

Stateflow[®] and Stateflow[®] Coder

For Use with Simulink[®]

- Modeling
- Simulation
- Implementation

How to Contact The MathWorks:



www.mathworks.com Web
comp.soft-sys.matlab Newsgroup



support@mathworks.com 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. Mail
3 Apple Hill Drive
Natick, MA 01760-2098

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

Stateflow and Stateflow Coder User's Guide

© COPYRIGHT 1997 - 2005 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.

MATLAB, Simulink, Stateflow, Handle Graphics, Real-Time Workshop, and xPC TargetBox are registered trademarks of The MathWorks, Inc.

Other product or brand names are trademarks or registered trademarks of their respective holders.

Printing History:	May 1997	First printing	New
	January 1999	Second printing	Revised for Stateflow 2.0 (Release 11)
	September 2000	Third printing	Revised for Stateflow 4.0 (Release 12)
	June 2001	Fourth printing	Revised for Stateflow 4.1 (Release 12.1)
	July 2002	Fifth printing	Revised for Stateflow 5.0 (Release 13)
	January 2003	Online only	Revised for Stateflow 5.1 (Release 13SP1) (Renamed from Stateflow User's Guide)
	June 2004	Online only	Revised for Stateflow 6.0 (Release 14)
	October 2004	Online only	Revised for Stateflow 6.1 (Release 14SP1)
	March 2005	Online only	Revised for Stateflow 6.2 (Release 14SP2)

Stateflow Concepts

1

Finite State Machine Concepts	1-2
What Is a Finite State Machine?	1-2
Finite State Machine Representations	1-2
Stateflow Representations	1-3
Notation	1-3
Semantics	1-4
References	1-4
Stateflow and Simulink	1-5
The Simulink Model and the Stateflow Machine	1-5
Stateflow Data Dictionary of Objects	1-6
Defining Stateflow Interfaces to Simulink	1-7
Stateflow Diagram Objects	1-9
States	1-10
Transitions	1-12
Default Transitions	1-13
Events	1-14
Data	1-14
Conditions	1-15
History Junction	1-16
Actions	1-17
Connective Junctions	1-18
Stateflow Hierarchy of Objects	1-20
Exploring a Real-World Stateflow Application	1-22
Overview of the “fuel rate controller” Model	1-22
Control Logic of the “fuel rate controller” Model	1-26
Simulating the “fuel rate controller” Model	1-29

Overview of Stateflow Objects	2-2
Graphical Objects	2-2
Nongraphical Objects	2-3
The Stateflow Data Dictionary of Objects	2-4
States	2-5
What Is a State?	2-5
State Hierarchy	2-6
State Decomposition	2-7
State Labels	2-9
Transitions	2-12
What Is a Transition?	2-12
Transition Hierarchy	2-13
Transition Label Notation	2-14
Valid Transitions	2-16
Transition Connections	2-17
Transitions to and from Exclusive (OR) States	2-17
Transitions to and from Junctions	2-18
Transitions to and from Exclusive (OR) Superstates	2-18
Transitions to and from Substates	2-20
Self-Loop Transitions	2-21
Inner Transitions	2-21
Default Transitions	2-25
What Is a Default Transition?	2-25
Drawing Default Transitions	2-25
Labeling Default Transitions	2-26
Default Transition Examples	2-26
Connective Junctions	2-30
What Is a Connective Junction?	2-30
Flow Diagram Notation with Connective Junctions	2-31
History Junctions	2-37
What Is a History Junction?	2-37

History Junctions and Inner Transitions	2-38
Boxes	2-39
Graphical Functions	2-40

Stateflow Semantics

3

Executing an Event	3-3
Sources for Stateflow Events	3-3
Processing Events	3-4
Executing a Chart	3-6
Executing an Inactive Chart	3-6
Executing an Active Chart	3-6
Executing a Transition	3-7
Transition Flow Graph Types	3-7
Executing a Set of Flow Graphs	3-8
Transition Testing Order	3-10
Implicit Order Mode	3-10
Explicit Order Mode	3-14
Entering, Executing, and Exiting a State	3-20
Entering a State	3-20
Executing an Active State	3-22
Exiting an Active State	3-23
State Execution Example	3-23
Early Return Logic for Event Broadcasts	3-26
Semantic Examples	3-29

Transitions to and from Exclusive (OR) States Examples	3-31
Label Format for a State-to-State Transition Example	3-31
Transitioning from State to State with Events Example	3-32
Transitioning from a Substate to a Substate with Events Example	3-35
Condition Action Examples	3-37
Condition Action Example	3-37
Condition and Transition Actions Example	3-39
Condition Actions in For Loop Construct Example	3-40
Condition Actions to Broadcast Events to Parallel (AND) States Example	3-40
Cyclic Behavior to Avoid with Condition Actions Example . . .	3-41
Default Transition Examples	3-43
Default Transition in Exclusive (OR) Decomposition Example	3-43
Default Transition to a Junction Example	3-44
Default Transition and a History Junction Example	3-45
Labeled Default Transitions Example	3-47
Inner Transition Examples	3-49
Processing Events with an Inner Transition in an Exclusive (OR) State Example	3-49
Processing Events with an Inner Transition to a Connective Junction Example	3-53
Inner Transition to a History Junction Example	3-56
Connective Junction Examples	3-58
Label Format for Transition Segments Example	3-58
If-Then-Else Decision Construct Example	3-59
Self-Loop Transition Example	3-61
For Loop Construct Example	3-62
Flow Diagram Notation Example	3-63
Transitions from a Common Source to Multiple Destinations Example	3-65
Transitions from Multiple Sources to a Common Destination Example	3-67
Transitions from a Source to a Destination Based on a Common Event Example	3-68
Backtracking Behavior in Flow Graphs Example	3-69

Event Actions in a Superstate Example	3-71
Parallel (AND) State Examples	3-73
Event Broadcast State Action Example	3-73
Event Broadcast Transition Action with a Nested Event Broadcast Example	3-77
Event Broadcast Condition Action Example	3-81
Directed Event Broadcasting Examples	3-85
Directed Event Broadcast Using send Example	3-85
Directed Event Broadcasting Using Qualified Event Names Example	3-87

Creating Stateflow Chart Diagrams

4

Creating a Stateflow Chart	4-2
Creating States in Stateflow Charts	4-5
Creating a State	4-5
Moving and Resizing States	4-7
Creating Substates and Superstates	4-7
Grouping States	4-8
Specifying Substate Decomposition	4-8
Specifying Activation Order for Parallel States	4-9
Changing State Properties	4-9
Labeling States	4-11
Outputting State Activity to Simulink	4-14
Creating Transitions in Stateflow Charts	4-16
Creating a Transition	4-16
Creating Straight Transitions	4-17
Labeling Transitions	4-17
Moving Transitions	4-19
Changing Transition Arrowhead Size	4-20
Creating Self-Loop Transitions	4-21
Creating Default Transitions	4-22

Changing Transition Properties	4-22
Creating Flowcharts with Connective Junctions	4-25
Creating a Connective Junction	4-25
Changing Connective Junction Size	4-26
Changing Junction Properties	4-26
Using the Stateflow Editor	4-28
Stateflow Diagram Editor Window	4-29
Displaying the Context Menu for Objects	4-30
Specifying Colors and Fonts	4-31
Selecting and Deselecting Objects	4-34
Cutting and Pasting Objects	4-35
Copying Objects	4-35
Editing Object Labels	4-36
Viewing Stateflow Objects in the Model Explorer	4-36
Zooming a Diagram	4-37
Undoing and Redoing Editor Operations	4-39
Keyboard Shortcuts for Stateflow Diagrams	4-40

Extending Stateflow Chart Diagrams

5

Using History Junctions to Extend Charts and States	5-2
Creating a History Junction	5-2
Changing History Junction Size	5-3
Changing History Junction Properties	5-3
Using Subcharts to Extend Charts	5-5
What Is a Subchart?	5-5
Creating a Subchart	5-7
Manipulating Subcharts as Objects	5-9
Opening a Subchart	5-9
Editing a Subchart	5-10
Navigating Subcharts	5-10

Using Supertransitions to Extend Transitions	5-12
What Is a Supertransition?	5-12
Drawing a Supertransition Into a Subchart	5-13
Drawing a Supertransition Out of a Subchart	5-16
Labeling Supertransitions	5-18
Extending Transitions with Smart Behavior	5-19
Setting Smart Behavior in Transitions	5-19
What Smart Transitions Do	5-20
What Nonsmart Transitions Do	5-26
Using Functions to Extend Actions	5-29
Creating a Function	5-29
Programming Functions	5-33
Defining Function Data	5-39
Calling Functions in Stateflow	5-41
Exporting Functions	5-41
Specifying Function Properties	5-43
Using Boxes to Extend Chart Diagrams	5-45
Creating a State	5-45
Changing a State to a Box	5-46
Using Boxes in Stateflow	5-47
Using Notes to Extend Chart Diagrams	5-48
Creating Notes	5-48
Editing Existing Notes	5-49
Changing Note Font and Color	5-49
Moving Notes	5-50
Deleting Notes	5-50
Reporting Chart Diagrams	5-51
Printing and Reporting on Stateflow Charts	5-51
Generating a Model Report in Stateflow	5-53
Printing the Current Stateflow Diagram	5-56
Printing a Stateflow Book	5-56

Adding Events	6-3
Adding Events in the Stateflow Diagram Editor	6-3
Adding Events in the Model Explorer	6-4
 Setting Event Properties in the Event Dialog	 6-7
 Sharing Events with Simulink	 6-11
Defining Input Events	6-11
Associating Input Events with Control Signals	6-13
Defining Output Events	6-13
Associating an Output Event with an Output Port	6-15
 Sharing Events with Stateflow External Code	 6-16
Exporting Events to Stateflow External Code	6-16
Importing Events from Stateflow External Code	6-17
 Defining Implicit Events	 6-18
Referencing Implicit Events	6-18
Example of an Implicit Event	6-19
 Adding Data	 6-21
Adding Data to the Data Dictionary	6-21
Defining Temporary Data	6-24
 Setting Data Properties in the Data Dialog	 6-25
Setting General Data Properties	6-26
Setting Value Attribute Data Properties	6-30
Setting Description Data Properties	6-33
 Sharing Stateflow Data with Simulink and MATLAB	 6-35
Sharing Input and Output Data with Simulink	6-35
Resolving Signal Objects for Output Data	6-38
Bringing Simulink Parameters into Stateflow	6-39
Initializing Data from the MATLAB Base Workspace	6-40
Saving Data to the MATLAB Workspace	6-41

Sharing Data with Stateflow External Code	6-42
Exporting Data to External Stateflow Code	6-42
Importing Data from External Stateflow Code	6-43
Typing Stateflow Data	6-45
Selecting Stateflow Data Types	6-45
Inheriting Input and Output Data Type from Simulink	6-46
Typing Data by Using the Type of Other Data	6-48
Typing Data by Using an Alias	6-49
Sizing Stateflow Data	6-50
Inheriting Input and Output Data Size from Simulink	6-50
Sizing Data by Expression	6-51

Using Actions in Stateflow

7

Defining Action Types	7-3
State Action Types	7-3
Transition Action Types	7-7
Example of Action Type Execution	7-9
Using Operations in Actions	7-12
Binary and Bitwise Operations	7-12
Unary Operations	7-15
Unary Actions	7-15
Assignment Operations	7-15
Pointer and Address Operations	7-16
Type Cast Operations	7-17
Using Special Symbols in Actions	7-19
Comment Symbols	7-19
Hexadecimal Notation Symbols	7-19
Infinity Symbol, inf	7-20
Line Continuation Symbol,	7-20
Literal Code Symbol, \$	7-20
MATLAB Display Symbol, ;	7-20

Single-Precision Floating-Point Number Symbol, F	7-20
Time Symbol, t	7-21
Calling C Functions in Actions	7-22
Calling C Library Functions	7-22
Calling the abs Function	7-23
Calling min and max Functions	7-23
Calling User-Written C Code Functions	7-24
Using MATLAB Functions and Data in Actions	7-27
ml Namespace Operator	7-27
ml Function	7-28
ml Expressions	7-30
Which ml Should I Use?	7-31
ml Data Type	7-32
Inferring Return Size for ml Expressions	7-35
Using Data and Event Arguments in Actions	7-40
Using Arrays in Actions	7-42
Array Notation	7-42
Arrays and Custom Code	7-43
Broadcasting Events in Actions	7-44
Event Broadcasting	7-44
Directed Event Broadcasting	7-46
Using Temporal Logic in Actions	7-50
Rules for Using Temporal Logic Operators	7-50
after Temporal Logic Operator	7-52
before Temporal Logic Operator	7-53
at Temporal Logic Operator	7-54
every Temporal Logic Operator	7-55
Conditional and Event Notation	7-56
Temporal Logic Events	7-57
Using Bind Actions to Control Function-Call Subsystems	7-58
Binding a Function-Call Subsystem	7-58
Simulating a Bound Function-Call Subsystem	7-60

Using Stateflow Logic with Binding	7-64
Avoiding Muxed Trigger Events with Binding	7-68

Using Fixed-Point Data in Stateflow

8

What Is Fixed-Point Data?	8-2
Fixed-Point Numbers	8-2
Fixed-Point Operations	8-3
Using Fixed-Point Data in Stateflow	8-5
How Stateflow Defines Fixed-Point Data	8-5
Specifying Fixed-Point Data in Stateflow	8-7
Fixed-Point Context-Sensitive Constants	8-8
Tips for Using Fixed-Point Data in Stateflow	8-9
Overflow Detection for Fixed-Point Types	8-11
Sharing Fixed-Point Data with Simulink	8-12
Fixed-Point “Bang-Bang Control” Example	8-14
Opening the Fixed-Point “Bang-Bang Control” Example ...	8-14
Exploring the Fixed-Point “Bang-Bang Control” Example ...	8-15
Operations with Fixed-Point Data	8-18
Supported Operations with Fixed-Point Operands	8-18
Promotion Rules for Fixed-Point Operations	8-21
Assignment (=, :=) Operations	8-31
Fixed-Point Conversion Operations	8-36
Autoscaling of Stateflow Fixed-Point	8-37

Defining Interfaces to Simulink and MATLAB

9

Overview of Stateflow Interfaces	9-2
Stateflow Interfaces	9-2

Typical Tasks to Define Stateflow Interfaces	9-3
Where to Find More Information on Events and Data	9-3
Specifying Chart Properties	9-4
Setting Properties for Individual Charts	9-4
Setting Properties for All Charts in the Model	9-8
Setting the Stateflow Block Update Method	9-11
Implementing Simulink Update Interfaces	9-12
Defining a Triggered Stateflow Block	9-12
Defining a Sampled Stateflow Block	9-13
Defining an Inherited Stateflow Block	9-14
Defining a Continuous Stateflow Block	9-15
Defining Function Call Output Events	9-17
Defining Edge-Triggered Output Events	9-20
Creating Chart Libraries	9-23
MATLAB Workspace Interfaces	9-24
Examining the MATLAB Workspace in MATLAB	9-24
Interfacing the MATLAB Workspace in Stateflow	9-24
Interface to External Sources	9-25
Exported Events	9-25
Imported Events	9-27
Exported Data	9-29
Imported Data	9-31

Truth Table Functions

10

What Is a Truth Table?	10-2
Building a Simulink Model with a Stateflow Truth Table	10-4
Creating a Simulink Model	10-5
Creating a Stateflow Truth Table	10-8

Calling a Truth Table in a Stateflow Action	10-11
Creating Truth Table Data in Stateflow and Simulink	10-14
Programming a Truth Table	10-19
Opening a Truth Table for Editing	10-19
Entering Truth Table Conditions	10-20
Entering Truth Table Decisions	10-23
Entering Truth Table Actions	10-26
Assigning Truth Table Actions to Decisions	10-29
Adding Initial and Final Actions	10-31
Debugging a Truth Table	10-34
Checking Truth Tables for Errors	10-34
Specifying a Breakpoint for the Call to a Truth Table	10-35
Debugging a Truth Table During Simulation	10-36
Options for Assigning Actions to Decisions in Truth Tables	10-43
Truth Table Editor Operations	10-47
Truth Table Editor Reference	10-47
Searching and Replacing in Truth Tables	10-52
Using Row and Column Tooltip Identifiers	10-54
Correcting Over- and Underspecified Truth Tables	10-55
Defining an Overspecified Truth Table	10-55
Defining an Underspecified Truth Table	10-56
Model Coverage for Truth Tables	10-58
How Stateflow Realizes Truth Tables	10-62
Viewing the Graphical Function for a Truth Table	10-62
How Stateflow Generates Graphical Functions for Truth Tables	10-62

Using Embedded MATLAB Functions

11

Introduction to Embedded MATLAB Functions	11-2
Building a Simulink Model with a Stateflow Embedded MATLAB Function	11-5
Programming a Stateflow Embedded MATLAB Function	11-13
Debugging a Stateflow Embedded MATLAB Function ..	11-18
Checking Embedded MATLAB Functions for Syntax Errors	11-18
Run-Time Debugging for Embedded MATLAB Functions	11-20
Model Coverage for an Embedded MATLAB Function ..	11-29
Types of Model Coverage in Embedded MATLAB Functions	11-30
Creating a Model with Embedded MATLAB Function Decisions	11-30
Understanding Embedded MATLAB Function Model Coverage	11-35

Building Targets

12

Overview of Stateflow Targets	12-3
What Is a Simulink RTW Target?	12-3
What Is a Stateflow Target?	12-3
How Do You Build a Target?	12-5
How Does Stateflow Build into Targets?	12-7
Adding a Stateflow Target	12-8
Configuring a Simulation Target for Stateflow	12-11

Configuring Real-Time Workshop for Stateflow	12-14
Configuring Stateflow Blocks in Nonlibrary Models for Real-Time Workshop	12-14
Configuring the Stateflow Blocks in Library Models for Real-Time Workshop	12-17
Configuring a Custom Target in Stateflow	12-24
Integrating Custom Code with Stateflow Targets	12-29
Specifying Custom Code Options for Stateflow Targets	12-29
Specifying Relative Paths for Custom Code	12-32
Including Custom C++ Code	12-33
Calling Graphical Functions from Custom Code	12-35
Starting the Build	12-36
Starting a Simulation Target Build	12-36
Starting an RTW Target Build	12-37
Parsing Stateflow Diagrams	12-38
Calling the Stateflow Parser	12-38
Parser Error Checking	12-39
Parsing Diagram Example	12-40
Resolving Event, Data, and Function Symbols	12-44
Error Messages	12-46
Parser Error Messages	12-46
Code Generation Error Messages	12-47
Compilation Error Messages	12-48
Generated Files	12-49
DLL Files	12-49
Code Files	12-50
Makefiles	12-51

Debugging with the Debugging Window	13-2
Setting Breakpoints for Debugging	13-3
Setting Error Checking in the Debugging Window	13-5
Starting Simulation in the Debugging Window	13-5
Controlling Animation in the Debugging Window	13-7
Controlling the Execution Rate in the Debugging Window ...	13-7
Setting the Output Display Pane	13-8
Debugging Run-Time Errors Example	13-9
Create the Model and Stateflow Diagram	13-9
Debugging the Stateflow Diagram	13-11
Correcting the Run-Time Error	13-12
Debugging State Inconsistencies	13-14
Causes of State Inconsistency	13-14
Detecting State Inconsistency	13-14
State Inconsistency Example	13-15
Debugging Conflicting Transitions	13-16
Detecting Conflicting Transitions	13-16
Conflicting Transition Example	13-16
Debugging Data Range Violations	13-18
Detecting Data Range Violations	13-18
Data Range Violation Example	13-18
Debugging Cyclic Behavior	13-20
Cyclic Behavior Example	13-21
Flow Cyclic Behavior Not Detected Example	13-22
Noncyclic Behavior Flagged as a Cyclic Example	13-23
Watching Data Values with Debuggers	13-24
Watching Data in the Stateflow Debugger	13-24
Watching Stateflow Data in MATLAB	13-25

Monitoring Stateflow Test Points	13-30
Setting Test Points for Stateflow States and Local Data with Model Explorer	13-31
Logging Data Values and State Activity	13-33
Using a Floating Scope to Monitor Data Values and State Activity	13-38
Understanding Model Coverage for Stateflow Charts ...	13-42
Making Model Coverage Reports	13-43
Specifying Coverage Report Settings	13-43
Cyclomatic Complexity	13-44
Decision Coverage	13-44
Condition Coverage	13-49
MCDC Coverage	13-49
Coverage Reports for Stateflow Charts	13-50
Colored Stateflow Diagram Coverage Display	13-59

Exploring and Modifying Charts

14

Using the Model Explorer with Stateflow Objects	14-2
Viewing Stateflow Objects in the Model Explorer	14-3
Editing States or Charts in the Model Explorer	14-5
Adding Data and Events in the Model Explorer	14-6
Adding a Target in the Model Explorer	14-6
Renaming Objects in the Model Explorer	14-8
Setting Properties for Stateflow Objects in the Model Explorer	14-8
Moving and Copying Data, Events, and Targets in the Model Explorer	14-9
Changing the Port Order of Input and Output Data and Events	14-10
Deleting Data, Events, and Targets in the Model Explorer .	14-11
Using the Stateflow Search & Replace Tool	14-12
Opening the Search & Replace Tool	14-12
Using Different Search Types	14-15

Specify the Search Scope	14-17
Using the Search Button and View Area	14-19
Specifying the Replacement Text	14-22
Using the Replace Buttons	14-23
Search and Replace Messages	14-24
Using the Stateflow Finder Tool	14-27
Opening Stateflow Finder	14-27
Using Stateflow Finder	14-28
Finder Display Area	14-31

The Stateflow Block

A	Stateflow	A-2
----------	-----------------	-----

Semantic Rules Summary

B	Semantic Rules Summary for Stateflow	B-2
	Entering a Chart	B-2
	Executing an Active Chart	B-2
	Entering a State	B-2
	Executing an Active State	B-3
	Exiting an Active State	B-3
	Executing a Set of Flow Graphs	B-4
	Executing an Event Broadcast	B-5

Glossary

Stateflow Concepts

This chapter acquaints you with the concepts involved in defining a finite state machine and follows this with a look at the hierarchical organization of Stateflow[®] objects. Later, it introduces you to a real-world Stateflow example. The sections in this chapter are as follows:

- Finite State Machine Concepts (p. 1-2) Stateflow is an example of a finite state machine. This section examines what it means to be a finite state machine and what that designation requires from Stateflow.
- Stateflow and Simulink (p. 1-5) Examines how Stateflow functions in the Simulink[®] environment.
- Stateflow Diagram Objects (p. 1-9) This section describes most of the graphical and nongraphical objects in a Stateflow diagram along with the concepts that relate them.
- Stateflow Hierarchy of Objects (p. 1-20) Stateflow supports a containment hierarchy for both charts and states. Charts can contain states, transitions, and the other Stateflow objects. States can contain other states, transitions, and so on as if they were charts themselves. This containment hierarchy applies to all Stateflow objects except targets, which are the sole possession of the Stateflow machine.
- Exploring a Real-World Stateflow Application (p. 1-22) The modeling of a real-world fault-tolerant fuel control system demonstrates how Simulink and Stateflow can be used to efficiently model hybrid systems containing both continuous dynamics and complex logical behavior.

Finite State Machine Concepts

Stateflow is an example of a finite state machine. The following topics examine what it means to be a finite state machine and what that designation requires from Stateflow:

- “What Is a Finite State Machine?” on page 1-2 — Introduces you to the concept of a finite state machine of which Stateflow is a variant.
- “Finite State Machine Representations” on page 1-2 — Discusses traditional approaches to representing finite state machines.
- “Stateflow Representations” on page 1-3 — Discusses Stateflow’s representation for its version of the finite state machine.
- “Notation” on page 1-3 — Describes how Stateflow uses its notation to represent the objects that it uses.
- “Semantics” on page 1-4 — Stateflow semantics describe how Stateflow notation is interpreted and implemented into a designed behavior.
- “References” on page 1-4 — Provides more information on finite state machine theory.

What Is a Finite State Machine?

A *finite state machine* is a representation of an event-driven (reactive) system. In an event-driven system, the system makes a transition from one state (mode) to another prescribed state, provided that the condition defining the change is true.

For example, you can use a state machine to represent a car’s automatic transmission. The transmission has a number of operating states: park, reverse, neutral, drive, and low. As the driver shifts from one position to another the system makes a transition from one state to another, for example, from park to reverse.

Finite State Machine Representations

Traditionally, designers used truth tables to represent relationships among the inputs, outputs, and states of a finite state machine. The resulting table describes the logic necessary to control the behavior of the system under study. Another approach to designing event-driven systems is to model the behavior of the system by describing it in terms of transitions among states. The state

that is active is determined based on the occurrence of events under certain conditions. State-transition diagrams and bubble diagrams are graphical representations based on this approach.

Stateflow Representations

Stateflow uses a variant of the finite state machine notation established by Harel [1]. Using Stateflow, you create Stateflow diagrams. A Stateflow diagram is a graphical representation of a finite state machine, where *states* and *transitions* form the basic building blocks of the system. You can also represent flow (stateless) diagrams using Stateflow. Stateflow provides a block that you include in a Simulink model. The collection of Stateflow blocks in a Simulink model is the Stateflow machine.

Additionally, Stateflow enables the representation of hierarchy, parallelism, and history. Hierarchy enables you to organize complex systems by defining a parent/offspring object structure. For example, you can organize states within other higher-level states. A system with parallelism can have two or more orthogonal states active at the same time. History provides the means to specify the destination state of a transition based on historical information. These characteristics enhance the usefulness of this approach and go beyond what state-transition diagrams and bubble diagrams provide.

Notation

Notation defines a set of objects and the rules that govern the relationships between those objects. Stateflow notation provides a common language to communicate the design information conveyed by a Stateflow diagram.

Stateflow notation consists of the following:

- A set of graphical objects
- A set of nongraphical text-based objects
- Defined relationships between those objects

See “Stateflow Notation” on page 2-1 for detailed information on Stateflow notations.

Semantics

Semantics describe how the notation is interpreted and implemented. A completed Stateflow diagram illustrates how the system will behave. A Stateflow diagram contains actions associated with transitions and states. The semantics describe in what sequence these actions take place during Stateflow diagram execution.

Knowledge of the semantics is important to make sound Stateflow diagram design decisions for code generation. Different use of notations results in different ordering of simulation and generated code execution.

The default semantics provided with the product are described in Chapter 3, “Stateflow Semantics.”

References

For more information on finite state machine theory, consult these sources:

[1] Harel, David, “Statecharts: A Visual Formalism for Complex Systems,” *Science of Computer Programming* 8, 1987, pages 231-274.

[2] Hatley, Derek J., and Imtiaz A. Pirbhai, *Strategies for Real-Time System Specification*, Dorset House Publishing Co., Inc., NY, 1988.

Stateflow and Simulink

Stateflow functions as a finite state machine within a Simulink model. The following topics examine how Stateflow functions in this environment and how Simulink and Stateflow communicate with each other:

- “The Simulink Model and the Stateflow Machine” on page 1-5 — Describes the Stateflow machine which is the collection of Stateflow blocks in a Simulink model.
- “Stateflow Data Dictionary of Objects” on page 1-6 — Describes the Stateflow data dictionary, which is the hierarchical collection of all Stateflow objects in a Simulink model.
- “Defining Stateflow Interfaces to Simulink” on page 1-7 — Tells you how each Stateflow block in Simulink interfaces with the rest of the Simulink model.
- “Stateflow Diagram Objects” on page 1-9 — Describes most of the Stateflow objects used in creating Stateflow diagrams.

The Simulink Model and the Stateflow Machine

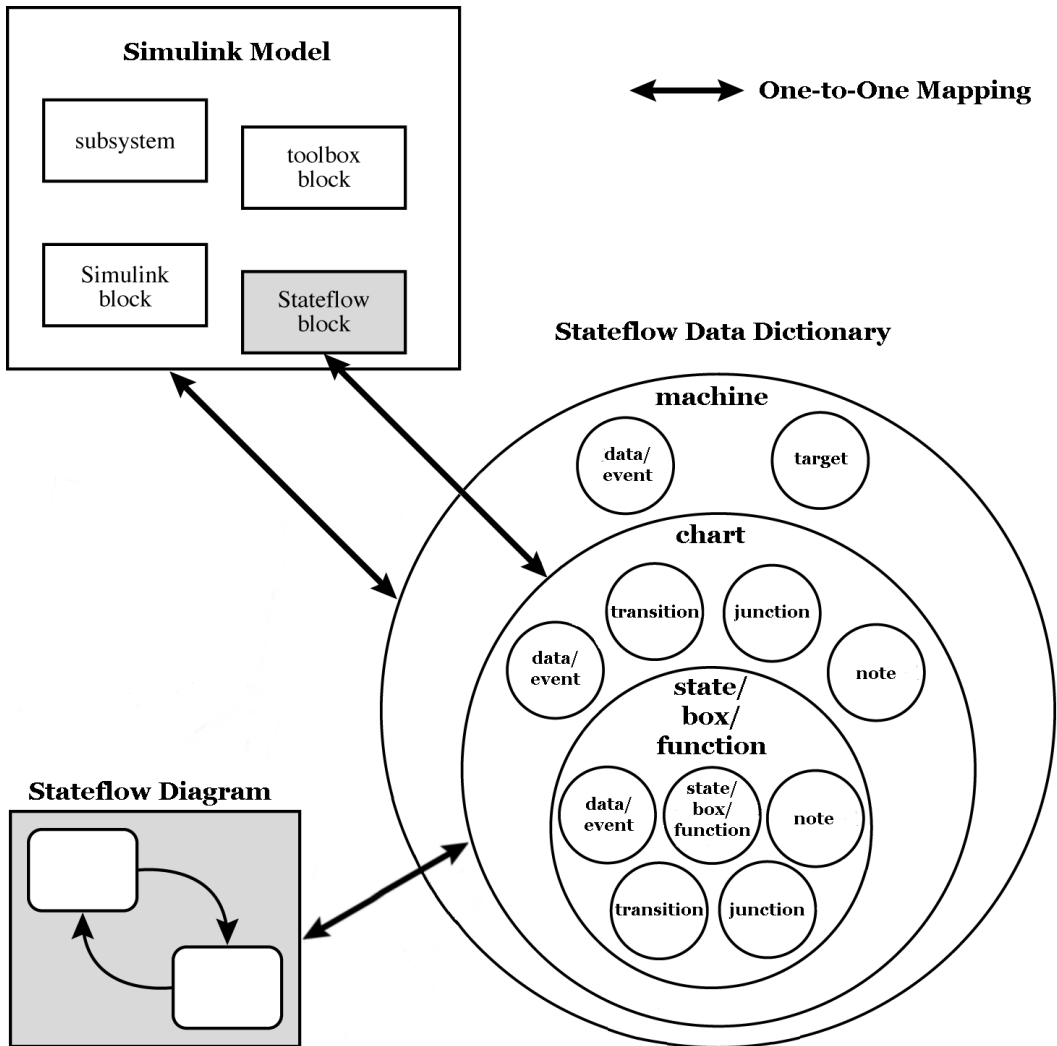
The Stateflow machine is the collection of Stateflow blocks in a Simulink model. The Simulink model and Stateflow machine work seamlessly together. Running a simulation automatically executes both the Simulink and Stateflow portions of the model.

A Simulink model can consist of combinations of Simulink blocks, toolbox blocks, and Stateflow blocks (Stateflow diagrams). In Stateflow, the chart (Stateflow diagram) consists of a set of graphical objects (states, boxes, functions, notes, transitions, connective junctions, and history junctions) and nongraphical objects (events, data, and targets).

There is a one-to-one correspondence between the Simulink model and the Stateflow machine. Each Stateflow block in the Simulink model is represented in Stateflow by a single chart (Stateflow diagram). Each Stateflow machine has its own object hierarchy. The Stateflow machine is the highest level in the Stateflow hierarchy. The object hierarchy beneath the Stateflow machine consists of combinations of graphical and nongraphical objects.

Stateflow Data Dictionary of Objects

The Stateflow data dictionary is the internal representation for the hierarchy of all Stateflow objects, graphical and nongraphical, that reside in a Simulink model. It is represented by the following diagram:



Stateflow scoping rules dictate where different nongraphical objects can exist in the hierarchy. For example, data and events can be parented by the machine, the chart (Stateflow diagram), or by a state. Targets can only be parented by the machine. Once a parent is chosen, that object is known in the hierarchy from the parent downward (including the parent's offspring). For example, a data object parented by the machine is accessible by that machine, by any charts within that machine, and by any states within that machine.

The hierarchy of the graphical objects is easily and automatically handled for you by the graphics editor. You manage the hierarchy of nongraphical objects through the Explorer or the graphics editor **Add** menu. See “Stateflow Hierarchy of Objects” on page 1-20.

Defining Stateflow Interfaces to Simulink

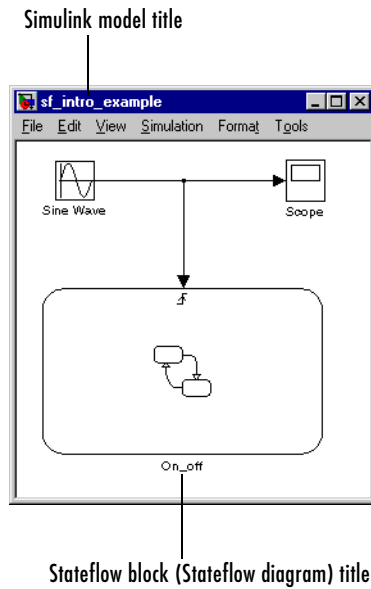
Each Stateflow block corresponds to a single Stateflow diagram. The Stateflow block interfaces to its Simulink model. The Stateflow block can interface to code sources external to the Simulink model (data, events, custom code).

Stateflow diagrams are event driven. Events can be local to the Stateflow block or can be propagated to and from Simulink and code sources external to Simulink. Data can be local to the Stateflow block or can be shared with and passed to the Simulink model and to code sources external to the Simulink model.

You must define the interface to each Stateflow block. Defining the interface for a Stateflow block can involve some or all of these tasks:

- Defining the Stateflow block update method
- Defining **Output to Simulink** events
- Adding and defining nonlocal events and nonlocal data within the Stateflow diagram
- Defining relationships with any external sources

In the following example, the Simulink model titled `sf_intro_example` consists of a Simulink Sine Wave source block, a Simulink Scope sink block, and a single Stateflow block, titled `On_off`.



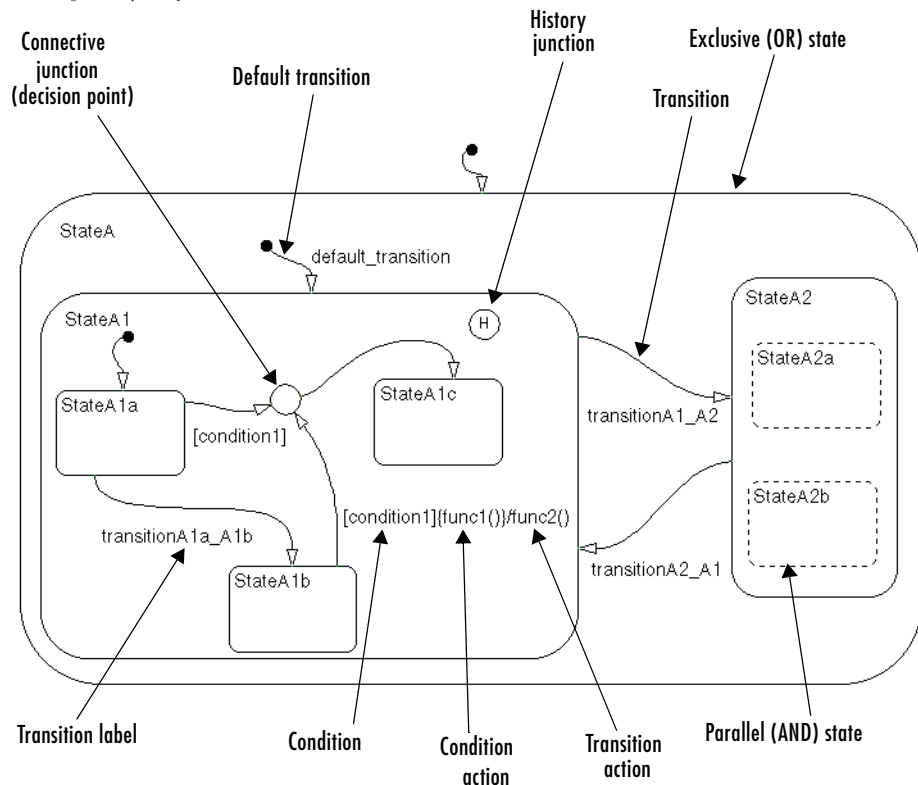
See “Defining Input Events” on page 6-11 and “Defining Interfaces to Simulink and MATLAB” on page 9-1 for more information.

Stateflow Diagram Objects

Stateflow diagrams are made of objects. Some of these objects graphical, that is, you draw them in a Stateflow diagram. Some of these objects are nongraphical, that is, they do not have a graphical appearance, but are referred to in the Stateflow diagram.

The following sample Stateflow diagram displays some of the key graphical objects of a Stateflow diagram.

Stateflow Diagram of Graphical Objects



This section introduces you to the graphical and nongraphical objects in a Stateflow diagram in the following topics:

- “States” on page 1-10 — Describes the role of *states* in a Stateflow diagram that represent the current mode of an executing diagram.
- “Transitions” on page 1-12 — Describes the role of a *transitions* in a Stateflow diagram that provide a path for an executing diagram to change from one mode (state) to another.
- “Default Transitions” on page 1-13 — Describes the role of a *default transition* that specifies which exclusive (OR) state in an executing diagram is active on startup.
- “Events” on page 1-14 — Describes the role of *events* that drive Stateflow diagram execution.
- “Data” on page 1-14 — Describes the role of *data* that serves as variables in a Stateflow diagram.
- “Conditions” on page 1-15 — Describes the *conditions* that, along with events, drive Stateflow diagram execution.
- “History Junction” on page 1-16 — Describes the role of *history junctions* that record the most recently active substate of a chart or superstate.
- “Actions” on page 1-17 — Describes the role of *actions* that take place as part of Stateflow diagram execution.
- “Connective Junctions” on page 1-18 — Describes the role of *connective junctions* that provide decision points for transitions taking place during diagram execution.

All Stateflow objects are arranged in a hierarchy of objects. See “Stateflow Hierarchy of Objects” on page 1-20.

States

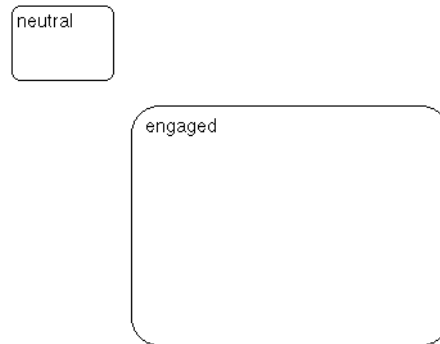
A *state* describes a mode of an event-driven system. The activity or inactivity of the states dynamically changes based on events and conditions.

Every state has a parent. In a Stateflow diagram consisting of a single state, that state’s parent is the Stateflow diagram itself (also called the Stateflow diagram root). You can place states within other higher-level states. In the preceding figure, StateA1 is a child of StateA.

A state can have its activity history recorded in a *history junction*. History provides an efficient means of basing future activity on past activity. See “History Junction” on page 1-16.

States have labels that can specify actions executed in a sequence based upon action type. The action types are entry, during, exit, and on. See “Actions” on page 1-17.

The *decomposition* of a state defines the kind of state that a state can contain and the next level of containment. Stateflow provides two types of states: exclusive (OR) and parallel (AND) states. Exclusive (OR) states are used to describe modes that are mutually exclusive. A chart or state that contains exclusive (OR) states is said to have exclusive decomposition. The following transmission example has exclusive (OR) states.

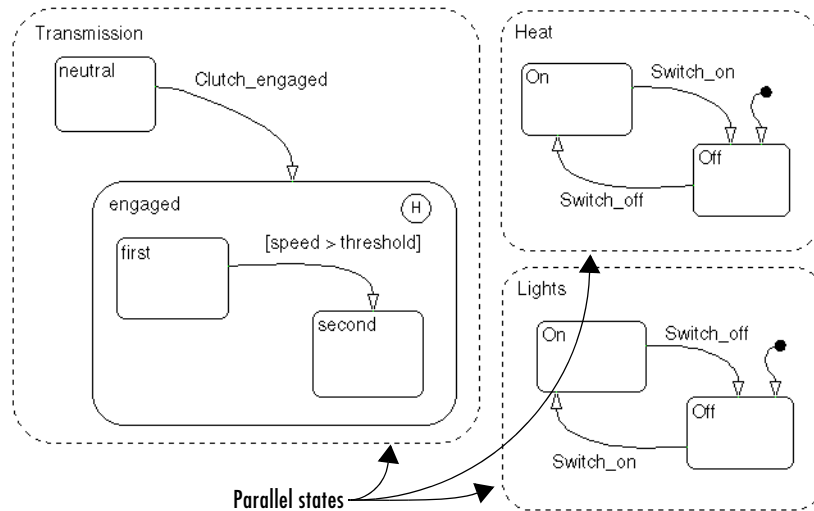


An automatic transmission can be set to either neutral or engaged. In this example either the neutral state or the engaged state is active at any one time. Both cannot be active at the same time.

A chart or state with *parallel* states has two or more states that can be active at the same time. A chart or state that contains parallel (AND) states is said to have parallel decomposition.

Parallel (AND) states are displayed as dashed rectangles. The activity of each parallel state is essentially independent of other states. In the diagram in “Stateflow Diagram of Graphical Objects” on page 1-9, StateA2 has parallel (AND) state decomposition. Its states, StateA2a and StateA2b, are parallel (AND) states.

The following Stateflow diagram has parallel superstate decomposition.

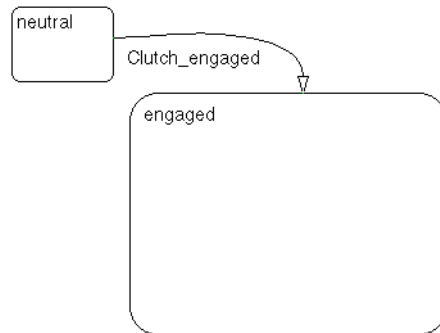


In this example, the transmission, heating, and light systems are parallel subsystems in a car. They are active at the same time and are physically independent of each other. There are many other parallel components in a car, such as the braking and windshield wiper subsystems.

Transitions

A *transition* is a graphical object that, in most cases, links one object to another. One end of a transition is attached to a source object and the other end to a destination object. The *source* is where the transition begins and the *destination* is where the transition ends. A *transition label* describes the circumstances under which the system moves from one state to another. It is always the occurrence of some event that causes a transition to take place. In the diagram in “Stateflow Diagram of Graphical Objects” on page 1-9, the transition from StateA1 to StateA2 is labeled with the event `transitionA1_A2` that triggers the transition to occur.

Consider again the automatic transmission system. `clutch_engaged` is the event required to trigger the transition from `neutral` to `engaged`.

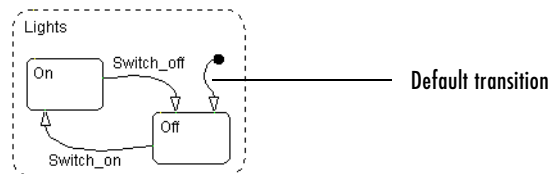


Default Transitions

Default transitions specify which exclusive (OR) state is to be active when there is ambiguity between two or more exclusive (OR) states at the same level in the hierarchy.

For example, in the diagram in “Stateflow Diagram of Graphical Objects” on page 1-9, the default transition to StateA1 resolves the ambiguity that exists with regard to whether StateA1 or StateA2 should be active when State A becomes active. In this case, when StateA is active, by default StateA1 is also active.

In the following Lights subsystem, the default transition to the Lights.Off substate indicates that when the Lights superstate becomes active, the Off substate becomes active by default.



Note History junctions override default transition paths in superstates with exclusive (OR) decomposition.

In parallel (AND) states, a default transition must always be present to indicate which of its exclusive (OR) states is active when the parallel state becomes active.

Events

Events drive the Stateflow diagram execution but are nongraphical objects and are thus not represented directly in a Stateflow chart. All events that affect the Stateflow diagram must be defined. The occurrence of an event causes the status of the states in the Stateflow diagram to be evaluated. The broadcast of an event can trigger a transition to occur or can trigger an action to be executed. Events are broadcast in a top-down manner starting from the event's parent in the hierarchy.

Events are created and modified using the Stateflow Explorer. Events can be created at any level in the hierarchy. Events have properties such as a scope. The scope defines whether the event is

- Local to the Stateflow diagram
- An input to the Stateflow diagram from its Simulink model
- An output from the Stateflow diagram to its Simulink model
- Exported to a (code) destination external to the Stateflow diagram and Simulink model
- Imported from a code source external to the Stateflow diagram and Simulink model

Data

Data objects are used to store numerical values for reference in the Stateflow diagram. They are nongraphical objects and are thus not represented directly in a Stateflow chart.

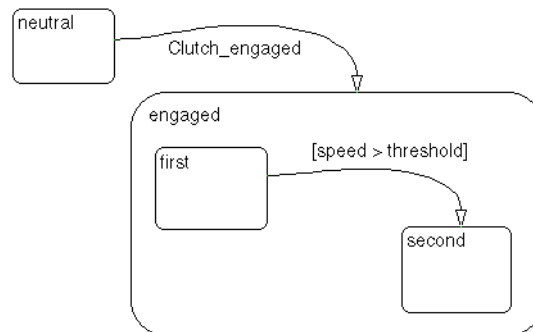
You create and modify data objects for Stateflow diagrams in Stateflow Explorer. Data objects have a property called scope that defines whether the data object is

- Local to the Stateflow diagram
- An input to the Stateflow diagram from its Simulink model
- An output from the Stateflow diagram to its Simulink model
- Nonpersistent temporary data
- Defined in the MATLAB[®] workspace
- A constant
- Exported to a (code) destination external to the Stateflow diagram and Simulink model
- Imported from a code source external to the Stateflow diagram and Simulink model

Conditions

A *condition* is a Boolean expression specifying that a transition occurs, given that the specified expression is true. In the component summary Stateflow diagram, [condition1] represents a Boolean expression that must be true for the transition to occur.

In the automatic transmission system, the transition from `first` to `second` occurs if the transition condition `[speed > threshold]` is true.

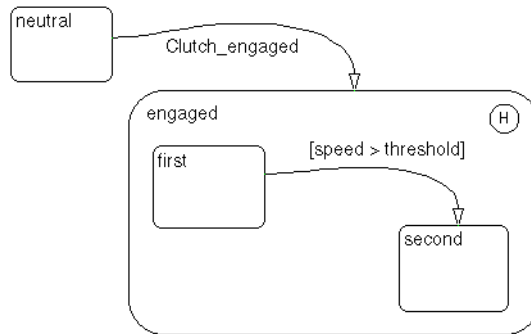


History Junction

A *history junction* records the most recently active state of a chart or superstate.

If a superstate with exclusive (OR) decomposition has a history junction, the destination substate is defined to be the substate that was most recently visited. A history junction applies to the level of the hierarchy in which it appears. The history junction overrides any default transitions. In the component summary Stateflow diagram, the history junction in StateA1 indicates that when a transition to StateA1 occurs, the substate that becomes active (StateA1a, StateA1b, or StateA1c) is based on which of those substates was most recently active.

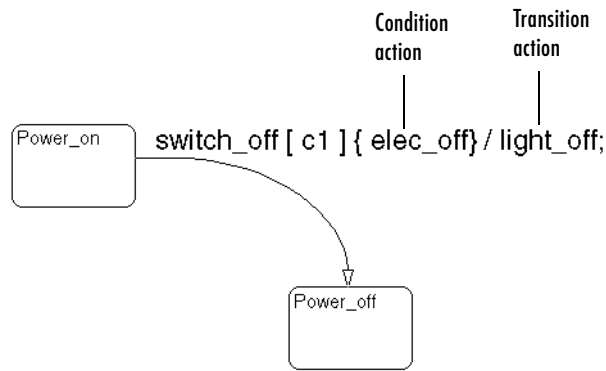
In the automatic transmission system, history indicates that when `clutch_engaged` causes a transition from `neutral` to the engaged superstate, the substate that becomes active, either `first` or `second`, is based on which of those substates was most recently active.



Actions

Actions take place as part of Stateflow diagram execution. The action can be executed either as part of a transition from one state to another or based on the activity status of a state.

Transitions ending in a state can have *condition* actions and *transition* actions, as shown in the following example:



In the diagram in “Stateflow Diagram of Graphical Objects” on page 1-9, the transition segment from StateA1b to the connective junction is labeled with the condition action `func1()` and the transition action `func2()`. The semantics of how and why actions take place are discussed throughout the examples listed in “Semantic Examples” on page 3-29.

States can have entry, during, exit, and on *event_name* actions. For example,

```
Power_on/
entry: ent_action();
during: dur_action();
exit: exit_action();
on Switch_off: on_action();
```

Action language defines the types of actions you can specify and their associated notations. An action can be a function call, the broadcast of an event, the assignment of a value to a variable, and so on.

Stateflow supports both Mealy and Moore finite state machine modeling paradigms. In the Mealy model, actions are associated with transitions, whereas in the Moore model they are associated with states. Stateflow supports state actions, transition actions, and condition actions. For more information, see the following:

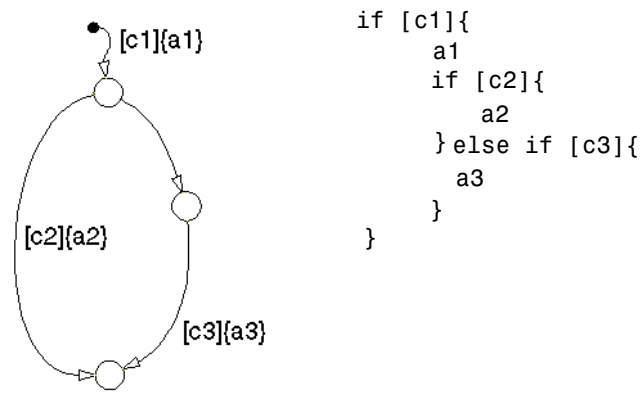
- “State Labels” on page 2-9 — Describes action language for states, which is included in the label for a state
- “Transition Label Notation” on page 2-14 — Describes action language for transitions which is included in the label of a transition.
- “Labeling States” on page 4-11 — Shows you to label states with its name and actions in the Stateflow diagram editor.
- “Labeling Transitions” on page 4-17 — Shows you how to label transitions with actions in the Stateflow diagram editor.

Connective Junctions

Connective junctions are decision points in the system. A connective junction is a graphical object that simplifies Stateflow diagram representations and facilitates generation of efficient code. Connective junctions provide alternative ways to represent desired system behavior. In the diagram in “Stateflow Diagram of Graphical Objects” on page 1-9, the connective junction is used as a decision point for two transition segments that complete at StateA1c.

Transitions connected to junctions are called *transition segments*. Transitions, apart from default transitions, must go state to state. However, once the transition segments taken complete a state to state transition, the accumulation of the transition segments taken forms a complete transition.

The following example shows how connective junctions (displayed as small circles) are used to represent the flow of an if - else code structure shown in accompanying pseudocode.



```

if [c1]{
    a1
    if [c2]{
        a2
    }else if [c3]{
        a3
    }
}
  
```

This example executes as follows:

- 1** If condition [c1] is true, condition action a1 is executed and the default transition to the top junction is taken.
- 2** Stateflow now considers which transition segment to take out of the top junction (it can take only one). Junctions with conditions have priority over junctions without conditions, so the transition with the condition [c2] is considered first.
- 3** If condition [c2] is true, action a2 is executed and the transition segment to the bottom junction is taken. Because there are no outgoing transition segments from the bottom junction, the diagram is finished executing.
- 4** If condition [c2] is false, the empty transition segment on the right is taken (because it has no condition at all).
- 5** If condition [c3] is true, condition action a3 is executed and the transition segment from the middle to the bottom junction is taken. Because there are no outgoing transition segments from the bottom junction, the diagram is finished executing.
- 6** If condition [c3] is false, execution is finished at the middle junction.

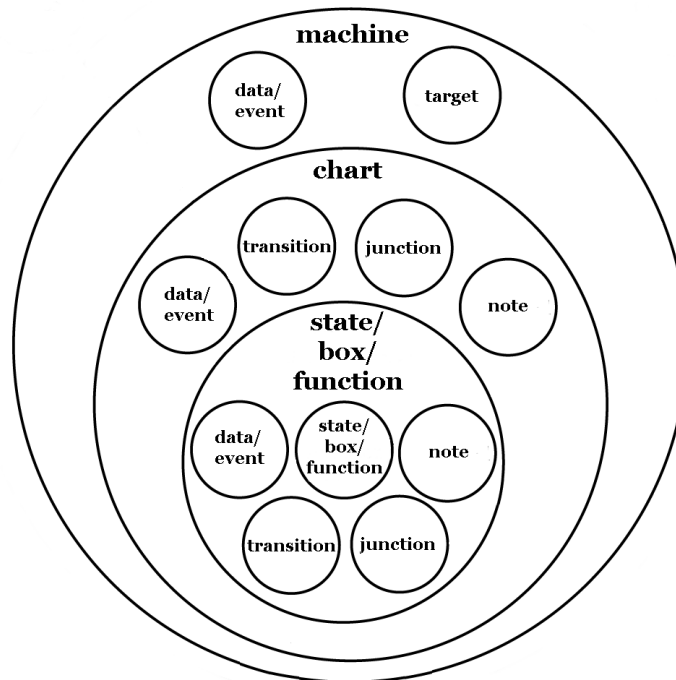
The above steps describe the execution of the example diagram for connective junctions with Stateflow semantics. Stateflow semantics describe how objects in diagrams relate to each other during execution. See “Stateflow Semantics” on page 3-1.

Stateflow Hierarchy of Objects

Stateflow diagrams arrange Stateflow objects in a hierarchy of objects. This hierarchy is based on containment. That is, one Stateflow object can contain other Stateflow objects.

The object hierarchy for all Stateflow objects in a Simulink model is referred to as the Stateflow Data Dictionary. This dictionary defines which objects a particular object can contain in a Stateflow diagram. The Stateflow Data Dictionary is depicted in the following diagram.

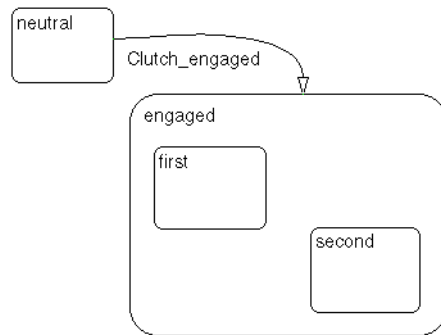
Stateflow Data Dictionary



The highest object in Stateflow hierarchy is the Stateflow machine. It is defined as an object that contains all other Stateflow objects in a Simulink model. This means that the Stateflow machine contains all the Stateflow charts (diagrams) in a Simulink model. In addition, the Stateflow machine for

a model can also contain its own data, event, and target objects. Only a simulation target (named `sfun`) is added to the Stateflow machine by default when the model is created. All other data, event, and target objects must be added to the machine.

Similarly, charts can contain state, box, function, data, event, transition, junction, and note events. You use all of these objects to create a Stateflow diagram. Continuing with the Stateflow hierarchy, states can contain all of these objects as well, including other states. Stateflow represents state hierarchy with superstates and substates. For example, this Stateflow diagram has a superstate that contains two substates.



In the preceding Stateflow diagram, the engaged superstate contains the first and second substates. The engaged superstate is the parent in the hierarchy to the states first and second. When the event `clutch_engaged` occurs, the system transitions out of the neutral state to the engaged superstate. Transitions within the engaged superstate are intentionally omitted from this example for simplicity.

A transition out of a superstate implies transitions out of any of its active substates. Transitions can cross superstate boundaries to specify a substate destination. If a substate is made active its parent superstate is also made active.

The Stateflow hierarchy of objects lets you organize complex Stateflow diagrams by defining a containment structure. A hierarchical design usually reduces the number of transitions and produces neat, manageable diagrams. Stateflow supports a hierarchical organization of both charts and states.

Exploring a Real-World Stateflow Application

The modeling of a fault-tolerant fuel control system demonstrates how Simulink and Stateflow can be used to efficiently model hybrid systems containing both continuous dynamics and complex logical behavior.

Simulink elements model behavior based on a given sample time. Each loop of its block diagram is assigned an increment of sample time. Stateflow execution makes no consideration for sample time. Internally, its application might take many cycles of execution, which are assumed to take place during the sample time assigned in Simulink.

The model described represents a fuel control system for a gasoline engine. This robust control system reacts to the detection of individual sensor failures and is dynamically reconfigured for uninterrupted operation. This section describes how Stateflow is used to implement supervisory logic control system to deal with the sensor failures and contains the following topics:

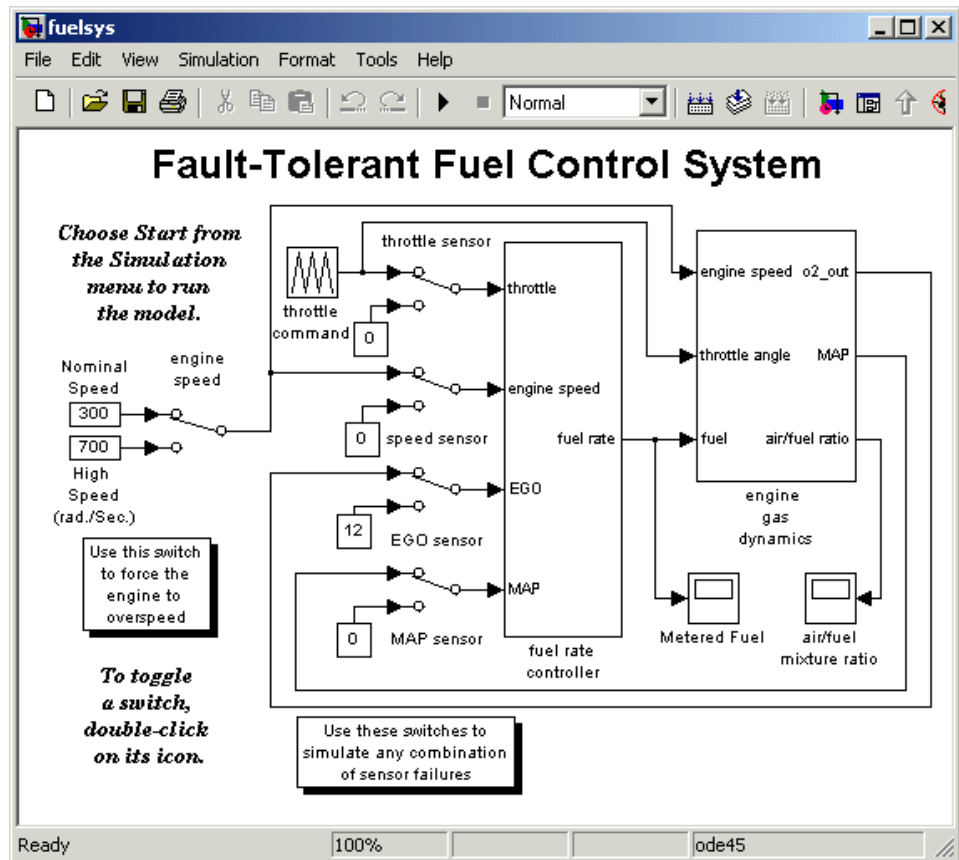
- “Overview of the “fuel rate controller” Model” on page 1-22 — Gives you a top-down look at the fuel rate controller demo model along with an introduction to the Simulink logic of the fuel rate controller model and how failures are simulated.
- “Control Logic of the “fuel rate controller” Model” on page 1-26 — Introduces you to the control logic Stateflow diagram of the fuel rate controller model and its response mechanisms for system failure.
- “Simulating the “fuel rate controller” Model” on page 1-29 — Takes you step by step through the simulation of the fuel rate controller model focusing on Stateflow simulation in which you can witness state changes in response to simulated failures.

Overview of the “fuel rate controller” Model

The mass flow rate of air pumped from the intake manifold, divided by the fuel rate, which is injected at the valves, gives the air/fuel ratio. The ideal mixture ratio provides a good compromise between power, fuel economy, and emissions. A target air/fuel ratio of 14.6 is assumed in this system.

A sensor (EGO) determines the amount of residual oxygen present in the exhaust gas. This gives a good indication of the air/fuel ratio and provides a feedback measurement for closed-loop control. If the sensor indicates a high oxygen level, the controller increases the fuel rate. If the sensor detects a fuel-rich mixture (corresponding to a very low level of residual oxygen), the controller decreases the fuel rate.

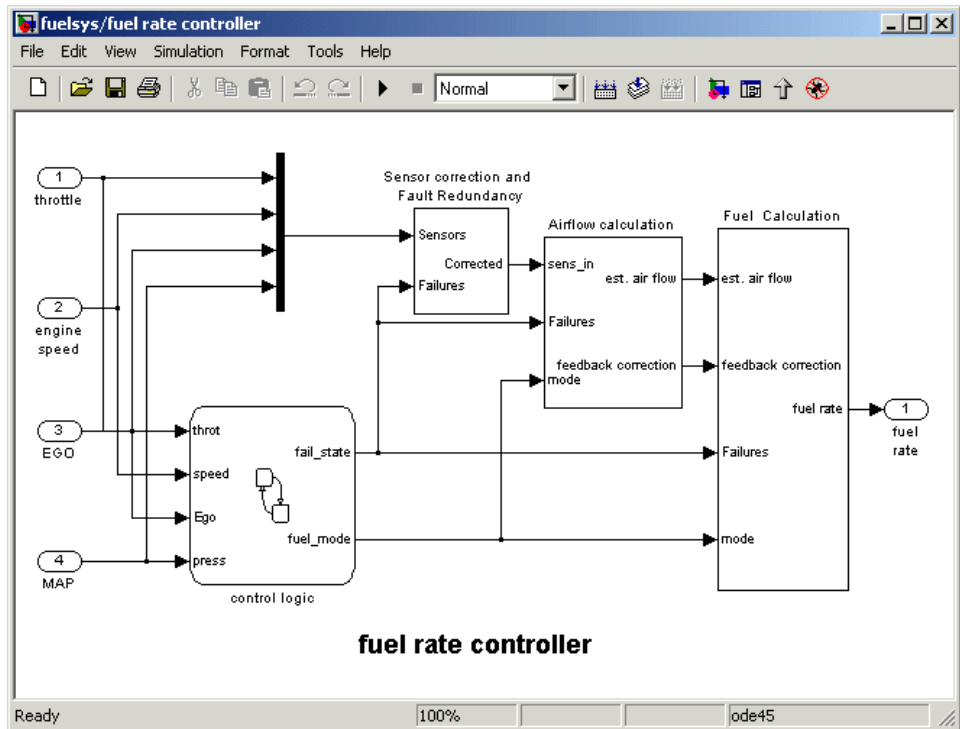
The following figure shows the top level of the Simulink model (`fuelsys.mdl`). The model is modularized into a fuel rate controller and a subsystem to simulate engine gas dynamics.



The fuel rate controller uses signals from the system's sensors to determine the fuel rate that gives an ideal mixture. The fuel rate combines with the actual air flow in the engine gas dynamics model to determine the resulting mixture ratio as sensed at the exhaust.

To simulate failures in the system, the user can selectively disable each of the four sensors: throttle angle, speed, exhaust gas (EGO), and manifold absolute pressure (MAP). Simulink accomplishes this with Manual Switch blocks. The user can toggle the position of a switch by double-clicking its icon prior to or during a simulation. Similarly, the user can induce the failure condition of a high engine speed by toggling the switch on the far left.

The controller uses the sensor input and feedback signals to adjust the fuel rate to provide an ideal ratio. The model uses four subsystems to implement this strategy: control logic, sensor correction, airflow calculation, and fuel calculation. Under normal operation, the model estimates the airflow rate and multiplies the estimate by the reciprocal of the desired ratio to give the fuel rate. Feedback from the oxygen sensor provides a closed-loop adjustment of the rate estimation in order to maintain the ideal mixture ratio.



A detailed explanation of the Simulink part of the fault-tolerant control system is given in *Using Simulink and Stateflow in Automotive Applications*, a Simulink-Stateflow Technical Examples booklet published by The MathWorks. This section concentrates on the supervisory logic part of the system that is implemented in Stateflow, but the following points are crucial to the interaction between Simulink and Stateflow:

- The supervisory logic monitors the input data readings from the sensors.
- The logic determines from these readings the sensors that have failed and outputs a failure state Boolean array as `fail_state`.
- Given the current failure state, the logic determines in which fueling mode the engine should be run.

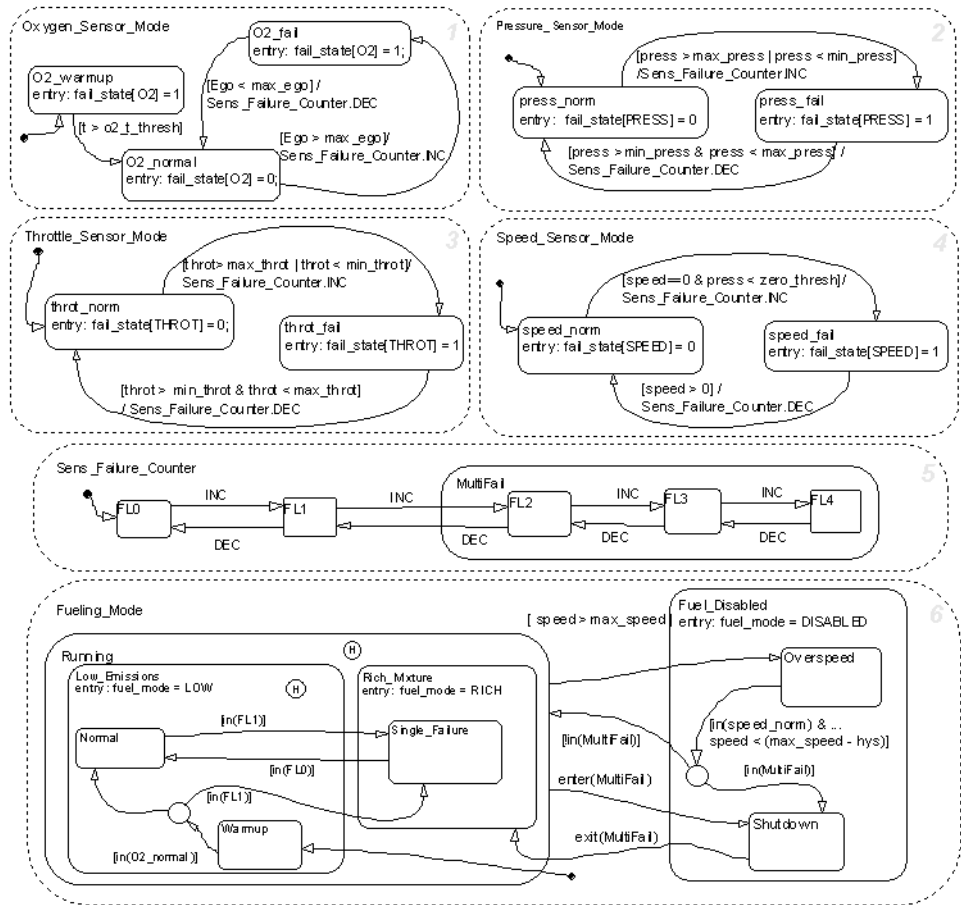
The fueling mode can be one of the following modes:

- **Low emissions mode** is the normal mode of operation where no sensors have failed.
- **Rich mixture mode** occurs when a sensor has failed, to ensure smooth running of the engine.
- **Shutdown mode** occurs when more than one sensor has failed, rendering the engine inoperable.

The fueling mode and failure state are output from Stateflow as `fuel_mode` and `fail_state` respectively into the algorithmic part of the model, where they determine the fueling calculations.

Control Logic of the “fuel rate controller” Model

The single Stateflow chart that implements the entire control logic for the `fuelsys` model is shown in the following diagram:



The chart consists of six parallel states with dashed boundaries that represent concurrent modes of operation.

The four parallel states at the top of the diagram correspond to the four individual sensors. Each of these states has a substate that represents the functioning or failing status of that sensor. These substates are mutually exclusive. For example, if the throttle sensor fails then the lone active substate of the `Throttle_Sensor_Mode` state is `thrott_fail`.

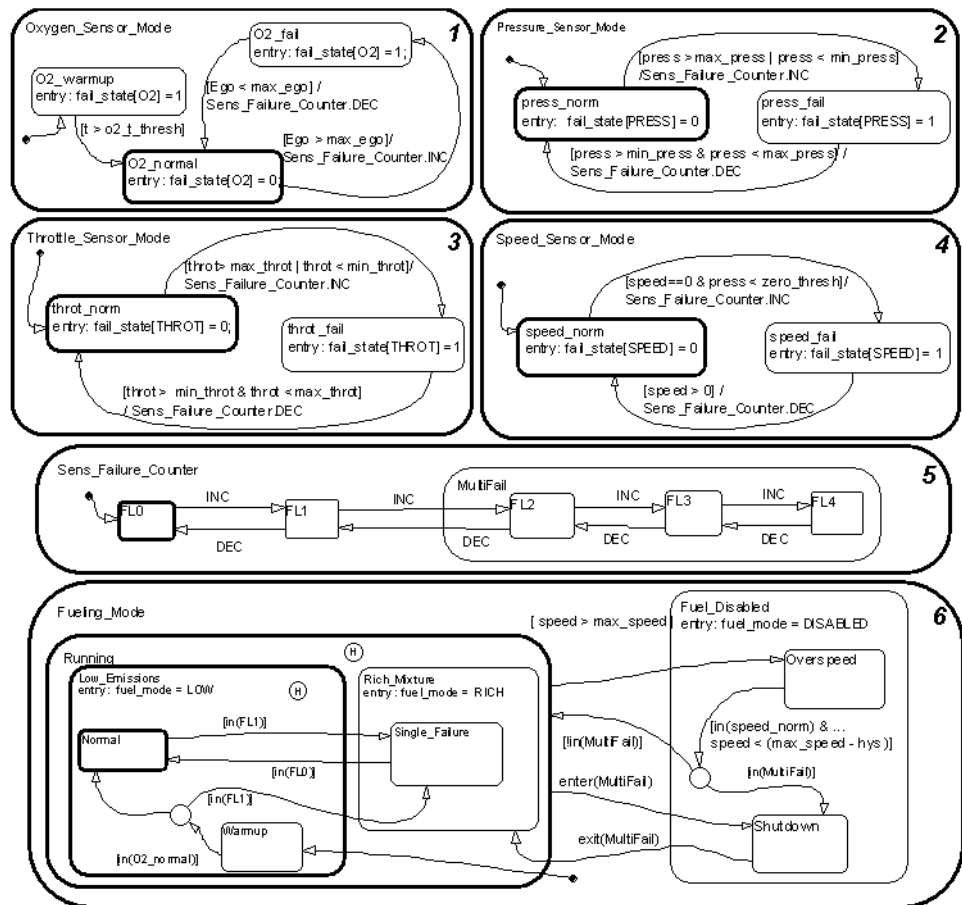
Transitions determine how states can change and can be guarded by conditions. For example, the active state can change from the `throt_norm` state to the `throt_fail` state when the measurement from the throttle sensor exceeds `max_throt` or is below `min_throt`.

The remaining two parallel states at the bottom consider the status of the four sensors simultaneously and determine the overall system operating mode. The `Sens_Failure_Counter` superstate acts as a store for the resultant number of sensor failures. This state is polled by the `Fueling_Mode` state that determines the fueling mode of the engine. If a single sensor fails, operation continues but the air/fuel mixture is richer to allow smoother running at the cost of higher emissions. If more than one sensor has failed, the engine shuts down as a safety measure, because the air/fuel ratio cannot be controlled reliably.

Although it is possible to run Stateflow charts asynchronously by injecting events from Simulink when required, the fueling control logic is polled synchronously at a rate of 100 Hz. Consequently, the sensors are checked every 1/100 second to see if they have changed status, and the fueling mode is adjusted accordingly.

Simulating the “fuel rate controller” Model

On starting the simulation, and assuming no sensors have failed, the Stateflow diagram initializes in the Warmup mode in which the oxygen sensor is deemed to be in a warmup phase. If Stateflow is placed into animation mode, the current state of the system can clearly be seen highlighted on the Stateflow diagram, as shown.



After a given time period, defined by `o2_t_thresh`, the sensor is deemed to have reached operating temperature and the system settles into the normal mode of operation, shown above, in which the fueling mode is set to NORMAL.

As the simulation progresses, the chart is woken up synchronously every 0.01 second. The events and conditions that guard the transitions are evaluated and if a transition is valid, it is taken and animated on the Stateflow diagram.

To illustrate this, you can provoke a transition by switching one of the sensors to a failure value on the top-level Simulink model. The system detects throttle and pressure sensor failures when their measured values fall outside their nominal ranges. A manifold vacuum in the absence of a speed signal indicates a speed sensor failure. The oxygen sensor also has a nominal range for failure conditions but, because zero is both the minimum signal level and the bottom of the range, failure can be detected only when it exceeds the upper limit.

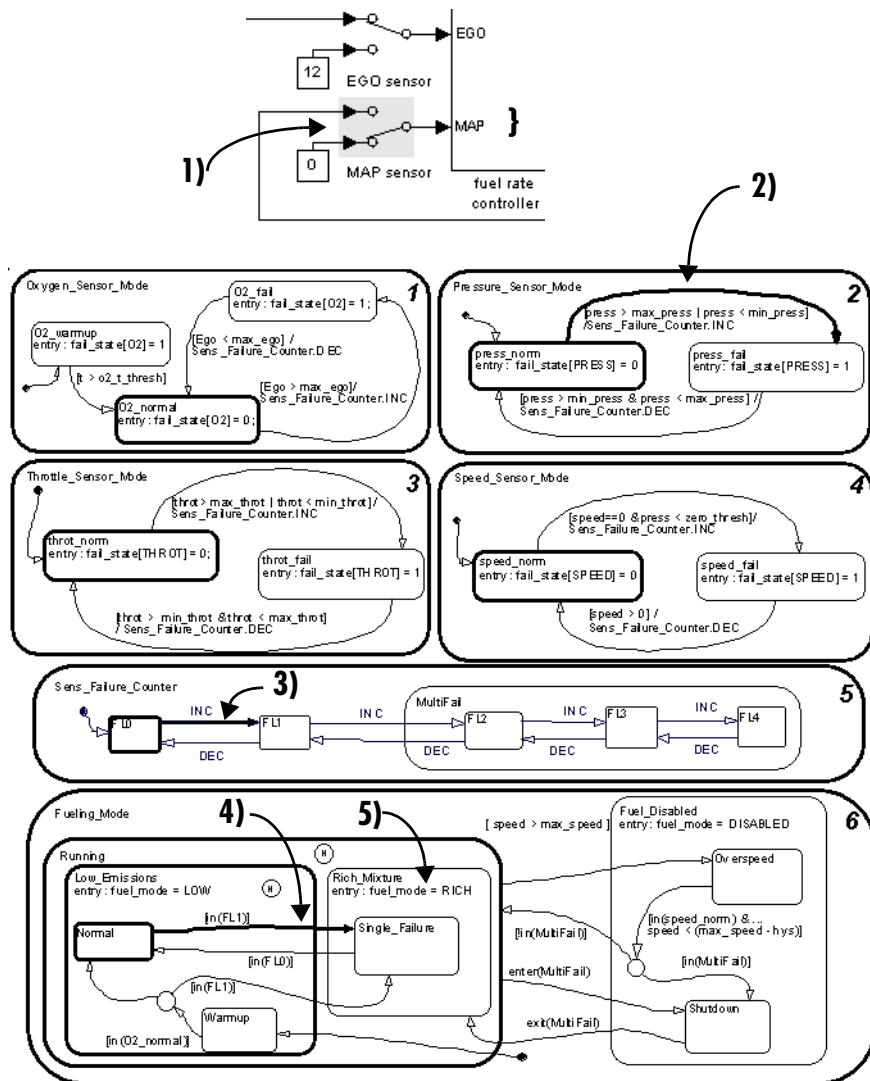
Switch the Simulink switch for the manifold air pressure sensor to the off position to witness the following sequence of transitions (note the diagram that follows).

- 1** Switching the Simulink manifold air pressure sensor switch causes a value of zero to be read by the fuel rate controller.
- 2** When the chart is next woken up, the transition from the `press_norm` state becomes valid as the reading is now out of bounds and the transition is taken to the `press_fail` state, as shown.
- 3** Regardless of which sensor fails, the model always generates the directed event broadcast `Sens_Failure_Counter.INC`, which makes the triggering of the universal sensor failure logic independent of the sensor.

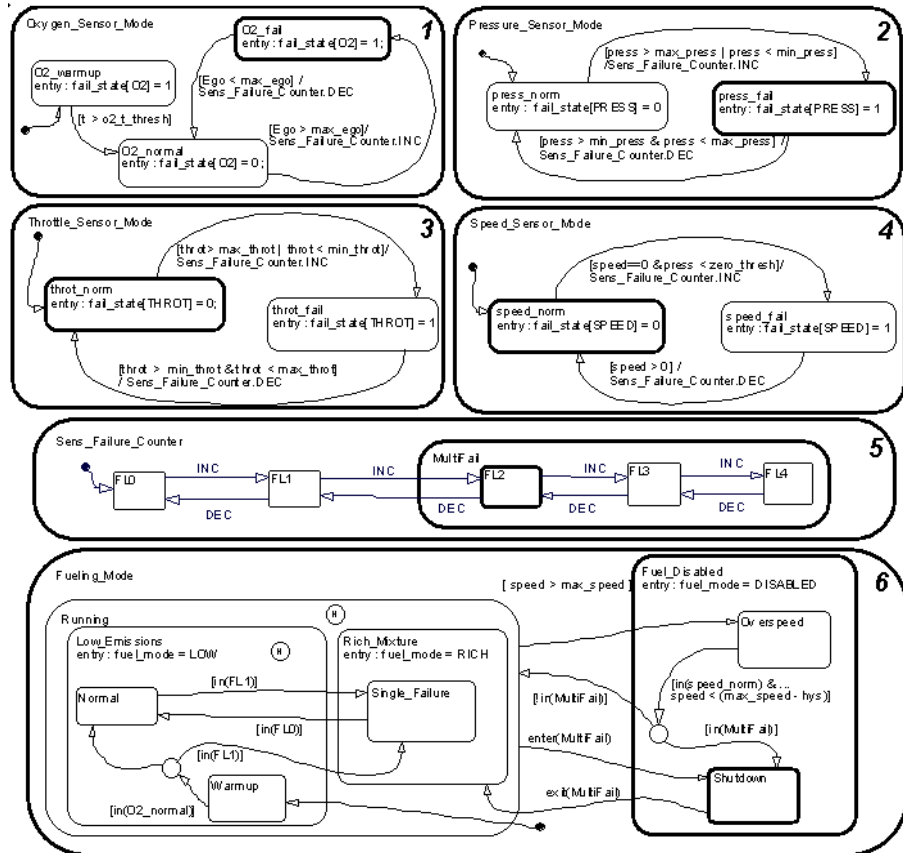
This event causes a second transition from `FL0` to `FL1` in the `Sens_Failure_Counter` superstate. Both transitions are animated on the Stateflow diagram.

- 4** With the `Sens_Failure_Counter` state showing one failure, the condition that guards the transition from the `Low_Emissions.Normal` state to the `Rich_Mixture.Single_Failure` state is now valid and is therefore taken.
- 5** As the `Fuel_Disabled` state is entered, the data `fuel_mode` is set to `RICH`, as shown.

The transitions taken in the preceding steps are depicted in the following simulation diagram. Step numbers appear next to the dashed indicator line.



A second sensor failure causes the `Sens_Failure_Counter` to enter the `Multifail` state, broadcasting an implicit event that immediately triggers the transition from the `Running` state to the `Shutdown` state. On entering the `Fuel_Disabled` superstate the Stateflow data `fuel_mode` is set to `DISABLED`.

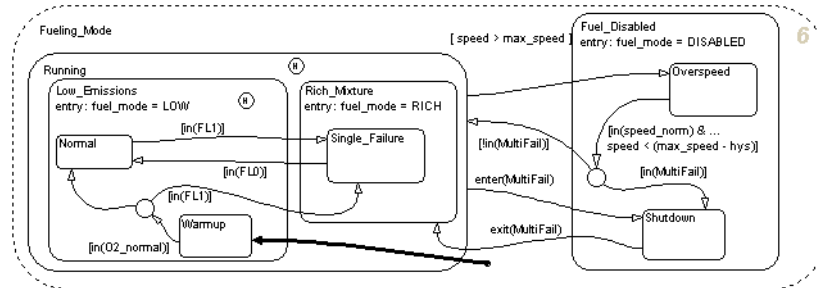


Implicit Event Broadcasts

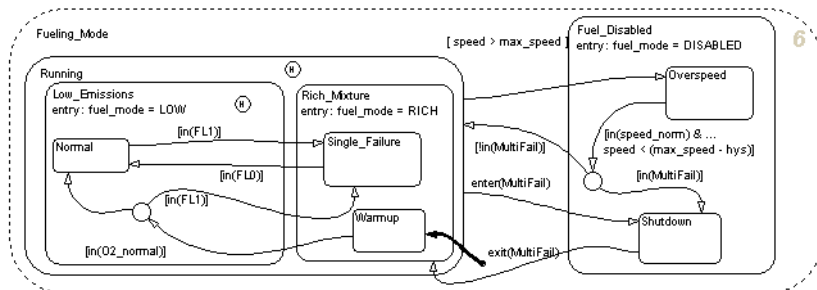
The preceding example shows how the control logic can be represented in a clear and intuitive manner. The Stateflow diagram (or chart) has been developed in a way that allows the user, or a reviewer, to easily understand how the logic is structured. Implicit event broadcasts (such as `enter(multiFail)`) and implicit conditions (`in(FL0)`) make the diagram easy to read and the generated code more efficient.

Modifying the Model

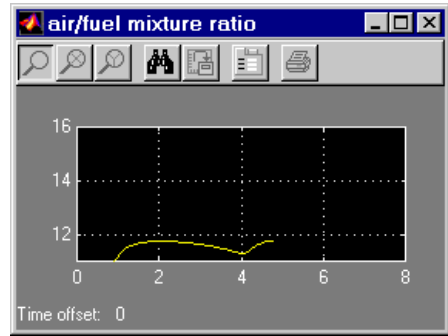
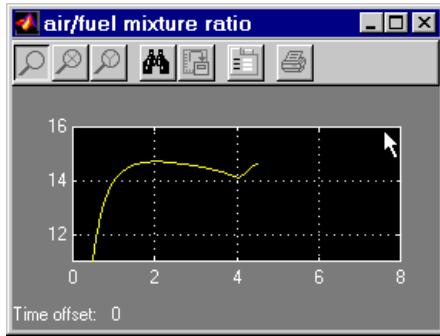
To illustrate how easy it is to modify the model, consider the Warmup fueling state in the fuel control logic. At the moment the fueling is set to the low emissions mode (note the highlighted default transition at the bottom).



You might decide that when the oxygen sensor is warming up, changing the warmup fueling mode to a rich mixture would be beneficial. In the Stateflow chart you can easily achieve this by changing the parent of the Warmup state to the Rich_Mixture state. This is accomplished by enlarging the Rich_Mixture state and moving the Warmup state into it from the Low_Emissions state. This alteration is obvious to all who need to inspect or maintain the code as shown in the following result (note the highlighted default transition at the bottom):



The results of changing the algorithm can be seen in the following graphs of air/fuel mixture ratio for the first few seconds of engine operation after startup. The left graph shows the air/fuel ratio for the unaltered system. The right graph for the altered system shows how the air/fuel ratio stays low in the warming up phase indicating a rich mixture.



Stateflow Notation

You compose Stateflow diagrams with the symbolic objects of Stateflow notation. Learning Stateflow notation is the first step to designing and implementing efficient Stateflow diagrams. Use the following sections to introduce yourself to the objects of Stateflow diagrams:

Overview of Stateflow Objects (p. 2-2)	Stateflow contains objects that are graphical and nongraphical that are organized into an hierarchical structure.
States (p. 2-5)	States are the primary objects of Stateflow. They represent modes of a system.
Transitions (p. 2-12)	A transition is a pathway for a chart or state to change from one mode (state) to another.
Transition Connections (p. 2-17)	Stateflow supports a wide variety of connections with other Stateflow objects.
Default Transitions (p. 2-25)	Default transitions tell Stateflow which of several possible states to enter first for a chart or superstate.
Connective Junctions (p. 2-30)	A connective junction represents a decision point between alternative transition paths.
History Junctions (p. 2-37)	A history junction records the most recently active state of the chart or superstate in which it is placed.
Boxes (p. 2-39)	You use boxes to group parts of a diagram.
Graphical Functions (p. 2-40)	For convenience, Stateflow provides functions that are graphically defined by a flow graph.

Overview of Stateflow Objects



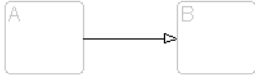


This section describes the different types of available Stateflow objects. Its topics are as follows:

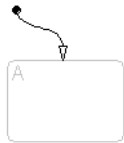



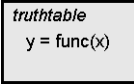

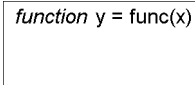

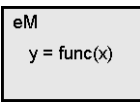



- “Graphical Objects” on page 2-2 — Introduces you to Stateflow’s graphical objects (charts, states, boxes, functions, transitions, and junctions) that you draw in the Stateflow diagram editor.
- “Nongraphical Objects” on page 2-3 — Introduces you to Stateflow’s nongraphical objects (data and events) that are represented textually in the Stateflow diagram editor or in the Stateflow Explorer tool.
- “The Stateflow Data Dictionary of Objects” on page 2-4 — Introduces you to the data dictionary that unites all Stateflow objects (graphical and nongraphical) in a hierarchical database.

While this chapter defines most of these relationships, they are dealt with in more detail in Chapter 3, “Stateflow Semantics,” which describes the behavior of Stateflow charts.

Graphical Objects

The following table gives the name of each graphical object in Stateflow, its appearance when drawn in the diagram editor (Notation), and the toolbar icon used in drawing the object:

Name	Notation	Toolbar Icon
State		
Transition		NA
History Junction		

Name	Notation	Toolbar Icon
Default Transition		
Connective Junction		
Truth Table Function		
Graphical Function		
Embedded MATLAB Function		
Box		

Nongraphical Objects

Stateflow defines event, data, and target objects that do not have graphical representations in the Stateflow diagram editor. However, you can see them in the Stateflow Explorer. See “Using the Model Explorer with Stateflow Objects” on page 14-2.

Event Objects

An event is a Stateflow object that can trigger a whole Stateflow chart or individual actions in a chart. Because Stateflow charts execute by reacting to events, you specify and program events into your charts to control their execution. You can broadcast events to every object in the scope of the object sending the event, or you can send an event to a specific object. You can define explicit events that you specify directly, or you can define implicit events to take place when certain actions are performed, such as entering a state. For a full description of events, see “Adding Events” on page 6-3.

Data Objects

A Stateflow chart stores and retrieves data that it uses to control its execution. Stateflow data resides in its own workspace, but you can also access data that resides externally in the Simulink model or application that embeds the Stateflow machine. When creating a Stateflow model, you must define any internal or external data that you use in the action language of a Stateflow chart. For a full description of data objects, see “Adding Data” on page 6-21.

Target Objects

You build targets in Stateflow to execute the application you program in Stateflow charts and the Simulink model that contains them. A target is a program that executes a Stateflow model or a Simulink model containing a Stateflow machine. You build a simulation target (named `sfun`) to execute a simulation of your model. You build a Real-Time Workshop[®] target (named `rtw`) to execute the Simulink model on a supported processor environment. You build custom targets (with names other than `sfun` or `rtw`) to pinpoint your application to a specific environment. For a full description of target objects in Stateflow and Simulink, see “Overview of Stateflow Targets” on page 12-3.

The Stateflow Data Dictionary of Objects

The data dictionary is a database containing all the information about the graphical and nongraphical objects. Data dictionary entries for graphical objects are created automatically as the objects are added and labeled. You explicitly define nongraphical objects in the data dictionary by using the Stateflow Explorer. The parser evaluates entries and relationships between entries in the data dictionary to verify that the notation is correct.


States

This section describes Stateflow’s primary object, the state. States represent modes of a reactive system. See the following topics for information about states and their properties:

- “What Is a State?” on page 2-5 — Describes states as modes of behavior that can be active or inactive.
- “State Hierarchy” on page 2-6 — Explains Stateflow’s hierarchy of objects and how it is represented for graphical and nongraphical objects.
- “State Decomposition” on page 2-7 — Shows you how states can be exclusive of each other or parallel with each other in behavior in an executing diagram.
- “State Labels” on page 2-9 — Shows you how to specify the name of a state and its actions through its label.

What Is a State?

A *state* describes a mode of a reactive Stateflow chart. States in a Stateflow chart represent these modes. The following table shows the button icon for a drawing a state in the Stateflow diagram editor and a short description.

Name	Button Icon	Description
State		Use a state to depict a mode of the system.

States can be active or inactive. When a state is active, the chart takes on that mode. When a state is inactive, the chart is not in that mode. The activity or inactivity of a chart’s states dynamically changes based on events and conditions. The occurrence of events drives the execution of the Stateflow diagram by making states become active or inactive. At any point in the execution of a Stateflow diagram, there is a combination of active and inactive states.

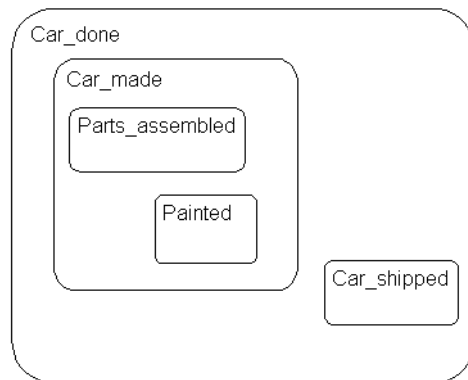
State Hierarchy

States can contain all other Stateflow objects except targets. Stateflow notation supports the representation of graphical object hierarchy in Stateflow diagrams with containment. A state is a superstate if it contains other states. A state is a substate if it is contained by another state. A state that is neither a superstate nor a substate of another state is a state whose parent is the Stateflow diagram itself.

States can also contain nongraphical data and event objects. The hierarchy of this containment is represented in the Explorer tool. Data and event containment is defined by specifying the parent object when you create it. See Chapter 6, “Defining Events and Data” and Chapter 9, “Defining Interfaces to Simulink and MATLAB” for information and examples on representing data and event objects in the Explorer tool.

Representing State Hierarchy Example

In the following example, drawing one state within the boundaries of another state indicates that the inner state is a substate or child of the outer state or superstate and the outer state is the parent of the inner state:



In this example, the Stateflow diagram is the parent of the state `Car_done`. The state `Car_done` is the parent state of the `Car_made` and `Car_shipped` states. The state `Car_made` is also the parent of the `Parts_assembled` and `Painted` states. You can also say that the states `Parts_assembled` and `Painted` are children of the `Car_made` state.

Stateflow hierarchy can also be represented textually, in which the Stateflow diagram is represented by the slash (/) character and each level in the hierarchy of states is separated by the period (.) character. The following is a textual representation of the hierarchy of objects in the preceding example:

- /Car_done
- /Car_done.Car_made
- /Car_done.Car_shipped
- /Car_done.Car_made.Parts_assembled
- /Car_done.Car_made.Painted

State Decomposition

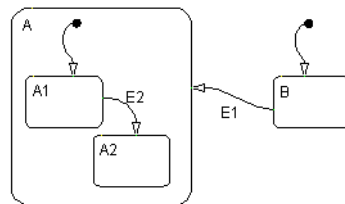
Every state (and chart) has a *decomposition* that dictates what kind of substates it can contain. All substates of a superstate must be of the same type as the superstate's decomposition. Decomposition for a state can be exclusive (OR) or parallel (AND). These types of decomposition are described in the following topics:

- “Exclusive (OR) State Decomposition” on page 2-7
- “Parallel (AND) State Decomposition” on page 2-8

Exclusive (OR) State Decomposition

Exclusive (OR) state decomposition for a superstate (or chart) is indicated when its substates have solid borders. Exclusive (OR) decomposition is used to describe system modes that are mutually exclusive. When a state has exclusive (OR) decomposition, only one substate can be active at a time. The children of exclusive (OR) decomposition parents are OR states.

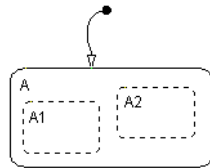
In the following example, either state A or state B can be active. If state A is active, either state A1 or state A2 can be active at any one time.



Parallel (AND) State Decomposition

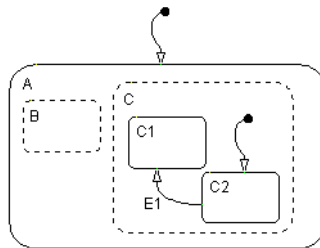
The children of parallel (AND) decomposition parents are parallel (AND) states. Parallel (AND) state decomposition for a superstate (or chart) is indicated when its substates have dashed borders. This representation is appropriate if all states at that same level in the hierarchy are always active at the same time.

In the following example, when state A is active, A1 and A2 are both active at the same time:



The activity within parallel states is essentially independent, as demonstrated in the following example.

In the following example, when state A becomes active, both states B and C become active at the same time. When state C becomes active, either state C1 or state C2 can be active.

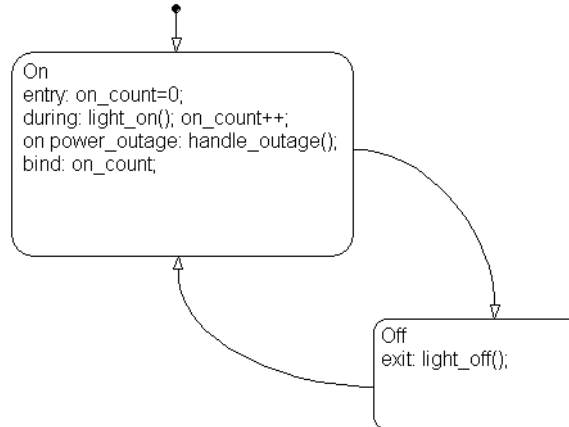


State Labels

The label for a state appears on the top left corner of the state rectangle with the following general format:

```
name /  
entry:entry actions  
during:during actions  
exit:exit actions  
bind:events, data  
on event_name:on event_name actions
```

The following example demonstrates the components of a state label.



Each of the above actions is described in the subtopics that follow. For more information on state actions, see the following topics:

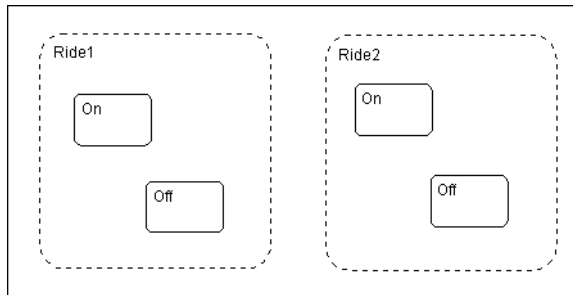
- “Entering, Executing, and Exiting a State” on page 3-20 — Describes how and when entry, during, exit, and *on event_name* actions are taken.
- “State Action Types” on page 7-3 — Gives more detailed descriptions of each type of state action.

State Name

A state label starts with the name of the state followed by an optional / character. In the preceding example, the state names are On and Off. Valid state names consist of alphanumeric characters and can include the underscore (_) character, for example, Transmission or Green_on.

The use of hierarchy provides some flexibility in the naming of states. The name that you enter as part of the label must be unique when preceded by the hierarchy of its ancestor states. The name stored in the data dictionary is the text you enter as the label on the state, preceded by the hierarchy of its parent states separated by periods. Each state can have the same name appear in the label of the state, as long as their full names within the data dictionary are unique. Otherwise, the parser indicates an error.

The following example shows how hierarchy supports unique naming of states.



Each of these states has a unique name because of its location in the hierarchy of the Stateflow diagram. Although the name portion of the label on these states is not unique, when the hierarchy is prefixed to the name in the data dictionary, the result is unique. The full names for these states as seen in the data dictionary are as follows:

- Ride1.On
- Ride1.Off
- Ride2.On
- Ride2.Off

State Actions

After the name, you enter optional action statements for the state with a keyword label that identifies the type of action. You can specify none, some, or all of them. The colon after each keyword is required. The slash following the state name is optional as long as it is followed by a carriage return.

For each type of action, you can enter more than one action by separating each action with a carriage return, semicolon, or a comma. You can specify actions for more than one event by adding additional *on event_name* lines for different events.

If you enter the name and slash followed directly by actions, the actions are interpreted as entry action(s). This shorthand is useful if you are only specifying entry actions.

Entry Action. Preceded by the prefix `entry` or `en` for short. In the preceding example, state `On` has entry action `on_count=0`. This means that the value of `on_count` is reset to 0 whenever state `On` becomes active (entered).

During Action. Preceded by the prefix `during` or `du` for short. In the preceding label example, state `On` has two during actions, `light_on()` and `on_count++`. These actions are executed whenever state `On` is already active and any event occurs.

Exit Action. Preceded by the prefix `exit` or `ex` for short. In the preceding label example, state `Off` has the exit action `light_off()`. If the state `Off` is active, but becomes inactive (exited), this action is executed.

On Event_Name Action. Preceded by the prefix `on event_name`, where *event_name* is a unique event. In the preceding label example, state `On` has an `on power_outage` action. If state `On` is active and the event `power_outage` occurs, the action `handle_outage()` is executed.

Bind Action. Preceded by the prefix `bind`. In the preceding label example, the data `on_count` is bound to the state `On`. This means that only the state `On` or a child of `On` can change the value of `on_count`. Other states, such as the state `Off`, can use `on_count` in its actions, but it cannot change its value in doing so.

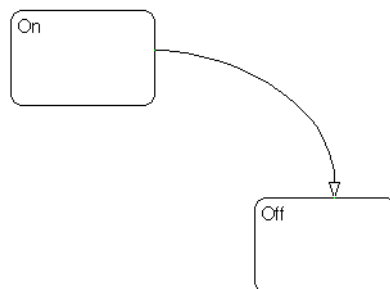
Transitions

You model the behavior of reactive systems by changing from one state to another through an object called a transition. This section contains the following topics on the transitions in Stateflow diagrams:

- “What Is a Transition?” on page 2-12 — Gives a description and examples of transitions and transition segments.
- “Transition Label Notation” on page 2-14 — Gives the syntax for a transition label, which can include conditions and actions. Also defines a condition and the different types of actions in a transition label.
- “Valid Transitions” on page 2-16 — Describes the situations under which transitions become valid and are taken during execution of the Stateflow diagram.

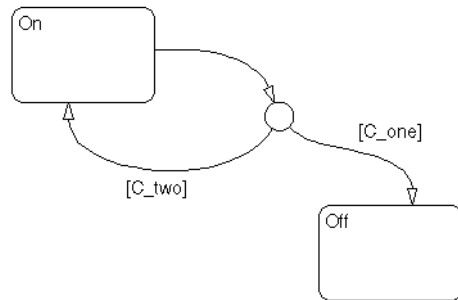
What Is a Transition?

A *transition* is a curved line with an arrowhead that links one graphical object to another. In most cases, a transition represents the passage of the system from one mode (state) object to another. A transition is attached to a source and a destination object. The *source* object is where the transition begins and the *destination* object is where the transition ends. This is an example of a transition from a source state, On, to a destination state, Off.



Junctions divide a transition into transition segments. In this case, a full transition consists of the segments taken from the origin to the destination state. Each segment is evaluated in the process of determining the validity of a full transition.

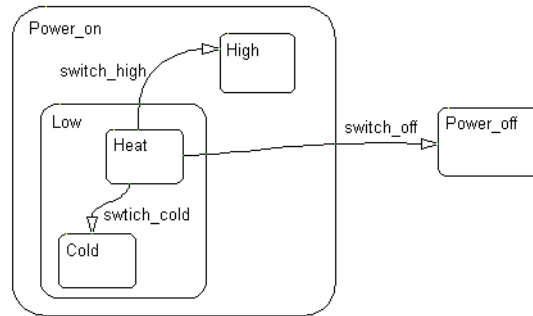
The following example has two segmented transitions: one from state On to state Off, and the other from state On to itself:



A default transition is a special type of transition that has no source object. See “Default Transitions” on page 2-25 for a description of a default transition.

Transition Hierarchy

Transitions cannot contain other objects like states can. However, transitions are contained by states. A transition’s hierarchy is described in terms of the transition’s parent, source, and destination. The parent is the lowest level that contains the source and destination of the transition. Consider the parents for the transitions in the following example:



The following table resolves the parentage of each transition in the preceding example. The Stateflow diagram is represented by the / character. Each level in the hierarchy of states is separated by the period (.) character.

Transition Label	Transition Parent	Transition Source	Transition Destination
switch_off	/	/Power_on.Low.Heat	/Power_off
switch_high	/Power_on	/Power_on.Low.Heat	/Power_on.High
switch_cold	/Power_on.Low	/Power_on.Low.Heat	/Power_on.Low.Cold

Transition Label Notation

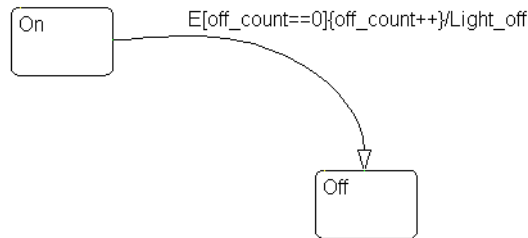
A transition is characterized by its *label*. The label can consist of an event, a condition, a condition action, and/or a transition action. The ? character is the default transition label. Transition labels have the following general format:

event[condition]{condition_action}/transition_action

You replace the names for *event*, *condition*, *condition_action*, and *transition_action* with appropriate contents as shown in the example “Transition Label Example” on page 2-15. Each part of the label is optional.

Transition Label Example

Use the transition label in the following example to understand the parts of a transition label.



Event Trigger. specifies an event that causes the transition to be taken, provided the condition, if specified, is true. Specifying an event is optional. The absence of an event indicates that the transition is taken upon the occurrence of any event. Multiple events are specified using the OR logical operator (|).

In the preceding example, the broadcast of event E triggers the transition from On to Off provided the condition [off_count==0] is true.

Condition. specifies a boolean expression that, when true, validates a transition to be taken for the specified event trigger. Enclose the condition in square brackets ([]). See “Conditions” on page 7-8 for information on the condition notation.

In the preceding example, the condition [off_count==0] must evaluate as true for the condition action to be executed and for the transition from the source to the destination to be valid.

Condition Action. A *condition action* follows the condition for a transition and is enclosed in curly braces ({}). It is executed as soon as the condition is evaluated as true and before the transition destination has been determined to be valid. If no condition is specified, an implied condition evaluates to true and the condition action is executed.

In the preceding example, if the condition [off_count==0] is true, the condition action off_count++ is immediately executed.

Transition Action. The transition action is executed after the transition destination has been determined to be valid provided the condition, if specified, is true. If the transition consists of multiple segments, the transition action is only executed when the entire transition path to the final destination is determined to be valid. Precede the transition action with a backslash.

In the preceding example, if the condition `[off_count==0]` is true, and the destination state `Off` is valid, the transition action `Light_off` is executed.

Valid Transitions

In most cases, a transition is valid when the source state of the transition is active and the transition label is valid. Default transitions are slightly different because there is no source state. Validity of a default transition to a substate is evaluated when there is a transition to its superstate, assuming the superstate is active. This labeling criterion applies to both default transitions and general case transitions. The following are possible combinations of valid transition labels.

Transition Label	Is Valid If...
Event only	That event occurs
Event and condition	That event occurs and the condition is true
Condition only	Any event occurs and the condition is true
Action only	Any event occurs
Not specified	Any event occurs

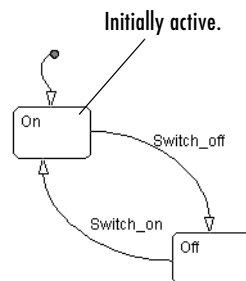
Transition Connections

Stateflow notation supports a wide variety of transition connections, which are demonstrated by the examples in the following sections:

- “Transitions to and from Exclusive (OR) States” on page 2-17 — Describes the behavior of transitions that take place between exclusive (OR) states.
- “Transitions to and from Junctions” on page 2-18 — Describes the behavior of transitions that take place between exclusive (OR) states and a connective junction.
- “Transitions to and from Exclusive (OR) Superstates” on page 2-18 — Describes the behavior of transitions that take place between a state and a superstate.
- “Transitions to and from Substates” on page 2-20 — Describes the behavior of a transition from one substate to another.
- “Self-Loop Transitions” on page 2-21 — Describes the behavior of a self-loop transition through a connective junction.
- “Inner Transitions” on page 2-21 — Describes several examples that demonstrate the behavior of inner transitions to substates and a history junction and why you want to use them.

Transitions to and from Exclusive (OR) States

This example shows simple transitions to and from exclusive (OR) states.

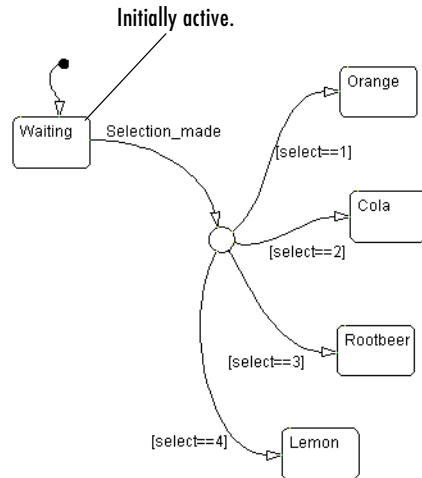


The transition `On→Off` is valid when state `On` is active and the event `Switch_off` occurs. The transition `Off→On` is valid when state `Off` is active and event `Switch_on` occurs.

See “Transitions to and from Exclusive (OR) States Examples” on page 3-31 for more information on the semantics of this notation.

Transitions to and from Junctions

This example shows transitions to and from a connective junction.

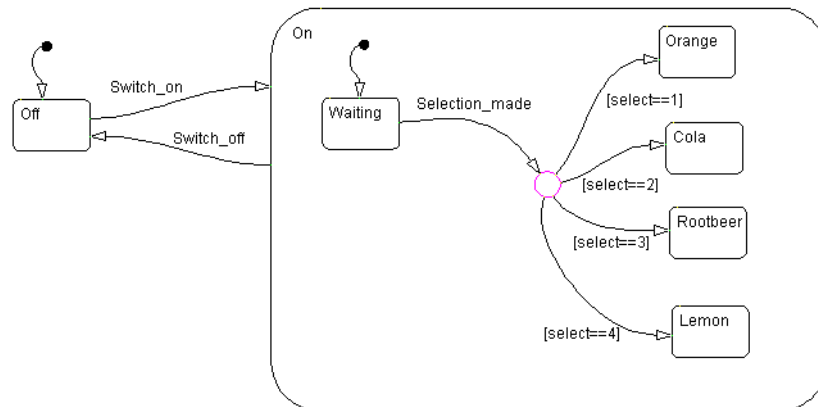


This is a Stateflow diagram of a soda machine. The Stateflow diagram is called when the external event `Selection_made` occurs. The Stateflow diagram awakens with the `Waiting` state active. The `Waiting` state is a common source state. When the event `Selection_made` occurs, the Stateflow diagram transitions from the `Waiting` state to one of the other states based on the value of the variable `select`. One transition is drawn from the `Waiting` state to the connective junction. Four additional transitions are drawn from the connective junction to the four possible destination states.

See “Transitions from a Common Source to Multiple Destinations Example” on page 3-65 for more information on the semantics of this notation.

Transitions to and from Exclusive (OR) Superstates

This example shows transitions to and from an exclusive (OR) superstate and the use of a default transition.



This is an expansion of the soda machine Stateflow diagram that includes the initial example of the **On** and **Off** exclusive (OR) states. **On** is now a superstate containing the **Waiting** and soda choices states. The transition **Off**→**On** is valid when state **Off** is active and event **Switch_on** occurs. Now that **On** is a superstate, this is an explicit transition to the **On** superstate.

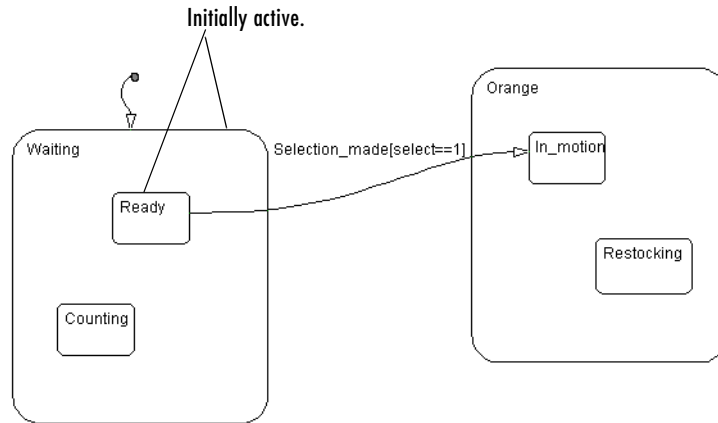
For a transition to a superstate to be a valid, the destination substate must be implicitly defined. The destination substate for **On** is implicitly defined by making the **Waiting** substate the destination state of a default transition. This notation defines that the resultant transition is made from the **Off** state to the state **On.Waiting**.

The transition from **On** to **Off** is valid when state **On** is active and event **Switch_off** occurs. However, when the **Switch_off** event occurs, a transition to the **Off** state must take place no matter which of the substates of **On** is active. This top-down approach simplifies the Stateflow diagram by looking at the transitions out of the superstate without considering all the details of states and transitions within the superstate.

See “Default Transition Examples” on page 3-43 for more information on the semantics of this notation.

Transitions to and from Substates

The following example shows transitions to and from exclusive (OR) substates.

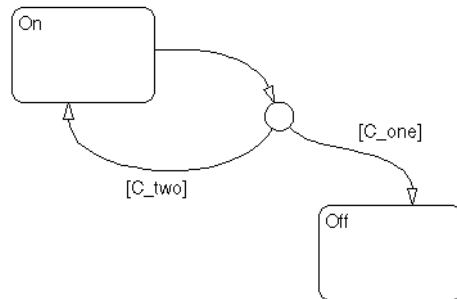


This Stateflow diagram shows a transition from one OR substate to another OR substate: the transition from `Waiting.Ready` to `Orange`. The transition to the state `In_motion` is valid when state `Waiting.Ready` is active and the event `Selection_made` occurs, providing that the variable `select` equals 1. This transition defines an explicit exit from the `Waiting.Ready` state and an implicit exit from the `Waiting` superstate. On the destination side, this transition defines an implicit entry into the `Orange` superstate and an explicit entry into the `Orange.In_motion` substate.

See “Transitioning from a Substate to a Substate with Events Example” on page 3-35 for more information on the semantics of this notation.

Self-Loop Transitions

A transition segment from a state to a connective junction that has an outgoing transition segment from the connective junction back to the state is a self-loop transition as shown in the following example:



See these sections for examples of self-loop transitions:

- “Connective Junction—Self-Loop Example” on page 2-33
See “Self-Loop Transition Example” on page 3-61 for information on the semantics of this notation.
- “Connective Junction and For Loops Example” on page 2-33
See “For Loop Construct Example” on page 3-62 for information on the semantics of this notation.

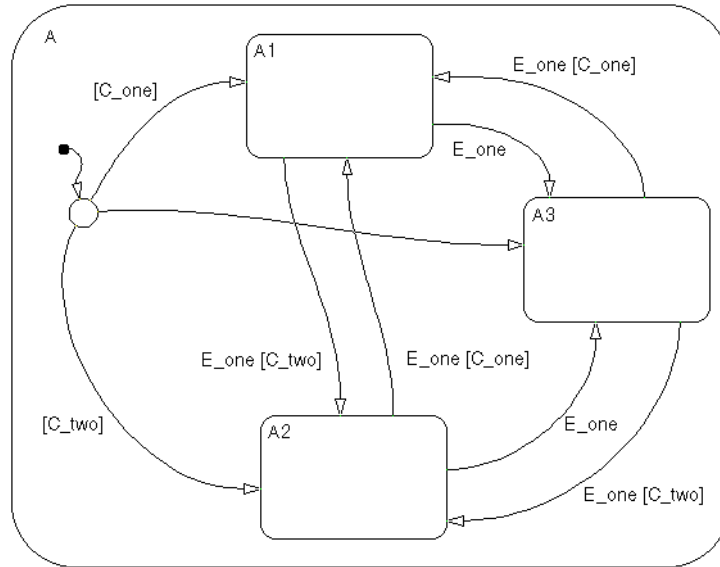
Inner Transitions

An *inner transition* is a transition that does not exit the source state. Inner transitions are most powerful when defined for superstates with exclusive (OR) decomposition. Use of inner transitions can greatly simplify a Stateflow diagram, as shown by the following examples:

- “Before Using an Inner Transition” on page 2-22
- “After Using an Inner Transition to a Connective Junction” on page 2-23
- “Using an Inner Transition to a History Junction” on page 2-24

Before Using an Inner Transition

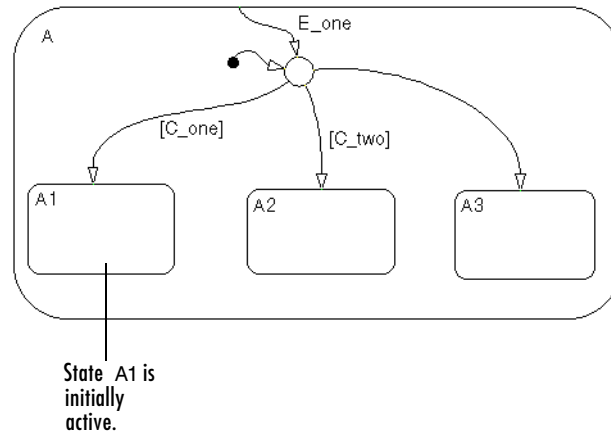
This is an example of a Stateflow diagram that could be simplified by using an inner transition.



Any event occurs and awakens the Stateflow diagram. The default transition to the connective junction is valid. The destination of the transition is determined by [C_one] and [C_two]. If [C_one] is true, the transition to A1 is true. If [C_two] is true, the transition to A2 is valid. If neither [C_one] nor [C_two] is true, the transition to A3 is valid. The transitions among A1, A2, and A3 are determined by E_one, [C_one], and [C_two].

After Using an Inner Transition to a Connective Junction

This example simplifies the preceding example using an inner transition to a connective junction.



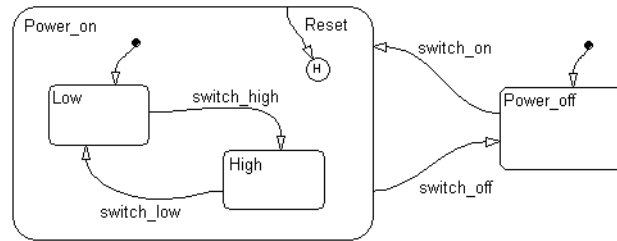
Any event occurs and awakens the Stateflow diagram. The default transition to the connective junction is valid. The destination of the transitions is determined by [C_one] and [C_two].

The Stateflow diagram is simplified by using an inner transition in place of the many transitions among all the states in the original example. If state A is already active, the inner transition is used to reevaluate which of the substates of state A is to be active. When event E_one occurs, the inner transition is potentially valid. If [C_one] is true, the transition to A1 is valid. If [C_two] is true, the transition to A2 is valid. If neither [C_one] nor [C_two] is true, the transition to A3 is valid. This solution is much simpler than the previous one.

See “Processing the First Event with an Inner Transition to a Connective Junction” on page 3-53 for more information on the semantics of this notation.

Using an Inner Transition to a History Junction

This example shows an inner transition to a history junction.



State `Power_on.High` is initially active. When event `Reset` occurs, the inner transition to the history junction is valid. Because the inner transition is valid, the currently active state, `Power_on.High`, is exited. When the inner transition to the history junction is processed, the last active state, `Power_on.High`, becomes active (is reentered). If `Power_on.Low` was active under the same circumstances, `Power_on.Low` would be exited and reentered as a result. The inner transition in this example is equivalent to drawing an outer self-loop transition on both `Power_on.Low` and `Power_on.High`.

See “Use of History Junctions Example” on page 2-37 for another example using a history junction.

See “Inner Transition to a History Junction Example” on page 3-56 for more information on the semantics of this notation.

Default Transitions

You use default transitions to tell Stateflow which one of several states you enter when you first enter a chart or a state that has substates. See the following topics for information on default transitions:

- “What Is a Default Transition?” on page 2-25 — Defines and describes a default transition.
- “Drawing Default Transitions” on page 2-25 — Gives you the steps for drawing a default transition.
- “Labeling Default Transitions” on page 2-26 — Gives you the steps for editing the label of a default transition.
- “Default Transition Examples” on page 2-26 — Provides some examples of default transitions.

What Is a Default Transition?


Default transitions are primarily used to specify which exclusive (OR) state is to be entered when there is ambiguity among two or more neighboring exclusive (OR) states. They have a destination but no source object. For example, default transitions specify which substate of a superstate with exclusive (OR) decomposition the system enters by default in the absence of any other information such as a history junction. Default transitions are also used to specify that a junction should be entered by default.

Drawing Default Transitions

Click the **Default transition** button in the toolbar, and click a location in the drawing area close to the state or junction you want to be the destination for the default transition. Drag the mouse to the destination object to attach the default transition. In some cases it is useful to label default transitions.

One of the most common Stateflow programming mistakes is to create multiple exclusive (OR) states without a default transition. In the absence of the default transition, there is no indication of which state becomes active by default. Note that this error is flagged when you simulate the model using the Debugger with the **State Inconsistencies** option enabled.

This table shows the button icon and briefly describes a default transition.

Name	Button Icon	Description
Default transition		Use a default transition to indicate, when entering this level in the hierarchy, which object becomes active by default.

Labeling Default Transitions

In some circumstances, you might want to label default transitions. You can label default transitions as you would other transitions. For example, you might want to specify that one state or another should become active depending upon the event that has occurred. In another situation, you might want to have specific actions take place that are dependent upon the destination of the transition.

Note When labeling default transitions, take care to ensure that there is always at least one valid default transition. Otherwise, a Stateflow chart can transition into an inconsistent state.

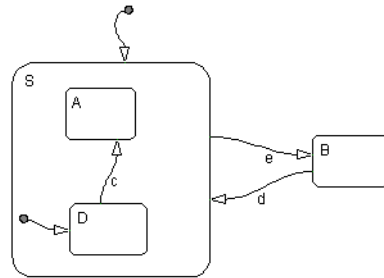
Default Transition Examples

The following examples show the use of default transitions in Stateflow diagrams:

- “Default Transition to a State Example” on page 2-27
- “Default Transition to a Junction Example” on page 2-28
- “Default Transition with a Label Example” on page 2-28

Default Transition to a State Example

This example shows a use of default transitions.



When the Stateflow diagram is first awakened, it must decide whether to activate state S or state B since they are exclusive (OR) states. The answer is given by the default transition to superstate S, which is taken if valid. Because there are no conditions on this default transition, it is taken.

State S, which is now active, has two substates, A and D. Which substate becomes active? Only one of them can be active because they are exclusive (OR) states. The answer is given by the default transition to substate D, which is taken if valid. Because there are no conditions on this default transition, it is taken.

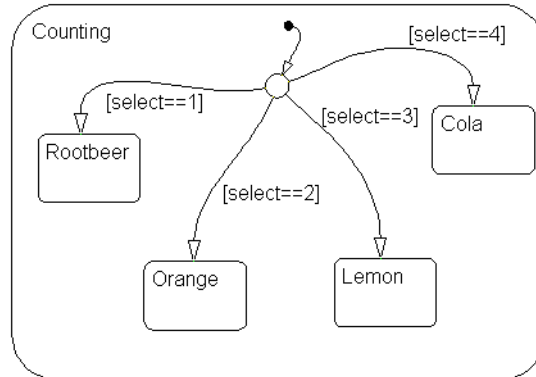
Suppose at a different execution point the Stateflow diagram is awakened by the occurrence of event d and state B is active. The transition from state B to state S is valid. When the system enters state S, it enters substate D because the default transition is defined.

See “Default Transition Examples” on page 3-43 for more information on the semantics of this notation.

The default transitions are required for the Stateflow diagram to execute. Without the default transition to state S, when the Stateflow diagram is awakened, none of the states becomes active. You can detect this situation at run-time by checking for state inconsistencies. See “Controlling Animation in the Debugging Window” on page 13-7 for more information.

Default Transition to a Junction Example

This example shows a default transition to a connective junction.

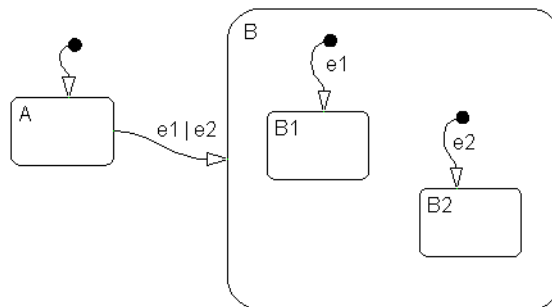


In this example, the default transition to the connective junction defines that upon entering the Counting state, the destination is determined by the condition on each transition segment.

See “Default Transition to a Junction Example” on page 3-44 for more information on the semantics of this notation.

Default Transition with a Label Example

The following example shows the labeling of default transitions.



If state A is initially active and either e1 or e2 occurs, the transition from state A to superstate B is valid. The substates B1 and B2 both have default transitions. The default transitions are labeled to specify the event that triggers the transition. If event e1 occurs, the transition A to B1 is valid. If event e2 occurs, the transition A to B2 is valid.

See “Labeled Default Transitions Example” on page 3-47 for more information on the semantics of this notation.

Connective Junctions

A connective junction represents a decision point between alternate transition paths taken for a single transition. See the following topics for more information on connective junctions:

- “What Is a Connective Junction?” on page 2-30 — Defines a connective junction and describes some of the transition constructs it can be used to create.
- “Flow Diagram Notation with Connective Junctions” on page 2-31 — Introduces you to the concept of using connective junctions to create flow diagrams that create the logic of common code structures and provides extensive examples of their use.

What Is a Connective Junction?

The connective junction enables representation of different possible transition paths for a single transition. Connective junctions are used to help represent the following:

- Variations of an if - then -else decision construct, by specifying conditions on some or all of the outgoing transitions from the connective junction
- A self-loop transition back to the source state if none of the outgoing transitions is valid
- Variations of a for loop construct, by having a self-loop transition from the connective junction back to itself
- Transitions from a common source to multiple destinations
- Transitions from multiple sources to a common destination
- Transitions from a source to a destination based on common events

Note An event cannot trigger a transition from a connective junction to a destination state.

See “Connective Junction Examples” on page 3-58 for a summary of the semantics of connective junctions.

Flow Diagram Notation with Connective Junctions

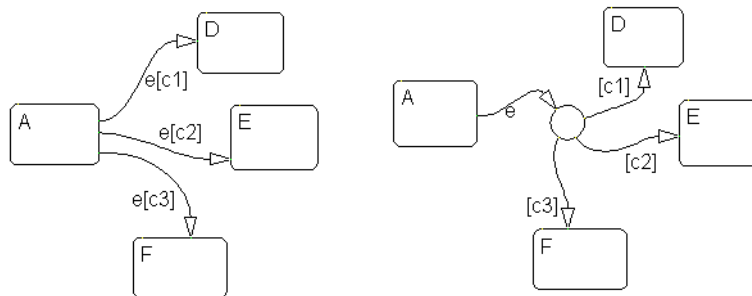
Flow diagram notation uses connective junctions to represent common code structures like for loops and if-then-else constructs without the use of states. And by reducing the number of states in your Stateflow diagrams, flow diagram notation produces more efficient generated code that helps optimize memory use.

Flow diagram notation employs combinations of the following:

- Transitions to and from connective junctions
- Self-loops to connective junctions
- Inner transitions to connective junctions

Flow diagram notation, states, and state-to-state transitions seamlessly coexist in the same Stateflow diagram. The key to representing flow diagram notation is in the labeling of the transitions (specifically the use of action language) as shown by the following examples.

Connective Junction with All Conditions Specified Example



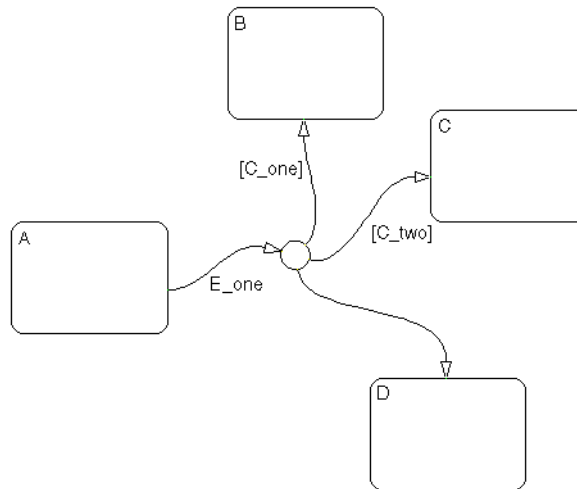
In the example on the left, if state A is active when event e occurs, the transition from state A to any of states D, E, or F takes place if one of the conditions [c1], [c2], or [c3] is met.

In the equivalent representation on the right, a transition from the source state to a connective junction is labeled by the event. Transitions from the connective junction to the destination states are labeled by the conditions. If state A is active when event e occurs, the transition from A to the connective junction occurs first. The transition from the connective junction to a destination state follows based on which of the conditions $[c1]$, $[c2]$, or $[c3]$ is true. If none of the conditions is true, no transition occurs and state A remains active.

See “If-Then-Else Decision Construct Example” on page 3-59 for more information on the semantics of this notation.

Connective Junction with One Unconditional Transition Example

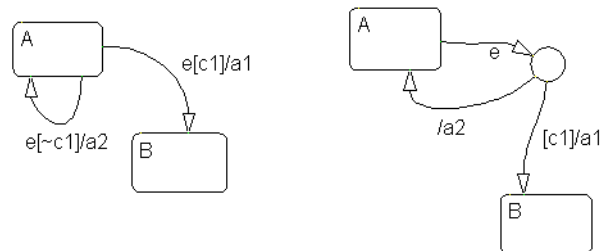
The transition A to B is valid when A is active, event E_one occurs, and $[C_one]$ is true. The transition A to C is valid when A is active, event E_one occurs, and $[C_two]$ is true. Otherwise, given A is active and event E_one occurs, the transition A to D is valid. If you do not explicitly specify condition $[C_three]$, it is implicit that the transition condition is not $[C_one]$ and not $[C_two]$.



See “If-Then-Else Decision Construct Example” on page 3-59 for information on the semantics of this notation.

Connective Junction — Self-Loop Example

In some situations, the transition event occurs but a condition is not met. No transition is taken, but an action is generated. You can represent this situation by using a connective junction or a self-loop transition (transition from state to itself).



In the example on the left, if State A is active and event e occurs and the condition $[c1]$ is met, the transition from A to B is taken, generating action $a1$. The transition from state A to state A is valid if event e occurs and $[c1]$ is not true. In this self-loop transition, the system exits and reenters state A, and executes action $a2$.

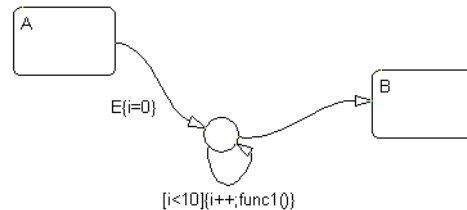
In the equivalent representation on the right, the use of a connective junction makes it unnecessary to specify the implied condition $[~c1]$ explicitly.

See “Self-Loop Transition Example” on page 3-61 for more information on the semantics of this notation.

Connective Junction and For Loops Example

This example shows a combination of flow diagram notation and state transition notation. Self-loop transitions to connective junctions can be used to represent for loop constructs.

In state A, event E occurs. The transition from state A to state B is valid if the conditions along the transition path are true. The first segment of the transition does not have a condition, but does have a condition action. The condition action, $\{i=0\}$, is executed. The condition on the self-loop transition is evaluated as true and the condition actions $\{i++;func1()\}$ execute. The condition actions execute until the condition $[i<10]$ is false. The condition actions on both the first segment and the self-loop transition to the connective junction effectively execute a for loop (for i values 0 to 9 execute $func1()$). The for loop is executed outside the context of a state. The remainder of the path is evaluated. Because there are no conditions, the transition completes at the destination, state B.



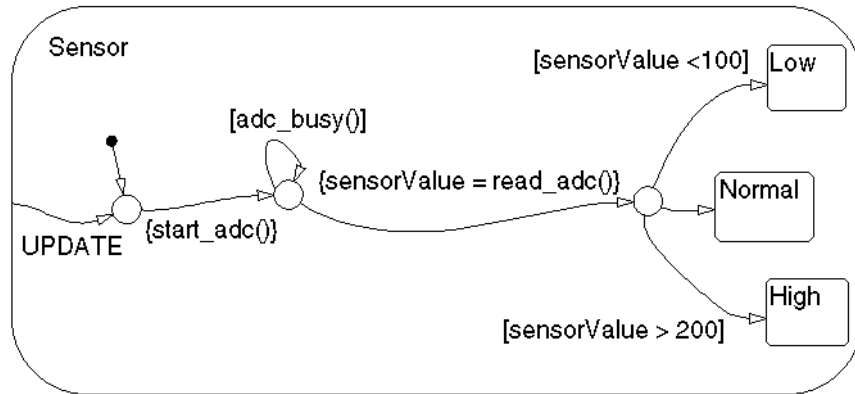
See “For Loop Construct Example” on page 3-62 for information on the semantics of this notation.

Flow Diagram Notation Example

This example shows a real-world use of flow diagram notation and state transition notation. This Stateflow diagram models an 8-bit analog-to-digital converter (ADC).

Consider the case when state `Sensor.Low` is active and event `UPDATE` occurs. The inner transition from `Sensor` to the connective junction is valid. The next transition segment has a condition action, $\{start_adc()\}$, which initiates a reading from the ADC. The self-loop on the second connective junction repeatedly tests the condition $[adc_busy()]$. This condition evaluates as true once the reading settles (stabilizes) and the loop completes. This self-loop transition is used to introduce the delay needed for the ADC reading to settle. The delay could have been represented by another state with some sort of counter. Using flow notation in this example avoids an unnecessary use of a state and produces more efficient code.

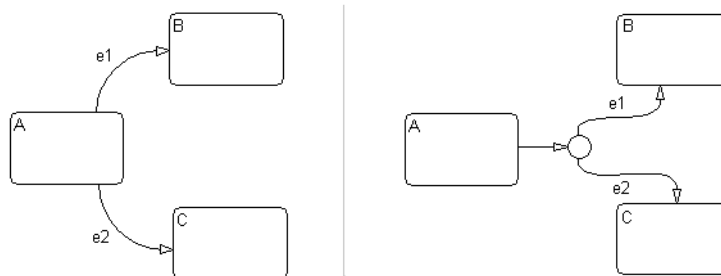
The next transition segment condition action, `{sensorValue=read_adc()}`, puts the new value read from the ADC in the data object `sensorValue`. The final transition segment is determined by the value of `sensorValue`. If `[sensorValue <100]` is true, the state `Sensor.Low` is the destination. If `[sensorValue >200]` is true, the state `Sensor.High` is the destination. Otherwise, state `Sensor.Normal` is the destination state.



See “Flow Diagram Notation Example” on page 3-63 for information on the semantics of this notation.

Connective Junction from a Common Source to Multiple Destinations Example

Transitions A to B and A to C share a common source state A. An alternative representation uses one arrow from A to a connective junction, and multiple arrows labeled by events from the junction to the destination states B and C.



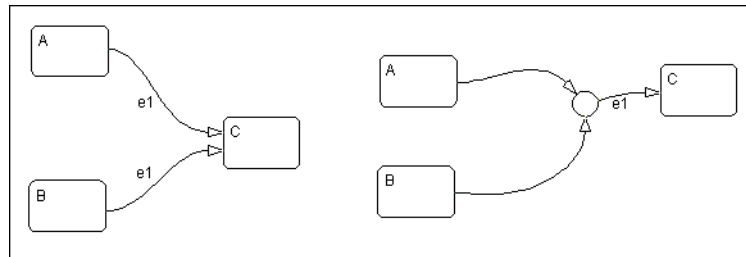
See “Transitions from a Common Source to Multiple Destinations Example” on page 3-65 for information on the semantics of this notation.

Connective Junction Common Events Example

Suppose, for example, that when event e_1 occurs, the system, whether it is in state A or B, transfers to state C. Suppose that transitions A to C and B to C are triggered by the same event e_1 , so that both destination state and trigger event are common to the transitions. There are three ways to represent this:

- By drawing transitions from A and B to C, each labeled with e_1
- By placing A and B in one superstate S, and drawing one transition from S to C, labeled with e_1
- By drawing transitions from A and B to a connective junction, then drawing one transition from the junction to C, labeled with e_1

This Stateflow diagram shows the simplification using a connective junction.



See “Transitions from a Source to a Destination Based on a Common Event Example” on page 3-68 for information on the semantics of this notation.

History Junctions

History junctions record the previously active state of the state in which they are resident. See the following sections for information on history junctions:

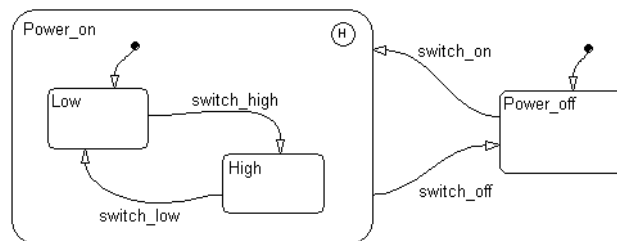
- “What Is a History Junction?” on page 2-37 — Presents a defining description and example of a history junction.
- “History Junctions and Inner Transitions” on page 2-38 — Shows how an inner transition to a history junction can immediately exit and reenter a state as a special behavior.

What Is a History Junction?

A history junction is used to represent historical decision points in the Stateflow diagram. The decision points are based on historical data relative to state activity. Placing a history junction in a superstate indicates that historical state activity information is used to determine the next state to become active. The history junction applies only to the level of the hierarchy in which it appears.

Use of History Junctions Example

The following example uses a history junction:



Superstate `Power_on` has a history junction and contains two substates. If state `Power_off` is active and event `switch_on` occurs, the system could enter either `Power_on.Low` or `Power_on.High`. The first time superstate `Power_on` is entered, substate `Power_on.Low` is entered because it has a default transition. At some point afterward, if state `Power_on.High` is active and event `switch_off` occurs, superstate `Power_on` is exited and state `Power_off` becomes active. Then event `switch_on` occurs. Since `Power_on.High` was the last active state, it becomes active again. After the first time `Power_on` becomes active, the choice between entering `Power_on.Low` or `Power_on.High` is determined by the history junction.

See “Default Transition and a History Junction Example” on page 3-45 for more information on the semantics of this notation.

History Junctions and Inner Transitions

By specifying an inner transition to a history junction, you can specify that, based on a specified event and/or condition, the active state is to be exited and then immediately reentered.

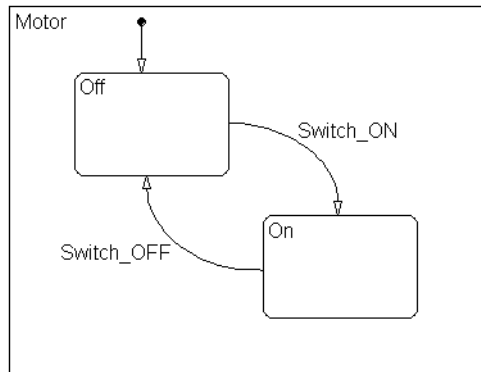
See “Using an Inner Transition to a History Junction” on page 2-24 for an example of this notation.

See “Inner Transition to a History Junction Example” on page 3-56 for more information on the semantics of this notation.

Boxes

You use boxes to graphically organize your diagram. Beyond this organizational use, boxes contribute little to how Stateflow diagrams execute.

The following is an example of a Stateflow Box object:

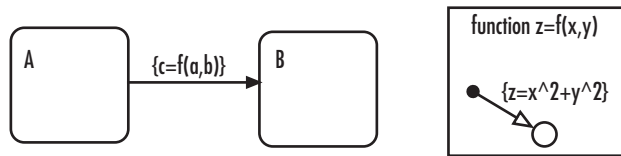


In this example, a box labeled Motor groups all the objects needed to control a simple motor. There can be many more objects displayed on the Stateflow chart along with this box, but now everything needed to control the motor is kept separate.

Graphical Functions

A *graphical function* is a function defined graphically by a flow diagram that provides convenience and power to Stateflow action language.

The following example shows a graphical function side by side in a Stateflow diagram with the transition that calls it:



In this example the function $z = f(x, y)$ is called in the condition action of the transition from state A to state B. The function is defined using symbols that are valid only within the function itself. The function is called using data objects available to states A and B and their parent states (if any).

Graphical functions are similar to textual functions such as MATLAB and C functions in the following ways:

- Graphical functions can accept arguments and return results.
- You can invoke graphical functions in transition and state actions.

Unlike C and MATLAB functions, however, graphical functions are full-fledged Stateflow graphical objects. You use the Stateflow editor to create them and they reside in your Stateflow model along with the diagrams that invoke them. This makes graphical functions easier to create, access, and manage than textual custom code functions, whose creation requires external tools, and whose definition resides separately from the model.

