

STATEFLOW[®]

For Use with SIMULINK[®]

Modeling
└───┘

Simulation
└───┘

Implementation
└───┘



User's Guide

Version 2

How to Contact The MathWorks:



508-647-7000 Phone



508-647-7001 Fax



The MathWorks, Inc. Mail
24 Prime Park Way
Natick, MA 01760-1500



<http://www.mathworks.com> Web
<ftp.mathworks.com> Anonymous FTP server
<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
subscribe@mathworks.com Subscribing user registration
service@mathworks.com Order status, license renewals, passcodes
info@mathworks.com Sales, pricing, and general information

Stateflow User's Guide

© COPYRIGHT 1997 - 1999 by The MathWorks, Inc. All Rights Reserved.

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.

U.S. GOVERNMENT: If Licensee is acquiring the Programs on behalf of any unit or agency of the U.S. Government, the following shall apply: (a) For units of the Department of Defense: the Government shall have only the rights specified in the license under which the commercial computer software or commercial software documentation was obtained, as set forth in subparagraph (a) of the Rights in Commercial Computer Software or Commercial Software Documentation Clause at DFARS 227.7202-3, therefore the rights set forth herein shall apply; and (b) For any other unit or agency: NOTICE: Notwithstanding any other lease or license agreement that may pertain to, or accompany the delivery of, the computer software and accompanying documentation, the rights of the Government regarding its use, reproduction, and disclosure are as set forth in Clause 52.227-19 (c)(2) of the FAR.

MATLAB, Simulink, Stateflow, Handle Graphics, and Real-Time Workshop are registered trademarks and the Target Language Compiler is a trademark of The MathWorks, Inc.

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

Printing History:	May 1997	First printing
	January 1998	Revised for 5.2 (online version)
	January 1999	Revised for Stateflow 2.0 (Release 11)

Before You Begin

System Requirements	2
Using Stateflow on a Laptop Computer	3
Related Products	4
MATLAB	4
Simulink	5
Real-Time Workshop	6
Using This Guide	7
Chapter Quick Reference	7
Typographical Conventions	8
Installing Stateflow	9

Introduction

1

Overview	1-2
What Is Stateflow?	1-2
Examples of Stateflow Applications	1-2
Stateflow Components	1-3
Design Approaches	1-3
Finite State Machine Concepts	1-5
What Is a Finite State Machine?	1-5
FSM Representations	1-5
Stateflow Representations	1-5
Notations	1-6
Semantics	1-6

References	1-6
Anatomy of a Model and Machine	1-7
The Simulink Model and Stateflow Machine	1-7
Defining Stateflow Interfaces	1-9
Stateflow Diagram Objects	1-10
Quick Start	1-21
The Power Switch Model	1-21
Creating a Simulink Model	1-22
Creating a Stateflow Diagram	1-25
Defining Input Events	1-28
Defining the Stateflow Interface	1-28
Defining Simulink Parameters	1-29
Parsing the Stateflow Diagram	1-30
Running a Simulation	1-31
Debugging	1-32
Generating Code	1-34

Creating Charts

2

Creating a Chart	2-2
Using the Stateflow Editor	2-5
Displaying Shortcut Menus	2-6
Drawing Objects	2-6
Specifying Object Styles	2-7
Selecting and Deselecting Objects	2-9
Cutting and Pasting Objects	2-10
Copying Objects	2-11
Editing Object Labels	2-11
Exploring Objects in the Editor Window	2-12
Zooming a Diagram	2-12
Creating States	2-13
Moving and Resizing States	2-13

Creating Substates	2-14
Grouping States	2-14
Hiding a State's Contents	2-14
Specifying State Decomposition	2-14
Specifying Activation Order for Parallel States	2-15
Labeling States	2-15
Using the State Properties Dialog	2-16
Naming States	2-17
Defining State Actions	2-17
Outputting State Activity to Simulink	2-19
Creating Boxes	2-20
Creating Transitions	2-21
What Is a Default Transition?	2-21
Creating Default Transitions	2-22
Editing Transition Attach Points	2-22
Labeling Transitions	2-22
Valid Labels	2-23
Changing Arrowhead Size	2-23
Moving Transition Labels	2-24
Using the Transition Properties Dialog	2-24
Creating Junctions	2-26
Changing Size	2-26
Changing Arrowhead Size	2-27
Moving a Junction	2-27
Editing Junction Properties	2-27
Specifying Chart Properties	2-29
Waking Up Charts	2-31
Creating Chart Libraries	2-32
Stateflow Printing Options	2-33
Printing a Stateflow Diagram	2-33
Printing a Stateflow Book	2-33

