

SimEvents®

User's Guide

R2012a

MATLAB®
& **SIMULINK®**

How to Contact MathWorks



www.mathworks.com Web
comp.soft-sys.matlab Newsgroup
www.mathworks.com/contact_TS.html Technical Support



suggest@mathworks.com Product enhancement suggestions
bugs@mathworks.com Bug reports
doc@mathworks.com Documentation error reports
service@mathworks.com Order status, license renewals, passcodes
info@mathworks.com Sales, pricing, and general information



508-647-7000 (Phone)



508-647-7001 (Fax)



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

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

SimEvents® *User's Guide*

© COPYRIGHT 2005–2012 by The MathWorks, Inc.

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

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

Trademarks

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

Patents

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

Revision History

November 2005	Online only	New for Version 1.0 (Release 14SP3+)
March 2006	Online only	Revised for Version 1.1 (Release 2006a)
September 2006	Online only	Revised for Version 1.2 (Release 2006b)
March 2007	Online only	Revised for Version 2.0 (Release 2007a)
September 2007	Online only	Revised for Version 2.1 (Release 2007b)
March 2008	Online only	Revised for Version 2.2 (Release 2008a)
October 2008	Online only	Revised for Version 2.3 (Release 2008b)
March 2009	Online only	Revised for Version 2.4 (Release 2009a)
September 2009	Online only	Revised for Version 3.0 (Release 2009b)
March 2010	Online only	Revised for Version 3.1 (Release 2010a)
September 2010	Online only	Revised for Version 3.1.1 (Release 2010b)
April 2011	Online only	Revised for Version 3.1.2 (Release 2011a)
September 2011	Online only	Revised for Version 4.0 (Release 2011b)
March 2012	Online only	Revised for Version 4.1 (Release 2012a)

Working with Entities

1

Generating Entities When Events Occur	1-2
Overview	1-2
Sample Use Cases for Event-Based Generation of Entities	1-2
Specifying Generation Times for Entities	1-4
Overview	1-4
Procedure for Generating Entities at Specified Times	1-4
Setting Attributes of Entities	1-6
Role of Attributes in SimEvents Models	1-6
Blocks That Set Attributes	1-6
Example: Setting Attributes	1-8
Example: Attaching Data Instead of Branching a Signal ..	1-10
Manipulating Attributes of Entities	1-12
Choice of Approaches for Manipulating Attributes	1-12
Writing Functions to Manipulate Attributes	1-12
Using Block Diagrams to Manipulate Attributes	1-15
Accessing Attributes of Entities	1-17
Counting Entities	1-19
Counting Departures Across the Simulation	1-19
Counting Departures per Time Instant	1-19
Resetting a Counter Upon an Event	1-21
Associating Each Entity with Its Index	1-23
Combining Entities and Allocating Resources	1-24
Overview of the Entity-Combining Operation	1-24
Example: Waiting to Combine Entities	1-25
Example: Copying Timers When Combining Entities	1-26
Example: Managing Data in Composite Entities	1-27

Replicating Entities on Multiple Paths	1-32
Sample Use Cases	1-32
Modeling Notes	1-32
Attribute Value Support	1-34

Working with Events

2

Supported Events in SimEvents Models	2-2
Types of Supported Events	2-2
Signal-Based Events	2-4
Function Calls	2-5
Example: Event Calendar Usage for a Queue-Server	
Model	2-7
Overview of Example	2-7
Start of Simulation	2-8
Generation of First Entity	2-8
Generation of Second Entity	2-9
Completion of Service Time	2-10
Generation of Third Entity	2-11
Generation of Fourth Entity	2-12
Completion of Service Time	2-13
Observing Events	2-15
Techniques for Observing Events	2-15
Example: Observing Service Completions	2-19
Example: Detecting Collisions by Comparing Events	2-22
Generating Function-Call Events	2-25
Role of Explicitly Generated Events	2-25
Generating Events When Other Events Occur	2-25
Generating Events Using Intergeneration Times	2-27
Manipulating Events	2-29
Reasons to Manipulate Events	2-29
Blocks for Manipulating Events	2-31

Creating a Union of Multiple Events	2-31
Translating Events to Control the Processing Sequence ..	2-34
Conditionalizing Events	2-36

Managing Simultaneous Events

3

Overview of Simultaneous Events	3-2
Exploring Simultaneous Events	3-4
Using Nearby Breakpoints to Focus on a Particular Time	3-5
For Further Information	3-5
Choosing an Approach for Simultaneous Events	3-7
Assigning Event Priorities	3-8
Procedure for Assigning Event Priorities	3-8
Tips for Choosing Event Priority Values	3-8
Procedure for Specifying Equal-Priority Behavior	3-9
Example: Choices of Values for Event Priorities	3-11
Overview of Example	3-11
Arbitrary Resolution of Signal Updates	3-12
Selecting a Port First	3-12
Generating Entities First	3-19
Randomly Selecting a Sequence	3-24
Example: Effects of Specifying Event Priorities	3-25
Overview of the Example	3-25
Default Behavior	3-26
Deferring Gate Events	3-27

Role of Event-Based Signals in SimEvents Models	4-2
Overview of Event-Based Signals	4-2
Comparison with Time-Based Signals	4-2
Tips for Using Event-Based Signals	4-3
Signal Restrictions for Event-Based Signals	4-3
Generating Random Signals	4-4
Generating Random Event-Based Signals	4-4
Examples of Random Event-Based Signals	4-5
Using Data Sets to Create Event-Based Signals	4-7
Behavior of the Event-Based Sequence Block	4-7
Generating Sequences Based on Arbitrary Events	4-8
Converting Between Time-Based and Event-Based Signals	4-10
When to Convert Signals	4-10
When Not to Convert Signals	4-11
How to Convert Signals	4-11
Signal Conversion When Using Custom Library Blocks	4-12
Manipulating Signals	4-14
Specifying Initial Values of Event-Based Signals	4-14
Example: Resampling a Signal Based on Events	4-15
Sending Data to the MATLAB Workspace	4-17
Behavior of the Discrete Event Signal to Workspace Block	4-17
Example: Sending Queue Length to the Workspace	4-17
Working with Multivalued Signals	4-20
Zero-Duration Values of Signals	4-20
Importance of Zero-Duration Values	4-21
Detecting Zero-Duration Values	4-21
Working with Bus Signals	4-24

Modeling Queues and Servers

5

Example: LIFO Queue Waiting Time	5-2
Sorting by Priority	5-4
Behavior of the Priority Queue Block	5-4
Example: FIFO and LIFO as Special Cases of a Priority Queue	5-4
Example: Serving Preferred Customers First	5-7
Preempting an Entity in a Server	5-10
Definition of Preemption	5-10
Criteria for Preemption	5-10
Residual Service Time	5-11
Queuing Disciplines for Preemptive Servers	5-11
Example: Preemption by High-Priority Entities	5-11
Determining Whether a Queue Is Nonempty	5-17
Modeling Multiple Servers	5-18
Blocks that Model Multiple Servers	5-18
Example: M/M/5 Queuing System	5-18
Modeling the Failure of a Server	5-20
Server States	5-20
Using a Gate to Implement a Failure State	5-20
Using Stateflow Charts to Implement a Failure State	5-21

Routing Techniques

6

Output Switching Based on a Signal	6-2
Routing Entities with an Output Switch	6-2
Specifying an Initial Port Selection	6-4
Using the Storage Option to Prevent Latency Problems ..	6-5

Example: Cascaded Switches with Skewed Distribution	6-9
Example: Compound Switching Logic	6-10
Example: Choosing the Shortest Queue	6-13

Regulating Arrivals Using Gates

7

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

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
Schematic Illustrating Procedure	8-4
Step 1: Designate the Entity Path	8-5
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
Common Requirements for Handling Timed-Out Entities	8-10
Techniques for Handling Timed-Out Entities	8-10
Example: Rerouting Timed-Out Entities to Expedite Handling	8-11
Example: Limiting the Time Until Service Completion	8-13

Computations on Event-Based Signals

9

Performing Computations in Atomic Subsystems	9-2
When to Use Atomic Subsystems for Computations on Event-Based Signals	9-2
How to Set Up Atomic Subsystems for Computations	9-2
Behavior of Computations in Atomic Subsystems	9-3
Refining the Behavior	9-4
Examples That Use Atomic Subsystems	9-5
Suppressing Computations By Filtering Out Events ..	9-6
When to Suppress Computations	9-6
How to Set Up Event Filter Blocks	9-7
Behavior of Event Filter Blocks	9-8
Evaluating the Behavior	9-9
Examples That Use Event Filter Blocks	9-10

Performing Computations in Function-Call	
Subsystems	9-11
When to Use Function-Call Subsystems for Computations	
on Event-Based Signals	9-11
How to Set Up Function-Call Subsystems for	
Computations	9-11
Behavior of Computations in Function-Call Subsystems ..	9-12
Refining the Behavior	9-13
Examples That Use Function-Call Subsystems	9-13
Blocks Inside Subsystems with Event-Based Input	
Signals	9-14
Performing Computations Without Using	
Subsystems	9-15
When to Perform Computations on Event-Based Signals	
Without Using Subsystems	9-15
How to Set Up Blocks for Computations	9-15
Behavior of Computations	9-15
Refining the Behavior	9-16
Examples That Perform Computations Without Using	
Subsystems	9-16

Plotting Data

10

Choosing and Configuring Plotting Blocks	10-2
Sources of Data for Plotting	10-2
Comparison of Blocks for Plotting Signals Against Time ..	10-3
Inserting and Connecting Scope Blocks	10-5
Connections Among Points in Plots	10-6
Varying Axis Limits Automatically	10-7
Caching Data in Scopes	10-8
Examples Using Scope Blocks	10-8
Working with Scope Plots	10-10
Customizing Plots	10-10
Exporting Plots	10-11

Using Plots for Troubleshooting	10-12
Example: Plotting Entity Departures to Verify Timing	10-13
Example: Plotting Event Counts to Check for Simultaneity	10-15

Using Statistics

11

Statistics for Data Analysis	11-2
Statistics for Run-Time Control	11-3
Statistical Tools for Discrete-Event Simulation	11-4
Accessing Statistics from SimEvents Blocks	11-5
Deriving Custom Statistics	11-7
Overview of Approaches to 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	11-9
Example: Resetting an Average Periodically	11-12
Measuring Point-to-Point Delays	11-18
Overview of Timers	11-18
Basic Example Using Timer Blocks	11-19
Basic Procedure for Using Timer Blocks	11-20
Timing Multiple Entity Paths with One Timer	11-21
Restarting a Timer from Zero	11-22
Timing Multiple Processes Independently	11-22
Varying Simulation Results by Managing Seeds	11-24

Connection Between Random Numbers and Seeds	11-24
Making Results Repeatable by Storing Sets of Seeds	11-25
Setting Seed Values Programmatically	11-26
Sharing Seeds Among Models	11-26
Working with Seeds Not in SimEvents Blocks	11-27
Choosing Seed Values	11-29

Regulating the Simulation Length	11-30
Overview	11-30
Setting a Fixed Stop Time	11-30
Stopping Upon Processing a Fixed Number of Entities ...	11-31
Stopping Upon Reaching a Particular State	11-32

Using Stateflow Charts in SimEvents Models

12

Role of Stateflow Charts in SimEvents Models	12-2
Guidelines for Using Stateflow and SimEvents	
Blocks	12-3
Examples Using Stateflow Charts and SimEvents	
Blocks	12-4
Failure State of Server	12-4
Go-Back-N ARQ Model	12-4

Debugging Discrete-Event Simulations

13

Overview of Debugging Resources	13-2
Overview of the SimEvents Debugger	13-3
Starting the SimEvents Debugger	13-5

The Debugger Environment	13-7
Debugger Command Prompt	13-7
Simulation Log in the Debugger	13-8
Identifiers in the Debugger	13-19
Independent Operations and Consequences in the Debugger	13-21
Significance of Independent Operations	13-21
Independent Operations	13-21
Consequences of Independent Operations	13-22
Stopping the Debugger	13-25
How to End the Debugger Session	13-25
Comparison of Simulation Control Functions	13-25
Stepping Through the Simulation	13-27
Overview of Stepping	13-27
How to Step	13-28
Choosing the Granularity of a Step	13-29
Tips for Stepping Through the Simulation	13-30
Inspecting the Current Point in the Debugger	13-32
Viewing the Current Operation	13-32
Obtaining Information Associated with the Current Operation	13-32
Inspecting Entities, Blocks, and Events	13-34
Inspecting Entities	13-34
Inspecting Blocks	13-36
Inspecting Events	13-38
Obtaining Identifiers of Entities, Blocks, and Events	13-38
Working with Debugging Information in Variables ...	13-41
Comparison of Variables with Inspection Displays	13-41
Functions That Return Debugging Information in Variables	13-41
How to Create Variables Using State Inspection Functions	13-42
Tips for Manipulating Structures and Cell Arrays	13-43
Example: Finding the Number of Entities in Busy Servers	13-43

Viewing the Event Calendar	13-46
For Further Information	13-46
Customizing the Debugger Simulation Log	13-47
Customizable Information in the Simulation Log	13-47
Tips for Choosing Appropriate Detail Settings	13-48
Effect of Detail Settings on Stepping	13-50
How to View Current Detail Settings	13-52
How to Change Detail Settings	13-52
How to Save and Restore Detail Settings	13-53
Debugger Efficiency Tips	13-55
Executing Commands Automatically When the Debugger Starts	13-55
Creating Shortcuts for Debugger Commands	13-56
Defining a Breakpoint	13-57
What Is a Breakpoint?	13-57
Identifying a Point of Interest	13-57
Setting a Breakpoint	13-59
Viewing All Breakpoints	13-61
Using Breakpoints During Debugging	13-63
Running the Simulation Until the Next Breakpoint	13-63
Ignoring or Removing Breakpoints	13-64
Enabling a Disabled Breakpoint	13-65
Block Operations Relevant for Block Breakpoints	13-67
Animating	13-74
Introduction	13-74
Starting and Stopping Animation	13-75
Animating Signals and Entities	13-76
Controlling Animation Speed	13-76
Animating Without Debugger Text	13-76
Animating the Output Switching Using Signal Model Without Debugger Text	13-77
Common Problems in SimEvents Models	13-79
Unexpectedly Simultaneous Events	13-79
Unexpectedly Nonsimultaneous Events	13-80

Unexpected Processing Sequence for Simultaneous Events	13-80
Unexpected Use of Old Value of Signal	13-81
Effect of Initial Value on Signal Loops	13-84
Loops in Entity Paths Without Sufficient Storage Capacity	13-85
Unexpected Timing of Random Signal	13-88
Unexpected Correlation of Random Processes	13-90
Blocks that Require Event-Based Signal Input	13-91
Invalid Connections of Gateway Blocks	13-91
Race Conditions Involving Entity Departure Function Calls	13-95
Saving Models at Earlier Versions of SimEvents	13-96
Blocks That Reference Simulation Time	13-97
Recognizing Latency in Signal Updates	13-98

Learning More About SimEvents Software

14

Solvers for Discrete-Event Systems	14-2
Variable-Step Solvers for Discrete-Event Systems	14-3
Fixed-Step Solvers for Discrete-Event Systems	14-4
Execution of Blocks Having Event-Based Input Signals	14-6
Response to Event-Based Input Signals	14-6
Event Sequencing	14-9
Processing Sequence for Simultaneous Events	14-9
Role of the Event Calendar	14-10
For Further Information	14-12
Choosing How to Resolve Simultaneous Signal Updates	14-14
Resolution Sequence for Input Signals	14-15
Detection of Signal Updates	14-15

Effect of Simultaneous Operations	14-16
Resolving the Set of Operations	14-17
Specifying Event Priorities to Resolve Simultaneous Signal Updates	14-17
Resolving Simultaneous Signal Updates Without Specifying Event Priorities	14-20
For Further Information	14-23
Livelock Prevention	14-24
Overview	14-24
Permitting Large Finite Numbers of Simultaneous Events	14-25
Signal-Based Event Cycle Prevention	14-26
Notifications and Queries Among Blocks	14-30
Overview of Notifications and Queries	14-30
Querying Whether a Subsequent Block Can Accept an Entity	14-30
Notifying Blocks About Status Changes	14-31
Notifying, Monitoring, and Reactive Ports	14-33
Overview of Signal Input Ports of SimEvents Blocks	14-33
Notifying Ports	14-33
Monitoring Ports	14-34
Reactive Ports	14-35
Connecting Event-Based Signal Generators to Reactive Ports	14-36
Interleaving of Block Operations	14-39
Overview of Interleaving of Block Operations	14-39
How Interleaving of Block Operations Occurs	14-39
Example: Sequence of Departures and Statistical Updates	14-40
Update Sequence for Output Signals	14-45
Determining the Update Sequence	14-45
Example: Detecting Changes in the Last-Updated Signal	14-46
SimEvents Support for Simulink Subsystems	14-48

Discrete-Event Blocks in Virtual Subsystems	14-48
Discrete-Event Blocks in Nonvirtual Subsystems	14-48
Discrete-Event Blocks in Variant Subsystems	14-50
Storage and Nonstorage Blocks	14-51
Storage Blocks	14-51
Nonstorage Blocks	14-51
Blocks That Support Event-Based Input Signals	14-53
Computational Blocks	14-53
Sink Blocks	14-54
SimEvents Blocks	14-55
Other Blocks	14-55

Migrating SimEvents Models

15

Introduction	15-2
Legacy Behavior in SimEvents Models	15-3
Visual Appearance of Legacy SimEvents Blocks	15-3
Multifiring Behavior	15-4
Entities Arriving at Empty Queue Blocks	15-10
No Events at Time 0	15-11
Checking Your Model for Legacy Behavior	15-13
Model Advisor Checks for SimEvents	15-13
Migration Using seupdate	15-14
Before Migration	15-15
Preparing to Migrate a Model	15-15
Expected Changes to Model Contents	15-15
Running seupdate	15-17
Resolving Migration Failure	15-19

After You Migrate	15-21
Model Behavior Changes	15-22
Eliminating Multifiring Behavior	15-22
Time-Based Execution in Previous Model	15-22
Algebraic Loops	15-23
Changed Behavior of Entities Arriving at Empty Queue	
Blocks	15-23
Initial Values	15-24
Bus Signals Converted to Non-Bus Signals	15-26
Events at Time 0	15-27
Migration Limitations	15-30
Position of Gateway Blocks	15-30
Cascaded Mux or Bus Blocks	15-30

Examples

A

Attributes of Entities	A-2
Counting Entities	A-3
Queuing Systems	A-4
Working with Events	A-5
Working with Signals	A-6
Server States	A-7
Routing Entities	A-8
Gates	A-9
Timeouts	A-10

Troubleshooting	A-11
Statistics	A-12
Timers	A-13

Index



Working with Entities

- “Generating Entities When Events Occur” on page 1-2
- “Specifying Generation Times for Entities” on page 1-4
- “Setting Attributes of Entities” on page 1-6
- “Manipulating Attributes of Entities” on page 1-12
- “Accessing Attributes of Entities” on page 1-17
- “Counting Entities” on page 1-19
- “Combining Entities and Allocating Resources” on page 1-24
- “Replicating Entities on Multiple Paths” on page 1-32
- “Attribute Value Support” on page 1-34

Generating Entities When Events Occur

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

Overview

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

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

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

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

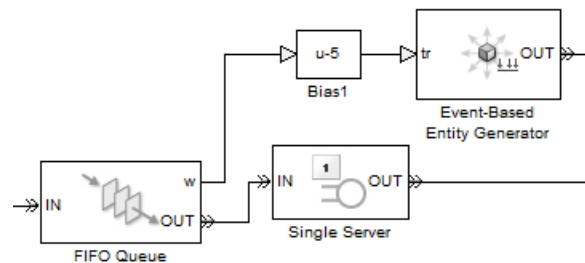
To specify intergeneration times between pairs of successive entities instead of using an event-based approach, use the Time-Based Entity Generator block. For more information, see “Creating Entities Using Intergeneration Times” in the SimEvents® getting started documentation.

Sample Use Cases for Event-Based Generation of Entities

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

- To generate an entity each time a Stateflow® chart transitions from state A to state B, configure the chart to output a function call upon such a transition. Then configure the Event-Based Entity Generator block to react to each function call by generating an entity.
- To generate an entity each time a real-valued statistical signal crosses a threshold, configure the Event-Based Entity Generator block to react to triggers. Connect the **tr** signal input port of the block to the statistical signal minus the threshold.

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



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

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

Specifying Generation Times for Entities

In this section...

“Overview” on page 1-4

“Procedure for Generating Entities at Specified Times” on page 1-4

Overview

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

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

Procedure for Generating Entities at Specified Times

To generate entities at specified times, follow this procedure:

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

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

An example of a column vector listing generation times is:

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

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

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

- 5 Insert an Event-Based Sequence block in the model and connect it to the `t` input port of the Time-Based Entity Generator block.
- 6 In the Event-Based Sequence block, set **Vector of output values** to `intergentimes`. Set the **Form output after final data value by** parameter to `Setting to infinity`. The `Setting to infinity` option halts the generation process if the simulation time exceeds your maximum generation time.

Setting Attributes of Entities

In this section...
“Role of Attributes in SimEvents Models” on page 1-6
“Blocks That Set Attributes” on page 1-6
“Example: Setting Attributes” on page 1-8
“Example: Attaching Data Instead of Branching a Signal” on page 1-10

Role of Attributes in SimEvents Models

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

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

Blocks That Set Attributes

To assign attributes on each arriving entity, use one of the blocks listed in the following table. Assignments can create new attributes or change the values of existing attributes.

Data	Blocks to Use	More Information and Examples
Constant value	Set Attribute	Set Attribute block reference page “Example: Setting Attributes” on page 1-8
Random numbers	Event-Based Random Number followed by Set Attribute	“Generating Random Signals” on page 4-4 Set Attribute block reference page “Example: A Packet Switch”

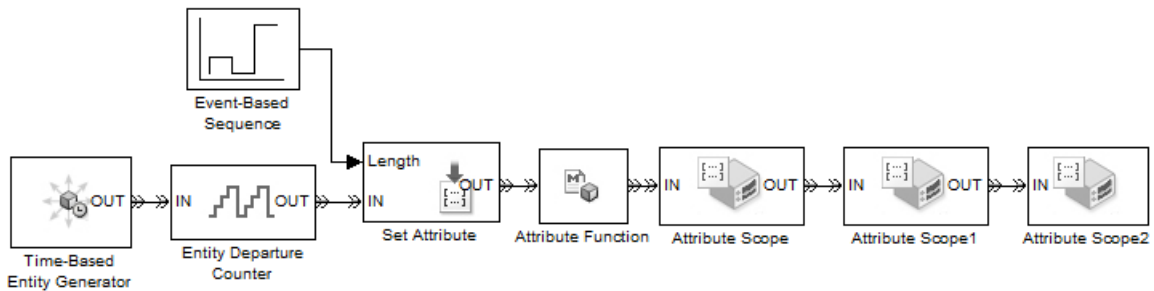
Data	Blocks to Use	More Information and Examples
Elements of either a vector in the MATLAB workspace or a vector that you can type in a block dialog box	Event-Based Sequence followed by Set Attribute	“Using Data Sets to Create Event-Based Signals” on page 4-7 Set Attribute block reference page “Example: Setting Attributes” on page 1-8
Values of an output argument of a MATLAB function that you write	Attribute Function, or MATLAB Function followed by Set Attribute	“Writing Functions to Manipulate Attributes” on page 1-12 “Example: Setting Attributes” on page 1-8
Values of an event-based signal	Set Attribute	Set Attribute block reference page Communication Protocol Modeling in an Ethernet LAN demo, in MAC controller subsystems
Values of a time-based signal	Set Attribute preceded by Timed to Event Signal	Set Attribute block reference page “Converting Between Time-Based and Event-Based Signals” on page 4-10 “Building a Simple Hybrid Model”
Entity count	Entity Departure Counter with Write count to attribute check box selected	“Associating Each Entity with Its Index” on page 1-23 “Example: Setting Attributes” on page 1-8

For Further Information

- “Attribute Value Support” on page 1-34 — Characteristics of data an entity can store in an attribute
- “Accessing Attributes of Entities” on page 1-17 — How to query entities for attribute values
- “Combining Entities and Allocating Resources” on page 1-24 — How to aggregate attributes from distinct entities

Example: Setting Attributes

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

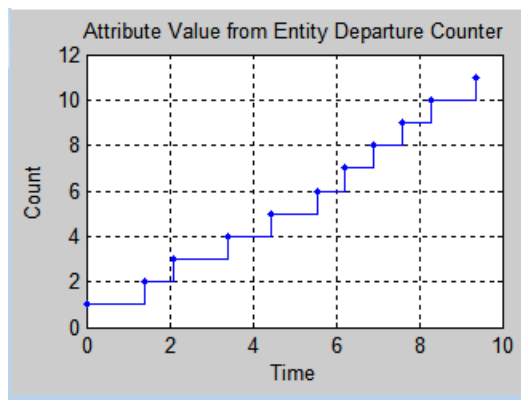


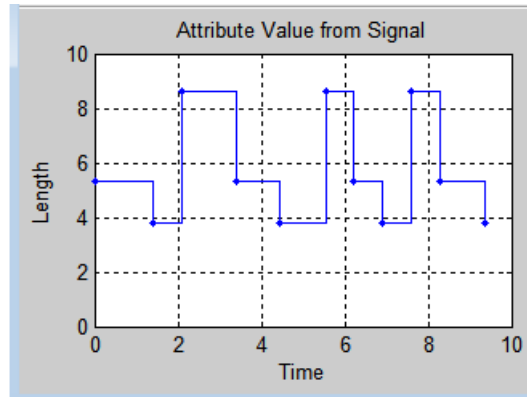
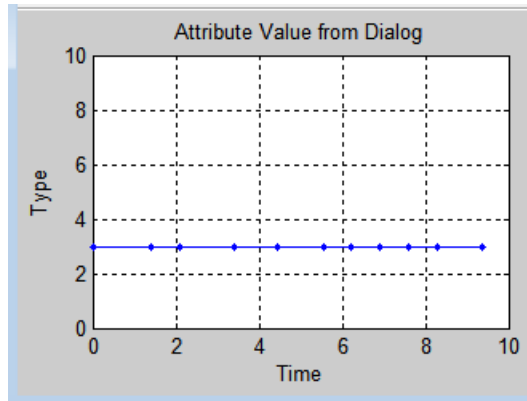
After each entity departs from the Attribute Function block, it possesses the attributes listed in the table.

Attribute Name	Attribute Value	Method for Setting Attribute Value
Count	N, for the Nth entity departing from the Time-Based Entity Generator block	In Entity Departure Counter dialog box: Write count to attribute check box selected 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

Attribute Name	Attribute Value	Method for Setting Attribute Value
		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
LengthInt	floor(Length)	Attribute Function block whose function is function [out_LengthInt] = fcn(Length) out_LengthInt = floor(Length);

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





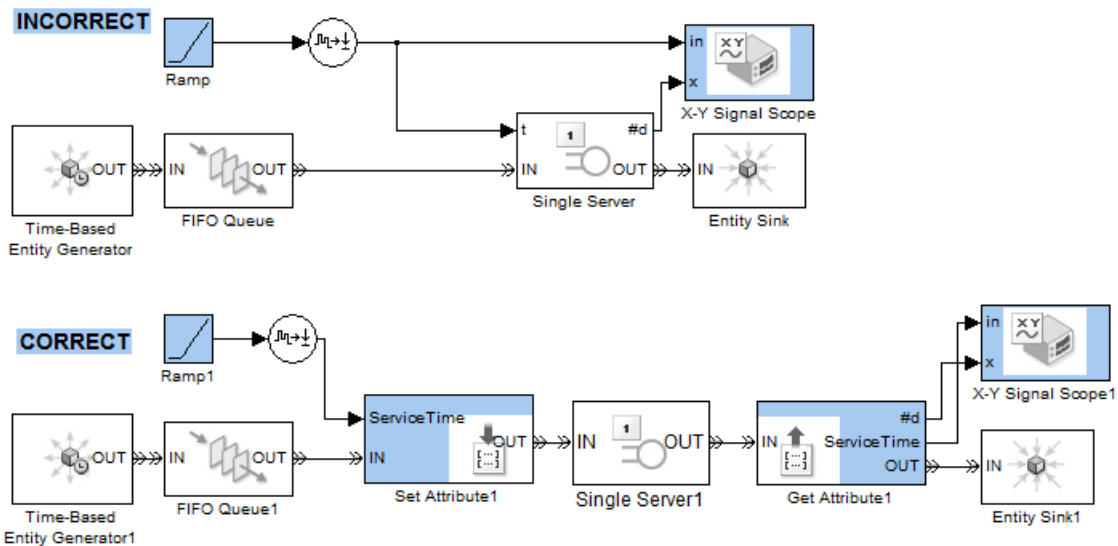
Example: Attaching Data Instead of Branching a Signal

In some modeling situations, it is important to attach data to an entity instead of using a signal directly. This example shows the importance of considering not only the topology of your block diagrams, but also the timing of data signals.

The model contains a server with varying service times. Suppose you want to plot the service time against entity count for each entity departing from the server. A signal specifies the service time to use for each entity. Although connecting the same signal to the Signal Scope block appears correct topologically, the timing in such an arrangement is incorrect. The

incorrectness arises from the delay at the server. That is, the signal has one value when a given entity arrives at the server and another value when the same entity arrives at the scope.

To implement the example correctly, attach the service time to each entity using an attribute and retrieve the attribute value when needed from each entity. That way, the scope receives the service time associated with each entity, regardless of the delay between arrival times at the server and the scope.



Manipulating Attributes of Entities

In this section...

“Choice of Approaches for Manipulating Attributes” on page 1-12

“Writing Functions to Manipulate Attributes” on page 1-12

“Using Block Diagrams to Manipulate Attributes” on page 1-15

Choice of Approaches for Manipulating Attributes

The table compares approaches for manipulating attributes of entities.

Approach	Advantages	Details
Write a function that uses MATLAB code to modify an attribute or define a new attribute. Use the Attribute Function block to invoke your function for each entity that arrives at the block.	<ul style="list-style-type: none"> • Setup requires only one block. • Many computations are simpler to express in code than in blocks. 	“Writing Functions to Manipulate Attributes” on page 1-12
Use the Get Attribute, Set Attribute, and Single Server blocks to structure the computation correctly. Use blocks inside the subsystem to perform the specific computation.	<ul style="list-style-type: none"> • Computation can use signal data that is not in an attribute. 	“Using Block Diagrams to Manipulate Attributes” on page 1-15

Writing Functions to Manipulate Attributes

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

For examples that use the block, see

- “Example: Setting Attributes” on page 1-8

- “Attribute Names Unique and Accessible in Composite Entity” on page 1-28.
- Modeling Multicomponent Parts using Combiners and Splitters demo
- Distributing Multi-Class Jobs to Service Stations demo, within the Distribution Center subsystem and its Service History Monitor subsystem

Procedure for Using the Attribute Function Block

The Attribute Function block has one entity input port and one entity output port. The block manipulates attributes of each arriving entity. See “Attribute Value Support” on page 1-34 for the characteristics of attribute values. To edit the computation, use this procedure:

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

Argument-Naming Rules

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

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

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

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

- 3** Write the function to implement your specific computation. The value of each attribute can be a real- or complex-valued array of any fixed dimension and double data type, but cannot be a structure. For each

attribute, the dimensions and complexity must be consistent throughout the model. Also, the function must use only those MATLAB functions and operators that are suitable for code generation. For more information, see “MATLAB Language Features Supported for Code Generation” and “MATLAB Language Features Not Supported for Code Generation” in the code generation documentation.

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

Example: Incorporating Legacy Code

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

- Represents your desired attribute manipulation
- Uses only those MATLAB functions and operators that are suitable for code generation

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

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

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

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

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

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

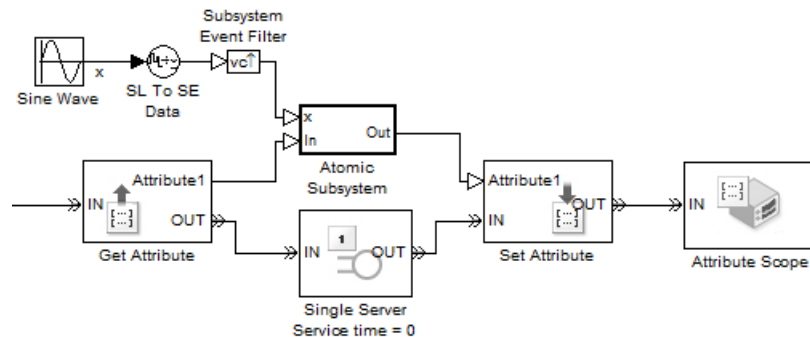
In your legacy code, the function name and the number of input and output arguments must be compatible with the invocation above. However, the legacy code does not need to follow the argument-naming rules for the Attribute Function block. Below is an example of a function that is compatible with the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

```
function outp = mylegacyfcn(inp)

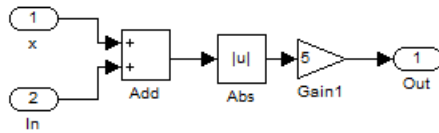
if inp < 0
    outp = 0;
else
    if ((inp >= 0) && (inp < 1))
        outp = 0.25;
    else
        outp = 5;
    end
end
end
```

Using Block Diagrams to Manipulate Attributes

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



Top-Level Model



Subsystem Contents

In your own model, you can vary:

- The number of outputs of the Get Attribute block
- The number of inputs of the Set Attribute block
- The connections or contents of the subsystem, where all input signals of the subsystem are event-based signals

The key components are in the table.

Block	Role
Get Attribute	Queries the entity for its attribute value.
Atomic Subsystem	Ensures that the computation executes as an atomic unit. To learn more, see “Performing Computations in Atomic Subsystems” on page 9-2.
Content of the subsystem	Models your specific computation. The earlier figure shows one example: $5 Attr+x $, where <i>Attr</i> is an attribute value and <i>x</i> is a time-based signal converted into an event-based signal. The subsystem executes if and only if <i>Attr</i> has a sample time hit.
Single Server with Service time set to 0	Ensures that the Set Attribute block uses the up-to-date results of the computation. For details, see “Interleaving of Block Operations” on page 14-39.
Set Attribute	Assigns new value to the attribute.

Accessing Attributes of Entities

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

Use Attribute Values To...	Technique
Create a signal	<p>Use the Get Attribute block.</p> <p>For example, see the subsystem of the model described in “Adding Event-Based Behavior” in the SimEvents getting started documentation.</p>
Create a plot	<p>Use the Attribute Scope block and name the attribute in the Y attribute name parameter of the block.</p> <p>Alternatively, use the X-Y Attribute Scope block and name two attributes in the X attribute name and Y attribute name parameters of the block.</p> <p>For example, see the reference page for the X-Y Attribute Scope block.</p>
Compute a different attribute value	<p>Use the Attribute Function block.</p> <p>For example, see “Attribute Names Unique and Accessible in Composite Entity” on page 1-28.</p>
Help specify behavior of a block that supports the use of attribute values for block parameters. Examples are the service time for a server and the selected port for an output switch.	<p>Name the attribute in the block dialog box as applicable.</p> <p>For example, see “Example: Using an Attribute to Select an Output Port” in the SimEvents getting started documentation.</p>

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

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

Use the attribute directly. This is better than creating a signal with the attribute value and ensuring that the signal is up-to-date when the entity arrives. For a comparison of the two approaches, see “Example: Using a Signal or an Attribute” on page 13-82.

Counting Entities

In this section...

“Counting Departures Across the Simulation” on page 1-19

“Counting Departures per Time Instant” on page 1-19

“Resetting a Counter Upon an Event” on page 1-21

“Associating Each Entity with Its Index” on page 1-23

Counting Departures Across the Simulation

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

- “Building a Simple Discrete-Event Model” in the SimEvents getting started documentation
- “Example: First Entity as a Special Case” on page 7-11
- “Stopping Upon Processing a Fixed Number of Entities” on page 11-31

Counting Departures per Time Instant

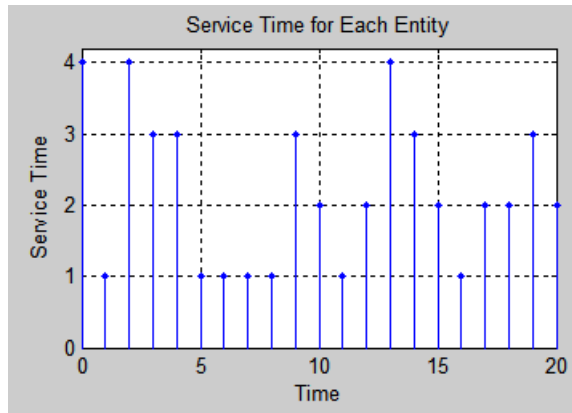
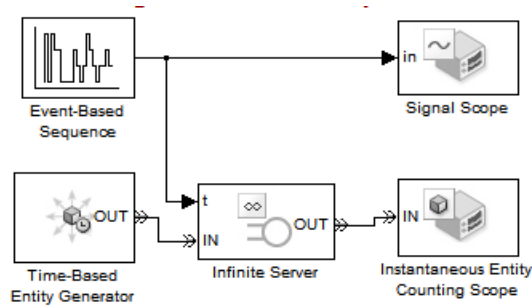
Suppose you want to visualize entity departures from a particular block, and you want to reset (that is, restart) the counter at each time instant. Visualizing departures per time instant can help you:

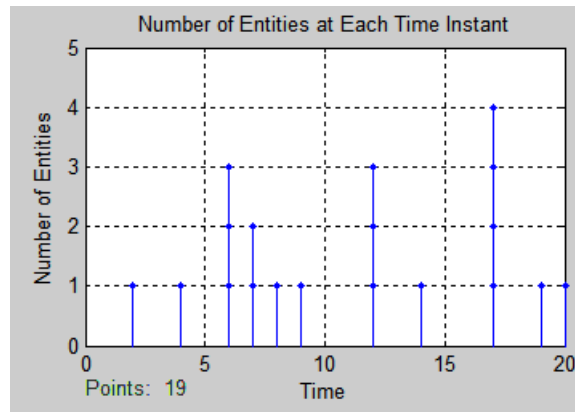
- Detect simultaneous departures
- Compare the number of simultaneous departures at different time instants
- Visualize the departure times while keeping the plot axis manageable

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

Example: Counting Simultaneous Departures from a Server

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





Resetting a Counter Upon an Event

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

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

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

These resources can help you choose which type of events:

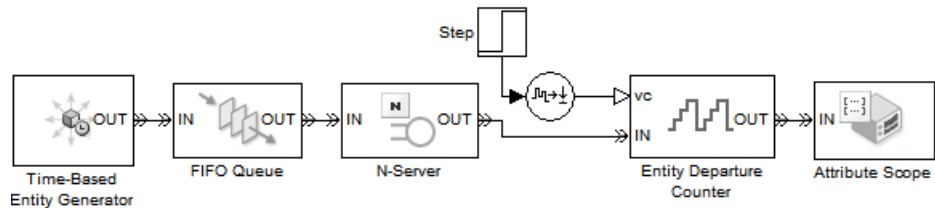
- To learn the difference among the kinds of signal-based events, see “Signal-Based Events” on page 2-4 and “Function Calls” on page 2-5.
- To build a signal that has events at the times that you need them, see “Manipulating Events” on page 2-29.

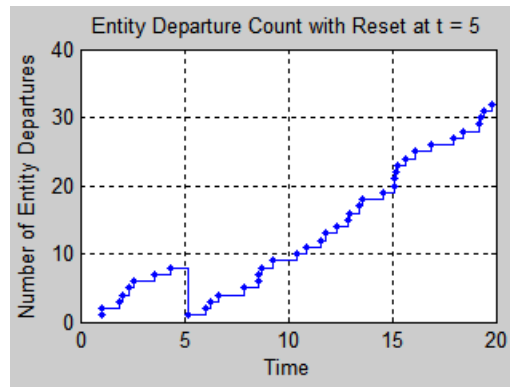
- To visualize the events of a signal, use the Instantaneous Event Counting Scope block.
- 2 In the Entity Departure Counter block, indicate which signal-based events cause the counter to reset:
 - To reset upon sample time hits of an input signal, set **Reset counter upon** to Sample time hit from port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

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

Example: Resetting a Counter After a Transient Period

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





Associating Each Entity with Its Index

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

For an example, see “Example: Setting Attributes” on page 1-8.

For an example in which the Entity Departure Counter block is more straightforward than storing the **#d** output signal in an attribute, see “Example: Sequence of Departures and Statistical Updates” on page 14-40.

Combining Entities and Allocating Resources

In this section...
“Overview of the Entity-Combining Operation” on page 1-24
“Example: Waiting to Combine Entities” on page 1-25
“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. Alternatively, you can model resource allocation by combining an entity that represents a resource with an entity that represents a part or other item.

The Entity Combiner block and its surrounding blocks automatically detect when all necessary component entities are present and the result of the combining operation would be able to advance to a storage block or a block that destroys entities. In effect, the blocks synchronize the component entities.

The Entity Combiner block provides options for managing information (attributes and timers) associated with the component entities. You can also configure the Entity Combiner block to make the combining operation reversible via the Entity Splitter block.

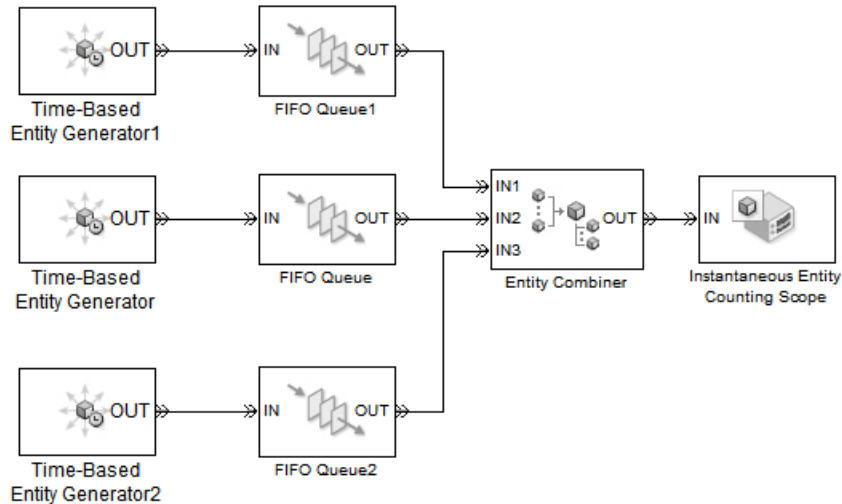
For Further Information

- Entity Combiner reference page
- Entity Splitter reference page
- Batch Production Process
- Packet Creation, Transmission and Error Analysis
- Kanban Production System

- Resource Allocation from Multiple Pools

Example: Waiting to Combine Entities

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



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

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

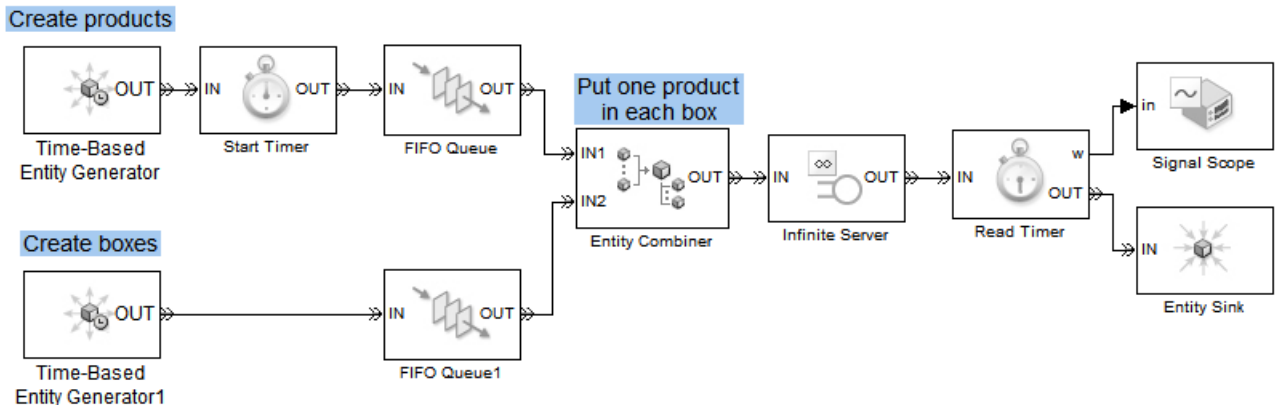
The bottom entity generator has the largest intergeneration time among the three entity generators, and is the limiting factor that determines when the Entity Combiner block accepts one entity from each queue. The top and middle queues store pending entities while waiting for the bottom entity generator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

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

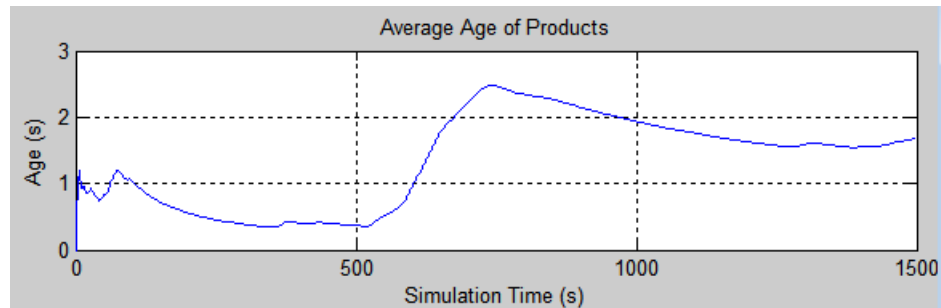
While you could alternatively synchronize the departures from the three queues using appropriately configured gates, it is simpler and more intuitive to use the Entity Combiner block as shown.

Example: Copying Timers When Combining Entities

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



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

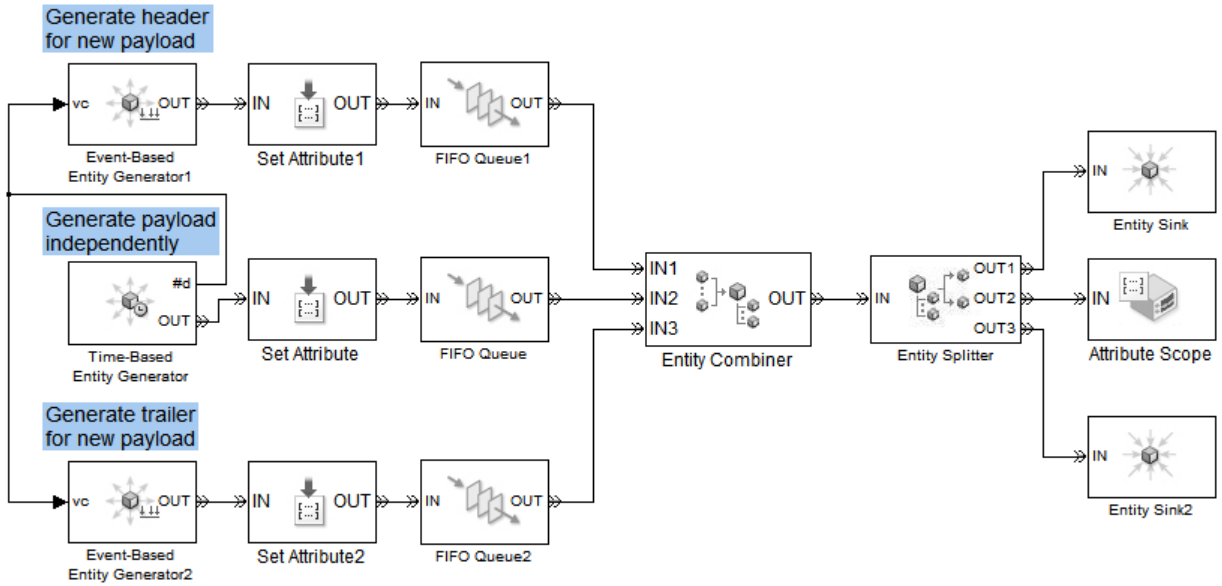


Example: Managing Data in Composite Entities

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

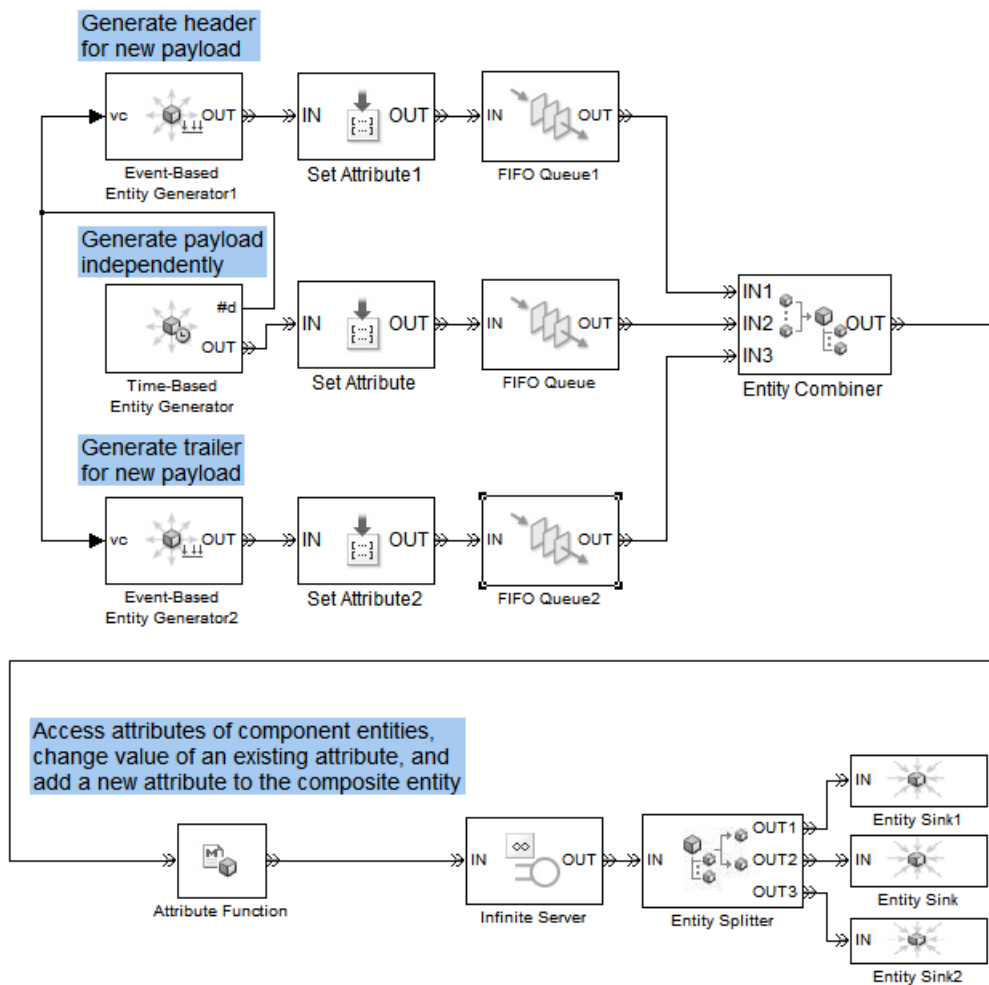
Attribute Names Nonunique and Inaccessible in Composite Entity

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



Attribute Names Unique and Accessible in Composite Entity

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



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

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

Function Underlying the Attribute Function Block.

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

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

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

Replicating Entities on Multiple Paths

In this section...
“Sample Use Cases” on page 1-32
“Modeling Notes” on page 1-32

Sample Use Cases

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

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

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

Modeling Notes

- Unlike the Output Switch block, the Replicate block has departures at all of its entity output ports that are not blocked, not just a single selected entity output port.
- If your model routes the replicates such that they use a common entity path, then be aware that blockages can occur during the replication process. For example, connecting all ports of a Replicate block, Path Combiner block, and Single Server block in that sequence can create a blockage because the server can accommodate at most one of the replicates at a time. The blockage causes fewer than the maximum number of replicates to depart from the block.

- Each time the Replicate block replicates an entity, the copies depart in a sequence whose start is determined by the **Departure port precedence** parameter. Although all copies depart at the same time instant, the sequence might be significant in some modeling situations. For details, see the reference page for the Replicate block.

Attribute Value Support

These lists summarize the characteristics of attribute values.

Permitted Characteristics of Attribute Values

- Real or complex
- Array of any dimension, where the dimensions remain fixed throughout the simulation
- double data type

For a given attribute, the characteristics of the value must be consistent throughout the discrete-event system in the model.

Not Permitted as Attribute Values

- Structure
- Data types other than double. In particular, strings are not valid as attribute values.
- Bus
- Variable-size signals or variable-size arrays
- Frame

Working with Events

- “Supported Events in SimEvents Models” on page 2-2
- “Example: Event Calendar Usage for a Queue-Server Model” on page 2-7
- “Observing Events” on page 2-15
- “Generating Function-Call Events” on page 2-25
- “Manipulating Events” on page 2-29

To learn about working with sets of simultaneous events, see Chapter 3, “Managing Simultaneous Events”. To view information about events during the simulation, see Chapter 13, “Debugging Discrete-Event Simulations”.

Supported Events in SimEvents Models

In this section...
“Types of Supported Events” on page 2-2
“Signal-Based Events” on page 2-4
“Function Calls” on page 2-5

Types of Supported Events

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

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

Event	Description
Function call	Discrete invocation request carried from block to block by a special signal called a function-call signal. For more information, see “Function Calls” on page 2-5.
Gate (opening or closing)	Opening or closing of the gate represented by the Enabled Gate block.
Memory read	Reading of memory in the Signal Latch block.
Memory write	Writing of memory in the Signal Latch block.
New head of queue	Scheduled when there is a new entity at the head of a queue. Upon execution, it causes the entity at the head of a queue to attempt to depart.
Port selection	Selection of a particular entity port in the Input Switch, Output Switch, or Path Combiner block. In the case of the Path Combiner block, the selected entity input port is the port that the block notifies first, whenever its entity output port changes from blocked to unblocked.
Preemption	Replacement of an entity in a server by a higher priority entity.
Release	Opening of the gate represented by the Release Gate block.
Sample time hit	Update in the value of a signal that is connected to a block configured to react to signal updates.
Service completion	Completion of service on an entity in a server.
Storage completion	Change in the state of the Output Switch block, making it attempt to advance the stored entity.
Subsystem	Execution of Atomic Subsystem block caused by an appropriate signal-based event in the input signal of a Event Filter block that connects to the Atomic Subsystem block.
Timeout	Departure of an entity that has exceeded a previously established time limit.

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

During a simulation, the application maintains a list, called the *event calendar*, of selected upcoming events that are scheduled for the current simulation time or future times. By referring to the event calendar, the application executes events at the correct simulation time and in an appropriate sequence. If a model has multiple discrete-event systems (in which signals change when events occur), each discrete-event system maintains its own event calendar. The application executes the event calendar of each discrete-event system independently of the other discrete-event systems.

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.
- It is unreliable to induce signal-based events based on sample time hits from a time-based signal connected to the discrete-event system via a

gateway block. For more information see “Invalid Connections of Gateway Blocks” on page 13-91.

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

Function Calls

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

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

Within a discrete-event system, function-calls can generate from these kinds of blocks:

- Entity Departure Function-Call Generator
- Signal-Based Function-Call Generator
- Time-Based Function-Call Generator
- Stateflow blocks
- MATLAB Function blocks
- Atomic Subsystem block containing a Function-Call Generator block

Note The software does not support asynchronous function-calls in discrete-event systems. It also does not support asynchronous function-calls at discrete-event system boundaries (as identified by gateway blocks).

You can combine or manipulate function-call signals. To learn more, see “Manipulating Events” on page 2-29.

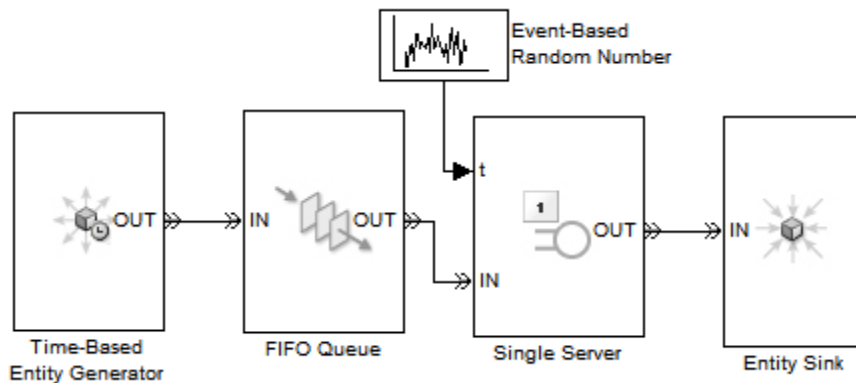
Example: Event Calendar Usage for a Queue-Server Model

In this section...

“Overview of Example” on page 2-7
 “Start of Simulation” on page 2-8
 “Generation of First Entity” on page 2-8
 “Generation of Second Entity” on page 2-9
 “Completion of Service Time” on page 2-10
 “Generation of Third Entity” on page 2-11
 “Generation of Fourth Entity” on page 2-12
 “Completion of Service Time” on page 2-13

Overview of Example

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



Assume that the blocks are configured so that:

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

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

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

Start of Simulation

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

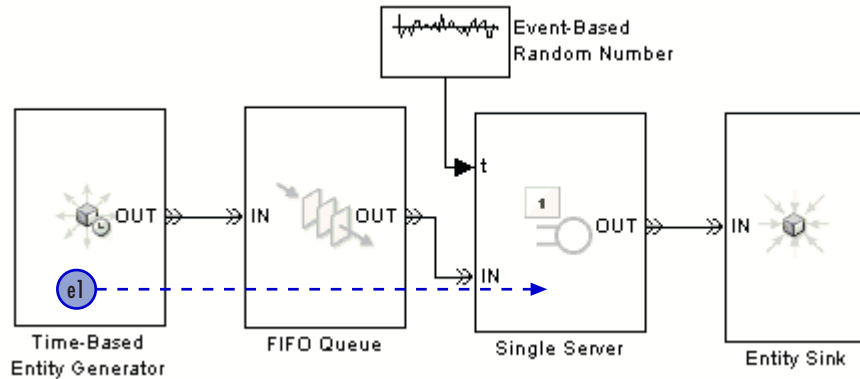
Time of Event (s)	Type of Event
0.9	Time-Based Entity Generator block generates an entity.

Generation of First Entity

At $T = 0.9$,

- The entity generator generates an entity and attempts to output it.
- The queue is empty, so the entity advances from the entity generator to the queue.
- The queue schedules an event that indicates that the entity has been placed at the head of the queue. This event is called a `NewHeadOfQueue` event.
- The server is empty, so the entity advances from the queue to the server.
- The server's entity input port becomes temporarily unavailable to future entities.
- The server schedules an event that indicates when the entity's service time is completed. The service time is 1.3 seconds, so service is complete at $T = 2.2$.
- The entity generator schedules its next entity-generation event, at $T = 1.7$.

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



The event calendar looks like this.

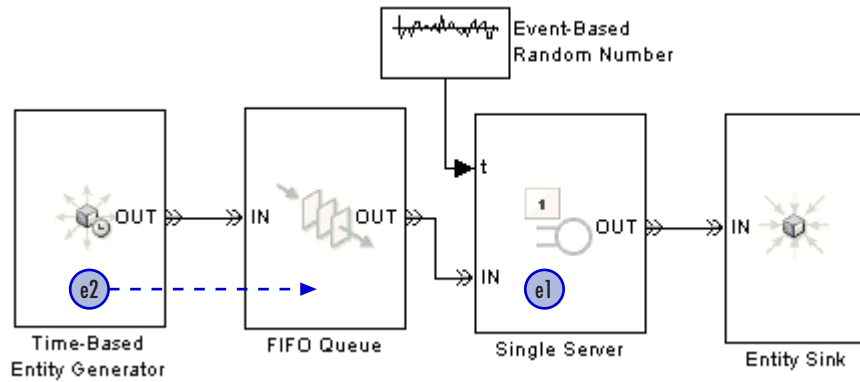
Time of Event (s)	Event Description
0.9	FIFO Queue block schedules a <code>NewHeadOfQueue</code> event that indicates arrival of first entity at empty queue.
1.7	Time-Based Entity Generator block generates second entity.
2.2	Single Server block completes service on the first entity.

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 queue schedules a `NewHeadOfQueue` event that indicates arrival of second entity at empty queue.

- 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
1.7	FIFO Queue block schedules a NewHeadOfQueue event.
2.2	Single Server block completes service on the first entity.
3.8	Time-Based Entity Generator block generates the third entity.

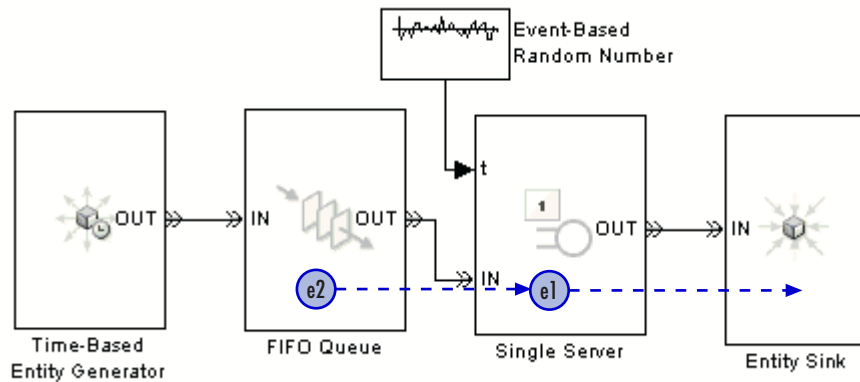
Completion of Service Time

At $T = 2.2$,

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

- The queue is now able to output the second entity to the server. As a result, the queue becomes empty and the server becomes busy again.
- The server's entity input port becomes temporarily unavailable to future entities.
- The server schedules an event that indicates when the second entity's service time is completed. The service time is 2.0 seconds.

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

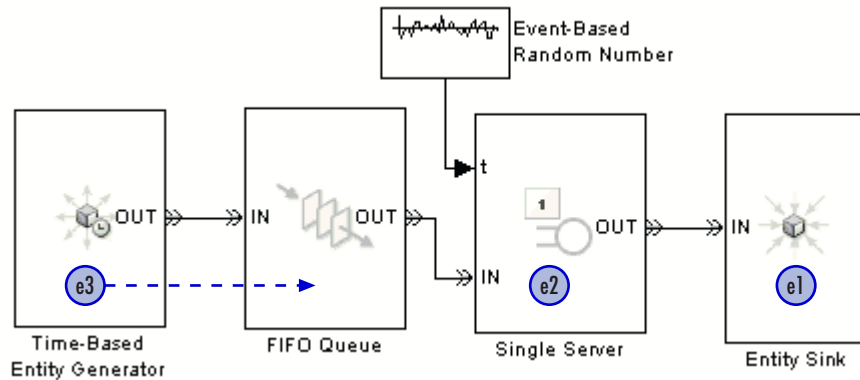


Time of Event (s)	Event Description
3.8	Time-Based Entity Generator block generates the third entity.
4.2	Single Server block completes service on the second entity.

Generation of Third Entity

At $T = 3.8$,

- The entity generator generates an entity and attempts to output it.
- The queue is empty, so the entity advances from the entity generator to the queue.
- The queue schedules a `NewHeadOfQueue` event that indicates arrival of third entity at empty queue.
- 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 $T = 3.9$.



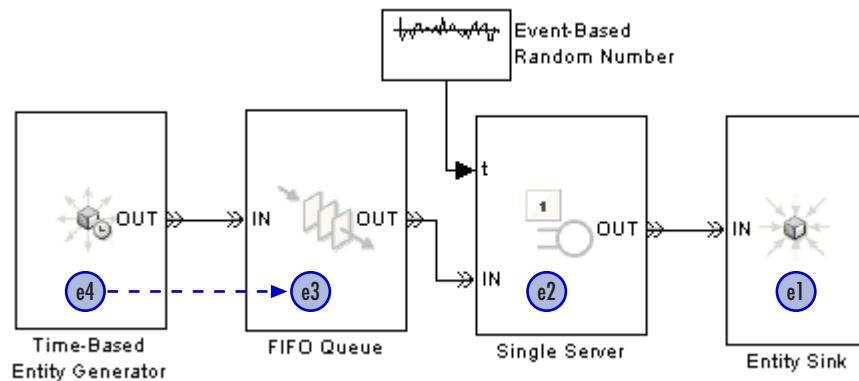
Time of Event (s)	Event Description
3.8	FIFO Queue block schedules a <code>NewHeadOfQueue</code> event.
3.9	Time-Based Entity Generator block generates the fourth entity.
4.2	Single Server block completes service on the second entity.