Stateflow Semantics

Stateflow semantics describe how the notation in Stateflow charts is interpreted and implemented into a behavior. Knowledge of Stateflow semantics is important to make sound Stateflow diagram design decisions for code generation. Different notations result in different behavior during simulation and generated code execution. This chapter contains the following sections:

Executing an Event (p. 3-3)	Describes the behavior of events that drive Stateflow chart execution.
Executing a Chart (p. 3-6)	Describes how charts become active, execute, and become inactive.
Executing a Transition (p. 3-7)	Describes the processes for grouping and executing a transition.
Transition Testing Order (p. 3-10)	Describes different ways and rules for ordering transitions.
Entering, Executing, and Exiting a State (p. 3-20)	Describes how states become active, execute, and become inactive.
Early Return Logic for Event Broadcasts (p. 3-26)	Describes the logic employed when events interrupt the normal execution behavior of Stateflow charts.
Semantic Examples (p. 3-29)	A list of the semantic (behavioral) examples provided in the remainder of this chapter.
Transitions to and from Exclusive (OR) States Examples (p. 3-31)	Examples that describe the behavior of transitions that exit and enter exclusive (OR) states.
Condition Action Examples (p. 3-37)	Examples that describe the behavior of Stateflow diagrams using condition actions.
Default Transition Examples (p. 3-43)	Examples that describe the behavior of Stateflow diagrams using default transitions.
Inner Transition Examples (p. 3-49)	Examples that describe the behavior of Stateflow diagrams using inner transitions.

Connective Junction Examples (p. 3-58)	Examples that describe the behavior of Stateflow diagrams using connective junctions.
Event Actions in a Superstate Example (p. 3-71)	Example that describes the behavior of Stateflow diagrams using event actions.
Parallel (AND) State Examples (p. 3-73)	Example Stateflow diagrams demonstrating the behavior of parallel (AND) states.
Directed Event Broadcasting Examples (p. 3-85)	Example Stateflow diagrams demonstrating how to use directed event broadcasting.

Executing an Event

A Stateflow chart executes only in response to an event. This occurs on two levels. First, Simulink updates the chart, which awakens it for execution. Second, once the chart is awakened, it continues to respond to events until there are no more events. The chart then goes to sleep. When another event occurs, the chart is awakened (from sleep) to respond to the event.

Because Stateflow runs on a single thread, actions that take place based on an event are atomic to that event. This means that all activity caused by the event in the chart is completed before returning to whatever activity was taking place prior to reception of the event. Once action is initiated by an event, it is completed unless interrupted by an early return.

See the following topics to continue with the behavior of events:

- “Sources for Stateflow Events” on page 3-3 — Describes the sources for events that drive Stateflow diagrams.
- “Processing Events” on page 3-4 — Describes how Stateflow diagrams handle events.

Sources for Stateflow Events

Stateflow charts are awakened by Simulink events. From the Stateflow perspective, this is an event like any other event, with the exception that it comes from Simulink. After the chart is made active, it can respond to the occurrence of this event in its action language.

You can also use events to control the processing of your Stateflow diagrams by broadcasting events in the action language associated with states and transitions in the chart itself. For the mechanics of broadcasting events in action language, see “Broadcasting Events in Actions” on page 7-44. For examples using event broadcasting and directed event broadcasting, see the following:

- “Condition Actions to Broadcast Events to Parallel (AND) States Example” on page 3-40
- “Cyclic Behavior to Avoid with Condition Actions Example” on page 3-41
- “Event Broadcast State Action Example” on page 3-73
- “Event Broadcast Transition Action with a Nested Event Broadcast Example” on page 3-77

- “Event Broadcast Condition Action Example” on page 3-81
- “Directed Event Broadcasting” on page 7-46

Before you can broadcast events in the action language of a Stateflow chart, you must first add them. You can do this in the **Add** menu of the Stateflow diagram editor or through the **Model Explorer**. See “Adding Data to the Data Dictionary” on page 6-21 for a description of the ways that you add events in Stateflow.

Events have hierarchy (a parent) and scope. The parent and scope together define a range of access to events. It is primarily the event’s parent that determines who can trigger on the event (has receive rights). See the **Name** and **Parent** fields for an event in “Setting Event Properties in the Event Dialog” on page 6-7 for more information.

Processing Events

When an event occurs, it is processed from the top or root of the Stateflow diagram down through the hierarchy of the diagram. At each level in the hierarchy, any during and on *event_name* actions for the active state are executed and completed and then a check for the existence of a valid explicit or implicit transition among the children of the state is conducted. The examples in this chapter demonstrate the top-down processing of events.

All events, with the exception of the output edge trigger to Simulink (see the following note), have the following execution in a Stateflow diagram:

- 1 If the *receiver* of the event is active, then it is executed (see “Executing an Active Chart” on page 3-6 and “Executing an Active State” on page 3-22). (The event *receiver* is the parent of the event unless the event was explicitly directed to a *receiver* using the `send()` function.)
- 2 If the receiver of the event is not active, nothing happens.
- 3 After broadcasting the event, the broadcaster performs early return logic based on the type of action statement that caused the event.

For an understanding of early return logic, see “Early Return Logic for Event Broadcasts” on page 3-26.

Note Output edge trigger event execution in Simulink is equivalent to toggling the value of an output data value between 1 and 0. It is not treated as a Stateflow event. See “Defining Edge-Triggered Output Events” on page 9-20.

Executing a Chart

A Stateflow chart executes when it is triggered by an event from Simulink. Like all events, this event is processed top down in the updated chart. See “Executing an Event” on page 3-3.

A chart is inactive when it is first triggered by an event from the Simulink model and has no active states within it. After the chart executes and completely processes its initial trigger event from the Simulink model, it exits to the model and goes to sleep, but still remains active. A sleeping chart has active states within it, but no events to process. When Simulink triggers the chart the next time, it is an active but sleeping chart.

Executing an Inactive Chart

When a chart is inactive and first triggered by an event from Simulink, it first executes its set of default flow graphs (see “Executing a Set of Flow Graphs” on page 3-8). If this does not cause an entry into a state and the chart has parallel decomposition, then each parallel state is entered (see “Entering a State” on page 3-20).

If executing the default flow paths does not cause state entry, a state inconsistency error occurs.

Executing an Active Chart

After a chart has been triggered the first time by the Simulink model, it is an active chart. When it receives another event from Simulink, it executes again as an active chart. If the chart has no states, each execution is equivalent to initializing a chart. Otherwise, the active children are executed. Parallel states are executed in the same order that they are entered.

Executing a Transition

Transitions play a large role in defining the animation or execution of a system. If your chart has exclusive (OR) states, its execution begins with the default transitions that point to the first active states in your chart.

Transitions have sources and destinations; thus, any actions associated with the sources or destinations are related to the transition that joins them. The type of the source and destination is equally important to define the semantics.

See the following topics:

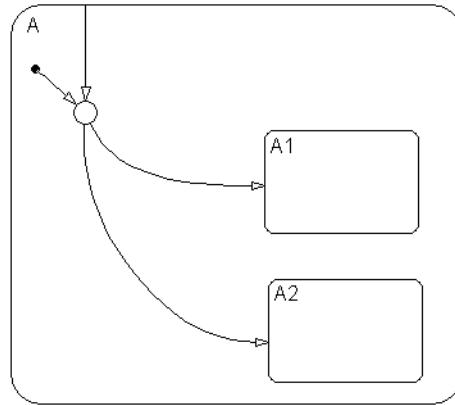
- “Transition Flow Graph Types” on page 3-7 — Defines the groups used for ordering transitions in order to select a valid transition for execution.
Outgoing transitions from an active state are first grouped according to their flow graph type.
- “Executing a Set of Flow Graphs” on page 3-8 — Describes the process of selecting a valid transition from a group.
The transitions in each flow graph group are checked for a valid, executable transition, starting with the highest priority group, and ending with the lowest priority group, until a valid transition is found.
- “Transition Testing Order” on page 3-10 — Describes the process through which transitions with the same source are ordered before selecting a valid transition for execution.

Transition Flow Graph Types

Before transitions are executed for an active state or for a chart, they are grouped by the following types:

- Default flow graphs are all default transition segments that start with the same parent.
- Inner flow graphs are all transition segments that originate on a state and reside entirely within that state.
- Outer flow graphs are all transition segments that originate on the respective state but reside at least partially outside that state.

Each set of flow graphs includes other transition segments connected to a qualifying transition segment through junctions and transitions. Consider the following example:



In this example, state A has both an inner and a default transition that connect to a junction with outgoing transitions to states A.A1 and A.A2. If state A is active, its set of inner flow graphs includes the inner transition and the outgoing transitions from the junction to state A.A1 and A.A2. In addition, state A's set of default flow graphs includes the default transition to the junction along with the two outgoing transitions from the junction to state A.A1 and A.A2. In this case, the two outgoing transition segments from the junction become members of more than one flow graph type.

Executing a Set of Flow Graphs

Each flow graph group is executed in the order of group priority until a valid transition is found. The default transitions group is executed first, followed by the outer transitions group and then the inner transitions group. Each flow graph group is executed with the following procedure.

- 1 Order the group's transition segments for the active state.

An active state can have several possible outgoing transitions. These are ordered before checking them for a valid transition. See "Transition Testing Order" on page 3-10.

- 2 Select the next transition segment in the set of ordered transitions.
- 3 Test the transition segment for validity.

- 4** If the segment is invalid, go to step 2.
- 5** If the destination of the transition segment is a state, do the following:
 - a** No more transition segments are tested and a transition path is formed by including the transition segment from each preceding junction back to the starting transition.
 - b** The states that are the immediate children of the parent of the transition path are exited (see “Exiting an Active State” on page 3-23).
 - c** The transition action for the final transition segment of the full transition path is executed.
 - d** The destination state is entered (see “Entering a State” on page 3-20).
- 6** If the destination is a junction with no outgoing transition segments, do the following:
 - a** Testing stops without any states being exited or entered.
- 7** If the destination is a junction with outgoing transition segments, repeat step 1 for the set of outgoing segments from the junction.
- 8** After testing all outgoing transition segments at a junction, back up the incoming transition segment that brought you to the junction and continue at step 2, starting with the next transition segment after the backup segment. The set of flow graphs is done executing when all starting transitions have been tested.

Transition Testing Order

If there is more than one transition from a source (state or junction) in a Stateflow diagram, then the transitions are numbered 1, 2, 3, ..., according to their testing order.

By default, Stateflow determines the order of testing transitions from a single source based on a set of internal rules. This is called the implicit order. The rules for determining the implicit order of transitions are discussed in “Implicit Order Mode” on page 3-10.

You have an option to switch from implicit ordering of transitions to a user-defined, or explicit, ordering. In this mode, you control the order in which transitions are tested for execution. Explicitly changing the transition testing order is discussed in “Explicit Order Mode” on page 3-14.

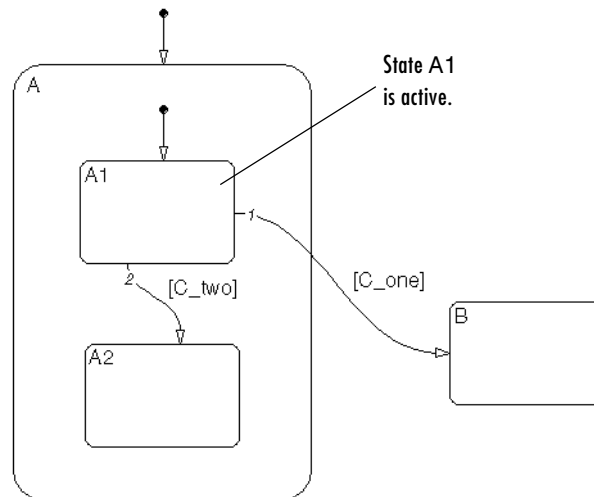
Implicit Order Mode

Implicit order mode is the default mode both for old models and for newly created models. Transitions from a single source are ordered for testing according to the following three sorting guidelines, which appear in order of their precedence (first step is highest priority):

- 1 Endpoint Hierarchy** — Transitions whose end points are attached to higher hierarchical levels are placed first in testing order. See the topic “Ordering by Hierarchy” on page 3-10.
- 2 Label** — Transitions are ordered for testing according to the types of action language present in their labels. See “Ordering by Label” on page 3-11.
- 3 Angular Surface Position of Transition Source** — Transitions are ordered for testing based on the angular position of the transition source on the surface of the originating object. See “Ordering by Geometric Position of Source” on page 3-12.

Ordering by Hierarchy

Transitions are evaluated in a top-down manner based on hierarchy. In the following example, an event occurs while state A1 is active.



Because state B is a sibling of state A and at a higher hierarchical level than state A2, a sibling of A1, the transition from state A1 to state B takes precedence over the transition from state A1 to state A2.

Ordering by Label

Transitions of equal endpoint hierarchical level are evaluated based on their labels, in the following order of precedence:

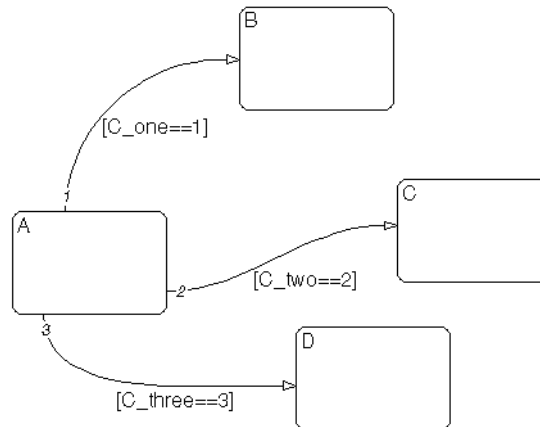
- 1 Labels with events and conditions
- 2 Labels with events
- 3 Labels with conditions
- 4 No label

The following example demonstrates ordering of single source transitions by the angular surface position of the source.

Ordering by Geometric Position of Source

Equivalent transitions (based on their labels and the hierarchy of their source and endpoints) are ordered based on the angular position on the surface of the source object for the outgoing transitions. The smallest clock position has the highest priority. For example, a transition with a 2 o'clock source position has a higher priority than a transition with a 4 o'clock source position. A transition with a 12 o'clock source position has the lowest possible priority.

Multiple outgoing transitions from states that are of equivalent label and source and end point hierarchy priority are evaluated in a clockwise progression starting at the upper left corner of the source state.



In this example, the transitions are of equivalent label priority. The conditions `[C_one == 1]` and `[C_two == 2]` are both false and the condition `[C_three == 3]` is true. Also, the hierarchical level of the endpoint of each transition is the same because all the states in the example are siblings.

The outgoing transitions from state A in the preceding diagram are evaluated in the following order:

- 1 Starting at the upper left corner of the source state A in a clockwise progression, the first transition is the transition from state A to state B.

Since the condition `[C_one == 1]` is false, this transition is not valid.

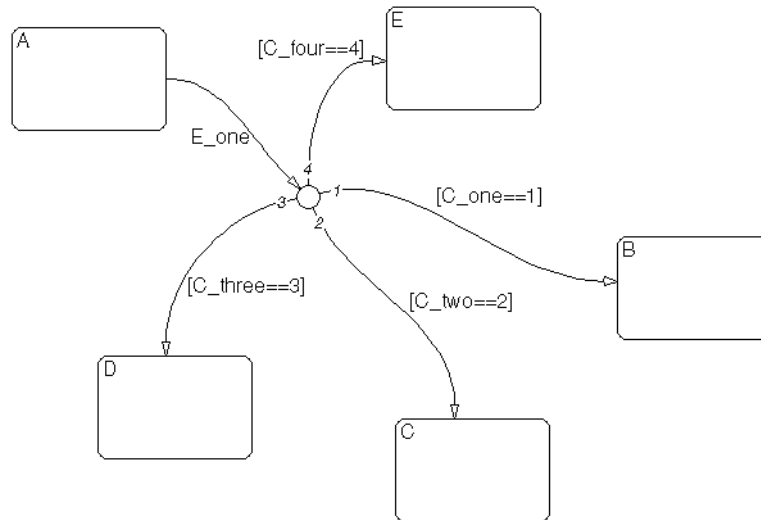
- 2** The next transition in a clockwise progression is the transition from state A to state C.

Since the condition $[C_two == 2]$ is false, this transition is not valid.

- 3** The next transition is the transition from state A to state D.

Since the condition $[C_three == 3]$ is true, this transition is valid and is taken.

Multiple outgoing transitions from junctions that are of equivalent label priority are evaluated according to the same angular position prioritization.



In this example,

- All outgoing transitions from the junction have conditions, which makes them equal in label priority.
- The conditions $[C_three == 3]$ and $[C_four == 4]$ are true.
- The junction source point for the transition to state E is exactly 12 o'clock.

The outgoing transitions from the junction are evaluated in the following order:

- 1 The transition to state B is evaluated. Since the condition `[C_one == 1]` is false, this transition is not valid.
- 2 The transition to state C is evaluated. Since the condition `[C_two == 2]` is false, this transition is not valid.
- 3 The transition to state D is evaluated. Since the condition `[C_three == 3]` is true, this transition is valid and taken.

Since the transition to D is taken, the transition to state E is not evaluated.

Explicit Order Mode

Implicit ordering rules are complex and hard to master. Additionally, when you modify a diagram for cosmetic purposes, rearranging transitions may inadvertently change execution. This is why you have an option to switch from implicit ordering of transitions to a user-defined, or explicit, ordering. In this mode, you control the order in which transitions are tested for execution.

Note You can reorder transitions only within their type (inner, outer, or default). For more information, see “Transition Flow Graph Types” on page 3-7.

To control the order of transitions, you have to perform the following steps:

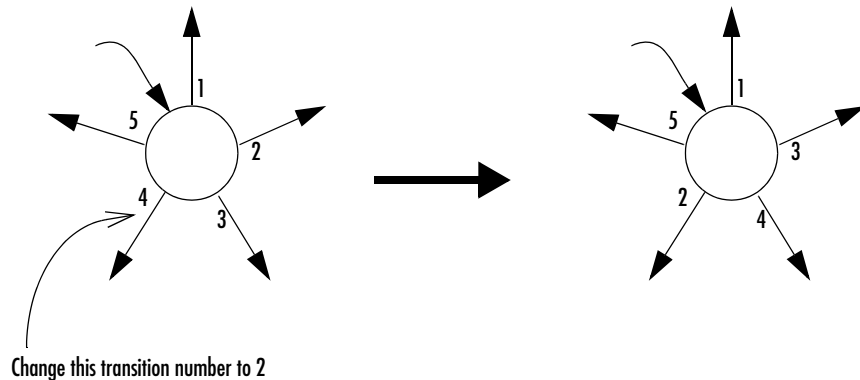
- 1 “Switching to Explicit Order Mode” on page 3-15
- 2 “Changing the Transition Order” on page 3-17

When you switch to explicit transition ordering, Stateflow stops evaluating the implicit rules. All the existing transitions in a diagram retain their current order numbers until you explicitly change them. All the newly created transitions for a source are automatically numbered in the order you create them, starting with the next available number for the source.

You can change the order of outgoing transitions for a source by explicitly renumbering them. When you change a transition number, Stateflow automatically renumbers the other outgoing transitions for the source by preserving their relative order. This is similar to taking a book out of a stack

and then putting it back in a different place: the order of the other books is unchanged, and the books are renumbered accordingly. This behavior is consistent with the automatic renumbering rules for the Simulink ports.

For example, if you have a source with five outgoing transitions, as shown below, changing transition 4 to 2 results in the following automatic renumbering.



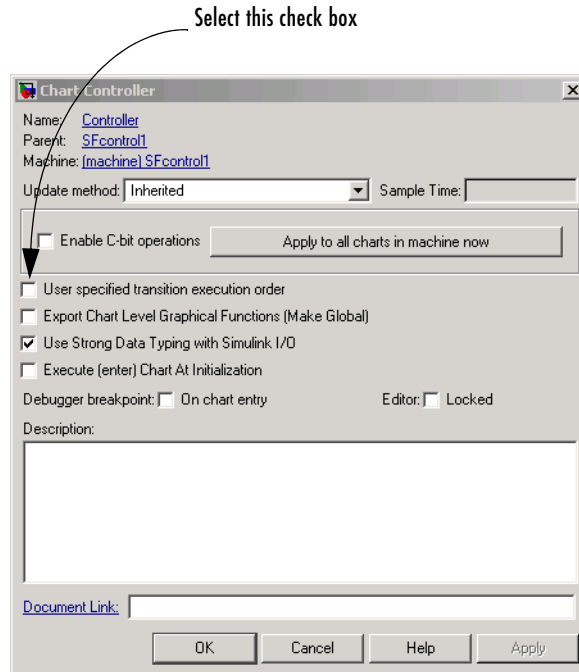
Automatic Renumbering of Transitions During Explicit Reordering

Switching to Explicit Order Mode

You switch to explicit ordering of transitions by setting your chart property preferences.

- 1 In the Stateflow diagram editor, from the **File** menu, select **Chart Properties**.

The properties dialog for the chart appears, as shown:



2 Select the **User specified transition execution order** check box.

3 Click **OK** to apply the change and close the dialog.

Now you can change the transition execution order for any source in the chart.

Switching between modes. If you switch back to implicit order mode after having explicitly reordered the transitions, the transition order is reset to follow the implicit rules.

Similarly, if you eventually change back to explicit order mode, without making any changes to the diagram, Stateflow restores the previous explicit transition order. Whenever you switch from one transition ordering mode to another, Stateflow displays the warnings about the changes in transition numbers in the diagnostic viewer.

Note If you change back to explicit order mode after having made changes to the diagram, Stateflow may not be able to restore the previous explicit transition order. Review the warnings in the diagnostic viewer and change the transition order, as necessary.

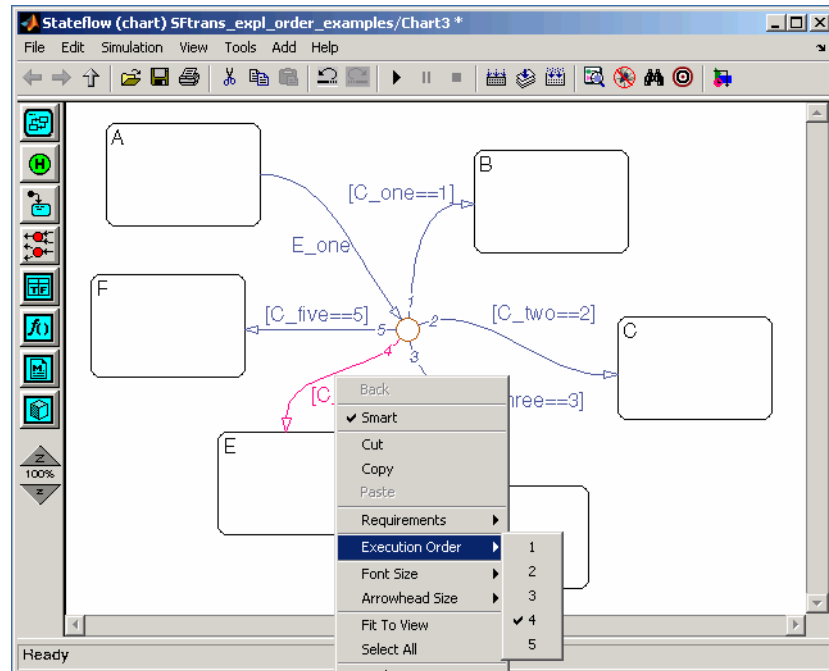
Changing the Transition Order

You change the order of transitions originating from a source by explicitly renumbering them.

- 1 Right-click a transition and select **Execution Order**.

Note If you select **Execution Order** while in implicit order mode, the only option available is **Enable ‘User specified execution order’ for this chart**. This opens the properties dialog for the chart, to let you switch to explicit order mode, as described in “Switching to Explicit Order Mode” on page 3-15.

A context menu of available transition numbers appears, with a checkmark next to the current number for this transition. For example, if there are five transitions originating from a source, and you are modifying the execution order for transition 4, the context menu will contain numbers from 1 to 5, with a checkmark next to number 4, as shown in the following illustration.

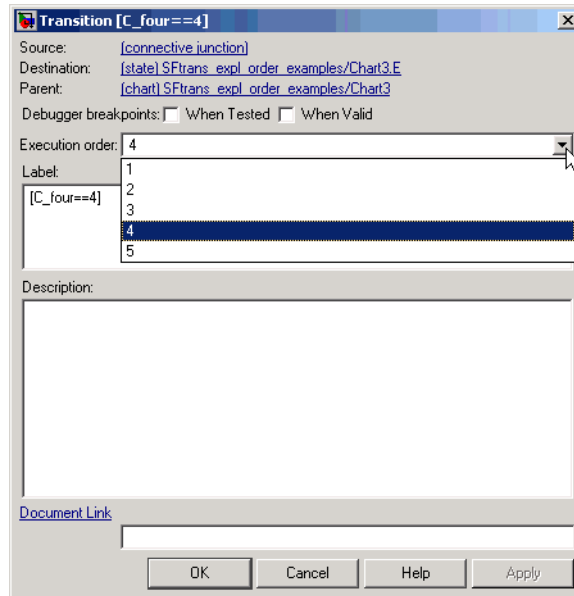


- 2 Select the new transition number. Stateflow automatically renumbers the other transitions for the source by preserving the relative transition order, as shown in the illustration “Automatic Renumbering of Transitions During Explicit Reordering” on page 3-15.
- 3 Repeat this procedure to renumber the other transitions as necessary.

Another way to access the transition order number is through its properties:

- 1 Right-click a transition and select **Properties**. The properties dialog for the transition appears.

- 2 Click in the **Execution order** box. A drop-down list of valid transition numbers appears, as shown in the following illustration.



- 3 Select the new transition number and click **Apply**. If the explicit order mode is enabled, Stateflow assigns the new number to the current transition and automatically renumbers the other transitions. If Stateflow is in the implicit order mode, an error dialog is displayed and the old number is retained.

Entering, Executing, and Exiting a State

States are either active or inactive. The following sections describe the stages of state execution that take place between becoming active and becoming inactive:

- “Entering a State” on page 3-20 — Describes the execution involved when a transition causes a state to be entered.
- “Executing an Active State” on page 3-22 — Describes the execution of a state when it receives an event that does not result in an outgoing transition from that state to another.
- “Exiting an Active State” on page 3-23 — Describes the execution involved when a transition causes a state to be exited.
- “State Execution Example” on page 3-23 — Gives a detailed description of an example of state execution activities during the execution of a Stateflow diagram.

Entering a State

A state is entered (becomes active) in one of the following ways:

- Its boundaries are crossed by an incoming executed transition.
- Its boundary terminates the arrow end of an incoming transition.
- It is the parallel state child of an activated state.

A state performs its entry action (if specified) when it becomes active. The state is marked active before its entry action is executed and completed.

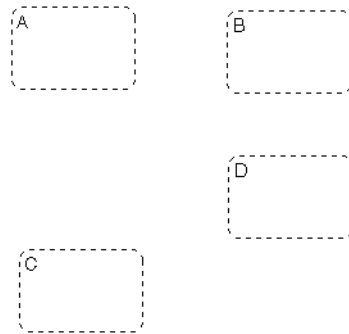
The execution steps for entering a state are as follows:

- 1** If the parent of the state is not active, perform steps 1 through 4 for the parent first.

- 2** If this is a parallel state, check if a sibling parallel state previous in entry order is active. If so, start at step 1 for this parallel state.

Parallel (AND) states are ordered for entry based on their vertical top-to-bottom position in the diagram editor. Parallel states that occupy the same vertical level are ordered for entry from left to right.

In the following example, parallel states A and B are aligned at the same vertical level while states A and C and states B and D are aligned at the same horizontal position.



Based on their top-down positions in the diagram editor, the order of entry for these states is A or B, then C, then D. Because A is left of B, A is evaluated first and the order of entry is A, B, D, C. Stateflow marks this order with an order number in the upper right-hand corner of the state (1, 2, 3, 4, respectively)

Step 2 says that if you are entering state D in step 1, check if state B is active. If it is not, start at step 1 for state B. Step 2 repeats for state B and, if A is not active, start at step 1 for A. Since there are no parallel states of lesser entry order, continue with step 3 for state A.

- 3** Mark the state active.
- 4** Perform any entry actions.
- 5** Enter children, if needed:

- a Execute the default flow paths for the state unless it contains a history junction.
 - b If the state contains a history junction and there is an active child of this state at some point after the most recent chart initialization, perform the entry actions for that child.
 - c If this state has children that are parallel states (parallel decomposition), perform entry steps 1 to 5 for each state according to its entry order.
- 6 If this is a parallel state, perform all entry steps for the sibling state next in entry order if one exists.
 - 7 If the transition path parent is not the same as the parent of the current state, perform entry steps 6 and 7 for the immediate parent of this state.
 - 8 The chart goes to sleep.

Executing an Active State

Active states that receive an event that does not result in an exit from that state execute a during action to completion if a during action is specified for that state. An *on event_name* action executes to completion when the event specified, *event_name*, occurs and that state is active. An active state executes its during and *on event_name* actions before processing any of its children's valid transitions. During and *on event_name* actions are processed based on their order of appearance in the state label.

The execution steps for executing a state that receives an event while it is active are as follows:

- 1 The set of outer flow graphs is executed (see “Executing a Set of Flow Graphs” on page 3-8).

If this causes a state transition, execution of the state stops.

Note This step is never required for parallel states.

- 2 During actions and valid *on event name* actions are preformed.

- 3 The set of inner flow graphs is executed. If this does not cause a state transition, the active children are executed, starting at step 1. Parallel states are executed in the same order that they are entered.

Exiting an Active State

A state is exited (becomes inactive) in one of the following ways:

- Its boundary is the origin of an outgoing executed transition.
- Its boundary is crossed by an outgoing executed transition.
- It is a parallel state child of an activated state.

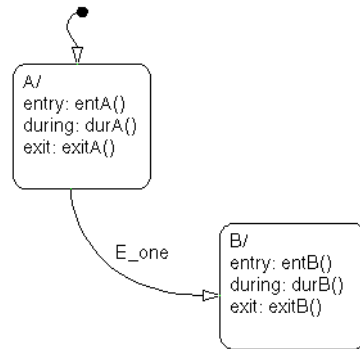
A state performs its `exit` action (if specified) before it becomes inactive. The state is marked inactive after the `exit` action has executed and completed.

The execution steps for exiting a state are as follows:

- 1 If this is a parallel state, and one of its sibling states was entered before this state, exit the siblings starting with the last-entered and progressing in reverse order to the first-entered. See step 2 of “Entering a State” on page 3-20.
- 2 If there are any active children, perform the exit steps on these states in the reverse order they were entered.
- 3 Perform any exit actions.
- 4 Mark the state as inactive.

State Execution Example

The following example demonstrates the execution semantics (behavior) of event reactive behavior by active and inactive states.



Inactive Diagram Event Reaction

Initially the Stateflow diagram and its states are inactive. This is the semantic sequence that follows an event:

- 1 An event occurs and the Stateflow diagram is awakened.
- 2 The Stateflow diagram checks to see if there is a valid transition as a result of the event.

A valid default transition to state A is detected.
- 3 State A is marked active.
- 4 State A entry actions (entA()) execute and complete.
- 5 The Stateflow diagram goes back to sleep.

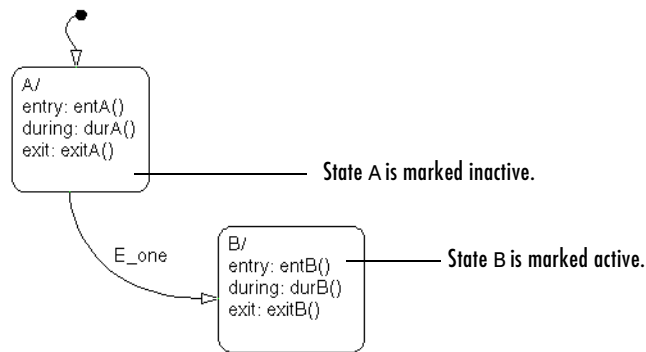
Sleeping Diagram Event Reaction

The Stateflow diagram is now asleep and waiting to be awakened by another event.

- 1 Event E_one occurs and the Stateflow diagram is awakened.

State A is active from the preceding steps 1 to 5.
- 2 The Stateflow diagram root checks to see if there is a valid transition as a result of E_one. A valid transition is detected from state A to state B.

- 3 State A exit actions (`exitA()`) execute and complete.
- 4 State A is marked inactive.
- 5 State B is marked active.
- 6 State B entry actions (`entB()`) execute and complete.
- 7 The Stateflow diagram goes back to sleep, to be awakened by the next event.



Early Return Logic for Event Broadcasts

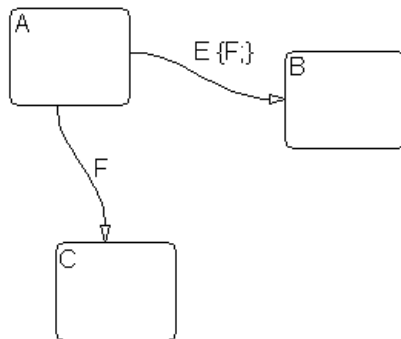
Stateflow employs early return logic in order to satisfy conflicts with proper diagram behavior that result from event broadcasts in state or transition actions.

The following statements are primary axioms of proper Stateflow behavior:

- 1 Whenever a state is active, its parent should also be active.
- 2 A state (or chart) with exclusive (OR) decomposition must never have more than one active child.
- 3 If a parallel state is active, siblings with higher priority (higher graphical position in the Stateflow diagram) must also be active.

Because Stateflow runs on a single thread, when it receives an event it must interrupt its current activity to process all activity resulting from the broadcast event before returning to its original activity. However, activity resulting from an event broadcast can conflict with the current activity, giving rise to the event broadcast. This conflict is resolved through early return logic.

The need for early return logic is best illustrated with an example like the following:



In this example, assume that state A is initially active. An event, E, occurs and the following behavior is expected:

- 1** The Stateflow diagram root checks to see if there is a valid transition out of the active state A as a result of event E.
- 2** A valid transition terminating in state B is found.
- 3** The condition action of the valid transition executes and broadcasts event F.

Stateflow must now interrupt the anticipated transition from A to B and take care of any behavior resulting from the broadcast of the event F before continuing with the transition from A to B.

- 4** The Stateflow diagram root checks to see if there is a valid transition out of the active state A as a result of event F.
- 5** A valid transition terminating in state C is found.
- 6** State A executes its `exit` action.
- 7** State A is marked inactive.
- 8** State C is marked active.
- 9** State C executes and completes its entry action.

State C is now the only active child of its chart. Stateflow cannot return to the transition from state A to state B and continue after the condition action that broadcast event F (step 3). First, its source, state A, is no longer active. Second, if Stateflow were to allow the transition, state B would become the second active child of the chart. This violates the second Stateflow axiom that a state (or chart) with exclusive (OR) decomposition can never have more than one active child. Consequently, early return logic is employed, and the transition from state A to state B is halted.

In order to maintain primary axiomatic behavior in Stateflow diagrams, Stateflow employs early return logic for event broadcasts in each of its action types as follows:

Action Type	Early Return Logic
Entry	If the state is no longer active at the end of the event broadcast, any remaining steps for entering a state are not performed.
Exit	If the state is no longer active at the end of the event broadcast, any remaining exit actions or steps in transitioning from state to state are not performed.
During	If the state is no longer active at the end of the event broadcast, any remaining steps in the execution of active state are not performed.
Condition	If the origin state of the inner or outer flow graph or parent state of the default flow graph is no longer active at the end of the event broadcast, the remaining steps in the execution of the set of flow graphs are not performed.
Transition	If the parent of the transition path is not active or if that parent has an active child, the remaining transition actions and state entry are not performed.

Semantic Examples

The following is a list of the examples provided to demonstrate the semantics (behavior) of Stateflow charts.

“Transitions to and from Exclusive (OR) States Examples” on page 3-31

- “Transitioning from State to State with Events Example” on page 3-32
- “Transitioning from a Substate to a Substate with Events Example” on page 3-35

“Condition Action Examples” on page 3-37

- “Condition Action Example” on page 3-37
- “Condition and Transition Actions Example” on page 3-39
- “Condition Actions in For Loop Construct Example” on page 3-40
- “Condition Actions to Broadcast Events to Parallel (AND) States Example” on page 3-40
- “Cyclic Behavior to Avoid with Condition Actions Example” on page 3-41

“Default Transition Examples” on page 3-43

- “Default Transition in Exclusive (OR) Decomposition Example” on page 3-43
- “Default Transition to a Junction Example” on page 3-44
- “Default Transition and a History Junction Example” on page 3-45
- “Labeled Default Transitions Example” on page 3-47

“Inner Transition Examples” on page 3-49

- “Processing One Event in an Exclusive (OR) State” on page 3-49
- “Processing a Second Event in an Exclusive (OR) State” on page 3-50
- “Processing a Third Event in an Exclusive (OR) State” on page 3-51
- “Processing the First Event with an Inner Transition to a Connective Junction” on page 3-53
- “Processing a Second Event with an Inner Transition to a Connective Junction” on page 3-54
- “Inner Transition to a History Junction Example” on page 3-56

“Connective Junction Examples” on page 3-58

- “If-Then-Else Decision Construct Example” on page 3-59
- “Self-Loop Transition Example” on page 3-61
- “For Loop Construct Example” on page 3-62
- “Flow Diagram Notation Example” on page 3-63
- “Transitions from a Common Source to Multiple Destinations Example” on page 3-65
- “Transitions from Multiple Sources to a Common Destination Example” on page 3-67
- “Transitions from a Source to a Destination Based on a Common Event Example” on page 3-68

“Event Actions in a Superstate Example” on page 3-71

- “Event Actions in a Superstate Example” on page 3-71

“Parallel (AND) State Examples” on page 3-73

- “Event Broadcast State Action Example” on page 3-73
- “Event Broadcast Transition Action with a Nested Event Broadcast Example” on page 3-77
- “Event Broadcast Condition Action Example” on page 3-81

Directed Event Broadcasting

- “Directed Event Broadcast Using send Example” on page 3-85
- “Directed Event Broadcasting Using Qualified Event Names Example” on page 3-87

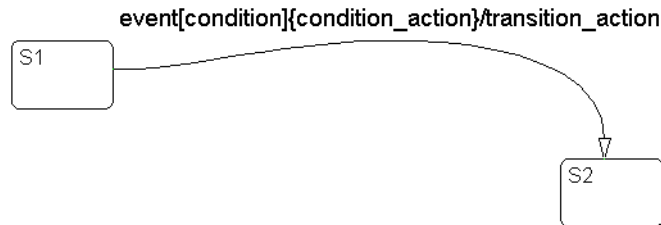
Transitions to and from Exclusive (OR) States Examples

The following examples demonstrate the use of state-to-state transitions to and from exclusive (OR) states in Stateflow:

- “Label Format for a State-to-State Transition Example” on page 3-31 — Shows the semantics for a transition with a general label format.
- “Transitioning from State to State with Events Example” on page 3-32 — Shows the behavior of a simple transition focusing on the implications of whether states are active or inactive for handling events.
- “Transitioning from a Substate to a Substate with Events Example” on page 3-35 — Shows the behavior of a transition from an OR substate to an OR substate.

Label Format for a State-to-State Transition Example

The following example shows the general label format for a transition entering a state:



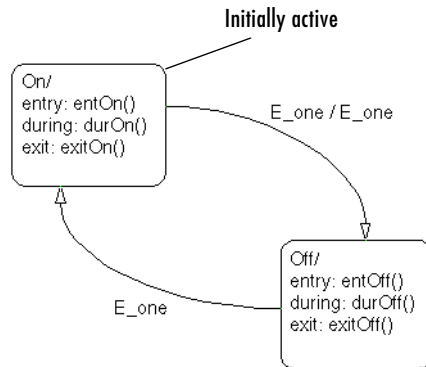
Execution of the above transition occurs as follows:

- 1 When an event occurs, state S1 checks for an outgoing transition with a matching event specified.
- 2 If a transition with a matching event is found, the condition for that transition (`[condition]`) is evaluated.

- 3 If the condition `condition` evaluates to true, the condition action `condition_action ({condition_action})` is executed.
- 4 If the destination state is determined to be a valid destination, the transition is taken.
- 5 State `S1` is exited.
- 6 The transition action `transition_action` is executed when the transition is taken.
- 7 State `S2` is entered.

Transitioning from State to State with Events Example

The following example shows the behavior of a simple transition focusing on the implications of whether states are active or inactive.



Processing of a First Event

Initially the Stateflow diagram is asleep. State `On` and state `Off` are OR states. State `On` is active. Event `E_one` occurs and awakens the Stateflow diagram. Event `E_one` is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram:

- 1 The Stateflow diagram root checks to see if there is a valid transition as a result of E_one. A valid transition from state On to state Off is detected.
- 2 State On exit actions (ExitOn()) execute and complete.
- 3 State On is marked inactive.
- 4 The event E_one is broadcast as the transition action.

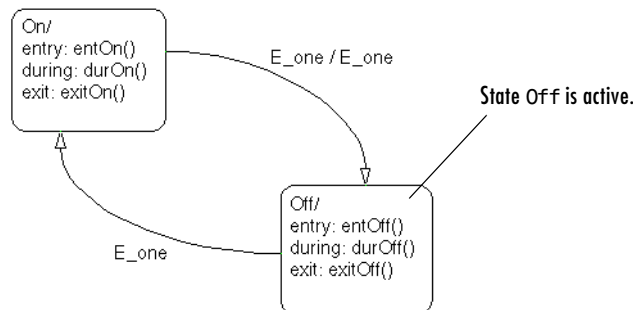
This second event E_one is processed, but because neither state is active, it has no effect. Had a valid transition been possible as a result of the broadcast of E_one, the processing of the first broadcast of E_one would be preempted by the second broadcast of E_one. See “Early Return Logic for Event Broadcasts” on page 3-26.

- 5 State Off is marked active.
- 6 State Off entry actions (entOff()) execute and complete.
- 7 The Stateflow diagram goes back to sleep, waiting to be awakened by another event.

This sequence completes the execution of the Stateflow diagram associated with event E_one when state On is initially active.

Processing of a Second Event

Using the same example, what happens when the next event, E_one, occurs while state Off is active?



Again, initially the Stateflow diagram is asleep. State `Off` is active. Event `E_one` occurs and awakens the Stateflow diagram. Event `E_one` is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram with the following steps:

- 1 The Stateflow diagram root checks to see if there is a valid transition as a result of `E_one`.

A valid transition from state `Off` to state `On` is detected.

- 2 State `Off` exit actions (`exitOff()`) execute and complete.

- 3 State `Off` is marked inactive.

- 4 State `On` is marked active.

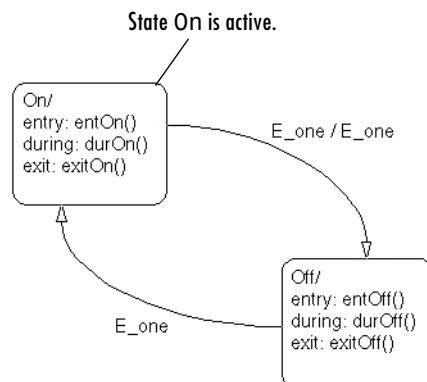
- 5 State `On` entry actions (`entOn()`) execute and complete.

- 6 The Stateflow diagram goes back to sleep, waiting to be awakened by another event.

This sequence completes the execution of the Stateflow diagram associated with the second event `E_one` when state `Off` is initially active.

Processing of a Third Event

Using the same example, what happens when a third event, `E_two`, occurs?



Notice that the event `E_two` is not used explicitly in this example. However, its occurrence (or the occurrence of any event) does result in behavior. Initially, the Stateflow diagram is asleep and state `On` is active.

- 1 Event `E_two` occurs and awakens the Stateflow diagram.

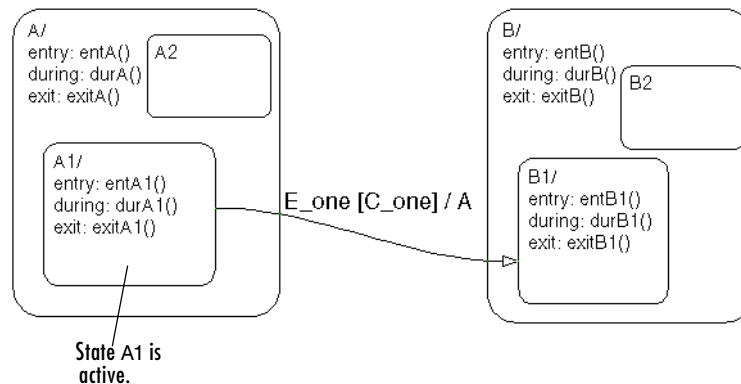
Event `E_two` is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram:

- 2 The Stateflow diagram root checks to see if there is a valid transition as a result of `E_two`. There is none.
- 3 State `On` during actions (`durOn()`) execute and complete.
- 4 The Stateflow diagram goes back to sleep, waiting to be awakened by another event.

This sequence completes the execution of the Stateflow diagram associated with event `E_two` when state `On` is initially active.

Transitioning from a Substate to a Substate with Events Example

This example shows the behavior of a transition from an OR substate to an OR substate.



Initially the Stateflow diagram is asleep. State A.A1 is active. Event E_one occurs and awakens the Stateflow diagram. Condition C_one is true. Event E_one is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram:

- 1** The Stateflow diagram root checks to see if there is a valid transition as a result of E_one. There is a valid transition from state A.A1 to state B.B1. (Condition C_one is true.)
- 2** State A during actions (durA()) execute and complete.
- 3** State A.A1 exit actions (exitA1()) execute and complete.
- 4** State A.A1 is marked inactive.
- 5** State A exit actions (exitA()) execute and complete.
- 6** State A is marked inactive.
- 7** The transition action, A, is executed and completed.
- 8** State B is marked active.
- 9** State B entry actions (entB()) execute and complete.
- 10** State B.B1 is marked active.
- 11** State B.B1 entry actions (entB1()) execute and complete.
- 12** The Stateflow diagram goes back to sleep, waiting to be awakened by another event.

This sequence completes the execution of this Stateflow diagram associated with event E_one.

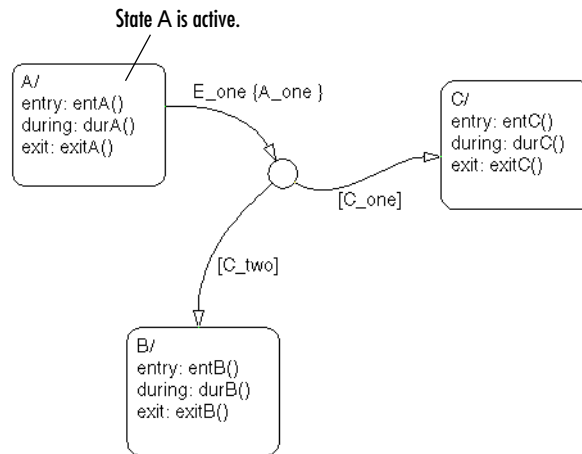
Condition Action Examples

The following examples demonstrate the use of condition actions in Stateflow:

- “Condition Action Example” on page 3-37 — Shows the behavior of a simple condition action in a multiple segment transition.
- “Condition and Transition Actions Example” on page 3-39 — Shows the behavior of a simple condition and transition action specified on a transition from one exclusive (OR) state to another.
- “Condition Actions in For Loop Construct Example” on page 3-40 — Shows the use of a condition action and connective junction to create a for loop construct.
- “Condition Actions to Broadcast Events to Parallel (AND) States Example” on page 3-40 — Shows the use of condition actions used to broadcast events immediately to parallel (AND) states.
- “Cyclic Behavior to Avoid with Condition Actions Example” on page 3-41 — Shows a notation to avoid when using event broadcasts as condition actions because the semantics result in cyclic behavior.

Condition Action Example

This example shows the behavior of a simple condition action in a multiple segment transition.



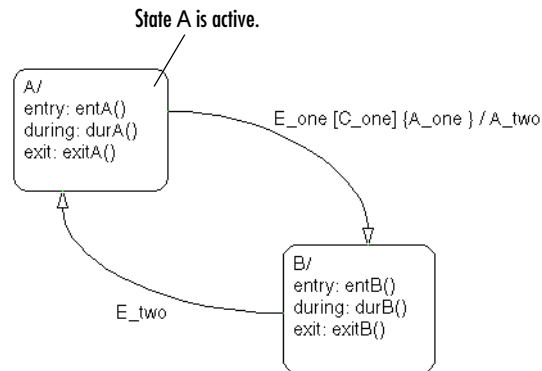
Initially the Stateflow diagram is asleep. State A is active. Event `E_one` occurs and awakens the Stateflow diagram. Conditions `C_one` and `C_two` are false. Event `E_one` is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram:

- 1 The Stateflow diagram root checks to see if there is a valid transition as a result of `E_one`. A valid transition segment from state A to a connective junction is detected. The condition action `A_one` is detected on the valid transition segment and is immediately executed and completed. State A is still active.
- 2 Because the conditions on the transition segments to possible destinations are false, none of the complete transitions is valid.
- 3 State A during actions (`durA()`) execute and complete.
State A remains active.
- 4 The Stateflow diagram goes back to sleep, waiting to be awakened by another event.

This sequence completes the execution of this Stateflow diagram associated with event `E_one` when state A is initially active.

Condition and Transition Actions Example

This example shows the behavior of a simple condition and transition action specified on a transition from one exclusive (OR) state to another.



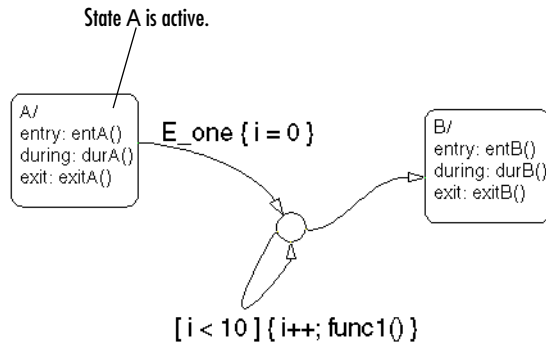
Initially the Stateflow diagram is asleep. State A is active. Event `E_one` occurs and awakens the Stateflow diagram. Condition `C_one` is true. Event `E_one` is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram:

- 1 The Stateflow diagram root checks to see if there is a valid transition as a result of `E_one`. A valid transition from state A to state B is detected. The condition `C_one` is true. The condition action `A_one` is detected on the valid transition and is immediately executed and completed. State A is still active.
- 2 State A exit actions (`ExitA()`) execute and complete.
- 3 State A is marked inactive.
- 4 The transition action `A_two` is executed and completed.
- 5 State B is marked active.
- 6 State B entry actions (`entB()`) execute and complete.
- 7 The Stateflow diagram goes back to sleep, waiting to be awakened by another event.

This sequence completes the execution of this Stateflow diagram associated with event E_one when state A is initially active.

Condition Actions in For Loop Construct Example

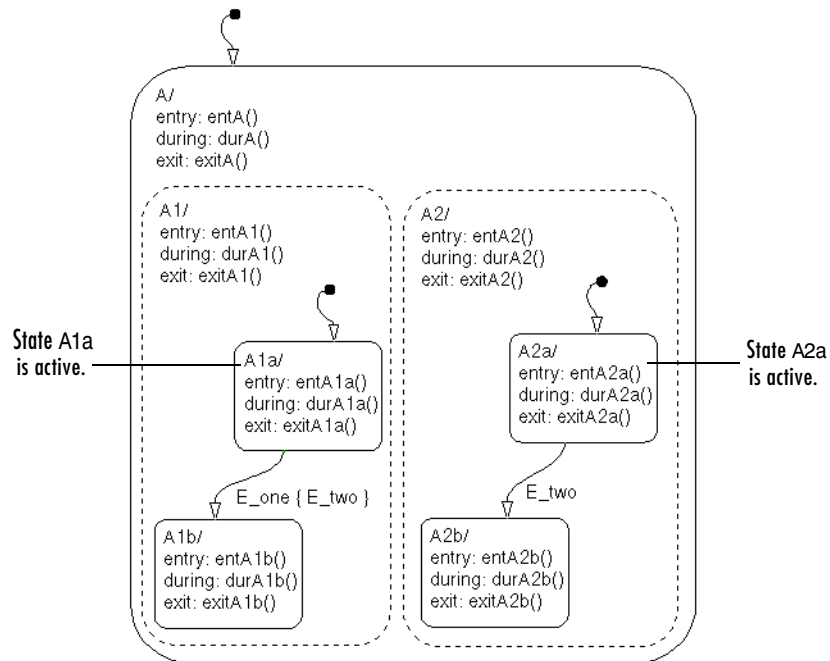
Condition actions and connective junctions are used to design a for loop construct. This example shows the use of a condition action and connective junction to create a for loop construct.



See “For Loop Construct Example” on page 3-62 to see the behavior of this example.

Condition Actions to Broadcast Events to Parallel (AND) States Example

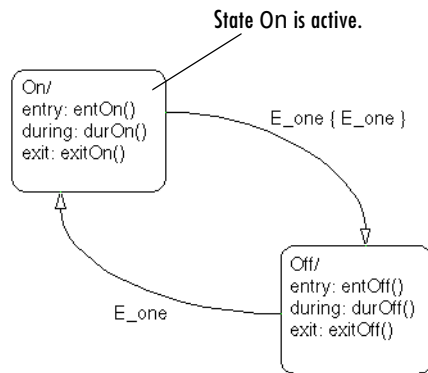
This example shows the use of condition actions used to broadcast events immediately to parallel (AND) states.



See “Event Broadcast Condition Action Example” on page 3-81 to see the behavior of this example.

Cyclic Behavior to Avoid with Condition Actions Example

This example shows a notation to avoid when using event broadcasts as condition actions because the semantics result in cyclic behavior.



Initially the Stateflow diagram is asleep. State On is active. Event **E_one** occurs and awakens the Stateflow diagram. Event **E_one** is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram:

- 1** The Stateflow diagram root checks to see if there is a valid transition as a result of **E_one**.

A valid transition from state On to state Off is detected.

- 2** The condition action on the transition broadcasts event **E_one**.
- 3** Event **E_one** is detected on the valid transition, which is immediately executed. State On is still active.
- 4** The broadcast of event **E_one** awakens the Stateflow diagram a second time.
- 5** Go to step 1.

Step 1 to 5 continue to execute in a cyclical manner. The transition label indicating a trigger on the same event as the condition action broadcast event results in unrecoverable cyclic behavior. This sequence never completes when event **E_one** is broadcast and state On is active.

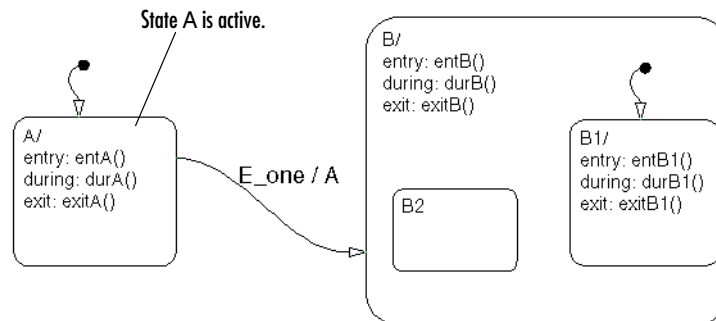
Default Transition Examples

The following examples demonstrate the use of default transitions in Stateflow:

- “Default Transition in Exclusive (OR) Decomposition Example” on page 3-43 — Shows the behavior of a transition from an OR state to a superstate with exclusive (OR) decomposition, where a default transition to a substate is defined.
- “Default Transition to a Junction Example” on page 3-44 — Shows the behavior of a default transition to a connective junction.
- “Default Transition and a History Junction Example” on page 3-45 — Shows the behavior of a superstate with a default transition and a history junction.
- “Labeled Default Transitions Example” on page 3-47 — Shows the use of a default transition with a label.

Default Transition in Exclusive (OR) Decomposition Example

This example shows a transition from an OR state to a superstate with exclusive (OR) decomposition, where a default transition to a substate is defined.



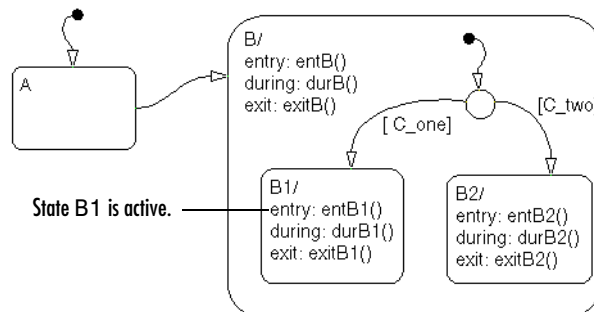
Initially the Stateflow diagram is asleep. State A is active. Event `E_one` occurs and awakens the Stateflow diagram. Event `E_one` is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram:

- 1 The Stateflow diagram root checks to see if there is a valid transition as a result of E_one. There is a valid transition from state A to superstate B.
- 2 State A exit actions (exitA()) execute and complete.
- 3 State A is marked inactive.
- 4 The transition action, A, is executed and completed.
- 5 State B is marked active.
- 6 State B entry actions (entB()) execute and complete.
- 7 State B detects a valid default transition to state B.B1.
- 8 State B.B1 is marked active.
- 9 State B.B1 entry actions (entB1()) execute and complete.
- 10 The Stateflow diagram goes back to sleep, waiting to be awakened by another event.

This sequence completes the execution of this Stateflow diagram associated with event E_one when state A is initially active.

Default Transition to a Junction Example

The following example shows the behavior of a default transition to a connective junction.



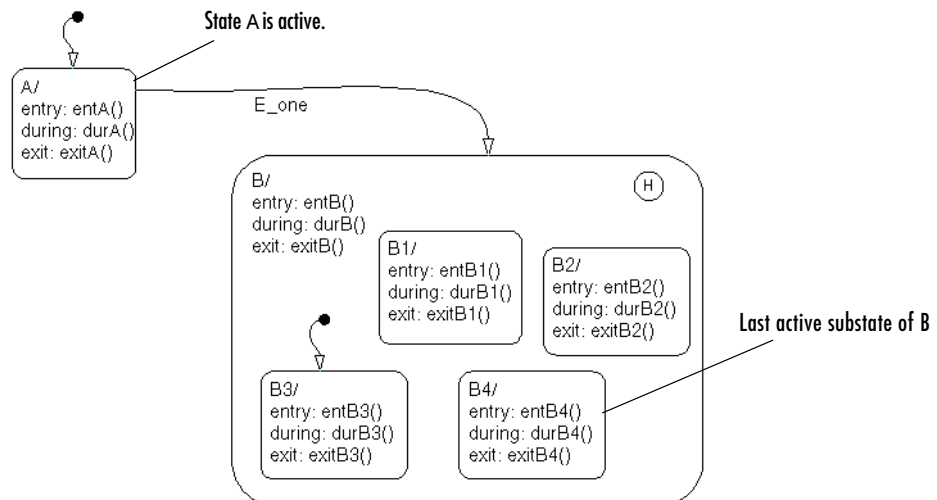
Initially the Stateflow diagram is asleep. State B.B1 is active. An event occurs and awakens the Stateflow diagram. Condition [C_two] is true. The event is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram:

- 1 State B checks to see if there is a valid transition as a result of any event. There is none.
- 2 State B1 during actions (durB1 ()) execute and complete.

This sequence completes the execution of this Stateflow diagram associated with the occurrence of any event.

Default Transition and a History Junction Example

This example shows the behavior of a superstate with a default transition and a history junction.



Initially the Stateflow diagram is asleep. State A is active. There is a history junction and state B4 was the last active substate of superstate B. Event E_one occurs and awakens the Stateflow diagram. Event E_one is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram:

- 1** The Stateflow diagram root checks to see if there is a valid transition as a result of E_one.

There is a valid transition from state A to superstate B.

- 2** State A exit actions (exitA()) execute and complete.
- 3** State A is marked inactive.
- 4** State B is marked active.
- 5** State B entry actions (entB()) execute and complete.
- 6** State B uses the history junction to determine the substate destination of the transition into the superstate.

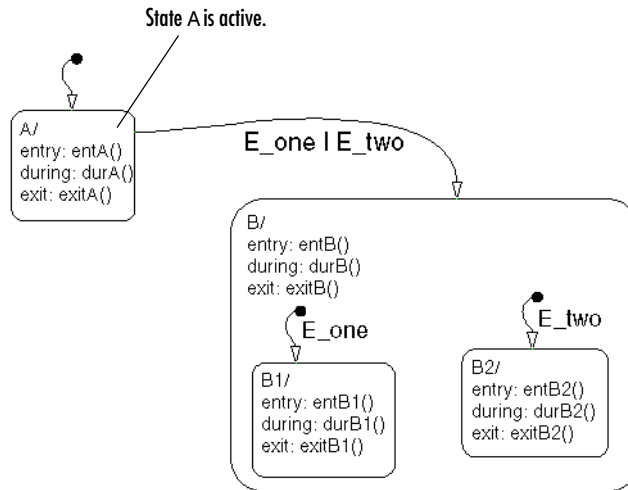
The history junction indicates that substate B.B4 was the last active substate, which becomes the destination of the transition.

- 7** State B.B4 is marked active.
- 8** State B.B4 entry actions (entB4()) execute and complete.
- 9** The Stateflow diagram goes back to sleep, waiting to be awakened by another event.

This sequence completes the execution of this Stateflow diagram associated with event E_one.

Labeled Default Transitions Example

This example shows the use of a default transition with a label.



Initially the Stateflow diagram is asleep. State A is active. Event `E_one` occurs, awakening the Stateflow diagram. Event `E_one` is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram with the following steps:

- 1 The Stateflow diagram root checks to see if there is a valid transition as a result of `E_one`.
There is a valid transition from state A to superstate B. The transition is valid if event `E_one` or `E_two` occurs.
- 2 State A exit actions execute and complete (`exitA()`).
- 3 State A is marked inactive.
- 4 State B is marked active.
- 5 State B entry actions execute and complete (`entB()`).
- 6 State B detects a valid default transition to state B.B1. The default transition is valid as a result of `E_one`.

- 7 State B.B1 is marked active.
- 8 State B.B1 entry actions execute and complete (entB1()).
- 9 The Stateflow diagram goes back to sleep, waiting to be awakened by another event.

This sequence completes the execution of this Stateflow diagram associated with event E_one when state A is initially active.

Inner Transition Examples

The following examples demonstrate the use of inner transitions in Stateflow:

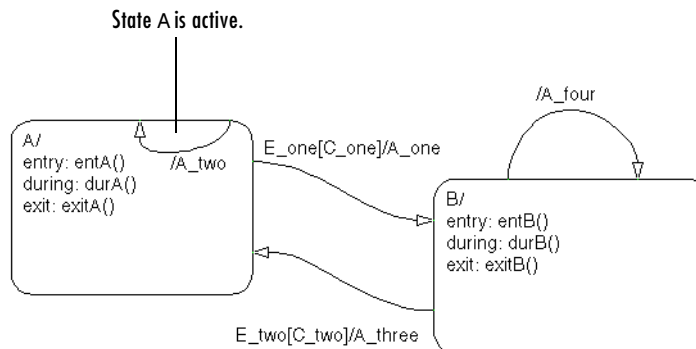
- “Processing Events with an Inner Transition in an Exclusive (OR) State Example” on page 3-49 — Shows what happens when processing repeated events using an inner transition in an exclusive (OR) state.
- “Processing Events with an Inner Transition to a Connective Junction Example” on page 3-53 — Shows the behavior of handling repeated events using an inner transition to a connective junction.
- “Inner Transition to a History Junction Example” on page 3-56 — Shows the behavior of an inner transition to a history junction.

Processing Events with an Inner Transition in an Exclusive (OR) State Example

This example shows what happens when processing three events using an inner transition in an exclusive (OR) state.

Processing One Event in an Exclusive (OR) State

This example shows the behavior of an inner transition.



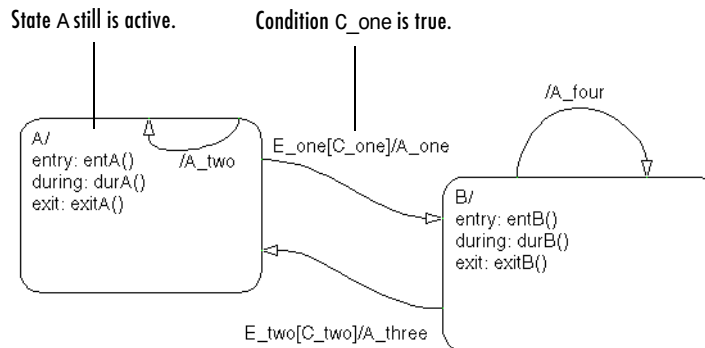
Initially the Stateflow diagram is asleep. State A is active. Event `E_one` occurs and awakens the Stateflow diagram. Condition `[C_one]` is false. Event `E_one` is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram:

- 1 The Stateflow diagram root checks to see if there is a valid transition as a result of E_one. A potentially valid transition from state A to state B is detected. However, the transition is not valid, because [C_one] is false.
- 2 State A during actions (durA()) execute and complete.
- 3 State A checks its children for a valid transition and detects a valid inner transition.
- 4 State A remains active. The inner transition action A_two is executed and completed. Because it is an inner transition, state A's exit and entry actions are not executed.
- 5 The Stateflow diagram goes back to sleep, waiting to be awakened by another event.

This sequence completes the execution of this Stateflow diagram associated with event E_one.

Processing a Second Event in an Exclusive (OR) State

Using the previous example, this example shows what happens when a second event E_one occurs.



Initially the Stateflow diagram is asleep. State A is still active. Event E_one occurs and awakens the Stateflow diagram. Condition [C_one] is true. Event E_one is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram with the following steps:

- 1 The Stateflow diagram root checks to see if there is a valid transition as a result of E_one.

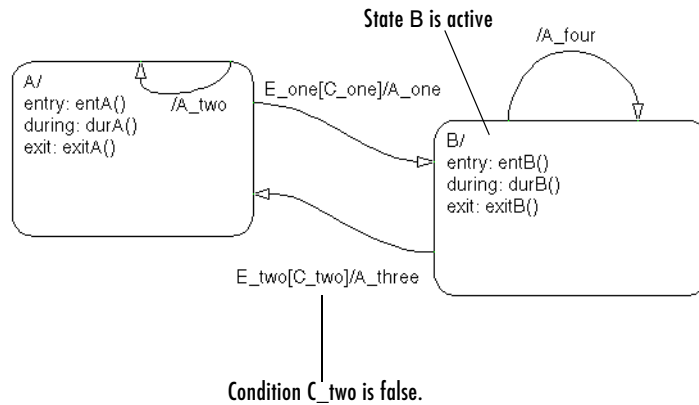
The transition from state A to state B is now valid because [C_one] is true.

- 2 State A exit actions (exitA()) execute and complete.
- 3 State A is marked inactive.
- 4 The transition action A_one is executed and completed.
- 5 State B is marked active.
- 6 State B entry actions (entB()) execute and complete.
- 7 The Stateflow diagram goes back to sleep, waiting to be awakened by another event.

This sequence completes the execution of this Stateflow diagram associated with event E_one.

Processing a Third Event in an Exclusive (OR) State

Using the previous example, this example shows what happens when a third event, E_two, occurs.



Initially the Stateflow diagram is asleep. State B is now active. Event E_two occurs and awakens the Stateflow diagram. Condition [C_two] is false. Event E_two is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram:

- 1 The Stateflow diagram root checks to see if there is a valid transition as a result of E_two.
 - A potentially valid transition from state B to state A is detected. The transition is not valid because [C_two] is false. However, active state B has a valid self-loop transition.
- 2 State B exit actions (exitB()) execute and complete.
- 3 State B is marked inactive.
- 4 The self-loop transition action, A_four, executes and completes.
- 5 State B is marked active.
- 6 State B entry actions (entB()) execute and complete.
- 7 The Stateflow diagram goes back to sleep, waiting to be awakened by another event.

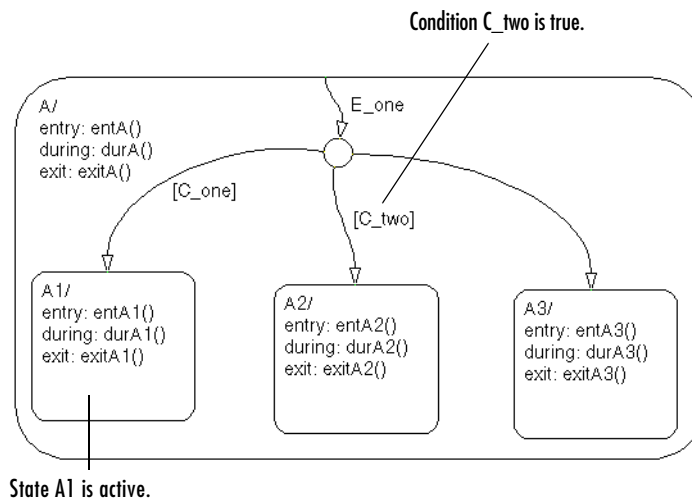
This sequence completes the execution of this Stateflow diagram associated with event E_two. This example shows the difference in behavior between inner and self-loop transitions.

Processing Events with an Inner Transition to a Connective Junction Example

This example shows the behavior of handling repeated events using an inner transition to a connective junction.

Processing the First Event with an Inner Transition to a Connective Junction

This example shows the behavior of an inner transition to a connective junction for an initial event.



Initially the Stateflow diagram is asleep. State A1 is active. Event E_one occurs and awakens the Stateflow diagram. Condition [C_two] is true. Event E_one is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram:

- 1 The Stateflow diagram root checks to see if there is a valid transition at the root level as a result of E_one. There is no valid transition.

- 2** State A during actions (`durA()`) execute and complete.
- 3** State A checks itself for valid transitions and detects that there is a valid inner transition to a connective junction.

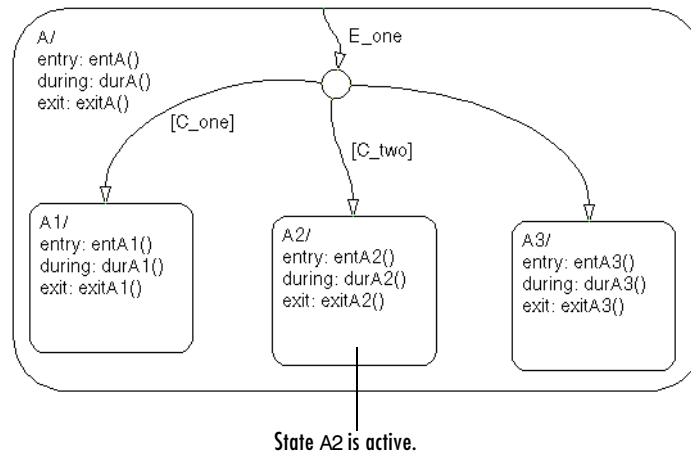
The conditions are evaluated to determine whether one of the transitions is valid. The segments labeled with a condition are evaluated before the unlabeled segment. The evaluation starts from a twelve o'clock position on the junction and progresses in a clockwise manner. Because `[C_two]` is true, the inner transition to the junction and then to state A.A2 is valid.

- 4** State A.A1 exit actions (`exitA1()`) execute and complete.
- 5** State A.A1 is marked inactive.
- 6** State A.A2 is marked active.
- 7** State A.A2 entry actions (`entA2()`) execute and complete.
- 8** The Stateflow diagram goes back to sleep, waiting to be awakened by another event.

This sequence completes the execution of this Stateflow diagram associated with event `E_one` when condition `C_two` is true.

Processing a Second Event with an Inner Transition to a Connective Junction

Continuing the previous example, this example shows the behavior of an inner transition to a junction when a second event `E_one` occurs.



Initially the Stateflow diagram is asleep. State A2 is active. Event E_one occurs and awakens the Stateflow diagram. Neither [C_one] nor [C_two] is true. Event E_one is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram:

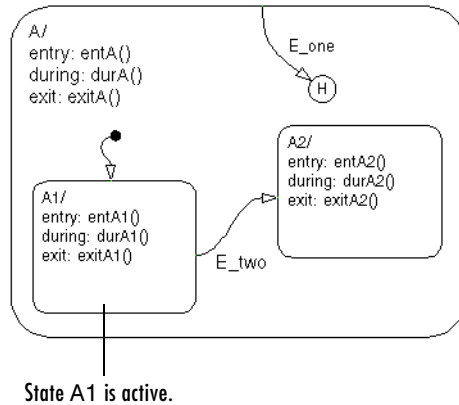
- 1 The Stateflow diagram root checks to see if there is a valid transition at the root level as a result of E_one. There is no valid transition.
- 2 State A during actions (durA()) execute and complete.
- 3 State A checks itself for valid transitions and detects a valid inner transition to a connective junction. The segments labeled with a condition are evaluated before the unlabeled segment. The evaluation starts from a twelve o'clock position on the junction and progresses in a clockwise manner. Because neither [C_one] nor [C_two] is true, the unlabeled transition segment is evaluated and is determined to be valid. The full transition from the inner transition to state A.A3 is valid.
- 4 State A.A2 exit actions (exitA2()) execute and complete.
- 5 State A.A2 is marked inactive.
- 6 State A.A3 is marked active.

- 7** State A.A3 entry actions (entA3 ()) execute and complete.
- 8** The Stateflow diagram goes back to sleep, waiting to be awakened by another event.

This sequence completes the execution of this Stateflow diagram associated with event E_one when neither [C_one] nor [C_two] is true.

Inner Transition to a History Junction Example

This example shows the behavior of an inner transition to a history junction.



Initially the Stateflow diagram is asleep. State A.A1 is active. There is history information because superstate A is active. Event E_one occurs and awakens the Stateflow diagram. Event E_one is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram:

- 1** The Stateflow diagram root checks to see if there is a valid transition as a result of E_one. There is no valid transition.
- 2** State A during actions execute and complete.
- 3** State A checks itself for valid transitions and detects that there is a valid inner transition to a history junction. According to the behavior of history junctions, the last active state, A.A1, is the destination state.

- 4** State A.A1 exit actions execute and complete.
- 5** State A.A1 is marked inactive.
- 6** State A.A1 is marked active.
- 7** State A.A1 entry actions execute and complete.
- 8** The Stateflow diagram goes back to sleep, waiting to be awakened by another event.

This sequence completes the execution of this Stateflow diagram associated with event E_one when there is an inner transition to a history junction and state A.A1 is active.

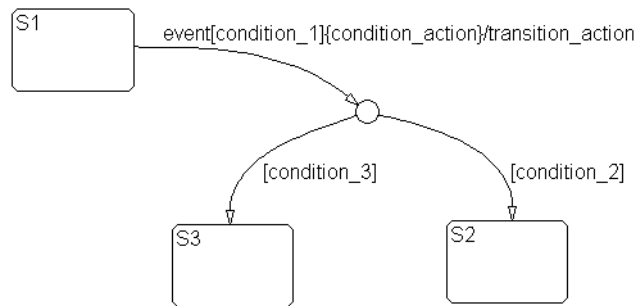
Connective Junction Examples

The following examples demonstrate the use of connective junctions in Stateflow:

- “Label Format for Transition Segments Example” on page 3-58 — Shows the general label format for a transition segment.
- “If-Then-Else Decision Construct Example” on page 3-59 — Shows the behavior of an if-then-else decision construct using a connective junction.
- “Self-Loop Transition Example” on page 3-61 — Shows the behavior of a self-loop transition using a connective junction.
- “For Loop Construct Example” on page 3-62 — Shows the behavior of a for loop using a connective junction.
- “Flow Diagram Notation Example” on page 3-63 — Shows the behavior of a flow notation in a Stateflow diagram.
- “Transitions from a Common Source to Multiple Destinations Example” on page 3-65 — Shows the behavior of transitions from a common source to multiple conditional destinations using a connective junction.
- “Transitions from Multiple Sources to a Common Destination Example” on page 3-67 — Shows the behavior of transitions from multiple sources to a single destination using a connective junction.
- “Transitions from a Source to a Destination Based on a Common Event Example” on page 3-68 — Shows the behavior of transitions from multiple sources to a single destination based on the same event using a connective junction.
- “Backtracking Behavior in Flow Graphs Example” on page 3-69 — Shows the behavior of transitions with junctions that force back-tracking behavior in flow graphs.

Label Format for Transition Segments Example

The general label format for a transition segment entering a junction is the same as for transitions entering states, as shown in the following example:

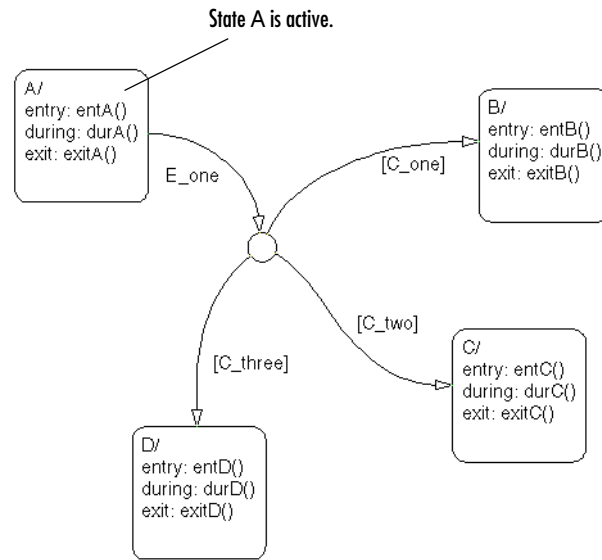


Execution of a transition in this example occurs as follows:

- 1 When an event occurs, state S1 is checked for an outgoing transition with a matching event specified.
- 2 If a transition with a matching event is found, the transition condition for that transition (in brackets) is evaluated.
- 3 If condition_1 evaluates to true, the condition action condition_action (in braces) is executed.
- 4 The outgoing transitions from the junction are checked for a valid transition. Since condition_2 is true, a valid state-to-state transition (S1 to S2) is found.
- 5 State S1 is exited (this includes the execution of S1's exit action).
- 6 The transition action transition_action is executed.
- 7 The completed state-to-state transition (S1 to S2) is taken.
- 8 State S2 is entered (this includes the execution of S2's entry action).

If-Then-Else Decision Construct Example

This example shows the behavior of an if-then-else decision construct.



Initially the Stateflow diagram is asleep. State A is active. Event E_one occurs and awakens the Stateflow diagram. Condition [C_two] is true. Event E_one is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram:

- 1 The Stateflow diagram root checks to see if there is a valid transition as a result of E_one.

There is a valid transition segment from state A to the connective junction. The transition segments beginning from a twelve o'clock position on the connective junction are evaluated for validity. The first transition segment, labeled with condition [C_one], is not valid. The next transition segment, labeled with the condition [C_two], is valid. The complete transition from state A to state C is valid.

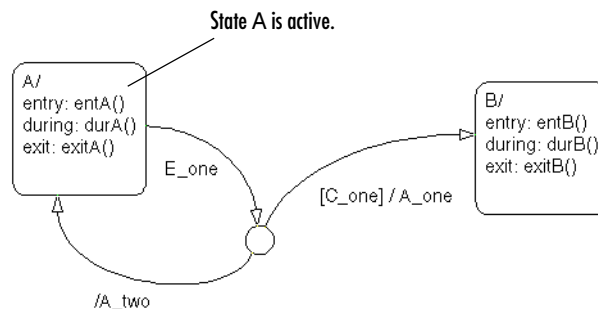
- 2 State A exit actions (exitA()) execute and complete.
- 3 State A is marked inactive.
- 4 State C is marked active.

- 5 State C entry actions (entC()) execute and complete.
- 6 The Stateflow diagram goes back to sleep, waiting to be awakened by another event.

This sequence completes the execution of this Stateflow diagram associated with event E_one.

Self-Loop Transition Example

This example shows the behavior of a self-loop transition using a connective junction.



Initially the Stateflow diagram is asleep. State A is active. Event E_one occurs and awakens the Stateflow diagram. Condition [C_one] is false. Event E_one is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram:

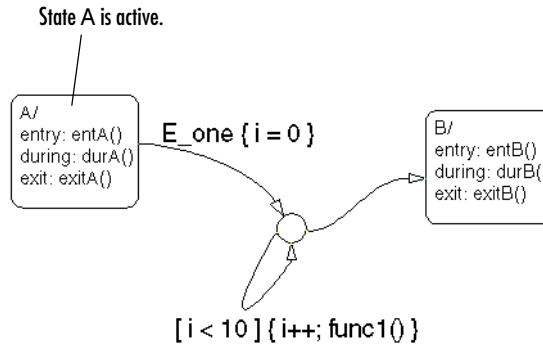
- 1 The Stateflow diagram root checks to see if there is a valid transition as a result of E_one. There is a valid transition segment from state A to the connective junction. The transition segment labeled with a condition and action is evaluated for validity. Because the condition [C_one] is not valid, the complete transition from state A to state B is not valid. The transition segment from the connective junction back to state A is valid.
- 2 State A exit actions (exitA()) execute and complete.
- 3 State A is marked inactive.

- 4 The transition action A_two is executed and completed.
- 5 State A is marked active.
- 6 State A entry actions (entA()) execute and complete.
- 7 The Stateflow diagram goes back to sleep, waiting to be awakened by another event.

This sequence completes the execution of this Stateflow diagram associated with event E_one.

For Loop Construct Example

This example shows the behavior of a for loop using a connective junction.



Initially the Stateflow diagram is asleep. State A is active. Event E_one occurs and awakens the Stateflow diagram. Event E_one is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram:

- 1 The Stateflow diagram root checks to see if there is a valid transition as a result of E_one. There is a valid transition segment from state A to the connective junction. The transition segment condition action, $i = 0$, is executed and completed. Of the two transition segments leaving the connective junction, the transition segment that is a self-loop back to the connective junction is evaluated next for validity. That segment takes

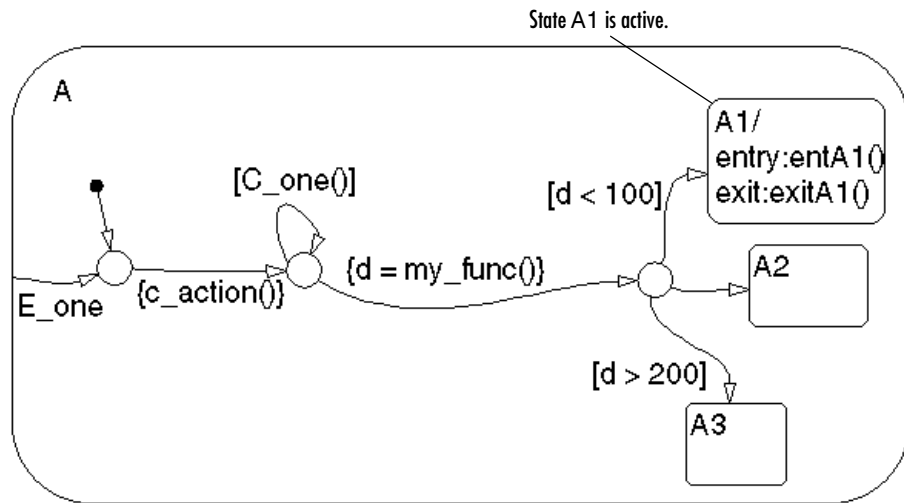
priority in evaluation because it has a condition specified, whereas the other segment is unlabeled.

- 2** The condition `[i < 10]` is evaluated as true. The condition actions `i++` and a call to `func1` are executed and completed until the condition becomes false. A connective junction is not a final destination; thus the transition destination remains to be determined.
- 3** The unconditional segment to state B is now valid. The complete transition from state A to state B is valid.
- 4** State A exit actions (`exitA()`) execute and complete.
- 5** State A is marked inactive.
- 6** State B is marked active.
- 7** State B entry actions (`entB()`) execute and complete.
- 8** The Stateflow diagram goes back to sleep, waiting to be awakened by another event.

This sequence completes the execution of this Stateflow diagram associated with event `E_one`.

Flow Diagram Notation Example

This example shows the behavior of a Stateflow diagram that uses flow notation.



Initially the Stateflow diagram is asleep. State A.A1 is active. The condition [C_one()] is initially true. Event E_one occurs and awakens the Stateflow diagram. Event E_one is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram:

- 1 The Stateflow diagram root checks to see if there is a valid transition as a result of E_one. There is no valid transition.
- 2 State A checks itself for valid transitions and detects a valid inner transition to a connective junction.
- 3 The next possible segments of the transition are evaluated. There is only one outgoing transition and it has a condition action defined. The condition action is executed and completed.
- 4 The next possible segments are evaluated. There are two outgoing transitions; one is a conditional self-loop transition and the other is an unconditional transition segment. The conditional transition segment takes precedence. The condition [C_one()] is tested and is true; the self-loop transition is taken. Since a final transition destination has not been reached, this self-loop continues until [C_one()] is false.

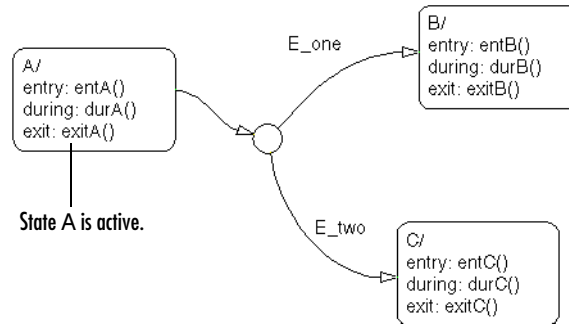
Assume that after five iterations [C_one()] is false.

- 5 The next possible transition segment (to the next connective junction) is evaluated. It is an unconditional transition segment with a condition action. The transition segment is taken and the condition action, `{d=my_func() }`, is executed and completed. The returned value of `d` is 84.
- 6 The next possible transition segment is evaluated. There are three possible outgoing transition segments to consider. Two are conditional; one is unconditional. The segment labeled with the condition `[d<100]` is evaluated first based on the geometry of the two outgoing conditional transition segments. Because the return value of `d` is 84, the condition `[d<100]` is true and this transition (to the destination state `A.A1`) is valid.
- 7 State `A.A1` exit actions (`exitA1()`) execute and complete.
- 8 State `A.A1` is marked inactive.
- 9 State `A.A1` is marked active.
- 10 State `A.A1` entry actions (`entA1()`) execute and complete.
- 11 The Stateflow diagram goes back to sleep, waiting to be awakened by another event.

This sequence completes the execution of this Stateflow diagram associated with event `E_one`.

Transitions from a Common Source to Multiple Destinations Example

This example shows the behavior of transitions from a common source to multiple conditional destinations using a connective junction.



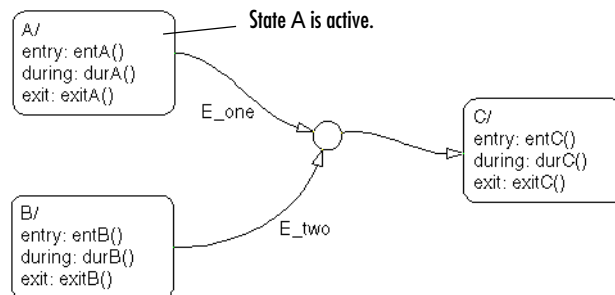
Initially the Stateflow diagram is asleep. State A is active. Event E_two occurs and awakens the Stateflow diagram. Event E_two is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram:

- 1 The Stateflow diagram root checks to see if there is a valid transition as a result of E_two. There is a valid transition segment from state A to the connective junction. Given that the transition segments are equivalently labeled, evaluation begins from a twelve o'clock position on the connective junction and progresses clockwise. The first transition segment, labeled with event E_one, is not valid. The next transition segment, labeled with event E_two, is valid. The complete transition from state A to state C is valid.
- 2 State A exit actions (exitA()) execute and complete.
- 3 State A is marked inactive.
- 4 State C is marked active.
- 5 State C entry actions (entC()) execute and complete.
- 6 The Stateflow diagram goes back to sleep, waiting to be awakened by another event.

This sequence completes the execution of this Stateflow diagram associated with event E_two.

Transitions from Multiple Sources to a Common Destination Example

This example shows the behavior of transitions from multiple sources to a single destination using a connective junction.



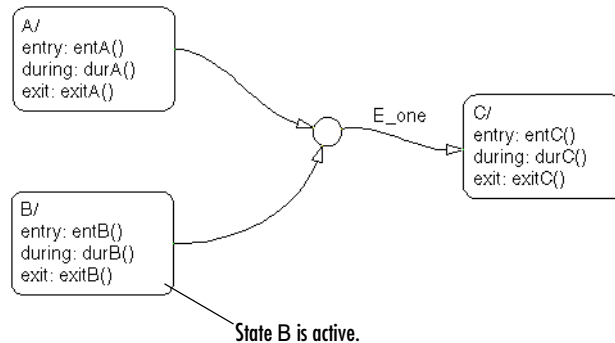
Initially the Stateflow diagram is asleep. State A is active. Event E_one occurs and awakens the Stateflow diagram. Event E_one is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram:

- 1 The Stateflow diagram root checks to see if there is a valid transition as a result of E_one. There is a valid transition segment from state A to the connective junction and from the junction to state C.
- 2 State A exit actions (exitA()) execute and complete.
- 3 State A is marked inactive.
- 4 State C is marked active.
- 5 State C entry actions (entC()) execute and complete.
- 6 The Stateflow diagram goes back to sleep, waiting to be awakened by another event.

This sequence completes the execution of this Stateflow diagram associated with event E_one.

Transitions from a Source to a Destination Based on a Common Event Example

This example shows the behavior of transitions from multiple sources to a single destination based on the same event using a connective junction.



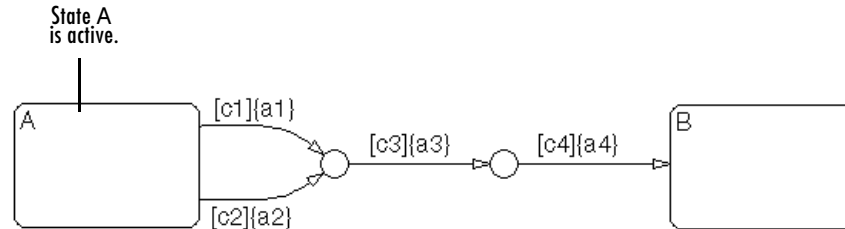
Initially the Stateflow diagram is asleep. State B is active. Event E_one occurs and awakens the Stateflow diagram. Event E_one is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram:

- 1 The Stateflow diagram root checks to see if there is a valid transition as a result of E_one. There is a valid transition segment from state B to the connective junction and from the junction to state C.
- 2 State B exit actions (exitB()) execute and complete.
- 3 State B is marked inactive.
- 4 State C is marked active.
- 5 State C entry actions (entC()) execute and complete.
- 6 The Stateflow diagram goes back to sleep, waiting to be awakened by another event.

This sequence completes the execution of this Stateflow diagram associated with event E_one.

Backtracking Behavior in Flow Graphs Example

This example shows the behavior of transitions with junctions that force back-tracking behavior in flow graphs.



Initially, state A is active and conditions c1, c2, and c3 are true:

- 1 The Stateflow diagram root checks to see if there is a valid transition from state A.

There is a valid transition segment marked with the condition c1 from state A to a connective junction.

- 2 Condition c1 is true, therefore action a1 is executed.
- 3 Condition c3 is true, therefore action a3 is executed.

- 4 Condition c4 is not true, therefore control flow is back-tracked to state A.

- 5 The Stateflow diagram root checks to see if there is another valid transition from state A.

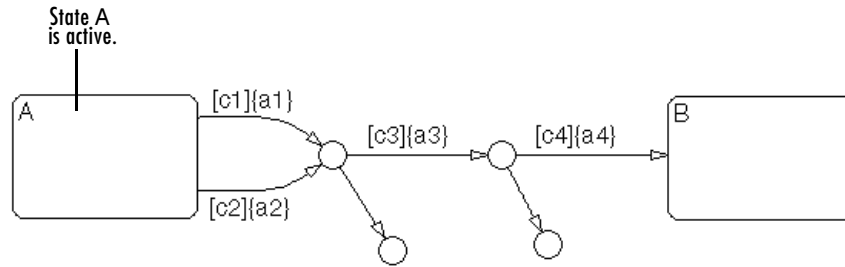
There is a valid transition segment marked with the condition c2 from state A to a connective junction.

- 6 Condition c2 is true, therefore action a2 is executed.
- 7 Condition c3 is true, therefore action a3 is executed.

- 8 Condition c4 is not true, therefore control flow is back-tracked to state A.

- 9 The Stateflow chart goes to sleep.

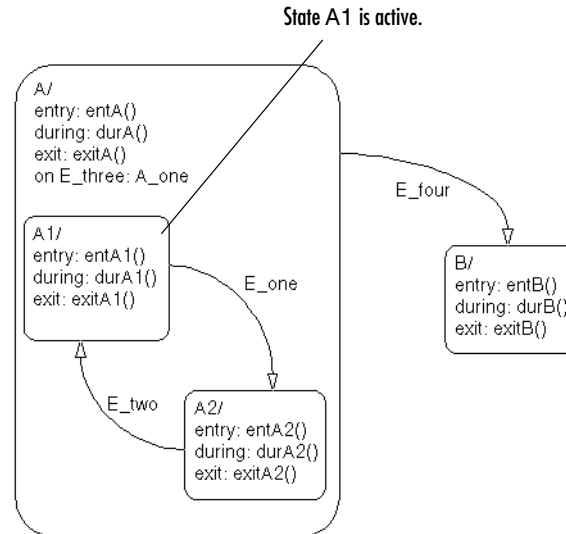
The preceding example shows the unanticipated behavior of executing both actions a1 and a2 and executing action a3 twice. To resolve this problem, consider the following.



In this example, the previous example is amended with two terminating junctions that allow flow to terminate if either c3 or c4 is not true. This leaves state A active without taking any unnecessary actions.

Event Actions in a Superstate Example

The following example demonstrates the use of event actions in a superstate:



Initially the Stateflow diagram is asleep. State A.A1 is active. Event `E_three` occurs and awakens the Stateflow diagram. Event `E_three` is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram:

- 1 The Stateflow diagram root checks to see if there is a valid transition as a result of `E_three`. There is no valid transition.
- 2 State A during actions (`durA()`) execute and complete.
- 3 State A executes and completes the on event `E_three` action (`A_one`).
- 4 State A checks its children for valid transitions. There are no valid transitions.
- 5 State A1 during actions (`durA1()`) execute and complete.

- 6 The Stateflow diagram goes back to sleep, waiting to be awakened by another event.

This sequence completes the execution of this Stateflow diagram associated with event E_three.

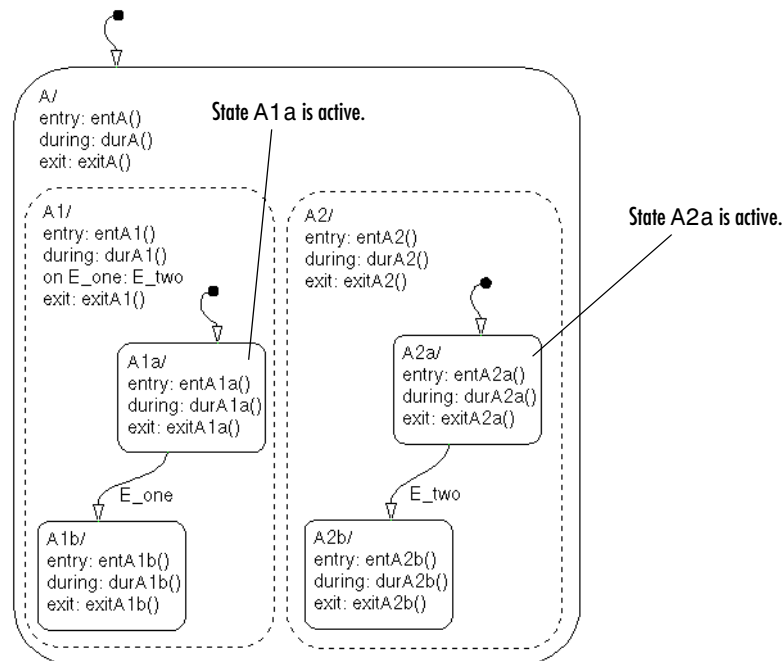
Parallel (AND) State Examples

The following examples demonstrate the use of parallel (AND) states:

- “Event Broadcast State Action Example” on page 3-73 — Shows the behavior of event broadcast actions in parallel states.
- “Event Broadcast Transition Action with a Nested Event Broadcast Example” on page 3-77 — Shows the behavior of an event broadcast transition action that includes a nested event broadcast in a parallel state.
- “Event Broadcast Condition Action Example” on page 3-81 — Shows the behavior of a condition action event broadcast in a parallel (AND) state.

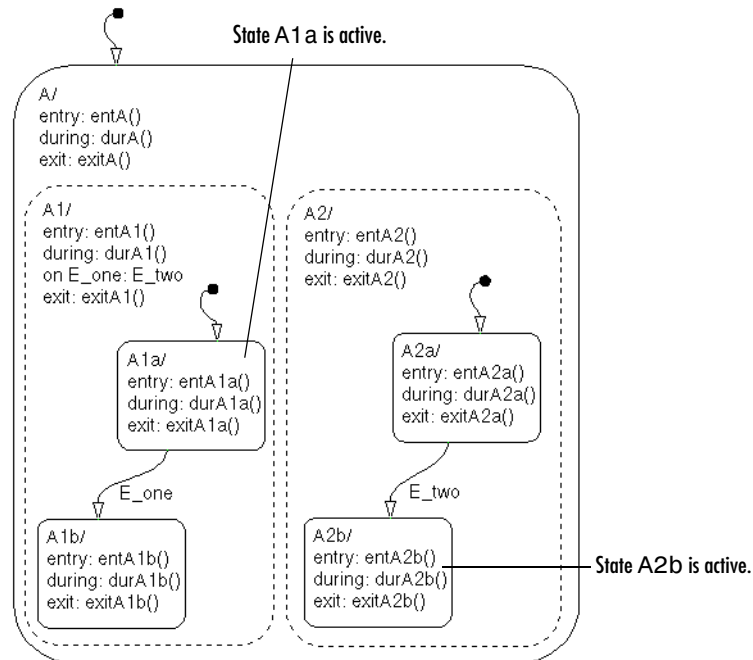
Event Broadcast State Action Example

This example shows the behavior of event broadcast actions in parallel states.



Initially the Stateflow diagram is asleep. Parallel substates A.A1.A1a and A.A2.A2a are active. Event E_one occurs and awakens the Stateflow diagram. Event E_one is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram:

- 1** The Stateflow diagram root checks to see if there is a valid transition at the root level as a result of E_one. There is no valid transition.
- 2** State A during actions (durA()) execute and complete.
- 3** State A's children are parallel (AND) states. They are evaluated and executed from left to right and top to bottom. State A.A1 is evaluated first. State A.A1 during actions (durA1()) execute and complete. State A.A1 executes and completes the on E_one action and broadcasts event E_two. during and on *event_name* actions are processed based on their order of appearance in the state label:
 - a** The broadcast of event E_two awakens the Stateflow diagram a second time. The Stateflow diagram root checks to see if there is a valid transition as a result of E_two. There is no valid transition.
 - b** State A during actions (durA()) execute and complete.
 - c** State A checks its children for valid transitions. There are no valid transitions.
 - d** State A's children are evaluated starting with state A.A1. State A.A1 during actions (durA1()) execute and complete. State A.A1 is evaluated for valid transitions. There are no valid transitions as a result of E_two within state A1.
 - e** State A.A2 is evaluated. State A.A2 during actions (durA2()) execute and complete. State A.A2 checks for valid transitions. State A.A2 has a valid transition as a result of E_two from state A.A2.A2a to state A.A2.A2b.
 - f** State A.A2.A2a exit actions (exitA2a()) execute and complete.
 - g** State A.A2.A2a is marked inactive.
 - h** State A.A2.A2b is marked active.
 - i** State A.A2.A2b entry actions (entA2b()) execute and complete. The Stateflow diagram activity now looks like this:



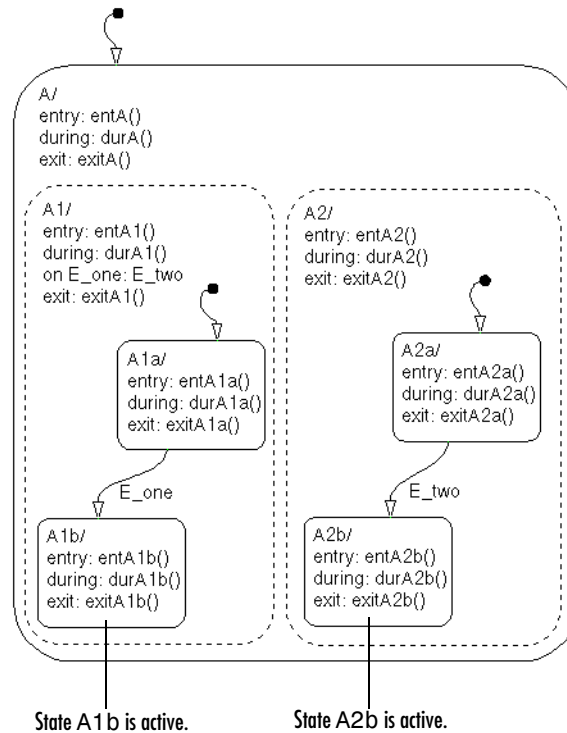
- 4 State A.A1.A1a executes and completes exit actions (exitA1a).
- 5 The processing of E_one continues once the on event broadcast of E_two has been processed. State A.A1 checks for any valid transitions as a result of event E_one. There is a valid transition from state A.A1.A1a to state A.A1.A1b.
- 6 State A.A1.A1a is marked inactive.
- 7 State A.A1.A1b entry actions (entA1b()) execute and complete.
- 8 State A.A1.A1b is marked active.
- 9 Parallel state A.A2 is evaluated next. State A.A2 during actions (durA2()) execute and complete. There are no valid transitions as a result of E_one.

10 State A.A2.A2b during actions (durA2b()) execute and complete.

State A.A2.A2b is now active as a result of the processing of the on event broadcast of E_two.

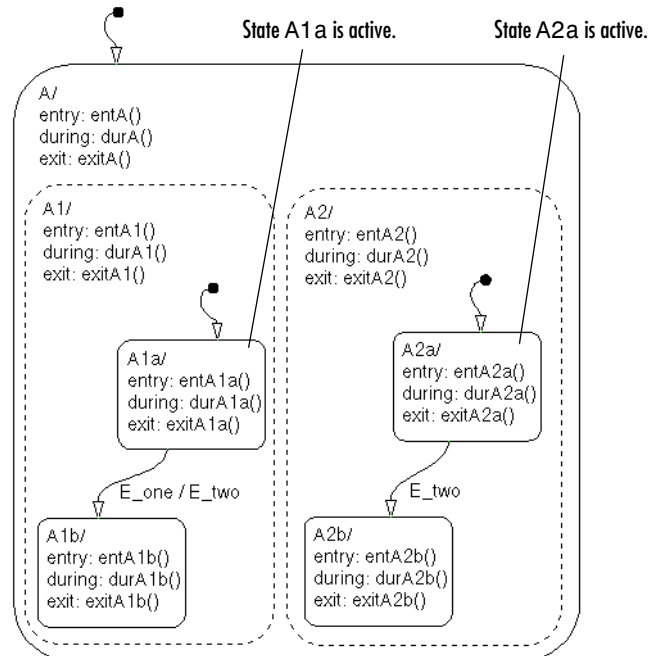
11 The Stateflow diagram goes back to sleep, waiting to be awakened by another event.

This sequence completes the execution of this Stateflow diagram associated with event E_one and the on event broadcast to a parallel state of event E_two. The final Stateflow diagram activity looks like this.



Event Broadcast Transition Action with a Nested Event Broadcast Example

This example shows the behavior of an event broadcast transition action that includes a nested event broadcast in a parallel state.



Start of Event **E_one** Processing

Initially the Stateflow diagram is asleep. Parallel substates A.A1.A1a and A.A2.A2a are active. Event **E_one** occurs and awakens the Stateflow diagram. Event **E_one** is processed from the root of the Stateflow diagram through the hierarchy of the Stateflow diagram:

- 1 The Stateflow diagram root checks to see if there is a valid transition as a result of **E_one**. There is no valid transition.
- 2 State A during actions (**durA()**) execute and complete.

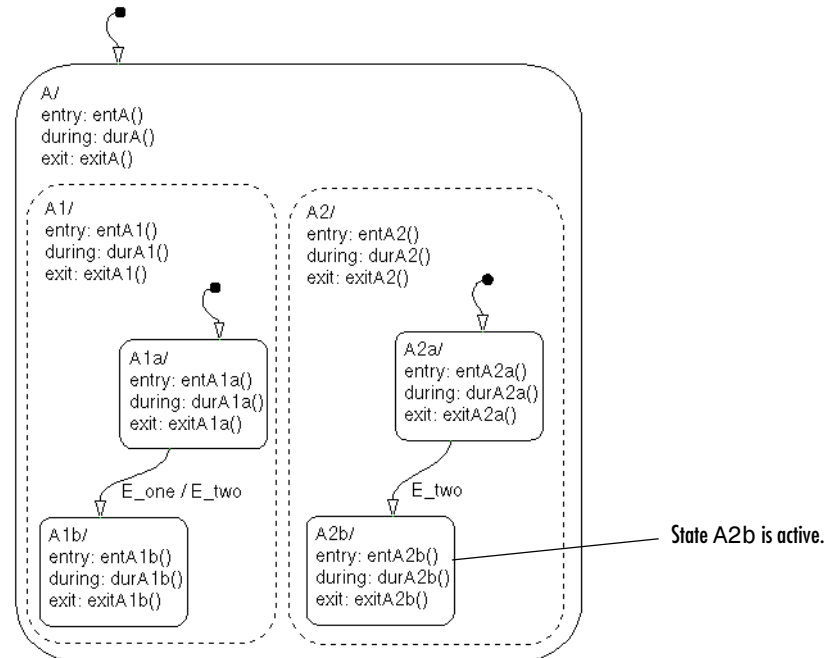
- 3** State A's children are parallel (AND) states. They are evaluated and executed from left to right and top to bottom. State A.A1 is evaluated first. State A.A1 during actions (durA1()) execute and complete.
- 4** State A.A1 checks for any valid transitions as a result of event E_one. There is a valid transition from state A.A1.A1a to state A.A1.A1b.
- 5** State A.A1.A1a executes and completes exit actions (exitA1a).
- 6** State A.A1.A1a is marked inactive.

Event E_two Preempts E_one

- 7** Transition action generating event E_two is executed and completed:
 - a** The transition from state A1a to state A1b (as a result of event E_one) is now preempted by the broadcast of event E_two.
 - b** The broadcast of event E_two awakens the Stateflow diagram a second time. The Stateflow diagram root checks to see if there is a valid transition as a result of E_two. There is no valid transition.
 - c** State A during actions (durA()) execute and complete.
 - d** State A's children are evaluated starting with state A.A1. State A.A1 during actions (durA1()) execute and complete. State A.A1 is evaluated for valid transitions. There are no valid transitions as a result of E_two within state A1.
 - e** State A.A2 is evaluated. State A.A2 during actions (durA2()) execute and complete. State A.A2 checks for valid transitions. State A.A2 has a valid transition as a result of E_two from state A.A2.A2a to state A.A2.A2b.
 - f** State A.A2.A2a exit actions (exitA2a()) execute and complete.
 - g** State A.A2.A2a is marked inactive.
 - h** State A.A2.A2b is marked active.
 - i** State A.A2.A2b entry actions (entA2b()) execute and complete.

Event E_two Processing Ends

The Stateflow diagram activity now looks like this.



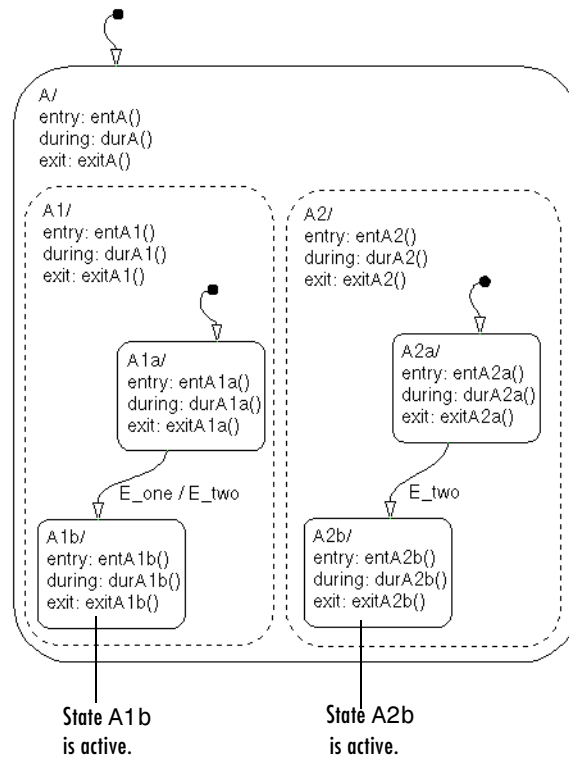
Event E_one Processing Resumes

- 8** State A.A1.A1b is marked active.
- 9** State A.A1.A1b entry actions (entA1b()) execute and complete.
- 10** Parallel state A.A2 is evaluated next. State A.A2 during actions (durA2()) execute and complete. There are no valid transitions as a result of E_one.
- 11** State A.A2.A2b during actions (durA2b()) execute and complete.

State A.A2.A2b is now active as a result of the processing of the transition action event broadcast of E_two.

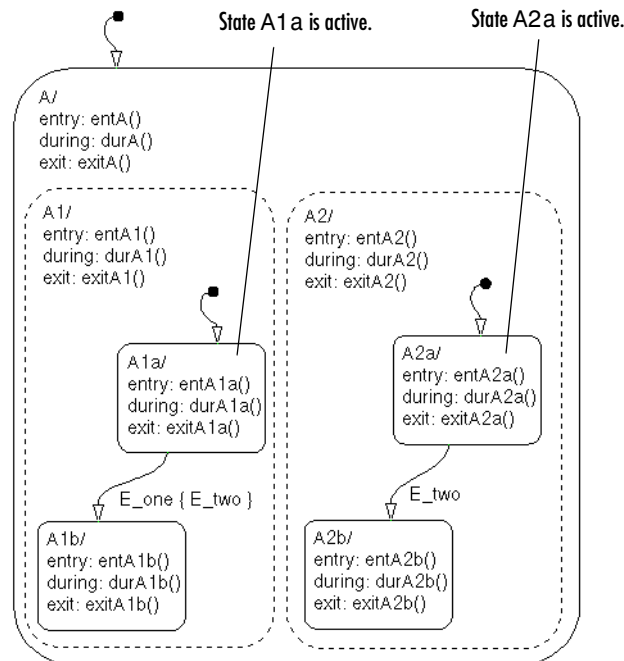
12 The Stateflow diagram goes back to sleep, waiting to be awakened by another event.

This sequence completes the execution of this Stateflow diagram associated with event E_one and the transition action event broadcast to a parallel state of event E_two. The final Stateflow diagram activity now looks like this.



Event Broadcast Condition Action Example

This example shows the behavior of a condition action event broadcast in a parallel (AND) state.

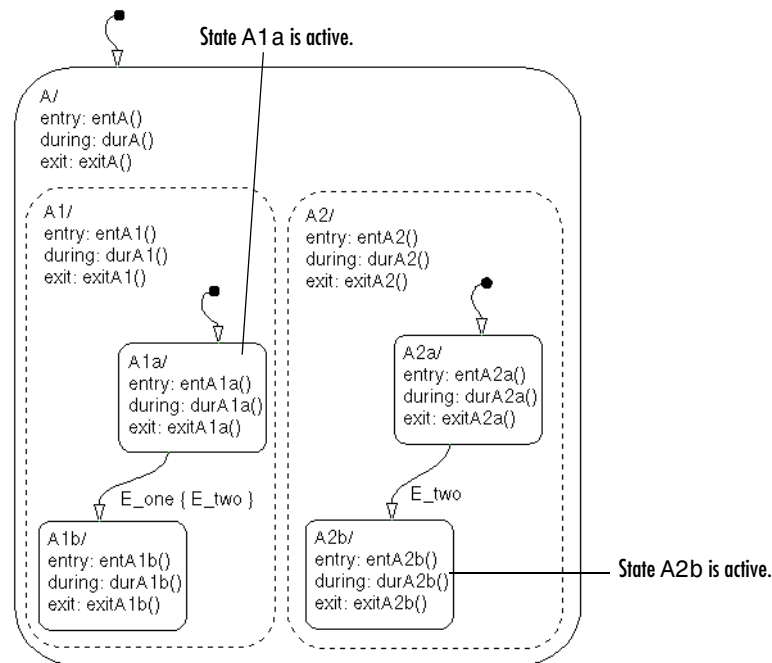


Initially the Stateflow diagram is asleep. Parallel substates A.A1.A1a and A.A2.A2a are active. Event E_one occurs and awakens the Stateflow diagram. Event E_one is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram:

- 1 The Stateflow diagram root checks to see if there is a valid transition as a result of E_one. There is no valid transition.
- 2 State A during actions (durA()) execute and complete.

- 3** State A's children are parallel (AND) states. Parallel states are evaluated and executed from top to bottom. In the case of a tie, they are evaluated from left to right. State A.A1 is evaluated first. State A.A1 during actions (durA1()) execute and complete.
- 4** State A.A1 checks for any valid transitions as a result of event E_one. There is a valid transition from state A.A1.A1a to state A.A1.A1b. There is also a valid condition action. The condition action event broadcast of E_two is executed and completed. State A.A1.A1a is still active:
 - a** The broadcast of event E_two awakens the Stateflow diagram a second time. The Stateflow diagram root checks to see if there is a valid transition as a result of E_two. There is no valid transition.
 - b** State A during actions (durA()) execute and complete.
 - c** State A's children are evaluated starting with state A.A1. State A.A1 during actions (durA1()) execute and complete. State A.A1 is evaluated for valid transitions. There are no valid transitions as a result of E_two within state A1.
 - d** State A.A2 is evaluated. State A.A2 during actions (durA2()) execute and complete. State A.A2 checks for valid transitions. State A.A2 has a valid transition as a result of E_two from state A.A2.A2a to state A.A2.A2b.
 - e** State A.A2.A2a exit actions (exitA2a()) execute and complete.
 - f** State A.A2.A2a is marked inactive.
 - g** State A.A2.A2b is marked active.
 - h** State A.A2.A2b entry actions (entA2b()) execute and complete.

The Stateflow diagram activity now looks like this.

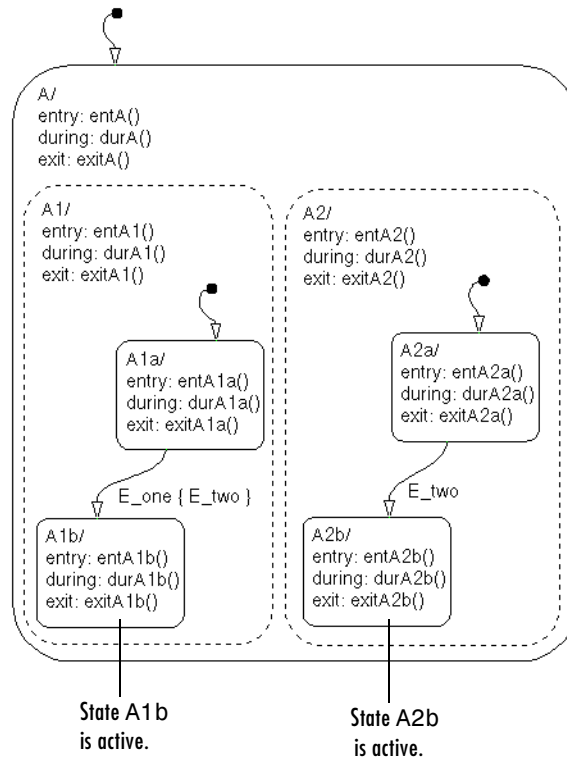


- 5 State A.A1.A1a executes and completes exit actions (`exitA1a()`).
- 6 State A.A1.A1a is marked inactive.
- 7 State A.A1.A1b entry actions (`entA1b()`) execute and complete.
- 8 State A.A1.A1b is marked active.
- 9 Parallel state A.A2 is evaluated next. State A.A2 during actions (`durA2()`) execute and complete. There are no valid transitions as a result of `E_one`.
- 10 State A.A2.A2b during actions (`durA2b()`) execute and complete.

State A.A2.A2b is now active as a result of the processing of the condition action event broadcast of `E_two`.

- 11** The Stateflow diagram goes back to sleep, waiting to be awakened by another event.

This sequence completes the execution of this Stateflow diagram associated with event E_one and the condition action event broadcast to a parallel state of event E_two. The final Stateflow diagram activity now looks like this.



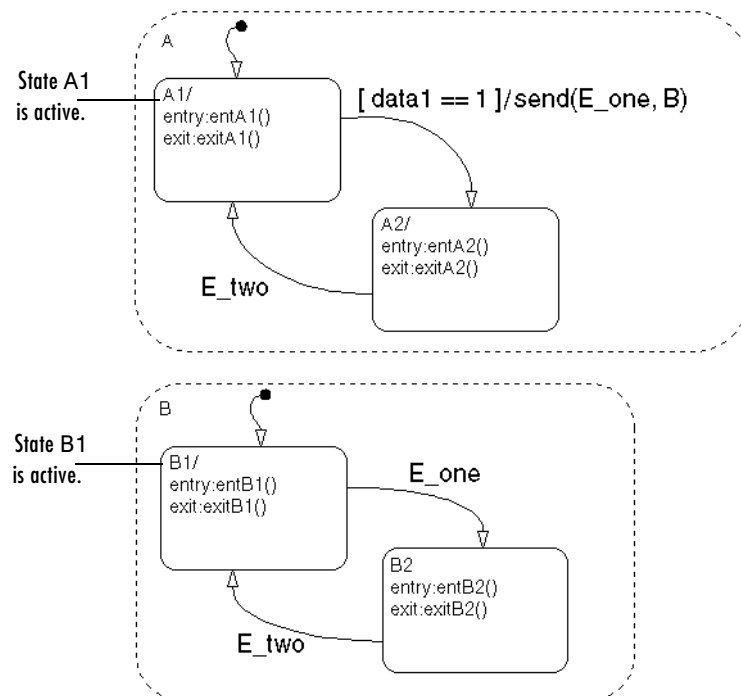
Directed Event Broadcasting Examples

The following examples demonstrate the use of directed event broadcasting:

- “Directed Event Broadcast Using send Example” on page 3-85 — Shows the behavior of directed event broadcast using the send function in a transition action
- “Directed Event Broadcasting Using Qualified Event Names Example” on page 3-87 — Shows the behavior of directed event broadcast using a qualified event name in a transition action

Directed Event Broadcast Using send Example

This example shows the behavior of directed event broadcast using the `send(event_name, state_name)` function in a transition action.



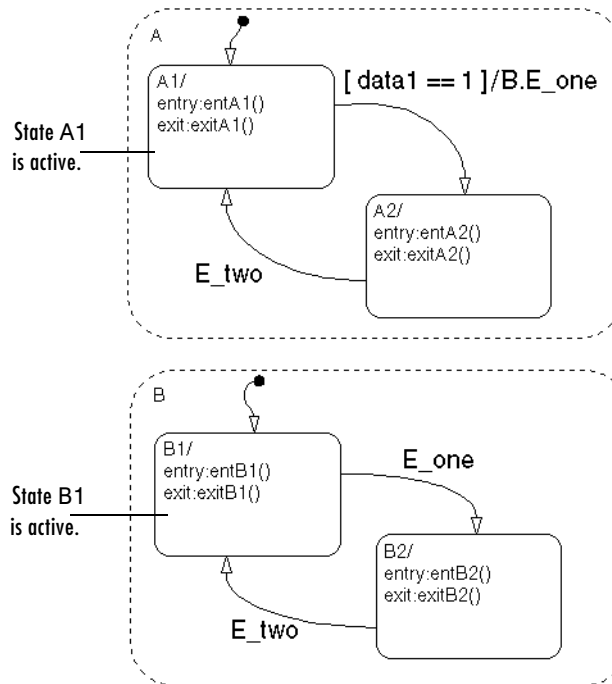
Initially the Stateflow diagram is asleep. Parallel substates A.A1 and B.B1 are active. By definition, this implies that parallel (AND) superstates A and B are active. An event occurs and awakens the Stateflow diagram. The condition [data1==1] is true. The event is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram:

- 1** The Stateflow diagram root checks to see if there is a valid transition as a result of the event. There is no valid transition.
- 2** State A checks for any valid transitions as a result of the event. Because the condition [data1==1] is true, there is a valid transition from state A.A1 to state A.A2.
- 3** State A.A1 exit actions (exitA1()) execute and complete.
- 4** State A.A1 is marked inactive.
- 5** The transition action send(E_one, B) is executed and completed:
 - a** The broadcast of event E_one awakens state B. (This is a nested event broadcast.) Because state B is active, the directed broadcast is received and state B checks to see if there is a valid transition. There is a valid transition from B.B1 to B.B2.
 - b** State B.B1 exit actions (exitB1()) execute and complete.
 - c** State B.B1 is marked inactive.
 - d** State B.B2 is marked active.
 - e** State B.B2 entry actions (entB2()) execute and complete.
- 6** State A.A2 is marked active.
- 7** State A.A2 entry actions (entA2()) execute and complete.

This sequence completes the execution of this Stateflow diagram associated with an event broadcast and the directed event broadcast to a parallel state of event E_one.

Directed Event Broadcasting Using Qualified Event Names Example

This example shows the behavior of directed event broadcast using a qualified event name in a transition action.



Initially the Stateflow diagram is asleep. Parallel substates A.A1 and B.B1 are active. By definition, this implies that parallel (AND) superstates A and B are active. An event occurs and awakens the Stateflow diagram. The condition `[data1==1]` is true. The event is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram:

- 1 The Stateflow diagram root checks to see if there is a valid transition as a result of the event. There is no valid transition.

- 2** State A checks for any valid transitions as a result of the event. Because the condition `[data1==1]` is true, there is a valid transition from state A.A1 to state A.A2.
- 3** State A.A1 exit actions (`exitA1()`) execute and complete.
- 4** State A.A1 is marked inactive.
- 5** The transition action, a qualified event broadcast of event `E_one` to state B (represented by the notation `B.E_one`), is executed and completed:
 - a** The broadcast of event `E_one` awakens state B. (This is a nested event broadcast.) Because state B is active, the directed broadcast is received and state B checks to see if there is a valid transition. There is a valid transition from B.B1 to B.B2.
 - b** State B.B1 exit actions (`exitB1()`) execute and complete.
 - c** State B.B1 is marked inactive.
 - d** State B.B2 is marked active.
 - e** State B.B2 entry actions (`entB2()`) execute and complete.
- 6** State A.A2 is marked active.
- 7** State A.A2 entry actions (`entA2()`) execute and complete.

This sequence completes the execution of this Stateflow diagram associated with an event broadcast using a qualified event name to a parallel state.

Creating Stateflow Chart Diagrams

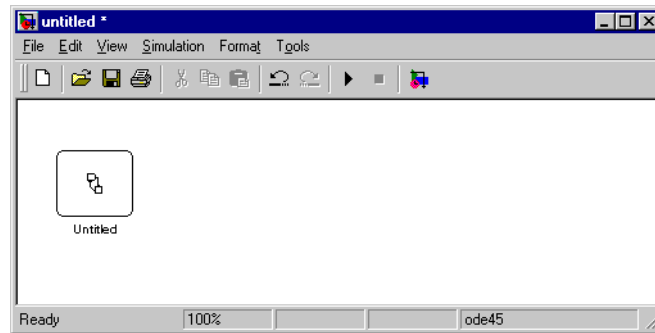
This chapter takes you through the steps of creating graphical Stateflow objects in the Stateflow diagram editor. It includes the following sections:

Creating a Stateflow Chart (p. 4-2)	Gives a step-by-step procedure for creating an empty Stateflow chart.
Creating States in Stateflow Charts (p. 4-5)	Describes how to create and specify a state in your new chart. Stateflow diagrams react to events by changing states, which are modes of a chart.
Creating Transitions in Stateflow Charts (p. 4-16)	Describes how to create, move, change, and specify properties for Stateflow transitions. Charts change active states using pathways called transitions.
Creating Flowcharts with Connective Junctions (p. 4-25)	Describes how to create, move, and specify properties for Stateflow junctions. Junctions provide decision points between alternate transition paths. History junctions record the activity of states inside states.
Using the Stateflow Editor (p. 4-28)	Describes each part of the Stateflow diagram editor window that displays the chart you create.

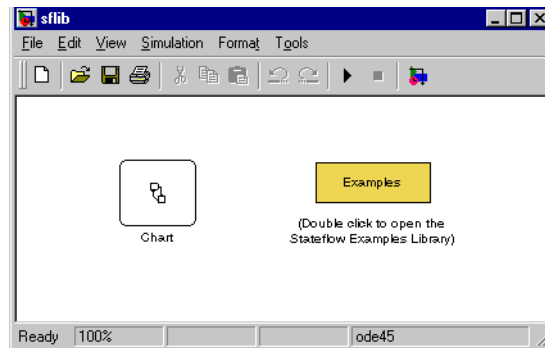
Creating a Stateflow Chart

Charts contain a Stateflow diagram that you build with Stateflow objects. You create charts by adding them to a Simulink system. Create a Stateflow chart in a Simulink system with the following steps:

- 1 Enter `sfnew` or `stateflow` at the MATLAB command prompt to create a new empty model with a Stateflow chart.



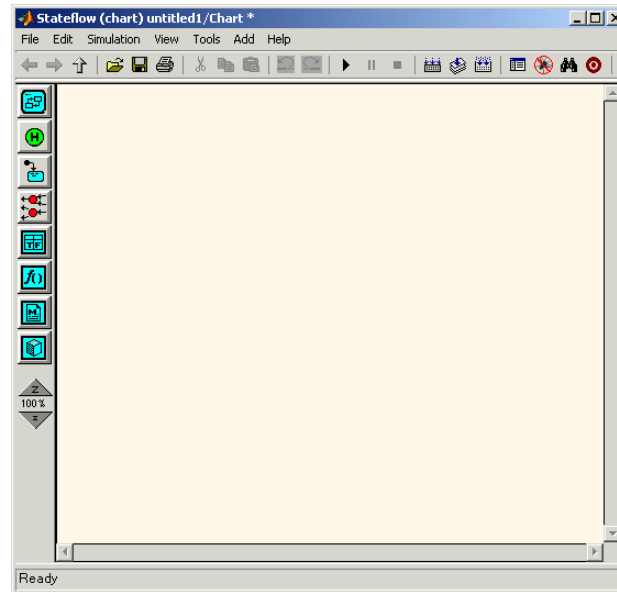
The `stateflow` command also displays the Stateflow block library.



You can drag and drop additional charts in your Simulink system from this library in case you want to create multiple charts in your model. You can also drag and drop new charts into existing systems from the Stateflow library in the Simulink Library browser. For information on creating your own chart libraries, see “Creating Chart Libraries” on page 9-23.

- 2 Open the chart by double-clicking the Chart block.

Stateflow opens the empty chart in a Stateflow editor window.



- 3 Use the Stateflow editor to draw a Stateflow chart diagram.

See “Using the Stateflow Editor” on page 4-28 and the remaining sections in this chapter for more information on how to draw Stateflow diagrams.

Once you have constructed a Stateflow diagram, continue with the remaining steps to integrate your chart into its Simulink model:

- 4 Specify an update method for the chart.

You set the update method for a chart by setting the **Update Method** field in the properties dialog for the chart. This value determines when and how often the chart is called during the execution of the Simulink model that calls this chart. See “Specifying Chart Properties” on page 9-4.

- 5 Interface the chart to other blocks in your Stateflow model, using events and data.

See “Defining Events and Data” on page 6-1 and “Defining Interfaces to Simulink and MATLAB” on page 9-1 for more information.

- 6 Rename and save the model chart by selecting **Save Model As** from the Stateflow editor menu or **Save As** from the Simulink menu.

Note Trying to save a model with more than 25 characters produces an error. Loading a model with more than 25 characters produces a warning.

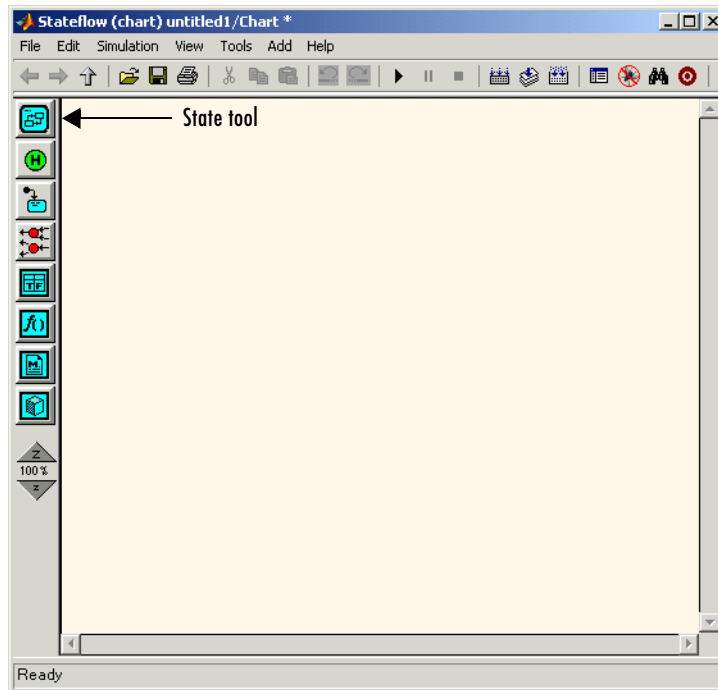
Creating States in Stateflow Charts

Stateflow diagrams react to events by changing states, which are modes of a chart. This section describes how to create and specify a state in your new chart and contains the following topics:

- “Creating a State” on page 4-5 — Shows you how to draw states in the Stateflow diagram editor with the State drawing tool
- “Moving and Resizing States” on page 4-7 — Tells you how to move and resize states in the diagram editor
- “Creating Substates and Superstates” on page 4-7 — Tells you how to create substates of owning superstates
- “Grouping States” on page 4-8 — Tells you how to group (and ungroup) states, boxes, and functions for convenience
- “Specifying Substate Decomposition” on page 4-8 — Tells you how to specify the substate decomposition type (parallel or exclusive) for a state or chart
- “Specifying Activation Order for Parallel States” on page 4-9 — Tells you how to specify the order of activation for parallel (AND) substates in a state or chart with parallel (AND) decomposition
- “Changing State Properties” on page 4-9 — Tells you how to change the properties for a state
- “Labeling States” on page 4-11 — Tells you how to label a state with its name and actions
- “Outputting State Activity to Simulink” on page 4-14 — Tells you how to output a state’s activity as an output port signal on a Stateflow block

Creating a State

You create states by drawing them in the Stateflow diagram editor for a particular Stateflow chart (block). The following is a depiction of the Stateflow diagram editor:



1 Select the State tool.

2 Move the mouse cursor into the drawing area.

In the drawing area, the mouse cursor becomes state-shaped (rectangular with oval corners).

3 Click in a particular location to create a state.

The created state appears with a question mark (?) label in its upper left-hand corner.

4 Click the question mark.

A text cursor appears in place of the question mark.

- 5 Enter a name for the state and click outside of the state when finished.

The label for a state specifies its required name and optional actions. See “Labeling States” on page 4-11 for more detail.

To delete a state, click it to select it and choose **Cut (Ctrl+X)** from the **Edit** or any shortcut menu or press the **Delete** key.

Moving and Resizing States

To move a state, do the following:

- 1 Click and drag the state.
- 2 Release it in a new position.

To resize a state, do the following:

- 1 Place the cursor over a corner of the state.

When the cursor is over a corner, it appears as a double-ended arrow (PC only; cursor appearance varies with other platforms).

- 2 Click and drag the state’s corner to resize the state and release the left mouse button.

Creating Substates and Superstates

A *substate* is a state that can be active only when another state, called its parent, is active. States that have substates are known as *superstates*. To create a substate, click the State tool and drag a new state into the state you want to be the superstate. Stateflow creates the substate in the specified parent state. You can nest states in this way to any depth. To change a substate’s parentage, drag it from its current parent in the state diagram and drop it in its new parent.

Note A parent state must be graphically large enough to accommodate all its substates. You might need to resize a parent state before dragging a new substate into it. You can bypass the need for a state of large graphical size by declaring a superstate to be a subchart. See “Using Subcharts to Extend Charts” on page 5-5 for details.

Grouping States

Grouping a state causes Stateflow to treat the state and its contents as a graphical unit. This simplifies editing a state diagram. For example, moving a grouped state moves all its substates as well.

To group a state, double-click the state or its border.

Stateflow thickens the grouped state’s border and grays its contents to indicate that it is grouped.

You can also group a state by right-clicking it and then selecting **Make Contents** and then **Grouped** from the resulting shortcut menu.

You must ungroup a superstate to select objects within the superstate. To ungroup a state, double-click it or its border or select **Ungrouped** from the **Make Contents** shortcut menu.

Specifying Substate Decomposition

You specify whether a superstate contains parallel (AND) states or exclusive (OR) states by setting its decomposition. A state whose substates are all active when it is active is said to have parallel (AND) decomposition. A state in which only one substate is active when it is active is said to have exclusive (OR) decomposition. An empty state’s decomposition is exclusive.

To alter a state’s decomposition, select the state, right-click to display the state’s shortcut menu, and choose either **Parallel (AND)** or **Exclusive (OR)** from the menu.

You can also specify the state decomposition of a chart. In this case, Stateflow treats the chart's top-level states as substates of the chart. Stateflow creates states with exclusive decomposition. To specify a chart's decomposition, deselect any selected objects, right-click to display the chart's shortcut menu, and choose either **Parallel (AND)** or **Exclusive (OR)** from the menu.

The appearance of a superstate's substates indicates the superstate's decomposition. Exclusive substates have solid borders, parallel substates, dashed borders. A parallel substate also contains a number in its upper right corner. The number indicates the activation order of the substate relative to its sibling substates.

Specifying Activation Order for Parallel States

You specify the activation order of parallel states by arranging them from top to bottom and left to right in the state diagram. Stateflow activates the states in the order in which you arrange them. In particular, a top-level parallel state activates before all the states whose top edges reside at a lower level in the state diagram. A top-level parallel state also activates before any other state that resides to the right of it at the same vertical level in the diagram. The same top to bottom, left to right activation order applies to parallel substates of a state.

Note Stateflow displays the activation order of a parallel state in its upper right corner.

Changing State Properties

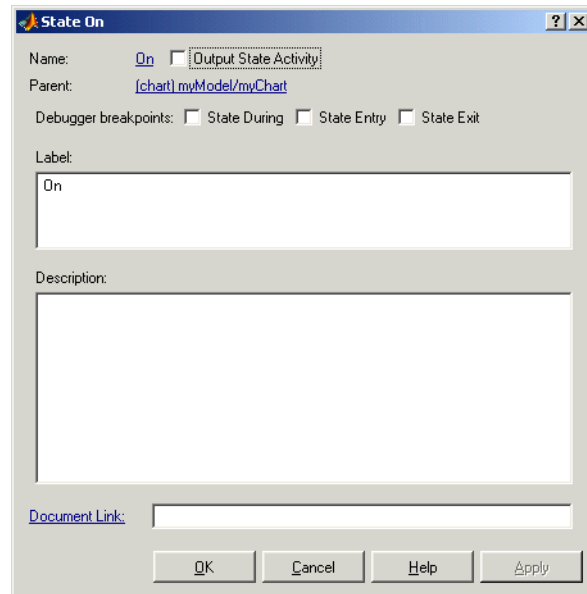
Use the **State** dialog box to view and change the properties for a state. To access the **State** dialog for a particular state, do the following:

- 1 Right-click the state.

A shortcut pop-up menu appears.

- 2 Choose **Properties** from the shortcut menu.

Stateflow displays the **State** dialog for the state as shown.



The **State** dialog contains the following properties for a state:

Field	Description
Name	Stateflow diagram name; read-only; click this hypertext link to bring the state to the foreground.
Output State Activity	Select this check box to cause Stateflow to output the activity status of this state to Simulink via a data output port on the Chart block containing the state. See “Outputting State Activity to Simulink” on page 4-14 for more information.
Parent	Parent of this state; a / character indicates the Stateflow diagram is the parent; read-only; click this hypertext link to bring the parent to the foreground.

Field	Description
Debugger breakpoints	Click the check boxes to set debugging breakpoints on the execution of state entry, during, or exit actions during simulation. See Chapter 13, “Debugging and Testing,” for more information.
Label	The label for the state. This includes the name of the state and its associated actions. See the section titled “Labeling States” on page 4-11 for detailed information.
Description	Textual description/comment.
Document Link	Enter a URL address or a general MATLAB command. Examples are <code>www.mathworks.com</code> , <code>mailto:email_address</code> , and <code>edit /spec/data/speed.txt</code> .

3 After making changes, select one of the following:

- **Apply** to save the changes and keep the **State** dialog open.
- **Revert** to return to the previous settings
- **Close** to save the changes and close the dialog box
- **Help** to display the Stateflow documentation in an HTML browser window.

Labeling States

The label for a state specifies its required name for the state and the optional actions executed when the state is entered, exited, or receives an event while it is active.

State labels have the following general format.

```

name /
entry:entry actions
during:during actions
exit:exit actions
bind:data and events
on event_name:on event_name actions

```

The italicized entries in this format have the following meanings:

Keyword	Entry	Description
NA	<i>name</i>	A unique reference to the state with optional slash
entry or en	<i>entry actions</i>	Actions executed when a particular state is entered as the result of a transition taken to that state
during or du	<i>during actions</i>	Actions that are executed when a state receives an event while it is active with no valid transition away from the state
exit or ex	<i>exit actions</i>	Actions executed when a state is exited as the result of a transition taken away from the state
bind	<i>events or data</i>	Binds the specified events or data to this state. Bound data can only be changed by this state or its children, but can be read by other states. Bound events can be broadcast only by this state or its children.
on	<i>event_name</i> and <i>on event_name actions</i>	A specified event and Actions executed when a state is active and the specified event <i>event_name</i> occurs See “Adding Events” on page 6-3 for information on defining and using events.