Defining Events and Data

3

Defining Events	3-2
Adding Events to the Data Dictionary	3-2
Setting Event Properties	3-4
Event Properties	3-5
Naming Events	3-7
Defining Local Events	3-7
Defining Input Events	3-7
Defining Output Events	3-8
Exporting Events	3-9
Importing Events	3-9
Specifying Trigger Types	3-10
Describing Events	3-11
Documenting Events	3-11
Implicit Events	3-11
Defining Data	3-14
Adding Data to the Data Dictionary	3-14
Setting Data Properties	3-16
Data Properties	3-18
Defining Data Arrays	3-20
Defining Input Data	3-21
Defining Output Data	3-22
Associating Ports with Data	3-23
Defining Temporary Data	3-23
Exporting Data	3-24
Importing Data	3-25
Documenting Data	3-25

Defining Stateflow Interfaces

4

Overview	4-2
Interfaces to Stateflow	4-2
Typical Tasks to Define Stateflow Interfaces	4-2

Where to Find More Information on Events and Data	4-3
Defining the Stateflow Block Update Method	4-4
Stateflow Block Update Methods	4-4
Defining a Triggered Stateflow Block	4-5
Defining a Sampled Stateflow Block	4-5
Defining an Inherited Stateflow Block	4-6
Defining a Continuous Stateflow Block	4-7
Defining Output to Simulink Event Triggers	4-10
Overview	4-10
Defining Function Call Output Events	4-10
Defining Edge-Triggered Output Events	4-13
Event Inputs from Simulink	4-16
Add an Event Choosing a Chart as the Parent	4-16
Choose Input from Simulink as the Scope	4-16
Select the Trigger	4-17
Apply the Changes	4-17
Data Inputs from Simulink	4-18
Add a Data Object Choosing a Chart as the Parent	4-18
Choose Input from Simulink as the Scope	4-18
Specify Data Attributes	4-19
Apply and Save the Changes	4-19
Event Outputs to Simulink	4-20
Add an Event Choosing a Chart as the Parent	4-20
Choose Output to Simulink as the Scope	4-20
Apply the Changes	4-21
Data Outputs to Simulink	4-22
Add a Data Object Choosing a Chart as the Parent	4-22
Choose Output to Simulink as the Scope	4-22
Specify Data Attributes	4-23
Apply the Changes	4-23
MATLAB Workspace	4-24
What Is the MATLAB Workspace?	4-24

Using the MATLAB Workspace	4-24
Defining the Interface to External Sources	4-25
What Are External Sources?	4-25
Exported Events	4-25
Imported Events	4-27
Exported Data	4-28
Imported Data	4-29

Exploring and Searching

5

Overview	5-2
Exploring and Searching	5-2
Stateflow Explorer User Interface	5-4
Explorer Main Window	5-4
Moving Objects/Changing Parent	5-7
Moving Objects/Changing Index and Port Order	5-7
Deleting Objects	5-7
Editing Objects	5-7
Stateflow Finder User Interface	5-8
Finder Dialog Box	5-8
Finder Display Area	5-11
Searching for a Wildcard String	5-14
Example	5-14
Searching for an Exact String Match	5-16
Example	5-16
Searching for All Objects of a Specified Object Type	5-18
Example	5-18
Searching Based on Multiple Conditions	5-20
Example	5-20

Searching for an Object to Edit Properties	5-22
Example	5-22

Notations

6

Overview	6-2
What Is Meant by Notation?	6-2
Motivation Behind the Notation	6-2
How Is the Notation Checked?	6-3
What Are the Graphical Objects?	6-3
What Are the Nongraphical Objects?	6-4
What Is the Data Dictionary?	6-4
Keywords	6-5
How Is Hierarchy Represented?	6-6
States	6-9
Overview	6-9
What Is a State Decomposition?	6-9
Active and Inactive States	6-10
What Are Combination States?	6-11
Labeling a State	6-12
Transitions	6-16
What Is a Transition?	6-16
Labeling a Transition	6-17
What Is a Valid Transition?	6-18
Types of Transitions	6-19
What Is a Default Transition?	6-23
Labeling of Default Transitions	6-23
What Is an Inner Transition?	6-26
What Is a Self Loop Transition?	6-30
Connective Junctions	6-31
What Is a Connective Junction?	6-31
What Is Flow Diagram Notation?	6-31