Generation of Fourth Entity

At $T = 3.9$,

- The entity generator generates an entity and attempts to output it.

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



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

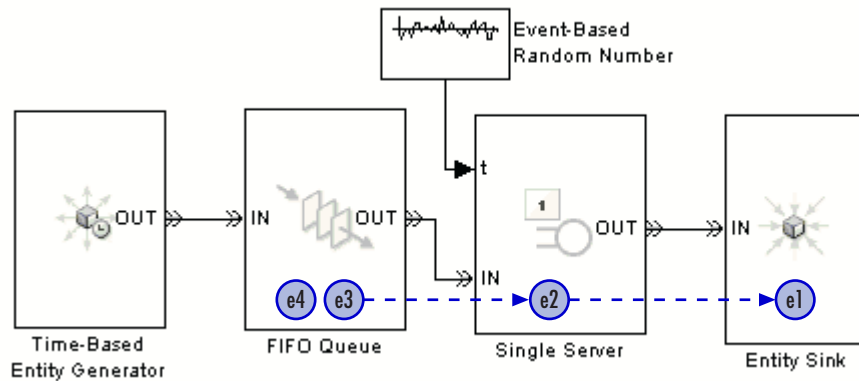
Completion of Service Time

At $T = 4.2$,

- The server finishes serving its entity and attempts to output it.
- The sink block accepts all entities by definition, so the entity advances from the server to the sink.
- The server's entity input port becomes available, so it executes an entity request event. The queue's entity output port becomes unblocked. The queue is now able to output the third entity to the server. As a result, the queue length becomes one, and the server becomes busy.

- The server's entity input port becomes temporarily unavailable to future entities.
- The server schedules an event that indicates when the entity's service time is completed. The service time is 0.7 seconds.
- The queue attempts to output the fourth entity. However, the server's entity input port is unavailable, so this entity stays in the queue. The queue's entity output port becomes blocked.

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



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

Observing Events

In this section...
“Techniques for Observing Events” on page 2-15
“Example: Observing Service Completions” on page 2-19
“Example: Detecting Collisions by Comparing Events” on page 2-22

Techniques for Observing Events

The SimEvents debugger can help you observe events and the relative sequencing of simultaneous events. For details, see “Overview of the SimEvents Debugger” on page 13-3.

The next table describes some additional observation techniques. Each technique focuses on a particular kind of event. These techniques indicate the simulation time at which events occur but do not indicate relative sequencing of simultaneous events. Key tools are the Instantaneous Event Counting Scope block, Signal Scope block, and Discrete Event Signal to Workspace block.

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

Event	Observation Technique
Entity generation	<p>In the entity generator block, observe values of the pe and #d output signals. Upon entity generation, you see one of the following outcomes:</p> <ul style="list-style-type: none"> • The generated entity departs immediately. #d increases and pe does not change. • The generated entity cannot depart immediately. pe increases. <p>To build a concrete example, adapt the technique described in “Example: Observing Service Completions” on page 2-19.</p>
Entity request	
Function call	<p>If the block issuing the function call provides a #f1 output signal, observe its increases. Otherwise, add a Function-Call Split block from the Simulink library to your model. Connect the input port of the Function-Call Split block to the block issuing the function call. Connect one output port of the Function-Call Split block to the block reacting to the function-call. Connect the second output port of the Function-Call Split block to an Instantaneous Event Counting Scope block.</p>
Gate (opening or closing)	<p>In the Enabled Gate block, use a branch line to connect the en input signal to an Instantaneous Event Counting Scope block. Rising trigger edges of the input signal indicate gate-opening events, while falling trigger edges of the input signal indicate gate-closing events.</p>
Head of queue push	
Memory read	<p>In the Signal Latch block, observe sample time hits in the out output signal.</p>
Memory write	<p>In the Signal Latch block, observe sample time hits in the mem output signal.</p>

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 that is configured to plot value changes. For the Input Switch or Output Switch block, an alternative is to observe the last output signal.
Preemption	In the server block, observe increases in the #p output signal.
Release	In the Release Gate block, use a branch line to connect the input signal to an Instantaneous Event Counting Scope block.
Sample time hit	Use a branch line to connect the signal to an Instantaneous Event Counting Scope block.
Service completion	<p>For Single Server blocks, observe values of the pe and #d output signals. Upon service completion, you see one of the following outcomes:</p> <ul style="list-style-type: none"> • The entity departs immediately. #d increases, and pe does not change. • The entity cannot depart immediately. pe increases. <p>To build a concrete example, adapt the technique described in “Example: Observing Service Completions” on page 2-19.</p>
	<p>For Infinite Server and N-Server blocks, observe values of the #pe and #d output signals. Upon service completion, you see one of the following outcomes:</p> <ul style="list-style-type: none"> • The entity departs immediately. #d increases, and #pe does not change. • The entity cannot depart immediately. #pe increases. <p>For a concrete example, see “Example: Observing Service Completions” on page 2-19.</p>

Event	Observation Technique
Storage completion	<p>In the switch block, observe values of the pe and #d output signals. Upon storage completion, you see one of the following outcomes:</p> <ul style="list-style-type: none"> • The entity departs immediately. #d increases, and pe does not change. • The entity cannot depart immediately. pe increases. <p>To build a concrete example, adapt the technique in “Example: Observing Service Completions” on page 2-19.</p>
Subsystem	<p>In any output signal from the subsystem, observe sample time hits. Alternatively, connect a Discrete Event Signal to Workspace block to any signal inside the subsystem. Then observe the times at which the variable in the workspace indicates a sample time hit of the signal.</p>
Timeout	<p>In the storage block from which the entity times out, observe increases in the #to output signal.</p>
Trigger	<p>Use a branch line to connect the signal to an Instantaneous Event Counting Scope block.</p>
Value change	<p>Use a branch line to connect the signal to an Instantaneous Event Counting Scope block.</p>

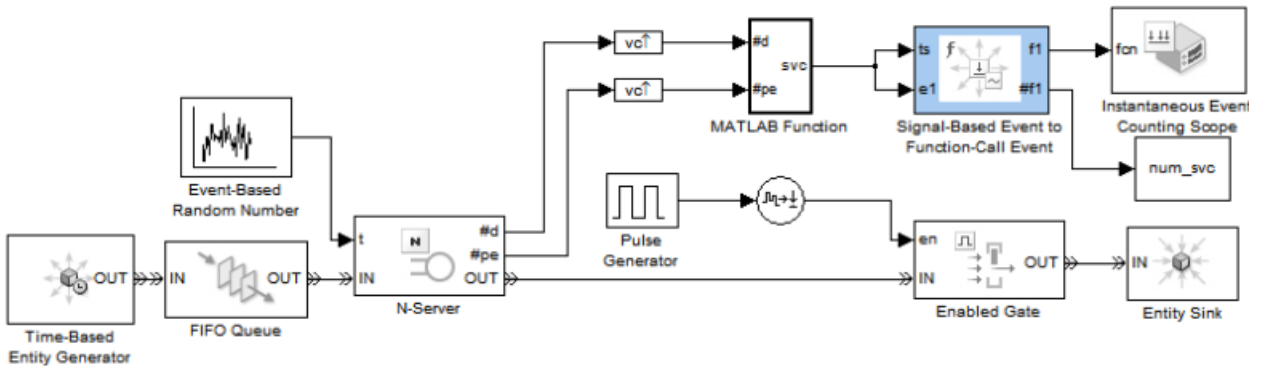
For examples that use one or more of these techniques, see:

- “Example: Plotting Event Counts to Check for Simultaneity” on page 10-15
- “Example: Observing Service Completions” on page 2-19

Also, “Example: Detecting Collisions by Comparing Events” on page 2-22 describes how to use a Signal Latch block to observe which of two types of events are more recent.

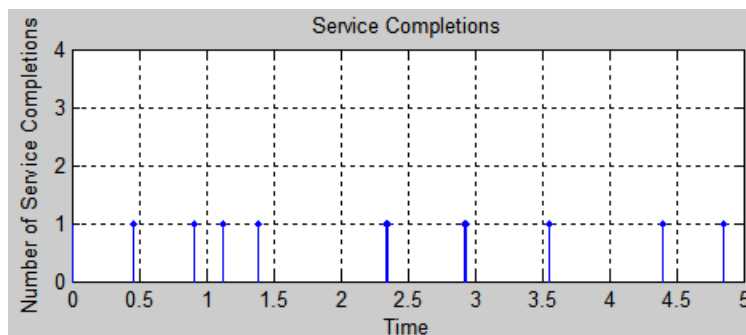
Example: Observing Service Completions

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



Example Results

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



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

```
t_svcpc = num_svc.time
```

The output is:

```
t_svcpc =  
  
      0  
    0.4542  
    0.9077  
    1.1218  
    1.3868  
    2.3430  
    2.3570  
    2.9251  
    2.9370  
    3.5592  
    4.3933  
    4.8554
```

The first value in the `t_svcpc` vector represents the initial value of the `#f1` signal. Subsequent values represent service completion times.

Computation Details

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

- MATLAB Function
- Signal-Based Function-Call Generator

As inputs, the MATLAB Function block uses the `#d` and `#pe` output signals from the server block.

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

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

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

The MATLAB Function block contains this code:

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

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

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

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

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

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

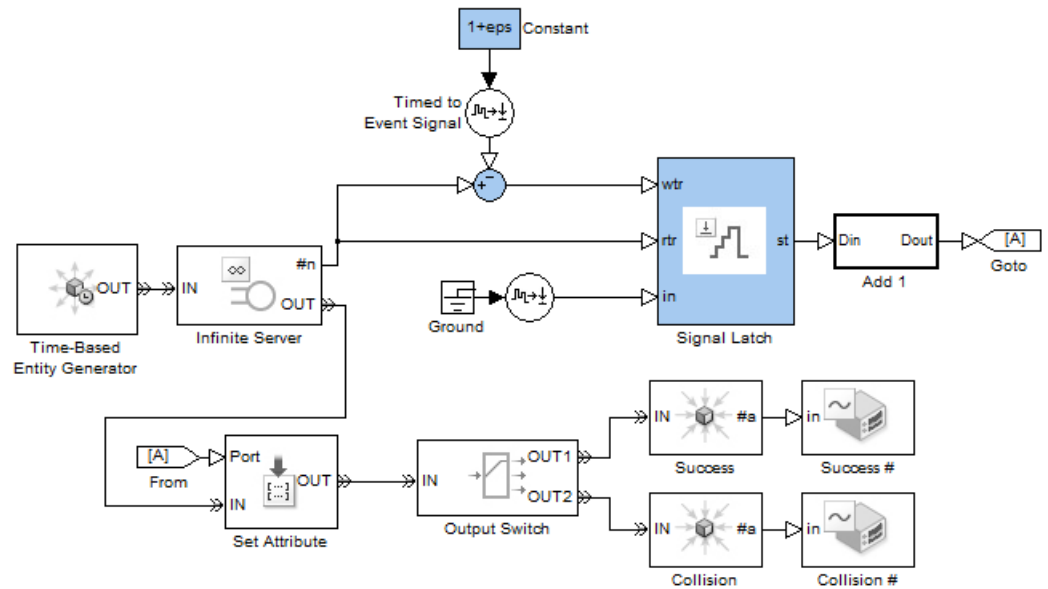
Note The MATLAB Function block does not execute upon decreases in **#pe**. A decrease in **#pe** means that a pending entity has departed. The departure causes a simultaneous increase in **#d**, but the block updates **#pe** before **#d**. Executing the MATLAB Function block upon decreases in **#pe**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

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

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

Example: Detecting Collisions by Comparing Events

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



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

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

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

This example is similar to the CSMA/CD subsystem in the Communication Protocol Modeling in an Ethernet LAN demo.

Generating Function-Call Events

In this section...

“Role of Explicitly Generated Events” on page 2-25

“Generating Events When Other Events Occur” on page 2-25

“Generating Events Using Intergeneration Times” on page 2-27

Role of Explicitly Generated Events

You can generate an event and use it to

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

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

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

Generating Events When Other Events Occur

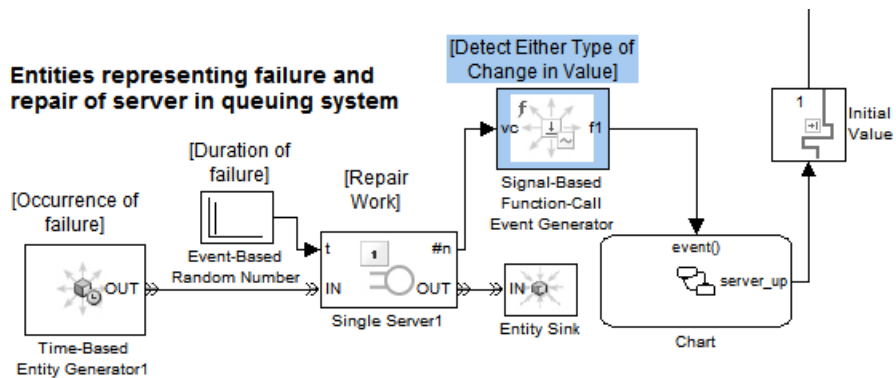
The table below indicates which blocks generate function calls when other events occur.

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

Example: Calling a Stateflow Block Upon Changes in Server Contents

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

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



Using function-calls rather than triggers to call a Stateflow block in discrete-event simulations is preferable because an event-based signal can experience a trigger edge due to a zero-duration value that a time-based block does 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 in a time-based manner, use the Time-Based Function-Call Generator block.

When you want to generate events using intergeneration times from a signal or a statistical distribution, set the **Event generation mode** parameter of the Time-Based Function-Call Generator block to **Period** from **port**. Connect the signal or statistical distribution to the input port **t** of the Time-Based Function-Call Generator block.

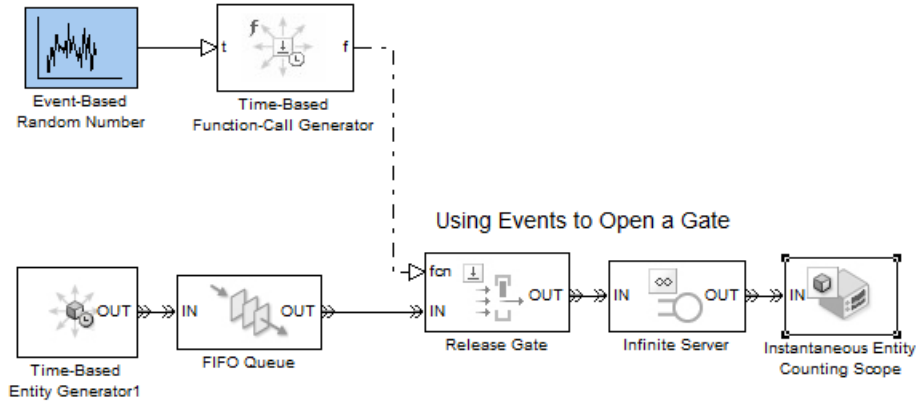
When the intergeneration time is constant, set the **Event generation mode** parameter of the Time-Based Function-Call Generator to **Period** from **dialog**. In the block dialog box, enter a value for the **Period** parameter. The period is the time interval between successive function-call events that the block generates, in seconds.

Opening a Gate Upon Random Events

The following model uses the Time-Based Entity Generator block to generate entities that enter a gated queuing system. In this case, the Time-Based Function-Call Generator block generates function-call events with intergeneration times drawn from a statistical distribution. Each of these function-call events triggers the Release Gate block and an entity advances from the queue.

Opening a Gate Upon Random Events

Generating Events with Random Intergeneration Times



Manipulating Events

In this section...

- “Reasons to Manipulate Events” on page 2-29
- “Blocks for Manipulating Events” on page 2-31
- “Creating a Union of Multiple Events” on page 2-31
- “Translating Events to Control the Processing Sequence” on page 2-34
- “Conditionalizing Events” on page 2-36

Reasons to Manipulate Events

You can manipulate events to accomplish any of these goals:

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

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

- To create a union of events from multiple sources. See “Creating a Union of Multiple Events” on page 2-31.
- To prioritize the reaction to one event relative to other simultaneous events. See “Translating Events to Control the Processing Sequence” on page 2-34.
- To delay the reactions to events. See the **Function-call time delay** parameter on the Signal-Based Function-Call Generator blocks reference page.
- To conditionalize the reactions to events. See “Conditionalizing Events” on page 2-36.

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 Function-Call Generator Event
Signal-based event	Signal-Based Function-Call Generator
Function call	Signal-Based Function-Call Generator
	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 and below.

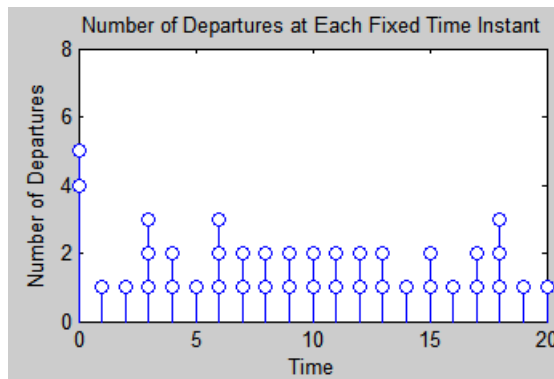
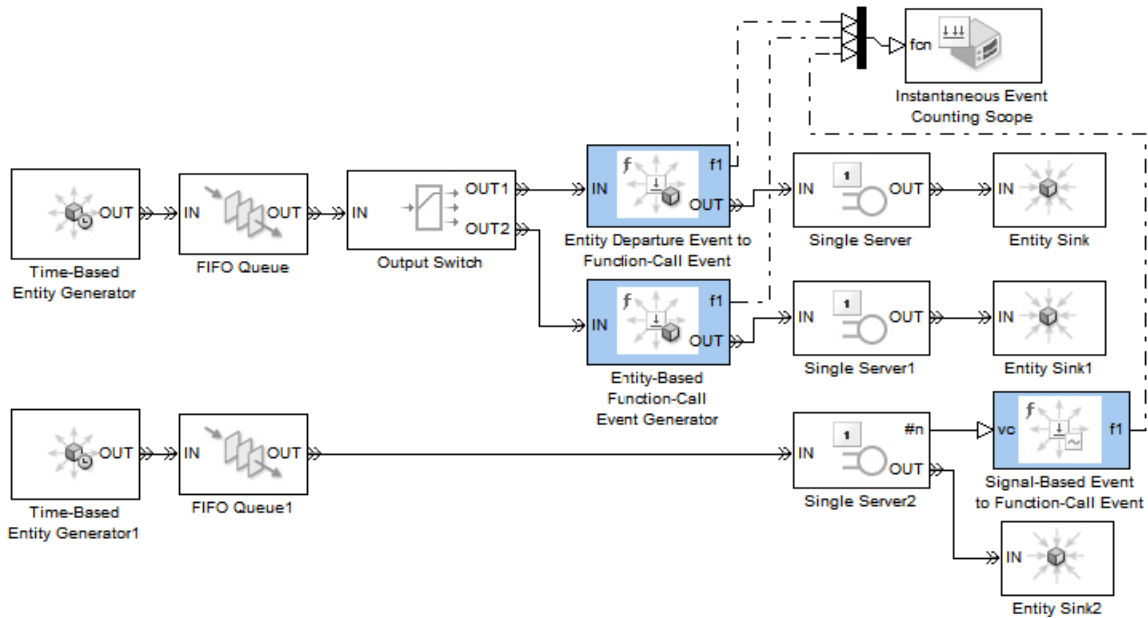
Example: Counting Events from Multiple Sources

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.

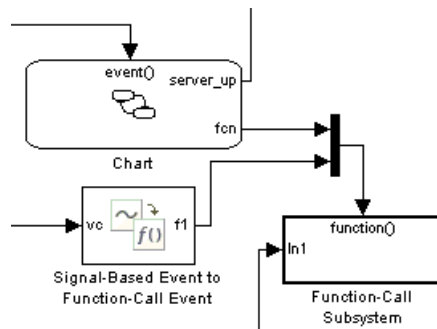
The three server paths use these methods for translating an entity arrival into a function call:

- One path uses the Entity Departure Function-Call Generator 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 Function-Call Generator 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 Function-Call Subsystem block to detect function calls from one or more sources. The following model fragment uses an event translation block to convert a signal-based event into a function call. When you perform this conversion, you create a subsystem that detects a function call from a Stateflow block and a signal-based event from another source. When either the Stateflow block generates a function call, or the signal connected to the `vc` port of the Signal-Based Function-Call Generator block changes, the model executes this subsystem. If both events occur simultaneously, the model executes the subsystem twice.



A similar example is in .

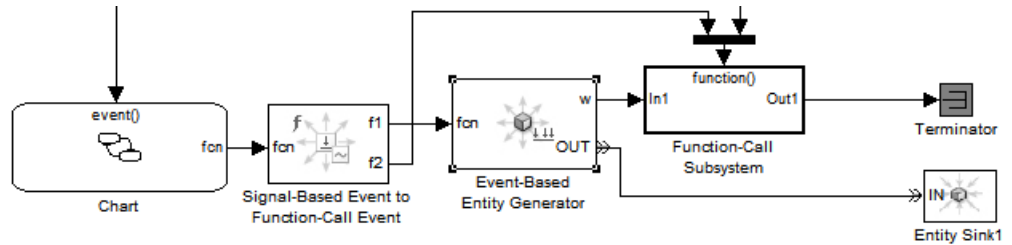
Translating Events to Control the Processing Sequence

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

Example: Issuing Two Function Calls in Sequence

In the next model, entity generation and the execution of a function-call subsystem can occur simultaneously. At such times, it is important that the entity generation occur first, so that the entity generator updates the value of the `w` signal before the function-call subsystem uses `w` in its computation. This model ensures a correct processing sequence by using the

same Signal-Based Function-Call Generator block to issue both function calls and by relying on the fact that the block always issues the **f1** function call before the **f2** function call.



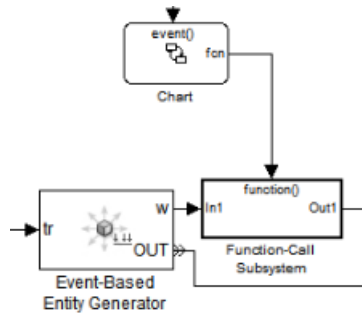
In this example, the Signal-Based Function-Call Generator block has this configuration:

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

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

Example: Generating a Function Call with an Event Priority

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



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

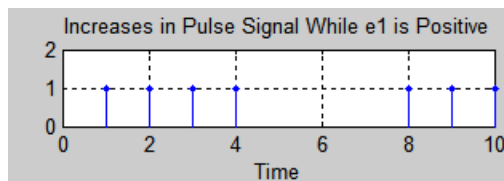
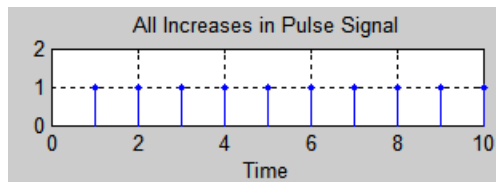
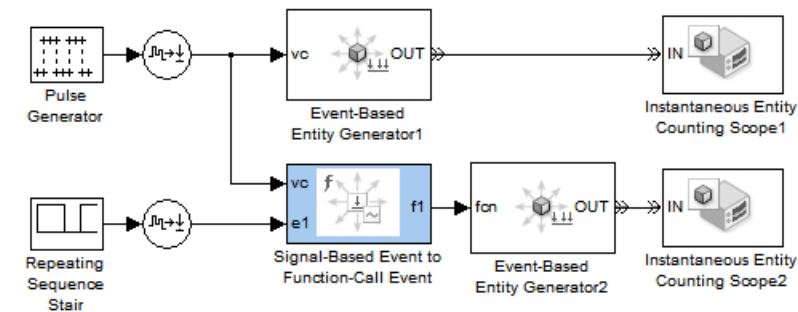
Conditionalizing Events

The Entity Departure Function-Call Generator Event and Signal-Based Function-Call Generator 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 14-46 uses conditional function calls to prevent division-by-zero warnings.
- Model an inoperative state of a component of your system. See the next example.

Example: Modeling Periodic Shutdown of an Entity Generator

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



Managing Simultaneous Events

- “Overview of Simultaneous Events” on page 3-2
- “Exploring Simultaneous Events” on page 3-4
- “Choosing an Approach for Simultaneous Events” on page 3-7
- “Assigning Event Priorities” on page 3-8
- “Example: Choices of Values for Event Priorities” on page 3-11
- “Example: Effects of Specifying Event Priorities” on page 3-25

Overview of Simultaneous Events

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

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

To Read About...	Refer to...	Description
Background	“Supported Events in SimEvents Models” on page 2-2	Overview of event types and the event calendar
Behavior	“Event Sequencing” on page 14-9	How the application determines which events to process first, when time and causality alone do not specify a unique sequence
Examples	“Example: Event Calendar Usage for a Queue-Server Model” on page 2-7	Illustrates basic functionality of the event calendar
	“Example: Choices of Values for Event Priorities” on page 3-11	Examines the role of event priority values
	“Example: Effects of Specifying Event Priorities” on page 3-25	Compares simulation behaviors when you specify and do not specify event priorities
Tips	“Choosing an Approach for Simultaneous Events” on page 3-7 and “Tips for Choosing Event Priority Values” on page 3-8	Tips to help you decide how to configure your model
Techniques	“Exploring Simultaneous Events” on page 3-4 and “Assigning Event Priorities” on page 3-8	Viewing behavior and working with explicit event priorities

When one of the simultaneous events is a signal update, information in “Choosing How to Resolve Simultaneous Signal Updates” on page 14-14 is also relevant.

Exploring Simultaneous Events

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

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

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

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

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

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

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

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

Using Nearby Breakpoints to Focus on a Particular Time

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

```
tbreak(3)
```

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

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

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

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

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

For Further Information

- “Overview of the SimEvents Debugger” on page 13-3 — More information about the debugger

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

Choosing an Approach for Simultaneous Events

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

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

For details on how the application treats simultaneous events, see “Processing Sequence for Simultaneous Events” on page 14-9 and “Resolution Sequence for Input Signals” on page 14-15.

Assigning Event Priorities

In this section...

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

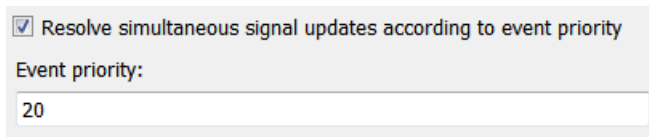
“Tips for Choosing Event Priority Values” on page 3-8

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

Procedure for Assigning Event Priorities

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

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



Resolve simultaneous signal updates according to event priority

Event priority:

20

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

Note Some events have event priorities that are not numerical, such as SYS1 and SYS2. For more information about these priority values, see “System-Priority Events on the Event Calendar” on page 14-20 and “Processing Sequence for Simultaneous Events” on page 14-9.

Tips for Choosing Event Priority Values

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

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

- If Event Y does not have a numerical event priority, then any value for the event priority of Event X causes Event X to be processed later. (See “Processing Sequence for Simultaneous Events” on page 14-9 for details.)
- If Event Y has a numerical event priority, then choosing a larger value for the event priority of Event X causes Event X to be processed later. Simultaneous events having distinct numerical event priorities are processed in ascending order of the event priority values.
- Leaving gaps in the set of numerical values you choose lets you include additional events that require intermediate-value priorities. For example, if Event Y has priority 1 and Event X has priority 2, then you cannot force an Event Z to be processed after Event Y and before Event X. On the other hand, priority values of 100 and 200 would better accommodate future growth of your model.

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

Procedure for Specifying Equal-Priority Behavior

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

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

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

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

Example: Choices of Values for Event Priorities

In this section...

“Overview of Example” on page 3-11

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

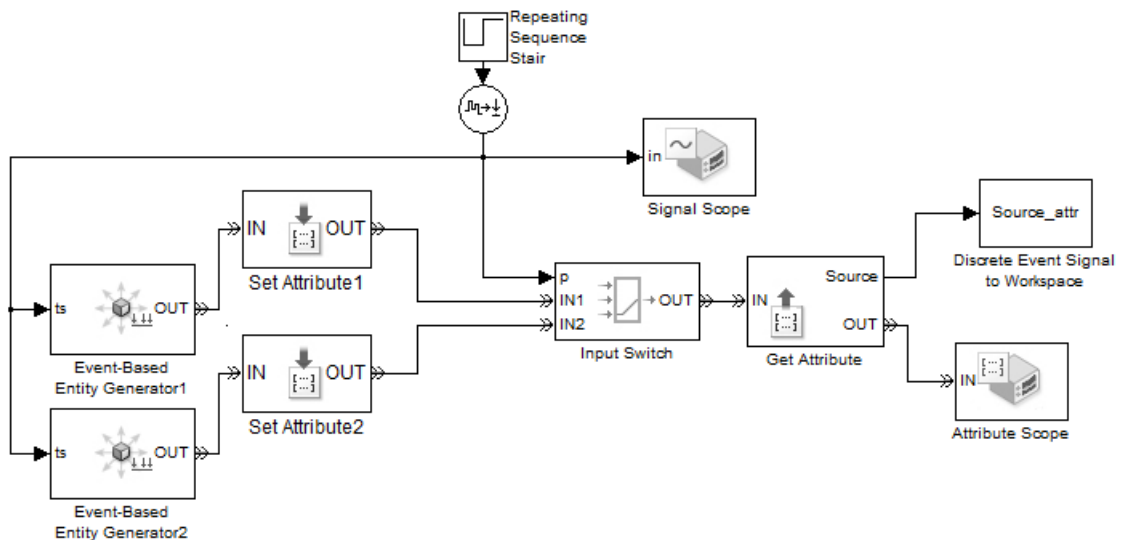
“Selecting a Port First” on page 3-12

“Generating Entities First” on page 3-19

“Randomly Selecting a Sequence” on page 3-24

Overview of Example

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



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

- The top entity generator generates an entity.
- The bottom entity generator generates an entity.
- The Input Switch block selects a different entity input port.

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

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

Arbitrary Resolution of Signal Updates

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

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

Selecting a Port First

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

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

The following describes what happens at $T=1, 2, 3$ using messages from the simulation log of the SimEvents debugger. To learn more about the debugger, see “Overview of the SimEvents Debugger” on page 13-3.

Behavior at T=1

- The output signal from the Repeating Sequence Stair block changes from 1 to 2 and blocks that connect to it detect the relevant update.

```
%=====
Detected Sample Time Hit                Time = 1.0000000000000000
: Block = Input Switch
```

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

```
%-----%
Discrete-Event System ID: 0 Highlight
  ID      EventTime      EventType      Priority Entity      Block
=> ev6    1.0000000000000000 PortSelection    200   <none>   Input Switch
      ev4    1.0000000000000000 EntityGeneration 300   <none>   Event-Based
Entity Generator1
      ev5    1.0000000000000000 EntityGeneration 310   <none>   Event-Based
Entity Generator2
```

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

```
%=====
Executing PortSelection Event (ev6)      Time = 1.0000000000000000
: Entity = <none>                        Priority = 200
: Block = Input Switch
```

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

```
%=====
Executing EntityGeneration Event (ev4)   Time = 1.0000000000000000
: Entity = <none>                        Priority = 300
: Block = Event-Based Entity Generator1
%.....%
Generating Entity (en1)
```

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

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

```
%=====
Executing EntityGeneration Event (ev5)           Time = 1.000000000000000
: Entity = <none>                               Priority = 310
: Block = Event-Based Entity Generator2
%.....%
Generating Entity (en2)
: Block = Event-Based Entity Generator2
%.....%
Entity Advancing (en2)
: From = Event-Based Entity Generator2
: To = Set Attribute2
%.....%
[...other messages...]
%.....%
Entity Advancing (en2)
: From = Get Attribute
: To = Attribute Scope
%.....%
Destroying Entity (en2)
: Block = Attribute Scope
```

Behavior at T=2

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

```
%-----%
Discrete-Event System ID: 0 Highlight
  ID      EventTime      EventType      Priority Entity      Block
=> ev10   2.000000000000000   PortSelection   200    <none>   Input Switch
      ev8    2.000000000000000   EntityGeneration 300    <none>   Event-Based
Entity Generator1
      ev9    2.000000000000000   EntityGeneration 310    <none>   Event-Based
Entity Generator2
```

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

```

%=====
Executing PortSelection Event (ev10)                Time = 2.0000000000000000
: Entity = <none>                                  Priority = 200
: Block = Input Switch
  %.....%
  Entity Advancing (en1)
  : From = Event-Based Entity Generator1
  : To   = Set Attribute1
  %.....%
[...other messages...]
  %.....%
  Entity Advancing (en1)
  : From = Get Attribute
  : To   = Attribute Scope
  %.....%
  Destroying Entity (en1)
  : Block = Attribute Scope

```

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

```

%=====
Executing EntityGeneration Event (ev8)             Time = 2.0000000000000000
: Entity = <none>                                  Priority = 300
: Block = Event-Based Entity Generator1
  %.....%
  Generating Entity (en3)
  : Block = Event-Based Entity Generator1
  %.....%
  Entity Advancing (en3)
  : From = Event-Based Entity Generator1
  : To   = Set Attribute1
  %.....%
[...other messages...]
  %.....%
  Entity Advancing (en3)

```

```

: From = Get Attribute
: To   = Attribute Scope
%.....%
Destroying Entity (en3)
: Block = Attribute Scopee

```

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

```

%=====
Executing EntityGeneration Event (ev9)           Time = 2.000000000000000
: Entity = <none>                               Priority = 310
: Block  = Event-Based Entity Generator2
%.....%
Generating Entity (en4)
: Block  = Event-Based Entity Generator2

```

Behavior at T=3

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

```

%-.....-%
Discrete-Event System ID: 0 Highlight

```

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

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

```

%=====
Executing PortSelection Event (ev15)           Time = 3.000000000000000
: Entity = <none>                               Priority = 200
: Block  = Input Switch
%.....%
Entity Advancing (en4)

```

```

: From = Event-Based Entity Generator2
: To   = Set Attribute2
%.....%
[...other messages...]
%.....%
Entity Advancing (en4)
: From = Get Attribute
: To   = Attribute Scope
%.....%
Destroying Entity (en4)
: Block = Attribute Scope

```

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

```

%=====
Executing EntityGeneration Event (ev13)           Time = 3.000000000000000
: Entity = <none>                                Priority = 300
: Block  = Event-Based Entity Generator1
%.....%
Generating Entity (en5)
: Block  = Event-Based Entity Generator1

```

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

```

%=====
Executing EntityGeneration Event (ev14)           Time = 3.000000000000000
: Entity = <none>                                Priority = 310
: Block  = Event-Based Entity Generator2
%.....%
Generating Entity (en6)
: Block  = Event-Based Entity Generator2
%.....%
Entity Advancing (en6)
: From = Event-Based Entity Generator2
: To   = Set Attribute2
%.....%
[...other messages...]

```

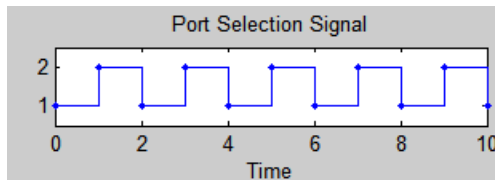
```

%.....%
Entity Advancing (en6)
: From = Get Attribute
: To  = Attribute Scope
%.....%
Destroying Entity (en6)
: Block = Attribute Scope

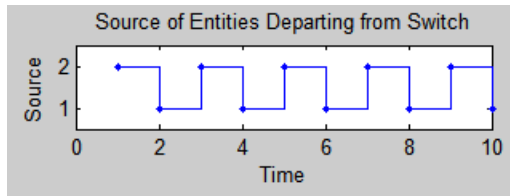
```

Evidence from Plots and Signals

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



Port Selection Signal



Switch Departures When Port Selection Is Processed First

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

```
ans =
```

```

0    0
1    2
2    1

```



```

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

Generating Entities First

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

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

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

Behavior at T=1

- The output signal from the Repeating Sequence Stair block changes from 1 to 2 and blocks that connect to it detect the relevant update.

```

%=====
Detected Sample Time Hit                               Time = 1.0000000000000000
: Block = Input Switch
    
```

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

```

%-----%
Discrete-Event System ID: 0 Highlight
  ID      EventTime      EventType      Priority Entity      Block
=> ev4    1.0000000000000000 EntityGeneration 300    <none>    Event-Based
Entity Generator1
  ev5     1.0000000000000000 EntityGeneration 310    <none>    Event-Based
Entity Generator2
  ev6     1.0000000000000000 PortSelection    4000   <none>    Input Switch

```

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

```

%=====
Executing EntityGeneration Event (ev4)          Time = 1.0000000000000000
: Entity = <none>                             Priority = 300
: Block = Event-Based Entity Generator1
%.....%
Generating Entity (en1)
: Block = Event-Based Entity Generator1
%.....%
Entity Advancing (en1)
: From = Event-Based Entity Generator1
: To   = Set Attribute1
%.....%
[...other messages...]
%.....%
Entity Advancing (en1)
: From = Get Attribute
: To   = Attribute Scope

```

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

```

%=====
Executing EntityGeneration Event (ev5)          Time = 1.0000000000000000
: Entity = <none>                             Priority = 310
: Block = Event-Based Entity Generator2

```

```
%.....%
Generating Entity (en2)
: Block = Event-Based Entity Generator2
```

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

```
%=====
Executing PortSelection Event (ev3)           Time = 0.000000000000000
: Entity = <none>                             Priority = 4000
: Block = Input Switch2

%.....%
[...other messages...]

%.....%
Entity Advancing (en1)
: From = Get Attribute
: To = Attribute Scope
```

Behavior at T=2

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

```
%-----%
Discrete-Event System ID: 0 Highlight
  ID      EventTime      EventType      Priority Entity      Block
=> ev9    2.000000000000000  EntityGeneration  300    <none>    Event-Based
Entity Generator1
  ev10    2.000000000000000  EntityGeneration  310    <none>    Event-Based
Entity Generator2
  ev11    2.000000000000000  PortSelection    4000   <none>    Input Switch
```

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

```
%=====
Executing EntityGeneration Event (ev9)       Time = 2.000000000000000
: Entity = <none>                             Priority = 300
: Block = Event-Based Entity Generator1
```

```

%.....%
Generating Entity (en3)
: Block = Event-Based Entity Generator1

```

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

```

%=====
Executing EntityGeneration Event (ev10)           Time = 2.0000000000000000
: Entity = <none>                                Priority = 310
: Block = Event-Based Entity Generator2
%.....%
Generating Entity (en4)
: Block = Event-Based Entity Generator2
%.....%
Entity Advancing (en4)
: From = Event-Based Entity Generator2
: To = Set Attribute2
%.....%
[...other messages...]
%.....%
Entity Advancing (en4)
: From = Get Attribute
: To = Attribute Scope

```

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

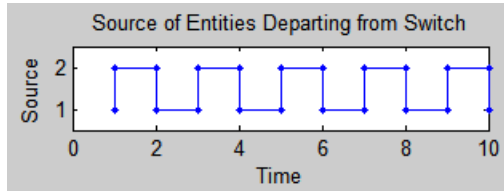
```

%=====
Executing PortSelection Event (ev11)           Time = 2.0000000000000000
: Entity = <none>                                Priority = 4000
: Block = Input Switch
%.....%
[...other messages...]
%.....%
Entity Advancing (en5)
: From = Get Attribute
: To = Attribute Scope

```

Plots and Signals

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



Switch Departures When Entity Generations Are Processed First

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

```
ans =
```

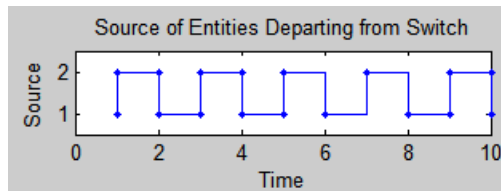
```

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

Randomly Selecting a Sequence

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

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



Switch Departures When Processing Sequence is Random

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

```
ans =
```

```

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

Example: Effects of Specifying Event Priorities

In this section...

“Overview of the Example” on page 3-25

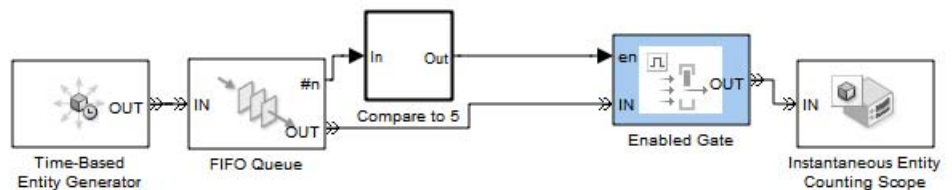
“Default Behavior” on page 3-26

“Deferring Gate Events” on page 3-27

Overview of the Example

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

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



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

Default Behavior

By default, the Enabled Gate block at the top level has the **Resolve simultaneous signal updates according to event priority** option not selected. This block does not have any events with numerical priority values. The simulation behaves as follows:

Simulation Behavior

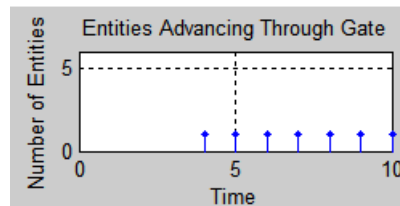
- 1 The queue accumulates entities until it updates the queue length signal, #n, to 5.
- 2 The subsystem executes immediately because the execution is not scheduled on the event calendar. The subsystem reports, and the gate detects, that the queue length is at the threshold.
- 3 The gate schedules an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

Note Some events have event priorities that are not numerical, such as SYS1 and SYS2. For more information about these priority values, see “System-Priority Events on the Event Calendar” on page 14-20 and “Processing Sequence for Simultaneous Events” on page 14-9.

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

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

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



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

Deferring Gate Events

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

Procedure

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

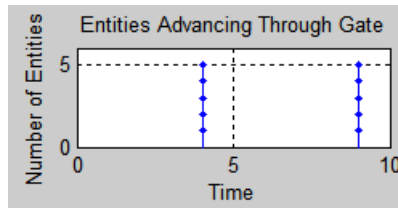
The change causes the gate to prioritize its events differently. The application processes events with priority SYS1 before processing events with numerical priority values. As a result, the simulation behaves as follows:

Simulation Behavior

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

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

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



Working with Signals

- “Role of Event-Based Signals in SimEvents Models” on page 4-2
- “Generating Random Signals” on page 4-4
- “Using Data Sets to Create Event-Based Signals” on page 4-7
- “Converting Between Time-Based and Event-Based Signals” on page 4-10
- “Manipulating Signals” on page 4-14
- “Sending Data to the MATLAB Workspace” on page 4-17
- “Working with Multivalued Signals” on page 4-20
- “Working with Bus Signals” on page 4-24

Role of Event-Based Signals in SimEvents Models

In this section...

“Overview of Event-Based Signals” on page 4-2

“Comparison with Time-Based Signals” on page 4-2

“Tips for Using Event-Based Signals” on page 4-3

“Signal Restrictions for Event-Based Signals” on page 4-3

Overview of Event-Based Signals

Discrete-event simulations often involve signals that change when events occur; for example, the number of entities in a server is a statistical output signal from a server block and the signal value changes when an entity arrives at or departs from the server. An event-based signal is a signal that can change in response to discrete events. A discrete-event system is one in which signals change when events occur. One model can have one or more discrete-event systems.

Most output signals from SimEvents blocks are event-based signals. Exceptions are the output signals from the Event to Timed Signal and Event to Timed Function-Call blocks, whose explicit purpose is to convert event-based signals into time-based signals.

Comparison with Time-Based Signals

Unlike time-based signals, event-based signals:

- Do not have a true sample time.
- 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.

Tips for Using Event-Based Signals

- The sample time coloration feature makes the signal connection line gray, which normally indicates fixed in minor step. Also, querying the sample time indicates that the signal is fixed in minor step.
- If your model includes both event-based and time-based signals, see “Converting Between Time-Based and Event-Based Signals” on page 4-10.

Signal Restrictions for Event-Based Signals

- In the SimEvents libraries, most blocks process only signals whose data type is `double`. (Exceptions are the Timed to Event Signal and Event to Timed Signal blocks. These blocks accept input signals of any data type and produce output signals of the same data type.)

To convert between data types, use the Data Type Conversion block.

- In the SimEvents libraries, blocks process only fixed-size signals and do not support variable-size signals.
- An event-based signal cannot be an element of a nonvirtual bus while retaining event-based timing. The reason is that a nonvirtual bus is an inherently time-based signal.

Generating Random Signals

In this section...
“Generating Random Event-Based Signals” on page 4-4
“Examples of Random Event-Based Signals” on page 4-5

Generating Random 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 14-33. You must connect the Event-Based Random Number block, directly or indirectly, to exactly one notifying port. The notifying port tells this block when to generate a new output value. An indirect connection must be via a block listed in “Computational Blocks” on page 14-53 having exactly one input signal and no function-call output signals.

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:

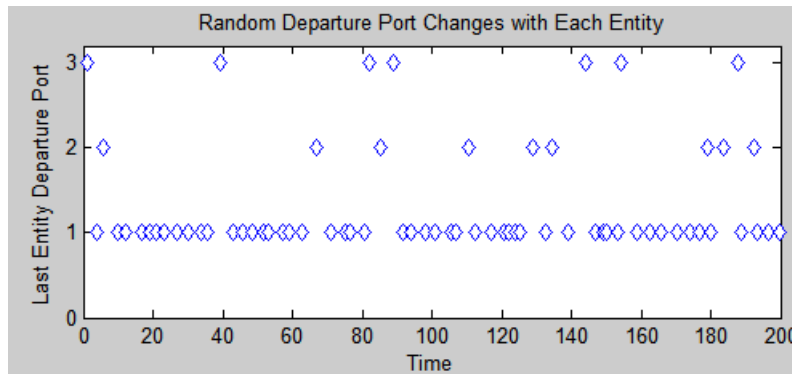
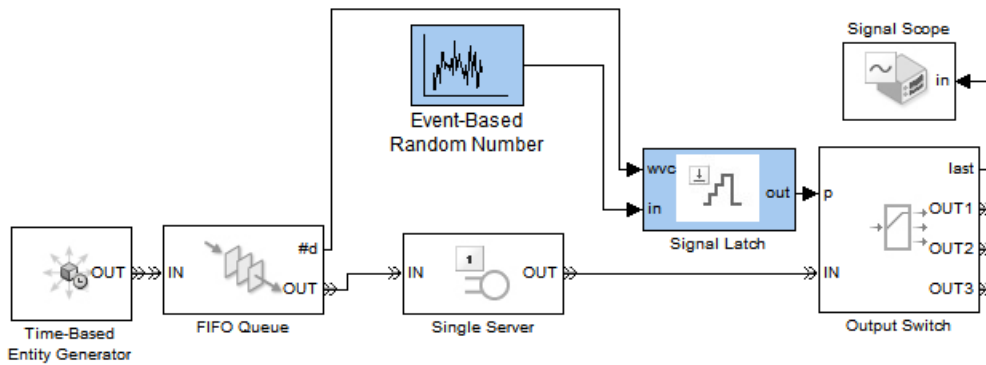
- “Example: Using an Arbitrary Discrete Distribution as Intergeneration Time” in the SimEvents getting started documentation
- “Example: A Packet Switch” in the SimEvents getting started documentation
- “Example: Using Random Service Times in a Queuing System” in the SimEvents getting started documentation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

The model in “Example: Compound Switching Logic” on page 6-10 also illustrates how to use the Signal Latch block as described in “Generating Random Signals Based on Arbitrary Events” on page 4-4, to generate a random number upon each departure from an Input Switch block.

The models in “Example: Invalid Connection of Event-Based Random Number Generator” on page 13-88 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 , 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.



Using Data Sets to Create Event-Based Signals

In this section...
“Behavior of the Event-Based Sequence Block” on page 4-7
“Generating Sequences Based on Arbitrary Events” on page 4-8

Behavior of the Event-Based Sequence Block

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

More generally, you can use the Event-Based Sequence block to incorporate your data into a simulation via event-based signals, where the block infers from a subsequent block when to output the next data value. You must connect the Event-Based Sequence block, directly or indirectly, to exactly one notifying port. The *t* input port of a Single Server block is an example of a notifying port; for a list, see “Notifying Ports” on page 14-33. The notifying port tells this block when to generate a new output value. An indirect connection must be via a block listed in “Computational Blocks” on page 14-53 having exactly one input signal and no function-call output signals.

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

For examples using this block, see these sections:

- “Specifying Generation Times for Entities” on page 1-4
- “Example: Counting Simultaneous Departures from a Server” on page 1-20
- “Example: Setting Attributes” on page 1-8

Generating Sequences Based on Arbitrary Events

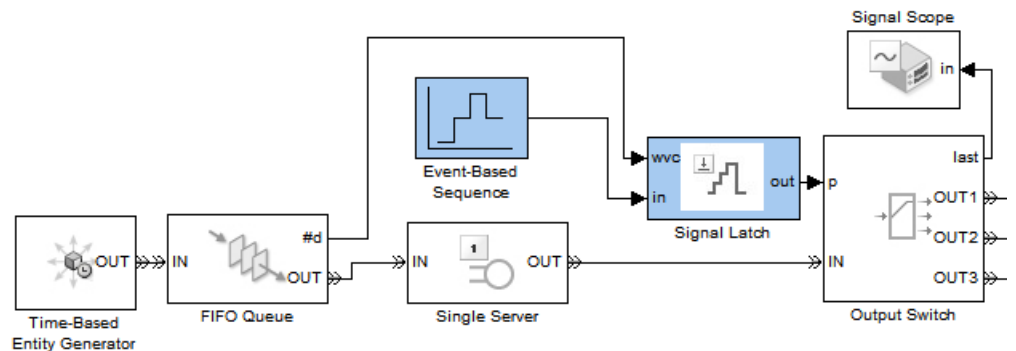
A flexible way to generate event-based sequences is to use the Signal Latch block to indicate explicitly which events cause the Event-Based Sequence block to generate a new output value. Use this procedure:

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

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

Example

You can modify the model in “Example: Creating a Random Signal for Switching” on page 4-6 by replacing the Event-Based Random Number block with the Event-Based Sequence block.



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

Converting Between Time-Based and Event-Based Signals

In this section...

“When to Convert Signals” on page 4-10

“When Not to Convert Signals” on page 4-11

“How to Convert Signals” on page 4-11

“Signal Conversion When Using Custom Library Blocks” on page 4-12

When to Convert Signals

Time-based signals and event-based signals have different characteristics, as described in “Comparison with Time-Based Signals” on page 4-2. Here are some indications that you might need to convert a time-based signal into an event-based signal, or vice versa:

- You want to connect a time-based signal to an input port of a SimEvents block. Instead of making the connection directly, you must insert a conversion block before the input port of the SimEvents block. The relevant conversion blocks are Timed to Event Signal for data signals and Timed to Event Function-Call for function-call signals.
- You are using data from an event-based signal to affect time-based dynamics. Instead of making the connection directly, you must insert a conversion block before the input port of the SimEvents block. The relevant conversion blocks are Event to Timed Signal for data signals and Event to Timed Function-Call for function-call signals.
- You want to perform a computation involving both time-based signals and event-based signals. You must decide how you want the software to perform the computation, and then insert conversion blocks as needed:
 - If you want the software to perform the computation in an event-based manner, insert a conversion block on each time-based signal line that is an input to the computation. The relevant conversion blocks are Timed to Event Signal for numerical signals and Timed to Event Function-Call for function-call signals.
 - If you want the software to perform the computation in a time-based manner, insert a conversion block on each event-based signal line that is

an input to the computation. The relevant conversion blocks are Event to Timed Signal for numerical signals and Event to Timed Function-Call for function-call signals.

When Not to Convert Signals

Here are some indications that converting a time-based signal into an event-based signal, or vice versa, might be inappropriate:

- You want to convert an event-based signal into a time-based signal because a computational block is not in the list of supported blocks in “Computational Blocks” on page 14-53. Before converting the signal, consider whether you want the software to perform the computation in an event-based manner. If you do, try connecting your event-based input signal to an Atomic Subsystem block and putting the computational block inside the subsystem.
- Converting an event-based signal into a time-based signal causes the loss of zero-duration values that you want to retain.
- Converting an event-based signal into a time-based signal is unnecessary and less efficient. If the software is capable of performing a computation on that signal in an event-based manner with equivalent results, the conversion might be unnecessary. For details, see Chapter 9, “Computations on Event-Based Signals” and “Computational Blocks” on page 14-53. If the conversion causes computations to occur when the inputs to the computations have not changed, the conversion might reduce simulation efficiency.

Note When using gateway blocks, you might notice behavior differences. In particular, you will notice more sample time hits in time-based systems than event-based systems.

How to Convert Signals

Use the blocks in the tables to convert between time-based and event-based signals. These blocks exist in the Gateways library. The documentation refers to these blocks collectively as gateway blocks.

Converting Numerical Data Signals

From	To	Conversion Block
Time-based signal	Event-based signal	Timed to Event Signal
Event-based signal	Time-based signal	Event to Timed Signal

Converting Function-Call Signals

From	To	Conversion Block
Time-based signal	Event-based signal	Timed to Event Function-Call
Event-based signal	Time-based signal	Event to Timed Function-Call

Signal Conversion When Using Custom Library Blocks

When you use custom library blocks in a SimEvents model, be aware of how you use time-based and event-based signals with such blocks. Consider these best-practice guidelines:

- If you are designing a new custom library block, consider using a naming convention for input and output ports, which indicates the type of signal intended at each port.
 - You can create a custom library block which performs computations that occur in either an event-based or time-based manner. In some cases, you might want this flexibility. However, using a name such as *IN1-ESignal* for the inport of a custom library block, gives a clear indication to other users that an event-based signal is expected. This approach may eliminate situations where custom library blocks intended for use with event-based signals are used instead with time-based signals, and vice versa.
- If you are using a custom library block in an event-based model and you require conversion between a time-based signal and an event-based signal, placing the conversion block within the custom library block is not recommended:

- It will reduce the flexibility of the custom block, by always converting the incoming signal to either a time-based, or event-based signal, depending on the conversion block used.
- It may confuse other users unfamiliar with the design of the custom library block.
- If signal conversion is required to use a custom library block in an event-based model, then the required conversion block should be placed outside the custom library block. Inspect the model and determine where in the signal computation path to place the conversion block.

Manipulating Signals

In this section...

“Specifying Initial Values of Event-Based Signals” on page 4-14

“Example: Resampling a Signal Based on Events” on page 4-15

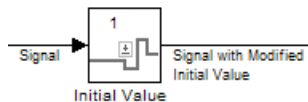
Specifying Initial Values of Event-Based Signals

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

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

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

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



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

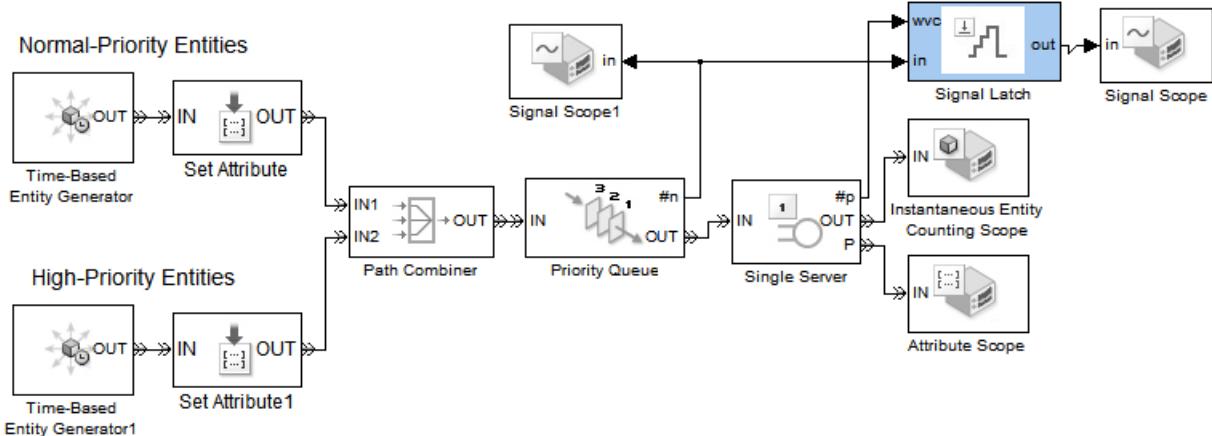
The following examples illustrate this technique:

- “Example: Controlling Joint Availability of Two Servers” on page 7-4 initializes an event-based signal for use in a feedback loop.

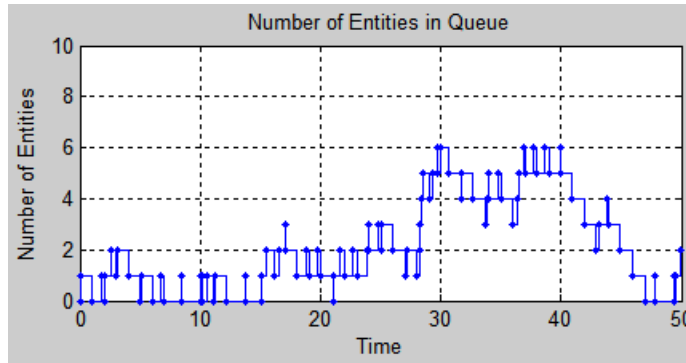
- “Example: Failure and Repair of a Server” on page 5-22 initializes an event-based signal that is the output of a Stateflow block.

Example: Resampling a Signal Based on Events

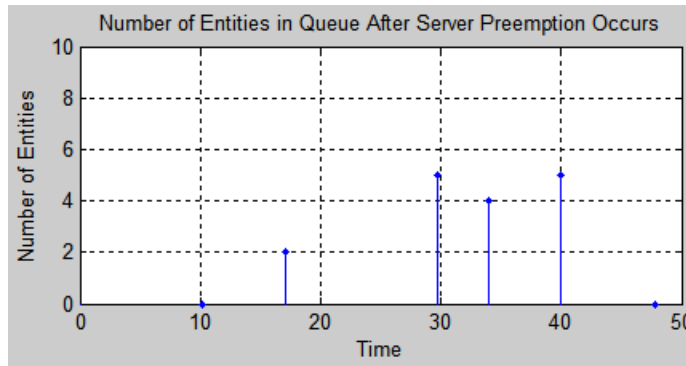
The example below contains a server that supports preemption of normal-priority entities by high-priority entities. This is similar to “Example: Preemption by High-Priority Entities” on page 5-11. Suppose that a preemption and the subsequent service of a high-priority entity represent a time interval 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. Note that new head of queue events also affect the number of entities in the queue.



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



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



Sending Data to the MATLAB Workspace

In this section...

“Behavior of the Discrete Event Signal to Workspace Block” on page 4-17

“Example: Sending Queue Length to the Workspace” on page 4-17

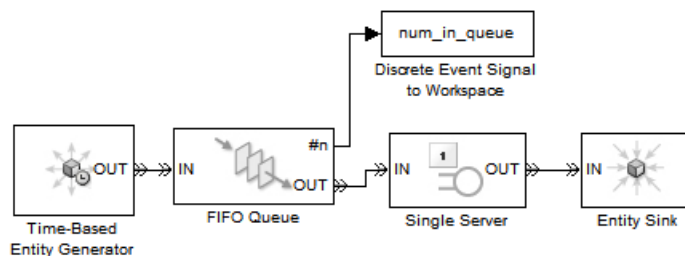
Behavior of the Discrete Event Signal to Workspace Block

The Discrete Event Signal to Workspace block writes event-based signals to the MATLAB workspace when the simulation stops or pauses. One way to pause a running simulation is to select **Simulation > Pause**.

Note To learn how to read data from the workspace during a discrete-event simulation, see “Using Data Sets to Create Event-Based Signals” on page 4-7.

Example: Sending Queue Length to the Workspace

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



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

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

The output 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. The first row of the `times_values` matrix represents the initial value of the `#n` signal.

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

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

Working with Multivalued Signals

In this section...
“Zero-Duration Values of Signals” on page 4-20
“Importance of Zero-Duration Values” on page 4-21
“Detecting Zero-Duration Values” on page 4-21

Zero-Duration Values of Signals

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

Scenario: Server Departure and New Arrival

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

Scenario: Queue Length

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

Importance of Zero-Duration Values

The values of signals, even values that do not persist for a positive duration, can help you understand or debug your simulations. In the example described in “Scenario: Server Departure and New Arrival” on page 4-20, the zero-duration value of 0 in the signal tells you that the server experienced a departure. If the signal assumed only the value 1 at time T (because 1 is the final value at time T), then the constant values before, at, and after time T would fail to indicate the departure. While you could use a departure count signal to detect departures specifically, the zero-duration value in the number-in-block signal provides you with more information in a single signal.

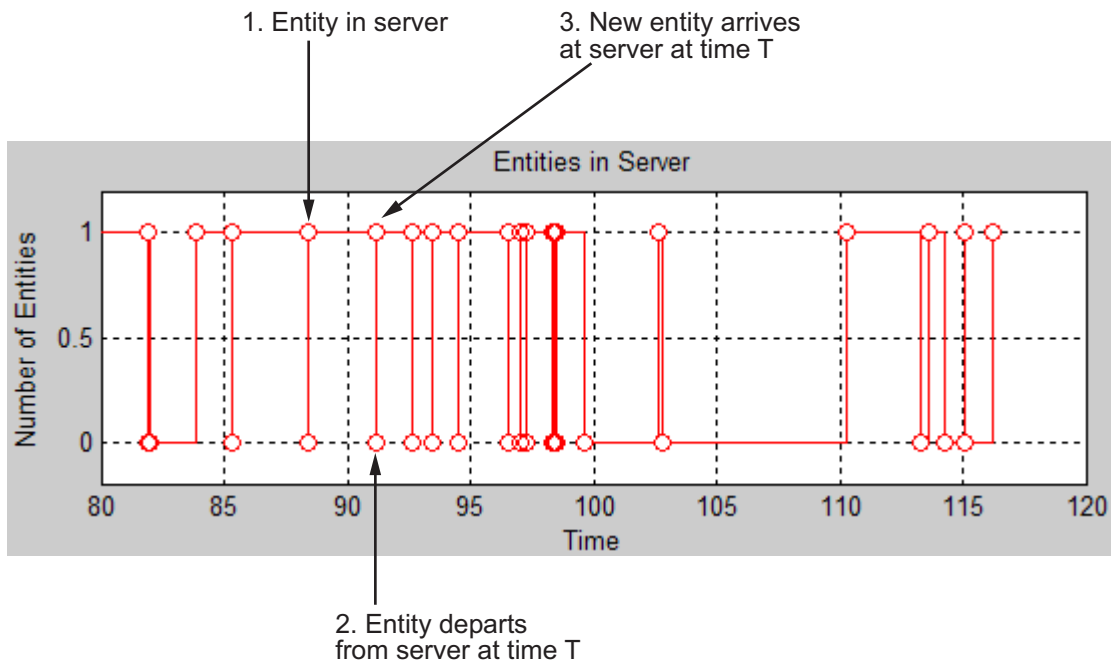
Detecting Zero-Duration Values

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

- “Plotting Signals that Exhibit Zero-Duration Values” on page 4-21
- “Plotting the Number of Signal Changes Per Time Instant” on page 4-22
- “Viewing Zero-Duration Values in the MATLAB Workspace” on page 4-23

Plotting Signals that Exhibit Zero-Duration Values

One way to visualize event-based signals, including signal values that do not persist for a positive duration, is to use the Signal Scope or X-Y Signal Scope block. Either of these blocks can produce a plot that includes a marker for each signal value (or each signal-based event, in the case of the event counting scope). For example, the figure below uses a plot to illustrate the situation described in “Scenario: Server Departure and New Arrival” on page 4-20.



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.

Plotting the Number of Signal Changes Per Time Instant

To detect the presence of zero-duration values, but not the values themselves, use the Instantaneous Event Counting Scope block with the **Type of change in signal value** parameter set to **Either**. When the input signal assumes multiple values at an instant of time, the plot shows a stem of height of two or greater.

For an example using this block, see “Example: Plotting Event Counts to Check for Simultaneity” on page 10-15.

Viewing Zero-Duration Values in the MATLAB Workspace

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

See “Sending Data to the MATLAB Workspace” on page 4-17 for instructions and an example.

Working with Bus Signals

When you select the SimEvents configuration parameter **Prevent duplicate events on multiport blocks and branched signals**, you can connect Simulink bus signals to Timed to Event Signal blocks in a discrete-event system. Also, when you want to route a bus signal out of the discrete-event system, you can connect it to an Event to Timed Signal gateway block.

To manipulate bus signals in the discrete-event system, from the Simulink block library, you can use the following blocks:

- Bus Assignment
- Bus Creator
- Bus Selector

With **Prevent duplicate events on multiport blocks and branched signals** selected, the software enforces the behavior that the entire discrete-event system—including any time-based signals feeding into the discrete-event system via gateway blocks—executes at the base rate of the Simulink model. Therefore, a bus signal undergoes an implicit rate transition when connecting to a Timed to Event Signal gateway block. For more information on why the software enforces this rate transition behavior for both non-bus and bus signals, see “Invalid Connections of Timed to Event Signal Block” on page 13-91.

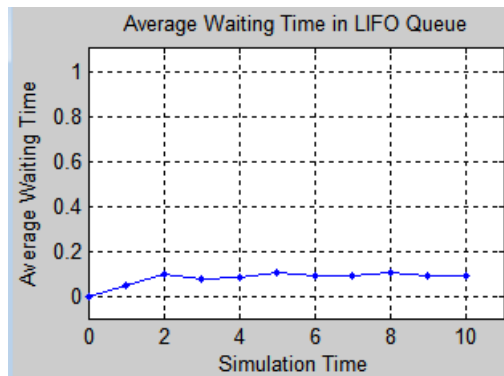
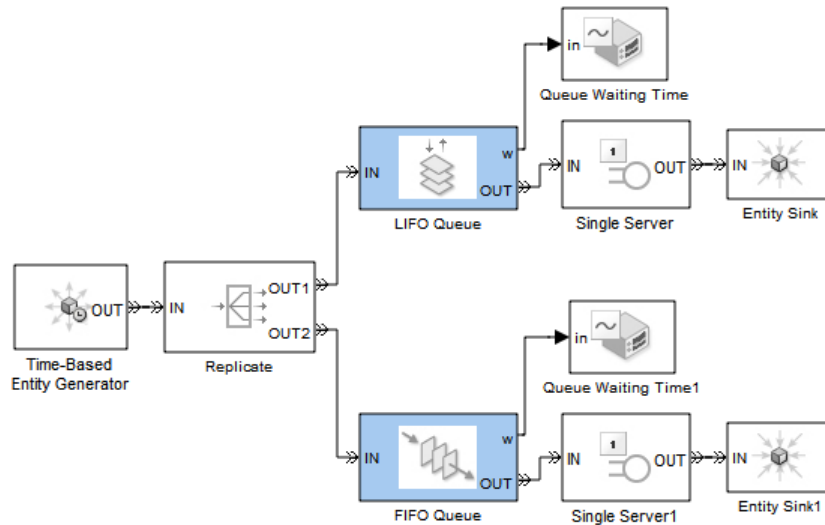
Modeling Queues and Servers

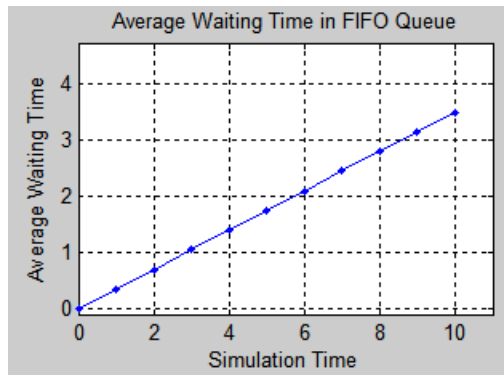
The topics below supplement the discussion in “Basic Queues and Servers” in the SimEvents getting started documentation.

- “Example: LIFO Queue Waiting Time” on page 5-2
- “Sorting by Priority” on page 5-4
- “Preempting an Entity in a Server” on page 5-10
- “Determining Whether a Queue Is Nonempty” on page 5-17
- “Modeling Multiple Servers” on page 5-18
- “Modeling the Failure of a Server” on page 5-20

Example: LIFO Queue Waiting Time

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





Sorting by Priority

In this section...
“Behavior of the Priority Queue Block” on page 5-4
“Example: FIFO and LIFO as Special Cases of a Priority Queue” on page 5-4
“Example: Serving Preferred Customers First” on page 5-7

Behavior of the Priority Queue Block

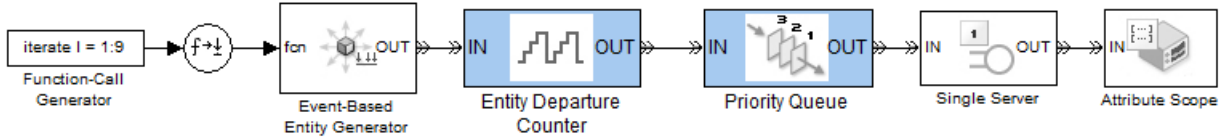
The Priority Queue block supports queuing in which entities' positions in the queue are based primarily on their attribute values. Arrival times are relevant only when attribute values are equal. You specify the attribute and the sorting direction using the **Sorting attribute name** and **Sorting direction** parameters in the block's dialog box. To assign values of the attribute for each entity, you can use the Set Attribute block as described in “Setting Attributes of Entities” on page 1-6.

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

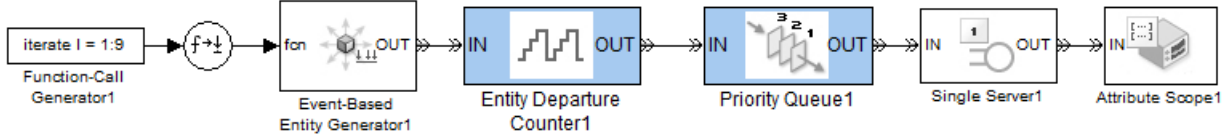
Example: FIFO and LIFO as Special Cases of a Priority Queue

Two familiar cases are shown in the example below, in which a priority queue acts like a FIFO or LIFO queue. At the start of the simulation, the FIFO and LIFO sections of the model each generate nine entities. The first entity advances 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.

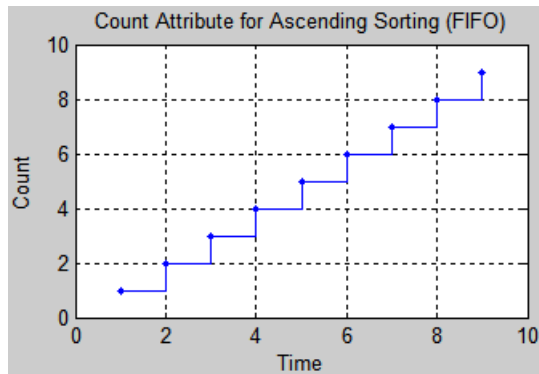
Priority Queue acts like FIFO Queue

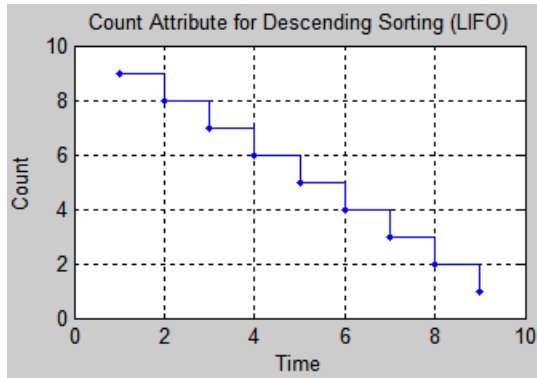


Priority Queue acts like LIFO Queue



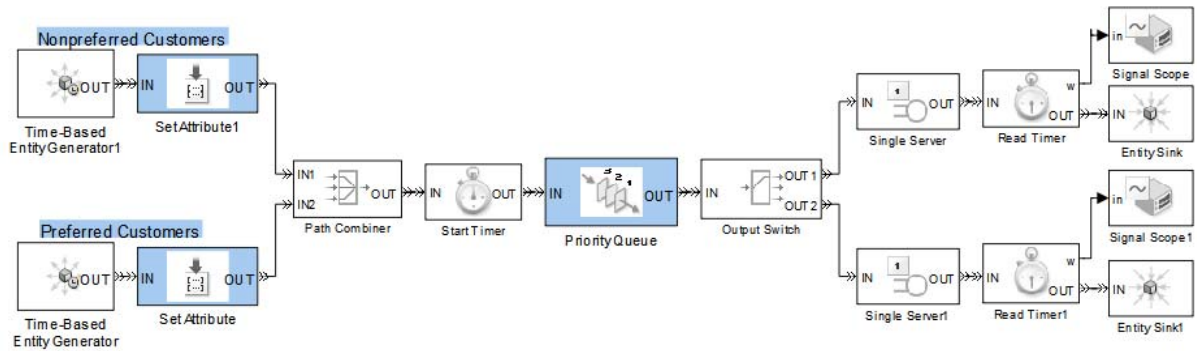
The FIFO plot reflects an increasing sequence of Count values. The LIFO plot reflects a descending sequence of Count values.



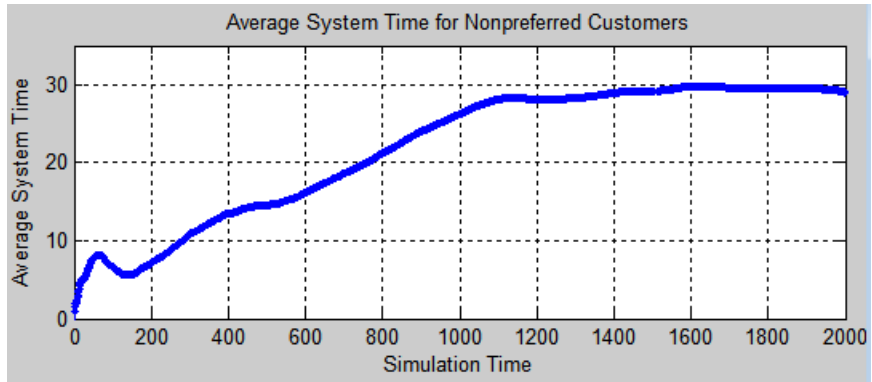


Example: Serving Preferred Customers First

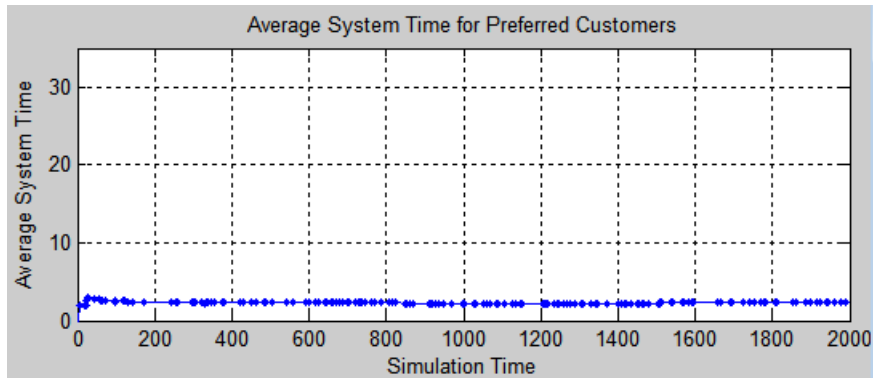
In the example below, two types of customers enter a queuing system. One type, considered to be preferred customers, are less common but require longer service. The priority queue places preferred customers ahead of nonpreferred customers. The model plots the average system time for the set of preferred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



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



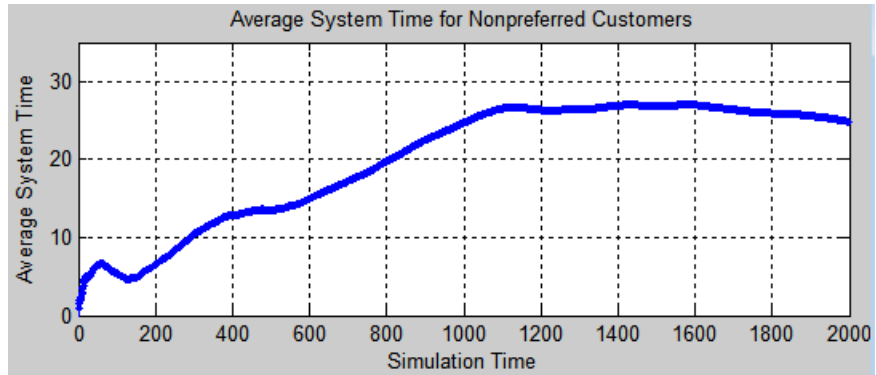
Average System Time for Nonpreferred Customers Sorted by Priority



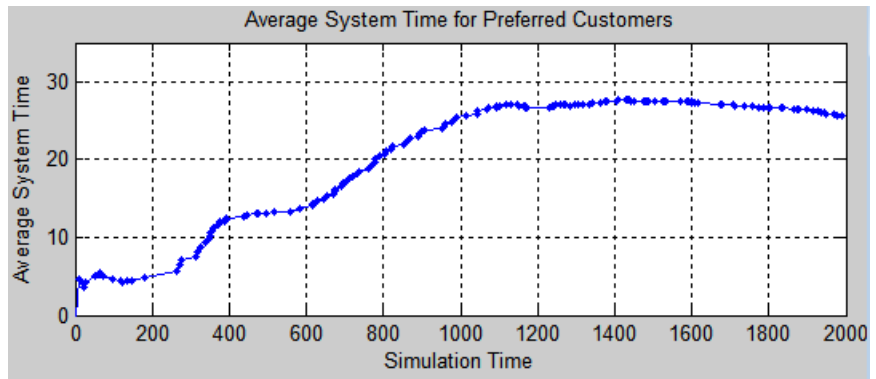
Average System Time for Preferred Customers Sorted by Priority

Comparison with Unsorted Behavior

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



Average System Time for Nonpreferred Customers Unsorted



Average System Time for Preferred Customers Unsorted

Preempting an Entity in a Server

In this section...
“Definition of Preemption” on page 5-10
“Criteria for Preemption” on page 5-10
“Residual Service Time” on page 5-11
“Queuing Disciplines for Preemptive Servers” on page 5-11
“Example: Preemption by High-Priority Entities” on page 5-11

Definition of Preemption

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

Criteria for Preemption

Whether preemption occurs depends on attribute values of the entity in the server and of the entity attempting to arrive at the server. You specify the attribute using the **Sorting attribute name** parameter in the Single Server block’s dialog box. You use the **Sorting direction** parameter to indicate whether the preempting entity has a smaller (Ascending) or larger (Descending) value of the attribute, compared to the entity being replaced. (Both parameters are available after you select **Permit preemption based on attribute**.) To assign values of the sorting attribute for each entity, you can use the Set Attribute block, as described in “Setting Attributes of Entities” on page 1-6. Valid values for the sorting attribute are any real numbers, Inf, and -Inf.

If the attribute values are equal, no preemption occurs.

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

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

Residual Service Time

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

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

Queuing Disciplines for Preemptive Servers

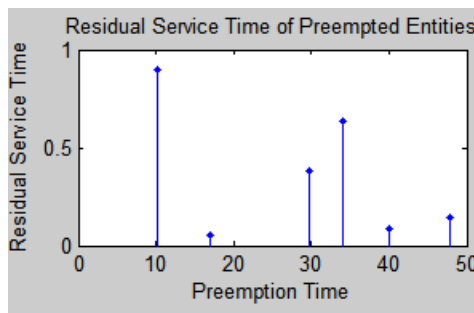
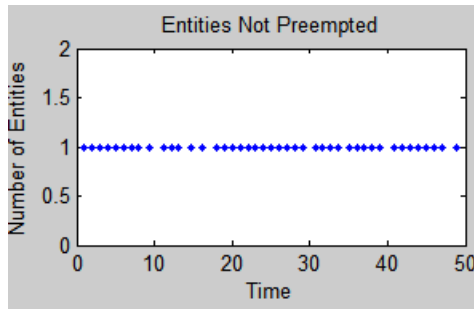
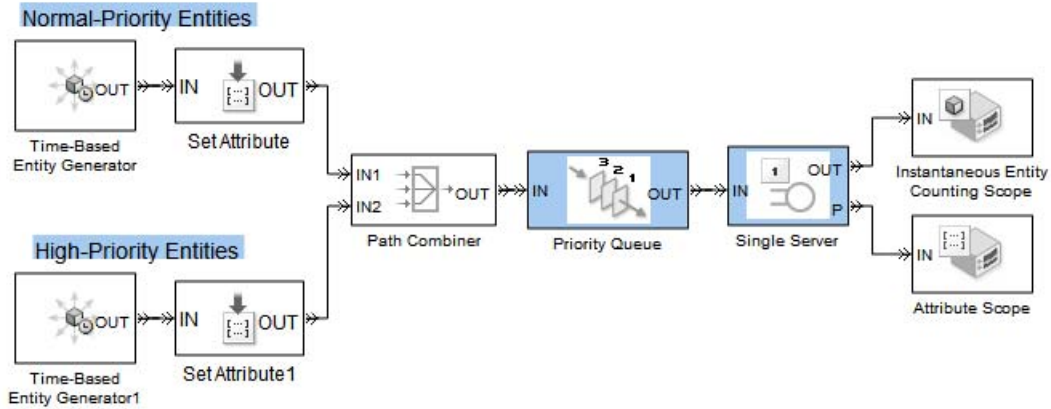
When you permit preemption in a Single Server block preceded by a queue, only the entity at the head of the queue can preempt an entity in the server.

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

Example: Preemption by High-Priority Entities

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



Appearance of Preemption-Related Operations in Debugger

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

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

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

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