Entering the Name

Initially, a state’s label is empty. Stateflow indicates this by displaying a ? in the state’s label position (upper left corner). Begin the labeling the state by entering a name for the state with the following steps:

- 1 Click the state.

The state turns to its highlight color and a question mark character appears in the upper left-hand corner of the state.

- 2 Click the ? to edit the label.

An editing cursor appears. You are now free to type a label.

Enter the state's name in the first line of the state's label. Names are case sensitive. To avoid naming conflicts, do not assign the same name to sibling states. However, you can assign the same name to states that do not share the same parent.

If you are finished labeling the state, click outside of the state. Otherwise, continue entering actions. To reedit the label, simply click the label text near the character position you want to edit.

Entering Actions

After entering the name of the state in the state's label, you can enter actions for any of the following action types:

- **Entry Actions** — begin on a new line with the keyword `entry` or `en`, followed by a colon, followed by one or more action statements on one or more lines. To separate multiple actions on the same line, use a comma or a semicolon. You can begin entry actions on the same line as the state's name. In this case, begin the entry action with a forward slash (/) instead of the entry keyword.
- **Exit Actions** — begin on a new line with the keyword `exit` or `ex`, followed by a colon, followed by one or more action statements on one or more lines. To separate multiple actions on the same line, use a comma or a semicolon.
- **During Actions** — begin on a new line with the keyword `entry` or `en`, followed by a colon, followed by one or more action statements on one or more lines. To separate multiple actions on the same line, use a comma or a semicolon.
- **Bind Actions** — begin on a new line with the keyword `bind` followed by a colon, followed by one or more data or events on one or more lines. To separate multiple actions on the same line, use a comma or a semicolon.

- **On <event_name> Actions** — begin with the keyword `on`, followed by a space and the name of an event, followed by a colon, followed by one or more action statements on one or more lines, for example

```
on ev1: exit();
```

To separate multiple actions on the same line, use a comma or a semicolon. If you want different events to trigger different actions, enter multiple `event_name` blocks in the state's label, each specifying the action for a particular event or set of events, for example:

```
on ev1: action1(); on ev2: action2();
```

Note The execution of the actions you enter for a state is dependent only on their action type, and not on the order in which you enter them in the label.

You can also edit the state's label through the properties dialog for the state. See "Changing State Properties" on page 4-9.

Outputting State Activity to Simulink

Stateflow allows a chart to output the activity of its states to Simulink via a data port on the state's Chart block. To enable output of a particular state's activity, first name the state (see "Entering the Name" on page 4-12), if unnamed, then select the **Output State Activity** check box on the state's property dialog (see "Changing State Properties" on page 4-9). Stateflow creates a data output port on the Chart block containing the state. The port has the same name as the state. Stateflow also adds a corresponding data object of type `State` to the Stateflow data dictionary. During simulation of a model, the port outputs 1 at each time step in which the state is active; 0, otherwise. Attaching a scope to the port allows you to monitor a state's activity visually during the simulation. See "Sharing Input and Output Data with Simulink" on page 6-35 for more information.

Note If a chart has multiple states with the same name, only one of those states can output activity data. If you check the `Output State Activity` property for more than one state with the same name, Stateflow outputs data only from the first state whose `Output State Activity` property you specified.

Creating Transitions in Stateflow Charts

Charts change active states using pathways called transitions. This section describes how to create, move, change, and specify properties for Stateflow transitions with the following topics:

- “Creating a Transition” on page 4-16 — Tells you how to create transitions between states.
- “Creating Straight Transitions” on page 4-17 — Gives you the methods for straightening curved transitions.
- “Labeling Transitions” on page 4-17 — Tells you how to edit the default ? label for each transition and the correct format for a transition label.
- “Moving Transitions” on page 4-19 — Tells you how to move the line, the attach points, and the label of a transition.
- “Changing Transition Arrowhead Size” on page 4-20 — Gives you the procedures for changing the arrowhead size of transitions.
- “Creating Self-Loop Transitions” on page 4-21 — Shows you how to create self-loop transitions that loop back to their source.
- “Creating Default Transitions” on page 4-22 — Tells you how to create default transitions.

Creating a Transition

Use the following procedure for creating transitions between states and junctions:

- 1 Place the cursor on or close to the border of a source state or junction.

The cursor changes to crosshairs.

- 2 Click and drag a transition to a destination state or junction.
- 3 Release on the border of the destination state or junction.

Notice that the source of the transition sticks to the initial source point for the transition.

Attached transitions obey the following rules:

- Transitions do not attach to the corners of states. Corners are used exclusively for resizing.
- Transitions exit a source and enter a destination at angles perpendicular to the source or destination surface.
- Newly created transitions have smart behavior. See “Setting Smart Behavior in Transitions” on page 5-19.

To delete a transition, select it and choose **Ctrl+X** or **Cut** from the **Edit** menu, or press the **Delete** key.

See the following sections for help with creating *self-loop* and *default* transitions:

- “Creating Self-Loop Transitions” on page 4-21
- “Creating Default Transitions” on page 4-22

Creating Straight Transitions

While creating a transition, notice that the source of the transition sticks to the initial source point. This often results in a curved transition. To create a perfectly straight transition, while clicking and dragging from one state to another, do one of the following:

- Press the **S** key (works on all platforms).
- Right-click the mouse (works on *most* platforms).

Either of these actions straightens the transition perpendicular to the transition’s source state or junction surface, if possible, and allows the transition source point to slide to maintain straightness. For states, if the cursor is out of range of perpendicularity with the source state, the transition is unaffected.

Labeling Transitions

Transition labels contain Stateflow action language that accompanies the execution of a transition. Creating and editing transition labels is described in the following topics:

- “Editing Transition Labels” on page 4-18
- “Transition Label Format” on page 4-18

For more information on transition concepts, see “Transition Label Notation” on page 2-14.

For more information on transition label contents, see “Using Actions in Stateflow” on page 7-1.

Editing Transition Labels

Label unlabeled transitions as follows:

- 1 Select (left-click) the transition.

The transition turns to its highlight color and a question mark (?) appears on the transition. The ? character is the default empty label for transitions.

- 2 Left-click the ? to edit the label.

An editing cursor appears. You are now free to type a label.

To apply and exit the edit, deselect the object. To reedit the label, simply left-click the label text near the character position you want to edit.

Transition Label Format

Transition labels have the following general format:

```
event [condition]{condition_action}/transition_action
```

Specify, as appropriate, relevant names for event, condition, condition_action, and transition_action.

Label Field	Description
event	Event that causes the transition to be evaluated.
condition	Defines what, if anything, has to be true for the condition action and transition to take place.
condition_action	If the condition is true, the action specified executes and completes.
transition_action	This action executes after the source state for the transition is exited but before the destination state is entered.

Transitions do not have to have labels. You can specify some, all, or none of the parts of the label. Valid transition labels are defined by the following:

- Can have any alphanumeric and special character combination, with the exception of embedded spaces
- Cannot begin with a numeric character
- Can have any length
- Can have carriage returns in most cases
- Must have an ellipsis (...) to continue on the next line

Moving Transitions

You can move transition lines with a combination of several individual movements. These movements are described in the following topics:

- “Bowing the Transition Line” on page 4-19
- “Moving Transition Attach Points” on page 4-20
- “Moving Transition Labels” on page 4-20

In addition, transitions move along with the movements of states and junctions. See “Setting Smart Behavior in Transitions” on page 5-19 for a description of *smart* and *nonsmart* transition behavior.

Bowing the Transition Line

You can move or “bow” transition lines with the following procedure:

- 1** Place the cursor on the transition at any point along the transition except the arrow or attach points.
- 2** Click and drag the mouse to move the transition point to another location.

Only the transition line moves. The arrow and attachment points do not move.
- 3** Release the mouse button to specify the transition point location.

The result is a bowed transition line. Repeat the preceding steps to move the transition back into its original shape or into another shape.

Moving Transition Attach Points

You can move the source or end points of a transition to place them in exact locations as follows:

- 1 Place the cursor over an attach point until it changes to a small circle.
- 2 Click and drag the mouse to move the attach point to another location.
- 3 Release the mouse button to specify the new attach point.

The appearance of the transition changes from a solid to a dashed line when you detach and release a destination attach point. Once you attach the transition to a destination, the dashed line changes to a solid line.

The appearance of the transition changes to a default transition when you detach and release a source attach point. Once you attach the transition to a source, the appearance returns to normal.

Moving Transition Labels

You can move transition labels to make the Stateflow diagram more readable. To move a transition label, do the following:

- 1 Click and drag the label to a new location.
- 2 Release the left mouse button.

If you mistakenly click and then immediately release the left mouse button on the label, you will be in edit mode for the label. Press the **Esc** key to deselect the label and try again. You can also click the mouse on an empty location in the diagram editor to deselect the label.

Changing Transition Arrowhead Size

The arrowhead size is a property of the destination object. Changing one of the incoming arrowheads of an object causes all incoming arrowheads to that object to be adjusted to the same size. The arrowhead size of any selected transitions, and any other transitions ending at the same object, is adjusted.

To adjust arrowhead size from the **Transition** shortcut menu:

- 1 Select the transitions whose arrowhead size you want to change.

- 2 Place the cursor over a selected transition and right-click to display the shortcut menu.

A menu of arrowhead sizes appears.

- 3 Select an arrowhead size from the menu.

To adjust arrowhead size from the **Junction** shortcut menu:

- 1 Select the junctions whose incoming arrowhead size you want to change.
- 2 Place the cursor over a selected junction and right-click.
- 3 In the resulting submenu, place the cursor over **Arrowhead Size**.

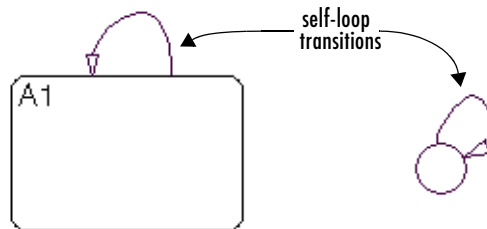
A menu of arrowhead sizes appears.

- 4 Select a size from the menu.

Creating Self-Loop Transitions

A self-loop transition is a transition whose source and destination are the same state or junction.

The following is an example of a self-loop transition:




To create a self-loop transition, do the following:

- 1 Create the transition as usual by clicking and dragging it out from the source state or junction.
- 2 Continue dragging the transition tip back to a location on the source state or junction.

For the semantics of self-loops, see “Self-Loop Transitions” on page 2-21.

Creating Default Transitions

A default transition is a transition with a destination (a state or a junction), but no apparent source object. See “Default Transitions” on page 1-13 for an explanation of default transitions.

Click the **Default Transition** button  in the toolbar and click a location in the drawing area close to the state or junction you want to be the destination for the default transition. Drag the mouse to the destination object to attach the default transition.

The size of the endpoint of the default transition is proportional to the arrowhead size. See “Changing Transition Arrowhead Size” on page 4-20.

Default transitions can be labeled just like other transitions. See “Labeling Default Transitions” on page 2-26 for an example.

Changing Transition Properties

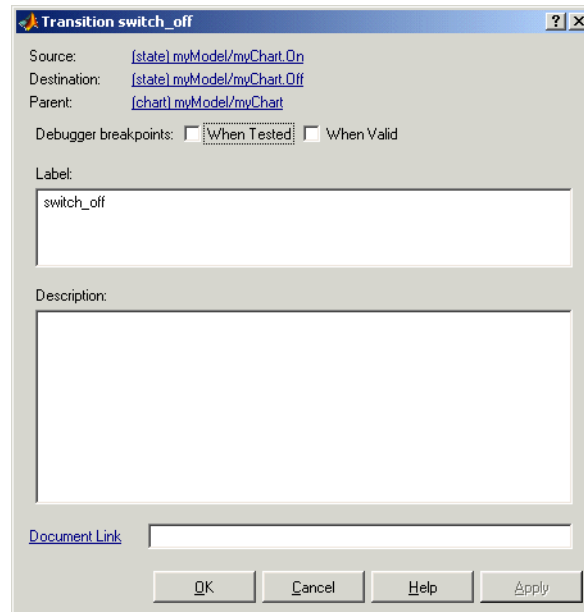
Use the **Transition** dialog box to view and change the properties for a transition. To access the **Transitions** dialog for a particular transition, do the following:

- 1 Right-click on the transition.

A shortcut pop-up menu appears.

- 2 Choose **Properties** from the shortcut menu.

Stateflow displays the **Transition** dialog for the transitions as shown.



The following table lists and describes the properties displayed for a transition in its **Transition** dialog:

Field	Description
Source	Source of the transition; read-only; click the hypertext link to bring the transition source to the foreground.
Destination	Destination of the transition; read-only; click the hypertext link to bring the transition destination to the foreground.
Parent	Parent of this state; read-only; click the hypertext link to bring the parent to the foreground.

Field	Description
Debugger breakpoints	Select the check boxes to set debugging breakpoints either when the transition is tested for validity or when it is valid.
Label	The transition's label. See "Transition Label Notation" on page 2-14 for more information on valid label formats.
Description	Textual description/comment.
Document Link	Enter a Web URL address or a general MATLAB command. Examples are <code>www.mathworks.com</code> , <code>mailto:email_address</code> , and <code>edit/spec/data/speed.txt</code> .

- 3** After making changes, select one of the following:
- **Apply** to save the changes and keep the **Transition** dialog open.
 - **Revert** to return to the previous settings for the dialog
 - **Close** to save the changes and close the dialog box
 - **Help** to display Stateflow online help in an HTML browser window.

Creating Flowcharts with Connective Junctions


Connective junctions provide a decision point between alternate transition paths. You use them to handle situations where transitions out of one state into one of two or more states are taken based on the same event but guarded by different conditions.

The following sections describe how to create, move, and specify properties for connective junctions:

- “Creating a Connective Junction” on page 4-25 — Shows you how to create a junction in the Stateflow diagram editor
- “Changing Connective Junction Size” on page 4-26 — Tells you how to change the diameter of a junction in the Stateflow diagram editor
- “Changing Junction Properties” on page 4-26 — Tells you how to access and change the properties of a junction

Creating a Connective Junction

To create a junction, do the following:

- 1 In the diagram toolbar, click the **Connective Junction** tool  .
- 2 Move the cursor into the diagram editor.

The cursor takes on the shape of a connective junction.

- 3 Click to place a connective junction in the desired location in the drawing area.

To create multiple connective junctions, do the following:

- 1 In the diagram toolbar, double-click the **Connective Junction** tool.
- 2 The button is now in multiple object mode.
- 3 Click anywhere in the drawing area to place a connective junction in the drawing area.
- 4 Move to and click another location to create an additional connective junction.

- 5 Click the **Connective Junction** tool or press the **Esc** key to cancel the operation.

To move a connective junction to a new location, click and drag it to the new position.

Changing Connective Junction Size

To change the size of connective junctions, do the following:

- 1 Select the connective junctions whose size you want to change.
- 2 Place the cursor over one of the connective junctions and right-click.
- 3 In the resulting submenu, place the cursor over **Junction Size**.

A menu of junction sizes appears.

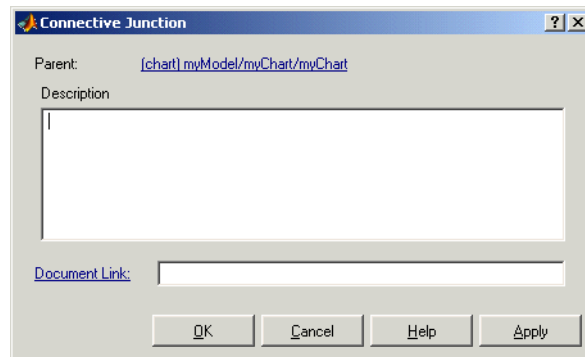
- 4 Select a size from the menu of junction sizes.

Changing Junction Properties

To edit the properties for a connective junction, do the following:

- 1 Right-click a connective junction.
- 2 In the resulting submenu select **Properties**.

The **Connective Junction** dialog box appears as shown.



- 3 Edit the fields in the properties dialog, which are described in the following table:

Field	Description
Parent	Parent of this state; read-only; click the hypertext link to bring the parent to the foreground.
Description	Textual description/comment.
Document Link	Enter a URL address or a general MATLAB command. Examples are <code>www.mathworks.com</code> , <code>mailto:email_address</code> , and <code>edit/spec/data/speed.txt</code> .

- 4 When finished editing, select one of the following:
- Select the **Apply** button to save the changes.
 - Select the **Cancel** button to cancel any changes you've made.
 - Select **OK** to save the changes and close the dialog box.
 - Select the **Help** button to display the Stateflow online help in an HTML browser window.

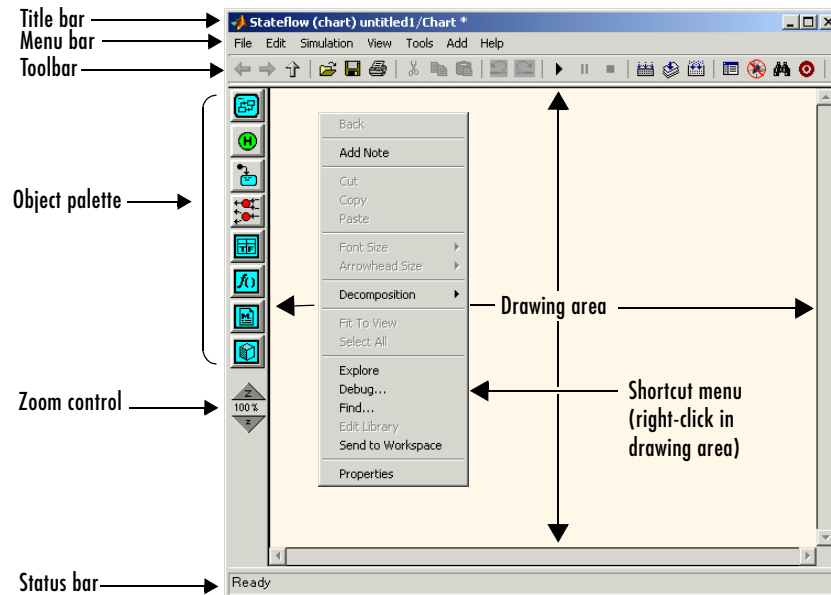
Using the Stateflow Editor

You edit your Stateflow chart diagrams in the Stateflow Editor. This section describes each part of the Stateflow diagram editor window displaying the chart you created. It contains the following topics:

- “Stateflow Diagram Editor Window” on page 4-29 — Describes the parts of the window in which you edit Stateflow diagrams.
- “Displaying the Context Menu for Objects” on page 4-30 — Shows you how to use the right-click context menu for doing operations on that object.
- “Specifying Colors and Fonts” on page 4-31 — Shows you how to change the colors and fonts for all the graphical objects in a Stateflow diagram.
- “Selecting and Deselecting Objects” on page 4-34 — Shows you how to select or deselect objects in the diagram editor to edit them.
- “Cutting and Pasting Objects” on page 4-35 — Tells you how to cut and paste objects in a Stateflow diagram from one location to another.
- “Copying Objects” on page 4-35 — Shows you how to copy objects in the diagram editor.
- “Editing Object Labels” on page 4-36 — Teaches you how to edit the labels of state and transition objects in the diagram editor.
- “Viewing Stateflow Objects in the Model Explorer” on page 4-36 — Use the Stateflow Explorer to add, change, or view data and events for a Stateflow diagram.
- “Zooming a Diagram” on page 4-37 — Shows you how to zoom in and navigate to parts of your Stateflow diagram.
- “Undoing and Redoing Editor Operations” on page 4-39 — Shows you how to undo and redo operations you perform in the Stateflow diagram editor.
- “Keyboard Shortcuts for Stateflow Diagrams” on page 4-40 — Provides a reference list of all keyboard shortcuts available in the Stateflow diagram editor.

Stateflow Diagram Editor Window

You use the Stateflow diagram editor to draw, zoom, modify, print, and save a state diagram displayed in the window. It has the following appearance:



The Stateflow diagram editor window includes the following elements:

- Title bar

The full chart name appears here in *model name / chart name** format. The * character appears on the end of the chart name for a newly created chart or for an existing chart that has been edited but not saved yet.

- Menu bar

Most editor commands are available from the menu bar.

- Toolbar

Contains buttons for cut, copy, paste, and other commonly used editor commands. You can identify each tool of the toolbar by placing the mouse cursor over it until an identifying tool tip appears.

The toolbar also contains buttons for navigating a chart's subchart hierarchy (see "Navigating Subcharts" on page 5-10).

- **Object palette**
Displays a set of tools for drawing states, transitions, and other state chart objects.
- **Drawing area**
Displays an editable copy of a state diagram.
- **Zoom control**
See “Viewing Stateflow Objects in the Model Explorer” on page 4-36 for information on using the zoom control.
- **Shortcut menus**
These menus pop up from the drawing area when you right-click an object. They display commands that apply only to that object. If you right-click an empty area of the diagram editor, the shortcut menu applies to the chart object. See “Displaying the Context Menu for Objects” on page 4-30 for more information.
- **Status bar**
Displays tool tips and status information.

Displaying the Context Menu for Objects

Every object that you create in a state diagram has a shortcut menu associated with it. To display the shortcut (context) menu, do the following:

- 1 Move the cursor over the object.
- 2 Right-click the object.

Stateflow pops up a menu of operations that apply to the object.

To display the context menu for the chart object, do the following:

- 1 Move the cursor to an unoccupied location in the diagram.
- 2 Right-click the location.

Stateflow pops up a menu of operations that apply to the chart.

Specifying Colors and Fonts

You can specify the color and font for items in the diagram editor, as described in the following topics:

- “Changing Fonts for an Individual Text Item” on page 4-31 — Tells you how to set color and font for an individual item in the Stateflow diagram editor.
- “Using the Colors & Fonts Dialog” on page 4-31 — Shows you how to set default colors and fonts for all Stateflow diagram editor items in the Colors and Fonts dialog

Changing Fonts for an Individual Text Item

You can change the font for an individual text item as follows:

- 1 Right-click the individual item.
- 2 From the resulting submenu, select **Font Size** -> *size of font*.

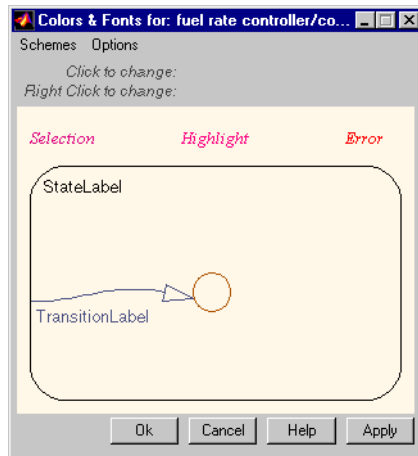
You can also specify the label font size of a particular object:

- 1 Left-click an individual text item in the editor.
- 2 From the editor’s **Edit** menu, select **Set Font Size**.
- 3 From the resulting submenu, select the font size.

Using the Colors & Fonts Dialog

The Stateflow **Colors & Fonts** dialog allows you to specify a color scheme for a chart as a whole, or colors and label fonts for different types of objects in a chart.

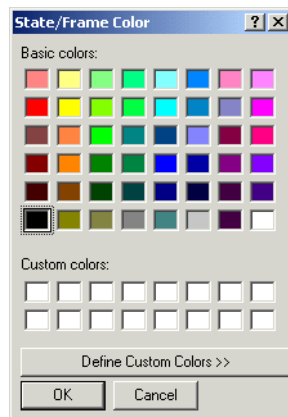
To display the **Colors & Fonts** dialog, select **Style** from the Stateflow editor’s **Edit** menu.



The drawing area of the dialog displays examples of the types of objects whose colors and font labels you can specify. The examples use the colors and label fonts specified by the current color scheme for the chart. To choose another color scheme, select the scheme from the dialog's **Schemes** menu. The dialog displays the selected color scheme. Click **Apply** to apply the selected scheme to the chart or **OK** to apply the scheme and dismiss the dialog.

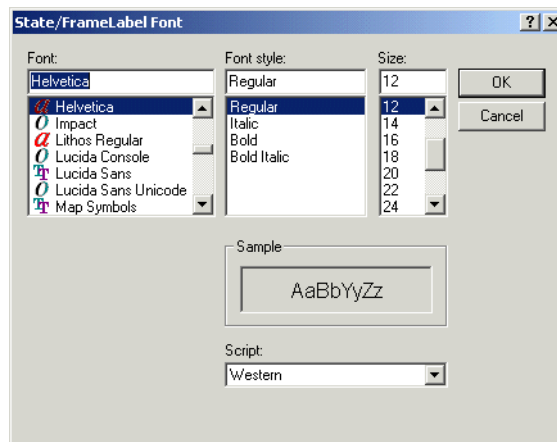
To make the selected scheme the default scheme for all Stateflow charts, select **Make this the "Default" scheme** from the dialog's **Options** menu.

To modify the current scheme, position the cursor over the example of the type of object whose color or label font you want to change. Then left-click to change the object's color or right-click to change the object's font. If you left-click, Stateflow displays a color chooser dialog.



Use the dialog to select a new color for the selected object type.

If the selected object is a label and you right-click, Stateflow displays a font selection dialog.



Use the font selector to choose a new font for the selected label.

To save changes to the default color scheme, select **Save defaults to disk** from the **Colors & Fonts** dialog's **Options** menu.

Note Choosing **Save defaults to disk** has no effect if the modified scheme is not the default scheme.

Selecting and Deselecting Objects

Once an object is in the drawing area, you need to select it to make any changes or additions to that object.

Select objects in the Stateflow diagram editor as follows:

- To select an object, click anywhere inside of the object.
- To select multiple adjacent objects, click and drag a selection rubberband so that the rubberband box encompasses or touches the objects you want to select, and then release the mouse button.

All objects or portions of objects within the rubberband are selected.

- To select multiple separate objects, simultaneously press the **Shift** key and click an object or rubberband a group of objects.

This adds objects to the list of already selected objects unless an object was already selected, in which case, the object is deselected. This type of multiple object selection is useful for selecting objects within a state without selecting the state itself when you rubberband select a state and all of its objects and then Shift-click inside the containing state to deselect it.

- To select all objects in the Stateflow diagram, from the **Edit** menu select **Select All**.

You can also select all objects by selecting **Select All** from the right-click shortcut menu.

- To deselect all selected objects, press the **Esc** key.

Pressing the **Esc** key again displays the parent of the current chart.

When an object is selected, it is highlighted in the color set as the selection color (blue by default; see “Specifying Colors and Fonts” on page 4-31 for more information).

Cutting and Pasting Objects

You can cut objects from the drawing area or cut and then paste them as many times as you like. You can cut and paste objects from one Stateflow diagram to another. Stateflow retains a selection list of the most recently cut objects. The objects are pasted in the drawing area location closest to the current mouse location.

To cut an object, select the object and choose **Cut** from one of the following:

- The **Edit** menu on the main window
- The right-click shortcut menu

Pressing the **Ctrl** and **X** keys simultaneously is the keyboard equivalent to the **Cut** menu item.

To paste the most recently cut selection of objects, choose **Paste** from either of the following:

- The **Edit** menu on the main window
- The right-click shortcut menu

Pressing the **Ctrl** and **V** keys simultaneously is the keyboard equivalent to the **Paste** menu item.

Copying Objects

To copy and paste an object in the drawing area, select the objects and right-click and drag them to the desired location in the drawing area. This operation also updates the Stateflow clipboard.

Alternatively, to copy from one Stateflow diagram to another, choose the **Copy** and then **Paste** menu items from either of the following:

- The **Edit** menu on the Stateflow graphics editor window
- The right-click shortcut menu

Pressing the **Ctrl** and **C** keys simultaneously is the keyboard equivalent to the **Copy** menu item. States that contain other states (superstates) can be grouped together.

Editing Object Labels

Some Stateflow objects (for example, states and transitions) have labels. To change these labels, place the cursor anywhere in the label and click. The cursor changes to an I-beam. You can then edit the text.

The shortcut (context) menus allows you to change a label's font size:

- 1 Select the states whose label font size you want to change.
- 2 Right-click to display the shortcut menu.
- 3 Place the cursor over the **Font Size** menu item.

A menu of font sizes appears.

- 4 Select the desired font size from the menu.

Stateflow changes the font size of all labels on all selected states to the selected size.

Viewing Stateflow Objects in the Model Explorer

To view or modify Stateflow diagram editor objects in the **Model Explorer**, do the following:

- 1 Position the mouse cursor over the state.
- 2 Right-click to display the state's context menu.
- 3 Select **Explore** from the context menu.

The **Model Explorer** opens (if not already open) and highlights the state in the left hierarchy pane to show any events or data defined by the state.

To view events and data defined by the parent state of a transition or junction, select **Explore** from the transition or junction's context menu.

The **Model Explorer** is the only place where you can view and modify events and data. See “Defining Events and Data” on page 6-1 for more details on using the **Model Explorer** to view, add, delete, and modify data and events for Stateflow objects. See also “Using the Model Explorer with Stateflow Objects” on page 14-2 for more details on using the **Model Explorer** to view Stateflow objects.

Zooming a Diagram

You can magnify or shrink a diagram, using the following zoom controls:

- **Zoom Factor Selector.** Selects a zoom factor (see “Using the Zoom Factor Selector” on page 4-37).
- **Zoom In** button. Zooms in by the current zoom factor.
You can also press the **R** key to increase the zoom factor.
- **Zoom Out** button. Zooms out by the current zoom factor.
You can also press the **V** key to decrease the zoom factor.

Using the Zoom Factor Selector

The **Zoom Factor Selector** allows you to specify the zoom factor by

- Choosing a value from a menu.
Click the selector to display the menu.
- Double-clicking the **Zoom Factor Selector** selects the zoom factor that will fit the view to all selected objects or all objects if none are selected.
You can achieve the same effect by choosing **Fit to View** from the right-click context menu or by pressing the **F** key to apply the maximum zoom that includes all selected objects. Press the space bar to fit all objects to the view.
- Clicking the **Zoom Factor Selector** and dragging up or down.
Dragging the mouse upward increases the zoom factor. Dragging the mouse downward decreases the zoom factor. Alternatively, right-clicking and dragging on the percentage value resizes while you are dragging.

Zooming with Shortcut Keys

The following is a summary of the shortcut keys you can use to perform some of the zooming operations described above:

Key	Zoom Operation
F	Highlight (select) an object and press the F key to fit it to view.
space bar	Set to full view of diagram.
R or +	Increase zoom factor.
V or -	Decrease zoom factor.

Moving in Zoomed Diagrams with Shortcut Keys

You can also use number keys to move in zoomed diagrams according to their layout in the number keypad:


7 ↖	8 ↑	9 ↗
4 ←	5 fit to view	6 →
1 ↙	2 ↓	3 ↘

You can enter numbers for moving from the number keys above the alphabetic keys at any time or from the number keypad if NumLock is engaged for the keyboard. The **5** key fits the currently selected object to full view. If no object is selected, the entire diagram is fit to view.


Undoing and Redoing Editor Operations

You can undo and redo operations you perform in the Stateflow diagram editor. When you undo an operation in the Stateflow diagram editor, you reverse the last edit operation you performed. After you undo operations in the diagram editor, you can also redo them one at a time.

To undo an operation in the Stateflow diagram editor, do one of the following:

- Select the **Undo** icon in the toolbar of the Stateflow diagram editor . When you place your mouse cursor over the **Undo** button, the tool tip that appears indicates the nature of the operation to undo.
- From the **Edit** menu, select **Undo**.

To redo an operation in the Stateflow diagram editor, do one of the following:

- Select the **Redo** icon in the toolbar of the Stateflow diagram editor . When you place your mouse cursor over the **Redo** button, the tool tip that appears indicates the nature of the operation to redo.
- From the **Edit** menu, select **Redo**.

Exceptions for Undo

You can undo or redo all diagram editor operations, with the following exceptions:

- You cannot undo the operation of turning subcharting off for a state previously subcharted. To understand subcharting, see “Using Subcharts to Extend Charts” on page 5-5.
- You cannot undo the drawing of a supertransition or the splitting of an existing transition. Splitting of an existing transition refers to the redirection of the source or destination of a transition segment that is part of a supertransition. For a description of supertransitions, see “Drawing a Supertransition Into a Subchart” on page 5-13 and “Drawing a Supertransition Out of a Subchart” on page 5-16.

- You cannot undo any changes made to the diagram editor through the Stateflow API.

For a description of the Stateflow API (Application Programming Interface), see “Using the Stateflow API” on page 20-1.

Caution When you perform one of the preceding operations, the undo and redo buttons are disabled from undoing and redoing any prior operations.

Keyboard Shortcuts for Stateflow Diagrams

The following table is a comprehensive list of keyboard shortcuts for the Stateflow diagram editor:

Key	Action
.. (two periods)	Displays the parent of the currently displayed chart or subchart. There is no limit on the time between the entry of each period.
+	Zooms diagram by an incremental amount. Same as R key.
-	Unzooms diagram by an incremental amount. Same as V key.
0	Fit to view for entire diagram. Same as Space Bar key.
1	Moves the current diagram editor view down and to the left within the full diagram.
2	Moves the current diagram editor view down within the full diagram.
3	Moves the current diagram editor view down and to the right within the full diagram.
4	Moves the current diagram editor view left within the full diagram.

Key	Action
5	Fits the currently selected object to full view. If no object is selected, the chart is fit to full view.
6	Moves the current diagram editor view to the right within the full diagram.
7	Moves the current diagram editor view up and to the left within the full diagram.
8	Moves the current diagram editor view up within the full diagram.
9	Moves the current diagram editor view up and to the right within the full diagram.
Delete	Deletes the selected objects.
Enter	Accesses the contents of the currently highlighted subchart or truth table.
Esc	<p>Does any of the following:</p> <ul style="list-style-type: none"> • If you are editing the label of an object, the Esc key disables label editing but leaves the object selected. • If objects are selected, the Esc key deselects all objects in the current view. • If the current diagram view is the contents of a subchart and no object is selected, the Esc key changes the view to the parent of the subchart. • If the current diagram view is at the chart level and no object is selected, the Esc key displays the Simulink window for that chart's block.
Space Bar	Fit to view for entire chart. Same as 0 key.
F	Fits the currently selected object to full view. If no object is selected, the chart is fit to full view.

Key	Action
J (jump)	Selects the first state, function, truth table, or box parented (contained) by the currently selected object in the same diagram. Selection order of contained objects is top-down, left-right. See also U key.
N (next)	Selects the next state, function, truth table, or box at the same containment level. Selection order of objects is top-down, left-right.
P (previous)	Selects the previous state, function, truth table, or box at the same containment level. Selection order of objects is top-down, left-right.
R	Zooms diagram by an incremental amount. Same as + key.
U (up)	Selects the parent object of the currently highlighted object in the same diagram. See also J key.
V	Unzooms diagram by an incremental amount. Same as - key.

Extending Stateflow Chart Diagrams

This chapter takes you through the steps of extending Stateflow states, transitions, and junctions with subcharts, boxes, functions, and notes. It includes the following sections:

- | | |
|--|--|
| Using History Junctions to Extend Charts and States (p. 5-2) | Describes how to create, move, and specify properties for Stateflow history junctions. History junctions extend the abilities of charts and states by recording their most recently active substate. |
| Using Subcharts to Extend Charts (p. 5-5) | Shows you how to create and work with charts within charts, that is, subcharts. Subcharts are a convenience for compacting your diagrams. |
| Using Supertransitions to Extend Transitions (p. 5-12) | Shows you how to make a supertransition to connect transitions from outside a subchart to a state or junction inside a subchart. |
| Extending Transitions with Smart Behavior (p. 5-19) | Shows you how smart transitions maintain their shapes and uniqueness while you rearrange chart objects. |
| Using Functions to Extend Actions (p. 5-29) | Describes how Stateflow graphical functions are created, called, and made available to Simulink. Graphical functions are a convenience, as functions are to programs. |
| Using Boxes to Extend Chart Diagrams (p. 5-45) | Describes Stateflow boxes and how to create them in your new chart. Boxes are a convenience for grouping items in your charts. |
| Using Notes to Extend Chart Diagrams (p. 5-48) | Shows you how to create, edit, and delete descriptive notes for your Stateflow chart. |
| Reporting Chart Diagrams (p. 5-51) | Shows you the options Stateflow offers for printing part or all of your Stateflow chart. |

Using History Junctions to Extend Charts and States


History junctions extend the ability of charts and states by recording the activity of substates inside superstates. Use a history junction in a chart or superstate to indicate that its last active substate becomes active when the chart or superstate becomes active.

The following sections describe how to create, and specify properties for history junctions in Stateflow:

- “Creating a History Junction” on page 5-2 — Shows you how to create a junction in the Stateflow diagram editor
- “Changing History Junction Size” on page 5-3 — Tells you how to change the diameter of a junction in the Stateflow diagram editor
- “Changing History Junction Properties” on page 5-3 — Tells you how to access and change the properties of a junction

Creating a History Junction

To create a junction, do the following:

- 1 In the diagram toolbar, click the **History Junction** tool  .
- 2 Move the cursor into the diagram editor.

The cursor takes on the shape of a junction.
- 3 Click to place a history junction inside the state whose last active substate it records.

To create multiple history junctions, do the following:

- 1 In the diagram toolbar, double-click the **History Junction** tool.
- 2 The button is now in multiple object mode.
- 3 Click anywhere in the drawing area to place a history junction.
- 4 Move to and click another location to create an additional history junction.
- 5 Click the **History Junction** tool or press the **Esc** key to cancel the operation.

To move a history junction to a new location, click and drag it to the new position.

Changing History Junction Size

To change the size of junctions, do the following:

- 1 Select the history junctions whose size you want to change.
- 2 Place the cursor over one of the junctions and right-click.
- 3 In the resulting submenu, place the cursor over **Junction Size**.

A menu of junction sizes appears.

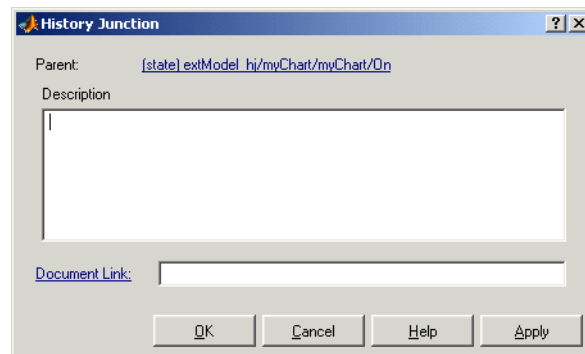
- 4 Select a size from the menu of junction sizes.

Changing History Junction Properties

To edit the properties for a junction, do the following:

- 1 Right-click a junction.
- 2 In the resulting submenu select **Properties**.

The **History Junction** dialog appears as shown.



- 3 Edit the fields in the properties dialog, which are described in the following table:

Field	Description
Parent	Parent of this history junction; read-only; click the hypertext link to bring the parent to the foreground.
Description	Textual description/comment.
Document Link	Enter a URL address or a general MATLAB command. Examples are <code>www.mathworks.com</code> , <code>mailto:email_address</code> , and <code>edit/spec/data/speed.txt</code> .

- 4 When finished editing, select one of the following:
- Select the **Apply** button to save the changes.
 - Select the **Cancel** button to cancel any changes you've made.
 - Select **OK** to save the changes and close the dialog box.
 - Select the **Help** button to display the Stateflow online help in an HTML browser window.

Using Subcharts to Extend Charts

Subcharts are charts within charts. They provide a convenience for compacting your diagrams. This section shows you how to create and work with subcharts in the following topics:

- “What Is a Subchart?” on page 5-5 — Describes a subchart and its hierarchical position in a chart.
- “Creating a Subchart” on page 5-7 — Shows you how to create a subchart by converting a state, box, or graphical function into a subchart.
- “Manipulating Subcharts as Objects” on page 5-9 — Tells you how to move, copy, cut, paste, relabel, and resize subcharts.
- “Opening a Subchart” on page 5-9 — Tells you how to open a subchart to view and change its contents.
- “Editing a Subchart” on page 5-10 — Describes the way in which you edit the contents of subcharts.
- “Navigating Subcharts” on page 5-10 — Describes a set of buttons for navigating a chart’s subchart hierarchy.

What Is a Subchart?

Stateflow allows you to create charts within charts. A chart that is embedded in another chart is called a *subchart*. The subchart can contain anything a top-level chart can, including other subcharts. In fact, you can nest subcharts to any level.

A subcharted state is a superstate of the states and charts that it contains. It appears as a block with its name in the block center. However, you can define actions and default transitions for subcharts just as you can for superstates. You can also create transitions to and from subcharts just as you can create transitions to and from superstates. Further, you can create transitions between states residing outside a subchart and any state within a subchart. The term *supertransition* refers to a transition that crosses subchart boundaries in this way. See “Using Supertransitions to Extend Transitions” on page 5-12 for more information.

Subcharts enable you to reduce a complex chart to a set of simpler, hierarchically organized diagrams. This makes the chart easier to understand and maintain. Nor do you have to worry about changing the semantics of the chart in any way. Stateflow ignores subchart boundaries when simulating and generating code from Stateflow models.

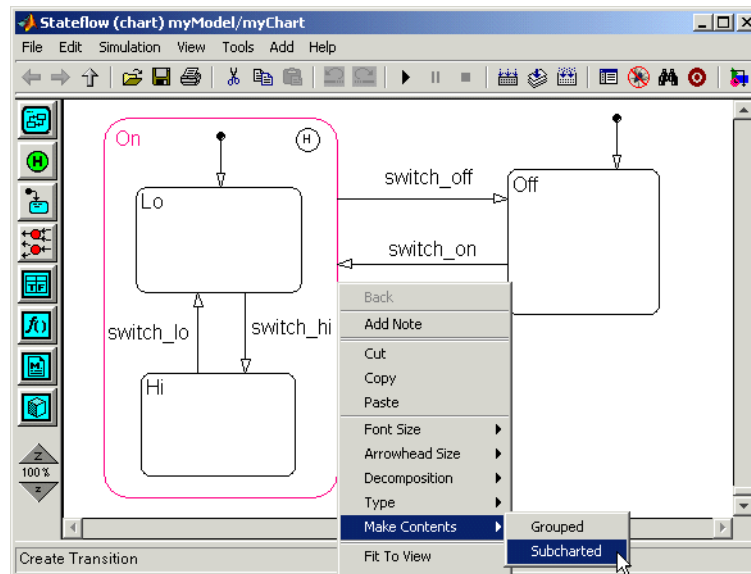
Subcharts define a containment hierarchy within a top-level chart. A subchart or top-level chart is said to be the *parent* of the charts it contains at the first level and an *ancestor* of all the subcharts contained by its children and their descendants at lower levels.

Creating a Subchart

You create a subchart by converting an existing state, box, or graphical function into the subchart. The object to be converted can be one that you have created expressly for the purpose of making a subchart or it can be an existing object whose contents you want to turn into a subchart.

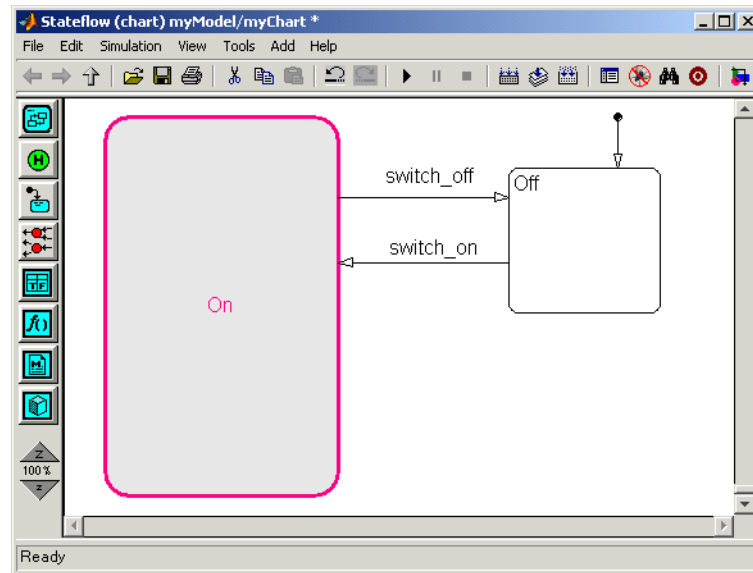
To convert a new or existing state, box, or graphical function to a subchart:

- 1 Select the object and right-click a state (SC1 in the example below) to display the Stateflow shortcut menu for that state.



- 2 Select **Make Contents** from the resulting menu.
- 3 Select **Subchartd** from the resulting submenu.

Stateflow converts the state (or a graphical function or box) to a subchart.



Note When you convert a box to a subchart, the subchart retains the attributes of a box. In particular, the resulting subchart's position in the chart determines its activation order (see “Using Boxes to Extend Chart Diagrams” on page 5-45 for more information).

To convert the subchart back to its original form, right-click the subchart. In the pop-up menu that results, select **Make Contents**. In the resulting submenu select the **Subcharted** item.

Caution You cannot undo the operation of converting a subchart back to its original form. When you perform this operation, the undo and redo buttons are disabled from undoing and redoing any prior operations.

Manipulating Subcharts as Objects

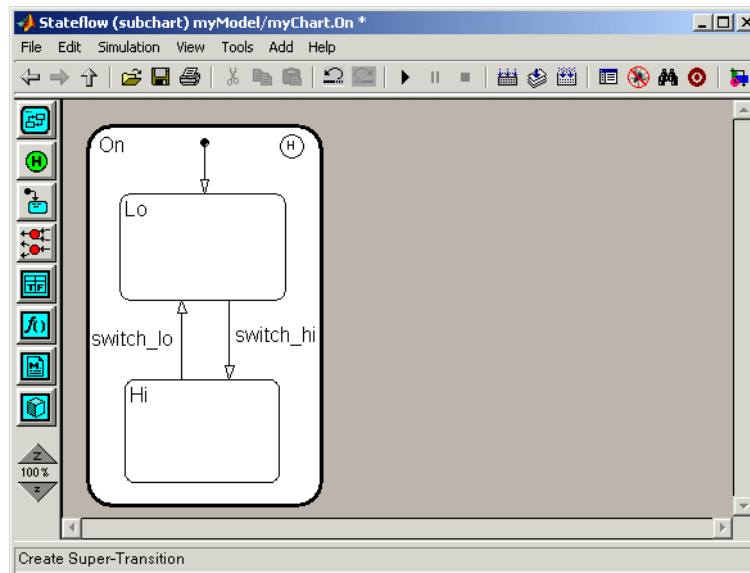
Subcharts also act as individual objects in Stateflow. You can move, copy, cut, paste, relabel, and resize subcharts as you would states and boxes. You can also draw transitions to and from a subchart and any other state or subchart at the same or different levels in the chart hierarchy (see “Using Supertransitions to Extend Transitions” on page 5-12).

Opening a Subchart

Opening a subchart allows you to view and change its contents. To open a subchart, do one of the following:

- Double-click anywhere in the box that represents the subchart.
- Select the box representing the subchart and press the **Enter** key.

Stateflow replaces the current diagram editor display with the contents of the subchart, as shown.



A shaded border surrounds the contents of the subchart. Stateflow uses the border to display supertransitions.

To return to the previous view, select **Back** from the Stateflow shortcut menu, press the **Esc** key on your keyboard, or select the up or back arrow on the Stateflow toolbar.

Editing a Subchart




After you open a subchart (see “Opening a Subchart” on page 5-9), you can perform any editing operation on its contents that you can perform on a top-level chart. This means that you can create, copy, paste, cut, relabel, and resize the states, transitions, and subcharts in a subchart. You can also group states, boxes, and graphical functions inside subcharts.

You can also cut and paste objects between different levels in your chart. For example, to copy objects from a top-level chart to one of its subcharts, first open the top-level chart and copy the objects. Then open the subchart and paste the objects into the subchart.

Transitions from outside subcharts to states or junctions inside subcharts are called *supertransitions*. You create supertransitions differently than you do ordinary transitions. See “Using Supertransitions to Extend Transitions” on page 5-12 for information on creating supertransitions.

Navigating Subcharts

The Stateflow toolbar contains a set of buttons for navigating a chart’s subchart hierarchy.

Tool	Description
	If the Stateflow editor is displaying a subchart, replaces the subchart with the subchart’s parent in the editor. If the editor is displaying a top-level chart, this button raises the Simulink model window containing that chart.
	Returns to the chart that you visited before the current chart. Lets you navigate up the hierarchy.
	Returns to the chart that you visited after visiting the current chart. Lets you navigate down the hierarchy.

Note You can also use the key sequence `..` (that is, press the period key twice) to navigate up to the parent object for a subcharted state, box, or function.

Using Supertransitions to Extend Transitions

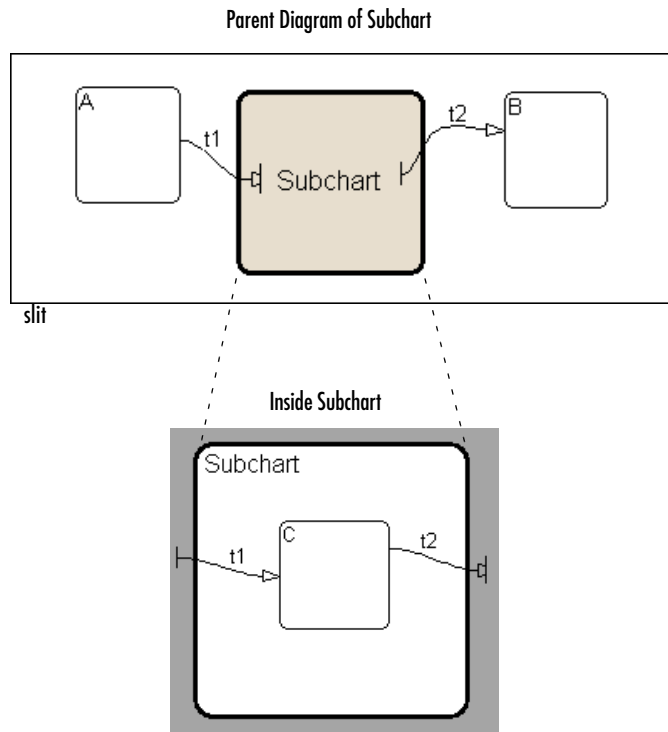
To connect transitions from outside a subchart to an object inside a subchart, you'll need to know how to make a supertransition. Learn how to make supertransitions in the following topics:

- “What Is a Supertransition?” on page 5-12 — Describes supertransitions and gives you an example.
- “Drawing a Supertransition Into a Subchart” on page 5-13 — Gives you a procedure for drawing a supertransition into a subchart.
- “Drawing a Supertransition Out of a Subchart” on page 5-16 — Gives you a procedure for drawing a supertransition out of a subchart.
- “Labeling Supertransitions” on page 5-18 — Tells you how to label a supertransition composed of multiple transition segments.

What Is a Supertransition?

A *supertransition* is a transition between different levels in a chart, for example, between a state in a top-level chart and a state in one of its subcharts, or between states residing in different subcharts at the same or different levels in a diagram. Stateflow allows you to create supertransitions that span any number of levels in your chart, for example, from a state at the top level to a state that resides in a subchart several layers deep in the chart.

The point where a supertransition enters or exits a subchart is called a *slit*. Slits divide a supertransition into graphical segments. For example, the following diagram shows two supertransitions as seen from the perspective of a subchart and its parent chart, respectively.



In this example, supertransition t1 goes from state A in the parent chart to state C in the subchart and supertransition t2 goes from state C in the subchart to state B in the parent chart. Note that both segments of t1 and t2 have the same label.

Drawing a Supertransition Into a Subchart

Use the following steps to draw a supertransition from an object outside a subchart to an object inside the subchart.

Caution You cannot undo the operation of drawing a supertransition. When you perform this operation, the undo and redo buttons are disabled from undoing and redoing any prior operations.

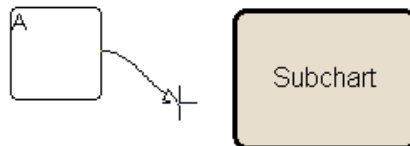
- 1 Position the mouse cursor over the border of the state.

The cursor assumes the crosshairs shape.



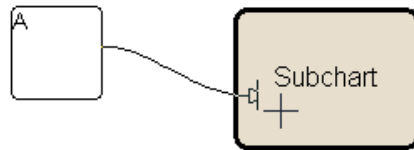
- 2 Drag the mouse.

Dragging the mouse causes a supertransition segment to appear. The segment looks like a regular transition. It is curved and is tipped by an arrowhead.



- 3 Drag the segment's tip anywhere just inside the border of the subchart.

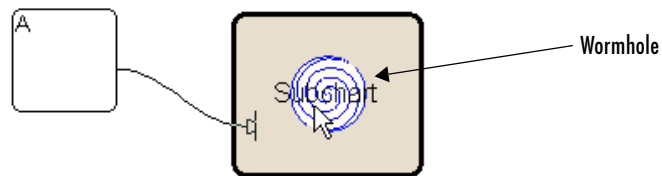
The arrowhead now penetrates the slit.



If you are not happy with the initial position of the slit, you can continue to drag the slit around the inside edge of the subchart to the desired location.

- 4 Continue dragging the cursor toward the center of the subchart.

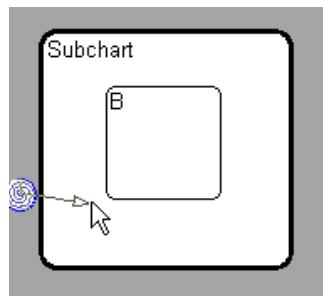
A wormhole appears in the center of the subchart.



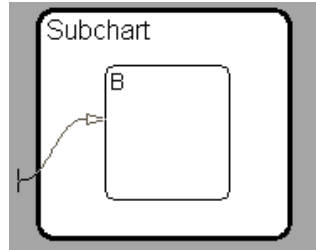
A *wormhole* allows you to open a subchart while drawing a supertransition.

- 5 Drag the mouse pointer over the center of the wormhole.

The subchart opens. Now the wormhole and supertransition are visible inside the subchart.



- 6 Drag and drop the tip of the supertransition anywhere on the border of the object that you want to terminate the transition.



Note If the terminating object resides within a subchart in the current subchart, continue to drag the tip of the supertransition through the wormhole of the inner subchart and complete the connection inside the inner chart. In this way, you can draw a supertransition to an object at any subchart depth in the chart.

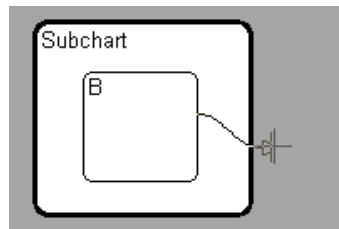
Drawing a Supertransition Out of a Subchart

Use the following steps to draw a supertransition out of a subchart.

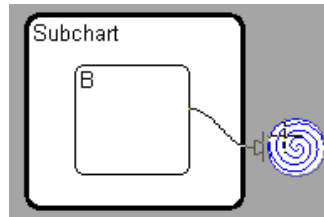
Caution You cannot undo the operation of drawing a supertransition. When you perform this operation, the undo and redo buttons are disabled from undoing and redoing any prior operations.

- 1 Draw an inner transition segment from the source object anywhere just outside the border of the subchart

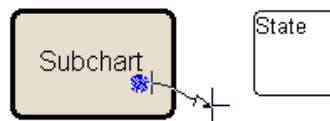
A slit appears as shown.



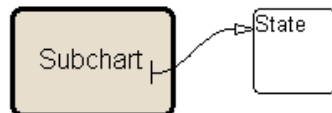
- 2** Keep dragging the transition away from the border of the subchart.
A wormhole appears.



- 3** Drag the transition down the wormhole.
The parent of the subchart appears.



- 4** Complete the connection.



Note If the parent chart is itself a subchart and the terminating object resides at a higher level in the subchart hierarchy, you can continue drawing by dragging the supertransition into the border of the parent subchart. This allows you to continue drawing the supertransition at the higher level. In this way, you can connect objects separated by any number of layers in the subchart hierarchy.

Labeling Supertransitions

A supertransition is displayed with multiple resulting transition segments for each layer of containment traversed. For example, if you create a transition between a state outside a subchart and a state inside a subchart of that subchart, you create a supertransition with three segments, each displayed at a different containment level.

You can label any one of the transition segments constituting a supertransition using the same procedure used to label a regular transition (see “Labeling Transitions” on page 4-17). The resulting label appears on all the segments that constitute the supertransition. Also, if you change the label on any one of the segments, the change appears on all segments.

Extending Transitions with Smart Behavior

Transitions with smart behavior, that is, smart transitions, allow their ends to slide around the surfaces of Stateflow objects. This allows transitions to maintain their shapes and uniqueness while you rearrange chart objects. Without this ability, rearranging a Stateflow chart full of states and junctions attached by transitions can soon become a very arduous task. Now, objects move but stay attached in cleaner and more efficient ways. The result is a chart that is cleaner and easier to rearrange.

Examine the behavior of smart transitions in the following topics:

- “Setting Smart Behavior in Transitions” on page 5-19
- “What Smart Transitions Do” on page 5-20
- “What Nonsmart Transitions Do” on page 5-26

Setting Smart Behavior in Transitions

Transitions are automatically created with smart behavior, on the assumption that this behavior is desirable in most circumstances. You can disable or enable smart behavior in existing transitions with the following procedure:

- 1 Right-click a transition.

On the resulting menu, observe the selection titled **Smart**. If a check mark appears in front of **Smart**, the transition has smart behavior.

- 2 If **Smart** is not checked, select it to enable smart behavior.

To disable smart transition behavior, select **Smart** if it is already checked.

See the following sections for a comparison of behavior between smart and nonsmart transitions:

- “What Smart Transitions Do” on page 5-20
- “What Nonsmart Transitions Do” on page 5-26

Note Transitions with smart behavior differ graphically only. Apart from graphical behavior, there is no difference in meaning between a transition with and without smart behavior.

What Smart Transitions Do

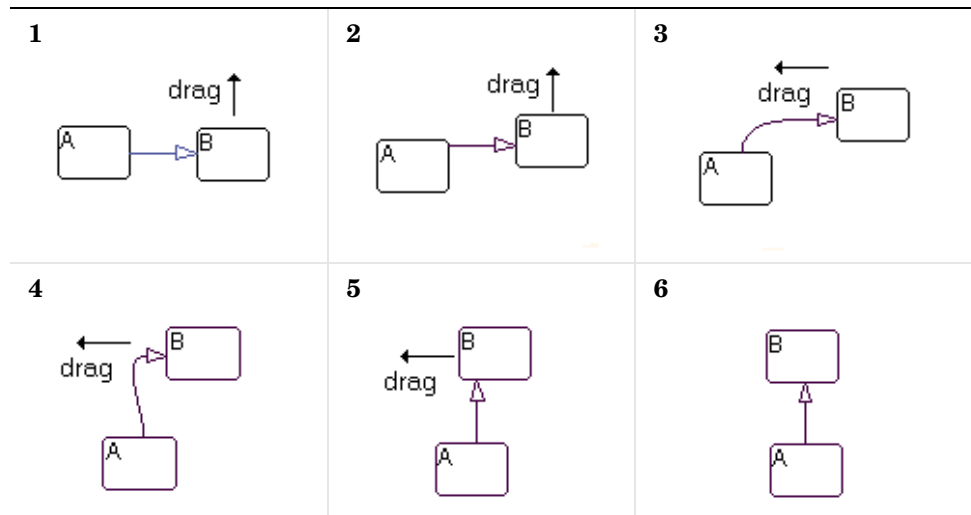
The following topics discuss some of the behaviors of smart transitions:

- “Smart Transitions Slide Around Surfaces” on page 5-20
- “Smart Transitions Slide and Maintain Shape” on page 5-22
- “Smart Transitions Connect States to Junctions at 90 Degree Angles” on page 5-23
- “Smart Transitions Snap to an Invisible Grid” on page 5-24
- “Smart Transitions Bow Symmetrically” on page 5-25

Smart Transitions Slide Around Surfaces

In the following example, state B is attached to state A by a smart transition. The example shows state B being dragged counterclockwise around the upper right corner of state A. When this occurs, state B turns to its selection color and

the transition turns to a very light shade of gray, a sure sign of smart behavior. Dragging direction is shown by the arrows.



Note the following step-by-step behavior for the preceding example:

- 1** The first capture shows states A and B at the beginning of movement.
- 2** As B moves upward, the transition's back end slides upward on A, maintaining the transition straight.
- 3** As B moves around A's corner, the back end of the transition suddenly hops around A's upper right-hand corner. The transition is now curved from A's top surface to B's left side, maintaining perpendicularity with each attached state side.

Note A hop around a state's corner is a necessity because transitions are restricted from attaching at corners of states.

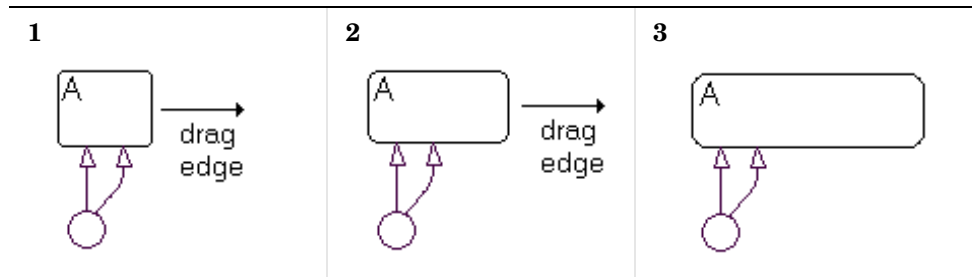
- 4** As B moves on top of A, the transition stays curved but its front end slides down to B's lower left-hand corner.

- 5 As B continues to move to the left over A, the transition's front end hops around B's lower left-hand corner.
- 6 Finally, as B moves directly over A, the transition's front end slides onto B's bottom edge.

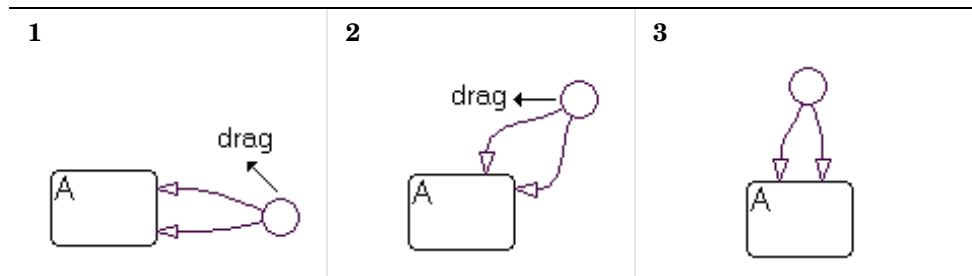
As B continues to circle A, steps 1 through 6 repeat for each of A's remaining sides.

Smart Transitions Slide and Maintain Shape

While transitions with smart behavior allow their ends to slide around the surfaces of their connected objects, they also attempt to maintain their original shape during moving. In the following example, a pair of transitions with smart behavior slide during a resizing to maintain their original shape.

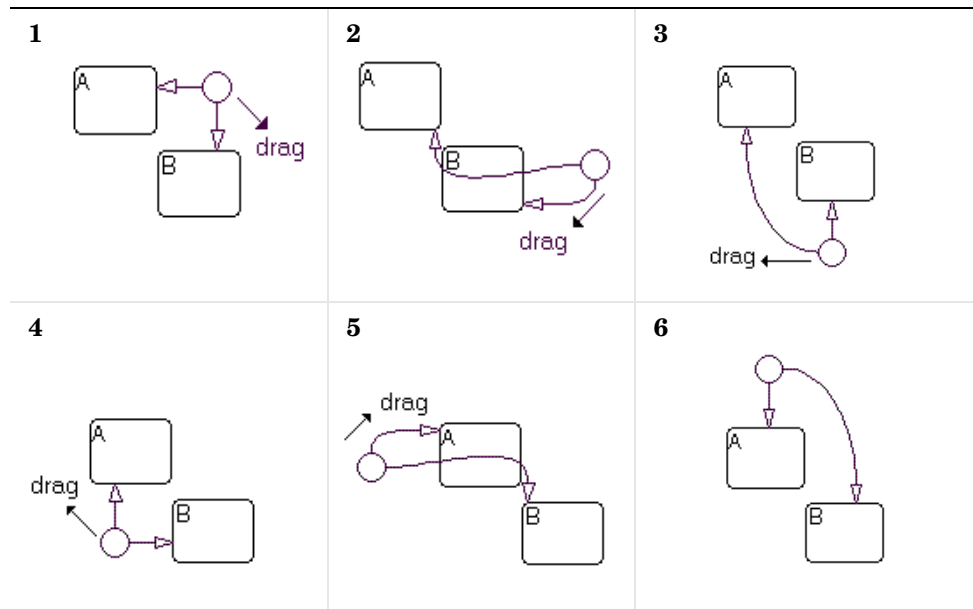


In the following example, the ends of a pair of transitions with smart behavior emanate from a junction and terminate in a state. As the junction is dragged around the state, the ends slide around the state and maintain the same relative spacing between each other. Direction is indicated by the arrows.



Smart Transitions Connect States to Junctions at 90 Degree Angles

Straight-line connections to states must be in one of four directions: left, right, up, or down. To maintain their straightness, smart transitions from junctions always seek to connect to a state through equivalent locations on the junction (left, right, top, bottom). In the following example, a junction is connected to two states, A and B. Watch the behavior of two straight smart transitions as the junction is moved to different locations.



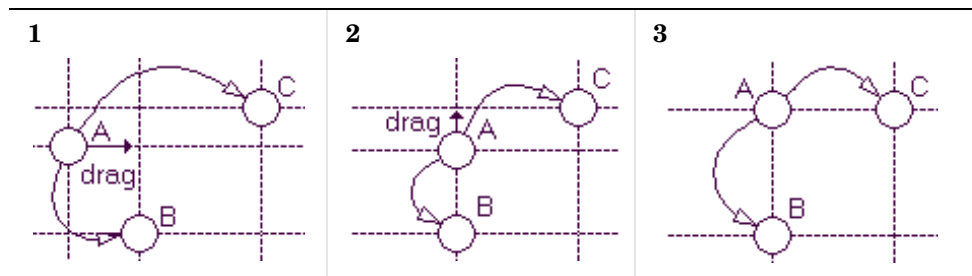
- 1 The junction starts with two straight smart transition connections to states A and B.
- 2 Stateflow chooses to connect the junction to state A through the junction's left side. Since the junction is below A, only a curved connection is possible.

State B could be connected by a straight line through the junction's left side, but this is already occupied by the connection to A. Therefore, B is connected through the junction's bottom, and must be curved.

- 3 Stateflow connects the junction to B by a straight transition through the junction's top connection. No straight-line connection to A is possible, therefore the junction is connected to state A with a curved transition through its left side.
- 4 At this location (under A, to the left of B), straight-line transitions to A and B are possible from the junction's top and right connection points, respectively.
- 5 At the location left of state A, Stateflow chooses to connect to state B through its right connection point. Since the junction is above B, only a curved connection is possible.
- 6 Above A, a straight-line transition to state A is possible through the junction's bottom connector. A straight-line connection to state B is not possible, so the junction is connected to state B through a curved transition from its right connection.

Smart Transitions Snap to an Invisible Grid

Junctions that are connected to other junctions with smart transitions will snap to an invisible grid consisting of horizontal and vertical lines that pass through the center of each junction. The following example depicts this behavior.

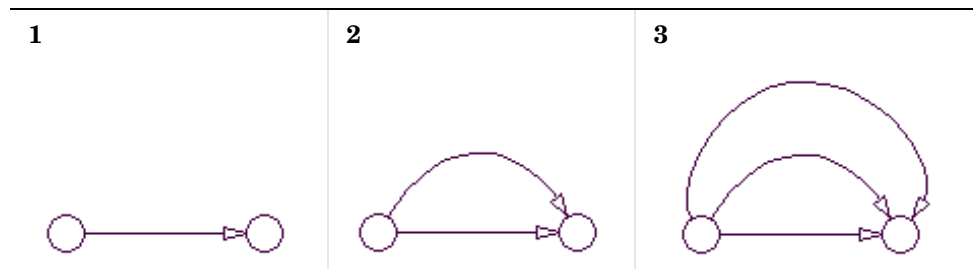


Here, the invisible grid is depicted for each of the three junctions by dashed vertical and horizontal lines. Each junction is connected to each other through nonlinear smart transitions:

- 1 In the first scene, the snap grid for each junction does not overlap. The arrow indicates that junction A is being moved toward the vertical snap line for junction B.
- 2 When A is within a very small distance of B's snap line, A snaps into position directly above B and centered in its vertical snap line. The arrow indicates that A is now being moved toward the horizontal snap line for junction C.
- 3 When A is within a very small distance of C's horizontal snap line, A snaps into position directly to the side of C and centered in its horizontal snap line.

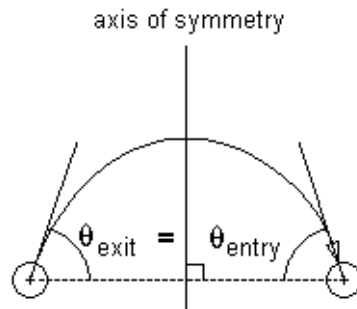
Smart Transitions Bow Symmetrically

Transitions with smart behavior bow symmetrically between junctions. In the following examples, transitions with smart behavior are drawn between two junctions:



- 1 In the first case, a transition originates at the junction on the left and terminates on the left side of the right junction. This results in a straight line.
- 2 In the second case, a transition originates at the junction on the left and terminates on the top of the right junction. This results in a transition line bowed up.
- 3 In the third case, a transition originates at the junction on the left and terminates on the right side of the right junction. This results in a transition line bowed up even more.

Bowed smart transitions maintain symmetry by maintaining equality between transition entry and exit angles as shown below.



You can bow a smart transition between two junctions to any degree by placing the mouse cursor on any point in the transition (except the attachment points) and clicking and dragging in a direction perpendicular to a straight line connecting the two junctions. You can move the mouse in any direction to bow the transition but Stateflow only uses the component perpendicular to the line connecting the two junctions.

Disabling smart behavior for a transition allows you to distort the transition asymmetrically (see section “Nonsmart Transitions Distort Asymmetrically” on page 5-28). However, if you enable smart behavior again, the transition automatically returns to its prior symmetric bowed shape.

What Nonsmart Transitions Do

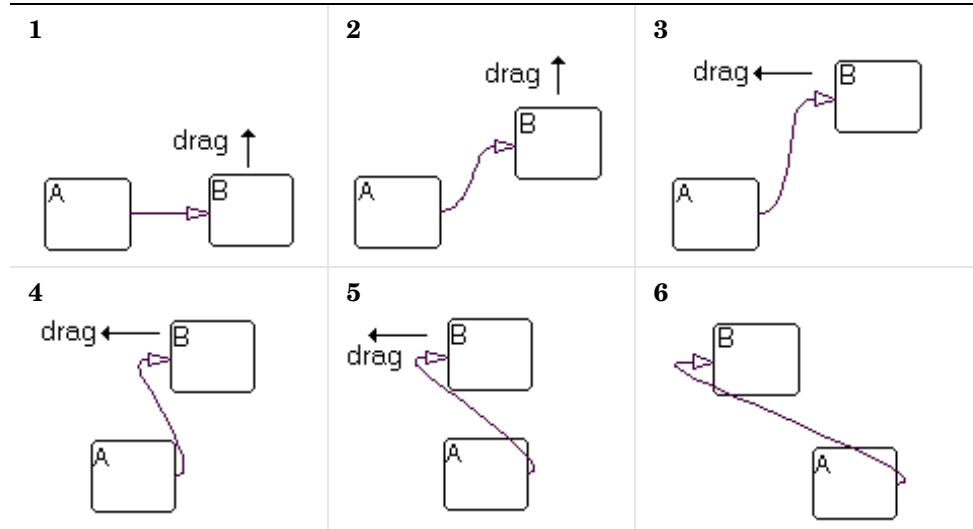
The following topics describe some of the behavior exhibited by transitions without smart behavior.

- “Nonsmart Transitions Anchor Connection Points” on page 5-27
- “Nonsmart Transitions Distort Asymmetrically” on page 5-28

You can disable and enable smart behavior in transitions. See the section “Setting Smart Behavior in Transitions” on page 5-19.

Nonsmart Transitions Anchor Connection Points

Contrast the example in the section “Smart Transitions Slide Around Surfaces” on page 5-20 with the example shown below.

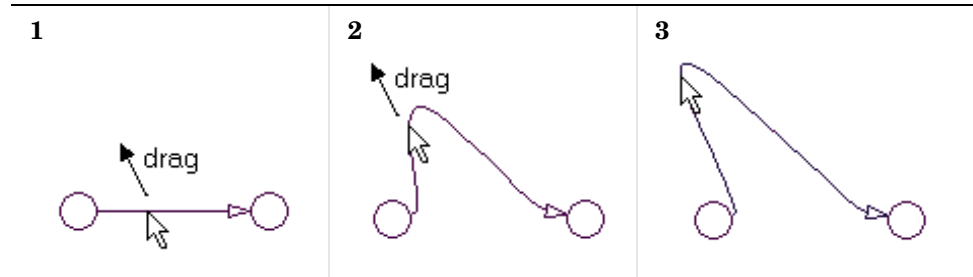


A nonsmart transition connects state A to state B. The mouse cursor is then placed over state B and clicked and dragged to new locations counterclockwise around A. When this occurs, state B turns to its highlight color but the transition remains unchanged, a sure sign of a nonsmart transition.

As B is moved around A, the transition changes into a distorted curve that seeks to maintain the original attachment points. These remain unchanged in position, although the angle of attachment is always perpendicular to the side of the state.

Nonsmart Transitions Distort Asymmetrically

Simply by clicking and dragging on different locations along a transition without smart behavior, you can reshape it into an asymmetric curve suited to your individual preferences. This is illustrated in the following example:



For this example, use the following procedure:

- 1** Drag a horizontal transition between two junctions.
- 2** Right-click the transition and select **Smart** from the resulting shortcut menu to disable smart behavior.
- 3** Place the mouse cursor on any point on the transition.
- 4** Click and drag the mouse cursor up and to the left.

Using Functions to Extend Actions

A function in Stateflow is an extension of Stateflow actions. You define a program once in a function, but call it as many times as you need in Stateflow action language. This provides convenience and power to Stateflow action language.

Stateflow defines three types of functions: graphical, truth table, and embedded MATLAB. Truth tables are described in Chapter 10, “Truth Table Functions” and embedded MATLAB functions are described in Chapter 11, “Using Embedded MATLAB Functions”. This section describes graphical functions.




Learn how to create and use a functions in Stateflow diagrams in the following topics:

- “Creating a Function” on page 5-29 — Shows you how to create a graphical function and describes the significance of its location.
- “Programming Functions” on page 5-33 — Introduces you to the programming concepts for each type of function: graphical, truth table, and embedded MATLAB.
- “Calling Functions in Stateflow” on page 5-41 — Shows you where and how graphical functions are called.
- “Exporting Functions” on page 5-41 — Shows you how to share the graphical function of one Stateflow chart with all the charts in a model.
- “Specifying Function Properties” on page 5-43 — Shows you how to access and set properties for graphical functions.

Creating a Function

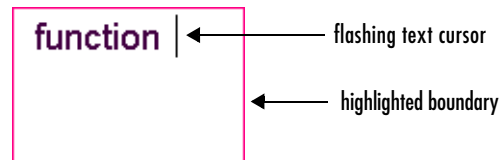
You create a Stateflow graphical, truth table, or embedded MATLAB function in Stateflow diagrams with the following steps:

- 1 Select a drawing tool for the function from the Stateflow drawing toolbar as follows:

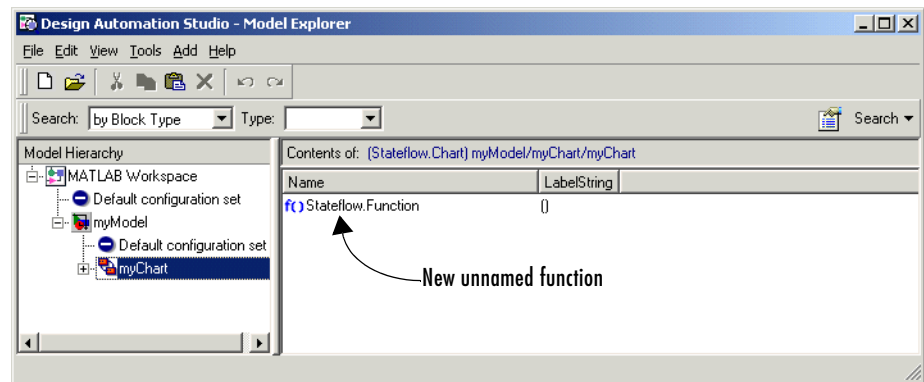
Tool	Function
	Graphical
	Truth Table
	Embedded MATLAB

- 2 Move the cursor to the location for the new function and click to place it.

The new function appears as an unnamed object in the Stateflow diagram editor with a flashing text cursor as seen in the following graphical function example.



The new function also appears in the **Model Explorer** as a child of the chart or state in which it is drawn. In the following example, a graphical function is added to its parent Stateflow chart, myChart.



A function can reside anywhere in a chart, state, or subchart. The location of a function determines its scope, that is, the set of states and transitions that can call the function. In particular, functions are visible to the chart, to the parent state and its parents, and to sibling transitions and states with the following exceptions.

- If the chart containing the function exports its graphical functions, the scope of the function is the entire Stateflow machine, which encompasses all the charts in the model. See “Exporting Functions” on page 5-41 for more information.
- A function defined in a state or subchart overrides any functions of the same name defined in the parents and ancestors of that state or subchart.

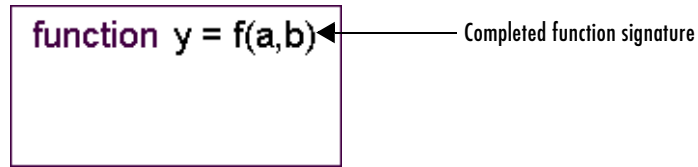
3 Enter the function signature and click outside of the function box.

The function signature specifies a name for the function and formal names for its arguments and return value. A signature has the following syntax:

$$r = \text{func}(a_1, a_2, \dots, a_n)$$

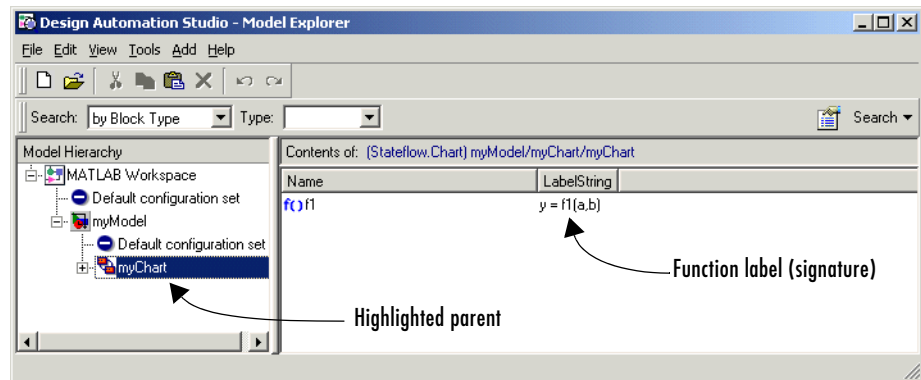
where `func` is the function’s name, a_1 , a_2 , a_n are formal names for its arguments, and r is the formal name for its return value (only one return is allowed). Arguments and return values can be scalars, vectors, or 2-dimensional matrices of any type. Matrices with a row or column dimension of 1 are treated as row or column vectors, respectively.

The following example shows a signature for a graphical function named `f1` that takes two arguments, a and b , and returns a value y .

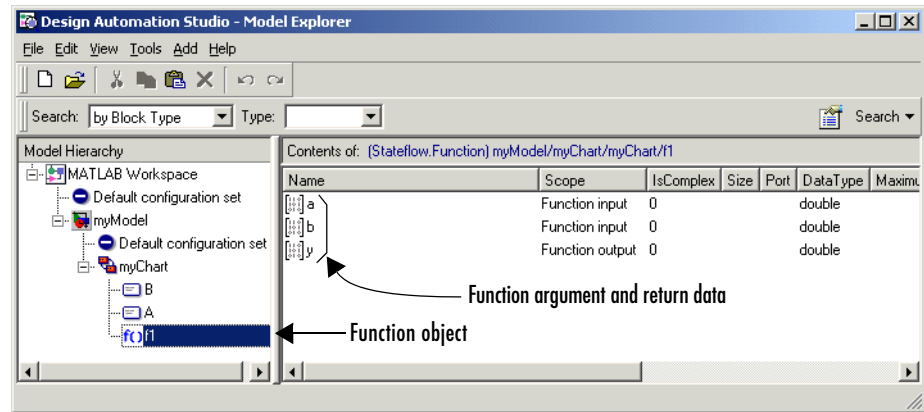


Notice that the function box in the preceding example has been enlarged to accommodate the width of its label signature. You enlarge function boxes in their corners, just as for state and box objects.

The signature for a function appears as a property of its owning object in the **Model Explorer**.



If you expand the parent object in the **Model Explorer**, you can see the return values and arguments that you declare in the signature for a function as data items parented by the function.



The **Scope** field in the Explorer indicates the role of each argument or return value. Arguments have the scope `Function input`, and return values have scope `Function output`.

Note You can use the Stateflow diagram editor to change the signature of a graphical function at any time. When you are done editing the signature, Stateflow updates the data dictionary and the **Model Explorer** to reflect the changes.

Programming Functions

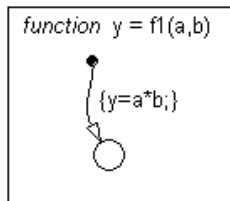
In “Creating a Function” on page 5-29, you learn how to create a function with a return value and arguments in a Stateflow diagram. In the context of programming a function, arguments and return values have local scope. They are visible only in the environment that programs the function. This topic introduces you to the environment for each function in which you program its behavior with the function subtopics that follow.

Graphical Functions

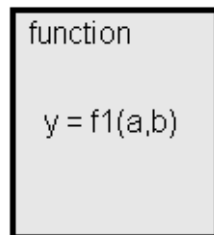
A Stateflow graphical function is a program written with Stateflow flow graphs using connective junctions and transitions. You create a graphical function, fill it with a flow diagram, and call it repeatedly in the actions of states and transitions.

Because a function must execute completely when it is called, states are not allowed in graphical functions. When a state is entered, execution stops until an event occurs.

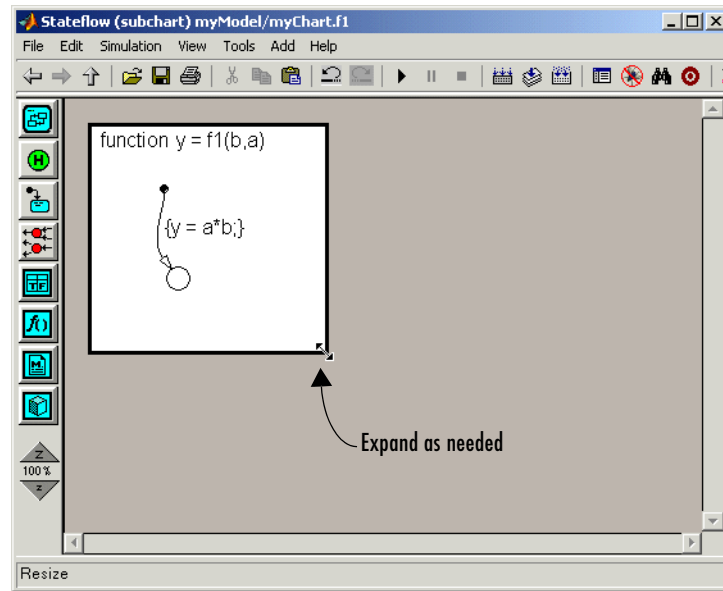
At a minimum, the flow diagram must include a default transition with a terminating junction. The following example shows a minimal flow diagram for a graphical function that returns the product of its arguments.




You can make a graphical function as complicated and as long as you want. However, because complicated graphical functions can become very large, it may be difficult to fit it into the Stateflow diagram. To make the function smaller, hide the function's contents by selecting **Subcharted** from the **Make Contents** menu of the function's shortcut menu. This makes the graphical function opaque as shown.



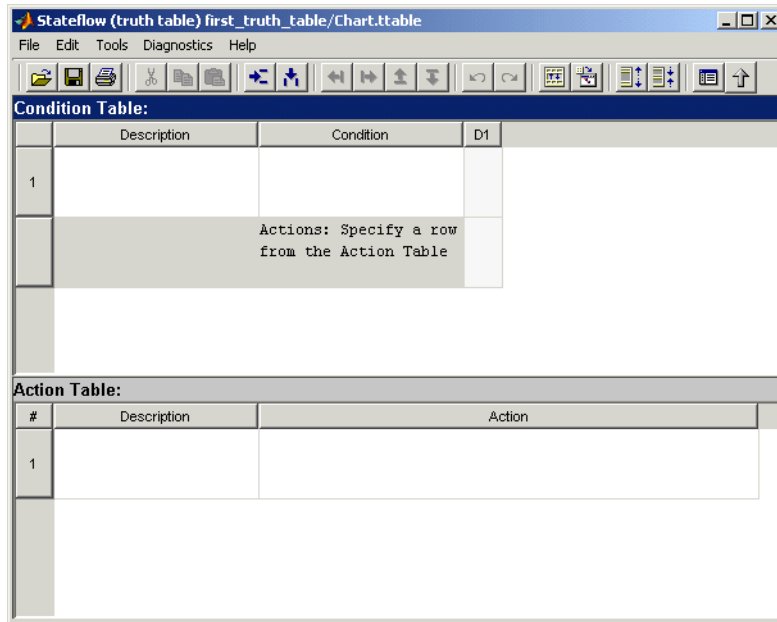
To access the programming of a subcharted graphical function, double-click it. This dedicates the entire Stateflow diagram window to programming the function, as shown.



To access the original Stateflow diagram, select the Back button .

Truth Table Functions

You program a truth table in the truth table editor. Access this editor for a truth table by double-clicking it in the Stateflow diagram, and the truth table editor appears.



Truth table functions in Stateflow are very suitable for implementing functions with logical behavior. You program truth tables with logical behavior in the form of action language conditions, decisions, and actions. For example, the following truth table

The screenshot shows the Stateflow interface with two tables. The top table is the 'Condition Table' and the bottom table is the 'Action Table'.

Condition Table:						
	Description	Condition	D1	D2	D3	D4
1	x is equal to 1	XEQ1: x == 1	T	F	F	-
2	y is equal to 1	YEQ1: y == 1	F	T	F	-
3	z is equal to 1	ZEQ1: z == 1	F	F	T	-
	Actions: Specify a row from the Action Table		A1	A2	A3	A4

Action Table:		
#	Description	Action
1	set t to 1	A1: t=1;
2	set t to 2	A2: t=2;
3	set t to 3	A3: t=3;
4	set t to 4	A4: t=4;

implements the following logic expressed in pseudo-code:

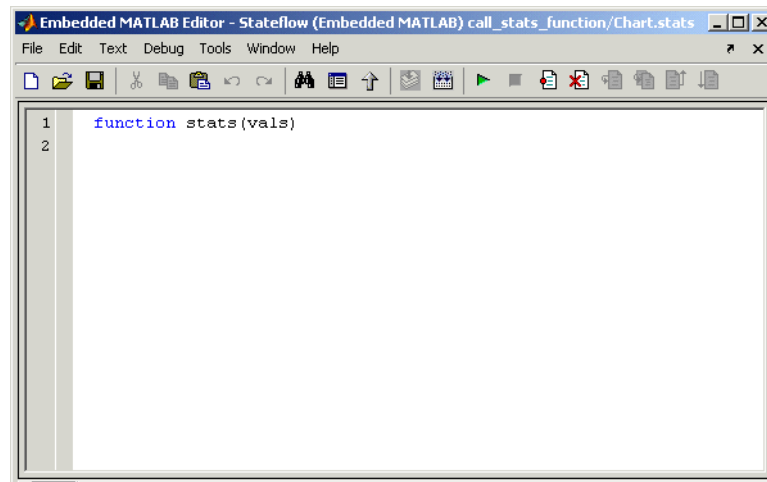
Description	Pseudo-Code
Decision 1 Decision 1 Action	if ((x == 1) & !(y == 1) & !(z == 1)) t = 1;
Decision 2 Decision 2 Action	elseif (!(x == 1) & (y == 1) & !(z == 1)) t = 2;
Decision 3 Decision 3 Action	elseif (!(x == 1) & !(y == 1) & (z == 1)) t = 3;
Default Decision Default Decision Action	else t = 4; endif

When you try to simulate your truth tables, Stateflow detects whether they are underspecified or overspecified. Underspecified truth tables contain fewer than all possible decisions for the conditions you specify. Overspecified truth tables contain redundant decisions that prevent other decisions from being evaluated. Stateflow detects both underspecified and overspecified truth tables and tells you the decisions you need to specify or remove in the truth table.

See Chapter 10, “Truth Table Functions” for a thorough understanding of truth table functions in Stateflow.

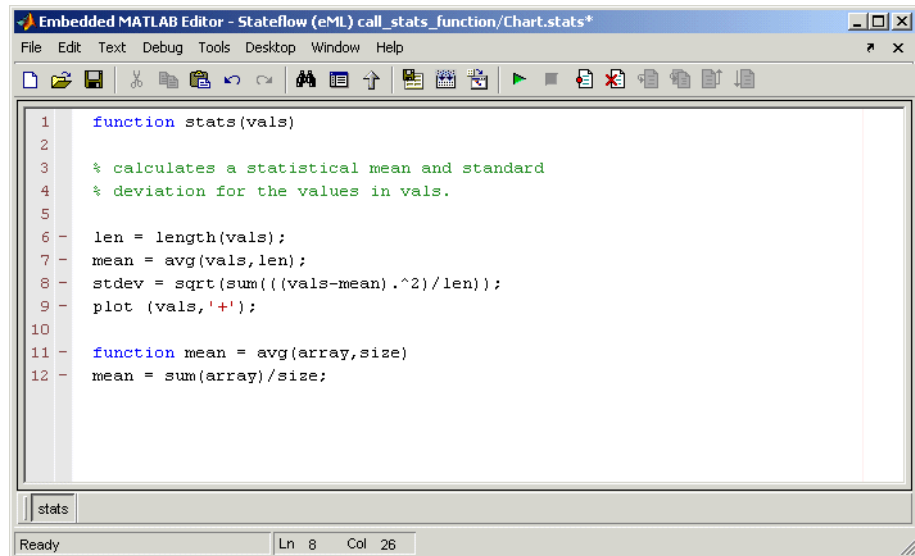
Embedded MATLAB Functions

You program an embedded MATLAB function in the Embedded MATLAB Editor. Access this editor for an embedded MATLAB function by double-clicking it in the Stateflow diagram. The following is an example of a newly created function in Stateflow that you open to program:



You program embedded MATLAB functions just like you would a MATLAB function. Embedded MATLAB functions in Stateflow use a rich subset of MATLAB language and functions to generate code for applications that can reside in target applications with operating systems and platforms that have strict memory and data type requirements.

The following embedded MATLAB function calculates a mean and a standard deviation for a set of value input to the function as a vector argument:



```

1  function stats(vals)
2
3  % calculates a statistical mean and standard
4  % deviation for the values in vals.
5
6  len = length(vals);
7  mean = avg(vals, len);
8  stdev = sqrt(sum((vals-mean).^2)/len);
9  plot (vals, '+');
10
11 function mean = avg(array, size)
12 mean = sum(array)/size;

```

For a detailed example and information on programming embedded MATLAB functions in Stateflow, see “Using Embedded MATLAB Functions” on page 11-1.

Defining Function Data

Before you can finish a function, you need to define all or some of its data. Use the following steps to defined data for a Stateflow function:

- 1 Specify the data properties (data type, initial value, and so on) for the function arguments and return value.
 - Change data properties directly in the **Contents** pane of the **Model Explorer** by clicking values in the displayed columns for a data row.
 - Change data properties in the **Data** dialog for a data.

If you right-click a data row in the **Model Explorer** and select **Properties** from the resulting pop-up menu, its **Data** dialog appears. See “Setting

Data Properties in the Data Dialog” on page 6-25 for information on setting data properties.

The following restrictions apply to setting properties for the arguments and return value of a function:

- Each argument and the return value can be a scalar or matrix of values.
 - A function can have only one return.
 - Arguments cannot have initial values.
- 2** Create any additional data items that the function might need to process its programming when it is called.

A function can access its own data or data belonging to parent states or the chart. The items that you create for the function itself can have any of the following scopes:

- **Local**

A local data item persists from invocation to invocation. For example, if the item is equal to 1 when the function returns from one invocation, the item will equal 1 the next time the function is invoked.

- **Temporary**

Stateflow creates and initializes a copy of a temporary item for each invocation of the function.

- **Constant**

Constant data retains its initial value through all invocations of the function.

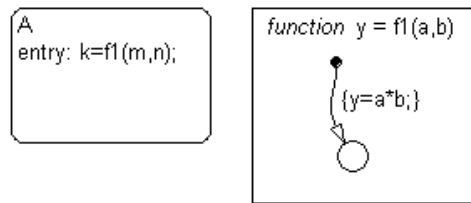
All data items (other than arguments and return values) parented by a function can be initialized from the workspace. However, only local items can be saved to the workspace.

It is not necessary to assign local data to an embedded MATLAB function. You can declare local data in an embedded MATLAB function simply by using it. In this case, Stateflow uses first use assignments to determine the type, size, and value of the local data. See “Declaring Variables Implicitly” in Simulink documentation for specific rules in declaring local data in Embedded MATLAB functions.

Calling Functions in Stateflow

Once you create a graphical function, you use it by calling it in Stateflow action language. Any state or transition action in the scope of a function can call that function. The calling syntax is the same as that of the function signature, with actual arguments replacing the formal parameters specified in the signature. If the data types of the actual and formal argument differ, Stateflow casts the actual argument to the type of the formal parameter.

The following example shows a state entry action that invokes a graphical function that returns the product of its arguments.

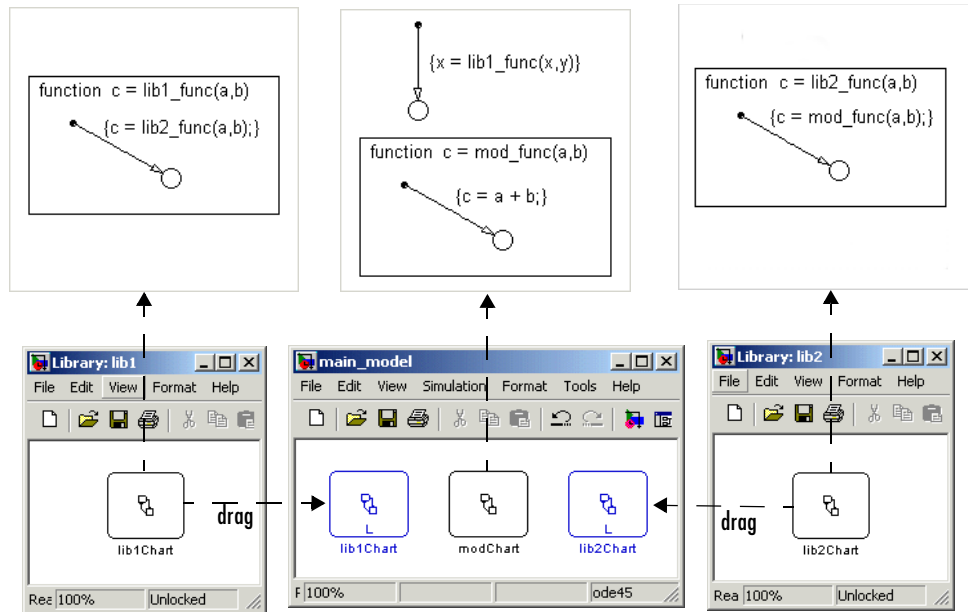


Exporting Functions

You can export the root-level functions of a chart to the remaining charts in the chart's model. Exporting a chart's functions extends their scope to include all other charts in the same model.

You can also export functions in library charts to a model as long as the library charts are present in the model. To export a chart's root-level functions, select **Export Chart Level Functions** on the chart's **Chart Properties** dialog box (see "Specifying Chart Properties" on page 9-4).

In the following example, the model `main_model` has two library stateflow charts, `lib1Chart` and `lib2Chart`:



Both lib1Chart and lib2Chart are dragged into the model main_model from the library models lib1 and lib2 in which they were created. In the properties dialog for all three charts, the **Export Chart Level Functions** option is selected. Each chart now defines a graphical function that can be called by any other chart placed in main_model.

The sequence of action in simulation of main_model is as follows:

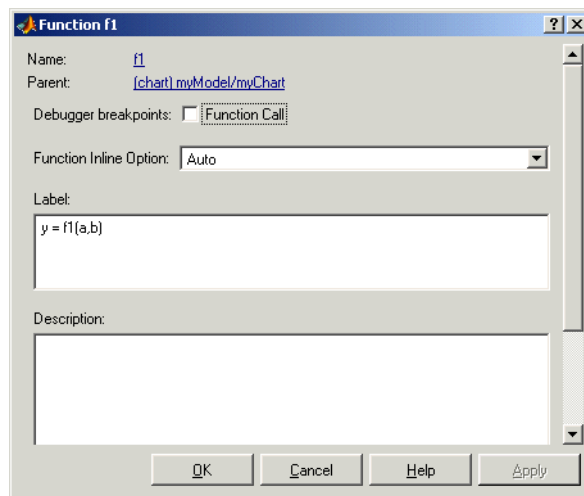
- The chart modChart calls the graphical function lib1_func, with the two arguments, x and y.
- lib1_func calls the graphical function lib2_func, passing the same two arguments.
- lib2_func calls the graphical function mod_func, which adds x and y.
- The result of the addition is assigned to x.

Specifying Function Properties

You can set general properties for a function through its function dialog as follows:

- 1 Right-click the function box.
- 2 Select **Properties** from the resulting submenu.

The **Function Properties** dialog for the function appears, as shown:



The fields in the **Function Properties** dialog are as follows:

Field	Description
Name	Function name; read-only; click this hypertext link to bring the function to the foreground in its native diagram.
Parent	Parent of this function; read-only; a forward slash character (/) indicates that the Stateflow diagram is the parent; click this hypertext link to bring the parent to the foreground in its native diagram.

Field	Description
Debugger breakpoints	Select Function Call to set a breakpoint to pause execution during simulation when the function is called.
Function Inline Option	<p>This option controls the inlining of this function in generated code through the following selections:</p> <ul style="list-style-type: none"> • Auto Stateflow decides whether or not to inline the function based on an internal calculation. • Inline Stateflow inlines the function as long as it is not exported to other charts and is not part of a recursion. A recursion exists if the function calls itself either directly or indirectly through another called function. • Function The function is not inlined.
Label	You can specify the signature label for the function through this field. See “Creating a Function” on page 5-29 for more information.
Description	Textual description/comment.
Document Link	Enter a URL address or a general MATLAB command. Examples are <code>www.mathworks.com</code> , <code>mailto:email_address</code> , and <code>edit/spec/data/speed.txt</code> .

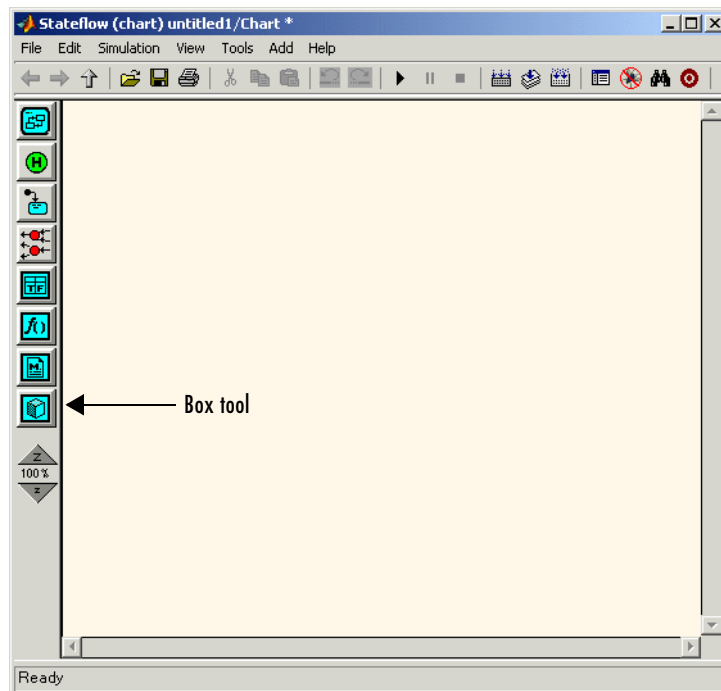
Using Boxes to Extend Chart Diagrams

Boxes are a convenience for organizing related items in your charts. Use the following topics to understand and use boxes in Stateflow diagrams:

- “Changing a State to a Box” on page 5-46
- “Using Boxes in Stateflow” on page 5-47

Creating a State

Like states, you create boxes by drawing them in the Stateflow diagram editor with the box tool shown below:



- 1 Select the Box tool.

- 2 Move your cursor into the drawing area.

In the drawing area, the mouse cursor becomes box-shaped.

- 3 Click in a particular location to create a box.

The created box appears with a question mark (?) name in its upper left-hand corner.

- 4 Click the question mark label.

A text cursor appears in place of the question mark.

- 5 Enter a name for the box and click outside of the box when finished.

To delete a box, click it to select it and choose **Cut (Ctrl+X)** from the **Edit** or any shortcut menu or press the **Delete** key.

You can change a state to a box or a box to a state. See “Changing a State to a Box” on page 5-46 for details.

Changing a State to a Box

You can change an existing state to a box and back to a state with the following procedure:

- 1 Right-click the state.

A shortcut pop-up menu appears.

- 2 From the pop-up menu, select **Type**.

A submenu appears adjacent to the pop-up menu.

- 3 From the submenu, select **Box**.

Stateflow converts the state to a box, redrawing its border with sharp corners to indicate its changed status.

- 4 Repeat the preceding steps on the box and select **State** from the submenu instead of **Box** to change the box to a state

Using Boxes in Stateflow

Once you create a box you can use it in one of the following ways:

- You can move or draw objects inside of a box to organize your diagram.
You can draw the box first as a state around the objects you want inside it and then convert it to a box.
- You can add data to a box so that all the elements in the box can share the same data.
- You can group a box and its contents into a single graphical element .
See “Grouping States” on page 4-8.
- You subchart a box to hide its elements.
See “Using Subcharts to Extend Charts” on page 5-5 for more information.

For the most part, boxes do not contribute to the semantics of a Stateflow diagram. They do, however, affect the activation order of a diagram’s parallel states. A box wakes up before any parallel states or boxes that are lower or to the right of it in a Stateflow diagram. Within a box, parallel states still wake up in top-down, left to right order.

Using Notes to Extend Chart Diagrams

Learn how to create, edit, and delete descriptive notes for your Stateflow chart with the following topics:

- “Creating Notes” on page 5-48 — Tells you how to create a chart note at any location in the Stateflow diagram editor.
- “Editing Existing Notes” on page 5-49 — Tells you how to edit an existing chart note.
- “Changing Note Font and Color” on page 5-49 — Shows you how to change the font and color of a chart note and also format it with TeX format.
- “Moving Notes” on page 5-50 — Tells you how to move a chart note from one location to another in the Stateflow diagram editor.
- “Deleting Notes” on page 5-50 — Shows you how to delete an existing chart note.

Note Chart notes are descriptive only. They do not contribute in any way to the behavior or code generation of a chart.

Creating Notes

You can enter comments/notes in any location on the chart with the following procedure:

- 1 Place the cursor at the desired location in the Stateflow chart.
- 2 Right-click the mouse.
- 3 From the resulting menu, select **Add Note**.

A blinking cursor appears at the location you selected. Default text is italic, 9 point.

- 4 Begin typing your comments.

As you type, the text moves left to right.

- 5 Press the **Return** key to start a new line.
- 6 When finished typing, click outside the typed note text.

Editing Existing Notes

To edit existing note text,

- 1 Left-click the mouse on the comment location you want to edit.
- 2 Once the blinking cursor appears, begin typing or use the arrow keys to move to a new text location.

Changing Note Font and Color

To change font and color for your Stateflow chart notes, follow the procedures described in the section “Specifying Colors and Fonts” on page 4-31.

You can also change your note text to bold or italic text by doing the following:

- 1 Right-click the note text.
- 2 From the resulting shortcut menu, select **Text Format**.
- 3 From the resulting submenu, select **Bold** or **Italic** (default).

TeX Instructions

In the preceding procedure, note a third selection of the **Text Format** submenu called **TeX Instructions**. This selection sets the text Interpreter property to Tex, which allows you to use a subset of TeX commands embedded in the string to produce special characters such as Greek letters and mathematical symbols.

The **TeX Instructions** selection is used in the following example:

- 1 Right-click the text of an example note.
- 2 In the resulting shortcut menu, select **Text Format**.
- 3 In the submenu that results, make sure that **TeX Instructions** has a check mark positioned in front of it. Otherwise, select it.
- 4 Click the note text to place the cursor in it.

- 5 Replace the existing note text with the following expression.

$$\text{\it{\omega}_N = e^{\{-2\pi i\}/N}}$$

- 6 Click outside the note.

The note now has the following appearance:

$$\omega_N = e^{(-2\pi i)/N}$$

Moving Notes

To move your notes,

- 1 Place the cursor over the text of the note.
- 2 Click and drag the note to a new location.
- 3 Release the left mouse button.

Deleting Notes

To delete your notes, do the following:

- 1 Place the mouse cursor over the text of the note.
- 2 Click and hold the left mouse button on the note.
A dim rectangle appears surrounding the note.
- 3 Select the **Delete** key.

Alternatively, you can also do the following:

- 1 Place the mouse cursor over the text of the note.
- 2 Right-click the note.
- 3 From the resulting shortcut menu, select **Cut**.

Reporting Chart Diagrams

Stateflow offers the following reports for printing parts or all of your Stateflow chart:

- “Printing and Reporting on Stateflow Charts” on page 5-51 — Tells you how to print the Simulink block diagram containing the Stateflow diagram currently displayed in the Stateflow diagram editor.
- “Generating a Model Report in Stateflow” on page 5-53 — Tells you how to make a comprehensive report of Stateflow objects relative to the currently displayed chart or subchart in the Stateflow diagram editor.
- “Printing the Current Stateflow Diagram” on page 5-56 — Tells you how to print the Stateflow diagram currently displayed in the Stateflow diagram editor.
- “Printing a Stateflow Book” on page 5-56 — Tells you how to generate a report that documents the Stateflow components of a Stateflow model.

You can also use the Report Generator for MATLAB and Simulink to generate a report that documents an entire Stateflow model, including both Simulink and Stateflow components. See the Report Generator for MATLAB documentation.

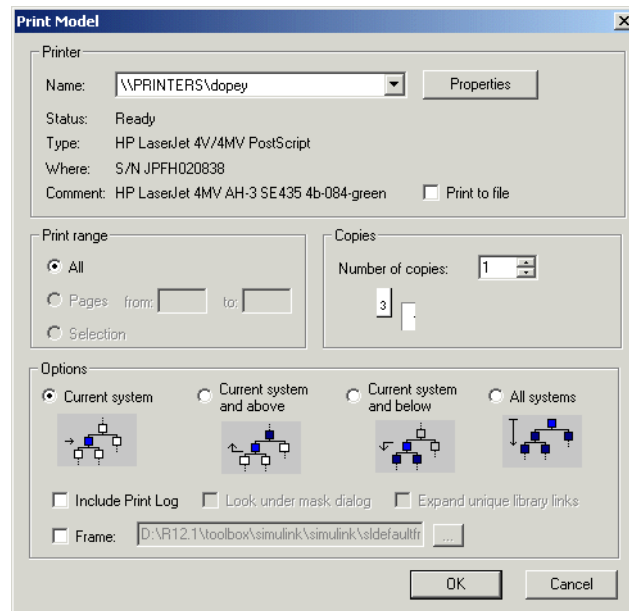
Printing and Reporting on Stateflow Charts

The **Print** option prints a copy of the current Stateflow diagram loaded in the Stateflow diagram editor. You can also select to print subcharts of the current diagram or the chart, subcharts, and Simulink subsystems that contain the current diagram.

Print a copy of a Stateflow diagram by doing the following:

- 1 Open the Stateflow chart or subchart you want to print.
- 2 Select **Print** from the **File** menu.

The **Print Model** dialog box appears as follows:



In the resulting **Print Model** window, select the printer for this report and one of the following options for the type of report you receive:

- **Current system** — Prints the current chart or subchart in view in the Stateflow editor.
- **Current system and above** — Prints the current chart or subchart in view in the Stateflow editor and all the subcharts and Simulink subsystems that contain it.
- **Current system and below** — Prints the current chart or subchart in view in the Stateflow editor and all the subcharts that it contains.
- **All systems** — Prints the current chart or subchart in view in the Stateflow editor, all the subcharts that it contains, and all the subcharts and Simulink subsystems that contain it.

Further enhance the above reports with the following options:

- **Include Print Log** — Includes a list of all printed diagrams as a preface to the print report.

- **Look under mask dialog** — Applies only to the masked subsystems that might appear in Simulink subsystems that are printed with the report options **Current system and below** and **All systems**.
- **Expand unique library links** — Applies only to the library blocks that might appear in Simulink subsystems that are printed with the report options **Current system and below** and **All systems**.
- **Frame** — Prints a title block frame that you specify in the adjacent field on each diagram in the report.

Note This option is also available in the Simulink window. See the topic “Printing a Block Diagram” in the Using Simulink documentation for more information on the preceding options and on the behavior of this command as used in Simulink. The information in this topic describes the behavior of this option only when it is used in a Stateflow diagram editor window.

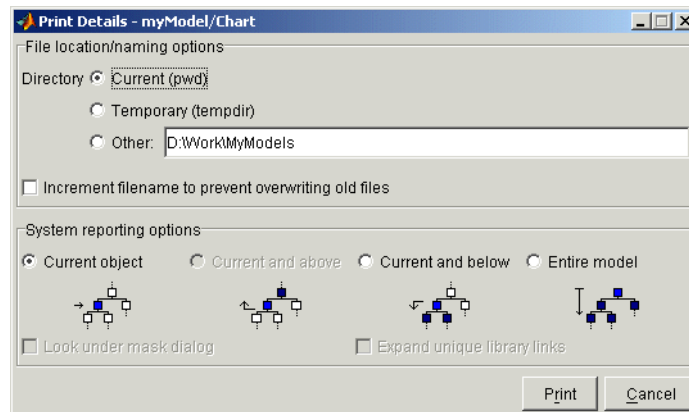
Generating a Model Report in Stateflow

The **Print Details** report in Stateflow is an extension to the **Print Details** report in Simulink. It provides a report of Stateflow and Simulink model objects relative to the Stateflow diagram currently in view in the Stateflow diagram editor from which you select the report.

To generate a model report on Stateflow diagram objects, do the following:

- 1 Open the Stateflow chart or subchart whose objects you want to report on.
- 2 In the diagram editor window, select **Print Details** from the **File** menu.

The **Print Details** dialog box appears as follows:



- 3 Make selections for the destination directory of the report file and reporting options that determine what objects get reported.

For details on setting the fields in the **File locations/naming options** section of this dialog, see “Generating a Model Report” in the Using Simulink documentation. For details on the report you receive from the report option you choose in the **System reporting options** section, see “System Report Options” on page 5-54 and “Report Format” on page 5-55.

- 4 Select **Print**.

The **Print Details** dialog box appears and tracks the activity of the report generator during report generation. See “Generating a Model Report” in the Using Simulink documentation for more details on this window.

If no serious errors are encountered, the resulting HTML report is displayed in your default browser.

System Report Options

Reports for the current Stateflow diagram vary with your choice of one of the **System reporting options** fields as follows:

- **Current** — Reports on the chart or subchart in the current Stateflow diagram editor and its immediate parent Simulink system.

- **Current and above** — This option is grayed out and unavailable for printing chart details in Stateflow.
- **Current and below** — Reports on the chart or subchart in the current Stateflow diagram editor and all contents at lower levels of containment (children) along with the immediate Simulink system.
- **Entire model** — Reports on the entire model including all Stateflow charts in the model for the chart in the current Stateflow diagram editor and all Simulink systems.

If this option is selected, the following options are enabled to modify this report:

- **Look under mask dialog** — Include the contents of masked subsystems in the report.
- **Expand unique library links** — Include the contents of library blocks that are subsystems in the report.

The report includes a library subsystem only once even if it occurs in more than one place in the model.

Report Format

The general top-down format of the **Print Details** report in Stateflow is as follows:

- The report is titled with the system in Simulink containing the chart or subchart in current view in Stateflow.
- A representation of Simulink hierarchy for the containing system and its subsystems follows. Each subsystem in the hierarchy is linked to the report of its Stateflow diagrams.
- The report section for the Stateflow diagrams of each system or subsystem begins with a small report on the system or subsystem and is followed by a report of each contained diagram.
- Each Stateflow diagram report includes a reproduction of its diagram with links for subcharted states that have reports of their own.
- Covered Stateflow and Simulink objects are tabulated and counted in a concluding appendix to the report.

Printing the Current Stateflow Diagram

The **Print Current View** option prints an individual Stateflow chart or subchart diagram as follows:

- 1 Open the chart or subchart that you want to print.
- 2 Select **Print Current View** from the Stateflow editor's **File** menu.
- 3 In the resulting submenu, choose one of the following destination options:
 - **To File** — Converts the current view to a graphics file.
Select the format of the graphics file from a resulting submenu of graphical file types.
 - **To Clipboard** — Copies the current view to the system clipboard.
Select the graphical format for the copy to the clipboard from a resulting submenu of graphical formats.
 - **To Figure** — Converts the current view to a MATLAB figure window.
 - **To Printer** — Prints the current view on the current printer.

You can also print the current view from the MATLAB command line using the `sfprint` command. See the “Using the Stateflow API” on page 20-1.

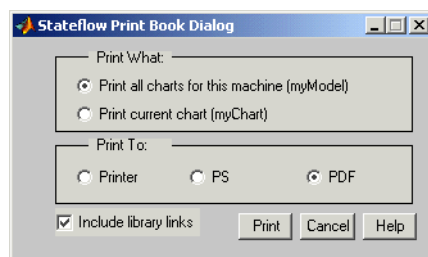
Printing a Stateflow Book

The **Print Book** report documents all the elements of a Stateflow chart, including states, transitions, junctions, events, and data. You can generate a book documenting a specific chart or all charts in a model.

To generate a book report of the objects of a Stateflow diagram, do the following:

- 1 Select and open a chart or subchart that you want to document.
- 2 Select **Print Book** from the Stateflow editor's **File** menu.

The **Print Book** dialog box appears as follows:



- 3 Select the desired print options on the dialog.
- 4 Click the **Print** button to generate the report.

Defining Events and Data

Stateflow diagrams respond to events from Simulink and other Stateflow objects. Stateflow responds to events by taking transitions and/or performing actions. Actions use data variables to perform computations that control diagram behavior and generate output handed on to the Simulink model. Before you can use events and data in Stateflow, you must first define them. Learn how to define events and data in Stateflow in the following sections:

Adding Events (p. 6-3)	Learn how to define the events you need to trigger actions in Stateflow or its environment.
Setting Event Properties in the Event Dialog (p. 6-7)	A reference to the fields of the Event dialog for setting the properties of an event in Stateflow.
Sharing Events with Simulink (p. 6-11)	Shows you how to define the input and output events for a Stateflow chart that allow it to communicate with other Simulink blocks.
Sharing Events with Stateflow External Code (p. 6-16)	Shows you how to define events in Stateflow that enable external code to send events to other charts in the model and receive events from other charts in the model.
Defining Implicit Events (p. 6-18)	Describes events that Stateflow triggers implicitly for actions such as entry in or exit from a state.
Adding Data (p. 6-21)	Learn how to define the data that Stateflow stores internally in its own workspace.
Setting Data Properties in the Data Dialog (p. 6-25)	A reference to the fields of the Data dialog for setting the properties of a data in Stateflow.
Sharing Stateflow Data with Simulink and MATLAB (p. 6-35)	Learn different ways that you can share data with Simulink and MATLAB in a Stateflow diagram.
Sharing Data with Stateflow External Code (p. 6-42)	Learn how to export data to Stateflow external code and import data from Stateflow external code to a Stateflow chart in the model.

Typing Stateflow Data (p. 6-45)

Learn how to specify the type of your Stateflow data.

Sizing Stateflow Data (p. 6-50)

Learn different ways to specify the size of your Stateflow data as a vector or matrix.

Adding Events

An event is a Stateflow object that triggers actions in a Stateflow machine or its environment. Stateflow defines a set of events that typically occur whenever a Stateflow machine executes (see “Defining Implicit Events” on page 6-18). You can define other types of events that occur only during execution of a specific Stateflow machine or its environment through the Scope property for an event, which can have one of the following values:

- **Local** — Event is broadcast in the Stateflow diagram to the diagram.
- **Input from Simulink** — Event is broadcast by another Simulink block to the Stateflow diagram.
- **Output to Simulink** — Event is broadcast in the Stateflow diagram to another Simulink block.

Events that you add to a parent object are visible to all of its children. Events that you add to the Stateflow machine are visible to all Stateflow diagrams in the model and all of their states and substates. Events that you add to the Stateflow diagram are visible to the diagram and all of its states and substates. Events that you add to a state is visible to the state and all of its substates.

You can add events in the Stateflow diagram editor and in the **Model Explorer**. In the **Model Explorer** you can add events to all levels of Stateflow hierarchy including the Stateflow machine, the Stateflow diagram (chart), and individual states in the Stateflow diagram. In the Stateflow diagram editor, you can add events only to the Stateflow diagram.

See one of the following topics to learn how to add events from the Stateflow diagram editor or **Model Explorer**:

Adding Events in the Stateflow Diagram Editor

If you add events in the Stateflow diagram editor, they are parented by the Stateflow diagram and are visible to all objects in the chart. To add an event in the Stateflow diagram editor, do the following:

- 1 From the **Add** menu of the Stateflow editor, select **Event**.
- 2 In the resulting pop-up menu, select one of the following scopes for the use of the event:

- **Local** — Event is broadcast in the Stateflow diagram to the diagram.
- **Input from Simulink** — Event is broadcast by another Simulink block to the Stateflow diagram.
- **Output to Simulink** — Event is broadcast in the Stateflow diagram to another Simulink block.

Stateflow adds a default definition of the new event to the Stateflow data dictionary and displays the **Event** dialog box.

3 Specify properties for the event specified in the **Event** dialog box.

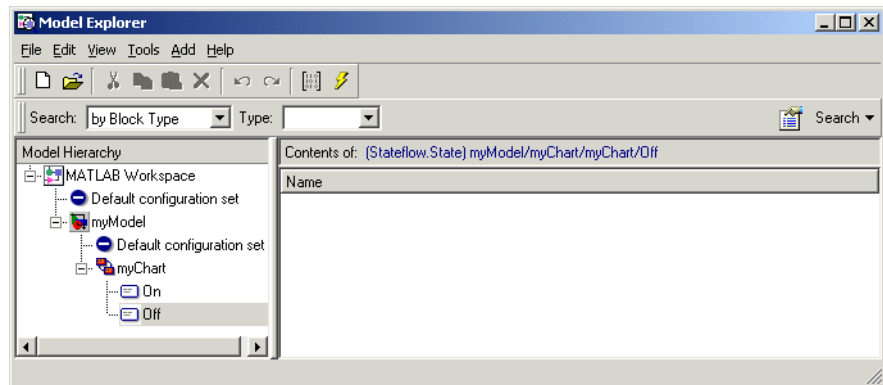
See “Setting Event Properties in the Event Dialog” on page 6-7 for a description of the property fields of the **Event** dialog box.

Adding Events in the Model Explorer

If you add events in the **Model Explorer**, you can choose any Stateflow object to parent the new event. This event is visible to the parent object and all child objects of the parent. To add an event in the **Model Explorer**, do the following:

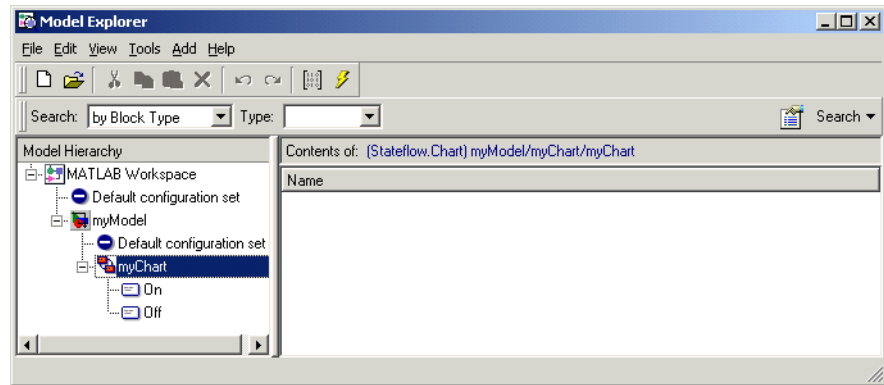
- 1** Select **Explore** from the Stateflow editor’s **Tools** menu.

Stateflow opens the **Model Explorer**.



If no object is selected, the current diagram or subchart is highlighted in the **Model Hierarchy** pane of the **Model Explorer**. Otherwise, the selected object is highlighted. In the preceding example, state Off is selected in the Stateflow diagram myChart when the **Model Explorer** is called.

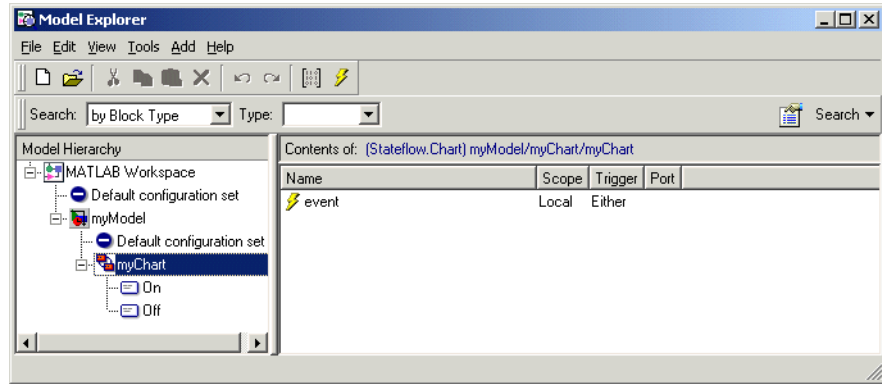
- 2 Select the object (machine, chart, or state) in the object hierarchy pane where you want the new event to be visible.



In the preceding example, the Stateflow chart myChart is selected to receive the new event.

- 3 From the **Add** menu of the **Model Explorer**, select **Event**.

Stateflow adds a default definition for the new event in the data dictionary and displays an entry row for the new event in the Explorer's Contents pane.



- 4 Change the properties of the event you add in one of the following ways:
- Right-click the event row and select **Properties** to access all properties for the event through its **Event** dialog.
See “Setting Event Properties in the Event Dialog” on page 6-7 for a description of each property for an event.
 - Click individual entries in the entry row to set just the **Name**, **Scope**, **Trigger Type**, and **Port** properties for the event.

Setting Event Properties in the Event Dialog

You set all properties for an event through its **Event** dialog. You access the **Event** dialog for an event in one of the following ways:

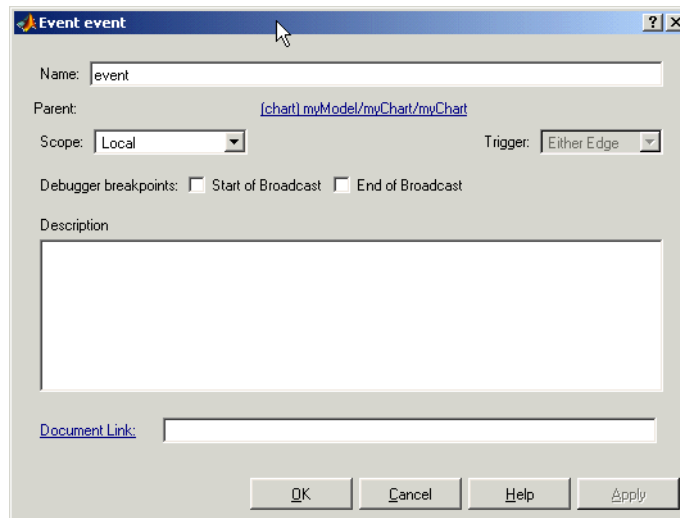
- When you add a new event from the Stateflow diagram editor, its **Event** dialog appears.

See “Adding Events in the Stateflow Diagram Editor” on page 6-3 for a description of how to add an event in the Stateflow diagram editor.

- In the **Contents** pane (right pane) of the **Model Explorer** window, right-click an entry row for a new or existing event and select **Properties** from the resulting pop-up menu.

See “Adding Events in the Model Explorer” on page 6-4 for a description of how to open the **Model Explorer** and add an event.

The **Event** dialog for an event has the following appearance:



Name

Name of this event. Event names enable actions to reference specific events. You assign a name to an event by setting its Name property. You can assign any name that begins with an alphabetic character, does not include spaces, and is not shared by sibling events.

Parent

Clicking this read-only field displays the parent of this event in the Stateflow editor. The parent is the object in which this event is visible. When an event is triggered, Stateflow broadcasts the event to the parent and all the parent's descendants. The parent for an event can be a Stateflow machine, a chart, or a state. You specify the parent for an event when you add it to the data dictionary (see “Adding Events” on page 6-3).

Scope

Scope of this event. The scope specifies where the event occurs relative to its parent. The following sections describe each scope setting.

Local. A local event is an event that can occur anywhere in a Stateflow machine but is visible only in its parent (and its parent's descendants).

Input from Simulink. This event occurs in one Simulink block and is broadcast in another. The first block can be any type of Simulink block. The second block must be a Chart block. See “Defining Input Events” on page 6-11 for more information.

Output to Simulink. This event occurs in one Simulink block and is broadcast in another. The first block is a Chart block. The second block can be any type of Simulink block. See “Defining Output Events” on page 6-13 for more information.

Exported. An exported event is a Stateflow event that can be broadcast by external code built into a stand-alone or Real-Time Workshop target. See “Exporting Events to Stateflow External Code” on page 6-16 for more information.

Imported. An imported event is an externally defined event that can be broadcast by a Stateflow machine embedded in the external code. See “Importing Events from Stateflow External Code” on page 6-17 for more information.

Note If you copy a Stateflow Chart block from one Simulink model to another, all objects in the chart hierarchy are copied except those parented by the Stateflow machine. This means that events parented by the original Stateflow machine (local, imported, exported) are not copied to the new machine. You can, however, use the **Model Explorer** tool to copy individual events by clicking and dragging them from machine to machine.

Trigger

Type of signal that triggers an input or output event. See “Defining Input Events” on page 6-11 or “Defining Output Events” on page 6-13 for more information.

Index

Index of the input signal that triggers this event. This option applies only to input events and appears when you select Input from Simulink as the scope of this event. See “Associating Input Events with Control Signals” on page 6-13 for more information.

Port

Index of the port that outputs this event. This property applies only to output events and appears when you select Output to Simulink as the scope of this event. See “Associating an Output Event with an Output Port” on page 6-15 for more information.

Description

Description of this event. Stateflow allows you to store brief descriptions of events in the data dictionary. To describe a particular event, enter its description in this field.

Document Link

Stateflow allows you to provide online documentation for events defined by a model. To document a particular event, set its `Documentation` property to a MATLAB expression that displays documentation in some suitable online format (for example, an HTML file or text in the MATLAB command window). Stateflow evaluates the expression when you click the event's documentation link (the blue text that reads "Document Link" displayed at the bottom of the event's **Event** dialog box).

Sharing Events with Simulink

Before a Stateflow block can send events or receive them from other Simulink blocks, you need to define them for the Stateflow chart. Use the following topics to define input and output events for a Stateflow chart:

- “Defining Input Events” on page 6-11 — Describes input events that allow other Simulink blocks, including other Stateflow blocks, to notify a particular chart of events that occur outside it.
- “Associating Input Events with Control Signals” on page 6-13 — Tells you how to associate multiple control signals with a single trigger port for the Stateflow block.
- “Defining Output Events” on page 6-13 — Describes output events that allow a chart to notify other blocks in a model of occurrences in that chart.
- “Associating an Output Event with an Output Port” on page 6-15 — Tells you how to position the output ports for multiple output events.

Defining Input Events

An input event occurs outside a chart and is visible only in that chart. This type of event allows other Simulink blocks, including other Stateflow blocks, to notify a particular chart of events that occur outside it. To define an event as an input event, set its **Scope** property to **Input from Simulink**.

The following steps describe how to add an input event to a chart in the **Model Explorer**:

- 1 Add an event to the Stateflow chart.

You must add an input event to the chart and not to one of its objects. Adding events with the Stateflow diagram editor adds them to the chart open in the diagram editor. In the **Model Explorer**, you can add an event to any object. This means that you must select a chart object in the left **Model Hierarchy** pane before adding an input event. See “Adding Events” on page 6-3 for a complete description of both ways to add an event.

- 2 Set the **Scope** property for the event to **Input** (short for **Input from Simulink**).

A single trigger port is added to the top of the Stateflow block.

- 3** If you want a Simulink block to trigger the Stateflow block through this input event, do one of the following:
- To trigger the Stateflow block through a change in control signal, set the **Trigger** property to one of the following edge triggers:

Trigger Type	Description
Rising Edge	Rising edge trigger, where the control signal changes from either 0 or a negative value to a positive value.
Falling Edge	Falling edge trigger, where the control signal changes from either 0 or a positive value to a negative value.
Either Edge	Either rising or falling edge trigger.

In all cases, the signal must cross 0 to constitute a valid edge trigger. For example, a change from -1 to 1 constitutes a valid rising edge, but a change from 1 to 2 does not.

An edge trigger triggers execution of a subsystem at the beginning of the next simulation time step, regardless of when triggering actually occurred during the previous time step. If it is not critical that the subsystem be executed immediately, you can define the events as edge-triggered.

- To trigger the Stateflow block through a Simulink block that outputs function-call events, set the **Trigger** property to **Function Call**.

In this case, Stateflow changes all other input events for the Chart to **Function Call**.

If you choose to add an event in the **Model Explorer**, you can change its **Name**, **Scope**, **Trigger** and **Port** properties directly in the **Model Explorer**. See “Setting Properties for Stateflow Objects in the Model Explorer” on page 14-8 for a description.

Associating Input Events with Control Signals

You can define multiple input events for a chart. When you define one or more input events for a chart, Stateflow adds a single trigger port to the chart block. External Simulink blocks can trigger the chart's input events via a signal or vector of signals connected to the chart's single trigger port by associating input events with control signals.

When defining input events for a chart, you must specify how control signals connected to the chart trigger the input events. An input event's **Index** property associates the event with a specific element of a control signal vector connected to the trigger port of the chart that parents the event. The first element of the signal vector triggers the input event whose index is 1; the second, the event whose index is 2; and so on. Stateflow assigns 1 as the index of the first input event that you define for a chart, 2 as the index of the second event, and so on. You can change the default association for an event by setting the event's **Index** property to the index of the signal that you want to trigger the event.

Input events occur in ascending order of their indexes when more than one such event occurs during update of a chart (see "Setting the Stateflow Block Update Method" on page 9-11). For example, suppose that when defining input events for a chart, you assign the indexes 3, 2, and 1 to events named A, B, and C, respectively. Now, suppose that, during simulation of the model containing the chart, that events A and C occur in a particular update. Then, in this case, the order of occurrence of the events is C first followed by A.

Defining Output Events

An output event is an event that occurs in a specific chart and is visible in specific Simulink blocks outside the chart. This type of event allows a chart to notify other blocks in a model of events that occur in the chart. To define an event as an output event, set its Scope property to Output to Simulink. You can define multiple output events for a given chart. Stateflow creates a chart output port for each output event that you define (see "Port" on page 6-9). Your model can use the output ports to trigger the output events in other Simulink blocks in the same model.

The following steps describe how to add and define the necessary fields for an event output to Simulink:

1 Add an event to the Stateflow chart.

You must add the event to the chart and not to any other object in the chart. Adding events with the Stateflow diagram editor adds them only to the chart open in the diagram editor. In the **Model Explorer**, you must select the chart object in the left **Model Hierarchy** pane before adding the event. See “Adding Events” on page 6-3 for a complete description of both ways to add an event.

2 Set the **Scope** field to **Output to Simulink**.

When you define an event to be **Output to Simulink**, a Simulink output port is added to the Stateflow block. Output events from the Stateflow block to the Simulink model are scalar.

3 If you want this chart to call a subsystem, do one of the following:

- To call an edge-triggered subsystem, set the **Trigger** property of the output event to **Either Edge**.

See “Defining Edge-Triggered Output Events” on page 9-20 for an example of a Stateflow block calling an edge-triggered subsystem in Simulink.

- To call a function-call subsystem, set the **Trigger** property of the output event to **Function Call**.

Function-call triggers call function-call subsystems the moment the call is made, even if it is made in the middle of a time step. For an example of a Stateflow block calling a function-call subsystem, see “Defining Function Call Output Events” on page 9-17.

If you choose to add an event in the **Model Explorer**, you can change its **Name**, **Scope**, **Trigger** and **Port** properties directly in the **Model Explorer**. See “Setting Properties for Stateflow Objects in the Model Explorer” on page 14-8 for a description.

Associating an Output Event with an Output Port

The **Port** property for an output event associates it with an output port on the chart block that parents the event. It specifies the position of the output port relative to other output event ports on the Chart block. Output event ports appear below output data ports on the right side of a chart block.

Stateflow numbers all output ports sequentially from top to bottom, starting with output data followed by output events. As you add output events, their default **Port** property is assigned sequentially to the end of the existing port list.

You can change the default port assignment of an event by resetting its **Port** property. When you change the **Port** property for an output event, the **Port** property for the remaining output events is automatically renumbered. For example, assume that there are three output events, OE1, OE2, and OE3, associated with the output ports 4, 5, and 6, respectively. If you change the **Port** property for OE2 to 6, the Port property for the remaining events, OE1 and OE3, is renumbered to 4 and 5, respectively.

Sharing Events with Stateflow External Code

Stateflow allows external code defined for the Stateflow machine to send exported events to trigger Stateflow diagrams in the model. It allows external code to receive imported events from other charts to trigger parts of external code. Use the following topics to learn how to export and import events for external code in Stateflow.

- “Exporting Events to Stateflow External Code” on page 6-16 — Describes how to export events that enable external code to trigger events in the Stateflow machine.
- “Importing Events from Stateflow External Code” on page 6-17 — Describes how to import events that allow a Stateflow machine built into a stand-alone or Real-Time Workshop target to trigger an event in external code.

Exporting Events to Stateflow External Code

Stateflow allows a Stateflow machine to export events. Exporting events enables external code to trigger events in the Stateflow machine.

Exported events can be parented only by a Stateflow machine. To export an event, add the event to the Stateflow machine and set its **Scope** property to Exported.

Note You must use the **Model Explorer** to add exported events to a Stateflow machine. For information on adding events in the **Model Explorer**, see “Adding Events in the Model Explorer” on page 6-4.

The Stateflow code generator generates a function for each exported event. The C prototype for the exported event function has the form

```
void external_broadcast_EVENT()
```

where `EVENT` is the name of the exported event. External code built into a target can trigger the event by invoking the event function. For example, suppose you define an exported event named `switch_on`. External code can trigger this event by invoking the generated function `external_broadcast_trigger_on`. See “Exported Events” on page 9-25 for an example of how to trigger an exported event.

See “Exported Events” on page 9-25 for an example of a Stateflow event exported to Stateflow external code.

Importing Events from Stateflow External Code

A Stateflow machine serves as a surrogate parent for imported events defined by external code. Importing an event allows Stateflow to build a custom or Real-Time Workshop (RTW) target, that triggers the imported event in external code.

To import an event, add an event to the Stateflow machine that needs to trigger the event and set its **Scope** property to Imported.

Note You must use the **Model Explorer** to add imported events to the Stateflow machine. For information on adding events in the **Model Explorer**, see “Adding Events in the Model Explorer” on page 6-4.

Stateflow assumes that external code defines each imported event as a function whose prototype is of the form

```
void external_broadcast_EVENT
```

where EVENT is the Stateflow name of the imported event. For example, suppose that the Stateflow machine imports an external event named `switch_on`. In this case, Stateflow assumes that external code defines a function named `external_broadcast_switch_on` that broadcasts the event to external code. Later, when you build a target for the Stateflow machine, the Stateflow code generator encodes actions that signal imported events as calls to the corresponding external broadcast event functions defined by the external code.

See “Imported Events” on page 9-27 for an example of a Stateflow external code event imported into Stateflow.

Defining Implicit Events

Stateflow defines and triggers the following events that typically occur whenever a chart executes:

- Chart waking up
- Entry into a state
- Exit from a state
- Value assigned to an internal (noninput) data object

These events are called *implicit events* because you do not have to define or trigger them explicitly. Implicit events are children of the chart in which they occur. Thus, they are visible only in the charts in which they occur.

Use the following topics to understand implicit events:

- “Referencing Implicit Events” on page 6-18
- “Example of an Implicit Event” on page 6-19

Referencing Implicit Events

Action expressions can use the following syntax to reference implicit events.

```
event(object)
```

where *event* is the name of the implicit event and *object* is the state or data in which the event occurred.

Each of the following keywords generates implicit events in the action language notation for states and transitions.

Implicit Event	Meaning
<code>change(data_name)</code> or <code>chg(data_name)</code>	Specifies and implicitly generates a local event when the value of <code>data_name</code> changes.
<code>enter(state_name)</code> <code>en(state_name)</code>	Specifies and implicitly generates a local event when the specified <code>state_name</code> is entered.
<code>exit(state_name)</code> <code>ex(state_name)</code>	Specifies and implicitly generates a local event when the specified <code>state_name</code> is exited.

Implicit Event	Meaning
tick	Same as wakeup keyword.
wakeup	Specifies and implicitly generates a local event when the chart of the action being evaluated awakens.

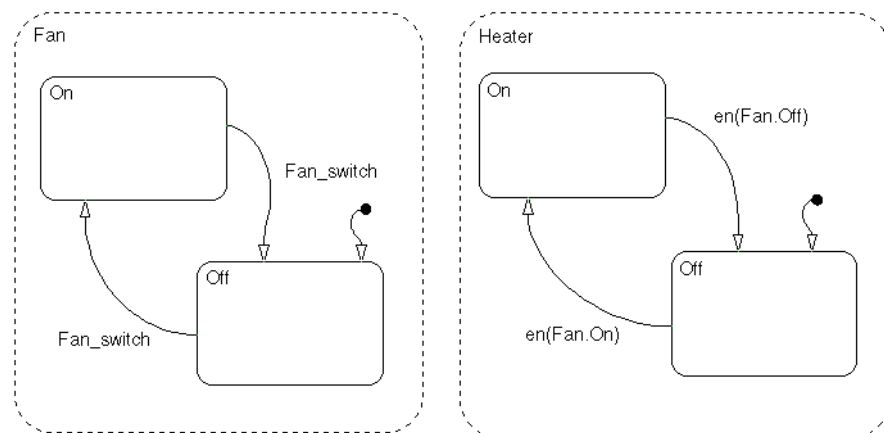
If more than one object has the same name, the event reference must qualify the object's name with that of its ancestor. The following are some examples of valid implicit event references.

```
enter(switch_on)
en(switch_on)
change(engine.rpm)
```

Note The wakeup (or tick) event always refers to the chart of the action being evaluated. It cannot reference a different chart by argument.

Example of an Implicit Event

This example illustrates use of an implicit enter event.



Fan and Heater are parallel (AND) superstates. By default, the first time the Stateflow diagram is awakened by an event, the states Fan.Off and Heater.Off become active. The first time event Fan_switch occurs, the transition from Fan.Off to Fan.On occurs. When Fan.On's entry action executes, an implicit local event is broadcast (i.e., `en(Fan.On) == 1`). This event broadcast triggers the transition from Heater.Off to Heater.On (triggered by the condition `en(Fan.On)`). Similarly, when the system transitions from Fan.On to Fan.Off and the implicit local event Fan.Off is broadcast, the transition from Heater.On to Heater.Off is triggered.

Adding Data

Stateflow can store and retrieve data that resides internally in its own workspace. It can also access data that resides externally in the Simulink model or application that embeds the Stateflow chart. When creating a Stateflow model, define any internal or external data referenced by Stateflow actions as described in the following topics:

- “Adding Data to the Data Dictionary” on page 6-21 — Describes how to add a data object to any object in a Stateflow chart, to the chart itself, or to the Stateflow machine.
- “Defining Temporary Data” on page 6-24 — Describes temporary data that you can only define only a function.
- “Sharing Input and Output Data with Simulink” on page 6-35 — Describes how to specify data input from and output to Simulink in a Stateflow diagram.

Adding Data to the Data Dictionary

You can use either the Stateflow editor or Explorer to add data that is accessible only in a specific chart. You must use the Stateflow Explorer to add data that is accessible everywhere in a Stateflow chart or in a specific object such as a state.

Adding Data in the Stateflow Diagram Editor

If you add events in the Stateflow diagram editor, they are parented by the Stateflow diagram and are visible to all objects in the chart. To add an event in the Stateflow diagram editor, do the following:

- 1 From the **Add** menu of the Stateflow editor, select **Event**.

The Data properties dialog appears.

- 2 In the resulting pop-up menu, select one of the following scopes for this data:
 - **Local** — Data is defined for use in this Stateflow diagram only.
 - **Constant** — Data is defined as a constant.
 - **Input from Simulink** — The Simulink model provides the value for this data by an input port to the Stateflow block for this diagram.

- **Output to Simulink** — Stateflow provides this data to the Simulink model by an output port on the Stateflow block for this diagram.
- **Parameter** — Data is defined as a constant whose value is taken from a parameter for the parent masked Simulink subsystem or from the MATLAB base workspace variable of the same name. See “The Mask Editor” in Simulink documentation for information on how to assign a parameter to a masked subsystem.

Stateflow adds a default definition of the new event to the Stateflow data dictionary and displays the **Data** dialog box.

- 3 Specify properties for the data specified in the **Data** dialog box.

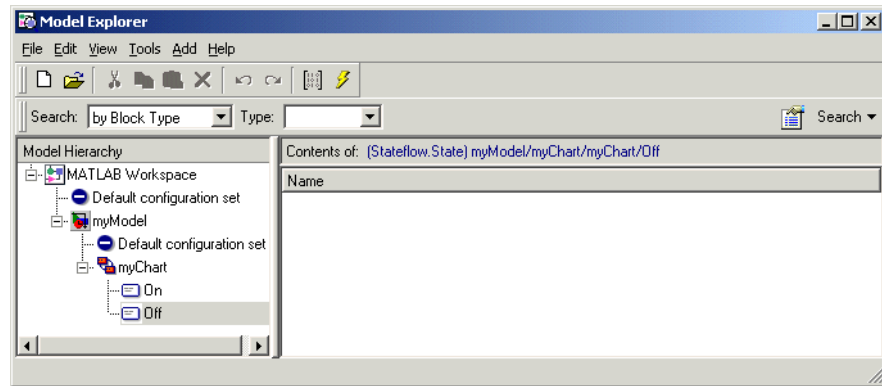
See “Setting Data Properties in the Data Dialog” on page 6-25 for a description of the property fields of the **Data** dialog box.

Adding Data in the Model Explorer

If you add events in the **Model Explorer**, you can choose any Stateflow object to parent the new event. This event is visible to the parent object and all child objects of the parent. To add an event in the **Model Explorer**, do the following:

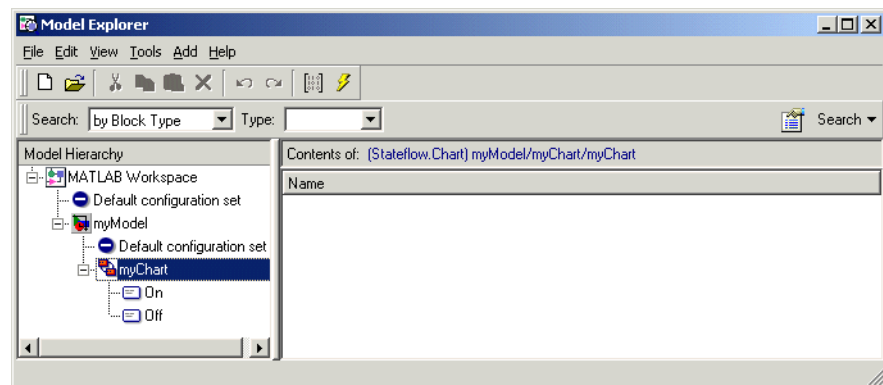
- 1 Select **Explore** from the Stateflow editor’s **Tools** menu.

The **Model Explorer** opens.



If no object is selected, the current diagram or subchart is highlighted in the **Model Hierarchy** pane of the **Model Explorer**. Otherwise, the selected object is highlighted. In the preceding example, state **Off** is selected in the Stateflow diagram **myChart** when the **Model Explorer** is called.

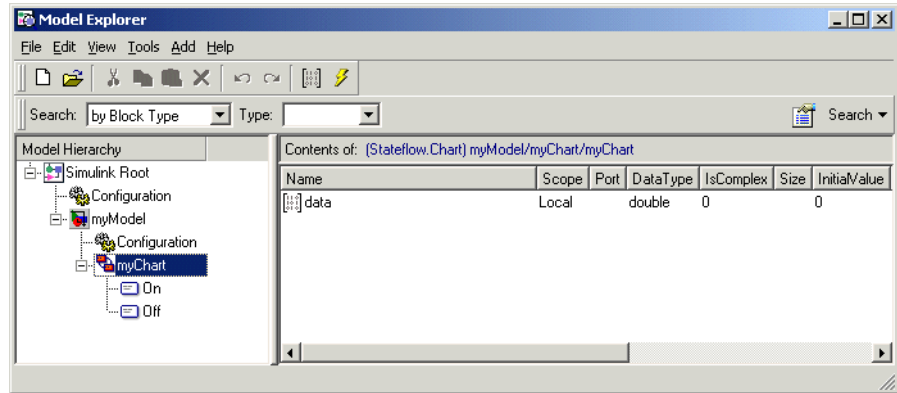
- 2 Select the object (machine, chart, or state) in the object hierarchy pane where you want the new data to be visible.



In the preceding example, the Stateflow chart **myChart** is selected to receive the new event.

3 From the **Add** menu of the **Model Explorer**, select **Data**.

Stateflow adds a default definition for the new data in the data dictionary and displays an entry row for the new data in the Explorer's Contents pane.



4 Change the properties of the event in one of the following ways:

- To change the **Name**, **Scope**, **Trigger Type**, or **Port** property for the event, click individual entries in the entry row in the **Model Explorer**.
- To change any or all properties for the event, right-click the event row and select **Properties** to access the **Data** dialog for the event.

See “Setting Data Properties in the Data Dialog” on page 6-25 for a description of each property for an event.

Defining Temporary Data

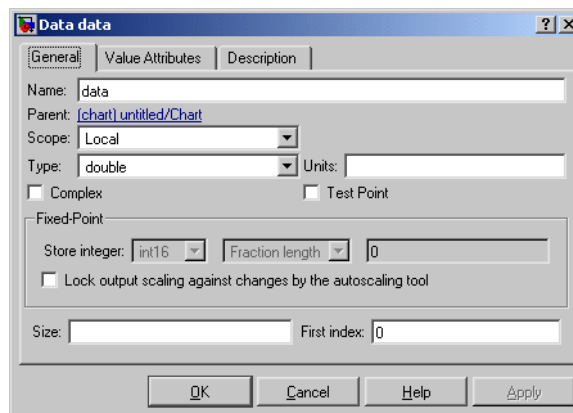
Stateflow allows functions (graphical, truth table, and Embedded MATLAB) to define temporary data that persists only as long as the function is active. Only the function can access its temporary data. For example, defining a loop counter to have **Temporary** scope might be a good idea if it is used only as a temporary counter that does not need to persist after the function completes.

Setting Data Properties in the Data Dialog

You set data properties in the **Data** dialog when you

- Add new data in the Stateflow Diagram editor
 - In the Stateflow diagram editor, from the **Add** menu, select **Data**, and, in the resulting pop-up menu, select a scope for the data. The **Data** dialog appears. See “Adding Data in the Stateflow Diagram Editor” on page 6-21 for more details.
- Change the definition of new or existing data in the **Model Explorer**
 - See “Adding Data in the Model Explorer” on page 6-22 for details on how to create data in the **Model Explorer**. Once you select the new or existing data in the **Contents** pane in the **Model Explorer**, you can see the **Data** dialog.
 - In the far right **Properties** pane (if not, from the **View** menu select **Show Dialog View**).
 - When you right-click the data in the **Contents** pane and select **Properties**.

The **Data** dialog appears with the **General** page tabbed to the forefront.

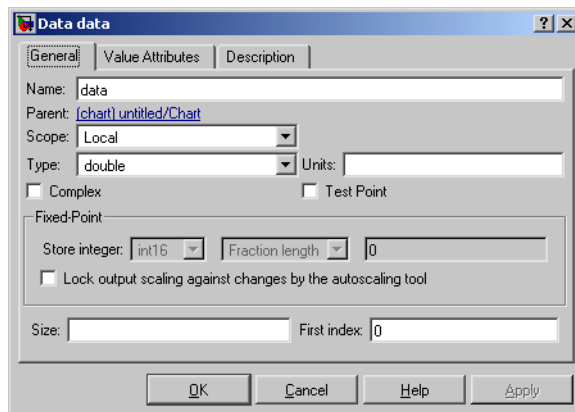


Each tabbed page of the Data properties dialog is discussed in the following topics:

- “Setting General Data Properties” on page 6-26 — Define the type of data.
- “Setting Value Attribute Data Properties” on page 6-30 — Set the value of the data.
- “Setting Description Data Properties” on page 6-33 — Enter a description for the data.

Setting General Data Properties

The window tabbed **General** in the **Data** dialog has the following appearance.



It includes the property settings described in the following topics.

Name Property

A data name can be of any length and can consist of any alphanumeric and special character combination, with the exception of embedded spaces. The name cannot begin with a numeric character.

Parent Property

You specify the parent of a data item when you add the item to the data dictionary. The parent object for a data determines the objects that can access it. Only the parent of the data and its can access the item. This is a read-only field. Click it to display the parent Stateflow object in the Stateflow diagram editor.

Scope Property

The scope for a data object specifies where it resides in memory relative to its parent. It can be set to one of the following values, depending on its parent:

Local. A local data object can be accessed only by its parent Stateflow object.

Input from Simulink. This is a data parented by a Stateflow diagram. The diagram reads the value of this data from the Simulink model with an input port associated with the data item on the Chart block. See “Importing Data from External Stateflow Code” on page 6-43 for more information.

Output to Simulink. This data is parented by the Stateflow diagram. The diagram sends the value of this data to the Simulink model with an output port associated with the data item on the Chart block. See “Sharing Input and Output Data with Simulink” on page 6-35 for more information.

Temporary. A temporary data object exists only while its parent function (graphical, truth table, embedded MATLAB) is executing. See “Defining Temporary Data” on page 6-24 for more information.

Constant. Constant data is read-only data for its parent Stateflow object and its children. It retains the initial value set in its **Data** properties dialog box with the **Initialize from** field in the window tabbed **Value Attributes**.

Exported. Exported data is data present in the Simulink model that is made available to external code defined in Stateflow. Exporting data enables external code to access this data as well. See “Exporting Data to External Stateflow Code” on page 6-42 for more information.

Imported. Imported data is data parented by the Simulink model that is defined by external code embedded in the Stateflow machine. Importing externally defined data enables the Stateflow machine to access this data as well. See “Importing Data from External Stateflow Code” on page 6-43 for more information.

Input. This scope applies to argument values for a function (graphical, truth table, and embedded MATLAB).

Output. This scope applies to the return value of a function (graphical, truth table, and embedded MATLAB).

Note Copying a Stateflow Chart block from one Simulink model to another copies all objects in the chart hierarchy, including data and events. Data parented by the original Simulink model accessed in the Stateflow chart is not copied to the new model. Use the **Model Explorer** to copy this data to the new model.

Type Property

The type of the data specifies its bit size and how its value is represented. You can select from a list of provided types or enter an expression of type. See “Typing Stateflow Data” on page 6-45 for an explanation on each type of entry.

Fixed-Point Properties

You select an integer base for the fixed-point data in the **Stored Integer** field. The scaling type for the fixed-point number is entered in the drop-down field below which can take one of the two selectable values, **Fraction Length** or **Scaling**. The scaling value is entered in the field to the right. For a detailed description of fixed-point data, see “What Is Fixed-Point Data?” on page 8-2. See also “Specifying Fixed-Point Data in Stateflow” on page 8-7.

Stored Integer. Select the integer word type to hold the fixed-point data quantized integer. The size of the word in bits is indicated by a suffix of 8, 16, or 32. Larger word sizes can represent large quantities with greater precision than smaller word sizes. Unsigned types (uint8, uint16, uint32) can represent only positive quantities. Signed types (int8, int16, int32) can represent negative quantities.

Fraction length. Select this value to interpret the entry in the right adjacent field as the binary point location for binary-point-only scaling. A positive integer entry moves the binary point left of the rightmost bit by that amount. For example, an entry of 2 sets the binary point in front of the second bit from the right. A negative integer entry moves the binary point further right of the rightmost bit by that amount. Only integers can be entered (see note below).

Scaling. Select this value to interpret the entry in the right adjacent field as scaling with separate slope and bias entered in [Slope Bias] format (can be entered with or without brackets []). For example, the entry [2 3] sets the slope to 2 and the bias to 3. You can also enter the slope or bias as an arithmetic expression, for example, [1.2*2^-2 3.4]. If you enter a single value, it is interpreted as the slope. The value entered for the slope can be any positive real number. The value for the bias can be any real number. See following note.

Note Values entered for **Fraction length** or **Scaling** that exceed the ranges specified in the previous sections are flagged immediately with red text.

Lock output scaling against changes by the autoscaling tool. Simulink autoscales Stateflow fixed-point data with the Simulink autoscaling tool. Selecting this option prevents Simulink from replacing the current fixed-point type with a Simulink chosen type in the autoscaling tool. See “Automatic Scaling” in Fixed-Point Blockset documentation for instructions on autoscaling fixed-point data in Simulink.

Port

Index of the port associated with this data item (see “Sharing Input and Output Data with Simulink” on page 6-35). This control applies only to input and output data.

Units

Units, for example, inches, centimeters, and so on, represented by this data item. The value of this field is stored with the item in the Stateflow machine’s data dictionary.

Size

Size of this data. The value of this property can be a scalar or a MATLAB vector of values. If it is a scalar (for example, 5), it specifies the data as a one-dimensional array (that is, a vector) of that size. If it is a MATLAB vector (for example, [2 3 7]), it specifies the data as a multidimensional array. In this case, the number of dimensions is equal to the length of the vector, and the size of each dimension corresponds to the value of each element of the vector.

The allowed size of the data array that you specify depends on the **Scope** property of the data, as follows:

Scope	Scalar	Vector	Matrix 2-dim	Matrix n-dim
Constant	Yes	No	No	No
Input from Simulink/ Output to Simulink	Yes	Yes	Yes	No
Temporary/Local	Yes	Yes	Yes	Yes
Imported/Exported	Yes	Yes	Yes	Yes
Function input/ Function output)	Yes	Yes	Yes	Yes *

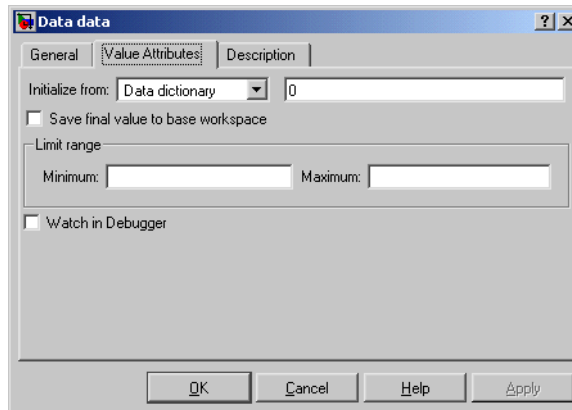
* Applies only to Graphical functions and Truth Tables. Embedded MATLAB functions can accept or return only 2-dimensional matrices.

First Index

Specifies the index of the first element of this array. For example, the first element index of a zero-based array is 0.

Setting Value Attribute Data Properties

The window tabbed **Value Attributes** in the **Data** dialog has the following appearance.



It includes the following settings:

Initialize from

Source of the initial value for this data item: either the Stateflow data dictionary or the MATLAB workspace.

Data Dictionary. If the source is the data dictionary, enter the initial value in the adjacent text field. Stateflow stores the value that you enter in the data dictionary.

MATLAB Workspace. If the source is the MATLAB workspace, this item gets its initial value from a similarly named variable in the MATLAB workspace. For example, suppose that the name of a Stateflow data item is A. If the MATLAB workspace has a variable named A, its value is used to initialize A in Stateflow.

If the initialized data item is an array, Stateflow sets each element of the Stateflow array to the value of the same element of the MATLAB array. Also, Stateflow vectors that you specify in Stateflow with one dimension are compatible with MATLAB row, and column vectors of the same size. For example, a Stateflow vector of entered size 5 is compatible with a MATLAB row vector of size [1, 5], or column vector of size [5, 1].

The following table summarizes the time of initialization for each data scope.

Data Parent	Scope	When Initialized
Machine	Local	Start of simulation
	Exported	Start of simulation
	Imported	Not applicable
Chart	Input	Not applicable
	Output Local	Start of simulation or when chart is reinitialized as part of an enabled Simulink subsystem
State with History Junction	Local	Start of simulation or when chart is reinitialized as part of an enabled Simulink subsystem
State without History Junction	Local	State activation (state entered)
Function (graphical, truth table, and embedded MATLAB)	Input	When function is invoked
	Output	When function is invoked
	Local	Start of simulation or when chart is reinitialized as part of an enabled Simulink subsystem

Note You can also use the Stateflow Explorer to set the **Initialize from** field.

Save final value to base workspace

Selecting this option causes the value of the data item to be assigned to a similarly named variable in the model's base workspace at the end of simulation.

Limit Range

This control group specifies values used by a Stateflow machine to check the validity of this data item. It includes the next two controls.

Minimum. Minimum value that this data item can have during execution or simulation of the Stateflow machine it belongs to.

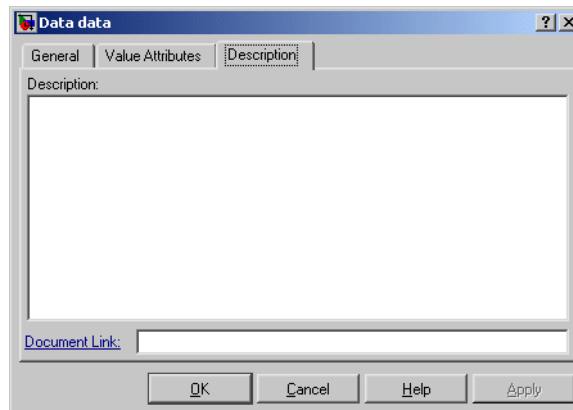
Maximum. Maximum value that this data item can have during execution or simulation of the Stateflow machine it belongs to.

Watch in debugger

If selected, this option causes the debugger to halt if this data item is modified.

Setting Description Data Properties

The window tabbed **Description** in the **Data** dialog has the following appearance:



It includes the following settings:

Description

Description of this data item.

Document link

Stateflow allows you to provide online documentation for data defined by a model. To document a particular item of data, set this property to a MATLAB expression that displays documentation in some suitable online format (for example, an HTML file or text in the MATLAB Command Window). Stateflow evaluates the expression when you click the item's documentation link (the blue text that reads Document Link displayed at the bottom of the event's **Data** dialog box).

Sharing Stateflow Data with Simulink and MATLAB

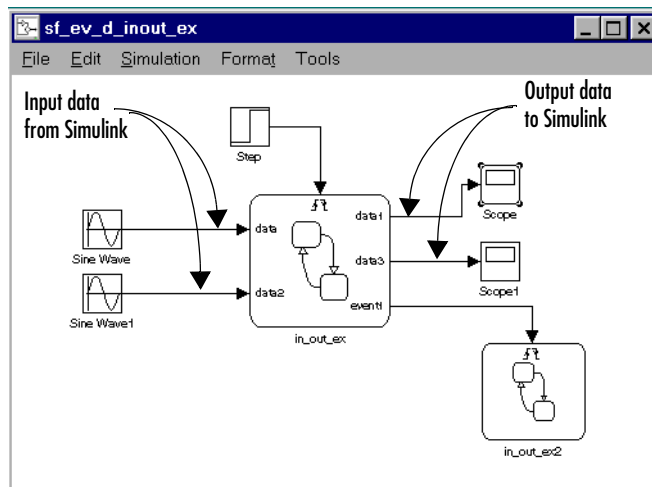
Stateflow charts can receive input data from a Simulink input signal or a Simulink parameter defined for a masked subsystem. Stateflow charts can also send output data through a Simulink output signal. Use the following topics to understand how Stateflow shares data with Simulink:

- “Sharing Input and Output Data with Simulink” on page 6-35
- “Resolving Signal Objects for Output Data” on page 6-38
- “Bringing Simulink Parameters into Stateflow” on page 6-39
- “Initializing Data from the MATLAB Base Workspace” on page 6-40
- “Saving Data to the MATLAB Workspace” on page 6-41

Sharing Input and Output Data with Simulink

In “Setting Data Properties in the Data Dialog” on page 6-25, you define data in its properties dialog. In this topic, you continue to use the properties dialog to define Stateflow data that is input from or output to the Simulink model.

Input data is data that a Simulink model supplies to a chart with input ports on the chart block. Output data is data that a chart supplies to other blocks via its output ports. The following example shows a Stateflow chart block with input and output data ports connected to the Simulink model.



To define data as input from or output to Simulink, do the following:

- 1 Add a data object to the chart using the Stateflow diagram editor or Stateflow Explorer (see “Adding Data to the Data Dictionary” on page 6-21).

You must add the data to the chart and not to any other object in the chart. Adding a data object with the Stateflow diagram editor adds it only to the chart that is visible in the diagram editor. If you add the data in the **Model Explorer**, you must select (highlight) the chart object in the **Model Hierarchy** pane of the **Model Explorer** before adding the event.

- 2 In the **Model Explorer**, set the property **Scope** to one of the following:
 - **Input** — Short for **Input from Simulink** for the **Scope** property in the **Data** dialog.
 - **Output** — Short for **Output to Simulink** for the **Scope** property in the **Data** dialog.

When you define data to be **Input to Simulink**, a Simulink input port is added to the Stateflow block. When you define a data object to be **Output to Simulink**, a Simulink output port is added to the Stateflow block.

By default, Stateflow associates the first input port with the first input item you define, the second input port with the second input item, and so on. The same applies to output ports. Both the **Model Explorer** and the **Data** dialog show the current port assignment in the **Port** field. You can alter this assignment by editing the value displayed in the **Port** field.

- 3 Set the type of the input or output data.

You use the **Type** property for input or output data, to size it directly by an expression or specify that Stateflow inherit the type from Simulink. For more information on defining data types in Stateflow, see “Typing Stateflow Data” on page 6-45.

If the **Use Strong Data Typing with Simulink I/O** option for the chart is selected (see “Specifying Chart Properties” on page 9-4), the following apply:

- Input signals from Simulink must match the specified type, or a mismatch error occurs.

- Output signals from the chart to Simulink are of the same data type as the output data in Stateflow.

If the **Use Strong Data Typing with Simulink I/O** option is not selected for the chart, the following apply:

- Input signals must be of type `double`.
In this case, Stateflow converts the input value to the specified type.
- The Stateflow Chart block converts all output data to Simulink type `double`.

4 Set the size of the input or output data.

You use the **Size** property for input or output data to size it directly by a mathematical expression or specify that Stateflow inherit the size from Simulink. For more information on defining data sizes in Stateflow, see “Sizing Stateflow Data” on page 6-50.

If you specify the size of input or output data directly by a mathematical expression, you can set it to be a scalar or a matrix (two-dimensional, row-by-column array). The size of the Simulink signal connected to a Stateflow input port must match the size of the Stateflow data associated with the port. The size of Stateflow data associated with an output port must match the size expected by the destination block in Simulink it is connected to.

Data arrays you specify with only one dimension are compatible with Simulink row or column vectors of the same size. For example, if Stateflow defines input or output data of size 3, it is compatible with a Simulink row vector of size `[1,3]`, or column vector of size `[3,1]`.

Note You cannot type or size Stateflow input data to accept frame-based data from Simulink.

Resolving Signal Objects for Output Data

Stateflow blocks participate in signal resolution with Simulink signal objects, which are used to specify the attributes of any signal in Simulink. By default, if a Simulink signal object exists with the same name as an output data from a Stateflow block, the output data becomes associated with the signal object. This is referred to as *implicit signal resolution*. See Simulink documentation for a more detailed explanation of implicit signal resolution.

By default, implicit signal resolution of matching Stateflow output data receives a warning when the Simulink diagram is updated. Use the subtopics that follow to control implicit signal resolution and its warnings for the Stateflow output data signals in your model.

Eliminating the Warnings for Implicit Signal Resolution for the Model

If you want implicit signal resolution for all model signals, including Stateflow output data, but want to eliminate the warnings for all signals in the model, do the following:

- 1 In Simulink, from the **Simulation** menu, select **Configuration Parameters**.

The **Configuration Parameters** dialog appears.

- 2 In the left pane of the dialog, under the **Diagnostics** node, select the **Data Integrity** node.

The **Data Integrity** configuration parameters appear in the right pane of the **Configuration Parameters** dialog. By default, the **Signal resolution control** field is set to **Try resolve all signals & states (warn for implicit resolution)**.

- 3 For the **Signal resolution control** field, select **Try resolve all signals & states**.

Disabling Implicit Signal Resolution for a Stateflow Chart

If you enable implicit signal resolution for the model but want to eliminate it for an individual Stateflow chart, do the following:

- 1 Right-click the Stateflow block with output data that you do not want resolved.

- 2 From the resulting context menu select **Subsystem Parameters**.
- 3 In the resulting **Block Parameters** dialog, for the **Permit hierarchical resolution** field, select **ParametersOnly** or **None**.

Enabling Explicit Signal Resolution for an Individual Output Data Signal

If you want to force signal resolution for an individual output signal for a Stateflow block, do the following:

- 1 In Simulink, right-click the signal line connected to output data that you want to resolve.
- 2 From the resulting pop-up menu select **Signal Properties**.

The **Signal Properties** dialog appears.

- 3 In the **Signal Properties** dialog, enter a name for the signal corresponding to its signal object.
- 4 Select the **Signal name must resolve to Simulink signal object** check box and select **OK** to save and apply your settings and close the **Signal Properties** dialog.

Bringing Simulink Parameters into Stateflow

You might want to use Simulink parameters for a parent masked subsystem in a Stateflow diagram. To bring these parameters into Stateflow diagrams, do the following:

- 1 In the Simulink mask editor for the parent subsystem, define and initialize a Simulink parameter.
- 2 In the Stateflow diagram for the Stateflow block in the masked subsystem, define data with the same name as the parameter.
- 3 Select the scope **Parameter** for the data.

Data of scope **Parameter** is defined as a constant in the Stateflow diagram workspace. It cannot be changed during application execution. This scope is primarily designed for Stateflow developers who want to use Simulink

parameters with Stateflow blocks and maintain consistency with the Simulink model.

Note Stateflow data of scope **Parameter** is not a Stateflow parameter. It is a mechanism for bringing Simulink parameters into a Stateflow diagram.

When simulation starts, the data of scope **Parameter** is resolved at the lowest level by a parameter with the name of the Stateflow data defined for the masked subsystem containing the Stateflow block. If the parameter of a masked subsystem parent does not resolve the value of the data, it continues to be resolved by higher level masked subsystems.

Initializing Data from the MATLAB Base Workspace

Stateflow lets you initialize data from the MATLAB base workspace. Initializing data from the MATLAB base workspace requires that you define data in both the MATLAB base workspace and Stateflow as follows:

- 1** In Stateflow, define data that you want to initialize from the MATLAB workspace with a scope of **Local** or **Output**.
- 2** In the **Data** properties dialog for the data, in the **Value Attributes** page, for the **Initialize from** field, select **Workspace**.

You can also set this data property in the **Contents** pane of the **Model Explorer** as follows:

- a** Click the row for the data to highlight it.
 - b** In the highlighted row click the 0 value under the **InitFromWorkspace** field.
A check box appears in place of the 0 value.
 - c** Click the check box to check it and click outside the row to see a value of 1 replace the check box.
- 3** Define a variable of the same name in the MATLAB base workspace.

When simulation starts, the data is resolved.

Saving Data to the MATLAB Workspace

For data of all scopes except **Constant** and **Parameter**, you can have Stateflow save its final value at the end of simulation in the MATLAB base workspace (not as a masked subsystem parameter) by doing one of the following:

- In the **Data** properties dialog, in the **Value Attributes** panel, for the **Initialize from** field, select **Save final value to base workspace**.
- In the **Model Explorer**, in the middle **Contents** pane, do the following:
 - Select the row for the data to highlight it.
 - Click the 0 value under the **SaveToWorkspace** field to see a check box materialize in its place.
 - Click the check box to check it.
 - Click outside the row to see a value of 1 replace the check box.

Sharing Data with Stateflow External Code

The Stateflow machine is a Stateflow object that parents all Stateflow charts in a model. It includes all Stateflow charts in the model along with external code that you include with a target for the model. You can import and export Stateflow data to share it with any other chart in the Stateflow machine or with Stateflow external code assigned to the machine. Use the following topics to learn how to export and import data in Stateflow across the charts and external code in a Stateflow machine.

- “Exporting Data to External Stateflow Code” on page 6-42 — Describes exported data that enables external code and any object in the Stateflow machine to access the data
- “Importing Data from External Stateflow Code” on page 6-43 — Describes imported data that allows a chart in the Stateflow machine to import definitions of data defined by external code that embeds the Stateflow machine

Exporting Data to External Stateflow Code

A Stateflow machine can export definitions of Stateflow machine data to external code that embeds the machine. Exporting data enables external code and the Stateflow machine to access the exported data. This means that external data is visible to every Stateflow diagram in the Simulink model of the Stateflow machine.

To export data from the Stateflow machine to externally defined code, do the following:

- 1** In the **Model Explorer**, add a data to the Simulink model.
- 2** Set the **Scope** of the data to **Exported**.

Note You can add exported data to the Stateflow machine only in the **Model Explorer**, by adding it to a Simulink model.

For each exported data item, the Stateflow code generator generates a C declaration of the form

```
type data;
```

where `type` is the C type of the exported item (for example, `int16`, `double`, and so on) and `data` is the item's Stateflow name. For example, suppose that your Stateflow machine defines an exported `int16` item named `counter`. The Stateflow code generator exports the item as the C declaration

```
int16_T counter;
```

where `int16_T` is a defined type for `int16` integers in Stateflow.

The code generator includes declarations for exported data in the generated target's global header file, thereby making the declarations visible to external code compiled into or linked to the target.

See “Exported Data” on page 9-29 for an example of Stateflow data exported to Stateflow external code.

Importing Data from External Stateflow Code

A Stateflow machine can import definitions of data defined by external code that embeds the Stateflow machine. Importing externally defined data enables a Stateflow machine to access data defined by the system in which it is embedded.

To import externally defined data into the Stateflow machine, do the following:

- 1 In the **Model Explorer**, add a data to the Simulink model.
- 2 Set the **Scope** property of the data to Imported.

Note You can add imported data to the Stateflow machine only in the **Model Explorer**, by adding it to a Simulink model.

For each imported item, the Stateflow code generator assumes that external code provides a prototype of the form

```
type data;
```

where `type` is the C data type corresponding to the Stateflow data type of the imported item (for example, `int` for Integer, `double` for Double, and so on) and `data` is the item's Stateflow name. For example, suppose that your Stateflow machine defines an imported `int32` integer named `counter`. The Stateflow code generator expects the item to be defined in the external C code as

```
int counter;
```

See “Imported Data” on page 9-31 for an example of Stateflow external code data imported into Stateflow.

Typing Stateflow Data

You specify the type of Stateflow data in the **Type** field of the **Data** properties dialog. You can access this dialog when you create data in Stateflow or when you access the **Model Explorer** for existing data. See “Adding Data” on page 6-21 for details on creating and defining data in Stateflow. In the **Model Explorer**, you can also specify it in the middle **Contents** pane by clicking the cell in the **Type** column of the highlighted variable row and editing it.

Use the following topics to decide how to type your data in Stateflow:

- “Selecting Stateflow Data Types” on page 6-45 — Select one of a list of supported data types in Stateflow.
- “Inheriting Input and Output Data Type from Simulink” on page 6-46 — Have Stateflow use Simulink signals to type input and output data.
- “Typing Data by Using the Type of Other Data” on page 6-48 — Specify the data type with the type of another data.
- “Typing Data by Using an Alias” on page 6-49 — Specify the data type by using a data type alias.

Selecting Stateflow Data Types

In the **Model Explorer**, when you click the **Type** field for data in its **Data** properties dialog or click the **Data Type** element in the row for data in the **Contents** pane, a selectable list of data types appears. You can directly specify the type of the data by selecting one of the following types:

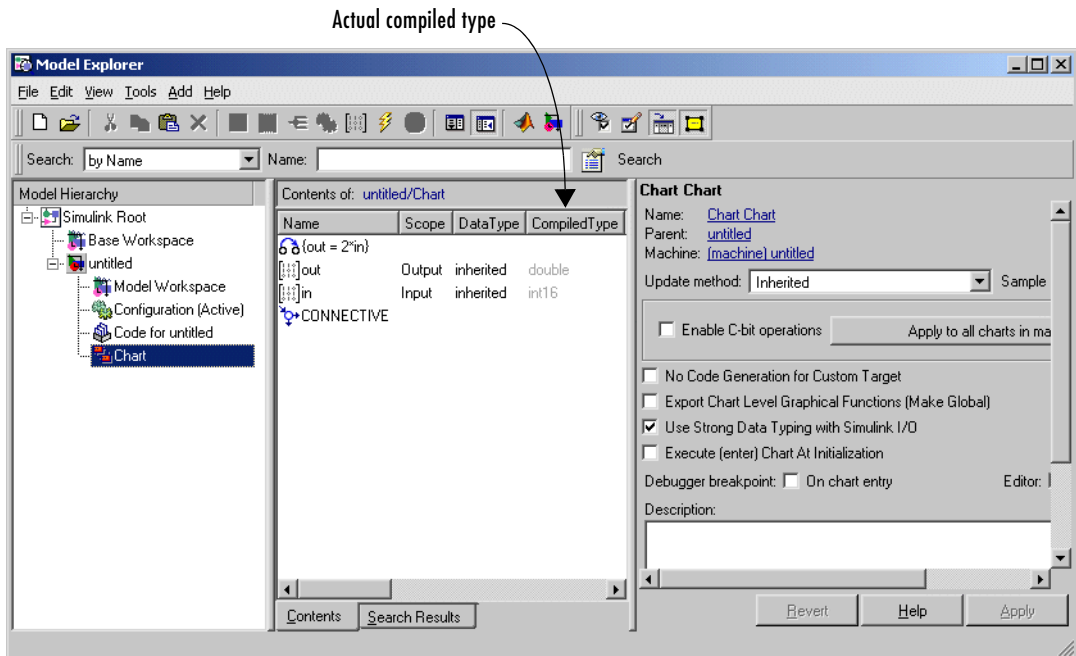
Entry	Description
double	64-bit double-precision floating point
single	32-bit single-precision floating point
int32	32-bit signed integer
int16	16-bit signed integer
int8	8-bit signed integer
uint32	32-bit unsigned integer

Entry	Description
uint16	16-bit unsigned integer
uint8	8-bit unsigned integer
boolean	Boolean (1 = true; 0 = false)
fixpt	<p>Fixed-point data</p> <p>Specifying <code>fixpt</code> enables the Stored Integer and Scaling fields in the adjacent Fixed-Point field section, which are used to specify the fixed-point type. See “Using Fixed-Point Data in Stateflow” on page 8-1 for a description of the Stateflow fixed-point type and its use in Stateflow.</p> <p>For fixed-point data, the Use Strong Data Typing with Simulink IO option in the chart properties dialog (see “Specifying Chart Properties” on page 9-4) is always on. This means that if input or output fixed-point data in Stateflow does not match its counterpart data in Simulink, a mismatch error results.</p>
inherited	<p>Inherit type of input or output data from Simulink</p> <p>See “Inheriting Input and Output Data Type from Simulink” on page 6-46 for more details.</p>
ml	<p>This data is designed to hold MATLAB data in Stateflow. See “ml Data Type” on page 7-32.</p>

Inheriting Input and Output Data Type from Simulink

You can tell the Stateflow chart to inherit the type of input data and output data from Simulink by selecting `Inherited` in the **Type** field for that data. This is the default setting for input and output data that you add to the Stateflow chart. Once you connect Simulink signals to the ports for input and output data and build the model, the **CompiledType** column of the **Model Explorer** gives the actual type inherited.

In the following example, a Stateflow diagram inherits an input signal from a Constant block of type `int16`, multiplies it by the numerical constant 2, and assigns the result to the output.



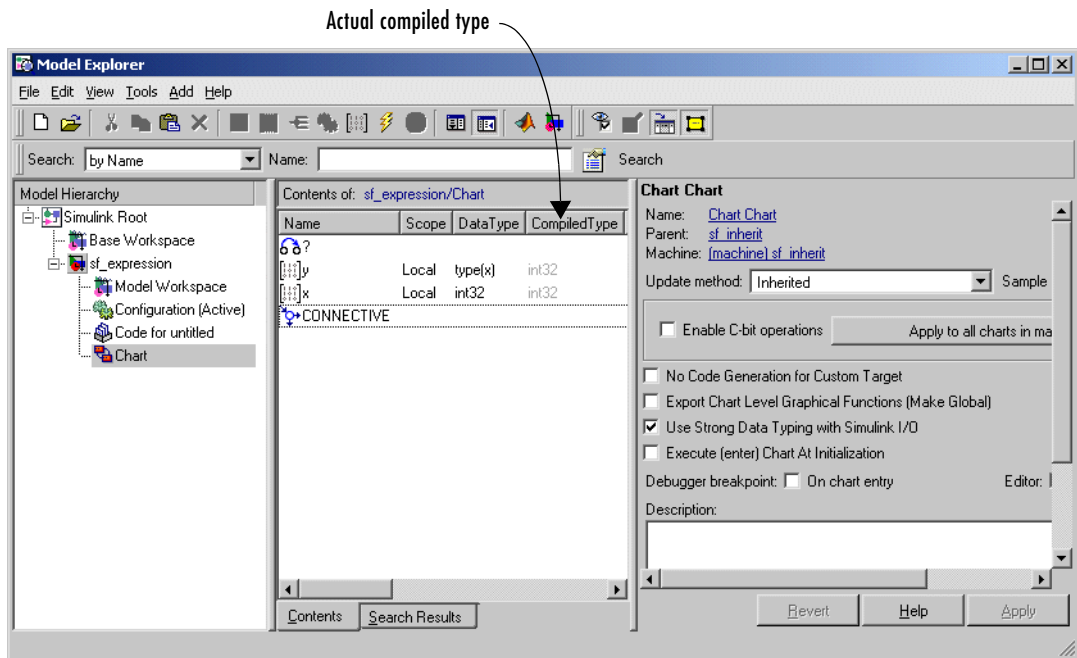
Inheriting the type of input data is successful in all cases. According to the **CompiledType** field in the preceding example, the Stateflow diagram inherits the input type correctly at `int16`.

The inherited type of output data is inferred from diagram actions that store values in the specified output. Because numerical constants in Stateflow are of type `double`, the inherited output type in the example is `double`. If the expected output type in Simulink matches the inferred type, inheritance is successful. In all other cases, a mismatch error occurs during build time.

Note Stateflow blocks in a library cannot inherit input or output data types.

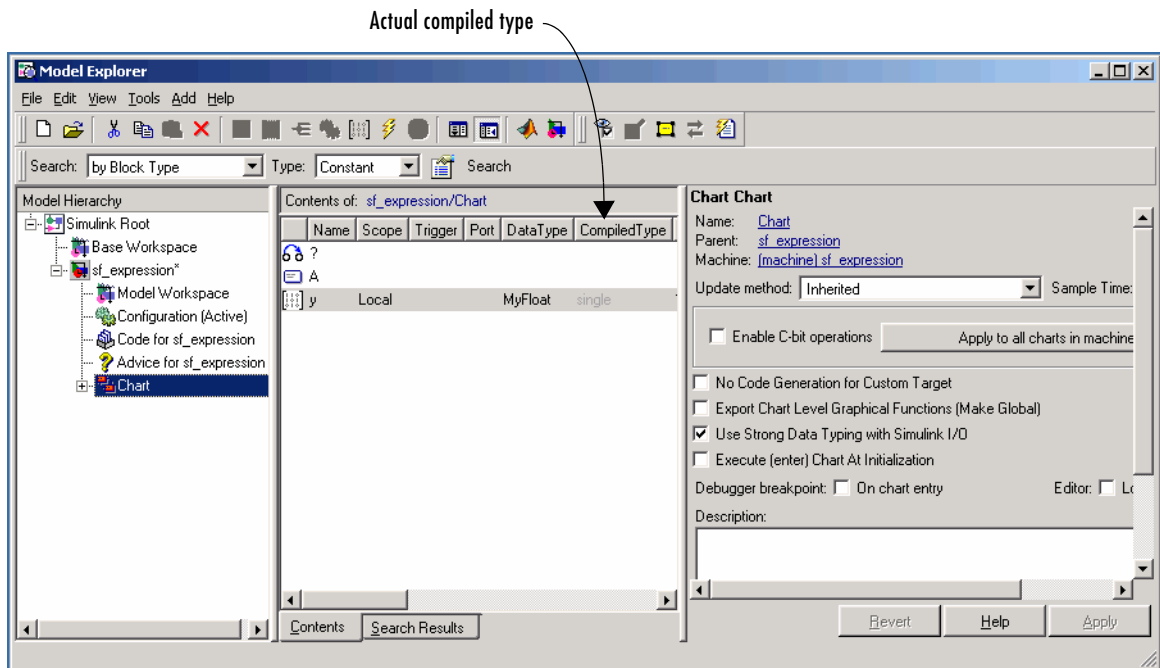
Typing Data by Using the Type of Other Data

You can specify the type of any Stateflow data with the type of other Stateflow data returned by the type operator. Once you build the model, the **CompiledType** column of the **Model Explorer** gives the actual type used in the compiled simulation application. In the following example, the Stateflow block local data *y* is typed by the expression type (*x*), where *x* is typed as an `int32`.



Typing Data by Using an Alias

You can specify the type of Stateflow data by using a Simulink data type alias (for more information, see `Simulink.AliasType` in the Simulink Reference documentation). Once you build the model, the **CompiledType** column of the **Model Explorer** gives the actual type used in the compiled simulation application. In the following example, the Stateflow block local data `y` is typed by the alias `MyFloat`, with `BaseType` set to `single`.



Sizing Stateflow Data

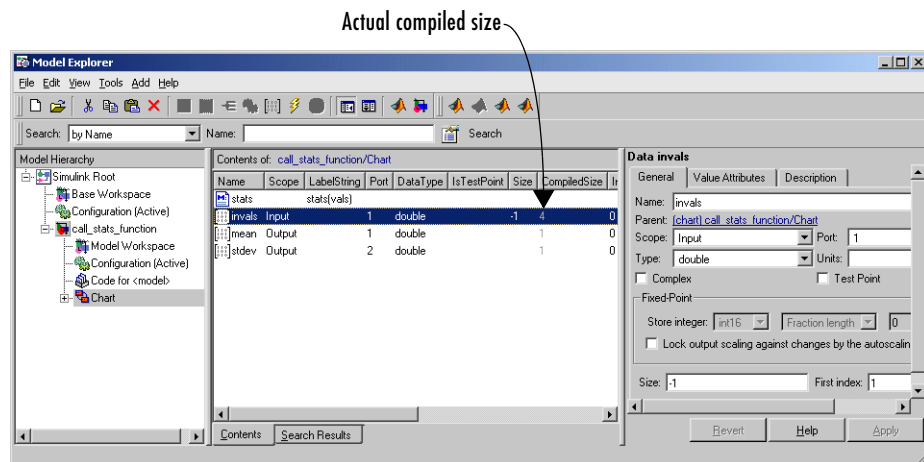
You specify the size of Stateflow data in the **Size** field of the **Data** properties dialog. You can access this dialog when you create data in Stateflow or when you access the **Model Explorer** for existing data. In the **Model Explorer**, you can also specify data size in the middle **Contents** pane by clicking the cell in the **Size** column of the highlighted variable row and editing it. See “Adding Data” on page 6-21 for details on creating and defining data in Stateflow.

You can size data to be a scalar, vector, or an n-dimensional matrix. You can also specify that the data inherit its size from a Simulink signal that it is connected to. Use the following topics to decide on what to enter for the size of your data in Stateflow:

- “Inheriting Input and Output Data Size from Simulink” on page 6-50 — Have Stateflow use Simulink inputs and outputs to size the data.
- “Sizing Data by Expression” on page 6-51 — Use expressions with constants and operators to define the size of Stateflow data.

Inheriting Input and Output Data Size from Simulink

You can tell the Stateflow block to inherit the size of input data and output data from Simulink by entering a -1 in the **Size** field for that data. This is the default setting for input and output data that you add to a Stateflow chart. Once you build the model, the **CompiledSize** column of the **Model Explorer** gives the actual size used in the compiled simulation application.



In the preceding example, the input data `invals` is connected to a Constant block that specifies a four-element vector. The **CompiledSize** field displays the correct inherited size of 4 for `invals`.

Inheriting the size of input data is complete for all cases. The inherited size of output data is inferred from diagram actions that store values in the specified output. If the expected size in Simulink matches the inferred size, inheritance is successful. In all other cases, a mismatch occurs during build time.

Note Stateflow blocks in a library cannot inherit input or output data size. Also, Stateflow cannot inherit frame-based data sizes from Simulink.

Sizing Data by Expression

You set the **Size** property for an array in `[row column page]` format. For example, a value of `[2 4]` defines a 2-by-4 matrix in `[row column]` format. To define a row vector of size 5, set the **Size** to `[1 5]`. To define a column vector of size 6, set the **Size** property to `[6 1]` or just 6.

You can enter a MATLAB expression for each element (row, column, page) in the **Size** field that uses one or more of the following:

- Numeric constants — For example, 1, 3, 7.54, and so on.

- Arithmetic operators — Restricted to +, -, *, and /.
- Stateflow constants — Read-only variables for Stateflow declared in the **Model Explorer** with a scope of **Constant**. They retain the value they are initialized with, which is set in the **Data** dialog or **Contents** pane of the **Model Explorer**. See “Setting Value Attribute Data Properties” on page 6-30 for details on setting the initial value in the **Data** dialog.
- Parameters — Read-only data for the Stateflow chart declared in the **Model Explorer** with the scope **Parameter**. They take their value from a Simulink parameter of the same name for a parent masked subsystem or from a variable in the MATLAB base workspace. See “Initializing Data from the MATLAB Base Workspace” on page 6-40 for more information on data with the scope **Parameter** in Stateflow.
- Calls to the MATLAB functions `min`, `max`, and `size`.

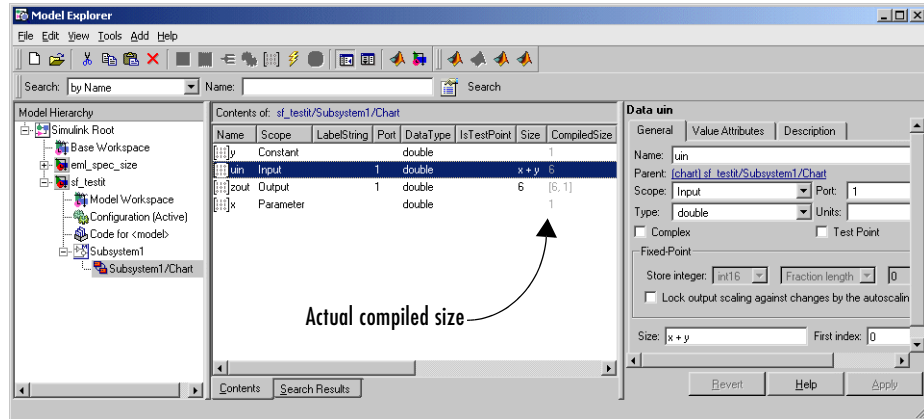
Note The result of the expression you enter for the size of data must resolve to a positive integer, or an error results. For example, the expression $x+y$, where $x=4.21$ and $y=3.79$, results in a size of 8.00 , which is acceptable. However, if $x=4.22$ instead, this results in a size of 8.01 , which flags an error.

By using parameterized sizing, you can quickly resize large numbers of data by changing the size of a single Simulink parameter or MATLAB variable. If k , x , and y are Stateflow data of scope **Constant** or **Parameter**, the following examples are valid expressions that you can enter in the **Size** field of other data:

```
k+1
size(x)
min(size(y),k)
```

Note You cannot size Stateflow input data with an expression that accepts frame-based data from Simulink.

Once you build the model, the **CompiledSize** column of the **Model Explorer** gives the actual size used in the compiled simulation application. In the following example, the Stateflow block input variable `uin` is sized by the expression $x+y$.



`x` is a local data with the scope **Parameter** that takes its value from the parameter `x` for the masked subsystem containing the Stateflow block. Its value is initialized to 2.1. `y` is a local variable of scope **Constant** initialized to 3.9. $x+y$ is therefore 6.

Using Actions in Stateflow

Stateflow attaches actions to a state or transition through its label. The actions in action language can be broadcast events, condition statements, function calls, variable assignments and operations, and so on. Many are very similar to statements in C or MATLAB. Stateflow also defines categories for the actions that you specify, known as action types. This chapter describes the action types of states and transitions and the actions that they contain in the following sections:

Defining Action Types (p. 7-3)	Gives a description of each type of action language available to states and transitions and shows their behavior in an example.
Using Operations in Actions (p. 7-12)	Describes the available data operations in Stateflow action language.
Using Special Symbols in Actions (p. 7-19)	Learn the special symbols Stateflow uses to provide the user with special features in action language notation.
Calling C Functions in Actions (p. 7-22)	Describes the C functions that you can call directly in Stateflow action language.
Using MATLAB Functions and Data in Actions (p. 7-27)	Tells you how you can call MATLAB functions and access MATLAB workspace variables in action language, using the <code>m1</code> namespace operator or the <code>m1</code> function.
Using Data and Event Arguments in Actions (p. 7-40)	Tells you how to reference data defined at different levels of containment in a Stateflow chart when you use them as arguments for functions that you call in action language.
Using Arrays in Actions (p. 7-42)	Describes how to use Stateflow data arrays in action language.
Broadcasting Events in Actions (p. 7-44)	Describes event broadcasting and directed event broadcasting in action language.

Using Temporal Logic in Actions
(p. 7-50)

You can test the occurrence of a specified multiple of events. Learn how to use temporal logic in Stateflow action language.

Using Bind Actions to Control
Function-Call Subsystems (p. 7-58)

Gives a quick example of each type of action language available to Stateflow.

Defining Action Types

Stateflow attaches actions to states and transitions through the syntax of their labels. Each action is entered as an action of a particular type. States specify actions through five action types: entry, during, exit, bind, and on *event_name*. Transitions specify actions through four action types: event trigger, condition, condition action, and transition action. This section describes and gives examples of the action language types for states and transitions in the following topics:

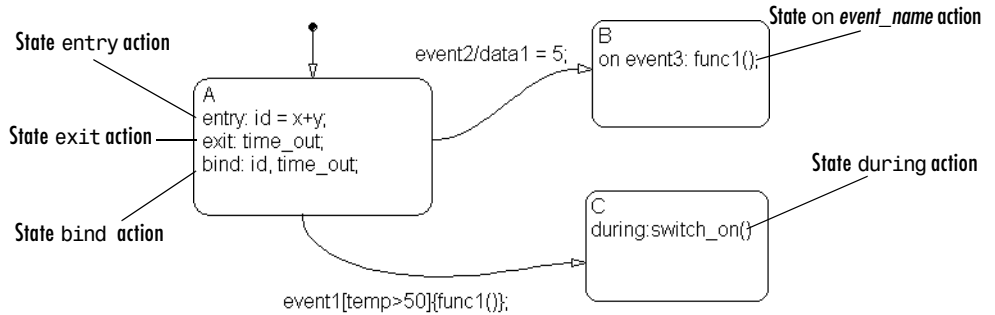
- “State Action Types” on page 7-3 — Introduces you to the notation and meaning of actions that accompany states.
- “Transition Action Types” on page 7-7 — Introduces you to the notation and meaning of actions that accompany transitions.
- “Example of Action Type Execution” on page 7-9 — Shows how the different action types interact in executing example Stateflow diagram.

State Action Types

States can have different action types, which include entry, during, exit, bind, and, on *event_name* actions. The actions for states are assigned to an action type using label notation with the following general format:

```
name /  
entry:entry actions  
during:during actions  
exit:exit actions  
bind:data_name, event_name  
on event_name:on event_name actions
```

The following example shows examples of state action types:



After you enter the name in the state’s label, enter a carriage return and specify the actions for the state. A description of each action type is given in the following topics:

Note The order that you use to enter action types in the label is irrelevant.

Entry Actions

Entry actions are preceded by the prefix `entry` or `en` for short, followed by a required colon (:), followed by one or more actions. Separate multiple actions with a carriage return, semicolon (;), or a comma (,). If you enter the name and slash followed directly by actions, the actions are interpreted as entry action(s). This shorthand is useful if you are specifying entry actions only.

Entry actions are executed for a state when the state is entered (becomes active). In the preceding example in “State Action Types” on page 7-3, the entry action `id = x+y` is executed when the state A is entered by the default transition.

For a detailed description of the semantics of entering a state, see “Entering a State” on page 3-20 and “State Execution Example” on page 3-23.

Exit Actions

Exit actions are preceded by the prefix `exit` or `ex` for short, followed by a required colon (:), followed by one or more actions. Separate multiple actions with a carriage return, semicolon (;), or a comma (,).

Exit actions for a state are executed when the state is active and a transition out of the state is taken.

For a detailed description of the semantics of exiting a state, see “Exiting an Active State” on page 3-23 and “State Execution Example” on page 3-23.

During Actions

During actions are preceded by the prefix `during` or `du` for short, followed by a required colon (:), followed by one or more actions. Separate multiple actions with a carriage return, semicolon (;), or a comma (,).

During actions are executed for a state when it is active and an event occurs and no valid transition to another state is available.

For a detailed description of the semantics of executing an active state, see “Executing an Active State” on page 3-22 and “State Execution Example” on page 3-23.

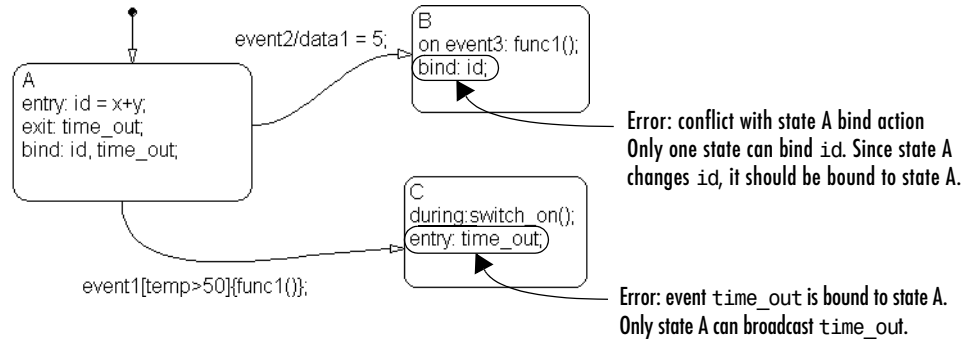
Bind Actions

Bind actions are preceded by the prefix `bind`, followed by a required colon (:), followed by one or more events or data. Separate multiple data/events with a carriage return, semicolon (;), or a comma (,).

Bind actions bind the specified data and events to a state. Data bound to a state can be changed by the actions of that state or its children. Other states and their children are free to read the bound data, but they cannot change it. Events bound to a state can be broadcast only by that state or its children. Other states and their children are free to listen for the bound event, but they cannot send it.

Bind actions are applicable to a Stateflow diagram whether the binding state is active or not. In the preceding example in “State Action Types” on page 7-3, the bind action `bind: id, time_out` for state A binds the data `id` and the event `time_out` to state A. This forbids any other state (or its children) in the Stateflow diagram from changing `id` or broadcasting event `time_out`.

If another state includes actions that change data or send events that are bound to another state, a parsing error results. The following example demonstrates a few of these error conditions:



Binding a function-call event to a state also binds the function-call subsystem that it calls. In this case, the function-call subsystem is enabled when the binding state is entered and disabled when the binding state is exited. For a detailed description of this feature, see “Using Bind Actions to Control Function-Call Subsystems” on page 7-58.

On Event_Name Actions

On *event_name* actions are preceded by the prefix `on`, followed by a unique event, *event_name*, followed by one or more actions. Separate multiple actions with a carriage return, semicolon (;), or a comma (,). You can specify actions for more than one event by adding additional `on event_name` lines for different events. If you want different events to trigger different actions, enter multiple `on event_name` action statements in the state’s label, each specifying the action for a particular event or set of events, for example:

```
on ev1: action1();
on ev2: action2();
```

On *event_name* actions for a state are executed when the state is active and the event *event_name* is received by the state. This is also accompanied by the execution of any `during` actions for the state.

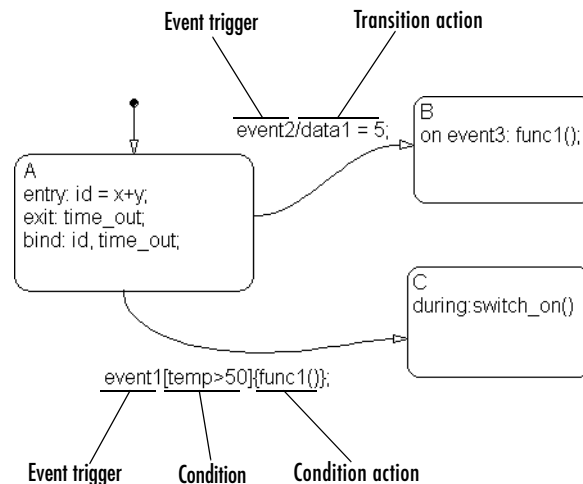
For a detailed description of the semantics of executing an active state, see “Executing an Active State” on page 3-22.

Transition Action Types

In “State Action Types” on page 7-3, you see how Stateflow attaches actions to the label for a state. Stateflow also attaches actions to a transition through its label. Transitions can have different action types, which include event triggers, conditions, condition actions, and transition actions. The actions for transitions are assigned to an action type using label notation with the following general format:

```
event_trigger[condition]{condition_action}/transition_action
```

The following example shows examples of transition action types:



Event Triggers

In transition label syntax, event triggers appear first as the name of an event. They have no distinguishing special character to separate them from other actions in a transition label. In the example in “Transition Action Types” on page 7-7, both transitions from state A have event triggers. The transition from state A to state B has the event trigger `event2` and the transition from state A to state C has the event trigger `event1`.

Event triggers specify an event that causes the transition to be taken, provided the condition, if specified, is true. Specifying an event is optional. The absence of an event indicates that the transition is taken upon the occurrence of any event. Multiple events are specified using the OR logical operator (|).

Conditions

In transition label syntax, conditions are Boolean expressions enclosed in square brackets ([]). In the example in “Transition Action Types” on page 7-7, the transition from state A to state C has the condition `temp > 50`.

A condition is a Boolean expression to specify that a transition occurs given that the specified expression is true. The following are some guidelines for defining and using conditions:

- The condition expression must be a Boolean expression of some kind that evaluates to either true (1) or false (0).
- The condition expression can consist of any of the following:
 - Boolean operators that make comparisons between data and numeric values
 - A function that returns a Boolean value
 - The `in(state_name)` condition function that is evaluated as true when the state specified as the argument is active. The full state name, including any ancestor states, must be specified to avoid ambiguity.

Note A chart cannot use the In condition function to trigger actions based on the activity of states in other charts.

- Temporal conditions (see “Using Temporal Logic in Actions” on page 7-50)
- The condition expression should not call a function that causes the Stateflow diagram to change state or modify any variables.
- Boolean expressions can be grouped using `&` for expressions with AND relationships and `|` for expressions with OR relationships.
- Assignment statements are not valid condition expressions.
- Unary increment and decrement actions are not valid condition expressions.

Condition Actions

In transition label syntax, condition actions follow the transition condition and are enclosed in curly braces (`{}`). In the example in “Transition Action Types” on page 7-7, the transition from state A to state B has the condition action `func()`, a function call.

Condition actions are executed as soon as the condition is evaluated as true, but before the transition destination has been determined to be valid. If no condition is specified, an implied condition evaluates to true and the condition action is executed.

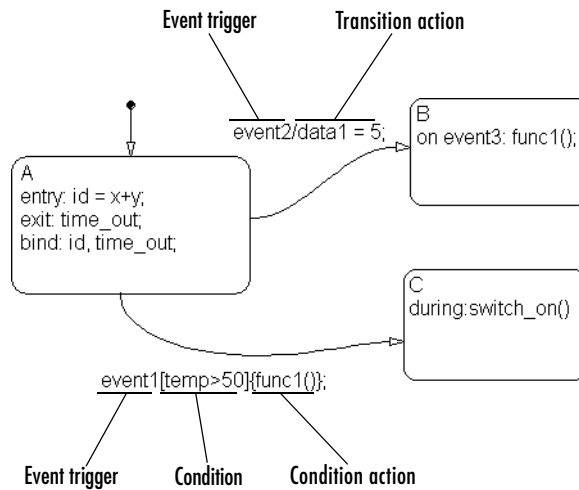
Transition Actions

In transition label syntax, transition actions are preceded with a forward slash (`/`). In the example in “Transition Action Types” on page 7-7, the transition from state A to state B has the transition action `data1 = 5`.

Transition actions are executed when the transition is actually taken. They are executed after the transition destination has been determined to be valid, and the condition, if specified, is true. If the transition consists of multiple segments, the transition action is only executed when the entire transition path to the final destination is determined to be valid.

Example of Action Type Execution

In “State Action Types” on page 7-3 and “Transition Action Types” on page 7-7, you are introduced to the notation and meaning of the action language types in Stateflow. In this topic, you see how Stateflow action language types interact when you execute the following example Stateflow diagram:



If the Stateflow diagram is turned on, the following takes place:

- 1 The default transition to state A is taken.
- 2 The entry action `id = x+y` is executed.
- 3 The event `time_out` is bound to state A.
- 4 State A is active.

If state A is active and the Stateflow diagram receives the event `event2`, the following takes place:

- 1 The exit action broadcast of the event `time_out` is executed.
- 2 State A becomes inactive.
- 3 The transition action `data1 = 5` is executed.
- 4 State B becomes active.

If state A is active and the Stateflow diagram receives the event `event1`, the following takes place:

- 1** The condition action call to the function `func1 ()` is executed.
- 2** The condition `temp > 50` is evaluated. If it is true, do the remaining steps. If it is false, stop here.
- 3** The exit action broadcast of the event `time_out` is executed.
- 4** State A becomes inactive.
- 5** The transition action `data1 = 5` is executed.
- 6** State B becomes active.

Using Operations in Actions

Stateflow maintains a set of allowable operations between Stateflow data in action language. The following sections categorize the operations you can use in Stateflow action language:

- “Binary and Bitwise Operations” on page 7-12 — Lists and describes the supported action language operations that require two operands.
- “Unary Operations” on page 7-15 — Lists and describes the supported action language operations that require one operand.
- “Unary Actions” on page 7-15 — Lists and describes the supported action language operations that require one operand and an operator to the right.
- “Assignment Operations” on page 7-15 — Lists and describes the supported action language operations that assign the results of an operation to an operand.
- “Pointer and Address Operations” on page 7-16 — Lists and describes the supported action language operations that point to the location of data.
- “Type Cast Operations” on page 7-17 — Lists and describes the supported action language operations that change the data type of an operand.

Binary and Bitwise Operations

The table that follows summarizes the interpretation of all binary operators in Stateflow action language. Table order gives relative operator precedence; highest precedence (10) is at the top of the table. Binary operators are evaluated left to right (left associative).

You can specify that the binary operators `&`, `^`, `|`, `&&`, and `||` are interpreted as bitwise operators in Stateflow generated C code for a chart or for all the charts in a model. See these individual operators in the table that follows for specific binary or bitwise operator interpretations.

Example	Precedence	Description
<code>a * b</code>	10	Multiplication
<code>a / b</code>	10	Division
<code>a %% b</code>	10	Modulus

Example	Precedence	Description
<code>a + b</code>	9	Addition
<code>a - b</code>	9	Subtraction
<code>a >> b</code>	8	Shift operand a right by b bits. Noninteger operands for this operator are first cast to integers before the bits are shifted.
<code>a << b</code>	8	Shift operand a left by b bits. Noninteger operands for this operator are first cast to integers before the bits are shifted.
<code>a > b</code>	7	Comparison of the first operand greater than the second operand
<code>a < b</code>	7	Comparison of the first operand less than the second operand
<code>a >= b</code>	7	Comparison of the first operand greater than or equal to the second operand
<code>a <= b</code>	7	Comparison of the first operand less than or equal to the second operand
<code>a == b</code>	6	Comparison of equality of two operands
<code>a ~= b</code>	6	Comparison of inequality of two operands
<code>a != b</code>	6	Comparison of inequality of two operands
<code>a <> b</code>	6	Comparison of inequality of two operands

Example	Precedence	Description
$a \& b$	5	<p>One of the following:</p> <ul style="list-style-type: none"> Bitwise AND of two operands <p>Enabled when Enable C-bit operations is selected in chart properties dialog. See “Specifying Chart Properties” on page 9-4.</p> <ul style="list-style-type: none"> Logical AND of two operands <p>Enabled when Enable C-bit operations is cleared in chart properties dialog.</p>
$a \wedge b$	4	<p>One of the following:</p> <ul style="list-style-type: none"> Bitwise XOR of two operands <p>Enabled when Enable C-bit operations is selected in chart properties dialog. See “Specifying Chart Properties” on page 9-4.</p> <ul style="list-style-type: none"> Operand a raised to power b <p>Enabled when Enable C-bit operations is cleared in chart properties dialog.</p>
$a b$	3	<p>One of the following:</p> <ul style="list-style-type: none"> Bitwise OR of two operands <p>Enabled when Enable C-bit operations is selected in chart properties dialog. See “Specifying Chart Properties” on page 9-4.</p> <ul style="list-style-type: none"> Logical OR of two operands <p>Enabled when Enable C-bit operations is cleared in chart properties dialog.</p>
$a \&\& b$	2	Logical AND of two operands
$a b$	1	Logical OR of two operands

Unary Operations

The following unary operators are supported in Stateflow action language. Unary operators have higher precedence than binary operators and are evaluated right to left (right associative).

Example	Description
<code>~a</code>	Logical NOT of a Complement of a (if bitops is enabled)
<code>!a</code>	Logical NOT of a
<code>-a</code>	Negative of a

Unary Actions

The following unary actions are supported in Stateflow action language.

Example	Description
<code>a++</code>	Increment a
<code>a--</code>	Decrement a

Assignment Operations

The following assignment operations are supported in Stateflow action language.

Example	Description
<code>a = expression</code>	Simple assignment
<code>a := expression</code>	Used primarily with fixed-point numbers. See “Assignment (=, :=) Operations” on page 8-31 for a detailed description.
<code>a += expression</code>	Equivalent to <code>a = a + expression</code>

Example	Description
<code>a -= expression</code>	Equivalent to <code>a = a - expression</code>
<code>a *= expression</code>	Equivalent to <code>a = a * expression</code>
<code>a /= expression</code>	Equivalent to <code>a = a / expression</code>

The following assignment operations are supported in Stateflow action language when **Enable C-bit operations** is selected in the properties dialog for the chart. See “Specifying Chart Properties” on page 9-4.

Example	Description
<code>a = expression</code>	Equivalent to <code>a = a expression</code> (bit operation). See operation <code>a b</code> in “Binary and Bitwise Operations” on page 7-12.
<code>a &= expression</code>	Equivalent to <code>a = a & expression</code> (bit operation). See operation <code>a & b</code> in “Binary and Bitwise Operations” on page 7-12.
<code>a ^= expression</code>	Equivalent to <code>a = a ^ expression</code> (bit operation). See operation <code>a ^ b</code> in “Binary and Bitwise Operations” on page 7-12.

Pointer and Address Operations

The address operator is available for use with both custom code variables and Stateflow variables. The pointer operator is available for use with custom code variables only.

Note The action language parser uses a relaxed set of restrictions. As a result, many syntax errors are not trapped until compilation.

The following examples show syntax that is valid for use with *custom code* variables only.

```
varStruct.field = <expression>;
```

```
(*varPtr) = <expression>;
varPtr->field = <expression>;
myVar = varPtr->field;
varPtrArray[index]->field = <expression>;
varPtrArray[expression]->field = <expression>;
myVar = varPtrArray[expression]->field;
```

The following examples show syntax that is valid for use both with custom code variables and with Stateflow variables.

```
varPtr = &var;
ptr = &varArray[<expression>];
*(&var) = <expression>;
function(&varA, &varB, &varC);
function(&sf.varArray[<expression>]);
```

Type Cast Operations

Stateflow provides type cast operators to convert a value of one type to a value that can be represented in another type. Normally, you do not need to use type cast operators in actions because Stateflow checks whether the types involved in a variable assignment differ and compensates by inserting the required type cast operator of the target language (typically C) in the generated code.

However, external (custom) code might require data in a different type from those currently available in Stateflow. In this case, Stateflow cannot determine the required type casts and you must explicitly use a Stateflow action language type cast operator to tell Stateflow the target language type cast operator to generate.

For example, you might have a custom code function that requires integer RGB values for a graphic plot. You might have these values in Stateflow data, but only in data of type `double`. To call this function, you must type cast the original data and assign the result to integers, which you use as arguments to the function.

In Stateflow, type cast operations have two forms: the MATLAB type cast form and the explicit form using the cast operator. These operators and the special type operator, which works with the explicit cast operator, are described in the topics that follow.

MATLAB Form Type Cast Operators

The MATLAB type casting form in Stateflow has the general form

```
<type_op>(<expression>)
```

<type_op> is a conversion type operator that can be double, single, int32, int16, int8, uint32, uint16, uint8, or boolean. <expression> is the expression to be converted. For example, you can cast the expression `x+3` to a 16-bit unsigned integer and assign its value to the data `y` as follows:

```
y = uint16(x+3)
```

Explicit Type Cast Operator

You can also type cast with the explicit cast operator, which has the following general form:

```
cast(<expression>, <type>)
```

As in the preceding example, the statement

```
y = cast(x+3, uint16)
```

tells Stateflow to cast the expression `x+3` to a 16-bit unsigned integer and assign it to `y`, which can be of any type.

type Operator

To make type casting more convenient, Stateflow also provides a type operator that works with the explicit type cast operator `cast` to let you assign types to data based on the types of other data.

The type operator returns the type of an existing Stateflow data according to the general form

```
type(<data>)
```

where <data> is the Stateflow data whose type you want to return.

The return value from a type operation can be used only in an explicit cast operation. For example, if you want to convert the data `y` to the same type as that of data `z`, use the following statement:

```
cast(y, type(z))
```

In this case, the data `z` can have any acceptable Stateflow type.

Using Special Symbols in Actions

Stateflow notation uses the symbols `inf`, `NaN`, `t`, `$`, `...`, `%`, `//`, `/*`, `;`, `F`, and hexadecimal notation to provide the user with special features in action language notation as described in the following topics:

- “Comment Symbols”
- “Hexadecimal Notation Symbols”
- “Infinity Symbol, `inf`”
- “Line Continuation Symbol, `...`”
- “Literal Code Symbol, `$`”
- “MATLAB Display Symbol, `;`”
- “Single-Precision Floating-Point Number Symbol, `F`”
- “Time Symbol, `t`”

Comment Symbols

Use the the symbols `%`, `//`, and `/*` to represent comments in Stateflow action language as shown in the following examples:

```
% MATLAB comment line
// C++ comment line
/* C comment line */
```

You can optionally include comments in Stateflow generated code for a Real-Time Workshop target (see “Real-Time Workshop Options” in the Real-Time Workshop documentation) or a Stateflow custom target (see “Configuring a Custom Target in Stateflow” on page 12-24). Stateflow action language comments in generated code are represented with multibyte character code. This means that you can have comments in code with characters for non-English alphabets such as Japanese Kanji characters.

Hexadecimal Notation Symbols

Stateflow action language supports C style hexadecimal notation. For example, `0xFF`. You can use hexadecimal values wherever you can use decimal values.

Infinity Symbol, `inf`

Use the MATLAB symbol `inf` to represent infinity in Stateflow action language. Calculations like `n/0`, where `n` is any nonzero real value, result in `inf`.

Line Continuation Symbol, `...`

Use the characters `...` at the end of a line of action language to indicate that the expression continues on the next line.

Literal Code Symbol, `$`

Use `$` characters to mark action language that you want the parser to ignore but you want to appear in the generated code. For example,

```
$  
ptr -> field = 1.0;  
$
```

The parser is completely disabled during the processing of anything between the `$` characters. Frequent use of literals is discouraged.

MATLAB Display Symbol, `;`

Omitting the semicolon after an expression displays the results of the expression in the MATLAB Command Window. If you use a semicolon, the results are not displayed.

Single-Precision Floating-Point Number Symbol, `F`

Use a trailing `F` to specify single-precision floating-point numbers in Stateflow action language. For example, you can use the action statement `x = 4.56F;` to specify a single-precision constant with the value 4.56. If a trailing `F` does not appear with a number, it is assumed to be double-precision.

Time Symbol, t

Use the letter t to represent absolute time inherited from Simulink in simulation targets. For example, the condition $[t - \text{On_time} > \text{Duration}]$ specifies that the condition is true if the value of On_time subtracted from the simulation time t is greater than the value of Duration .

Note The meaning of t for nonsimulation targets is undefined since it is dependent upon the specific application and target hardware.

Calling C Functions in Actions

This section describes the C functions that you can call directly in Stateflow action language. See the following topics:

- “Calling C Library Functions” on page 7-22 — Tells you how to call C functions from the math library of your compiler or from custom libraries that you provide.
- “Calling min and max Functions” on page 7-23 — Describes the special min and max macros that you can call.
- “Calling User-Written C Code Functions” on page 7-24 — Tells you how to call C functions you provide in custom code for the target you build.

Calling C Library Functions

You can call the following small subset of the C Math Library functions:

abs ^a	acos	asin	atan	atan2	ceil
cos	cosh	exp	fabs	floor	fmod
labs	ldexp	log	log10	pow	rand
sin	sinh	sqrt	tan	tanh	

a. Stateflow extends the abs function beyond that of its standard C counterpart with its own built in functionality. See “Calling the abs Function” on page 7-23.

You can call the above C Math Library functions without doing anything special as long as you are careful to call them with the right data types. In case of a type mismatch, Stateflow replaces the input argument with a cast of the original argument to the expected type. For example, if you call the sin function with an integer argument, Stateflow replaces the argument with a cast of the original argument to a floating-point number of type double.

If you call other C library functions not specified above, be sure to include the appropriate `#include . . .` directive in the **Custom code included at the top of generated code** field of the **Target Options** dialog. See the section “Specifying Custom Code Options for Stateflow Targets” on page 12-29.

Calling the abs Function

Stateflow extends the interpretation of its `abs` function beyond the standard C version to include integer and floating-point arguments of all types as follows:

- If `x` is an integer of type `int32`, Stateflow evaluates `abs(x)` with the standard C function `abs` applied to `x`, or `abs(x)`.
- If `x` is an integer of type other than `int32`, Stateflow evaluates `abs(x)` with the standard C `abs` function applied to a cast of `x` as an integer of type `int32`, or `abs((int32)x)`.
- If `x` is a floating point number of type `double`, Stateflow evaluates `abs(x)` with the standard C function `fabs` applied to `x`, or `fabs(x)`.
- If `x` is a floating point number of type `single`, Stateflow evaluates `abs(x)` with the standard C function `fabs` applied to a cast of `x` as a `double`, or `fabs((double)x)`.
- If `x` is a fixed-point number, Stateflow evaluates `abs(x)` with the standard C function `fabs` applied to a cast of the fixed-point number as a `double`, or `fabs((double)Vx)`, where V_x is the real-world value of `x`.

If you want to use the `abs` function in Stateflow in the strict sense of standard C, be sure to cast its argument or return values to integer types. See “Type Cast Operations” on page 7-17.

Note If `x` is declared in Stateflow custom code, Stateflow evaluates `abs(x)` with the standard C `abs` function in all cases. For instructions on inserting custom code into Stateflow diagrams, see “Integrating Custom Code with Stateflow Targets” on page 12-29.

Calling min and max Functions

Although `min` and `max` are not C library functions, Stateflow enables them by emitting the following macros automatically at the top of generated code.

```
#define min(x1,x2) ((x1) > (x2)) ? (x2) : (x1)
#define max(x1,x2) ((x1) > (x2)) ? (x1) : (x2)
```

To allow compatibility with user graphical functions named `min()` or `max()`, Stateflow generates code for them with a mangled name of the following form: `<prefix>_min`. However, if you export `min()` or `max()` graphical functions to other Stateflow charts in the Stateflow machine, the name of these functions can no longer be emitted with mangled names in generated code and conflict occurs. To avoid this conflict, rename the `min()` and `max()` graphical functions.

Calling User-Written C Code Functions

To install your own C code functions for use in Stateflow action language, do the following:

- 1 From the Tools menu, select the **Open (RTW or Simulation) Target** dialog.
- 2 When the **Open Target** dialog appears, select **Target Options**.
- 3 Enter the following:
 - Include a header file containing the declarations of your C code functions in the **Custom code included at the top of generated code** field.
 - Specify the source file name containing your C code functions in the **Custom source files** field.

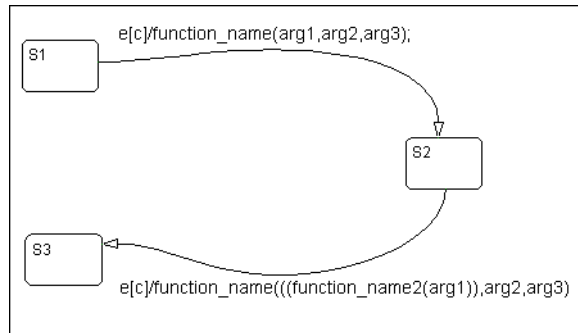
See “Specifying Custom Code Options for Stateflow Targets” on page 12-29.

To use your own C code functions in Stateflow action language, follow these guidelines:

- Define a function by its name, any arguments in parentheses, and an optional semicolon.
- String parameters to user-written functions are passed between single quotation marks. For example, `func('string')`.
- An action can nest function calls.
- An action can invoke functions that return a scalar value (of type `double` in the case of MATLAB functions and of any type in the case of C user-written functions).

Function Call Transition Action Example

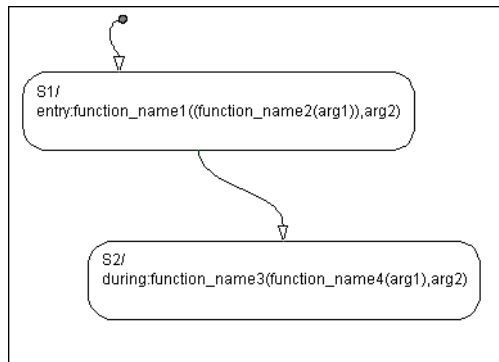
These are example formats of function calls using transition action notation.



If S1 is active, event *e* occurs, *c* is true, and the transition destination is determined, then a function call is made to `function_name` with `arg1`, `arg2`, and `arg3`. The transition action in the transition from S2 to S3 shows a function call nested within another function call.

Function Call State Action Example

These are example formats of function calls using state action notation.



When the default transition into S1 occurs, S1 is marked active and then its entry action, a function call to `function_name1` with the specified arguments, is executed and completed. If S2 is active and an event occurs, the during action, a function call to `function_name3` with the specified arguments, executes and completes.

Passing Arguments by Reference

A Stateflow action can pass arguments to a user-written function by reference rather than by value. In particular, an action can pass a pointer to a value rather than the value itself. For example, an action could contain the following call

```
f(&x);
```

where `f` is a custom-code C function that expects a pointer to `x` as an argument.

If `x` is the name of a data item defined in the Stateflow data dictionary, the following rules apply.

- Do not use pointers to pass data items input from Simulink.
 - If you need to pass an input item by reference, for example, an array, assign the item to a local data item and pass the local item by reference.
- If `x` is a Simulink output data item having a data type other than `double`, the chart property **Use strong data typing with Simulink IO** must be on (see “Specifying Chart Properties” on page 9-4).
- If the data type of `x` is `boolean`, the coder option **Use bitsets to store state-configuration** must be turned off (see “Configuring Real-Time Workshop for Stateflow” on page 12-14).
- If `x` is an array with its first index property set to 0 (see “Size” on page 6-29), then the function must be called as follows.

```
f(&(x[0]));
```

This passes a pointer to the first element of `x` to the function.

- If `x` is an array with its first index property set to a nonzero number (for example, 1), the function must be called in the following way:

```
f(&(x[1]));
```

This passes a pointer to the first element of `x` to the function.

Using MATLAB Functions and Data in Actions

You can call MATLAB functions and access MATLAB workspace variables in action language, using the `m1` namespace operator or the `m1` function. See the following sections:

- “`m1` Namespace Operator” on page 7-27 — Shows you how to use MATLAB workspace variables or call MATLAB functions through the `m1` namespace operator.
- “`m1` Function” on page 7-28 — Shows you how to use MATLAB functions through the `m1` function.
- “`m1` Expressions” on page 7-30 — Shows you how to mix `m1` namespace operator and `m1` function expressions along with Stateflow data in larger expressions.
- “`m1` Data Type” on page 7-32 — Shows you how to use the MATLAB data type to keep MATLAB data in Stateflow instead of in the MATLAB workspace.
- “Inferring Return Size for `m1` Expressions” on page 7-35 — Gives you the rules for providing enough information to infer the size of the values returned from the `m1` namespace operator and the `m1` function in larger action language expressions.

Caution Because MATLAB functions are not available in a target environment, do not use the `m1` namespace operator and the `m1` function if you plan to build an RTW target that includes code from Stateflow Coder.

`m1` Namespace Operator

The `m1` namespace operator uses standard dot (`.`) notation to reference MATLAB variables and functions in action language. For example, the statement `y = m1.x` returns the value of the MATLAB workspace variable `x` to the Stateflow data `y`. The statement `y = m1.matfunc(arg1, arg2)` passes the return value from the MATLAB function `matfunc` to Stateflow data `y`.

If the MATLAB function you call does not require arguments, you must still include the parentheses, as shown in the preceding examples. If the parentheses are omitted, Stateflow interprets the function name as a workspace variable, which, when not found, generates a run-time error during simulation.

In the following examples, *x*, *y*, and *z* are workspace variables and *d1* and *d2* are Stateflow data:

- `a = ml.sin(ml.x)`

In this example, the `sin` function of MATLAB evaluates the sine of *x*, which is then assigned to Stateflow data variable *a*. However, because *x* is a workspace variable, you must use the namespace operator to access it. Hence, `ml.x` is used instead of just *x*.

- `a = ml.sin(d1)`

In this example, the `sin` function of MATLAB evaluates the sine of *d1*, which is assigned to Stateflow data variable *a*. Because *d1* is Stateflow data, you can access it directly.

- `ml.x = d1*d2/ml.y`

The result of the expression is assigned to *x*. If *x* does not exist prior to simulation, it is automatically created in the MATLAB workspace.

- `ml.v[5][6][7] = ml.matfunc(ml.x[1][3], ml.y[3], d1, d2, 'string')`

The workspace variables *x* and *y* are arrays. `x[1][3]` is the (1,3) element of the two-dimensional array variable *x*. `y[3]` is the third element of the one-dimensional array variable *y*. The last argument, `'string'`, is a literal string.

The return from the call to `matfunc` is assigned to element (5,6,7) of the workspace array, *v*. If *v* does not exist prior to simulation, it is automatically created in the MATLAB workspace.

ml Function

You can use the `ml` function to specify calls to MATLAB functions through a string expression in the action language. The format for the `ml` function call uses standard function notation as follows:

```
ml(evalString, arg1,arg2,...);
```

`evalString` is a string expression that is evaluated in the MATLAB workspace. It contains a MATLAB command (or a set of commands, each separated by a semicolon) to execute along with format specifiers (`%g`, `%f`, `%d`, etc.) that provide formatted substitution of the other arguments (`arg1`, `arg2`, etc.) into `evalString`.

The format specifiers used in `m1` functions are the same as those used in the C functions `printf` and `sprintf`. The `m1` function call is equivalent to calling the MATLAB `eval` function with the `m1` namespace operator if the arguments `arg1`, `arg2`, ... are restricted to scalars or string literals in the following command:

```
m1.eval(m1.sprintf(evalString,arg1,arg2,...))
```

Stateflow assumes scalar return values from `m1` namespace operator and `m1` function calls when they are used as arguments in this context. See “Inferring Return Size for `m1` Expressions” on page 7-35.

In the following examples, `x` is a MATLAB workspace variable, and `d1` and `d2` are Stateflow data:

- `a = m1('sin(x)')`

In this example, the `m1` function calls the `sin` function of MATLAB to evaluate the sine of `x` in the MATLAB workspace. The result is then assigned to Stateflow data variable `a`. Because `x` is a workspace variable, and `sin(x)` is evaluated in the MATLAB workspace, you enter it directly in the `evalString` argument (`'sin(x)'`).

- `a = m1('sin(%f)', d1)`

In this example, the `sin` function of MATLAB evaluates the sine of `d1` in the MATLAB workspace and assigns the result to Stateflow data variable `a`. Because `d1` is Stateflow data, its value is inserted in the `evalString` argument (`'sin(%f)'`) using the format expression `%f`. This means that if `d1 = 1.5`, the expression evaluated in the MATLAB workspace is `sin(1.5)`.

- `a = m1('matfunc(%g, 'abcdfg', x, %f)', d1, d2)`

In this example, the string `'matfunc(%g, 'abcdfg', x, %f)'` is the `evalString` shown in the preceding format statement. Stateflow data `d1` and `d2` are inserted into that string with the format specifiers `%g` and `%f`, respectively. The string `'abcdfg'` is a string literal with two single pairs of quotation marks used to enclose it because it is part of the evaluation string, which is already enclosed in single quotation marks.

- `sfmat_44 = ml('rand(4)')`

In this example, a square 4-by-4 matrix of random numbers between 0 and 1 is returned and assigned to the Stateflow data `sf_mat44`. Stateflow data `sf_mat44` must be defined in Stateflow as a 4-by-4 array before simulation. If its size is different, a size mismatch error is generated during run-time.

ml Expressions

You can mix `ml` namespace operator and `ml` function expressions along with Stateflow data in larger expressions. The following example squares the sine and cosine of an angle in workspace variable `X` and adds them:

```
ml.power(ml.sin(ml.X),2) + ml('power(cos(X),2)')
```

The first operand uses the `ml` namespace operator to call the `sin` function. Its argument is `ml.X`, since `X` is in the MATLAB workspace. The second operand uses the `ml` function. Because `X` is in the workspace, it is included in the `evalString` expression as `X`. The squaring of each operand is performed with the MATLAB `power` function, which takes two arguments: the value to square, and the power value, 2.

Expressions using the `ml` namespace operator and the `ml` function can be used as arguments for `ml` namespace operator and `ml` function expressions. The following example nests `ml` expressions at three different levels:

```
a = ml.power(ml.sin(ml.X + ml('cos(Y)')),2)
```

In composing your `ml` expressions, follow the levels of precedence set out in “Binary and Bitwise Operations” on page 7-12. To repeat a warning note in that section, be sure to use parentheses around power expressions with the `^` operator when you use them in conjunction with other arithmetic operators.

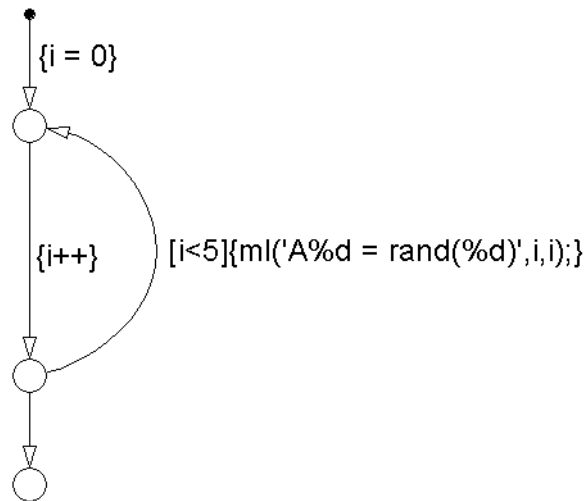
Stateflow checks expressions for data size mismatches in your action language during parsing of your charts and during run-time. Because the return values for `ml` expressions are not known till run-time, Stateflow must infer the size of their return values. See “Inferring Return Size for `ml` Expressions” on page 7-35.

Which ml Should I Use?

In most cases the notation of the `ml` namespace operator is more straightforward. However, using the `ml` function call does offer a few advantages:

- Use the `ml` function to dynamically construct workspace variables.

The following example creates four new MATLAB matrices:



This example demonstrates the use of a Stateflow `for` loop to create four new matrix workspace variables in MATLAB. The default transition initializes the Stateflow counter `i` to 0 while the transition segment between the top two junctions increments it by 1. If `i` is less than 5, the transition segment back to the top junction is taken and evaluates the `ml` function call `ml('A%d = rand(%d)', i, i)` for the current value of `i`. When `i` is greater than or equal to 5, the transition segment between the bottom two junctions is taken and execution stops.

This results in the following MATLAB commands, which create a workspace scalar (A1) and three matrices (A2, A3, A4):

```
A1 = rand(1)
A2 = rand(2)
A3 = rand(3)
A4 = rand(4)
```

- Use the `m1` function with full MATLAB notation.

You cannot use full MATLAB notation with the `m1` namespace operator, as demonstrated by the following example:

```
m1.A = m1.magic(4)
B = m1('A + A''')
```

This example sets the workspace variable `A` to a magic 4-by-4 matrix using the `m1` namespace operator. Stateflow data `B` is then set to the addition of `A` and its transpose matrix, `A'`, which produces a symmetric matrix. Because the `m1` namespace operator cannot evaluate the expression `A'`, the `m1` function is used instead. However, you can call the MATLAB function `transpose` with the `m1` namespace operator in the following equivalent expression:

```
B = m1.A + m1.transpose(m1.A)
```

As another example, you cannot use arguments with cell arrays or subscript expressions involving colons with the `m1` namespace operator. However, these can be included in an `m1` function call.

m1 Data Type

Stateflow data of type `m1` is typed internally with the MATLAB type `mxAarray`. This means that you can assign (store) any type of data available in Stateflow to a data of type `m1`. This includes any type of data defined in Stateflow or returned from MATLAB with the `m1` namespace operator or `m1` function.

The following features and limitations apply to Stateflow data of type `m1`.

- `m1` data can be initialized from the MATLAB workspace just like other data in Stateflow (see the **Initialize from** property in “Setting Data Properties in the Data Dialog” on page 6-25).

- Any numerical scalar or array of m1 data in Stateflow can participate in any kind of unary operation and any kind of binary operation with any other data in Stateflow.

If m1 data participates in any numerical operation with other data, the size of the m1 data must be inferred from the context in which it is used, just as return data from the m1 namespace operator and m1 function are. See “Inferring Return Size for m1 Expressions” on page 7-35.

Note The preceding feature does not apply to m1 data storing MATLAB 64 bit integers. m1 data can store 64 bit MATLAB integers but cannot be used in Stateflow action language.

- m1 data cannot be defined with the scope **Constant** or specified as an array. These options are disabled in the data properties dialog and Stateflow Explorer for Stateflow data of type m1.
- m1 data can be used in building a simulation target (sfun) but not in building an RTW target (rtw).
- If data of type m1 contains an array, the elements of the array can be accessed through indexing with the following rules:
 - a Only arrays with numerical elements can be indexed.
 - b Numerical arrays can be indexed according to their dimension only.
This means that only one-dimensional arrays can be accessed by a single index value. You cannot access a multidimensional array with a single index value.
 - c The first index value for each dimension of an array is 1, and not 0, as in C-language arrays.

In the examples that follow, m1data is a Stateflow data of type m1, ws_num_array is a 2-by-2 MATLAB workspace array with numerical values, and ws_str_array is a 2-by-2 MATLAB workspace array with string values.

```
m1data = m1.ws_num_array; /* OK */
n21 = m1data[2][1]; /* OK for numerical data of type m1 */
n21 = m1data[3]; /* NOT OK for 2-by-2 array data */
m1data = m1.ws_str_array; /* OK */
s21 = m1data[2][1]; /* NOT OK for string data of type m1*/
```

- m1 data cannot have a scope outside Stateflow; that is, it cannot be scoped as **Input to Simulink** or **Output to Simulink**.

Place Holder for Workspace Data

Both the m1 namespace operator and the m1 function can access data directly in the MATLAB workspace and return it to Stateflow. However, maintaining data in the MATLAB workspace can present Stateflow users with conflicts with other data already resident in the workspace. Consequently, with the m1 data type, you can maintain m1 data in Stateflow and use it for MATLAB computations in Stateflow action language.

As an example, in the following Stateflow action language statements, m1data1 and m1data2 are Stateflow data of type m1:

```
m1data1 = m1.rand(3);  
m1data2 = m1.transpose(m1data1);
```

In the first line of this example, m1data1 receives the return value of the rand function in MATLAB, which, in this case, returns a 3-by-3 array of random numbers. Note that m1data1 is not specified as an array or sized in any way. It can receive any MATLAB workspace data or the return of any MATLAB function because it is defined as a Stateflow data of type m1.

In the second line of the example, m1data2, also of Stateflow data type m1, receives the transpose matrix of the matrix in m1data1. It is assigned the return value of the MATLAB function transpose in which m1data1 is the argument.

Note the differences in notation if the preceding example were to use MATLAB workspace data (wsdata1 and wsdata2) instead of Stateflow m1 data to hold the generated matrices:

```
m1.wsdata1 = m1.rand(3);  
m1.wsdata2 = m1.transpose(m1.wsdata);
```

In this case, each workspace data must be accessed through the m1 namespace operator.

Inferring Return Size for m1 Expressions

Stateflow expressions using the `m1` namespace operator and the `m1` function are evaluated in the MATLAB workspace at run-time. This means that the actual size of the data returned from the following expression types is known only at run-time:

- MATLAB workspace data or functions using the `m1` namespace operator or the `m1` function call
For example, the size of the return values from the expressions `m1.var`, `m1.func()`, or `m1(evalString, arg1, arg2, ...)`, where `var` is a MATLAB workspace variable and `func` is a MATLAB function, cannot be known until run-time.
- Stateflow data of type `m1`
- Graphical functions that return Stateflow data of type `m1`

When any of these expressions is used in action language, Stateflow code generation must create temporary Stateflow data, invisible to the user, to hold their intermediate returns for evaluation of the full expression of which they are a part. Because the size of these return values is not known till run-time, Stateflow must employ context rules to infer their size for the creation of the temporary data.

During run-time, if the actual returned value from one of these commands differs from the inferred size of the temporary variable chosen to store it, a size mismatch error results. To prevent these run-time errors, use the following guidelines in constructing action language statements with MATLAB commands or `m1` data:

- 1 The return sizes of MATLAB commands or data in an expression must match the return sizes of peer expressions.

For example, in the expression `m1.func() * (x + m1.y)`, if `x` is a 3-by-2 matrix, then `m1.func()` and `m1.y` are also assumed to evaluate to 3-by-2 matrices. If either returns a value of different size (other than a scalar), an error results during run-time.

2 Expressions that return a scalar never produce an error.

You can combine matrices and scalars in larger expressions because MATLAB practices scalar expansion. For example, in the larger expression `m1.x + y`, if `y` is a 3-by-2 matrix and `m1.x` returns a scalar, the resulting value is determined by adding the scalar value of `m1.x` to every member of `y` to produce a matrix with the size of `y`, that is, a 3-by-2. The same rule applies to subtraction (-), multiplication (*), division (/), and any other binary operations.

3 MATLAB commands or Stateflow data of type `m1` can be members of the following independent levels of expression, for which the return size must be resolved:**- Arguments**

The expression for each function argument is a larger expression for which the return size of MATLAB commands or Stateflow data of type `m1` must be determined. For example, in the expression `z + func(x + m1.y)`, the size of `m1.y` has nothing to do with the size of `z`, because `m1.y` is used at the function argument level. However, the return size for `func(x + m1.y)` must match the size of `z`, because they are both at the same expression level.

- Array indices

The expression for an array index is an independent level of expression that is required to be scalar in size. For example, in the expression `x + arr[y]`, the size of `y` has nothing to do with the size of `x` because `y` and `x` are at different levels of expression, and `y` must be a scalar.

4 The return size for an indexed array element access must be a scalar.

For example, the expression `x[1][1]`, where `x` is a 3-by-2 array, must evaluate to a scalar.

5 MATLAB command or data elements used in an expression for the input argument for a MATLAB function called through the `m1` namespace operator are resolved for size using the rule for peer expressions (preceding rule 1) for the expression itself, because there is no size definition prototype available.

For example, in the function call `m1.func(x + m1.y)`, if `x` is a 3-by-2 array, `m1.y` must return a 3-by-2 array or a scalar.

- 6** MATLAB command or data elements used for the input argument for a graphical function in an expression are resolved for size by the function's prototype.

For example, if the graphical function `gfunc` has the prototype `gfunc(arg1)`, where `arg1` is a 2-by-3 Stateflow data array, then the calling expression, `gfunc(m1.y + x)`, requires that both `m1.y` and `x` evaluate to 2-by-3 arrays (or scalars) during run-time.

- 7** `m1` function calls can take only scalar or string literal arguments. Any MATLAB command or data used to specify an argument for the `m1` function must return a scalar value.
- 8** In an assignment, the size of the right-hand expression must match the size of the left-hand expression, with one exception: if the left-hand expression is a single MATLAB variable such as `m1.x` or a single Stateflow data of type `m1`, then the sizes of both left-hand expression and right-hand expression are determined by the right-hand expression.

For example, in the expression `s = m1.func(x)`, where `x` is a 3-by-2 matrix and `s` is scalar data in Stateflow, `m1.func(x)` must return a scalar to match the left-hand expression, `s`. However, in the expression `m1.y = x + s`, where `x` is a 3-by-2 data array and `s` is scalar, the left-hand expression, workspace variable `y`, is assigned the size of a 3-by-2 array to match the size of the right-hand expression, `x+s`, a 3-by-2 array.

- 9** In an assignment, Stateflow column vectors on the left-hand side are compatible with MATLAB row or column vectors of the same size on the right-hand side.

A matrix you define with a row dimension of 1 is considered a row vector. A matrix you define with one dimension or with a column dimension of 1 is considered a column vector. For example, in the expression `s = m1.func()`, where `m1.func()` returns a 1-by-3 matrix, if `s` is a vector of size 3, the assignment is valid.

- 10** If you cannot resolve the return size of MATLAB command or data elements in a larger expression by any of the preceding rules, they are assumed to return scalar values.

For example, in the expression `m1.x = m1.y + m1.z`, none of the preceding rules can be used to infer a common size among `m1.x`, `m1.y`, and `m1.z`. In this case, both `m1.y` and `m1.z` are assumed to return scalar values. And even if `m1.y` and `m1.z` return matching sizes at run-time, if they return nonscalar values, a size mismatch error results.

- 11** The preceding rules for resolving the size of member MATLAB commands or Stateflow data of type `m1` in a larger expression apply only to cases in which numeric values are expected for that member. For nonnumeric returns, a run-time error results.

For example, the expression `x + m1.str`, where `m1.str` is a string workspace variable, produces a run-time error stating that `m1.str` is not a numeric type.

Note Member MATLAB commands or data of type `m1` in a larger expression are limited to numeric values (scalar or array) only if they participate in numeric expressions.

- 12** Stateflow has special cases in which it does no size checking to resolve the size of MATLAB command or data expressions that are members of larger expressions. In the cases shown, use of a singular MATLAB element such as `m1.var`, `m1.func()`, `m1(evalString, arg1, arg2, ...)`, Stateflow data of type `m1`, or a graphical function returning a Stateflow data of type `m1`, does not require that size checking be enforced at run-time. In these cases, assignment of a return to the left-hand side of an assignment statement or to a function argument is made without consideration for a size mismatch between the two:

- An assignment in which the left-hand side is a MATLAB workspace variable

For example, in the expression `m1.x = m1.y`, `m1.y` is a MATLAB workspace variable of any size and type (structure, cell array, string, and so on).

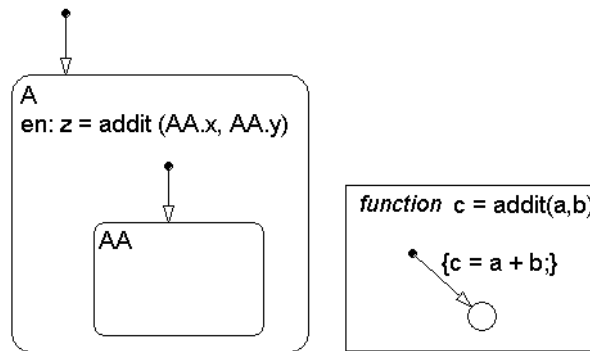
- An assignment in which the left-hand side is a data of type `m1`
For example, in the expression `m_x = m1.func()`, `m_x` is a Stateflow data of type `m1`.
- Input arguments of a MATLAB function
For example, in the expression `m1.func(m_x, m1.x, gfunc())`, `m_x` is a Stateflow data of type `m1`, `m1.x` is a MATLAB workspace variable of any size and type, and `gfunc()` is a Stateflow graphical function that returns a Stateflow data of type `m1`. Even though Stateflow does nothing to check the size of the input type, if the passed-in data is not of the expected type, an error results from the function call `m1.func()`.
- Arguments for a graphical function that are specified as Stateflow data of type `m1` in its prototype statement

Note If inputs in the preceding cases are replaced with non-MATLAB numeric Stateflow data, a conversion to an `m1` type is performed.

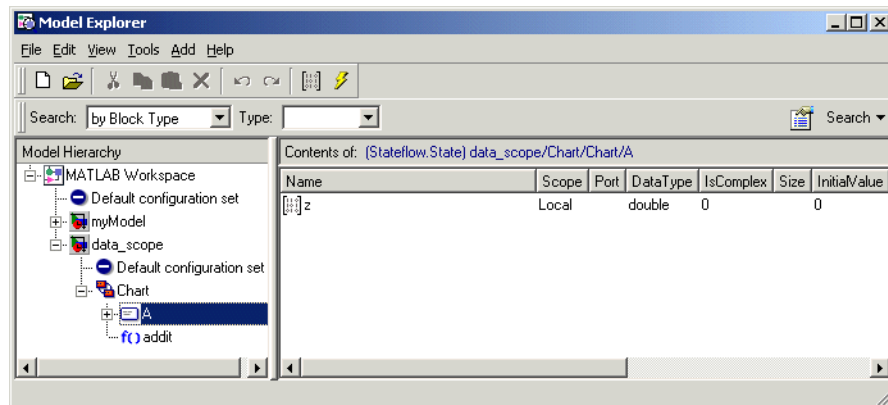
Using Data and Event Arguments in Actions

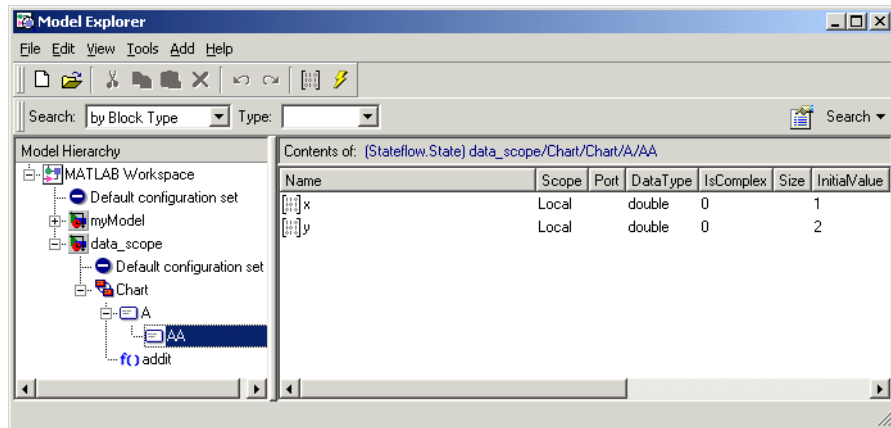
When you use data and event objects as arguments to functions that you call in action language, they are assumed to be defined at the same level in the hierarchy as the action language that references them. If they are not found at that level, Stateflow attempts to resolve the object name by searching up the hierarchy. Data or event object arguments that are parented anywhere else must have their path hierarchies defined explicitly.

In the following example, state A calls the graphical function `addit` to add the Stateflow data `x` and `y` and store the result in data `z`.



The following **Model Explorer** windows show the data `z` is defined for state **A**, but the data `x` and `y` are defined for state **AA**, a substate of **A**.





The call to function `addit` from state `A` can resolve `z` because it is owned by `A`. However, it cannot resolve `x` and `y` by looking above state `A`. Therefore, the function call must reference `x` and `y` explicitly to their owner, state `AA`.

There are a variety of functions that you can call in Stateflow action language that use data as arguments. See the following sections:

- “Using Functions to Extend Actions” on page 5-29
- “Calling C Functions in Actions” on page 7-22
- “Using MATLAB Functions and Data in Actions” on page 7-27

Only the temporal logic operators take events as an argument. See “Using Temporal Logic in Actions” on page 7-50.

Using Arrays in Actions

This section tells you how to use Stateflow data arrays in action language. See the following topics:

- “Array Notation” on page 7-42 — Gives examples of array notation in action language.
- “Arrays and Custom Code” on page 7-43 — Shows you how to access arrays provided by custom code that you build into a Stateflow target.

Array Notation

Use C style syntax in the action language to access array elements.

```
local_array[1][8][0] = 10;  
  
local_array[i][j][k] = 77;  
  
var = local_array[i][j][k];
```

As an exception to this style, **scalar expansion** is available within the action language. This statement assigns a value of 10 to all the elements of the array `local_array`.

```
local_array = 10;
```

Scalar expansion is available for performing general operations. This statement is valid if the arrays `array_1`, `array_2`, and `array_3` have the same value for the **Sizes** property.

```
array_1 = (3*array_2) + array_3;
```

Note Use the same notation for accessing arrays in Stateflow, from Simulink, and from custom code.

Arrays and Custom Code

Stateflow action language provides the same syntax for Stateflow arrays and custom code arrays.

See also “Integrating Custom Code with Stateflow Targets” on page 12–29.

Note Any array variable that is referred to in a Stateflow chart but is not defined in the data dictionary is identified as a custom code variable.

Broadcasting Events in Actions

You can specify an event to be broadcast in the action language. Events have hierarchy (a parent) and scope. The parent and scope together define a range of access to events. It is primarily the event's parent that determines who can trigger on the event (has receive rights). See “Name” on page 6-7 for more information.

See the following sections for an understanding of broadcasting events in action language:

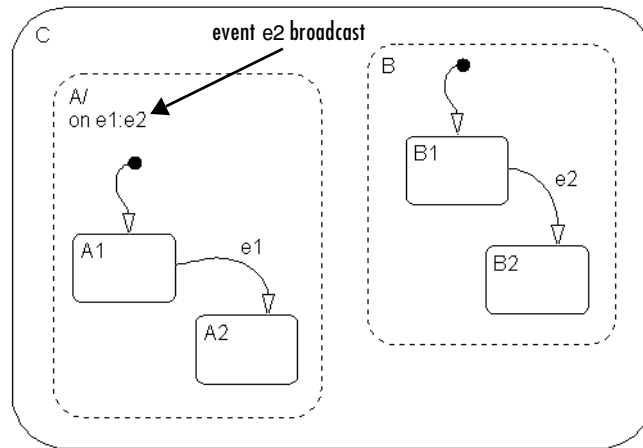
- “Event Broadcasting” on page 7-44 — Gives examples of using broadcast events to synchronize behavior between AND (parallel) states.
- “Directed Event Broadcasting” on page 7-46 — Shows you how to use the send function to send an event to a specific state.

Event Broadcasting

Broadcasting an event in the action language is most useful as a means of synchronization among AND (parallel) states. Recursive event broadcasts can lead to definition of cyclic behavior. Cyclic behavior can be detected only during simulation.

Event Broadcast State Action Example

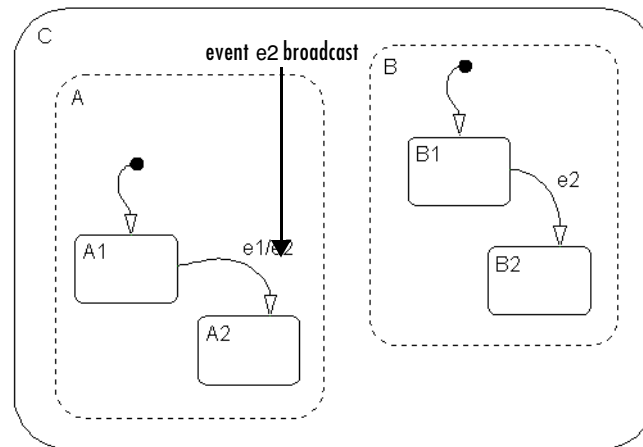
The following is an example of the state action notation for an event broadcast:



See “Event Broadcast State Action Example” on page 3-73 for information on the semantics of this notation.

Event Broadcast Transition Action Example

The following is an example of transition action notation for an event broadcast.



See “Event Broadcast Transition Action with a Nested Event Broadcast Example” on page 3-77 for information on the semantics of this notation.

Directed Event Broadcasting

You can specify a directed event broadcast in actions. Using a directed event broadcast, you can broadcast a specific event to a specific receiver state in the same chart. The receiving state must be active at the time the broadcast is executed to receive and potentially act on the directed event broadcast.

Directed event broadcasting is a more efficient means of synchronization among parallel (AND) states. Using directed event broadcasting improves the efficiency of the generated code. As is true in event broadcasting, recursive event broadcasts can lead to definition of cyclic behavior.

Note An action in one chart cannot broadcast events to states defined in another chart.

Directed Event Broadcasting Using send

The format of the directed event broadcast with `send` is as follows:

```
send(event_name, state_name)
```

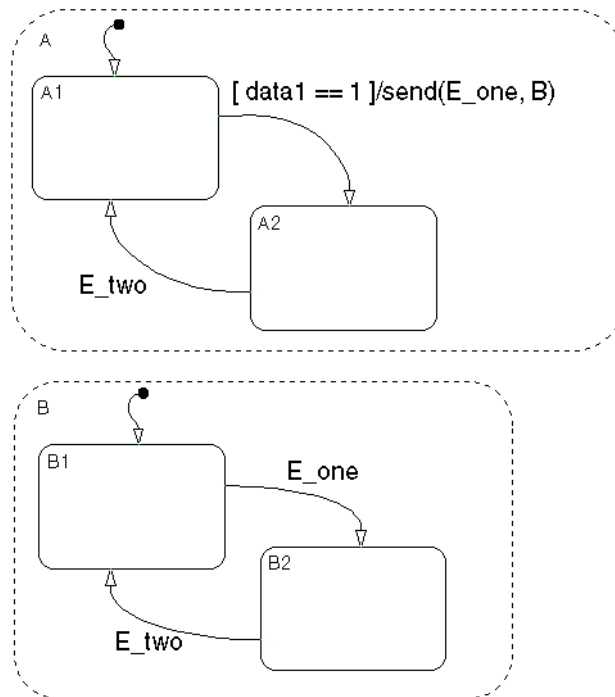
where `event_name` is broadcast to `state_name` and any offspring of that state in the hierarchy. The event sent must be visible to both the sending state and the receiving state (`state_name`).

The `state_name` argument can include a full hierarchy path to the state. For example, if the state `A` contains the state `A1`, send an event `e` to state `A1` with the following broadcast:

```
send(e, A.A1)
```


Note Do not use the chart name in the full hierarchy path to a state. Formal chart names include the subsystem they are in. For example, in the demo model `fuelsys` the chart `control logic` is in the subsystem `fuel rate controller`. This means that the formal name for the chart `control logic` is `fuel rate controller/control logic`. This name includes the forward slash character (`/`), which is not a valid character in Stateflow identifiers.

This is an example of a directed event broadcast using the `send(event_name, state_name)` transition action as a transition action.



In this example, event `E_one` must be visible in both A and B. See “Directed Event Broadcast Using `send` Example” on page 3-85 for information on the semantics of this notation.

Directed Event Broadcasting Using Qualified Event Names

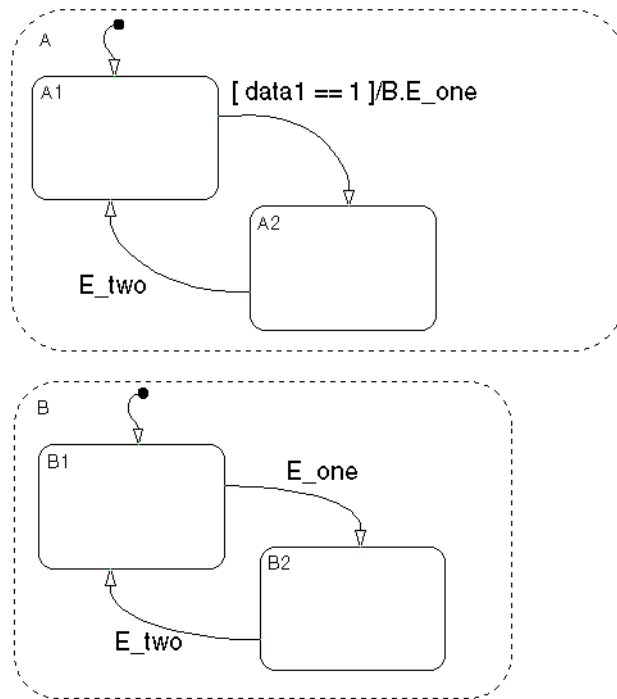
The format of the direct event broadcast using qualified event names is as follows:

```
state_name.event_name
```

where event_name is broadcast to its owning state (state_name) and any offspring of that state in the hierarchy. The event sent is visible only to the receiving state (state_name).

The state_name argument can also include a full hierarchy path to the receiving state. Again, do not use the chart name in the full path name of the state.

The following example illustrates the use of a qualified event name in a directed event broadcast.



In this example, event E_one is visible only to state B. See “Directed Event Broadcasting Using Qualified Event Names Example” on page 3-87 for information on the semantics of this notation.

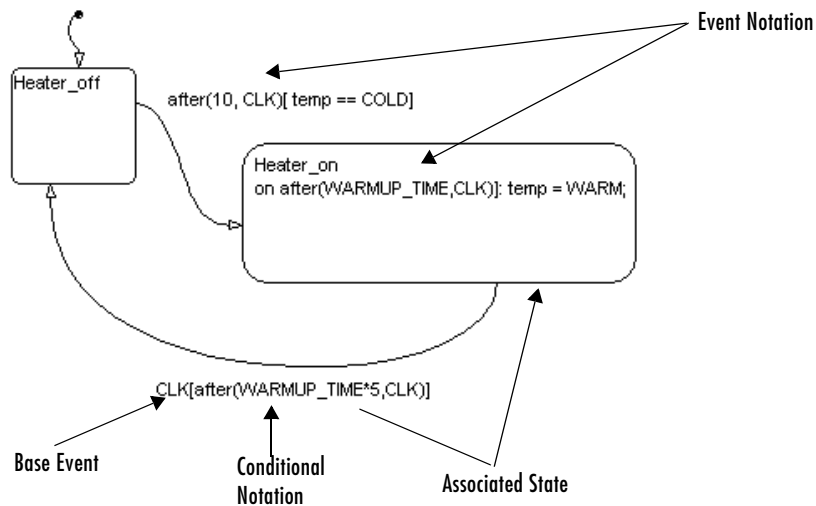
Using Temporal Logic in Actions

Temporal logic operators are Boolean operators that operate on recurrence counts of Stateflow events. See the following subsections for individual descriptions of each temporal logic operator:

- “Rules for Using Temporal Logic Operators” on page 7-50 — Gives you the rules for using temporal logic operators.
- “after Temporal Logic Operator” on page 7-52 — Shows you how to use the after temporal logic operator.
- “before Temporal Logic Operator” on page 7-53 — Shows you how to use the before temporal logic operator.
- “at Temporal Logic Operator” on page 7-54 — Shows you how to use the at temporal logic operator.
- “every Temporal Logic Operator” on page 7-55 — Shows you how to use the every temporal logic operator.
- “Conditional and Event Notation” on page 7-56 — Describes the two types of notation you can use for temporal logic operations in action language.
- “Temporal Logic Events” on page 7-57 — Shows you how you temporal logic is a single event equivalent to accumulated base events.

Rules for Using Temporal Logic Operators

The following diagram illustrates the use of temporal logic operators in action language:



The following rules apply generally to the use of temporal logic operators:

- The recurring event on which a temporal operator operates is called the *base event*. Any Stateflow event can serve as a base event for a temporal operator.

Note Temporal logic operators can also operate on recurrences of implicit events, such as state entry or exit events or data change events. See “Defining Implicit Events” on page 6-18.

- For a chart with no Simulink input events, you can use the wakeup (or tick) event to denote the implicit event of a chart waking up.
- Temporal logic operators can appear only in conditions on transitions originating from states and in state actions.

Note This means you cannot use temporal logic operators as conditions on default transitions or flow graph transitions.

The state on which the temporally conditioned transition originates or in whose during action the condition appears is called the temporal operator's *associated state*.

- You must use event notation (see “Temporal Logic Events” on page 7-57) to express temporal logic conditions on events in state during actions.

after Temporal Logic Operator

The syntax for the after operator is as follows:

```
after(n, E)
```

where E is the base event for the after operator and n is one of the following:

- A constant integer greater than 0
- An expression that evaluates to an integer value greater than or equal to 0

The after operator is true if the base event E has occurred n times since activation of its associated state. Otherwise, it is false. In a chart with no input events, `after(n, wakeup)` (or `after(n, tick)`) evaluates to true after the chart has woken up n times.

Note The after operator resets its counter for E to 0 each time the associated state is activated.

The following example illustrates use of the after operator in a transition expression.

```
CLK[after(10, CLK) && temp == COLD]
```

This example permits a transition out of the associated state only if there have been 10 occurrences of the CLK event since the state was activated and the temp data item has the value COLD.

The next example illustrates usage of event notation for temporal logic conditions in transition expressions.

```
after(10, CLK)[temp == COLD]
```

This example is semantically equivalent to the first example.

The next example illustrates setting a transition condition for any event visible in the associated state while it is activated.

```
[after(10, CLK)]
```

This example permits a transition out of the associated state on any event after 10 occurrences of the CLK event since activation of the state.

The next two examples underscore the semantic distinction between an after condition on its own base event and an after condition on a nonbase event.

```
CLK[after(10,CLK)]
ROTATE[after(10,CLK)]
```

The first expression says that the transition must occur *as soon as* 10 CLK events have occurred after activation of the associated state. The second expression says that the transition can occur *no sooner than* 10 CLK events after activation of the state, but possibly later, depending on when the ROTATION event occurs.

The next example illustrates usage of an after event in a state's during action.

```
Heater_on
on after(5*BASE_DELAY, CLK): status('heater on');
```

This example causes the Heater_on state to display a status message each CLK cycle, starting 5*BASE_DELAY clock cycles after activation of the state. Note the use of event notation to express the after condition in this example. Use of conditional notation is not allowed in state during actions.

before Temporal Logic Operator

The before operator has the following syntax:

```
before(n, E)
```

where E is the base event for the before operator and n is one of the following:

- A constant integer greater than 0
- An expression that evaluates to an integer value greater than or equal to 0

The `before` operator is true if the base event `E` has occurred less than `n` times since activation of its associated state. Otherwise, it is false. In a chart with no input events, `before(n, wakeup)` or `before(n, tick)` evaluates to true before the chart has woken up `n` times.

Note The `before` operator resets its counter for `E` to 0 each time the associated state is activated.

The following example illustrates the use of the `before` operator in a transition expression.

```
ROTATION[before(10, CLK)]
```

This expression permits a transition out of the associated state only on occurrence of a `ROTATION` event but *no later than* 10 CLK cycles after activation of the state.

The next example illustrates usage of a `before` event in a state's during action.

```
Heater_on  
on before(MAX_ON_TIME, CLK): temp++;
```

This example causes the `Heater_on` state to increment the `temp` variable once per CLK cycle until the `MAX_ON_TIME` limit is reached.

at Temporal Logic Operator

The syntax for the `at` operator is as follows:

```
at(n, E)
```

where `E` is the base event for the `at` operator and `n` is one of the following:

- A constant integer greater than 0
- An expression that evaluates to an integer value greater than or equal to 0

The `at` operator is true only at the n^{th} occurrence of the base event `E` since activation of its associated state. Otherwise, it is false. In a chart with no input events, `at(n, wakeup)` (or `at(n, tick)`) evaluates to true when the chart wakes up for the n^{th} time.

Note The `at` operator resets its counter for `E` to 0 each time the associated state is activated.

The following example illustrates the use of the `at` operator in a transition expression.

```
ROTATION[at(10, CLK)]
```

This expression permits a transition out of the associated state only if a `ROTATION` event occurs *exactly* 10 `CLK` cycles after activation of the state.

The next example illustrates usage of an `at` event in a state's during action.

```
Heater_on
on at(10, CLK): status('heater on');
```

This example causes the `Heater_on` state to display a status message 10 `CLK` cycles after activation of the associated state.

every Temporal Logic Operator

The syntax for the `every` operator is as follows:

```
every(n, E)
```

where `E` is the base event for the `every` operator and `n` is one of the following:

- A constant integer greater than 0
- An expression that evaluates to an integer value greater than or equal to 0

The `every` operator is true at every n^{th} occurrence of the base event `E` since activation of its associated state. Otherwise, it is false. In a chart with no input events, `every(n, wakeup)` (or `every(n, tick)`) evaluates to true whenever the chart wakes up an integer multiple `n` times.

Note The every operator resets its counter for E to 0 each time the associated state is activated. As a result, this operator is useful only in state during actions.

The following example illustrates the use of the every operator in a state.

```
Heater_on
  on every(10, CLK): status('heater on');
```

This example causes the Heater_on state to display a status message every 10 CLK cycles after activation of the associated state.

Conditional and Event Notation

Stateflow treats the following notations as equivalent,

```
E[tlo(n, E) && C]
tlo(n, E)[C]
```

where *tlo* is a temporal logic operator (after, before, at, every), E is the operator's base event, n is the operator's occurrence count, and C is any conditional expression. For example, the following expressions are functionally equivalent in Stateflow:

```
CLK[after(10, CLK) && temp == COLD]
after(10, CLK)[temp == COLD]
```

The first notation is referred to as the conditional notation for temporal logic operators and the second notation as the event notation.

Note You can use conditional and event notation interchangeably in transition expressions. However, you must use the event notation in state during actions.

Temporal Logic Events

Although temporal logic does not introduce any new events into a Stateflow model, it is useful to think of the change of value of a temporal logic condition as an event. For example, suppose that you want a transition to occur from state A exactly 10 clock cycles after activation of the state. One way to achieve this would be to define an event called ALARM and to broadcast this event 10 CLK events after state A is entered. You would then use ALARM as the event that triggers the transition out of state A.

An easier way to achieve the same behavior is to set a temporal logic condition on the CLK event that triggers the transition out of state A.

```
CLK[after(10, CLK)]
```

Note that this approach does not require creation of any new events. Nevertheless, conceptually it is useful to think of this expression as equivalent to creation of an implicit event that triggers the transition. Hence, Stateflow supports the equivalent event notation (see “Temporal Logic Events” on page 7-57).

```
after(10, CLK)
```

Note that the event notation allows you to set additional constraints on the implicit temporal logic “event,” for example,

```
after(10, CLK)[temp == COLD]
```

This expression says, “Exit state A if the temperature is cold but no sooner than 10 clock cycles.”

Using Bind Actions to Control Function-Call Subsystems

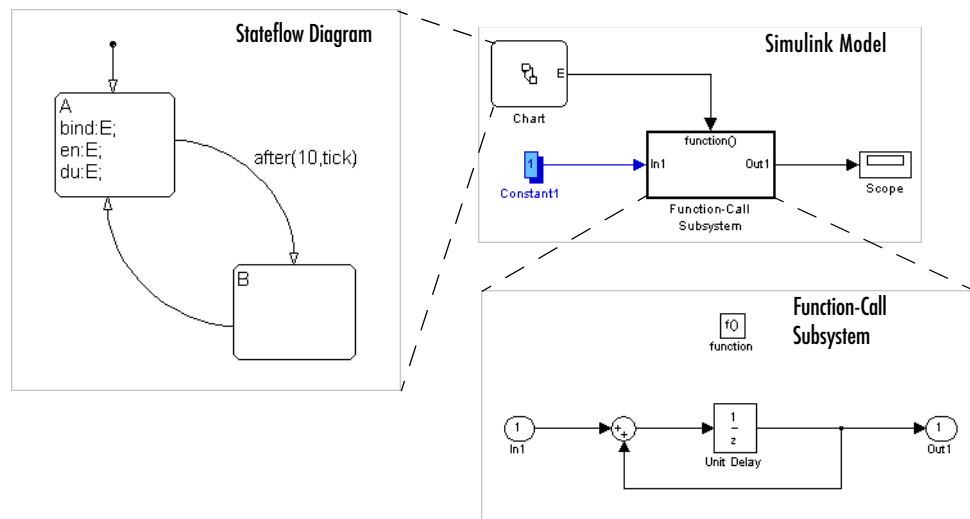
Bind actions in a state bind specified data and events to that state. Events bound to a state can be broadcast only by the actions in that state or its children. You can also bind a function-call event to a state to add enabling and disabling control of the function-call subsystem that it triggers. The function-call subsystem enables when the state with the bound event is entered and disables when that state is exited. This means that the execution of the function-call subsystem is fully bound to the activity of the state that calls it.

Examine the effects of binding a function-call subsystem trigger event in the following topics:

- “Binding a Function-Call Subsystem” on page 7-58
- “Simulating a Bound Function-Call Subsystem” on page 7-60
- “Avoiding Muxed Trigger Events with Binding” on page 7-68


Binding a Function-Call Subsystem

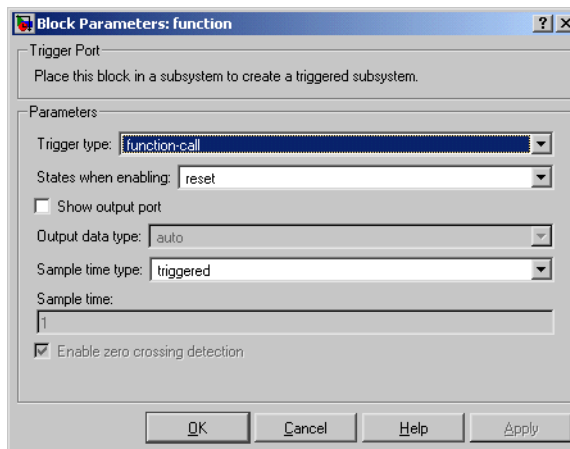
The control of a Stateflow state that binds a function-call subsystem trigger is best understood through the creation and execution of an example model. In the following example, a Simulink model triggers a function-call subsystem with a function-call trigger event E bound to state A of a Stateflow diagram.



The function subsystem contains a trigger port block, an input port, an output port, and a simple block diagram. The block diagram increments a count by 1 each time, using a Unit Delay block to store the count.

The Stateflow diagram contains two states, A and B, and connecting transitions, along with some actions. Notice that state A binds and event E to itself with the binding action `bind:E`. Event E is defined for the Stateflow diagram in the example with a scope of **Output to Simulink** and a trigger type of **Function Call**.

Double-click the trigger port block for the function-call subsystem  to display the **Block Parameters** dialog for the trigger port as shown.



Notice that the **States when enabling** field is set to the default value **reset**. This resets the state values for the function-call subsystem to zero when it is enabled. The alternative value for this field, **held**, tells the function-call subsystem to retain its state values when it is enabled. This feature, when coupled with state binding, gives full control of state variables for the function-call subsystem to the binding state.

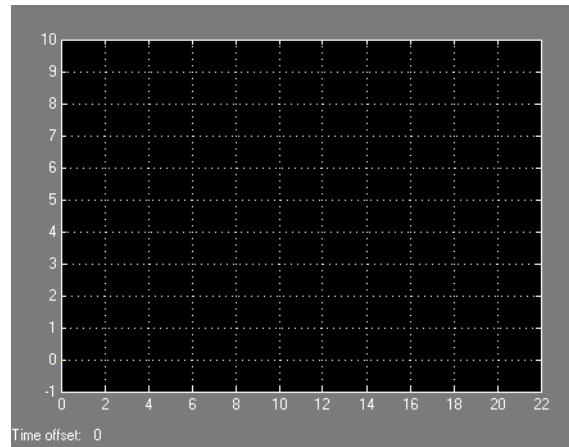
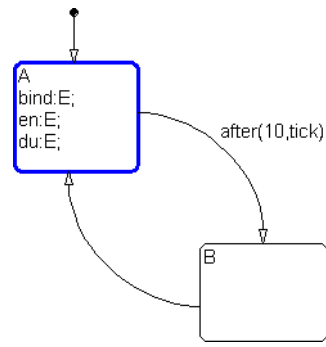
Notice also that the **Sample time type** field is set to the default value **triggered**. This sets the function-call subsystem to execute only when it is triggered by a calling event while it is enabled.

Setting **Sample time type** to **periodic** enables the **Sample time** field below it, which defaults to 1. These settings force the function-call subsystem to execute for each time step specified in the **Sample time** field while it is enabled. To accomplish this, the state that binds the calling event for the function-call subsystem must send an event for the time step coinciding with the specified sampling rate in the **Sample time** field. States can send events with entry or during actions at the simulation sample rate. Therefore, for fixed step sampling, the sample time you enter in the **Sample time** field must be an integer multiple of the fixed step size. For variable step sampling, there are no limits on what you enter in the **Sample time** field.

Simulating a Bound Function-Call Subsystem

To see the control that a state can have over the function-call subsystem whose trigger event it binds, begin simulating the example model in “Binding a Function-Call Subsystem” on page 7-58. For the purposes of display, the simulation parameters for this model specify a fixed-step solver with a fixed-step size of 1. Take note of model behavior in the following steps, which record the simulating Stateflow diagram and the output of the subsystem.

- 1** The default transition to state A is taken.
- 2** State A becomes active as shown.

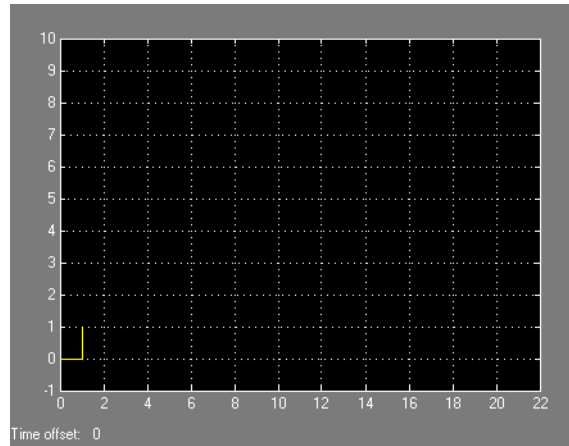
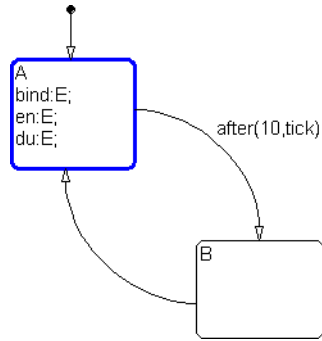


When state A becomes active, it executes its bind and entry actions. The binding action, `bind:E`, binds event E to state A. This enables the function-call subsystem and resets its state variables to 0.

State A also executes its entry action, `en:E`, which sends an event E to trigger the function-call subsystem and execute its block diagram. The block diagram increments a count by 1 each time using a Unit Delay block. Since the previous content of the Unit Delay block is 0 after the reset, the starting output point is 0 and the current value of 1 is held for the next call to the subsystem.

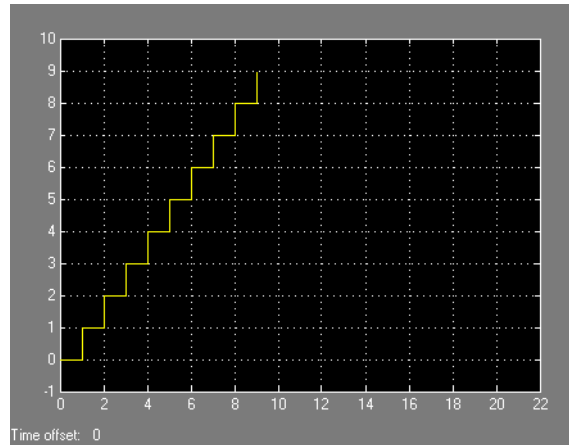
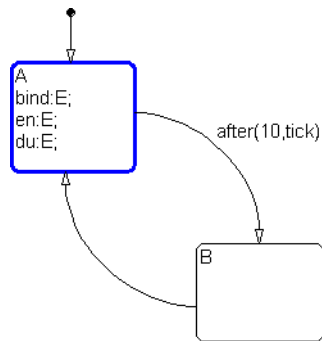
- 3 The next update event from Simulink tests state A for an outgoing transition.

The temporal operation on the transition to state B, `after(10, tick)`, allows the transition to be taken only after ten update events are received. This means that for the second update, the during action of state A, `du:E`, is executed, which sends an event to trigger the function-call subsystem. The held content of the Unit Delay block, 1, is output to the scope as shown.

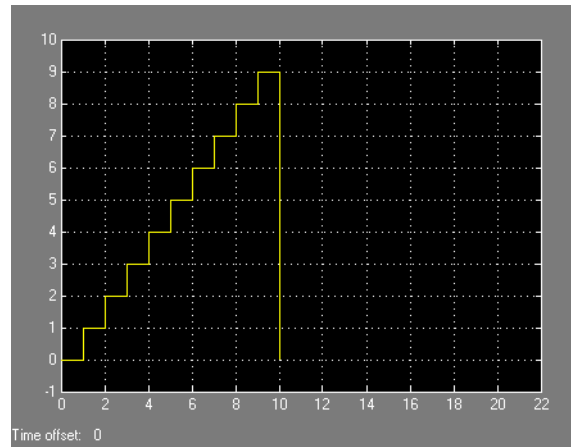
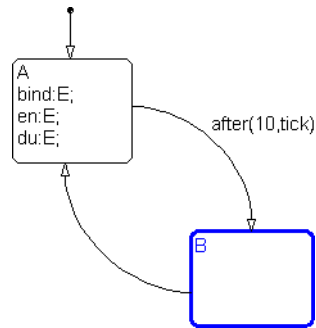


The subsystem also adds 1 to the held value to produce the value 2, which is held by the Unit Delay block for the next triggered execution.

- 4 The next eight update events repeat step 2, which increment the subsystem output by 1 each time as shown.