```
tbreak 17
cont
```

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

```
Hit b1 : Breakpoint for first operation at or after time 17

%=====
Executing EntityGeneration Event (ev43)           Time = 17.043502632805254
: Entity = <none>                                Priority = 300
: Block = Time-Based Entity Generator1
```

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

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

```
step over
step over
step
```

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

```
%.....%
Generating Entity (en19)
```

```

: Block = Time-Based Entity Generator1
%.....%
Entity Advancing (en19)
: From = Time-Based Entity Generator1
: To   = Set Attribute1
%.....%
Setting Attribute on Entity (en19)
: EntityPriority = -Inf
: Block = Set Attribute1
%.....%
Entity Advancing (en19)
: From = Set Attribute1
: To   = Path Combiner
%.....%
Entity Advancing (en19)
: From = Path Combiner
: To   = Priority Queue
%.....%
Queuing Entity (en19)
: Priority Pos = 1 of 3
: Capacity = 25
: Block    = Priority Queue
%.....%
Scheduling NewHeadOfQueue Event (ev68)
: EventTime = 17.043502632805254 (Now)
: Priority   = SYS2
: Entity    = <none>
: Block    = Priority Queue
%.....%
Scheduling EntityGeneration Event (ev69)
: EventTime = 29.662073718098188
: Priority   = 300
: Entity    = <none>
: Block    = Time-Based Entity Generator1
%=====
Executing NewHeadOfQueue Event (ev68)                Time = 17.043502632805254
: Entity = <none>                                     Priority = SYS2
: Block  = Priority Queue
%.....%
Scheduling NewHeadOfQueue Event (ev70)

```

```

: EventTime = 17.043502632805254 (Now)
: Priority   = SYS2
: Entity    = <none>
: Block     = Priority Queue
%.....%
Entity Advancing (en19)
: From     = Priority Queue
: To       = Single Server
%.....%
Preempting Entity (en16)
: NewEntity = en19 (EntityPriority = -Inf)
: OldEntity = en16 (EntityPriority = 0)
: Block     = Single Server
%.....%
Canceling ServiceCompletion Event (ev66)
: EventTime = 17.094640781246184
: Priority   = 500
: Entity    = en16
: Block     = Single Server

```

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

```

%.....%
Canceling ServiceCompletion Event (ev66)
: EventTime = 17.094640781246184
: Priority   = 500
: Entity    = en16
: Block     = Single Server
%.....%
Setting Attribute on Entity (en16)
: ResidualServiceTime = 0.0511381484409306
: Block     = Single Server
%.....%
Entity Advancing (en16)
: From     = Single Server
: To       = Attribute Scope
%.....%

```

```
Executing Scope
: Block = Attribute Scope
%.....%
Destroying Entity (en16)
: Block = Attribute Scope
```

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