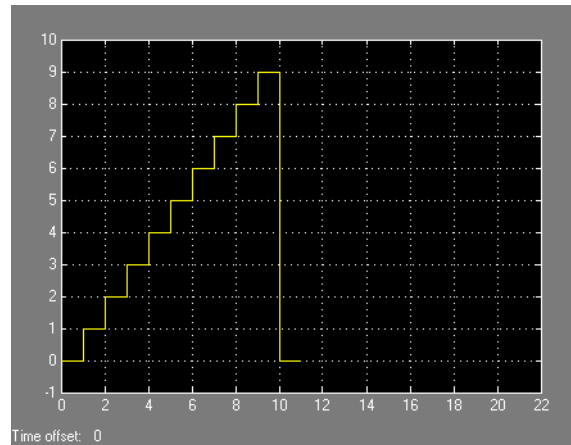
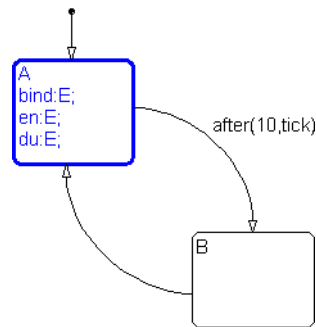
- 5 On the 11th update event, the transition to state B is taken as shown.



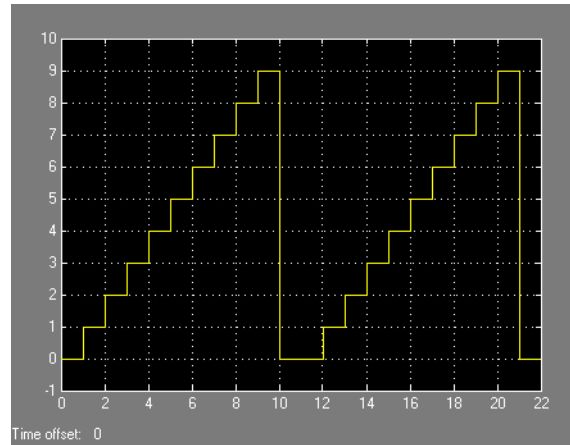
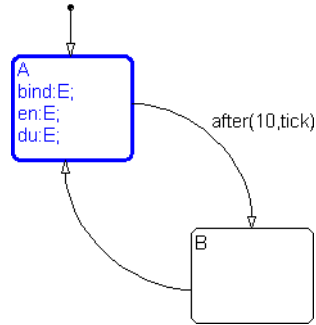
This makes state B active. Since the binding state A is no longer active, the function-call subsystem is disabled, and its output drops to 0.

- When the next sampling event occurs, the transition from state B to state A is taken.

Once again, the binding action, bind: E, enables the subset and resets its output to 0 as shown.

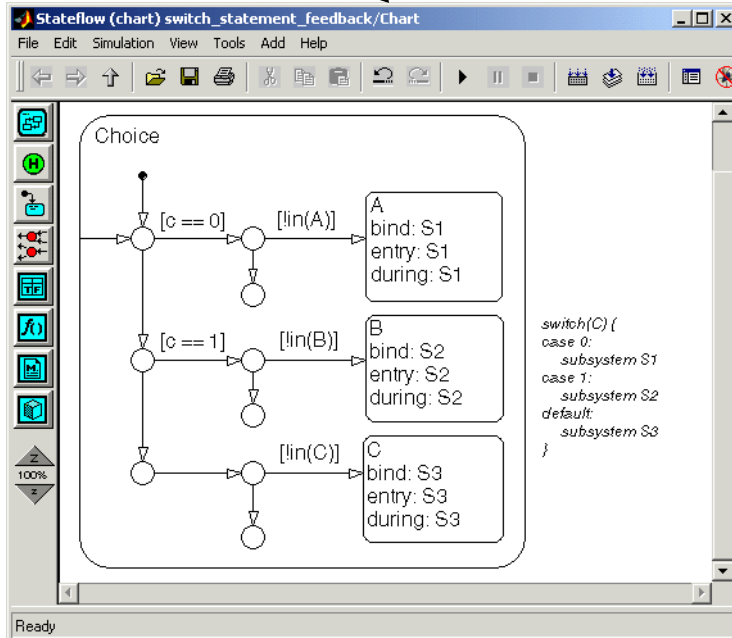
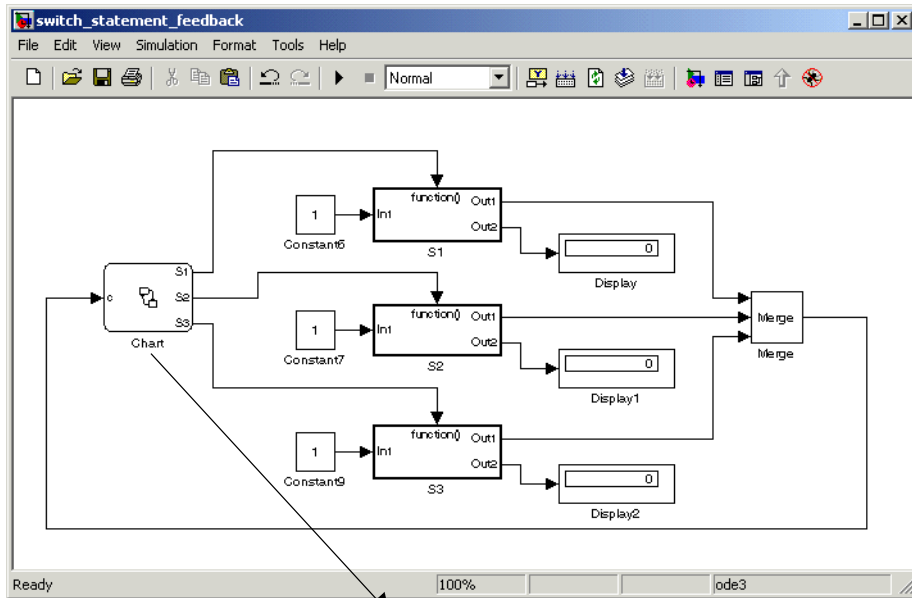


- 7 With the next 10 update events, steps 2 through 5 repeat, producing the following output:



Using Stateflow Logic with Binding

You can use Stateflow logic to control function-call subsystems to compose C-like switch, if-else, for, and while statements in Simulink. For example, the following Simulink model demonstrates a Simulink switch statement with subsystems controlled by bind actions:



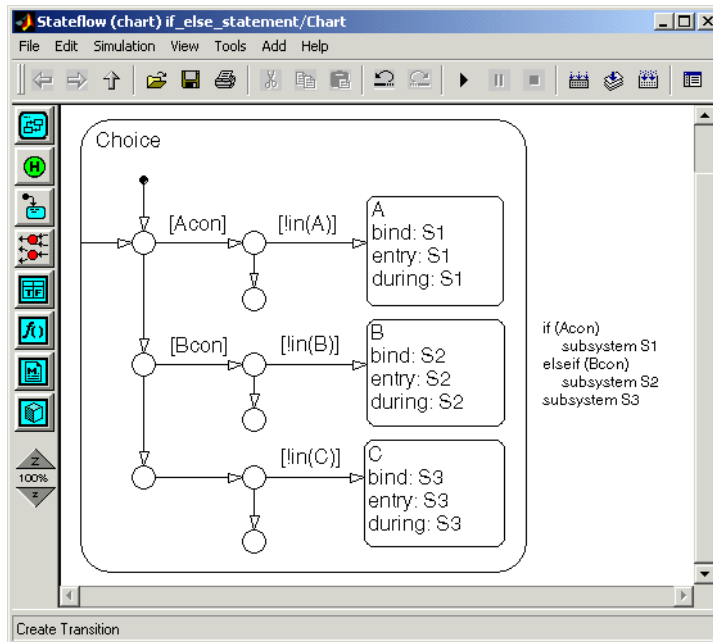
The preceding Stateflow diagram controls three subsystems, S1, S2, and S3, through the bind actions for three states, A, B, and C, respectively. In this example, the value of the case argument *c* is used to determine the subsystem to execute. State A becomes active and stays active when *c* is 0. State B becomes active and stays active when *c* is 1. State C becomes active and stays active when *c* has any other value.

When state A is active, the event S1 is bound to state A, which enables subsystem S1. The entry and during actions for A broadcast the event S1 whenever the model is updated for sampling. This means that while A is active, the subsystem S1 is executed for each sample time. The same applies to subsystem S2 for state B, and to subsystem S3 for state C. This creates the following if-else statement in Simulink:

This creates the following switch statement in Simulink:

```
switch(c)
case 0:
    subsystem S1
case 1:
    subsystem S2
default:
    subsystem S3
```

You can modify the previous Stateflow diagram to control a Simulink model with an if-else statement, as shown.



In this example, State A becomes active and stays active when the condition Acon is true. State B becomes active and stays active when the condition Bcon is true and the condition Acon is false. State C becomes active and stays active when both conditions Acon and Bcon are false. This creates the following if-else statement in Simulink:

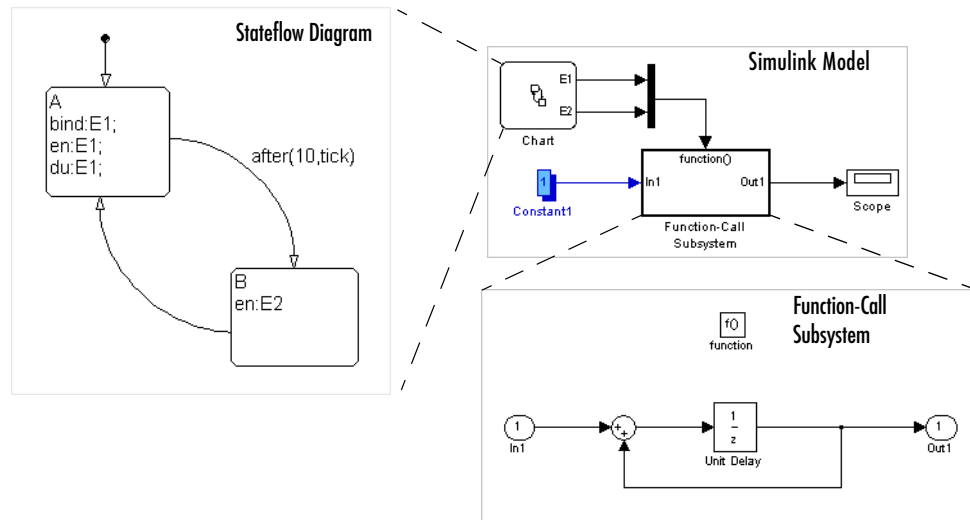
```

if (Acon)
    subsystem S1
elseif (Bcon)
    subsystem S2
subsystem S3

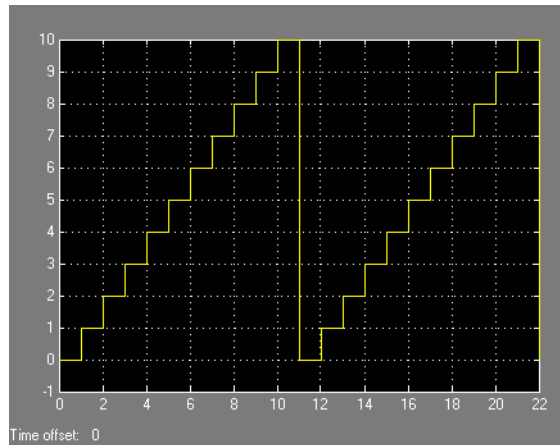
```

Avoiding Muxed Trigger Events with Binding

The simulated example in “Simulating a Bound Function-Call Subsystem” on page 7-60 shows how binding events gives control of a function-call subsystem to a single state in a Stateflow diagram. This control can be undermined if you allow other events to trigger the function-call subsystem through a mux. For example, the following Simulink diagram defines two function-call events to trigger a function-call subsystem through a mux:



In the Stateflow diagram, E1 is bound to state A, but E2 is not. This means that state B is free to send the triggering event E2 in its entry action. When you simulate this model, you receive the following output:



Notice that broadcasting E2 in state B changes the output, which now rises to a height of 10 before the binding action in state A resets the data.

Note Binding is not recommended when users provide multiple trigger events to a function-call subsystem through a mux. Muxed trigger events can interfere with event binding and cause undefined behavior.

Using Fixed-Point Data in Stateflow

Fixed-point numbers use integers and integer arithmetic to approximate real numbers. They are an efficient means for performing computations involving real numbers without requiring floating-point support in underlying system hardware. The following topics describe fixed-point data in Stateflow. Be sure to read “Tips for Using Fixed-Point Data in Stateflow” on page 8-9 when you are ready to begin using it.

What Is Fixed-Point Data? (p. 8-2)

Provides a general discussion of the underlying arithmetic of fixed-point numbers.

Using Fixed-Point Data in Stateflow (p. 8-5)

Shows you how to define and use fixed-point data in Stateflow

Fixed-Point “Bang-Bang Control” Example (p. 8-14)

Examines a Stateflow demo that uses fixed-point data in Stateflow to perform limited floating-point operations on an 8 bit processor

Operations with Fixed-Point Data (p. 8-18)

Lists the operations supported for Stateflow fixed-point data and the means by which they are implemented in Stateflow generated code

What Is Fixed-Point Data?

This section presents a high-level overview of fixed-point arithmetic. Use this section to understand the theory of fixed-point numbers and their operations with the following topics:

- “Fixed-Point Numbers” on page 8-2
- “Fixed-Point Operations” on page 8-3

For more discussion on the theory of fixed-point numbers, see “Fixed-Point Numbers” in the Fixed-Point Blockset documentation.

Fixed-Point Numbers

Fixed-point numbers use integers and integer arithmetic to represent real numbers and arithmetic with the following encoding scheme:

$$V \approx \tilde{V} = SQ + B$$

where

- V is a precise real-world value that you want to approximate with a fixed-point number.
- \tilde{V} is the approximate real-world value that results from fixed-point representation.
- Q is an integer that encodes \tilde{V} . It is referred to as the *quantized integer*. Q is the actual stored integer value used in representing the fixed-point number; that is, if a fixed-point number changes, its quantized integer, Q , changes — S and B remain unchanged.
- S is a coefficient of Q referred to as the *slope*.
- B is an additive correction referred to as the *bias*.

Fixed-point numbers encode real quantities (for example, 15.375) using the stored integer Q . You set Q 's value by solving the preceding equation $V = SQ + B$ for Q and rounding the result to an integer value as follows:

$$Q = \text{round}((V - B)/S)$$

For example, suppose you want to represent the number 15.375 in a fixed-point type with the slope $S = 0.5$ and the bias $B = 0.1$. This means that

$$Q = \text{round}((15.375 - 0.1)/0.5) = 30$$

However, because Q is rounded to an integer, you have lost some precision in representing the number 15.375. If you calculate the number that Q actually represents, you now get a slightly different answer.

$$V \approx \tilde{V} = SQ + B = 0.5 \cdot 30 + 0.1 = 15.1$$

So using fixed-point numbers to represent real numbers with integers involves the loss of some precision. However, if you choose S and B correctly, you can minimize this loss to acceptable levels.

Fixed-Point Operations

Now that you can express fixed-point numbers as $\tilde{V} = SQ + B$, you can define operations between two fixed-point numbers.

The general equation for an operation between fixed-point operands is as follows:

$$c = a \text{ <op> } b$$

where a , b , and c are all fixed-point numbers, and <op> refers to one of the binary operations: addition, subtraction, multiplication, or division.

The general form for a fixed-point number x is $S_x Q_x + B_x$ (see “Fixed-Point Numbers” on page 8-2). Substituting this form for the result and operands in the preceding equation yields the following:

$$(S_c Q_c + B_c) = (S_a Q_a + B_a) \text{ <op> } (S_b Q_b + B_b)$$

The values for S_c and B_c are usually chosen by Stateflow for each operation (see “Promotion Rules for Fixed-Point Operations” on page 8-21) and are based on the values for S_a , S_b , B_a , and B_b , which are entered for each fixed-point data (see “Specifying Fixed-Point Data in Stateflow” on page 8-7).

Note Stateflow also offers a more precise means for choosing the values for S_c and B_c when you use the $:=$ assignment operator (that is, $c := a \text{ <op> } b$). See “Assignment ($=$, $:=$) Operations” on page 8-31 for more detail.

Using the values for S_a , S_b , S_c , B_a , B_b , and B_c , you can solve the preceding equation for Q_c for each binary operation as follows:

- The operation $c=a+b$ implies that
$$Q_c = ((S_a/S_c)Q_a + (S_b/S_c)Q_b + (B_a + B_b - B_c)/S_c)$$
- The operation $c=a-b$ implies that
$$Q_c = ((S_a/S_c)Q_a - (S_b/S_c)Q_b - (B_a - B_b - B_c)/S_c)$$
- The operation $c=a*b$ implies that
$$Q_c = ((S_a S_b/S_c)Q_a Q_b + (B_a S_b/S_c)Q_a + (B_b S_a/S_c)Q_b + (B_a B_b - B_c)/S_c)$$
- The operation $c=a/b$ implies that
$$Q_c = ((S_a Q_a + B_a)/(S_c(S_b Q_b + B_b)) - (B_c/S_c))$$

The fixed-point approximations of the real number result of the operation $c = a \text{ <op> } b$ are given by the preceding solutions for the value Q_c . In this way, all fixed-point operations are performed using only the stored integer Q for each fixed-point number and integer operation.

Using Fixed-Point Data in Stateflow

In “What Is Fixed-Point Data?” on page 8-2 you learn the theory behind fixed-point numbers. In this section you learn how Stateflow implements these numbers in the following topics:

- “How Stateflow Defines Fixed-Point Data” on page 8-5 — Describes the parameters that Stateflow uses to define fixed-point data.
- “Specifying Fixed-Point Data in Stateflow” on page 8-7 — Tells you where and how to specify fixed-point data in Stateflow.
- “Fixed-Point Context-Sensitive Constants” on page 8-8 — Tells you how to specify fixed-point constants that take their data type from the context in which they are used.
- “Tips for Using Fixed-Point Data in Stateflow” on page 8-9 — Gives you guidelines for using fixed-point data in Stateflow.
- “Overflow Detection for Fixed-Point Types” on page 8-11 — Tells you how to detect errors that result from exceeding the numeric capacity of fixed-point numbers.
- “Sharing Fixed-Point Data with Simulink” on page 8-12 — Tells you how to specify fixed-point data in Simulink that it shares with Stateflow as input from Simulink and output to Simulink data

How Stateflow Defines Fixed-Point Data

The preceding example in “What Is Fixed-Point Data?” on page 8-2 does not answer the question of how the values for the slope, S , the quantized integer, Q , and the bias, B , are implemented in Stateflow as integers. These values are implemented through the following:

- Stateflow defines a fixed-point data’s type from values that you specify. You specify values for S , B , and the base integer type for Q . The available base types for Q are the unsigned integer types `uint8`, `uint16`, and `uint32`, and the signed integer types `int8`, `int16`, and `int32`. For specific instructions on how to enter fixed-point data, see “Specifying Fixed-Point Data in Stateflow” on page 8-7.

Notice that if a fixed-point number has a slope $S = 1$ and a bias $B = 0$, it is equivalent to its quantized integer Q , and behaves exactly as its base integer type.

- Stateflow implements an integer variable for the Q value of each fixed-point data in generated code.

This is the only part of a fixed-point number that varies in value. The quantities S and B are constant and appear only as literal numbers or expressions in generated code.

- The slope, S , is factored into an integer power of two, E , and a coefficient, F , such that $S = F \cdot 2^E$ and $1 \leq F < 2$.

The powers of 2 are implemented as bit shifts, which are more efficient than multiply instructions. Setting $F = 1$ avoids the computationally expensive multiply instructions for values of $F > 1$. This is referred to as *binary-point-only* scaling, which is implemented with bit shifts only, and is highly recommended.

- Operations for fixed-point types are implemented with solutions for the quantized integer as described in “Fixed-Point Operations” on page 8-3.

To generate efficient code, the fixed-point promotion rules choose values for S_c and B_c that conveniently cancel out difficult terms in the solutions. See “Addition (+) and Subtraction (-)” on page 8-24 and “Multiplication (*) and Division (/)” on page 8-27.

Stateflow provides a special assignment operator ($:=$) and context-sensitive constants to help you maintain as much precision as possible in your fixed-point operations. See “Assignment (=, :=) Operations” on page 8-31 and “Fixed-Point Context-Sensitive Constants” on page 8-8.

- Any remaining numbers, such as the fractional slope, F , that cannot be expressed as a pure integer or a power of 2, are converted into fixed-point numbers.

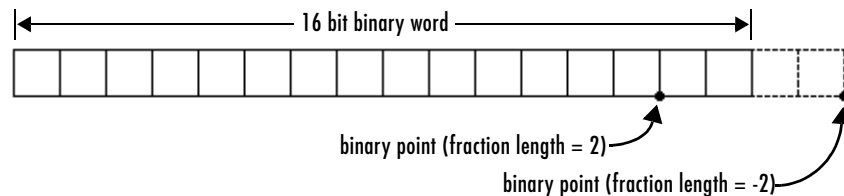
These remaining numbers can be computationally expensive in multiplication and division operations. That is why the practice of using binary-point-only scaling in which $F = 1$ and $B = 0$ is recommended.

- During simulation, Stateflow detects when the result of a fixed-point operation *overflows* the capacity of its fixed-point type. See “Overflow Detection for Fixed-Point Types” on page 8-11.

Specifying Fixed-Point Data in Stateflow

You can specify fixed-point data in Stateflow as follows:

- 1 Add data to Stateflow as described in “Adding Data to the Data Dictionary” on page 6-21.
- 2 Set the properties for the data in the data properties dialog as described in “Setting Data Properties in the Data Dialog” on page 6-25. For fixed-point data, set the following fields:
 - In the **Type** field, select **fixpt**.
This enables the Fixed-Point field **Stored Integer** and optional fields **Fraction length** and **Scaling** directly below the **Type** field.
 - In the **Stored Integer** field, select the desired integer type to hold Q .
Choose between the unsigned integer types `uint8`, `uint16`, and `uint32`, and the signed integer types `int8`, `int16`, `int32`.
 - If you want binary-point-only scaling for a fixed-point data (see note below), select the optional **Fraction length** field. You specify the number of bits to the right of the binary point in this fixed-point data by entering an integer value in the field to the right (noninteger values that you enter for **Fraction length** are flagged immediately with red text). A positive value moves the binary point left of the rightmost bit (least significant bit) by that amount. A negative value moves the binary point further right of the rightmost bit by that amount.



- If you want to enter separate slope and bias values (see note below), select the **Scaling** option. In the field to the right, enter a two-element array, that is, [s b], in which the first element, s, is the slope, and the second element, b, is the bias. The slope must be greater than zero (values that you enter for **Fraction length** that are less than 0 are flagged immediately with red text).

For example, to specify the fixed-point data $2.4*Q + 3.4$, enter the slope and bias as [2.4 3.4] in the **Scale** field. You can also enter the slope or bias as an arithmetic expression such as [1.2* -2^1 3.4].

Note It is recommended that you use binary-point-only scaling whenever possible to simplify the implementation of fixed-point data in generated code. Operations with fixed-point data using binary-point-only scaling are performed with simple bit shifts and eliminate the expensive code implementations required for separate slope and bias values.

You can also specify a fixed-point constant indirectly in action language by using a fixed-point context-sensitive constant. See “Fixed-Point Context-Sensitive Constants” on page 8-8.

Fixed-Point Context-Sensitive Constants

You can conveniently use fixed-point constants without using the data properties dialog or Stateflow Explorer, by using context-sensitive constants. Context-sensitive constants are constants that infer their types from the context in which they occur. They are written like ordinary constants, but have the suffix C or c. For example, the numbers 4.3C and 123.4c are valid fixed-point context-sensitive constants you can use in action language operations.

The following rules apply to context-sensitive constants:

- If any type in the context is a double then the context-sensitive constant is cast to type double.
- In an addition or subtraction operation, the type of the context-sensitive constant is the type of the other operand.

- In a multiplication or division operation with a fixed-point number, they obtain the best possible precision for a fixed-point result.
In the Fixed-Point Blockset in Simulink, this functionality is provided by the function `fixptbestexp`.
- In a cast, the context is the type to which the constant is being cast.
- As an argument in a function call, the context is the type of the formal argument. In an assignment, the context is the type of the left-hand operand.
- Context-sensitive constants may not be used on the left-hand side of an assignment.
- Both operands of a binary operation cannot be context-sensitive constants.

Note Both operands of a binary operation cannot be context-sensitive constants.

While fixed-point context-sensitive constants can be used in context with any types (for example, `int32` or `double`), the primary motivation for using them is with fixed-point numbers. The algorithm that computes the type to assign to a fixed-point context-sensitive constant depends on the operator, the types in the context, and the value of the constant. It provides a “natural” type, providing maximum accuracy without overflow.

Tips for Using Fixed-Point Data in Stateflow

Once you specify fixed-point data (see “Specifying Fixed-Point Data in Stateflow” on page 8-7), you can use it just as you would any data in Stateflow. However, because of the limitations of fixed-point numbers, it is a good idea to follow these guidelines when using them:

- 1 Develop and test your application using double- or single-precision floating-point numbers.

Using double- or single-precision floating-point numbers does not limit the range or precision of your computations. You need this while you are building your application.

- 2 Once your application works well, start substituting fixed-point data for double-precision data during the simulation phase, as follows:

- a Set the integer word size for the simulation environment to the integer size of the intended target environment.

Stateflow uses this integer size in generated code to select result types for your fixed-point operations. See “Setting the Integer Word Size for a Target” on page 8-22.

- b Add the suffix 'C' to literal numeric constants.

This suffix casts a literal numeric constant in the type of its context. For example, if x is fixed-point data, the expression $y = x / 3.2C$ first converts the numerical constant 3.2 to the fixed-point type of x and then performs the division with a fixed-point result. See “Fixed-Point Context-Sensitive Constants” on page 8-8 for more information.

Note If you do not use context-sensitive constants with fixed-point types, noninteger numeric constants (for example, constants that have a decimal point) can force fixed-point operations to produce floating-point results.

- 3 When you simulate, use overflow detection.

See “Overflow Detection for Fixed-Point Types” on page 8-11 for instructions on how to set overflow detection in simulation.

- 4 If you encounter overflow errors in fixed-point data, you can do one of the following to add range to your data.

- Increase the number of bits in the overflowing fixed-point data.

For example, change the base type for Q from `int16` to `int32`.

- Increase the range of your fixed-point data by increasing the power of 2 value, E .

For example, you might want to increase E from -2 to -1. This decreases the available precision in your fixed-point data.

- 5 If you encounter problems with model behavior stemming from inadequate precision in your fixed-point data, you can do one of the following to add precision to your data:

- Increase the precision of your fixed-point data by decreasing the value of the power of 2 binary point E .
For example, you might want to decrease E from -2 to -3. This decreases the available range in your fixed-point data.
 - If you decrease the value of E , you might also want to increase the number of bits in the base data type for Q to prevent overflow.
For example, change the base type for Q from `int16` to `int32`.
- 6 If you cannot avoid overflow for lack of precision, consider using the `:=` assignment operator in place of the `=` operator for assigning the results of multiplication and division operations.

You can use the `:=` operator to increase the range and precision of the result of fixed-point multiplication and division operations at the possible expense of computational efficiency. See “Assignment Operator `:=`” on page 8-31.

Overflow Detection for Fixed-Point Types

Overflow occurs when the magnitude of a result assigned to a data exceeds the numeric capacity of that data. You enable Stateflow to detect overflow of integer and fixed-point operations during simulation with the following steps:

- 1 In the Stateflow diagram editor, from the Tools menu, select Open Simulation Target.

The **Stateflow Target Builder** dialog opens with `sfun` entered in the **Target Name** field.


- 2 Select **Coder Options**.

The **Stateflow sfun Coder Options** dialog opens.

- 3 Select both the **Enable debugging/animation** and **Enable overflow detection (for debugging)** options.

For descriptions of these options, see “Configuring a Simulation Target for Stateflow” on page 12-11.

- 4 Select **OK** to close the **Stateflow sfun Coder Options** dialog.

- 5 In the **Stateflow Target Builder** dialog, select **Build** to build the simulation target.
- 6 In the Stateflow diagram editor toolbar, select Debug  to open the **Debugging** window.
- 7 In the **Debugging** window, select **Data Range**.

See “Setting Error Checking in the Debugging Window” on page 13-5 for a description of this option.
- 8 In the Debugging window, select **Start** to begin simulating the model.

Simulation breaks execution when an overflow occurs.

Sharing Fixed-Point Data with Simulink

If you plan on sharing fixed-point data with Simulink, use one of the following methods:

- Define the data that you input from Simulink or output to Simulink identically in both Stateflow and Simulink.

This means that the values that you enter for the **Stored Integer** and **Scaling** fields in the shared data’s properties dialog in Stateflow (see “Specifying Fixed-Point Data in Stateflow” on page 8-7) must match similar fields that you enter for fixed-point data in Simulink. See “Fixed-Point “Bang-Bang Control” Example” on page 8-14 for an example of this method of sharing input from Simulink data using a Gateway In block in Simulink.

For some Simulink blocks, you can specify the type of input or output data directly. For example, you can set fixed-point output data directly in the block parameters dialog of the Simulink Constant block when you select **Specify via dialog** for the **Output data type mode** field (under **Show additional parameters**).

- Define the data as **Input from Simulink** or **Output to Simulink** in the data's properties dialog in Stateflow and instruct the sending or receiving block in Simulink to inherit its type from Stateflow.

Many blocks allow you to set their data types and scaling through inheritance from the driving block, or through backpropagation from the next block. This can be a good way to set the data type of a Simulink block to match the data type of the Stateflow port it connects to.

For example, you can set the Simulink Constant block to inherit its type from the Stateflow **Input to Simulink** port that it supplies by selecting **Inherit via back propagation** for the **Output data type mode** field in its block parameters dialog (under **Show additional parameters**).

For a general discussion of data compatibility between Simulink blocks, see “Unified Simulink and Fixed-Point Blockset Blocks” and “Compatibility with Simulink Blocks” in the Fixed-Point Blockset documentation.

Fixed-Point “Bang-Bang Control” Example

In this section you open and explore a Stateflow fixed-point demo model that shows you how Stateflow fixed-point data is used in the following topics:

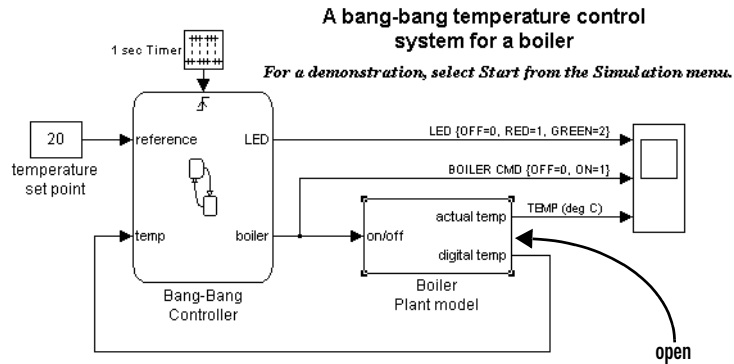
- “Opening the Fixed-Point “Bang-Bang Control” Example” on page 8-14
- “Exploring the Fixed-Point “Bang-Bang Control” Example” on page 8-15

Opening the Fixed-Point “Bang-Bang Control” Example

Stateflow includes demo models with applications of fixed-point data. For this example, load the `sf_boiler` demo model (“Bang-Bang control using Temporal Logic”) into Simulink with the following steps:

- 1** In the MATLAB window, in the **Help** menu, select **Demos**.
- 2** In the left pane of the resulting **Help** dialog, click the + sign in front of **Simulink** to expand that node.
- 3** Continue by expanding the **Stateflow** node under the **Simulink** node.
- 4** Continue by expanding the **Examples** node under the **Stateflow** node.
- 5** Double-click the demos node under the **Stateflow** node.
- 6** Double-click the node marked **Bang-Bang control using Temporal Logic**.

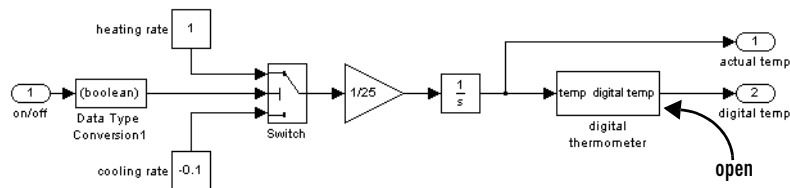
A Simulink window opens as shown.



Exploring the Fixed-Point "Bang-Bang Control" Example

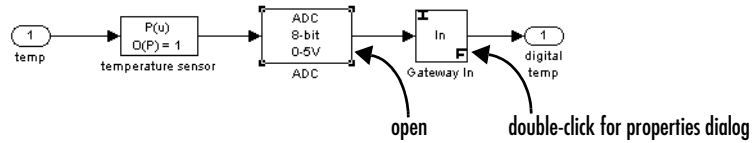
The Stateflow block performs almost all the logic of the bang-bang boiler model with the exception of the Boiler Plant model subsystem block.

- 1 Double-click the Boiler Plant model subsystem block.



The Boiler Plant model block simulates the temperature reaction of the boiler to periods of heating or cooling dictated by the Stateflow block. Depending on the Boolean value coming from the Controller, a temperature increment (+1 for heating, -0.1 for cooling) is added to the previous boiler temperature. The resulting boiler temperature is sent to the digital thermometer subsystem block.

- 2 Double-click the digital thermometer subsystem block.



The digital thermometer subsystem produces an 8 bit fixed-point representation of the input temperature with the blocks described in the sections that follow.

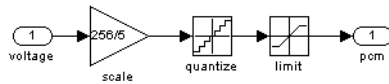
temperature sensor Block

The temperature sensor block converts input boiler temperature (T) to an intermediate analog voltage output T_{volts} with a first-order polynomial that results in the following output:

$$T_{volts} = 0.05 * T + 0.75$$

ADC Block

Double-click the ADC block to reveal the following contents:



The ADC subsystem digitizes the analog voltage from the temperature sensor block by multiplying the analog voltage by $256/5$, rounding it to its integer floor, and limiting it to a maximum of 255 (the largest unsigned 8 bit integer value). Using the value for the output T_{volts} from the temperature sensor block, the new digital coded temperature output by the ADC block, $T_{digital}$, is given by the following equation:

$$T_{digital} = (256/5) * T_{volts} = (256 * 0.05 / 5) * T + (256 / 5) * 0.75$$

Gateway In Block

An examination of the Block Parameters dialog for the Gateway In block shows that it informs the rest of the model that $T_{digital}$ is now a fixed-point number with a slope value of $5/256/0.05$ and an intercept value of $-0.75/0.05$. The Stateflow block Bang Bang Controller receives this output and interprets it as

a fixed-point number through the Stateflow data temp, which is scoped as **Input from Simulink** and set as an unsigned 8 bit fixed-point data with the same values for S and B set in the Gateway In block.

The values for S and B are determined from the general expression for a fixed-point number, which is as follows:

$$V = S*Q + B$$

Therefore,

$$Q = (V - B)/S = (1/S)*V + (-1/S)*B$$

Since $T_{digital}$ is now a fixed-point number, it is now the quantized integer Q of a fixed-point type. This means that $T_{digital} = Q$ of its fixed-point type and results in the following identity:

$$(1/S)*V + (-1/S)*B = (256*0.05/5)*T + (256/5)*0.75$$

Since T is the real-world value for the environment temperature, the above equation implies the following identifications:

$$V = T$$

and

$$1/S = (256*0.05)/5$$

$$S = 5/(256*0.05) = 0.390625$$

and

$$(-1/S)*B = (256/5)*0.75$$

$$B = -(256/5)*0.75*5/(256*0.05) = -0.75/0.05 = 15$$

By setting $T_{digital}$ to be a fixed-point data both as the output of the Gateway In block in Simulink and the input of the Stateflow Bang Bang Controller block, Stateflow interprets and processes this data automatically in an 8 bit environment with no need for any explicit conversions.

Operations with Fixed-Point Data

This section lists the supported operations for fixed-point data in Stateflow and describes the data conversions required in order to perform these operations in the following topics:

- “Supported Operations with Fixed-Point Operands” on page 8-18 — Provides a reference list of all supported operations with fixed-point numbers.
- “Promotion Rules for Fixed-Point Operations” on page 8-21 — Describes the data types automatically selected by Stateflow to receive the results of operations with fixed-point numbers.
- “Assignment (=, :=) Operations” on page 8-31 — Describes the data types that result from assignment operations with fixed-point numbers.
- “Fixed-Point Conversion Operations” on page 8-36 — Describes conversions of fixed-point numbers from one type to another that take place during code generation (offline) and execution (online).

Supported Operations with Fixed-Point Operands

Stateflow supports the operations listed in the topics that follow.

Binary Operations

Stateflow supports the following binary operations with the listed precedence:

Example	Precedence	Description
$a * b$	10	Multiplication
a / b	10	Division
$a + b$	9	Addition
$a - b$	9	Subtraction
$a > b$	7	Comparison, greater than
$a < b$	7	Comparison, less than
$a \geq b$	7	Comparison, greater than or equal to

Example	Precedence	Description
a <= b	7	Comparison, less than or equal to
a == b	6	Comparison, equality
a ~= b	6	Comparison, inequality
a != b	6	Comparison, inequality
a <> b	6	Comparison, inequality
a & b	5	<p>One of the following:</p> <ul style="list-style-type: none"> • Bitwise AND Enabled when Enable C-bit operations is selected in the chart properties dialog. See “Specifying Chart Properties” on page 9-4. Operands are cast to integers before the operation is performed. • Logical AND Enabled when Enable C-bit operations is cleared in chart properties dialog.
a b	3	<p>One of the following:</p> <ul style="list-style-type: none"> • Bitwise OR Enabled when Enable C-bit operations is selected in chart properties dialog. See “Specifying Chart Properties” on page 9-4. Operands are cast to integers before the operation is performed. • Logical OR Enabled when Enable C-bit operations is cleared in chart properties dialog.
a && b	2	Logical AND
a b	1	Logical OR

Unary Operations and Actions

Stateflow supports the following unary operations and actions:

Example	Description
<code>~a</code>	Unary minus
<code>!a</code>	Logical not
<code>a++</code>	Increment
<code>a--</code>	Decrement

Assignment Operations

Stateflow supports the following assignment operations:

Example	Description
<code>a = expression</code>	Simple assignment
<code>a := expression</code>	See “Assignment Operator :=” on page 8-31.
<code>a += expression</code>	Equivalent to <code>a = a + expression</code>
<code>a -= expression</code>	Equivalent to <code>a = a - expression</code>
<code>a *= expression</code>	Equivalent to <code>a = a * expression</code>
<code>a /= expression</code>	Equivalent to <code>a = a / expression</code>
<code>a = expression</code>	Equivalent to <code>a = a expression</code> (bit operation). See operation <code>a b</code> in “Binary Operations” on page 8-18.
<code>a &= expression</code>	Equivalent to <code>a = a & expression</code> (bit operation). See operation <code>a & b</code> in “Binary Operations” on page 8-18.

Promotion Rules for Fixed-Point Operations

Operations with at least one fixed-point operand require rules for selecting the type of the intermediate result for that operation. For example, in the action statement $c = a + b$, where a or b is a fixed-point number, an intermediate result type for $a + b$ must first be chosen before the result is calculated and assigned to c .

The rules for selecting the numeric types used to hold the results of operations with a fixed-point number are referred to as the *fixed-point promotion rules*. The primary goal of these rules is to maintain good computational efficiency with reasonable usability.

Note You can use the `:=` assignment operator to override the fixed-point promotion rules in the interest of obtaining greater accuracy. However, in this case, greater accuracy might require more computational steps. See “Assignment Operator `:=`” on page 8-31.

The following topics describe the process of selecting an intermediate result type for all binary operations with at least one fixed-point operand:

Default Selection of the Number of Bits of the Result Type

A fixed-point number with $S = 1$ and $B = 0$ is treated as an integer. In operations with integers, the C language promotes any integer input with fewer bits than the type `int` to the type `int` and then performs the operation.

The type `int` is the *integer word size* for C on a given platform. Result word size is increased to the integer word size because processors can perform operations at this size efficiently.

Stateflow maintains consistency with the C language by using the following default rule to assign the number of bits for the result type of an operation with fixed-point numbers:

When both operands are fixed-point numbers, the number of bits in the result type is the maximum number of bits in the input types or the number of bits in the integer word size for the target machine, whichever is larger. The preceding rule has the following mathematical form:

$$N_c = \max(N_{int}, N_a, N_b)$$

where

- N_c is the bit size for the result of the operation $c = a \langle \text{op} \rangle b$.
- N_a, N_b are the bit sizes of the operands a and b , respectively.
- N_{int} is the integer word size for the target.
-

Note The preceding rule is a default rule for selecting the bit size of the result for operations with fixed-point numbers. This rule is overruled for specific operations as described in the sections that follow.

Setting the Integer Word Size for a Target. The preceding default rule for selecting the bit size of the result for operations with fixed-point numbers relies on the definition of the integer word size for your target. You can set the integer word size for the targets that you build in Simulink with the following procedure:

- 1** Select **Simulink parameters** from the **Simulink** menu in your Simulink model window.
- 2** In the resulting **Simulation Parameters** dialog, select the **Advanced** tab.
- 3** At the bottom of the **Advanced** panel, select **Microprocessor** for the **Production hardware characteristics** field.
- 4** In the window pane at the bottom of the panel, select **Number of bits for C 'int'** to highlight it.
- 5** In the **Value** field to the right, select the target integer size in bits.
- 6** Select **OK** to accept the changes.

When you build any target after making this change, Stateflow uses this integer size in generated code to select result types for your fixed-point operations.

Note It is recommended that you set all the available sizes in the **Value** field because they affect code generation, although they do not affect the implementation of the fixed-point promotion rules in generated code.

Unary Promotions

Only the unary minus (-) operation requires a promotion of its result type. The word size of the result is given by the default procedure for selecting the bit size of the result type for an operation involving fixed-point data. See “Default Selection of the Number of Bits of the Result Type” on page 8-21. The bias, *B*, of the result type is the negative of the bias of the operand.

Binary Operation Promotion for Integer Operand with Fixed-Point Operand

Integers as operands in binary operations with fixed-point numbers are treated as fixed-point numbers of the same word size with slope, *S*, equal to 1, and a bias, *B*, equal to 0. The operation now becomes a binary operation between two fixed-point operands. See “Binary Operation Promotion for Two Fixed-Point Operands” on page 8-24.

Binary Operation Promotion for Double Operand with Fixed-Point Operand

When one operand is of type `double` in a binary operation with a fixed-point type, the result type is `double`. In this case, the fixed-point operand is cast to type `double`, and the operation is performed.

Binary Operation Promotion for Single Operand with Fixed-Point Operand

When one operand is of type `single` in a binary operation with a fixed-point type, the result type is `single`. In this case, the fixed-point operand is cast to type `single`, and the operation is performed.

Binary Operation Promotion for Two Fixed-Point Operands

Operations with both operands of fixed-point type produce an intermediate result of fixed-point type. The resulting fixed-point type is chosen through the application of a set of operator-specific rules. The procedure for producing an intermediate result type from an operation with operands of different fixed-point types is summarized in the following topics:

- “Setting the Integer Word Size for a Target” on page 8-22
- “Addition (+) and Subtraction (-)” on page 8-24
- “Multiplication (*) and Division (/)” on page 8-27
- “Relational Operations (>, <, >=, <=, ==, !=, <>)” on page 8-29
- “Logical Operations (&, |, &&, ||)” on page 8-30

Addition (+) and Subtraction (-). The output type for addition and subtraction is chosen so that the maximum positive range of either input can be represented in the output while preserving maximum precision. The base word type of the output follows the rule in “Default Selection of the Number of Bits of the Result Type” on page 8-21. To simplify calculations and yield efficient code, the biases of the two inputs are added for an addition operation and subtracted for a subtraction operation.

Note Mixing signed and unsigned operands might yield unexpected results and is not recommended.

Selecting the Bias of the Result Type

The bias of the result of an addition or subtraction of two fixed-point operands is evaluated by adding or subtracting the biases of the individual operands as follows:

$$B_c = B_a + B_b, \text{ for the addition operation } c = a + b$$

$$B_c = B_a - B_b, \text{ for the subtraction operation } c = a - b$$

where

- B_a and B_b are the biases of the operands a and b , respectively.
- B_c is the bias for the result of the operation, c .

Selecting the Slope and Sign Bit of the Result Type

The process of selecting the slope for the result type of an addition or subtraction of fixed-point numbers attempts to preserve as much range as possible in the result to prevent overflow. It does so at the expense of precision. The procedure for selecting the slope of the result of an addition and subtraction is as follows:

- 1 The result type attempts to match the precision of the most precise operand.
- 2 If the resulting range is greater than the maximum range of the operands, the result is signed.
- 3 If the resulting range is equal to the maximum range, and one operand is nonsigned, the result is nonsigned.
- 4 If the resulting range is less than the maximum range, the range of the result is increased at the expense of precision until the range is equal to the maximum range without a sign bit.

The precision for any fixed-point number x depends on the radix point location E_x , such that the number of bits to the right of the radix point is equal to $-E_x$. The range for any fixed-point number x , depends on the number of the most significant nonsign bit to the left of the radix point, \overline{MSB}_x , which is defined as follows:

$$\overline{MSB}_x = N_x + E_x - \sigma_x$$

where

- N_x is the number of bits in x .
- σ_x is the sign for x (1 if signed, 0 if not).

To maximize the result, c , for an addition or subtraction of two fixed-point operands, E_c , σ_c , and N_c are chosen to satisfy the following relationships:

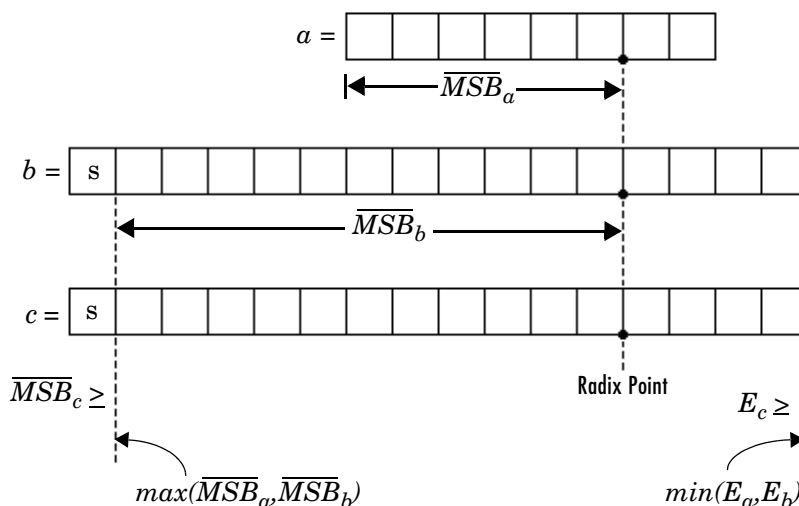
$$\overline{MSB}_c \geq \max(\overline{MSB}_a, \overline{MSB}_b)$$

$$E_c \geq \min(E_a, E_b)$$

The preceding relations are used to set the type of the result, c , in the following example, which adds (or subtracts) two fixed-point numbers, a and b , under the conditions

- a is an 8 bit unsigned integer with a slope of 2^{-2} .
- b is a 16 bit signed integer with a slope of 2^{-4} .
- The number of bits in the natural word size on the target machine is 16 bits.

Selection of the result type for the addition of a and b is summarized in the following figure:

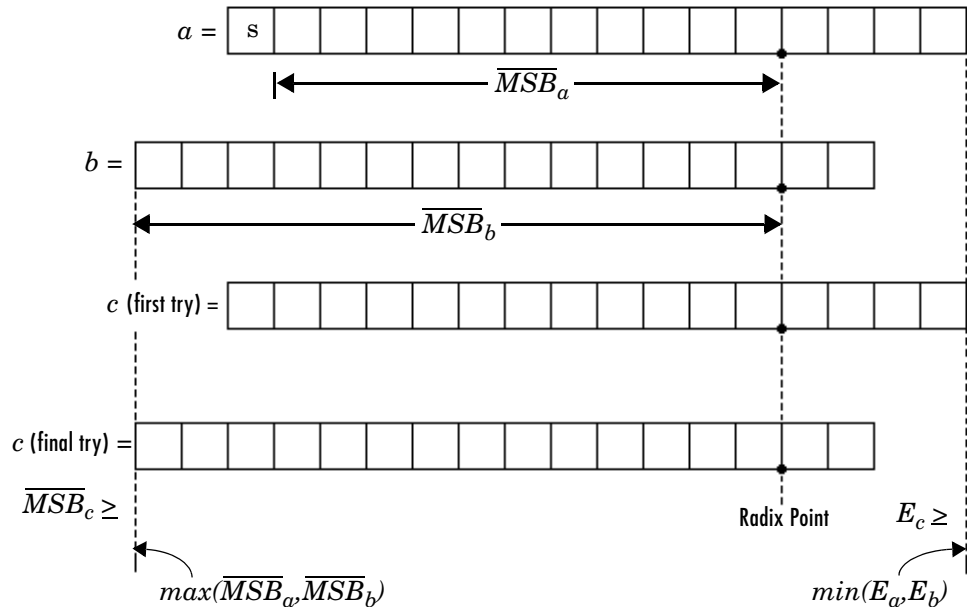


Since both operands are less than or equal to the natural word bit size for the processor, the result bit size of the operation is 16 bits. The result, c , first takes on the precision of the most precise operand, b , which satisfies the requirement $E_c \geq \min(E_a, E_b)$. In doing so, the requirement $\overline{MSB}_c \geq \max(\overline{MSB}_a, \overline{MSB}_b)$ is satisfied at the location of the most significant bit for c , which is unaffected by the additional sign bit. This means that the result type for c is signed with a radix point precision of 2^{-4} .

The following example adds (or subtracts) two fixed-point numbers, a and b under the following conditions:

- a is a 16 bit signed integer with a slope of 2^{-4}
- b is a 16 bit unsigned integer with a slope of 2^{-2}
- The number of bits in the natural word size on the target machine is 16 bits.

Selection of the result type for the addition of a and b is summarized in the following figure:



Since both operands are less than or equal to the natural word size for the processor, the result bit size of the operation is 16 bits. The result, c , first takes on the precision of the most precise operand, b , which satisfies the requirement $E_c \geq \min(E_a, E_b)$. However, the requirement $\overline{MSB}_c \geq \max(\overline{MSB}_a, \overline{MSB}_b)$ is not satisfied at the location of the most significant bit for c . Therefore, c is moved two bits to the left until its most significant nonsigned bit now satisfies $\overline{MSB}_c \geq \max(\overline{MSB}_a, \overline{MSB}_b)$, which reduces its precision to 2^{-2} .

Multiplication (*) and Division (/). The output type for multiplication and division is chosen to yield the most efficient code implementation. Nonzero biases are not supported for multiplication and division by Stateflow (see note).

The slope for the result type of the product of the multiplication of two fixed-point numbers is the product of the slopes of the operands. Similarly, the slope of the result type of the quotient of the division of two fixed-point numbers is the quotient of the slopes. The base word type is chosen to conform to the rule in “Default Selection of the Number of Bits of the Result Type” on page 8-21.

Note Because nonzero biases are computationally very expensive, they are not supported for multiplication and division by Stateflow.

Selecting the Bias of the Result Type

The requirement that fixed-point operands in multiplication or division operations have zero bias means that $B_a = B_b = B_c = 0$, which results in the following simplifications to the solutions for Q_c in “Fixed-Point Operations” on page 8-3:

<op>	Q_c
.	$(S_a S_b / S_c) Q_a Q_b$
/	$(S_a Q_a) / (S_c (S_b Q_b))$

If you attempt to use a fixed-point number of nonzero bias in a multiplication or division operation with another fixed-point number, an error results. However, as a workaround, you can do one of the following:

- Specify only fixed-point numbers of zero bias for an operand in a multiplication or division operation.
- Assign a fixed-point number of nonzero bias to a fixed-point number of zero bias and perform the multiplication or division with the latter as an operand.

Stateflow chooses $S_c = S_a S_b$ for the intermediate results of multiplication and $S_c = S_a / S_b$ for the intermediate results of the division of fixed-point operands. This results in the following simplifications to the preceding solutions for Q_c :

<op>	Q_c
.	$Q_a Q_b$
/	Q_a / Q_b

Selecting the Sign Bit of the Result

Once the word size in bits of the result type of a binary operation between two fixed-point numbers, a and b , is chosen (see “Default Selection of the Number of Bits of the Result Type” on page 8-21), that value is used to select the sign for the result type as follows:

The result type of a binary operation between two fixed-point numbers is assigned a sign bit if the result type word size in bits (not including the sign bit) exceeds the word size in bits of both operands (not including the sign bit). Otherwise, the result has no sign bit.

The preceding rule for the result type, c , of the operation $a \langle \text{op} \rangle b$, has the following mathematical form:

$$\sigma_c = 1 \text{ if } N_c - 1 \geq \max(N_a - \sigma_a, N_b - \sigma_b), \text{ otherwise, } \sigma_c = 0$$

where

- σ_c indicates the presence of a sign bit for the result of the operation, c . Its value is 1 if signed, and 0 if not.
- σ_a, σ_b indicate the presence of a sign bit for the operands, a and b , respectively. Their values are 1 if signed, and 0 if not.
- N_c is the bit size for the result of the operation, c .
- N_a and N_b are the bit sizes of the operands a and b , respectively.

Relational Operations ($>$, $<$, $>=$, $<=$, $==$, $!=$, $<>$). Stateflow supports the following relational (comparison) operations on all fixed-point types: $>$, $<$, $>=$, $<=$, $==$, $!=$, $<>$. See “Supported Operations with Fixed-Point Operands” on page 8-18 for an example and description of these operations. Stateflow requires that both operands in a comparison have equal biases (see note).

Comparing fixed-point values of different types can yield unexpected results because Stateflow must convert each operand to a common type for comparison. Because of rounding or overflow errors during the conversion, values that do not appear equal might be equal and values that appear to be equal might not be equal.

Note To preserve precision and minimize unexpected results, Stateflow requires both operands in a comparison operation to have equal biases.

For example, compare the following two unsigned 8 bit fixed-point numbers, a and b, in an 8 bit target environment:

Fixed-Point Number a	Fixed-Point Number b
$S_a = 2^{-4}$	$S_b = 2^{-2}$
$B_a = 0$	$B_b = 0$
$V_a = 43.8125$	$V_b = 43.75$
$Q_a = 701$	$Q_b = 175$

By rule, the result type for comparison is 8 bit. Converting b, the least precise operand, to the type of a, the most precise operand, could result in overflow. Consequently, a is converted to the type of b. Because the bias values for both operands are 0, the conversion is made as follows:

$$S_b(\text{new}Q_a) = S_a Q_a$$

$$\text{new}Q_a = (S_a/S_b)Q_a = (2^{-4}/2^{-2})701 = 701/4 = 175$$

Although they represent different values, a and b are considered equal as fixed-point numbers.

Logical Operations (&, |, &&, ||). If a is a fixed-point number used in a logical operation, it is interpreted with the equivalent substitution $a \text{ != } 0.0C$ where 0.0C is an expression for zero in the fixed-point type of a (see “Fixed-Point Context-Sensitive Constants” on page 8-8). For example, if a is a fixed-point number in the logical operation $a \ \&\& \ b$, this operation is equivalent to the following:

$$(a \text{ != } 0.0C) \ \&\& \ b$$

The preceding operation is not a check to see whether the quantized integer for a , Q_a , is not 0. If the real-world value for a fixed-point number a is 0, this implies that $V_a = S_a Q_a + B_a = 0.0$. Therefore, the expression $a \neq 0$, for fixed-point number a , is actually equivalent to the following expression:

$$Q_a \neq -B_a / S_a$$

For example, if a fixed-point number, a , has a slope of 2^{-2} , and a bias of 5, the test $a \neq 0$ is equivalent to the test if $Q_a \neq -20$.

Assignment (=, :=) Operations

Stateflow supports the assignment operations $LHS = RHS$ and $LHS := RHS$ between a left-hand side (LHS) and a right-hand side (RHS). These are described in the following topics:

- “Assignment Operator =” on page 8-31.
- “Assignment Operator :=” on page 8-31
- “:= Multiplication Example” on page 8-32
- “:= Division Example” on page 8-34
- “:= Assignment and Context-Sensitive Constants” on page 8-35

Assignment Operator =

An assignment statement of the type $LHS = RHS$ is equivalent to casting the right-hand side to the type of the left-hand side. Stateflow supports any assignment between fixed-point types and therefore, implicitly, any cast.

A cast converts the stored integer Q from its original fixed-point type while preserving its value as accurately as possible using the online conversions (see “Fixed-Point Conversion Operations” on page 8-36). Assignments are most efficient when both types have the same bias, and slopes that are equal or both powers of 2.

Assignment Operator :=

Ordinarily, Stateflow uses the fixed-point promotion rules to choose the result type for an operation. Using the $:=$ assignment operator overrides this behavior by using the type of the LHS as the result type of the RHS operation.

This type of assignment is particularly useful in retaining useful range and precision in the result of a multiplication or division that ordinary assignment might not retain. It is less useful with addition or subtraction but can avoid overflow or the loss of memory to store a result even in these cases.

Use of the `:=` assignment operator is governed by the following rules:

- The RHS can contain at most one binary operator.
- If the RHS contains anything other than a multiplication (`*`), division (`/`), addition (`+`), or subtraction (`-`) operation, or a constant, then the `:=` assignment behaves exactly like regular assignment (`=`).
- Constants on the RHS of an LHS `:=` RHS assignment are converted to the type of the left-hand side using offline conversion (see “Fixed-Point Conversion Operations” on page 8-36). Ordinary assignment always casts the RHS using online conversions.

For examples contrasting the LHS `:=` RHS and the LHS `=` RHS assignment operations, see the following:

- “`:=` Multiplication Example” on page 8-32
- “`:=` Division Example” on page 8-34

Caution Using the `:=` assignment operator to produce a more accurate result might generate code that is less efficient than the code generated using the normal fixed-point promotion rules.

`:=` Multiplication Example

The following example contrasts the `:=` and `=` assignment operators for multiplication. Here, the `:=` operator is used to avoid overflow in the results of the multiplication `c = a * b` in which `a` and `b` are of two fixed-point operands.

The operands and result for this operation are 16 bit unsigned integers with the following assignments:

Fixed-Point Number a	Fixed-Point Number b	Fixed-Point Number c
$S_a = 2^{-4}$	$S_b = 2^{-4}$	$S_c = 2^{-5}$
$B_a = 0$	$B_b = 0$	$B_c = 0$
$V_a = 20.1875$	$V_b = 15.3125$	$V_c = ?$
$Q_a = 323$	$Q_b = 245$	$Q_c = ?$

where S is the slope, B is the bias, V is the real-world value, and Q is the quantized integer.

c = a*b. In this case, first calculate an intermediate result for $a*b$ in the fixed-point type given by the rules in the section “Fixed-Point Operations” on page 8-3, and then cast that result into the type for c .

The intermediate value is calculated as follows:

$$\begin{aligned} Q_{iv} &= Q_a Q_b \\ &= 323 \cdot 245 = 79135 \end{aligned}$$

Because the maximum value of a 16 bit unsigned integer is $2^{16} - 1 = 65535$, the preceding result overflows its word size. An operation that overflows its type produces an undefined result.

You can capture overflow errors like the preceding example during simulation with the Debugger window. See “Overflow Detection for Fixed-Point Types” on page 8-11.

c := a*b. In this case, calculate $a*b$ directly in the type of c . Use the solution for Q_c given in “Fixed-Point Operations” on page 8-3 with the requirement of zero bias, which is as follows:

$$\begin{aligned} Q_c &= ((S_a S_b / S_c) Q_a Q_b) \\ &= (2^{-4} \cdot 2^{-4} / 2^{-5}) (323 \cdot 245) \\ &= 79135 / 8 = 9892 \quad (\text{rounded to floor}) \end{aligned}$$

No overflow occurs in this case, and the approximate real-world value is as follows:

$$\tilde{V}_c = S_c Q_c = 2^{-5} \cdot 9892 = 9892/32 = 309.125$$

This value is very close to the actual real-world result of 309.121.

:= Division Example

The following example contrasts the := and = assignment operators for division. The := operator is used to obtain more precise results for the division of two fixed-point operands, b and c, in the statement c := a/b.

This example uses the following fixed-point numbers, where S is the slope, B is the bias, V is the real-world value, and Q is the quantized integer:

Fixed-Point Number a	Fixed-Point Number b	Fixed-Point Number c
$S_a = 2^{-4}$	$S_b = 2^{-3}$	$S_c = 2^{-6}$
$B_a = 0$	$B_b = 0$	$B_c = 0$
$V_a = 2$	$V_b = 3$	$V_c = ?$
$Q_a = 32$	$Q_b = 24$	$Q_c = ?$

c = a/b. In this case, first calculate an intermediate result for a/b in the fixed-point type given by the rules in the section “Fixed-Point Operations” on page 8-3, and then cast that result into the type for c.

The intermediate value is calculated as follows:

$$\begin{aligned} Q_{iv} &= Q_a / Q_b \\ &= 32 / 24 = 1 \end{aligned}$$

The intermediate value is then cast to the result type for c as follows:

$$\begin{aligned} S_c Q_c &= S_{iv} Q_{iv} \\ Q_c &= (S_{iv} / S_c) Q_{iv} \end{aligned}$$

The slope of the intermediate value for a division operation is calculated as

$$S_{iv} = S_a/S_b = 2^{-4}/2^{-3} = 2^{-1}$$

Substitution of this value into the preceding result yields the final result.

$$Q_c = 2^{-1}/2^{-6} = 2^5 = 32$$

In this case, the approximate real-world value is $\tilde{V}_c = 32/64 = 0.5$, which is not a very good approximation of the actual result, $2/3 = 0.667$.

c := a/b. In this case, calculate a/b directly in the type of c. Use the solution for Q_c given in “Fixed-Point Operations” on page 8-3 with the simplification of zero bias, which is as follows:

$$\begin{aligned} Q_c &= (S_a Q_a)/(S_c(S_b Q_b)) \\ &= (S_a/(S_b \cdot S_c)) \cdot Q_a/Q_b \\ &= (2^{-4}/(2^{-3} \cdot 2^{-6})) \cdot 32/24 \\ &= 2^5 \cdot 32/24 = 42 \end{aligned}$$

In this case, the approximate real-world value $\tilde{V}_c = 42/64 = 0.6563$, a much better approximation to the precise result, $2/3 = 0.667$.

:= Assignment and Context-Sensitive Constants

In a := assignment operation, the type of the left-hand side (LHS) determines part of the context used for inferring the type of a right-hand side (RHS) context-sensitive constant.

The following rules apply to RHS context-sensitive constants in assignments with the := operator:

- If the LHS is a floating-point data (type `double` or `single`), the RHS context-sensitive constant becomes a floating-point constant.
- For addition and subtraction, the type of the LHS determines the type of the context-sensitive constant on the RHS.
- For multiplication and division, the type of the context-sensitive constant is chosen independent of the LHS.

Fixed-Point Conversion Operations

Stateflow converts real numbers into fixed-point data during data initialization and as part of casting operations in the application. These conversions compute a quantized integer, Q , from a real number input. Stateflow uses offline conversions to initialize data and online conversions for casting operations in the running application. The topics that follow describe each conversion type and give examples of the results.

Offline Conversions for Initialized Data

Offline conversions are performed during code generation, and are designed to maximize accuracy. They round the resulting quantized integer to its nearest integer value. If the conversion overflows, the result saturates the value for Q .

Offline conversions are performed for the following operations:

- Initialization of data (both variables and constants) in the data dictionary
- Initialization of constants or variables from the MATLAB workspace

Online Conversions for Casting Operations

Online conversions are performed for casting operations that take place during execution of the application. Designed to maximize computational efficiency, they are faster and more efficient than offline conversions, but less precise. Instead of rounding Q to its nearest integer, online conversions round to the floor (with the possible exception of division, which might round to 0, depending on the C compiler you have). If the conversion overflows the type converted to, the result is undefined.

Offline and Online Conversion Examples

The following examples show the difference in the results of of offline and online conversions of real numbers to a fixed-point type defined by a 16 bit word size, a slope (S) equal to 2^{-4} , and a bias (B) equal to 0:

		Offline Conversion		Online Conversion	
V	V/S	Q	\tilde{V}	Q	\tilde{V}
3.45	55.2	55	3.4375	55	3.4375

		Offline Conversion		Online Conversion	
V	V/S	Q	\tilde{V}	Q	\tilde{V}
1.0375	16.6	17	1.0625	16	1
2.06	32.96	33	2.0625	32	2

In the preceding example,

- V is the real-world value represented as a fixed-point value.
- V/S is the floating-point computation for the quantized integer Q .
- Q is the rounded value of V/S .
- \tilde{V} is the approximate real-world value resulting from Q for each conversion.

Autoscaling of Stateflow Fixed-Point

Simulink autoscales Stateflow fixed-point data with the Simulink autoscaling tool. See “Automatic Scaling” in Fixed-Point Blockset documentation for instructions on autoscaling fixed-point data in Simulink.

You can prevent Stateflow fixed-point data from being autoscaled by selecting the **Lock output scaling against changes by the autoscaling tool** checkbox in the **Data** dialog for a fixed-point data. Selecting this option prevents Simulink from replacing the current fixed-point type with a Simulink chosen type in the autoscaling tool. See “Setting Data Properties in the Data Dialog” on page 6-25 for a description of the properties for data.

Defining Interfaces to Simulink and MATLAB

Each Stateflow chart is a block in a Simulink diagram that sits on top of MATLAB. You can share data with MATLAB and Simulink and also determine how and when Simulink executes your charts through Stateflow interfaces with the following sections:

Overview of Stateflow Interfaces
(p. 9-2)

Each Stateflow block interfaces to its Simulink model. Take an overview look at Stateflow interfaces to Simulink and MATLAB.

Specifying Chart Properties (p. 9-4)

Tells you how to specify properties for your chart. Part of the interface for a chart to its Simulink model is set when you specify the properties for a chart.

Setting the Stateflow Block Update Method (p. 9-11)

Implementing different Stateflow interfaces in Simulink requires you to set the update method for your chart. This section describes each of the settings for the update method of your chart.

Implementing Simulink Update Interfaces (p. 9-12)

This section summarizes all the settings necessary for implementing any possible interface in Simulink to your Stateflow chart block.

Creating Chart Libraries (p. 9-23)

Shows you how to save Stateflow charts that you can place in the Simulink block library for repeated use in a Simulink model.

MATLAB Workspace Interfaces
(p. 9-24)

The MATLAB workspace is an area of memory normally accessible from the MATLAB command line. This section describes ways that Stateflow can access the data and functions of the MATLAB workspace.

Interface to External Sources (p. 9-25)

Describes the ways in which a Stateflow chart can interface data and events outside its Simulink model.

Overview of Stateflow Interfaces

Each Stateflow block interfaces to its Simulink model. Take an overview look at Stateflow interfaces to Simulink and MATLAB with the following topics:

- “Stateflow Interfaces” on page 9-2 — Lists the interfaces that Stateflow has to Simulink blocks, MATLAB data, and external code sources.
- “Typical Tasks to Define Stateflow Interfaces” on page 9-3 — Describes the tasks used to define Stateflow interfaces.
- “Where to Find More Information on Events and Data” on page 9-3 — Gives you references to further information on defining Stateflow interfaces in the Stateflow documentation.

Stateflow Interfaces

Each Stateflow block interfaces to its Simulink model. Each Stateflow block can interface to sources external to the Simulink model (data, events, custom code). Events and data are the Stateflow objects that define the interface from the Stateflow block’s point of view.

Events can be local to the Stateflow block or can be propagated to and from Simulink and sources external to Simulink. Data can be local to the Stateflow block or can be shared with and passed to the Simulink model and to sources external to the Simulink model.

The Stateflow interfaces includes the following:

- Physical connections between Simulink blocks and the Stateflow block
- Event and data information exchanged between the Stateflow block and external sources
- The properties of a Stateflow chart.
- Graphical functions exported from a chart
See “Exporting Functions” on page 5-41 for more details.
- The MATLAB workspace
See “Using MATLAB Functions and Data in Actions” on page 7-27 for more details.
- Definitions in external code sources

Typical Tasks to Define Stateflow Interfaces

Defining the interface for a Stateflow block in a Simulink model involves some or all the tasks described in the following topics:

- Specify the update method for a Stateflow block in a Simulink model.
This task is described in “Setting the Stateflow Block Update Method” on page 9-11.
- Define the input and output data and events that you need.
See the following topics for detailed information:
 - “Defining Input Events” on page 6-11
 - “Defining Output Events” on page 6-13
 - “Sharing Input and Output Data with Simulink” on page 6-35
- Add and define any nonlocal data and events your Stateflow diagram must interact with.
- Define relationships with any external sources.
See the topics “MATLAB Workspace Interfaces” on page 9-24 and “Interface to External Sources” on page 9-25.

The preceding task list could be a typical sequence. You might find that another sequence better complements your model development.

See “Implementing Simulink Update Interfaces” on page 9-12 for examples of implemented interfaces to Simulink.

Where to Find More Information on Events and Data

The following references are relevant to defining the interface for a Stateflow Chart block in Simulink:

- “Defining Input Events” on page 6-11
- “Defining Output Events” on page 6-13
- “Importing Events from Stateflow External Code” on page 6-17
- “Exporting Events to Stateflow External Code” on page 6-16
- “Sharing Input and Output Data with Simulink” on page 6-35
- “Importing Data from External Stateflow Code” on page 6-43
- “Exporting Data to External Stateflow Code” on page 6-42

Specifying Chart Properties

Part of the interface for a Stateflow block to its Simulink model is set when you specify the properties for the chart of a Stateflow block. You can specify properties for individual charts or for all charts in a model as described in the following topics:

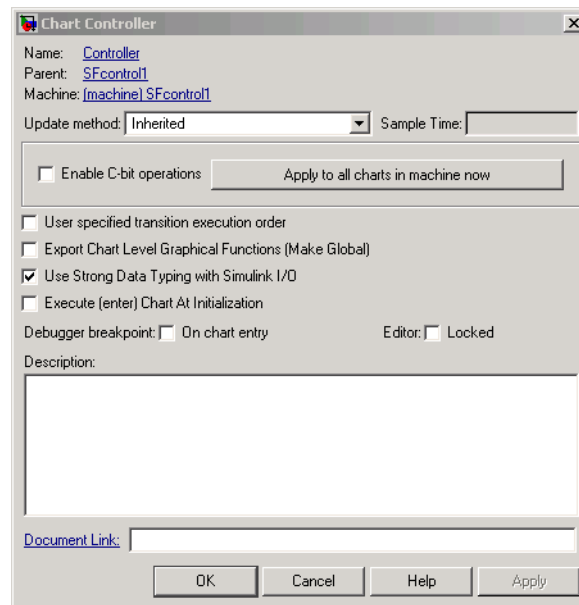
- “Setting Properties for Individual Charts” on page 9-4
- “Setting Properties for All Charts in the Model” on page 9-8

Setting Properties for Individual Charts

You set most of the properties for a chart as an individual chart. To specify properties for a chart of a Stateflow block, do the following:

- 1 Double-click on a Stateflow block to open its diagram in the Stateflow diagram editor.
- 2 Right-click an open area of the Stateflow diagram.
- 3 From the resultant context menu, select **Properties**.

The properties dialog for the chart appears, as shown:



4 Enter properties for the chart based on the following descriptions:

Field	Description
Name	Stateflow diagram name; read-only; click this hypertext link to bring the chart to the foreground.
Parent	Parent name; read-only; click this hypertext link to display the parent's property dialog box.
Machine	Simulink subsystem name; read-only; click this hypertext link to bring the Simulink subsystem to the foreground.

Field	Description (Continued)
Update method	Stateflow lets you specify the method by which a simulation updates (wakes up) a chart in Simulink. Choose from Inherited , Discrete , or Continuous . For more information, see “Setting the Stateflow Block Update Method” on page 9-11.
Sample Time	If Update method is Sampled , enter a sample time.
Enable C-bit operations	<p>Select this box to recognize C bitwise operators (~, &, , ^, >>, and so on) in action language statements and encode them as C bitwise operations.</p> <p>If this box is not selected, the following occurs:</p> <ul style="list-style-type: none"> • & and are interpreted as logical operators. • ^ is interpreted as the power operator (for example, 2^3 = 8). • The remaining expressions (>>, <<, and so on) result in parse errors. <p>To specify this interpretation for all charts in the model (machine), select the Apply to all charts in machine now button.</p>
User specified transition execution order	Select this box to switch to explicit ordering of transitions. In this mode, you have complete control of the order in which transitions originating from a source are tested for execution. For more information, see “Transition Testing Order” on page 3-10.
Export Chart Level Graphical Functions	Exports graphical functions defined at the chart’s root level. See “Exporting Functions” on page 5-41 for more information.

Field	Description (Continued)
Use Strong Data Typing with Simulink I/O	<p>If this option is selected, the Chart block for this chart can accept input signals of any data type supported by Simulink, provided that the type of the input signal matches the type of the corresponding chart input data item (see “Sharing Input and Output Data with Simulink” on page 6-35). If the types do not match, a type mismatch error occurs.</p> <p>If this item is cleared, the chart accepts and outputs only signals of type double. In this case, Stateflow converts Simulink input signals to the data types of the corresponding chart input data items. Similarly, Stateflow converts chart output data (see “Sharing Input and Output Data with Simulink” on page 6-35) to type double if this option is not selected.</p> <p>For fixed-point data, see the note following this table.</p>
Execute (enter) Chart at Initialization	<p>Select this option if you want a chart’s state configuration to be initialized at time 0 instead of at the first occurrence of an input event.</p>
Debugger breakpoint: On chart entry	<p>Select to set a debugging breakpoint on entry to this chart.</p>
Editor: Locked	<p>Select to mark the Stateflow diagram as read-only and prohibit any write operations.</p>

Field	Description (Continued)
Description	Textual description/comment.
Document Link	Enter a Web URL address or a general MATLAB command. Examples are <code>www.mathworks.com</code> , <code>mailto:email_address</code> , and <code>edit/spec/data/speed.txt</code> .

Note For fixed-point data, the **Use Strong Data Typing with Simulink I/O** option is always on. Therefore, if an input or output fixed-point data in Stateflow does not match its counterpart data in Simulink, a mismatch error results.

5 Select one of the following buttons:

- **Apply** to save the changes
- **Cancel** to cancel any changes since the last apply
- **OK** to save the changes and close the dialog box
- **Help** to display the Stateflow online help in an HTML browser window

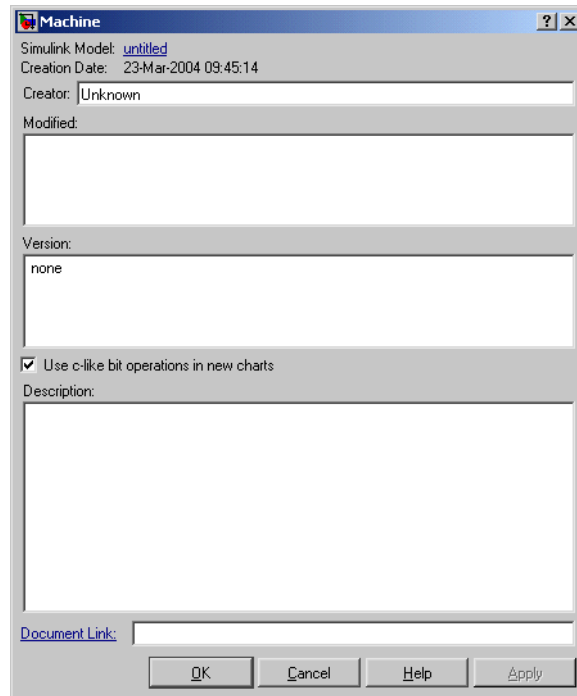
Setting Properties for All Charts in the Model

You can set some properties for all charts in the model by setting properties for the Stateflow machine for a model. The Stateflow machine for a model represents all of the Stateflow blocks in a model.

To set properties for the Stateflow machine, do the following:

- 1 In the **Chart** properties dialog for a particular Stateflow chart, select the **Machine** link at the top of the dialog.

The **Machine properties** dialog box appears.



See “Setting Properties for Individual Charts” on page 9-4 for instructions on accessing the **Chart** properties dialog for a Stateflow chart.

- 2 Enter information in the fields provided as described below.

Field	Description
Simulink Model	Name of the Simulink model that defines this Stateflow machine, which is read-only. You change the model name in Simulink when you save the model under a chosen file name.
Creation Date	Date on which this machine was created.
Creator	Name of the person who created this Stateflow machine.

Field	Description
Modified	Time of the most recent modification of this Stateflow machine.
Version	Version number of this Stateflow machine.
Enable C-like bit operations	<p>If you select this box, all new charts recognize C bitwise operators (~, &, , ^, >>, and so on) in action language statements and encode these operators as C bitwise operations.</p> <p>You can enable or disable this option for individual charts or all charts in the model in an individual chart's property dialog box. See “Setting Properties for Individual Charts” on page 9-4 for a detailed explanation of this property.</p>
Description	Brief description of this Stateflow machine, which is stored with the model that defines it.
Document Link	MATLAB expression that, when evaluated, displays documentation for this Stateflow machine.

3 Click one of the following:

- **Apply** saves the changes.
- **Cancel** closes the dialog without making any changes.
- **OK** saves the changes and closes the dialog box.
- **Help** displays the Stateflow online help in an HTML browser window.

Setting the Stateflow Block Update Method

Stateflow blocks are Simulink subsystems. Simulink events wake up subsystems for execution. To specify a wakeup method for a chart, set the chart's **Update method** property in the **Chart** dialog for the chart (see "Specifying Chart Properties" on page 9-4). Choose from the following wakeup methods:

- **Inherited**

This is the default update method. Specifying this method causes input from the Simulink model to determine when the chart wakes up during a simulation.

If you define input events for the chart, the Stateflow block is explicitly triggered by a signal on its trigger port originating from a connected Simulink block. This trigger input event can be set in the Stateflow Explorer to occur in response to a Simulink signal that is **Rising**, **Falling**, or **Either** (rising and falling), or in response to a **Function Call**. See "Defining Input Events" on page 6-11.

If you do not define input events, the Stateflow block implicitly inherits triggers from the Simulink model. These implicit events are the sample times (discrete or continuous) of the Simulink signals providing inputs to the chart. If you define data inputs (see "Sharing Input and Output Data with Simulink" on page 6-35), the chart awakens at the rate of the fastest data input. If you do not define any data inputs for the chart, the chart wakes up as defined by its parent subsystem's execution behavior.

- **Discrete**

Simulink awakens (samples) the Stateflow block at the rate you specify as the block's **Sample Time** property. An implicit event is generated by Simulink at regular time intervals corresponding to the specified rate. The sample time is in the same units as the Simulink simulation time. Note that other blocks in the Simulink model can have different sample times.

- **Continuous**

Simulink wakes up (samples) the Stateflow block at each step in the simulation, as well as at intermediate time points that can be requested by the Simulink solver. This method is consistent with the continuous method in Simulink.

See "Interface to External Sources" on page 9-25 for more information.

Implementing Simulink Update Interfaces

Stateflow diagrams execute when they are updated by the Simulink model during simulation. A Stateflow diagram can be updated when it is triggered or sampled by the Simulink model. This section summarizes all the settings necessary for implementing any of the following possible Simulink updates for a Stateflow chart block:

- “Defining a Triggered Stateflow Block” — Provides an example of a triggered Stateflow block in a Simulink model.
- “Defining a Sampled Stateflow Block” — Provides an example of a sampled Stateflow block in a Simulink model.
- “Defining an Inherited Stateflow Block” — Provides an example of a Stateflow block that inherits its sample time in a Simulink model.
- “Defining a Continuous Stateflow Block” — Provides an example of a continuously sampled Stateflow block in a Simulink model.
- “Defining Function Call Output Events” — Provides an example of a Stateflow block that triggers a subsystem in a Simulink model with a function call.
- “Defining Edge-Triggered Output Events” — Provides an example of a Stateflow block that triggers a subsystem in a Simulink model with a signal edge.

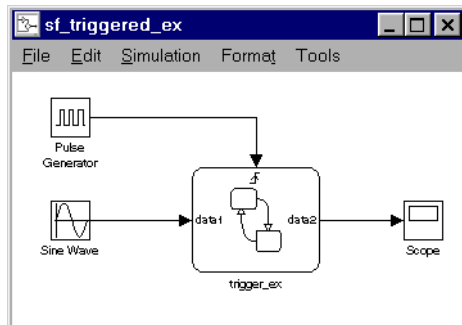
Defining a Triggered Stateflow Block

These are essential conditions that define an edge-triggered Stateflow block:

- The chart **Update method** (set in the **Chart Properties** dialog box) is set to **Triggered or Inherited**. (See “Specifying Chart Properties” on page 9-4.)
- The chart has an **Input from Simulink** event defined and an edge-trigger type specified. (See “Defining Input Events” on page 6-11.)

Triggered Stateflow Block Example

A Pulse Generator block connected to the trigger port of the Stateflow block is an example of an edge-triggered Stateflow block.



The **Input from Simulink** event has a **Rising Edge** trigger type. If more than one **Input from Simulink** event is defined, the sample times are determined by Simulink to be consistent with various rates of all the incoming signals. The outputs of a triggered Stateflow block are held after the execution of the block.

Defining a Sampled Stateflow Block

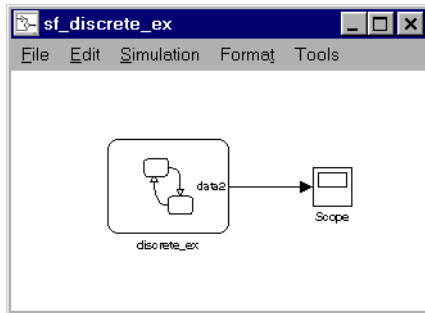
There are two ways you can define a sampled Stateflow block. Setting the chart **Update method** (set in the **Chart Properties** dialog box) to **Sampled** and entering a **Sample Time** value define a sampled Stateflow block. (See “Specifying Chart Properties” on page 9-4.)

Alternatively, you can add and define an **Input from Simulink** data object. Data is added and defined using either the graphics editor **Add** menu or the Explorer. (See “Sharing Input and Output Data with Simulink” on page 6-35.) The chart sample time is determined by Simulink to be consistent with the rate of the incoming data signal.

The **Sample Time** (set in the **Chart Properties** dialog box) takes precedence over the sample time of any **Input from Simulink** data.

Sampled Stateflow Block Example

You specify a discrete sample rate to have Simulink trigger a Stateflow block that is not explicitly triggered via the trigger port. You can specify a **Sample Time** in the Stateflow diagram’s **Chart properties** dialog box. The Stateflow block is then called by Simulink at the defined, regular sample times.



The outputs of a sampled Stateflow block are held after the execution of the block.

Defining an Inherited Stateflow Block

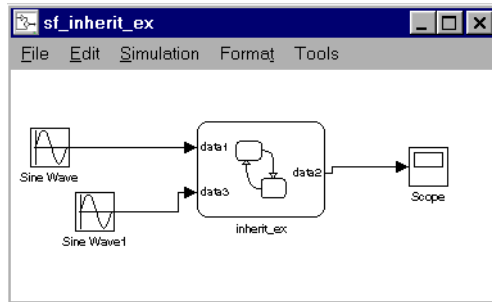
These are essential conditions that define an inherited trigger Stateflow block:

- The chart **Update method** (set in the **Chart Properties** dialog box) is set to **Triggered** or **Inherited**. (See “Specifying Chart Properties” on page 9-4)
- The chart has an **Input from Simulink** data object defined (added and defined using either the graphics editor **Add** menu or the Explorer). (See “Sharing Input and Output Data with Simulink” on page 6-35.) The chart sample time is determined by Simulink to be consistent with the rate of the incoming data signal.

Inherited Stateflow Block Example

Simulink can trigger a Stateflow block that is not explicitly triggered by a trigger port or a specified discrete sample time. In this case, the Stateflow block is called by Simulink at a sample time determined by Simulink.

In this example, more than one **Input from Simulink** data object is defined. The sample times are determined by Simulink to be consistent with the rates of both incoming signals.



The outputs of an inherited trigger Stateflow block are held after the execution of the block.

Defining a Continuous Stateflow Block

To define a continuous Stateflow block, set the chart **Update method** in the **Chart** dialog to **Continuous**. See “Specifying Chart Properties” on page 9-4.

Considerations in Choosing Continuous Update

The availability of intermediate data makes it possible for the solver to back up in time to precisely locate a zero crossing. Refer to the Using Simulink documentation for further information on zero crossings. Use of the intermediate time point information can provide increased simulation accuracy.

To support the **Continuous** update method, Stateflow keeps an extra copy of all its data.

In most cases, including continuous-time simulations, the **Inherited** method provides consistent results. The timing of state and output changes of the Stateflow block is entirely consistent with that of the continuous plant model.

There are situations, such as the following, when changes within the Stateflow block must be felt immediately by the plant and a **Continuous** update is needed:

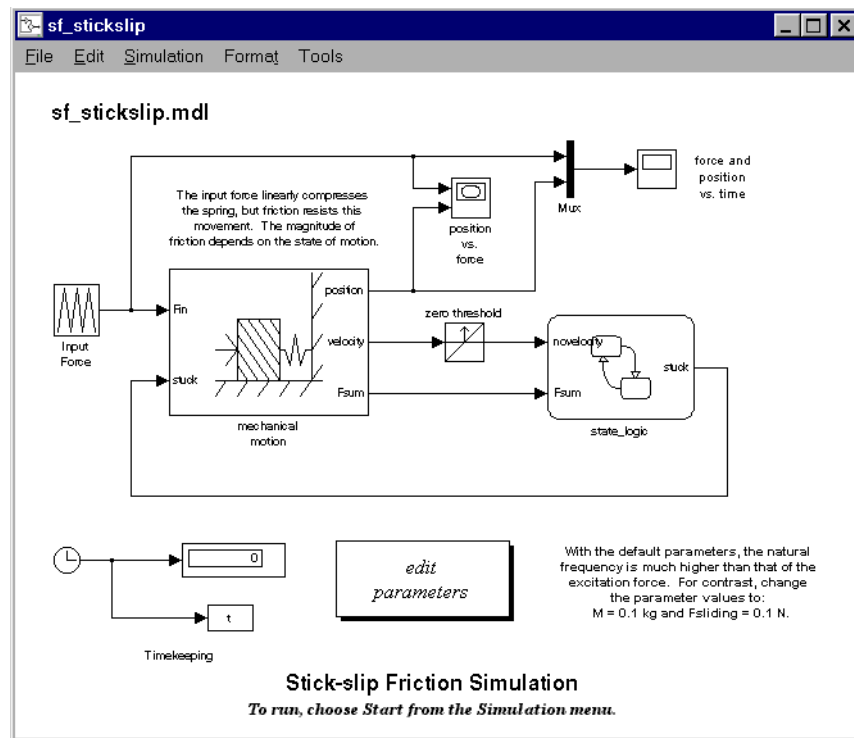
- Data output to Simulink that is a direct function of data input from Simulink and then updated by the Stateflow diagram (state **during** actions in particular)

- Models in which Stateflow states correspond to intrinsic physical states, such as the onset of static friction or the polarity of a magnetic domain. These states are in contrast to states that are assigned, for example, as modes of control strategy.

Continuous Stateflow Block Example

Simulink awakens (samples) the Stateflow block at each step in the simulation, as well as at intermediate time points that might be requested by the Simulink solver. This method is consistent with the continuous method in Simulink.

In the following example (provided in the Examples/Stick-Slip Friction Demonstration block), the chart **Update method** (set in the **Chart Properties** dialog box) is set to **Continuous**.



Defining Function Call Output Events

This topic shows you how to trigger a function-call subsystem in Simulink with a Function Call output event in a Stateflow diagram. It assumes that you already have in place a programmed function-call subsystem and a Stateflow block in the Simulink model. Use the following steps to connect the Stateflow block to the function-call subsystem and trigger it during simulation.

- 1 In the Stateflow diagram editor, from the **Add** menu, select **Data**.

A pop-up menu of different event scopes appears.

- 2 From the pop-up menu select **Output to Simulink**.

The **Event** dialog appears with a default name of event and a **Scope** of **Output to Simulink**.

- 3 In the **Event** dialog, in the **Trigger** field, select **Function Call** .

- 4 Name the event appropriately and select **OK** to close the dialog.

An output port with the name of the event you add appears on the right side of the Stateflow block.

- 5 From the Simulink library browser **Ports & Subsystems** library, place a function-call subsystem in the Simulink model.

You can also create a function-call subsystem by adding a subsystem to the model and adding a Trigger port to the subsystem. In the **Triggerport parameters** dialog for the Trigger block, set the **Trigger type** field to **function-call**.

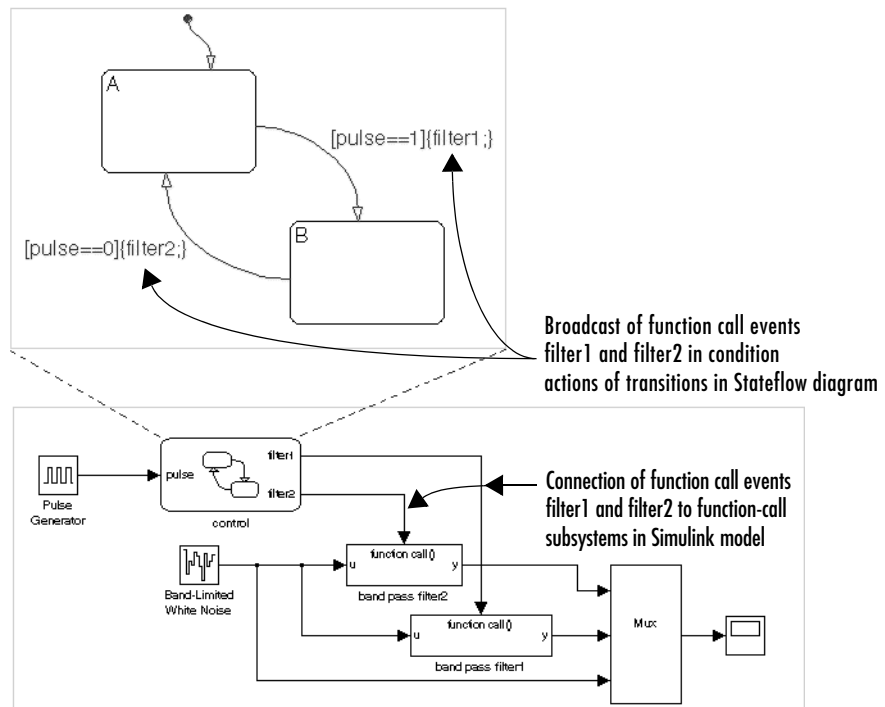
- 6 Connect the output port on the Stateflow block for the **Function Call** trigger **Output to Simulink** event you add to the function-call trigger input port of the subsystem.

You should avoid placing any other blocks in the connection lines between the Stateflow block and the function-call subsystem for Stateflow blocks that have feedback loops from a block triggered by a function call event.

Note You cannot connect a function-call output event from Stateflow to a Simulink Demux block in order to trigger multiple subsystems.

- 7 To execute the function-call subsystem, include an event broadcast of the function call output event in the actions of the Stateflow diagram as shown in the following example.

Function Call Output Events Example

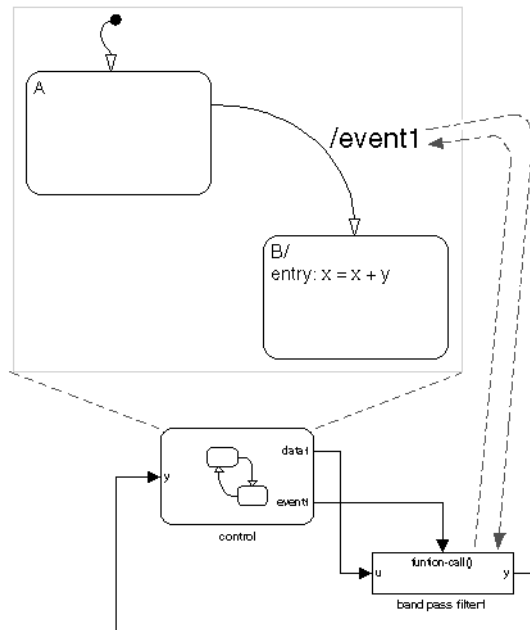


The control Stateflow block has one data input called `pulse` and two function call output events called `filter1` and `filter2`. A pulse generator provides input data to the control block. Each function call output event is attached to a subsystem in the Simulink model that is set to trigger by a function call.

Each transition in the control chart has a condition based on the size of the input pulse. When taken, each transition broadcasts a function call output event that determines whether to make a function call to filter1 or filter2. If the **Output to Simulink** function call event filter1 is broadcast, the band pass filter1 subsystem executes. If the **Output to Simulink** function call event filter2 is broadcast, the band pass filter2 subsystem executes. When either of these subsystems is finished executing, control is returned to the control Stateflow block for the next execution step. In this way, the Stateflow block controls the execution of band pass filter1 and band pass filter2.

Function Call Semantics Example

In this example the transition from state A to state B (in the Stateflow diagram) has a transition action that specifies the broadcast of event1. event1 is defined in Stateflow to be an **Output to Simulink** with a **Function Call** trigger type. The Stateflow block output port for event1 is connected to the trigger port of the band pass filter1 Simulink block. The band pass filter1 block has its **Trigger type** field set to **Function Call**.



This sequence is followed when state A is active and the transition from state A to state B is valid and is taken:

- 1 State A exit actions execute and complete.
- 2 State A is marked inactive.
- 3 The transition action is executed and completed.

In this case the transition action is a broadcast of event1. Because event1 is an event output to Simulink with a function call trigger, the band pass `filter1` block executes and completes, and then returns to the next statement in the execution sequence. The value of `y` is fed back to the Stateflow diagram.

- 4 State B is marked active.
- 5 State B entry actions execute and complete ($x = x + y$). The value of `y` is the updated value from the band pass `filter1` block.
- 6 The Stateflow diagram goes back to sleep, waiting to be awakened by another event.

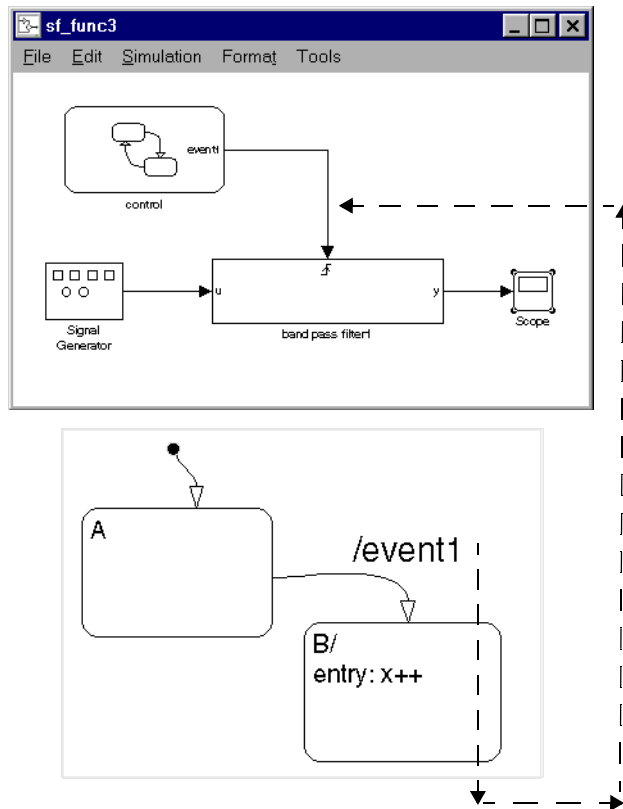
Defining Edge-Triggered Output Events

Simulink controls the execution of edge-triggered subsystems with output events. These are essential conditions that define this use of triggered output events:

- The chart has an **Output to Simulink** event with the trigger type **Either Edge**. See “Defining Output Events” on page 6-13.
- The Simulink block connected to the edge-triggered **Output to Simulink** event has its own trigger type set to the equivalent edge triggering.

Edge-Triggered Semantics Example

In this example the transition from state A to state B (in the Stateflow diagram) has a transition action that specifies the broadcast of event1. event1 is defined in Stateflow to be an **Output to Simulink** with an **Either edge** trigger type. The Stateflow block output port for event1 is connected to the trigger port of the band pass filter1 Simulink block. The band pass filter1 block has its **Trigger type** field set to **Either edge**.



This sequence is followed when state A is active and the transition from state A to state B is valid and is taken:

- 1 State A exit actions execute and complete.

- 2 State A is marked inactive.
- 3 The transition action, an edge-triggered **Output to Simulink** event broadcast, is registered (but not executed). Simulink is controlling the execution and execution control does not shift until the Stateflow block completes.
- 4 State B is marked active.
- 5 State B entry actions execute and complete ($x = x++$).
- 6 The Stateflow diagram goes back to sleep, waiting to be awakened by another event.
- 7 The band pass `filter1` block is triggered, executes, and completes.

Creating Chart Libraries

A Stateflow chart library is a Simulink block library that contains Stateflow Chart blocks (and, optionally, other types of Simulink blocks as well). Just as Simulink libraries serve as repositories of commonly used blocks, chart libraries serve as repositories of commonly used charts.

You create a chart library in the same way you create other types of Simulink libraries. First, create an empty chart library by selecting **Library** from the **New** submenu of Simulink's **File** menu. Then create or copy Chart blocks into the library just as you would create or copy Chart blocks into a Stateflow model.

You use chart libraries in the same way you use other types of Simulink libraries. To include a chart from a library in your Stateflow model, copy or drag the chart from the library to the model. Simulink creates a link from the instance in your model to the instance in the library. This allows you to update all instances of the chart simply by updating the library instance.

Note Events parented by a library Stateflow machine are invalid. Stateflow allows you to define such events but flags them as errors when parsing a model.

MATLAB Workspace Interfaces

The MATLAB workspace is an area of memory normally accessible from the MATLAB command line. It maintains a set of variables built up during a MATLAB session.

Examining the MATLAB Workspace in MATLAB

Two commands, `who` and `whos`, show the current contents of the workspace. The `who` command gives a short list, while `whos` also gives size and storage information.

To delete all the existing variables from the workspace, enter `clear` at the MATLAB command line.

See the MATLAB online or printed documentation for more information.

Interfacing the MATLAB Workspace in Stateflow

Stateflow charts have the following access to the MATLAB workspace:

- You can access MATLAB data or MATLAB functions in Stateflow action language with the `m1` namespace operator or the `m1` function.
See “Using MATLAB Functions and Data in Actions” on page 7-27 for more information.
- You can use the MATLAB workspace to initialize chart data at the beginning of a simulation.
See the subsection “Initialize from” on page 6-31 of the topic “Setting Data Properties in the Data Dialog” on page 6-25.
- You can save chart data to the workspace at the end of a simulation.
See “Save final value to base workspace” on page 6-32 for more information.

Interface to External Sources

Any source of data, events, or code that is outside a Stateflow diagram, its Stateflow machine, or its Simulink model, is considered external to that Stateflow diagram. Stateflow can interface data and events from external sources to your Stateflow chart, as described in the following topics:

- “Exported Events” on page 9-25 — Describes events that a Stateflow chart exports to locations outside itself.
- “Imported Events” on page 9-27 — Describes events that a Stateflow chart imports from locations outside itself.
- “Exported Data” on page 9-29 — Describes data that a Stateflow chart exports to locations outside itself.
- “Imported Data” on page 9-31 — Describes data that a Stateflow chart imports from locations outside itself.

See Chapter 6, “Defining Events and Data,” for information on defining events and data.

You can include external source code in the **Target Options** section of the **Target Builder** dialog box. (See “Integrating Custom Code with Stateflow Targets” on page 12-29.)

Exported Events

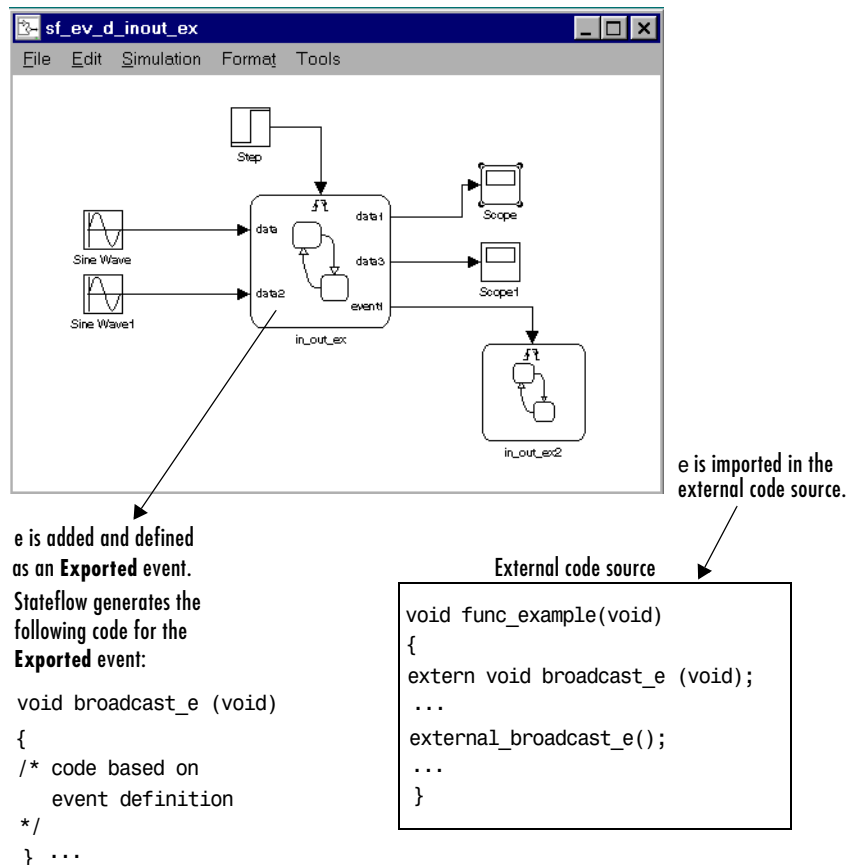
You might want an external source (outside the Stateflow diagram, its Stateflow machine, and its Simulink model) to be able to broadcast an event. By defining an event’s scope to be **Exported**, you make that event available to external sources for broadcast purposes. Exported events must be parented by the Stateflow machine, because the machine has the highest level in the Stateflow hierarchy and can interface to external sources. The Stateflow machine also retains the ability to broadcast the exported event. Exporting the event does not imply anything about what the external source does with the information. It is the responsibility of the external source to include the **Exported** event (in the manner appropriate to the source) to make use of the right to broadcast the event.

If the external source for the event is another Stateflow machine, then that machine must define the event as an **Exported** event and the other machine must define the same event as **Imported**. Stateflow generates the appropriate export and import event code for both machines.

Consider a real-world example to clarify when to define an **Exported** event. You have purchased a communications pager. There are a few people you want to be able to page you, so you give those people your personal pager number. These people now know your pager number and can call that number and page you at any time. You do not usually page yourself, but you can do so. Telling someone the pager number does not mean they have heard and recorded the number. It is the other person's responsibility to retain the number.

Exported Event Example

This example shows the format required in the external code source (custom code) to take advantage of an **Exported** event generated in Stateflow.



Imported Events

You might want to broadcast an event that is defined externally (outside the Stateflow diagram, its Stateflow machine, and its Simulink model). By defining an event's scope to be **Imported**, you can broadcast the event anywhere within the hierarchy of that machine (including any offspring of the machine).

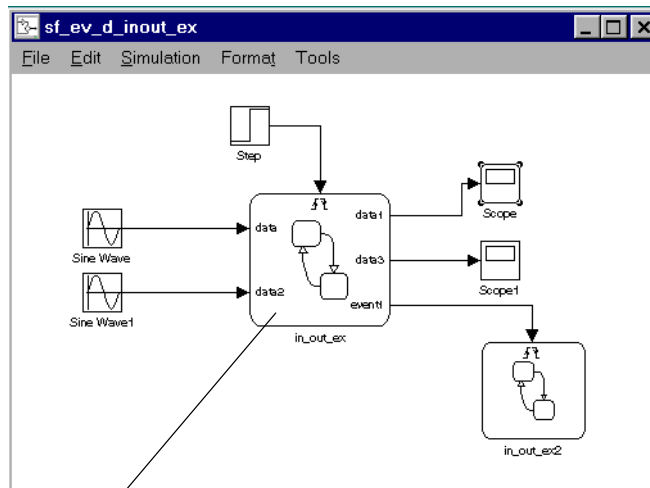
An imported event's parent is external. However, the event needs an adoptive parent to resolve symbols for code generation. An imported event's adoptive parent must be the Stateflow machine, because the machine has the highest level in the Stateflow hierarchy and can interface to external sources. It is the responsibility of the external source to make the imported event available (in the manner appropriate to the source).

If the external source is another Stateflow machine, the source machine must define the same event as **Exported**. Stateflow generates the appropriate import and export event code for both machines.

The preceding pager example for exported events can clarify the use of imported events. For example, someone buys a pager and tells you that you might want to use this number to page them in the future and they give you the pager number to record. You can then use that number to page that person.

Imported Event Example

The following example shows the format required in an external code source (custom code) to generate an **Imported** event in Stateflow.



e is added and defined as an Imported event.

Stateflow generates the following code for the **Imported event**:

```
extern void broadcast_e (void);
```

External code source

```
void broadcast_e (void)
{
  ...
}
```

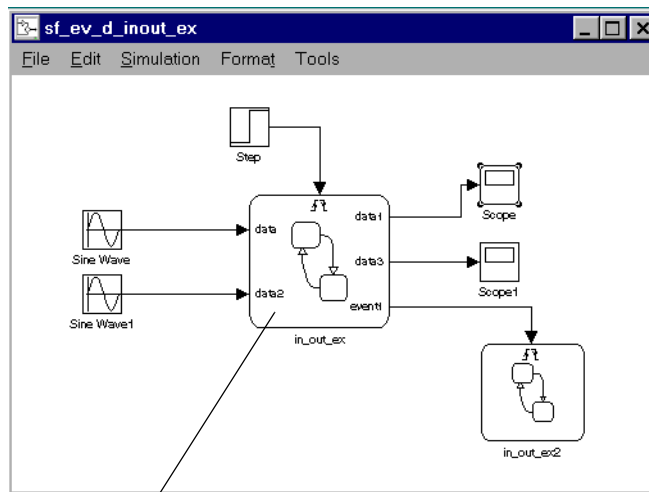
Exported Data

You might want an external source (outside the Stateflow diagram, its Stateflow machine, and its Simulink model) to be able to access a data object. By defining a data object's scope as **Exported**, you make it accessible to external sources. Exported data must be parented by the Stateflow machine, because the machine has the highest level in the Stateflow hierarchy and can interface to external sources. The Stateflow machine also retains the ability to access the exported data object. Exporting the data object does not imply anything about what the external source does with the data. It is the responsibility of the external source to include the exported data object (in the manner appropriate to the source) to make use of the right to access the data.

If the external source is another Stateflow machine, then that machine defines an exported data object and the other machine defines the same data object as imported. Stateflow generates the appropriate export and import data code for both machines.

Exported Data Example

The following example shows the format required in the external code source (custom code) to import a Stateflow exported data object:



ext_data is added and defined as an **Exported** data.

Stateflow generates the following code for the **Exported** data:

```
int ext_data;
```

External code source

```
extern int ext_data;
void func_example(void)
{
...
ext_data = 123;
...
}
```

ext_data is defined as imported in the external code source.

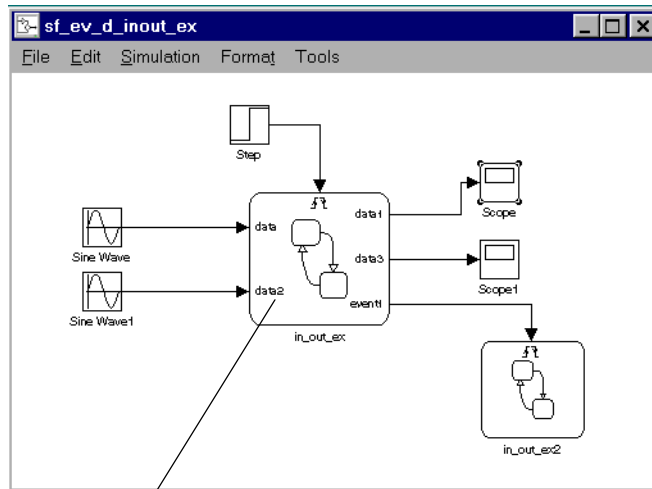
Imported Data

Similarly, you might want to access a data object that is externally defined outside the Stateflow diagram, its Stateflow machine, and its Simulink model. If you define the data's scope as **Imported**, the data can be accessed anywhere within the hierarchy of the Stateflow machine (including any offspring of the machine). An imported data object's parent is external. However, the data object needs an adoptive parent to resolve symbols for code generation. An imported data object's adoptive parent must be the Stateflow machine, because the machine has the highest level in the Stateflow hierarchy and can interface to external sources. It is the responsibility of the external source to make the imported data object available (in the manner appropriate to the source).

If the external source for the data is another Stateflow machine, that machine must define the same data object as **Exported**. Stateflow generates the appropriate import and export data code for both machines.

Imported Data Example

This example shows the format required to retrieve imported data from an external code source (custom code).



ext_data is added and defined as an imported data
Stateflow generates the following code for the imported data:
extern int ext_data;

External code source

```
int ext_data;  
void func_example(void)  
{  
  ...  
}
```

ext_data is defined as exported in the external code source.

Truth Table Functions

Stateflow uses functions to capture programming steps that you call repeatedly in a Stateflow diagram. In truth tables, you specify logical behavior with conditions, decisions, and actions. For purely logical behavior, truth tables are easier to program, easier to maintain, and easier for others to read than graphical functions. Truth tables also help you complete your function by telling you whether you have specified enough or too many decisions for the conditions you specify. To learn how to construct successful truth tables, see the following sections:

What Is a Truth Table? (p. 10-2)	Describes the truth tables that Stateflow truth table functions implement.
Building a Simulink Model with a Stateflow Truth Table (p. 10-4)	Shows you how to create a Simulink model that calls and executes a Stateflow diagram with a truth table.
Programming a Truth Table (p. 10-19)	Describes procedures for programming a truth table.
Debugging a Truth Table (p. 10-34)	Shows you how to use the Parser and Debugging window during simulation to debug a truth table.
Options for Assigning Actions to Decisions in Truth Tables (p. 10-43)	Gives you a complete set of rules and options for assigning actions to decisions in a truth table.
Truth Table Editor Operations (p. 10-47)	Describes the editing operations available to you in the truth table editor.
Correcting Over- and Underspecified Truth Tables (p. 10-55)	Describes over- and underspecified truth tables detected by Stateflow error checking.
Model Coverage for Truth Tables (p. 10-58)	Describes and interprets the results of an example model coverage report for a truth table.
How Stateflow Realizes Truth Tables (p. 10-62)	Describes the way that Stateflow generates graphical functions to realize truth tables.

What Is a Truth Table?

Stateflow uses truth table functions to realize logical decision-making behavior that you call in action language. Stateflow truth tables contain conditions, decisions, and actions arranged like the following:

Condition	Decision 1	Decision 2	Decision 3	Default Decision
<code>x == 1</code>	T	F	F	-
<code>y == 1</code>	F	T	F	-
<code>z == 1</code>	F	F	T	-
Action	<code>t = 1</code>	<code>t = 2</code>	<code>t = 3</code>	<code>t = 4</code>

Each of the conditions entered in the Condition column must evaluate to true (nonzero value) or false (zero value). Outcomes for each condition are specified as T (true), F (false), or - (true or false). Each of the decision columns combines an outcome for each condition with a logical AND into a compound condition, that is referred to as a decision.

You evaluate a truth table one decision at a time, starting with Decision 1. If one of the decisions is true, you perform its action and truth table execution is complete. For example, if conditions 1 and 2 are false and condition 3 is true, Decision 3 is true and the variable `t` is set equal to 3. The remaining decisions are not tested and evaluation of the truth table is finished.

The last decision in the preceding example, Default Decision, covers all possible remaining decisions. If Decisions 1, 2, and 3 are false, then the Default Decision is automatically true and its action ($t = 4$) is executed. You can see this behavior when you examine the following equivalent pseudocode for the evaluation of the preceding truth table example:

Description	Pseudocode
Decision 1 Decision 1 Action	<code>if ((x == 1) & !(y == 1) & !(z == 1)) t = 1;</code>
Decision 2 Decision 2 Action	<code>elseif (!(x == 1) & (y == 1) !(z == 1)) t = 2;</code>
Decision 3 Decision 3 Action	<code>elseif (!(x == 1) & !(y == 1) (z == 1)) t = 3;</code>
Default Decision Default Decision Action	<code>else t = 4; endif</code>

Building a Simulink Model with a Stateflow Truth Table

Truth tables are a special Stateflow function that you program with logical behavior. You implement truth tables by specifying action language conditions, decisions, and actions. In this section, you learn how to create truth tables in Stateflow by building a simple model with a truth table in the following topic steps:

- 1** “What Is a Truth Table?” on page 10-2 — Describes the truth table concept using an example truth table that you create in “Building a Simulink Model with a Stateflow Truth Table” on page 10-4.
- 2** “Creating a Simulink Model” on page 10-5 — Starts by creating a model that executes a Stateflow block.
- 3** “Creating a Stateflow Truth Table” on page 10-8 — Creates a Stateflow diagram with a truth table for the Stateflow block.
- 4** “Calling a Truth Table in a Stateflow Action” on page 10-11 — Adds a flow diagram to the Stateflow diagram that calls the truth table.
- 5** “Creating Truth Table Data in Stateflow and Simulink” on page 10-14 — Shows you how to create the data in Stateflow and Simulink that the truth table changes during execution.

Once you build a model in this section, finish it by programming the truth table with its behavior in “Programming a Truth Table” on page 10-19.

Note The Stateflow diagram you create in this section contains the least amount of Stateflow programming possible. This might appeal to Simulink users who know little about Stateflow but want to use it to execute a truth table function.

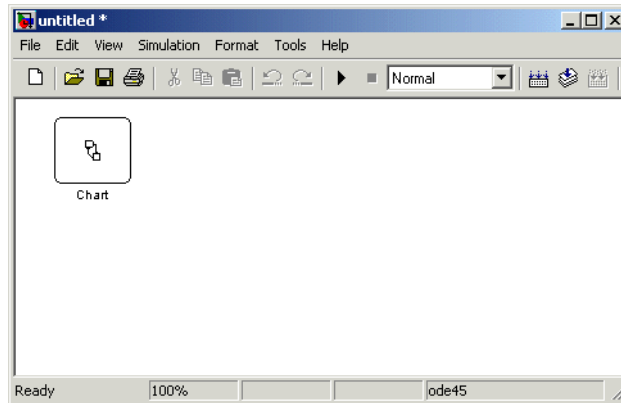
Creating a Simulink Model

To execute a truth table, you first need a Simulink model that calls a Stateflow block. Later, you create a Stateflow diagram for the Stateflow block that calls a truth table function. In this topic, you create a Simulink model that calls a Stateflow block with the following procedure:

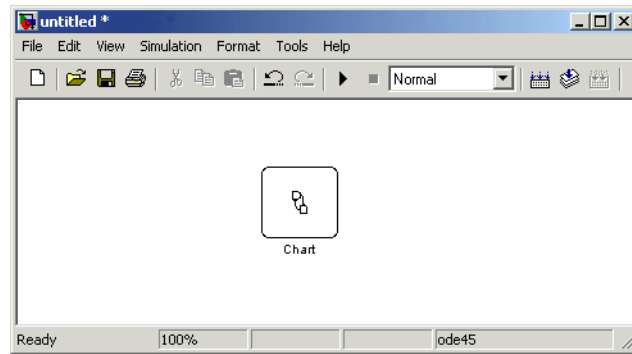
- 1 At the MATLAB prompt, enter the following command:

```
sfnew
```

An untitled Simulink model with a Stateflow block appears as shown.



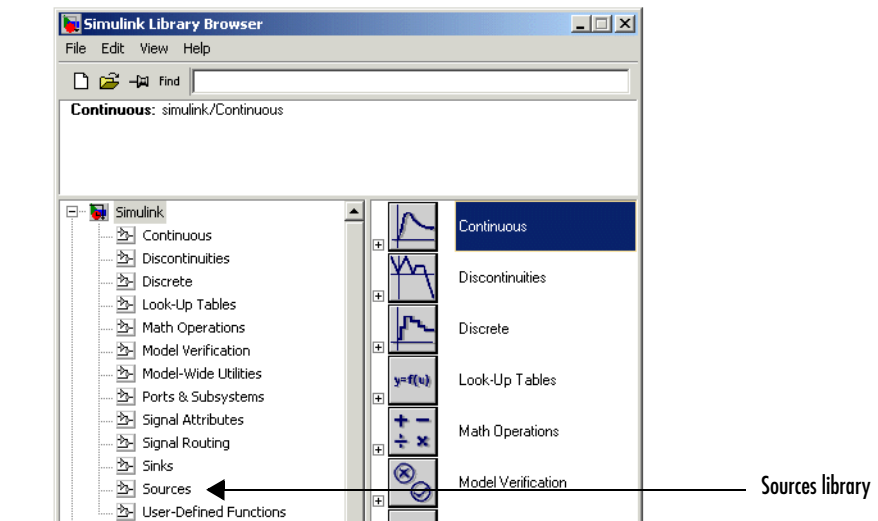
- 2 Click and drag the Stateflow block to the center of the Simulink window as shown.



This makes room for the blocks you add in the steps that follow.

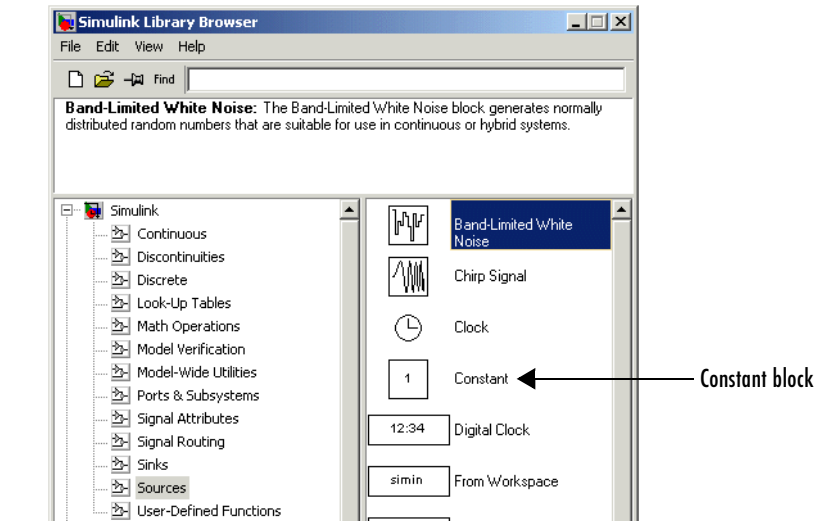
- 3 In the Simulink window, from the **View** menu, select **Library browser**.

The **Simulink Library Browser** window opens with the **Simulink** node expanded.



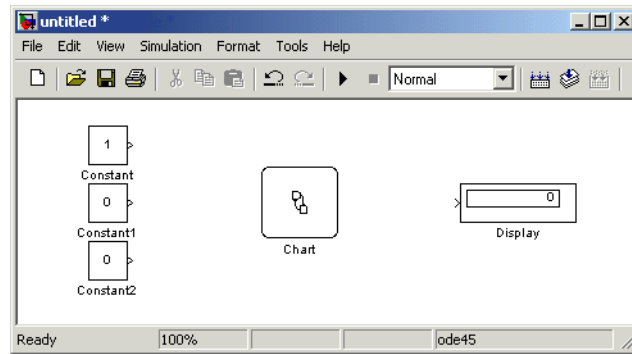
- 4 Under the **Simulink** node, select the **Sources** library.

The right pane of the **Simulink Library Browser** window displays the block members of the **Sources** library.



- 5 From the right pane of the Simulink Library Browser window, click and drag the Constant block to the left of the Stateflow block in the Simulink model.
- 6 Add two more Constant blocks to the left of the Chart block and a Display block (from the Sinks library) to the right of the Chart block.

Your model should now have the following appearance:



- 7 In the Simulink model window, double-click the top Constant block.
- 8 In the resulting **Block Parameters** dialog, change the **Constant value** field to 1 and select **OK** to close the dialog.
- 9 In the Simulink model window, from the **Simulation** menu, select **Simulation Parameters**.

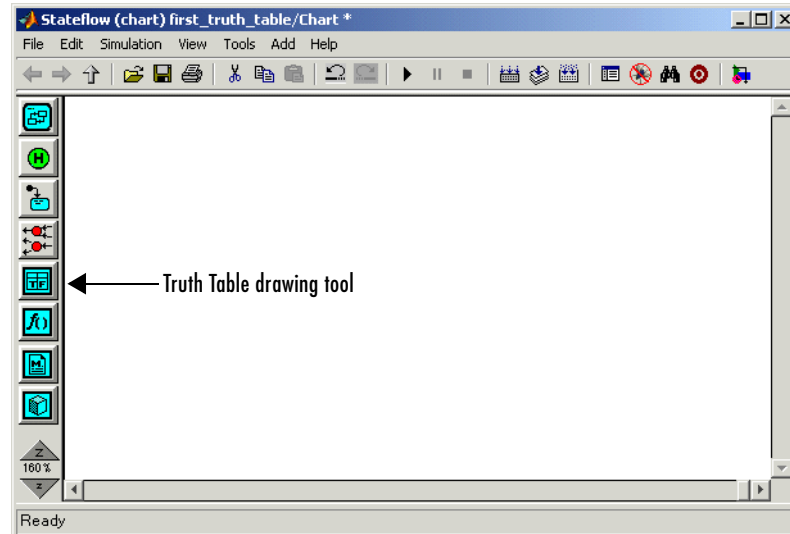
The **Simulation Parameters** dialog opens.


- 10 In the resulting **Simulation Parameters** dialog,
 - Leave the **Solver Options** field set at Variable-step.
 - Change the **Stop Time** to `inf`.
- 11 Click **OK** to accept these values and close the **Simulation Parameters** dialog.
- 12 From the Simulink **File** menu, select **Save As** and save the model as `first_truth_table.mdl`.

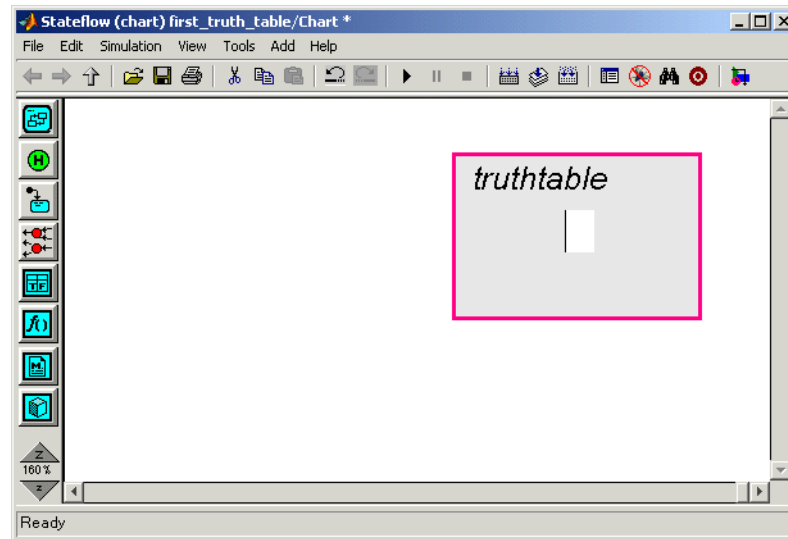
Creating a Stateflow Truth Table

You create a Simulink model in “Creating a Simulink Model” on page 10-5 that contains a Stateflow block. Now you need to open the Stateflow diagram for the block and specify a truth table for it in the following steps:

- 1 In the Simulink model, double-click the Stateflow block named Chart.
An empty Stateflow diagram editor appears.



- 2 In the Stateflow diagram editor, select the Truth Table drawing tool .
- 3 Move the cursor into the empty diagram area and notice that the cursor takes on the shape of a box.
- 4 Click to place a new truth table as shown.

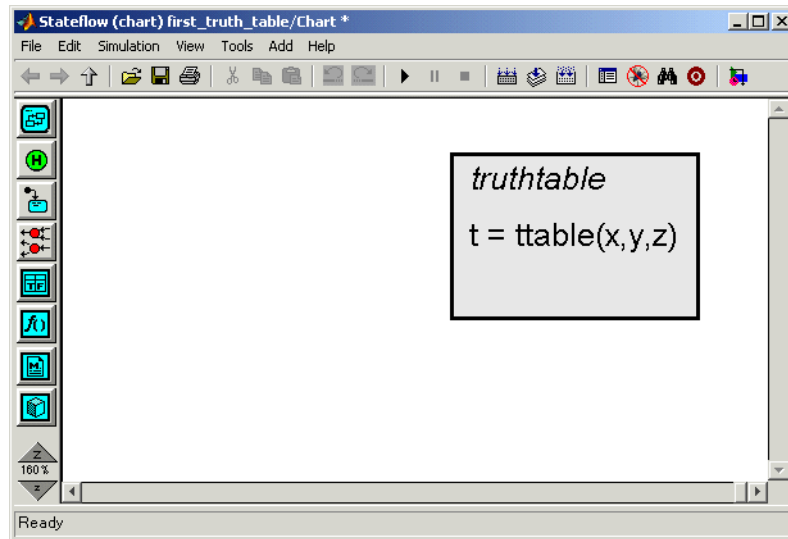


A shaded box appears with the title **truthtable** and a flashing text cursor in the middle of the box.

5 Enter the label text

```
t = ttable(x,y,z)
```

and click outside the truth table box.



The signature label you enter for the truth table defines its name (`ttable`), its arguments (`x`, `y`, and `z`), and its return value (`t`). Argument and return values can each be a scalar value or a matrix of values. Multiple return values are not allowed.

Note If you need to reedit the truth table label at any time, click the label to place an editing cursor in the text of the label.

Calling a Truth Table in a Stateflow Action


In “Creating a Stateflow Truth Table” on page 10-8, you create the truth table function `ttable` with the signature

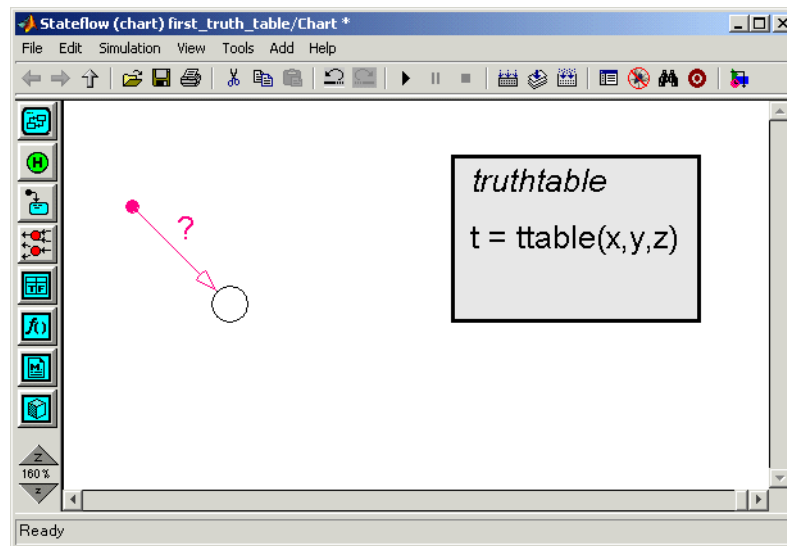
```
t = ttable(x,y,z)
```

Now you need to specify a call to the truth table function in the Stateflow diagram. Later, when the diagram executes during simulation, it calls the truth table.

You can call truth table functions from the actions of any state or transition. You can also call truth tables from other functions, including graphical functions and other truth tables. Also, if you export a truth table, you can call it from any Stateflow chart in the model.

Use the following steps to call the `ttable` function from the default transition of its own Stateflow diagram.

- 1 Select the Default Transition button  from the drawing toolbar.
- 2 Move the cursor to a location left of the truth table function and notice that the cursor takes on the shape of a downward-pointing arrow.
- 3 Click to place a default transition into a terminating junction.



- 4 Click the question mark character (?) that appears on the highlighted default transition.

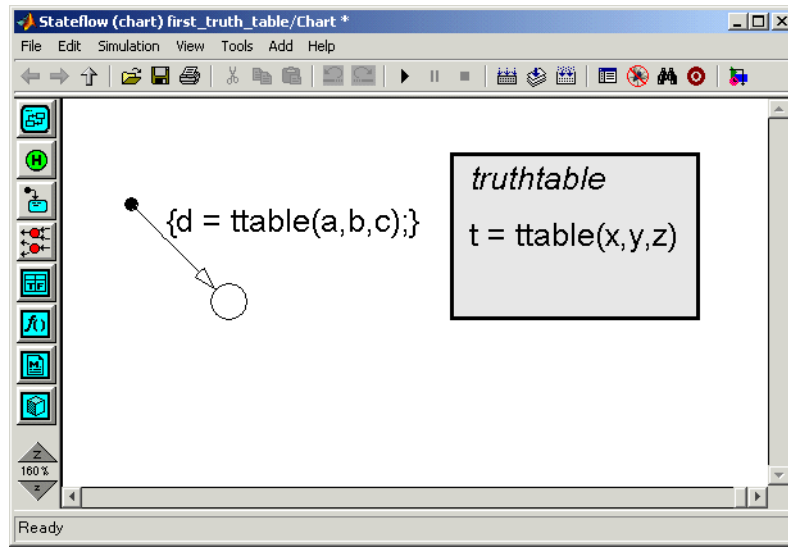
A blinking cursor in a text field appears for entering the label of the default transition.

5 Enter the text

```
{d = ttable(a,b,c);}
```

and click outside the transition label to finish editing it.

You might want to adjust the label's position by clicking and dragging it to a new location. The finished Stateflow diagram should have the following appearance:




The label on the default transition that you entered provides a condition action that calls the truth table with arguments and a return value. When Simulink triggers the Stateflow block during simulation, the default transition is taken and a call to the truth table `ttable` is made.

The call to the truth table in Stateflow action language must match the truth table signature. This means that the type of the return value `c` must match the type of the signature return value `z`, and the type of the arguments `a` and `b` must match the type of the signature arguments `x` and `y`. You ensure this with a later step in this section when you create the data that you use in Stateflow.

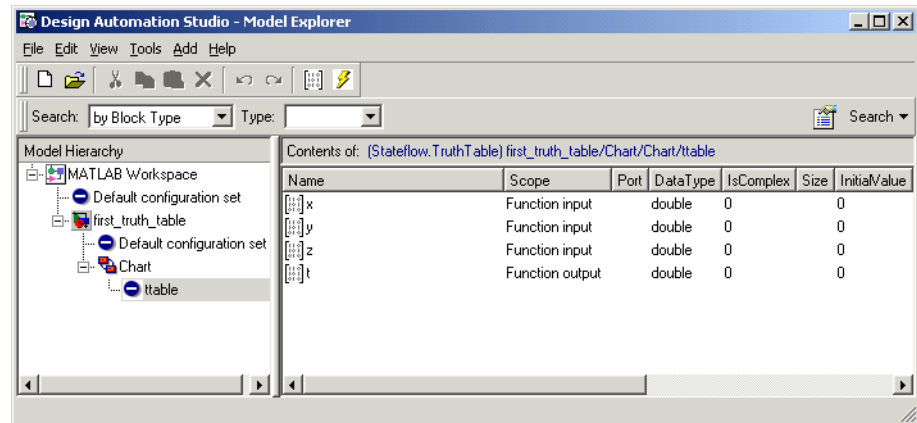
- From the **File** menu, select **Save** to save the model.

Creating Truth Table Data in Stateflow and Simulink

When you create a truth table with its own signature, you specify data for it in the form of a return value (t) and argument values (x, y, z). When you specify a call to a truth table, as you did in “Calling a Truth Table in a Stateflow Action” on page 10-11, you specify data that you pass to the return and argument values of the truth table (d, a, b, and c). Now you need to define this data for the Stateflow diagram in the following steps:

- In the Truth Table editor, select the Goto Explorer tool .

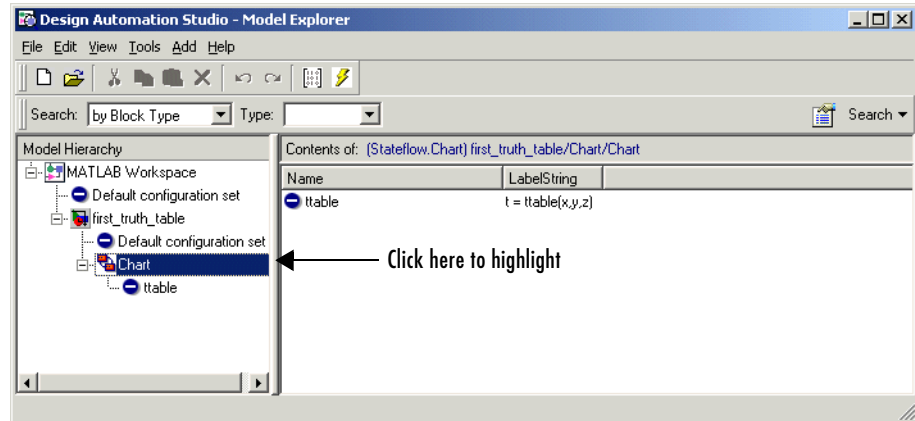
The **Model Explorer** window appears as shown.



The Goto Explorer button is a convenience for accessing the **Model Explorer** from the truth table editor. Notice in the **Model Hierarchy** pane (left pane) that the node for the function ttable is highlighted and that the **Contents** pane (right pane) displays the inputs (x, y, z) and outputs (t) for ttable. By default, these data are defined as scalars of type double. If you want to redefine these data with a different array size and type, you do it in the **Model Explorer**. However, no changes are necessary for this example.

Notice also in the **Model Hierarchy** pane that the node above the function ttable is a node for the Stateflow block Chart. This is the name of the Stateflow diagram that contains the truth table ttable.

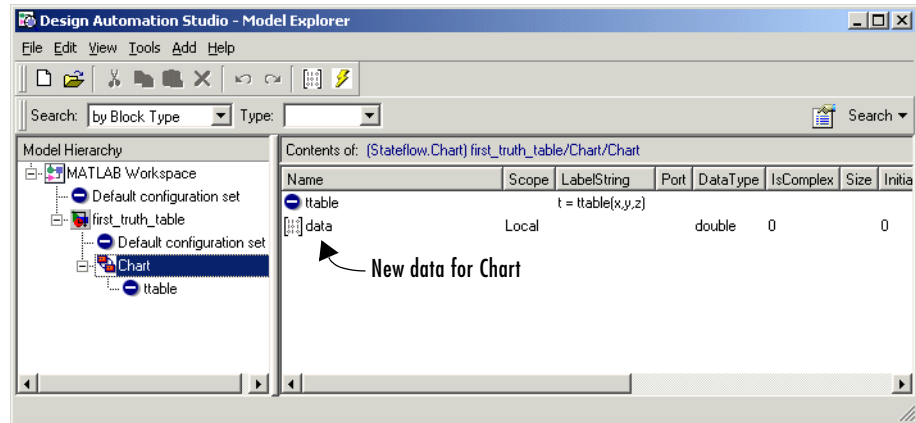
2 In the **Model Hierarchy** pane, select **Chart** as shown.



Notice that Chart contains no data in the **Contents** pane. You need to add the return and argument data used in calling `ttable`.

3 From the **Add** menu, select **Data**.

A scalar data of type double is added to the chart in the **Contents** pane of Explorer with the default name `data`.



This data matches argument x in type and size, but still needs a change in name and scope.

- 4 In the **Contents** pane, double-click the entry **data** in the **Name** column.

A small text field opens with the name **data** highlighted.

- 5 In the text field, change the name to **a** and press **Enter**.

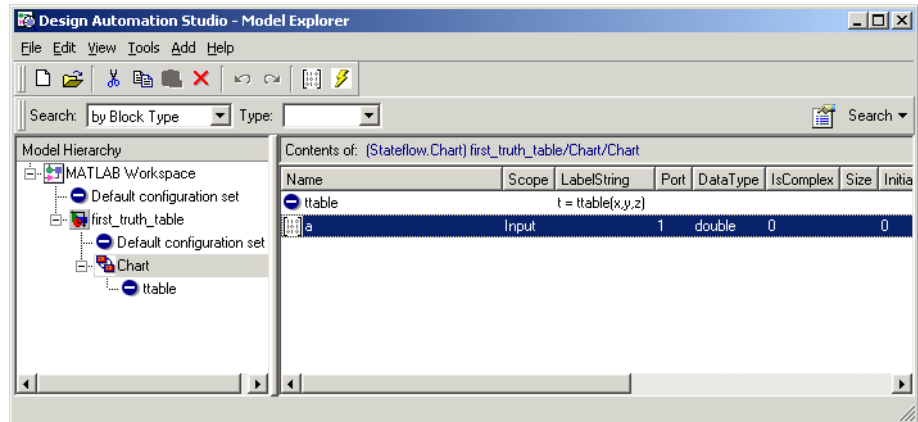
- 6 Click the entry **Local** under the **Scope** column.

A menu of selectable scopes appears with **Local** selected.

- 7 Select **Input** and press **Enter**.

The scope **Input** (short for **Input from Simulink**) means that Simulink provides the value for this data, which it passes to the Stateflow diagram through an input port on the Stateflow block.

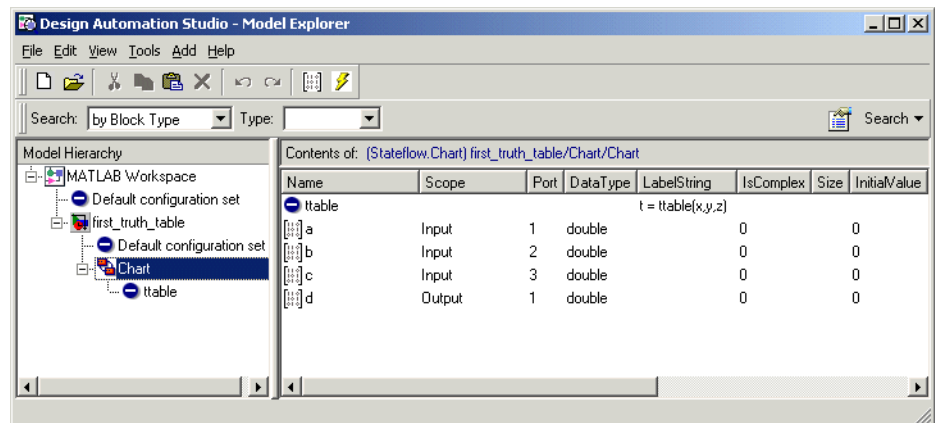
You should now see the new data **a** in the **Contents** pane as shown.



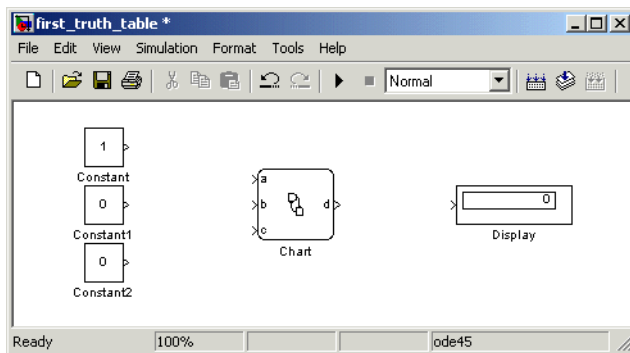
- 8 Use steps 3 through 7 to add the data b and c with the scope **Input**, and data d with a scope of **Output to Simulink**.

The scope **Output to Simulink** means that the Stateflow chart provides this data and passes it to Simulink through an output port on the Stateflow block.

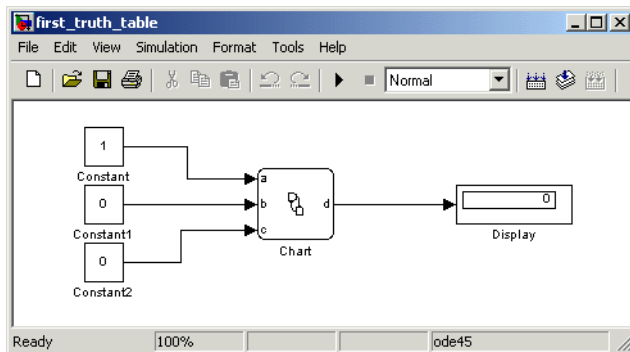
You should now see the following data in Stateflow Explorer:



The data a, b, c, and d match their counterparts x, y, z, and t in the truth table signature in size (scalar) and type (double), but have a source outside the Stateflow block. Notice that input ports for a, b, and c, and an output port for d appear for the Stateflow block in the Simulink model.



9 Complete connections to the Simulink diagram as shown.



10 From the **File** menu, select **Save** to save the model.

Programming a Truth Table

In the steps of “Building a Simulink Model with a Stateflow Truth Table” on page 10-4, you create a Simulink model with a Stateflow diagram that calls the truth table `ttable`. This truth table is empty and requires programming to specify its behavior. This section shows you how to program this truth table, with the topics in the following steps:

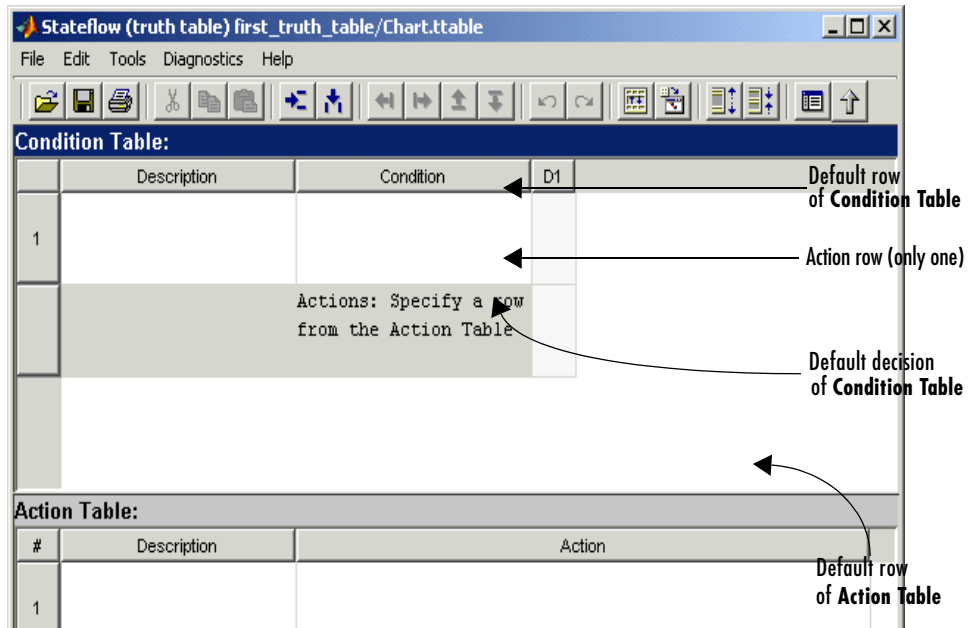
- 1 “Opening a Truth Table for Editing” on page 10-19 — You open a truth table editor for a truth table object in the Stateflow diagram to begin programming it.
- 2 “Entering Truth Table Conditions” on page 10-20 — Shows you how to enter conditions in the **Condition Table** of a truth table.
- 3 “Entering Truth Table Decisions” on page 10-23 — Shows you how to enter decisions in the **Condition Table** of a truth table.
- 4 “Entering Truth Table Actions” on page 10-26 — Shows you how to enter actions for each decision in the **Action Table** of a truth table.
- 5 “Assigning Truth Table Actions to Decisions” on page 10-29 — Shows you how to assign actions that you entered in the **Action Table** to each decision in the **Condition Table**.
- 6 “Adding Initial and Final Actions” on page 10-31 — Shows you how to specify special initial and final actions in the **Action Table** of a truth table.

When you finish programming `ttable` in this section, the example model you start in “Building a Simulink Model with a Stateflow Truth Table” on page 10-4 is finished. Continue by debugging the truth table in “Debugging a Truth Table” on page 10-34.

Opening a Truth Table for Editing

Once you create and label a truth table in a Stateflow diagram, you specify its logical behavior by editing its contents in the truth table editor. To open a truth table function in a Stateflow diagram, double-click it.

A blank truth table editor window appears.



The truth table editor is titled in *<model name>/<truth table name>* format in its header. An empty default truth table contains a **Condition Table** and an **Action Table**, each with one row. The **Condition Table** also contains a single decision column, **D1**, and a single action row.

Entering Truth Table Conditions

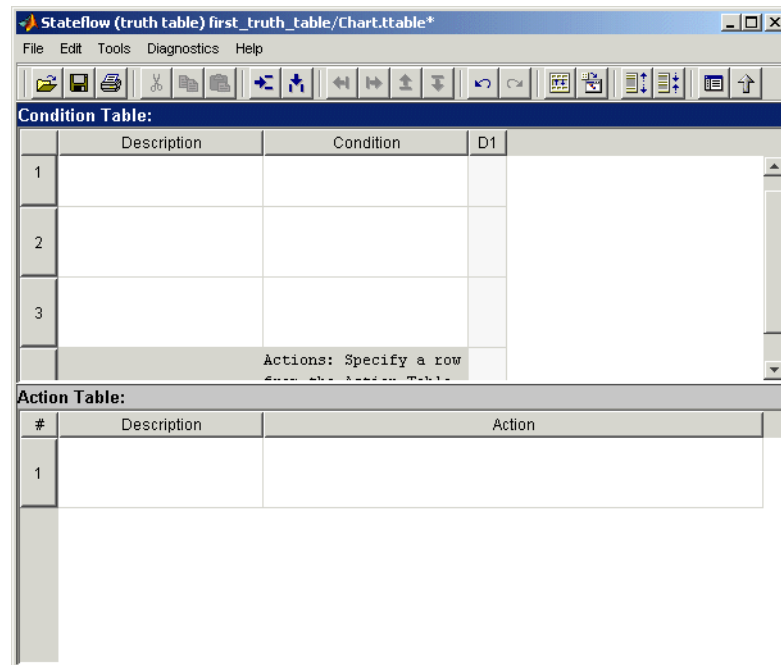
Conditions are the starting point for specifying logical behavior in a truth table. You open the truth table ttable for editing in “Opening a Truth Table for Editing” on page 10-19. In this topic, you start programming the behavior of ttable by specifying its conditions.

You enter conditions in the **Condition** column of the **Condition Table**. For each condition that you enter, you can also enter an optional description in the **Description** column. Use the following procedure to enter the conditions of the truth table ttable:

- 1 Click anywhere in the **Condition Table** to select it.

- 2 Click the Append Row tool  twice.

Two rows are appended to the bottom of the **Condition Table** for a total of three, as shown.



The Action row of the **Condition Table** is slightly obscured by the pane of the **Action Table**.

- 3 Click and drag the bar separating the **Condition Table** and the **Action Table** panes down to enlarge the **Condition Table** pane.
- 4 In the **Condition Table**, click the top cell of the **Description** column.

The cell is highlighted and a flashing text cursor appears in the cell.

- 5 Enter the following text:

```
x is equal to 1
```

Condition descriptions are optional, but they are carried into the generated code for the truth table as code comments.

- 6 Press the **Tab** key to select the next cell on the right in the **Condition** column.

Use **Shift+Tab** to select a cell on the left.

- 7 In the first row cell of the **Condition** column, enter the following text:

```
XEQ1 :
```

This is an optional condition label that you include with the condition. Press **Enter** after the label to place the label and condition on separate lines.

Condition labels must begin with an alphabetic character ([a-z][A-Z]) followed by any number of alphanumeric characters ([a-z][A-Z][0-9]) or an underscore (_). In the generated code for a truth table, the condition label becomes the name of a temporary data variable that stores the outcome of its condition. If no label is entered, Stateflow names a temporary variable of its own.

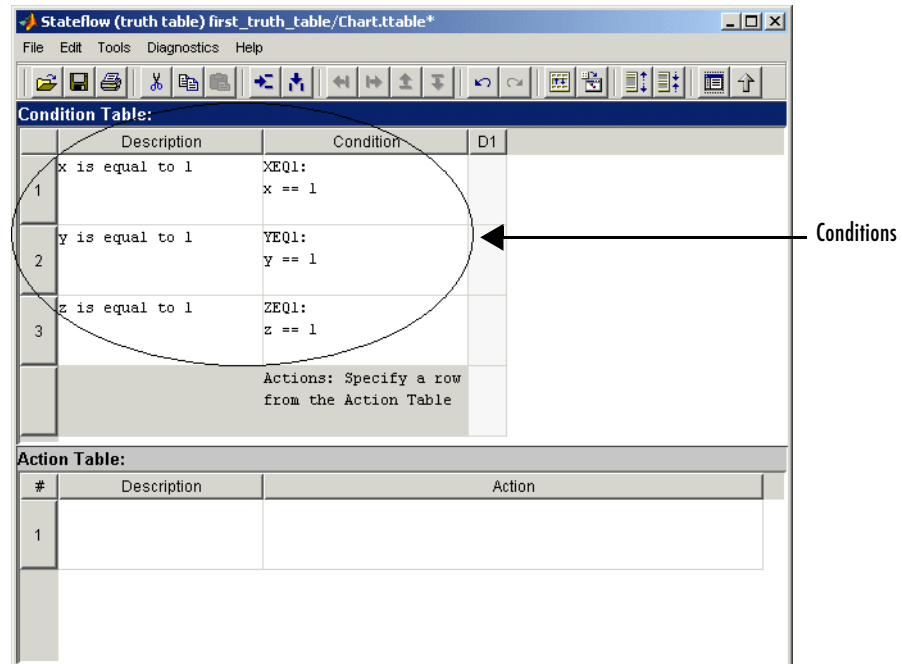
- 8 Press **Enter** and enter the following text:

```
x == 1
```

This is the actual condition. Each condition you enter must evaluate to zero (false) or nonzero (true). You can use optional brackets in the condition (for example, [x == 1]) as you would in Stateflow action language.


You can use data passed to the truth table function through its arguments in truth table conditions. The preceding condition tests whether the argument `x` is equal to 1. You can also use data defined for parent objects of the truth table in Stateflow, including the Stateflow diagram.

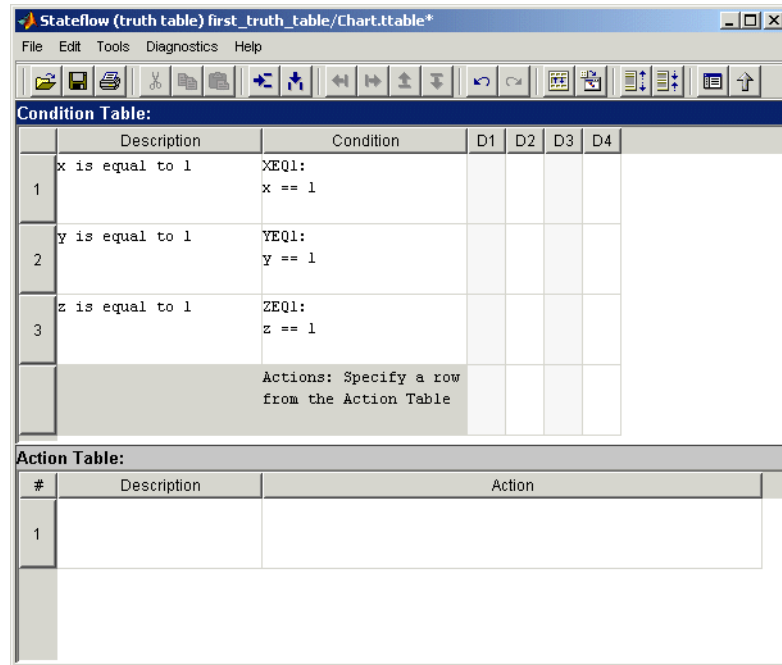
- 9 Use the preceding steps to enter the remaining two conditions, as shown in the following:



Entering Truth Table Decisions

Each decision column (**D1**, **D2**, and so on) binds a group of condition outcomes together with an AND relationship into a decision. The allowed values for the condition outcomes in a decision are T (true), F (false), and - (true or false). In “Entering Truth Table Conditions” on page 10-20 you enter conditions for the truth table `ttable` in the truth table editor. Continue by entering decisions in the decision columns with the following steps:

- 1 Click anywhere in the **Condition Table** to make sure it is selected.
- 2 Click the Append Column toolbar button  three times to add three columns to the right end of the **Condition Table** for a total of four as shown.



- 3 Click the top cell in decision column **D1**.

The cell is highlighted and a flashing text cursor appears in the cell.

- 4 Press the space bar once to enter a value of T.

Pressing the space bar toggles through the possible values of F, -, and T. You can also enter these characters directly. All other entries are rejected.

- 5 Press the down arrow key to advance to the next cell down in the **D1** column.

In the decision columns, you can use the arrow keys to advance to another cell in any direction. You can also use **Tab** and **Shift+Tab** to advance left or right in these cells.

- 6 Enter the remaining values for the decision columns, as shown in the following:

Stateflow (truth table) first_truth_table/Chart.ttable

File Edit Tools Diagnostics Help

Condition Table:

	Description	Condition	D1	D2	D3	D4
1	x is equal to 1	XEQ1: x == 1	T	F	F	-
2	y is equal to 1	YEQ1: y == 1	F	T	F	-
3	z is equal to 1	ZEQ1: z == 1	F	F	T	-
	Actions: Specify a row from the Action Table					

Action Table:

#	Description	Action
1		

During execution of the truth table, decisions are tested in left to right order. The order of testing for individual condition outcomes within a decision is undefined. The current implementation of truth tables in Stateflow evaluates the conditions for each decision in top-down order (first condition 1, then condition 2, and so on). Because this implementation is subject to change in the future, you must not depend on a particular evaluation order.

The Default Decision Column

The last decision column in `ttable`, **D4**, is the default decision for this truth table. The default decision covers any remaining decisions not tested for in preceding decision columns to the left. You enter a default decision as the last decision column on the right with an entry of - for all conditions in the decision, where - represents any outcome for the condition.

In the preceding example, the following decisions are specified by the default decision column, **D4**:


Condition	Decision 4	Decision 5	Decision 6	Decision 7	Decision 8
x == 1	F	T	F	T	T
y == 1	F	F	T	T	T
z == 1	F	T	T	F	T

Note The default decision column must be the last column on the right in the **Condition Table**.

Entering Truth Table Actions

During execution of the truth table, decisions are tested in left to right order. When a decision is realized during execution of the truth table, the action in the **Action Table** specified in the **Actions** row for that decision column is executed and the truth table is exited.

In “Entering Truth Table Decisions” on page 10-23 you enter decisions in the truth table editor. The next step is to enter the actions you want to occur for each decision in the **Action Table**. Later, you assign these actions to their decisions in the **Actions** row of the **Condition Table**.

- 1 Click anywhere in the **Action Table** to select it.
- 2 Click the Append Row toolbar button  three times to add three rows to the bottom of the **Action Table**.
- 3 Click and drag the bottom border of the truth table editor window down to enlarge it and clearly show all rows of the **Action Table**, as shown.

Condition Table:

	Description	Condition	D1	D2	D3	D4
1	x is equal to 1	XEQ1: x == 1	T	F	F	-
2	y is equal to 1	YEQ1: y == 1	F	T	F	-
3	z is equal to 1	ZEQ1: z == 1	F	F	T	-
		Actions: Specify a row from the Action Table				

Action Table:

#	Description	Action
1		
2		
3		
4		

- 4** Click the top cell in the **Description** column of the **Action Table**.

The cell is highlighted and a flashing text cursor appears in the cell.

- 5** Enter the following description:

```
set t to 1
```

Action descriptions are optional, but are carried into the generated code for the truth table as code comments.

- 6** Press **Tab** to select the next cell on the right, in the **Action** column.

- 7** Enter the following text:

A1:

You begin an action with an optional label followed by a colon (:). Later, you enter these labels in the **Actions** row of the **Condition Table** to specify an action for each decision column. Like condition labels, action labels must begin with an alphabetic character ([a-z][A-Z]) followed by any number of alphanumeric characters ([a-z][A-Z][0-9]) or an underscore (_).

- 8** Press **Enter** and enter the following text:

```
t=1;
```

You can use data passed to the truth table function through its arguments and return value in truth table actions. The preceding action, `t=1`, sets the value of the return value `t`. You can also specify actions with data defined for a parent object of the truth table, including the Stateflow diagram. Truth table actions can also broadcast or send events that are defined for the truth table, or for a parent, such as the diagram (chart) itself.

If you omit the semicolon at the end of an action, the result of the action is echoed to the MATLAB window when it is executed during simulation. Use this as a debugging tool.

- 9** Enter the remaining actions in the **Action Table**, as shown in the following:

Condition Table:

	Description	Condition	D1	D2	D3	D4
1	x is equal to 1	XEQ1: x == 1	T	F	F	-
2	y is equal to 1	YEQ1: y == 1	F	T	F	-
3	z is equal to 1	ZEQ1: z == 1	F	F	T	-
	Actions: Specify a row from the Action Table					

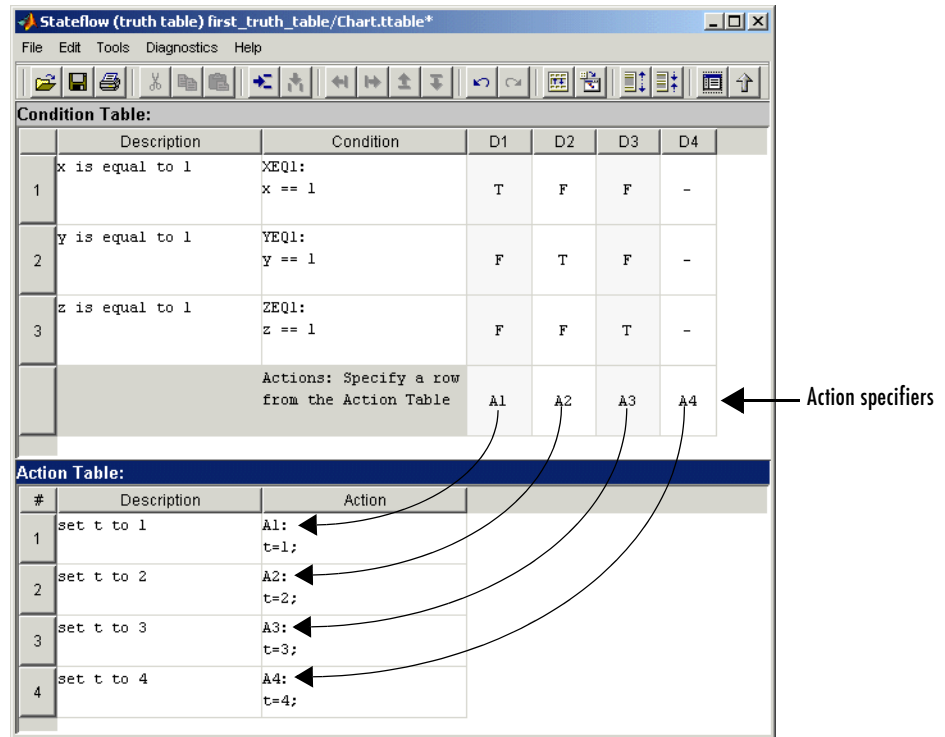
Action Table:

#	Description	Action
1	set t to 1	A1: t=1;
2	set t to 2	A2: t=2;
3	set t to 3	A3: t=3;
4	set t to 4	A4: t=4;

Assigning Truth Table Actions to Decisions

You must assign at least one action from the **Action Table** to each decision in the **Condition Table**. In “Entering Truth Table Actions” on page 10-26, you enter actions in the rows of the **Action Table**. In this topic, you learn how to link the actions you enter in the **Action Table** to the decisions you enter in the **Condition Table** using the **Actions** row in the **Condition Table**.

- 1 Click the bottom cell in decision column **D1**, the first cell of the **Actions** row of the **Condition Table**.
- 2 Enter the action specifier **A1** for decision column **D1**, that links the action labeled **A1** in the **Action Table** to decision **D1**.
- 3 Enter the action specifiers for the remaining decision columns as shown in the following:



In this example, the **Actions** row cell for each decision column contains a label specified for each action in the **Action Table**. Decision **D1** is assigned the action $t=1$. Decision **D2** is assigned the action $t=2$, and so on.

When the truth table executes, the decisions are tested in left to right column order. If a decision is tested as true, its action is immediately executed and the truth table is exited.

You can also specify the action for a decision with the row number for the appropriate action in the **Action Table**. For example, you can enter 1 in place of A1 for decision column **D1** to specify the same action. These and other options are described with examples in the section “Options for Assigning Actions to Decisions in Truth Tables” on page 10-43. For now, continue with the final step in programming a truth table, “Adding Initial and Final Actions” on page 10-31.

Adding Initial and Final Actions

In addition to the actions for decisions, Stateflow also lets you add initial and final actions to the truth table function. Initial actions specify an action that executes before any decisions are tested. Final actions specify an action that executes as the last action before the truth table is exited. To specify initial and final actions for a truth table, use the action labels `INIT` and `FINAL` in the **Action Table**.

Use the following procedure to add initial and final actions to display diagnostic messages in the MATLAB Command Window before and after the execution of the truth table `ttable`:

- 1** In the truth table editor for the truth table `ttable`, right-click row 1 of the **Action Table**.

A pop-up menu appears.

- 2** From the pop-up menu, select **Insert Row**.

A blank row is inserted at the beginning of the **Action Table**.

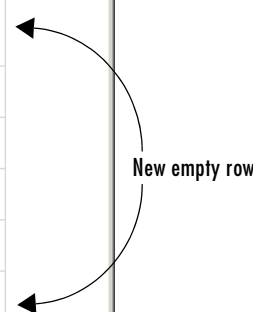
- 3** Click the Append Row tool .

A blank row is appended to the bottom of the **Action Table**.

- 4** Click and drag the bottom border of the truth table editor to expose all six rows of the **Action Table**, as shown.

Condition Table:						
	Description	Condition	D1	D2	D3	D4
1	x is equal to 1	x == 1	T	F	F	-
2	y is equal to 1	y == 1	F	T	F	-
3	z is equal to 1	z == 1	F	F	T	-
	Actions: Specify a row from the Action Table		A1	A2	A3	A4

Action Table:		
#	Description	Action
1		
2	set t to 1	A1: t=1;
3	set t to 2	A2: t=2;
4	set t to 3	A3: t=3;
5	set t to 4	A4: t=4;
6		



- 5** Edit the contents of rows 1 and 6 of the **Action Table**, as shown in the following:

Condition Table:						
	Description	Condition	D1	D2	D3	D4
1	x is equal to 1	x == 1	T	F	F	-
2	y is equal to 1	y == 1	F	T	F	-
3	z is equal to 1	z == 1	F	F	T	-
	Actions: Specify a row from the Action Table		A1	A2	A3	A4

Action Table:		
#	Description	Action
1	Initial action: Display message	INIT: ml.disp('truth table ttable entered'); ← Initial action
2	set t to 1	A1: t=1;
3	set t to 2	A2: t=2;
4	set t to 3	A3: t=3;
5	set t to 4	A4: t=4;
6	Final action: Display message	FINAL: ml.disp('truth table ttable exited'); ← Final action

In the **Action Table**, click and drag the right border of the **Action** column to increase its size as needed.

Even though the initial and final actions for the preceding truth table example are shown in the first and last rows of the **Action Table**, you can enter these actions in any row. You can also explicitly assign the initial and final actions to decisions by using the action specifier INIT or FINAL in the **Actions** row of the **Condition Table**.


Debugging a Truth Table

In “Programming a Truth Table” on page 10-19 you finish a Simulink model with a truth table by programming the truth table. Now you need to begin the process of debugging the truth table. Use the following topics to debug the truth table `ttable` in its Stateflow diagram.

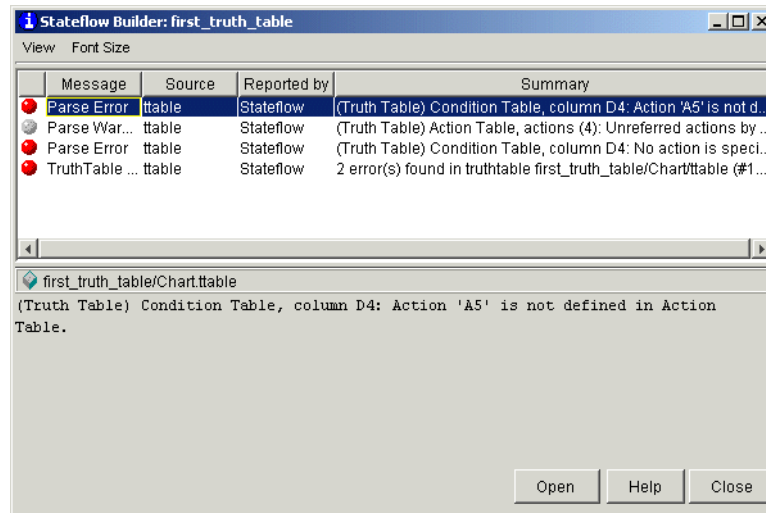
- 1 “Checking Truth Tables for Errors” on page 10-34 — Shows you how Stateflow runs its own diagnostics on truth tables to check for syntax errors.
- 2 “Specifying a Breakpoint for the Call to a Truth Table” on page 10-35 — Shows you how to access the properties dialog for a truth table and set a breakpoint to occur when the truth table is called.
- 3 “Debugging a Truth Table During Simulation” on page 10-36 — Shows you how to debug each condition and decision of a truth table during simulation.

Checking Truth Tables for Errors

Once you completely specify your truth tables you need to begin the process of debugging them. Stateflow runs its own diagnostics to check truth tables for syntax errors with the following procedure:

- 1 In the truth table editor toolbar, select the Run Diagnostics tool  .

If there are no errors or warnings, the **Builder** window appears and reports a message of success. If errors are found, the **Builder** window lists them. For example, if you change the action specifier for decision column **D4** to **A5** (specifier does not exist) instead of **A4**, the following **Builder** window appears:



Each detected error begins with a red button. Warnings begin with a gray button. The first error message is highlighted in the top pane and the diagnostic message is displayed in the bottom pane. Click another error or warning message to see its content displayed in the bottom pane.

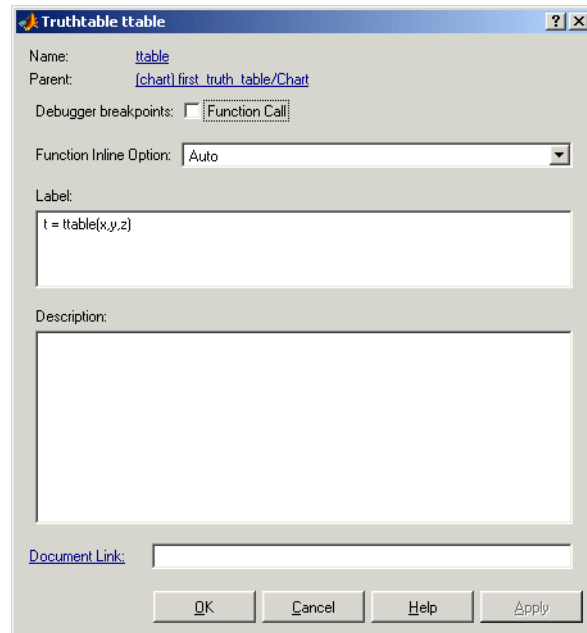
Truth table diagnostics are automatically run when you start simulation of the model with a new or modified truth table. If no errors are found, the **Builder** window does not appear and simulation commences immediately.

Specifying a Breakpoint for the Call to a Truth Table

Before you debug the truth table during simulation, you must set a breakpoint for the truth table in its properties dialog as follows:

- 1 In the Stateflow diagram editor, right-click the truth table.
- 2 In the resulting pop-up shortcut menu, select **Properties**.

The **TruthTable** dialog appears, as shown.



3 For Debugger breakpoints, select Function Call.


This sets a breakpoint to occur when this truth table function is called in the Stateflow diagram during simulation.

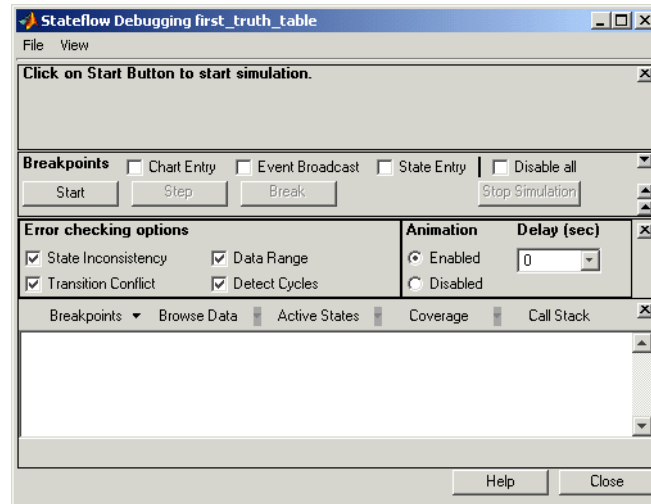
4 Select OK to save settings and close the Truthtable dialog.

Debugging a Truth Table During Simulation

The last step in developing the truth table model is to test it during a simulated run. In “Specifying a Breakpoint for the Call to a Truth Table” on page 10-35, you set a function call breakpoint for the truth table `ttable`. This pauses execution during simulation so that you can debug each execution step of a truth table using the Stateflow Debugger.

Simulate and debug the individual operation steps of a truth table with the following steps:

- 1 Select the Debug button  in the Stateflow diagram editor toolbar to start the **Stateflow Debugging** window as shown.



- 2 From the **Stateflow Debugging** window, select the **Start** button to begin simulation of your model.

When you simulate your model, Stateflow checks the truth tables automatically for syntactical errors if they have changed since the last simulation. If you receive errors or warnings, make corrections before you try to simulate again.

If Stateflow finds no syntactical errors in the truth table, Stateflow builds a simulation application and begins the simulation of your model.

- 3 Wait until the breakpoint for the call to the truth table is reached.

When this breakpoint is encountered, the truth table `ttable` appears and the **Start** button in the Stateflow Debugger changes to the **Continue** button.

- 4 In the **Stateflow Debugging** window, click the **Step** button three times to advance simulation through the call to the truth table.

The `INIT` action of the truth table is highlighted prior to its execution.

Action Table:		
#	Description	Action
1	Initial action: Display message	INIT: ml.disp('truth table ttable entered');
2	set t to 1	A1: t=1;
3	set t to 2	A2: t=2;
4	set t to 3	A3: t=3;
5	set t to 4	A4: t=4;
6	Final action: Display message	FINAL: ml.disp('truth table ttable exited');

- 5 Click **Step** to execute the INIT action and advance truth table execution to the first condition, as shown.

Condition Table:						
	Description	Condition	D1	D2	D3	D4
1	x is equal to 1	x == 1	T	F	F	-
2	y is equal to 1	y == 1	F	T	F	-
3	z is equal to 1	z == 1	F	F	T	-

- 6 Click **Step** to evaluate the first condition and advance truth table execution to the second condition.

Condition Table:						
	Description	Condition	D1	D2	D3	D4
1	x is equal to 1	x == 1	T	F	F	-
2	y is equal to 1	y == 1	F	T	F	-
3	z is equal to 1	z == 1	F	F	T	-

- 7 Click **Step** to evaluate the second condition and advance truth table execution to the third condition.

Condition Table:						
	Description	Condition	D1	D2	D3	D4
1	x is equal to 1	x == 1	T	F	F	-
2	y is equal to 1	y == 1	F	T	F	-
3	z is equal to 1	z == 1	F	F	T	-

- 8 Click **Step** to evaluate the third condition and advance truth table execution to the first decision.

Condition Table:						
	Description	Condition	D1	D2	D3	D4
1	x is equal to 1	x == 1	T	F	F	-
2	y is equal to 1	y == 1	F	T	F	-
3	z is equal to 1	z == 1	F	F	T	-
	Actions: Specify a row from the Action Table		A1	A2	A3	A4

- 9 Click **Step** twice.

Because the first decision is true, truth table execution advances to its action, which is labeled A1.

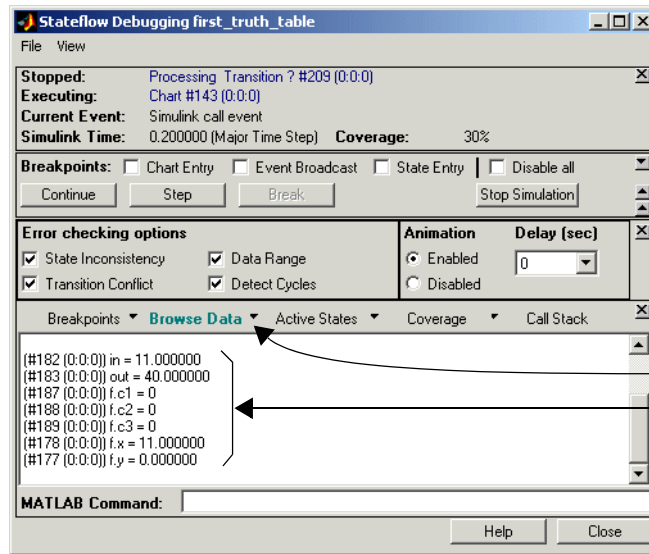
Action Table:		
#	Description	Action
1	Initial action: Display message	INIT: ml disp('truth table ttable entered');
2	set t to 1	A1: t=1;
3	set t to 2	A2: t=2;
4	set t to 3	A3: t=3;
5	set t to 4	A4: t=4;
6	Final action: Display message	FINAL: ml disp('truth table ttable exited');

10 Click **Step** three times to execute action A1 and advance to the FINAL action.

Action Table:		
#	Description	Action
1	Initial action: Display message	INIT: ml disp('truth table ttable entered');
2	set t to 1	A1: t=1;
3	set t to 2	A2: t=2;
4	set t to 3	A3: t=3;
5	set t to 4	A4: t=4;
6	Final action: Display message	FINAL: ml disp('truth table ttable exited');

11 In the **Stateflow Debugging** window, from the **Browse Data** pull-down, select **All Data (Current Chart)**.

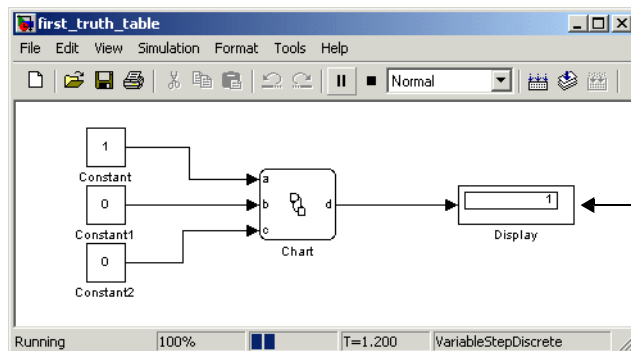
A continuously updated display appears in the bottom pane of the **Stateflow Debugging** window, as shown.



You can use this display to monitor Stateflow data during simulation.

12 In the **Stateflow Debugging** window, click **Step**.

This executes the final action and exits the truth table. The Display block in the Simulink window should now display the number 1, as shown.



- 13 Change the values of the Constant blocks and continue stepping through the simulation.

For example, you might want to set the Constant1 block to 1 (sets b to 1) and the other Constant blocks to 0 (sets a and c to 0). Or you might want to set all the Constant blocks to 0. To enter a new value for a Constant block, double-click it. In the resulting **Block Parameters** dialog enter the new value in the **Constant value** field.

Options for Assigning Actions to Decisions in Truth Tables

In “Assigning Truth Table Actions to Decisions” on page 10-29, you assign actions to decisions in a truth table as part of programming a truth table. Stateflow allows you to be very creative in assigning actions. The following is a list of rules that apply to assigning actions to decisions in a truth table.

- You specify actions for decisions by entering a row number or a label in the **Actions** row cell of a decision column.

If you use a label specifier, the label must be entered with the action in the **Action Table**.

- You must specify at least one action for each decision.

Actions for decisions are not optional. Each decision must have at least one action specifier that points to an action in the **Action Table**. If you want to specify no action for a decision, specify a row that contains no action statements.

- You can specify multiple actions for a decision with multiple specifiers separated by a comma.

For example, for the decision column **D1** you can specify A1, A2, A3 or 1, 2, 3 to execute the first three actions if decision **D1** is true.

- You can mix row number and label action specifiers interchangeably in any order.

The following example uses both row and label action specifiers.

Condition Table:						
	Description	Condition	D1	D2	D3	D4
1	x is equal to 1	x == 1	T	F	F	-
2	y is equal to 1	y == 1	F	T	F	-
3	z is equal to 1	z == 1	F	F	T	-
	Actions: Specify a row from the Action Table		A1	2	A3	4

Action Table:		
#	Description	Action
1	set t to 1	A1: t=1;
2	set t to 2	A2: t=2;
3	set t to 3	A3: t=3;
4	set t to 4	A4: t=4;

- You can specify the same action for more than one decision, as shown.

Condition Table:						
	Description	Condition	D1	D2	D3	D4
1	x is equal to 1	x == 1	T	F	F	-
2	y is equal to 1	y == 1	F	T	F	-
3	z is equal to 1	z == 1	F	F	T	-
	Actions: Specify a row from the Action Table		A1	1	A2	2


Action Table:		
#	Description	Action
1	set t to 1	A1: t=1;
2	set t to 2	A2: t=2;

- Row number action specifiers in the **Actions** row of the **Condition Table** automatically adjust to changes in the row order of the **Actions Table**.

In the following example, decisions **D3** and **D4** are assigned the actions in rows 3 and 4 of the **Action Table**, respectively.

Condition Table:						
	Description	Condition	D1	D2	D3	D4
1	x is equal to 1	x == 1	T	F	F	-
2	y is equal to 1	y == 1	F	T	F	-
3	z is equal to 1	z == 1	F	F	T	-
	Actions: Specify a row from the Action Table		1	2	3	4

Action Table:		
#	Description	Action
1	set t to 1	A1: t=1;
2	set t to 2	A2: t=2;
3	set t to 3	A3: t=3;
4	set t to 4	A4: t=4;

Select row 4 in the **Action Table** and select the Move Row Up tool  to reverse rows 3 and 4, and notice the change in the action specifiers for columns **D3** and **D4**, as shown.

Condition Table:						
	Description	Condition	D1	D2	D3	D4
1	x is equal to 1	x == 1	T	F	F	-
2	y is equal to 1	y == 1	F	T	F	-
3	z is equal to 1	z == 1	F	F	T	-
	Actions: Specify a row from the Action Table		1	2	4	3

Action Table:		
#	Description	Action
1	set t to 1	A1: t=1;
2	set t to 2	A2: t=2;
3	set t to 4	A4: t=4;
4	set t to 3	A3: t=3;

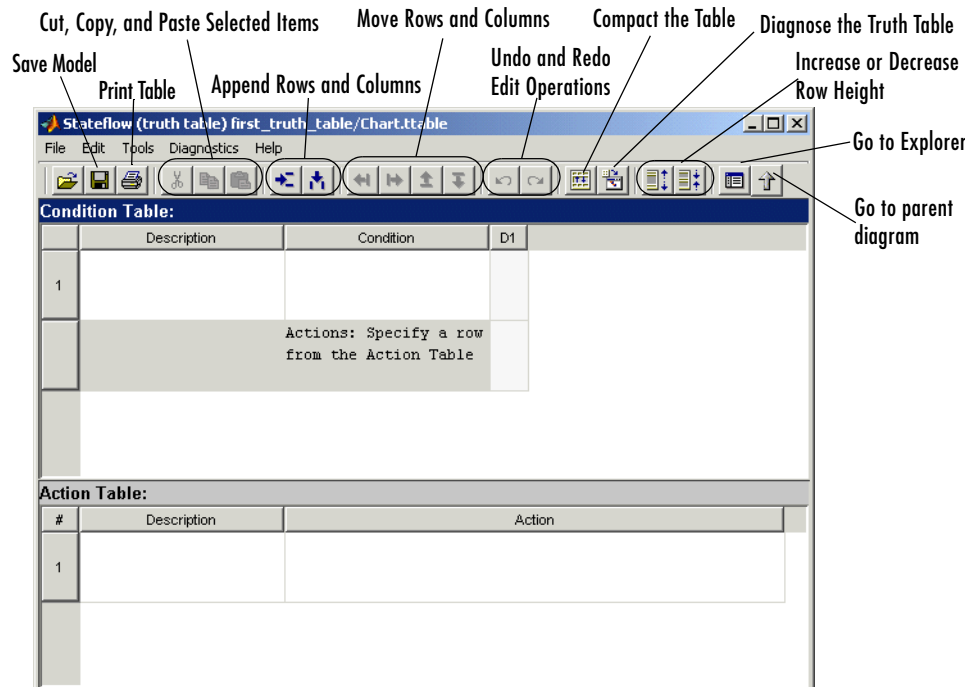
Truth Table Editor Operations

This section describes the edit operations that you can perform in the truth table editor. Use the following topics to learn operations that you can do in the truth table editor:

- “Truth Table Editor Reference” on page 10-47 — Identifies operations that you can perform in a truth table editor with tables and toolbar icon tables.
- “Searching and Replacing in Truth Tables” on page 10-52 — Shows you how the Stateflow Search & Replace tools work with Stateflow truth tables.
- “Using Row and Column Tooltip Identifiers” on page 10-54 — Shows you how row and column tooltips can help you navigate in big truth tables.

Truth Table Editor Reference

This topic is a reference of the parts of a truth table and the edit operations that you can perform in the truth table editor as shown.



The major editing operations in the truth table editor are described in the topics that follow.

Adding or Modifying Stateflow Data



Select the Goto Explorer tool to add or modify Stateflow data with the **Model Explorer** tool.

Appending Rows and Columns



Click the Append Column tool to append a column on the right end of the selected table.



Click the Append Row tool to append a row to the bottom of the selected table.

Compacting the Table



Select the Compact Table tool to remove the empty rows and columns of the selected table.

Copy, Cut, and Paste Table Element.



Click the Copy tool to copy the selected row, column, or text.



Click the Cut tool to cut the selected row, column, or text.



Click the Paste tool to paste the cut or copied element as follows:

- Cut or copied text is inserted at the cursor location.
- If you select a decision column before pasting a cut or copied decision column, the pasted decision column is inserted to the left of the selected column.
- If you select a row before pasting a cut or copied row, the pasted row is inserted above the selected row.

Deleting Text, Rows, and Columns

To delete the contents of a cell, do the following:

- 1 Right-click the cell.
- 2 From the resulting pop-up menu, select **Delete Cell**.

To delete an entire row or column, do the following:

- 1 Right-click the row or column header.
- 2 From the resulting pop-up menu, select **Delete Row** or **Delete Column**.

You can also click the row or column header to select the entire row or column and press the **Delete** key.

Diagnosing the Truth Table



Select the Run Diagnostics tool to check the truth table for syntax errors.

Editing Tables

Both the default **Condition Table** and the default **Action Table** have one empty row. Click a cell to edit its text contents. Use **Tab** and **Shift+Tab** to move horizontally between cells. To add rows and columns to either table, see “Appending Rows and Columns” on page 10-48.

You set the Truth Table editor to display only one of the two tables by double-clicking the header of the table to display. To revert to the display of both tables, double-click the header of the displayed table.

Cells for the numbered rows in decision columns like **D1** can take values of T, F, or -. Once you select one of these cells, you can use the spacebar to step through the T, F, and - values. In these cells you can use the left, right, up, and down arrow keys to advance to another cell in any direction.

Increasing and Decreasing Row Height



Select the Increase Row Height tool to increase the row height of the selected row.



Select the Decrease Row Height tool to decrease the row height of the selected row.

Inserting Rows and Columns

To insert a blank row above an existing table row,

- 1 Right-click any cell in the row (including the row header).
- 2 From the resulting pop-up menu, select **Insert Row**.

To insert a blank decision column to the left of an existing decision column,

- 1 Right-click any cell in the existing decision column (including the column header).
- 2 From the resulting pop-up menu, select **Insert Column**.

Moving Rows and Columns



Click the Move Up tool to move a selected row up one row.



Click the Move Down tool to move a selected row down one row.



Click the Move Right tool to move a selected column right one column.



Click the Move Left tool to move a selected column left one column.

Printing Tables



Select the Print tool to make a printed copy or an online viewable copy (HTML file) of the truth table.

Selecting and Deselecting Table Elements

- To select a cell for editing, click the cell.
- To select text in a cell, click and drag the text as usual.
- To select a row, click the header for the row.
- To select a decision column in the **Condition Table**, click the column header (**D1**, **D2**, and so on).
- To deselect a selected cell, row, or column, press **Esc**, or click another table, cell, row, or column.

Undoing and Redoing Edit Operations



Select the Undo tool or press **Ctrl+Z** to reverse the effects of the preceding operation.



Select the Redo tool or press **Ctrl+Y** to reverse the effects of the most recently undone edit operation.

Viewing the Stateflow Diagram for the Truth Table



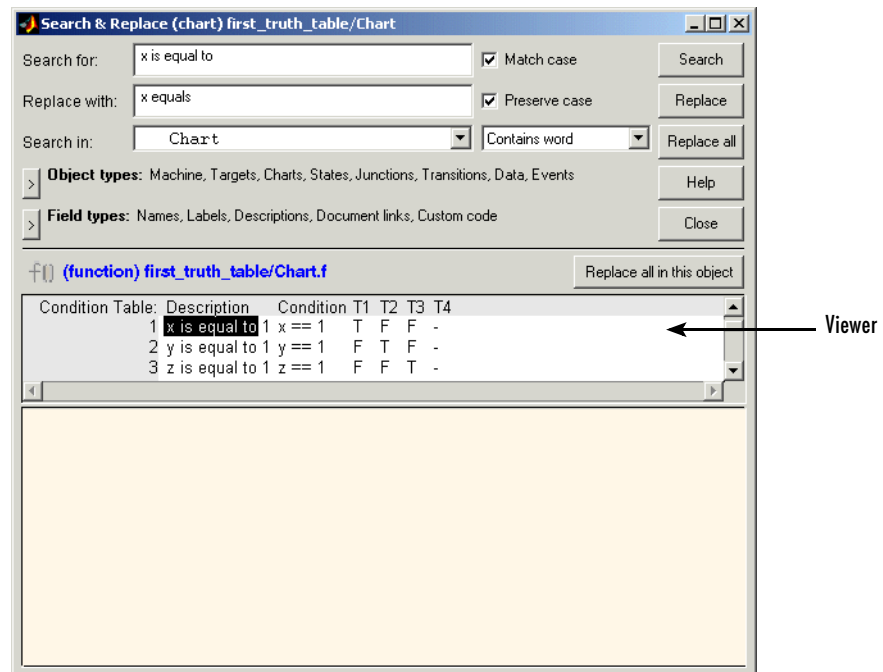
Select the Go to Diagram tool to display the current truth table function in its native Stateflow diagram.

Searching and Replacing in Truth Tables

You can use the Search & Replace tool in Stateflow to search for text in the **Description**, **Condition**, and **Action** columns of a truth table and replace it with a substitute string. For example, you can search the model for the string `x is equal to` and replace it with the string `x equals` with the following procedure:

- 1 In the Stateflow diagram editor, select **Search & Replace** from the **Tools** menu.
- 2 In the resulting **Search & Replace** window, enter the text `x is equal to` in the **Search** field, and the text `x equals` in the **Replace** field.
- 3 Select the **Search** button.

You see something like the following in the **Search & Replace** window:



Notice that in the Viewer pane of the **Search & Replace** window the first occurrence of the string `x is equal to` is highlighted normally and the remaining matches are highlighted lightly.

- 4 Select **Replace** to replace the first match with `x equals`.
- 5 Select **Replace All** to replace all matches in the model (not just in the truth table) with `x equals`.

Note The Search & Replace tool is fully described in “Using the Stateflow Search & Replace Tool” on page 14-12.

Using Row and Column Tooltip Identifiers

Stateflow gives you row and column header tooltips to aid truth table navigation when you are scrolling to other columns or rows when you are editing a large truth table. When you place your mouse cursor over row or column headers, the following tooltips appear:

Table	Row or Column	Tooltip
Condition	Condition row	Condition entered for this row
Condition	Decision column (D1 , D2 ,...)	Row or label entered for this decision in the Actions row
Condition	Actions row	Actions: specify a row from the Action Table
Action	Any row	Description entered for this action

Correcting Over- and Underspecified Truth Tables

An *overspecified truth table* contains a decision that will never be executed because it is already specified in a previous decision in the **Condition Table**. An *underspecified truth table* lacks one or more possible decisions that might require an action to avoid undefined behavior in the application. Stateflow helps you correct over- and underspecified truth tables with the special error messages described in the following topics:

- “Defining an Overspecified Truth Table” on page 10-55 — Gives you an example of an overspecified truth table and a truth table that is not overspecified.
- “Defining an Underspecified Truth Table” on page 10-56 — Gives you an example of an underspecified truth table and a truth table that is not underspecified.

Defining an Overspecified Truth Table

An overspecified truth table contains at least one decision that will never be executed because it is already specified in a previous decision in the **Condition Table**. The following is the **Condition Table** of an overspecified truth table:

Condition Table:					
	Description	Condition	D1	D2	D3
1	Condition C1	C1: x == 0	F	T	-
2	Condition C2	C2: y == 0	T	-	T
3	Condition C3	C3: z == 0	T	T	T
	Actions: Specify a row from the Action Table		A1	A2	A3

The decision in column **D3** (-TT) specifies the decisions FTT and TTT. These decisions have already been specified by decisions **D1** (FTT) and **D2** (TTT and TFT). Therefore column **D3** is an overspecification.

The following is the **Condition Table** of a truth table that appears overspecified but is not:

Condition Table:						
	Description	Condition	D1	D2	D3	D4
1	Condition C1	C1: x == 0	F	T	T	-
2	Condition C2	C2: y == 0	T	F	T	T
3	Condition C3	C3: z == 0	T	T	F	T
	Actions: Specify a row from the Action Table		A1	A2	A3	A4

In this case, the decision **D4** specifies two decisions (TTT and FTT). The second of these is specified by decision **D1**. However, the first is not specified in a previous decision column. Therefore, this **Condition Table** is not overspecified.

Defining an Underspecified Truth Table

An underspecified truth table lacks one or more possible decisions that might require an action to avoid undefined behavior. The following is the **Condition Table** of an underspecified truth table:

Condition Table:					
	Description	Condition	D1	D2	D3
1	Condition C1	C1: x == 0	T	T	F
2	Condition C2	C2: y == 0	T	F	T
3	Condition C3	C3: z == 0	F	T	T
	Actions: Specify a row from the Action Table		A1	A2	A3

Complete coverage of the conditions in the preceding truth table requires a **Condition Table** with every possible decision, like the following example:

Condition Table:											
	Description	Condition	D1	D2	D3	D4	D5	D6	D7	D8	
1	Condition C1	C1: x == 0	T	T	T	F	F	T	F	F	
2	Condition C2	C2: y == 0	T	T	F	T	F	F	T	F	
3	Condition C3	C3: z == 0	T	F	T	T	F	F	F	T	
	Actions: Specify a row from the Action Table		A1	A2	A3	A4	A5	A6	A7	A8	

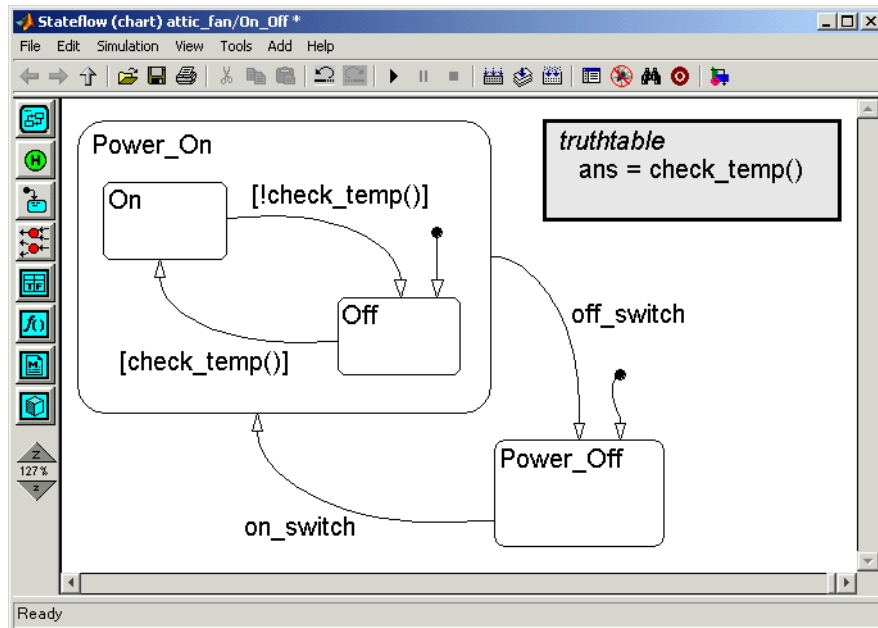
A possible workaround is to specify an action for all other possible decisions through a default decision, as in the following example:

Condition Table:						
	Description	Condition	D1	D2	D3	D4
1	Condition C1	C1: x == 0	T	T	T	-
2	Condition C2	C2: y == 0	T	T	F	-
3	Condition C3	C3: z == 0	T	F	T	-
	Actions: Specify a row from the Action Table		A1	A2	A3	DA

The last decision column is the default decision for the truth table. The default decision covers any remaining decisions not tested for in the preceding decision columns. See “The Default Decision Column” on page 10-25 for an example and more complete description of the default decision column for a **Condition Table**.

Model Coverage for Truth Tables

Stateflow reports model coverages for the decisions made by the objects in a Stateflow diagram during model simulation. The model coverage report includes coverage for the decisions made by a truth table function. This section examines model coverage for an example truth table, `check_temp`, which is tested during simulation in the following Stateflow diagram:



The following shows the contents of the `check_temp` truth table:

Condition Table:				
	Description	Condition	D1	D2
1	Check ambient temp for cooling time	CAMTSC: amb_temp > min_amb_temp	T	-
2	Check attic temp setting for cooling	attic_temp > attic_temp_set	T	-
3	Check ambient temp for cooling capable	amb_temp < attic_temp - 5	T	-
	Actions: Specify a row from the Action Table		ON	OFF

Action Table:		
#	Description	Action
1	Stay on or turn on	ON: ans = 1;
2	Stay off or turn off	OFF: ans = 0;

You generate model coverage reports for a model during simulation. You first specify the creation of the reports in Simulink and then simulate the model. When simulation ends, a model coverage report appears in a browser window. See “Making Model Coverage Reports” on page 13-43 for information on how to set up a model coverage report.

Note The Model Coverage tool requires a Simulink Verification and Validation license.

The following is the part of a model coverage report that reports on the check_temp truth table:

4. Function "check_temp"

Parent: [attic_fan/On_Off](#)

Metric	Coverage (this object)	Coverage (inc. descendants)
Cyclomatic Complexity	0	3
Decision (D1)	NA	100% (2/2) decision outcomes
Condition (C1)	NA	83% (5/6) condition outcomes
MCDC (C1)	NA	67% (2/3) conditions reversed the outcome

Predicate table analysis (missing values are in parentheses)

Check ambient temp for cooling time	CAMTSC: amb_temp > min_amb_temp	T (ok)	-
Check attic temp setting for cooling	attic_temp > attic_temp_set	T (ok)	-
Check ambient temp for cooling capable	amb_temp < attic_temp - 5	T (F)	-
	Actions	ON (ok)	OFF

← red F

Coverage for the truth table function in the **Coverage (this object)** column shows no valid coverage values. The reason for this is that the container object for the truth table function, the chart, makes no decision on whether to execute the check_temp truth table or not.

Stateflow implements a truth table by generating a graphical function for it. The decision logic of the truth table is implemented internally in the transitions of the graphical function generated for the truth table. See “How Stateflow Realizes Truth Tables” on page 10-62 for a description of the generated graphical function for a truth table.

The transitions of the generated graphical function for a truth table contain the decisions and conditions of the truth table. Coverage for the descendants in the **Coverage (inc. descendants)** column includes coverage for these conditions and decisions, which are tested when the truth table function is called.

In the case of the check_temp truth table, the only decision covered in the model coverage report is the **D1** decision. There is no model coverage for the default decision, **D2**.

Note All logic that leads to taking a default decision is based on a false outcome for all preceding decisions. This means that no logic is required for the default decision, which receives no model coverage.

Coverages for the **D1** decision and its individual conditions in the `check_temp` truth table function are as follows:

- Decision coverage for the **D1** decision is 100% because this decision was tested both true and false during simulation.
- Condition coverage for the three conditions of the **D1** decision indicate that 5 of 6 possible T/F values were tested.
Because each condition can have an outcome value of T or F, three conditions can have 6 possible values. During simulation, only 5 of 6 were tested. The **D1** decision coverage column shows that the last condition received partial condition coverage by not evaluating to false (F) during simulation. The missing occurrence of the false (F) condition outcome is indicated by the appearance of a red F character.
- MCDC coverage looks for decision reversals that occur because one condition outcome changes from T to F or from F to T. The **D1** decision reverses when any of the conditions changes from T to F. This means that the outcomes FTT, TFT, and TTF reverse this decision by a change in the value of one condition. The top two conditions for the D1 decision tested both true (T) and false (F) with a resulting reversal in the decision from true (T) to false (F). However, the bottom condition tested only a true (T) outcome but no false (F) outcome (appearance of red F character). Therefore, two of a possible three reversals were observed and coverage is $2/3 = 67\%$.
- The (ok) next to the ON action label indicates that its decision realized both true (T) and false (F) during simulation. Because the default decision is based on no logic of its own, it does not receive the (ok) mark.

How Stateflow Realizes Truth Tables

Stateflow realizes the logical behavior specified in a truth table by generating a graphical function for it. See the following sections if it is important for you to know the mechanics of truth table generation in Stateflow:

- “Viewing the Graphical Function for a Truth Table” on page 10-62 — Tells you when Stateflow generates a graphical function for a truth table and how to view the graphical function.
- “How Stateflow Generates Graphical Functions for Truth Tables” on page 10-62 — Takes a particular example truth table and describes the structure of the resulting graphical function.

Viewing the Graphical Function for a Truth Table

Stateflow generates a graphical function for a truth table when you simulate your model. If a truth table changes, its graphical function is regenerated when you simulate your model. If a truth table remains unchanged, simulating does not generate a new graphical function for it.

After you simulate the model, you can see the generated graphical function for a truth table as follows:

- 1 Right-click the truth table box in its Stateflow diagram.
- 2 From the resulting pop-up menu, select **View Contents**.

The truth table graphical function appears with a shaded background to indicate that it is not editable. See “How Stateflow Generates Graphical Functions for Truth Tables” on page 10-62 for an example of a generated graphical function for a truth table.

How Stateflow Generates Graphical Functions for Truth Tables

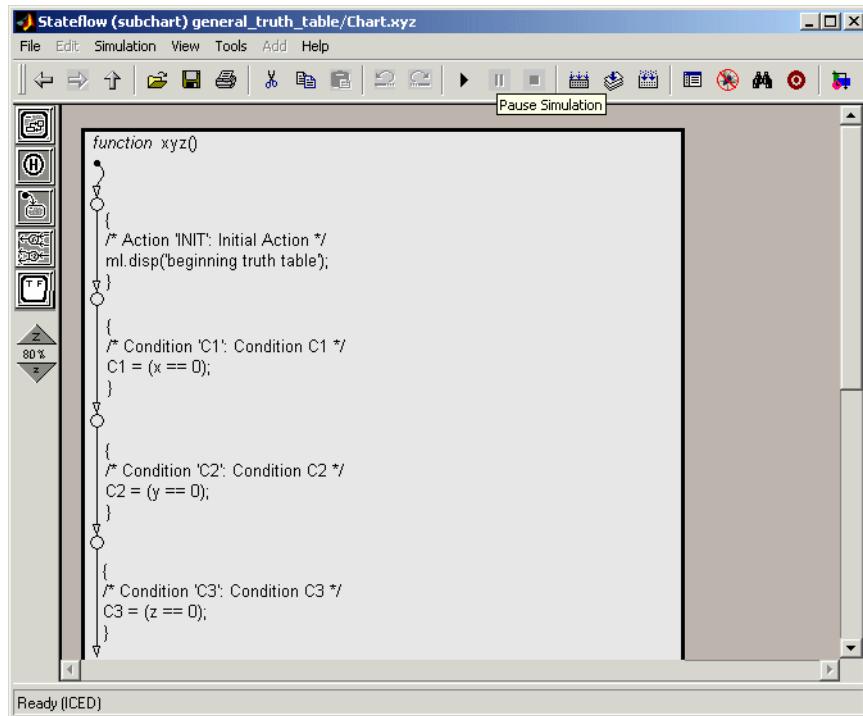
This topic shows you through an example how the logic of a truth table translates into a graphical function. See “Viewing the Graphical Function for a Truth Table” on page 10-62 to find out how to access the generated graphical function.

The following example truth table has three conditions, four decisions and actions, and initial and final actions:

Condition Table:						
	Description	Condition	D1	D2	D3	D4
1	Condition C1	C1: x == 0	T	F	F	
2	Condition C2	C2: y == 0	-	T	F	
3	Condition C3	C3: z == 0	-	-	T	
	Enter Row number for associated action	Action number:	A1	A2	A3	DA

Action Table:		
#	Description	Action
1	Initial Action	INIT: ml_disp('beginning truth table');
2	Action 1	A1: x = 1;
3	Action 2	A2: y = 1;
4	Action 3	A3: z = 1;
5	Default Action	DA: x = 0; y = 0; z = 0;
6	Final Action	FINAL: ml_disp('ending truth table');

Stateflow generates the following graphical function for the preceding truth table:

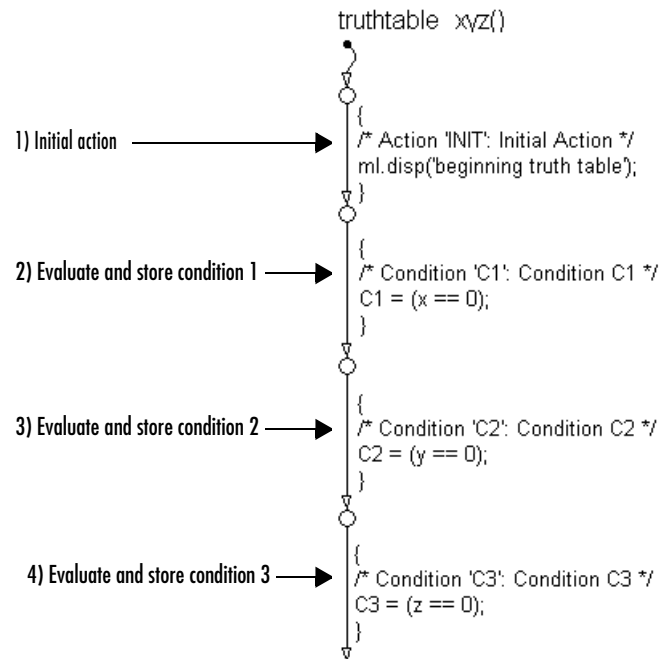


Stateflow uses the top half of the generated function to do the following:

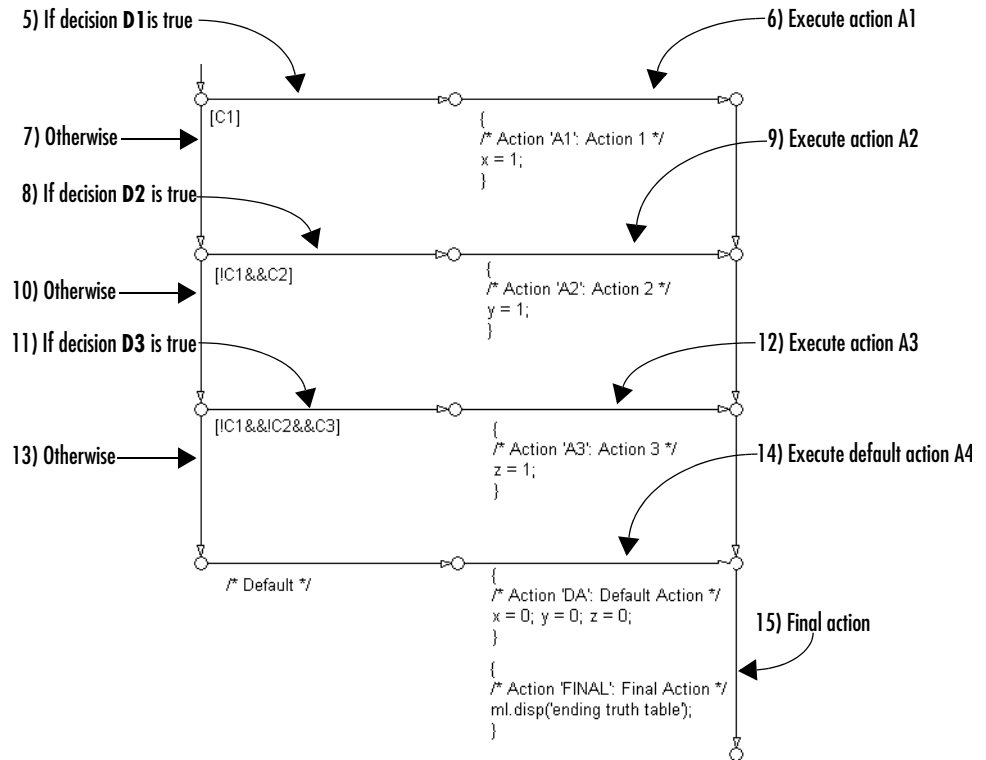
- Perform initial actions
- Evaluate the conditions and store the results in temporary data variables.

The temporary data for storing conditions is based on the labels that you enter for the conditions. If no labels are specified, temporary data variables are named by Stateflow.

The following generated flow diagram for the `check_temp` truth table shows this process, which includes number steps to show the order of execution:



The stored values for the conditions are used in the bottom half of the function to make decisions on which action to perform. The following shows the remaining half of the generated function, along with numbered steps for showing the order of consideration for each condition and action:



Each decision is implemented as a fork from a connective junction with one of two possible paths:

- A transition segment with a decision followed by a segment with the consequent action
 The action is specified as a condition action that leads to the FINAL action and termination of the flow diagram
- A transition segment that flows to the next fork for an evaluation of the next decision
 This transition segment has no condition or action.

The preceding implementation continues from the first decision through the remaining decisions in left to right column order. When a specified decision is matched, the action specified for that decision is executed as a condition action of its transition segment. Once the action is performed, the flow diagram performs the final action for the truth table and terminates. This means that only one action results from a call to a truth table graphical function. This also implies that no data dependencies are possible between different decisions.

When you start constructing your own truth tables, you will need to troubleshoot them. When you simulate your model, Stateflow checks your truth tables for errors. Once you finish building a truth table and eliminate all detectable errors from it, you can debug it during simulation.

Stateflow also provides another type of debugging that is useful in making sure that your truth table conditions, decisions, and actions are completely tested. This type of debugging is referred to as *model coverage*. The Model Coverage tool reports on model coverage for Stateflow diagrams and their objects. See “Model Coverage for Truth Tables” on page 10-58 for a description and example of truth table model coverage.

Finally, Stateflow realizes truth tables by generating a graphical function with the logical behavior you specify. To see how Stateflow generates these graphical functions, see “How Stateflow Realizes Truth Tables” on page 10-62.

Using Embedded MATLAB Functions

An Embedded MATLAB function lets you compose a MATLAB language function in Stateflow. In a Stateflow Embedded MATLAB function, you create functions with a rich subset of the MATLAB language that generates efficient C code meeting the strict memory and data type requirements of embedded target environments. In this way, Embedded MATLAB functions bring the power of MATLAB into Stateflow diagrams.

To learn how to use an Embedded MATLAB function in a Stateflow diagram, see the following sections:

Introduction to Embedded MATLAB Functions (p. 11-2)

Introduction to Embedded MATLAB functions through an example that shows you how to use them in Stateflow

Building a Simulink Model with a Stateflow Embedded MATLAB Function (p. 11-5)

Procedure building a Simulink model with a Stateflow diagram that calls an Embedded MATLAB function

Programming a Stateflow Embedded MATLAB Function (p. 11-13)

Procedure for programming an example Embedded MATLAB function that gives you an overview of program syntax, local data, and callable functions

Debugging a Stateflow Embedded MATLAB Function (p. 11-18)

Procedure for debugging an example Embedded MATLAB function with its own debugging tools

Model Coverage for an Embedded MATLAB Function (p. 11-29)

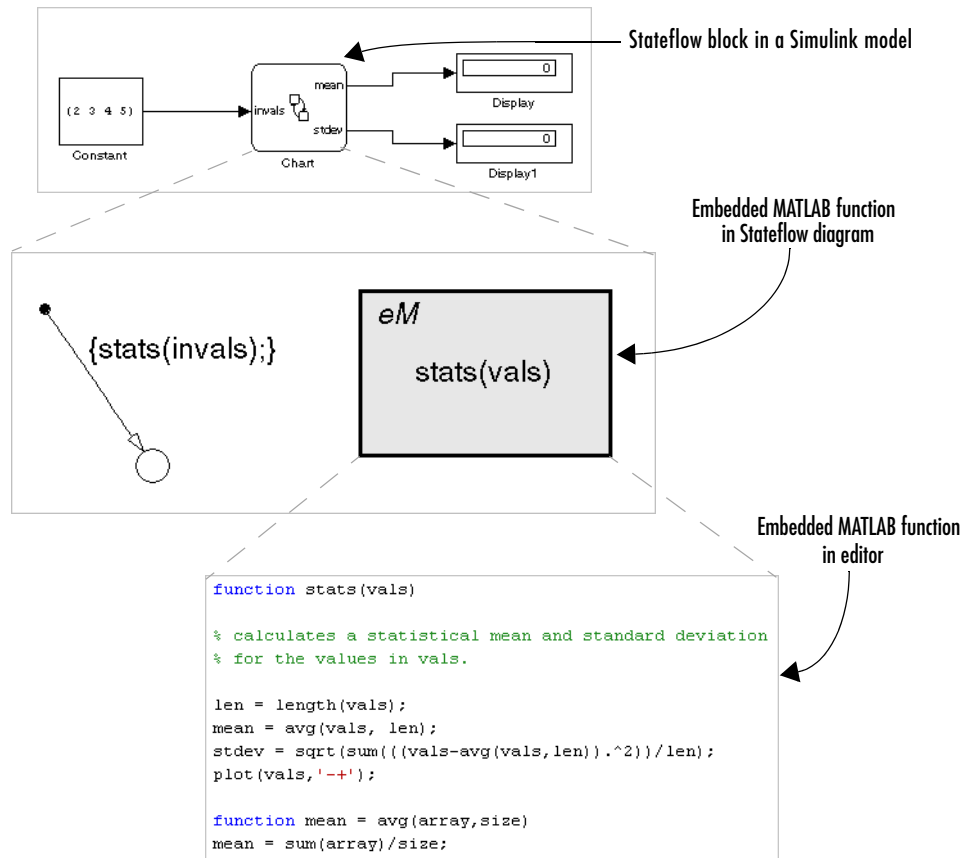
Interpretation of the results of a coverage report that you generate for an example model

Introduction to Embedded MATLAB Functions

A Stateflow Embedded MATLAB function contains an Embedded MATLAB language function. The Embedded MATLAB language is a powerful subset of the MATLAB language that, like the rest of Stateflow, generates efficient C code for embeddable applications.

Simulink uses Stateflow Embedded MATLAB functions to implement Embedded MATLAB blocks in Simulink models. See “Using the Embedded MATLAB Function Block” in Simulink documentation for more information on Embedded MATLAB blocks in Simulink.

You build the following model in “Building a Simulink Model with a Stateflow Embedded MATLAB Function” on page 11-5:



In addition to supporting MATLAB syntax, a Stateflow Embedded MATLAB function can call any of the following functions:

- Subfunctions

Subfunctions are defined in the body of the Embedded MATLAB function. In the preceding example, `avg` is a subfunction.

- Embedded MATLAB run-time library functions

Embedded MATLAB run-time library functions are a subset of the functions that you can call in MATLAB. They generate C code for building targets that conforms to the memory and data type requirements of embedded

environments. In the preceding example, `length`, `sqrt`, and `sum` are examples of Embedded MATLAB run-time library functions.

- **Stateflow functions**

Includes graphical, truth table, and other Embedded MATLAB functions in the scope of the current Embedded MATLAB function.

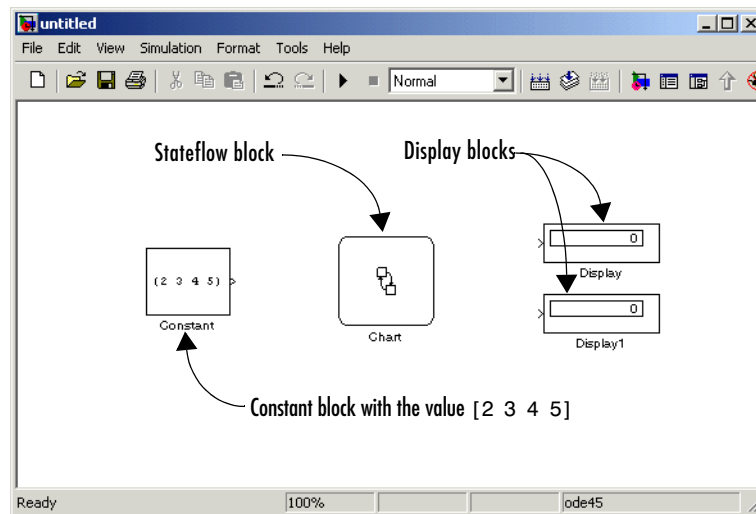
- **Some MATLAB functions**

Function calls that cannot be resolved as subfunctions, Embedded MATLAB run-time library functions, or Stateflow functions are resolved in the MATLAB workspace. These functions do not generate code; they execute only in the MATLAB workspace during simulation of the model.

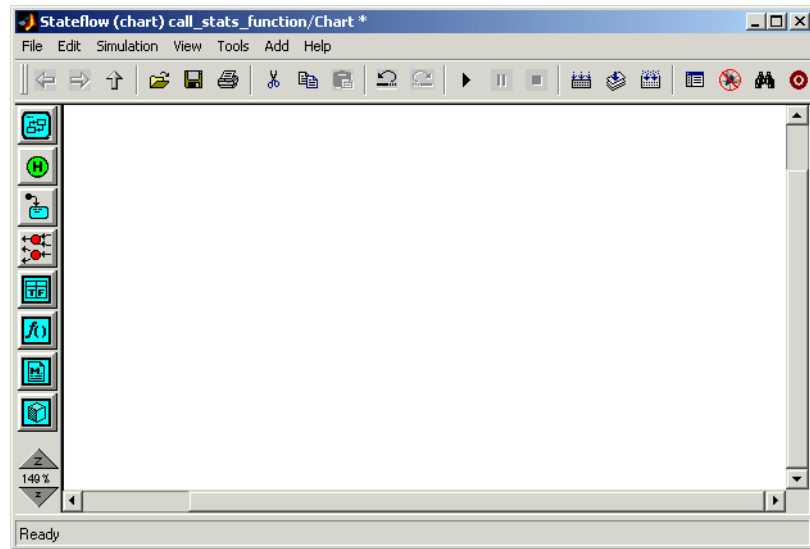
Building a Simulink Model with a Stateflow Embedded MATLAB Function


Embedded MATLAB functions are special Stateflow functions that use the Embedded MATLAB Language, a rich subset of the MATLAB programming language. This section takes you through the steps of creating a Simulink model with a Stateflow block that calls an Embedded MATLAB function, `stats`. `stats` calculates a mean and a standard deviation for the values in `vals` and stores them in the Stateflow data `mean` and `stdev`, respectively. Use the following steps to create a model with a Stateflow diagram that calls an Embedded MATLAB function. In the process, learn how to use embedded MATLAB functions in Stateflow.

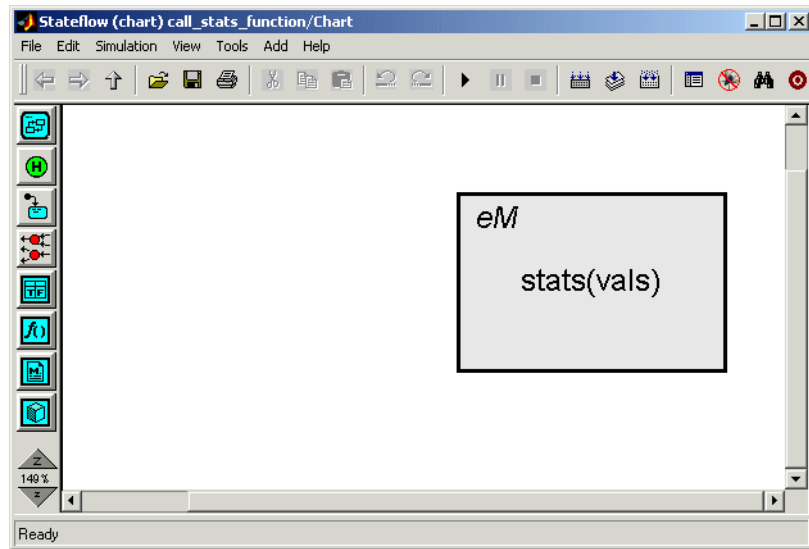
- 1 Create a new Simulink model with the following blocks:



- 2 Save the model as `call_stats_function`.
- 3 In the Simulink model, double-click the Stateflow block to open its empty Stateflow diagram.



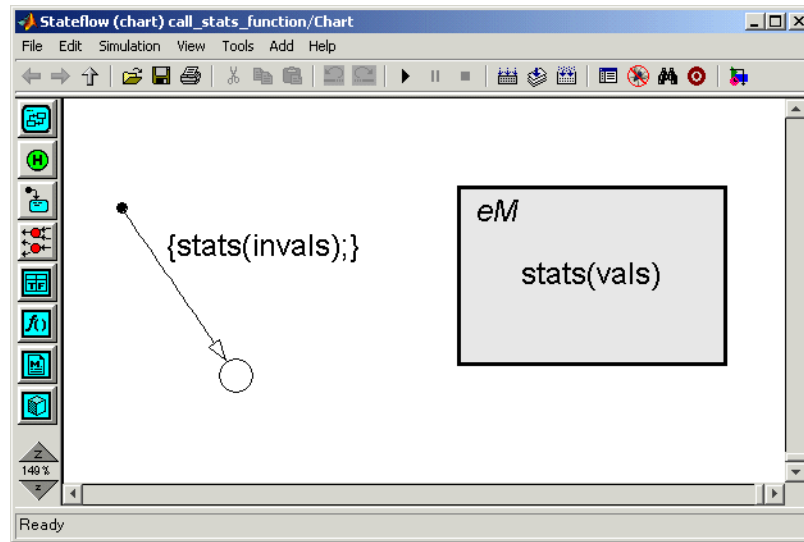
- 4 In the Stateflow diagram editor, select the Embedded MATLAB Function tool  and move the cursor into the empty diagram area to click and place a new Embedded MATLAB function.
- 5 A text field with a flashing cursor appears in the middle of the new Embedded MATLAB function.
- 6 Enter the label `stats(vals)` for the new Embedded MATLAB function, as shown.



- 7 In the Stateflow diagram, draw a default transition into a terminating junction with the following condition action.

```
{stats(invals);}
```


The finished Stateflow diagram now has the following appearance:



Like all Stateflow functions, the signature of an Embedded MATLAB function is entered in its label with the following syntax:

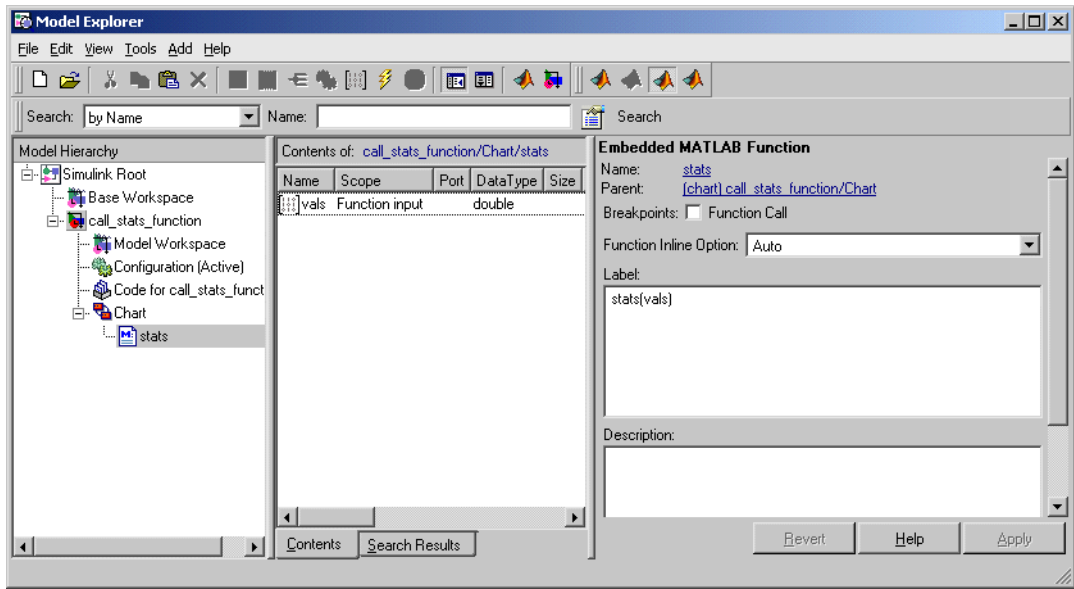
```
return_value = function_name (argument1, argument2,...)
```

Each argument and the return value can be a scalar, vector, or matrix of values. A matrix is a two-dimensional array of values. A vector is a matrix with a row or column dimension of 1. Multiple return values are not allowed.

- 8 In the Stateflow diagram, double-click the function stats to edit its function body.
- 9 In the **Embedded MATLAB Editor**, select the Explore/Define Data tool  to open the **Model Explorer**.

The **Model Explorer** window appears, as shown.

Notice that the function stats is highlighted in the **Model Hierarchy** pane (left). The **Contents** pane (right) displays the argument data vals, a scalar of type double with a scope of Function Input that was added when you labeled the Embedded MATLAB function.



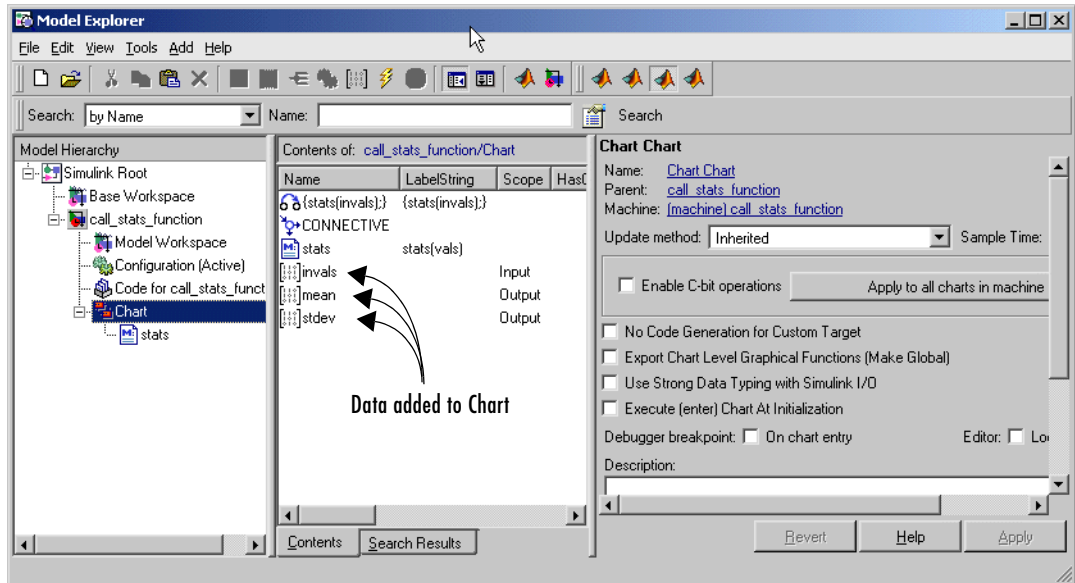
- 10** Double-click the vals row under the **Size** column to change the size of vals to 4.

Because vals is defined for an Embedded MATLAB function, its first index, which is not displayed, is 1 by default, as in MATLAB arrays.

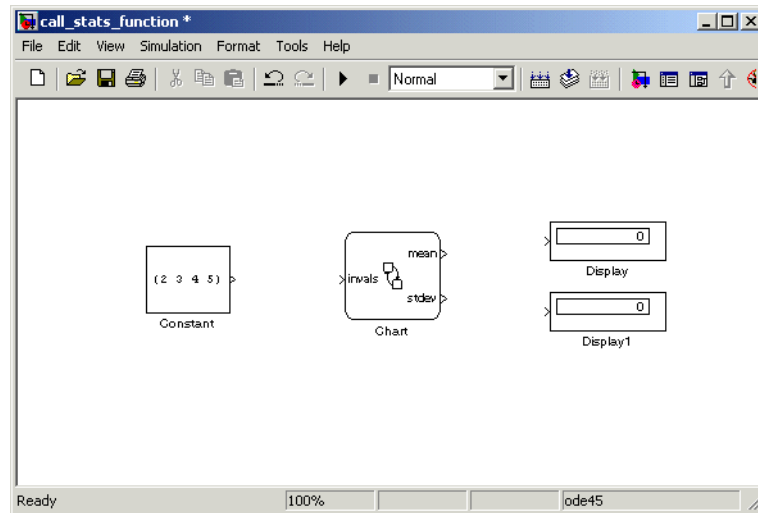
- 11** In the **Model Explorer** window, select the **Chart** level in the **Model Hierarchy** pane on the left and add the following data:

Name	Scope	Size
invals	Input from Simulink	4
mean	Output to Simulink	Scalar (no change)
stdev	Output to Simulink	Scalar (no change)

You should now see the following data in the **Model Explorer**:

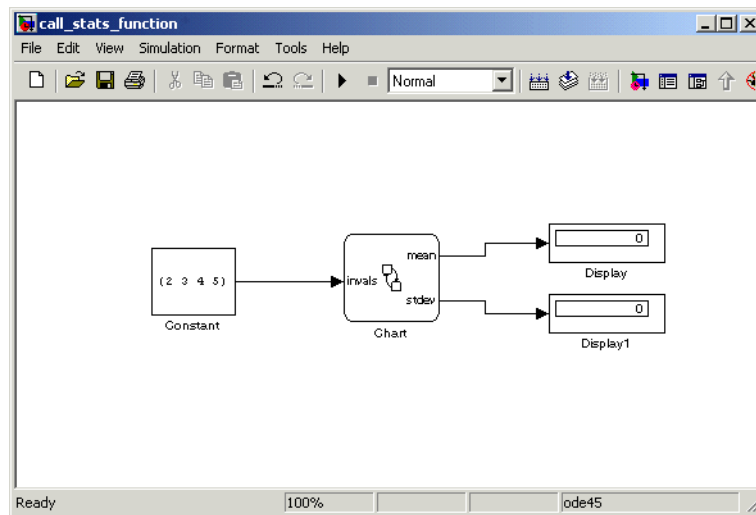


Now that you have added the data `invals`, `mean`, and `stdev` to the Stateflow diagram `Chart`, notice that an input port for `invals` and output ports for `mean` and `stdev` appear on the Stateflow block in Simulink.



Notice in the **Model Explorer** that the value of **Port** for mean is 1 and for stdev is 2. The value of **Port** orders the ports from the top to the bottom of the block for both input and output data. You can change the order of the ports by entering a new value for the **Port** field.

- 12 Connect the Constant block and the Display blocks to the ports of the Stateflow block, as shown, and save the model (call_stats_function).

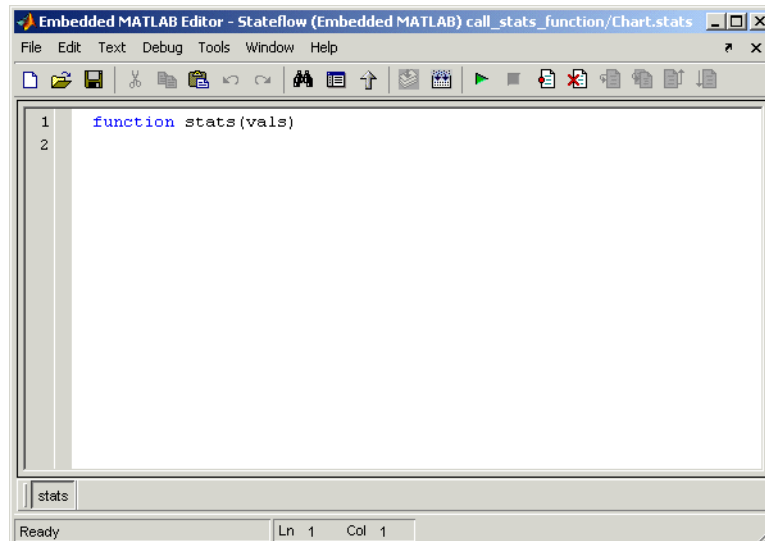


Programming a Stateflow Embedded MATLAB Function

You program an Embedded MATLAB function with the Embedded MATLAB language, a rich subset of MATLAB. Specify the contents of the Embedded MATLAB function stats that you create in “Building a Simulink Model with a Stateflow Embedded MATLAB Function” on page 11-5 with the following steps:

- 1 If not already open, open the `call_stats_function` model that you save at the end of “Building a Simulink Model with a Stateflow Embedded MATLAB Function” on page 11-5, and open its Stateflow block and the Embedded MATLAB function stats inside.
- 2 In the Stateflow diagram, double-click the Embedded MATLAB function stats to open it for editing.


The **Embedded MATLAB Editor** appears with a function header for the function stats as shown.



The **Embedded MATLAB Editor** is titled with the syntax *<model name>/<Embedded MATLAB function name>*. In this case, the model name is `call_stats_function`, and the function name is `Chart.stats`, where `Chart` is the name of the owning Stateflow block in Simulink.

The first line of the function, the function header, is already present:

```
function stats(vals)
```

This function header is taken from the label that you add to the function in the Stateflow diagram. You can edit it directly in the **Embedded MATLAB Editor**. Changing the function header in the editor changes the Embedded MATLAB function label in the Stateflow diagram editor when you close the window or select the Update Diagram tool  in the editor toolbar.

- 3 After the function header, enter a line space and the following comment lines:

```
% calculates a statistical mean and a standard  
% deviation for the values in vals.
```

Enter text in the **Embedded MATLAB Editor** as you would for a normal text editor. If you require more advanced operations while using this editor, see the detailed reference section “The Embedded MATLAB Function Editor” in Simulink documentation.

- 4 Enter a line space and replace the default function line `y = u;` with `len = length(vals);`

The function `length` is an example of a built-in function supported by the Embedded MATLAB run-time function library. This `length` function works just like the MATLAB function `length`, and returns the vector length of its argument `vals`. However, when you build a simulation target for this function, the function `length` is implemented with generated C code. Callable functions supported for Embedded MATLAB blocks are listed in the topic “Using the Run-Time Library of Functions.”

The variable `len` is an example of implicitly declared local data. Because the Embedded MATLAB run-time library function `length` returns a scalar double, `len` is scalar data of type double. You can declare `len` to have a different type and size by changing the way you declare it. See “Declaring Local Variables Implicitly” in Simulink documentation for a description and examples.

Implicitly declared local data is temporary data. It comes into existence only when the function is called and disappears when the function is exited. You can declare local data for an Embedded MATLAB function to be persistent by using `persistent`. You can also declare local data for an Embedded MATLAB function that is persistent by declaring it in the **Model Explorer**.

- 5 Enter the following lines to calculate the value of `mean` and `stdev`:

```
mean = avg(vals, len);  
stdev = sqrt(sum((vals-mean).^2), len)/len);
```

`stats` stores the mean and standard deviation for the values in `vals` in the Stateflow data `mean` and `stdev`. Since these data are defined for the chart, which is the parent of `stats`, you can use them directly in the Embedded MATLAB function. You can use Stateflow data of all types and sizes that are in the scope of an Embedded MATLAB function directly, except for data of type `m1`, which is not supported in Embedded MATLAB functions.

Two-dimensional arrays with a single row or column of elements are treated as vectors or as matrices in Embedded MATLAB functions. For example, in the Embedded MATLAB function `stats`, the argument `vals` is a four element vector. You can access the fourth element of this vector with the matrix notation `vals(4,1)` or the vector notation `vals(4)`.

The preceding line uses the functions `avg`, `sqrt`, and `sum` to compute the values of `mean` and `stdev`. `sqrt` and `sum` are Embedded MATLAB run-time library functions. `avg` is a subfunction of the Embedded MATLAB function `stats` that you define in the next step. When resolving function names, Embedded MATLAB functions look for subfunctions first, followed by Embedded MATLAB run-time library functions.

Note For simulation targets, if a called function in an Embedded MATLAB function cannot be resolved as a subfunction or Embedded MATLAB run-time library function, it is resolved as a MATLAB function, which is executed in the MATLAB workspace by the simulation application during simulation. See “Calling MATLAB Functions” in Simulink documentation for a description of this process and an example.

- 6** Enter the following code line to plot the input values in `vals`.

```
plot (vals, '+' );
```

This line calls the function `plot` to plot the input values sent to `stats` against their vector index. Because the Embedded MATLAB run-time library has no `plot` function, the Embedded MATLAB function cannot resolve this call with a subfunction or an Embedded MATLAB run-time function. Instead, it replaces this call with a call to the MATLAB `plot` function in the generated code for the simulation target.

See “Calling MATLAB Functions” in documentation for more details on using this mechanism to call MATLAB functions from Embedded MATLAB functions.

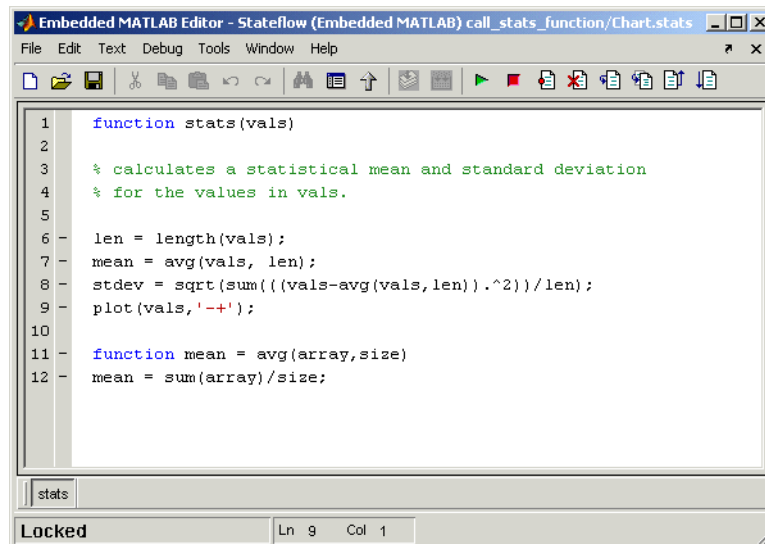
- 7** Enter a line space followed by the subfunction `avg` called in an earlier line.

```
function mean = avg(array,size)
mean = sum(array)/size;
```

The preceding code lines define the subfunction `avg` at the end of the top-level function `stats`. The header for `avg` defines two arguments, `array` and `size`, and a single return value, `mean`. The only line of programming in the subfunction `avg` calculates the average of the elements of the `array` argument by summing them with a call to the Embedded MATLAB library function `sum` and dividing the result with the argument `size`.

Embedded MATLAB functions support subfunctions with multiple return values. As in MATLAB, separate the return values by commas or spaces and surround the list with square brackets. For more information on creating subfunctions, see “Subfunctions” in MATLAB online documentation.

The finished Embedded MATLAB function stats has the following appearance:



```
Embedded MATLAB Editor - Stateflow (Embedded MATLAB) call_stats_function/Chart.stats
File Edit Text Debug Tools Window Help
[Icons]
1 function stats(vals)
2
3 % calculates a statistical mean and standard deviation
4 % for the values in vals.
5
6 len = length(vals);
7 mean = avg(vals, len);
8 stdev = sqrt(sum((vals-avg(vals, len)).^2)/len);
9 plot(vals, '-+');
10
11 function mean = avg(array, size)
12 mean = sum(array)/size;
stats
Locked Ln 9 Col 1
```

8 Save the model (call_stats_function).

Debugging a Stateflow Embedded MATLAB Function


In “Programming a Stateflow Embedded MATLAB Function” on page 11-13, you finish a Simulink model by programming the Embedded MATLAB function stats for its Stateflow block. Now you need to begin the process of debugging stats in its model.

Use the topics in the following steps to debug the Embedded MATLAB function stats in its Stateflow diagram.

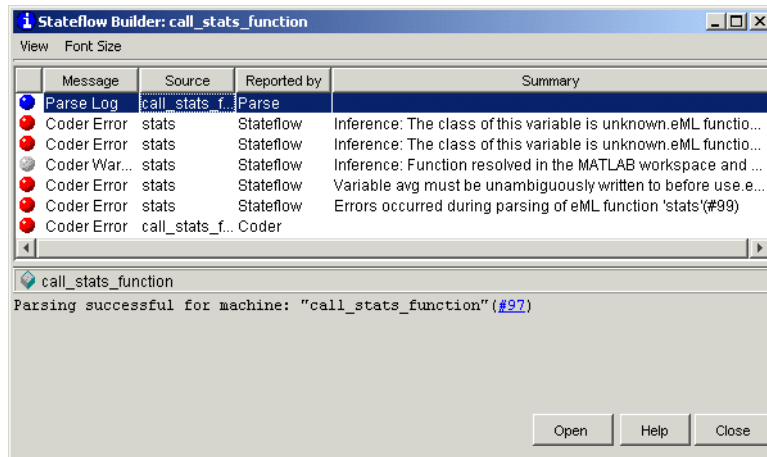
- “Checking Embedded MATLAB Functions for Syntax Errors” on page 11-18 — Describes the way that Stateflow checks Embedded MATLAB functions for syntactical errors.
- “Run-Time Debugging for Embedded MATLAB Functions” on page 11-20 — Executes the model and tests the Embedded MATLAB function with different input data values.

Checking Embedded MATLAB Functions for Syntax Errors

Before you can build a simulation application for a model, you need to fix any possible syntax or grammar errors that keep the application from building. Use the following steps to check the Embedded MATLAB function stats for syntax and grammar errors that keep it from building:

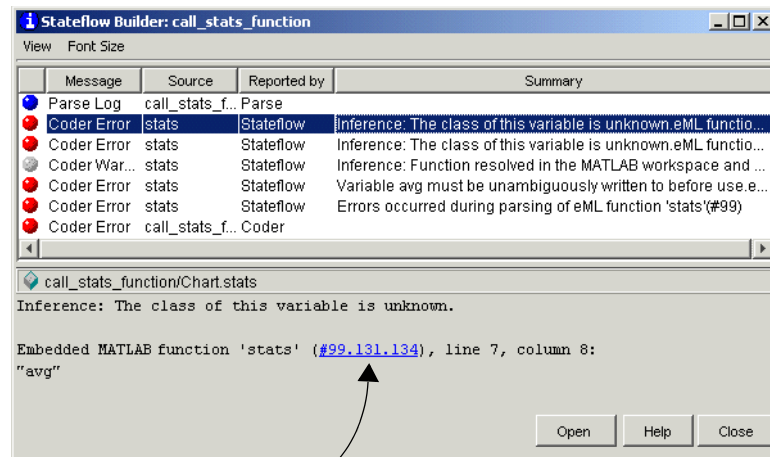
- 1 If not already open, open the `call_stats_function` model that you save at the end of “Programming a Stateflow Embedded MATLAB Function” on page 11-13, and open its Stateflow block and the Embedded MATLAB function stats inside.
- 2 In the **Embedded MATLAB Editor**, select the Build tool  to build a simulation application for the example Simulink model.

If there are no errors or warnings, the **Builder** window appears and reports a message of success. If errors are found, the **Builder** window lists them. For example, if you change the call to the subfunction `avg` to a call to a nonexistent subfunction `aug` in `stats`, the following **Builder** window appears:



Each detected error begins with a red button.

- 3 Click the first error line to highlight it and display information for it.



Click to display error source in Embedded MATLAB Editor

The diagnostic message explaining the error appears in the bottom pane.

- 4 Click the link in the diagnostic message to display the offending line highlighted in the **Embedded MATLAB Editor**, as shown.

```

1  function stats(vals)
2
3  % calculates a statistical mean and standard deviation
4  % for the values in vals.
5
6 - len = length(vals);
7  mean = avg(vals, len);
8  stdev = sqrt(sum((vals-avg(vals, len)).^2)/len);
9  plot(vals, '-+');
10
11 function mean = avg(array, size)
12 mean = sum(array)/size;

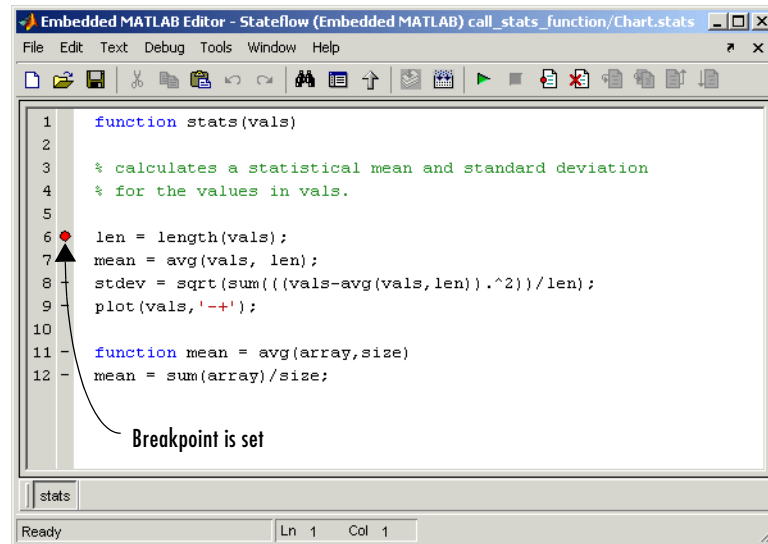
```

Run-Time Debugging for Embedded MATLAB Functions

You use simulation to test your Embedded MATLAB functions for run-time errors that are not detectable by Stateflow diagnostics. When you start simulation of your model, Simulink performs diagnostic tests of your completed Embedded MATLAB functions for missing or undefined information and possible logical conflicts as described in “Checking Embedded MATLAB Functions for Syntax Errors” on page 11-18. If no errors are found, Stateflow begins the simulation of your model.

Use the following procedure to simulate and debug the stats Embedded MATLAB function during run-time conditions:

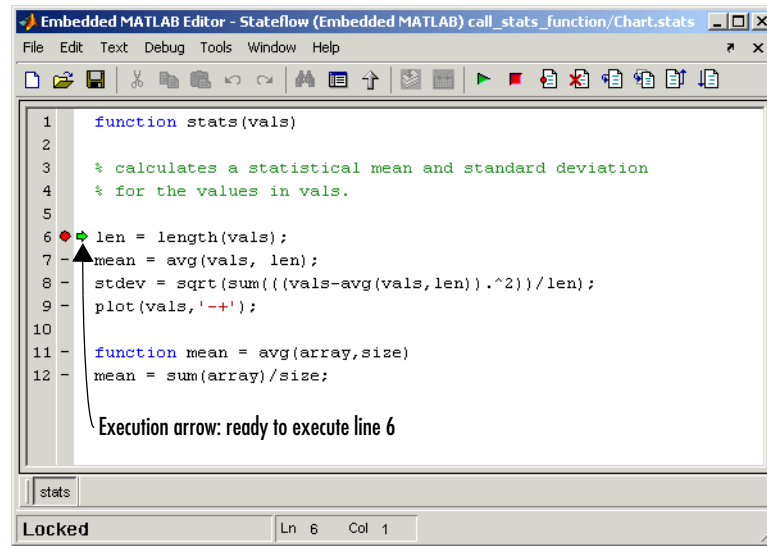
- 1 In the **Embedded MATLAB Editor**, in the left margin of line 6, click the dash (-) character.



A small red ball appears in the margin of line 6 indicating that you have set a breakpoint.


- 2 Click the Start Simulation tool  to begin simulating the model.

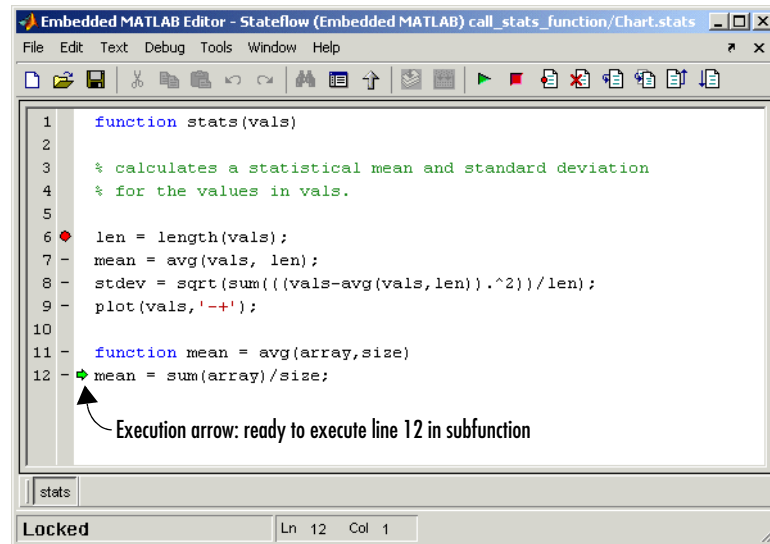
If you get any errors or warnings, make corrections before you try to simulate again. Otherwise, simulation pauses when execution reaches the breakpoint you set. This is indicated by a small green arrow in the left margin as shown.




- 3 Click the Step tool  to advance execution one line to line 7.

Notice that line 7 calls the subfunction avg. If you click Step here, execution advances to line 8, past the execution of the subfunction avg. In order to track execution of the lines in the subfunction avg, you need to click the Step In tool.

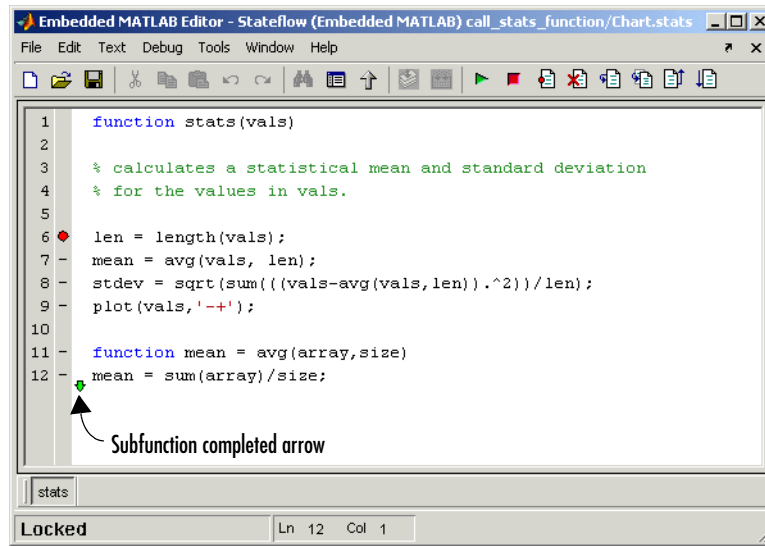
- 4 Click the Step In tool  to advance execution to the first line of the called subfunction avg as shown.



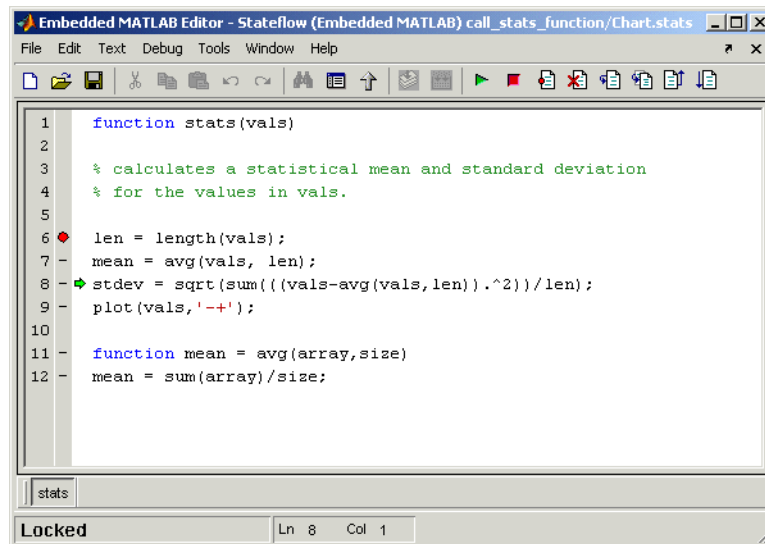
Once you are in the subfunction, you can advance through the subfunction one line at a time with the Step tool. If the subfunction calls another subfunction, use the Step In tool to step into it. If you want to continue through the remaining lines of the subfunction and go back to the line after the subfunction call, click the Step Out tool .

- 5 Click the Step tool to execute the only line in the subfunction avg.

The subfunction avg finishes its execution, and you see a green arrow pointing down under its last line as shown.



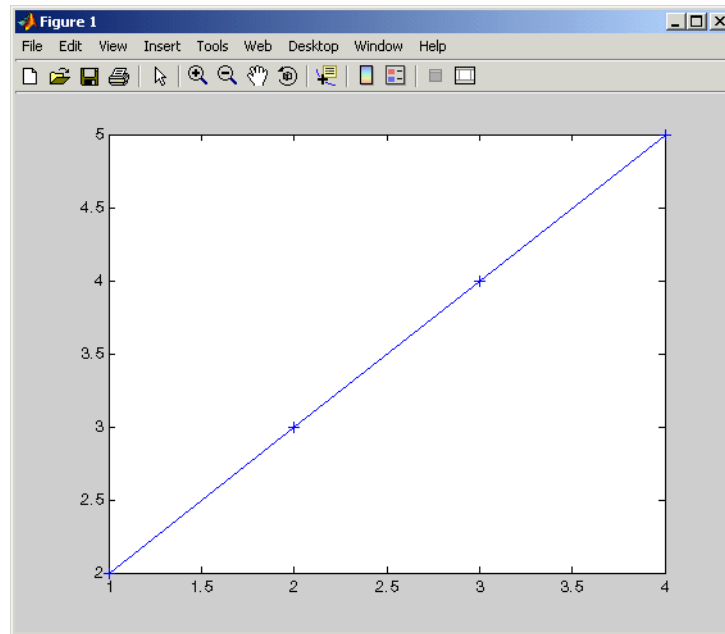
- 6 Click the Step tool to return to the function stats.



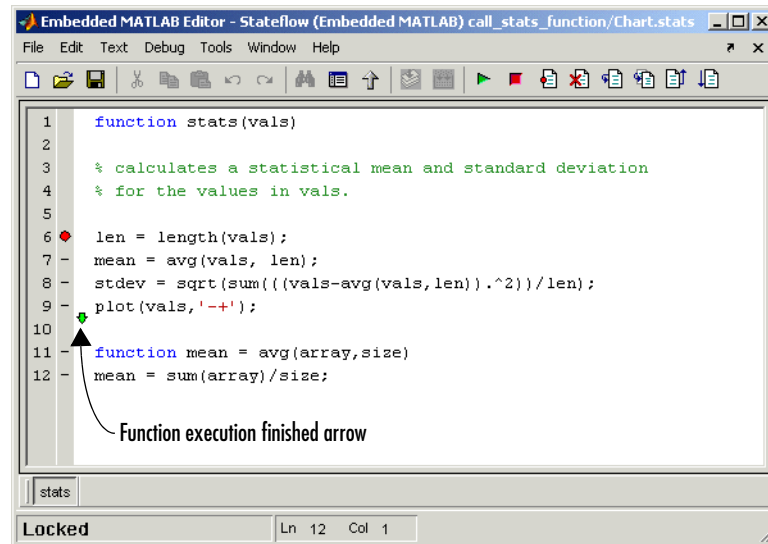
Execution advances to the line after to the call to the subfunction avg, line 8.

- Click the Step tool twice to execute line 8 and the plot function in MATLAB in line 9.

The plot function executes in MATLAB, and you see the following plot.

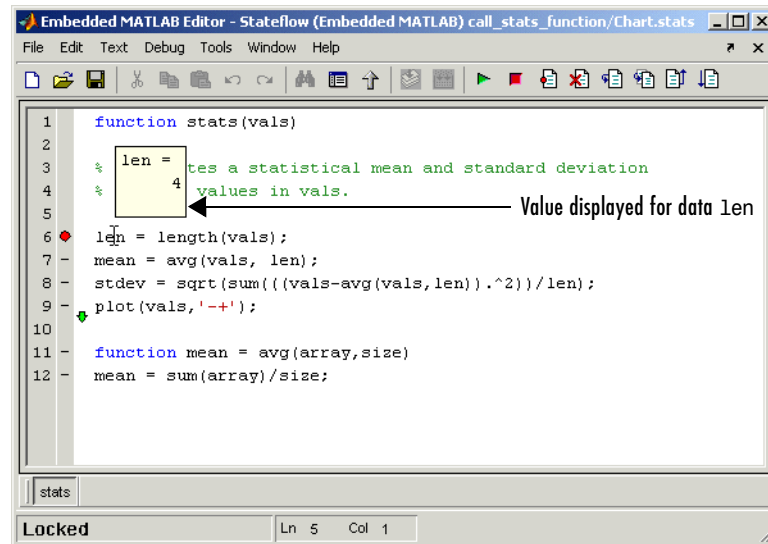


In the **Embedded MATLAB Editor**, a green arrow pointing down under line 9, indicating the completion of the function stats.



- 8 To display the value of the variable `len`, place the mouse cursor over the text `len` in line 6 for at least a second.

The value of `len` appears adjacent to the cursor as shown:



You can display the value for any data in the Embedded MATLAB block function in this way, no matter where it appears in the function. For example, you can display the values for the vector `vals` by placing the cursor over it as an argument to the function `length` in line 6, or as an argument in the function header.


You can also report the values for Embedded MATLAB block function data in the MATLAB window during simulation. When you reach a breakpoint, the `DB>>` command prompt appears in the MATLAB window (you might have to press **Enter** to see it). At this prompt you can inspect data defined for the Embedded MATLAB block by entering the name of the data as shown in the following example:

```

DB>> len
len =
     4
DB>>

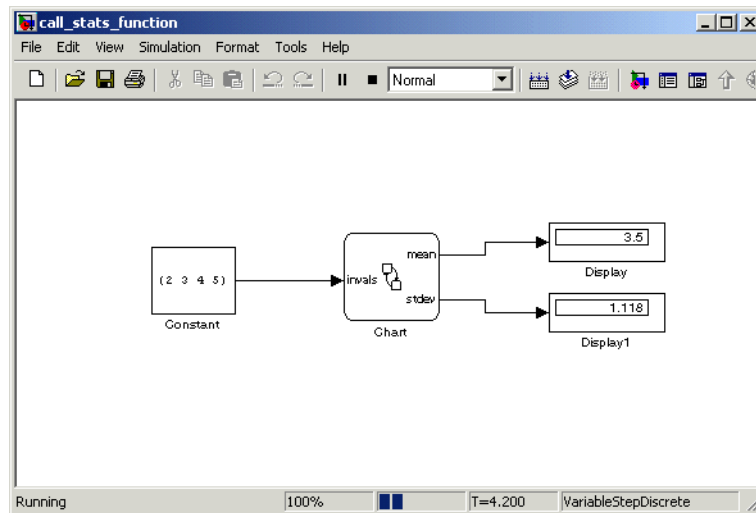
```


As another debugging alternative, you can display the execution result of an Embedded MATLAB function line by omitting the terminating semicolon. If you do, execution results for the line are echoed to the MATLAB window during simulation.

- 9 Click the Continue tool  to leave the function until it is called again and the breakpoint on line 6 is reached.

At any point in a function, you can advance through the execution of the remaining lines of the function with the Continue tool. If you are at the end of the function, clicking the Step tool accomplishes the same thing.

In the Simulink window, the computed values of mean and stdev now appear in the Display blocks.



- 10 In the **Embedded MATLAB Editor**, select the Exit Debug Mode tool  to stop simulation.

Model Coverage for an Embedded MATLAB Function

The **Model Coverage** tool in Simulink reports model coverages for the decisions and conditions of Embedded MATLAB functions in Stateflow. For example, the Embedded MATLAB function if statement

```
if (x > 0 || y > 0)
    reset = 1;
```

contains a decision with two conditions ($x > 0$ and $y > 0$). You use the **Model Coverage** tool for Embedded MATLAB functions to make sure that all decisions and conditions are taken during simulation of the model.

See the following topics for a description of model coverage for an example Embedded MATLAB function:

- “Types of Model Coverage in Embedded MATLAB Functions” on page 11-30 — Lists the types of elements in an Embedded MATLAB function that receive model coverage during simulation.
- “Creating a Model with Embedded MATLAB Function Decisions” on page 11-30 — Shows you an example model that you use to examine model coverage for Embedded MATLAB functions.
- “Understanding Embedded MATLAB Function Model Coverage” on page 11-35 — Describes the individual coverages for the Model Coverage report of the example Embedded MATLAB function.

For a description of model coverage for other Stateflow objects, see “Understanding Model Coverage for Stateflow Charts” on page 13-42.

Note The Model Coverage tool requires a Simulink Verification and Validation license.

Types of Model Coverage in Embedded MATLAB Functions

During simulation, the following Embedded MATLAB block function statements are tested for Decision Coverage:

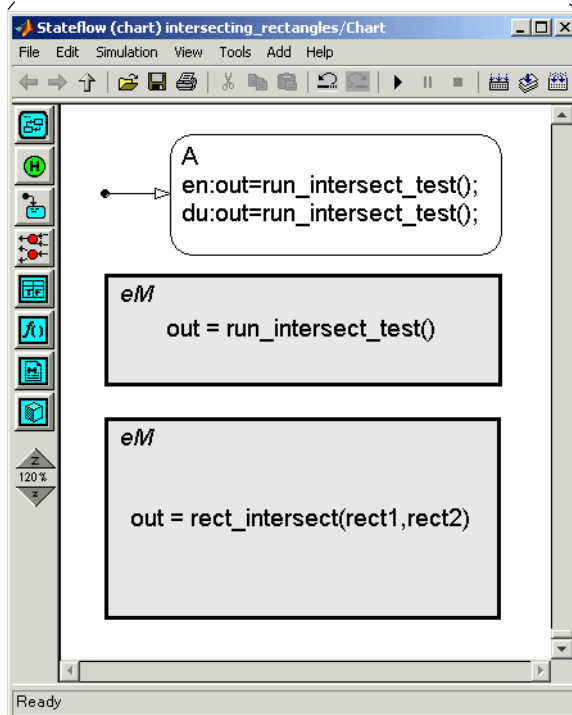
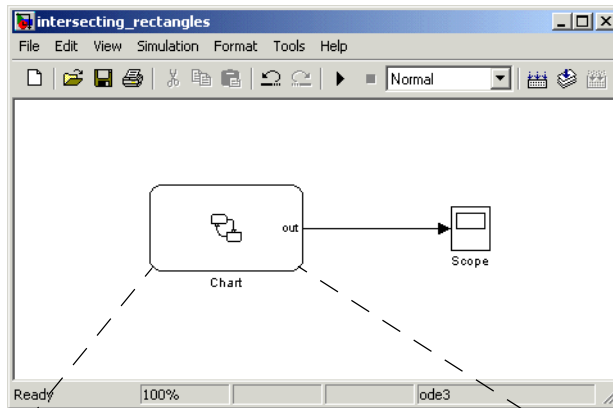
- **function header** — Decision coverage is 100% if the function or subfunction is executed.
- **if**— Decision coverage is 100% if the `if` expression evaluates to true at least once and false at least once.
- **switch**— Decision coverage is 100% if every switch case is taken, including the fall-through case.
- **for**— Decision coverage is 100% if the equivalent loop condition evaluates to true at least once, and false at least once.
- **while** — Decision coverage is 100% if the equivalent loop condition evaluates to true at least once, and false at least once.

During simulation, the following logical conditions are tested for Condition Coverage and MCDC in the Embedded MATLAB block function:

- `if` statement conditions
- `while` statement conditions, if present

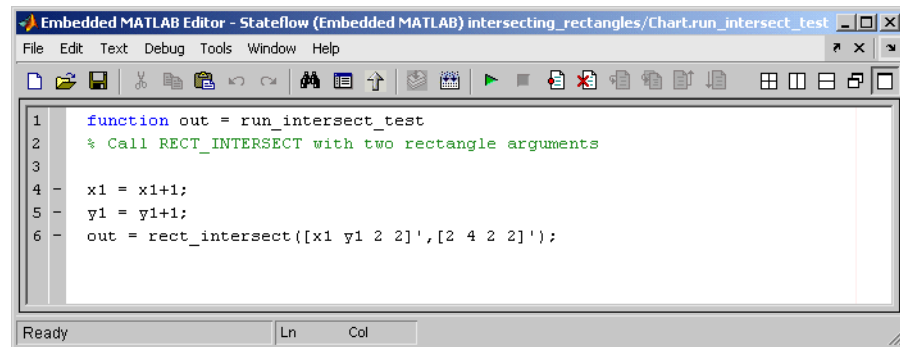
Creating a Model with Embedded MATLAB Function Decisions

In this topic you examine an example model you can use to generate a model coverage report for two Embedded MATLAB functions in Stateflow. The following model is named `intersecting_rectangles`. It contains a single Stateflow block with output data sent to a Scope block as shown.



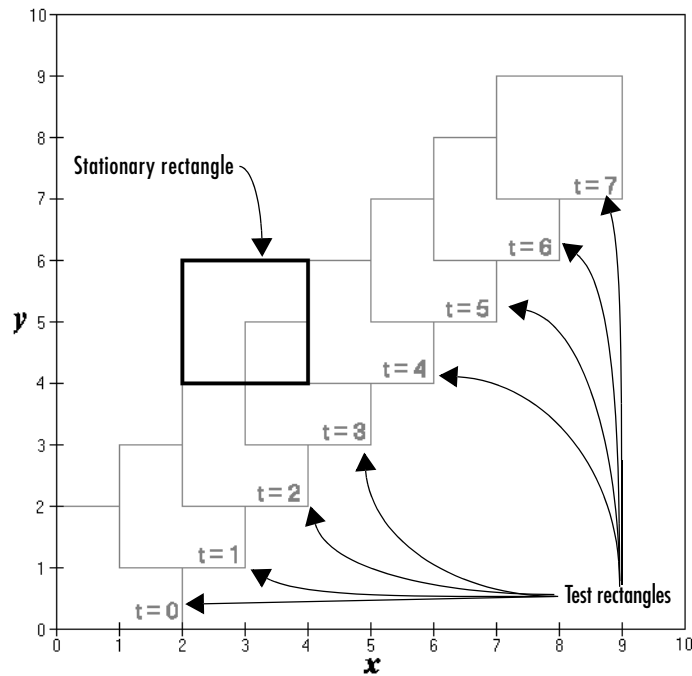
The preceding Stateflow diagram has a state with a default transition and entry and during actions. The state executes its entry action the first time that it is entered for the first time sample. Each succeeding time sample calls the during action of the active state.

The entry and during actions of state A call the Embedded MATLAB function `run_intersect_test`, which appears as follows in the **Embedded MATLAB Editor** window:



```
Embedded MATLAB Editor - Stateflow (Embedded MATLAB) intersecting_rectangles/Chart.run_intersect_test
File Edit Text Debug Tools Window Help
[Icons]
1 function out = run_intersect_test
2 % Call RECT_INTERSECT with two rectangle arguments
3
4 - x1 = x1+1;
5 - y1 = y1+1;
6 - out = rect_intersect([x1 y1 2 2]',[2 4 2 2]');
```

`run_intersect_test` calls the function `rect_intersect` with two rectangle arguments that each consist of coordinates for the lower left corner of the rectangle (origin), and its width and height. The first rectangle is a test rectangle, and the second is a stationary rectangle. The coordinates for the origin of the test rectangle are represented by the Stateflow data `x1` and `y1`, which are both initialized to -1. This means that `x1` and `y1` are 0 for the first sample. The progression of rectangle arguments during simulation is as follows:



In the preceding display, the stationary rectangle is shown in bold with a lower left origin of $(2, 4)$ and a width and height of 2. At time $t = 0$, the first test rectangle has an origin of $(0, 0)$ and a width and height of 2. For each succeeding sample, the origin of the test rectangle is incremented by $(1, 1)$. The rectangles at sample times $t = 2, 3$, and 4 intersect with the test rectangle.

The function `rect_intersect`, as shown, checks to see if two rectangles intersect.

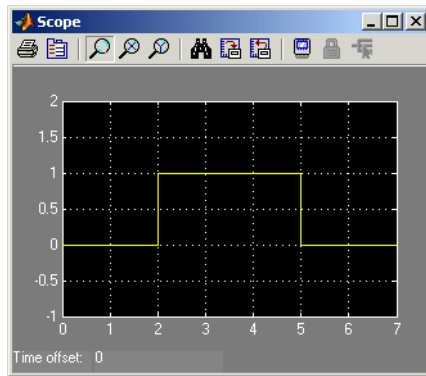
```

1  function out = rect_intersect(rect1,rect2)
2  % Return 1 if two rectangles intersect; 0 if not.
3
4  left1 = rect1(1);
5  bottom1 = rect1(2);
6  right1 = left1 + rect1(3);
7  top1 = bottom1 + rect1(4);
8
9  left2 = rect2(1);
10 bottom2 = rect2(2);
11 right2 = left2 + rect2(3);
12 top2 = bottom2 + rect2(4);
13
14 if (top1 < bottom2 || top2 < bottom1)
15     out = 0;
16 else
17     if (right1 < left2 || right2 < left1)
18         out = 0;
19     else
20         out = 1;
21     end
22 end

```

rect_intersect receives the two rectangle arguments from run_intersect_test. It first calculates horizontal (x) coordinates for the left and right sides, and vertical (y) values for the top and bottom sides for each rectangle and compares them in the nested if-else decisions shown. The function returns a logical value of 1 if the rectangles intersect and 0 if they do not.

Scope output during simulation, which plots the return value against the sample time, confirms the intersecting rectangles for sample 2, 3, and 4.



Understanding Embedded MATLAB Function Model Coverage

Model coverage reports are generated during simulation if you specify them (see “Making Model Coverage Reports” on page 13-43). When simulation is finished, the model coverage report appears in a browser window. After the summary for the model, the Details section reports on each of the parts of the model. Model coverage for the parts of the example model in “Creating a Model with Embedded MATLAB Function Decisions” on page 11-30 appears in the following order:

- Model “intersecting_rectangles”
- Subsystem “Chart”
- Chart “Chart”
 - Function `rect_intersect`
 - #1: function out = `rect_intersect(rect1,rect2)`
 - #14: if (top1 < bottom2 || top2 < bottom1)
 - #17: if (right1 < left2 || right2 < left1)
 - Function “run_intersect_test”
 - #1: function out = `run_intersect_test`

The reports for the Embedded MATLAB functions `rect_intersect` and `run_intersect_test` appear in alphabetical order as part of the report on their parent, the Stateflow block Chart. The reports on individual decisions for each function appear in numerical line order. Line numbers are indicated by the # character.

The following subtopics examine the model coverage report for the example model in reverse order of the report. Reversing the order helps you make sense of the summary information that appears at the top of each section.

Model Coverage for Embedded MATLAB Function `run_intersect_test`

Model coverage for the Embedded MATLAB function `run_intersect_test`, which sends test rectangles to the function `rect_intersect`, appears in the model coverage report as shown.

The screenshot shows a web browser window titled "intersecting_rectangles Coverage Report". The address bar shows the location: "C:/TEMP/tp302987_intersecting_rectangles_navigation.html". The page has two tabs: "Summary" and "Details". The main content area displays the following information:

Function "run_intersect_test"

Parent: [intersecting_rectangles/Chart](#)

Metric	Coverage
Cyclomatic Complexity	1
Decision (D1)	100% (1/1) decision outcomes

```

1 function out = run_intersect_test
2 % Call RECT_INTERSECT with two rectangle arguments
3
4 x1 = x1+1;
5 y1 = y1+1;
6 out = rect_intersect([x1 y1 2 2],[2 4 2 2]);
    
```

#1: function out = run_intersect_test

Decisions analyzed:

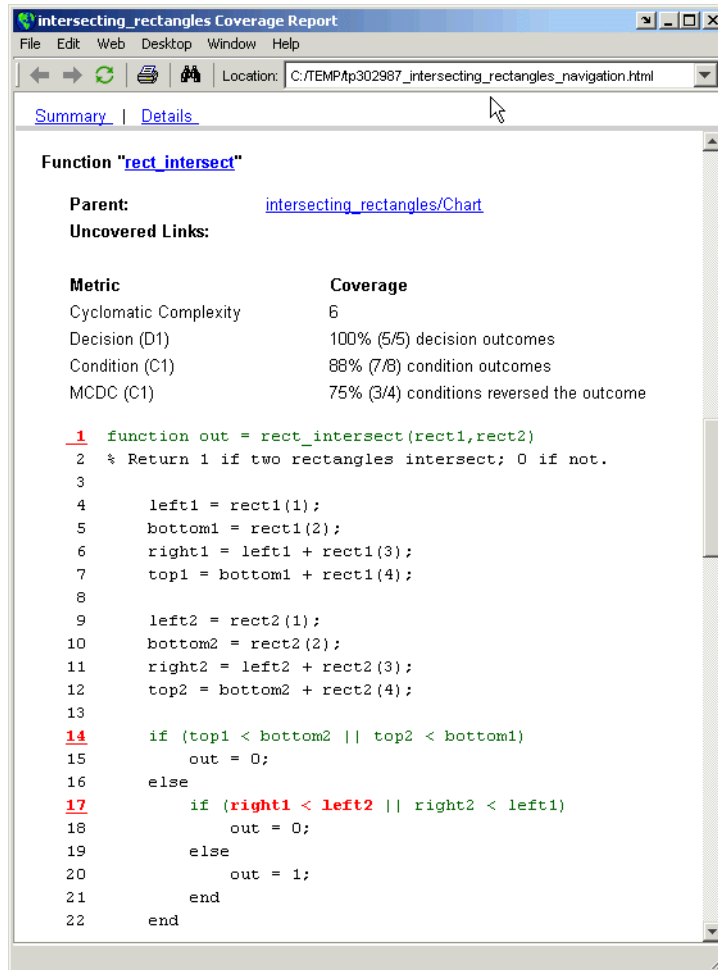
function out = run_intersect_test	100%
executed	8/8

The report on `run_intersect_test` begins with the function name `run_intersect_test`, which links to an **Embedded MATLAB Editor** for the function. Following the name is a link to the model coverage report for the parent of `run_intersect_test`, the Stateflow chart `Chart`. Coverage continues with a summary of the coverage metrics for the function followed by a code listing. The line number for the first line of the listing is highlighted in bold red, which is actually a link to an analysis for that decision below.

The first line of every function receives coverage analysis indicative of the decision to run the function. Its coverage here indicates that the function `run_intersect_test` executed 8 out of 8 times for the samples taken at 0 through 7 seconds. Because this is the only decision in the function `run_intersect_test`, coverage for it in the metrics table above indicates the occurrence of 1 out of 1 possible outcomes. In this case, the only possible outcome is function execution. In other words, the function was executed during simulation.

Coverage for Embedded MATLAB Function `rect_intersect`

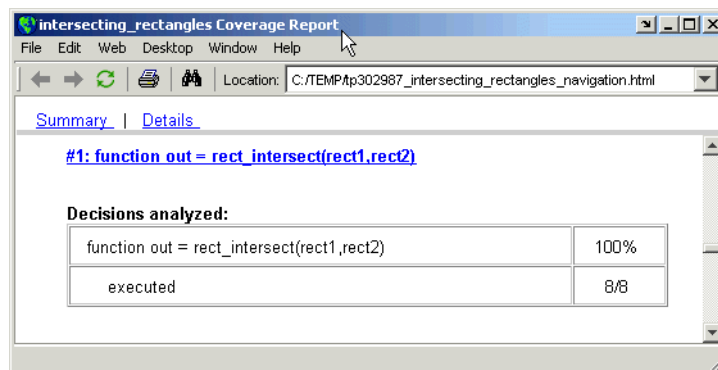
Model coverage for the Embedded MATLAB function `rect_intersect`, which tests rectangles for intersection, appears first in the model coverage report as shown.



The coverage metrics for `rect_intersect` include Decision, Condition, and MCDC coverage. These are best understood after you examine the coverage for the decisions in `rect_intersect` highlighted in the listing below.

The listing for `rect_intersect` includes three highlighted line numbers. The first line is highlighted as the decision on whether or not to execute the function. Highlighted line numbers 14 and 17 indicate decisions in a nested `if-else` statement. Notice that the condition `right1 < left2` in line 17 is highlighted in red. This means that this condition did not test both the true and false possible outcomes for this decision during simulation. Exactly which of the outcomes was not tested is answered by the metrics for the decision in line 17. The metrics for line 17 and the remaining decisions appear below listed in numerical line order, and are easily accessed through the line number links in the listing.

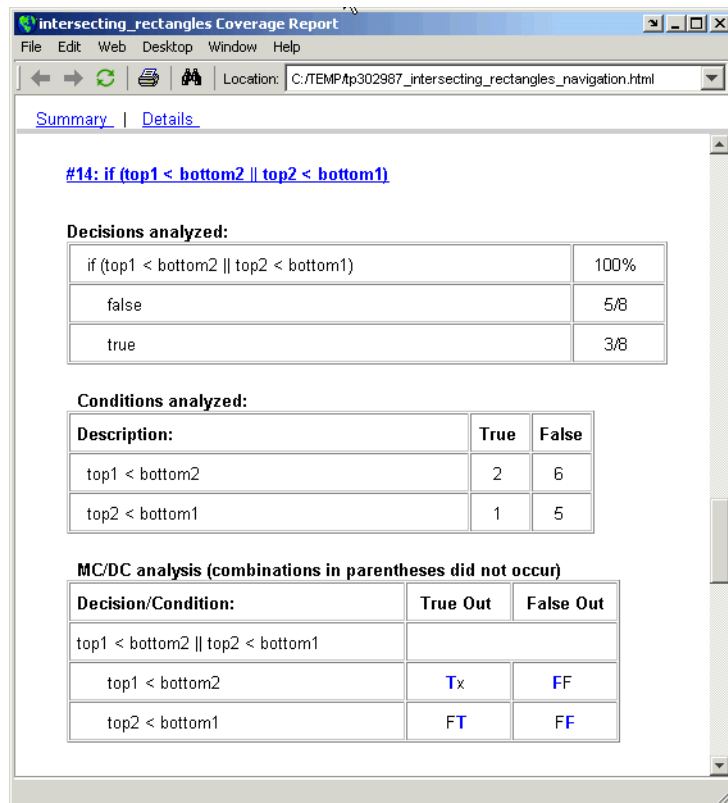
Coverage for Line 1. Coverage metrics for line 1 appear directly below the listing for `rect_intersect` as shown.