```
sedb.quit
```

Determining Whether a Queue Is Nonempty

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

- 1** Enable the **#n** output signal from the queue block. In the block dialog box, on the **Statistics** tab, select the **Number of entities in queue, #n** check box.
- 2** From the Math Operations library in the Simulink library set, insert a Sign block into the model. Connect the **#n** output port of the queue block to the input port of the Sign block.

The Sign block output has values of 0 and 1. A value of 1 indicates that the queue is storing one or more entities. A value of 0 indicates that the queue is not storing any entities.

Modeling Multiple Servers

In this section...

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

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

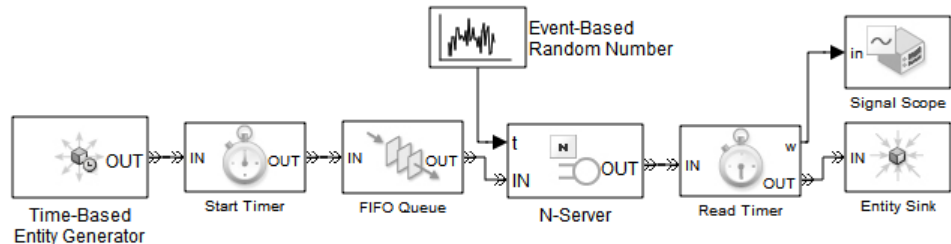
Blocks that Model Multiple Servers

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

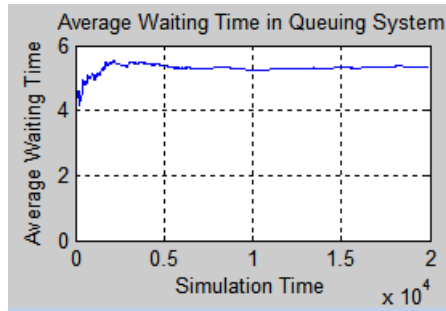
To model multiple servers that are not identical to each other, you must use multiple blocks. For example, to model a pair of servers whose service times do not share the same distribution, use a pair of Single Server blocks rather than a single N-Server block. The example in “Example: Selecting the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

Example: M/M/5 Queuing System

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



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



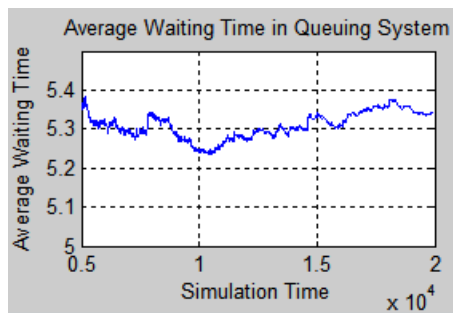
You can compare the empirical values shown in the plot with the theoretical value, $E[S]$, of the mean system time for an M/M/m queuing system with an arrival rate of $\lambda=1/2$ and a service rate of $\mu=1/5$. Using expressions in [2], the computation is as follows.

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

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

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

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



Modeling the Failure of a Server

In this section...

“Server States” on page 5-20

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

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

Server States

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

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

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

Using a Gate to Implement a Failure State

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

The gate prevents entity access to the server whenever the gate's control signal at the **en** input port is zero or negative. The logic that creates the **en** signal determines whether or not the server is in a failure state. You can implement such logic using the MATLAB Function block, using a subsystem containing logic blocks, or using Stateflow charts to transition among a finite number of server states.

For an example in which an Enabled Gate block precedes a server, see “Example: Controlling Joint Availability of Two Servers” on page 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 Charts to Implement a Failure State

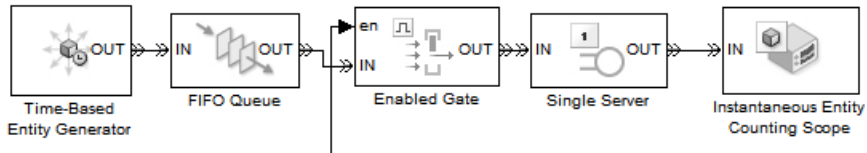
Stateflow software is suitable for implementing transitions among a finite number of server states. If you need to support more than just two states, then a Stateflow block might be more natural than a combination of Enabled Gate and logic blocks.

When modeling interactions between the state chart and discrete-event aspects of the model, note that a function call is one 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.

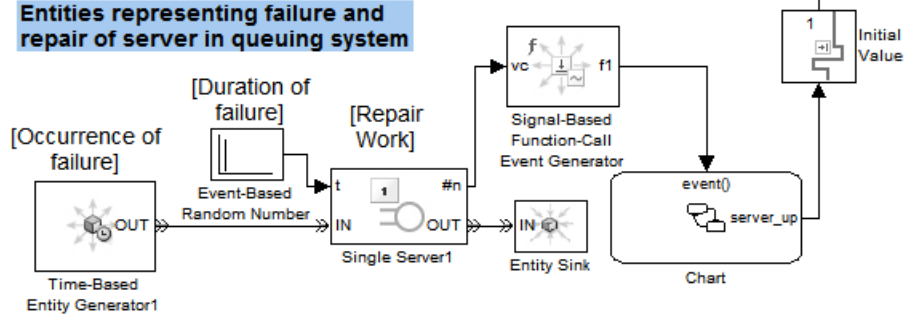
Example: Failure and Repair of a Server

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

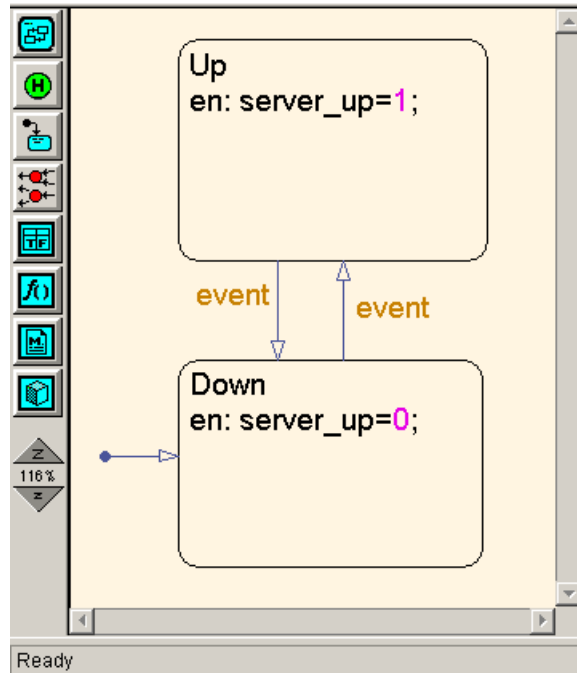


Entities representing failure and repair of server in queuing system

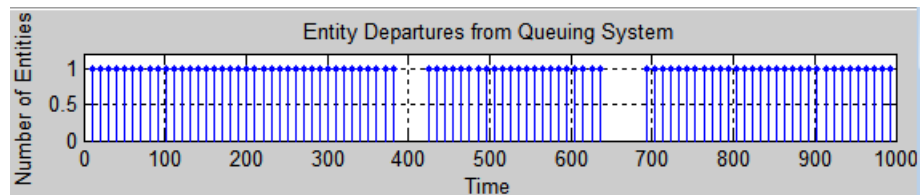


The lower portion of the model contains a parallel queuing system. The entities in the lower queuing system represent failures, not customers. Generation of a failure entity represents a failure occurrence in the upper queuing system. Service of a failure entity represents the time during which the server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

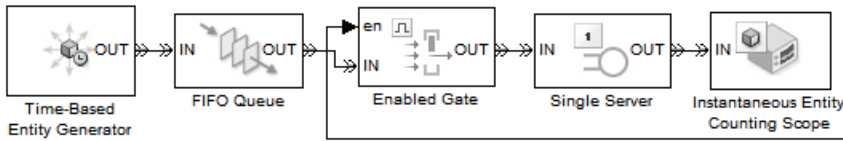


Although this two-state machine could be modeled more concisely with a Signal Latch block instead of a Stateflow block, the Stateflow chart scales more easily to include additional states or other complexity.

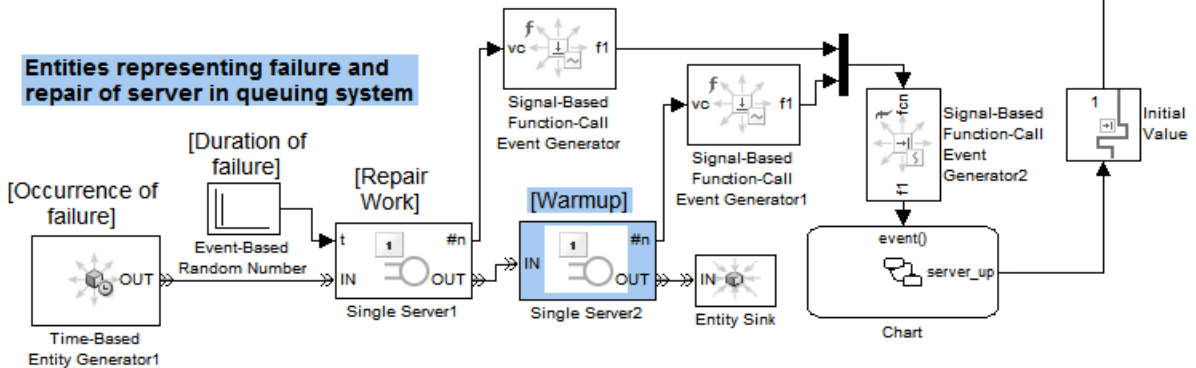
Example: Adding a Warmup Phase

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

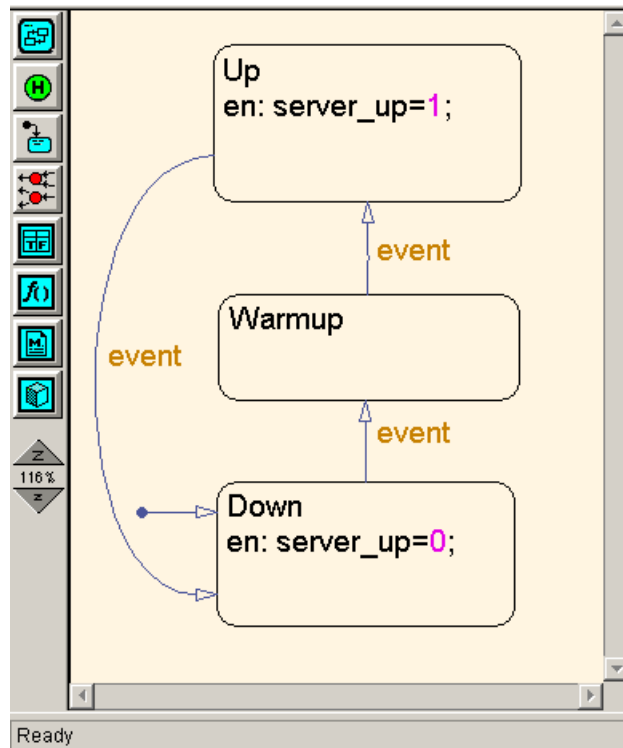
Entities representing customers in queuing system



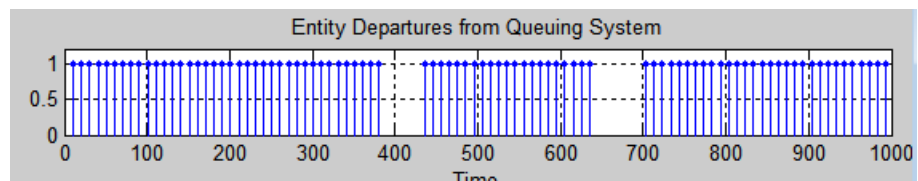
Entities representing failure and repair of server in queuing system



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



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



Routing Techniques

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

- “Output Switching Based on a Signal” on page 6-2
- “Example: Cascaded Switches with Skewed Distribution” on page 6-9
- “Example: Compound Switching Logic” on page 6-10
- “Example: Choosing the Shortest Queue” on page 6-13

Output Switching Based on a Signal

In this section...

“Routing Entities with an Output Switch” on page 6-2

“Specifying an Initial Port Selection” on page 6-4

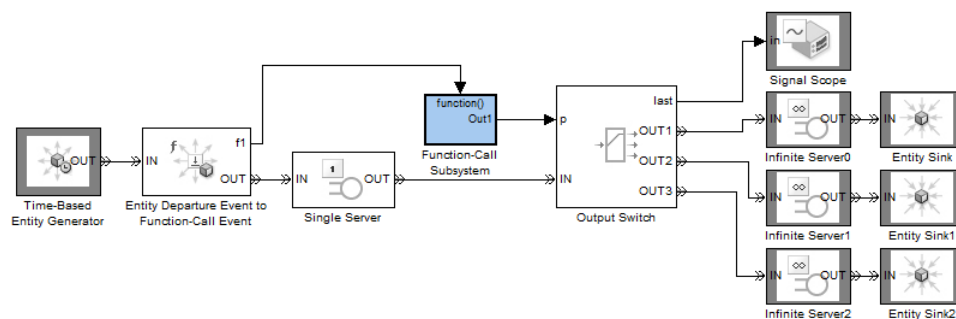
“Using the Storage Option to Prevent Latency Problems” on page 6-5

Routing Entities with an Output Switch

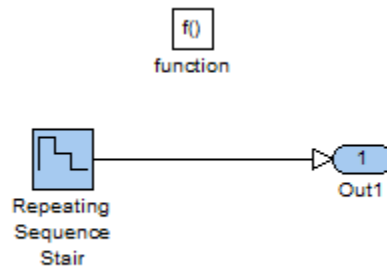
You can change the selected output port of an Output Switch block to route entities along different paths. The software selects the path on a per-entity basis, not on a predetermined time schedule.

The following model shows three paths available to an entity when it departs the Output Switch block.

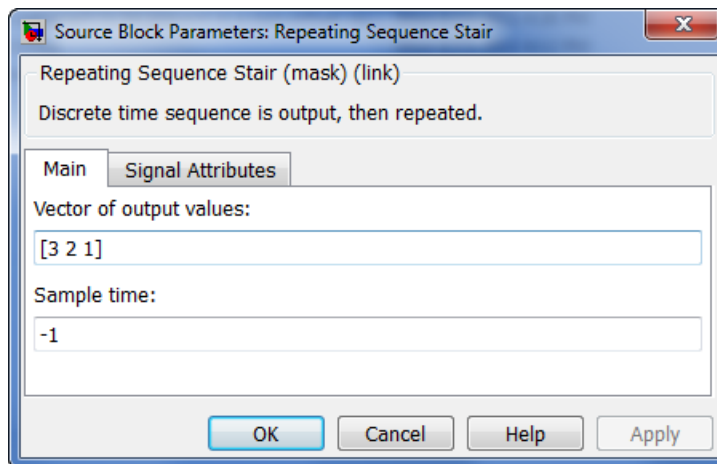
When the software generates an entity, the entity advances to the Single Server block. Meanwhile, the Entity Departure Event to Function-Call Event block issues a function call that executes the Function-Call Subsystem block.



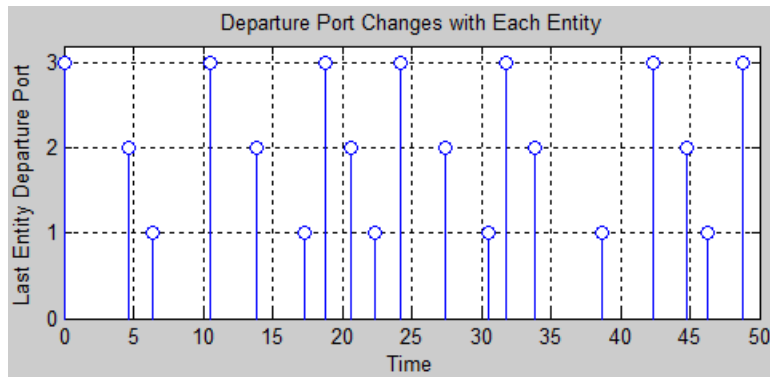
The Function-Call Subsystem block contains a single Repeating Sequence Stair block, whose **Sample time** parameter is set to -1 (inherited).



When the Function-Call Subsystem block executes, it outputs the next number from a repeating sequence. In this model, the output signal value is 3, 2 or 1, based on the sequence of values specified in the Repeating Sequence Stair block.



When service in the Single Server block is complete, the entity advances to the Output Switch block. The output signal of the Function-Call Subsystem block determines which output port the entity uses when it departs the Output Switch block. When the simulation is complete, this repeating port selection is shown by the output of the Signal Scope block.



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 6-13 for an example of this scenario.

Using the Storage Option to Prevent Latency Problems

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

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

Effect of Enabling Storage

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

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

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

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

- The selected port becomes unblocked.
- The selection changes to a port that is not blocked.
- The stored entity times out. For details on timeouts, see Chapter 8, “Forcing Departures Using Timeouts”.

Storage for a Time Interval. A stored entity can stay in the block for a nonzero period of time if the selected port is blocked. The design of your model should account for the effect of this phenomenon on statistics or other simulation behaviors. For an example scenario, see the discussion of average wait in “Example Without Storage” on page 6-7.

Even if the stored entity departs at the same time that it arrives, step 2 on page 6-5 is important for preventing latency.

Example Using Storage. The model in “Example: Choosing the Shortest Queue” on page 6-13 uses the **Store entity before switching** option in the Output Switch block. Suppose the queues have sufficient storage capacity so that the Output Switch block never stores an entity for a nonzero period of time. When an entity arrives at the Output Switch block, it does the following:

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

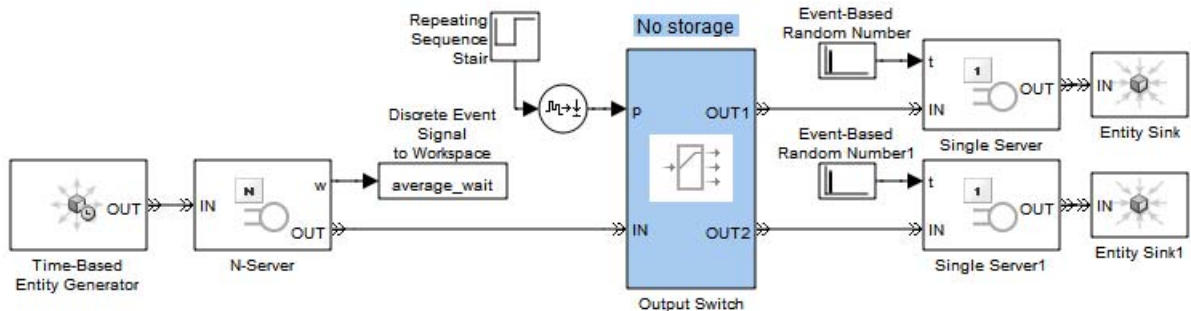
Effect of Disabling Storage

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

For common problems and troubleshooting tips, see “Unexpected Use of Old Value of Signal” on page 13-81 in the SimEvents user guide documentation.

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

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



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

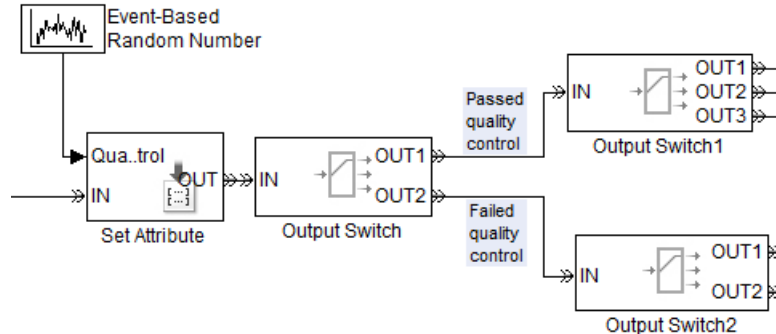
have not yet reached the Single Server blocks, then the model would need to account for entities stored in the switch for a nonzero period of time.

Example: Cascaded Switches with Skewed Distribution

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

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

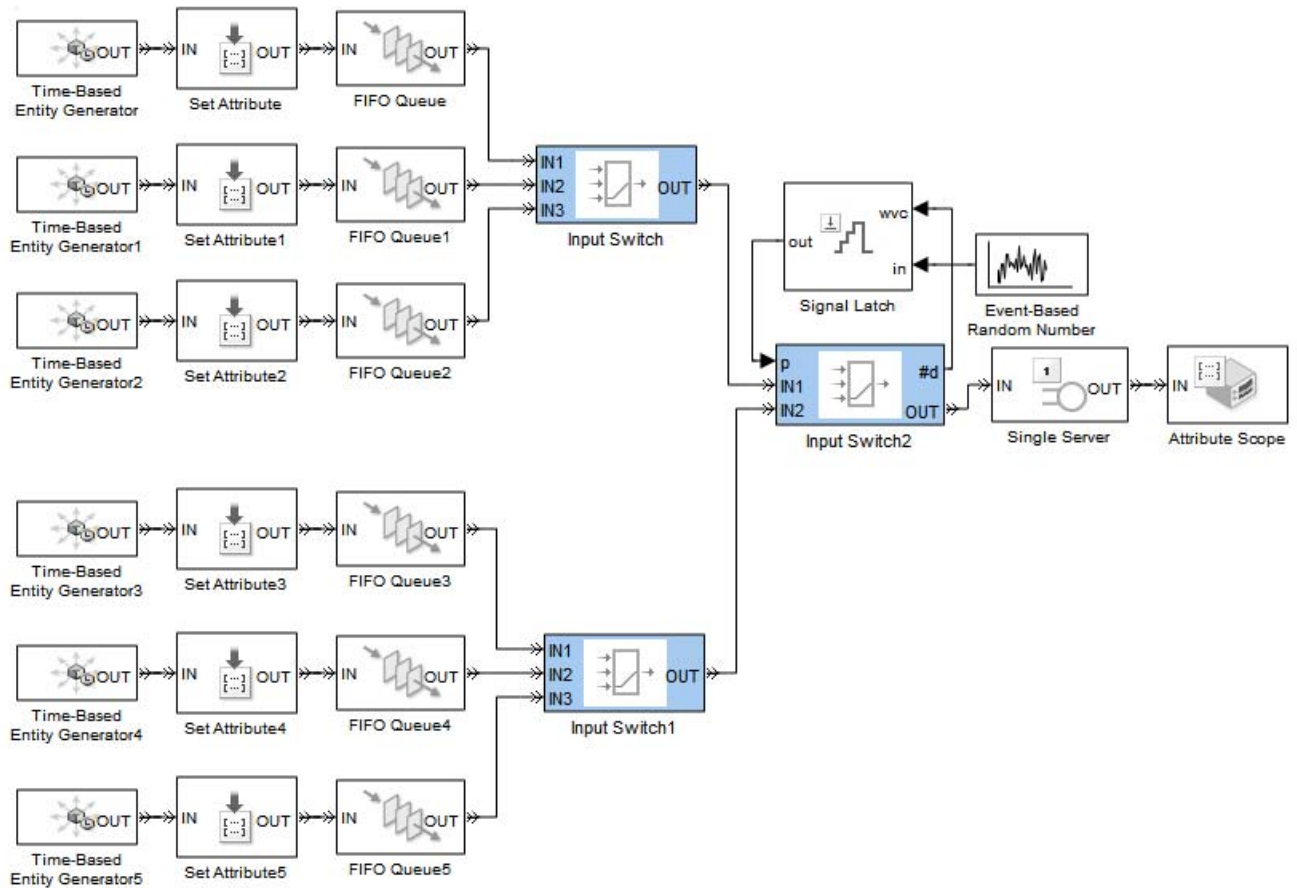
The figure below illustrates the switches and their switching criteria.



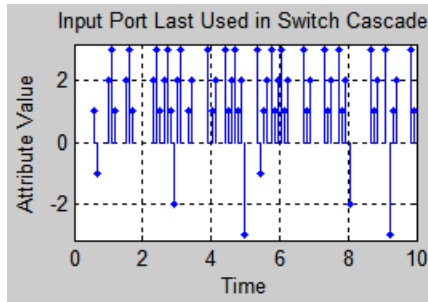
Example: Compound Switching Logic

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

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



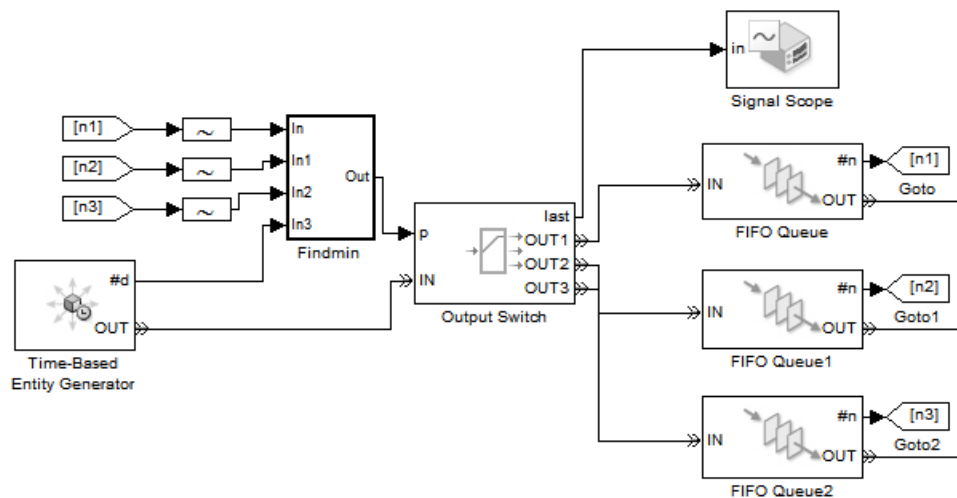
For tracking purposes, the model assigns an attribute to each entity based on its source. The attribute values are 1, 2, and 3 for entities in the first group and -1, -2, and -3 for entities in the second group. You can see from the plot below that negative values occur less frequently than positive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



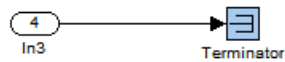
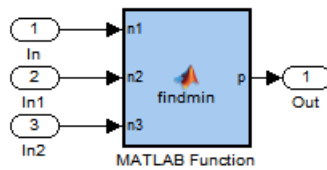
Example: Choosing the Shortest Queue

The model below directs entities to the shortest of three queues. It uses an Output Switch block to create the paths to the different queues. To implement the choice of the shortest queue, a discrete event subsystem queries each queue for its current length, determines which queue or queues achieve the minimum length, and provides that information to the Output Switch block. To ensure that the information is up to date when the Output Switch block attempts to output the arriving entity, the block uses the **Store entity before switching** option; for details, see “Using the Storage Option to Prevent Latency Problems” on page 6-5.

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 the MATLAB Function block.



Subsystem Contents

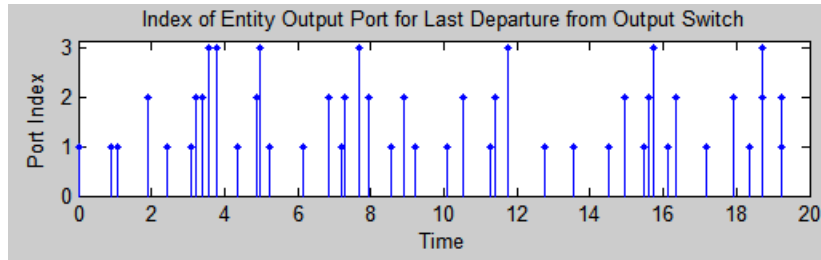
If you double-click the MATLAB Function block in a model, an editor window shows the MATLAB function that specifies the block. In this example, the following MATLAB function computes the index of a queue having the shortest length, where the individual queue lengths are n_1 , n_2 , and n_3 . If more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

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

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

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

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.



Regulating Arrivals Using Gates

- “Role of Gates in SimEvents Models” on page 7-2
- “Keeping a Gate Open Over a Time Interval” on page 7-4
- “Opening a Gate Instantaneously” on page 7-6
- “Adding Gating Logic Using Combinations of Gates” on page 7-9

Role of Gates in SimEvents Models

In this section...
“Overview of Gate Behavior” on page 7-2
“Types of Gate Blocks” on page 7-3

Overview of Gate Behavior

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

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

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

For example, you might use a gate

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

To learn about enabled gates, which can remain open for a time interval of nonzero length, see “Keeping a Gate Open Over a Time Interval” on page 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 “Adding Gating Logic Using Combinations of Gates” on page 7-9.

Types of Gate Blocks

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

- The Enabled Gate block, which uses a control signal to determine time intervals over which the gate is open or closed. For more information, see “Keeping a Gate Open Over a Time Interval” on page 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.

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

Keeping a Gate Open Over a Time Interval

In this section...
“Behavior of Enabled Gate Block” on page 7-4
“Example: Controlling Joint Availability of Two Servers” on page 7-4

Behavior of Enabled Gate Block

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

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

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

Depending on your application, the gating logic can arise from time-driven dynamics, state-driven dynamics, a SimEvents block’s statistical output signal, or a computation involving various types of signals. The **en** signal must be an event-based signal. To convert a time-based signal into an event-based signal, use the Timed to Event Signal block.

Example: Controlling Joint Availability of Two Servers

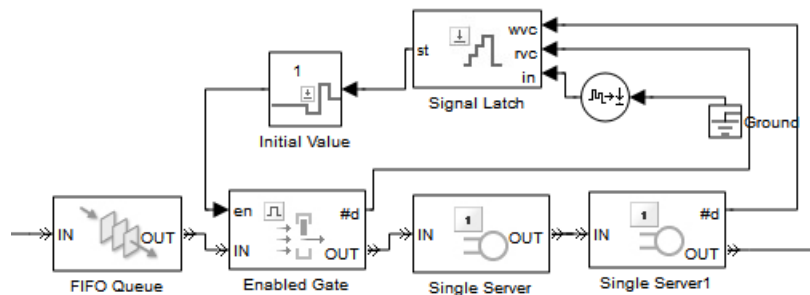
Suppose that each entity undergoes two processes, one at a time, and that the first process does not start if the second process is still in progress for the previous entity. Assume for this example that it is preferable to model the two processes using two Single Server blocks in series rather than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

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

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

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

This arrangement is shown below.



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

Opening a Gate Instantaneously

In this section...

“Behavior of Release Gate Block” on page 7-6

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

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

Behavior of Release Gate Block

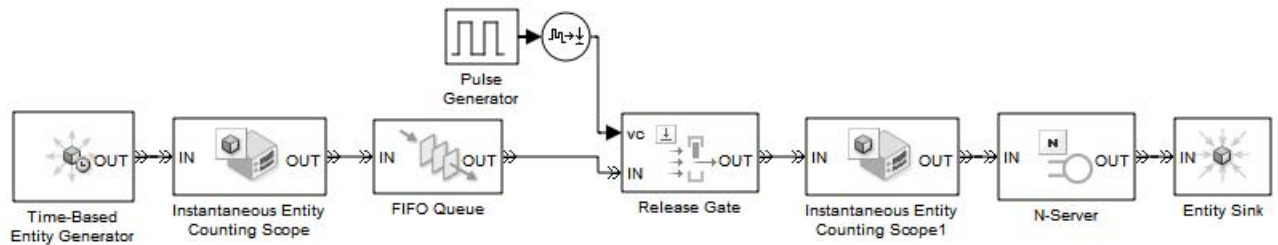
The Release Gate block opens instantaneously at a discrete set of times during the simulation and is closed at all other times. The gate opens when a signal-based event or a function call occurs. By definition, the gate’s opening permits one pending entity to arrive if able to advance immediately to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

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

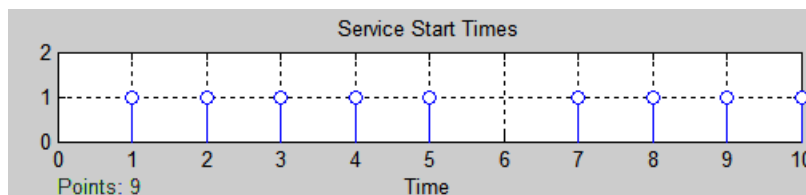
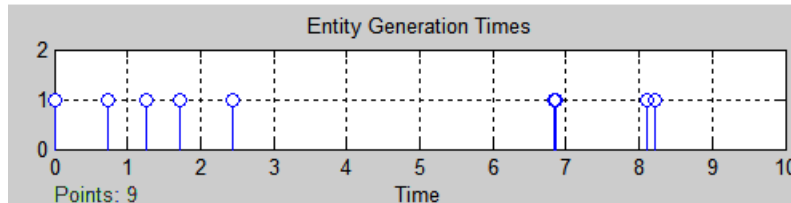
Example: Synchronizing Service Start Times with the Clock

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

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



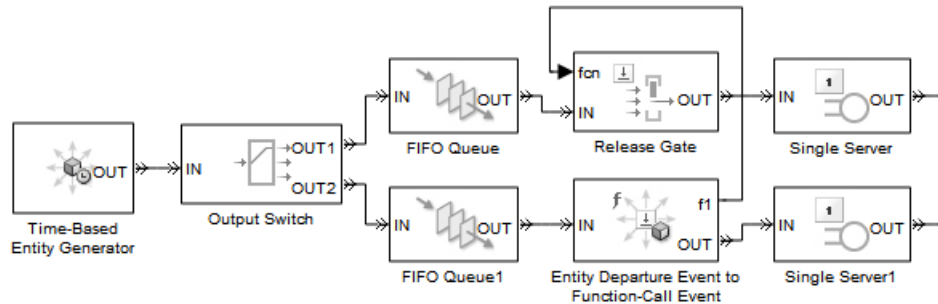
The plots below show that the entity generation times can be noninteger values, but the service beginning times are always integers.



Example: Opening a Gate Upon Entity Departures

In the model below, two queue-server pairs operate in parallel and an entity departs from the top queue only in response to a departure from the bottom queue. In particular, departures from the bottom queue block cause the Entity Departure Function-Call Generator 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.

Adding Gating Logic Using Combinations of Gates

In this section...

“Effect of Combining Gates” on page 7-9

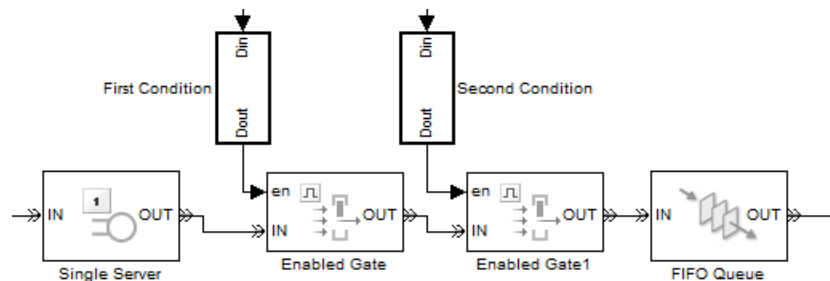
“Example: First Entity as a Special Case” on page 7-11

Effect of Combining Gates

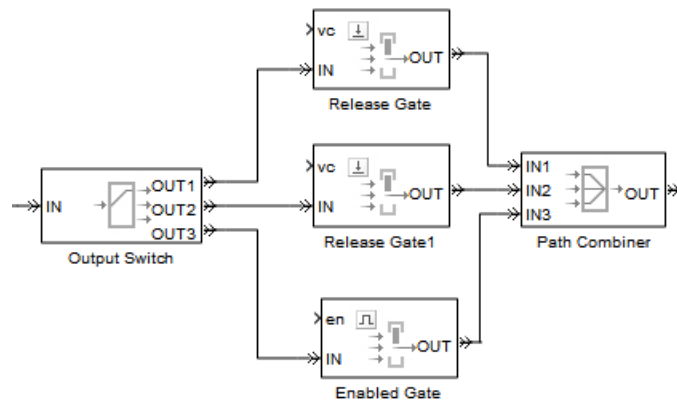
You can use multiple gate blocks in combination with each other:

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

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



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



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

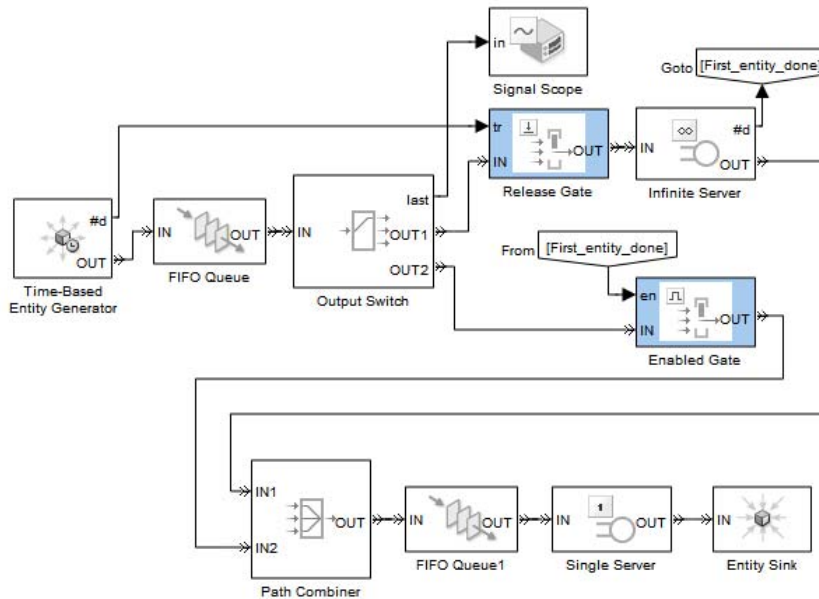
Note The figure above uses two Release Gate blocks and one Enabled Gate block, but you can use whatever combination is suitable for the logic of your application and whatever sequence you prefer. Also, the figure above omits the control signals (**vc** and **en**) for visual clarity but in your model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

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

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

Example: First Entity as a Special Case

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

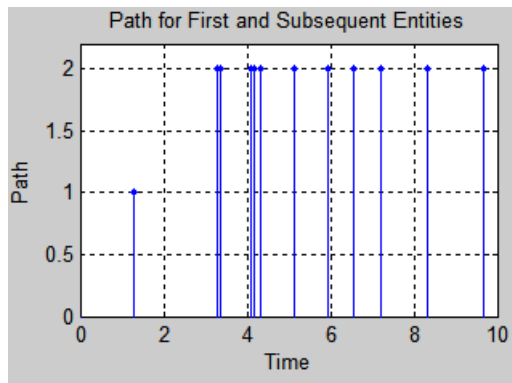


The Release Gate block is open precisely when the **#d** output signal from the Time-Based Entity Generator block rises from 0 to 1. That is, the gate is open for the first entity of the simulation and no other entities. The first entity arrives at an Infinite Server block, which represents the warmup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

Subsequent entities find the Release Gate block's entity input port unavailable, so they attempt to arrive at the Enabled Gate block. The Enabled Gate block is open during the entire simulation, except when the first entity has not yet departed from the Infinite Server block. This logic is necessary to prevent the second entity from jumping ahead of the first entity before the warmup period is over.

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

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



Forcing Departures Using Timeouts

- “Role of Timeouts in SimEvents Models” on page 8-2
- “Basic Example Using Timeouts” on page 8-3
- “Basic Procedure for Using Timeouts” on page 8-4
- “Defining Entity Paths on Which Timeouts Apply” on page 8-7
- “Handling Entities That Time Out” on page 8-10
- “Example: Limiting the Time Until Service Completion” on page 8-13

Role of Timeouts in SimEvents Models

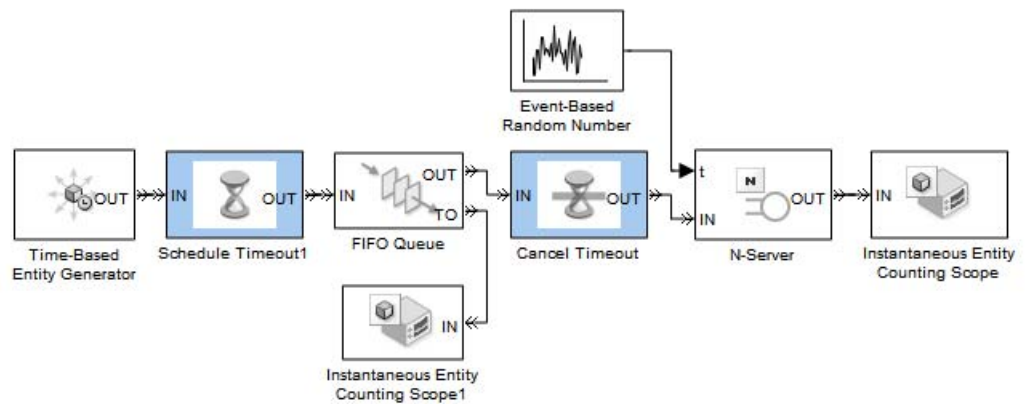
You can limit the amount of time an entity spends during the simulation on designated entity paths. Exceeding the limit causes a *timeout event* and the entity is said to have *timed out*. The duration of the time limit is called the *timeout interval*.

You might use timeout events to

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

Basic Example Using Timeouts

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



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

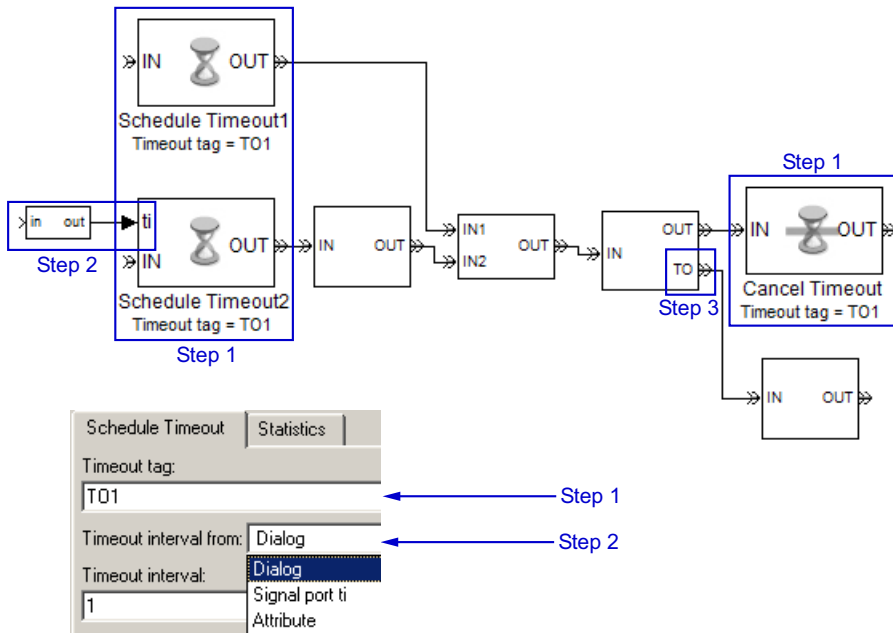
- **Entity Times Out** — If the customer is still in the queue when the clock reaches the time limit, the customer hangs up without reaching an operator. In generic terms, the entity times out, departs from the FIFO Queue block via the **TO** port, and does not reach the server.
- **Entity Advances to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

Basic Procedure for Using Timeouts

In this section...
“Schematic Illustrating Procedure” on page 8-4
“Step 1: Designate the Entity Path” on page 8-5
“Step 2: Specify the Timeout Interval” on page 8-5
“Step 3: Specify Destinations for Timed-Out Entities” on page 8-6

Schematic Illustrating Procedure

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



Step 1: Designate the Entity Path

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

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

For sample topologies, see “Defining Entity Paths on Which Timeouts Apply” on page 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 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 14-39.

- 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 13-81.

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

In this section...

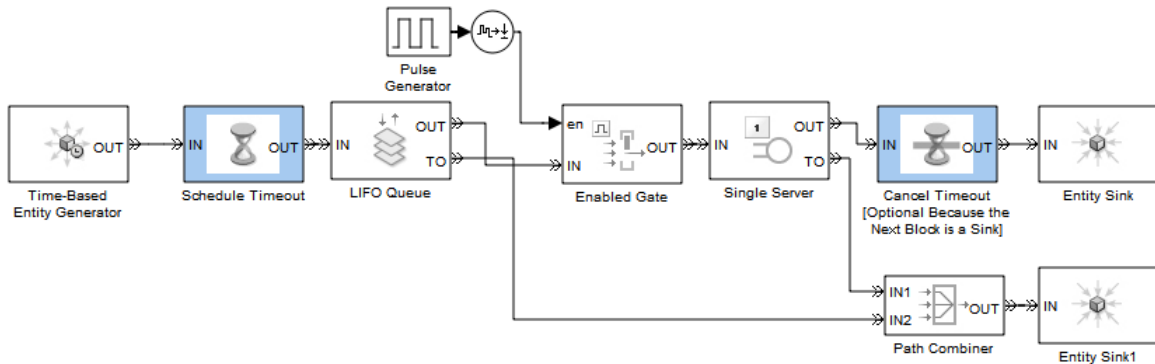
“Linear Path for Timeouts” on page 8-7

“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.

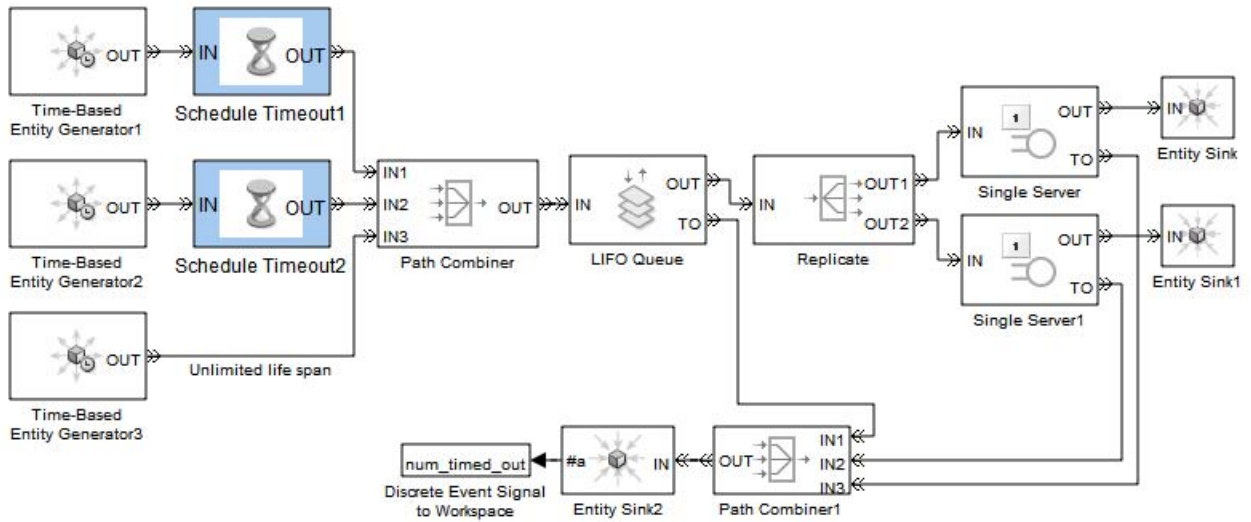
Other examples of timeouts on linear entity paths include these:

- “Basic Example Using Timeouts” on page 8-3
- “Example: Limiting the Time Until Service Completion” on page 8-13

Branched Path for Timeouts

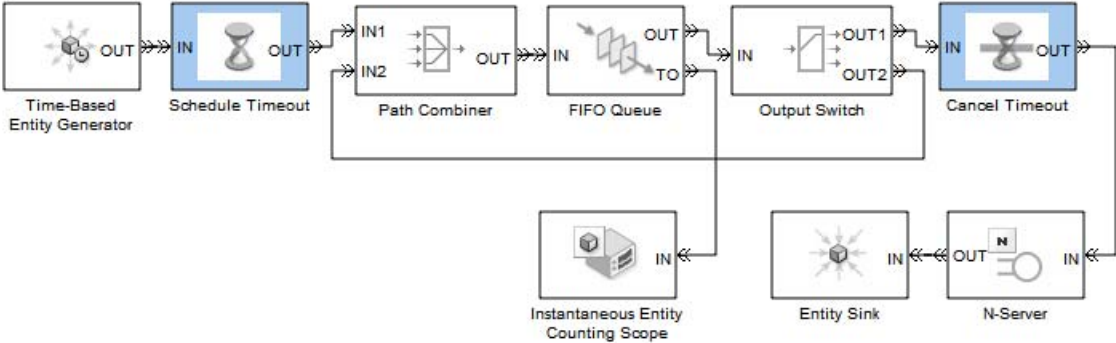
In the example below, entities from two sources have limited lifespans. Entities from a third source do not have limited lifespans.

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



Feedback Path for Timeouts

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



Handling Entities That Time Out

In this section...

“Common Requirements for Handling Timed-Out Entities” on page 8-10

“Techniques for Handling Timed-Out Entities” on page 8-10

“Example: Rerouting Timed-Out Entities to Expedite Handling” on page 8-11

Common Requirements for Handling Timed-Out Entities

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

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

Techniques for Handling Timed-Out Entities

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

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

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

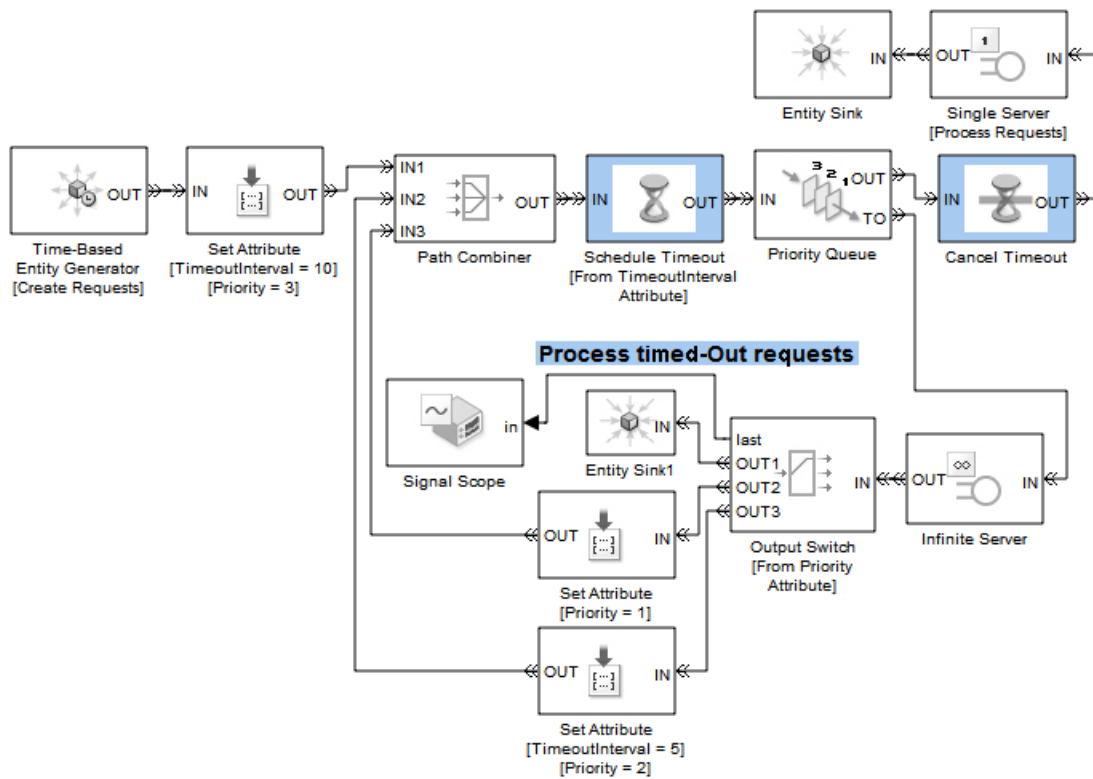
Note If an entity times out while it is in a block that has no **TO** port, then the Schedule Timeout block's **If entity has no destination when timeout occurs** parameter indicates whether the simulation halts with an error or discards the entity while issuing a warning.

Example: 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.

8 Forcing Departures Using Timeouts



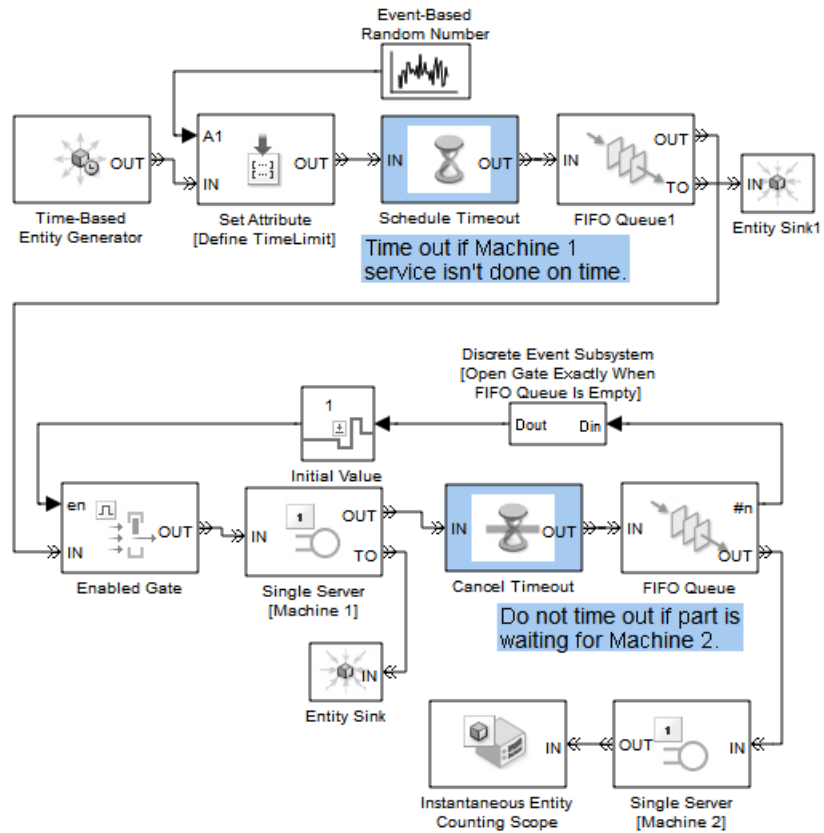
Example: Limiting the Time Until Service Completion

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.

8 Forcing Departures Using Timeouts



Computations on Event-Based Signals

- “Performing Computations in Atomic Subsystems” on page 9-2
- “Suppressing Computations By Filtering Out Events” on page 9-6
- “Performing Computations in Function-Call Subsystems” on page 9-11
- “Blocks Inside Subsystems with Event-Based Input Signals” on page 9-14
- “Performing Computations Without Using Subsystems” on page 9-15

Performing Computations in Atomic Subsystems

In this section...
“When to Use Atomic Subsystems for Computations on Event-Based Signals” on page 9-2
“How to Set Up Atomic Subsystems for Computations” on page 9-2
“Behavior of Computations in Atomic Subsystems” on page 9-3
“Refining the Behavior” on page 9-4
“Examples That Use Atomic Subsystems” on page 9-5

When to Use Atomic Subsystems for Computations on Event-Based Signals

In many situations, the clearest and most flexible way to perform a computation involving event-based signals is to place the computational blocks inside an Atomic Subsystem block.

How to Set Up Atomic Subsystems for Computations

The alternate ways to set up atomic subsystems are described in these topics:

- “Creating an Atomic Subsystem by Adding the Atomic Subsystem Block” on page 9-2
- “Creating an Atomic Subsystem from Existing Blocks” on page 9-3

Creating an Atomic Subsystem by Adding the Atomic Subsystem Block

If you have not started adding the computational blocks to the model, use the procedure in “Creating a Subsystem by Grouping Existing Blocks”, with one change: instead of using the Subsystem block, use the Atomic Subsystem block.

Creating an Atomic Subsystem from Existing Blocks

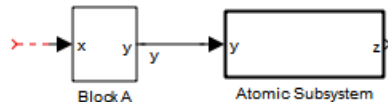
If your model already contains a connected set of computational blocks, use this procedure to convert the set of blocks into a subsystem like that of the Atomic Subsystem block.

- 1 Create a virtual subsystem containing the set of blocks. Follow the steps in “Creating a Subsystem by Grouping Existing Blocks”.
- 2 Open the subsystem parameters dialog box. With the new subsystem block selected, select **Edit > Subsystem Parameters**.
- 3 Select **Treat as atomic unit** and click **OK**. The block outline becomes bold to indicate that the subsystem is a nonvirtual subsystem.

Behavior of Computations in Atomic Subsystems

When an Atomic Subsystem block has event-based input signals, it behaves as follows:

- At time 0, the simulation executes all blocks within the Atomic Subsystem and updates values of the subsystem’s output ports.
- The subsystem executes in any of the circumstances listed in the following table:



In the Schematic, if Block A Is...	...This Occurs
Not the Event Filter block	Signal y has a sample time hit event.
The Event Filter block	Signal x has a qualifying signal-based event, according to the parameters of Block A.

- Whenever the subsystem executes, all the blocks in the subsystem execute. In this sense, the computation is an atomic operation. During this atomic

operation, each block executes once, using the current values of its input signals. The sequence in which the blocks in the subsystem execute depends on the sorted order that the application determines.

- At any given value of the simulation clock, conditions that cause the subsystem to execute can occur zero, one, or multiple times. Such flexibility and aperiodicity are characteristic of discrete-event simulations. Multiple executions can arise from multiple signal-based events in a single input signal or from signal-based events in multiple input signals. However, each signal-based event can execute the subsystem only once, regardless of whether the signal is scalar or nonscalar.

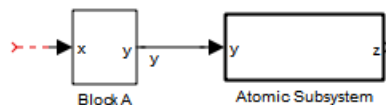
Refining the Behavior

Initial Value of the Subsystem Output

To replace the initial value for an output signal of the Atomic Subsystem block, connect the signal to the Initial Value block.

Types of Events That Cause Subsystem Execution

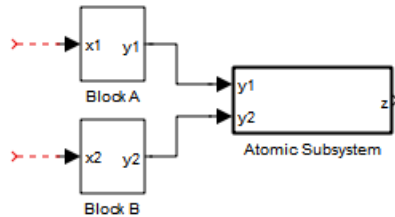
In the following schematic, if Block A is not the Event Filter block and signal y is an event-based signal, each sample time hit of the signal causes the subsystem to execute.



To make the subsystem respond to changes or triggers in this signal instead of responding to each sample time hit, connect the signal to the Event Filter block. For details, see “Suppressing Computations By Filtering Out Events” on page 9-6.

Subsystems Having Multiple Input Ports

If your Atomic Subsystem block has multiple input signals, each signal can cause the subsystem to execute. For example, in the following graphic, the subsystem can execute based on signal-based events in signal y1 and signal y2.



Examples That Use Atomic Subsystems

- “Example: Computing a Time Average of a Signal” on page 11-9
- “Example: Resetting an Average Periodically” on page 11-12
- “Example: Fraction of Dropped Messages” on page 11-8
- “Example: Observing Service Completions” on page 2-19

Suppressing Computations By Filtering Out Events

In this section...

“When to Suppress Computations” on page 9-6

“How to Set Up Event Filter Blocks” on page 9-7

“Behavior of Event Filter Blocks” on page 9-8

“Evaluating the Behavior” on page 9-9

“Examples That Use Event Filter Blocks” on page 9-10

When to Suppress Computations

This section describes situations in which you should consider selectively preventing execution of an Atomic Subsystem block that has event-based input signals.

Note If the computations that you want to suppress are in a Function-Call Subsystem block instead of an Atomic Subsystem block, see “Conditionalizing Events” on page 2-36.

States and Persistent Variables in Computation

When the computation uses persistent variables of a MATLAB function, Memory blocks, or other blocks with state, consider whether performing the computation at inappropriate points in the simulation causes it to store the wrong data for future use or corrupt the state of a block. If that issue affects your computation, try one of these solutions:

- Introduce additional logic in your model to prevent the subsystem from executing or prevent it from corrupting a variable or state.
- Use the techniques in “How to Set Up Event Filter Blocks” on page 9-7. The techniques are applicable when you can characterize the appropriate points at which to perform the computation, in terms of a type of signal-based event of an input to an Atomic Subsystem block. For example, suppose increases in a signal value are appropriate points at which to perform the computation, while other sample time hits are not.

For an example that uses the solution involving the Event Filter block, see the model in “Example: Observing Service Completions” on page 2-19. The model uses a MATLAB function that compares the current and previous values of the input arguments. At the end, the function stores the current values in persistent variables, to use (as previous values) in the next invocation of the function. If the example invoked the function upon irrelevant sample time hits in the second input argument, the function inappropriately overwrites the stored values and causes the next invocation of the function to produce incorrect results. To avoid this issue, the example uses the Event Filter block to avoid invoking the function when the second input argument has sample time hits that are not increases in value.

Logic Corresponding to Changes or Triggers

Execute an Atomic Subsystem block when a particular input signal has a particular type of signal-based event, and not when the signal has other types of events, as in the following cases:

- When your implementation of the subsystem implicitly assumes that the signal has had a particular type of event.
- When your model requires the result of the computation only when the signal has had a particular type of event.

For example, if you want the simulation to respond to worsening backlogs in a queue by recomputing a routing path, the relevant signal-based events of the queue length signal are the increases, not decreases or repeated values. When the queue length increases, perform the computation, thereby avoiding disrupting the routing when the queue length does not increase.

How to Set Up Event Filter Blocks

The Event Filter block enables you to adjust the behavior of an Atomic Subsystem block by restricting the type of signal-based events that cause the subsystem to execute. The restriction applies to the signal that connects to a particular input port of the Atomic Subsystem block. To apply restrictions to multiple input ports, use multiple Event Filter blocks.

To set up such a restriction:

- 1 Locate the Atomic Subsystem input port whose behavior you want to adjust. Also, locate the signal that provides data for this input port.
- 2 Insert the Event Filter block into your model. Connect the data signal to the Event Filter input port. Connect the Event Filter output port to the Atomic Subsystem input port.
- 3 Configure the Event Filter block by setting parameters in its dialog box.

If You Want the Subsystem to...	Set These Parameters
Execute when the data signal exhibits a qualifying signal-based event, and the signal is real-valued	<ul style="list-style-type: none"> • Set Execute atomic subsystem to one of these values: <ul style="list-style-type: none"> - Upon sample time hit - Upon trigger - Upon change in signal <p>In the case of triggers and changes, an additional parameter lets you specify the direction of the trigger or change: rising, falling, or either.</p>
Execute when the data signal exhibits a qualifying signal-based event, and the signal is complex-valued	<ul style="list-style-type: none"> • Set Execute atomic subsystem to Upon sample time hit.
Use the most recent value of the data signal, but the signal must not cause the subsystem to execute	Set Execute atomic subsystem to Never.

Behavior of Event Filter Blocks

When the input signal of an Event Filter block has a sample time hit, it does the following:

- 1 Updates its output signal with the value of the input signal. This value is available to the Atomic Subsystem block to which the Event Filter block connects.
- 2 Determines whether to execute the Atomic Subsystem block, based on the settings in the block dialog box of the Event Filter block. If the Event Filter block is not supposed to execute the Atomic Subsystem block, the Event Filter does nothing further, until the next sample time hit of the input signal. Otherwise, processing continues to the next step.
- 3 Determines when to execute the Atomic Subsystem block.
 - If you did not select the **Resolve simultaneous signal updates according to event priority** option, the Event Filter block executes the Atomic Subsystem block immediately.
 - If you select the **Resolve simultaneous signal updates according to event priority** option, the Event Filter block schedules an event on the event calendar. The event time is the current simulation time. The event priority is the value of the **Event priority** parameter in the Event Filter block. When the event calendar executes this event, the Atomic Subsystem block performs its computation.
 - If you select both the **Resolve simultaneous signal updates according to event priority** option, and the configuration parameter **Prevent duplicate events on multiport blocks and branched signals**, the software uses the **Event priority** parameter to help Simulink to sort blocks in the model. In this case, the software no longer schedules an event on the event calendar.

Evaluating the Behavior

Use the SimEvents debugger to examine any of these occurrences:

- A signal-based event causes an Atomic Subsystem block to execute immediately.
- An Event Filter block suppresses execution of the subsystem because a signal-based event is not a qualifying event.

You can also use the Instantaneous Event Counting Scope to view signal-based events of these signals:

- Input signal to an Event Filter block
- Input signal to an Atomic Subsystem block, when the signal is not the output of a Event Filter block

Examples That Use Event Filter Blocks

- “Example: Resetting an Average Periodically” on page 11-12
- “Example: Observing Service Completions” on page 2-19
- “Example: Effects of Specifying Event Priorities” on page 3-25

Performing Computations in Function-Call Subsystems

In this section...

“When to Use Function-Call Subsystems for Computations on Event-Based Signals” on page 9-11

“How to Set Up Function-Call Subsystems for Computations” on page 9-11

“Behavior of Computations in Function-Call Subsystems” on page 9-12

“Refining the Behavior” on page 9-13

“Examples That Use Function-Call Subsystems” on page 9-13

When to Use Function-Call Subsystems for Computations on Event-Based Signals

In some situations, the most appropriate way to implement a numerical computation involving event-based signals is to place the computational blocks inside the Function-Call Subsystem block. When using this block, set the **Sample time type** parameter of the Trigger block to **triggered**.

How to Set Up Function-Call Subsystems for Computations

- 1 From the Ports & Subsystems library, copy the Function-Call Subsystem block from the Simulink into your model.
- 2 Open the Function-Call Subsystem block by double-clicking it. Initially, the subsystem contains:
 - An Inport block connected to an Outport block. The Inport block represents an input signal that provides data but does not cause the subsystem to execute.
 - A block labeled “function”. This block represents a function-call signal that causes the subsystem to execute.
- 3 In the empty Function-Call Subsystem window, create the contents of the subsystem. Use Inport blocks to represent input from outside the subsystem that provides data but does not cause the subsystem to execute.

Use Output blocks to represent external output. To learn which blocks are suitable for use inside the subsystem, see “Blocks Inside Subsystems with Event-Based Input Signals” on page 9-14.

- 4 At the upper level of your model hierarchy, create or identify the function-call signal that causes the subsystem to execute. To create this signal, you might need to do any of the following:
 - Create a function-call signal that indicates the occurrence of signal-based events or entity departure events. Use the Time-Based Function-Call Generator, Signal-Based Function-Call Generator or Entity Departure Function-Call Generator block.
 - Create a union of function-call signals using the Mux block.
 - Convert a function-call signal that originates from a block in the Simulink library set into an event-based function-call signal, using the Timed to Event Function-Call block.
- 5 Connect data signals to the ports that correspond to the Inport blocks inside the subsystem. Connect the function-call signal to the port labeled `function()`.

Behavior of Computations in Function-Call Subsystems

When a Function-Call Subsystem block has event-based input signals, it behaves as follows:

- The initial value of each output signal of the subsystem comes from the **Initial output** parameter of the corresponding Output block inside the subsystem. Even if there is no function call to invoke the subsystem at $T = 0$, the Output block creates a sample time hit for the initial value.
- The subsystem executes whenever the function-call input signal has a function call. Sample time hits of the data input signals of the subsystem do not cause the subsystem to execute and do not cause sample time hits of the output signals.
- Whenever the subsystem executes, all the blocks in the subsystem execute once, using the current values of their input signals. The sequence in which the blocks in the subsystem execute depends on the sorted order that the application determines.

- At any given value of the simulation clock, conditions that cause the subsystem to execute can occur zero, one, or multiple times. Such flexibility and aperiodicity are characteristic of discrete-event simulations.

Refining the Behavior

Initial Value of the Subsystem Output

To change the initial value for an output signal of the subsystem, change the **Initial output** parameter of the corresponding Outport block.

Timing of Function Calls Versus Signal Updates

If your function-call signal executes the Function-Call Subsystem block before all the data signals are up to date, try prioritizing the function-call signal. By doing so, you defer the subsystem execution relative to other simultaneous events, such as signal updates. To prioritize a function-call signal, insert a Signal-Based Event to Function-Call Event block and configure it as in “Example: Generating a Function Call with an Event Priority” on page 2-35.

Suppressing Computations Selectively

To prevent certain function calls in the function-call signal from executing the Function-Call Subsystem block, use the techniques in “Conditionalizing Events” on page 2-36.

Examples That Use Function-Call Subsystems

- Modeling Load Within a Dynamic Voltage Scaling Application demo
- “Example: Choosing the Shortest Queue” on page 6-13
- “Example: Detecting Changes in the Last-Updated Signal” on page 14-46

Blocks Inside Subsystems with Event-Based Input Signals

When an atomic subsystem or function-call subsystem has event-based input signals, the subsystem has a restricted set of possible contents. The subsystem can contain:

- 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. 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 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.

The subsystem cannot contain:

- Continuous-time blocks
- Discrete-time blocks with a **Sample time** parameter value that is positive and finite.

In some cases, you can work around these restrictions by entering a **Sample time** parameter value of -1 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.

Performing Computations Without Using Subsystems

In this section...

“When to Perform Computations on Event-Based Signals Without Using Subsystems” on page 9-15

“How to Set Up Blocks for Computations” on page 9-15

“Behavior of Computations” on page 9-15

“Refining the Behavior” on page 9-16

“Examples That Perform Computations Without Using Subsystems” on page 9-16

When to Perform Computations on Event-Based Signals Without Using Subsystems

In some situations, you can perform computations on an event-based signal by connecting it directly to computational blocks, without putting the blocks in an Atomic Subsystem or Function-Call Subsystem block. Direct connections make the model easier to construct.

How to Set Up Blocks for Computations

Insert computational blocks into your model and connect them directly to event-based signals. If the block has a **Sample time** parameter in the block dialog box, you must set **Sample time** to -1 to indicate an inherited sample time.

Behavior of Computations

When a connected set of computational blocks has event-based input signals, the behavior depends on whether the blocks are in a nonvirtual subsystem, such as an Atomic Subsystem or Function-Call Subsystem block.

- When a nonvirtual subsystem executes, the application determines an execution sequence that accounts for data dependencies among the blocks. Each block in the subsystem executes once, in that predetermined sequence.

- Without the nonvirtual subsystem, event-based input signals execute blocks upon the sample time hits of those input signals.

For details about how blocks behave when they have event-based input signals and are not in a nonvirtual subsystem, see “Execution of Blocks Having Event-Based Input Signals” on page 14-6.

Refining the Behavior

Initial Value of Event-Based Signals

To change the initial value for any event-based signal that is not in a nonvirtual subsystem, connect the signal to the Initial Value block.

Migrating from Direct Connections to Atomic Subsystem

As you build your model, it might evolve from a situation in which direct connections of event-based signals to computational blocks is appropriate, to a situation in which it is better to perform the computation in an Atomic Subsystem block. Be alert to changes that your computation involves a block that has any of these characteristics:

- The block does not support event-based input signals.
- You do not want the block to respond immediately to each sample time hit of each event-based input signal.

To migrate your computation to an Atomic Subsystem block, see “Creating an Atomic Subsystem from Existing Blocks” on page 9-3.

Examples That Perform Computations Without Using Subsystems

- “Example: Detecting Collisions by Comparing Events” on page 2-22

Plotting Data

- “Choosing and Configuring Plotting Blocks” on page 10-2
- “Working with Scope Plots” on page 10-10
- “Using Plots for Troubleshooting” on page 10-12
- “Example: Plotting Entity Departures to Verify Timing” on page 10-13
- “Example: Plotting Event Counts to Check for Simultaneity” on page 10-15

Choosing and Configuring Plotting Blocks

In this section...

“Sources of Data for Plotting” on page 10-2

“Comparison of Blocks for Plotting Signals Against Time” on page 10-3

“Inserting and Connecting Scope Blocks” on page 10-5

“Connections Among Points in Plots” on page 10-6

“Varying Axis Limits Automatically” on page 10-7

“Caching Data in Scopes” on page 10-8

“Examples Using Scope Blocks” on page 10-8

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 vs. time	Scope	
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	

Comparison of Blocks for Plotting Signals Against Time

The following table compares the capabilities of two blocks for plotting an event-based signal against time.

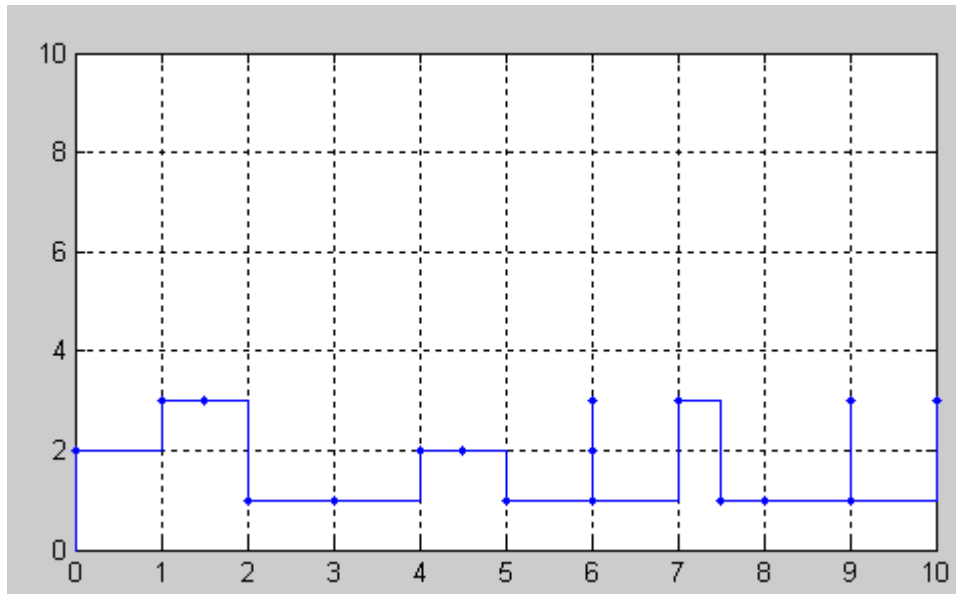
Capability	Signal Scope Block (SimEvents Sinks library)	Scope Block (Simulink Sinks library)
Includes markers to show sample time hits	Yes	No
Creates stair plots	Yes (default plot)	Yes
Creates stem plots	Yes (alternative to stair plot)	No
Creates continuous plots	Yes (alternative to stair plot)	No
Plots nonscalar signals	No	Yes
Plots multiple signals per window	No	Yes
Supports event-based signals	Yes	Yes

Capability	Signal Scope Block (SimEvents Sinks library)	Scope Block (Simulink Sinks library)
Supports time-based signals	No	Yes
Supports data types other than double	No	Yes
Available as a viewer	No	Yes

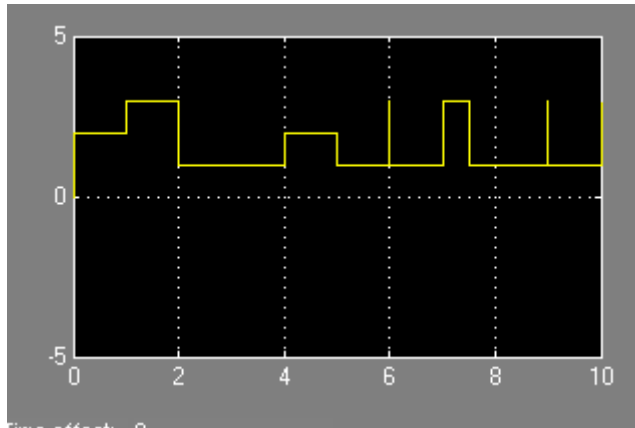
Markers in the Signal Scope plot are especially useful when your event-based signal:

- Assumes zero-duration values
- Assumes the same value in consecutive sample time hits at different times

The following plots of the same event-based signal illustrate the additional information that markers provide.



Signal Scope Plot with Markers



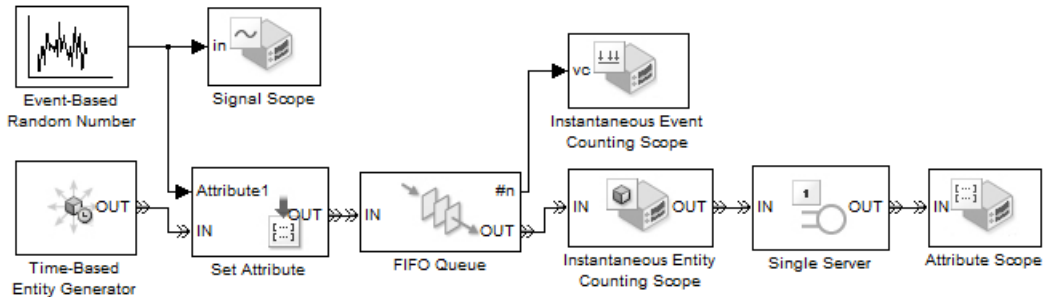
Scope Plot Without Markers

Inserting and Connecting Scope Blocks

The following table indicates the number, kind, and meaning of the input ports on each scope block.

Block	Input Ports	Port Description
Signal Scope	One signal input port	Signal representing the data to plot
Scope	One signal input port	Signal representing the data to plot
X-Y Signal Scope	Two signal input ports	Signals representing the data to plot
Attribute Scope	One entity input port	Entities containing the attribute value to plot
X-Y Attribute Scope	One entity input port	Entities containing the attribute values to plot
Instantaneous Entity Counting Scope	One entity input port	Entities whose arrivals the block counts
Instantaneous Event Counting Scope	One signal input port	Signal whose signal-based events or function calls the block counts

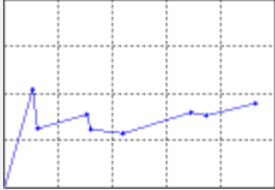
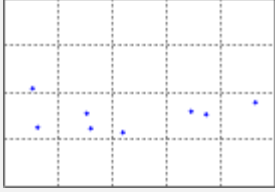
The following figure shows some typical arrangements of scope blocks in a model. Notice that the blocks that have entity input ports can have optional entity output ports, and that signal lines can branch whereas entity connection lines cannot.



Connections Among Points in Plots

You can configure certain scope blocks in the SimEvents Sinks library to determine whether and how the block connects the points that it plots. The following table indicates the options. To view or change the parameter settings, open the dialog box using the Parameters toolbar button in the plot window.

Connection Characteristics	Setting	Sample Plot
Stairstep across, then up or down. Also known as a zero-order hold.	Plot type = Stair in the block dialog box	
Vertical line from horizontal axis to point. No connection with previous or next plotted point. Also known as a stem plot.	Plot type = Stem in the block dialog box	

Connection Characteristics	Setting	Sample Plot
Line segment from point to point. Also known as a first-order hold.	Plot type = Continuous in the block dialog box	
No connection with other points or with axis. Also known as a scatter plot.	Style > Line > None in the plot window	

Note If no initial output, data value, or arriving entity indicates a value to plot at $T=0$, the plot shows no point at $T=0$. In this case, the plot does not connect the first plotted point to the $T=0$ edge of the plot.

Varying Axis Limits Automatically

Using parameters on the **Axes** tab of the dialog box of scope blocks in the SimEvents Sinks library, you set the initial limits for the axes of the plot. Also, these parameters let you choose how the block responds when a point does not fit within the current axis limits:

- **If X value is beyond limit**
- **If Y value is beyond limit**

Choices for the parameters are in the table.

Option	Description
Stretch axis limits	Maintain one limit while doubling the size of the displayed interval (without changing the size of the containing plot window)
Keep axis limits unchanged	Maintain both limits, which means that points outside the limits do not appear
Shift axis limits	Maintain the size of the displayed interval while changing both limits

Other operations can still affect axis limits, such as the autoscale, zoom, and pan features.

To store the current limits of both axes for the next simulation, select **Axes > Save axes limits** from the plot window menu.

Caching Data in Scopes

The **Data History** tab of the dialog box of scope blocks in the SimEvents Sinks library lets you balance data visibility with simulation efficiency. Parameters on the **Data History** tab determine how much data the blocks cache during the simulation. Caching data lets you view it later, even if the scope is not open during the simulation. Caching less or no data accelerates the simulation and uses less memory.

If you set the **Store data when scope is closed** parameter to **Limited**, uncached data points disappear when:

- The simulation ends
- You interact with the plot after pausing the simulation (using **Simulation > Pause**, for example)

Examples Using Scope Blocks

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

Example	Description
“Plotting the Queue-Length Signal” and “Observations from Plots” in the SimEvents getting started documentation	Stairstep and continuous plots of statistical signals
“Example: Round-Robin Approach to Choosing Inputs” in the SimEvents getting started documentation	Stem plot of data from an attribute
“Example: Preemption by High-Priority Entities” on page 5-11	Unconnected plot of a signal using dots
“Example: Setting Attributes” on page 1-8	Stairstep plots of data from attributes using Attribute Scope blocks as sinks
“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

Working with Scope Plots

In this section...

“Customizing Plots” on page 10-10

“Exporting Plots” on page 10-11

Customizing Plots

After a scope block in the SimEvents Sinks library opens its plot window, you can modify several aspects of the plot by using the menu and toolbar of the plot window:

- **Axes > Autoscale** resizes both axes to fit the range of the data plus some buffer space.
- The Zoom In and Zoom Out toolbar buttons change the axes as described in the MATLAB documentation about zooming in 2-D views.
- The Pan toolbar button moves your view of a plot.
- The **Style** menu lets you change the line type, marker type, and color of the plot. (You can also select **Style > Line > None** to create a plot of unconnected points.) Your changes become part of the block configuration and persist across sessions when you save the model.
- **Axes > Save axes limits** updates the following parameters on the **Axes** tab of the block dialog box to reflect the current limits of the axes:
 - **Initial X axis lower limit**
 - **Initial X axis upper limit**
 - **Initial Y axis lower limit**
 - **Initial Y axis upper limit**
- **Axes > Save position** updates the **Position** parameter on the **Figure** tab of the block dialog box to reflect the current position and size of the window.

Note Some menu options duplicate the behavior of a parameter in the block dialog box. In this case, selecting the menu option *replaces* the corresponding parameter value in the dialog. You can still edit the parameter values in the dialog manually. An example of a duplicate pair of menu option and dialog box parameter is **Show grid**.

Exporting Plots

The Save Figure toolbar button lets you export the current state of the plot to a file:

- Exporting to a FIG-file enables you to reload it in a different MATLAB software session. Reloading the file opens a new plot. The new plot is *not* associated with the original scope block. The new plot does not offer the same menu and toolbar options as in the original plot window.
- Exporting to a graphics file enables you to insert the graphic into a document.

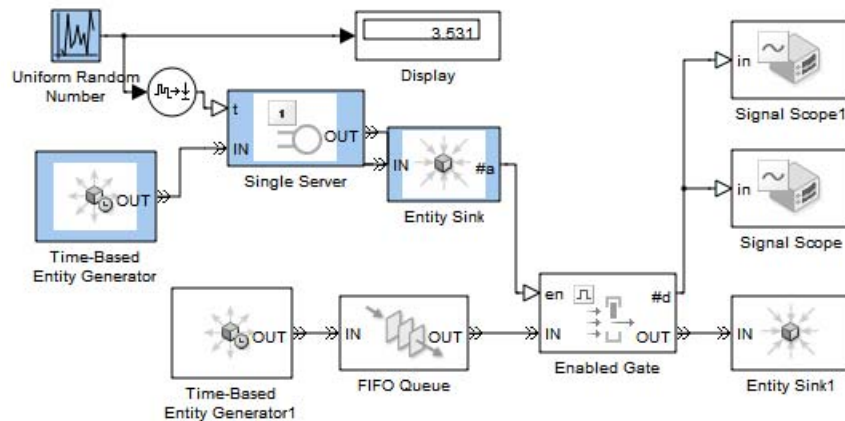
Using Plots for Troubleshooting

Here are typical ways to use plotting blocks in the SimEvents Sinks library to troubleshoot problems.

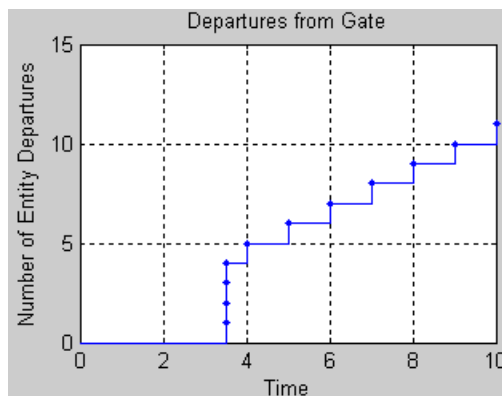
Technique	Example
Check when an entity departs from the block. To do this, plot the #d output signal of the block.	“Example: Plotting Entity Departures to Verify Timing” on page 10-13
Check whether operations such as service completion or routing are occurring as you expect. To do this, plot statistical output signals such as pe or last , if applicable.	
Check whether a block uses a control signal as you expect. To do this, plot input signals such as port selection, service time, or intergeneration time, and compare the values with observations of how the corresponding blocks use those signals.	“Example: Choices of Values for Event Priorities” on page 3-11
Check how long entities spend in a region of the model. To do this, plot the output of a Read Timer block.	“Example: M/M/5 Queuing System” on page 5-18
Check whether events you expect to be simultaneous are, in fact, simultaneous. To do this, use the Instantaneous Entity Counting Scope or Instantaneous Event Counting Scope block.	“Example: Counting Simultaneous Departures from a Server” on page 1-20 and “Example: Plotting Event Counts to Check for Simultaneity” on page 10-15

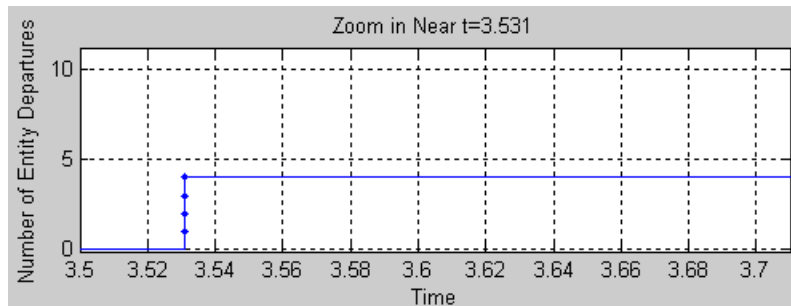
Example: Plotting Entity Departures to Verify Timing

This example shows how to verify the timing of a gate opening graphically. The model opens a gate at a random time and leaves the gate open for the rest of the simulation.



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.





Details about the model

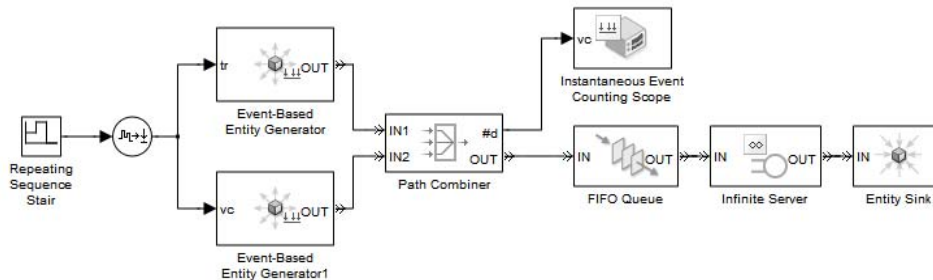
The modeling approach views the random opening of the gate as a discrete event, and models it via an entity departure from a server at a random time. The Time-Based Entity Generator block generates exactly one entity, at $T=0$. The Single Server block delays the entity for the amount of time indicated by the Uniform Random Number block, 3.531 s in this case. At $T=3.531$, the entity arrives at the Entity Sink block. This time is exactly when the **#a** signal of the sink block changes from 0 to 1, which in turn causes the gate to open.

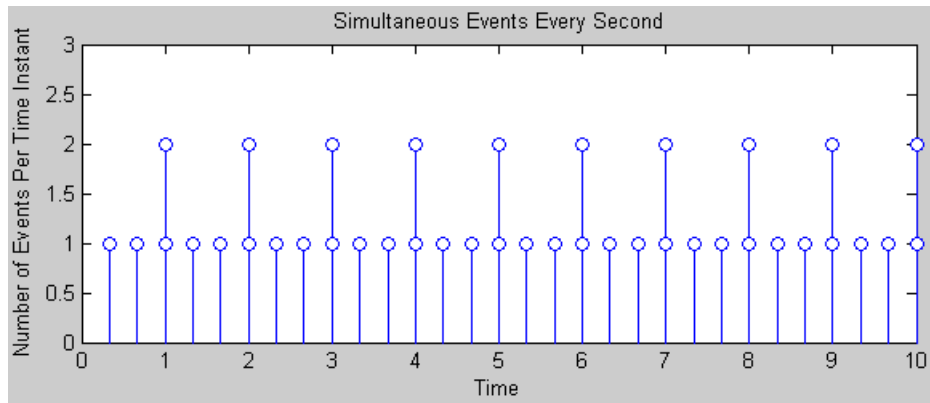
Example: Plotting Event Counts to Check for Simultaneity

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.

Using Statistics

- “Statistics for Data Analysis” on page 11-2
- “Statistics for Run-Time Control” on page 11-3
- “Statistical Tools for Discrete-Event Simulation” on page 11-4
- “Accessing Statistics from SimEvents Blocks” on page 11-5
- “Deriving Custom Statistics” on page 11-7
- “Measuring Point-to-Point Delays” on page 11-18
- “Varying Simulation Results by Managing Seeds” on page 11-24
- “Regulating the Simulation Length” on page 11-30

Statistics for Data Analysis

The purpose of creating a discrete-event simulation is often to improve understanding of the underlying system or guide 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, you might use the computed production and defect rates to help decide whether to change your maintenance schedule.
- If you simulate a communication bus under varying bus loads, you might use computed average delays in high- or low-priority messages to help determine whether a proposed architecture is viable.

When you design the statistical measures that you use to learn about the system, consider these questions:

- Which statistics are meaningful for your investigation or decision? For example, if you are trying to maximize efficiency, then what is an appropriate measure of efficiency in your system? As another example, does a mean give the best performance measure for your system, or is it also worthwhile to consider the proportion of samples in a given interval?
- How can you compute the desired statistics? For example, do you need to ignore any transient effects, does the choice of initial conditions matter, and what stopping criteria are appropriate for the simulation?
- To ensure sufficient confidence in the result, how many simulation runs do you need? One simulation run, no matter how long, is still a single sample and probably inadequate for valid statistical analysis.

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

Statistics for Run-Time Control

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

When you create simulations that use statistical signals to control the dynamics, you must have access to the current values of the statistics at key times throughout the simulation, not just at the end of the simulation. Some questions to consider while designing your model are:

- Which statistics are meaningful, and how should they influence the dynamics of the system?
- How can you compute the desired statistics at the right times during the simulation? It is important to understand when `SimEvents` blocks update each of their statistical outputs and when other blocks can access the updated values. For more information, see Chapter 4, “Working with Signals”.
- Do you need to account for initial conditions or extreme values in any special way? For example, if your control logic involves the number of entities in a queue, then be sure that the logic is sound even when the queue is empty or full.
- Will small perturbations result in large changes in the system’s behavior? When using statistics to control the model, you might want to monitor those statistics or other statistics to check whether the system is undesirably sensitive to perturbations.

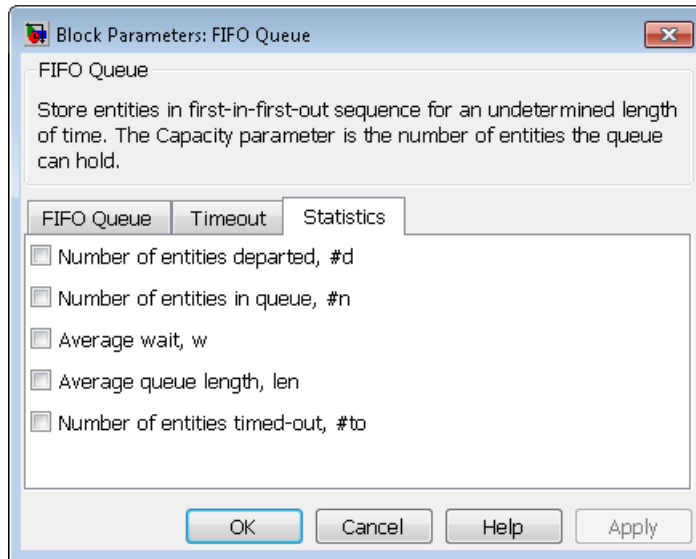
Statistical Tools for Discrete-Event Simulation

The table lists components that SimEvents models commonly use to gather or compute statistics.

Statistical Information	Available Tools	More Information
Number of entities in a queue or server	#n output signal from queue and server blocks	“Accessing Statistics from SimEvents Blocks” on page 11-5
Utilization of a server	util output signal from Single Server and N-Server blocks	“Accessing Statistics from SimEvents Blocks” on page 11-5
Number of entities that have departed from a block	<ul style="list-style-type: none"> • #d output signal from various SimEvents blocks • Entity Departure Counter • Instantaneous Entity Counting Scope 	<ul style="list-style-type: none"> • “Accessing Statistics from SimEvents Blocks” on page 11-5 • “Counting Entities” on page 1-19
Amount of time entities spend in a block or region (point-to-point delay)	<ul style="list-style-type: none"> • Start Timer • Read Timer 	“Measuring Point-to-Point Delays” on page 11-18
Events on a signal, such as changes in value	<ul style="list-style-type: none"> • Instantaneous Event Counting Scope • Signal-Based Function-Call Event Generator 	“Example: Plotting Event Counts to Check for Simultaneity” on page 10-15
Custom computation on event-based signals	<ul style="list-style-type: none"> • Atomic Subsystem • MATLAB Function • Event Filter • Attribute Function • Stateflow Chart* 	<ul style="list-style-type: none"> • “Deriving Custom Statistics” on page 11-7 • Chapter 9, “Computations on Event-Based Signals” • “Manipulating Attributes of Entities” on page 1-12 • Chapter 12, “Using Stateflow Charts in SimEvents Models”

Accessing Statistics from SimEvents Blocks

Most SimEvents blocks can produce one or more statistical output signals.



This procedure shows you how to access a statistical output signal for a given SimEvents block.

- 1** Determine which statistical output signal you want to access and find the associated parameter in the block dialog box. To see which statistics are available, open the block dialog box. In most cases, the list of available statistics appears as a list of parameters on the **Statistics** tab of the 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 parameters associated with statistics are straightforward to locate.
- 2** Select the check box. After you apply the change, the block has a new signal output port corresponding to that statistic.
- 3** Connect the new signal output port to the signal input port of another block. The table lists some common examples.

If You Want to...	Use this Block...
Create a plot using the statistic.	Signal Scope or X-Y Signal Scope
Show the statistic on the block icon throughout the simulation.	Display
Write the data set to the MATLAB workspace when the simulation stops or pauses. To learn more, see “Sending Data to the MATLAB Workspace” on page 4-17.	Discrete Event Signal to Workspace
Perform custom data processing. See “Deriving Custom Statistics” on page 11-7 for some specific examples.	Custom subsystem or computational block

For more information about when SimEvents blocks update their statistical signals and when other blocks react to the updated values, see Chapter 4, “Working with Signals”.

Deriving Custom Statistics

In this section...

“Overview of Approaches to Custom Statistics” on page 11-7

“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 11-9

“Example: Resetting an Average Periodically” on page 11-12

Overview of Approaches to Custom Statistics

You can use the built-in statistical signals from SimEvents blocks to derive more specialized or complex statistics that are meaningful in your model. One approach is to compute statistics during the simulation. You can implement your computations using a graphical block-diagram approach or a nongraphical coded approach. Alternatively, you can compute statistics after the simulation is complete.

Graphical Block-Diagram Approach

The Math Operations library in the Simulink library set and the Statistics library in the DSP System Toolbox™ library set can help you compute statistics using blocks. For examples using Simulink blocks, see

- “Example: Fraction of Dropped Messages” on page 11-8
- “Example: Detecting Changes in the Last-Updated Signal” on page 14-46, 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 Modeling Load Within a Dynamic Voltage Scaling Application demo
- The Arrival Rate Estimation Computation subsystem within the Arrival Rate Estimator subsystem in the Estimating Entity Arrival Rates demo

Coded Approach

The blocks in the User-Defined Functions library in the Simulink library set can help you compute statistics using code. For examples using the MATLAB Function block, see

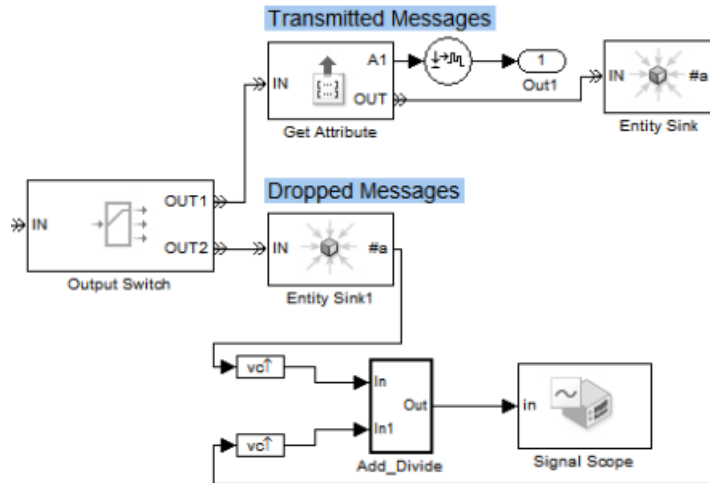
- “Example: Computing a Time Average of a Signal” on page 11-9
- “Example: Resetting an Average Periodically” on page 11-12

Post-Simulation Analysis

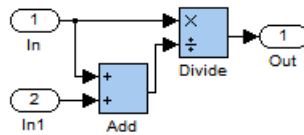
You can use the Discrete Event Signal to Workspace block to log data to the MATLAB workspace and compute statistics after the simulation is complete.

Example: Fraction of Dropped Messages

The example below shows how to compute a ratio of event-based signals in a subsystem that executes when either signal has a sample time hit. 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.

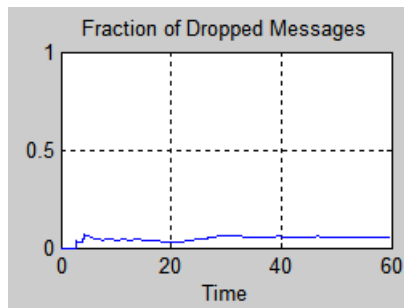


Upper-Level System



Subsystem Contents

Two Entity Sink blocks produce **#a** signals that indicate how many messages the communication link transmits or drops, respectively. The subsystem divides the number of dropped messages by the sum of the two **#a** signals. Because the subsystem performs the division only when one of the **#a** signals increases, no division-by-zero instances occur.

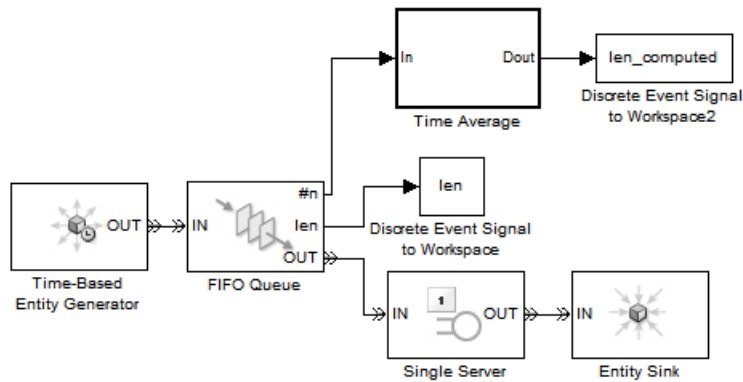


Example: Computing a Time Average of a Signal

This example illustrates how to compute a time average of a signal using the 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

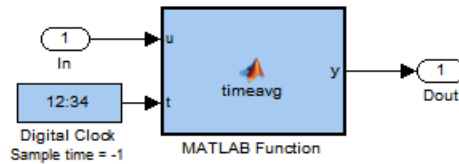
The subsystem uses **#n** to compute the time average. In this case, the time average should equal **len**. You can use a similar subsystem in your own models to compute the time averages of other signals.

Computation of the Time Average

In the example, the subsystem performs computations each time a customer arrives at or departs from the queue. Within the subsystem, the 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 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.

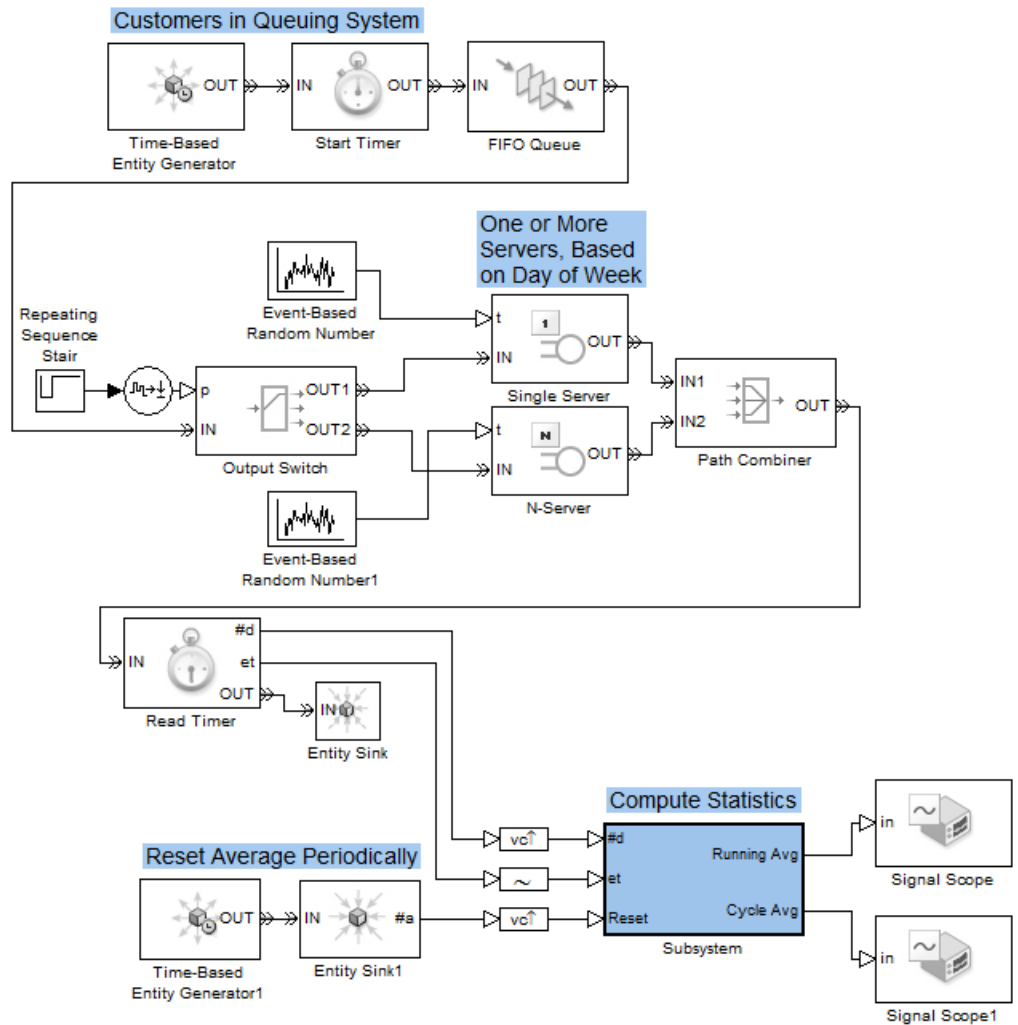
```
ans =
```

```
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.

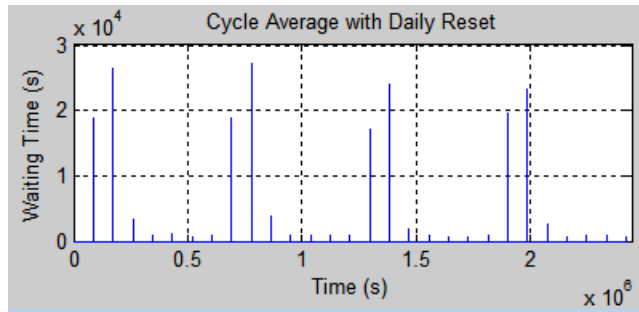
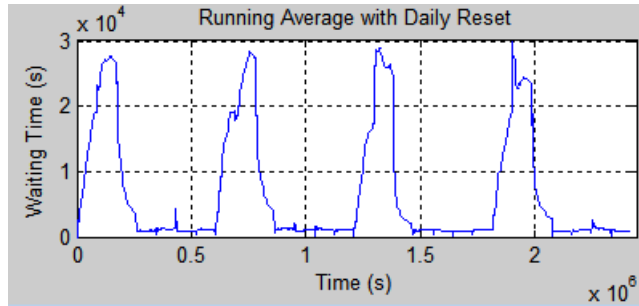
The model below uses a time-based Repeating Sequence Stair block to determine whether entities advance to a Single Server or N-Server block, thus creating variations in the number of operational servers. The Start Timer and Read Timer blocks compute each entity's waiting time in the queuing system. A computational 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 11-15.



Top-Level Model

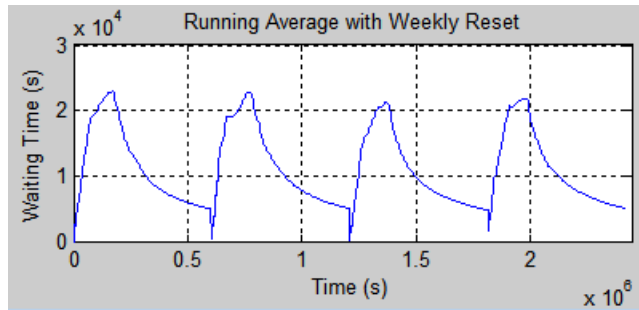
Performance of Daily Averages

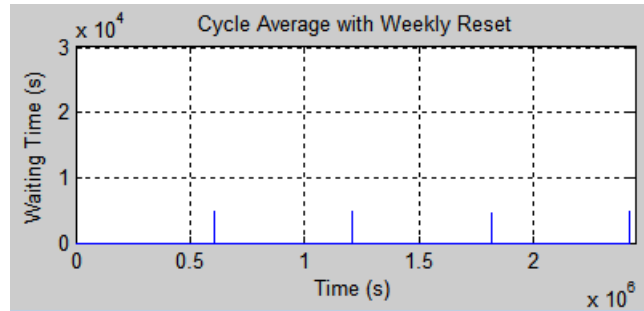
When considering daily cycles, you can see that the cycle averages do not stabilize at a single value.



Performance of Weekly Averages

When considering weekly cycles, you can see less variation in the cycle averages because each cycle contains the same pattern of changing service levels. To compute the cycle average over a weekly cycle, change the **Period** parameter in the Time-Based Entity Generator1 block at the bottom of the model to $60 \times 60 \times 24 \times 7$, which is the number of seconds in a week.