Decisions analyzed:	
function out = rect_intersect(rect1,rect2)	100%
executed	8/8

The first line of every function receives coverage analysis indicative of the decision to run the function in response to a call. The preceding table indicates that `rect_intersect` executed. Coverage for this decision, which is equivalent to decision D1 in the metrics table for `rect_intersect`, is therefore 100%.

Coverage for Line 14. Coverage metrics for line 14 appear directly below the coverage metrics for line 1 as shown.

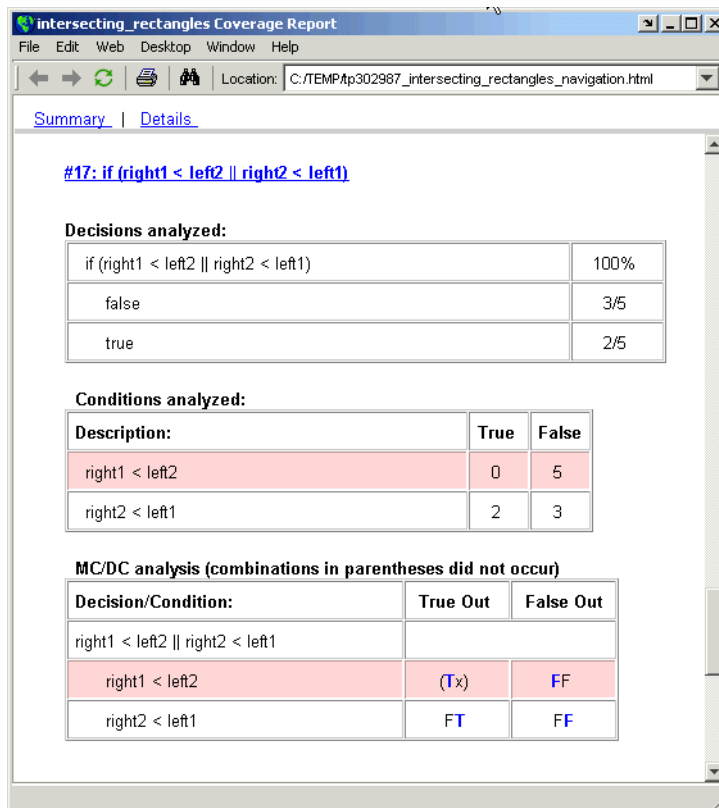


The **Decisions analyzed** table indicates that there are two possible outcomes for the decision in line 14: false and true. 5 of 8 times the decision evaluated false, and the remaining times (3) it evaluated true. Because both the true and false outcome occurred during simulation, Decision Coverage is 100%.

The following **Conditions analyzed** table sheds some light on the decision in line 14. Because this decision consists of two conditions linked by a logical or (||) operation, only one condition must evaluate true for the decision to be true. Also, if the first condition evaluates to true, there is no need to evaluate the second condition. The first condition, top1 < bottom2, was evaluated 8 times, and was true twice. This means that it was necessary to evaluate the second condition only 6 times. In only one case was it true, which brings the total of the true occurrence for the decision to 3 as reported in the **Decisions analyzed** table.

MCDC coverage looks for decision reversals that occur because one condition outcome changes from T to F or from F to T. The table identifies all possible combinations of outcomes for the conditions that lead to a reversal in the decision. The character x is used to indicate a condition outcome that is irrelevant to the decision reversal. Reversing condition outcomes that are not achieved during simulation are marked with a set of parentheses. There are no parentheses, so all decision reversing outcomes occurred, and MCDC Coverage is complete for this decision.

Coverage for `rect_intersect` Line 17. Coverage metrics for line 17 appear directly below the coverage metrics for line 14, as shown.



The screenshot shows a web browser window titled "intersecting_rectangles Coverage Report". The address bar shows the location: "C:/TEMP/p302987_intersecting_rectangles_navigation.html". The page content includes a navigation menu with "Summary" and "Details" links. The main content area displays the following information:

#17: `if (right1 < left2 || right2 < left1)`

Decisions analyzed:

<code>if (right1 < left2 right2 < left1)</code>	100%
false	3/5
true	2/5

Conditions analyzed:

Description:	True	False
<code>right1 < left2</code>	0	5
<code>right2 < left1</code>	2	3

MC/DC analysis (combinations in parentheses did not occur)

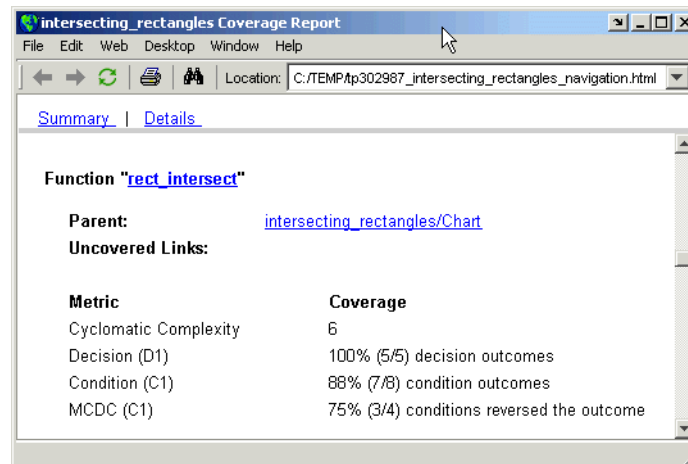
Decision/Condition:	True Out	False Out
<code>right1 < left2 right2 < left1</code>		
<code>right1 < left2</code>	(Tx)	FF
<code>right2 < left1</code>	FT	FF

The line 17 decision `if (right1 < left2 || right2 < left1)` is nested in the `if` statement of the line 14 decision and is evaluated only if the line 14 decision is false. Because the line 14 decision evaluated false 5 times, line 17 is evaluated 5 times, 3 of which were false. Because both the true and false outcomes were achieved, Decision Coverage for line 17 is 100%.

Because line 17, like line 14, has two conditions related by a logical OR operator (`||`), condition 2 is tested only if condition 1 is false. Because condition 1 tests false 5 times, condition 2 is tested 5 times. Of these, condition 2 tests true 2 times and false 3 times, which accounts for the two occurrences of the true outcome for this decision.

Because the first condition of the line 17 decision does not test true, both outcomes did not occur for that condition, and the Condition Coverage for the first condition is highlighted with a rose color. Because the first condition of the line 17 decision does not test true, MCDC coverage is also highlighted in the same way for a decision reversal based on the true outcome for that condition.

Coverage for All `rect_intersect` Lines. Reviewing the coverage report for each line of `rect_intersect` in the previous topics makes sense of the coverage metrics reported at the beginning of the model coverage report for `rect_intersect`, which is as shown.



Reflecting on the model coverage reports for each line of `rect_intersect`, you can draw the following conclusions:

- There are five decision outcomes reported for `rect_intersect` in the line reports: one for line 1 (execute), two for line 14 (true and false), and two for line 17 (true and false). The decision coverage for each line shows 100% coverage. This means that decision coverage for `rect_intersect` is 5 of 5 or 100%.
- There are four conditions reported for `rect_intersect` in the line reports. Lines 14 and 17 each have two conditions, and each condition has two condition outcomes (true and false), for a total of eight condition outcomes in `rect_intersect`. All conditions tested for both the true and false outcome, except for the first condition of line 17 (`right1 < left2`). This means that condition coverage for `rect_intersect` is 7 of 8 or 88%.
- The MCDC tables for each line list two cases of decision reversal for each condition. Only the decision reversal from changing the condition 1 outcome from true to false did not occur during simulation. This means that 3 of 4 or 75% of the possible reversal cases were tested for during simulation. Therefore, only three of a possible four reversals were observed and coverage is 75%.

Building Targets

Stateflow generates code that lets you execute Stateflow diagrams on target computers. Stateflow builds a special simulation target (`sfun`) that lets you simulate your Stateflow application in Simulink. Stateflow also works with Simulink to let you build a Real-Time Workshop (RTW) target application (`rtw`) for running Stateflow and Simulink applications on other computers. Finally, Stateflow lets you build custom targets from Stateflow applications only.

Overview of Stateflow Targets (p. 12-3)	Describes how Stateflow and its companion tools can build targets for virtually any computer.
Adding a Stateflow Target (p. 12-8)	Tells you how to add an RTW or custom target to a Simulink model. Simulation targets are created by default when you create a Simulink model.
Configuring a Simulation Target for Stateflow (p. 12-11)	Tells you how to configure the simulation target (named <code>sfun</code>) for a Simulink model to build a simulation application for the model.
Configuring Real-Time Workshop for Stateflow (p. 12-14)	Tells you how to configure the RTW target (named <code>rtw</code>) for a Simulink model to build an embedded application for the model.
Configuring a Custom Target in Stateflow (p. 12-24)	Tells you how to configure a custom target (named other than <code>sfun</code> or <code>rtw</code>) for a Simulink model to generate code for the Stateflow diagrams in the model.
Integrating Custom Code with Stateflow Targets (p. 12-29)	Tells you how to configure the custom code you include in the target you build. Also tells you how to invoke Stateflow graphical functions from your custom code.
Starting the Build (p. 12-36)	Describes how to start building a simulation target (<code>sfun</code>) or a Real-Time Workshop target (<code>rtw</code>).

Parsing Stateflow Diagrams (p. 12-38)	The parser evaluates the graphical and nongraphical objects in each Stateflow machine against the supported Stateflow notation and the action language syntax. This section describes the Stateflow parser and how its messages appear.
Resolving Event, Data, and Function Symbols (p. 12-44)	Describes the process Stateflow uses when attempting to resolve undefined data, event, and graphical function symbols when you choose to simulate the model, build a target, or generate code.
Error Messages (p. 12-46)	Describes the error messages you might receive when you build a target or parse a diagram.
Generated Files (p. 12-49)	Describes the files that Stateflow generates when you generate code for a target or build the target.

Overview of Stateflow Targets

This section gives you an overview understanding of what targets are in Stateflow and how you use them in the following topics:

- “What Is a Simulink RTW Target?” on page 12-3 — Describes the type of targets you can build from a model in Simulink.
- “What Is a Stateflow Target?” on page 12-3 — Describes the type of targets you can use to configure Stateflow contributions to a target you build for a library model in Simulink.

What Is a Simulink RTW Target?

A Simulink RTW target is a container for specifying the generated code, custom code, and build type used in building an application or producing generated code for a Simulink model. The model represented by an RTW target can include non-Stateflow as well as Stateflow blocks. An RTW target can also run on computers that do not have a floating-point instruction set. Building an RTW target requires the Real-Time Workshop and Stateflow Coder.

What Is a Stateflow Target?

A Stateflow target is a container in Stateflow for specifying the generated code, custom code, and build type used in building an application or producing generated code from the Stateflow blocks in a Simulink model. There are three types of Stateflow targets:

- Simulation target (sfun)
The Stateflow simulation target (named sfun) is a container in Stateflow for specifying the generated code, custom code, and build type used for Stateflow blocks in creating a Simulink simulation target. When you build a simulation target for a Simulink model, the information from a Stateflow simulation target is combined with the information for the rest of the model in a Simulink simulation target and the target is built.
- Real-Time Workshop (rtw) target
The Stateflow RTW target (named rtw) is a container in Stateflow for specifying the generated code, custom code, and build type used for Stateflow blocks in creating a Simulink RTW target for library models. When you build an RTW target for a Simulink model, the information from a Stateflow RTW

target is combined with the information for the rest of the model in a Simulink RTW target. Inputs to and outputs from Stateflow blocks in a library model can only be resolved when a Stateflow target is combined with the Simulink target during building. Using Stateflow RTW targets lets you specify target information for the Stateflow blocks in a library model before the model is used in building an application.

- Custom targets — A Stateflow custom target (named anything but `sfun` or `rtw`) is a convenience for collecting the generated code for the Stateflow charts used in a model. After you collect the code, you can use this code at your own discretion in building your own applications.

How Do You Build a Target?

This section gives an overview to the entire process of building a target for models with Stateflow blocks. Use the following steps to find the right procedure for building a target for your model:

1 Add a Stateflow target, if necessary.

You need to add a Stateflow target to the model for any of the following cases:

- You are configuring the Simulink RTW target for a library model with Stateflow blocks.

In this case, you need to add an RTW target (named `rtw`) in Stateflow or the **Model Explorer**.

- You want to generate code from the Stateflow blocks in a model with a custom target.

In this case, you need to add a custom target (named anything but `sfun` or `rtw`) to the model in the **Model Explorer**.

See “Adding a Stateflow Target” on page 12-8 for instructions on adding a Stateflow target to the model.

2 Configure the target.

See one of the following sections for the target you build:

- “Configuring a Simulation Target for Stateflow” on page 12-11
- “Configuring Real-Time Workshop for Stateflow” on page 12-14
- “Configuring a Custom Target in Stateflow” on page 12-24

3 Include your own custom code in the target.

This is actually a part of step 2 that requires special attention. You can include your own custom C code in the target when you configure it. See “Integrating Custom Code with Stateflow Targets” on page 12-29 for details.

4 Configure the target for the rest of the Simulink model.

The preceding steps configure a target only for the contribution from the Stateflow blocks in a model. You configure the rest of the model in the Simulink **Configuration Parameters** dialog. See the “The Configuration Parameters Dialog Box” in Simulink documentation.

5 Start the build process.

Stateflow automatically builds or rebuilds simulation targets when you initiate simulation of the Stateflow machine. You must explicitly initiate the build process for other types of targets. See “Starting the Build” on page 12-36 for more information.

How Does Stateflow Build into Targets?

This topic give you information on how Stateflow blocks contribute to the building of targets in the following steps:

- 1** Stateflow parses the charts in the model to ensure that their logic is valid.
- 2** If any errors are found, Stateflow displays the errors in the Build window and halts. See “Parsing Stateflow Diagrams” on page 12-38 for more details.
- 3** If the charts parse without error, Stateflow Coder generates C code from the charts.

The code generator accepts various options that control the code generation process. You specify these options when you configure Stateflow for targets.

- 4** Stateflow Coder generates a makefile to build the generated source code into an executable program.

The generated makefile can optionally build custom code that you specify into the target. See “Integrating Custom Code with Stateflow Targets” on page 12-29 for details.

- 5** The specified C compiler for MATLAB and a make utility build the code into an application for the target.

Building Simulink targets requires a C compiler that is supported by MATLAB. The Microsoft Windows version of Stateflow comes with a C compiler (`lcc.exe`) and a make utility (`lccmake`). Both tools are installed in the directory `matlabroot\sys\lcc`. If you do not configure MATLAB to use any other compiler, Stateflow uses `lcc` to build targets. For details on setting up your own C compiler, see “Setting Up Your Own Target Compiler” on page 1-21

Adding a Stateflow Target

You need to specify information for generating code and building Stateflow charts into the RTW target for the model. For nonlibrary models, you specify this information directly for the RTW target in the **Configuration Parameters** dialog. For library models you specify this information in a Stateflow RTW target. Later, when you build the RTW target for the model, this information is handed off to the model target and resolved there.

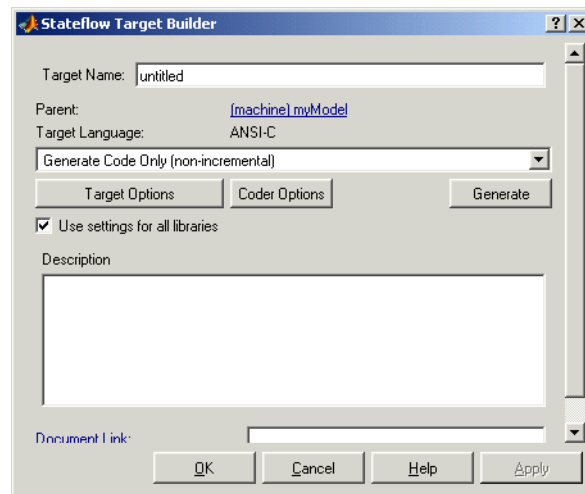
You also need a Stateflow target when you want to use the generated code for the Stateflow blocks in a model for your own use. You collect the generated code from the Stateflow blocks in a model with a Stateflow custom target.

When you create a Simulink model, a Stateflow simulation target named `sfun` is added to it by default. By default, Stateflow RTW and custom targets are not created with the model. You need to add them to the model. You create an RTW target when you add a target with the name `rtw` to the Simulink model. You create a custom target when you add a target with a name other than `sfun` or `rtw` to the Simulink model.

To create a Stateflow RTW target for a library model, or a custom target for any model, do the following:

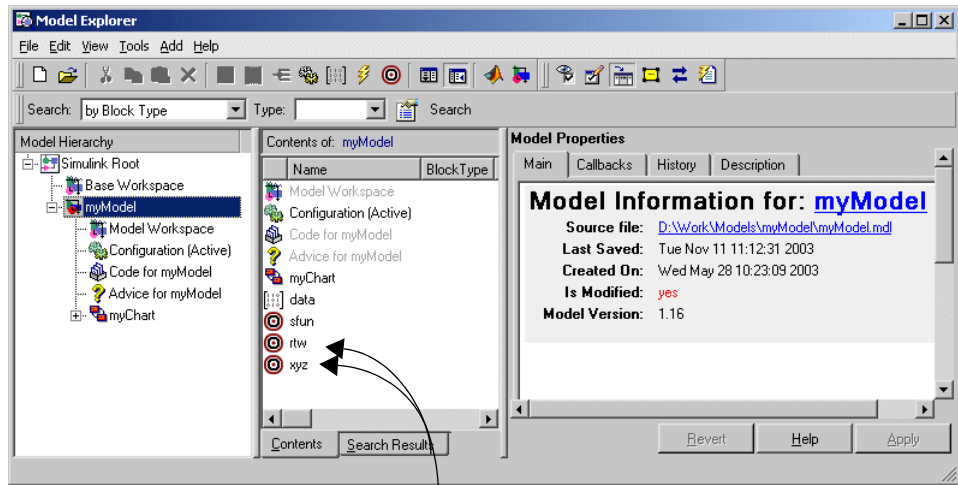
- 1 In the Stateflow diagram editor, from the **Add** menu, select **Target**.

The **Stateflow Target Builder** dialog appears, as shown.



- 2** In the **Target: Name** field, enter the name for the target.
 - To create a Real-Time Workshop (RTW) target, enter `rtw`.
 - To create a custom target, enter the name of your choice other than `sfun` or `rtw`.
- 3** Select **OK** to close the **Stateflow Target Builder** dialog and create the target.

The targets you add are displayed in the **Model Explorer** as members of the model object, as shown.



New RTW (rtw) and custom (xyz) targets

You can also create Stateflow RTW and custom targets in the **Model Explorer** by selecting **Stateflow Target** from the **Add** menu. See “Adding a Target in the Model Explorer” on page 14-6 for details.

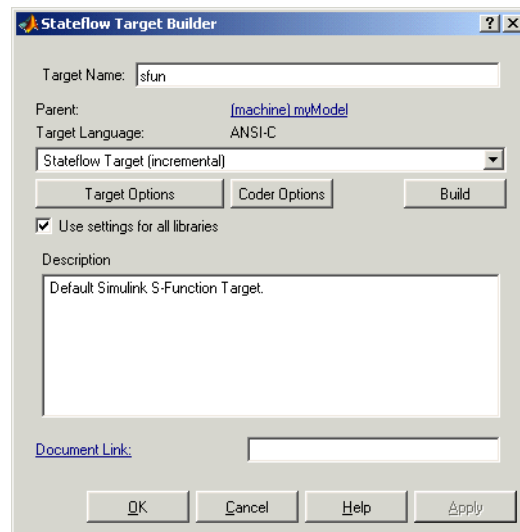
Configuring a Simulation Target for Stateflow

You can configure your model to simulate the Stateflow blocks in your model by configuring the Stateflow simulation target (sfun). Any model that contains one or more Stateflow blocks has a Stateflow simulation target. When you configure this target and build the model for simulation, this target configuration is carried into the simulation configuration for the entire model.

To configure the Stateflow simulation target, do the following:

- 1 From the Stateflow diagram editor **Tools** menu, select **Open Simulation Target**.

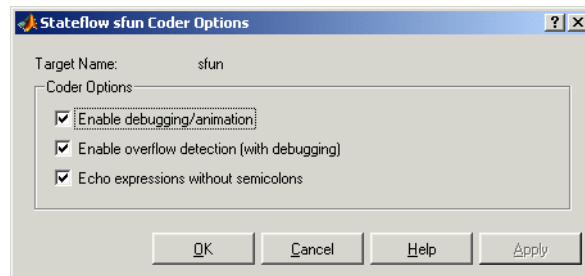
The **Stateflow Target Builder** dialog appears.



- 2 Select one of the following build options:
 - **Stateflow Target (incremental)** to rebuild only those portions of the target corresponding to charts that have changed logically since the last build.
 - **Rebuild All (including libraries)** to rebuild the target, including chart libraries, from scratch.

- **Make without generating code** to invoke the make process without generating code. This is useful when you have custom source files that need to be recompiled within a Stateflow incremental build mechanism that does not detect changes in custom software files.
- 3 To specify code generation options for a simulation target, select **Coder Options**.

The **Simulation Coder Options** dialog box appears.



- 4 Select **OK** or **Apply** for any or all of the following options:
- **Enable debugging/animation** — Enables chart animation and debugging. Stateflow enables debugging code generation when you use the debugger to start a model simulation. You can enable or disable chart animation separately in the debugger. (The Stateflow debugger does not work with stand-alone and RTW targets. Therefore, Stateflow and Real-Time Workshop do not generate debugging/animation code for these targets, even if this option is enabled.)
 - **Enable overflow detection (with debugging)** — Overflow occurs for data when a value is assigned to it that exceeds the numeric capacity of its type. If this check box is selected, Stateflow generates code for overflow detection of Stateflow data. If cleared, no code is generated for overflow detection.

The **Enable overflow detection (with debugging)** option is particularly important for fixed-point data. See “Overflow Detection for Fixed-Point Types” on page 8-11.

Note To actually detect overflow in data during simulation, you must also select the **Data Range** check box in the Debugger window. See “Debugging Data Range Violations” on page 13-18 for more details.

- **Echo expressions without semicolons** — Display run-time output in the MATLAB Command Window, specifically actions that are not terminated by a semicolon.
- 5 Select (check) the **Use settings for all libraries** option if you want the settings that you specify for this target to apply to all the Stateflow charts contributed by library models as well.
 - 6 To specify custom code options for the simulation target, select **Target Options**.

See “Integrating Custom Code with Stateflow Targets” on page 12-29 for details on using the simulation target to integrate custom code with generated code for the Stateflow diagrams in the model.
 - 7 To finish configuring the simulation target, do one of the following:
 - Click **Apply** to apply the selected options
 - Click **OK** to apply the options and close the dialog
 - Click **Build** to build the simulation target.

Note Use the **Chart Properties** dialog to tell the simulation target builder to recognize C bitwise operators (~, &, |, ^, >>, and so on) in action language statements and encode them as C bitwise operations.

Configuring Real-Time Workshop for Stateflow

If you have a Real-Time Workshop (RTW) license, you can configure the Stateflow blocks in your model to generate code for an RTW target. The RTW configuration you use depends on whether your model is a normal nonlibrary model or a library model. Use the following topics to specify the RTW configuration you use for the Stateflow blocks in your model:

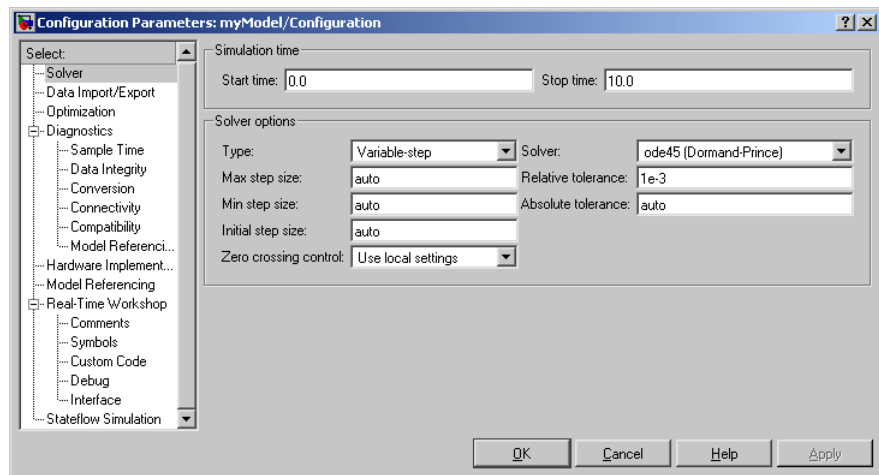
- “Configuring Stateflow Blocks in Nonlibrary Models for Real-Time Workshop” on page 12-14 — For a nonlibrary model, specify the RTW configuration for Stateflow in the Simulink **Configuration Parameters** dialog.
- “Configuring the Stateflow Blocks in Library Models for Real-Time Workshop” on page 12-17 — For a library model, specify the RTW configuration for Stateflow by configuring an RTW target in Stateflow.

Configuring Stateflow Blocks in Nonlibrary Models for Real-Time Workshop

For nonlibrary models, you configure the Stateflow blocks in your model to generate code for the RTW target in Simulink in the **Configuration Parameters** dialog with the following steps:

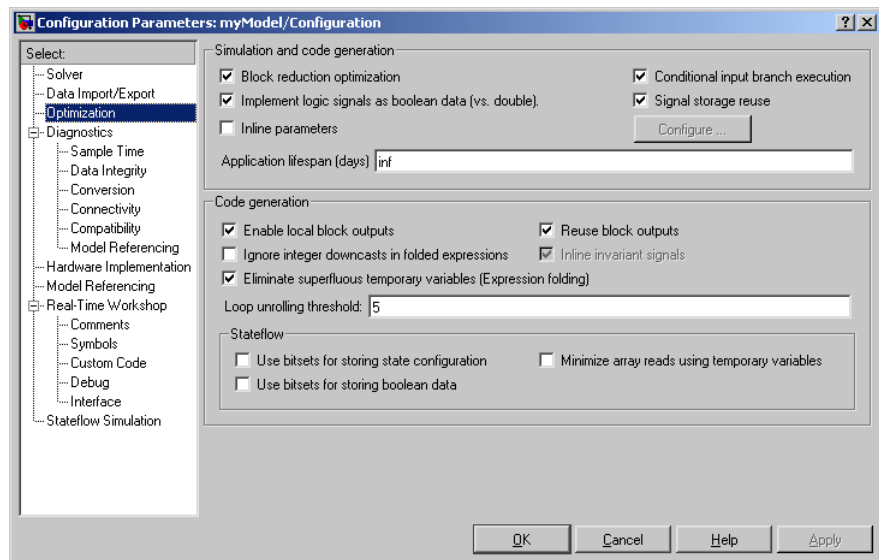
- 1 In a Simulink window or a Stateflow diagram editor, from the Simulation menu, select Configuration Parameters.

The Configuration Parameters dialog appears, as shown.



2 Select the Optimization node, as shown.

The Optimization configuration settings appear in the right pane, as shown.



3 In the right pane, in the **Stateflow** section, select from the following options:

- **Use bitsets for storing state configuration** — Enabling this option specifies that bitsets be used for storing state configuration variables. This can significantly reduce the amount of memory required to store the variables. However, it can increase the amount of memory required to store target code if the target processor does not include instructions for manipulating bitsets.
- **Use bitsets for storing boolean data** — Enabling this option specifies that bitsets be used for storing Boolean data. This can significantly reduce the amount of memory required to store Boolean variables. However, it can increase the amount of memory required to store target code if the target processor does not include instructions for manipulating bitsets.
- **Minimize array reads using temporary variables** — In certain microprocessors, global array read operations are more expensive than accessing a temporary variable on stack. Using this option minimizes array reads by using temporary variables when possible.

For example, the generated code

```
a[i] = foo();
if(a[i]<10 && a[i]>1) {
    y = a[i]+5;
}else{
z = a[i];
}
```

now becomes

```
a[i] = foo();
temp = a[i];
if(temp<10 && temp>1) {
    y = temp+5;
}else{
    z = temp;
}
```

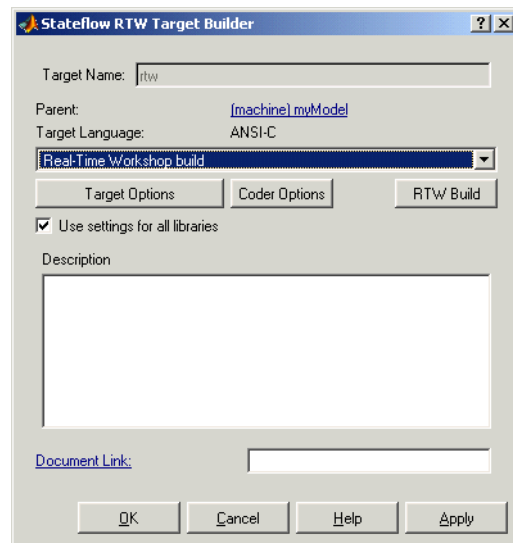
Configuring the Stateflow Blocks in Library Models for Real-Time Workshop

For library models you configure the Stateflow blocks in your model for Real-Time Workshop by configuring a Stateflow RTW target. When the model is built, the configuration for this target is carried into the Simulink RTW configuration and resolved under actual Simulink conditions.

To configure an existing Stateflow RTW target for a library model, do the following:

- 1 From the Stateflow diagram editor **Tools** menu, select **Open RTW Target**.

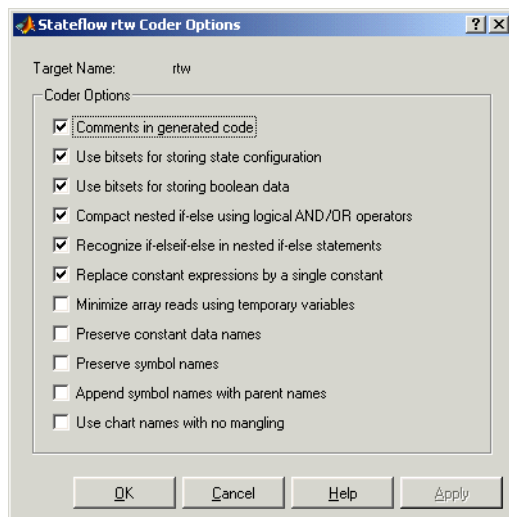
The **Stateflow RTW Target Builder** dialog appears, as shown.



- 2 Select one of the following build options:
 - **Real-Time Workshop build** — to build or rebuild the RTW target.
 - **Real-Time Workshop options** — to change parameters for your RTW build in Simulink. Once you select this option, the button to the right of this field is renamed **RTW Options**. Selecting this button displays the **Configuration Parameters** dialog for Simulink.
 - **Stateflow Target** — to build only Stateflow code in the RTW target.

- 3 Select **Coder Options** to specify Stateflow code generation options for the RTW target.

The **RTW Coder Options** dialog box appears, as shown.



- 4 Select any or all of the following coder options:
 - **Comments in generated code** — Include comments in the generated code.
 - **Use bitsets for storing state configuration** — Use bitsets for storing state configuration variables. This can significantly reduce the amount of memory required to store the variables. However, it can increase the amount of memory required to store target code if the target processor does not include instructions for manipulating bitsets.
 - **Use bitsets for storing boolean data** — Use bitsets for storing Boolean data. This can significantly reduce the amount of memory required to store Boolean variables. However, it can increase the amount of memory required to store target code if the target processor does not include instructions for manipulating bitsets.

- **Compact nested if-else using logical AND/OR operators** — Improves readability of generated code by compacting nested if-else statements using logical AND (&&) and OR (||) operators.

For example, the generated code

```
if(c1) {  
    if(c1) {  
        a1();  
    }  
}
```

now becomes

```
if(c1 && c2) {  
    a1();  
}
```

and the generated code

```
if(c1) {  
    /* fall through to do a1() */  
}else if(c2) {  
    /* fall through to do a1() */  
}else{  
    /* skip doing a1() */  
    goto label1;  
}  
a1();  
label1:  
    a2();
```

becomes

```
if(c1 || c2) {  
    a1();  
}  
a2();
```

- **Recognize if-elseif-else in nested if-else statements** — Improves readability of generated code by recognizing and emitting an if-elseif-else construct in place of deeply nested if-else statements.

For example, the generated code

```
if(c1) {
    a1();
}else{
    if(c2) {
        a2();
    }else{
        if(c3) {
            a3();
        }
    }
}
```

becomes

```
if(c1) {
    a1();
}else if(c2) {
    a2();
}else if(c3) {
    a3();
}
```

- **Replace constant expressions by a single constant** — Improves readability by preevaluating constant expressions and emitting a single constant. This optimization also opens up opportunities for eliminating dead code.

For example, the generated code

```
if(2+3<2) {
    a1();
}else {
    a2(4+5);
}
```

becomes

```
if(0) {
    a1();
}else {
    a2(9);
}
```

in the first phase of this optimization. The second phase eliminates the `if` statement, resulting in simply

```
a2(9);
```

- **Minimize array reads using temporary variables** — In certain microprocessors, global array read operations are more expensive than accessing a temporary variable on stack. Using this option minimizes array reads by using temporary variables when possible.

For example, the generated code

```
a[i] = foo();
if(a[i]<10 && a[i]>1) {
    y = a[i]+5;
}else{
    z = a[i];
}
```

becomes

```
a[i] = foo();
temp = a[i];
if(temp<10 && temp>1) {
    y = temp+5;
}else{
    z = temp;
}
```

- **Preserve constant data names** — Preserve the names of constant data in generated code. This option is useful when the target contains custom code that accesses Stateflow data.
- **Preserve symbol names** — (See note below before using.) Preserve symbol names (names of states and data) in generated code. This option is useful when the target contains custom code that accesses Stateflow data. This option can generate duplicate C symbols if the source chart contains duplicate symbols, for example, two substates with identical names. Enable the next option to avoid duplicate substate names.
- **Append symbol names with parent names** — (See note below before using.) Generate a state or data name by appending the name of the item's parent to the item's name.
- **Use chart names with no mangling** — (See note below before using.) Preserve the names of chart entry functions so that they can be invoked by user-written C code.

Note When you use the options **Preserve constant data names**, **Preserve symbol names**, **Append symbol names with parent names**, or **Use chart names with no mangling**, the names of symbols in generated code are not mangled to make them unique. Because these options do not check for symbol conflicts in generated code, use them only when your symbol names are unique within the model. Conflicts in generated names cause errors such as variable aliasing and compilation errors.

- 5 Select (check) the **Use settings for all libraries** option if you want the settings that you specify for this target to apply to all the Stateflow charts contributed by library models as well.
- 6 To integrate custom code with Stateflow generated code for the RTW target, select **Target Options**.

See “Integrating Custom Code with Stateflow Targets” on page 12-29 for details on using the RTW target to integrate custom code with generated code for the Stateflow diagrams in the model.

- 7 To finish configuring the simulation target, do one of the following:

- Click **Apply** to apply the selected options.
- Click **OK** to apply the options and close the dialog.
- Click **Build** to build the RTW target.

This creates a new RTW target if one did not previously exist.


Note To tell Stateflow to recognize C bitwise operators (~, &, |, ^, >>, and so on) in action language statements and encode them as C bitwise operations, select the **Enable C-bit operations** property in the **Chart Properties** dialog.

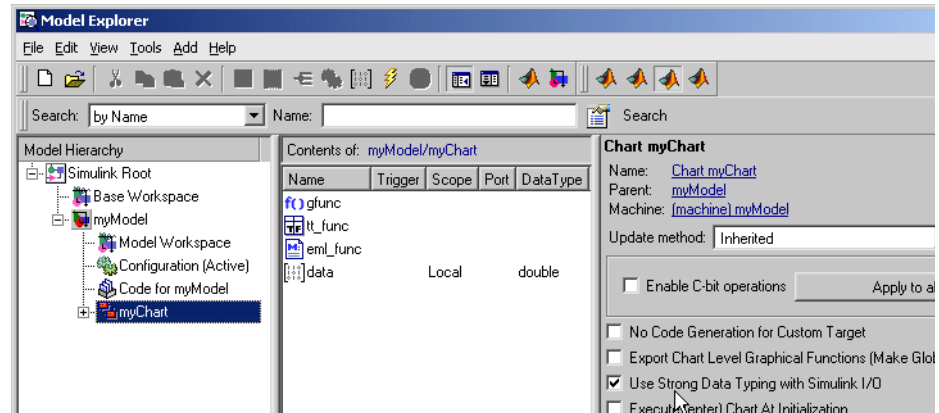
Custom code options are selected through the **Target Options** button. To specify custom code options for your rtw target, see “Integrating Custom Code with Stateflow Targets” on page 12-29.

Note Configuring an RTW target requires additional steps. See Real-Time Workshop documentation for more information.

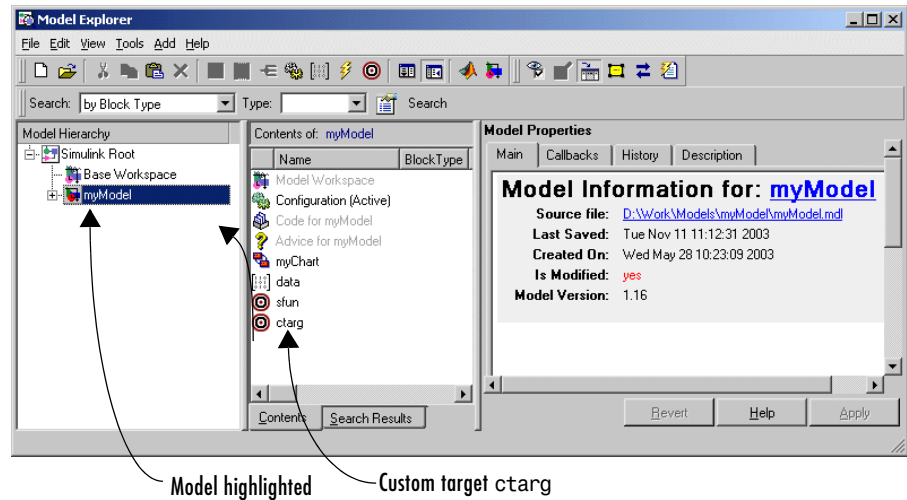
Configuring a Custom Target in Stateflow

A Stateflow custom target (named anything but `sfun` or `rtw`) is a convenience for collecting the generated code for the Stateflow charts in a model. After you collect the code, you can use this code at your own discretion in building your own applications. You configure a custom target in the **Model Explorer** with the following procedure:

- 1 From the Stateflow diagram editor toolbar, select Explore .
- 2 The **Model Explorer** appears with the Stateflow chart highlighted in the **Model Hierarchy** pane.



- 3 In the **Model Explorer**, in the left **Model Hierarchy** pane, select the Simulink model with the custom target.



Model highlighted

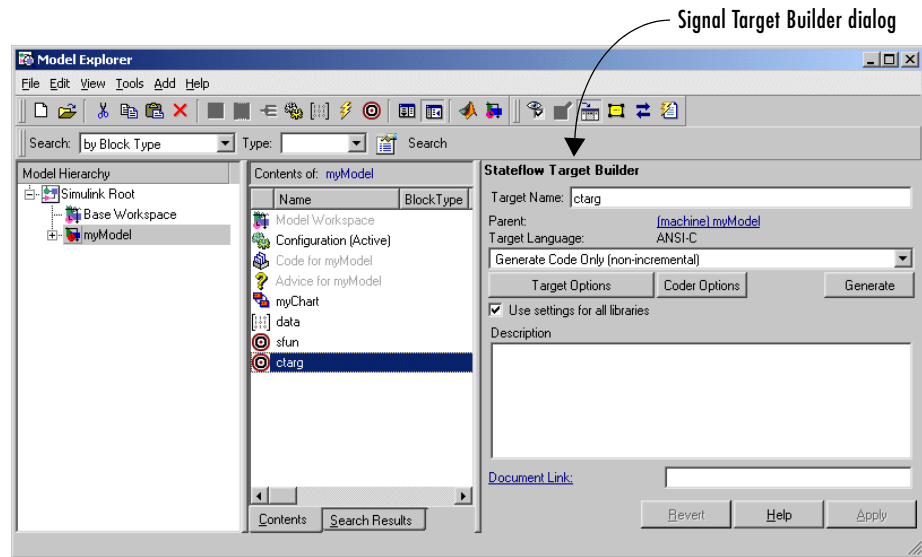
Custom target ctarg

The custom target (in this example, ctarg) appears as a child of the Simulink model. Custom targets have names other than sfun or rtw.

If you need to create a custom target in the **Model Explorer**, see “Adding a Target in the Model Explorer” on page 14-6.

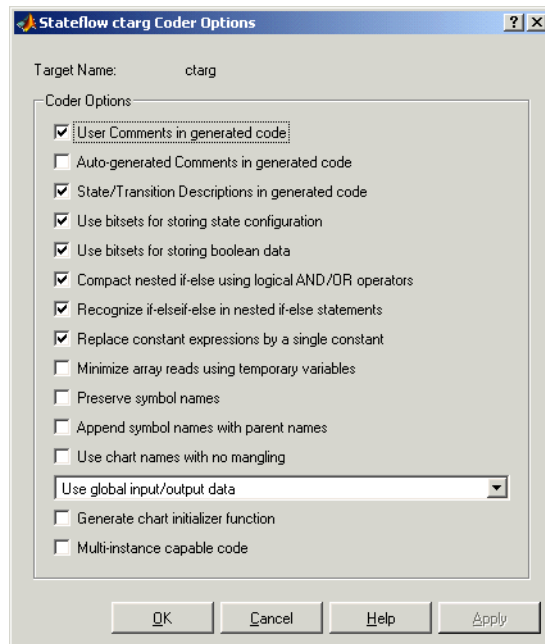
- 4 In the **Contents** pane, click the row for the custom target ctarg.

The **Stateflow Target Builder** dialog appears in the dynamic dialog on the right, as shown.



- 5 In the **Stateflow Target Builder** dynamic dialog, in the long field below the **Target Language** descriptor, select one of the following build options:
 - **Generate code only (non-incremental)** to regenerate code for all charts in the model.
 - **Rebuild All (including libraries)** to rebuild the target, including chart libraries, from scratch.
 - **Make without generating code** to invoke the Make process without generating code. This is useful when you have custom source files that need to be recompiled within a Stateflow incremental build mechanism that does not detect changes in custom software files.
- 6 To specify code generation options for a custom target, select the **Coder Options**.

The **Coder Options** dialog box for the custom target appears as follows:



7 Select any or all of the following options:

The following options are unique to custom targets:

- **I/O Format Options** — Can be any one of the following:

Select **Use global input/output data** to generate chart input and output data as global variables.

Select **Pack input/output data into structures** to generate structures for chart input data and chart output data.

The **Separate argument for input/output data** generates input and output data as separate arguments to a function.

- **Generate chart initializer function** — Generates a function initializer of data.

- **Multiinstance capable code** — Generates multiply instantiable chart objects instead of a static definition.

The following options are also available for RTW targets. See “Configuring Real-Time Workshop for Stateflow” on page 12-14 for a description.

- **Comments in generated code**
- **Use bitsets for storing state configuration**
- **Use bitsets for storing boolean data**
- **Compact nested if-else using logical AND/OR operators**
- **Recognize if-elseif-else in nested if-else statements**
- **Replace constant expressions by a single constant**
- **Minimize array reads using temporary variables**
- **Preserve symbol names**
- **Append symbol names with parent names**
- **Use chart names with no mangling**

Custom code options are selected through the **Target Options** button. To specify custom code options for your rtw target, see “Integrating Custom Code with Stateflow Targets” on page 12-29.

- 8** Select (check) the **Use settings for all libraries** option if you want the settings that you specify for this target to apply to all the Stateflow charts contributed by library models as well, s.
- 9** To specify custom code options for the simulation target, select **Target Options**.

See “Integrating Custom Code with Stateflow Targets” on page 12-29 for details on using the custom target to integrate custom code with generated code for the Stateflow diagrams in the model.

- 10** To finish configuring the custom target, do one of the following:
 - Click **Apply** to apply the selected options.
 - Click **OK** to apply the options and close the dialog.
 - Click **Build** to build the RTW target.

Integrating Custom Code with Stateflow Targets

For all Stateflow targets (simulation, RTW, and custom) you can configure custom code options that let you incorporate custom C or C++ code into the target you build for a model. This lets you take advantage of legacy code that augments model capabilities and lets you define and include custom global variables that can be shared by both Stateflow generated code and your custom code.

Integrate custom code with Stateflow targets as described in the following topics:

- “Specifying Custom Code Options for Stateflow Targets” on page 12-29 — Describes the target options you need to set and specify in order to build custom code into your target.
- “Specifying Relative Paths for Custom Code” on page 12-32 — Shows you where Stateflow searches for the custom code files you specify with a relative path.
- “Including Custom C++ Code” on page 12-33 — Shows you how to specify and include C++ custom code with your library model.
- “Calling Graphical Functions from Custom Code” on page 12-35 — Tells you how you can export graphical functions from your Stateflow charts so that they can be called from custom code that is added to your target.

Specifying Custom Code Options for Stateflow Targets

You include custom code in Stateflow simulation, RTW, and custom targets in the tabbed option pages of the Stateflow **Target Options** dialog. Use the following procedure to specify configuration options to build custom code into a Stateflow target:

- 1 Open the **Target Builder** dialog for the Stateflow simulation, RTW, or custom target as described in the following sections:

You do this when you configure one of the targets as described in the following topics:

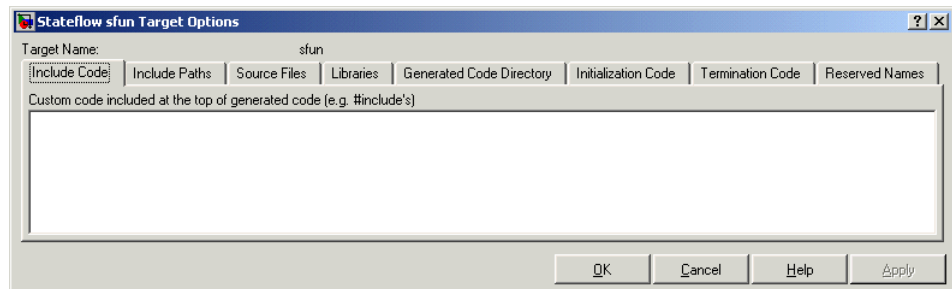
- “Configuring a Simulation Target for Stateflow” on page 12-11

- “Configuring the Stateflow Blocks in Library Models for Real-Time Workshop” on page 12-17
- “Configuring a Custom Target in Stateflow” on page 12-24

You also access the **Target Builder** dialog for an RTW or custom target when you first create it. See “Adding a Stateflow Target” on page 12-8 for instructions.

- 2 In the **Target Builder** dialog, select the **Target Options** button.

The **Target Options** dialog appears, as shown:



- 3 Specify your custom code in the edit boxes of the following tabbed pages of the **Target Options** dialog:

- **Include Code** — Enter code lines (for example, `#include test.h`) to include at the top of a generated header file that declares custom functions and data used by the generated code. These code lines are included at the top of all generated source code files and are visible to all generated code.

Since the code specified in this option is included in multiple source files that are linked into a single binary, there are some limitations on what you can and cannot include. For example, you should not include a global variable definition such as `int x;` or a function body such as

```
void myfun(void)
{
...
}
```

These code lines cause linking errors because their symbols are defined multiple times in the source files of the generated code. You can, however, include extern declarations of variables or functions such as `extern int x;` or `extern void myfun(void);`.

- **Include Paths** — Enter a space-separated list of the directory paths that contain custom header files to be included either directly (see Include Code option) or indirectly in the compiled target. See “Specifying Relative Paths for Custom Code” on page 12-32 for instructions on entering directory path names.
- **Source Files** — Enter a list of source files to be compiled and linked into the target. You can separate source files with either a comma, a space, or a new line. See “Specifying Relative Paths for Custom Code” on page 12-32 for instructions on entering directory path names.
- **Libraries** — Enter a space-separated list of static libraries containing custom object code to be linked into the target. See “Specifying Relative Paths for Custom Code” on page 12-32 for instructions on entering directory path names.
- **Generated Code Directory** — For custom targets you can specify an optional directory to receive the generated code.
- **Initialization Code** — Code statements that are executed once at the start of simulation. You can use this initialization code to invoke functions that allocate memory or perform other initializations of your custom code. This option does not apply to custom targets.
- **Termination Code** — Enter code statements that are executed at the end of simulation. You can use this code to invoke functions that free memory allocated by the custom code or perform other cleanup tasks. This option does not apply to custom targets.
- **Reserved Names** — Sometimes the names of variables and functions in Stateflow generated code matches the names of variables or functions specified in custom code. When this occurs, the compiler reports a “redeclaration” or “redefinition” of the variable or function name. When you encounter an error like this, enter the name that causes this conflict. Stateflow changes the name in its generated code.

- 4 Click **Apply** to apply the specification to the target or **OK** to apply the specifications and close the dialog.

If you make a change in one of your custom code options, to force the target rebuild to incorporate your changes you must either change one of the charts slightly (this forces a rebuild when you simulate again) or go to the **Simulation Target Builder** dialog box and select the **Rebuild All** option.

References

For additional information on specifying custom code for your target, see the following online articles:

- “Integrating Custom C Code Using Stateflow 2.0” by V. Raghavan
- “Automatic Code Generation from Stateflow for Palm OS Handhelds: a Tutorial” by D. Maclay

Specifying Relative Paths for Custom Code

You specify custom code options for Stateflow targets in “Specifying Custom Code Options for Stateflow Targets” on page 12-29. If you specify paths and files with absolute paths and later move them, you have to change these paths to point to new locations. Because of this, it is recommended that you use relative paths for custom code options that specify paths or files.

If the entered files or paths are specified with a relative path, Stateflow searches paths relative to the following directories for them:

- The current directory
- The model directory (if different from the current directory)
- The list of directories specified for the **Include Path** option
- All the directories on MATLAB’s search path, excluding the toolbox directories

You can use the forward slash (/) or backward slash (\) as a file separator regardless of whether you are on a UNIX or PC platform. The makefile generator parses these strings and returns the path names with the correct platform-specific file separators.

Paths can also contain `$...$` enclosed tokens that are evaluated in the MATLAB workspace. For example, the entry `$mydir1$dir1`, where `mydir1` is a string variable defined in the MATLAB workspace as `'d:\work\source\module1'`, declares the directory `d:\work\source\module1\dir1` as a custom include path.

Including Custom C++ Code

You specify custom C code for Stateflow targets as described in “Specifying Custom Code Options for Stateflow Targets” on page 12-29. Use the following procedure to include custom C++ code with a target:

- 1 Add a C function wrapper to your existing custom code. This wrapper function executes the C++ code that you are including.

The C function wrapper should have the following form:

```
int my_c_function_wrapper()
{
    .
    .
    .
    //C++ code
    .
    .
    .
    return result;
}
```

- 2 Create a header file that prototypes the C function wrapper in the previous step.

The header file should have the following form:

```
int my_c_function_wrapper();
```

The value `_cplusplus` is defined if your compiler supports C++ code. The `extern "C"` wrapper specifies C linkage with no name mangling.

- 3 Configure the Stateflow simulation target. You can access this dialog as follows:

- a From the Stateflow chart, select **Tools -> Open Simulation Target** and click the **Target Options** button.
 - b Add the header file to the **Include Code** tab. Click **Apply**.
 - c Add the custom C++ files to the **Include Paths** or **Source Files** tabs. Click **Apply**.
 - d If you need to make additional configurations in this dialog, do so now. Click **OK** when you are done.
- 4 Select C++ in the **Language** option of the Real-Time Workshop configuration dialog. You can access this dialog as follows:
- a From the model, select **Simulation -> Configuration Parameters**.
 - b Select the **Real-Time Workshop** pane.
The general Real-Time Workshop pane is displayed.
 - c Select C++ from the **Language** pull-down menu. Click **Apply**.
 - d If you need to make additional configurations in this dialog, do so now. Click **OK** when you are done.

The preceding procedure describes how to include custom C++ code with a target for a simulation. If you also want to generate code with Real-Time Workshop, use the following procedure to duplicate the Stateflow chart header file and C++ file entries for Real-Time Workshop:

- 1 Be sure that you have performed the preceding procedure.
- 2 Configure the Real-Time Workshop configuration set. You can access this dialog as follows:
 - a From the model, select **Simulation -> Configuration Parameters**.
 - b Select the **Custom Code** subpane under the **Real-Time Workshop** pane.
 - c Add the header file to the **Header File** option. Click **Apply**.
 - d Add the custom C++ files to the **Include directories** or **Source files** option. Click **Apply**.

- e If you need to make additional configurations in this dialog, do so now. Click **OK** when you are done.

You can now generate code for the model.

Calling Graphical Functions from Custom Code

You specify custom C code for Stateflow targets as described in “Specifying Custom Code Options for Stateflow Targets” on page 12-29. Use the following procedure to call a graphical function from custom C code:

- 1 Create the graphical function at the root level of the chart that defines the function (see “Creating a Function” on page 5-29).
- 2 Export the function from the chart that defines the function (see “Exporting Functions” on page 5-41).

This option implicitly forces the chart and function names to be preserved.

- 3 Include the generated header file `chart_name.h` at the top of your custom code, where `chart_name` is the name of the chart that contains the graphical function.

The chart header file contains the prototypes for the graphical functions that the chart defines.

Starting the Build

Once you have completely configured a Stateflow simulation or RTW target, you want to start the build process to generate code from your model and compile it. This section describes how to start a build for a simulation target or Real-Time Workshop target in Stateflow with the following topics:

- “Starting a Simulation Target Build” on page 12-36 — Tells you how to start a build for the simulation target (sfun).
- “Starting an RTW Target Build” on page 12-37 — Tells you how to start a build for the RTW target (rtw).

Starting a Simulation Target Build

You can start a target build for a Stateflow simulation target (sfun) in one of the following ways:

- Select **Start** from the Stateflow or Simulink editor’s **Simulation** menu
Automatically builds and runs a simulation target.
- Select **Debug** from the Stateflow editor’s **Tools** menu
Automatically builds and runs a simulation target. This is equivalent to the previous method.
- Select the **Build** button on the **Simulation Target Builder** dialog box for the target
Use this option if you want to build a simulation target without running it. You would typically want to do this to ensure that Stateflow can build a target containing custom code.

Using the target builder to launch the build allows you to choose between an incremental build, a full build, and a build without code generation. See “Configuring a Simulation Target for Stateflow” on page 12-11 for more information.

Starting an RTW Target Build

You can start a target build for a Stateflow Real-Time Workshop (rtw) target in one of the following ways:

- By selecting the **Build RTW** button on the **RTW Target Builder** dialog box for the target

You must use this option to build stand-alone targets. Using the target builder to launch the build allows you to choose between full build or rebuild and a build of Stateflow code only. See “Configuring Real-Time Workshop for Stateflow” on page 12-14 for more information.

- By selecting the **Build** button on the **Real-time Workshop** panel of the **Simulation Parameters** dialog box in Simulink.

Select **Real-Time Workshop** options on the **RTW Target Builder** dialog box to access the **Simulation Parameters** dialog box of Simulink. See “Configuring Real-Time Workshop for Stateflow” on page 12-14 for more information.

Parsing Stateflow Diagrams

When you begin a build for a target as described in “Starting the Build” on page 12-36, the parser evaluates the graphical and nongraphical objects in each Stateflow machine against the supported Stateflow notation and the action language syntax. This section describes the Stateflow parser and how its messages appear with the following topics:

- “Calling the Stateflow Parser” on page 12-38 — Shows you how to call the Stateflow Parser to parse your current diagram at any time.
- “Parser Error Checking” on page 12-39 — Lists the types of errors that Stateflow parses the diagram for.
- “Parsing Diagram Example” on page 12-40 — Gives you an example of parsing an example Stateflow diagram with a parsing error.

Calling the Stateflow Parser

Apart from building a target, you can call the Stateflow parser to check the syntax of your Stateflow diagrams in one of the following ways:

- Parse an individual Stateflow diagram in the Stateflow diagram editor by selecting **Parse Diagram** from the **Tools** menu.
- Parse a Stateflow machine, that is, all the Stateflow charts in a model, by selecting **Parse** from the **Tools** menu in the Stateflow diagram editor.
- When you simulate a model, build a target, or generate code, the Stateflow machine is automatically parsed.

In all cases, the **Stateflow Builder** window displays when parsing is complete. If parsing is unsuccessful (that is, an error is detected), the Stateflow diagram editor automatically appears with the highlighted object causing the first parse error. In the **Stateflow Builder** window, each error is displayed with a leading red button icon. You can double-click any error in this window to bring its source Stateflow diagram to the front with the source object highlighted. See “Parsing Diagram Example” on page 12-40 for example displays of parsing results in the Stateflow Builder window.

Note Parsing informational messages are also displayed in the MATLAB Command Window.

Parser Error Checking

The parser evaluates the graphical and nongraphical objects in each Stateflow machine against the supported Stateflow notation and the action language syntax. Errors are displayed in informational pop-up windows. See “Parsing Stateflow Diagrams” on page 12-38 for more information.

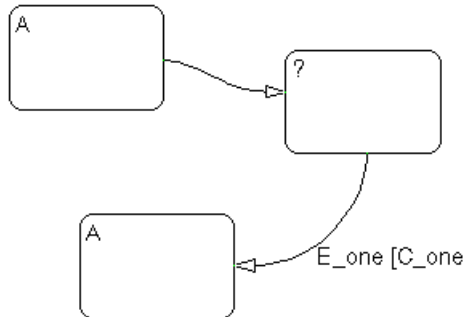
Some aspects of the notation are verified at run-time. Others are verified during application run-time. Using the Debugger, you can detect the following run-time errors during simulation:

- **State Inconsistency** — Most commonly caused by the omission of a default transition to a substate in superstates with XOR decomposition. See “Debugging State Inconsistencies” on page 13-14.
- **Transition Conflict** — Occurs when there are two equally valid transition paths from the same source. See “Debugging Data Range Violations” on page 13-18.
- **Data Range Violation** — Occurs when minimum and maximum values specified for a data in its properties dialog are exceeded or when fixed-point data overflows its base word size. See “Debugging Data Range Violations” on page 13-18.
- **Cyclical Behavior** — Occurs when a step or sequence of steps repeats itself indefinitely. See “Debugging Cyclic Behavior” on page 13-20.

You can modify the notation to resolve run-time errors. See Chapter 13, “Debugging and Testing,” for more information on debugging run-time errors.

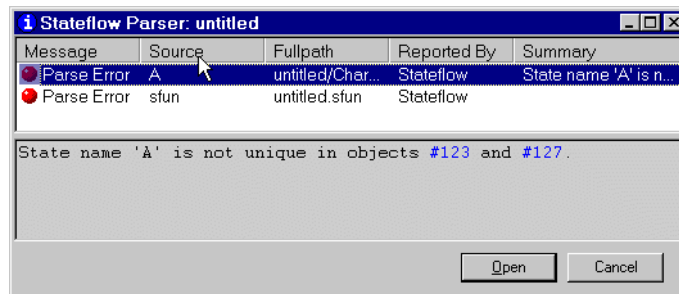
Parsing Diagram Example

For the following Stateflow diagram, the steps that follow describe the parsing process and its reported results.



- 1 Parse the Stateflow diagram.

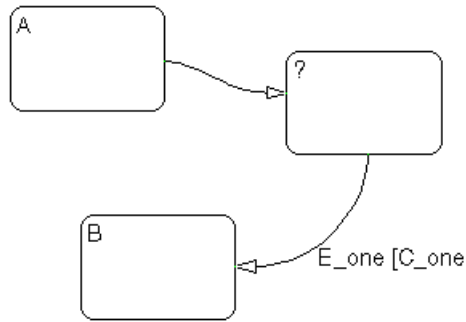
Choose **Parse Diagram** from the graphics editor **Tools** menu to parse the Stateflow diagram. State A in the upper left corner is selected and this message is displayed in the pop-up window and the MATLAB Command Window.



- 2 Fix the parse error.

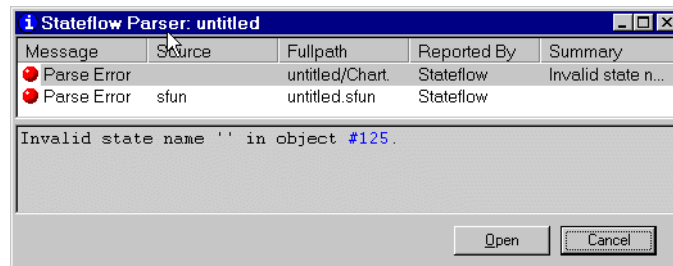
In this example, there are two states with the name A. Edit the Stateflow diagram and label the duplicate state with the text B.

The Stateflow diagram should look similar to this.



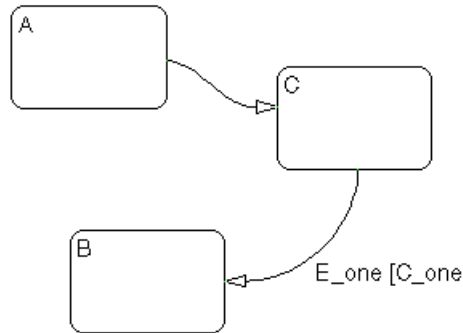
3 Reparse.

Choose **Parse Diagram** from the graphics editor **Tools** menu. This message displays in the pop-up menu and the MATLAB Command Window.



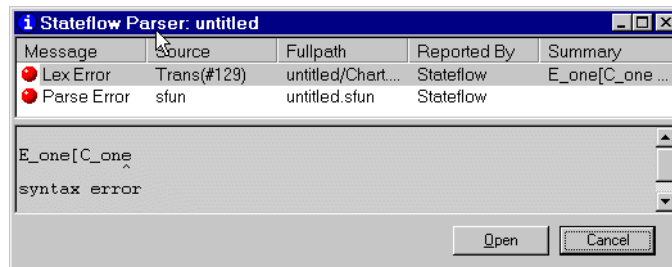
4 Fix the parse error.

In this example, the state with the question mark needs to be labeled with at least a state name. Edit the Stateflow diagram and label the state with the text C. The Stateflow diagram should look similar to this.



5 Reparse.

Choose **Parse Diagram** from the graphics editor **Tools** menu. This message is displayed in the pop-up window and the MATLAB Command Window.

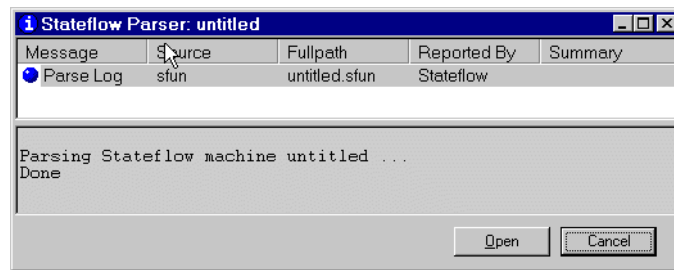


6 Fix the parse error.

In this example, the transition label contains a syntax error. The closing bracket of the condition is missing. Edit the Stateflow diagram and add the closing bracket so that the label is `E_one [C_one]`.

7 Reparse.

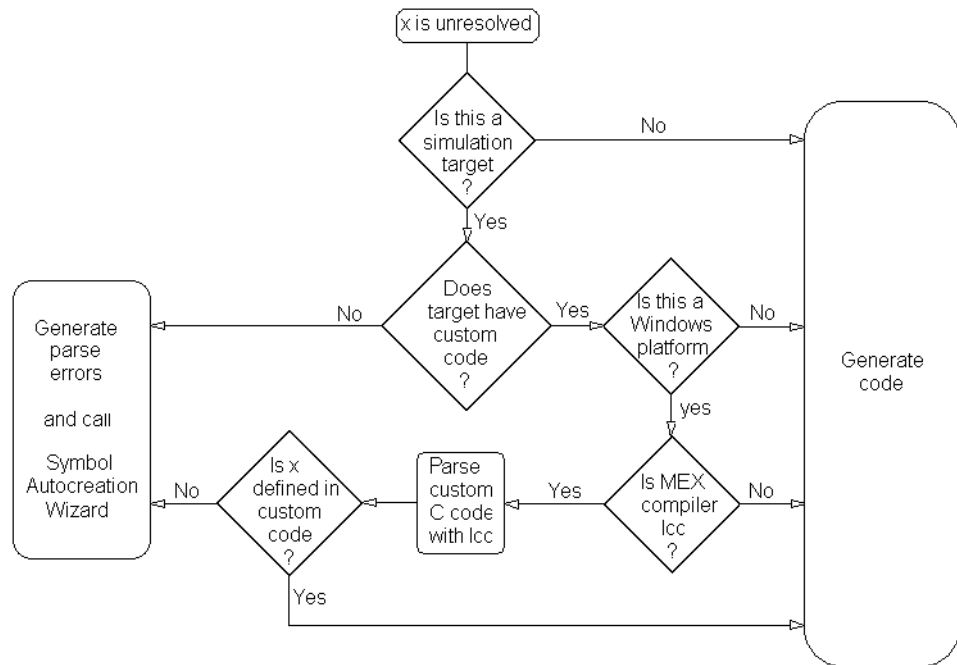
Choose **Parse Diagram** from the graphics editor **Tools** menu. This message is displayed in the pop-up window and the MATLAB Command Window.



The Stateflow diagram now has no parse errors.

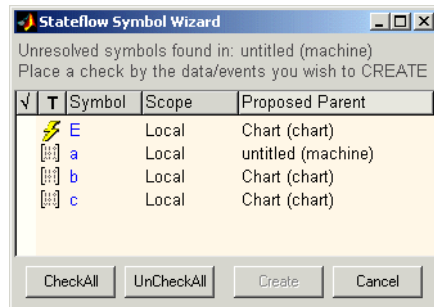
Resolving Event, Data, and Function Symbols

When you simulate a model, build a target, or generate code for a target, the Stateflow machine is automatically parsed (see “Parsing Stateflow Diagrams” on page 12-38). During that time, if Stateflow finds that your diagram does not resolve some of its symbols, it uses the following process to determine whether to report parse errors for the unresolved symbols or continue generating code:



Symbol Autocreation Wizard

The Symbol Autocreation Wizard helps you to add missing data and events to your Stateflow charts. When you parse or simulate a diagram, the Wizard detects references to data and events that are not already defined in the Stateflow Explorer and opens with a list of the recommended data or events that you need to define.



To accept, reject, or change a recommended item, do the following:

- To accept an item, select the space in front of the item under the check mark column.
To accept all items, select the **CheckAll** button.
- To reject an item, leave it unchecked.
- To change an item, select the value under the **T** (type), **Scope**, or **Proposed Parent** column for that item.

Each time you select the value, the Wizard replaces the entry with a different value. Keep selecting until the desired value appears.

When you are satisfied with the proposed symbol definitions, click the Wizard's **Create** button to add the symbols to Stateflow's data dictionary.

Error Messages

When building a target or parsing a diagram, you might see error messages from any of the following sources: the parser, the code generator, or external build tools (make utility, C compiler, linker). Stateflow displays errors in a dialog box and in the MATLAB Command Window. Double-clicking a message in the error dialog zooms the source Stateflow diagram to the object that caused the error.

This section contains the following topics:

- “Parser Error Messages” on page 12-46 — Lists some of the messages you can receive during parsing of your Stateflow diagram.
- “Code Generation Error Messages” on page 12-47 — Lists some of the messages you can receive during code generation for your Stateflow diagram.
- “Compilation Error Messages” on page 12-48 — Explains the difference between compilation error messages that you receive during parsing and code generation messages.

Parser Error Messages

The Stateflow parser flags syntax errors in a state chart. For example, using a backward slash (\) instead of a forward slash (/) to separate the transition action from the condition action generates a general parse error message.

Typical parse error messages include the following:

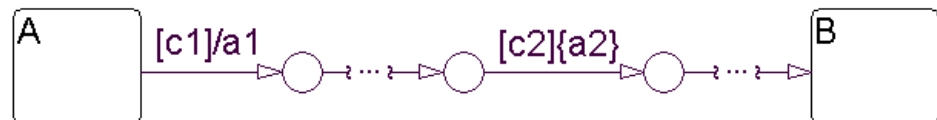
- "Invalid state name xxx" or "Invalid event name yyy" or "Invalid data name zzz"
A state, data, or event name contains a nonalphanumeric character other than underscore.
- "State name xxx is not unique in objects #yyy and #zzz"
Two or more states at the same hierarchy level have the same name.
- "Invalid transition out of AND state xxx (#yy)"
A transition originates from an AND (parallel) state.
- "Invalid intersection between states xxx and yyy"
Neighboring state borders intersect. If the intersection is not apparent, consider the state to be a cornered rectangle instead of a rounded rectangle.

- "Junction #x is sourcing more than one unconditional transition"
More than one unconditional transition originates from a connective junction.
- "Multiple history junctions in the same state #xxx"
A state contains more than one history junction.

Code Generation Error Messages

Typical code generation error messages include the following:

- "Failed to create file: modelName_sf.c"
The code generator does not have permission to generate files in the current directory.
- "Another unconditional transition of higher priority shadows transition # xx"
More than one unconditional inner, default, or outer transition originates from the same source.
- "Default transition cannot end on a state that is not a substate of the originating state."
A transition path starting from a default transition segment in one state completes at a destination state that is not a substate of the original state.
- "Input data xxx on left hand side of an expression in yyy"
A Stateflow expression assigns a value to an **Input from Simulink** data object. By definition, Stateflow cannot change the value of a Simulink input.
- "Transition <number> has a condition action which is preceded by a transition <number> containing a transition action. This is not allowed as it results in out-of-order execution, i.e., the condition action of <number> gets executed before the transition action of <number>."



The preceding Stateflow diagram flags this error. Assuming that there are no other actions than those indicated for the labeled transition segments

between state A and state B, the sequence of execution that takes place when state A is active is expressed by the following pseudocode:

```
    If (c1) {  
        if(c2) {  
            a2;  
            exit A;  
            a1;  
            enter B;  
        }  
    }
```

Because condition actions are evaluated when their guarding condition is true and transition actions are evaluated when the transition is actually taken, condition action a2 is executed prior to transition action a1. This violates the apparent graphical sequence of executing a1 then a2. In this case, the preceding diagram is flagged for an error during build time. As a remedy, the user can change a1 and a2 to be both condition or transition actions.

Compilation Error Messages

If compilation errors indicate the existence of undeclared identifiers, verify that variable expressions in state, condition, and transition actions are defined.

Consider, for example, an action language expression such as $a=b+c$. In addition to entering this expression in the Stateflow diagram, you must create data objects for a, b, and c using the Explorer. If the data objects are not defined, the parser assumes that these unknown variables are defined in the **Custom code** portion of the target (which is included at the beginning of the generated code). This is why the error messages are encountered at compile time and not at code generation time.

Generated Files

All generated files are placed in a subdirectory of the `sfprj` subdirectory of the MATLAB current directory. You set the MATLAB current directory in the **Start in** field for the properties of the MATLAB program icon that you used to start MATLAB. You can change it in MATLAB with a `cd` command as you would in DOS or UNIX.

Note Do not confuse the `sfprj` directory created in the MATLAB current directory for generated files with the `sfprj` directory in the directory containing the model file. The latter `sfprj` directory is used for storing information on the model, while the former stores generated files. However, if the MATLAB current directory is the model directory, the same `sfprj` directory (under the model directory) is used to store both model information and generated files.

This section contains the following topics:

- “DLL Files” on page 12-49 — Explains the origin of `.dll` files that you build as part of building a simulation target.
- “Code Files” on page 12-50 — Describes the code files that you build for your target as part of code generation.
- “Makefiles” on page 12-51 — Describes the makefiles that result when you build your target into an executable.

DLL Files

If you have a Simulink model named `mymodel.mdl`, which contains two Stateflow blocks named `chart1` and `chart2`, this means that you have a machine named `mymodel` that parents two charts named `chart1` and `chart2`.

When you simulate the Stateflow chart for `mymodel.mdl`, you actually generate code for `mymodel.mdl` that is compiled into an S-function DLL file known as a simulation target. On Windows PC platforms this file is named `mymodel_sfunsfun.dll`. On UNIX platforms the DLL file is named `mymodel_sfunsfun.$mexext$` where `$mexext$` is `so12` on Solaris, `mexsg` on SGI, `mex1x` on Linux, and so on.

Code Files

Code files for the Simulation target (sfun) are generated for each model and placed in the subdirectory `sfprj/build/<model>/sfun/src` of the current directory, where `<model>` represents the name of the model.

Note Do not keep any of your custom source files in the `sfprj` subdirectory of your model directory.

The code generated for the simulation target `sfun` is organized into the following files:

- `<model>_sfun.h` is the machine header file. It contains the following:
 - All the defined global variables needed for the generated code
 - Type definition of the Stateflow machine-specific data structure that holds machine-parented local data
 - External declarations of any Stateflow machine-specific global variables and functions
 - Custom code strings specified via the **Target Options** dialog box
- `<model>_sfun.c` is the machine source file. It includes the machine header file and all the chart header files (described below) and contains the following:
 - All the machine-parented event broadcast functions
 - Simulink interface code
- `<model>_sfun_registry.c` is a machine registry file that contains Simulink interface code.
- `<model>_sfun_cn.h` is the chart header file for the chart `chartn`, where $n = 1, 2, 3$, and so on, depending on how many charts your model has (see the following note). This file contains type definitions of the chart-specific data structures that hold chart-parented local data and states.
- `<model>_sfun_cn.c` is the chart source file for `chartn`, where $n = 1, 2, 3$, and so on, depending on how many charts your model has (see the following note). This chart source file includes the machine header file and the corresponding chart header file. It contains the following:

- Chart-parented data initialization code
- Chart execution code (state entry, during, and exit actions, and so on)
- Chart-specific Simulink interface code

Note Every chart is assigned a unique number at creation time by Stateflow. This number is used as a suffix for the chart source and chart header file names for every chart (where $n = 1, 2, 3$, and so on, depending on how many charts your model has).

Makefiles

Makefiles generated for your model are platform and compiler specific. On UNIX platforms, Stateflow generates a gmake-compatible makefile named `mymodel_sfunk.mku` that is used to compile all the generated code into an executable. On PC platforms, an ANSI C compiler-specific makefile is generated based on your C-MEX setup as follows:

- If your installed compiler is Microsoft Visual C++ 4.2, 5.0, or 6.0, the following files are generated:
 - MSVC-compatible makefile named `mymodel_sfunk.mak`
 - Symbol definition file named `mymodel_sfunk.def` (required for building S-function DLLs)
- If your installed compiler is Watcom 10.6 or 11.0, the following file is generated:
 - Watcom-compatible makefile named `mymodel_sfunk.wmk`
- If your installed compiler is Borland 5.0, the following files are generated:
 - Borland-compatible makefile named `mymodel_sfunk.bmk`
 - Symbol definition file named `mymodel_sfunk.def` (required for building S-function DLLs)
- If you choose `lcc-win32`, a bundled ANSI-C compiler shipped with Stateflow, the following file is generated:
 - An `lcc` compatible makefile named `mymodel_sfunk.lmk`

Stateflow Coder also generates another support file needed for the make process, named `<model>_sfunk.mol`.

Debugging and Testing

To ensure that your Stateflow diagrams are behaving as you expect them to, use the **Stateflow Debugging** window to evaluate code coverage and perform dynamic checking during simulation.

Debugging with the Debugging Window (p. 13-2)	Describes the parts of the Stateflow Debugging window during debugging.
Debugging Run-Time Errors Example (p. 13-9)	Shows you how to debug run-time errors in Stateflow diagrams with an actual example model.
Debugging State Inconsistencies (p. 13-14)	Describes how state inconsistencies due to faulty Stateflow notation are detected and debugged.
Debugging Conflicting Transitions (p. 13-16)	Describes how conflicting transitions, transitions that are equally valid during execution, are detected and debugged.
Debugging Data Range Violations (p. 13-18)	Describes how to debug for occurrences of the value of a data object exceeding its maximum value or dropping below its minimum value.
Debugging Cyclic Behavior (p. 13-20)	Shows you how the Stateflow Debugging window detects algorithms that lead to infinite recursions and looping caused by event broadcasts.
Watching Data Values with Debuggers (p. 13-24)	Shows you a variety of ways that you can keep track of the values for Stateflow data and state activity during simulation.
Monitoring Stateflow Test Points (p. 13-30)	Shows you how to specify local data or states as test points that you can plot with a floating scope or log to MATLAB workspace during simulation.
Understanding Model Coverage for Stateflow Charts (p. 13-42)	Describes how the Model Coverage tool determines the extent to which a model test case exercises simulation control flow paths through a model.

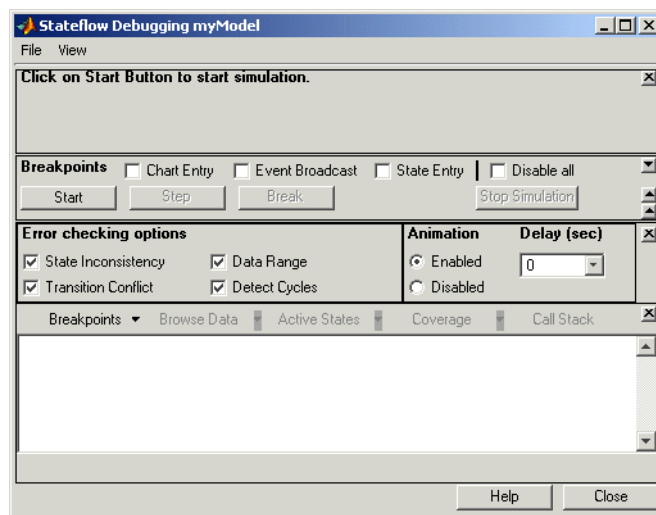
Debugging with the Debugging Window

You use the Stateflow Debugging window to control simulation to make sure that your Stateflow diagrams are behaving exactly as you expect them to.

To open the Stateflow Debugging window,

- 1 From the **Tools** menu, select **Debug**.

The Stateflow Debugging window opens as shown:



This section shows you how to use the Stateflow Debugging window to debug your Stateflow diagrams during simulation in the following topics:

- “Setting Breakpoints for Debugging” on page 13-3 — Shows you how to specify execution points in your target where execution stops for debugging purposes.
- “Setting Error Checking in the Debugging Window” on page 13-5 — Describes the optional error checking available in the Debugging window.
- “Controlling Animation in the Debugging Window” on page 13-7 — Shows you how to activate, deactivate, and control the speed of animation of Stateflow diagrams during simulation.

- “Starting Simulation in the Debugging Window” on page 13-5 — Shows you how to start simulation in the Debugging window and describes the items displayed in the Status Display Area of the **Stateflow Debugging** window when a breakpoint is encountered during simulation.
- “Controlling the Execution Rate in the Debugging Window” on page 13-7 — Describes the control buttons that you can use to control execution during simulation before and after breakpoints are encountered.
- “Setting the Output Display Pane” on page 13-8 — Describes the buttons you use during simulation to display information you use to debug your application.

Setting Breakpoints for Debugging

A breakpoint indicates a point at which the Stateflow Debugging window halts execution of a simulating Stateflow diagram. At this time you can inspect Stateflow and MATLAB workspace data and examine the status of a simulating Stateflow diagram.

The Stateflow Debugging window supports global and local breakpoints. Global breakpoints halt execution on any occurrence of the specific type of breakpoint. Local breakpoints halt execution on a specific object.

Setting Global Breakpoints

Use the **Breakpoint** controls in the Stateflow Debugging window to specify global breakpoints. When a global breakpoint is encountered during simulation, execution stops and the Debugger takes control. Select any or all of the following global breakpoints:

- **Chart Entry** — Simulation halts on chart entry.
- **Event Broadcast** — Simulation halts when an event is broadcast.
- **State Entry** — Simulation halts when a state is entered.

These breakpoints can be changed during run-time and are immediately enforced. When you save a Stateflow diagram, the breakpoint settings are saved with it.

Global breakpoints can be changed during run-time and are immediately enforced. When you save the Stateflow diagram, all the **Stateflow Debugging** window settings (including breakpoints) are saved, so that the next time you open the model, the breakpoints are as you left them.

Setting Local Breakpoints

You can set breakpoints for Stateflow objects that halt the execution of state actions, transitions, function calls, and event broadcasts.

- 1 In the **Model Explorer**, right-click a particular state, transition, function, or event.
- 2 From the resulting pop-up menu, select **Properties**.

A dialog appears for setting the properties of the object.

You can also access the properties dialog for states, transitions, and functions in the Stateflow diagram editor by right-clicking on them and selecting **Properties** from the resulting pop-up menu. You can only access the properties dialog for an existing event in the **Model Explorer**.

- 3 Select from the following **Debugger breakpoints** options:

For states, select one of the following breakpoints:

- **State During** — Stop execution before state during actions are performed.
- **State Entry** — Stop execution before state entry actions are performed.
- **State Exit** — Stop execution before state exit actions are performed.

For transitions, select one of the following breakpoints:

- **When Tested** — Stop execution before the transition is tested to see if it is a valid transition to take.
- **When Valid** — Stop execution after the transition tests valid, but before the transition is actually taken.

For functions, select **Function Call** to stop execution before the function is called.

For events, select one of the following breakpoints:

- **Start of Broadcast** — Stop execution before the event is broadcast.
- **End of Broadcast** — Stop execution after event is read by a Stateflow object.

Setting Error Checking in the Debugging Window

The options in the **Error checking options** section of the **Stateflow Debugging** window insert generated code in the simulation target to provide breakpoints to catch different types of errors that might occur during simulation. Select any or all of the following error checking options:

- **State inconsistency** — Check for state inconsistency errors that are most commonly caused by the omission of a default transition to a substate in superstates with XOR decomposition. See “Debugging State Inconsistencies” on page 13-14 for a complete description and example.
- **Transition Conflict** — Check whether there are two equally valid transition paths from the same source at any step in the simulation. See “Debugging Conflicting Transitions” on page 13-16 for a complete description and example.
- **Data Range** — Check whether the minimum and maximum values you specified for a data in its properties dialog are exceeded. Also check whether fixed-point data overflows its base word size. See “Debugging Data Range Violations” on page 13-18 for a complete description and example.
- **Detect Cycles** — Check whether a step or sequence of steps indefinitely repeats itself. See “Debugging Cyclic Behavior” on page 13-20 for a complete description and example.

To include the supporting code designated for these debugging options in the simulation application, select the **Enable debugging/animation** check box in the **Coder Options** dialog for the simulation target. This is described in “Configuring a Simulation Target for Stateflow” on page 12-11.

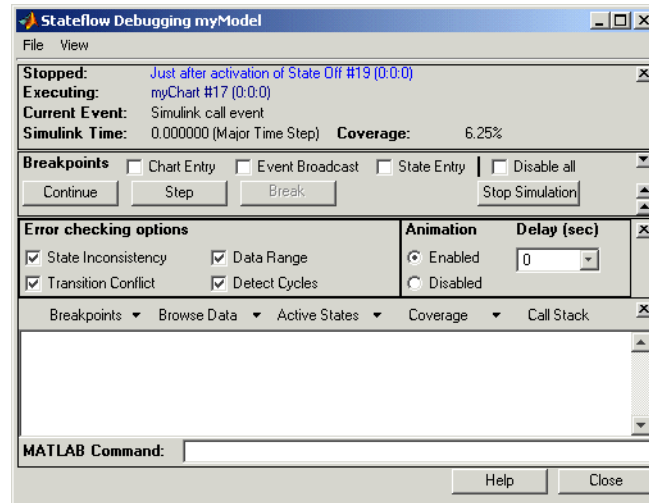
Note You must rebuild the target for any changes to any of the settings referred to above to take effect.

Starting Simulation in the Debugging Window

To debug the Stateflow diagrams in a model, you start simulation in the Debugging window with the following steps:

1 Select the **Start** button.

A debugging simulation session starts. When a breakpoint that you set is encountered, the Stateflow Debugging window takes on the following appearance:



At the breakpoint, the following status items are displayed in the upper portion of the Debugger window:

- **Stopped** — Displays the step executed just prior to breaking execution.
- **Executing** — Displays the currently executing Stateflow chart.
- **Current Event** — Displays the event being processed by the Stateflow chart.
- **Simulink Time** — Displays the current simulation time.
- **Code Coverage** — Displays the percentage of code covered in this simulation.

During simulation the Stateflow diagram is marked read-only. The appearance of the Stateflow diagram editor toolbar and menus changes so that object creation is not possible. When the diagram editor is in this read-only mode, its condition is referred to as *iced*.

Controlling Animation in the Debugging Window

During simulation of a Simulink model, you can animate the Stateflow diagrams. Animation highlights objects in Stateflow diagrams as they execute during simulation. You activate or deactivate animation for Stateflow diagrams during simulation in the **Debugging** window as follows:

- To activate animation for simulation, in the Animation section of the Debugging window, select **Enabled** to activate animation before the start of simulation.
- To deactivate animation for simulation, stop simulation and select **Disabled**.

Before, during, and after simulation, you control the speed of animation by selecting a value for the **Delay** field as follows:

- For the fastest animation, select a value of 0 seconds.
- For the slowest animation, select a value 1 seconds.

Controlling the Execution Rate in the Debugging Window

Once you start simulation as described in “Starting Simulation in the Debugging Window” on page 13-5, and a breakpoint is reached, you can control the rate of execution of Stateflow diagrams to execute step-by-step or continuously until another breakpoint is reached. Use the following buttons in the Stateflow Debugging window to control the rate of execution:

- **Continue** — After simulation has been started, and a breakpoint has been encountered, the **Start** button is marked **Continue**. Press **Continue** to continue simulation.
- **Step** — Execute the next execution step, and suspend the simulation.
- **Break** — Suspend the simulation and transfer control to the **Debugging** window.
- **Stop Simulation** — Stop simulation altogether and relinquish debugging control. When simulation stops, the Stateflow diagram editor toolbar and menus return to their normal appearance and operation so that object creation is again possible.

Setting the Output Display Pane

During simulation, the Debugging window monitors a variety of execution indicators in its output display in the bottom pane of the Debugging window. You select the contents of this display with the following pull-downs located just above the display, which are enabled only after a breakpoint is reached during simulation.

- **Breakpoints** — Display a list of the set breakpoints. You can set breakpoints in the Debugger and in the properties dialogs of individual objects such as states, transitions, and functions. See “Setting Breakpoints for Debugging” on page 13-3 for details. This option lists breakpoints for the currently executing chart or for all charts in the model.
- **Browse Data** — Display the current values of defined data objects. This pull-down list lets you filter displayed data between all data and watched data. Watched data has the **Data** property **Watch in Debugger** enabled for it. Each of these categories is further filtered by data for the currently executing chart, or all charts in the model. For more details see “Watching Data in the Stateflow Debugger” on page 13-24.
- **Active States** — Display a list of active states in the display area. Double-clicking any state causes the graphics editor to display that state. This pull-down lets you display active states in the current chart, or active states for all charts in the model.
- **Call Stack** — Display a sequential list of the **Stopped** and **Current Event** status items that occur with each single-step through the simulation.

Once you make a selection, the pulldown corresponding to the current display is highlighted. Once you select an output display button, that type of output is displayed until you choose a different display type. You can clear the display by selecting **Clear Display** from the **File** menu of the **Stateflow Debugging** window.

Debugging Run-Time Errors Example

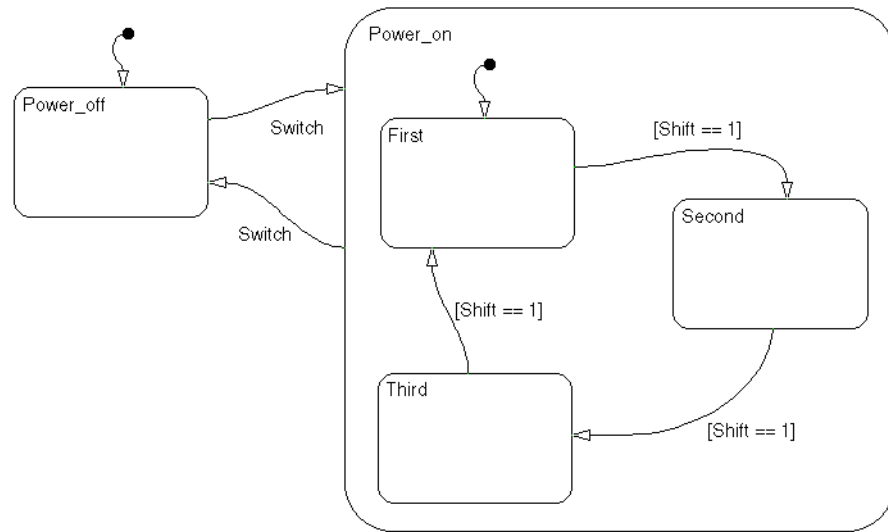
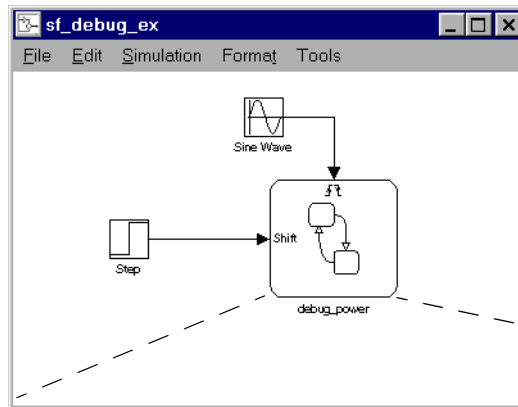
The following topics describe the steps used in a typical debugging scenario to resolve run-time errors in an example model:

- 1 “Create the Model and Stateflow Diagram” on page 13-9 — Create a Simulink model with a Stateflow diagram to debug.
- 2 “Debugging the Stateflow Diagram” on page 13-11 — Describes the individual steps that you take to inspect the behavior the Stateflow diagram during simulation.
- 3 “Correcting the Run-Time Error” on page 13-12 — Shows you how to deal with the run-time error you encounter.

Create the Model and Stateflow Diagram

In this topic you create an example model that you use as a debugging example. Use the following steps to create the example model:

- 1 Create the following Simulink model and Stateflow diagram:



- 2 From the Stateflow diagram editor, add an event Switch with a scope of **Input from Simulink**, and a **Rising Edge** trigger.
- 3 Also add a data Shift with a scope of **Input from Simulink**.

The Stateflow diagram has two states at the highest level in the hierarchy, `Power_off` and `Power_on`. By default `Power_off` is active. The event `Switch` toggles the system between the `Power_off` and `Power_on` states. `Power_on` has three substates, `First`, `Second`, and `Third`. By default, when `Power_on` becomes active, `First` also becomes active. When `Shift` equals 1, the system transitions from `First` to `Second`, `Second` to `Third`, `Third` to `First`, for each occurrence of the event `Switch`, and then the pattern repeats.

In the Simulink model, there is an event input and a data input. A Sine wave block is used to generate a repeating input event that corresponds with the Stateflow event `Switch`. The Step block is used to generate a repeating pattern of 1 and 0 that corresponds with the Stateflow data object `Shift`. Ideally, the `Switch` event occurs in a frequency that allows at least one cycle through `First`, `Second`, and `Third`.

Debugging the Stateflow Diagram

You create an example model with a Stateflow diagram that needs debugging in “Create the Model and Stateflow Diagram” on page 13-9. Use the steps that follow to debug the Stateflow diagram in this model.

- 1 In the Stateflow diagram editor, from the **Tools** menu, select **Open Simulation Target**.

The **Target Options** dialog appears.

- 2 In the **Target Options** dialog, select **Coder Options**.

The **Coder Options** dialog appears.

- 3 Make sure that **Enable Debugging/Animation** is selected.

- 4 Select **Close** in both the **Coder Options** and **Target Options** dialogs to close them and apply the changes.

- 5 In the Stateflow diagram editor, from the **Tools** menu, select **Debug**.

The Stateflow Debugging window opens.

- 6 Select the **Chart entry** option under the **Break Controls** border.

7 Under **Animation**, select **Enabled** to enable animation of Stateflow diagrams during simulation.

8 In the **Stateflow Debugging** window, select the **Start** button to start the simulation.

Informational messages are displayed in the MATLAB Command Window. The graphics editor toolbar and menus change appearance to indicate a read-only interface. The Stateflow diagram is parsed, the code is generated, and the target is built.

Because you specified a breakpoint on chart entry, the execution stops at that point and the Debugger display status indicates the following:

```
Stopped: Just after entering during function of Chart debug__power
Executing: sf_debug_ex_debug_power
Current Event: Input event Switch
```

9 Select the **Step** button.

The **Step** button executes the next execution step and stops.

10 Continue selecting the **Step** button and watching the animating Stateflow diagram.

After each step, watch the Stateflow diagram animation and the Debugger status area to see the sequence of execution.

Single-stepping shows that the Stateflow diagram does not exhibit the desired behavior. The transitions from the `First` to the `Second` to the `Third` state inside the state `Power_on` are not occurring because the transition from `Power_on` to `Power_off` takes priority. The output display of code coverage also confirms this observation.

Correcting the Run-Time Error

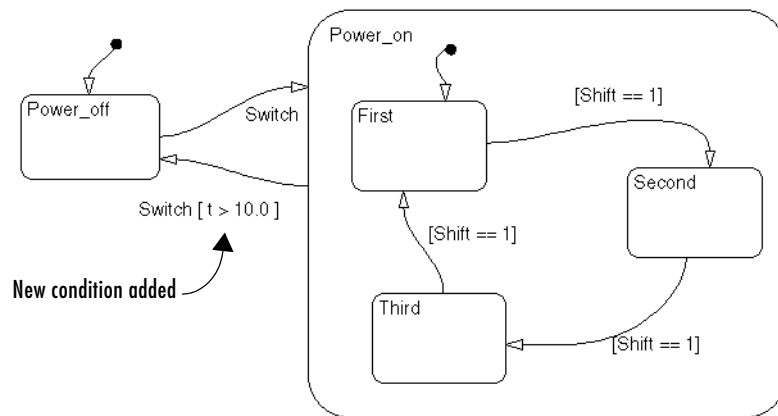
In “Debugging the Stateflow Diagram” on page 13-11, you step through a simulation of an example Stateflow diagram and find an error: the generation of the event `Switch` is driving the simulation and the simulation time is passing too quickly for the input data object `Shift` to have an effect.

Correct this error in the following steps:

- 1 Choose **Stop** from the **Simulation** menu of the graphics editor.

The Stateflow diagram editor is now writeable. The model might need to be completely rethought.

- 2 Add the condition `[t > 10.0]` to the transition from `Power_on` to `Power_off` as shown.



Now the transition from `Power_on` to `Power_off` is not taken until simulation time is greater than 10.0.

- 3 In the **Stateflow Debugging** window, select **Start** to begin simulation again.
- 4 Select **Step** repeatedly to observe the new behavior.

Debugging State Inconsistencies

Stateflow notations specify that states are consistent if

- An active state (consisting of at least one substate) with XOR decomposition has exactly one active substate.
- All substates of an active state with AND decomposition are active.
- All substates of an inactive state with either XOR or AND decomposition are inactive.

A state inconsistency error has occurred if, after a Stateflow diagram completes an update, the diagram violates any of the preceding notation rules.

Causes of State Inconsistency

State inconsistency errors are most commonly caused by the omission of a default transition to a substate in superstates with XOR decomposition.

Design errors in complex Stateflow diagrams can also result in state inconsistency errors. These errors are only detectable using the Debugger at run-time.

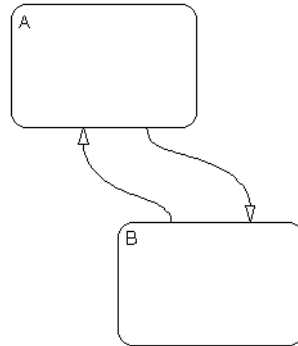
Detecting State Inconsistency

To detect the state inconsistency during a simulation,

- 1 Build the target with debugging enabled.
- 2 Invoke the Debugger and enable **State Inconsistency** checking.
- 3 Start the simulation.

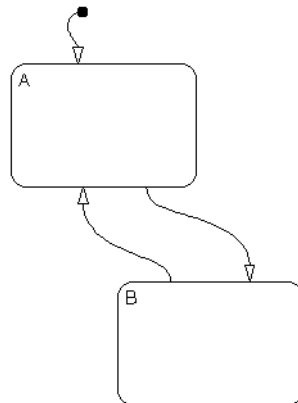
State Inconsistency Example

This Stateflow diagram has a state inconsistency.



In the absence of a default transition indicating which substate is to become active, the simulation encounters a run-time state inconsistency error.

Adding a default transition to one of the substates resolves the state inconsistency.



Debugging Conflicting Transitions

A transition conflict exists if, at any step in the simulation, there are two equally valid transition paths from the same source. In the case of a conflict, equivalent transitions (based on their labels) are evaluated based on the geometry of the outgoing transitions. See “Transition Testing Order” on page 3-10 for more information.

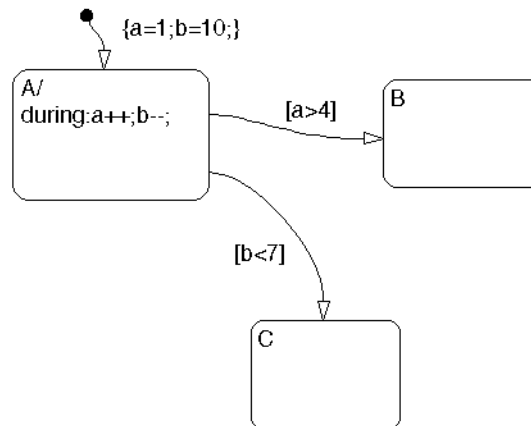
Detecting Conflicting Transitions

To detect conflicting transitions during a simulation, do the following:

- 1 Build the target with debugging enabled.
- 2 Invoke the Debugger and enable **Transition Conflict** checking.
- 3 Start the simulation.

Conflicting Transition Example

This Stateflow diagram has a conflicting transition.



The default transition to state A assigns data a equal to 1 and data b equal to 10. State A's during action increments a and decrements b. The transition from state A to state B is valid if the condition $[a > 4]$ is true. The transition from state A to state C is valid if the condition $[b < 7]$ is true. As the simulation proceeds, there is a point where state A is active and both conditions are true. This is a transition conflict.

Multiple outgoing transitions from states that are of equivalent label priority are evaluated in a clockwise progression starting from the twelve o'clock position on the state. In this example, the transition from state A to state B is taken.

Although the geometry is used to continue after the transition conflict, it is not recommended that you design your Stateflow diagram based on an expected execution order.

Debugging Data Range Violations

Stateflow detects the following data range violations during simulation:

- If a data object equals a value outside the range of the values set in the **Initial**, **Minimum**, and **Maximum** fields specified in the data properties dialog

See “Setting Data Properties in the Data Dialog” on page 6-25 for a description of the **Initial**, **Minimum**, and **Maximum** fields in the data properties dialog.

- If the fixed-point result of a fixed-point operation overflows its bit size
See “Overflow Detection for Fixed-Point Types” on page 8-11 for a description of the overflow condition in fixed-point numbers.

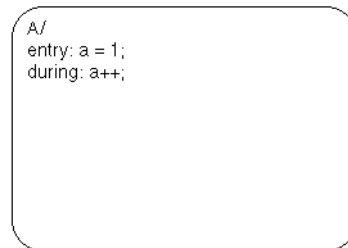
Detecting Data Range Violations

To detect data range violations during a simulation,

- 1 Build the target with debugging enabled.
- 2 Open the Debugger window.
- 3 In the **Error checking options** of the Debugger, select **Data Range**.
- 4 Start the simulation.

Data Range Violation Example

This Stateflow diagram has a data range violation.



The data `a` is defined to have an **Initial** and **Minimal** value of 0 and a **Maximum** value of 2. Each time an event awakens this Stateflow diagram and state A is active, `a` is incremented. The value of `a` quickly becomes a data range violation.

Debugging Cyclic Behavior

When a step or sequence of steps is indefinitely repeated (recursive), this is called cyclic behavior. The Debugger cycle detection algorithms detect a class of infinite recursions caused by event broadcasts.

To detect cyclic behavior during a simulation, do the following:

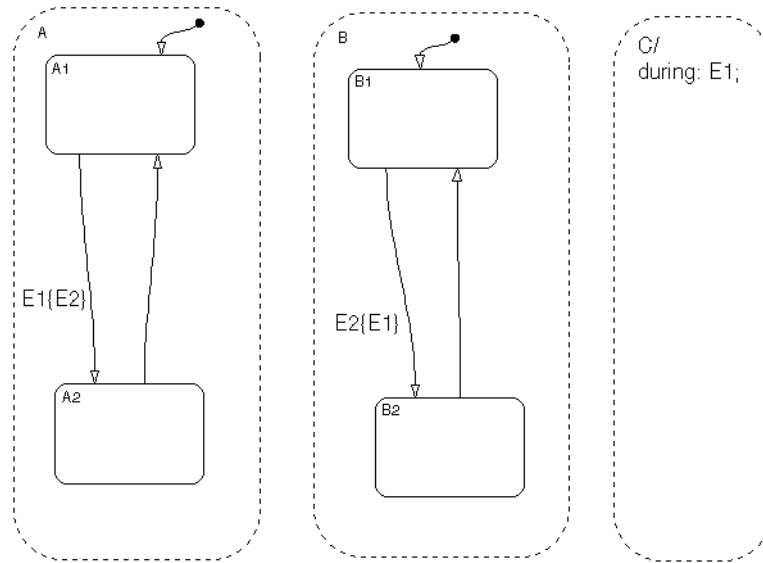
- 1** Build the target with debugging enabled.
- 2** Invoke the Debugger and enable **Detect Cycles**.
- 3** Start the simulation.

See the following sections for examples of cyclic behavior:

- “Cyclic Behavior Example” on page 13-21 — Shows a typical example of a cycle created by infinite recursions caused by an event broadcast.
- “Flow Cyclic Behavior Not Detected Example” on page 13-22 — Shows an example of cyclic behavior in a flow diagram that is not detected by the Debugger.
- “Noncyclic Behavior Flagged as a Cyclic Example” on page 13-23 — Shows an example of noncyclic behavior that the Debugger flags as being cyclic.

Cyclic Behavior Example

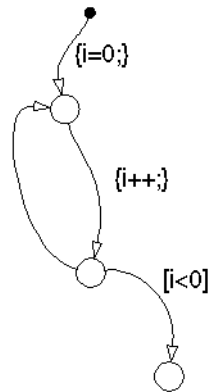
This Stateflow diagram shows a typical example of a cycle created by infinite recursions caused by an event broadcast.



When the state C during action executes, event E1 is broadcast. The transition from state A.A1 to state A.A2 becomes valid when event E1 is broadcast. Event E2 is broadcast as a condition action of that transition. The transition from state B.B1 to state B.B2 becomes valid when event E2 is broadcast. Event E1 is broadcast as a condition action of the transition from state B.B1 to state B.B2. Because these event broadcasts of E1 and E2 are in condition actions, a recursive event broadcast situation occurs. Neither transition can complete.

Flow Cyclic Behavior Not Detected Example

This Stateflow diagram shows an example of cyclic behavior in a flow diagram that is not detected by the Debugger.

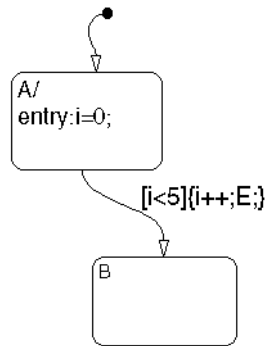


The data object `i` is set to 0 in the condition action of the default transition. `i` is incremented in the next transition segment condition action. The transition to the third connective junction is valid only when the condition `[i < 0]` is true. This condition will never be true in this flow diagram and there is a cycle.

This cycle is not detected by the Debugger because it does not involve event broadcast recursion. Detecting cycles that depend on data values is not currently supported.

Noncyclic Behavior Flagged as a Cyclic Example

This Stateflow diagram shows an example of noncyclic behavior that the Debugger flags as being cyclic.



State A becomes active and i is initialized to 0. When the transition is tested, the condition $[i < 5]$ is true. The condition actions that increment i and broadcast the event E are executed. The broadcast of E when state A is active causes a repetitive testing (and incrementing of i) until the condition is no longer true. The Debugger flags this as a cycle when in reality the apparent cycle is broken when i becomes greater than 5.

Watching Data Values with Debuggers

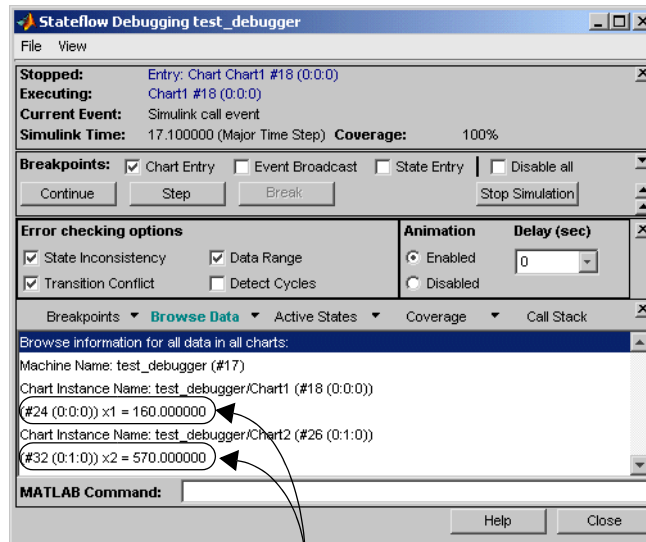
An important part of debugging is to observe the value of individual data during simulation. Use the following topics to display the value of Stateflow data while you are simulating a model:

- “Watching Data in the Stateflow Debugger” on page 13-24 — Display Stateflow data values in the Stateflow Debugger after a breakpoint is reached.
- “Watching Stateflow Data in MATLAB” on page 13-25 — Use the Command Line Debugger to report data values for the currently executing Stateflow block at the MATLAB prompt when a breakpoint is reached.

Watching Data in the Stateflow Debugger

The **Browse Data** pull-down menu in the Stateflow **Debugger** lets you display selected data in the bottom output display pane of the Stateflow Debugger during simulation, after a breakpoint is reached. Its selections filter the displayed data items between watched data and all data. Watched data has the property **Watch in Debugger** enabled for it. Watched data is further filtered between data for the currently executing chart, or data for all charts in the model.

The following example is set to display **All Data (All Charts)** for two executing charts, Chart1 and Chart2, for the simulating model `test_debugger.mdl`. Each chart has its own data value: `x1` and `x2`, respectively.



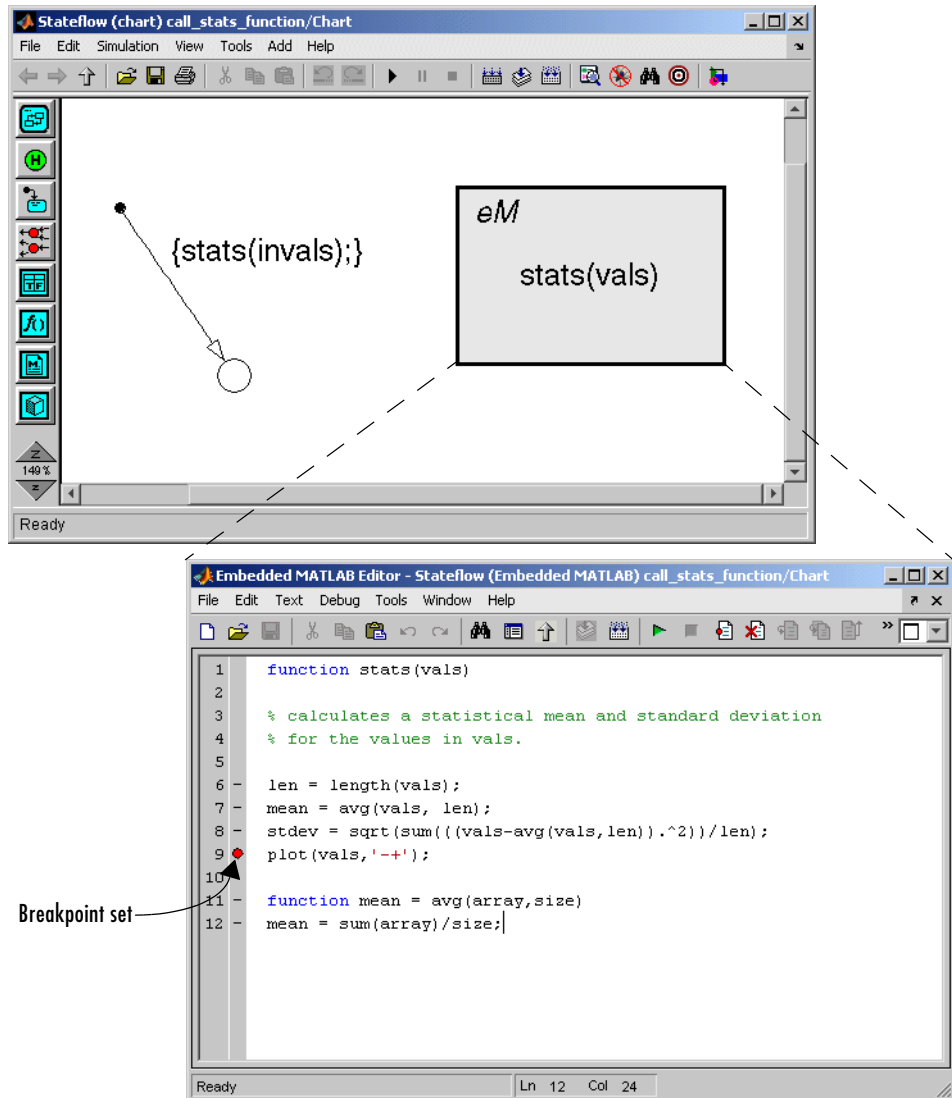
Data x1 and x2

The data for each chart is headed by its owning object. Each displayed object (chart, state, data, and so on) is accompanied by a unique identifier in the form (#id(xx:yy:zz)), which is used in linking the listed object to its appearance in the Stateflow diagram.

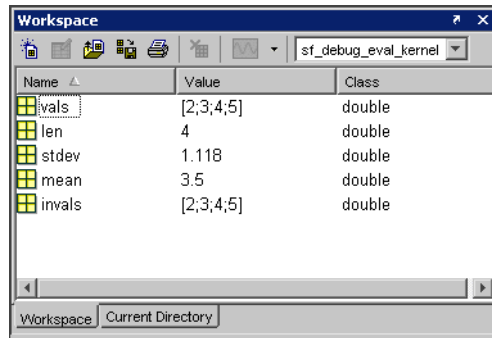
Note Fixed-point data is displayed with two values: the quantized integer value (stored integer) and the scaled real-world (actual) value. See “Using Fixed-Point Data in Stateflow” on page 8-5.

Watching Stateflow Data in MATLAB




When simulation reaches a breakpoint, you can see the values for Stateflow data in the MATLAB **Workspace Browser** and in the MATLAB Command Window. In the following example, a default transition calls an Embedded MATLAB function with a breakpoint set at the last executable line of the function:



When simulation reaches the breakpoint, the MATLAB Workspace Browser is updated with the values of Stateflow data in the scope of the breaking object (state, function, and so on), as shown.



All data displayed in the Workspace dialog at the break point cannot be modified. This means that the following tools for the Workspace dialog are disabled at this time:

Tool Icon	Tool Name
	Add New Variable
	Load Data File
	Delete

Once the breakpoint is reached you can also display Stateflow data in the MATLAB Command Window for the preceding example as follows:

- 1 At the MATLAB Command Window command prompt, press **Enter**.

A `debug>>` prompt appears.

- 2 Enter `stdev` at the prompt and press **Enter** to display its value, as shown.

```
debug>> stdev
```

```
1.1180
```

```
debug>>
```

- 3 Enter the MATLAB command `whos`.

```
debug>> whos
```

Name	Size	Bytes	Class
vals	4x1	32	double array
len	1x1	8	double array
stdev	1x1	8	double array
mean	1x1	8	double array
invals	4x1	32	double array

Grand total is 5 data in scope

debug>>

The Command Line Debugger provides the following commands during simulation:

Command	Description
dbstep	Advance to next program step after a breakpoint is encountered.
dbcont	Continue execution to next breakpoint.
dbquit (ctrl-c)	Stop simulation of the model. Press Enter after this command to return to the MATLAB prompt.
help	Displays help for command-line debugging.
print x	Display the value of the variable x. This is equivalent to typing x at the Command Line Debugger prompt. If x is a vector or matrix, you can also index into x. For example, x(1,2).
save	Saves all variables to the specified file. Follows the syntax of the MATLAB save command. To retrieve variables to the MATLAB base workspace, use load command after simulation has been ended.
whos	Display the size and class (type) of all variables in the scope of the halted Embedded MATLAB Fcn block.

You can issue any other MATLAB command at the `debug>>` prompt, but the results are executed in the Stateflow workspace. To issue a command in the MATLAB base workspace at the `debug>>` prompt, use the `evalin` command with the first argument `'base'` followed by the second argument command string, for example, `evalin('base','whos')`.

Note To return to the MATLAB base workspace, use the `dbquit` command.

Monitoring Stateflow Test Points

A Stateflow test point is a signal that Stateflow guarantees to be observable, for example, by a Floating Scope block, during a simulation. Stateflow allows you to designate a data or a state in a Stateflow diagram as a test point. While all Stateflow states can be test points, only Stateflow data of local scope qualify. They can be scalar, one-dimensional, or two-dimensional in size, and of any data type except type `m1`. Like states, test point data must also be a descendant of a Stateflow chart.

You implicitly declare all Stateflow data of local scope and all states as test points if the **Enable debugging/animation** code option is selected for the Stateflow simulation target. If this option is not set, you can specify individual local data or states as test points by setting their **TestPoint** property in the Stateflow API or through the Model Explorer (see “Setting Test Points for Stateflow States and Local Data with Model Explorer” on page 13-31).

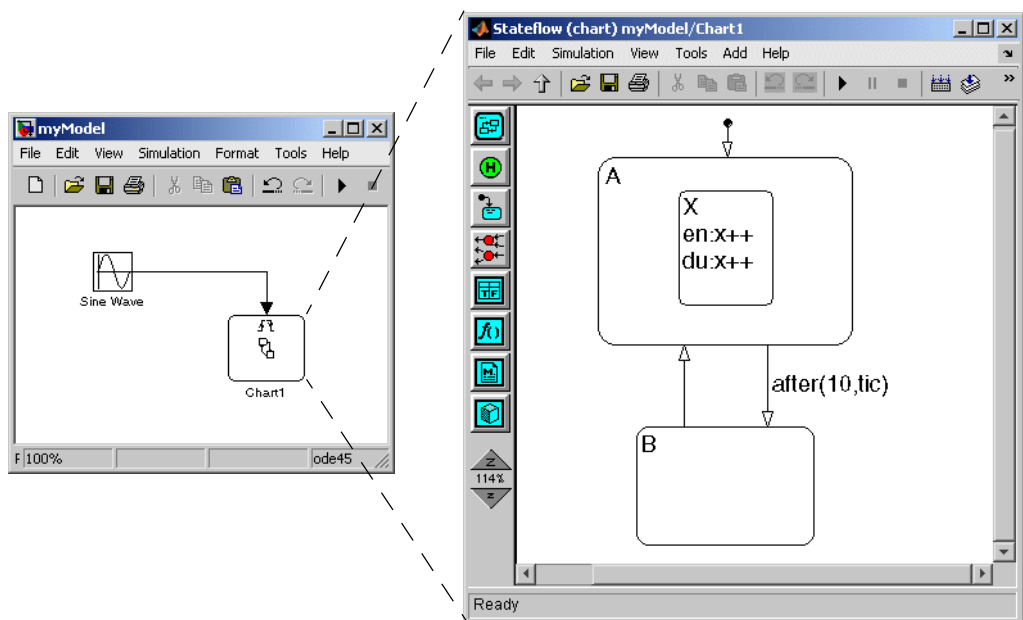
You can monitor individual Stateflow test points with a floating scope during model simulation. You can also log test point values into MATLAB workspace objects. Use the following topics to learn how to set test points and display the values of Stateflow test points while you are simulating a model.

- “Setting Test Points for Stateflow States and Local Data with Model Explorer” on page 13-31 — Through Model Explorer, set test points for states and local data for logging and monitoring.
- “Logging Data Values and State Activity” on page 13-33 — During simulation, log data values and state activity values into MATLAB objects for reporting and plotting.
- “Using a Floating Scope to Monitor Data Values and State Activity” on page 13-38 — Use a Floating Scope block to monitor Stateflow data values continuously during simulation.

Setting Test Points for Stateflow States and Local Data with Model Explorer

You can explicitly set individual states or local data as test points through the Model Explorer. Use the example you create in the following procedure to learn how to set individual test points for Stateflow states and data.

- 1 Create the following model:



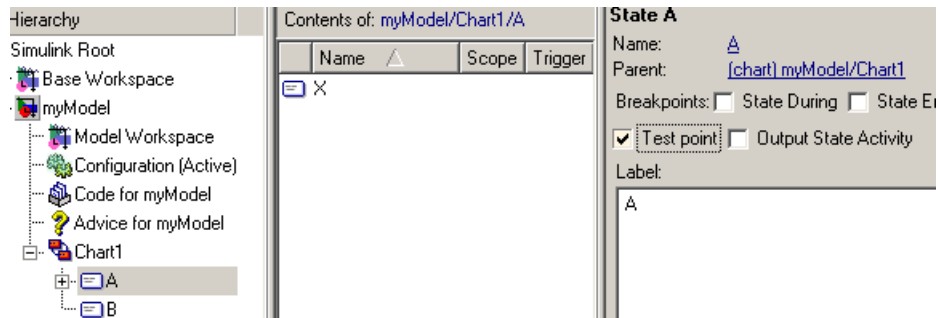
The model consists of a single Stateflow block named Chart1, which is triggered by a signal from a Sine Wave block through the input trigger event `tic`. In the Stateflow diagram, the state A and its substate X are entered for the first `tic` event. State A and substate X stay active until 10 `tic` events have taken place and then state B is entered. On the following event, state A and substate X are entered and the cycle continues.

The data `x` is added to the state X. The entry and during actions for substate X increment `x` while X is active for 10 `tic` events. When state B is entered, `x` is reinitialized to zero, and the cycle repeats.

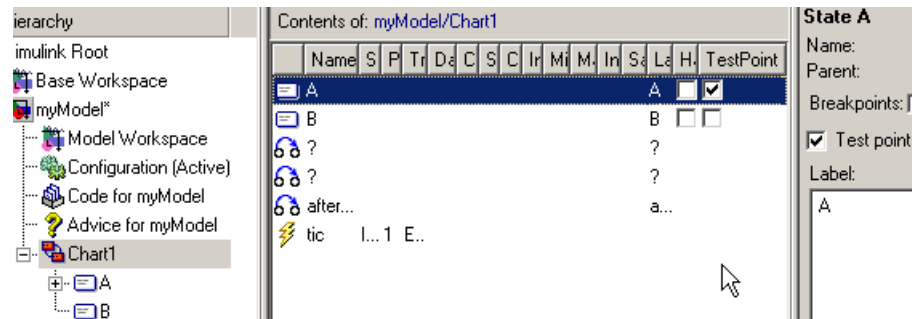
- 2 At the MATLAB Command Window prompt, type

```
myModel
```
 - 3 Start the Model Explorer. In the Simulink model, select **View -> Model Explorer**.
- The Model Explorer is displayed.
- 4 In the Model Explorer, expand myModel.
 - 5 Expand Chart1, then select A.
 - 6 In the right-most pane, **State A**, select the **Test point** check box. Click **Apply**.

This creates a test point for the A state.



Alternatively, you can access a test point through the middle pane. By default, the Model Explorer displays event and data child objects in the **Contents** pane for the selected object in the **Model Hierarchy** pane. You can set the test point for the A state through this pane by selecting the parent of A. If the states are not listed in the middle pane, select the **States/Functions/Boxes/Etc.** check box in the **View -> List View Options for All Stateflow Objects**.



- 7 Repeat step 6 for state X. Click **Apply**.
- 8 Select X again. Select the local data x in the **Contents** pane.
- 9 In the right-most pane for that data, select the **Value Attributes** tab, then select the **Test point** check box. Click **Apply**.
- 10 Repeat step 6 for state B. Click **Apply** and save the model.

You can now log these test points. See “Logging Data Values and State Activity” on page 13-33.

Logging Data Values and State Activity

During simulation, you can log values for data and state activity into Simulink objects. After simulation, you can access these objects in the MATLAB workspace and use them to report and plot the values.

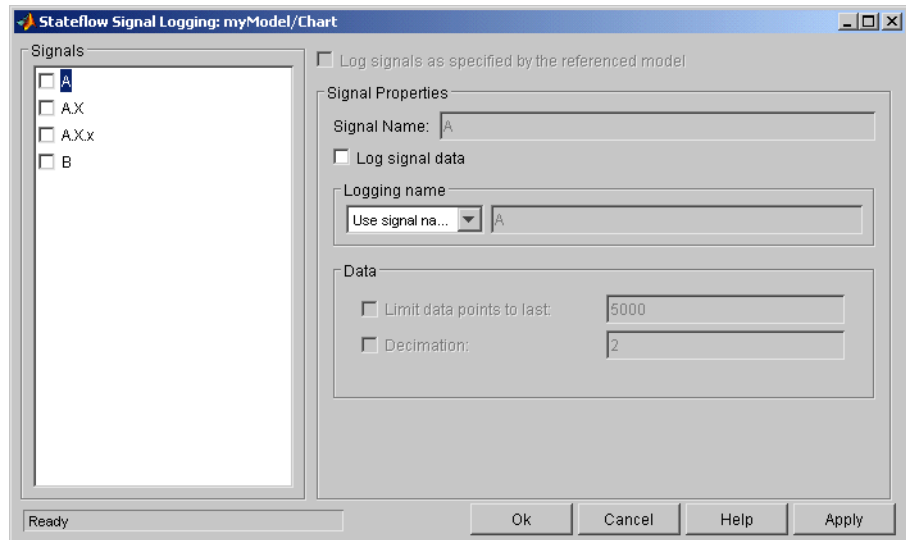
You can use the following procedure to learn how to access logged Stateflow data and state activity. This procedure uses the model, myModel, from the preceding topic, “Setting Test Points for Stateflow States and Local Data with Model Explorer” on page 13-31.

- 1 If myModel is not already open, in the MATLAB Command Window prompt, type
myModel

The model is displayed.

- In the Simulink window, right-click the Stateflow block and select **Log Chart Signals**.

The **Signal Logging** dialog appears, as shown.



- Select the check box next to A.

This is the state activity signal for state A. When A is active, its value is 1. When A is inactive, its value is 0.

After checking A, notice the following properties in the right pane of the **Signal Logging** dialog:

Signal Properties	Description
Signal Name	Name of the highlighted state or data.
Log signal data	Checking this selects the highlighted signal in the Signals pane.

Signal Properties	Description
Logging name	Name of the signal logged. By default, this is set to the name of the selected/highlighted state or data. You can select Custom for this property to rename the selected/highlighted signal in the adjacent field to the right.
Limit data points to last	Select this property to enter the number of most recent sample values to log in the adjacent field to the right for the selected/highlighted signal.
Decimation	Select this property to enter the level of decimation for the signal values logged for the selected/highlighted signal.

4 Select all the signals in the **Signal** pane and click **OK** to close the **Signal Logging** dialog.

5 Simulate the model.

During simulation, the Simulink model data log object `logsOut` is generated in the MATLAB workspace.

6 After simulation, enter the following at the MATLAB prompt:

```
>> logsOut
```

You see the following result:

```
logsOut =
```

```
Simulink.ModelDataLogs (myModel):
```

```
Name                Elements  Simulink Class
```

```
Chart1                4        StateflowDataLogs
```

The display identifies `logsOut` as a Simulink object of type `ModelDataLogs`. This is the highest level logging object. The object `Chart1` appears as the only contents of `logsOut`. It represents logged data for the Stateflow block `Chart1` and is identified as a Simulink object of type `StateflowDataLogs`.

7 At the MATLAB prompt, enter the following:

```
>> logsOut.Chart1
```

You see the following result:

```
ans =
```

```
Simulink.StateflowDataLogs (Chart1):
```

Name	Elements	Simulink Class
('A.X.x')	1	Timeseries
A	1	Timeseries
('A.X')	1	Timeseries
B	1	Timeseries

The signals that you selected in the **Signal Logging** dialog appear as Simulink objects of type Timeseries. Notice that the signals for the activity of state X and the value of data x appear as ('A.X') and ('A.X.x'), respectively. Because of the way that logged signals are stored for Stateflow, you need to use this notation to access logged data for Stateflow objects below chart level in the Stateflow diagram.

8 At the MATLAB prompt, enter the following:

```
>> logsOut.Chart1.('A.X.x')
```

You see the following result:

```
ans =
```

```

    Name: 'A.X.x'
  BlockName: 'StateflowChart/A.X.x'
  PortIndex: 1
  SignalName: 'A.X.x'
  ParentName: 'A.X.x'
  TimeInfo: [1x1 Simulink.TimeInfo]
         Time: [114x1 double]
         Data: [114x1 double]
```

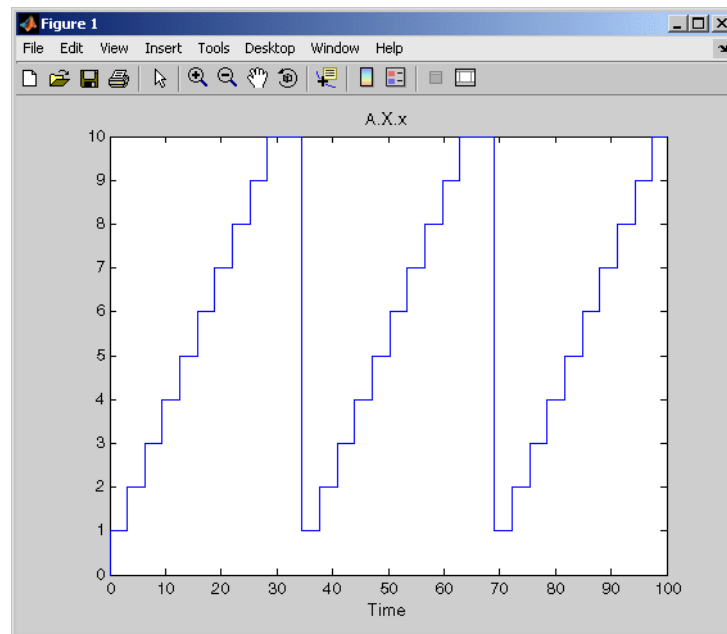
The logging object for the data x, ('A.X.x'), is actually a structure of logged data pertinent to x. The actual logged signal values for x are contained in the

Data object, a vector of 114 values. For example, if you were to enter the MATLAB command `logsOut.Chart1('A.X.x').Data`, a long stream of data would appear. A better way to see the logged values of `x` is to use the `plot` method shown in the next step.

- 9 At the MATLAB prompt, plot the values of `x` with the following command:

```
>> logsOut.Chart1('A.X.x').plot
```

You see the following result:

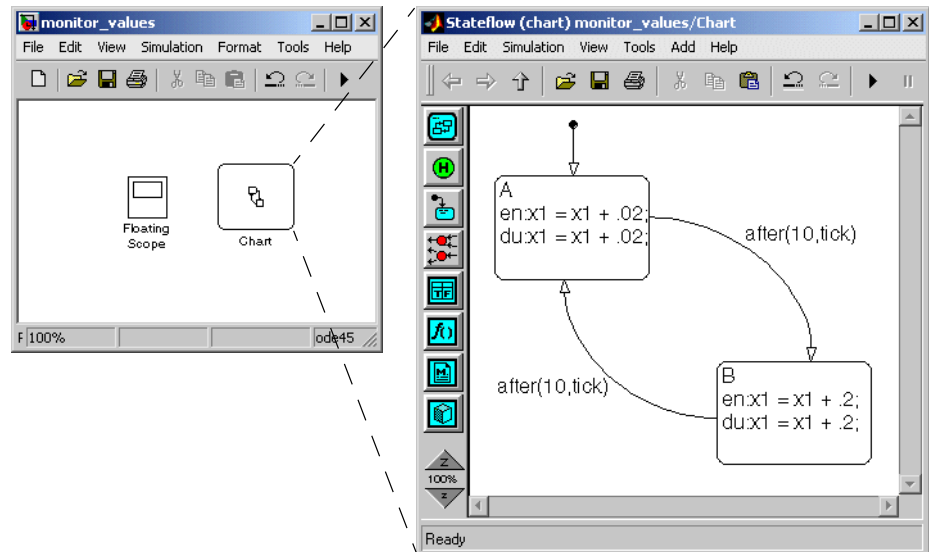


The preceding plot exhibits the expected results for the value of `x`. It is incremented for 10 time steps before resetting to 0 when states `X` and `A` are exited and state `B` is entered in the Stateflow diagram.

The preceding example is a demonstration of some of the capabilities you have for reporting logged Stateflow data. Stateflow data conforms to the general rules for handling logging signals in Simulink. For more information on how you can use and manipulate logged data with commands and scripts in MATLAB, see “Logging Signals” in the Simulink documentation.

Using a Floating Scope to Monitor Data Values and State Activity

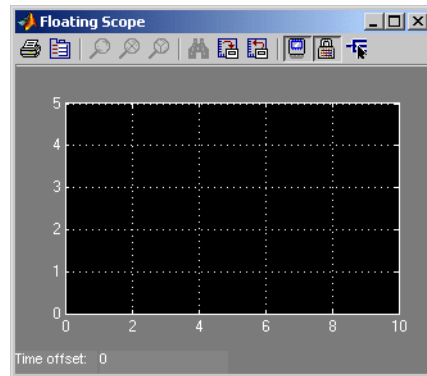
In the steps of this topic, you configure a Floating Scope block to monitor a data value and the activity of a state in the following example model:




The model consists of a Floating Scope block and a Stateflow block. The Stateflow diagram for the Stateflow block starts by adding an increment of .02 for 10 samples to the data $x1$. For the next 10 samples, an increment of .2 is added, and the cycle repeats.

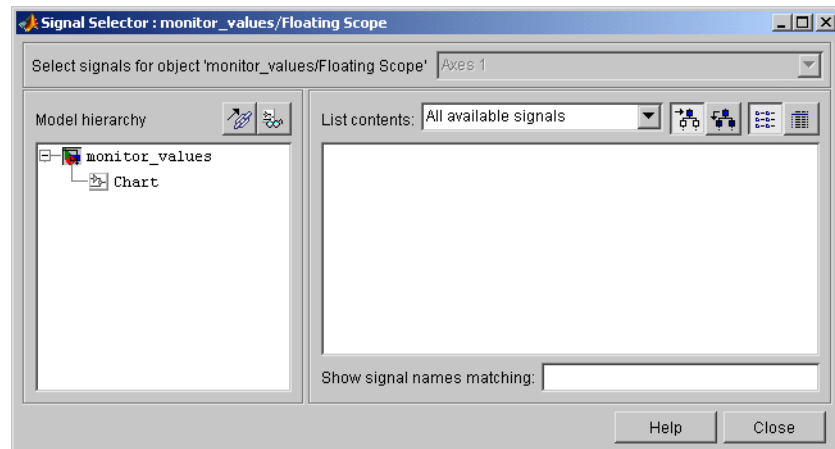
- 1 Double-click the Floating Scope block.

A **Floating Scope** window appears, already scaled for this example.

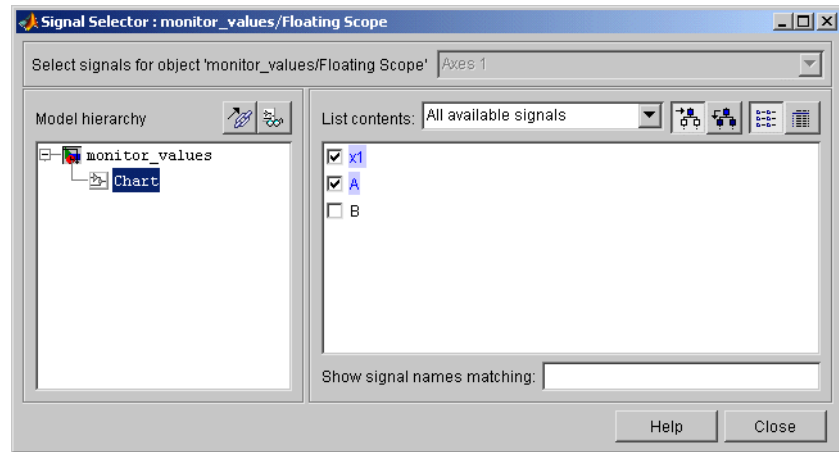


- 2 In the **Floating Scope** window, select the Signal Selection tool .

The **Signal Selector** dialog appears with a hierarchy of Simulink blocks for the model.



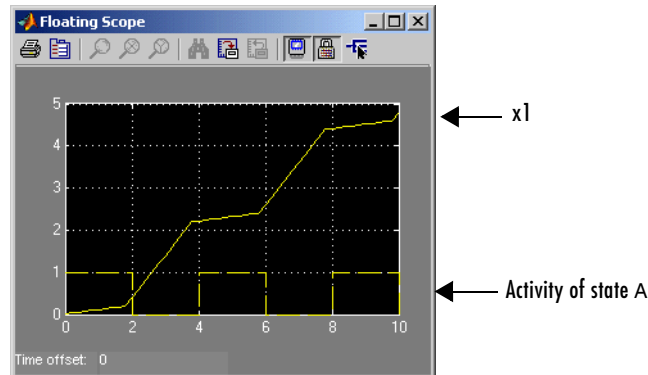
- 3 In the **Model hierarchy** pane, select the Stateflow block whose signals you want to monitor and, in the **List contents** pane, select the data you want to monitor.



In the preceding example, the block named Chart is selected in the **Model hierarchy** pane, and the data x1 and the activity of state A are selected in the **Contents** pane.

4 Simulate the model.

When the example model is simulated, you receive a signal trace for x1 and for the activity of state A, as shown.



Notice that when state A is active, its activity signal value is 1, and when it is inactive, its signal value is 0. Because this value is very low or very high

compared to some data, you might want to put it in a second Floating Scope block to compare it with other data.

Understanding Model Coverage for Stateflow Charts

Model coverage is a measure of how thoroughly a model is tested. The Model Coverage tool helps you to validate your model tests by measuring model coverage for your tests. It determines the extent to which a model test case exercises simulation control flow paths through a model. The percentage of paths that a test case exercises is called its *model coverage*.

Note The Model Coverage tool requires a Simulink Verification and Validation license.

For an understanding of how to generate and interpret model coverage reports for your Stateflow charts, see the following topics:

- “Making Model Coverage Reports” on page 13-43 — Gives you an overview of how model coverage reports are generated and how they are interpreted.
- “Specifying Coverage Report Settings” on page 13-43 — Gives you the settings you need to specify for each available model coverage report option.
- “Cyclomatic Complexity” on page 13-44 — Explains the cyclomatic complexity results you see on model coverage reports.
- “Decision Coverage” on page 13-44 — Explains the decision coverage results you see on model coverage reports if you select it for the report.
- “Condition Coverage” on page 13-49 — Explains the condition coverage results you see on model coverage reports if you select it for the report.
- “MCDC Coverage” on page 13-49 — Explains the MCDC (modified condition decision coverage) results you see on model coverage reports if you select it for the report.
- “Coverage Reports for Stateflow Charts” on page 13-50 — Describes different parts of a model coverage report for Stateflow charts.
- “Colored Stateflow Diagram Coverage Display” on page 13-59 — Describes an option for displaying model coverage information directly in Stateflow diagrams with context-sensitive access.

Stateflow provides model coverage for other Stateflow objects that incorporate logical decisions in the following sections:

- “Model Coverage for Truth Tables” on page 10-58
- “Model Coverage for an Embedded MATLAB Function” on page 11-29

Making Model Coverage Reports

Model Coverage reports are generated during simulation if you specify them (see “Specifying Coverage Report Settings” on page 13-43). For Stateflow charts, the Model Coverage tool records the execution of the chart itself and the execution of its states, transition decisions, and the individual conditions that compose each decision. When simulation is finished, the Model Coverage report tells you how thoroughly a model has been tested, in terms of how many times each exclusive substate is entered, executed, and exited based on the history of the superstate, how many times each transition decision has been evaluated as true or false, and how many times each condition (predicate) has been evaluated as true or false.

Note You must have the Simulink Performance Tools option installed on your system to use the Model Coverage tool.

Specifying Coverage Report Settings

Coverage report settings appear in the **Coverage Settings** dialog. Access this dialog by selecting **Coverage settings** from the **Tools** menu in a Simulink model window.

Selecting the **Generate HTML Report** option on the **Coverage Settings** dialog causes Simulink to create an HTML report containing the coverage data generated during simulation of the model. Simulink displays the report in the MATLAB Help browser at the end of simulation.

Selecting the **Generate HTML Report** option also enables the selection of different coverages that you can specify for your reports. The following sections address only those coverage metrics that have direct bearing on reports for the Stateflow charts. These include decision coverage, condition coverage, and MCDC coverage. For a complete discussion of all dialog fields and entries, consult the “Specifying Model Coverage Reporting Options” section Simulink Verification and Validation documentation.

Cyclomatic Complexity

Cyclomatic complexity is a measure of the complexity of a software module based on its edges, nodes, and components within a control-flow graph. It provides an indication of how many times you need to test the module.

The calculation of cyclomatic complexity is as follows:

$$CC = E - N + p$$

where CC is the cyclomatic complexity, E is the number of edges, N is the number of nodes, and p is the number of components.

Within the Model Coverage tool, each decision is exactly equivalent to a single control flow node, and each decision outcome is equivalent to a control flow edge. Any additional structure in the control-flow graph is ignored since it contributes the same number of nodes as edges and therefore has no effect on the complexity calculation. This allows cyclomatic complexity to be reexpressed as follows:

$$CC = \text{OUTCOMES} - \text{DECISIONS} + p$$

For analysis purposes, each chart is considered to be a single component.