Computation of the Cycle Average

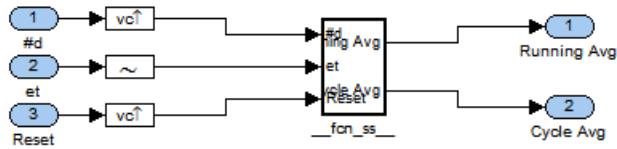
In the example, the subsystem performs computations each time a customer departs from the queuing system and at each boundary of a daily or weekly cycle. Within the subsystem, the MATLAB Function block counts the number of customers and the total waiting time among all customers at that point. The block 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 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 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 AtomicSubsystem 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
```

Measuring Point-to-Point Delays

In this section...
“Overview of Timers” on page 11-18
“Basic Example Using Timer Blocks” on page 11-19
“Basic Procedure for Using Timer Blocks” on page 11-20
“Timing Multiple Entity Paths with One Timer” on page 11-21
“Restarting a Timer from Zero” on page 11-22
“Timing Multiple Processes Independently” on page 11-22

Overview of Timers

Suppose you want to determine how long each entity takes to advance from one block to another, or how much time each entity spends in a particular region of your model. To compute these durations, you can attach a timer to each entity that reaches a particular spot in the model. Then you can

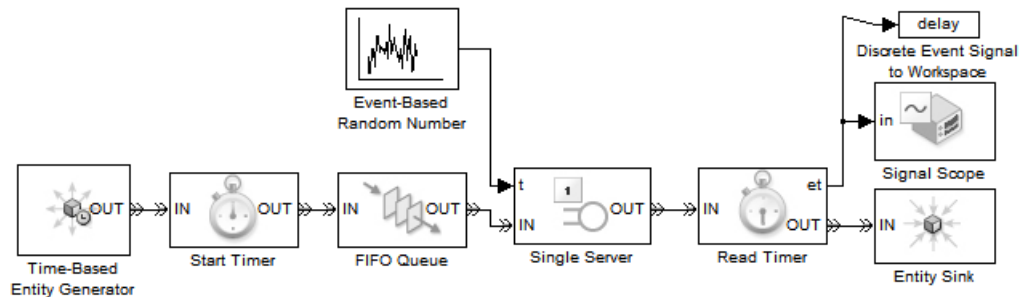
- Start the timer. The block that attaches the timer also starts it.
- Read the value of the timer whenever the entity reaches a spot in the model that you designate.
- Restart the timer, if desired, whenever the entity reaches a spot in the model that you designate.

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

Note Timers measure durations, or relative time. By contrast, clocks measure absolute time. For details about implementing clocks, see the descriptions of the Clock and Digital Clock blocks in the Simulink documentation.

Basic Example Using Timer Blocks

A typical block diagram for determining how long each entity spends in a region of the model is in the figure below. The Start Timer and Read Timer blocks jointly perform the timing computation.



The model above measures the time each entity takes between arriving at the queue and departing from the server. The Start Timer block attaches, or associates, a timer to each entity that arrives at the block. Each entity has its own timer. Each entity's timer starts timing when the entity departs from the Start Timer, or equivalently, when the entity arrives at the FIFO Queue block. Upon departing from the Single Server block, each entity arrives at a Read Timer block. The Read Timer block reads data from the arriving entity's timer and produces a signal at the **et** port whose value is the instantaneous elapsed time for that entity. For example, if the arriving entity spent 12 seconds in the queue-server pair, then the **et** signal assumes the value 12.

Basic Example of Post-Simulation Analysis of Timer Data

The model above stores data from the timer in a variable called `delay` in the base MATLAB workspace. After running the simulation, you can manipulate or plot the data, as illustrated below.

```
% First run the simulation shown above, to create the variable
% "delay" in the MATLAB workspace.

% Histogram of delay values
edges = (0:20); % Edges of bins in histogram
counts = histc(delay.signals.values, edges); % Number of points per bin
figure(1); bar(edges, counts); % Plot histogram.
```

```
title('Histogram of Delay Values')

% Cumulative histogram of delay values
sums = cumsum(counts); % Cumulative sum of histogram counts
figure(2); bar(edges, sums); % Plot cumulative histogram.
title('Cumulative Histogram of Delay Values')
```

Basic Procedure for Using Timer Blocks

A typical procedure for setting up timer blocks is as follows:

- 1** Locate the spots in the model where you want to begin timing and to access the value of the timer.
- 2** Insert a Start Timer block in the model at the spot where you want to begin timing.
- 3** In the Start Timer block's dialog box, enter a name for the timer in the **Timer tag** field. This timer tag distinguishes the timer from other independent timers that might already be associated with the same entity.

When an entity arrives at the Start Timer block, the block attaches a named timer to the entity and begins timing.

- 4** Insert a Read Timer block in the model at the spot where you want to access the value of the timer.
- 5** In the Read Timer block's dialog box, enter the same **Timer tag** value that you used in the corresponding Start Timer block.

When an entity having a timer with the specified timer tag arrives at the block, the block reads the time from that entity's timer. Using the **Statistics** tab of the Read Timer block's dialog box, you can configure the block to report this instantaneous time or the average of such values among all entities that have arrived at the block.

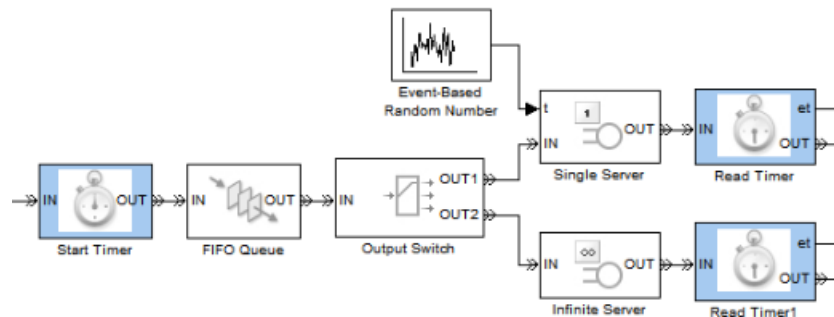
If you need multiple independent timers per entity (for example, to time an entity's progress through two possibly overlapping regions of the model), then follow the procedure above for each of the independent timers. For more information, see "Timing Multiple Processes Independently" on page 11-22.

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.

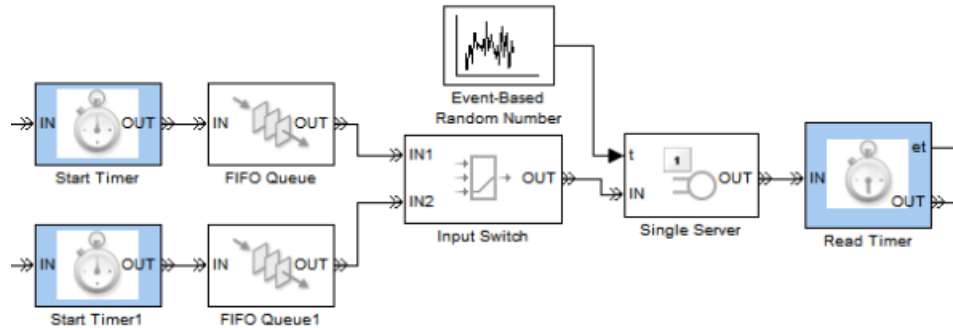
Output Switch Example

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

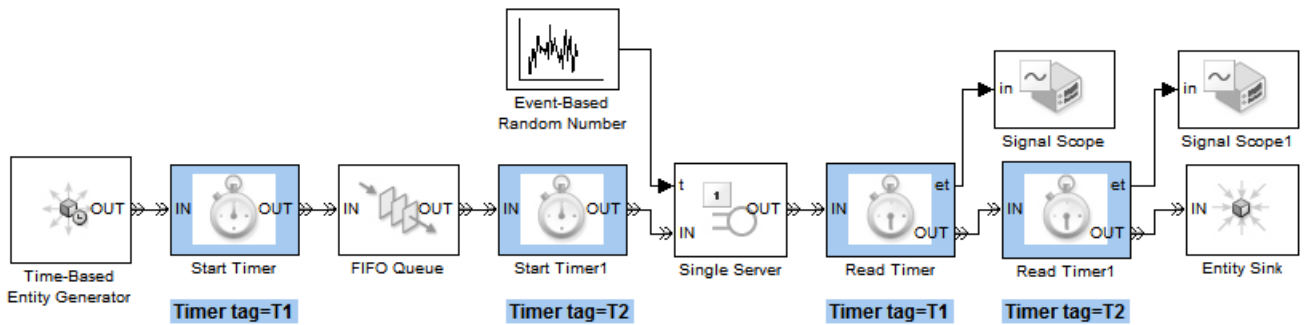
You can restart an entity's timer, that is, reset its value to zero, whenever the entity reaches a spot in the model that you designate. To do this, insert a Start Timer block in the model where you want to restart the timer. Then set the block's **If timer has already started** parameter to Restart.

Timing Multiple Processes Independently

You can measure multiple independent durations using the Start Timer and Read Timer blocks. To do this, create a unique **Timer tag** parameter for each independent timer. For clarity in your model, consider adding an annotation or changing the block names to reflect the **Timer tag** parameter in each timer block.

The figure below shows how to measure these quantities independently:

- The time each entity spends in the queue-server pair, using a timer with tag T1
- The time each entity spends in the server, using a timer with tag T2



The annotations beneath the blocks in the figure indicate the values of the **Timer tag** parameters. Notice that the T1 timer starts at the time when entities arrive at the queue, while the T2 timer starts at the time when entities depart from the queue (equivalently, at the time when entities arrive at the server). The two Read Timer blocks read both timers when entities depart from the server. The sequence of the Read Timer blocks relative to each other is not relevant in this example because no time elapses while an entity is in a Read Timer block.

Varying Simulation Results by Managing Seeds

In this section...

- “Connection Between Random Numbers and Seeds” on page 11-24
- “Making Results Repeatable by Storing Sets of Seeds” on page 11-25
- “Setting Seed Values Programmatically” on page 11-26
- “Sharing Seeds Among Models” on page 11-26
- “Working with Seeds Not in SimEvents Blocks” on page 11-27
- “Choosing Seed Values” on page 11-29

See also “Detecting Nonunique Seeds and Making Them Unique” on page 13-90.

Connection Between Random Numbers and Seeds

When a simulation uses random numbers and you compute statistical results from it, you typically want to use different sequences of random numbers in these situations:

- In the random processes of a single simulation run
- Across multiple simulation runs

To vary a sequence of random numbers, vary the *initial seed* on which the sequence of random numbers is based. SimEvents blocks that have a parameter called **Initial seed** include:

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

Some blocks in other library sets have parameters that represent initial seeds. For example, the Random Number and Uniform Random Number blocks in the Simulink Sources library have parameters called **Initial seed**.

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

Making Results Repeatable by Storing Sets of Seeds

If you need to repeat the results of a simulation run and expect to change random number sequences, then you should store the seeds before changing them. You can later repeat the simulation results by resetting the stored seeds; see “Setting Seed Values Programmatically” on page 11-26 to learn more.

When all seeds are parameters of SimEvents blocks, use this procedure to store the seeds:

- 1 Decide whether you want to store seeds from SimEvents blocks in a system (including subsystems at any depth) or from a single block.
- 2 Create a string variable (called `sysid`, for example) that represents the system name, subsystem path name, or block path name.

Tip To avoid typing names, use `gcb` or `gcs`:

- Select a subsystem or block and assign `sysid = gcb`.
 - Click in a system or subsystem and assign `sysid = gcs`.
-

- 3 Use the `se_getseeds` function with `sysid` as the input argument. The output is a structure having these fields:
 - `system` — Value of the `sysid` input to `se_getseeds`
 - `seeds` — Structure array, of which each element has these fields:
 - `block` — Path name of a block that uses a random number generator, relative to `system`
 - `value` — Numeric seed value of the block

- 4 Store the output in an array, cell array, or MAT-file. Use a MAT-file if you might need to recover the values in a different session.

For an example that uses `se_getseeds`, see the Seed Management Workflow for Random Number Generators demo.

If your model uses random numbers in contexts other than SimEvents blocks, see “Working with Seeds Not in SimEvents Blocks” on page 11-27.

Setting Seed Values Programmatically

To set seed values programmatically in blocks that use random numbers, use one or more of these approaches:

- If you have a seed structure in the same format as the output of the `se_getseeds` function, use the `se_setseeds` function to set seed values in the corresponding blocks.

For an example, see the Seed Management Workflow for Random Number Generators demo.

- If you want the application to choose seed values for you and then set the values in some or all SimEvents blocks, use the `se_randomizeseeds` function.

For examples, see the Managing Seeds For Random Number Generation demo. To learn about specific options for using `se_randomizeseeds`, see its reference page.

- If your model uses random numbers in contexts other than SimEvents blocks, use the `set_param` function to set seed values.

For examples, see “Working with Seeds Not in SimEvents Blocks” on page 11-27.

Sharing Seeds Among Models

Suppose you want to share seeds among multiple variants of a model or among models that have a common subsystem. The `se_getseeds` and `se_setseeds` functions provide a convenient way to apply seed values of the SimEvents blocks in one model to the corresponding blocks in a second model. Use this procedure:

- 1 Create string variables (for example, `sys1` and `sys2`) that represent the system names of the two models.
- 2 Open both models, if you have not already done so.
- 3 Use the `se_getseeds` function with `sys1` as the input argument. The result is a seed structure that represents the seeds in the SimEvents blocks in model `sys1`.
- 4 Use the `se_setseeds` function with the seed structure as the first input argument and `sys2` as the second input argument. The function uses information from the seed structure but overrides the system name stored in the seed structure. As a result, the function sets the seeds in model `sys2` to values from model `sys1`.

Working with Seeds Not in SimEvents Blocks

The seed management features in SimEvents software cover blocks in the SimEvents libraries. If your model uses random number sequences in other blocks or in the **Seed for event randomization** configuration parameter, you can use `get_param` and `set_param` commands to retrieve and set the seeds, respectively. These examples illustrate the techniques:

Example: Retrieving and Changing a Seed in a Custom Subsystem

This example illustrates how to identify relevant variable names for seed parameters, query seed values, and set seed values. The specific block in this example is the Uniform Random Number block within a custom masked subsystem in a demo model.

- 1 Open the demo model.

```
sedemo_md1
```

- 2 Select the block labeled Exponential Generation and store its path name. Exponential Generation is a custom masked subsystem that has a seed parameter related to a Uniform Random Number block under the mask.

```
blk = gcb; % Pathname of current block
```

- 3 Query the dialog parameters of the block.

```
vars = get_param(blk, 'DialogParameters')
```

```
vars =
```

```
    seed: [1x1 struct]
```

The term `seed` in the output indicates a parameter's underlying variable name, which can differ from the text label you see in the block dialog box. You might guess that `seed` represents the seed of a random number generator. Optionally, you can confirm that this variable name corresponds to the **Initial seed** text label in the dialog box using this command:

```
textlabel = vars.seed.Prompt
```

```
textlabel =
```

```
Initial seed
```

4 Query the seed parameter for its value.

```
thisseed = get_param(blk, 'seed')
```

```
thisseed =
```

```
60790
```

5 Change the value of the `seed` parameter to a constant.

```
newseed = '60791'; % String whose value is a number  
set_param(blk, 'seed', newseed);
```

See “Choosing Seed Values” on page 11-29 for criteria related to the values you choose for seeds.

6 Change the value of the `seed` parameter to the name of a variable in the workspace. As a result, the dialog box shows the name of the variable instead of the value stored in the variable. This approach might be useful if you want to use `set_param` once and then change the workspace variable repeatedly (for example, within a loop) to vary the seed value.

```
seedvariable = 60792; % Numeric variable  
set_param(blk, 'seed', ...
```

```
'seedvariable'); % Parameter refers to variable
```

Choosing Seed Values

Here are some recommendations for choosing appropriate values for seed parameters of blocks:

- If you choose a seed value yourself, choose an integer between 0 and $2^{32}-1$.
- To obtain the same sequence of random numbers the next time you run the same simulation, set the seed to a fixed value.
- To obtain a different sequence of random numbers the next time you run the same simulation, use one of these approaches:
 - Change the value of the seed, using the `se_randomize_seeds` function or any other means.
 - Set the value of the seed to a varying expression such as `mod(ceil(cputime*99999),2^32)`. See the `cputime` function for more details.
- If seed parameters appear in multiple places in your model, choose different values, or expressions that evaluate to different values, for all seed parameters. To have the application detect nonunique seeds in SimEvents blocks, use the “Identical seeds for random number generators” configuration parameter. To learn how to make seeds unique in SimEvents blocks across a model, see “Detecting Nonunique Seeds and Making Them Unique” on page 13-90.

Regulating the Simulation Length

In this section...

“Overview” on page 11-30

“Setting a Fixed Stop Time” on page 11-30

“Stopping Upon Processing a Fixed Number of Entities” on page 11-31

“Stopping Upon Reaching a Particular State” on page 11-32

Overview

When you gather statistics from a simulation, ending the simulation at the right time is more important than if you are only observing behavior qualitatively. Typical criteria for ending a discrete-event simulation include the following:

- A fixed amount of time passes
- The simulation processes a fixed number of packets, parts, customers, or other items that entities represent
- The simulation achieves a particular state, such as an overflow or a machine failure

Setting a Fixed Stop Time

To run a simulation interactively with a fixed stop time, do the following:

- 1** Open the **Configuration Parameters** dialog box by choosing **Simulation > Configuration Parameters** in the menu of the model window.
- 2** In the dialog box, set **Stop time** to the desired stop time.
- 3** Run the simulation by choosing **Simulation > Start**.

To fix the stop time when running a simulation programmatically, use syntax like

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

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

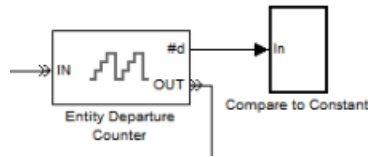
Stopping Upon Processing a Fixed Number of Entities

By counting entities, you can stop the simulation when the simulation processes a fixed number of entities. The basic procedure for stopping a simulation based on the total number of entity departures from a block is:

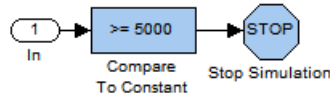
- 1 Find the parameter of the block that enables the departure counter as a signal output. Most blocks call the parameter **Number of entities departed**. Exceptions are in the following table.

Block	Parameter
Entity Departure Counter	Write count to signal port #d
Entity Sink	Number of entities arrived

- 2 Set the check box. This setting causes the block to have a signal output port corresponding to the entity count.
- 3 Connect the new signal output port to an Atomic Subsystem block.
- 4 Double-click the subsystem block to open the subsystem it represents.
- 5 Delete the Outport block labeled Out.
- 6 Connect the Inport block labeled In to a Compare To Constant block.
- 7 In the Compare To Constant block,
 - Set **Operator** to `>=`.
 - Set **Constant value** to the desired number of entity departures.
 - Set **Output data type mode** to `boolean`.
- 8 Connect the Compare To Constant block to a Stop Simulation block. 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

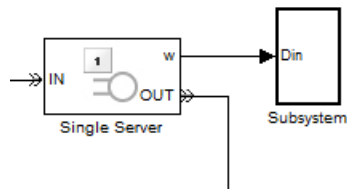
See the considerations discussed in “Tips for Using State-Based Stopping Conditions” on page 11-35 below. They are relevant if you are stopping the simulation based on an entity count, where “desired state” means the entity-count threshold.

Stopping Upon Reaching a Particular State

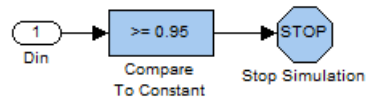
Suppose you want the simulation to end when it achieves a particular state, such as an overflow or a machine failure. The state might be the only criterion for ending the simulation, or the state might be one of multiple criteria, each of which is sufficient reason to end the simulation. An example that uses multiple criteria is a military simulation that ends when all identified targets are destroyed or all resources (ammunition, aircraft, etc.) are depleted, whichever occurs first.

Once you have identified a state that is relevant for ending the simulation, you typically create a Boolean signal that queries the state and connect the signal to a Stop Simulation block. Typical ways to create a Boolean signal that queries a state include the following:

- Connect a signal to a logic block to determine whether the signal satisfies some condition. See the blocks in the Simulink Logic and Bit Operations library. The following figure below illustrates one possibility.

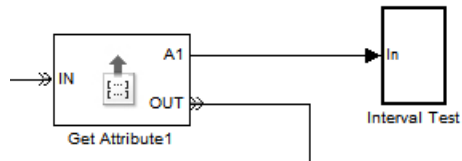


Top-Level Model

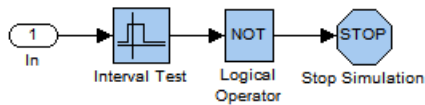


Subsystem Contents

- Use a Get Attribute block to query an attribute and a logic block to determine whether the attribute value satisfies some condition. The next figure illustrates one possibility.

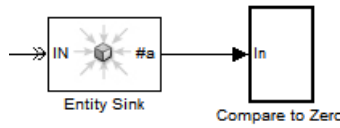


Top-Level Model

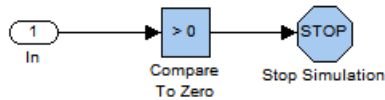


Subsystem Contents

- To end the simulation whenever an entity reaches a particular entity path, you can end that path with an Entity Sink block, enable that block's output signal to count entities, and check whether the output signal is greater than zero.

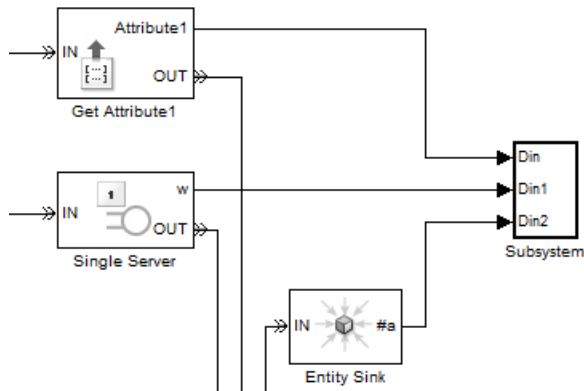


Top-Level Model

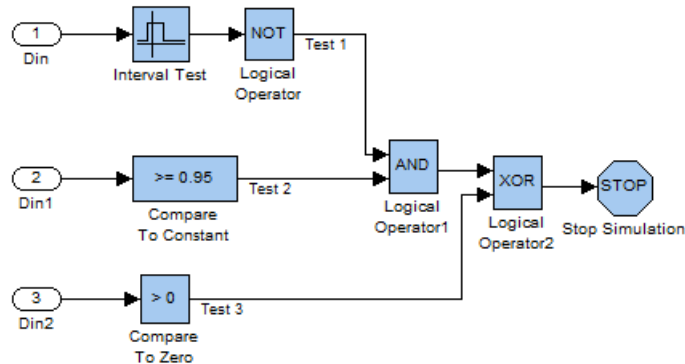


Subsystem Contents

- Logically combine multiple tests using logic blocks to build the final Boolean signal that connects to a Stop Simulation block. (A logical OR operation is implied if your model contains an independent Stop Simulation block for each of the multiple tests, meaning that the simulation ends when the first such block processes an input signal whose value is true.) The figure below illustrates one possibility using the exclusive-OR of two tests, one of which is in turn the logical AND of two tests.



Top-Level Model



Subsystem Contents

Tips for Using State-Based Stopping Conditions

When using a state rather than a time to determine when the simulation ends, keep in mind the following considerations:

- If the model has a finite stop time, then the simulation might end before reaching the desired state. Depending on your needs, this might be a desirable or undesirable outcome. If it is important that the simulation not stop too early, then you can follow the instructions in “Setting a Fixed Stop Time” on page 11-30 and use **Inf** as the **Stop time** parameter.
- If you set the **Stop time** parameter to **Inf**, then you should ensure that the simulation actually stops. For example, if you want to stop based on an entity count but the simulation either reaches a deadlock or sends most entities on a path not involving the block whose departure count is the stopping criterion, then the simulation might not end.
- Checking for the desired state throughout the simulation might make the simulation run more slowly than if you used a fixed stop time.

Using Stateflow Charts in SimEvents Models

- “Role of Stateflow Charts in SimEvents Models” on page 12-2
- “Guidelines for Using Stateflow and SimEvents Blocks” on page 12-3
- “Examples Using Stateflow Charts and SimEvents Blocks” on page 12-4

Role of Stateflow Charts in SimEvents Models

SimEvents software works with Stateflow software to represent systems containing state-transition diagrams that can produce or be controlled by discrete events. Both software products are related to event-driven modeling, but they play different roles:

- SimEvents blocks can model the movement of entities through a system so you can learn how such movement relates to overall system activity. Entities can carry data with them. Also, SimEvents blocks can generate events at times that are truly independent of the time steps dictated by the ODE solver in Simulink software.
- Stateflow charts can model the state of a block or system. Charts enumerate the possible values of the state and describe the conditions that cause a state transition. Runtime animation in a Stateflow chart depicts transitions but does not indicate movement of data.

For scenarios that combine SimEvents blocks with Stateflow charts, see “Examples Using Stateflow Charts and SimEvents Blocks” on page 12-4.

You can interpret the Signal Latch block with the `st` output signal enabled as a two-state machine that changes state when read and write events occur. Similarly, you can interpret Input Switch and Output Switch blocks as finite-state machines whose state is the selected entity port. However, Stateflow software offers more flexibility in the kinds of state machines you can model and an intuitive development environment that includes animation of state transitions during the simulation.

Guidelines for Using Stateflow and SimEvents Blocks

When your model contains Stateflow charts in addition to SimEvents blocks, you must follow these rules:

- Insert a Timed to Event Signal gateway block between Stateflow output and SimEvents input.
- If the chart is capable of propagating its execution context, select this option as follows:
 - 1 Select the Stateflow block and choose **Edit > Subsystem Parameters** from the model window's menu bar.
 - 2 In the dialog box that opens, select **Propagate execution context across subsystem boundary** if it appears and click **OK**. If this parameter does not appear in the dialog box, just click **OK**.

Note If the chart does not offer this option, you might see a delay in the response of other blocks to the chart's output signals. The duration of the delay is the time between successive calls to the chart.

- If an output of the chart connects to a SimEvents block, do not configure the chart to be entered at initialization. To ensure that this configuration is correct,
 - 1 Select the **File > Chart Properties** from the chart window's menu bar.
 - 2 In the dialog box that opens, clear **Execute (enter) Chart At Initialization** and click **OK**. This check box is cleared by default.

When you design default transitions in your chart, keep in mind that the chart will not be entered at initialization. For example, notice that the default transition in the example in “Example: Failure and Repair of a Server” on page 5-22 indicates the state corresponding to the first actual event during the simulation, not an initial state.

- If the chart has an output signal, you can provide a nonzero initial output using the Initial Value block as in “Specifying Initial Values of Event-Based Signals” on page 4-14. Because the chart is not entered at initialization, you cannot use the chart itself to provide a nonzero initial output.

Examples Using Stateflow Charts and SimEvents Blocks

In this section...
“Failure State of Server” on page 12-4
“Go-Back-N ARQ Model” on page 12-4

Failure State of Server

The examples in “Using Stateflow Charts to Implement a Failure State” on page 5-21 use Stateflow charts to implement the logic that determines whether a server is down, under repair, or operational. SimEvents blocks model the asynchronous arrival of customers, advancement of customers through a queue and server, and asynchronous failures of the server. While these examples could alternatively have represented the server’s states using signal values instead of states of a Stateflow chart, the chart approach is more intuitive and scales more easily to include additional complexity.

Go-Back-N ARQ Model

The Packet Communication Within a Go-Back-N ARQ System demo uses SimEvents and Stateflow blocks to model a communication system. SimEvents blocks implement the movement of data frames and acknowledgment messages from one part of the system to another. Stateflow blocks implement the logical transitions among finitely many state values of the transmitter and the receiver.

Receiver State

At the receiver, the chart decides whether to accept or discard an incoming frame of data, records the identifier of the last accepted frame, and regulates the creation of acknowledgment messages. Interactions between the Stateflow chart and SimEvents blocks include these:

- The arrival of an entity representing a data frame causes the generation of a function call that invokes the chart.
- The chart can produce a routing signal that determines which path entities take at an Output Switch block.

- The chart can produce a function call that causes the Event-Based Entity Generator block to generate an entity representing an acknowledgment message.

Transmitter State

At the transmitter, the chart controls the transmission and retransmission of frames. Interactions between the Stateflow chart and SimEvents blocks include these:

- The arrival of an entity representing a new data frame or an acknowledgment message causes the generation of a function call that invokes the chart.
- The completion of transmission of a frame (that is, the completion of service on an entity representing a frame) causes the generation of a function call that invokes the chart.
- The chart can produce a routing signal that determines which path entities take at an Output Switch block.
- The chart can produce a function call that causes the Release Gate block to permit the advancement of an entity representing a data frame to transmit (function call at Stateflow block's tx output port) or retransmit (function call at Stateflow block's retx output port).

Debugging Discrete-Event Simulations

- “Overview of Debugging Resources” on page 13-2
- “Overview of the SimEvents Debugger” on page 13-3
- “Starting the SimEvents Debugger” on page 13-5
- “The Debugger Environment” on page 13-7
- “Independent Operations and Consequences in the Debugger” on page 13-21
- “Stopping the Debugger” on page 13-25
- “Stepping Through the Simulation” on page 13-27
- “Inspecting the Current Point in the Debugger” on page 13-32
- “Inspecting Entities, Blocks, and Events” on page 13-34
- “Working with Debugging Information in Variables” on page 13-41
- “Viewing the Event Calendar” on page 13-46
- “Customizing the Debugger Simulation Log” on page 13-47
- “Debugger Efficiency Tips” on page 13-55
- “Defining a Breakpoint” on page 13-57
- “Using Breakpoints During Debugging” on page 13-63
- “Block Operations Relevant for Block Breakpoints” on page 13-67
- “Animating” on page 13-74
- “Common Problems in SimEvents Models” on page 13-79
- “Recognizing Latency in Signal Updates” on page 13-98

Overview of Debugging Resources

To Read About...	Refer to...
Running the simulation using the SimEvents debugger	“Overview of the SimEvents Debugger” on page 13-3
Some modeling errors and ways to avoid them	“Common Problems in SimEvents Models” on page 13-79
Plotting signals, attribute values, event information, or entity information during the simulation	“Using Plots for Troubleshooting” on page 10-12
Gathering data during the simulation that reflects block behavior	“Accessing Statistics from SimEvents Blocks” on page 11-5 and “Sending Data to the MATLAB Workspace” on page 4-17
Examining the set or processing sequence of simultaneous events	“Exploring Simultaneous Events” on page 3-4

Overview of the SimEvents Debugger

SimEvents software includes a debugger that lets you use MATLAB functions to suspend a simulation at each step or breakpoint, and query simulation behavior. The debugger also creates a simulation log with detailed information about what happens during the simulation.

This table indicates sources of relevant information about the debugger. For information about debugging resources other than the debugger, see “Overview of Debugging Resources” on page 13-2.

To Read About...	Refer to...	Description
Functions	Debugger Function Reference	List of functions related to debugging
Examples	“Confirming Event-Based Behavior Using the SimEvents Debugger”, part of an example in the SimEvents getting started documentation	Stepping through a simulation
	“Exploring the D/D/1 System Using the SimEvents Debugger”, part of an example in the SimEvents getting started documentation	Querying the final state of a simulation
	A video tutorial on the Web, in two parts: <ul style="list-style-type: none"> • Basic Single Stepping and Querying • Breakpoints and Advanced Querying 	Stepping, querying, and using breakpoints
	“Example: Choices of Values for Event Priorities” on page 3-11	Interpreting the simulation log
	“Example: Preemption by High-Priority Entities” on page 5-11	Stepping forward from a breakpoint and interpreting the simulation log
	“Example: Deadlock Resulting from Loop in Entity Path” on page 13-85	Inferring the reasons for a simulation deadlock

To Read About...	Refer to...	Description
Procedures	“Starting the SimEvents Debugger” on page 13-5	How to start the debugger
	“Stepping Through the Simulation” on page 13-27 and “Using Breakpoints During Debugging” on page 13-63	How to control the simulation
	“Inspecting Entities, Blocks, and Events” on page 13-34 and “Customizing the Debugger Simulation Log” on page 13-47	How to get information
Background	“The Debugger Environment” on page 13-7	Working in debug mode and interpreting debugger displays

Starting the SimEvents Debugger

Before you simulate a system using the SimEvents debugger, make sure that the system is not currently simulating and that you are not running the Simulink debugger on any system. Also ensure that the system contains at least one SimEvents block. The SimEvents debugger does not work on systems without these blocks.

To simulate a system using the SimEvents debugger, enter

```
sedebug(sys)
```

at the MATLAB command prompt. `sys` is a string representing the system name. An example is `sedebug('sedemo_outputswitch')`.

The results of starting the debugger are:

- The output in the MATLAB Command Window indicates that the debugger is active and includes hyperlinks to sources of information.

```
*** SimEvents Debugger ***
```

```
Functions | Help | Watch Video Tutorial
```

```
%=====
```

```
Initializing Model sedemo_outputswitch
```

```
sedebug>>
```

- The command prompt changes to `sedebug>>`. This notation is the debugger prompt where you enter commands.
- The system starts initializing. At this point, you can inspect initial states or configure the debugger before blocks have started performing operations.
- You cannot modify the system, modify parameters in the system or in its blocks, close the system, or end the MATLAB session until the debugger session ends. To end the debugger session, see “Stopping the Debugger” on page 13-25.

For more information on how to proceed, see one of these sections:

- “The Debugger Environment” on page 13-7
- “Stepping Through the Simulation” on page 13-27
- “Confirming Event-Based Behavior Using the SimEvents Debugger”, part of an example in the SimEvents getting started documentation
- A video tutorial on the Web, in two parts:
 - Basic Single Stepping and Querying
 - Breakpoints and Advanced Querying

The Debugger Environment

In this section...

“Debugger Command Prompt” on page 13-7

“Simulation Log in the Debugger” on page 13-8

“Identifiers in the Debugger” on page 13-19

Debugger Command Prompt

When the SimEvents debugger is active, the command prompt is `sedebug>>` instead of `>>`. The `sedebug>>` prompt reminds you that the simulation is suspended in debugging mode.

When you enter commands at the `sedebug>>` prompt, you can:

- Invoke debugger functions using only the function names, without the package qualifier, `sedb..` For example, you can enter `step` at the `sedebug>>` prompt even though the fully qualified name of the function is `sedb.step`.

Debugger functions that you invoke at the `sedebug>>` prompt are in the `sedb` package, so their fully qualified names start with “`sedb.`”. You can either include or omit the `sedb.` prefix when entering commands at the `sedebug>>` prompt because the `sedebug` function imports the `sedb` package.

- See a list of debugger functions by entering `help` with no input arguments.

At the `>>` prompt, the same list is available via the syntax `help sedb`.

When you enter commands at the `sedebug>>` prompt, follow these rules:

- Do not append additional debugger commands on the same line as a command that causes the simulation to proceed. For example, to step twice, you must enter `step` on two lines instead of entering the single command `step; step`.
- Do not invoke `sedebug` or `sldebug`.

In the `sedb` package, functions are valid only when the SimEvents debugger is active.

Simulation Log in the Debugger

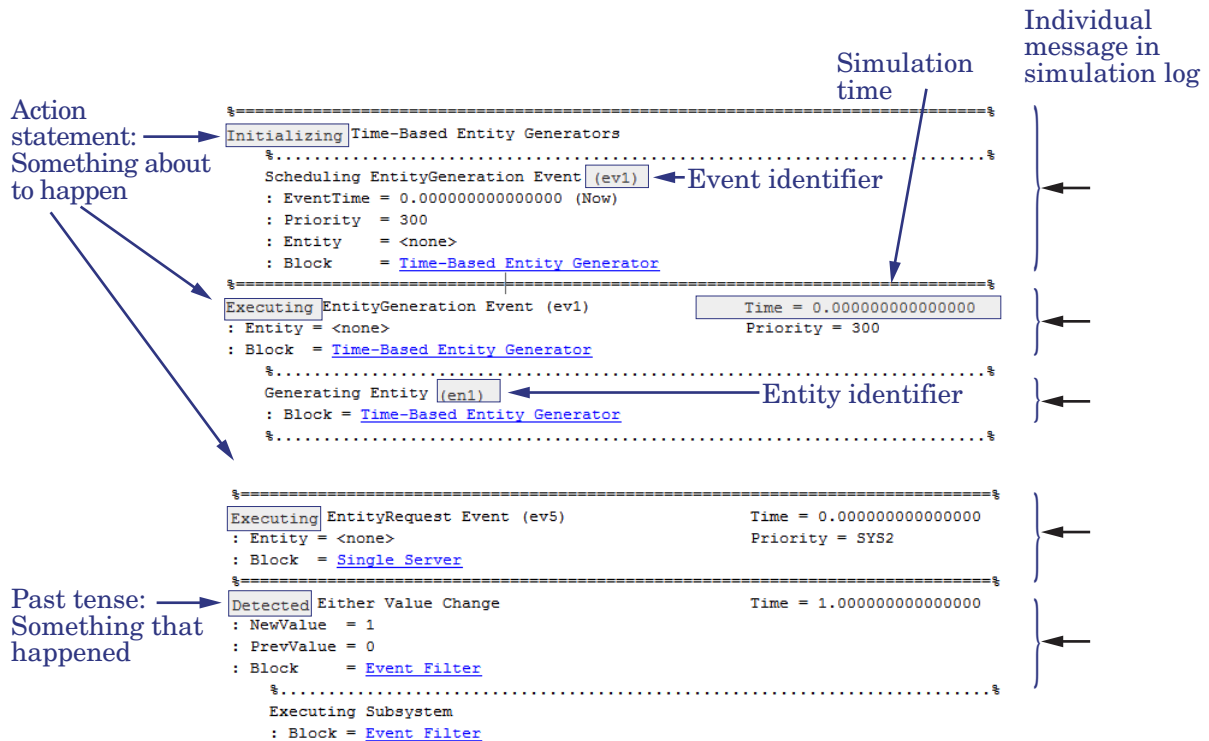
When the simulation proceeds during debugging mode, the debugger displays messages in the Command Window to indicate what is about to happen or what has just happened in the simulation. Collectively, these messages make up the simulation log. These topics describe what you see in the simulation log:

- “Key Parts of the Simulation Log” on page 13-8
- “Optional Displays of Event Calendar in the Simulation Log” on page 13-10
- “Interpreting the Simulation Log” on page 13-12
- “Event Scheduling, Execution, and Cancellation Messages” on page 13-13
- “Detection Messages” on page 13-15
- “Entity Operation Messages” on page 13-16
- “Monitoring Block Messages” on page 13-17
- “Initialization Messages” on page 13-18
- “Signal Operation Messages” on page 13-18
- “Execution of Time-Based Blocks in Event-based Systems Block Messages” on page 13-19

To learn how to obtain additional information via query commands, see “Inspecting Entities, Blocks, and Events” on page 13-34.

Key Parts of the Simulation Log

Here is a sample excerpt of the simulation log in its default configuration.



The sample highlights these typical features of messages in the simulation log:

- Individual messages typically span multiple lines in the Command Window. Lines that look like `%====%` or `%. . . . %` precede each individual message.
- The first line of each message provides a summary. The phrasing of the summary indicates whether the message describes something that is about to happen or that has just happened. When you inspect states, knowing the difference is important.

Phrasing of Summary	Examples	Interpretation
Action statement	Executing, scheduling, advancing, generating, and others	The action is about to happen. It has not happened yet.
Verb in the past tense	Detected	The action has just happened.

- Some messages include the simulation time. Messages that do not include the simulation time describe activities at the simulation time most recently included in the simulation log.
- Some messages use tokens in parentheses to identify events or entities uniquely. For details, see “Identifiers in the Debugger” on page 13-19.
- Some messages are indented to indicate operations that are consequences of other operations. For details, see “Independent Operations and Consequences in the Debugger” on page 13-21.
- Block names are underlined and act as hyperlinks.

If you click the path, the model window opens and highlights the block. To clear the highlighting, select **View > Remove Highlighting**.

Optional Displays of Event Calendar in the Simulation Log

You can configure the debugger to list the events on the event calendar, as part of the simulation log. Event calendar listings appear before execution messages. The following illustration is a sample event calendar listing.

```

Discrete-Event System ID: 0
=====
ID      EventTime      EventType      Priority Entity      Block
=> ev5   2.0000000000000000  Timeout      1700   en2      Infinite Server
ev13   2.5000000000000000  EntityGeneration  1      <none>   Time-Based Entity Generator
ev12   2.5813012975000028  ServiceCompletion  1      en4      Infinite Server
ev6    2.730384939625469  ServiceCompletion  1      en2      Infinite Server
ev11   3.0000000000000000  Timeout      1700   en4      Infinite Server
=====
Executing Timeout Event (ev5)                Time = 2.0000000000000000
: Entity = en2                               Priority = 1700
: Block = Infinite Server

```

The event calendar display spans multiple lines in the Command Window. A line that looks like % - - - % precedes the event calendar display.

Each listing in the event calendar display includes these event characteristics:

- Discrete-event system identifier, which identifies the discrete-event system during the debugger session. See “Discrete-Event Simulation Using SimEvents in Simulink Models” for more information on discrete-event systems.
- Event identifier, which is a token that identifies the event uniquely during the debugger session. For details, see “Identifiers in the Debugger” on page 13-19.
- Scheduled time of the event.
- Event type. For details, see “Supported Events in SimEvents Models” on page 2-2.
- Event priority. The value can be a number, SYS1, or SYS2. For details, see “Event Sequencing” on page 14-9 or “Overview of Simultaneous Events” on page 3-2.
- Entity identifier of an entity associated with the event, if applicable. If no entity is associated with the event, <none> appears.
- Partial path of the block that processes the event when it occurs. The partial path omits the name of the model, but is otherwise the same as the full path of the block.

Block paths are underlined and act as hyperlinks.

If you click the path, the model window opens and highlights the block. To clear the highlighting, select **View > Remove Highlighting**.

When comparing the simulation log with the event calendar display, the sequence on the calendar and the sequence in which the events were scheduled might differ, even for simultaneous events that have equal priority. For details, see “Procedure for Specifying Equal-Priority Behavior” on page 3-9.

To learn how to configure the debugger to show event calendar displays, see “Customizing the Debugger Simulation Log” on page 13-47.

Interpreting the Simulation Log

The most recent message in the simulation log shows you the simulation status while the debugger is waiting for you to enter the next command. When you inspect states, if the phrasing of the summary uses an action statement to indicate that the action has not happened yet, the state does not reflect the completion of the action. For example, if the most recent message in the simulation log indicates that a server block is scheduling a service completion event, the scheduling has not occurred yet and the `evcal` function does not list the event on the event calendar. If you want to see where the event is on the event calendar relative to other scheduled events, you must proceed in the simulation by one step and then call `evcal`.

The simulation log shows you what is happening during the simulation. When you simulate in debugging mode, the actions in the simulation log are the units of debugging behavior. By using the debugger on those units of debugging behavior, you can control the simulation process and query simulation behavior. When you proceed in the simulation step by step, the simulation is suspended before or after the actions that the simulation log reports.

By contrast, some approaches that the simulation log does *not* intend to reflect are:

- A block-by-block simulation process. In a discrete-event simulation, a block can take multiple actions at a given time, potentially interspersed with actions of other blocks. Alternatively, a block can take no action for long periods of time, even while other blocks in the system are active.

Messages in the simulation log might not be in the same sequence that you see in the topology of the block diagram, even if the topology of the block diagram is linear.

- An entity-by-entity simulation process. An entity can advance through multiple blocks or can be the object of multiple actions at a given time, potentially interspersed with actions on other entities. Alternatively, an entity can go for long periods of time without moving, changing, or affecting anything else in the simulation, even while other entities in the system are active.
- One message per time value. In a discrete-event simulation, many actions can occur at the same time. Therefore, the debugger lets you query simulation behavior at intermediate points.

The view into simulation behavior is useful during debugging because the simulation log reflects what is happening during the simulation. When you see what is happening, you can diagnose problems, explore solutions, improve your modeling proficiency, and learn about the underlying system that you are modeling.

Event Scheduling, Execution, and Cancellation Messages

When the event calendar is about to change during a debugging session, the simulation log displays a message that describes the event that the block is about to schedule, execute, or cancel. The event calendar has not yet changed.

When a block is about to execute a subsystem, function call, memory read, or memory write event that is not on the event calendar, the simulation log displays a message that describes the event. The execution, which is a dependent operation, has not yet occurred.

Sample Scheduling Message

The following sample shows that the Single Server block is about to add a service completion event to the event calendar. The event has priority 500, a scheduled time of $T=5$, and identifier `ev3`. The event represents the completion of service on the entity whose identifier is `en2`.

```
%.....%
Scheduling ServiceCompletion Event (ev3)
```

```
: EventTime = 5.0000000000000000
: Priority   = 500
: Entity    = en2
: Block     = Single Server
```

Sample Independent Execution Message

The following sample shows that the Single Server block is about to execute the service completion event for the entity whose identifier is `en2`. The event time of $T=5$ equals the current simulation time. The event has priority 500, which is relevant if multiple events on the event calendar share the same event time. The event identifier is `ev3`.

```
%=====
Executing ServiceCompletion Event (ev3)           Time = 5.0000000000000000
: Entity = en2                                   Priority = 500
: Block  = Single Server
```

Sample Cancellation Message

The following sample shows that the Single Server block is about to cancel the service completion event for the entity whose identifier is `en2`, because the entity has timed out. The service completion event has priority 500 and identifier `ev4`.

```
%=====
Executing Timeout Event (ev3)                     Time = 1.0000000000000000
: Entity = en2                                   Priority = 1700
: Block  = Single Server
%.....%
Canceling ServiceCompletion Event (ev4)
: EventTime = 5.0000000000000000
: Priority   = 500
: Entity    = en2
: Block     = Single Server
```

For more information, see “Event Sequencing” on page 14-9 and “Example: Event Calendar Usage for a Queue-Server Model” on page 2-7.

Detection Messages

Reactive ports, listen for relevant updates in the input signal and cause an appropriate reaction in the block possessing the port. For a list of reactive ports, see “Notifying, Monitoring, and Reactive Ports” on page 14-33. When labeled reactive ports detect relevant updates during a debugging session, the simulation log displays a message whose summary starts with **Detected**. These messages indicate that the update in the signal has happened and that the block has detected it. The block has not yet responded to the update. The response depends on the particular block. For details, see individual block reference pages.

The appearance of detection messages depends on their source:

- If the input signal is time-based and enters the discrete-event system through the Timed to Event Signal gateway block, the update is an independent operation and the detection message is not indented. This kind of detection message is one of the few ways in which time-based blocks affect the simulation log.
- If the input signal is event-based, the update is a dependent operation and the detection message is indented.

Note If a port not in the list of reactive ports, such as the **function()** input port of a Function-Call Subsystem block, detects a trigger or function call, the simulation log does not display a detection message.

Sample Detection Message

The following sample shows a detection message that indicates that the Event-Based Entity Generator block has detected that its input signal changed to a new value of 1 from a previous value of 0. Because the block is configured to respond to rising value changes, the signal update is relevant. The block responds by scheduling an entity generation event for the current simulation time (denoted by *Now*).

```
%.....%
Detected Rising Value Change
: NewValue = 1
```

```
: PrevValue = 0
: Block      = Event-Based Entity Generator
%.....%
Scheduling EntityGeneration Event (ev2)
: EventTime = 0.0000000000000000 (Now)
: Priority   = SYS1
: Block     = Event-Based Entity Generator
```

Entity Operation Messages

Various blocks produce messages in the simulation log that describe how the block are about to impact entities. The details of entity operation messages depend on what information is relevant for the particular block and operation.

Sample Entity Advancement Message

An entity whose identifier is en2 is about to depart from the Schedule Timeout block and arrive at the Single Server block. The entity has not yet advanced.

```
%.....%
Entity Advancing (en2)
: From = Schedule Timeout
: To   = Single Server
```

Sample Queuing Message

A FIFO Queue block places entity en4 in the third position. Two other entities are ahead en4 in the queue, which has a total capacity of 25.

```
%.....%
Queuing Entity (en4)
: FIFO Pos = 3 of 3
: Capacity = 25
: Block = FIFO Queue
```

Sample Attribute Assignment Message

A Set Attribute block assigns the value 3 to the attribute named RepCount of entity en1.

```
%.....%
```

```

Setting Attribute on Entity (en1)
: RepCount = 3
: Block = Set Attribute

```

Sample Preemption Message

When a preemption occurs, a Single Server block replaces entity en1 with entity en4, based on values of the entities' PriorityAttributeName attribute.

```

%.....%
Preempting Entity (en1)
: NewEntity = en4 (PriorityAttributeName = 1)
: OldEntity = en1 (PriorityAttributeName = 2)
: Block = Single Server1

```

Sample Entity Replication Message

A Replicate block replicates entity en1 to produce entity en2.

```

%.....%
Replicating Entity (en1)
: Replica 1 of 2 = en2
: Block = Replicate

```

Sample Entity Destruction Message

An Entity Sink block destroys entity en1.

```

%.....%
Destroying Entity (en1)
: Block = Entity Sink

```

Monitoring Block Messages

When a block is about to react to an update in a signal at a monitoring port, the block produces a message in the simulation log.

Sample Scope Message

A Signal Scope block updates its plot.

```

%.....%

```

```
Executing Scope
: Block = Signal Scope
```

Sample Workspace Message

A Discrete-Event Signal to Workspace block receives a new data value. The workspace variable is available only after the debugger session ends.

```
%.....%
Executing Discrete-Event Signal To Workspace
: Block = Discrete Event Signal to Workspace
```

Initialization Messages

Early in the debugging session, the debugger indicates initialization activities.

Message	Description
Initializing Model sedemo_timeout	Appears when the debugging session starts and the model is in the initialization stage.
Initializing Time-Based Entity Generators	Appears after the model initialization, if the model contains at least one Time-Based Entity Generator block. Initializing a Time-Based Entity Generator block means scheduling its first entity generation event.

Signal Operation Messages

Various blocks produce signal operation messages.

Sample Signal Update Detection Message

A sample time hit is detected in the Initial Value block.

```
Detected Sample Time Hit
: Block = Receiver/Initial Value
```

Sample Executing Signal Block Message

A signal operation is being performed in the Gain block, which is a time-based block in the event-based system:

```
Executing Signal Block
      : Block = Gain
```

Execution of Time-Based Blocks in Event-based Systems Block Messages

The debugger detects the execution of time-based blocks in event-based systems.

Message	Description
Executing Signal Block	Appears when the debugger detects the start of execution of such a block.

Identifiers in the Debugger

The simulation log uses tokens to identify blocks, entities, events on the event calendar, and breakpoints uniquely during the debugger session. When requesting information about blocks, entities, or events, or when manipulating breakpoints, you use these identifiers as input arguments.

If you repeat a debugging session in the same version of MATLAB without changing the model structure or parameters, all identifiers of blocks, entities, and events are the same from one session to the next.

This table summarizes the notation for identifiers.

Type of Identifier	Prefix of Identifier	Example
Block	blk	blk1
Entity	en	en2
Event	ev	ev3
Breakpoint	b	b4

In displays of state information in the debugger, the abbreviation ID refers to an identifier.

Note Discrete-event systems do not have identifiers. The debugger refers to discrete-event systems with numeric values, such as Discrete-Event System ID: 0. See for more information on discrete-event systems.

For more information about identifiers, see “Inspecting Entities, Blocks, and Events” on page 13-34 and “Using Breakpoints During Debugging” on page 13-63.

Independent Operations and Consequences in the Debugger

In this section...

“Significance of Independent Operations” on page 13-21

“Independent Operations” on page 13-21

“Consequences of Independent Operations” on page 13-22

Significance of Independent Operations

This section describes a hierarchy of operations that the SimEvents debugger uses when interpreting `step out` or `step over` commands, and that you see in the indentation of the simulation log. Learning these definitions can help you use the `step` function more effectively and gain more insight from the simulation log.

Independent Operations

The simulation clock and the event calendar processor jointly drive the simulation of a SimEvents model, and act as sources of these *independent operations* in the debugger simulation log:

- Initialization of the model or any Time-Based Entity Generator blocks in the model. For more information, see “Initialization Messages” on page 13-18.
- Execution of an event on the event calendar. However, if the application executes an event without scheduling it on the event calendar, the event cannot be the basis of an independent operation. To learn which events are scheduled on the event calendar, see “Role of the Event Calendar” on page 14-10.
- Execution of blocks that are fed time-based signals via time-based to event-based gateway blocks. (Simulation logs do not reflect gateway blocks.)

Other operations that appear in the simulation log are consequences of an independent operation.

In the simulation log, an independent operation is not indented after a line that looks like %====%.

Consequences of Independent Operations

Consequences of independent operations that appear in the simulation log include, but are not limited to, the following:

- Scheduling of an event on the event calendar
- Cancellation of an event on the event calendar
- Detection by a reactive port of a relevant update in an event-based input signal
- Execution of a block whose monitoring port connects to an event-based input signal
- Entity operations, such as:
 - Advancement of an entity from one block to another
 - Queuing of an entity in a queue block
 - Assignment of an attribute to an entity
 - Preemption of an entity in a queue by an arriving entity
 - Replication of an entity
 - Destruction of an entity
- Execution of these events when they are not on the event calendar:
 - Subsystem execution
 - Function call creation
 - Memory read event
 - Memory write event
- Detection by a reactive port of a relevant update in a time-based input signal through a gateway block. You can think of these relevant updates as zero crossings or level crossings. However, if the input signal is an event-based signal or if the input port is not a reactive port, the update is not an independent operation.

- Execution of a block whose monitoring port connects to a time-based input signal through a gateway block.

Consequences are also called dependent operations. In the simulation log, a dependent operation is indented, underneath the independent operation that causes it and after a line that looks like `%. . . %`.

Relationships Among Multiple Consequences

If an independent operation has multiple consequences, they appear in a sequence that reflects the simulation behavior.

Multiple consequences of an independent operation might or might not be causally related to each other. For example, in the following simulation log excerpt, each indented message represents a consequence of the execution of the `ev1` event. Among the indented messages, the `Scheduling ServiceCompletion Event` message is a direct consequence of the preceding message but is not directly related to the message that follows.

```

%=====
Executing EntityGeneration Event (ev1)                Time = 0.1000000000000000
: Entity = <none>                                     Priority = 300
: Block = Time-Based Entity Generator
sedebug>>step over
%. . . . .%
  Generating Entity (en1)
  : Block = Time-Based Entity Generator
%. . . . .%
  Entity Advancing (en1)
  : From = Time-Based Entity Generator
  : To   = Replicate
%. . . . .%
  Replicating Entity (en1)
  : Replica 1 of 2 = en2
  : Block = Replicate
%. . . . .%
  Entity Advancing (en2)
  : From = Replicate
  : To   = Set Attribute
%. . . . .%

```

```

Setting Attribute on Entity (en2)
: RepIndex = 1
: Block = Set Attribute
%.....%
Entity Advancing (en2)
: From = Set Attribute
: To   = Infinite Server
%.....%
Scheduling ServiceCompletion Event (ev2)
: EventTime = 1.1000000000000000
: Priority   = 500
: Entity    = en2
: Block     = Infinite Server
%.....%
Replicating Entity (en1)
: Replica 2 of 2 = en3
: Block = Replicate
%.....%
Entity Advancing (en3)
: From = Replicate
: To   = Infinite Server1
%.....%
Scheduling ServiceCompletion Event (ev3)
: EventTime = 1.1000000000000000
: Priority   = 500
: Entity    = en3
: Block     = Infinite Server1
%.....%
Destroying Entity (en1)
: Block = Replicate
%.....%
Scheduling EntityGeneration Event (ev4)
: EventTime = 0.2000000000000000
: Priority   = 300
: Block     = Time-Based Entity Generator
%=====
Executing EntityGeneration Event (ev4)                Time = 0.2000000000000000
: Entity = <none>                                     Priority = 300
: Block  = Time-Based Entity Generator

```

Stopping the Debugger

In this section...

“How to End the Debugger Session” on page 13-25

“Comparison of Simulation Control Functions” on page 13-25

How to End the Debugger Session

To end the debugger session without completing the simulation, enter one of these commands at the `sedebg>>` prompt:

```
sedb.quit
```

```
quit
```

The simulation ends, the debugging session ends, and the MATLAB command prompt returns.

At the `sedebg>>` prompt, `quit` is equivalent to `sedb.quit`. However, specifying the `sedb` package prevents you from inadvertently ending the MATLAB session if you enter the command at the incorrect command prompt.

Comparison of Simulation Control Functions

The functions in the next table have different behavior and purposes, but any of the functions can cause the debugger session to end.

Function	Behavior with Respect to Ending the Debugger Session	Primary Usage
<code>sedb.quit</code> or <code>quit</code>	Ends the debugger session without completing the simulation.	To end the debugger session immediately without spending time on further simulation log entries, plots, or other simulation behavior.
<code>runtoend</code>	Completes the simulation and then ends the debugger session.	To see the simulation log or plots but not enter commands.

Function	Behavior with Respect to Ending the Debugger Session	Primary Usage
cont	Ends the debugger session only if the simulation is suspended at the built-in breakpoint at the end of the simulation.	Primarily with breakpoints. For more information, see “Using Breakpoints During Debugging” on page 13-63.
step	Ends the debugger session only if the simulation is suspended at the built-in breakpoint at the end of the simulation.	Primarily for proceeding step by step through the simulation. For more information, see “Stepping Through the Simulation” on page 13-27.

Stepping Through the Simulation

In this section...

“Overview of Stepping” on page 13-27

“How to Step” on page 13-28

“Choosing the Granularity of a Step” on page 13-29

“Tips for Stepping Through the Simulation” on page 13-30

Overview of Stepping

Using the SimEvents debugger, you can proceed step by step in the simulation. After each step, you can inspect states or issue other commands at the `sedebug>>` prompt.

When to Step

Stepping is appropriate if one of these is true:

- You want to see what happens next in the simulation and you want frequent opportunities to inspect states as the simulation proceeds.
- You want the simulation to proceed but cannot formulate a condition suitable for a breakpoint.

When Not to Step

Stepping is not the best way to proceed with the simulation in the debugger if one of these is true:

- You want the simulation to proceed until it satisfies a condition that you can formulate using a breakpoint, and you do not need to enter debugging commands until the condition is satisfied. In this case, using breakpoints might require you to enter fewer commands compared to stepping; for details, see “Using Breakpoints During Debugging” on page 13-63.
- You want the simulation to proceed until the end, and you do not need to enter other commands. In this case, at the `sedebug>>` prompt, enter `runtoend` instead of stepping repeatedly.

- You want the simulation to proceed until the end, and you need to enter commands only at the end of the simulation. In this case, remove or disable any breakpoints you might have set earlier, and, at the `sedebug>>` prompt, enter `cont` instead of stepping repeatedly.

How to Step

If you have decided that stepping is appropriate for your debugging needs, at the `sedebug>>` prompt, enter one of the commands in the next table. To learn about the choices for granularity of steps, see “Choosing the Granularity of a Step” on page 13-29.

If Latest Message in Simulation Log Is...	And You Want to...	At <code>sedebug>></code> Prompt, Enter...
An independent operation (not indented)	Take the smallest possible step	<code>step</code> or <code>step in</code>
	Skip consequences of the current operation and stop at the next independent operation that appears in the simulation log	<code>step over</code>
A dependent operation (indented)	Take the smallest possible step	<code>step</code> or <code>step in</code> or <code>step over</code>
	Skip remaining consequences of the previous independent operation and stop at the next independent operation that appears in the simulation log	<code>step out</code>

As a result, the simulation proceeds and the simulation log displays one or more messages to reflect the simulation progress.

For an example, see “Building a Simple Hybrid Model” (“Confirming Event-Based Behavior Using the SimEvents Debugger” section).

Choosing the Granularity of a Step

Using the SimEvents debugger, you can proceed in the simulation by an amount that corresponds to one message in the simulation log, or a collection of messages. The endpoint of a step depends on these factors:

- What is happening in the simulation.

As the section “Interpreting the Simulation Log” on page 13-12 describes, the simulation log does not use a strictly time-based, block-based, or entity-based approach to determine the messages that appear. Similarly, proceeding step by step through a simulation in the SimEvents debugger does not use a strictly time-based, block-based, or entity-based approach to determine the endpoints of the steps.

- The detail settings in effect before you invoke `step`.

If you change the detail settings from their default values to cause the simulation log to omit entity messages or event messages, you cannot step to an operation that corresponds to an omitted message unless a breakpoint coincides with the operation. For instance, see Example: Skipping Entity Operations When Stepping on page 13-50.

In particular, if your detail settings cause the simulation log to omit all messages (equivalent to the `detail none` command), you cannot step to anything other than breakpoints.

- The step size you choose when you invoke `step`. Choices are described, following.

Taking the Smallest Possible Step

The smallest possible step in the SimEvents debugger corresponds to one message in the simulation log. Taking the smallest possible step is appropriate if one of these is true:

- You are not sure what the simulation will skip if you take a larger step, and you want as many opportunities as possible to inspect states as the simulation proceeds. This might be true if you are new to SimEvents software, new to the debugger, unfamiliar with the model you are debugging, or unsure where to look for a simulation problem you are trying to diagnose.

- You know that if you take a larger step, the simulation will skip a point in the simulation at which you want to inspect states.

Taking a Larger Step By Skipping Consequences

A potentially larger step in the SimEvents debugger corresponds to a series message in the simulation log. The last message in the series is not indented, while other messages in the series are indented to show that they represent consequences of an earlier operation. Taking a larger step is appropriate if one of these is true:

- The current operation is not relevant to you and you want to skip over its immediate consequences.
- You find it more efficient to scan a series of messages visually than enter a command interactively after viewing each message, and you do not need to inspect states at intermediate points in the larger step.
- You find it easier to understand a series of related messages when all are visible, and you do not need to inspect states at intermediate points in the larger step.

For Further Information

- “Simulation Log in the Debugger” on page 13-8
- “Customizing the Debugger Simulation Log” on page 13-47
- “Independent Operations and Consequences in the Debugger” on page 13-21

Tips for Stepping Through the Simulation

- **Ensuring the action has happened** — If you want to inspect states to confirm the effect of the action in the most recent message in the simulation log, first ensure that the action has happened. If the message uses an action statement such as “executing,” use `step` before inspecting states. If the message uses a verb in the past tense, such as “detected,” the extra step is not necessary because the action has already happened.
- **Clicking shortcuts** — If you use a certain `step` command frequently, a shortcut you can click might provide an efficient way to issue the command repeatedly. To learn about shortcuts, see “Create MATLAB Shortcuts

to Rerun MATLAB Commands”in the MATLAB Desktop Tools and Development Environment documentation.

- **Connection between step and detail** — If your detail settings cause the simulation log to omit all messages (equivalent to the `detail none` command), you cannot step to anything other than breakpoints. In the absence of breakpoints, a step causes the simulation to proceed until the end. If you inadvertently reach the end of the simulation in this way and want to return to the point in the simulation from which you tried to step, use information in the Command Window to set a breakpoint in a subsequent debugging session. For example, if the last message in the simulation log before you inadvertently stepped too far indicates the execution of event `ev5`, you can enter `evbreak ev5; cont` in the next debugger session.

Inspecting the Current Point in the Debugger

In this section...

“Viewing the Current Operation” on page 13-32

“Obtaining Information Associated with the Current Operation” on page 13-32

Viewing the Current Operation

The simulation log displays information about what is about to happen or what has just happened in the simulation. If the log is no longer visible in the Command Window because of subsequent commands and output displays, you can redisplay the most recent log entry by entering this command at the `sedebug>>` prompt:

```
currentop
```

If the most recent log entry represents a dependent operation, the output in the Command Window also includes the current top-level independent operation being executed. To learn more about dependent and independent operations, see “Independent Operations and Consequences in the Debugger” on page 13-21

Obtaining Information Associated with the Current Operation

You can get some information about the current operation in the form of variables in the workspace using commands like those listed in the next table. Variables containing identifiers can be useful as inputs to state inspection functions. For details, see “Inspecting Entities, Blocks, and Events” on page 13-34.

Information	At <code>sedebug>></code> Prompt, Enter...
Current simulation time	<code>t = simtime</code>
Identifier of the entity that undergoes the current operation	<code>enid = gcen</code>

Information	At sedebug>> Prompt, Enter...
Identifier of the block associated with the current operation	blkid = gcebid
Path name of the block associated with the current operation	blkname = gceb
Identifier of the event being scheduled, executed, or canceled on the event calendar as part of the current operation	evid = gcev

Inspecting Entities, Blocks, and Events

In this section...

“Inspecting Entities” on page 13-34

“Inspecting Blocks” on page 13-36

“Inspecting Events” on page 13-38

“Obtaining Identifiers of Entities, Blocks, and Events” on page 13-38

Inspecting Entities

These sections provide procedures and background information about inspecting entities:

- “Inspecting Location, Scalar Attributes, Timeouts, and Timers” on page 13-34
- “Inspecting Nonscalar Attribute Values” on page 13-35
- “Interpretation of Entity Location” on page 13-36

For an example, see the `sedb.eninfo` reference page.

Inspecting Location, Scalar Attributes, Timeouts, and Timers

If you expect all attributes of an entity to have scalar values or if knowing the sizes of nonscalar attribute values is sufficient, then this procedure is the simplest way to inspect the entity:

- 1 Find the entity identifier using the simulation log or one of the approaches listed in Obtaining Entity Identifiers on page 13-39.
- 2 At the `sedebg>>` prompt, enter this command, where `enid` is the entity identifier:

```
eninfo enid % enid is the identifier.
```

The resulting display includes this information:

- Current simulation time

- Location of the entity
- Names of attributes of the entity
- Scalar attribute values and sizes of nonscalar attribute values
- Tags, scheduled times, and event identifiers of timeouts
- Tags and elapsed times of timers

Inspecting Nonscalar Attribute Values

To inspect nonscalar values of attributes of an entity:

- 1** Find the entity identifier using the simulation log or one of the functions listed in Obtaining Entity Identifiers on page 13-39.
- 2** At the `sedebug>>` prompt, enter this command. `enid` is a string representing the entity identifier and `en_struct` is the name of a variable you want to create in the workspace:

```
en_struct = eninfo(enid); % enid is the identifier.
```

The output variable `en_struct` is a structure that stores this information:

- Current simulation time
 - Location of the entity
 - Names and values of the attributes of the entity
 - Tags, scheduled times, and event identifiers of timeouts
 - Tags and elapsed times of timers
- 3** Use dot notation to access values of any attribute as part of the `Attributes` field of `en_struct`.

For example, to access the value of an attribute called `Attribute1`, use the notation `en_struct.Attributes.Attribute1`. The field name `Attributes` is fixed but the names `en_struct` and `Attribute1` depend on names you choose for the variable and attribute.

Interpretation of Entity Location

Blocks that possess entity input ports act as storage or nonstorage blocks. Only storage blocks are capable of holding an entity for a nonzero duration. If the debugger reports a nonstorage block as the location of an entity, it means that the debugger has suspended the simulation while the entity is in the process of advancing through a nonstorage block toward a storage block or a block that destroys entities. Before the simulation clock moves ahead, the entity will either arrive at a storage block or be destroyed.

For lists of storage and nonstorage blocks, see “Storage and Nonstorage Blocks” on page 14-51.

To inspect blocks instead of entities, see “Inspecting Blocks” on page 13-36.

Inspecting Blocks

- “Procedure for Inspecting Blocks” on page 13-36
- “Result of Inspecting Queue Blocks” on page 13-37
- “Result of Inspecting Server Blocks” on page 13-37
- “Result of Inspecting Other Storage Blocks” on page 13-37
- “Result of Inspecting Nonstorage Blocks” on page 13-38

Procedure for Inspecting Blocks

Before inspecting a block, determine if it is capable of providing information. One way to do this is to see whether the block appears in the output of `blklist`. If the block is capable of providing information, use this procedure to get the information:

- 1 Find a unique way to refer to the block using one of these approaches:
 - Find the block identifier using one of the approaches listed in Obtaining Block Identifiers on page 13-39.
 - Find the path name of the block by selecting the block and using `gcb`, or by typing the path name. If you type the path name, be careful to reflect space characters and line breaks accurately.

- 2 At the `sedebug>>` prompt, enter this command , where `thisblk` is a string representing the block identifier or path name:

```
blkinfo(thisblk); % thisblk is the identifier or path name.
```

The resulting information depends on the kind of block you are inspecting.

Result of Inspecting Queue Blocks

When you inspect queue blocks, the display includes this information:

- Current simulation time
- Identifiers of entities that the block is currently storing
- Number of entities that the block can store at a time
- Status of each stored entity with respect to the block
- Length of time each stored entity has been in the queue

Result of Inspecting Server Blocks

When you inspect server blocks, the display includes this information:

- Current simulation time
- Identifiers of entities that the block is currently storing
- Number of entities that the block can store at a time
- Status of each stored entity with respect to the block
- Identifier and scheduled time of the service completion event for each entity

Result of Inspecting Other Storage Blocks

When you inspect storage blocks other than queues and servers, the display includes this information:

- Current simulation time
- Identifiers of entities that the block is currently storing

For the Output Switch block with the **Store entity before switching** option selected, the resulting display also indicates which entity output port is the selected port.

For a list of storage blocks, see “Storage and Nonstorage Blocks” on page 14-51.

Result of Inspecting Nonstorage Blocks

When you inspect nonstorage blocks, the display includes this information:

- Current simulation time
- Identifier of an entity that is currently advancing through the block. To learn what it means for a nonstorage block to be the location of an entity, see “Interpretation of Entity Location” on page 13-36.

Depending on the block, the display might also include additional information. For details, see the Block and Description columns of the Fields of Output Structure table on the `sedb.blkinfo` reference page.

Inspecting Events

To get details about a particular event on the event calendar:

- 1 Find the event identifier using the simulation log or one of the approaches listed in Obtaining Event Identifiers on page 13-40.
- 2 At the `sedebug>>` prompt, enter this command, where `evid` is the event identifier:

```
evinfo evid % evid is the identifier.
```

Alternatively, at the `sedebug>>` prompt, enter `evcal` to get details about all events on the event calendar. For more information, see the `sedb.evcal` reference page.

Obtaining Identifiers of Entities, Blocks, and Events

In some state inspection functions, you must refer to an entity, block, or event using its identifier. For background information about identifiers, see

“Identifiers in the Debugger” on page 13-19. The next tables suggest ways to obtain identifiers to use as input arguments in state inspection commands.

Obtaining Entity Identifiers

To Display Identifier of Entity Associated with...	At <code>sedebug>></code> Prompt, Enter...	Link to Reference Page
The current operation	<code>gcen</code>	<code>sedb.gcen</code>
A particular block whose identifier or path name you know	<code>blkinfo(...)</code> . Look in the ID column in the resulting tabular display.	<code>sedb.blkinfo</code>
Events on the event calendar	<code>evcal</code> . Look in the Entity column in the resulting tabular display.	<code>sedb.evcal</code>
A particular event whose identifier you know	<code>evinfo(...)</code> . Look at the Entity entry.	<code>sedb.evinfo</code>

Obtaining Block Identifiers

To Display Identifier of Block Associated with...	At <code>sedebug>></code> Prompt, Enter...	Link to Reference Page
The current operation	<code>gcebid</code>	<code>sedb.gcebid</code>
All blocks whose states you can inspect	<code>blklist</code> . Look in the first column in the resulting tabular display.	<code>sedb.blklist</code>

Obtaining Event Identifiers

To Display Identifier of Event Associated with...	At sedebug>> Prompt, Enter...	Link to Reference Page
The current operation	<code>gcev</code>	<code>sedb.gcev</code>
Events on the event calendar	<code>evcal</code> . Look in the ID column in the resulting tabular display.	<code>sedb.evc1</code>

Working with Debugging Information in Variables

In this section...

“Comparison of Variables with Inspection Displays” on page 13-41

“Functions That Return Debugging Information in Variables” on page 13-41

“How to Create Variables Using State Inspection Functions” on page 13-42

“Tips for Manipulating Structures and Cell Arrays” on page 13-43

“Example: Finding the Number of Entities in Busy Servers” on page 13-43

Comparison of Variables with Inspection Displays

State inspection functions in the SimEvents debugger enable you to view information in the Command Window, as the sections “Inspecting Entities, Blocks, and Events” on page 13-34 and “Viewing the Event Calendar” on page 13-46 describe. An alternative way to capture the same information is in a variable in the MATLAB base workspace. Capturing information in a workspace variable lets you accomplish these goals:

- Use information from one command as an input to another command, without having to type or paste information from the Command Window.
- Cache information at a certain point in the simulation, for comparison with updated information at a later point in the simulation. The workspace variable remains fixed as the simulation proceeds.
- Store information in a MAT-file and use it in a different MATLAB software session.

Functions That Return Debugging Information in Variables

The table lists state inspection functions that can return variables in the workspace.

Function	Returns	Class
<code>blkinfo</code>	Block information	Structure
<code>blklist</code>	Blocks and their identifiers	Cell

Function	Returns	Class
<code>eninfo</code>	Entity information	Structure
<code>evcal</code>	Event calendar	Structure
<code>evinfo</code>	Event information	Structure
<code>gceb</code>	Path name of the block associated with the current operation	String (char)
<code>gcebid</code>	Identifier of the block associated with the current operation	String (char)
<code>gcen</code>	Identifier of the entity that undergoes the current operation	String (char)
<code>gcev</code>	Identifier of the event associated with the current operation	String (char)
<code>simtime</code>	Current simulation time	Numeric (double)

How to Create Variables Using State Inspection Functions

To create a variable using a state inspection function, follow these rules:

- Name an output variable in the syntax that you use to invoke the function, such as `my_entity_id = gcen`. The function creates the output variable in the workspace.
- If the function requires input arguments, express them using the functional form of the syntax, such as `evinfo('ev1')`, rather than the command form, `evinfo ev1`.

To learn about functional and command forms of syntax, see “Command vs. Function Syntax” in the MATLAB Programming Fundamentals documentation.

To learn how to formulate input arguments for `blkinfo`, `eninfo`, and `evinfo`, see “Obtaining Identifiers of Entities, Blocks, and Events” on page 13-38.

For details about the information each function returns in the output variable, see the corresponding function reference page.

Tips for Manipulating Structures and Cell Arrays

Full details about structures and cell arrays are in “Structures” and “Cell Arrays”, both in the MATLAB Programming Fundamentals documentation. Tips that you might find useful for working with the variables that debugger functions return are:

- If a cell array shows something like [1x51 char] instead of exact block path names when you enter the variable name alone at the command prompt, use content indexing to display the cell contents explicitly. Content indexing uses curly braces, {}. For an example, see the `sedb.blklist` reference page.
- To access information in nested structures, append nested field names using dot notation. For an example, see the `sedb.eninfo` reference page.
- You can assign numeric values of like-named fields in a structure array to a numeric vector. To do this, enclose the `array.field` expression in square brackets, []. For an example, see the `sedb.evcal` reference page.
- You can assign string values of like-named fields in a structure array to a cell array. To do this, enclose the `array.field` expression in curly braces, {}. For examples, see the `sedb.blkinfo` and `sedb.breakpoints` reference pages.
- You can gather information about like-named fields in a structure array that satisfy certain criteria, by invoking `find` and using its output to index into the structure array. For examples, see the `sedb.evcal` and `sedb.breakpoints` reference pages.

Example: Finding the Number of Entities in Busy Servers

This example illustrates ways that you can use information from one command as an input to another command. Suppose your system includes several server blocks and you want to see how many entities are in each server block that is currently busy serving an entity. The following code inspects the event calendar to locate service completion events, uses the events to locate server blocks that are currently busy, and inspects server blocks to find out

how many entities are in them. An entity in the server might be in service or waiting to depart.

- 1** Begin a debugger session for a particular model by entering this command at the MATLAB command prompt:

```
sedebg('sedemo_star_routing')
```

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

```
tbreak 5  
cont
```

The output ends with a message describing the context of the simulation shortly after $T = 5$:

```
Hit b1 : Breakpoint for first operation at or after time 5.000000  
  
%===== %  
Executing ServiceCompletion Event (ev29)           Time = 5.189503930558476  
: Entity = en4                                     Priority = 5  
: Block = Distribution Center/Infinite Server
```

- 3** To find out how many entities are in each server that is currently busy serving, use a series of state inspection and variable manipulation commands:

```
% Get the event calendar.  
eventcalendar = evcal;  
  
% Combine executing and pending events, to search both.  
allevents = [eventcalendar.ExecutingEvent; eventcalendar.PendingEvents];  
  
% Find service completion events.  
idx = cellfun(@(x) isequal(x,'ServiceCompletion'), {allevents.EventType});  
svc_completions = allevents(idx);  
  
% Find the unique server blocks.  
svrs = unique({svc_completions.Block});  
  
% Compute the number of server blocks.
```

```
num = length(svrs);
% Preallocate an array for the results.
n = zeros(1,num);

% Loop over the server blocks and find the number of entities
% in each block.
for jj=1:num
    s = blkinfo(svrs{jj});
    n(jj)=length(s.Entities);
    disp(sprintf('%s: %d',svrs{jj},n(jj)))
end
```

The output is:

```
sedemo_star_routing/Distribution Center/Infinite Server: 1
sedemo_star_routing/Service Station 3/Infinite Server3: 1
sedemo_star_routing/Service Station 4/Infinite Server4: 2
```

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

```
sedb.quit
```

Viewing the Event Calendar

To view a list of events on the event calendar of the currently executing discrete-event system, at the `sedebug>>` prompt, enter this command:

```
evcal
```

The output in the Command Window includes the current simulation time and a tabular display of all events in the event calendar. Each listing in the event calendar display includes the event characteristics described in “Optional Displays of Event Calendar in the Simulation Log” on page 13-10.

Each discrete-event system has its own event calendar. See for more information on discrete-event systems. This means that one SimEvents model can have multiple event calendars.

The event in progress or selected for execution appears with notation `=>` to the left of the event identifier. This event appears in the display until the completion of all immediate consequences, and then the event is removed from the calendar.

When comparing the simulation log with the event calendar display, the sequence on the calendar and the sequence in which the events were scheduled might differ, even for simultaneous events having equal priority.

For Further Information

- “Event Sequencing” on page 14-9
- “Example: Event Calendar Usage for a Queue-Server Model” on page 2-7

Customizing the Debugger Simulation Log

In this section...

“Customizable Information in the Simulation Log” on page 13-47

“Tips for Choosing Appropriate Detail Settings” on page 13-48

“Effect of Detail Settings on Stepping” on page 13-50

“How to View Current Detail Settings” on page 13-52

“How to Change Detail Settings” on page 13-52

“How to Save and Restore Detail Settings” on page 13-53

Customizable Information in the Simulation Log

You can configure the SimEvents debugger to include or omit certain kinds of messages in its simulation log. You can focus on the messages that are relevant to you by omitting irrelevant messages.

The overall configuration regarding including or omitting messages is called the debugger’s *detail settings*. Each individual category of messages that you can include or omit corresponds to an individual detail setting. A particular detail setting is on if the simulation log includes messages in the corresponding category. The setting is off if the simulation log omits messages in the category.

The `detail` function lets you configure and view detail settings. The following table lists the available detail settings and the programmatic names that you see as inputs or field names when you use the `detail` function.

Category of Messages	Programmatic Name in detail Function	Further Information About Messages in Category
Event operations, except independent operations representing event executions	ev	“Event Scheduling, Execution, and Cancelation Messages” on page 13-13
Entity operations	en	“Entity Operation Messages” on page 13-16
Event calendar information	cal	“Optional Displays of Event Calendar in the Simulation Log” on page 13-10
Signal operations	sig	“Signal Operation Messages” on page 13-18

Messages that you cannot selectively omit from the simulation log include messages about independent operations, such as the execution of events on the event calendar. Messages about independent operations appear whenever any of the detail settings is on. The reason for including messages about independent operations is that they provide a context in which you can understand other messages. To learn more about independent operations, see “Independent Operations and Consequences in the Debugger” on page 13-21.

Tips for Choosing Appropriate Detail Settings

Different debugging scenarios benefit from different detail settings. Use these suggestions to help you choose appropriate settings for your needs:

- When using `step` to proceed in the simulation, include event or entity operations in the simulation log if you want the debugger to suspend the simulation at such operations to let you inspect states.
- When using breakpoints to skip a large portion of the simulation that is irrelevant to you, omitting all messages (`detail none`) can prevent the

Command Window from accumulating a lot of simulation log text that does not interest you.

Tip If you use `step` from a breakpoint onward, turn on at least one of the detail settings, or else you will likely step too far. For details, see “Effect of Detail Settings on Stepping” on page 13-50.

- If you want to inspect final states but are not interested in the simulation log, omit all messages (`detail none`) and then enter `cont`. The simulation proceeds until the built-in breakpoint at the end of the simulation, with minimal text in the simulation log.
- If you want to focus on entities instead of events, try `detail('en',1,'ev',0,'sig',0)`. The simulation log still shows independent operations, but the dependent operations in the simulation log exclude event scheduling, cancelation, and execution operations.
- If you want to focus on events instead of entities, try `detail('en',0,'ev',1,'sig',0)` or `detail('en',0,'ev',1,'sig',0,'cal',1)`.
- If you want to focus on signals instead, try `detail('en',0,'ev',0,'sig',1)` or `detail('en',0,'ev',0,'sig',1,'cal',1)`.
- If you are not sure which messages in the simulation log might be useful for later reference, consider these approaches:
 - If you include less information in the simulation log, the simulation might run more quickly and the simulation log might be more manageable. However, you risk missing important information and having to run the simulation again to see that information.
 - If you include more information in the simulation log, it might be harder for you to find the relevant portions. Using **Edit > Find** in the Command Window might help you see the portions of interest to you.
- Reducing textual output in the Command Window can save time.

Effect of Detail Settings on Stepping

The behavior of `detail` and `step` are related. If you change the detail settings from their default values to cause the simulation log to omit entity messages or event messages, then you cannot step to an operation that corresponds to an omitted message, unless a breakpoint coincides with the operation. A specific example of this behavior follows.

If your detail settings cause the simulation log to omit all messages (equivalent to the `detail none` command), you cannot step to anything other than breakpoints. In the absence of breakpoints, a step causes the simulation to proceed until the end. If you inadvertently reach the end of the simulation in this way and want to return to the point in the simulation from which you tried to step, use a time or event identifier that appears in the Command Window to set a breakpoint in a subsequent debugging session. For example, if the last message in the simulation log, before you inadvertently stepped too far, indicates the execution of event `ev5`, then you can exit the debugger session, restart it, and enter `evbreak ev5; cont`.

Example: Skipping Entity Operations When Stepping

When the detail setting for entity operations is off, the simulation log omits messages about entity operations and you cannot step to entity operations.

- 1 Open and debug a model. At the MATLAB command prompt, enter:

```
simeventsdocex('doc_slldemo_f14_des')
sedebug('doc_slldemo_f14_des')
```

- 2 Omit entity operation messages. At the `sedebug>>` prompt, enter:

```
detail('en',0)
```

- 3 Proceed with the simulation. At the `sedebug>>` prompt, enter the following command six times in succession:

```
step
```

The output reflects that when you step through the simulation, the debugger does *not* suspend the simulation upon the entity's generation, advancement, attribute assignment, or destruction. The reason is that the detail setting for entity operations is off.

```

Detected Sample Time Hit                               Time = 0.0000000000000000
: Block = Subsystem/Event-Based Entity Generator
sdebug>> step
%.....%
Scheduling EntityGeneration Event (ev1)
: EventTime = 0.0000000000000000 (Now)
: Priority   = SYS1
: Entity    = <none>
: Block    = Subsystem/Event-Based Entity Generator
%=====
Executing EntityGeneration Event (ev1)                 Time = 0.0000000000000000
: Entity    = <none>                                   Priority = SYS1
: Block    = Subsystem/Event-Based Entity Generator
sdebug>> step
%.....%
Scheduling ServiceCompletion Event (ev2)
: EventTime = 0.055383948677907
: Priority   = 500
: Entity    = en1
: Block    = Subsystem/Infinite Server
%=====
Executing ServiceCompletion Event (ev2)               Time = 0.055383948677907
: Entity    = en1                                     Priority = 500
: Block    = Subsystem/Infinite Server
%=====
Detected Sample Time Hit                               Time = 0.1000000000000000
: Block = Subsystem/Event-Based Entity Generator

```

To see which entity operations the debugger omits, compare your output with the items in the table in “Confirming Event-Based Behavior Using the SimEvents Debugger” in the SimEvents getting started documentation. Items 4, 5, 6, 7, 10, 11, and 12 in the table do not appear in this example because these items represent entity operations.

- 4** End the debugger session. At the `sdebug>>` prompt, enter:

```
sedb.quit
```

How to View Current Detail Settings

To view the current detail settings in the Command Window, at the `sedebg>>` prompt, enter:

```
detail
```

The output looks like this, where the `on` and `off` values depend on your current detail settings:

```
Event Operations (ev) : on
Entity Operations (en) : off
Signal Operations (sig) : on
Event Calendar (cal) : off
```

If a line in the output says `on`, the simulation log shows the corresponding type of message. Otherwise, the simulation log omits the corresponding type of message.

How to Change Detail Settings

To change detail settings, enter a `detail` command that includes one or more inputs. For available syntaxes, see the reference page for the `sedb.detail` function.

Tips to help you select a suitable syntax are:

- To change one or more detail settings but not all detail settings, use inputs that specify the name and new value of the detail settings you want to change. Detail settings you omit from the syntax remain unchanged.
- To change all detail settings, use inputs that specify the name and new value of all detail settings. One way to ensure that you include all settings in the command is to use a structure variable containing the current settings as a starting point. For an example, see [Example: Including Event Calendar Information Using a Structure](#) on page 13-53.
- To make it easier to restore previous settings later, use a syntax that includes an output. For details, see “[How to Save and Restore Detail Settings](#)” on page 13-53.

- To learn which messages in the simulation log correspond to each detail setting, see “Customizable Information in the Simulation Log” on page 13-47.
- For suggestions on choosing which messages to include or omit based on your debugging situation, see “Tips for Choosing Appropriate Detail Settings” on page 13-48.

Example: Including Event Calendar Information Using a Parameter/Value Pair

To cause the simulation log to include event calendar information but not change any other detail settings, at the `sedebug>>` prompt, enter:

```
detail('cal',1)
```

The output includes the following line confirming the change:

```
Event Calendar    (cal)  :  on
```

Example: Including Event Calendar Information Using a Structure

Entering these commands at the `sedebug>>` prompt has the same effect as the example above and also creates a structure variable, `s`, that records the new detail settings:

```
s = detail; % Get current detail settings.
s.cal = 1; % Change value of cal field of structure s.
detail(s); % Use s to change cal detail setting.
```

How to Save and Restore Detail Settings

If you expect to change detail settings frequently or temporarily during a debugger session, you can use an output from the `detail` function to facilitate restoring previous settings. Use this procedure:

- 1** When changing detail settings, enter a `detail` command that includes an output. The output variable records the settings *before* they change to correspond to the inputs that you specify in your command.
- 2** When you want to restore the earlier detail settings, use the variable as an input to `detail`.

As a special case, you can restore default detail settings. At the `sedebug>>` prompt, enter `detail default`.

Example: Omitting and Reinstating Entity Messages

To cause the simulation log to include event calendar information and also create a structure variable, `prev`, that records the previous detail settings, at the `sedebug>>` prompt, enter:

```
prev = detail('cal',1) % Record settings and then change them.
```

The next command restores the earlier settings:

```
detail(prev) % Restore previous settings.
```


Debugger Efficiency Tips

In this section...

“Executing Commands Automatically When the Debugger Starts” on page 13-55

“Creating Shortcuts for Debugger Commands” on page 13-56

Executing Commands Automatically When the Debugger Starts

If you want to execute one or more commands in the debugger immediately after initializing the model, you can include those commands when you invoke `sedebug`. You might find executing commands automatically to be useful for setting the same breakpoints or detail settings across multiple debugging sessions, or helping someone reproduce a problem that you are seeing in a model.

To start a debugger session that executes commands automatically:

- 1 Outside the debugger, create an empty options structure using the `se_getdbopts` function.

```
opts_struct = se_getdbopts;
```

- 2 Define the `StartFcn` field of the options structure as a cell array of strings, where each string is an individual command that you want to execute after initializing the model. Here is an example:

```
opts_struct.StartFcn = {'detail('cal',1)', 'tbreak 5'};
```

- 3 Start a debugger session using the `sedebug` function with the options structure as the second input argument. Here is an example:

```
sedebug('sedemo_preempt_policy', opts_struct)
```

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

```
sedb.quit
```

For an example, see the `se_getdbopts` reference page.

Tips for Creating a StartFcn Cell Array

- If you want to execute commands and then exit the debugging session automatically, include the string 'quit' at the end of the StartFcn array. If you want to execute commands automatically and then interact with the debugger, do not include the string 'quit' in the StartFcn array.
- If you want to set breakpoints at the beginning of the debugging session, have just ended a debugging session on the same model and have not changed the model, you can use the identifiers that occurred in the previous debugging session.

Creating Shortcuts for Debugger Commands

If you use a particular debugger command frequently, such as `step` or `step over`, a shortcut you can click might provide an efficient way to issue the command repeatedly. To learn about shortcuts, see “Create MATLAB Shortcuts to Rerun MATLAB Commands” in the MATLAB Desktop Tools and Development Environment documentation.

Defining a Breakpoint

In this section...

“What Is a Breakpoint?” on page 13-57

“Identifying a Point of Interest” on page 13-57

“Setting a Breakpoint” on page 13-59

“Viewing All Breakpoints” on page 13-61

What Is a Breakpoint?

In the SimEvents debugger, a breakpoint is a point of interest in the simulation at which the debugger can suspend the simulation and let you enter commands. You decide which points are of interest to you and then use debugger functions to designate those points as debugging breakpoints. After you define one or more breakpoints, you can use them to control the simulation process efficiently. At the `sedebug>>` prompt, the `cont` command causes the simulation to proceed until the next breakpoint, bypassing points that you are not interested in and letting you inspect states at a point of interest. To learn more about controlling the simulation after defining breakpoints, see “Using Breakpoints During Debugging” on page 13-63.

Identifying a Point of Interest

Before defining a breakpoint, you must decide what points in the simulation you want to inspect and then determine a way to refer to the point explicitly when invoking a function to define a breakpoint. The SimEvents debugger supports the following kinds of breakpoints.

Type of Breakpoint	Debugger Suspends Simulation Upon...	When You Might Use Breakpoint Type
Timed breakpoint	First operation whose associated time is equal to or greater than the value of the timed breakpoint	<ul style="list-style-type: none"> • You know a time at which something of interest to you occurs • You want to proceed in the simulation by a fixed amount of time • A point of interest is not associated with an event on the event calendar or a block in the model • You do not have an event identifier to use to define an event breakpoint
Event breakpoint	Execution or cancelation of the specified event	An event on the event calendar is a point of interest
Block breakpoint	Operation involving the specified block, for blocks that support block breakpoints	<ul style="list-style-type: none"> • You want to understand how a block behaves • A block seems to cause or reflect the problem you are investigating
Entity breakpoint	Operation that involves the specified entity	An operation that involves the entity is a point of interest

You cannot set a breakpoint for a Simulink controlled block. Instead, set the breakpoint in the SimEvents block that initiates the signal execution.

Tips for Identifying Points of Interest

- To see a list of all events on the event calendar and their event identifiers, at the `sedebg>>` prompt, enter `evcal`.
- To see a list of blocks in the model that support block breakpoints, at the `sedebg>>` prompt, enter `blklist`. The list also shows the block identifiers. (Note, the `blklist` output also contains the list of time-based blocks in event-based systems. These blocks do not support `blkbreak`.) For a list of block operations at which the debugger can suspend the simulation, see “Block Operations Relevant for Block Breakpoints” on page 13-67.

- To proceed in the simulation by a fixed amount of time, define a timed breakpoint whose value is relative to the current simulation time using syntax such as `tbreak(simtime + fixed_amount)`. For an example, see the `sedb.simtime` reference page.

To see a list of all entities and their IDs in a storage block, use `blkinfo`. Use these IDs to set entity breakpoints.

- To investigate the behavior of a block only during particular time intervals, use a combination of timed breakpoints and block breakpoints. During a time interval of interest, the block breakpoint helps you investigate the behavior of the block. When the simulation advances beyond that time interval, you can disable the block breakpoint and use a timed breakpoint to advance to another time interval of interest. To learn more about disabling breakpoints, see “Ignoring or Removing Breakpoints” on page 13-64.
- To see all actions that happen at a particular time, use a pair of timed breakpoints, as in “Using Nearby Breakpoints to Focus on a Particular Time” on page 3-5.
- You might need or want to iteratively refine your points of interest across multiple simulation runs. For example:
 - A plot of a signal against time might indicate when something of interest to you happens in the simulation. You can read the approximate time from the plot. You can use the approximate time when defining a timed breakpoint in a subsequent run of the simulation.
 - You can use a pair of timed breakpoints to examine simulation behavior in a time interval and find a relevant event on the event calendar. You can use the event identifier when defining an event breakpoint in a subsequent run of the simulation.
- An event breakpoint is not the same as a timed breakpoint whose value equals the scheduled time of the event. The two breakpoints can cause the simulation to stop at different points if the execution or cancelation of the event is not the first thing that happens at that value of time. For an example, see the `sedb.evbreak` reference page.

Setting a Breakpoint

After you have identified a point of interest, you can set a breakpoint by entering one of the commands in the table.

Type of Breakpoint	At <code>sedebug>></code> Prompt, Enter...
Timed breakpoint at simulation time $T = t_0$	<code>tbreak(t0)</code> or <code>tbreak t0</code>
Event breakpoint at event whose identifier is the string, <code>evid</code>	<code>evbreak(evid)</code>
Block breakpoint at block whose identifier is the string, <code>blkid</code> , or whose path name is the string, <code>blkname</code>	<code>blkbreak(blkid)</code> or <code>blkbreak(blkname)</code>
Entity breakpoint at entity whose identifier is the string, <code>enid</code>	<code>enbreak(enid)</code>

Warning When Setting Certain Breakpoints

The debugger warns you if it determines that it might not hit the breakpoint that you want to define or if it does not recognize an event identifier that you specify. The warning can alert you to a mistake in your command, but might also follow a correct command. For example, suppose you obtain an event identifier during one run of the simulation and set an event breakpoint on that event in a subsequent run of the simulation, *before* the event has been scheduled. Setting an event breakpoint before the event has been scheduled is legitimate because event identifiers are the same from one debugging session to the next. However, the debugger cannot distinguish this situation from a mistake in your input argument to `evbreak`.

If you often intentionally set breakpoints that cause this warning and you want to suppress such warnings in the future, enter `warning off last` immediately after the warning occurs. For more information about

this command, see “Warning Control” in the MATLAB Programming Fundamentals documentation.

Viewing All Breakpoints

To see a tabular display of all breakpoints that you have set, at the `sedebug>>` prompt, enter this command:

```
breakpoints
```

The output includes this information about each breakpoint.

Label	Description
ID	A token that uniquely identifies the breakpoint
Type	Block, Event, Timed, or Entity
Value	The block identifier of a block breakpoint
	The event identifier of an event breakpoint
	The time of a timed breakpoint
	The ID of the entry
Enabled	yes, if the debugger considers the breakpoint when determining where to suspend the simulation
	no, if the debugger ignores the breakpoint

The list of breakpoints does not guarantee that the simulation reaches each point before the simulation ends. The sequence of breakpoints in the list does not necessarily represent the sequence in which the simulation reaches each point.

The list of breakpoints does not show a special built-in breakpoint that the debugger always observes at the end of the simulation. You do not set this breakpoint explicitly and you cannot disable or remove it.

Sample Breakpoint List

The sample output of breakpoints shows six breakpoints. Two are timed breakpoints, one is an entity, one is an event breakpoint, and two are block breakpoints. One of the block breakpoints is disabled.

List of Breakpoints:

ID	Type	Value	Enabled
b1	Event	ev4	yes
b2	Block	blk11	yes
b3	Timed	100	yes
b4	Timed	101	yes
b5	Block	blk15	no
b6	Entity	en3	yes

To learn how to delete or disable breakpoints in the list, see “Ignoring or Removing Breakpoints” on page 13-64.

To learn how to enable breakpoints in the list, see “Enabling a Disabled Breakpoint” on page 13-65.

Using Breakpoints During Debugging

In this section...

“Running the Simulation Until the Next Breakpoint” on page 13-63

“Ignoring or Removing Breakpoints” on page 13-64

“Enabling a Disabled Breakpoint” on page 13-65

Running the Simulation Until the Next Breakpoint

By default, the debugger has a special built-in breakpoint at the end of the simulation. You can define your own breakpoints, as described in “Defining a Breakpoint” on page 13-57. To proceed in the simulation until the debugger reaches the next breakpoint, at the `sedebug>>` prompt, enter this command:

```
cont
```

You cannot set breakpoints for Simulink controlled block. Instead, set the breakpoint in the SimEvents block that initiates the signal execution.

Point at Which the Debugger Suspends the Simulation

When you enter a `cont` command, the debugger proceeds in the simulation until it reaches the first point in the simulation that meets one of these criteria:

- **At or after specified time** — The simulation time is equal to or greater than the specified time of a timed breakpoint, and the point in the simulation corresponds to an operation that the simulation log is able to show.

If no event executions or relevant updates in signals at reactive ports occur at the specified time of a timed breakpoint, the debugger reaches that breakpoint when the simulation time is strictly later. For example, if time-based blocks in a hybrid simulation have a discrete sample time of 1 and running the simulation without breakpoints causes the simulation log to report operations only at $T = 0, 2, 4, \dots$, then a timed breakpoint at $T = 3$ is equivalent to a timed breakpoint at $T = 4$.

- **At execution or cancellation** — The simulation is about to execute or cancel the specified event of an event breakpoint.
- **At operation of a block** — The block associated with a block breakpoint is about to perform an operation that the simulation log is able to show. For a list of block operations at which the debugger can suspend the simulation, see “Block Operations Relevant for Block Breakpoints” on page 13-67.
- **At operation on an entity** — The block associated with a entity breakpoint is about to perform an operation that the simulation log is able to show. For a list of entity operations at which the debugger can suspend the simulation, see “Block Operations Relevant for Block Breakpoints” on page 13-67.
- **At end** — The simulation is about to end. This condition corresponds to the built-in breakpoint at the end of the simulation.

The debugger reaches a given timed or event breakpoint zero or one time during the simulation. The debugger can reach a given block or entity breakpoint an arbitrary number of times during the simulation.

Unless all breakpoints are timed breakpoints, you might not be able to predict which breakpoint the debugger reaches next. Even though events have scheduled times, the debugger might reach an event breakpoint upon the cancelation of an event. You might not be able to predict the cancelation.

Ignoring or Removing Breakpoints

The table describes options for preventing the debugger from observing a particular breakpoint.

Treatment of Breakpoints	At <code>sedebug>></code> Prompt, Enter...	Result
Ignore a particular breakpoint while keeping it in the list of breakpoints and being able to reinstate it easily	<code>disable b1</code> , where <code>b1</code> is the breakpoint identifier	The list of breakpoints indicates the breakpoint as disabled and the debugger does not observe the breakpoint. You can reverse this operation using <code>enable b1</code> .
Ignore a particular breakpoint without keeping it in the list of breakpoints and without being able to reinstate it easily	<code>bdelete b1</code> , where <code>b1</code> is the breakpoint identifier	The breakpoint no longer appears in the list of breakpoints, so the debugger does not observe it.
Ignore all timed and event breakpoints <i>and</i> run the simulation until the end	<code>runtoend</code>	The simulation runs to completion and the debugger session ends.

To view breakpoint identifiers, at the `sedebug>>` prompt, enter `breakpoints`.

Tip You can apply `disable`, `enable`, or `bdelete` to multiple breakpoints in one command by using `all` or a cell array as an input argument. For exact syntax, see the reference page for each function.

Enabling a Disabled Breakpoint

To reinstate a breakpoint that you previously disabled:

- 1 View breakpoint identifiers. At the `sedebug>>` prompt, enter this command:
`breakpoints`
- 2 Enter a command like the following, replacing `b1` with the identifier of the breakpoint that you want to reinstate:

```
enable b1
```

You might want to disable and enable a breakpoint to focus on behavior of a block during a particular time interval. A block breakpoint helps you focus on that block. Disabling the block breakpoint, when the simulation time is outside the time interval of interest, helps you focus on only those periods that are relevant to you.

Block Operations Relevant for Block Breakpoints

For each block that supports block breakpoints, the following lists indicate the operations that the block can perform. These operations are the only operations that appear in the debugger simulation log and that can cause the debugger to suspend the simulation at a block breakpoint. The actual operations that occur during a given simulation depend on block configuration and simulation behavior.

Attribute Function

- Entity advancing
- Setting attribute on entity

Attribute Scope

- Destroying entity
- Canceling event
- Entity advancing

Cancel Timeout

- Canceling event
- Entity advancing

Discrete Event Signal to Workspace

- Executing discrete-event signal to workspace

Enabled Gate

- Scheduling event
- Executing event
- Entity advancing
- Detected signal update

Entity Combiner

- Destroying entity

- Entity advancing
- Combining entities
- Canceling event

Note When the Entity Combiner block performs one of the listed operations, the debugger also suspends the simulation if you originally set the breakpoint on one of the component entities.

Entity Departure Counter

- Scheduling event
- Executing event
- Entity advancing
- Detected signal update

Entity Departure Function-Call Generator

- Entity advancing

Entity Sink

- Destroying entity
- Canceling event
- Entity advancing

Entity Splitter

- Destroying entity
- Entity advancing
- Splitting entity
- Canceling event

Note When the Entity Splitter block performs one of the listed operations, the debugger also suspends the simulation if you originally set the breakpoint on one of the component entities.

Event-Based Entity Generator

- Generating entity
- Scheduling event
- Executing event
- Detected signal update
- Canceling event

Event Filter

- Scheduling event
- Executing event
- Detected signal update
- Executing subsystem

FIFO Queue

- Entity advancing
- Queuing entity
- Scheduling event
- Executing event

Get Attribute

- Entity advancing

Infinite Server

- Scheduling event
- Executing event

- Entity advancing
- Canceling event

Input Switch

- Scheduling event
- Executing event
- Entity advancing
- Detected signal update

Instantaneous Entity Counting Scope

- Destroying entity
- Canceling event
- Entity advancing

Instantaneous Event Counting Scope

- Executing scope

LIFO Queue

- Entity advancing
- Queuing entity
- Scheduling event
- Executing event

N-Server

- Scheduling event
- Executing event
- Entity advancing
- Canceling event

Output Switch

- Scheduling event

- Executing event
- Entity advancing
- Detected signal update

Path Combiner

- Scheduling event
- Executing event
- Entity advancing
- Detected signal update

Priority Queue

- Entity advancing
- Queuing entity
- Scheduling event
- Executing event

Read Timer

- Entity advancing
- Reading timer on entity

Release Gate

- Scheduling event
- Executing event
- Entity advancing
- Detected signal update

Replicate

- Destroying entity
- Scheduling event
- Executing event

- Entity advancing
- Replicating entity

Schedule Timeout

- Scheduling event
- Executing event
- Entity advancing

Set Attribute

- Entity advancing
- Setting attribute on entity

Signal Latch

- Scheduling event
- Executing event
- Detected signal update
- Executing memory read
- Executing memory write

Signal Scope

- Executing scope

Signal-Based Function-Call Event Generator

- Scheduling event
- Executing event
- Detected signal update
- Executing function call

Single Server

- Scheduling event
- Executing event

- Canceling event
- Entity advancing
- Preempting entity

Start Timer

- Entity advancing
- Starting timer on entity

Time-Based Entity Generator

- Generating entity
- Scheduling event
- Executing event
- Entity advancing

X-Y Attribute Scope

- Destroying entity
- Canceling event
- Entity advancing

X-Y Signal Scope



- Executing scope



Animating

In this section...
“Introduction” on page 13-74
“Starting and Stopping Animation” on page 13-75
“Animating Signals and Entities” on page 13-76
“Controlling Animation Speed” on page 13-76
“Animating Without Debugger Text” on page 13-76
“Animating the Output Switching Using Signal Model Without Debugger Text” on page 13-77

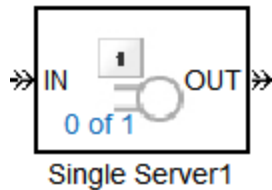
Introduction

The animation capability animates the entities and signals in a SimEvents model. With animation, you can observe SimEvents entities and signals as they move through the model. This behavior is enabled by default in the debugger. With animation, you can observe the following operations listed in this table.

Animation Icon	Operation
	Entity advancing through the block.
	Path history for the currently active entity motion. In the case of a virtual subsystem, this icon indicates that the entity is advancing within the subsystem; to observe that animation, open the subsystem. When the darker icon appears outside the subsystem, it indicates that the entity has exited the subsystem.

Animation Icon	Operation
	Signal update currently being executed.
	Signal blocks executed so far for currently independent operation. In the case of a virtual subsystem, this icon indicates that the signal updates are occurring within the subsystem; to observe that animation, open the subsystem. When the darker icon appears outside the subsystem, it indicates that the signal has exited the subsystem.

Enabling entity animation also displays the number of entities currently in the storage block in the bottom left corner of the block icon, as follows. This text appears on all storage blocks in the model. The software updates this number as simulation proceeds.



In this block, the 0 of 1 indicates that no entity is currently in the storage block.

Use the `sedb.animate` command to control animation settings. For example, you can turn the animation off and on, control the animation speed, and so forth.

Starting and Stopping Animation

By default, animation of a SimEvents model is enabled without the debugger. To view the animation, in the MATLAB Command Window, start the debugger for that model (`sedebg('model')`).

At the debugger prompt, to disable animation, type:

```
animate off
```

To restart animation, type:

```
animate on
```

Alternatively, you can type `animate` to toggle animation on and off.

Animating Signals and Entities

The `sedb.detail` command controls the amount of animation content in the model. By default, animation is enabled for signals and entities.

If you want to disable the animation of entities and signals, use the `detail` to disable them. For example:

```
detail('en', 0, 'sig', 0);
```

This command also disables all animation. Turning off entity details also turns off the display of the number of entities currently in storage blocks.

Controlling Animation Speed

By default, the animation delay is 0.0 seconds. You can change the animation speed by specifying a value between 0.0 and 5.0 seconds as the animation delay. For example, the following command slows the animation to update every 5 seconds:

```
animate 5.0
```

If you turn off the animation and then turn it on again, the animation delay is that of the previously specified delay value.

Animating Without Debugger Text

You can view a pure animation of your model that does not include text output from the debugger in the command window. At the MATLAB command prompt, type:

```
detail off
```

Animating the Output Switching Using Signal Model Without Debugger Text

To use the SimEvents debugger to animate the Output Switching Using Signal model:

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

```
sedemo_outputswitch
sedebug('sedemo_outputswitch')
```

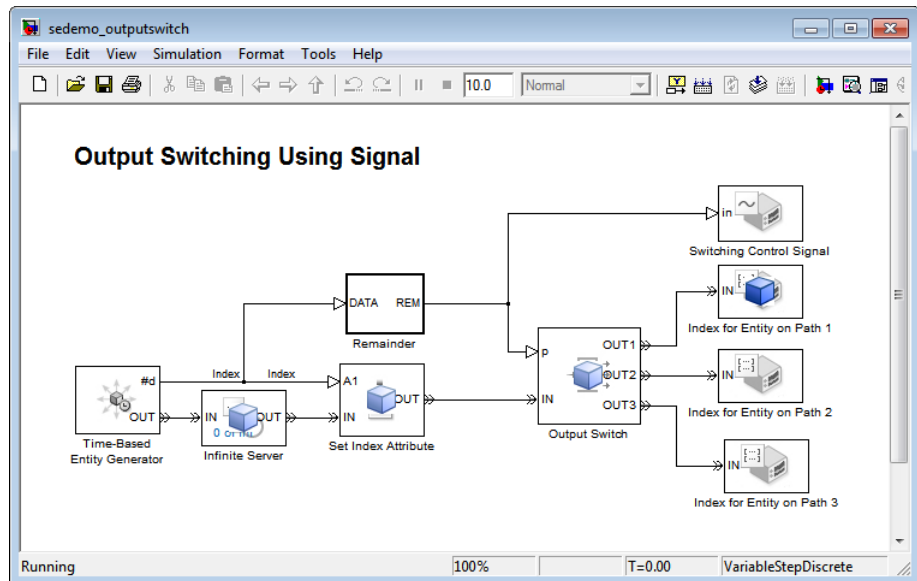
- 2 Turn on animation with a delay of 0.5 seconds.

```
animate on 0.5;
detail off
```

- 3 Simulate the model to the end.

```
runtoend
```

- 4 Observe the animation in the model, as illustrated.



Note When you animate your model without debugger text, there is no prompt visible in the command window that allows you to pause or exit the debugger. In this case, to pause the debugger, press CTRL+C. To exit, at the resulting prompt, type quit.

Common Problems in SimEvents Models

In this section...

“Unexpectedly Simultaneous Events” on page 13-79

“Unexpectedly Nonsimultaneous Events” on page 13-80

“Unexpected Processing Sequence for Simultaneous Events” on page 13-80

“Unexpected Use of Old Value of Signal” on page 13-81

“Effect of Initial Value on Signal Loops” on page 13-84

“Loops in Entity Paths Without Sufficient Storage Capacity” on page 13-85

“Unexpected Timing of Random Signal” on page 13-88

“Unexpected Correlation of Random Processes” on page 13-90

“Blocks that Require Event-Based Signal Input” on page 13-91

“Invalid Connections of Gateway Blocks” on page 13-91

“Race Conditions Involving Entity Departure Function Calls” on page 13-95

“Saving Models at Earlier Versions of SimEvents” on page 13-96

“Blocks That Reference Simulation Time” on page 13-97

Unexpectedly Simultaneous Events

An unexpected simultaneity of events can result from roundoff error in event times or other floating-point quantities, and might cause the processing sequence to differ from your expectation about when each event should occur. Computers’ use of floating-point arithmetic involves a finite set of numbers with finite precision. Events scheduled on the event calendar for times T and $T+\Delta t$ are considered simultaneous if $0 \leq \Delta t \leq 128 * \text{eps} * T$, where eps is the floating-point relative accuracy in MATLAB software and T is the simulation time.

If you have a guess about which events’ processing is suspect, adjusting event priorities or using the Instantaneous Event Counting Scope block can help you diagnose the problem. For examples involving event priorities, see “Example: Choices of Values for Event Priorities” on page 3-11. For an

example using the Instantaneous Event Counting Scope block, see “Example: Counting Events from Multiple Sources” on page 2-32.

Unexpectedly Nonsimultaneous Events

An unexpected lack of simultaneity can result from roundoff error in event times or other floating-point quantities. Computers’ use of floating-point arithmetic involves a finite set of numbers with finite precision. Events scheduled on the event calendar for times T and $T+\Delta t$ are considered simultaneous if $0 \leq \Delta t \leq 128 * \text{eps} * T$, where eps is the floating-point relative accuracy in MATLAB software and T is the simulation time.

If roundoff error is very small, the scope blocks might not reveal enough precision to confirm whether events are simultaneous or only close. An alternative technique is to use the Discrete Event Signal to Workspace block to collect data in the MATLAB workspace.

If your model requires that certain events be simultaneous, use modeling techniques aimed at effecting simultaneity. For an example, see “Example: Choices of Values for Event Priorities” on page 3-11.

Unexpected Processing Sequence for Simultaneous Events

An unexpected sequence for simultaneous events could result from the arbitrary or random handling of events having equal priorities, as described in “Processing Sequence for Simultaneous Events” on page 14-9. The sequence might even change when you run the simulation again. When the sequence is arbitrary, do not make any assumptions about the sequence or its repeatability.

If you copy and paste blocks that have an event priority parameter, the parameter values do not change unless you manually change them.

An unexpected processing sequence for simultaneous block operations, including signal updates, could result from interleaving of block operations. For information and examples, see “Interleaving of Block Operations” on page 14-39.

The processing sequence for simultaneous events could have unexpected consequences in the simulation. To learn more about the processing sequence that occurs in your simulation, use the SimEvents debugger. For tips on using the debugger to examine the processing sequence for simultaneous events, see “Exploring Simultaneous Events” on page 3-4.

To learn which events might be sensitive to priority, try perturbing the model by using different values of blocks’ **Resolve simultaneous signal updates according to event priority** or **Event priority** parameters. Then run the simulation again and see if the behavior changes.

Unexpected Use of Old Value of Signal

During a discrete-event simulation, multiple events or signal updates can occur at a fixed value of the simulation clock. If these events and signal updates are not processed in the sequence that you expect, you might notice that a computation or other operation uses a signal value from a previous time instead of from the current time. Some common situations occur when:

- A block defers the update of an output signal until a departing entity has either finished advancing to a subsequent storage block or been destroyed, but an intermediate nonstorage block in the sequence uses that signal in a computation or to control an operation. Such deferral of updates applies to most SimEvents blocks that have both an entity output port and a signal output port.

For examples, see “Example: Using a Signal or an Attribute” on page 13-82 and the Fine-Grained Management of Simultaneous Events demo.

For details, see “Interleaving of Block Operations” on page 14-39.

For a technique you can use when the situation involves the Output Switch block’s **p** input signal, see “Using the Storage Option to Prevent Latency Problems” on page 6-5.

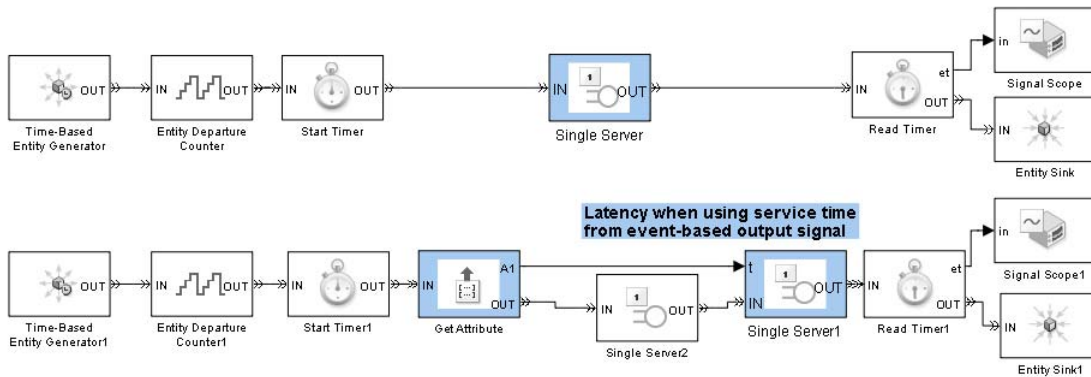
- A computation involving multiple signals is performed before all of the signals have been updated.

For details and an example, see “Update Sequence for Output Signals” on page 14-45.

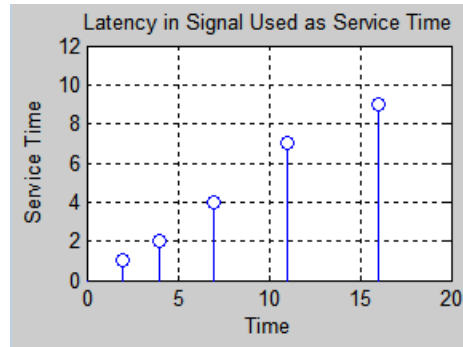
If you want notification of some of these situations, use the configuration parameters related to race conditions. For details, see “SimEvents Diagnostics Pane”.

Example: Using a Signal or an Attribute

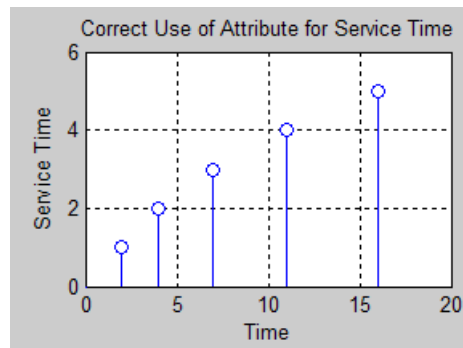
The goal in the next model is to use a service time of N seconds for the N th entity. The Entity Counter block stores each entity’s index, N , in an attribute. The top portion of the model uses the attribute directly to specify the service time. The bottom portion creates a signal representing the attribute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



The plot of the time in the bottom server block and a warning message in the Command Window both reveal a modeling error in the bottom portion of the model. The first entity’s service time is 0, not 1, while the second entity’s service time is 1, not 2. The discrepancy between entity index and service time occurs because the Get Attribute block processes the departure of the entity before the update of the signal at the A1 signal output port. That is, the server computes the service time for the newly arrived entity before the A1 signal reflects the index of that same entity. For more information about this phenomenon, see “Interleaving of Block Operations” on page 14-39.



The top portion of the model, where the server directly uses the attribute of each arriving entity, uses the expected service times. The sequential processing of an entity departure and a signal update does not occur because each entity carries its attributes with it.



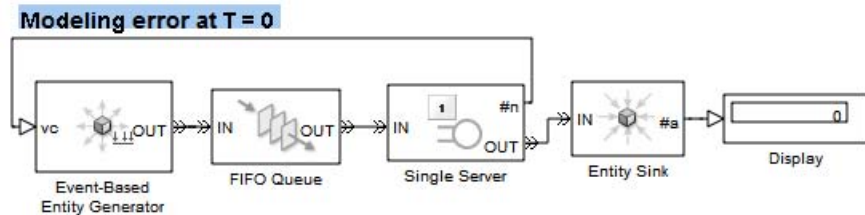
Tip If your entity possesses an attribute containing a desired service time, switching criterion, timeout interval, or other quantity that a block can obtain from either an attribute or signal, use the attribute directly rather than creating a signal with the attribute's value and having to ensure that the signal is up-to-date when the entity arrives.

Effect of Initial Value on Signal Loops

When you create a loop in a signal connection, consider the effect of initial values. If you need to specify initial values for event-based signals, see “Specifying Initial Values of Event-Based Signals” on page 4-14.

Example: Faulty Logic in Feedback Loop

The following model generates no entities because the logic is circular. The entity generator is waiting for a change in its input signal, but the server’s output signal never changes until an entity arrives or departs at the server.



To use the SimEvents debugger to see that the example has a modeling error:

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

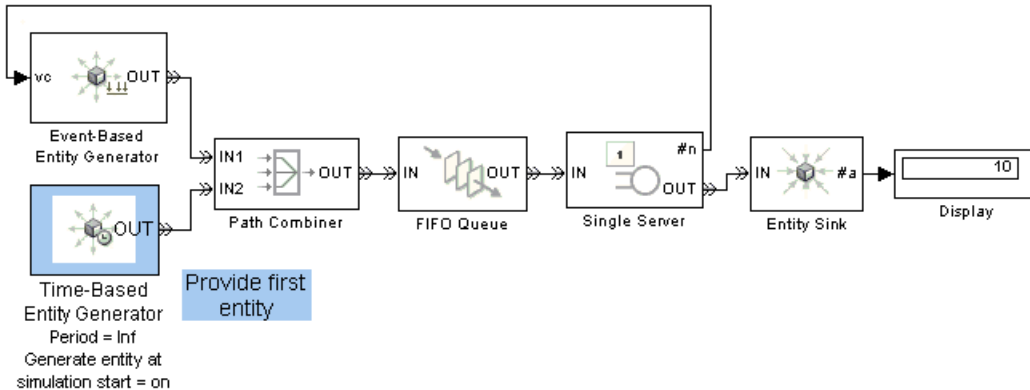
```
simeventsdocex('doc_ic_noentities')
sedebug('doc_ic_noentities')
```

- 2 Run the entire simulation. At the `sedebug>>` prompt, enter:

```
runtoend
```

If the simulation generated entities, the debugger would display messages in the Command Window to indicate that. The lack of output in this case shows that the simulation generates no entities.

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



You can use the debugger again to confirm that the revised model generates entities. If you use the preceding procedure, but substitute 'doc_ic_no_entities_fix' in place of 'doc_ic_no_entities', you can see that the debugger reports entity generations and other operations during the simulation of the revised model.

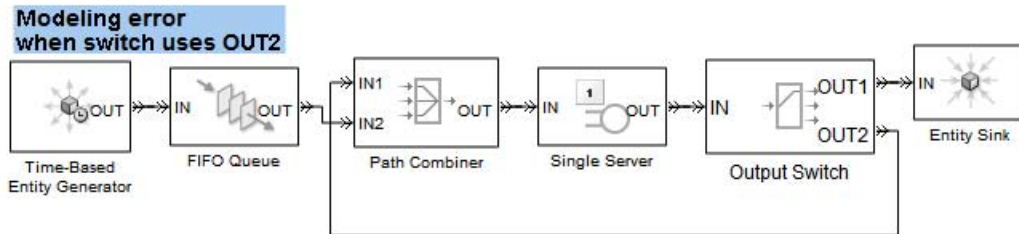
Loops in Entity Paths Without Sufficient Storage Capacity

An entity path that forms a loop should contain storage that will not become exhausted during the simulation. Storage blocks include queues and servers; for a list of storage blocks, see “Storage and Nonstorage Blocks” on page 14-51. The following example illustrates how the storage block can prevent a deadlock.

Example: Deadlock Resulting from Loop in Entity Path

The following model contains a loop in the entity path from the Output Switch block to the Path Combiner block. The problem occurs when the switch selects the entity output port **OUT2**. The entity attempting to depart from the server looks for a subsequent storage block where it can reside. It cannot reside in a routing block. Until the entity confirms that it can advance to a storage

block or an entity-destroying block, the entity cannot depart. However, until it departs, the server is not available to accept a new arrival. The result is a deadlock.



To use the SimEvents debugger to identify the deadlock:

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

```
simeventsdocex('doc_loop')
sedebug('doc_loop')
```

- 2 Run the simulation until the built-in breakpoint at the end of the simulation. At the `sedebug>>` prompt, enter:

```
cont
```

The debugger displays log messages in the Command Window so you can see what happens in the simulation. The latest time stamp in the messages is at $T = 3$:

```
%=====
Executing EntityGeneration Event (ev13)           Time = 3.000000000000000
: Entity = <none>                                Priority = 300
: Block = Time-Based Entity Generator
%.....%
Generating Entity (en4)
: Block = Time-Based Entity Generator
```


Hit built-in breakpoint for the end of simulation.

The lack of log messages after $T = 3$ reflects the deadlock.

- 3** If you inspect the final state of the switch, you see that it selects the entity output port **OUT2**:

```
blkinfo('doc_loop/Output Switch')
```

The output is:

```
OutputSwitch Current State          T = 10.000000000000000
Block (blk2): Output Switch

Advancing Entity    = <none>
Selected Output Port = 2
```

- 4** If you inspect the final state of the server, you see that it is holding an entity that completed its service at $T=2$:

```
blkinfo('doc_loop/Single Server')
```

The output is:

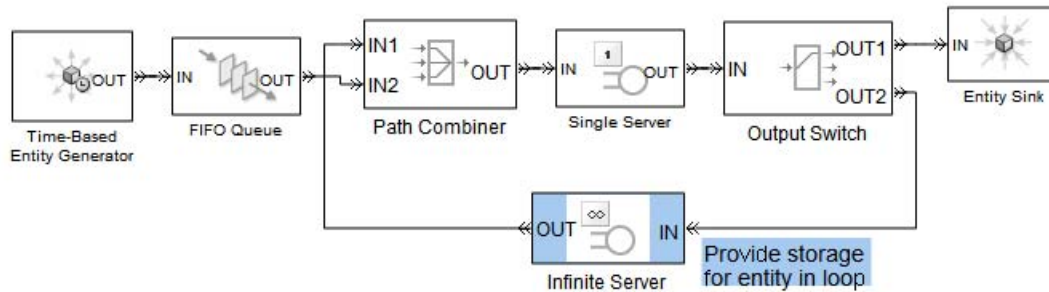
```
SingleServer Current State          T = 10.000000000000000
Block (blk6): Single Server

Entities (Capacity = 1):
Pos   ID      Status                Event   EventTime
1     en2     Service Completed    ev11   2
```

- 5** End the debugger session. At the `sedebg>>` prompt, enter:

```
sedb.quit
```

A better model includes a server with a service time of 0 in the looped entity path. This storage block provides a place for an entity to reside after it departs from the Output Switch block. After the service completion event is processed, the entity advances to the Path Combiner block and back to the Single Server block. The looped entity path connects to the Path Combiner block's **IN1** entity input port, not **IN2**. This ensures that entities on the looped path, not new entities from the queue, arrive back at the Single Server block.

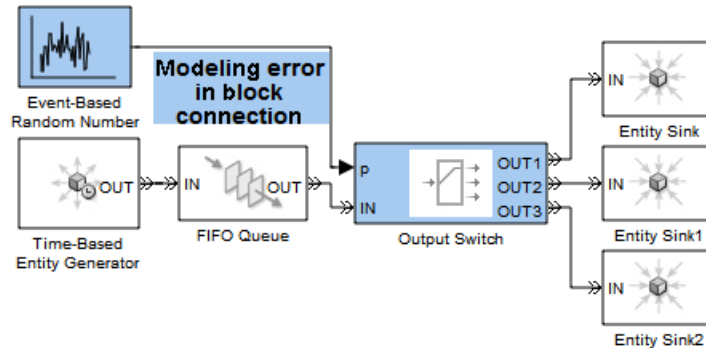


Unexpected Timing of Random Signal

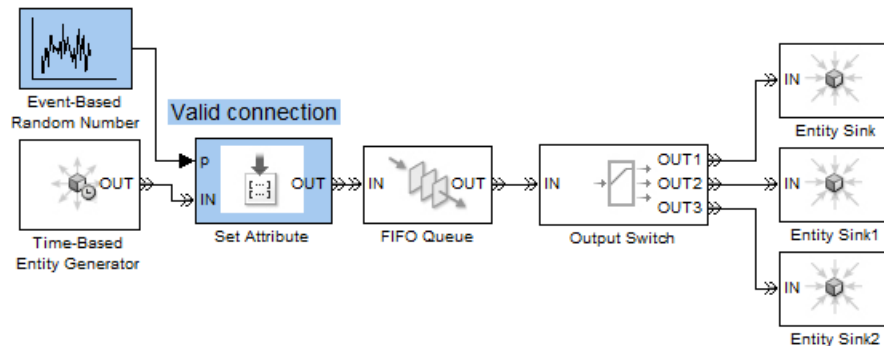
When you use the Event-Based Random Number block to produce a random event-based signal, the block infers from a subsequent block the events upon which to generate a new random number from the distribution. The sequence of times at which the block generates a new random number depends on the port to which the block is connected and on events occurring in the simulation. To learn how to use this block, see “Generating Random Signals” on page 4-4.

Example: Invalid Connection of Event-Based Random Number Generator

The following model is incorrect because the Event-Based Random Number block cannot infer from the **p** input port of an Output Switch block when to generate a new random number. The Output Switch block is designed to listen for changes in its **p** input signal and respond when a change occurs. The Output Switch cannot cause changes in the input signal value or tell the random number generator when to generate a new random number. The **p** input port of the Output Switch block is called a reactive port and it is not valid to connect a reactive signal input port to the Event-Based Random Number block.



If you want to generate a new random number corresponding to each entity that arrives at the switch, a better model connects the Event-Based Random Number block to a Set Attribute block and sets the Output Switch block's **Switching criterion** parameter to **From attribute**. The random number generator then generates a new random number upon each entity arrival at the Set Attribute block. The connection of the Event-Based Random Number block to the **A1** input port of the Set Attribute block is a supported connection because the **A2** port is a notifying port. To learn more about reactive ports and notifying ports, see the reference page for the Event-Based Random Number block.



Unexpected Correlation of Random Processes

An unexpected correlation between random processes can result from nonunique initial seeds in different dialog boxes. If you copy and paste blocks that have an **Initial seed** or **Seed** parameter, the parameter values do not change unless you manually change them. Such blocks include:

- Time-Based Entity Generator
- Event-Based Random Number
- Entity Splitter
- Blocks in the Routing library
- Uniform Random Number
- Random Number
- Masked subsystems that include any of the preceding blocks

Detecting Nonunique Seeds and Making Them Unique

To detect and correct some nonunique initial seeds, use a diagnostic setting:

- 1** Open the Configuration Parameters dialog box for the model using **Simulation > Configuration Parameters**.
- 2** Navigate to the **SimEvents Diagnostics** pane of the dialog box.
- 3** Set **Identical seeds for random number generators** to warning.

When you run the simulation, the application checks for nonunique **Initial seed** parameter values in SimEvents library blocks. Upon detecting any, the application issues a warning message that includes instructions for solving the problem.

Exceptions You must monitor seed values manually if your model contains random-number-generating blocks that come from libraries other than the SimEvents libraries.

To learn how to query and change seed values of such blocks, use the techniques illustrated in “Working with Seeds Not in SimEvents Blocks” on page 11-27.

Blocks that Require Event-Based Signal Input

If a SimEvents model contains a block that operates directly on an event-based signal, that block must be part of an event execution for it to update its output. Modify the model if the block cannot update its output during a simulation. An example block is an Atomic Subsystem block that does not have any input.

For such blocks to update their outputs during simulation, connect block input ports to event-based signals.

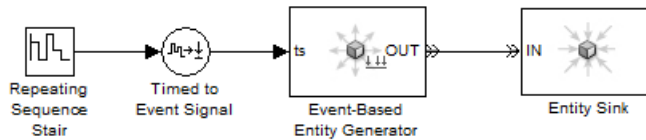
Invalid Connections of Gateway Blocks

When you select the configuration parameter **Prevent duplicate events on multiport blocks and branched signals** in your model, the software prohibits some connection types to other discrete-event blocks for the following gateway blocks:

- Timed to Event Signal
- Event to Timed Function-Call
- Timed to Event Function-Call

Invalid Connections of Timed to Event Signal Block

In the following model, the value of the **Generate Entities Upon:** parameter of the Event-Based Entity Generator block is set to **Sample time hit from port ts**. You cannot connect a time-based signal via a gateway block to a discrete-event block configured to sense sample time hits at its input port. If you run a model containing this type of connection, you see an error in the MATLAB command window.



The software does not allow the type of connection shown in this model because of a difference between the behavior you might expect when the simulation executes this configuration, and the *actual* simulation behavior when you also select the configuration parameter **Prevent duplicate events on multiport blocks and branched signals**. Because of this discrepancy, the simulation is likely to generate events at times that you do not intend.

For more information, see “Prevent duplicate events on multiport blocks and branched signals”.

In this model, you might expect the Event-Based Entity Generator block to sense sample time hits based on the sample rate of the Repeating Sequence Stair block and to generate entities based on those sample time hits. However, if the discrete-event system in the example is contained within a Simulink model, the software ensures that the entire discrete-event system—including any time-based signals fed into the discrete-event system via gateways—executes at the base rate of the Simulink model. As a result of this execution behavior, the Event-Based Entity Generator block senses sample time hits based on the execution rate of the Simulink model, *not* based on the sample rate of the incoming time-based signal.

The software enforces the behavior that all blocks of a discrete-event system contained in a Simulink model execute at the base rate of the Simulink model to ensure consistency between the two solver execution modes supported by discrete-event systems: *single-tasking* mode and *multitasking* mode.

Starting in SimEvents version 4.0 (R2012a), there is support for fixed-step solvers with discrete-event systems. Therefore, multitasking mode is now an available option for SimEvents® models.

For more information about:

- Choosing a solver for a SimEvents model, see *Solvers for Discrete-Event Systems* in the SimEvents documentation.

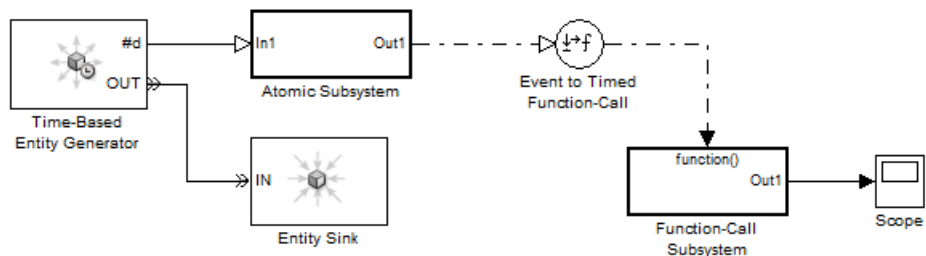
- Single tasking and multitasking modes, see “Single-Tasking and Multitasking Execution Modes” in the Simulink® Coder™ documentation.

In multitasking mode, the software executes tasks based on a priority it assigns to blocks in the model. The higher the sample rate of a block, the higher the priority the simulation assigns to it. In multitasking mode, without any enforcement of sample rates by the software, it might be possible for multiple gateway blocks to execute at different rates based on the sample rates of the signals connected to them. However, it is inadvisable to allow the simulation to read gateway blocks at rates that are slower than the base rate of the model. Serious data transfer problems might arise between tasks executing at different rates. In a discrete-event system, this behavior might lead to unpredictable production of events and values.

When a discrete-event system is contained in a Simulink model, the software constrains any time-based signal connected to the discrete-event system via a gateway block to be sampled at the base rate of the Simulink model. If you connect a time-based signal via a gateway block to a discrete-event block configured to sense sample time hits at its input port, you see an error in the MATLAB command window.

Invalid Connections of Event to Timed Function-Call Block

In the following model, the connection of the Event to Timed Function-Call gateway block is invalid. In this configuration, the gateway block accepts the **Out1** signal of the Atomic Subsystem block as an input. The role of the Event to Timed Function-Call gateway block in a model is to convert an incoming event-based function-call signal to a time-based signal. However, in this model, the **Out1** signal of the Atomic Subsystem block is already a time-based signal. The Event to Timed Function-Call gateway block performs no useful role.

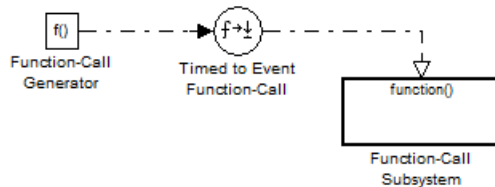


Connect the input port of the Event to Timed Function-Call gateway block to only the following blocks:

- Signal-Based Function-Call Generator
- Time-Based Function-Call Generator
- Entity Departure Function-Call Generator with value of **Timing of function call f1:** or **Timing of function call f2:** parameter set to After entity departure.

Invalid Connections of Timed to Event Function-Call Block

In the following model fragment, the output port of the Timed to Event Function-Call gateway block connects to a Function-Call Subsystem block from the Simulink block library. You can connect the output port of the Timed to Event Function-Call gateway block to only a block from the SimEvents library that accepts function-call input signals.

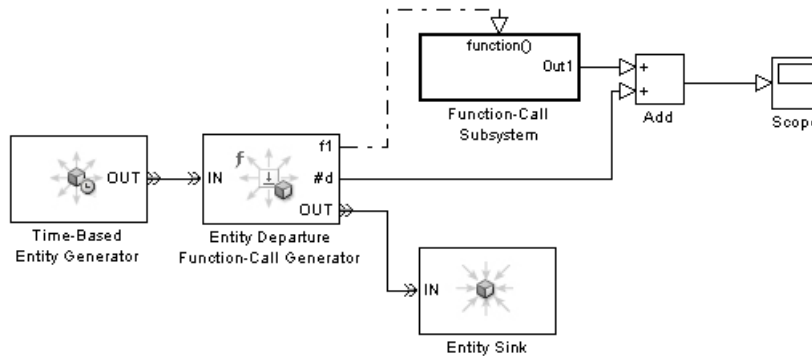


Connect the output port of the Timed to Event Function-Call gateway block to the function-call input port of only these blocks:

- Event-Based Entity Generator
- Release Gate
- Instantaneous Event Counting Scope
- Entity Departure Counter with value of **Reset counter upon:** parameter set to Function call from port fcn
- Signal-Based Function-Call Generator with value of **Function-call delay from:** parameter set to Signal port t or Dialog. If set to Dialog, enter a value greater than 0 for the **Function-call time delay** parameter

Race Conditions Involving Entity Departure Function Calls

The following model shows a connection of the Entity Departure Function-Call Generator block that, by default, causes an error in your simulation. In this modeling scenario, a race condition exists between the Entity Departure Function-Call Generator block and the Add block.

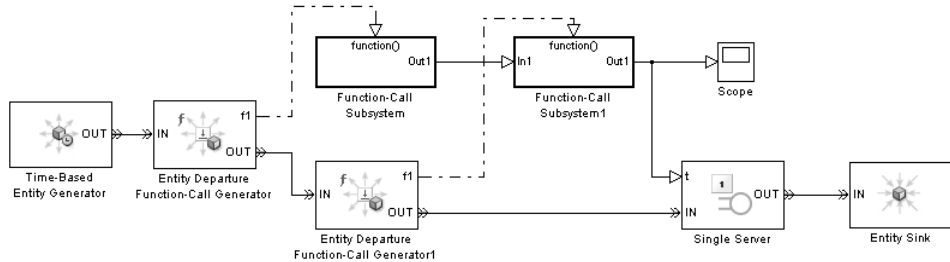


The **Timing of function call f1** parameter of the Entity Departure Function-Call Generator block is set to **Before entity departure**. Before an entity departs the generator block, the block produces a function call that causes the Add block to execute. When the Add block executes, it uses a value of the **#d** output of the generator block that is still awaiting update. This behavior might cause the Add block to produce an unexpected result.

If you do not require the **Before entity departure** behavior you can resolve this type of race condition shown by changing the value of the **Timing of function call f1** parameter of the block to **After entity departure**. If you require the **Before entity departure** behavior and have assessed the impact this race condition might have on your simulation results, you can disable the error that this race condition causes in your simulation. In the dialog box of the Entity Departure Function-Call Generator block, change the value of the **Use of signals awaiting update during function call** parameter from **error** to **none**.

The following model shows another type of connection using the Entity Departure Function-Call Generator block that, by default, causes an error in

your simulation. In this case, a race condition exists between both Entity Departure Function-Call Generator blocks.



The **Timing of function call f1** parameter of both generator blocks is set to **Before** entity departure. This setting causes a race condition because the function-call output of each block updates the inputs of the same block, the **Function-Call Subsystem1** block. If you use multiple **Before** entity departure function calls to update inputs of the same block, it is difficult to control which signal values are up to date when the simulation executes that block.

To resolve this race condition, if possible, set the **Timing of function call f1** parameter of both generator blocks to **After** entity departure. Or, disable the race condition error by changing the value of the **Use of signals awaiting update during function call** parameter from **error** to **none**.

Saving Models at Earlier Versions of SimEvents

Be aware that the following action cannot be reversed. If you use the **Save as type:** drop-down list to save your model at a version of SimEvents prior to 4.0 (R2011b), the software replaces unsupported blocks with empty subsystem blocks. The empty subsystem blocks are marked with the text **Replaced**. If you save your model in this way, the converted model is likely to generate incorrect results. To restore the functionality of your model, in the model editor, replace empty subsystem blocks with appropriate blocks from the version of SimEvents installed on your computer. When you have finished replacing blocks in your model, verify that all blocks are connected as you intend. Save the model to match the version of SimEvents installed on your computer.

Blocks That Reference Simulation Time

If your discrete-event system is contained within a Simulink model that uses a fixed-step solver, the discrete-event system cannot contain the following Simulink blocks that reference the current simulation time.

- Digital Clock
- Discrete-Time Integrator

A fixed-step solver processes events in the discrete-event system only at its major time steps. Therefore, for the listed blocks, the simulation does not accurately report the times at which events occur. If you run a model that uses a fixed-step solver and that contains one of the listed blocks, you see an error. To work around the issue, if possible, in the **Solver options** pane of the Configuration Parameters dialog box, change the value of the **Type** parameter to `Variable-step`.

For more information on choosing a solver for your SimEvents model, see “Solvers for Discrete-Event Systems” on page 14-2.

Recognizing Latency in Signal Updates

In some cases, the updating of an output signal or the reaction of a block to updates in its input signal can experience a delay:

- 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 14-39.
- 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 14-14.
- When the definition of a statistical signal suggests that its value can vary *continuously* as simulation time elapses, the block increases efficiency by updating the signal value only at key moments during the simulation. As a result, the signal has a somewhat outdated “approximate” value between such key moments, but corrects the value later.

The primary examples of this phenomenon are the signals that represent time averages, such as a server’s utilization percentage. The definitions of time averages involve the current time, but simulation performance would suffer drastically if the block recomputed the percentage at each time-based simulation step. Instead, the block recomputes the percentage only under these circumstances:

- Upon the arrival or departure of an entity
- When the simulation ends
- When you pause the simulation using **Simulation > Pause** or other means

For an example, see the reference page for the Single Server block.

When plotting statistics that, by definition, vary continuously as simulation time elapses, consider using a continuous-style plot. For example, set **Plot type** to **Continuous** in the Signal Scope block.

Learning More About SimEvents Software

Complementing the information in “How Simulink Works” and “Simulating Dynamic Systems” in the Simulink documentation, this section describes some aspects that are different for models that involve event-based processing.

- “Solvers for Discrete-Event Systems” on page 14-2
- “Execution of Blocks Having Event-Based Input Signals” on page 14-6
- “Event Sequencing” on page 14-9
- “Choosing How to Resolve Simultaneous Signal Updates” on page 14-14
- “Resolution Sequence for Input Signals” on page 14-15
- “Livelock Prevention” on page 14-24
- “Signal-Based Event Cycle Prevention” on page 14-26
- “Notifications and Queries Among Blocks” on page 14-30
- “Notifying, Monitoring, and Reactive Ports” on page 14-33
- “Interleaving of Block Operations” on page 14-39
- “Update Sequence for Output Signals” on page 14-45
- “SimEvents Support for Simulink Subsystems” on page 14-48
- “Storage and Nonstorage Blocks” on page 14-51
- “Blocks That Support Event-Based Input Signals” on page 14-53

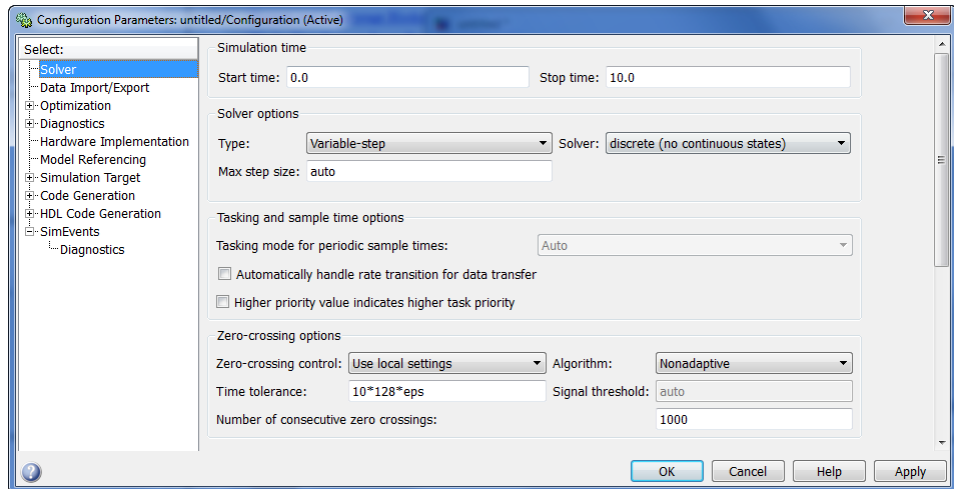
Solvers for Discrete-Event Systems

In this section...

“Variable-Step Solvers for Discrete-Event Systems” on page 14-3

“Fixed-Step Solvers for Discrete-Event Systems” on page 14-4

Depending on your configuration, you can use both variable-step and fixed-step solvers with discrete-event systems. To choose solver settings for your model, navigate to the **Solver** pane of the Configuration Parameters dialog box of the model.



When choosing a solver type for your model, use the following guidelines:

- If your model contains only event-based computation and excludes continuous and discrete time-based computation, choose the variable-step, discrete solver. In this case, if you select a variable-step continuous solver, the software detects that your model does not contain any blocks with continuous states (Simulink blocks) and automatically switches the solver to discrete (no continuous states). When the software makes this change, it notifies you with a message in the MATLAB command window.

- If your discrete-event system is within a Simulink model that also contains time-based modeling, choose either a variable-step or fixed-step solver, depending on your simulation requirements. For each solver type, the following sections describe the behavior of discrete-event systems when contained within such models.

Variable-Step Solvers for Discrete-Event Systems

If your discrete-event system is within a Simulink model that contains time-based modeling, and you choose a variable-step solver for the model, the Simulink solver has a major time step each time the discrete-event system needs to process events.

In addition, the SimEvents configuration parameter **Prevent duplicate events on multiport blocks and branched signals** affects the solver behavior. This parameter is introduced in R2012a. You can select the parameter once you use the `seupdate` function to migrate your model to the latest version of the software. For more information, see:

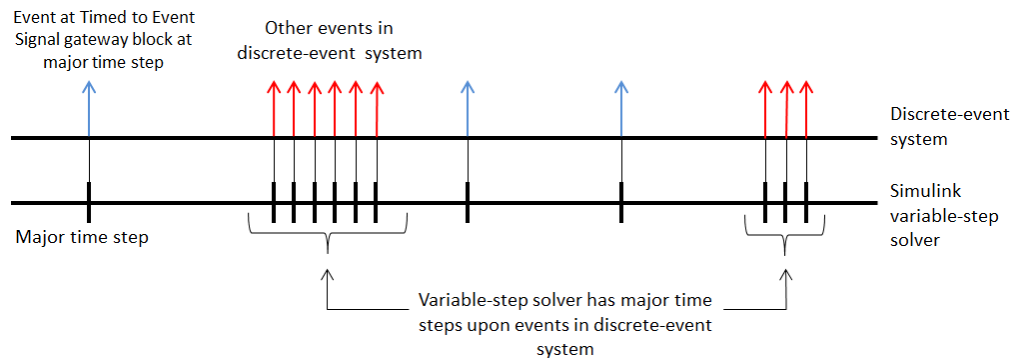
- “Migration Using `seupdate`” on page 15-14
- “Configuration Parameters”.

Depending on whether or not you select this configuration parameter in your model, there is a fundamental difference in how SimEvents uses Timed to Event Signal gateway blocks to convert time-based signals to event-based signals:

- If you do not select the configuration parameter, each Timed to Event Signal gateway block within the discrete-event system produces events based on the sample-time hits of the incoming time-based signal. In this scenario, the Simulink sample time diagnostic parameter **Single task rate transition** might generate a warning or error in your simulation, depending on the value you choose. For more information, see “Diagnostics Pane: Sample Time” in the Simulink documentation.
- If you select the configuration parameter, each gateway block produces a new event upon every major time step of the variable-step solver and *not* based on the sample rate of the incoming time-based signal. Because of this behavior, if you connect a time-based signal via a gateway block to a SimEvents block configured to sense a sample rate at its `ts` port, the

software identifies the connection as invalid and produces an error. For more information see “Invalid Connections of Gateway Blocks” on page 13-91. In this scenario, your simulation is not subject to the Simulink sample time diagnostic parameter, **Single task rate transition**.

The following graphic illustrates the behavior of the variable-step solver when used with a discrete-event system contained within a Simulink model.



Fixed-Step Solvers for Discrete-Event Systems

If you have a discrete-event system within a Simulink model that includes time-based modeling, you can choose a fixed-step solver for the model only if you also select the SimEvents configuration parameter **Prevent duplicate events on multiport blocks and branched signals**. If you do not select this configuration parameter, choosing a fixed-step solver causes an error in the simulation.

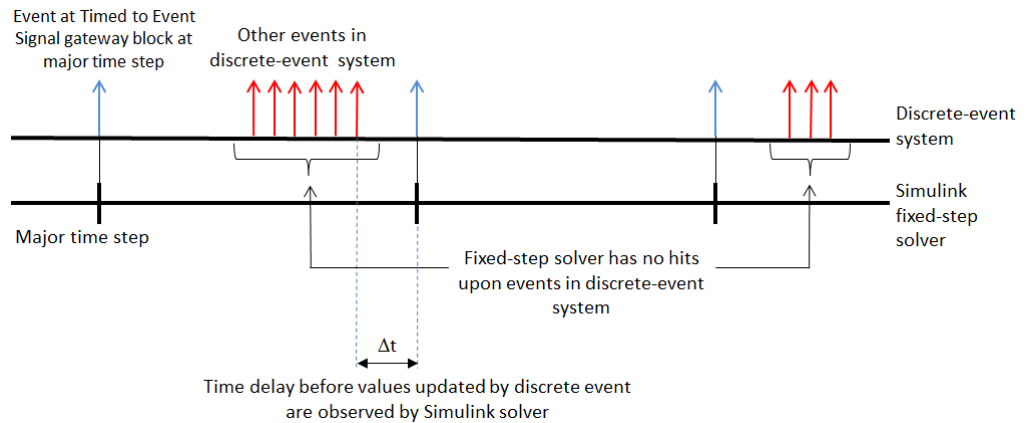
When you use a fixed-step solver, the simulation still executes events in the discrete-event system at the times at which they occur. However, these events do not cause the Simulink solver to have sample hits at those times. The software insulates the discrete-event system from the time-based portions of the Simulink model in this way by treating gateway blocks in a special manner:

- At Timed to Event Signal gateway blocks, for each major time step of the Simulink solver, the software produces a discrete event by sampling a fresh value at the gateway block. Any time-based signal that connects to a

gateway block, and has a sample time that is different to the base rate of the Simulink model, is subject to Simulink rate transition diagnostics.

- At Event to Timed Signal gateway blocks, the software converts event-based signals back to time-based signals at only the major time steps of the Simulink solver. This behavior means that there might be a delay between the time at which an event occurs in the discrete-event system and the time at which the time-based portion of the Simulink model observes this event.

The following graphic illustrates the behavior of the fixed-step solver when used with a discrete-event system.



Execution of Blocks Having Event-Based Input Signals

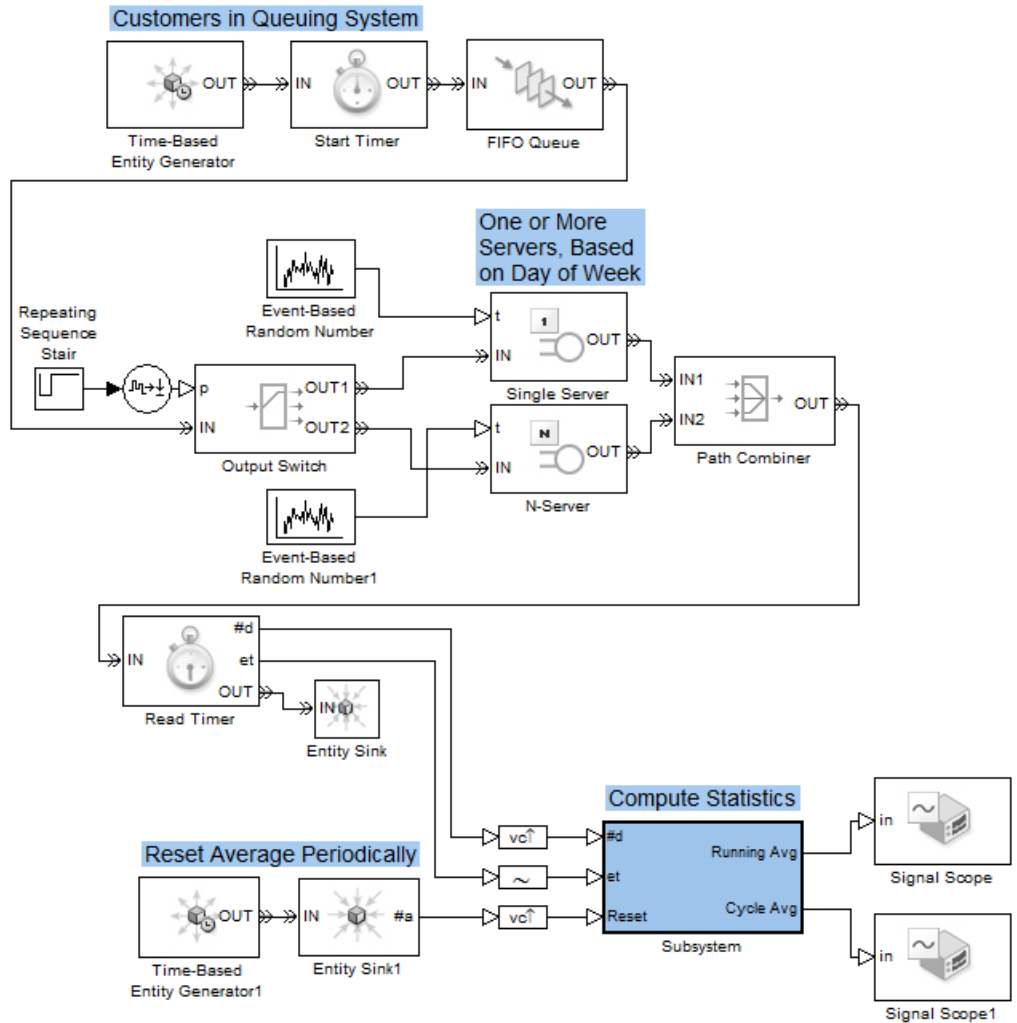
Response to Event-Based Input Signals

When a computational block or sink block has event-based input signals, the simulation process is slightly different from the process in “Simulating Dynamic Systems” in the Simulink documentation. The loop iteration process does not necessarily invoke Outputs methods and Update methods at each time step for a block that has event-based input signals. Instead, such a block executes only when an event-based input signal executes the block. The following table describes the circumstances under which an event-based input signal executes the block.

Event-Based Input Signal	When Signal Executes Block
Function-call signal	Upon each function call.
Signal that connects Event Filter block to Atomic Subsystem block, where the Event Filter block has the Execute downstream blocks upon signal-based events option selected	<p>The atomic subsystem executes upon each event that meets your criteria in the Type of signal-based event and other parameters of the Event Filter block.</p> <p>Events that do not meet your criteria do not execute the atomic subsystem. However, the values of the signal are visible if another input signal executes the subsystem.</p>
Signal that connects Event Filter block to Atomic Subsystem block, where the Event Filter block has the Execute downstream blocks upon signal-based events option deselected	<p>The signal never causes the atomic subsystem to execute.</p> <p>Values of the signal are visible to the subsystem if another input signal executes the subsystem.</p>
All other event-based signals	Upon each event.

Example: Execution of a Computational Block

In the following model from “Example: Resetting an Average Periodically” on page 11-12, the Atomic Subsystem block performs a computation on event-based input signals.



During the loop iteration process of the simulation, when an input signal executes the Atomic Subsystem block, the subsystem performs its computation and updates its output signals. For example, each time a customer leaves the queuing system, the **#d** signal of the Read Timer block has a sample time hit. This signal executes the Atomic Subsystem block, whose computation uses the latest values of all the input signals.

Input signals can potentially execute the Atomic Subsystem block multiple times at the same value of the simulation clock. In that case, the subsystem performs its computation multiple times. For example:

- If the 4th and 5th customers leave the queuing system simultaneously, the **#d** signal has two successive sample time hits at the same value of the simulation clock. The **#d** signal executes the Atomic Subsystem block twice: first with a **#d** value of 4 and then with a **#d** value of 5. When performing the computation with a **#d** value of 4, the subsystem does not know that the signal will change again before the simulation clock moves ahead.
- If a customer leaves the queuing system simultaneously with a reset of the average, the **#d** and **reset** signals both have sample time hits at the same value of the simulation clock. Suppose the simulation processes the sample time hit of the **#d** signal first. The **#d** signal executes the Atomic Subsystem block. The computation uses the new value of **#d** and the latest value of **reset** (before the sample time hit of **reset**). The subsystem does not know that the **reset** signal will change before the simulation clock moves ahead. After the **reset** signal has its sample time hit, the **reset** signal executes the Atomic Subsystem block. In this instance, the computation uses the new value of **reset** and the latest value of **#d**.

Event Sequencing

In this section...
“Processing Sequence for Simultaneous Events” on page 14-9
“Role of the Event Calendar” on page 14-10
“For Further Information” on page 14-12

Processing Sequence for Simultaneous Events

Even if simultaneous events occur at the same value of the simulation clock, the application processes them sequentially. The processing sequence must reflect causality relationships among events. This table describes the multiple-phase approach the application uses to determine a processing sequence for simultaneous events for which causality considerations alone do not determine a unique correct sequence.

Phase	Events Processed in This Phase	Processing Sequence for Multiple Events in This Phase
1	Events not scheduled on the event calendar	Arbitrary.
2	Events with priority SYS1	Same as the scheduling sequence (FIFO).
3	Events with priority SYS2	Same as the scheduling sequence (FIFO).
4	Events with numerical priority values	Ascending order of priority values. For equal-priority events, the sequence is random or arbitrary, depending on the model’s Execution order parameter.

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

The events with priority SYS1 enable the application to detect multiple signal updates before reacting to any of them. The events with priority SYS2 enable entities to advance in response to state changes.

For suggestions on how to use the information in the table when creating models, see “Choosing an Approach for Simultaneous Events” on page 3-7.

Role of the Event Calendar

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

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

Event Name	Event Type in Debugger	Scheduled On Event Calendar	How to Schedule Event on Event Calendar
Counter reset	CounterReset	Yes	
Delayed restart	DelayedRestart	Yes	
Entity advancement		No	
Entity destruction		No	
Entity generation	EntityGeneration	Yes	
Entity request	EntityRequest	Yes	
Function call	FunctionCall	Maybe	Select Resolve simultaneous signal updates according to event priority , if present, in the dialog box of the block that generates the function call. If the dialog box has no such option, the function call is not scheduled on the event calendar.
Gate (opening or closing)	Gate	Yes	
Memory read	MemoryRead	Maybe	Select Resolve simultaneous signal updates according to event priority on the Read tab of the Signal Latch block's dialog box.
Memory write	MemoryWrite	Maybe	Select Resolve simultaneous signal updates according to event priority on the Write tab of the Signal Latch block's dialog box.
New head of queue	NewHeadOfQueue	Yes	
Port selection	PortSelection	Yes	
Preemption		No	

Event Name	Event Type in Debugger	Scheduled On Event Calendar	How to Schedule Event on Event Calendar
Release	Release	Yes	
Sample time hit		No	
Service completion	ServiceCompletion	Yes	
Storage completion	StorageCompletion	Yes	
Subsystem	Subsystem	Maybe	Select Resolve simultaneous signal updates according to event priority in the dialog box of the Event Filter block that receives the signal that causes the subsystem execution.
Timeout	Timeout	Yes	
Trigger		No	
Value change		No	

When you use blocks that offer a **Resolve simultaneous signal updates according to event priority** option, your choice determines whether, or with what priority, particular events are scheduled on the event calendar. For information about this option, see “Resolution Sequence for Input Signals” on page 14-15 and “Choosing How to Resolve Simultaneous Signal Updates” on page 14-14.

For Further Information

- Chapter 3, “Managing Simultaneous Events” — Resolving race conditions in discrete-event simulations
- “Viewing the Event Calendar” on page 13-46 — Displaying event information in the MATLAB Command Window using the SimEvents debugger

- “Resolution Sequence for Input Signals” on page 14-15 — How the application resolves updates in input signals

Choosing How to Resolve Simultaneous Signal Updates

The **Resolve simultaneous signal updates according to event priority** option lets you defer certain operations until the application determines which other operations are supposed to be simultaneous. To use this option appropriately, you should understand your modeling goals, your model's design, and the way the application processes signal updates that are simultaneous with other operations in the simulation. The table indicates sources of relevant information that can help you use the **Resolve simultaneous signal updates according to event priority** option.

To Read About...	Refer to...	Description
Background	“Detection of Signal Updates” on page 14-15 and “Effect of Simultaneous Operations” on page 14-16	What simultaneous signal updates are, and the context in which the option is relevant
Behavior	“Specifying Event Priorities to Resolve Simultaneous Signal Updates” on page 14-17	How the simulation behaves when you select the option
	“Resolving Simultaneous Signal Updates Without Specifying Event Priorities” on page 14-20	How the simulation behaves when you do not select the option
Examples	“Example: Effects of Specifying Event Priorities” on page 3-25	Illustrates the significance of the option
	“Example: Choices of Values for Event Priorities” on page 3-11	Examines the role of event priority values, assuming you have selected the option
Tips	“Choosing an Approach for Simultaneous Events” on page 3-7	Tips to help you decide how to configure your model

For more general information about simultaneous events, information in “Overview of Simultaneous Events” on page 3-2 is also relevant.

Resolution Sequence for Input Signals

In this section...

“Detection of Signal Updates” on page 14-15

“Effect of Simultaneous Operations” on page 14-16

“Resolving the Set of Operations” on page 14-17

“Specifying Event Priorities to Resolve Simultaneous Signal Updates” on page 14-17

“Resolving Simultaneous Signal Updates Without Specifying Event Priorities” on page 14-20

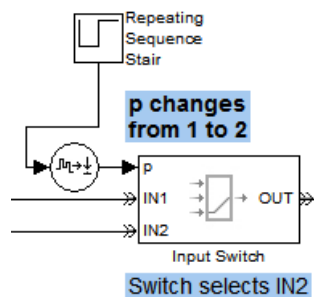
“For Further Information” on page 14-23

Detection of Signal Updates

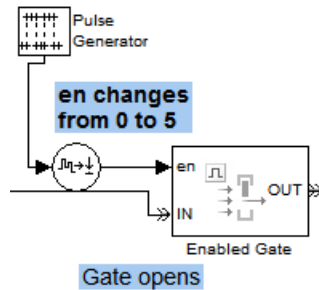
A block that possesses a reactive port listens for relevant updates in the input signal. A relevant update causes the block to react appropriately (for example, by opening a gate or generating a function call).

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



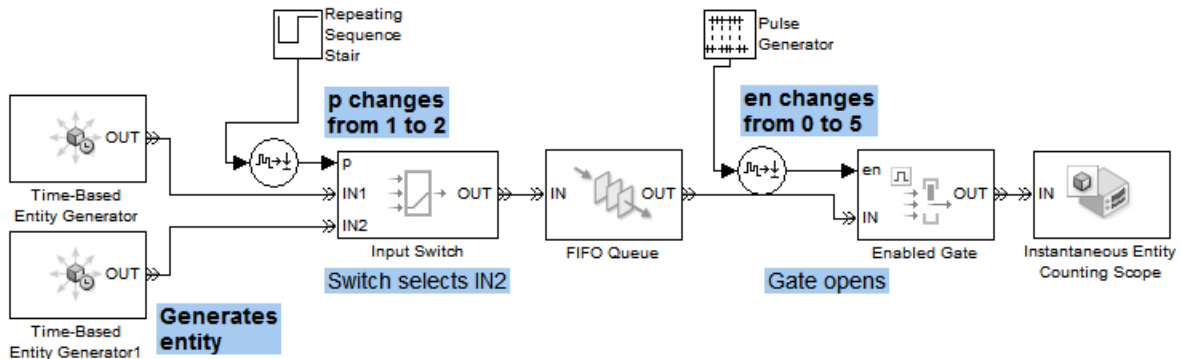
Signal Update That Causes a Gate to Open

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:

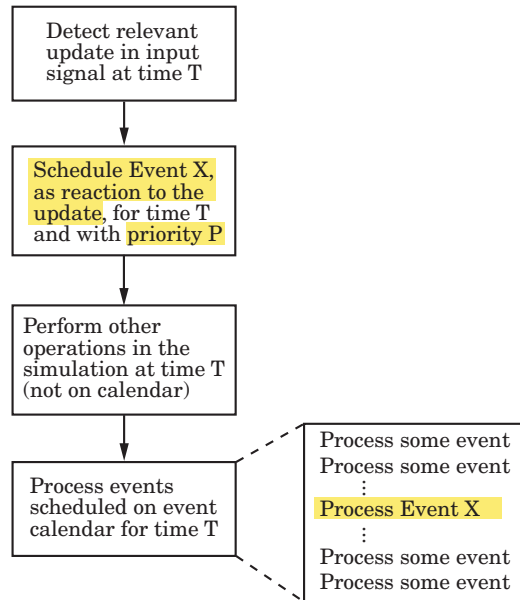
- The **Resolve simultaneous signal updates according to event priority** check box lets you choose the algorithm the application uses to resolve the reactions to signal updates, relative to other simultaneous operations in the simulation.
- If you select **Resolve simultaneous signal updates according to event priority**, the algorithm relies on the relative values of a set of numerical event priorities. You must set the event priority values using parameters in various blocks in the model.

Specifying Event Priorities to Resolve Simultaneous Signal Updates

If you select the **Resolve simultaneous signal updates according to event priority** option in a block that has a reactive port, and if the block detects a relevant update in the input signal that connects to the reactive port, then the application defers reacting to the update until it can determine which other operations are supposed to be simultaneous. Furthermore, the application sequences the reaction to the update using the numerical event priority that you specify.

Schematic Showing Application Processing

The next figure summarizes the steps the application takes when you select **Resolve simultaneous signal updates according to event priority** and the block has a relevant update in its input signal at a given time, T .



Processing for Numerical-Priority Events

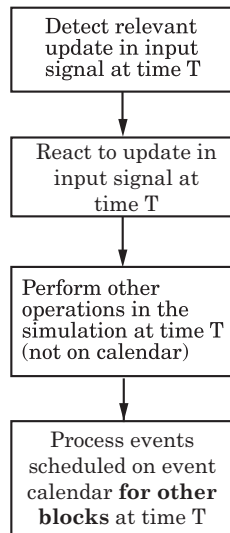
Contrast this with the schematics in Processing for System-Priority Events on page 14-21 and Processing for Immediate Updates on page 14-22.

The following blocks are exceptions to the processing sequence shown in the preceding graphic.

- Event Filter
- Signal-Based Function-Call Generator
- Signal Latch

For the listed blocks, if you select the configuration parameter **Prevent duplicate events on multiport blocks and branched signals** in your model, the software uses the **Event priority** parameter to help Simulink to

sort blocks in the model. The software no longer schedules an event that you can view on the SimEvents event calendar. The following graphic shows the processing sequence for these blocks.



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. The event priority of the event is the value of the **Event priority** or similarly named parameter in the block dialog box.

After scheduling the event, the application might perform other operations in the model at the current simulation time that are not scheduled on the event calendar. Examples of other operations can include updating other signals or processing the arrival or departure of entities.

Use of Event Priority Values

When the application begins processing the events that are scheduled on the event calendar for the current simulation time, event priority values influence the processing sequence. For details, see “Processing Sequence for Simultaneous Events” on page 14-9. As a result, the application is resolving

the update or change in the input signal (which might be simultaneous with other operations in the same block or in other blocks) according to the relative values of event priorities of all simultaneous events on the event calendar. A particular value of event priority is not significant in isolation; what matters is the ordering in a set of event priorities for a set of simultaneous events.

Resolving Simultaneous Signal Updates Without Specifying Event Priorities

If you do not select the **Resolve simultaneous signal updates according to event priority** option in a block that has a reactive port, and if the block detects a relevant update in the input signal that connects to the reactive port, then the block processes its reaction using one of these approaches:

- Defers reacting to the update until it can determine which other operations are supposed to be simultaneous. Furthermore, the application sequences the reaction to the update using an event priority value called a system priority, denoted `SYS1` or `SYS2`.

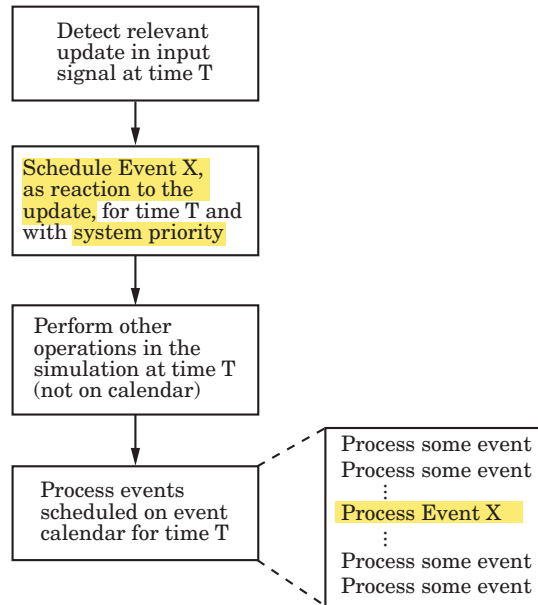
For details, see “System-Priority Events on the Event Calendar” on page 14-20.

- Reacts upon detecting the update (shown as “immediately” in the schematic illustrating this approach). The reaction, such as generation of a function call in the Signal-Based Function-Call Event Generator block, is not deferred, is not scheduled on the event calendar, and has no event priority. As a result, you are not resolving the sequence explicitly.

For details, see “Unprioritized Reactions to Signal Updates” on page 14-22.

System-Priority Events on the Event Calendar

The next figure summarizes the steps the application takes when you choose not to select **Resolve simultaneous signal updates according to event priority** in a block that uses system-priority events and that has a relevant update in its input signal at a given time, T .



Processing for System-Priority Events

Contrast this with the schematics in Processing for Numerical-Priority Events on page 14-18 and Processing for Immediate Updates on page 14-22. The difference between using a system priority and specifying a numerical priority for the same event is that the system priority causes earlier processing; see “Processing Sequence for Simultaneous Events” on page 14-9.

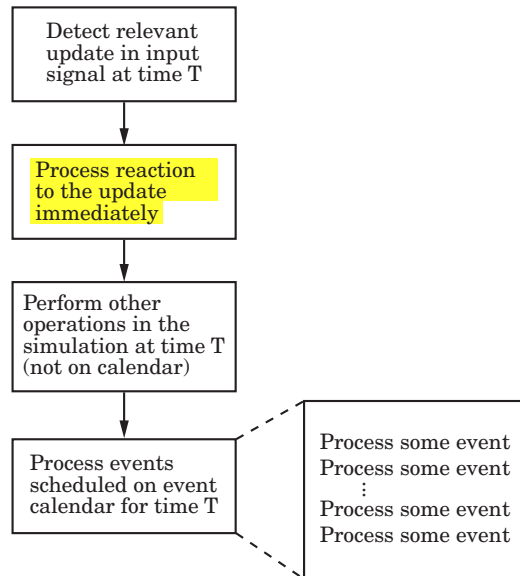
The blocks that can use system-priority events, unlike the blocks that can perform immediate updates, are characterized by the ability to directly change the state of an entity, including its location or attribute:

- Enabled Gate
- Entity Departure Counter
- Event-Based Entity Generator
- Input Switch
- Output Switch
- Path Combiner

- Release Gate

Unprioritized Reactions to Signal Updates

The next figure summarizes the steps the application takes when you do not select **Resolve simultaneous signal updates according to event priority** in a block that performs immediate updates and that has a relevant update in its input signal at a given time, T .



Processing for Immediate Updates

Contrast this with the schematics in Processing for Numerical-Priority Events on page 14-18 and Processing for System-Priority Events on page 14-21.

The blocks that can perform immediate updates, unlike the blocks that can use system-priority events, are characterized by the ability to produce signal outputs as a direct result of signal-based events:

- Discrete Event Inport
- Signal Latch
- Signal-Based Function-Call Event Generator

- Signal-Based Function-Call Generator

For Further Information

- Chapter 3, “Managing Simultaneous Events” — Resolving race conditions in discrete-event simulations
- “Reactive Ports” on page 14-35 — What constitutes a relevant update at a reactive port

Livelock Prevention

In this section...

“Overview” on page 14-24

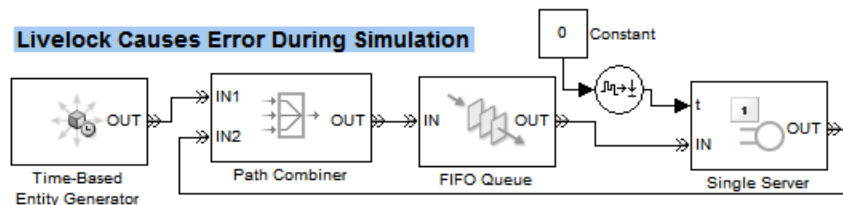
“Permitting Large Finite Numbers of Simultaneous Events” on page 14-25

Overview

SimEvents software includes features to prevent livelock. *Livelock* is a situation in which a block returns to the same state infinitely often at the same time instant. Typical cases include:

- An entity that moves along a looped entity path with no passage of time and no logic to stop the entity for a nonzero period of time
- An intergeneration time of 0 in an entity generator
- A Signal-Based Function-Call Generator block that calls itself with a time delay of 0
- A signal feedback loop in which event-based signals execute blocks in the loop repeatedly

The model below shows an example of livelock. The livelock prevention feature causes the simulation to halt with an error message. Without this error detection, an entity would move endlessly around the looped entity path without the simulation clock advancing.



Permitting Large Finite Numbers of Simultaneous Events

If your simulation creates a large, but not infinite, number of simultaneous events, consider increasing the model's thresholds related to livelock prevention.

For example, if you modify the Preloading Queues with Entities demo by setting both the capacity of the queue and the number of iterations of the function-call generator to 2000, then the simulation creates 2000 simultaneous events with no infinite loops. To prevent a spurious error in this situation, increase the model's limit on the number of events per block to at least 2000.

To change the thresholds related to livelock prevention, use this procedure:

- 1** Open the Configuration Parameters dialog box by selecting **Simulation > Configuration Parameters** from the model window's menu bar.
- 2** Navigate to the SimEvents pane of the Configuration Parameters dialog box.
- 3** Change the values of the **Maximum events per block** and **Maximum events per model** parameters.
- 4** Apply the change by clicking **OK** or **Apply**.

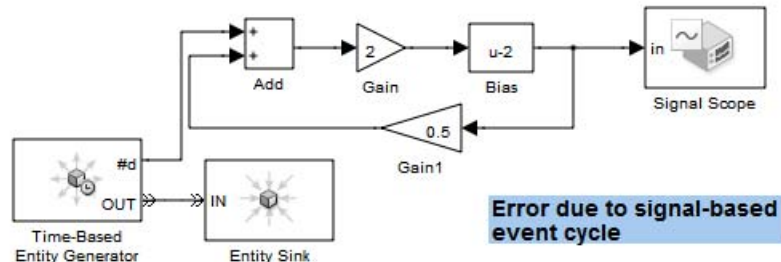
Signal-Based Event Cycle Prevention

This example shows how to eliminate an event-based signal cycle using the following resolution techniques. A signal-based event cycle is a loop formed by blocks that unconditionally update their outputs in response to a signal-based event. Such a cycle can cause an infinite loop during a simulation.

Error Caused by Signal-Based Event Cycle

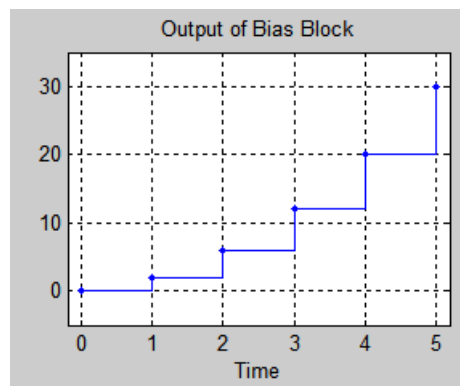
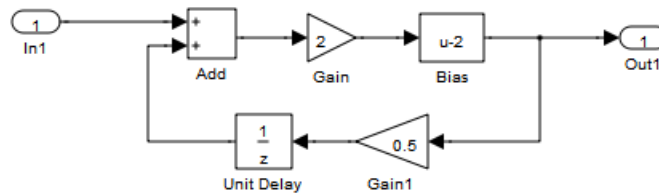
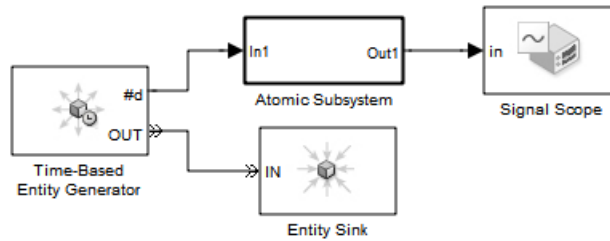
Simulating the following model causes an error because of a signal-based event cycle. If the simulation proceeds, a sample time hit of the **#d** signal causes the following actions to repeat in an infinite loop:

- The Add block executes because of a sample time hit of one of its input signals.
- The Gain, Bias, Signal Scope, and Gain1 blocks execute.



Resolution Using Atomic Subsystem and Unit Delay Blocks

The following model performs the computation in an Atomic Subsystem block, and includes a Unit Delay block on a signal line that connects to an input port of the Add block. During the simulation, a sample time hit of the **#d** signal causes the subsystem to execute. The subsystem executes each block inside the subsystem exactly once. The output of the Unit Delay block is the same as the output of the Gain1 block from the previous invocation of the subsystem.

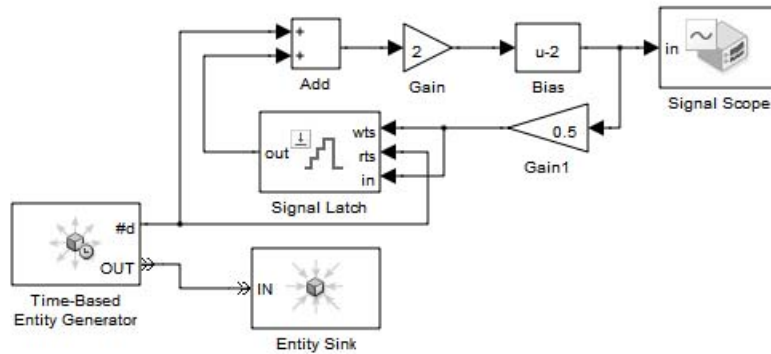


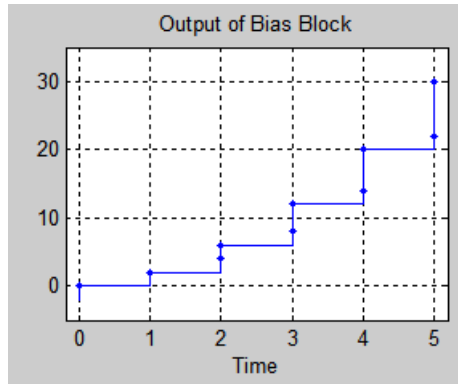
Alternative Resolution Using Signal Latch Block

The following model includes a Signal Latch block on a signal line that connects to the Add block. As a result, a sample time hit of the #d signal causes the following actions to occur once:

- The Add block executes because of a sample time hit of its first input signal.

- The Gain, Bias, Signal Scope, and Gain1 blocks execute.
- The sample time hit at the **rts** port of the Signal Latch block causes an update in the **out** output signal. The value of this signal is the same as the output of the Gain1 block from the previous time this entire sequence of actions occurred.
- The Add block executes again because of a sample time hit of its second input signal.
- The Gain, Bias, Signal Scope, and Gain1 blocks execute again.
- The two sample time hits at the **wts** port of the Signal Latch block cause the block to write the value of the **in** input signal to the block memory. This operation does not cause the block to update the **out** output signal.





Notifications and Queries Among Blocks

In this section...

“Overview of Notifications and Queries” on page 14-30

“Querying Whether a Subsequent Block Can Accept an Entity” on page 14-30

“Notifying Blocks About Status Changes” on page 14-31

Overview of Notifications and Queries

In a variety of situations, a SimEvents block notifies other blocks about changes in its status or queries other blocks about their status. These interactions among blocks occur automatically and are essential to the proper functioning of a discrete-event simulation. Entity request events are one kind of interaction among blocks. Entity request events appear on the event calendar, but other kinds of notifications and queries are not explicitly reported.

Querying Whether a Subsequent Block Can Accept an Entity

Before a SimEvents block outputs an entity, it queries the next block to determine whether that block can accept the entity. For example,

- When an entity arrives at an empty FIFO Queue block, the queue queries the next block. If that block can accept an entity, the queue outputs the entity at the head of the queue; otherwise, the queue holds the entity.
- While a Single Server block is busy serving, it does not query the next block. Upon completion of the service time, the server queries the next block. If that block can accept an entity, the server outputs the entity that has completed its service; otherwise, the server holds the entity.
- When an entity attempts to arrive at a Replicate block, the block queries each of the blocks connected to its entity output ports. If all of them can accept an entity, then the Replicate block copies its arriving entity and outputs the copies; otherwise, the block does not permit the entity to arrive there and the entity must stay in a preceding block.

- After a Time-Based Entity Generator block generates a new entity, it queries the next block. If that block can accept an entity, then the generator outputs the new entity; otherwise, the behavior of the Time-Based Entity Generator block depends on the value of its **Response when blocked** parameter.
- When a block (for example, a Single Server block) attempts to advance an entity to the Input Switch block, the server uses a query to check whether it is connected to the currently selected entity input port of the Input Switch block. If so, the Input Switch queries the next block to determine whether it can accept the entity because the Input Switch block cannot hold an entity for a nonzero duration.
- When an entity attempts to arrive at an Output Switch block, the block must determine which entity output port is selected for departure and whether the block connected to that port can accept the entity. If the **Switching criterion** parameter is set to **First port that is not blocked**, then the Output Switch block might need to query more than one subsequent block to determine whether it can accept the entity. If the **Switching criterion** parameter of the Output Switch block is set to **From attribute**, then the block also requires information about the entity that is attempting to arrive.

Notifying Blocks About Status Changes

When a SimEvents block undergoes certain kinds of status changes, it notifies other blocks of the change. This notification might cause the other blocks to change their behavior or status in some way, depending on the circumstances. For example,

- After an entity departs from a Single Server block, it schedules an entity request event to notify the preceding block that the server's entity input port has changed from unavailable to available.
- After an entity departs from a queue that was full to capacity, the queue schedules an entity request event to notify the preceding block that the queue's entity input port has changed from unavailable to available.
- After an entity departs from a switch or Enabled Gate block, it schedules an entity request event to determine whether another entity can advance from a preceding block. This process repeats until no further entity advancement can occur.

- When a Path Combiner block receives notification that the next block's entity input port has changed from unavailable to available, the Path Combiner block's entity input ports also become available. The block schedules an entity request event to notify preceding blocks that its entity input ports are available.

This case is subtle because the Path Combiner block usually has more than one block to notify, and the sequence of notifications can be significant. See the block's reference page for more information about the options.

- When an entity arrives at a Single Server block that has a **t** signal input port representing the service time, that port notifies the preceding block of the need for a new service time value. If the preceding block is the Event-Based Random Number block, then it responds by generating a new random number that becomes the service time for the arriving entity.

This behavior occurs because the **t** signal input port is a notifying port; see “Notifying, Monitoring, and Reactive Ports” on page 14-33 for details.

Notifying, Monitoring, and Reactive Ports

In this section...

“Overview of Signal Input Ports of SimEvents Blocks” on page 14-33

“Notifying Ports” on page 14-33

“Monitoring Ports” on page 14-34

“Reactive Ports” on page 14-35

“Connecting Event-Based Signal Generators to Reactive Ports” on page 14-36

Overview of Signal Input Ports of SimEvents Blocks

Signal input ports of SimEvents blocks fall into these categories:

- Notifying ports, which notify the preceding block when a certain event has occurred
- Monitoring ports, which help you observe signal values
- Reactive ports, which listen for updates or changes in the input signal and cause an appropriate reaction in the block possessing the port

The distinctions are relevant when you use the Event-Based Random Number or Event-Based Sequence block. For details, see these topics:

- Event-Based Random Number reference page
- Event-Based Sequence reference page
- “Generating Random Signals” on page 4-4
- “Using Data Sets to Create Event-Based Signals” on page 4-7

Notifying Ports

Notifying ports, listed in the table below, notify the preceding block when a certain event has occurred. When the preceding block is the Event-Based Random Number or Event-Based Sequence block, it responds to the notification by generating a new output value. Other blocks ignore the notification.

List of Notifying Ports

Signal Input Port	Block	Generate New Output Value Upon
<i>Attribute name</i> (e.g. Attribute1 , Attribute2)	Set Attribute	Entity arrival
in	Signal Latch	Write event
e1, e2	Entity Departure Function-Call Generator	Entity arrival
	Signal-Based Function-Call Generator	Relevant signal-based event, depending on configuration of block
t	Signal-Based Function-Call Generator	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
Unlabeled	Event to Timed Signal

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	

List of Reactive Ports (Continued)

Signal Input Port	Block	Relevant Update
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 Function-Call Generator	
	Signal-Based Function-Call Event Generator	
wts, wtr, wvc, rts, rtr, rvc	Signal Latch	Sample time hit at wts or rts port Appropriate trigger at wtr or rtr port Appropriate value change at wvc or rvc port
Unlabeled input port	Event Filter	Depends on block parameter values. Choices are: Sample time hit Appropriate trigger Appropriate value change
Unlabeled input port	Initial Value	Sample time hit

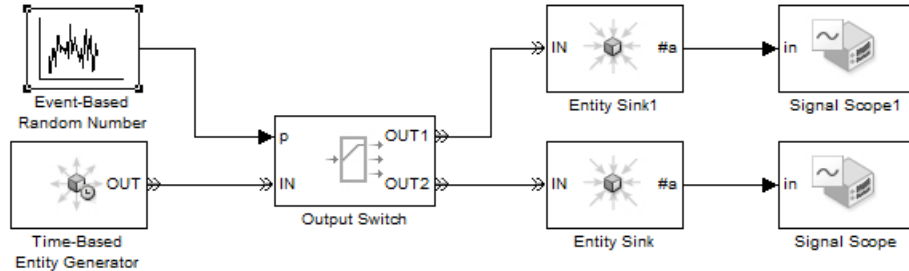
For triggers and value changes, “appropriate” refers to the direction you specify in a **Type of change in signal value** or **Trigger type** parameter in the block’s dialog box.

Connecting Event-Based Signal Generators to Reactive Ports

To connect an event-based signal generator to a reactive port, you must use a workaround because you cannot connect the Event-Based Random Number block directly to a reactive input port of a SimEvents block or indirectly, via

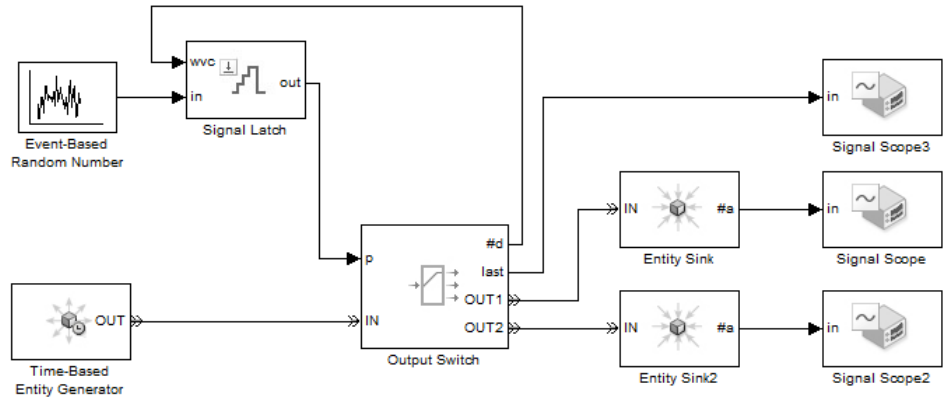
a single-input-single-output (SISO) block that supports event-based input signals.

The following model shows an invalid connection of the Event-Based Random Number block.



In this model, the Event-Based Random Number block is directly connected to the **p** input port of an Output Switch block. The **p** input port of the Output Switch block is a reactive port. This connection is invalid because the Event-Based Random Number block cannot infer from the **p** input port of the Output Switch block when to generate a new random number. The Output Switch block is designed to listen for changes in its **p** input signal and respond when a change occurs. The Output Switch cannot cause changes in the input signal value or inform the random number generator when to generate a new random number.

The following model shows a workaround to this invalid connection.



In this workaround, the Event-Based Random Number block is not directly connected to the **p** input port of the Output Switch block. Instead, a Signal Latch block stores each value that the Event-Based Random Number block generates.

When an entity departs the Output Switch block, the **#d** output signal of that block increments by a value of 1 and triggers a write-to-memory event in the Signal Latch block. In this model, the value of the **Read from memory upon** parameter of the Signal Latch block is set to Write to memory event. This setting means that each time the Event-Based Random Number block writes a new value to the Signal Latch block, the Signal Latch block outputs its current stored value to the **p** input port of the Output Switch block.

Interleaving of Block Operations

In this section...

“Overview of Interleaving of Block Operations” on page 14-39

“How Interleaving of Block Operations Occurs” on page 14-39

“Example: Sequence of Departures and Statistical Updates” on page 14-40

Overview of Interleaving of Block Operations

During the simulation of a SimEvents model, some sequences of block operations become interleaved when the application processes them. Interleaving can affect the simulation behavior. This section describes and illustrates interleaved block operations to help you understand the processing and make appropriate modeling choices.

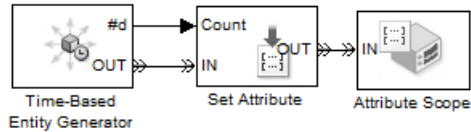
How Interleaving of Block Operations Occurs

At all simulation times from an entity’s generation to destruction, the entity resides in a block (or more than one block, if the entity advances from block to block at a given time instant). Blocks capable of holding an entity for a nonzero duration are called storage blocks. Blocks that permit an entity arrival but must output the entity at the same value of the simulation clock are called nonstorage blocks. During a simulation, whenever an entity departs from a block, the application processes enough queries, departures, arrivals, and other block operations until either a subsequent storage block detects the entity’s arrival or the entity is destroyed. Some block operations, including the updates of statistical output signals that are intended to be updated after the entity’s departure, are deferred until after a subsequent storage block detects the entity’s arrival or the entity is destroyed.

To change the sequence of block operations, you might need to insert storage blocks in key locations along entity paths in your model, as illustrated in “Example: Sequence of Departures and Statistical Updates” on page 14-40. A typical storage block to insert for this purpose is a server whose service time is 0.

Example: Sequence of Departures and Statistical Updates

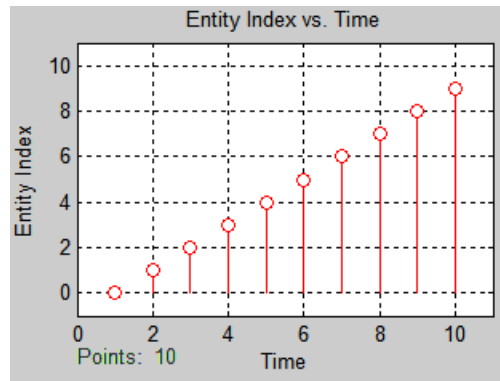
Consider the sequence of operations in the Time-Based Entity Generator, Set Attribute, and Attribute Scope blocks shown below.



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

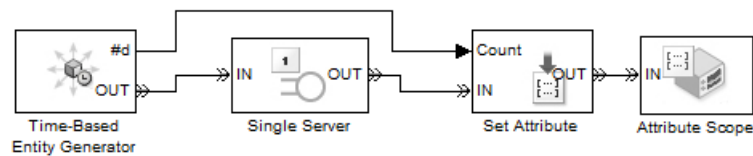
Order	Operation	Block
1	Entity generation	Time-Based Entity Generator
2	Entity advancement to nonstorage block	From Time-Based Entity Generator to Set Attribute
3	Assignment of attribute using value at A1 signal input port	Set Attribute
4	Entity advancement to nonstorage block	From Set Attribute to Attribute Scope
5	Entity destruction	Attribute Scope
6	Update of plot	Attribute Scope
7	Update of signal at #d signal output port	Time-Based Entity Generator

The final operation of the Time-Based Entity Generator block is deliberately processed *after* operations of subsequent blocks in the entity path are processed. This explains why the plot shows a value of 0, not 1, at $T=1$.



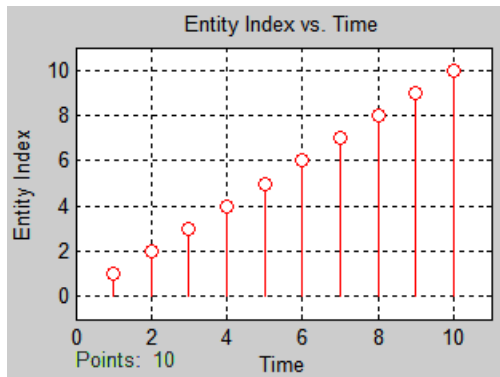
Altering the Processing Sequence

If you want to be sure that the Set Attribute block reads the value at the **A1** signal input port after the Time-Based Entity Generator block has updated its **#d** output signal, then insert a storage block between the two blocks. In this simple model, you can use a Single Server block with a **Service time** parameter of 0. The model, table, and plot are below.



Order	Operation	Block
1	Entity generation	Time-Based Entity Generator
2	Entity advancement to storage block	From Time-Based Entity Generator to Single Server
3	Update of signal at #d signal output port	Time-Based Entity Generator
4	Service completion	Single Server
5	Entity advancement to nonstorage block	From Single Server to Set Attribute

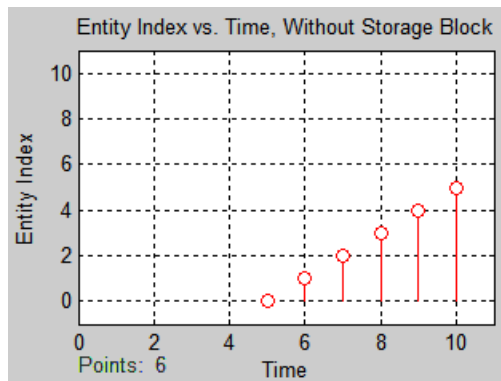
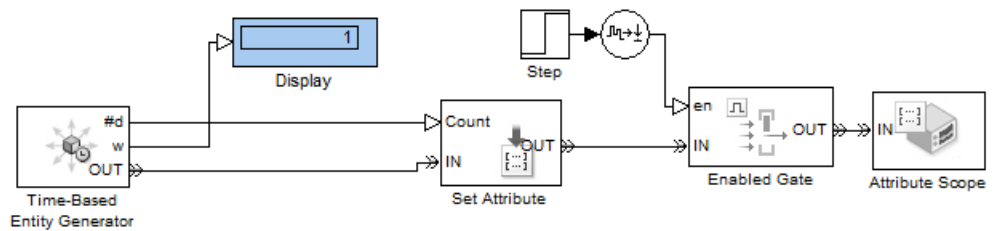
Order	Operation	Block
6	Assignment of attribute using value at A1 signal input port	Set Attribute
9	Entity advancement to nonstorage block	From Set Attribute to Attribute Scope
10	Entity destruction	Attribute Scope
11	Update of plot	Attribute Scope



Consequences of Inserting a Storage Block

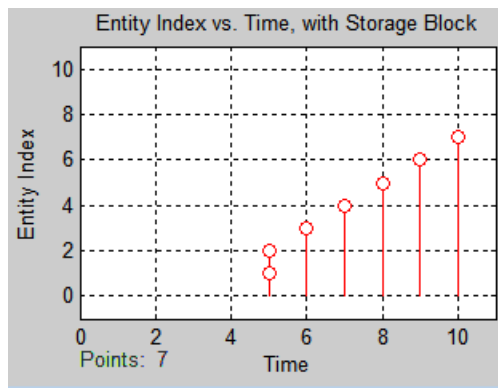
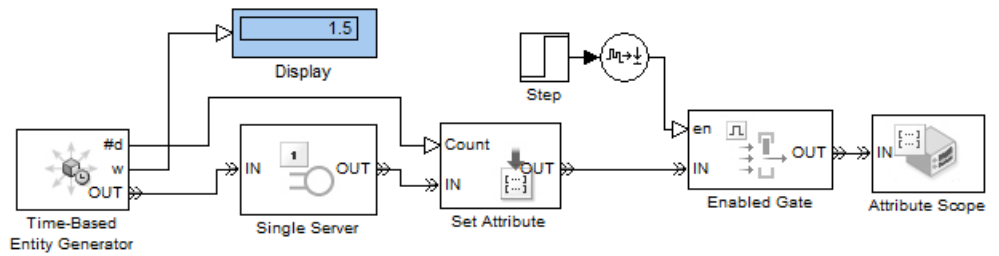
If the storage block you have inserted to alter the processing sequence holds the entity longer than you expect (beyond the zero-duration service time, for example), be aware that your simulation might change in other ways. You should consider the impact of either inserting or not inserting the storage block.

For example, suppose you add a gate block to the preceding example and view the average intergeneration time, w , of the entity generator block. When the gate is closed, a newly generated entity cannot advance immediately to the scope block. Whether this entity stays in the entity generator or a subsequent server block affects the w signal, as shown in the figures below.



Model with Gate and Without Storage Block

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



Model with Gate and Storage Block

Update Sequence for Output Signals

In this section...

“Determining the Update Sequence” on page 14-45

“Example: Detecting Changes in the Last-Updated Signal” on page 14-46

Determining the Update Sequence

When a block produces more than one output signal in response to events, the simulation behavior might depend on the sequence of signal updates relative to each other. This is especially likely if you use one of the signals to influence a behavior or computation that also depends on another one of the signals, as in “Example: Detecting Changes in the Last-Updated Signal” on page 14-46 and .

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.

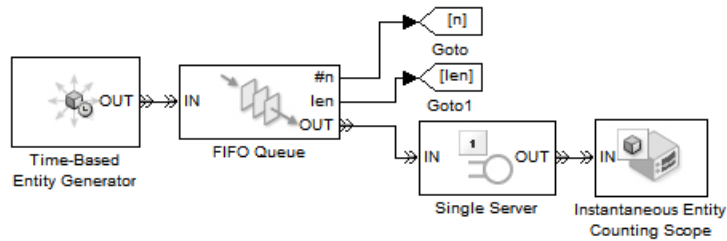
For example, the reference page for the N-Server block indicates that the **w** and **util** signals are updated in an arbitrary sequence relative to each other. The Order of Update column in the Signal Output Ports table lists the same number for both the **w** and **util** signals.

- 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

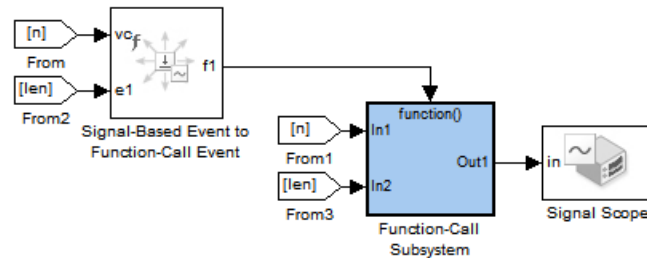
example, the reference page for the Enabled Gate block does not indicate an update sequence because the block can output only one signal.

Example: Detecting Changes in the Last-Updated Signal

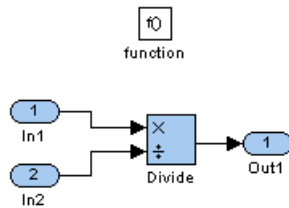
The example below plots the ratio of the queue's current length to the time average of the queue length. The FIFO Queue block produces **#n** and **len** signals representing the current and average lengths, respectively. The computation of the ratio occurs in a function-call subsystem that is called when the Signal-Based Function-Call Generator 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 Function-Call Generator 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.

SimEvents Support for Simulink Subsystems

You can use SimEvents blocks (discrete-event blocks) without restriction in Simulink Virtual Subsystems, and in Simulink® Nonvirtual Subsystems, observing some specific guidelines.

For more information about Simulink subsystems, see “Systems and Subsystems” in the Simulink documentation.

In this section...
“Discrete-Event Blocks in Virtual Subsystems” on page 14-48
“Discrete-Event Blocks in Nonvirtual Subsystems” on page 14-48
“Discrete-Event Blocks in Variant Subsystems” on page 14-50

Discrete-Event Blocks in Virtual Subsystems

You can use discrete-event blocks without restriction in a virtual subsystem.

Discrete-Event Blocks in Nonvirtual Subsystems

When the configuration parameter **Prevent duplicate events on branched signals and multiport blocks** is enabled, you can use discrete-event blocks in an atomic subsystem.

For more information about:

- The configuration parameter **Prevent duplicate events on multiport blocks and branched signals**, see “Prevent duplicate events on multiport blocks and branched signals” in the SimEvents documentation.
- Atomic Subsystems, see “Subsystem, Atomic Subsystem, Nonvirtual Subsystem, CodeReuse Subsystem” in the Simulink documentation.

When you use discrete-event blocks in an atomic subsystem, follow these guidelines:

- The entire discrete-event subsystem, which includes all discrete-event blocks and any blocks from the Simulink library that connect to

discrete-event blocks, must reside entirely within the atomic subsystem. You cannot route entities into, or out of, the atomic subsystem.

- If you want to route time-based signals *into* an atomic subsystem that contains discrete-event blocks, use a Timed to Event Signal gateway block within the atomic subsystem to perform signal conversion.
- If you want to route time-based signals *out of* an atomic subsystem that contains discrete-event blocks, use an Event to Timed Signal gateway block within the atomic subsystem to perform signal conversion.
- If you want to connect two or more atomic subsystems that contain discrete-event blocks, each atomic subsystem must meet all the preceding conditions.

In the following table, the types of nonvirtual subsystem block listed do not support the use of discrete-event blocks.

Type of Nonvirtual Subsystem	Examples	Further Information
Conditionally executed subsystem	<ul style="list-style-type: none"> • Triggered Subsystem • Enabled Subsystem • Function-Call Subsystem 	See “Creating Conditional Subsystems” in the Simulink documentation.
Subsystem to model control flow logic	<ul style="list-style-type: none"> • For Each Subsystem • If Action Subsystem • While Iterator Subsystem 	See “Modeling Control Flow Logic” in the Simulink documentation.

Type of Nonvirtual Subsystem	Examples	Further Information
	<ul style="list-style-type: none"> • Switch Case subsystem 	
Subsystem to reference other models or blocks	<ul style="list-style-type: none"> • Model subsystem • Model Variant subsystem • Configurable Subsystem 	See “Overview of Model Referencing”, Model Info and Configurable Subsystem in the Simulink documentation.

Discrete-Event Blocks in Variant Subsystems

You can use discrete-event blocks in a variant subsystem. The software permits both entities and time-based signals to enter and depart a virtual variant.

However, if you use an atomic subsystem as a variant, or within a variant, then that atomic subsystem must obey the rules for using discrete-event blocks in nonvirtual subsystems. These rules are described in “Discrete-Event Blocks in Nonvirtual Subsystems” on page 14-48. An atomic subsystem is the only type of nonvirtual subsystem that can contain discrete-event blocks, even when the nonvirtual subsystem is contained within a variant subsystem.

Storage and Nonstorage Blocks

In this section...
“Storage Blocks” on page 14-51
“Nonstorage Blocks” on page 14-51

For the significance of the distinction between storage and nonstorage blocks, see “Interleaving of Block Operations” on page 14-39.

Storage Blocks

These blocks are capable of holding an entity for a nonzero duration:

- Blocks in Queues library
- Blocks in Servers library
- Blocks in Entity Generators library
- Output Switch block with the **Store entity before switching** option selected

Nonstorage Blocks

These blocks permit an entity arrival but must output or destroy the entity at the same value of the simulation clock:

- Blocks in Attributes library
- Blocks in Routing library, except the Output Switch block with the **Store entity before switching** option selected
- Blocks in Gates library
- Blocks in the Entity Management library
- Blocks in Timing library
- Blocks in Probes library
- Blocks in SimEvents User-Defined Functions library
- Attribute Scope, X-Y Attribute Scope, and Instantaneous Entity Counting Scope blocks

- Entity Sink block
- Conn block
- Entity Departure Function-Call Generator block
- Entity-Based Function-Call Event Generator block

Blocks That Support Event-Based Input Signals

In this section...

“Computational Blocks” on page 14-53

“Sink Blocks” on page 14-54

“SimEvents Blocks” on page 14-55

“Other Blocks” on page 14-55

Computational Blocks

The following table lists blocks in the Simulink and Stateflow libraries that can operate directly on event-based input signals. If the block has an event-based input signal and a **Sample time** parameter in the block dialog box, you must set **Sample time** to -1 to indicate an inherited sample time.

Tip If the block of interest is not in the table but has an inherited or constant sample time, you can still use the block to perform a computation on an event-based signal. Connect the event-based signal to an Atomic Subsystem or Function-Call Subsystem block, and insert the computational block into the subsystem.

Block	Library
Add	Math Operations
Atomic Subsystem	Ports & Subsystems
Bias	Math Operations
Bus Creator	Signal Routing
Bus Selector	Signal Routing
Chart	Stateflow
Data Type Conversion	Signal Attributes
Demux	Signal Routing
Divide	Math Operations

Block	Library
Dot Product	Math Operations
Function-Call Split	Ports & Subsystems
MATLAB Function	User-Defined Functions
From	Signal Routing
Function-Call Subsystem	Ports & Subsystems
Gain	Math Operations
Goto	Signal Routing
Goto Tag Visibility	Signal Routing
Logical Operator	Logic and Bit Operations
Math Function	Math Operations
MinMax	Math Operations
Mux	Signal Routing
Product	Math Operations
Product of Elements	Math Operations
Reciprocal Sqrt	Math Operations
Relational Operator	Logic and Bit Operations
Sign	Math Operations
Slider Gain	Math Operations
Sqrt	Math Operations
Subsystem	Ports & Subsystems
Subtract	Math Operations
Sum	Math Operations
Unary Minus	Math Operations

Sink Blocks

These blocks in the Simulink Sinks library can display, report on, or terminate event-based input signals.

Block	Library
Display	Sinks
Floating Scope	Sinks
Scope	Sinks
Terminator	Sinks
To Workspace	Sinks

SimEvents Blocks

Except for the Timed to Event Signal and Timed to Event Function-Call blocks, all blocks in the SimEvents libraries that have signal input ports require the input signals to be event-based signals rather than time-based signals.

Other Blocks

A block that is not in one of the SimEvents libraries, Simulink libraries, or Stateflow library cannot directly operate on event-based signals. However, if the block has an inherited or constant sample time, you can connect the event-based signal to an Atomic Subsystem or Function-Call Subsystem block, and then insert the block that is not in one of the libraries into the subsystem.

Migrating SimEvents Models

- “Introduction” on page 15-2
- “Legacy Behavior in SimEvents Models” on page 15-3
- “Checking Your Model for Legacy Behavior” on page 15-13
- “Migration Using seupdate” on page 15-14
- “Before Migration” on page 15-15
- “Running seupdate” on page 15-17
- “Resolving Migration Failure” on page 15-19
- “After You Migrate” on page 15-21
- “Model Behavior Changes” on page 15-22
- “Migration Limitations” on page 15-30

Introduction

If your model was created in an earlier version of SimEvents, it does not have the benefit of feature updates and modeling syntax changes available in the latest version of the software.

Use the following information to:

- Learn about certain types of behavior in older versions of the software, but that are eliminated when you migrate your model to the latest version.
- Check your model for out-of-date behavior.
- Learn about expected changes to the contents of your model before you migrate.
- Migrate your model successfully.
- Learn about behavior changes in your model after you migrate.

Legacy Behavior in SimEvents Models

In this section...

“Visual Appearance of Legacy SimEvents Blocks” on page 15-3

“Multifiring Behavior” on page 15-4

“Entities Arriving at Empty Queue Blocks” on page 15-10

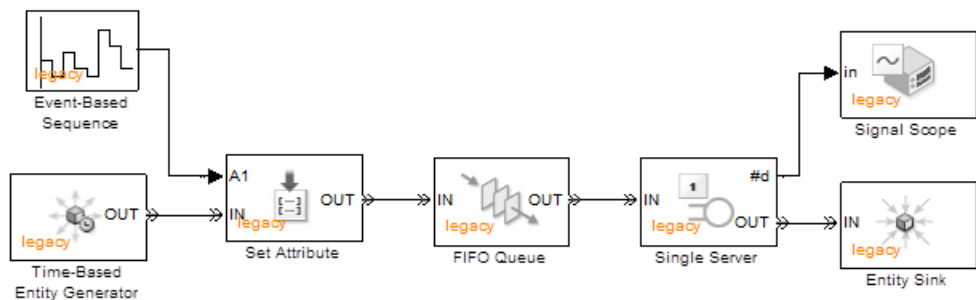
“No Events at Time 0” on page 15-11

The following sections describe certain types of behavior that you might see in models created in older versions of the software, but that are eliminated when you migrate your model to the latest version. Such behavior is called *legacy behavior*.

Visual Appearance of Legacy SimEvents Blocks

If your model contains blocks from a release prior to R2011b, the software visually labels these blocks so that when you view the model in the model editor, you can easily identify them. The software marks such blocks by placing an orange label, **legacy** in the lower left-hand corner of the block. This visual marker helps you assess the need to migrate your model to the latest version of SimEvents.

The model in the following graphic was created in a release earlier than R2011b, so each block is marked with the orange **legacy** label.



If your model contains blocks with the orange **legacy** label, it continues to function as originally designed, but does not have the benefit of feature updates and modeling syntax changes available in the latest version of SimEvents. To resolve this discrepancy, and to ensure that all blocks in your model are updated to the latest version, use the `seupdate` migration utility to migrate your model. For more information see “Migration Using `seupdate`” on page 15-14.

Multifiring Behavior

The configuration parameter **Prevent duplicate events on multiport blocks and branched signals** is introduced in R2012a. Once you use the `seupdate` function to migrate your model to the latest version of the software, you can select this configuration parameter. For more information, see “Migration Using `seupdate`” on page 15-14.

Prevent duplicate events on multiport blocks and branched signals eliminates a behavior called *multifiring* that is an implicit result of the way that the software executes particular modeling configurations. When a model shows multifiring behavior, the software might execute— or *fire*—a particular block more than once in response to a single discrete event. This multiple execution behavior causes duplication of subsequent events in your simulation

Multifiring behavior arises in a model that does not have this configuration parameter enabled and contains any of the following configurations:

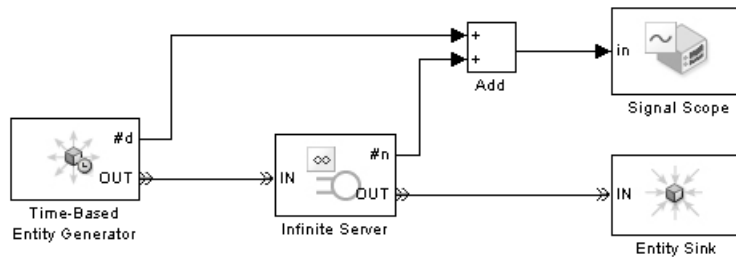
- Block with multiple inputs (multiport block) where two or more of the inputs separately update in response to the same event. If a multiport block has inputs that each update in response to different events, you do not see multifiring behavior.
- Branched signal that updates in response to a single discrete event, with two or more of the branches connected as inputs to the same multiport block.
- Function call with multiple iterations.

Note Although the name of the configuration parameter **Prevent duplicate events on multiport blocks and branched signals** refers to only the first two configuration types in the preceding list, this parameter also eliminates multifiring behavior in the third type of configuration; function calls with multiple iterations.

The following sections describe multifiring behavior that arises from the configuration types in the preceding list.

Multifiring on Multiport Blocks

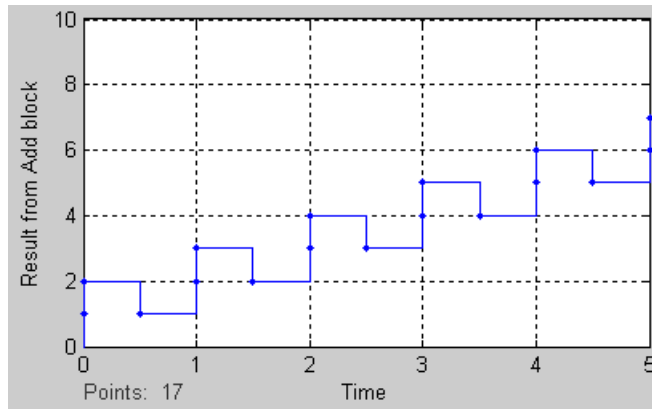
In the following model, the Add block is a multiport block. When a single entity moves between the generator block and the server block, each input of the Add block separately updates, resulting in multifiring behavior.



In this model, when an entity departs the Time-Based Entity Generator block and arrives at the Infinite Server block, the **#d** and **#n** outputs, respectively, of those blocks separately update. Multifiring behavior occurs because although just one entity moves from the generator block to the server block, the Add executes each time one of its input signals updates.

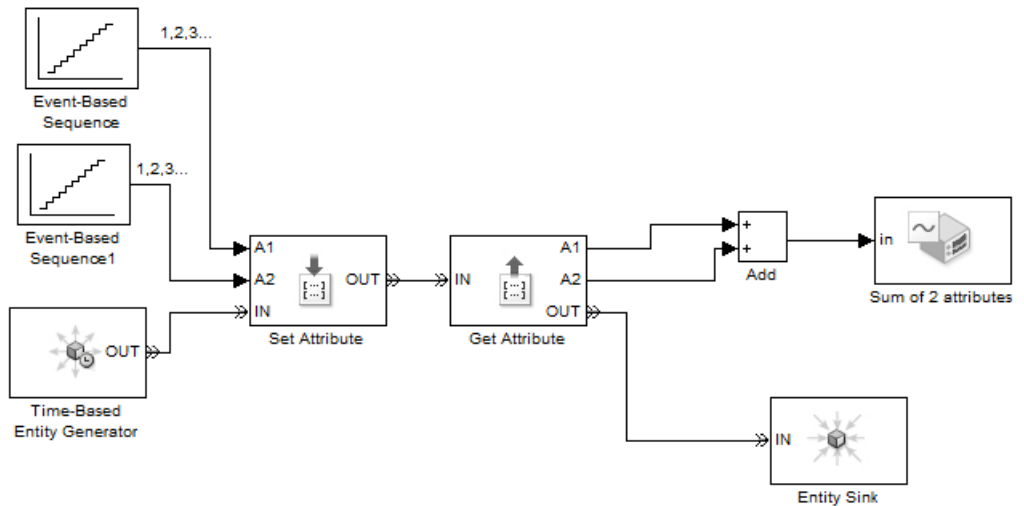
When the Add block first executes, the value of its second input—the **#n** output of the Infinite Server block—is still awaiting update. When this second input signal updates, the Add block executes again. The second time the block executes, all signal values are up to date and the block produces a new result. You can consider the first result to be an intermediate value that you might have no use for in your simulation.

The following output of the Signal Scope block shows this multifiring behavior, with two results displayed on the scope at simulation time 0, 1, 2, 3, etc. You can identify the intermediate results in your simulation as the points that a scope plot does not join to points at subsequent time steps. In this case, the intermediate result is the higher of the two values shown at simulation time 0, 1, 2, 3, etc.



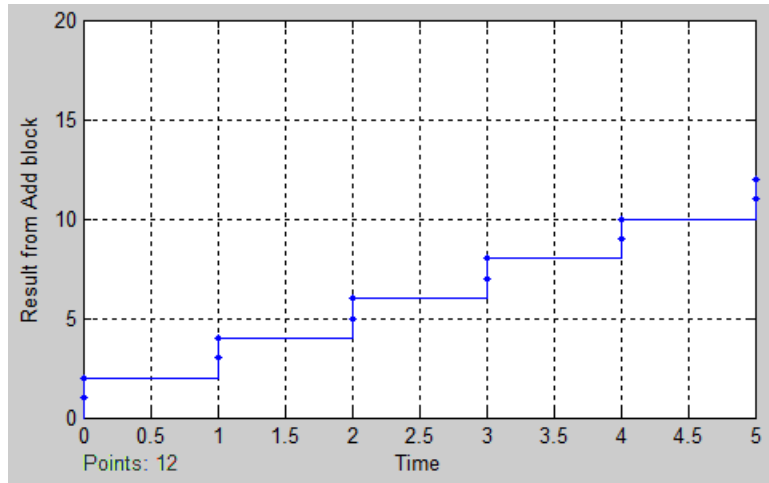
In the preceding model, another entity movement happens during the simulation that does not result in multifiring behavior. When an entity moves from the Single Server block to the Entity Sink block, the Add block executes once. Because no signal value is awaiting update when the Add block executes, the resulting output is final. The single result that the Add block produces is shown on the scope output at simulation time 0.5, 1.5, 2.5, etc.

The next model also exhibits multifiring behavior. Two output signals of the Get Attribute block are connected as inputs to the Add block. You might expect that when an entity departs the Get Attribute block, the Add block executes and calculates the sum of the two attribute values, **A1** and **A2**. However, for a single entity departure, each input of the Add block separately updates, resulting in multifiring behavior.



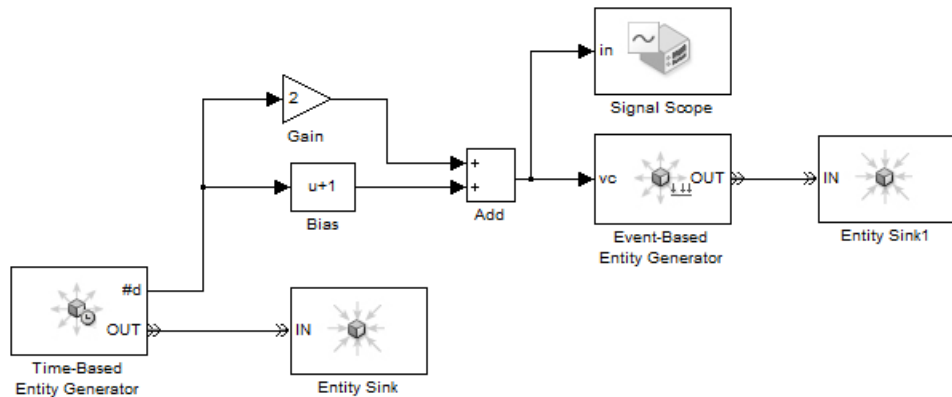
The first time that the Add block executes, for one of the input signals, it uses a value that is awaiting update. When this second input signal updates, the Add block executes again, and produces an updated result at the same simulation time.

The following output from the Signal Scope block shows the multifiring behavior, with two results at simulation time 0, 1, 2, 3, etc.



Multifiring on Branched Signals

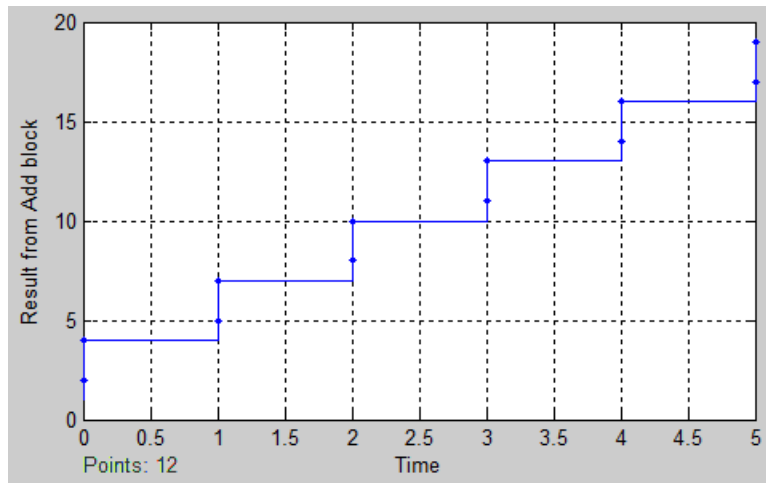
This model shows a configuration using a branched signal, with each branch connected as an input to the same multiport block, the Add block. This configuration results in multifiring behavior.



When an entity departs the Time-Based Entity Generator block, the value of the **#d** parameter of the block increments by a value of 1. The **#d** signal branches along two separate paths, with the simulation performing a mathematical operation on each branch.

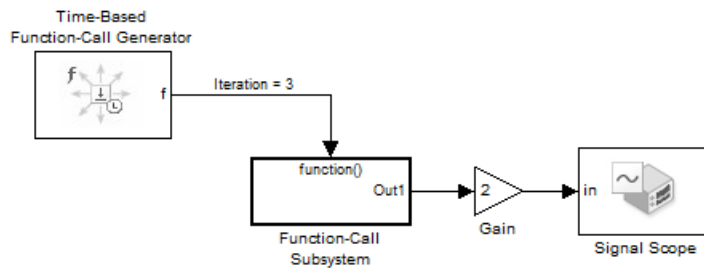
You might expect that once the simulation completes the mathematical operation on each branch, the Add block executes and calculates the sum of the two branches. Multifiring behavior occurs, however, because the Add block actually executes each time that one of its input signals updates. Therefore, the block produces two different results at the same simulation time.

The output from the Signal Scope block shows the multifiring behavior. Two results are displayed at simulation time 0, 1, 2, etc.



Multifiring when Using Function-Calls

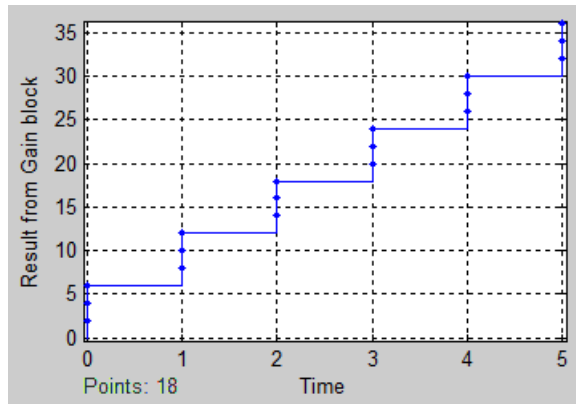
This model shows a configuration using a function call with multiple iterations that results in multifiring behavior.



You might expect that when the Time-Based Function-Call Generator block executes and makes a function call with multiple iterations, the Function-Call Subsystem block executes all iterations before updating its output signal, **Out1**, with a final value.

However, multifiring occurs because the simulation treats each function-call iteration as a separate event. Each time that an iteration completes, the value of the output signal **Out1** updates. This behavior causes three separate results at each time instant during the simulation.

The output from the Signal Scope block shows the multifiring behavior. Three separate results are displayed at simulation time 0, 1, 2, etc.



To eliminate multifiring behavior in your model:

- 1 Use the `seupdate` function to migrate all blocks in your model to the latest version. For more information, see “Migration Using `seupdate`” on page 15-14.
- 2 In the updated model, select the configuration parameter **Prevent duplicate events on multiport blocks and branched signals**. For more information, see “Configuration Parameters”.

Entities Arriving at Empty Queue Blocks

In a model created in a version of SimEvents prior to 4.0 (R2011b), when an entity arrives at an empty queue block, the software behaves as follows.

- If the output port of the queue block is not blocked:
 - The entity immediately advances.
 - The **#n** output statistic of the queue block, which is the number of entities in the queue, does not change in value.
- If the output port is blocked:
 - The entity is held in the queue.
 - The **#n** output of the queue block, changes from 0 to 1.

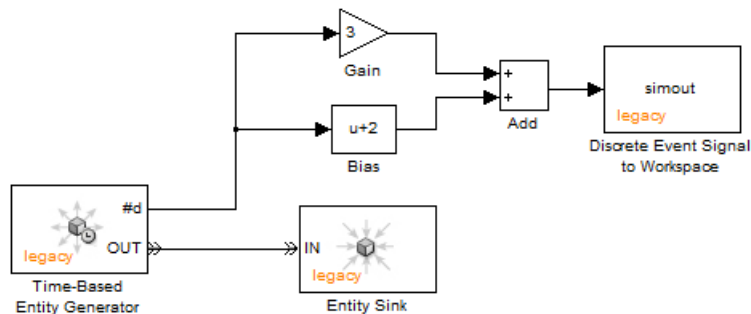
When you use the `seupdate` function to migrate your model to the latest version of the software, you see changes in this execution behavior of queue blocks.

For more information, see:

- “Migration Using `seupdate`” on page 15-14
- “Changed Behavior of Entities Arriving at Empty Queue Blocks” on page 15-23

No Events at Time 0

In versions of SimEvents prior to 4.0 (R2011b), the software did not have events at time 0. The following model—created in an older version of SimEvents—exhibits this behavior.



When you run this model, the simulation produces no events at time 0. This result is shown by the output of the Discrete Event Signal to Workspace block:

```
>> answers = [simout.time, simout.signals.values]

answers =

     1     6
     2    10
     3    14
     4    18
     5    22
     6    26
     7    30
     8    34
     9    38
    10    42
```

If you use the `seupdate` function to migrate your model to the latest version of the software, you see additional events at time 0.

For more information, see:

- “Migration Using `seupdate`” on page 15-14
- “Events at Time 0” on page 15-27

Checking Your Model for Legacy Behavior

Model Advisor Checks for SimEvents

You can use the Simulink Model Advisor to check the SimEvents portion of your model for the following legacy behavior:

- SimEvents blocks from releases prior to R2011b (legacy blocks).
- Implicit event duplication caused by multifiring.

If Model Advisor detects legacy blocks in your model, it includes an option to fix this condition by using `seupdate` to migrate the model. To learn more about the migration process using `seupdate` before proceeding with this option, see:

- “Migration Using `seupdate`” on page 15-14
- “Before Migration” on page 15-15

To learn more about the Model Advisor checks for SimEvents, see “Model Advisor Checks”.

For information on using Model Advisor, see *Consulting Model Advisor* in the Simulink documentation.

Migration Using seupdate

The seupdate function migrates all blocks in your model to the latest version. To migrate any Simulink components in your model, the function also runs the Simulink slupdate function.

For more information, see:

- “Before Migration” on page 15-15
- “Running seupdate” on page 15-17

Before Migration

In this section...
“Preparing to Migrate a Model” on page 15-15
“Expected Changes to Model Contents” on page 15-15

Preparing to Migrate a Model

The `seupdate` function overwrites the model that it is updating. Before beginning the update, the function completely backs up the model and associated custom libraries. The software stores this backup in a new folder, in the same directory as the model. The new folder uses the naming convention `seupdate_for_sys`.

Before you start the migration:

- Ensure that you have no unsaved changes in the model.
- Ensure that you have write permission for the model and associated libraries.
- Ensure that the model compiles without warnings or errors.

Expected Changes to Model Contents

SimEvents models cannot simultaneously contain blocks from releases prior to R2011b and the current release. After you update your model, you cannot use blocks from previously released SimEvents libraries.

The `seupdate` command:

- Runs `supdate` to perform all upgrades that are related to Simulink in the models. You do not need to separately perform `supdate` on any of the models that you update using `seupdate`.
- Replaces SimEvents and Simulink blocks with the latest versions.
- Replaces instances of the Discrete Event Subsystem block with the Simulink Atomic Subsystem block.

Note `seupdate` might convert a Discrete Event Subsystem block to a virtual subsystem that contains an Atomic Subsystem block and possibly Event Filter or gateway blocks. In these cases, the name of the virtual subsystem might still contain the string, `Discrete Event Subsystem`. However, the underlying subsystem has been converted.

- Inserts new gateway blocks to delineate the boundaries between event-based portions and time-based portions of the model.

For information on using the `seupdate` function to migrate your model, see “Running `seupdate`” on page 15-17.

Running seupdate

Use the following procedure to migrate all blocks in a model and its associated libraries to the latest version from the SimEvents library.

- 1 Run the seupdate function for the model that you want to migrate.

```
seupdate('doc_dd1')
```

- 2 Review the items under the heading `Before you update your model`.

```
Before you update your model:
```

1. Verify that the model compiles without any errors.
2. Learn how SEUPDATE changes your model (enter 'help' at the prompt below)

```
Update model 'doc_dd1'? ([y]/n/help):
```

If your model contains legacy queue blocks, the migration process identifies these blocks in your model and displays some additional output under the heading `Before you update your model`. This additional output provides you with instructions to check for potential changes in the results produced by queue blocks if you migrate your model. For more information, see “Migration of Model Containing Queue Blocks Using seupdate”.

- 3 Enter a value at the prompt. Valid responses are `y` to overwrite the model and associate libraries, `n` to cancel model migration without any changes, or `help` to read about how seupdate changes your model.

```
Model      : doc_dd1
Libraries  : --
```

```
Before you update your model:
```

1. Verify that the model compiles without any errors.
2. Learn how SEUPDATE changes your model (enter 'help' at the prompt below)

```
Update model 'doc_dd1'? ([y]/n/help):y
```

- 4 If migration of your model is successful, the software displays the status `UPDATE: Completed`.

UPDATE : Completed

Note If conditions exist in your model that cause the migration process to fail, the software displays the status UPDATE: Failed. For more information, see “Resolving Migration Failure” on page 15-19.

5 Review the report that the migration utility generates.

```
UPDATE REPORT -----  
  
Next steps:  
  
1. If the update failed, review any "ACTION NEEDED" items in this report.  
2. If the update completed, learn how the update might have changed your model and libraries (read)  
3. Read the more detailed update report (open)  
  
END UPDATE REPORT -----
```

Resolving Migration Failure

Your model might contain conditions that cause migration using `seupdate` to fail. Use the following procedure to troubleshoot a migration failure.

- 1 The software notifies you that the update process was unsuccessful.

```
Update model 'ex_eventcount'? ([y]/n/help):y

UPDATE : Checking file permissions ...
UPDATE : Backing up files in directory 'seupdate_for_ex_eventcount' ...
UPDATE : Replacing blocks with new versions ..

UPDATE : Failed
```

- 2 Each condition in your model that causes the update process to fail is identified in the update report, under the heading **FAILED: ACTION NEEDED**. The software provides you with instructions to resolve each failure condition.

```
==> FAILED: ACTION NEEDED:
The following blocks cannot be updated because the number of attributes specified
in each block is zero. In SimEvents 4.0, the number of attributes specified in
the Set Attribute and Get Attribute blocks cannot be less than one. To resolve
this, either increase the number of attributes specified in each block to a value
greater than zero, or delete each block from your model. Rerun SEUPDATE.
- 'ex\_eventcount/Set Attribute'
```

- 3 When you finish reviewing and correcting all the **FAILED: ACTION NEEDED** items in the update report, rerun the `seupdate` function.
- 4 If you have corrected all the previous failure conditions, when you rerun the `seupdate` function, the model migration is successful. If not, repeat this procedure until the output displays **UPDATE: Completed**.

```
Update model 'ex_eventcount'? ([y]/n/help):y

UPDATE : Checking file permissions ...
UPDATE : Backing up files in directory 'seupdate_for_ex_eventcount' ...
UPDATE : Replacing blocks with new versions ...
UPDATE : Adding gateway blocks to convert between time-based and event-based signals ...
UPDATE : Completed

Files changed : ex_eventcount
```


After You Migrate

After running the `seupdate` function, you might notice changes in behavior of the model. The primary areas where you might notice changes are:

- Signal computations that the SimEvents software performed in a time-based manner might now be performed more accurately in an event-based manner. This behavior change might now cause changes in your model output.
- The SimEvents software now computes initial values of various signals in the event-based portions of your model more robustly at time 0 of your simulation. These initial values might differ from previous releases.
- Architectural changes that more clearly delineate time-based modeling from event-based modeling also affect the event propagation from one block to another at time 0.
- Changes in sorted order within signal computations might improve accuracy in hybrid time-based and event-based models.
- Updated queue blocks have improved updating of their **#n** statistic port.
- Gateway blocks convert bus signals to non-bus signals

To ensure that the converted model continues to behave as you want:

- 1** Run the model and evaluate the simulation results.
- 2** If you notice simulation result changes that you do not want, modify your upgraded model to get the desired results.

Model Behavior Changes

This section provides you with a more detailed description of some behavior changes that you might see when you migrate your model. Where applicable, a workaround is included.

For more information on the most significant behavior and syntax changes introduced in recent versions of SimEvents, see “Version 4.0 (R2011b) SimEvents Software”.

Eliminating Multifiring Behavior

Once you use the `seupdate` function to migrate your model to the latest version of the software, you can select the configuration parameter **Prevent duplicate events on multiport blocks and branched signals**. This parameter eliminates each type of multifiring behavior that is described in “Multifiring Behavior” on page 15-4. For more information on this configuration parameter, see “Prevent duplicate events on multiport blocks and branched signals”.

Time-Based Execution in Previous Model

In past releases, signal computation blocks placed outside of a discrete-event subsystem executed exclusively in a time-based manner, even when the signal was between two SimEvents blocks. In the current release, several signal computation blocks are enabled for event-based execution. Signal computations that in previous releases, were inadvertently placed in time-based execution might now execute correctly in an event-based manner. This change might lead to subtle changes in behavior of the overall model.

You can use the special port notation to help clearly identify any signal computations that occur in an event-based manner. Additionally, using gateway blocks, the conversion places any computation blocks that do not support event-based computation within time-based regions of the model. If such computations actually belong within event-based computation, consider placing them within atomic subsystems to ensure event-based execution. You can then remove the corresponding gateway blocks.

Algebraic Loops

If the model has a time-driven section or block that is not contained in a discrete-event subsystem, such as an IC block, `seupdate` inserts a gateway before and after that section or block. When you simulate the model, you might notice algebraic loop errors.

To remove algebraic loop errors do one of the following:

- Replace the time-driven block, such as the IC block, with its SimEvents equivalent. For example, replace the IC block with the SimEvents Initial Value block.

When you replace a time-driven block with a SimEvents block, remove the gateway blocks.

- Add a Simulink Memory or Delay block adjacent to the gateway block, outside the discrete event system. However, this action might cause a delay that you do not want. If so, use the third option.
- Remove the gateway blocks and move the remaining blocks into an atomic subsystem.

Changed Behavior of Entities Arriving at Empty Queue Blocks

In previous releases, when an entity arrived at an empty queue block, if possible, the entity advanced immediately and the `#n` output of the queue block did not change in value. Now, when an entity arrives at an empty queue block, the software behaves as follows.

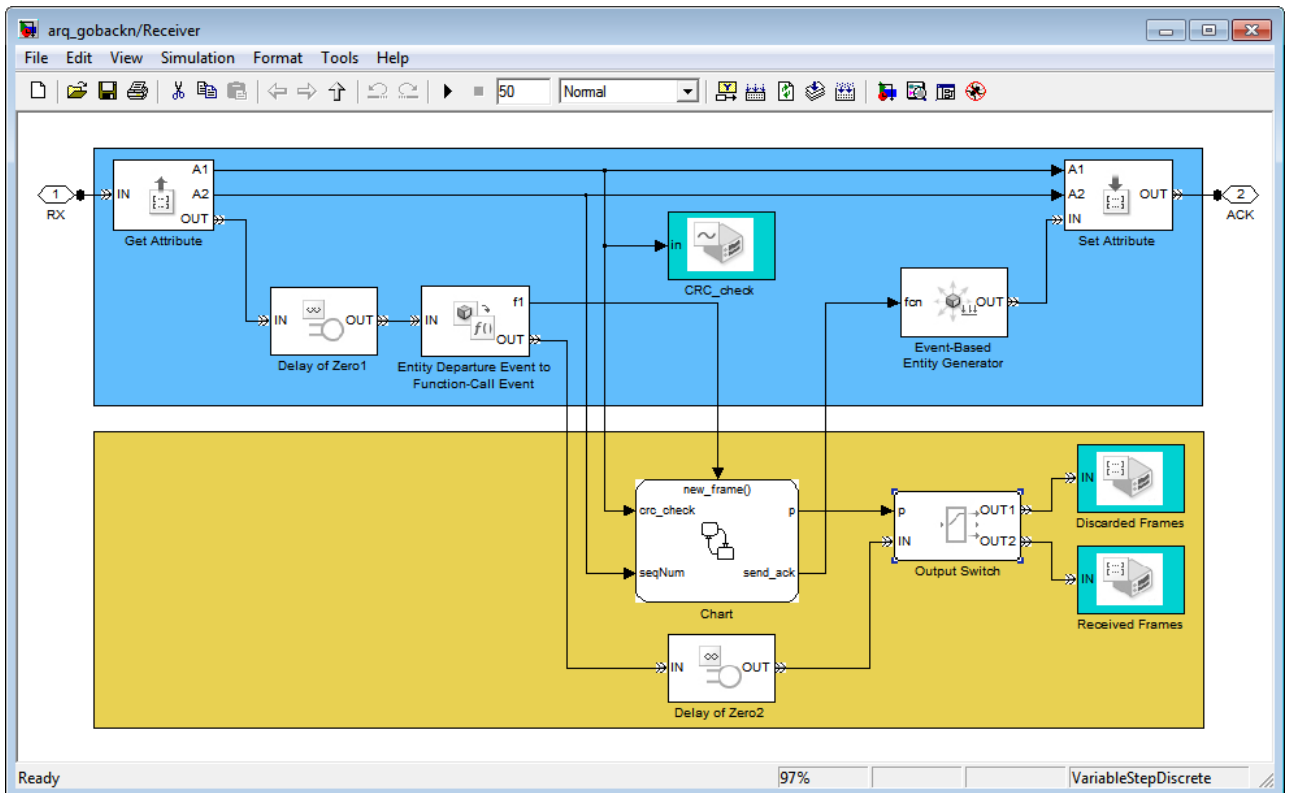
- If the output port of the queue block is not blocked:
 - The `#n` output of the queue block, which is the number of entities in the queue, changes from 0 to 1.
 - The entity immediately advances.
 - When the entity depart the block, the `#n` output changes from 1 back to 0.
- If the output port is blocked:
 - The entity is held in the queue.
 - The `#n` output of the queue block changes from 0 to 1.

Initial Values

In the current release, initial value changes might cause changes in simulation behavior. You might notice these differences in the following cases.

p Port Values

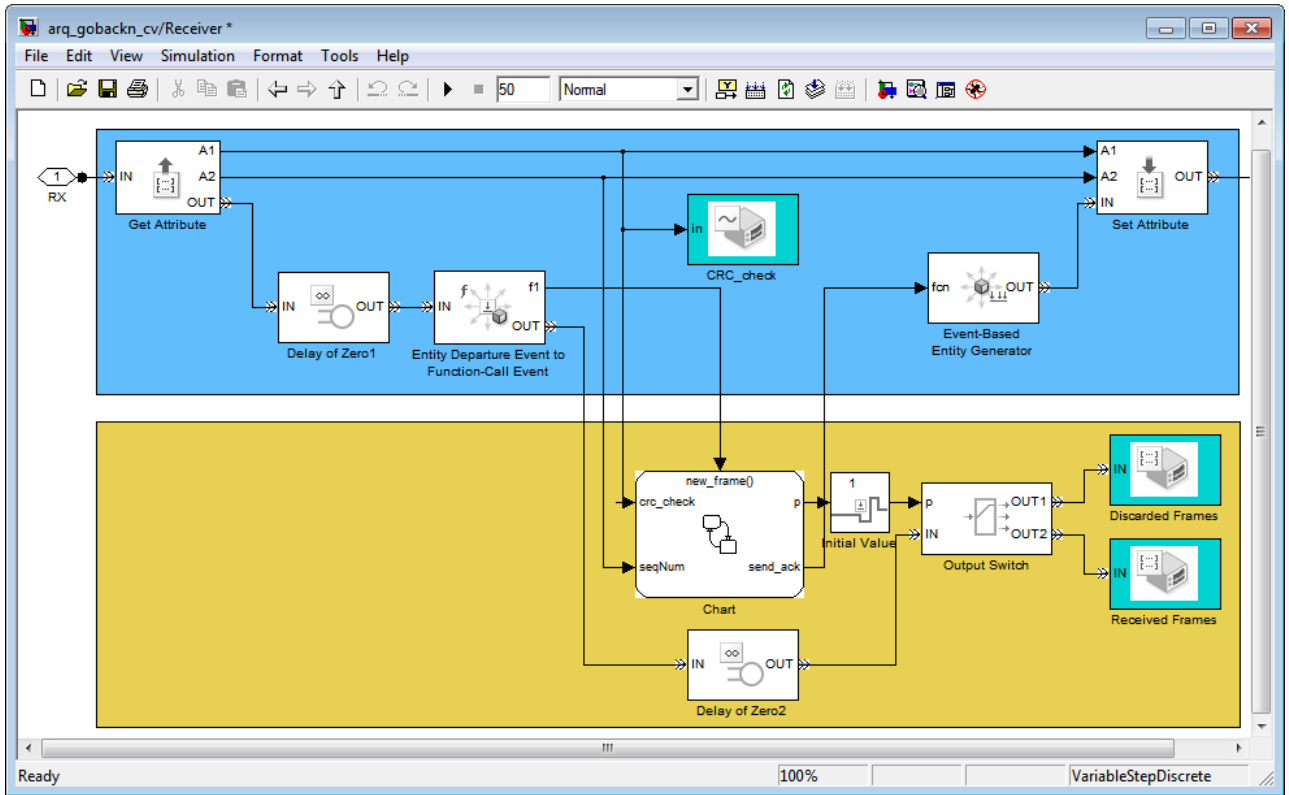
In previous releases, SimEvents models required that you inserted Stateflow charts in discrete event subsystems. If you did not insert a Stateflow chart in a discrete event subsystem, at time 0, that block had a hit with output at time 0, causing the p port to be nonzero. If the chart p port was connected to the input p port of an Output Switch block, the software did not emit an error message because the chart p port was nonzero. You did not need to explicitly specify a value for the p port of the Output Switch block.



Now, the SimEvents software requires you to specify a valid value for the p port of the Output Switch block. When you migrate the model to the current release, and you do not specify a valid value for the Output Switch p port, the software assigns a default value of 0 to the p port. This default assignment causes an error message.

Workaround. To restore the original block behavior, perform one of the following:

- Set an initial value for the block whose port is connected to the p port of the chart block, such as the Output Switch or Signal Latch block.
- Between the chart block and the block whose port is connected to the p port of the chart block, insert an Initial Value block. Configure it to output a legitimate initial value, such as 1.



Changes in Triggering

The changes in initial value calculation might cause value change (vc) ports and trigger (tr) ports to cause increased downstream execution at time 0. To avoid such execution, use Initial Value blocks to specify initial conditions for signals.

Bus Signals Converted to Non-Bus Signals

When seupdate adds a gateway block to the output port of a block that produces a bus signal, if possible, the gateway block converts the bus signal to a non-bus signal. In some cases, this conversion might not be possible. In such cases, because the bus signal remains in your updated model, when the software compiles the model, you see an error.

In the updated model, to connect a bus signal to a gateway block without seeing an error, select the configuration parameter, **Prevent duplicate events on multiport blocks and branched signals**.

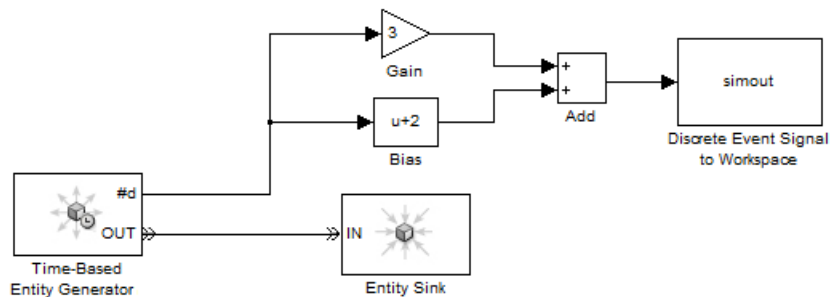
For more information about:

- Using bus signals in a discrete-event system, see *Working with Bus Signals*.
- The configuration parameter, **Prevent duplicate events on multiport blocks and branched signals**, see “Configuration Parameters”.

Events at Time 0

In previous versions of SimEvents, the software did not have events at time 0. Your converted model might now have events at this simulation time. In the updated model, the number of events at time 0 also depends on whether or not you select the configuration parameter **Prevent duplicate events on multiport blocks and branched signals**.

The following graphic shows the updated version of the model introduced in the section, “No Events at Time 0” on page 15-11.



In this updated version of the model, the software now produces events at time 0.

```
>> answers = [simout.time, simout.signals.values]
```

```
answers =
```

```
    0     2
    0     2
    1     3
    1     6
    2     7
    2    10
    3    11
    3    14
    4    15
    4    18
    5    19
    5    22
    6    23
    6    26
    7    27
    7    30
    8    31
    8    34
    9    35
    9    38
   10    39
   10    42
```

Because the Add block is a multiport block, at each time instant, the block exhibits multifiring behavior that causes it to produce two different results. At each simulation time, you can consider the first of these results to be an intermediate value that you might not want. For more information, see “Multifiring Behavior” on page 15-4.

If you select the configuration parameter **Prevent duplicate events on multiport blocks and branched signals**, the multifiring behavior is eliminated. The software produces a single result at each simulation time, including time 0 .


```
>> answers = [simout.time, simout.signals.values]
```

```
answers =
```

```
    0     2  
    1     6  
    2    10  
    3    14  
    4    18  
    5    22  
    6    26  
    7    30  
    8    34  
    9    38  
   10    42
```

When you select **Prevent duplicate events on multiport blocks and branched signals**, the software produces events at the following blocks:

- Atomic Subsystem
- MATLAB® Function
- Stateflow® Chart

Migration Limitations

In this section...

“Position of Gateway Blocks” on page 15-30

“Cascaded Mux or Bus Blocks” on page 15-30

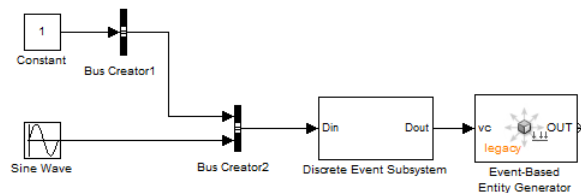
Position of Gateway Blocks

During migration of your model, the software inserts gateway blocks where signals transition from the event-based domain to the time-based domain, or vice-versa. To insert these gateway blocks, the software inspects existing lines and connections in your model. Based on the position of these existing connections, the software calculates the best position at which to insert the required gateway blocks. However, the software might not always place gateway blocks in the ideal position, or if the software inserts multiple gateway blocks, the gateway blocks might overlap.

Once the seupdate migration process is complete, review the list of new gateway blocks that the software includes in the update report. In the model editor, inspect the positioning of these new gateway blocks. If any blocks are not positioned as you expect, reposition them manually and save your model.

Cascaded Mux or Bus Blocks

In the following model fragment, two Bus Creator blocks are connected in a cascade configuration. In this case, if you use seupdate to migrate the model, the migration fails.



Migration of a model using seupdate fails if the model contains cascaded configurations of the following Simulink blocks:

- Bus Creator
- Mux

In this example, when migration of the model fails, the command window displays the following output:

```
==> FAILED: ACTION NEEDED:  
Could not complete the update because this model has the following cascaded  
Mux or Bus Creator blocks. The model and associated libraries may be in a  
partially updated state. To resolve this, reconfigure your model to remove  
cascaded Mux or Bus Creator blocks and rerun SEUPDATE.  
- ex\_CascadeBusCreator/Bus\_Creator2  
- ex\_CascadeBusCreator/Bus\_Creator1
```

If migration fails due to cascaded mux blocks, the software displays similar output. The output notifies you that as a result of the failed migration, the model might be in a partially updated state. To resolve the failure condition and successfully update the model, before rerunning `seupdate`, remove any cascaded bus or mux configurations from your model.

For information on using bus signals in SimEvents models, see [Working with Bus Signals](#).

Examples

Use this list to find examples in the documentation.

Attributes of Entities

“Example: Setting Attributes” on page 1-8

“Example: Attaching Data Instead of Branching a Signal” on page 1-10

Counting Entities

“Example: Counting Simultaneous Departures from a Server” on page 1-20

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

Queuing Systems

“Example: Event Calendar Usage for a Queue-Server Model” on page 2-7

“Example: LIFO Queue Waiting Time” on page 5-2

“Example: Serving Preferred Customers First” on page 5-7

“Example: Preemption by High-Priority Entities” on page 5-11

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

Working with Events

“Example: Observing Service Completions” on page 2-19

“Example: Detecting Collisions by Comparing Events” on page 2-22

“Opening a Gate Upon Random Events” on page 2-27

“Example: Counting Events from Multiple Sources” on page 2-32

“Example: Choices of Values for Event Priorities” on page 3-11

“Example: Effects of Specifying Event Priorities” on page 3-25

Working with Signals

“Example: Creating a Random Signal for Switching” on page 4-6

“Example: Resampling a Signal Based on Events” on page 4-15

“Example: Sending Queue Length to the Workspace” on page 4-17

Server States

“Example: Failure and Repair of a Server” on page 5-22

“Example: Adding a Warmup Phase” on page 5-24

Routing Entities

“Example: Cascaded Switches with Skewed Distribution” on page 6-9

“Example: Compound Switching Logic” on page 6-10

“Example: Choosing the Shortest Queue” on page 6-13

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 7-11

Timeouts

“Basic Example Using Timeouts” on page 8-3

“Defining Entity Paths on Which Timeouts Apply” on page 8-7

“Example: Rerouting Timed-Out Entities to Expedite Handling” on page 8-11

“Example: Limiting the Time Until Service Completion” on page 8-13

Troubleshooting

“Example: Plotting Entity Departures to Verify Timing” on page 10-13

“Example: Plotting Event Counts to Check for Simultaneity” on page 10-15

“Example: Faulty Logic in Feedback Loop” on page 13-84

“Example: Deadlock Resulting from Loop in Entity Path” on page 13-85

“Example: Invalid Connection of Event-Based Random Number Generator”
on page 13-88

“Example: Sequence of Departures and Statistical Updates” on page 14-40

Statistics

“Example: Fraction of Dropped Messages” on page 11-8

“Example: Computing a Time Average of a Signal” on page 11-9

“Example: Resetting an Average Periodically” on page 11-12

Timers

“Basic Example Using Timer Blocks” on page 11-19

“Timing Multiple Entity Paths with One Timer” on page 11-21

“Timing Multiple Processes Independently” on page 11-22

A

- arbitrary event sequence 3-9
 - troubleshooting 13-80
- ARQ
 - Stateflow charts 12-4
- attributes of entities
 - combining 1-24
 - MATLAB functions 1-12
 - permitted values 1-34
 - plots 10-2
 - reading values 1-17
 - setting values 1-6
 - subsystem for manipulating 1-15
 - usage 1-6
- autoscaling 10-10
- averaging signals
 - over samples 11-12
 - over time 11-9
- axis limits 10-7

B

- block breakpoints 13-57
- block identifiers 13-19
 - obtaining 13-38
- block-to-block interactions 14-30
- breakpoint identifiers 13-19
- breakpoints
 - clearing (removing) 13-64
 - disabling 13-64
 - discrete-event simulation 13-57
 - enabling 13-65
 - listing 13-61
 - running simulation until 13-63

C

- caching 10-8
- cancelation messages in debugger 13-13
- cascading switch blocks

- random 6-9
- combining entities 1-24
- combining events 2-31
- conditional events 2-36
- copying entities 1-32
- counter reset events 2-2
- counting entities
 - cumulative 1-19
 - instantaneous 1-19
 - reset 1-21
 - storing in attribute 1-23
- counting events 10-2
 - simultaneity 10-15

D

- data history 10-8
- data types 4-3
- debugger
 - discrete-event simulations 13-3
 - identifiers 13-19
 - indentation 13-21
 - sedb package 13-7
 - sedebg prompt 13-7
 - simulation log 13-8
 - starting SimEvents 13-5
 - startup commands 13-55
 - stepping in discrete-event simulation 13-27
 - stopping 13-25
- debugging
 - discrete-event simulations 13-2
- delayed restart events 2-2
- delays
 - signal updates 13-98
- dependent operations 13-21
- detail settings 13-47
- detection messages in debugger 13-15
- discrete state plots 10-6
- discrete-event plots 10-6
 - customizing 10-10

- exporting 10-11
- troubleshooting using 10-12

dropped messages 11-8

E

enabled gates 7-4

entities

- combining 1-24
- counting 1-19
- event-based generation 1-2
- replicating 1-32
- synchronizing 1-25
- timeouts 8-1

entity advancement events 2-2

entity collisions 2-22

entity data

- combining 1-24
- MATLAB functions 1-12
- permitted values 1-34
- plots 10-2
- reading values 1-17
- setting values 1-6
- subsystem for manipulating 1-15
- usage 1-6

entity destruction events 2-2

entity generation

- event-based 1-2
- vector of times 1-4

entity generation events 2-2

entity identifiers 13-19

- obtaining 13-38

entity messages in debugger 13-16

entity paths

- timeouts 8-7

entity request events 2-2

equal event priorities 3-9

- troubleshooting 13-80

event breakpoints 13-57

event calendar 14-9

- displaying 13-46
- displaying in simulation log 13-10
- events on 14-10
- events on/off 3-25
- example 2-7
- numerical/system-level priority 3-25

event identifiers 13-19

- obtaining 13-38

event-based sequences 4-7

event-based signals

- data sets 4-7
- deferring reactions 14-17
- description 4-2
- feedback loops 13-84
- initial values 4-14
- latency 13-81
- manipulating 4-14
- MATLAB® workspace 4-17
- random 4-4
- resampling 4-15
- resolving updates 14-14
- update sequence 14-45

events

- conditionalizing 2-36
- generating 2-25
- manipulating 2-29
- numerical/system-level priority 3-25
- observing 2-15
- on/off event calendar 3-25
- priorities 3-8
- reacting to signal updates 14-17
- resolving signal updates 14-14
- sequence 14-9
- modeling approaches 3-7
- simultaneous 3-2
- supported types 2-2
- translating 2-34
- troubleshooting 13-80
- union 2-31

execution messages in debugger 13-13

F

- failure modeling
 - conditional events 2-37
 - gates 5-20
 - Stateflow 5-21
- feedback entity paths
 - troubleshooting 13-85
- feedback loops
 - troubleshooting 13-84
- first-order-hold plots 10-6
- function call events
 - discrete-event simulation 2-2
- function calls 2-5
 - generating 2-25

G

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

H

- head of queue events 2-2

I

- identifiers in debugger 13-19
 - obtaining 13-38
- indentation
 - simulation log 13-21
- independent operations 13-21
- independent replications 11-24
- initial port selection
 - switching based on signal 6-4

- initial seeds 11-24
- initial values 4-14
 - feedback loops 13-84
- input signals
 - deferring reactions to updates 14-17
 - resolving updates 14-14
- inspecting states
 - blocks 13-36
 - current point 13-32
 - entities 13-34
 - events 13-38
- instantaneous gate openings 7-6
- intergeneration times
 - event generation 2-27
- interleaved operations 14-39

L

- latency
 - interleaved operations 14-39
 - signal updates 13-98
 - switching based on signal 6-5
 - troubleshooting 13-81
- LIFO queues 5-2
- livelock prevention 14-24
- loops in entity paths 13-85

M

- M/M/5 queuing systems 5-18
- MATLAB functions
 - attributes of entities 1-12
- maximum number of events 14-25
- memory read events 2-2
- memory write events 2-2
- monitoring block messages in debugger 13-17
- monitoring ports 14-34

N

- n-servers 5-18

nonstorage blocks 14-51
notifying ports 14-33

O

Output Switch block
 signal-based routing 6-2

P

plots 10-1
 customizing 10-10
 troubleshooting using 10-12
 zero-duration values 4-21
point-to-point-delays 11-18
port selection events 2-2
ports
 monitoring 14-34
 notifying 14-33
 reactive 14-35
preemption events 2-2
preemption in servers 5-10
priorities, entity
 priority queues with preemptive servers 5-11
 queue sequence 5-4
 server preemption 5-10
priorities, event 3-8
 reacting to signal updates 14-17
 resolving signal updates 14-14
 troubleshooting 13-80

Q

queues
 choosing shortest using MATLAB code 6-13
 LIFO vs. FIFO 5-2
 preemptive servers 5-11
 priority 5-4
queuing systems
 M/M/5 5-18

R

race conditions 3-11
random
 signals 4-4
random event sequence 3-9
 troubleshooting 13-80
random numbers
 event-based 4-4
 switch selection 4-6
reactive ports 14-35
release events 2-2
release gates 7-6
reneging in queuing 8-3
replication of entities 1-32
replications 11-24
resampling signals 4-15
resetting averages 11-12
residual service time 5-11
resolving simultaneous updates of signals 14-14
resource allocation 1-24

S

sample means 11-12
sample time 4-2
sample time hit events 2-2
scatter plots 10-6
scheduling messages in debugger 13-13
scope blocks 10-1
 zero-duration values 4-21
sedb package 13-7
sedebg prompt 13-7
seed of random number generator
 varying results 11-24
server states 5-20
servers
 failure states 5-20
 multiple 5-18
 preemption 5-10
service completion events 2-2

- signal-based events
 - definition 2-4
 - signals
 - deferring reactions to updates 14-17
 - event-based 4-2
 - event-based data 4-7
 - random 4-4
 - resolving updates 14-14
 - simulation log 13-8
 - filtering 13-47
 - indentation 13-21
 - simultaneous events 3-2
 - event priorities 3-8
 - input signal updates 14-14
 - interleaved operations 14-39
 - modeling approaches 3-7
 - numerical/system-level priority 3-25
 - on/off event calendar 3-25
 - output signal updates 4-20
 - sequential processing 14-9
 - troubleshooting 13-80
 - unexpected 13-79
 - splitting entities 1-24
 - stack 5-2
 - stairstep plots 10-6
 - state inspection
 - blocks 13-36
 - current point 13-32
 - entities 13-34
 - events 13-38
 - Stateflow and SimEvents® 5-21
 - Stateflow® 12-3
 - statistics 11-2
 - accessing from blocks 11-5
 - custom 11-7
 - interleaved updates 14-39
 - latency 13-98
 - Statistics tab 11-5
 - stem plots 10-6
 - stepping in discrete-event simulation 13-27
 - stop time 11-30
 - storage blocks 14-51
 - changing processing sequence 14-41
 - looped entity paths 13-85
 - storage completion events 2-2
 - subsystem execution events 2-2
 - switching entity paths
 - based on signal 6-2
 - initial port selection 6-4
 - random with cascaded blocks 6-9
 - storing entity 6-5
 - synchronizing entities 7-6
- T**
- time averages 11-9
 - timed breakpoints 13-57
 - timed-out entities 8-10
 - routing 8-11
 - timeout events 2-2
 - timeout intervals 8-4
 - timeout paths 8-7
 - timeout tags 8-4
 - timeouts of entities 8-1
 - role in modeling 8-2
 - timer tags 11-20
 - timers 11-18
 - combining 1-26
 - independent 11-22
 - multiple entity paths 11-21
 - restarting 11-22
 - trigger events
 - discrete-event simulation 2-2
- V**
- value change events 2-2
 - visualization 10-1
 - zero-duration values 4-21

W

workspace 4-17

Z

zero-duration values

definition 4-20

MATLAB workspace 4-23

visualization 4-21

zero-order hold

plots 10-6