Decision Coverage

Decision coverage interprets a model execution in terms of underlying decisions where behavior or execution must take one outcome from a set of mutually exclusive outcomes.

Note Full coverage for an object of decision means that every decision has had at least one occurrence of each of its possible outcomes.

Decisions belong to an object making the decision based on its contents or properties. The following table lists the decisions recorded for model coverage for the Stateflow objects owning them. The sections that follow the table describe these decisions and their possible outcomes.

Object	Possible Decisions
Chart	<p>If a chart is a triggered Simulink block, it must decide whether or not to execute its block. See “Chart as a Triggered Simulink Block Decision” on page 13-45.</p> <p>If a chart contains exclusive (OR) substates, it must decide which of its states to execute. See “Chart Containing Exclusive OR Substates Decision” on page 13-46.</p>
State	<p>If a state is a superstate containing exclusive (OR) substates, it must decide which substate to execute. See “Superstate Containing Exclusive OR Substates Decision” on page 13-46.</p> <p>If a state has on <i>event name</i> actions (which might include temporal logic operators), the state must decide whether or not to execute the actions. See “State with On Event_Name Action Statement Decision” on page 13-48.</p>
Transition	<p>If a transition is a conditional transition, it must decide whether or not to exit its active source state or junction and enter another state or junction. See “Conditional Transition Decision” on page 13-49.</p>

Chart as a Triggered Simulink Block Decision

If the chart is a triggered block in Simulink, the decision to execute the block is tested. If the block is not triggered, there is no decision to execute the block, and the measurement of decision coverage is not applicable (NA).

See “Chart as Subsystem Details Report Section” on page 13-52.

Chart Containing Exclusive OR Substates Decision

If the chart contains exclusive (OR) substates, the decision on which substate to execute is tested. If the chart contains only parallel AND substates, this coverage measurement is not applicable (NA).

See “Chart as Superstate Details Report Section” on page 13-52.

Superstate Containing Exclusive OR Substates Decision

Since a diagram is hierarchically processed from the top down, procedures such as exclusive (OR) substate entry, exit, and execution are sometimes decided by the parenting superstate.

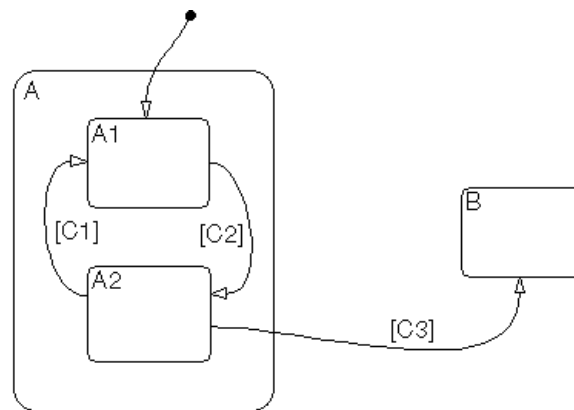
Note Decision coverage for superstates applies to exclusive (OR) substates only. A superstate makes no decisions for its parallel (AND) substates.

Since a superstate must decide which of its exclusive (OR) substates to process, the number of decision outcomes for the superstate is equal to the number of exclusive (OR) substates that it contains. In the examples following, the choice of which substate to process is made in one of three possible contexts.

Note Implicit transitions are shown as dashed lines in the following examples.

1 Active Call

In the following example, states A and A1 are active.



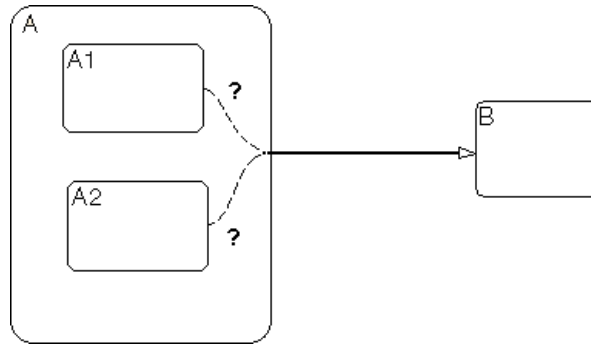
This gives rise to the following superstate/substate decisions:

- The parent of states A and B must decide which of these states to process. This decision belongs to the parent. Since A is active, it is processed.
- State A, the parent of states A1 and A2, must decide which of these states to process. This decision belongs to state A. Since A1 is active, it is processed.

During processing of state A1, all its outgoing transitions are tested. This decision belongs to the transition and not to its parent state A. In this case, the transition marked by condition C2 is tested and a decision is made whether to take the transition to A2 or not. See “Conditional Transition Decision” on page 13-49.

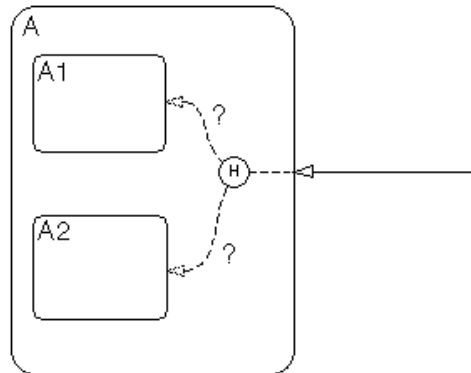
2 Implicit Substate Exit Context

In the following example, a transition takes place whose source is superstate A and whose destination is state B. If the superstate has two exclusive (OR) substates, it is the decision of superstate A as to which of these substates will perform the implicit transition from substate to superstate.



3 Substate Entry with a History Junction

A history junction, similar to the one shown in the example following, provides a superstate with the means of recording which of its substates was last active before the superstate was exited. If that superstate now becomes the destination of one or more transitions, the history junction provides it the means of deciding which previously active substate to enter.



See “State Details Report Section” on page 13-53.

State with On Event_Name Action Statement Decision

A state that has an on *event_name* action statement must decide whether to execute that statement based on the reception of a specified event or on an accumulation of the specified event when using temporal logic operators.

See “State Labels” on page 2-9 and “Using Temporal Logic in Actions” on page 7-50.

Conditional Transition Decision

A conditional transition is a transition with a triggering event and/or a guarding condition (see “Transition Label Notation” on page 2-14). In a conditional transition from one state to another, the decision to exit one state and enter another is credited to the transition itself.

See “Transition Details Report Section” on page 13-55.

Note Only conditional transitions receive decision coverage. Transitions without decisions are not applicable to decision coverage.

Condition Coverage

Condition coverage reports on the extent to which all possible outcomes are achieved for individual subconditions composing a transition decision.

Note Full condition coverage means that all possible outcomes occurred for each subcondition in the test of a decision.

For example, for the decision [A & B & C] on a transition, condition coverage reports on the true and false occurrences of each of the subconditions A, B, and C. This results in six possible outcomes: true and false for each of three subconditions.

See “Transition Details Report Section” on page 13-55.

MCDC Coverage

The Modified Condition Decision Coverage (MCDC) option reports a test's coverage of occurrences in which changing an individual subcondition within a transition results in changing the entire transition trigger expression from true to false or false to true.

Note If matching true and false outcomes occur for each subcondition, coverage is 100%.

For example, if a transition executes on the condition [C1 & C2 & C3 | C4 & C5], the MCDC report for that transition shows actual occurrences for each of the five subconditions (C1, C2, C3, C4, C5) in which changing its result from true to false is able to change the result of the entire condition from true to false.

See “Transition Details Report Section” on page 13-55.

Coverage Reports for Stateflow Charts

The following sections of a Model Coverage report were generated by simulating the Bang-Bang Boiler demonstration model, which includes the Stateflow Chart block Bang-Bang Controller. The coverage metrics for **Decision Coverage**, **Condition Coverage**, and **MCDC Coverage** are enabled for this report; the **Look-up Table Coverage** metric is Simulink dependent and not relevant to the coverage of Stateflow charts.

















This topic contains the following subtopics:

- “Summary Report Section” on page 13-51
- “Chart as Subsystem Details Report Section” on page 13-52
- “Chart as Superstate Details Report Section” on page 13-52
- “State Details Report Section” on page 13-53
- “Transition Details Report Section” on page 13-55

For information on the model coverage of truth tables, see “Model Coverage for Truth Tables” on page 10-58.

Summary Report Section

Summary

Model Hierarchy/Complexity:		Test 1					
		D1		C1		MCDC	
1. sf_boiler	18	100%		70%		40%	
2. ... Bang-Bang Controller	16	100%		70%		40%	
3. SF: Bang-Bang Controller	14	100%		70%		40%	
4. SF: Heater	11	100%		70%		40%	
5. SF: On	4	100%			NA		NA
6. SF: flash_LED	1	100%			NA		NA
7. SF: turn_boiler	1	100%			NA		NA
8. ... Boiler Plant model	1	100%			NA		NA

The Summary section shows coverage results for the entire test. It appears at the beginning of the Model Coverage report after the listing of the Start and End execution times for the test (simulation).

Each line in the hierarchy summarizes the coverage results at its level and the levels below it. It includes a hyperlink to a later section in the report with the same assigned hierarchical order number that details that coverage and the coverage of its children.

The top level, `sf_boiler`, is the model itself. The second level, Bang-Bang Controller, is the Simulink Stateflow chart block. The next levels are superstates within the Stateflow chart control logic in order of hierarchical containment. Each of these superstates uses an SF: prefix. The bottom level, Boiler Plant model, is an additional subsystem in the model.

Chart as Subsystem Details Report Section

2. Subsystem block "[Bang-Bang Controller](#)"

Parent: [/sf_boiler](#)
 Child Systems: [Bang-Bang Controller](#)

Metric	Coverage (this object)	Coverage (inc. descendants)
Cyclomatic Complexity	2	16
Decision (D1)	100% (2/2) decision outcomes	100% (24/24) decision outcomes
Condition (C1)	NA	70% (7/10) condition outcomes
MCDC (C1)	NA	40% (2/5) conditions reversed the outcome

The Subsystem report sees the chart as a block in a Simulink model, instead of a chart with states and transitions. You can confirm this by taking the hyperlink of the subsystem name in the title; it takes you to a highlighted Bang-Bang Controller Stateflow block sitting in its resident Simulink block diagram.

Chart as Superstate Details Report Section

3. Chart "[Bang-Bang Controller](#)"

Parent: [sf_boiler/Bang-Bang Controller](#)
 Child Systems: [flash_LED](#), [turn_boiler](#), [Heater](#)

Metric	Coverage (this object)	Coverage (inc. descendants)
Cyclomatic Complexity	1	14
Decision (D1)	100% (2/2) decision outcomes	100% (22/22) decision outcomes
Condition (C1)	NA	70% (7/10) condition outcomes
MCDC (C1)	NA	40% (2/5) conditions reversed the outcome

Decisions analyzed:

Substate executed	100%
State "Off"	571/699
State "On"	128/699

The Chart report sees a Stateflow chart as the superstate container of all of its states and transitions. You can confirm this through the hyperlinked chart name, which takes you to a display of the control logic chart in the Stateflow diagram editor.

Cyclomatic complexity and decision coverage are also displayed for the chart and for the chart including its descendants. Condition coverage and MCDC are both not applicable (NA) coverages for a chart, but apply to descendants.

State Details Report Section

5. State "[On](#)"

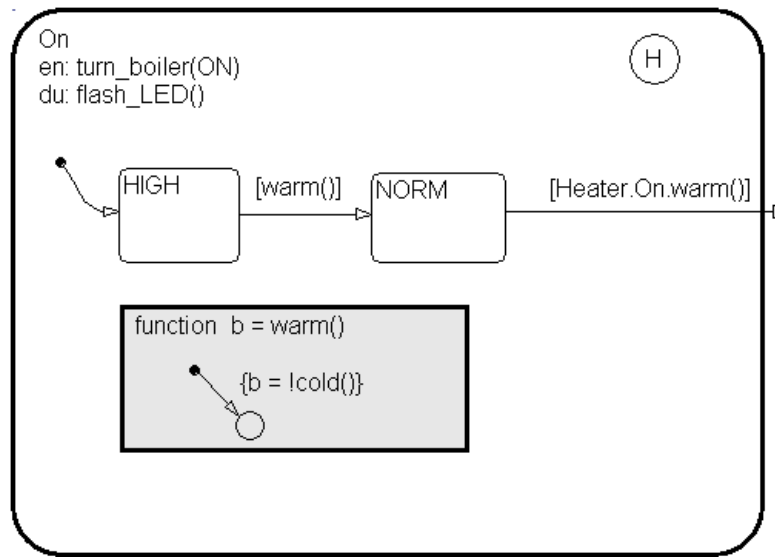
Parent: [sf_boiler/Bang-Bang_Controller.Heater](#)

Metric	Coverage (this object)	Coverage (inc. descendants)
Cyclomatic Complexity	3	4
Decision (D1)	100% (6/6) decision outcomes	100% (8/8) decision outcomes

Decisions analyzed:

Substate executed	100%
State "HIGH"	88/124
State "NORM"	36/124
Substate exited when parent exits	100%
State "HIGH"	4/14
State "NORM"	10/14
Previously active substate entered due to history	100%
State "HIGH"	4/13
State "NORM"	9/13

The example state section contains a report on the state On. The Stateflow diagram for On is as follows:



On resides in the box Heater, which has its own details report (not shown) because it contains other Stateflow objects. However, because On is a superstate containing the two states HIGH and NORM along with a history junction and the function warm, it has its own numbered report in the Details section.

The decision coverage for the On state tests the decision of which of its states to execute. The results indicate that six of a possible six outcomes were tested during simulation. Each decision is described as follows:

- 1 The choice of which substate to execute when On is executed
- 2 The choice of which state to exit when On is exited
- 3 The choice of which substate to enter when On is entered and the History junction has a record of the previously active substate

Because each of the above decisions can result in processing either HIGH or NORM, the total possible outcomes are $3 \times 2 = 6$.

The decision coverage tables also display the number of occurrences for each decision and the number of times each state was chosen. For example, the first decision was made 124 times. Of these, the HIGH state was executed 88 times and the NORM state was executed 36 times.

Cyclomatic complexity and decision coverage are also displayed for the On state including its descendants. This includes the coverage discussed above plus the decision required by the condition `[warm()]` for the transition from HIGH to NORM for a total of eight outcomes. Condition coverage and MCDC are both not applicable (NA) coverages for a state.

Note Nodes and edges that make up the cyclomatic complexity calculation have no direct relationship with model objects (states, transitions, and so on). Instead, this calculation requires a graph representation of the equivalent control flow.

Transition Details Report Section

Reports for transitions appear under the report sections of their owning states. They do not appear in the model hierarchy of the Summary section, since that is based entirely on superstates owning other Stateflow objects.

Transition "[after\(40,sec\) \[cold\(\)](#)" from "Off" to "On"

Parent: [sf_boiler/Bang-Bang_Controller.Heater](#)

Uncovered Links: 

Metric	Coverage
Cyclomatic Complexity	3
Decision (D1)	100% (2/2) decision outcomes
Condition (C1)	67% (4/6) condition outcomes
MCDC (C1)	33% (1/3) conditions reversed the outcome

Decisions analyzed:

Transition trigger expression	100%
false	557/571
true	14/571

Conditions analyzed:

Description:	#1 T	#1 F
Condition 1, "sec"	571	0
Condition 2, "after(40,sec)"	14	557
Condition 3, "cold()"	14	0

MC/DC analysis (combinations in parentheses did not occur)

Decision/Condition:	#1 True Out	#1 False Out
Transition trigger expression		
Condition 1, "sec"	TTT	(Fxx)
Condition 2, "after(40,sec)"	TTT	TFx
Condition 3, "cold()"	TTT	(TTF)

The decision for this transition is based on the broadcast of 40 sec events and the condition [cold ()]. If, after the reception of 40 sec events (equivalent to a 40 second delay) the environment is cold (cold () = 1), the decision to execute this transition and turn the Heater on is made. For other time intervals or environment conditions, the decision is made not to execute.

For decision coverage, both the true and false evaluations for the decision occurred. Because two of two decision outcomes occurred, coverage was full (that is, 100%).

Condition coverage shows that only 4 of 6 condition outcomes were tested. The temporal condition `after(40, sec)` is a short form expression for `sec[after(40, sec]` which is actually two conditions: the event `sec` and the accumulation condition `after(40, sec)`. Consequently, there are actually three conditions on the transition: `sec`, `after(40, sec)`, and `cold()`. Since each of these decisions can be true or false, there are now six possible outcomes.

A look at the **Decisions analyzed** table shows each of these conditions as a row with the recorded number of occurrences for each outcome for that decision (true or false). Decision rows in which a possible outcome did not occur are shaded. For example, the first and the third decision rows did not record an occurrence of a false outcome and are therefore shaded.

In the MC/DC report, all sets of occurrences of the transition conditions are scanned for a particular pair of decisions for each condition in which the following are true:

- The condition varies from true to false.
- All other conditions contributing to the decision outcome remain constant.
- The outcome of the decision varies from true to false, or the reverse.

For three conditions related by an implied AND operator, these criteria can be satisfied by the occurrence of the following conditions.

Condition Tested	True Outcome	False Outcome
1	TTT	Fxx
2	TTT	TFx
3	TTT	TTF

Notice that in each line, the condition tested changes from true to false while the other condition remains constant. Irrelevant contributors are coded with an “x” (discussed below). If both outcomes occur during testing, coverage is complete (100%) for the condition tested.

The preceding report example shows coverage only for condition 2. The false outcomes required for conditions 1 and 3 did not occur, and are indicated by parentheses for both conditions. Therefore the table lines for conditions (rows) 1 and 3 are shaded in red. Thus, while condition 2 has been tested, conditions 1 and 3 have not and MCDC is 33%.

For some decisions, the values of some conditions are irrelevant under certain circumstances. For example, in the decision [C1 & C2 & C3 | C4 & C5] the left side of the “|” is false if any one of the conditions C1, C2, or C3 is false. The same applies to the right side result if either C4 or C5 is false. When searching for matching pairs that change the outcome of the decision by changing one condition, holding some of the remaining conditions constant is irrelevant. In these cases, the MC/DC report marks these conditions with an “x” to indicate their irrelevance as a contributor to the result. This is shown in the following example.

Transition "[c1&c2&c3 | c4&c5]" . . .

MC/DC analysis (combinations in parentheses did not occur)

Decision/Condition:	#1 True Out	#1 False Out
Transition trigger expression		
Condition 1, "c1"	TTTxx	FxxFx
Condition 2, "c2"	TTTxx	TFxFx
Condition 3, "c3"	TTTxx	TTFFx
Condition 4, "c4"	FxxTT	FxxFx
Condition 5, "c5"	FxxTT	FxxTF

Consider the very first matched pair. Since condition 1 is true in the **True** outcome column, it must be false in the matching **False** outcome column. This makes the conditions C2 and C3 irrelevant for the false outcome since C1 & C2 & C3 is always false if C1 is false. Also, since the false outcome is required to evaluate to false, the evaluation of C4 & C5 must also be false. In this case, a match was found with C4 = F, making condition C5 irrelevant.

Colored Stateflow Diagram Coverage Display

The Model Coverage tool displays model coverage results for individual blocks directly in Simulink diagrams. If you enable this feature, the Model Coverage tool does the following:

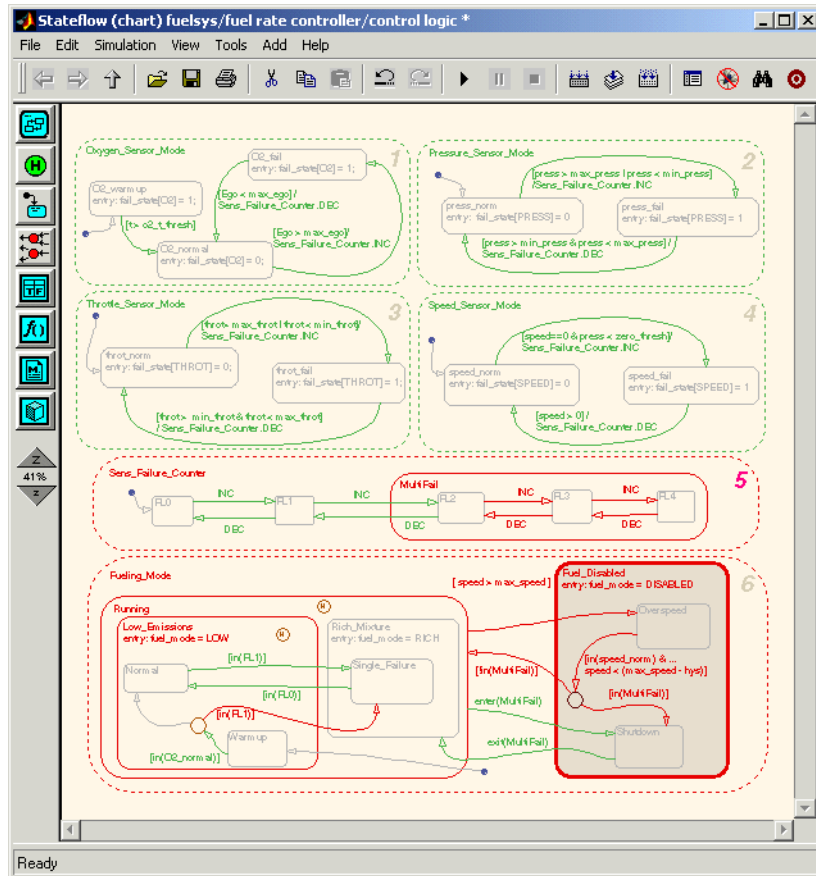
- Highlights (colors) Stateflow objects that have received model coverage during simulation
- Provides a context-sensitive display of summary model coverage information for each object

Caution The coverage tool only changes colors for open Stateflow at the time coverage information is reported. When you interact with the Stateflow diagram, such as selecting a transition or a state, colors revert to their default values.

For details on enabling and selecting this feature in Simulink, see “Enabling the Model Coverage Colored Diagram Display” in Simulink documentation.

Displaying Model Coverage with Model Coloring

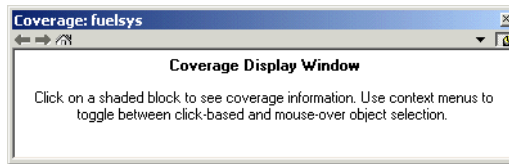
Once you enable display coverage with model coloring, anytime that the model generates a model coverage report, individual Stateflow objects receiving coverage are highlighted with light green or light red as shown in the following example:



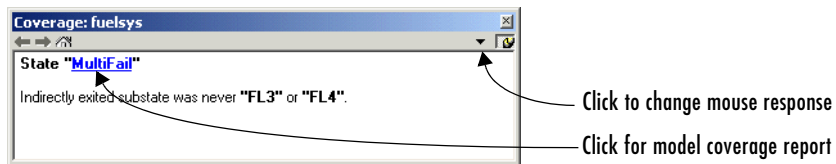
Objects highlighted in light green received full coverage during testing. Objects highlighted in light red received incomplete coverage. Objects with no color highlighting receive no coverage at all.

Note To revert the Stateflow diagram to show original colors, select and unselect its objects.

Along with the highlighted Stateflow diagram, a **Coverage Display Window** appears, as shown.



If you click a highlighted Stateflow object, its summarized coverage appears in the **Coverage Display Window**. In the preceding example, the following summary report appears when you click the MultiFail state:



Summary coverage information appears in the **Coverage Display Window** for the Stateflow object, whose hyperlinked name appears at the top of the window. Click the hyperlink to access the appropriate section of the coverage report for this object.

You can set the **Coverage Display Window** to appear for a block in response to a hovering mouse cursor instead of a mouse click in one of two ways:

- Select the downward arrow on right side of the **Coverage Display Window**, and, from the resulting menu, select **Focus**.
- Right-click a colored block and select **Coverage display on mouse-over** from the resulting context menu.

Exploring and Modifying Charts

Stateflow provides you with tools for searching for objects and replacing them with others. Learn how to search and replace objects in Stateflow in the following sections:

Using the Model Explorer with Stateflow Objects (p. 14-2)

Describes the Stateflow Explorer, a powerful tool for creating, displaying, modifying, and deleting Stateflow objects in a Simulink model.

Using the Stateflow Search & Replace Tool (p. 14-12)

Stateflow Search & Replace tool searches for and replaces text belonging to objects in Stateflow charts.

Using the Stateflow Finder Tool (p. 14-27)

Stateflow provides the Stateflow Finder tool on platforms that do not support the Simulink Find tool. This section describes how you use the Stateflow Finder tool to search for objects in Stateflow.

Using the Model Explorer with Stateflow Objects

The **Model Explorer** displays any object in the Stateflow hierarchy. You can also use the **Model Explorer** as a platform for creating, modifying, and deleting Stateflow objects. You can also display, create, modify, and delete target objects for generating code and building the simulation application in the **Model Explorer**.


You can create data, events, and targets in the Stateflow diagram editor and in the **Model Explorer**. However, the **Model Explorer** is the only location where you can modify and delete existing data, events, and targets.

The following topics describe the use of the **Model Explorer** for creating, modifying, and deleting Stateflow objects:

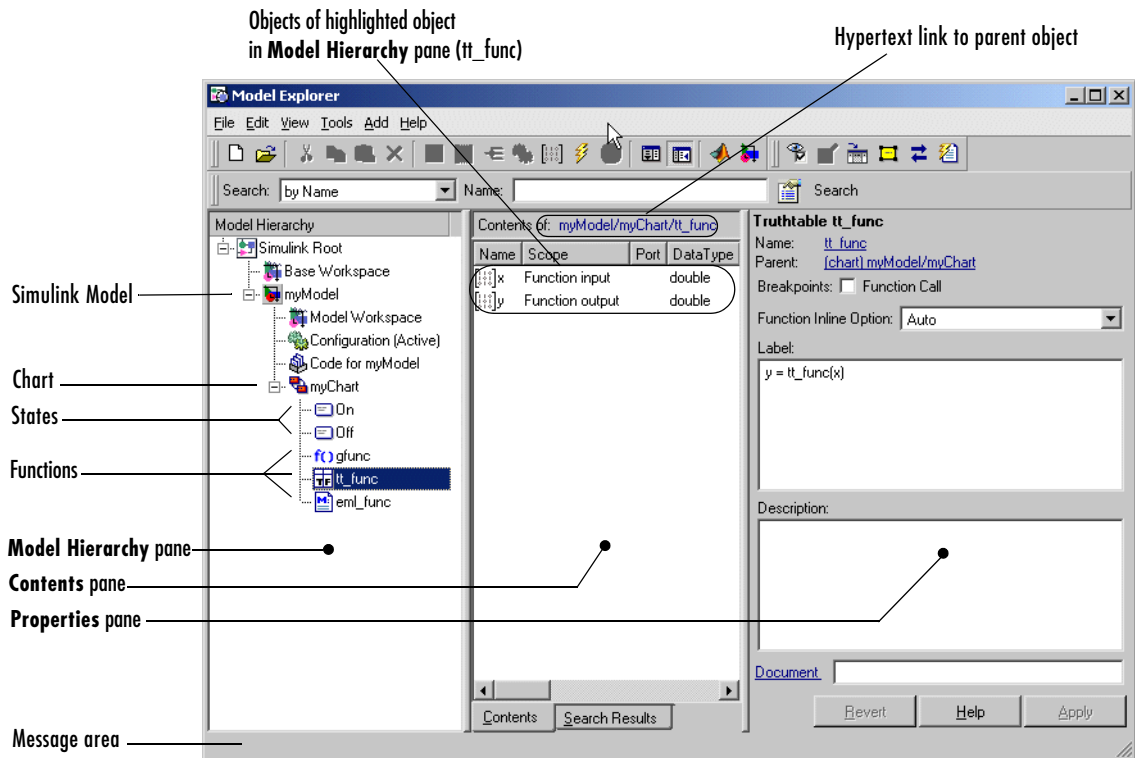
- “Viewing Stateflow Objects in the Model Explorer” on page 14-3 — Tells you how to open the Stateflow Explorer in the Stateflow diagram editor.
- “Editing States or Charts in the Model Explorer” on page 14-5 — Shows you how to edit states and charts from the Stateflow Explorer.
- “Adding Data and Events in the Model Explorer” on page 14-6 — Shows you how to create data in the **Model Explorer** and objects and which of their properties are displayed in the Stateflow Explorer.
- “Adding a Target in the Model Explorer” on page 14-6 — Describes target objects as they appear in the Stateflow Explorer.
- “Setting Properties for Stateflow Objects in the Model Explorer” on page 14-8 — Shows you how to set the properties of data, events, and targets from the Stateflow Explorer.
- “Moving and Copying Data, Events, and Targets in the Model Explorer” on page 14-9 — Shows you how to move and copy events, data, and targets to different owning objects in Stateflow Explorer.
- “Changing the Port Order of Input and Output Data and Events” on page 14-10 — Shows you how to change the order of input and output data and event ports as they appear on the Stateflow block in the Simulink model.
- “Deleting Data, Events, and Targets in the Model Explorer” on page 14-11 — Shows you how to delete events, data, and targets in the Explorer.

Viewing Stateflow Objects in the Model Explorer

Depending on what you are editing in Stateflow, you can use one of the following methods for opening **Model Explorer**:

- From the toolbar menu of the Stateflow diagram editor, truth table editor, or Embedded MATLAB Editor, select **Explore** .
- From the **Tools** menu of the Stateflow diagram editor or truth table editor, select **Explore**.
- Right-click an empty area in the Stateflow diagram. From the resulting pop-up menu, select **Explore**.

The **Model Explorer** window appears similar to the following:



The Explorer main window has two panes: a **Model Hierarchy** pane on the left and a **Contents** pane on the right. When you open the **Model Explorer**, the object you are editing in Stateflow (chart, truth table, or Embedded MATLAB function) is highlighted in the **Model Hierarchy** pane and its objects are displayed in the **Contents** pane. In the preceding example, the **Model Explorer** was opened from the truth table editor for the truth table `tt_func` in the Stateflow chart `myChart`.

The **Model Hierarchy** pane displays the elements of all loaded Simulink models, which includes Stateflow charts, and their states, boxes, and functions. A preceding plus (+) character for an object indicates that you can expand the display of its child objects by double-clicking the entry or by clicking the plus (+). A preceding minus (-) character for an object indicates that it has no child objects.

Clicking an entry in the **Model Hierarchy** pane selects that entry and displays its child objects in the **Contents** pane. For convenience, a hypertext link to the currently selected object in the **Model Hierarchy** pane is included following the **Contents of:** label at the top of the **Contents** pane. Click this link to display that object in its native editor. In the preceding example, selecting the link

```
(Stateflow.TruthTable) myModel/myChart/myChart/tt_func
```













displays the truth table `tt_func` in the truth table editor.

By default, the **Model Explorer** displays event and data child objects in the **Contents** pane for the selected object in the **Model Hierarchy** pane. To display additional or different child Stateflow objects in the **Contents** pane, do the following:

- 1** From the **Model Explorer View** menu, select **List View Options**.
- 2** In the resulting submenu, select any or all of the following individual options: **States**, **Transitions**, **Junctions**, **Events**, or **Data**.

To display all of the preceding Stateflow child objects, select **All Stateflow Objects**.

Each type of object, whether in the **Object Hierarchy** or **Contents** pane, is displayed with an adjacent icon. Objects that are subcharted (states, boxes, and graphical functions) have their appearance altered by shading.

Object	Icon	Icon for Subcharted Object
Chart		Not applicable
State		
Box		
Graphical Function		
Truth Table Function		Not applicable
Embedded MATLAB Function		Not applicable
Data		Not applicable
Event		Not applicable
Target		Not applicable

The display of child objects in the **Contents** pane includes properties for each object, most of which are directly editable. You can also access the properties dialog for an object from the **Model Explorer**. See “Setting Properties for Stateflow Objects in the Model Explorer” on page 14-8 for more details.

Editing States or Charts in the Model Explorer

To edit a state or chart displayed in the Explorer’s **Object Hierarchy** pane, do the following:

- 1 Right-click the object.
- 2 Select **Edit** from the resulting menu.

Stateflow displays the selected object highlighted in the Stateflow editor in the context of its parent.

Adding Data and Events in the Model Explorer

State, box, and function Stateflow objects can parent data and events. You can also add data and events to the Simulink model to make them globally available to all Stateflow objects in the model.

To add a data or an event to a Stateflow object or to the Simulink model, do the following:

- 1 In the **Model Hierarchy** pane of the **Model Explorer**, select a Simulink model or a Stateflow object.
- 2 From the **Add** menu, select **Data** or **Event**.

A data or event is added to the **Model Explorer Contents** pane with the default name data or event. If you continue adding more data, each new data or event is named with an integer suffix (data1, event1, data2, event2, and so on).

You can change the displayed properties for a data or event directly in the **Model Explorer**. You can also access the complete list of properties for a data or event from the **Model Explorer**. See “Setting Properties for Stateflow Objects in the Model Explorer” on page 14-8.

For more detailed examples of creating data and events in the **Model Explorer**, see “Adding Events in the Model Explorer” on page 6-4 and “Adding Data in the Model Explorer” on page 6-22.

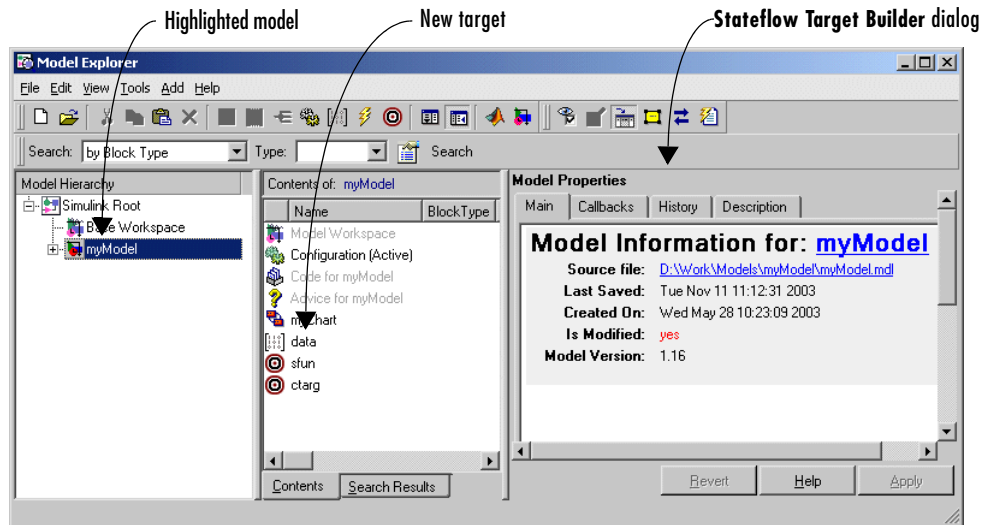
Adding a Target in the Model Explorer

Targets are parented exclusively by a Simulink model. A permanent simulation target (sfun) is automatically created for a model when you create the model. You can add an RTW target or multiple custom targets to a model in the **Model Explorer** as follows:

- 1 In the **Model Explorer**, in the left **Model Hierarchy** pane, select the Simulink model to receive the target.

2 From the Explorer's **Add** menu, select **Stateflow Target**.

The **Contents** pane of the **Model Explorer** displays the existing default simulation target `sfun` and the new custom target with the default name `untitled`.



3 In the **Stateflow Target Builder** dialog pane on the right, enter the name of the target and other properties. Click **Apply** when finished.

The simulation target for the model has the name `sfun`. The RTW target for a model has the name `rtw`. Custom targets have names other than `sfun` and `rtw`.

The properties you enter for the target depend on the kind of target you create. See “How Do You Build a Target?” on page 12-5 for a guide.

Renaming Objects in the Model Explorer

Use the following steps to rename a state, box, function, data, event, or target objects in the **Model Explorer**:

- 1 Right-click the object row in the **Contents** pane of the Explorer.

A pop-up menu appears.

- 2 From the resulting pop-up menu, select **Rename**.

The Explorer redisplay the name of the selected object in a text edit box that overlays the Name property for the object row.

- 3 Change the target's name in the edit box and click outside the edit box.

You can also change the name of an object in the **Model Explorer** by changing the value of its Name property. See “Setting Properties for Stateflow Objects in the Model Explorer” on page 14-8 for details.

Setting Properties for Stateflow Objects in the Model Explorer

To change one of the displayed properties of a displayed object in the **Contents** pane of the **Model Explorer**, do the following:

- 1 In the **Contents** pane, click anywhere in the row of the displayed object.

This highlights the row.

- 2 Click an individual entry for a property column in the highlighted row.
 - For text properties, such as the Name property, a text editing field with the current text value overlays the displayed value. Edit the field and press the **Return** key or click anywhere outside the edit field to apply the changes.
 - For properties with enumerated entries, such as the Scope, Trigger, or Type properties, select from a drop-down combo box that overlays the displayed value.
 - For Boolean properties (properties that are set on or off) check or uncheck the check box that appears in place of the displayed value.

To set all the properties for an object displayed in the **Model Hierarchy** or **Contents** pane of the **Model Explorer**, do the following:

- 1 Right-click the object.
- 2 Select **Properties** from the resulting menu.
The properties dialog for the object appears.
- 3 Edit the appropriate properties and select **Apply** or **OK** to apply the changes.

To display the property dialog dynamically for the selected object in the **Model Hierarchy** or **Contents** panes of the **Model Explorer**, do the following:

- 1 From the **View** menu, select **Show Dialog View**.

The property dialog for the selected object appears in the far right pane of the **Model Explorer**.

Moving and Copying Data, Events, and Targets in the Model Explorer

Note If you move an object to a level in the hierarchy that does not support the **Scope** property for that object, the **Scope** is automatically changed to **Local**.

You can move data, event, or target objects to another parent by doing the following:

- 1 Select the data, event, or target to move in the **Contents** pane of the Explorer.

You can select a contiguous block of items by highlighting the first (or last) item in the block and then using **Shift+click** for highlighting the last (or first) item.

- 2 Click and drag the highlighted objects from the **Contents** pane to a new location in the **Model Hierarchy** pane to change its parent.

A shadow copy of the selected objects accompanies the mouse cursor during dragging. If no parent is chosen or the parent chosen is the current parent, the mouse cursor changes to an X enclosed in a circle, indicating an invalid choice.

You can accomplish the same outcome by cutting or copying the selected events, data, and targets as follows:

- 1 Select the event, data, and targets to move in the **Contents** pane of the Explorer.
- 2 From the **Edit** menu of the Explorer, select **Edit -> Cut** or **Copy**.

If you select **Cut**, the selected items are deleted and are copied to the clipboard for copying elsewhere. If you select **Copy**, the selected items are left unchanged.

You can also right-click a single selection and select **Cut** or **Copy** from the resulting menu. Explorer also uses the keyboard equivalents of **Ctrl+X** (Cut) and **Ctrl+C** (Copy).

- 3 Select a new parent machine, chart, or state in the **Model Hierarchy** pane of the **Model Explorer**.
- 4 From the **Edit** menu of the Explorer, select **Edit -> Paste**. The cut items appear in the **Contents** pane of the Explorer.

You can also paste the cut items by right-clicking an empty part of the **Contents** pane of the Explorer and selecting **Paste** from the resulting menu. Explorer also uses the keyboard equivalents of **Ctrl+V** (Paste).

Changing the Port Order of Input and Output Data and Events

Input data, output data, input events, and output events each have numerical sequences of port index numbers. You can change the order of indexing for event or data objects with a scope of **Input to Simulink** or **Output to Simulink** in the **Contents** pane of the **Model Explorer** as follows:

- 1 Select one of the input or output data or event objects.
- 2 Click the **Port** property for the object.
- 3 Enter a new value for the Port property for the object.

The remaining objects in the affected sequence are automatically assigned a new value for their **Port** property.

Deleting Data, Events, and Targets in the Model Explorer

Delete event, data, and target objects in the **Contents** pane of the **Model Explorer** as follows:

- 1 Select the object.
- 2 Press the **Delete** key.

You can also select **Cut** from the **Edit** menu or **Ctrl+X** from the keyboard to delete an object.

Using the Stateflow Search & Replace Tool

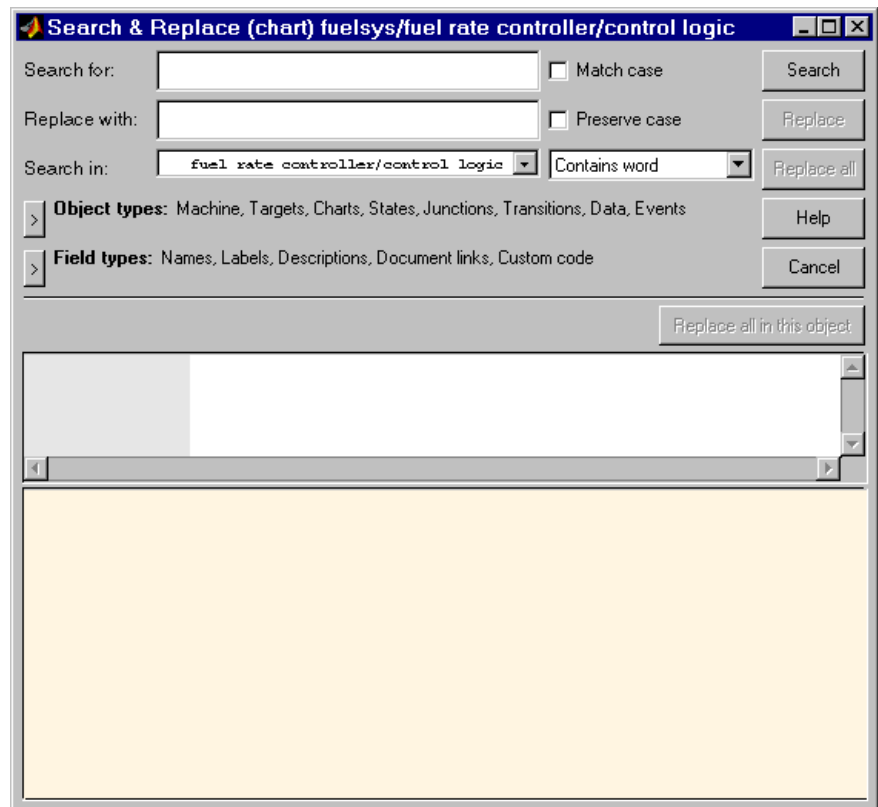
Based on textual criteria that you specify, the Stateflow Search & Replace tool searches for and replaces text belonging to objects in Stateflow charts. The following topics describe the Stateflow Search & Replace tool:

- “Opening the Search & Replace Tool” on page 14-12 — Tells you how to open the Search & Replace tool and describes the parts of the resulting dialog.
- “Using Different Search Types” on page 14-15 — Describes the different types of text searches available in the Search & Replace tool.
- “Specify the Search Scope” on page 14-17 — Tells you how to specify the parts of the model that you want to search.
- “Using the Search Button and View Area” on page 14-19 — Tells you how to use the Search button and how to interpret the display of found objects.
- “Specifying the Replacement Text” on page 14-22 — Tells you what you can specify for replacement text for the objects found by text.
- “Using the Replace Buttons” on page 14-23 — Interprets the replace buttons available and describes their use in text replacement.
- “Search and Replace Messages” on page 14-24 — Describes the text and defining icon for the informational and warning messages that appear in the Full Path Name Containing Object field.

Opening the Search & Replace Tool

To display the **Search & Replace** dialog box, do the following:

- 1** Open a Stateflow chart in the Stateflow chart editor.
- 2** Select **Search & Replace** from the Stateflow Editor’s **Tools** menu.



The window name for the **Search & Replace** dialog box contains a full path expression for the current Stateflow chart or machine in the following form.

(object) Machine/Subsystem/Chart

The **Search & Replace** dialog box contains the following fields:

- **Search for**

Enter search pattern text in the **Search for** text box. Interpretation of the search pattern is selected with the **Match case** check box and the **Match Options** field.

- **Match case**

If this check box is selected, the search is case sensitive and the Search & Replace tool finds only text matching the search pattern exactly. See “Match case (Case Sensitive)” on page 14-15.

- **Replace with**

Specify the text to replace the text found when you select any of the **Replace** buttons (**Replace**, **Replace All**, **Replace All in This Object**). See “Using the Replace Buttons” on page 14-23.

- **Preserve case**

This option modifies replacement text. For an understanding of this option, see “Replacing with Case Preservation” on page 14-22.

- **Search in**


By default, the Search & Replace tool searches for and replaces text only within the current Stateflow chart that you are editing in the Stateflow chart editor. You can select to search the machine owning the current Stateflow chart or any other loaded machine or chart by accessing this selection box.


- **Match options**

This field is unlabeled and just to the right of the **Search in** field. You can modify the meaning of your search text by entering one of the selectable search options. See “Using Different Search Types” on page 14-15.

- **Object types and Field types**

Under the **Search in** field are the selection boxes for **Object types** and **Field types**. These selections further refine your search and are described below. By default, these boxes are hidden; only current selections are displayed next to their titles.

Select the right-facing arrow button  in front of the title to expand a selection box and make changes.

Select the same button (this time with a left-facing arrow)  to compress the selection box to display the settings only, or, if you want, just leave the box expanded.

- **Search and Replace buttons**

These are described in “Using the Search Button and View Area” on page 14-19 and “Using the Replace Buttons” on page 14-23.

- **View Area**

The bottom half of the **Search & Replace** dialog box displays the result of a search. This area is described in “A Breakdown of the View Area” on page 14-20.

Using Different Search Types

Enter search pattern text in the **Search for** text box. You can use one of the following settings in the **Match options** field (unlabeled and just to the right of the **Search in** field) to further refine the meaning of the text entered.

Contains word

Select this option to specify that the search pattern text is a whole word expression used in a Stateflow chart with no specific beginning and end delimiters. In other words, find the specified text in any setting.

The following example is taken from the Sensor Failure Detection demo model.

```
throt_fail
entry: fail_state[THROT] = 1;
```

Searching for the string `fail` with the **Contains word** option set finds both occurrences of the string `fail`.

Match case (Case Sensitive)

By selecting the **Match case** option, you enable case-sensitive searching. In this case, the Search & Replace tool finds only text matching the search pattern exactly.

By clearing the **Match case** option, you enable case-insensitive searching. In this case, search pattern characters entered in lower- or uppercase find matching text strings with the same sequence of base characters in lower- or uppercase. For example, the search string "AnDrEw" finds the matching text "andrew" or "Andrew" or "ANDREW".

Match whole word

Select this option to specify that the search pattern text in the **Search for** field is a whole word expression used in a Stateflow chart with beginning and end delimiters consisting of a blank space or a character that is not alphanumeric and not an underscore character (`_`).

In the preceding example from the Sensor Failure Detection demo model, if **Match whole word** is selected, searching for the string `fail` finds no text within the above state. However, searching for the string `"fail_state"` does find the text `"fail_state"` as part of the second line since it is delimited as a word by a space on the front and a left square bracket (`[`) on the back.

Regular expression

Set the **Match options** field to **Regular expression** to search for text that varies from character to character within defined limits.

A regular expression is a string composed of letters, numbers, and special symbols that defines one or more string candidates. Some characters have special meaning when used in a regular expression, while other characters are interpreted as themselves. Any other character appearing in a regular expression is ordinary, unless a backslash (`\`) character precedes it.

If the **Match options** field is set to **Regular expression** in the preceding example from the Sensor Failure Detection demo model, searching for the string `"fail_"` matches the `"fail_"` string that is part of the second line, character for character. Searching with the regular expression `"\w*_"` also finds the string `"fail_"`. This search string uses the regular expression shorthand `"\w"` that represents any part-of-word character, an asterisk (`*`), which represents any number of any characters, and an underscore (`_`), which represents itself.

For a list of regular expression metacharacters, see the topic “Regular Expressions” in MATLAB documentation.

Searching with Regular Expression Tokens

Within a regular expression, you use parentheses to group characters or expressions. For example, the regular expression `"and(y|rew)"` matches the text `"andy"` or `"andrew"`. Parentheses also have the side effect of remembering what they match so that you can recall and reuse the found text with a special variable in the **Search for** field. These are referred to as *tokens*.

For an understanding of how to use tokens to enhance searching in the Search & Replace tool, see the topic “Tokens” in MATLAB documentation.

You can also use tokens in the **Replace with** field. See “Replacing with Tokens” on page 14-23 in for a description of using regular expression tokens for replacing.

Preserve case

This option actually modifies replacement text and not search text. For an understanding of this option, see “Replacing with Case Preservation” on page 14-22.

Specify the Search Scope

You specify the search scope for your search by selecting from the field regions discussed in the following topics:

- “Search in” on page 14-17 — Select a whole machine or individual Stateflow chart for searching.
- “Object Types” on page 14-18 — Limit your search to text matches in the selected object types.
- “Field Types” on page 14-18 — Limit your search to text matches for the specified fields

Search in

You can select a whole machine or individual Stateflow chart for searching in the **Search in** field. By default, the current Stateflow chart in which you entered the Search & Replace tool is selected.

To select a machine, do the following:

- 1 Select the down arrow of the **Search in** field.

A list of the currently loaded machines appears with the current machine expanded to reveal its underlying Stateflow charts.

- 2 Select a machine.

To select a Stateflow chart for searching, do the following:

- 1 Select the down arrow of the **Search in** field again.

This time the displayed list contains the previously selected machine expanded to reveal its Stateflow charts.

- 2 Select a chart from the expanded machine.

Object Types

Limit your search to text matches in the selected object types only when you do the following:

- 1 Expand the **Object types** field.
- 2 Select one or more object types.

Field Types

Limit your search to text matches for the specified fields only by doing the following:

- 1 Expand the **Field types** field.
- 2 Select one or more of the available field types

Available field types are as follows:

Names. Machines, charts, data, and events have valid **Name** fields. States have a **Name** defined as the top line of their labels. You can search and replace text belonging to the **Name** field of a state in this sense. However, if the Search & Replace tool finds matching text in a state's **Name** field, the remainder of the label is subject to succeeding searches for the specified text whether or not the label is chosen as a search target.

Note The **Name** field of machines and charts is an invalid target for the Search & Replace tool. Use Simulink to change the names of machines and charts.

Labels. Only states and transitions have labels.

Descriptions. All objects have searchable **Description** fields.

Document links. All objects have searchable **Link** fields.

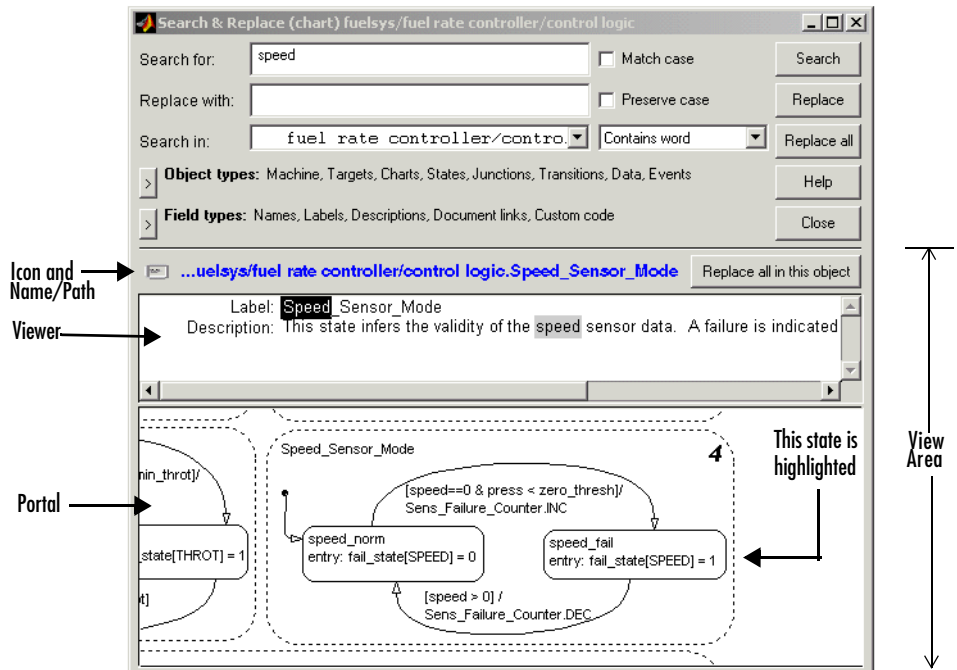
Custom code. Only target objects contain custom code.

Using the Search Button and View Area

This topic contains the following subtopics:

- “A Breakdown of the View Area” on page 14-20
- “The Search Order” on page 14-21
- “Additional Display Options” on page 14-21

Click **Search** to initiate a single-search operation. If an object match is made, its text fields are displayed in the **Viewer** pane in the middle of the **Search & Replace** dialog. If the object is graphical (state, transition, junction, chart), the matched object is displayed in a **Portal** pane below the **Viewer** pane.



A Breakdown of the View Area

The view area of the **Search & Replace** dialog box displays found text and its containing object, if viewable. In the preceding example, taken from the Sensor Fuel Detection demo model, a search for the word "speed" finds the **Description** field for the state Speed_Sensor_Mode. The resulting view area display consists of the following parts:

Icon. Displays an icon appropriate to the object containing the found text. These icons are identical to the icons used in the **Model Explorer** to represent Stateflow objects displayed in "Viewing Stateflow Objects in the Model Explorer" on page 14-3.

Full Path Name of Containing Object. This area displays the full path name for the object containing the found text in the following format:

```
<type> <machine name>/<subsystem>/<chart  
name>.[p1]...[pn].<object name> (<id>)
```

where p_1 through p_n denote the object's parent states.

To display the object, click the mouse once on the full path name of the object. If the object is a graphical member of a Stateflow chart, it is displayed in the Stateflow chart editor. Otherwise, it is displayed as a member of its Stateflow chart in the Stateflow Explorer.

Viewer. This area displays the found text as a highlighted part of all search-qualified text fields for the owner object. If other occurrences exist in these fields, they too are highlighted, but in lighter shades.

To invoke the **Properties** dialog box for the owner object, double-click anywhere in the view area.

Portal. This area contains a graphic display of the object containing the matching text. The object containing the found text is highlighted in blue (default).

To display the highlighted object in the Stateflow chart editor window, double-click anywhere in the portal.

The Search Order

If you specify an entire machine as your search scope in the **Search in** field, the Search & Replace tool starts searching at the beginning of the first chart of the model, regardless of the Stateflow chart displayed in the Stateflow chart editor when you begin your search. After searching the first chart, the Search & Replace tool continues searching each chart in model order until all charts for the model have been searched.

If you specify a Stateflow chart as your search scope, the Search & Replace tool begins searching at the beginning of the chart. The Search & Replace tool continues searching the chart until all the chart's objects have been searched.

The search order taken in searching an individual chart for matching text is equivalent to a depth-first search of the Stateflow Explorer. Starting at the highest level of the chart, the Explorer hierarchy is traversed downward from parent to child until an object with no child is encountered. At this point, the hierarchy is traversed upward through objects already searched until an unsearched sibling is found and the process is repeated.

Additional Display Options

Right-click anywhere in the **Search & Replace** dialog to display a menu with the following selections.

Selection	Result
Show Portal	A toggle switch that hides or displays the portal.
Edit	Displays the object with the matching text in the Stateflow chart editor. Applies to states, junctions, transitions, and charts.
Explore	Displays the object with the matching text in the Stateflow Explorer. Applies to states, data, events, machines, charts, and targets.
Properties	Displays the Properties dialog box for the object with the matching text.

Note The **Edit**, **Explore**, and **Properties** selections are enabled only after a successful search.

If the portal is not visible, you can select the **Show Portal** option to display it. You can also simply click and drag the border between the viewer and the portal (the cursor turns to a vertical double arrow), which resides just above the bottom boundary of the **Search & Replace** dialog. Moving this border allows you to exchange area between the portal and the viewer. If you click and drag the border with the left mouse button, the graphic display resizes after you reposition the border. If you click and drag the border with the right mouse button, the graphic display continuously resizes as you move the border.

Specifying the Replacement Text

The Search & Replace tool replaces found text with the exact (case-sensitive) text you entered in the **Replace With** field unless you choose one of the dynamic replacement options described below.

Replacing with Case Preservation

If you choose the **Case Preservation** option, matched text is replaced based on one of the following conditions discovered in the found text:

- **Whisper**

In this case, the found text has no uppercase characters, only lowercase. Found text is replaced entirely with the lowercase equivalent of all replacement characters. For example, if the replacement text is "ANDREW", the found text "bill" is replaced by "andrew".

- **Shout**

In this case, the found text contains only uppercase characters. Found text is replaced entirely with the uppercase equivalent of all replacement characters. For example, if the replacement text is "Andrew", the found text "BILL" is replaced by "ANDREW".

- Proper

In this case, the found text contains uppercase characters in the first character position of each word. Found text is replaced entirely with the case equivalent of all replacement characters. For example, if the replacement text is "andrew johnson", the found text "Bill Monroe" is replaced by "Andrew Johnson".

- Sentence

In this case, the found text contains an uppercase character in the first character position of a sentence with all remaining sentence characters in lowercase. Found text is replaced in like manner, with the first character of the sentence given an uppercase equivalent and all remaining sentence characters set to lowercase. For example, if the replacement text is "andrew is tall.", the found text "Bill is tall." is replaced by "Andrew is tall.".

Replacing with Tokens

Within a regular expression, you use parentheses to group characters or expressions. For example, the regular expression "and(y|rew)" matches the text "andy" or "andrew". Parentheses also have the side effect of remembering what they matched so that you can recall and reuse the found text with a special variable in the **Replace with** field. These are referred to as *tokens*.

Tokens outside the search pattern have the form \$1, \$2, . . . , \$n (n<17) and are assigned left to right from parenthetical expressions in the search string.

For example, the search pattern "(\w*)_(\w*)" finds all word expressions with a single underscore separating the left and right sides of the word. If you specify an accompanying replacement string of "\$2_\$1", you can replace all these expressions by their reverse expression with a single **Replace all**. For example, the expression "Bill_Jones" is replaced by "Jones_Bill" and the expression "fuel_system" is replaced by "system_fuel".

For a clearer understanding of how tokens are used in regular expression search patterns, see “Regular Expressions” in MATLAB documentation.

Using the Replace Buttons

You can activate the replace buttons (**Replace**, **Replace All**, **Replace All in This Object**) only after a search that finds text.

Replace

When you select the **Replace** button, the current instance of text matching the text string in the **Search for** field is replaced by the text string entered in the **Replace with** field. The Search & Replace tool then automatically searches for the next occurrence of the **Search for** text string.

Replace All

When you select the **Replace All** button, all instances of text matching the **Search for** field are replaced by the text string entered in the **Replace with** field. Replacement starts at the point of invocation to the end of the current Stateflow chart. This means that if you initially skip through some search matches with the **Search** button, they are also skipped when you select the **Replace All** button.

If the search scope is set to **Search Whole Machine**, then after finishing the current Stateflow chart, replacement continues to the completion of all remaining charts in your Simulink model.

Replace All in This Object

When you select the **Replace All in This Object** button, all instances of text matching the **Search for** field are replaced by text entered in the **Replace with** field everywhere in the current Stateflow object regardless of previous searches.

Search and Replace Messages

Informational and warning messages appear in the **Full Path Name Containing Object** field along with a defining icon.



- Informational Messages



- Warnings

The following messages are informational only:

Please specify a search string

A search was attempted without a search string specified.

No Matches Found

There are no matches within the selected search scope.

Search Completed

There are no more matches within the selected search scope.

The following messages are warnings that refer to invalid conditions for searching or replacing:

Invalid option set

The object types and field types that you have selected are incompatible. For example, a search on **Custom Code** fields without selecting target objects is invalid.

Match object not currently editable

The found object is not editable by replacement because of one of the following.

Problem	Solution
A simulation is running.	Stop the simulation.
You are editing a locked library block.	Unlock the library.
The current object or its parent has been manually locked.	Unlock the object or its parent.

The following messages are warnings that, when the Search & Replace tool performs a search or replacement immediately after finding an object, it must first refind the object and its matching text field. If that original found object is deleted or changed before an ensuing search or replacement, the Search & Replace tool cannot continue:

Search object not found

If you search for text, find it, and then delete the containing object, this warning results if you continue to search.

Match object not found

If you search for text, find it, and then delete the containing object, this warning results if you perform a replacement.

Match not found

If you search for text, find it, and then change the object containing the text, this warning results if you perform a replacement.

Search string changed

If you search for text, find it, and then change the **Search For** field, this warning results if you perform a replacement.

Using the Stateflow Finder Tool

There are two varieties of tools that search only for Stateflow, depending on your platform. These are as follows:

- On most platforms, when you select **Find** from the Stateflow Editor's **Tools** menu, the Simulink **Find** dialog appears. This tool allows you to search Stateflow models for Simulink and Stateflow objects, such as states and transitions, that meet criteria you specify. Simulink displays any objects that satisfy the search criteria in the dialog box's search results pane.
- On platforms that do not support the Simulink Find tool, the original Stateflow Finder appears when you select **Find** from the Stateflow Editor's **Tools** menu. The following topics explain how to use the original Stateflow Finder to search for objects.
 - "Opening Stateflow Finder" on page 14-27 — Tells you how to open the Stateflow Finder tool.
 - "Using Stateflow Finder" on page 14-28 — Describes the fields and selections available for describing the objects you want to find.
 - "Finder Display Area" on page 14-31 — Describes the display columns for found items in the display area.

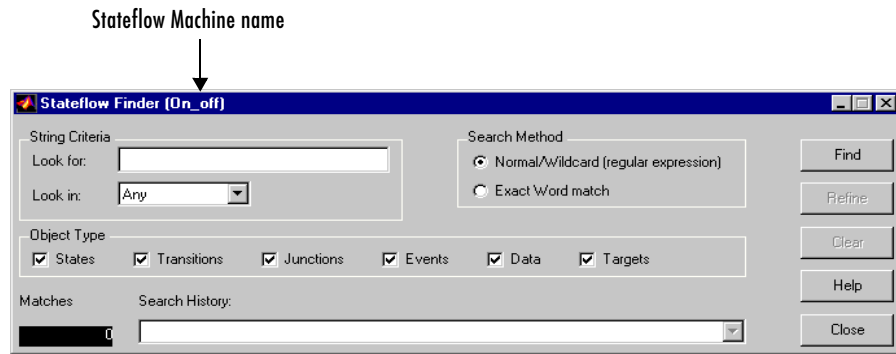
Note See the Simulink Release Notes in the online documentation for a list of platforms on which the Simulink Find tool is not available.

Opening Stateflow Finder

On platforms that do not support the Simulink Find tool (see preceding note), display the Stateflow **Finder** dialog box with one of the following:

- Select **Find** from the Stateflow Editor's **Tools** menu.
- Select **Find** from the Simulink model window's **Edit** menu.

The Finder operates on the machine whose name appears in the window title area of the **Finder** dialog as shown:



Using Stateflow Finder

The following topics in this section describe the parts of the Stateflow Finder:

- “String Criteria” on page 14-28
- “Search Method” on page 14-29
- “Object Type” on page 14-30
- “Find Button” on page 14-30
- “Matches” on page 14-30
- “Refine Button” on page 14-30
- “Search History” on page 14-30
- “Clear Button” on page 14-31
- “Close Button” on page 14-31
- “Help Button” on page 14-31

String Criteria

You specify the string by entering the text to search for in the **Look for** text box. The search is case sensitive. All text fields are included in the search by default. Alternatively, you can search in specific text fields by using the **Look in** list box to choose one of these options:

Any. Search the state and transition labels, object names, and descriptions of the specified object types for the string specified in the **Look for** field.

Label. Search the state and transition labels of the specified object types for the string specified in the **Look for** field.

Name. Search the **Name** fields of the specified object types for the string specified in the **Look for** field.

Description. Search the **Description** fields of the specified object types for the string specified in the **Look for** field.

Document link. Search the **Document** link fields of the specified object types for the string specified in the **Look for** field.

Custom Code. Search custom code for the string specified in the **Look for** field.

Search Method

By default the **Search Method** is **Normal/Wildcard** (regular expression). Alternatively, you can click the **Exact Word match** option if you are searching for a particular sequence of one or more words.

A regular expression is a string composed of letters, numbers, and special symbols that define one or more strings. Some characters have special meaning when used in a regular expression, while other characters are interpreted as themselves. Any other character appearing in a regular expression is ordinary, unless a `\` precedes it.

These are the special characters supported by Stateflow Finder.

Character	Description
^	Start of string
\$	End of string
.	Any character
\	Quote the next character
*	Match zero or more
+	Match one or more
[]	Set of characters

Object Type

Specify the object types to search by toggling the check boxes. A check mark indicates that the object is included in the search criteria. By default, all object types are included in the search criteria. **Object Types** include the following:

- States
- Transitions
- Junctions
- Events
- Data
- Targets

Find Button

Click the **Find** button to initiate the search operation. The data dictionary is queried and the results are listed in the display area.

Matches

The **Matches** field displays the number of objects that match the specified search criteria.

Refine Button

After the results of a search are displayed, enter additional search criteria and click **Refine** to narrow the previously entered search criteria. An ampersand (&) is prefixed to the search criteria in the **Search History** field to indicate a logical AND with any previously specified search criteria.

Search History

The **Search History** text box displays the current search criteria. Click the pull-down list to display search refinements. An ampersand is prefixed to the search criteria to indicate a logical AND with any previously specified search criteria. You can undo a previously specified search refinement by selecting a previous entry in the search history. By changing the **Search History** selection you force the Finder to use the specified criteria as the current, most refined, search output.

Clear Button

Click **Clear** to clear any previously specified search criteria. Results are removed and the search criteria are reset to the default settings.

Close Button

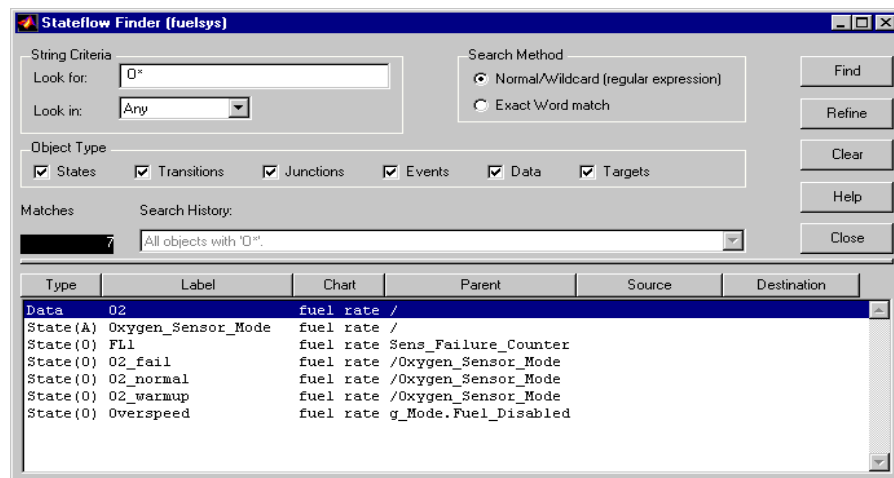
Click **Close** to close the Finder.

Help Button

Click **Help** to display the Stateflow documentation in an HTML browser window.

Finder Display Area

The Finder display area has an appearance similar to the following.



The display area displays found entries with the following columns:

Field	Description
Type	The object type is listed in this field. States with exclusive (OR) decomposition are followed by an (O). States with parallel (AND) decomposition are followed by (A).
Label	The string label of the object is listed in this field.
Chart	The title of the Stateflow diagram (Stateflow block) is listed in this field.
Parent	This object's parent in the hierarchy.
Source	Source object of a transition.
Destination	Destination object of a transition.

All fields are truncated to maintain column widths. The **Parent**, **Source**, and **Destination** fields are truncated from the left so that the name at the end of the hierarchy is readable. The entire field contents, including the truncated portion, are used for resorting.

Each field label is also a button. Click the button to have the list sorted based on that field. If the same button is pressed twice in a row, the sort ordering is reversed.

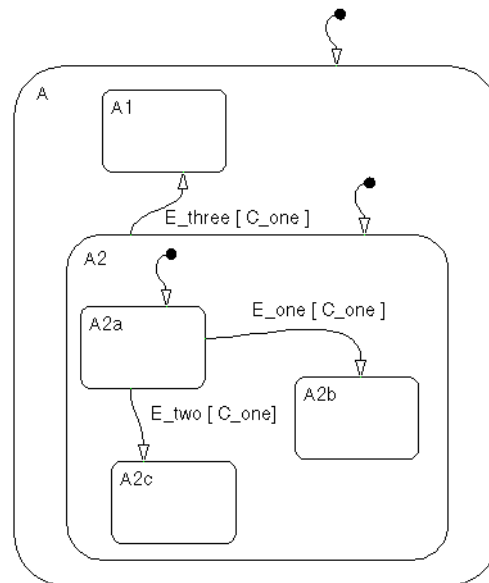
You can resize the Finder vertically to display more output rows, but you cannot expand it horizontally.

Click a graphical entry to highlight that object in the graphical editor window. Double-click an entry to invoke the **Properties** dialog box for that object. Right-click the entry to display a menu that allows you to explore, edit, or display the properties of that entry.

Representing Hierarchy

The Finder displays **Parent**, **Source**, and **Destination** fields to represent the hierarchy. The Stateflow diagram is the root of the hierarchy and is represented by the / character. Each level in the hierarchy is delimited by a period (.) character. The **Source** and **Destination** fields use the combination of the tilde (~) and the period (.) characters to denote that the state listed is relative to the **Parent** hierarchy.

Using the following Stateflow diagram as an example, what are the values for the **Parent**, **Source**, and **Destination** fields for the transition from A2a to A2b?



The A2a to A2b transition is within state A2. State A2's parent is state A and state A's parent is the Stateflow diagram itself. The notation for state A2a's parent is /A.A2. State A2a is the transition source and state A2b is the destination. These states are at the same level in the hierarchy. The relative hierarchy notation for the source of the transition is ~.A2a. The full path is /A.A2.A2a. The relative hierarchy notation for the destination of the transition is ~.A2b. The full path is /A.A2.A2b.

The Stateflow Block

This section contains a single block reference for the Stateflow block. Use it as an introduction to using Stateflow in Simulink models.

Stateflow

Purpose

A version of a finite state machine for controlling a physical plant

Library

Stateflow

Description



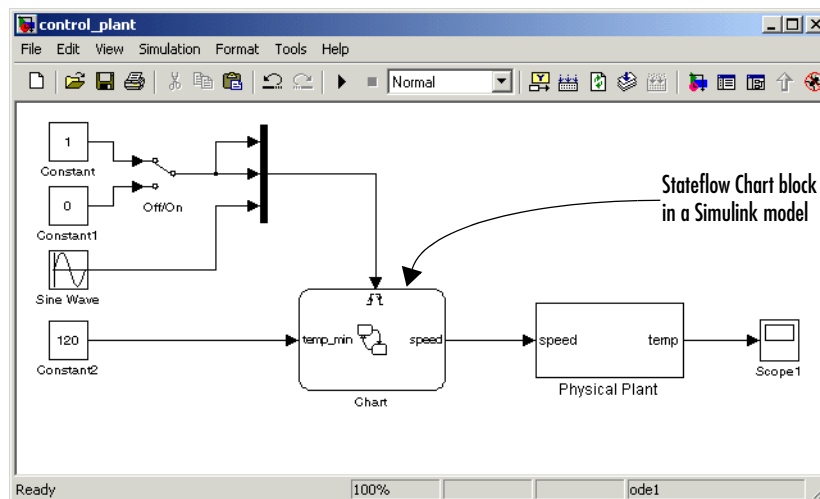
Chart1

A *finite state machine* is a representation of an event-driven (reactive) system. In an event-driven system, the system responds by making a transition from one state (mode) to another prescribed state in response to an event, provided that the condition defining the change is true.

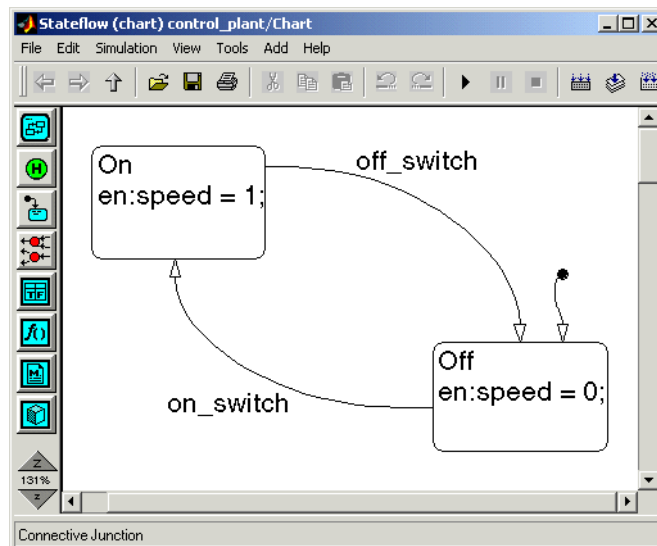
A Stateflow diagram is a graphical representation of a finite state machine, where *states* and *transitions* form the basic building blocks of the system. You can also represent flow (stateless) diagrams using Stateflow. Stateflow provides a block that you include in a Simulink model.

Stateflow charts are usually used to control a physical plant in response to events such as a temperature or pressure sensor, or clock or user-driven events. For example, you can use a state machine to represent a car's automatic transmission. The transmission has a number of operating states: park, reverse, neutral, drive, and low. As the driver shifts from one position to another the system makes a transition from one state to another, for example, from park to reverse.

The following diagram shows a simple Simulink model that has a Stateflow block named Chart (default) that responds to input from a manual switch:



If you double-click the Stateflow block in Simulink, the Stateflow diagram that programs the Stateflow block appears in the Stateflow diagram editor window.



During simulation of the Simulink model, you can interactively debug Stateflow diagrams with animated diagrams. Stateflow diagrams generate efficient C code for simulation, and also for Real-Time Workshop, and custom targets, that is suitable for embedded environments.

For an introduction to using Stateflow in Simulink models, see “Introduction to Stateflow” on page 1-1.

Data Type Support

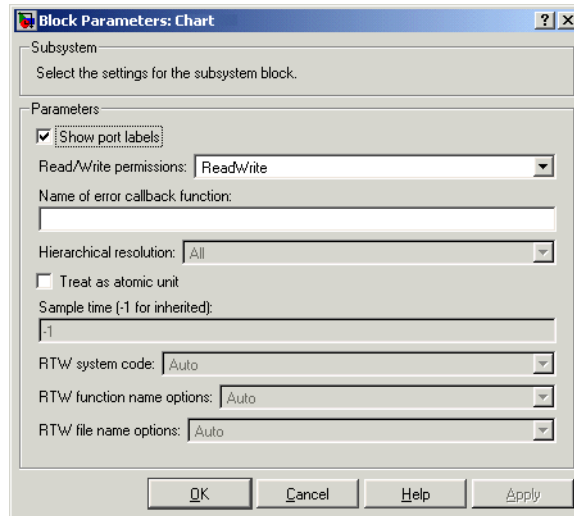
The Stateflow block accepts inputs of any type including two-dimensional matrices and fixed-point data. Floating-point inputs are passed through the block unchanged. Boolean inputs are treated as uint8 signals.

For a discussion on the variable types supported by Embedded MATLAB functions in Simulink, refer to “Data Types Supported by Simulink” in the Using Simulink documentation.

You can declare local data of any type or size.

Stateflow

Parameters and Dialog Box



Note It is highly recommended that the default settings for the block parameters of an Embedded MATLAB Function block not be changed.

Characteristics

Direct Feedthrough	Yes
Sample Time	Specified in the Sample time parameter
Scalar Expansion	N/A
Dimensionalized	Yes
Zero Crossing	No

Semantic Rules Summary

Stateflow semantics describe how the notation in Stateflow charts is interpreted and implemented into a behavior. Knowledge of Stateflow semantics is important to make sound Stateflow diagram design decisions for code generation. Different notations result in different behavior during simulation and generated code execution. This appendix contains a short summary of the rules that Stateflow abides by in executing a Stateflow diagram.

Semantic Rules Summary for Stateflow

Entering a Chart

The set of default flow paths is executed (see “Executing a Set of Flow Graphs” on page B-4). If this does not cause a state entry and the chart has parallel decomposition, then each parallel state is entered (see “Entering a State”).

If executing the default flow paths does not cause state entry, a state inconsistency error occurs.

Executing an Active Chart

If the chart has no states, each execution is equivalent to initializing a chart. Otherwise, the active children are executed. Parallel states are executed in the same order that they are entered.

Entering a State

- 1** If the parent of the state is not active, perform steps 1-4 for the parent.
- 2** If this is a parallel state, check that all siblings with a higher (i.e., earlier) entry order are active. If not, perform all entry steps for these states first.
- 3** Mark the state active.
- 4** Perform any entry actions.
- 5** Enter children, if needed:
 - a** If the state contains a history junction and there was an active child of this state at some point after the most recent chart initialization, perform the entry actions for that child. Otherwise, execute the default flow paths for the state.
 - b** If this state has parallel decomposition, i.e., has children that are parallel states, perform entry steps 1-5 for each state according to its entry order.

- 6** If this is a parallel state, perform all entry actions for the sibling state next in entry order if one exists.
- 7** If the transition path parent is not the same as the parent of the current state, perform entry steps 6 and 7 for the immediate parent of this state.

Executing an Active State

- 1** The set of outer flow graphs is executed (see “Executing a Set of Flow Graphs”). If this causes a state transition, execution stops. (Note that this step is never required for parallel states.)
- 2** During actions and valid on-event actions are performed.
- 3** The set of inner flow graphs is executed. If this does not cause a state transition, the active children are executed, starting at step 1. Parallel states are executed in the same order that they are entered.

Exiting an Active State

- 1** If this is a parallel state, make sure that all sibling states that were entered after this state have already been exited. Otherwise, perform all exiting steps on those sibling states.
- 2** If there are any active children, perform the exit steps on these states in the reverse order they were entered.
- 3** Perform any exit actions.
- 4** Mark the state as inactive.

Executing a Set of Flow Graphs

Flow graphs are executed by starting at step 1 below with a set of starting transitions. The starting transitions for inner flow graphs are all transition segments that originate on the respective state and reside entirely within that state. The starting transitions for outer flow graphs are all transition segments that originate on the respective state but reside at least partially outside that state. The starting transitions for default flow graphs are all default transition segments that have starting points with the same parent:

- 1 A set of transition segments is ordered.
- 2 While there are remaining segments to test, a segment is tested for validity. If the segment is invalid, move to the next segment in order. If the segment is valid, execution depends on the destination:

States

- a No more transition segments are tested and a transition path is formed by backing up and including the transition segment from each preceding junction until the respective starting transition.
- b The states that are the immediate children of the parent of the transition path are exited (see “Exiting an Active State”).
- c The transition action from the final transition segment is executed.
- d The destination state is entered (see “Entering a State”).

Junctions with no outgoing transition segments

Testing stops without any states being exited or entered.

Junctions with outgoing transition segments

Step 1 is repeated with the set of outgoing segments from the junction.

- 3 After testing all outgoing transition segments at a junction, back up the incoming transition segment that brought you to the junction and continue at step 2, starting with the next transition segment after the back up segment. The set of flow graphs is done executing when all starting transitions have been tested.

Executing an Event Broadcast

Output edge trigger event execution is equivalent to changing the value of an output data value. All other events have the following execution:

- 1 If the *receiver* of the event is active, then it is executed (see “Executing an Active Chart” on page B-2 and “Executing an Active State” on page B-3). (The event *receiver* is the parent of the event unless the event was explicitly directed to a *receiver* using the `send()` function.)

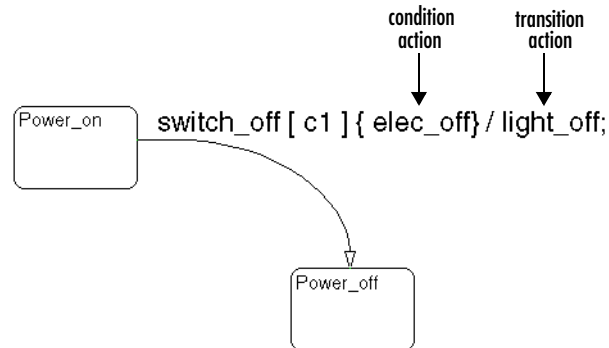
If the receiver of the event is not active, nothing happens.

- 2 After broadcasting the event, the broadcaster performs early return logic based on the type of action statement that caused the event.

Action Type	Early Return Logic
State Entry	If the state is no longer active at the end of the event broadcast, any remaining steps in entering a state are not performed.
State Exit	If the state is no longer active at the end of the event broadcast, any remaining exit actions and steps in state transitioning are not performed.
State During	If the state is no longer active at the end of the event broadcast, any remaining steps in executing an active state are not performed.
Condition	If the origin state of the inner or outer flow graph or parent state of the default flow graph is no longer active at the end of the event broadcast, the remaining steps in the execution of the set of flow graphs are not performed.
Transition	If the parent of the transition path is not active or if that parent has an active child, the remaining transition actions and state entry are not performed.

actions

Actions take place as a part of Stateflow diagram execution. The action can be executed as part of a transition from one state to another, or depending on the activity status of a state. Transitions can have condition actions and transition actions. For example,



Action language defines the categories of actions you can specify and their associated notations. For example, states can have entry, during, exit, and on *event_name* actions as shown by the following:

```
Power_on/  
entry:action1();  
during: action2();  
exit:action3();  
on switch_off:action4();
```

An action can be a function call, a broadcast event, a variable assignment, and so on. For more information on actions and action language, see Chapter 7, “Using Actions in Stateflow.”

API (application programming interface)

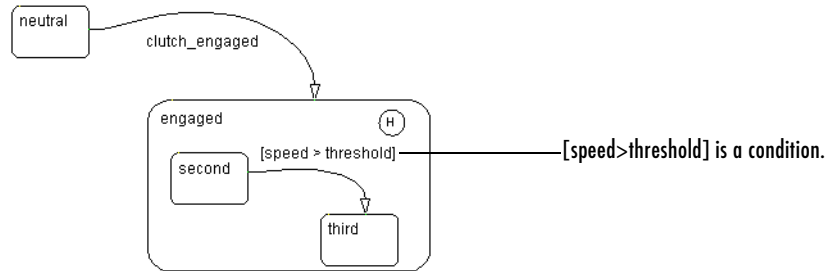
Format provided to access and communicate with an application program from a programming or script environment.

chart instance

Link from a Stateflow model to a chart stored in a Simulink library. A chart in a library can have many chart instances. Updating the chart in the library automatically updates all the instances of that chart.

condition

Boolean expression to specify that a transition occurs if the specified expression is true. For example,

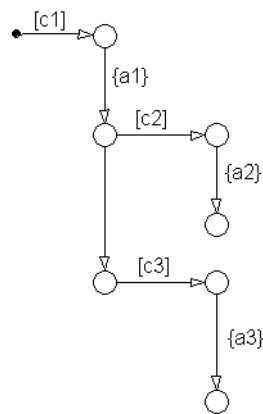


In the preceding example, assume that the state `second` is active. If an event occurs and the value for the data `speed` is greater than the value of the data `threshold`, the transition between states `second` and `third` is taken, and the state `third` becomes active.

connective junction

Decision points in the system. A connective junction is a graphical object that simplifies Stateflow diagram representations and facilitates generation of efficient code. Connective junctions provide alternative ways to represent desired system behavior.


This example shows how connective junctions (displayed as small circles) are used to represent the decision flow of an if code structure.



```

if [c1]{
    a1
    if [c2]{
        a2
    }else if [c3]{
        a3
    }
}

```

Name	Button Icon	Description
Connective junction		One use of a connective junction is to handle situations where transitions out of one state into two or more states are taken based on the same event but guarded by different conditions.

See “Connective Junctions” on page 2-30 for more information.

data

Data objects store numerical values for reference in the Stateflow diagram.

See “Adding Data” on page 6-21 for more information on representing data objects.

data dictionary

Database where Stateflow diagram information is stored. When you create Stateflow diagram objects, the information about those objects is stored in the data dictionary once you save the Stateflow diagram.

Debugger

See “Stateflow Debugger” on page -9.

decomposition


A state has a *decomposition* when it consists of one or more substates. A Stateflow diagram that contains at least one state also has decomposition. Representing hierarchy necessitates some rules around how states can be grouped in the hierarchy. A superstate has either parallel (AND) or exclusive (OR) decomposition. All substates at a particular level in the hierarchy must be of the same decomposition.

Parallel (AND) State Decomposition. *Parallel (AND) state decomposition* is indicated when states have dashed borders. This representation is appropriate if all states at that same level in the hierarchy are active at the same time. The activity within parallel states is essentially independent.

Exclusive (OR) State Decomposition. *Exclusive (OR) state decomposition* is represented by states with solid borders. Exclusive (OR) decomposition is used to describe system modes that are mutually exclusive. Only one state at the same level in the hierarchy can be active at a time.

default transition


Primarily used to specify which exclusive (OR) state is to be entered when there is ambiguity among two or more neighboring exclusive (OR) states. For example, default transitions specify which substate of a superstate with exclusive (OR) decomposition the system enters by default in the absence of any other information. Default transitions are also used to specify that a junction should be entered by default. A default transition is represented by selecting the default transition object from the toolbar and then dropping it to attach to a destination object. The default transition object is a transition with a destination but no source object.

Name	Button Icon	Description
Default transition		Use a default transition to indicate, when entering this level in the hierarchy, which state becomes active by default.

See “Default Transitions” on page 2-25 for more information.

events	<p><i>Events</i> drive the Stateflow diagram execution. All events that affect the Stateflow diagram must be defined. The occurrence of an event causes the status of the states in the Stateflow diagram to be evaluated. The broadcast of an event can trigger a transition to occur and/or can trigger an action to be executed. Events are broadcast in a top-down manner starting from the event's parent in the hierarchy.</p> <p>Events are added, removed, and edited through the Stateflow Explorer. See "Adding Events" on page 6-3 for more information.</p>
Explorer	<p>A tool for displaying, modifying, and creating data and event objects for any parent object in Stateflow. The Explorer also displays, modifies, and creates targets for the Stateflow machine. See "Stateflow Explorer" on page -9.</p>
Finder	<p>A tool to search for objects in Stateflow diagrams on platforms that do not support the Simulink Find tool. See "Stateflow Finder" on page -9.</p>
finite state machine (FSM)	<p>Representation of an event-driven system. FSMs are also used to describe reactive systems. In an event-driven or reactive system, the system transitions from one mode or state to another prescribed mode or state, provided that the condition defining the change is true.</p>
flow graph	<p>Set of decision flow paths that start from a transition segment that, in turn, starts from a state or a default transition segment.</p>
flow path	<p>Ordered sequence of transition segments and junctions where each succeeding segment starts on the junction that terminated the previous segment.</p>
flow subgraph	<p>Set of decision flow paths that start on the same transition segment.</p>
graphical function	<p>Function whose logic is defined by a flow graph. See "Using Functions to Extend Actions" on page 5-29.</p>
hierarchy	<p><i>Hierarchy</i> enables you to organize complex systems by placing states within other higher-level states. A hierarchical design usually reduces the number of transitions and produces neat, more manageable diagrams. See "Stateflow Hierarchy of Objects" on page 1-20 for more information.</p>
history junction	<p>Provides the means to specify the destination substate of a transition based on historical information. If a superstate has a history junction, the transition to the destination substate is defined to be the substate that was most recently</p>

visited. The history junction applies to the level of the hierarchy in which it appears.

Name	Button Icon	Description
History junction		Use a history junction to indicate, when entering this level in the hierarchy, that the last state that was active becomes the next state to be active.

See these sections for more information:

- “History Junctions” on page 2-37
- “Default Transition and a History Junction Example” on page 3-45
- “Labeled Default Transitions Example” on page 3-47
- “Inner Transition to a History Junction Example” on page 3-56

inner transitions Transition that does not exit the source state. Inner transitions are most powerful when defined for superstates with XOR decomposition. Use of inner transitions can greatly simplify a Stateflow diagram.

See “Inner Transitions” on page 2-21 and “Inner Transition to a History Junction Example” on page 3-56 for more information.

library link Link to a chart that is stored in a library model in a Simulink block library.

library model Stateflow model that is stored in a Simulink library. You can include charts from a library in your model by copying them. When you copy a chart from a library into your model, Stateflow does not physically include the chart in your model. Instead, it creates a link to the library chart. You can create multiple links to a single chart. Each link is called a *chart instance*. When you include a chart from a library in your model, you also include its Stateflow machine. Thus, a Stateflow model that includes links to library charts has multiple Stateflow machines. When Stateflow simulates a model that includes charts from a library model, it includes all charts from the library model even if there are links to only some of its models. However, when Stateflow generates a stand-alone or RTW target, it includes only those charts for which there are links. A model that includes links to a library model can be simulated only if all charts in the library model are free of parse and compile errors.

machine	Collection of all Stateflow blocks defined by a Simulink model. This excludes chart instances from library links. If a model includes any library links, it also includes the Stateflow machines defined by the models from which the links originate.
notation	<p>Defines a set of objects and the rules that govern the relationships between those objects. Stateflow notation provides a common language to communicate the design information conveyed by a Stateflow diagram.</p> <p>Stateflow notation consists of</p> <ul style="list-style-type: none">• A set of graphical objects• A set of nongraphical text-based objects• Defined relationships between those objects
parallelism	<p>A system with <i>parallelism</i> can have two or more states that can be active at the same time. The activity of parallel states is essentially independent. Parallelism is represented with a parallel (AND) state decomposition.</p> <p>See “State Decomposition” on page 2-7 for more information.</p>
Real-Time Workshop	Automatic C language code generator for Simulink. It produces C code directly from Simulink block diagram models and automatically builds programs that can be run in real time in a variety of environments. See the Real-Time Workshop documentation for more information.
rtw target	Executable built from code generated by the Real-Time Workshop. See Chapter 12, “Building Targets,” for more information.
S-function	<p>When using Simulink together with Stateflow for simulation, Stateflow generates an <i>S-function</i> (MEX-file) for each Stateflow machine to support model simulation. This generated code is a simulation target and is called the <i>sfun</i> target within Stateflow.</p> <p>For more information, see Using Simulink documentation.</p>
semantics	<p><i>Semantics</i> describe how the notation is interpreted and implemented behind the scenes. A completed Stateflow diagram communicates how the system will behave. A Stateflow diagram contains actions associated with transitions and states. The semantics describe in what sequence these actions take place during Stateflow diagram execution.</p>

Simulink

Software package for modeling, simulating, and analyzing dynamic systems. It supports linear and nonlinear systems, modeled in continuous time, sampled time, or a hybrid of the two. Systems can also be multirate, i.e., have different parts that are sampled or updated at different rates.

It allows you to represent systems as block diagrams that you build using your mouse to connect blocks and your keyboard to edit block parameters. Stateflow is part of this environment. The Stateflow block is a masked Simulink model. Stateflow builds an S-function that corresponds to each Stateflow machine. This S-function is the agent Simulink interacts with for simulation and analysis.


The control behavior that Stateflow models complements the algorithmic behavior modeled in Simulink block diagrams. By incorporating Stateflow diagrams into Simulink models, you can add event-driven behavior to Simulink simulations. You create models that represent both data and decision flow by combining Stateflow blocks with the standard Simulink blocksets. These combined models are simulated using Simulink.

The Using Simulink documentation describes how to work with Simulink. It explains how to manipulate Simulink blocks, access block parameters, and connect blocks to build models. It also provides reference descriptions of each block in the standard Simulink libraries.

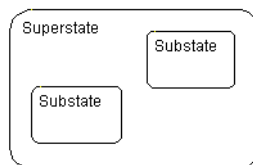
state

A *state* describes a mode of a reactive system. A reactive system has many possible states. States in a Stateflow diagram represent these modes. The activity or inactivity of the states dynamically changes based on transitions among events and conditions.

Every state has hierarchy. In a Stateflow diagram consisting of a single state, that state's parent is the Stateflow diagram itself. A state also has history that applies to its level of hierarchy in the Stateflow diagram. States can have actions that are executed in a sequence based upon action type. The action types are entry, during, exit, or on event_name actions.

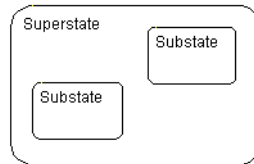
Name	Button Icon	Description
State		Use a state to depict a mode of the system.

- Stateflow block** Masked Simulink model that is equivalent to an empty, untitled Stateflow diagram. Use the Stateflow block to include a Stateflow diagram in a Simulink model.
- The control behavior that Stateflow models complements the algorithmic behavior modeled in Simulink block diagrams. By incorporating Stateflow blocks into Simulink models, you can add complex event-driven behavior to Simulink simulations. You create models that represent both data and decision flow by combining Stateflow blocks with the standard Simulink and toolbox block libraries. These combined models are simulated using Simulink.
- Stateflow Debugger** Use to debug and animate your Stateflow diagrams. Each state in the Stateflow diagram simulation is evaluated for overall code coverage. This coverage analysis is done automatically when the target is compiled and built with the debug options. The Debugger can also be used to perform dynamic checking. The Debugger operates on the Stateflow machine.
- Stateflow diagram** Using Stateflow, you create Stateflow diagrams. A *Stateflow diagram* is also a graphical representation of a finite state machine where *states* and *transitions* form the basic building blocks of the system. See “Stateflow and Simulink” on page 1-5 for more information on Stateflow diagrams.
- Stateflow Explorer** Use to add, remove, and modify data, event, and target objects. See “Using the Model Explorer with Stateflow Objects” on page 14-2 for more information.
- Stateflow Finder** Use to display a list of objects based on search criteria you specify. You can directly access the properties dialog box of any object in the search output display by clicking that object. See “Using the Stateflow Finder Tool” on page 14-27 for more information.
- subchart** Chart contained by another chart. See “Using Subcharts to Extend Charts” on page 5-5.
- substate** A state is a *substate* if it is contained by a superstate.



superstate

A state is a *superstate* if it contains other states, called substates.

**supertransition**

Transition between objects residing in different subcharts. See “Using Supertransitions to Extend Transitions” on page 5-12 for more information.

target

A container object for the generated code from the Stateflow diagrams in a model. Stateflow represents the collection of all Stateflow diagrams for a model as a Stateflow machine. This means that target objects belong to the Stateflow machine.

Stateflow generates code for all target types, which include Simulation, Real-Time Workshop, and Custom targets. See Chapter 12, “Building Targets,” for more information.

topdown processing

The way in which Stateflow processes states and events. In particular, Stateflow processes superstates before states. Stateflow processes a state only if its superstate is activated first.

transition

The circumstances under which the system moves from one state to another. Either end of a transition can be attached to a source and a destination object. The *source* is where the transition begins and the *destination* is where the transition ends. It is often the occurrence of some event that causes a transition to take place.

transition path

Flow path that starts and ends on a state.

transition segment

A state-to-junction, junction-to-junction, or junction-to-state part of a complete state-to-state transition. Transition segments are sometimes loosely referred to as transitions.

virtual scrollbar

Enables you to set a value by scrolling through a list of choices. When you move the mouse over a menu item with a virtual scrollbar, the cursor changes to a line with a double arrowhead. Virtual scrollbars are either vertical or horizontal. The direction is indicated by the positioning of the arrowheads. Drag the mouse either horizontally or vertically to change the value.

A

- abs
 - C library function in Stateflow action language 7-22
 - calling in action language 7-23
- acos in action language 7-22
- action language
 - array arguments 7-42
 - assignment operations 7-15
 - binary operations 7-12
 - bit operations 7-12
 - comment symbols *%,,/** 7-19
 - condition statements 7-8
 - data and event arguments 7-40
 - defined 1-17
 - directed event broadcasting 7-46
 - event broadcasting 7-44
 - floating-point number precision 7-20
 - hexadecimal notation 7-19
 - infinity symbol *inf* 7-20
 - keyword identifiers 7-12
 - line continuation symbol 7-20
 - literal code symbol *\$* 7-20
 - MATLAB display symbol *;* 7-20
 - pointer and address operations 7-16
 - semicolon symbol 7-20
 - single-precision floating point symbol *F* in action language 7-20
 - special symbols 7-19
 - temporal logic 7-50
 - time symbol *t* 7-21
 - types of 7-3
 - unary operations 7-15
- actions
 - assigning to decisions in truth table 10-29
 - binding function call subsystem 7-58
 - defined 1-17
 - during 2-9
 - entry 2-9
 - exit 2-9
 - on *event_name* 2-9
 - states 4-11
 - tracking rows in truth tables 10-45
 - unary 7-15
 - See also* condition actions
 - See also* transition actions
- activation order for parallel (AND) states 4-9
- active chart execution 3-6
- active states 2-5
 - display in debugger 13-8
 - execution 3-22
 - exiting 3-23
- addition (+) of fixed-point data 8-24
- addition operator (+) 7-13
- after temporal logic operator 7-52
- animation controls in debugger 13-7
- Append symbol names with parent names coder option 12-22
- arguments 7-40
- array arguments in action language 7-42
- Array property of data 6-29
- arrays
 - and custom code 7-43
 - indexing 7-42
- arrowhead size of transitions 4-20
- asin in action language 7-22
- assignment operations 7-15
 - fixed-point data 8-20, 8-31
- at temporal logic operator 7-54
- atan in action language 7-22
- atan2 in action language 7-22

B

- Back To button in diagram editor 5-10
- before temporal logic operator 7-53
- bias (B) in fixed-point data 8-2
- binary operations 7-12
 - fixed-point data 8-18
- binary point in fixed-point data 8-5
- binding function call subsystem
 - to state 7-58
- binding function call subsystem event
 - example 7-58
 - muxed events 7-68
 - subsystem sampling times 7-59
- bit operations 7-12
- bitwise & (AND) operator 7-14
- block
 - See also* Stateflow block
- bowing transitions 5-25
- boxes
 - creating 5-45
 - definition 2-39
 - grouping 5-45
- Break button on debugger 13-7
- breakpoints
 - chart entry 13-3
 - display in debugger 13-8
 - event broadcast 13-3
 - functions 5-44
 - overview 13-3
 - setting in debugger 13-3
 - state entry 13-3
 - states 4-11
 - transitions 4-24
- broadcasting directed events
 - examples using send keyword 7-46
 - send function 3-85
 - with qualified event names 3-87

- broadcasting events 7-44
 - in condition actions 3-40
 - in truth tables 10-11
- Browse Data display in debugger 13-8
- building targets 12-5, 12-7
 - options for custom target 12-26
 - starting 12-36

C

- C functions
 - library 7-22
 - user-written 7-24
- C++ code 12-33
- Call Stack display in debugger 13-8
- cast operation
 - and type operator 7-18
- cast operator 7-18
- ceil in action language 7-22
- change indicator (*) in title bar 4-29
- change(data_name) keyword 6-18
- chart notes
 - See* notes (chart)
- chart libraries 9-23
- charts
 - checking for errors 12-39
 - creating 4-2
 - decomposition 2-7
 - editing 4-28, 14-5
 - executing active charts 3-6
 - executing inactive charts 3-6
 - how they execute 3-6
 - printing 5-51
 - properties 9-4
 - saving model 4-2
 - setting properties for in Explorer 14-8
 - update method 4-2

- update methods for defining interface 9-11
- See also* subcharts
- code generation
 - error messages 12-47
- Code Generation Directory option 12-31
- code generation files 12-49
 - .dll files 12-49
 - code files 12-50
 - make files 12-51
- colors in diagram editor 4-31
- command line debugger 13-25
- command line debugger commands 13-28
- commands for command line debugger 13-28
- comment symbols `%,,/*` in action language 7-19
- comments (chart)
 - See notes* (chart)
- Comments in generated code coder option 12-18
- Compact nested if-else using logical AND/OR operators coder option 12-19
- comparison operators
 - (`>`, `<`, `>=`, `<=`, `==`, `!=`, `<>`) 7-13
- compilation error messages 12-48
- CompiledSize property 6-50
- CompiledType property
 - typing data
 - using CompiledType property 6-46
- condition actions
 - and transition actions 3-39
 - event broadcasts in 3-81
 - examples 3-37
 - in for loops 3-40
 - simple, example of 3-37
 - to broadcast events 3-40
 - with cyclic behavior to avoid 3-41
- condition coverage 13-49
 - definition 13-49
 - Embedded MATLAB functions 11-30, 11-43
 - example 13-55
 - truth tables 10-61
- conditional notation for temporal logic operators 7-57
- conditions
 - for transitions, defined 1-15
 - for transitions, guidelines 7-8
 - in function 7-8
 - outcomes for in truth tables 10-2
- configuring
 - custom target 12-24
 - rtw target 12-14
 - simulation target 12-11
- conflicting names in custom code 12-31
- conflicting transitions
 - definition 13-16
 - detecting 13-16
 - example 13-16
- connective junctions 2-30
 - backtracking transition segments to source 3-69
 - common events example 2-36
 - common source example 2-35
 - creating 4-25, 5-2
 - definition 2-30
 - described 1-18
 - examples of 3-58
 - flow diagrams 3-63
 - for loop 2-33
 - if-then-else decision 3-59
 - in flow diagrams 2-31
 - in for loops 3-62
 - in self-loop transitions 2-33
 - self-loop transitions 3-61
 - transitions based on common event 3-68
 - transitions from a common source 3-65
 - transitions from multiple sources 3-67

- with default transitions 3-44
- Contains word option in Search & Replace tool
 - 14-15
- context (shortcut) menu to properties 4-30
- context-sensitive constants in fixed-point data
 - 8-8
- Continue button on debugger 13-7
- continuous update method 9-11
- continuous update method for Stateflow block
 - 9-11
- copying objects in the diagram editor 4-35
- corners of states 4-17
- cos in action language 7-22
- cosh in action language 7-22
- Creation Date property of machines 9-9
- Creator property of machines 9-9
- custom code
 - building into target 12-5, 12-29
 - building into targets 12-5
 - calling graphical functions 12-35
 - conflicting names 12-31
 - including C++ code 12-33
 - integrating with diagram 12-29
 - path names 12-32
- Custom code included at the top of generated code
 - option 12-30
- Custom include directory paths option 12-31
- Custom initialization code option 12-31
- Custom source files option 12-31
- custom target
 - configuring 12-24
- custom targets
 - code generation options 12-26
- Custom termination code option 12-31
- cutting objects in diagram editor 4-35
- cyclic behavior
 - debugging 13-20

- definition 13-20
- example 13-21
- example of nondetection 13-22
- in condition actions 3-41
- noncyclic behavior flagged as cyclic example
 - 13-23

cyclomatic complexity

- in model coverage reports 13-44

D

- dashed transitions 4-20
- data 6-35
 - adding (creating) 6-21
 - copying/moving in Explorer 14-9
 - defined 1-14
 - deleting 14-11
 - dictionary 2-4
 - displaying logged data values 13-35
 - exported 9-29
 - exporting to external code 6-42
 - fixed-point 8-1, 8-18
 - imported 9-31
 - importing from external code 6-43
 - inheriting size 6-50
 - input from other blocks 6-35
 - logging values to MATLAB workspace 13-33
 - monitor values with command line debugger
 - 13-25
 - monitoring with floating scope 13-38
 - operations in action language 7-12
 - properties of 6-25
 - range violations 13-18
 - renaming 14-8
 - setting properties for in Explorer 14-8
 - sizing 6-50
 - sizing by expression 6-51

- temporary data 6-24
- types 6-45
- typing 6-45
- viewing 4-36
- See also* fixed-point data
- data and events 9-3
- data dictionary
 - adding data 6-21
 - adding events 6-3
 - defined 1-6
- data input from Simulink port order 14-10
- data output to Simulink port order 14-10
- data range violations (debugging) 13-18
- data types
 - boolean 6-46
 - double 6-45
 - fixpt 6-46
 - inheritance 6-46
 - inherited 6-46
 - int16 6-45
 - int32 6-45
 - int8 6-45
 - ml 6-46
 - single 6-45
 - uint16 6-46
 - uint32 6-45
 - uint8 6-46
- data typing
 - with other data 6-48
- data values during simulation 13-24, 13-30
- Debugger
 - monitoring data values during simulation 13-24
- debugger
 - action control buttons 13-7
 - active states display 13-8
 - animation controls 13-7
 - Break button 13-7
 - breakpoints 13-3
 - breakpoints display 13-8
 - browse data display 13-8
 - call stack display 13-8
 - clear output display 13-8
 - Continue button 13-7
 - debugging run-time errors 13-9
 - display controls 13-8
 - main window 13-2
 - setting global breakpoints 13-3
 - Start button 13-5
 - status display area 13-5
 - Step button 13-7
 - Stop Simulation button 13-7
 - user interface 13-2
- Debugger breakpoint property
 - charts 9-7
- debugging
 - breakpoints in Embedded MATLAB function 11-20
 - conflicting transitions 13-16
 - cyclic behavior 13-20
 - data range violations 13-18
 - display variable values in Embedded MATLAB function 11-26
 - displaying Embedded MATLAB function variables in MATLAB 11-27
 - Embedded MATLAB function 11-18
 - Embedded MATLAB functions 11-20
 - error checking options 13-5
 - model coverage 13-42
 - state inconsistency 13-14
 - stepping through Embedded MATLAB function 11-22
 - truth table during simulation 10-36
- decision coverage 13-44

- chart as a triggered block 13-45
- chart containing substates 13-46
- conditional transitions 13-49
- Embedded MATLAB functions 11-30, 11-43
- example 13-55
- in model coverage reports 13-44
- state with `on event_name` statement 13-48
- superstates containing substates 13-46
- truth tables 10-61
- decision outcomes for truth tables 10-2
 - tracking action rows feature 10-45
- decisions
 - assigning actions in truth table 10-29
- decomposition
 - described 1-10
 - states and charts 2-7
 - substates 4-8
- default decision outcome for truth tables
 - concept 10-2
- default transitions
 - and exclusive (OR) decomposition 3-43
 - and history junctions 3-45
 - creating 4-22
 - defined 1-13
 - examples 2-26, 3-43
 - labeled 3-47
 - labeling 2-26
 - to a junction 3-44
- Description property
 - data 6-33
 - events 6-9
 - functions 5-44
 - junctions 4-27, 5-4
 - states 4-11
 - transitions 4-24
- Description property for charts 9-8
- Description property of machines 9-10
- Destination property of transitions 4-23
- diagram (Stateflow)
 - graphical components 1-9
 - objects 1-9
- diagram editor
 - copying objects 4-35
 - cutting and pasting objects 4-35
 - drawing area 4-30
 - elements 4-29
 - menu bar 4-29
 - selecting and deselecting objects 4-34
 - specifying colors and fonts 4-31
 - status bar 4-30
 - title bar 4-29
 - toolbar 4-29
 - undoing and redoing operations 4-39
 - zooming 4-37
- directed event broadcasting
 - examples 3-85
 - send function
 - examples 7-46
 - semantics 3-85
 - using qualified event names 3-87
 - with qualified names 7-46
- discrete update method 9-11
- display controls in debugger 13-8
- division (/) of fixed-point data 8-27
- division operator (/) 7-12
- .dll files 12-49
- Document Link property
 - charts 9-8
 - data 6-34
 - events 6-10
 - junctions 4-27, 5-4
 - states 4-11
 - transitions 4-24
- Document Link property for functions 5-44

Document Link property of machines 9-10

drawing area

in diagram editor 4-30

during action 2-9

example 2-11

E

E (binary point) in fixed-point data 8-5

early return logic for event broadcasts 3-26

Echo expressions without semicolons coder option
12-13

Edit property of Search & Replace tool 14-21

editing

charts 4-28

labels in diagram editor 4-36

truth tables 10-19

Editor property for charts 9-7

either edge trigger 6-12

Embedded MATLAB Editor

introduction 11-14

Embedded MATLAB functions

argument and return values 11-8

breakpoints in function 11-20

calling from Stateflow 11-7

calling MATLAB functions 11-16

calling other functions 11-3

checking for errors 11-18

condition coverage 11-30, 11-43

creating 11-6

debugging 11-20

debugging function for 11-18

decision coverage 11-30, 11-43

description 11-1

display variable value 11-26

displaying variable values in MATLAB 11-27

example 11-5

example model 11-5

function library 11-14

implicitly declared variables 11-14

introduction to 11-2

MCDC coverage 11-30, 11-43

model coverage 11-29

model coverage example 11-30

model coverage report 11-35

Model Explorer 11-8

persistent variables 11-14

programming 11-13

signature 11-8

simulation example 11-20

stepping through function 11-22

subfunctions 11-16

types of model coverage 11-30

Enable C-bit operations property

for charts 9-6

operations affected 7-15

Enable C-like bit operations property of machines
9-10

Enable debugging/animation coder option 12-12

Enable overflow detection (with debugging) coder
option 12-12

entry action 2-9

example 2-11, 7-4

error

redefinition or redeclaration 12-31

error checking

charts 12-39

in Embedded MATLAB functions 11-18

overspecified truth tables 10-55

underspecified truth tables 10-56

when it occurs for truth tables 10-34

error messages

code generation 12-47

compilation 12-48

- overview 12-46
- parsing 12-46
- target building 12-48
- errors
 - data range 13-5
 - debugging run-time errors 13-9
 - detect cycles 13-5
 - state inconsistency 13-5
 - transition conflict 13-5
- event actions
 - in a superstate 3-71
- event broadcasting
 - early return logic 3-26
 - examples
 - state action notation 7-44
 - transition action notation 7-45
 - in condition actions 3-81
 - in parallel state action 3-73
 - nested in transition actions 3-77
 - See also* directed event broadcasting
- event input from Simulink
 - port order 14-10
 - trigger 6-12
- event notation for temporal logic operators 7-57
- event output to Simulink port order 14-10
- event triggers
 - defining 9-20
 - function call example 9-18
 - function call output event 9-17
 - function call semantics 9-19
- events
 - adding (creating) 6-3
 - and transitions from substate to substate 3-35
 - broadcast in condition actions 3-40
 - broadcasting 7-44
 - causing transitions 3-32
 - copying/moving in Explorer 14-9
 - defined 1-14
 - defining edge-triggered output events 9-20
 - deleting 14-11
 - executing 3-3
 - exported 9-25
 - exporting events example 9-25
 - exporting to external code 6-16
 - function call output event to Simulink 9-17
 - how Stateflow processes them 3-4
 - imported 9-27
 - imported event example 9-27
 - importing from external code 6-17
 - processing with inner transition to junction 3-53
 - processing with inner transitions in exclusive (OR) states 3-49
 - properties 6-7
 - renaming 14-8
 - setting properties for in Explorer 14-8
 - sources for 3-3
 - triggering Simulink blocks with 6-13
 - viewing 4-36
 - See also* directed event broadcasting
 - See also* implicit events
 - See also* input events
 - See also* output events
- every temporal logic operator 7-55
- exclusive (OR) decomposition 2-7
 - and default transitions 3-43
- exclusive (OR) states
 - defined 1-10
 - transitions 2-17
 - transitions to and from 3-31
- exclusive (OR) substates
 - transitions 2-20
- exclusive (OR) superstates

- transitions 2-18
- Execute (enter) Chart at Initialization property for charts 9-7
- exit action 2-9
 - example 2-11, 7-5
- exp in action language 7-22
- Explore property of Search & Replace tool 14-21
- Explorer
 - object hierarchy list 14-4
 - opening 14-3
 - operations 14-2
 - overview 14-2
 - targets 14-6
 - user interface 14-2
- Export Chart Level Graphical Functions property for charts 9-6
- exporting data to external code 9-29
 - description 6-42
 - example 9-30
- exporting events to external code 9-25
 - example 9-26
- exporting graphical functions 5-41
- external code sources
 - defining interface for 9-25
 - definition 9-25
- F**
- F (fractional slope) in fixed-point data 8-5
- fabs in action language 7-22
- falling edge trigger 6-12
- Field types field of Search & Replace tool 14-14
- final action in truth tables 10-31
- Finder
 - dialog box 14-28
 - user interface 14-27
- finite state machine
 - described 1-2
 - introduction 1-2
 - references 1-4
 - representations 1-2
- First Index (of array) property of data 6-30
- fixed-point data 8-1, 8-18
 - arithmetic 8-2
 - bias B 8-2
 - context-sensitive constants 8-8
 - defined 8-2
 - example of using 8-14
 - how to use 8-9
 - implementation in Stateflow 8-5
 - offline conversions 8-36
 - online conversions 8-36
 - operation (+, -, *, /) equations 8-3
 - operations supported 8-18
 - overflow detection 8-11
 - properties 6-28
 - quantized integer, Q 8-2
 - Scaling property 6-29, 8-7
 - setting for Strong Data Typing with Simulink IO 9-7
 - sharing with Simulink 8-12
 - slope S 8-2
 - specifying in Stateflow 8-7
 - Stored Integer property 6-28, 8-7
 - Type property 8-7
- fixed-point operations 8-18
 - assignment 8-31
 - casting 8-31
 - logical (&, &&, |, ||) 8-30
 - promotions 8-21
 - special assignment
 - and context-sensitive constants 8-35
 - division example 8-34
 - multiplication example 8-32

- floating scope
 - select signals 13-39
 - floating scope monitor of data and states 13-38
 - floating-point numbers
 - precision in action language 7-20
 - floor in action language 7-22
 - flow diagrams
 - connective junctions in 2-31
 - cyclic behavior example 13-22
 - example 2-34
 - examples 2-31
 - for loops 2-33
 - with connective junctions 3-63
 - flow graphs
 - order of execution 3-8
 - types 3-7
 - fmod in action language 7-22
 - font size of labels 4-36
 - fonts in diagram editor 4-31
 - for loops
 - example 2-33
 - with condition actions 3-40
 - with connective junctions 3-62
 - Forward To button in diagram editor 5-10
 - function call events
 - example output event semantics 9-19
 - output event 9-17
 - output event example 9-18
 - function call subsystem
 - binding trigger event 7-58
 - mixing bound and muxed events 7-68
 - sampling times with bind action 7-59
 - functions
 - calling functions from Embedded MATLAB
 - functions 11-3
 - data and event arguments 7-40
 - Description property 5-44
 - Document Link property 5-44
 - Embedded MATLAB function example 11-5
 - Embedded MATLAB run-time library 11-14
 - Function Inline Option property 5-44
 - inlining 5-44
 - Label property 5-44
 - MATLAB 7-27
 - Name property 5-43
 - Parent property 5-43
 - setting breakpoints 5-44
 - truth table function 10-4
 - See also* graphical functions
- G**
- generated code files 12-49
 - graphical functions 2-40
 - calling from action language 5-41
 - calling from custom code 12-35
 - compared with truth tables 10-11
 - creating 5-29
 - example 2-40
 - exporting 5-41
 - properties 5-43
 - realizing truth tables 10-62
 - signature (label) 5-29
 - graphical objects 2-2
 - copying 4-35
 - cutting and pasting 4-35
 - grouping
 - boxes 5-45
 - states 4-8
- H**
- hexadecimal notation in action language 7-19
 - hierarchy

- described 1-20
- of objects 2-6
- of states 2-6
- state example 2-6
- transition example 2-13
- history junctions 2-37
 - and default transitions 3-45
 - and inner transitions 2-38
 - creating 4-25, 5-2
 - defined 1-16
 - definition 2-37
 - example of use 2-37
 - inner transitions to 2-24, 3-56
- I**
- if-then-else decision
 - examples 2-31, 2-32
 - with connective junctions 3-59
- implicit events
 - definition 6-18
 - example 6-18
 - referencing in action language 6-18
- importing data from external code 6-43, 9-31
 - example 9-31
- importing events from external code 9-27
 - example 9-28
- in function in conditions 7-8
- inactive chart execution 3-6
- inactive states 2-5
- Index property for events 6-9
- infinity symbol inf in action language 7-20
- inherited update method 9-11
- inherited update method for Stateflow block 9-11
- inheriting data size 6-50
 - CompiledSize property 6-50
- inheriting data type 6-46
- initial action in truth tables 10-31
- Initialize from property of data 6-31
- inlining functions
 - Function Inline Option property 5-44
- inner transitions
 - after using them 2-23
 - before using them 2-22
 - definition 2-21
 - examples 2-21, 3-49
 - processing events in exclusive (OR) states 3-49
 - to a history junction 3-56
 - to a junction, processing events with 3-53
 - to history junction 2-24
- input data from other blocks 6-35
- input events
 - associating with control signals 6-13
 - defining
- integer word size
 - setting for target 8-22
- interfaces 9-3
 - to external code 9-2, 9-25
 - to MATLAB data 9-2
 - typical tasks to define 9-3
 - update methods for Stateflow block 9-11
- interfaces to MATLAB
 - data 9-24
 - workspace 9-24
- interfaces to Simulink
 - continuous Stateflow block 9-15
 - defining 1-7
 - edge-triggered output event 9-20
 - function call output event 9-17
 - implementing 9-12
 - inherited Stateflow block 9-14
 - sampled Stateflow block 9-13
 - triggered Stateflow block 9-12

J

junctions

- properties 4-26, 5-3

- size 4-26, 5-3

- See also* connective junctions

- See also* history junctions

K

keyboard shortcuts

- in diagram editor 4-40

- moving in a zoomed diagram 4-38

- opening subcharts 5-9

- zooming 4-38

keywords

- `change(data_name)` 6-18

- during 7-5

- entry 7-4

- `entry(state_name)` 6-18

- `exit` 7-5

- `exit(state_name)` 6-18

- `in(state_name)` 7-8

- `m1()` 7-28

- `m1.` 7-27

- on *event_name* action 7-6

- `send` 7-46

- summary list 7-12

- `tick` 6-19

- `wakeup` 6-19

- default transitions 2-26, 3-47

- editing in diagram editor 4-36

- field 14-18

- font size 4-36

- format for transition segments 3-58

- format for transitions 3-31, 4-18

- graphical function signature 5-29

- state example 2-10

- states 2-9, 4-11

- transition 2-14

- transitions 4-17

- labs in action language 7-22

- `ldexp` in action language 7-22

- left bit shift (<<) operator 7-13

- Limit Range property of data 6-33

- line continuation symbol ... in action language 7-20

- literal code symbol \$ in action language 7-20

- `log` in action language 7-22

- `log10` in action language 7-22

- logging data

 - displaying values 13-35

- logging data values to MATLAB workspace 13-33

- logging state activity

 - displaying state activity 13-35

- logging state activity to MATLAB workspace 13-33

- logical AND operator (&) 7-14

M

M code

- Embedded MATLAB function example 11-13

machine

- adding targets to 12-8

- overview of Stateflow machine 1-20

- setting properties 9-8

L

Label property

- functions 5-44

- states 4-11

- transitions 4-24

labels

- make files 12-51
- Match case
 - field of Search & Replace tool 14-14
 - search option of Search & Replace tool 14-15
- Match options field of Search & Replace tool 14-14
- Match whole word option in Search & Replace tool 14-16
- MATLAB
 - functions and data in Stateflow 7-27
 - in Embedded MATLAB functions 11-16
 - `m1()` and full MATLAB notation 7-31
 - `m1()` function call 7-28
 - `m1.` namespace operator 7-27
 - See also* interfaces to MATLAB
 - See also* interfaces to MATLAB workspace
- MATLAB display symbol ; in action language 7-20
- max in action language 7-23
- Max property of data 6-33
- MCDC coverage
 - definition 13-49
 - Embedded MATLAB functions 11-30, 11-43
 - example 13-55
 - explanation 13-57
 - irrelevant conditions 13-58
 - specifying 13-43
 - truth tables 10-61
- menu bar
 - in diagram editor 4-29
- messages
 - error messages 12-46
 - of Search & Replace tool 14-24
- min in action language 7-23
- Min property of data 6-33
- Minimize array reads using temporary variables
 - coder option 12-21
- `m1` data type 7-32
 - and targets 7-32
 - inferring size 7-32
 - place holder for workspace data 7-34
 - scope 7-32
- `m1()` function 7-28
 - and full MATLAB notation 7-31
 - dynamically construct workspace variables 7-31
 - expressions 7-30
 - inferring return size 7-35
 - or `m1.` namespace operator, which to use? 7-31
- `m1.` namespace operator 7-27
 - expressions 7-30
 - inferring return size 7-35
 - or `m1()` function, which to use? 7-31
- model coverage 13-42
 - chart as subsystem report section 13-52
 - colored stateflow diagram example 13-59
 - colored stateflow diagrams 13-59
 - condition coverage 13-49
 - coverages for truth table function 10-58
 - cyclomatic complexity 13-44
 - decision coverage 13-44
 - definition 13-42
 - for Embedded MATLAB functions 11-29
 - for Stateflow charts 13-50
 - for truth tables 10-58
 - generate HTML report 13-43
 - MCDC coverage 13-49
 - report 13-43
 - report for truth table example 10-58
 - reporting on 13-43
 - specifying reports 13-43
 - truth tables 10-58
- model coverage report
 - chart as superstate section 13-52

- state sections 13-53
- Summary 13-51
- transition section 13-55

Model Explorer

- adding data 14-6
- adding events 14-6
- Embedded MATLAB functions 11-8
- object icons 14-5

Modified property of machines 9-10

modulus operator (%) 7-12

monitoring data values

- in the Debugger 13-24

monitoring data values during simulation 13-24, 13-30

monitoring data values with command line debugger 13-25

monitoring data values with floating scope 13-38

monitoring state activity with floating scope 13-38

multiplication (*) of fixed-point data 8-27

multiplication operator (*) 7-12

N

Name property

- charts 9-5
- data 6-26
- events 6-7
- functions 5-43
- states 2-10, 4-10

nongraphical objects (data, events, targets) 2-3

nonsmart transitions

- asymmetric distortion 5-28
- graphical behavior

notation

- defined 1-3

- introduction to Stateflow notation 2-1
- representing hierarchy 2-6

notes (chart)

- changing color 5-49
- changing font 5-49
- creating 5-48
- deleting 5-50
- editing existing notes 5-49
- moving 5-50
- TeX format 5-49

O

object palette

- in Stateflow diagram editor 4-30

Object types field of Search & Replace tool 14-14

objects

- hierarchy 2-6
- overview of Stateflow objects 2-2
- See also* graphical objects
- See also* nongraphical objects

offline conversions with fixed-point data 8-36

on *event_name* action 2-9

- example 2-11, 7-6

online conversions with and fixed-point data 8-36

operations

- assignment 7-15
- binary 7-12
- bit 7-12
- defined for fixed-point data 8-3
- enable C-bit operations 9-6
- exceptions to undo 4-39
- fixed-point data 8-18
- in action language 7-12
- pointer and address 7-16
- type cast 7-17
- unary 7-15

- undo and redo 4-39
 - with objects in Explorer 14-2
 - operators
 - addition (+) 7-13
 - bitwise AND (&) 7-14
 - bitwise OR (|) 7-14
 - bitwise XOR (^) 7-14
 - comparison (>, <, >=, <=, ==, !=, <>) 7-13
 - division (/) 7-12
 - explicit type cast cast operator 7-18
 - explicit typing with cast 7-18
 - left bit shift (<<) 7-13
 - logical AND (&&) 7-14
 - logical AND (&) 7-14
 - logical OR (|) 7-14
 - logical OR (||) 7-14
 - MATLAB type cast 7-18
 - modulus (%) 7-12
 - multiplication (*) 7-12
 - pointer and address 7-16
 - power (^) 7-14
 - right bit shift (>>) 7-13
 - subtraction (-) 7-13
 - type 7-18
 - output events
 - associating with output port 6-15
 - defining 6-13
 - Output State Activity property of states 4-10
 - overflow detection
 - fixed-point data 8-11
 - overspecified truth tables 10-55
- P**
- parallel (AND) states
 - activation order 4-9
 - decomposition 2-8
 - defined 1-10
 - entry execution 3-20
 - event broadcast action 3-73
 - examples of 3-73
 - order of execution 3-20
 - Parent property
 - charts 9-5
 - data 6-26
 - events 6-8
 - functions 5-43
 - junctions 4-27, 5-4
 - states 4-10
 - transitions 4-23
 - parsing diagrams
 - error messages 12-46
 - example 12-40
 - overview 12-38
 - starting the parser 12-38
 - tasks 12-40
 - passing arguments by reference
 - C functions
 - passing arguments by reference 7-26
 - pasting objects in the diagram editor 4-35
 - path names for custom code 12-32
 - pointer and address operations 7-16
 - Port property
 - data 6-29
 - events 6-9
 - ports
 - order of inputs and outputs 14-10
 - pow in action language 7-22
 - Preserve case
 - field of Search & Replace tool 14-14
 - search type in Search & Replace tool 14-17
 - Preserve symbol names coder option 12-22
 - printing
 - book report of elements 5-56

- charts 5-51
- current diagram 5-56
- details of chart 5-53
- diagram 5-51
- programming
 - Embedded MATLAB functions 11-13
- promotion rules for fixed-point operations 8-21
- properties
 - machine 9-8
 - of transitions 4-23
 - of truth tables 10-35
 - Search & Replace tool 14-21
 - states 4-9
- Properties property of Search & Replace tool 14-21

Q

- quantized integer (Q) in fixed-point data 8-2

R

- rand in action language 7-22
- range violations, data 13-18
- Recognize if-elseif-else in nested if-else statements coder option 12-20
- redo operation 4-39
- references 1-4
- regular expressions
 - Search & Replace tool 14-16
 - Stateflow Finder 14-29
 - tokens in Search & Replace tool 14-16
- relational operations
 - fixed-point data 8-29
- renaming targets 14-8
- Replace button of Search & Replace tool 14-14
- replace buttons in Search & Replace tool 14-23

- Replace constant expressions by a single constant coder option 12-20
- Replace with field of Search & Replace tool 14-14
- replacing text in Search & Replace tool 14-22
 - with case preservation 14-22
 - with tokens 14-23
- reports
 - book report of elements 5-56
 - charts 5-51
 - details of chart 5-53
 - model coverage 13-43
 - model coverage for Embedded MATLAB functions 11-35
 - model coverage for Stateflow charts 13-50
- reserved names in custom code 12-31
- resolving symbols in action language 12-44
- return size of m1 expressions 7-35
- right bit shift (>>) operator 7-13
- rising edge trigger 6-12
- rtw target
 - configuring 12-14
- rtw target
 - starting the build 12-37
- run-time errors
 - debugging 13-9

S

- Sample Time property for charts 9-6
- sampled update method for Stateflow block 9-11
- Save final value to base workspace property of data 6-32
- Scaling property of fixed-point data 6-29, 8-7
- Scope property
 - data 6-27
 - events 6-8
- Search & Replace tool 14-12

- containing object 14-20
- Contains word option 14-15
- Custom Code field 14-19
- Description field 14-19
- Document Links field 14-19
- Field types field 14-14
- icon of found object 14-20
- Match case field 14-14
- Match case option 14-15
- Match options field 14-14
- Match whole word option 14-16
- messages 14-24
- Name field 14-18
- object types 14-14
- Object types field 14-14
- opening 14-12
- portal area 14-20
- Preserve case field 14-14
- Preserve case option 14-17
- Regular expression option in Search & Replace tool 14-16
- regular expression tokens 14-16
- Replace All button 14-24
- Replace All in This Object button 14-24
- Replace button 14-14, 14-24
- Replace with field 14-14
- replacement text 14-22
- Search button 14-14, 14-19
- Search For field 14-13
- Search in field 14-14
- search order 14-21
- search scope 14-17
- search types 14-15
- view area 14-19
- View Area field 14-15
- viewer 14-20
- viewing a match 14-20
- Search button of Search & Replace tool 14-14
- Search for field of Search & Replace tool 14-13
- Search in field of Search & Replace tool 14-14
- search order in Search & Replace tool 14-21
- search scope in Search & Replace tool 14-17
- searching
 - chart 14-17
 - Finder user interface 14-27
 - machine 14-17
 - specific objects 14-18
 - text 14-12
 - text matches 14-18
- selecting and deselecting objects in the diagram
 - editor 4-34
- self-loop transitions 2-21
 - creating 4-21
 - delay 2-34
 - with connective junctions 3-61
 - with junctions 2-33
- semantics
 - defined 1-4
 - early return logic for event broadcasts 3-26
 - examples 3-29
 - executing a chart 3-6
 - executing a state 3-20
 - executing a transition 3-7
 - executing an event 3-3
- send function
 - and directed event broadcasting 7-46
 - directed event broadcasting 3-85
 - directed event broadcasting examples 7-46
- sfnew function 4-2
- shortcut keys
 - in diagram editor 4-40
 - moving in a zoomed diagram 4-38
 - opening subcharts 5-9
 - zooming 4-38

- shortcut menus
 - in Stateflow diagram editor 4-30
 - to properties 4-30
- Show Portal property of Search & Replace tool 14-21
- Signal Logging dialog 13-34
- signal selection in floating scope 13-39
- signature
 - graphical functions 5-29
- simulating truth tables 10-36
- simulation
 - Embedded MATLAB function 11-20
 - monitoring data values 13-24, 13-30
 - monitoring data values in the Debugger 13-24
- simulation target
 - code generation options 12-12
 - configuring 12-11
 - starting the build 12-36
- Simulink
 - See also* interfaces to Simulink
- Simulink model and Stateflow machine
 - relationship between 1-5
- Simulink Model property of machines 9-9
- Simulink Subsystem property for charts 9-5
- `./simulink/ref/#simulink.signal` 6-38
- `sin` in action language 7-22
- single-precision floating-point symbol `F` 7-20
- `sinh` in action language 7-22
- Sizes (of array) property of data 6-29
- sizing data 6-50
 - by expression 6-51
 - by inheritance 6-50
 - CompiledSize property 6-50
- slits (in supertransitions) 5-12
- slope (S) in fixed-point data 8-2
- smart transitions
 - bowing symmetrically 5-25
 - graphical behavior 5-20
- Source property of transitions
 - transitions
 - Source property 4-23
- `sqrt` in action language 7-22
- Start button on debugger 13-5
- starting the build 12-36
- state inconsistency
 - debugging 13-14
 - definition 13-14
 - detecting 13-14
 - example 13-15
- Stateflow
 - representations 1-3
 - Signal Logging dialog 13-34
- Stateflow blocks
 - considerations in choosing continuous update 9-15
 - continuous 9-15
 - continuous example 9-16
 - inherited 9-14
 - inherited example 9-14
 - sampled 9-13
 - sampled example 9-13
 - triggered 9-12
 - triggered example 9-12
 - update methods 9-11
- Stateflow diagram editor
 - object palette 4-30
 - shortcut menus 4-30
 - zoom control 4-30
- `stateflow` function 4-2
- states
 - actions 4-11
 - active and inactive 2-5
 - active state execution 3-22

- button (drawing) 2-5
- corners 4-17
- creating 4-5, 5-45
- debugger breakpoint property 4-11
- decomposition 2-5, 2-7
- definition 2-5
- displaying logged state activity 13-35
- during action 2-11
- editing 14-5
- entry action 2-11, 7-4
- entry execution 3-20
- exclusive (OR) decomposition 2-7
- execution example 3-23
- exit action 2-11, 7-5
- exiting active states 3-23
- grouping 4-8
- hierarchy 2-6
- how they are executed 3-20
- label 2-9, 4-11
- label example 2-10
- label notation 2-5
- label property 4-11
- logging activity to MATLAB workspace 13-33
- monitoring activity with floating scope 13-38
- moving and resizing 4-7
- Name property 2-10
- Name, entering 4-12
- on *event_name* action 2-11, 7-6
- output activity to Simulink 4-14
- parallel (AND) decomposition notation 2-8
- properties 4-9
- setting properties for in Explorer 14-8
 - See also* parallel states
- status bar
 - in diagram editor 4-30
- Step button on debugger 13-7
- Stop Simulation button on debugger 13-7
- Stored Integer property of fixed-point data 6-28
- Strong Data Typing with Simulink IO setting
 - fixed-point data 9-7
- subcharts
 - creating 5-5, 5-7
 - definition and description 5-5
 - editing contents 5-10
 - manipulating 5-9
 - navigating through hierarchy of 5-10
 - opening to edit contents 5-9
 - unsubcharting 5-7
 - and supertransitions 5-5
- subfunctions
 - in Embedded MATLAB functions 11-16
- substates
 - creating 4-7
 - decomposition 4-8
- subtraction (-) of fixed-point data 8-24
- subtraction operator (-) 7-13
- Summary of model coverage report 13-51
- superstates
 - event actions in 3-71
- supertransitions 5-12
 - definition and description 5-12
 - drawing into a subchart 5-13
 - drawing out of a subchart 5-16
 - labeling 5-18
 - slits 5-12
- Symbol Autocreation Wizard 12-44
- symbols
 - comment symbols *%,,/** in action language 7-19
 - hexadecimal notation in action language 7-19
 - infinity symbol *inf* in action language 7-20
 - line continuation symbol *...* in action language 7-20
 - literal code symbol *\$* in action language 7-20

- MATLAB display symbol ; in action language 7-20
 - single-precision floating-point symbol F in action language 7-20
 - time symbol t in action language 7-21
 - symbols in action language 7-19
- T**
- tan in action language 7-22
 - tanh in action language 7-22
 - targets
 - adding to machine 12-8
 - build options for custom targets 12-26
 - building 12-5
 - building custom code into 12-29
 - building error messages 12-48
 - building procedure 12-7
 - building with custom code 12-5
 - configuration custom target 12-24
 - configuration rtw target 12-14
 - configuration simulation target 12-11
 - copying/moving in Explorer 14-9
 - custom code 12-5
 - deleting 14-11
 - in Explorer 14-6
 - overview 12-3
 - renaming 14-8
 - setting integer word size for 8-22
 - setting properties for in Explorer 14-8
 - types of 12-3
 - See also* custom targets
 - See also* simulation targets
 - temporal logic events 7-57
 - temporal logic operators 7-50
 - after 7-52
 - at operator 7-54
 - before operator 7-53
 - event notation 7-57
 - every operator 7-55
 - rules for using 7-50
 - temporary data 6-24
 - text
 - replacing 14-12
 - searching 14-12
 - tick keyword 6-19
 - time symbol t in action language 7-21
 - title bar
 - in diagram editor 4-29
 - toolbar
 - in diagram editor 4-29
 - transition actions
 - and condition actions 3-39
 - event broadcasts nested in 3-77
 - notation 2-14
 - transition labels
 - condition 4-18
 - condition action 4-18
 - event 4-18
 - transition action 4-18
 - transition segments
 - backtracking to source 3-69
 - label format 3-58
 - transitions
 - and exclusive (OR) states 2-17, 3-31
 - and exclusive (OR) substates 2-20
 - and exclusive (OR) superstates 2-18
 - arrowhead size 4-20
 - based on events 3-32
 - bowing 4-19
 - breakpoints 4-24
 - changing arrowhead size 4-20
 - condition 4-18
 - condition action 4-18

- connection examples 2-17
 - creating 4-16
 - dashed 4-20
 - debugging conflicting 13-16
 - defined 1-12
 - deleting 4-16
 - Description property 4-24
 - Destination property 4-23
 - Document Link property 4-24
 - events 4-18
 - flow graph types 3-7
 - from common source with connective junctions 3-65
 - from connective junctions based on common event 3-68
 - from multiple sources with connective junctions 3-67
 - hierarchy 2-13
 - label format 4-18
 - Label property 4-24
 - labels
 - action semantics 3-31
 - format 4-18
 - overview 2-14, 4-17
 - moving 4-19
 - moving attach points 4-20
 - moving label 4-20
 - nonsmart
 - anchored connection points 5-27
 - notation 2-17
 - ordering by angular surface position 3-12
 - ordering by hierarchy 3-10
 - ordering by label 3-11
 - overview 2-12
 - Parent property 4-23
 - properties 4-22, 4-23
 - self-loop transitions 4-21
 - setting them smart
 - smart
 - connecting to junctions at 90 degree angles 5-23
 - sliding and maintaining shape 5-22
 - sliding around surfaces 5-20
 - snapping to an invisible grid 5-24
 - straight transitions 4-17
 - substate to substate with events 3-35
 - transition action 4-18
 - transition testing order 3-10
 - valid 2-16
 - valid labels 4-19
 - when they are executed 3-7
 - See also* default transitions
 - See also* inner transitions
 - See also* nonsmart transitions
 - See also* self-loop transitions
 - See also* smart transitions
- trigger
 - event input from Simulink 6-12
 - Trigger property
 - events 6-9
 - triggered update method for Stateflow block 9-11
 - truth tables
 - assigning actions to decisions 10-29
 - calling rules 10-11
 - compared with graphical functions 10-11
 - default decision 10-2
 - defined 10-4
 - editing 10-19
 - entering final actions 10-31
 - entering initial actions 10-31
 - how they are realized 10-62
 - how to interpret 10-2
 - model coverage 10-58
 - model coverage example report 10-58

- model coverage for 10-58
- overspecified 10-55
- properties dialog 10-35
- pseudocode example 10-2
- row and column tooltips 10-54
- seeing generated function for 10-62
- simulation 10-36
- underspecified 10-56
- when generated 10-62
- type cast operations 7-17
- type cast operators
 - explicit cast operator 7-18
 - MATLAB form 7-18
- type operator 7-18
 - using to type other data
 - typing data
 - with type operator 6-48
- Type property
 - data 6-28
 - fixed-point data 8-7
- types
 - inheriting 6-46
- types of data 6-45
- typing data 6-45
 - with other data 6-48

U

- unary actions 7-15
- unary operations 7-15
 - fixed-point data 8-20
- underspecified truth tables 10-56
- undo operation 4-39
 - exceptions 4-39
- Units property of data 6-29
- Up To button in diagram editor 5-10
- update method

- continuous 9-11
- discrete (sample time) 9-11
- inherited 9-11
- Update method property for charts 9-6
- update methods for Stateflow block 9-11
- Use chart names with no mangling coder option 12-22
- Use Strong Data Typing with Simulink I/O property for charts 9-7
- user-written code
 - and Stateflow arrays 7-43
 - C functions 7-24, 7-26

V

- valid transitions 2-16
- Version property of machines 9-10
- View Area field of Search & Replace tool 14-15
- view area of Search & Replace tool 14-19

W

- wakeup keyword 6-19
- Watch in debugger property of data 6-33
- workspace
 - examining the MATLAB workspace 9-24
- wormhole 5-15

Z

- zoom control
 - in Stateflow diagram editor 4-30
- zooming a diagram
 - overview 4-37
 - shortcut keys 4-38
 - using zoom factor selector 4-37