History	6-39
What Is a History Junction?	6-39
History Junctions and Inner Transitions	6-40
Action Language	6-41
What Is an Action Language?	6-41
Objects with Actions	6-41
Transition Action Notation	6-42
State Action Notation	6-42
Action Language Components	6-43
Bit Operations	6-43
Binary Operations	6-43
Unary Operations	6-46
Unary Actions	6-46
Assignment Operations	6-46
User-Written Functions	6-47
ml() Functions	6-48
MATLAB Name Space Operator	6-51
Choosing Between the ml() Function and the ml Name Space Operator	6-53
Data and Event Arguments	6-54
Arrays	6-54
Pointer and Address Operators	6-55
Hexadecimal Notation	6-56
Event Broadcasting	6-56
Directed Event Broadcasting	6-58
Conditions	6-60
Time Symbol	6-61
Literals	6-61
Continuation Symbols	6-62
Comments	6-62
Use of the Semicolon	6-62
Using the Notation Efficiently	6-63
Designing for Efficiency	6-63
Optimizing for Speed	6-63
Optimizing for Code Size	6-64
Optimizing for Memory Use	6-65

Overview	7-2
What Are Semantics?	7-2
List of Semantic Examples	7-2
Event-Driven Effects on Semantics	7-5
What Does Event-Driven Mean?	7-5
Top-Down Processing of Events	7-5
Semantics of Active and Inactive States	7-5
Semantics of State Actions	7-7
Semantics of Transitions	7-7
Transitions to and from Exclusive (OR) States	7-8
Condition Actions	7-13
Default Transitions	7-18
Inner Transitions	7-23
Connective Junctions	7-31
Event Actions	7-40
Parallel (AND) States	7-42
Directed Event Broadcasting	7-54
Execution Order	7-58
Overview	7-58
Execution Order Guidelines	7-58
Parallel (AND) States	7-61
Semantic Rules Summary	7-62
Entering a Chart	7-62
Executing an Active Chart	7-62
Entering a State	7-62

Executing an Active State	7-63
Exiting an Active State	7-63
Executing a Set of Flow Graphs	7-63
Executing an Event Broadcast	7-64

Tutorial: Entering a Stateflow Diagram

8

Overview	8-2
Objectives	8-2
Provided Models for This Chapter	8-2
Carousel Tutorial Example	8-2
Basic Operation	8-2
Designing the Carousel Stateflow Diagram	8-4
Overview	8-4
The Design Questions	8-4
What Are the States?	8-4
What Is a State Decomposition?	8-5
What Are the State Labels?	8-6
Will Junctions Simplify the Design?	8-6
What Are the Transitions?	8-6
Are There Default Transitions?	8-7
Are There Events?	8-7
Is There Data?	8-7
Procedure	8-8
Overview	8-8
Create a Simulink Model	8-8
Sample Stateflow Diagram	8-9
Create and Label the OR States	8-10
Create and Label the Superstates	8-11
Create and Label the AND States	8-13
Create the Transitions	8-14
Label the Transitions	8-15
Create Default Transitions	8-16
Add and Define Event Objects	8-17

Add and Define Data Objects	8-19
Save the Model	8-21
Run the Simulation for the Provided Solution	8-22
Overview	8-22
Carousel Simulink Model Solution	8-22
Carousel Operator's Console	8-22
Carousel Solution Stateflow Diagram	8-23
Procedure	8-23
Run the Carousel Demo	8-25
Overview	8-25
Carousel Operator's Console	8-25
Procedure	8-25

Building Targets

9

Overview	9-2
Target Types	9-2
Building a Target	9-2
How Stateflow Builds Targets	9-3
Setting Up Target Build Tools	9-5
Setting Up Build Tools on UNIX	9-5
Setting Up Build Tools on Windows	9-5
Starting a Build	9-7
Starting from a Target Builder Dialog Box	9-8
Configuring a Target	9-9
Specifying Code Generation Options	9-11
Specifying Custom Code Options	9-14
Parsing	9-17
Parser	9-17
Parse the Machine or the Stateflow Diagram	9-17

Error Messages	9-21
Parser Error Messages	9-21
Code Generation Error Messages	9-22
Compilation Error Messages	9-22

Debugging

10

Overview	10-2
Debugging	10-2
Typical Debugging Tasks	10-2
Including Debugging in the Target Build	10-2
Breakpoints	10-3
Runtime Debugging	10-3
Stateflow Debugger User Interface	10-5
Debugger Main Window	10-5
Status Display Area	10-6
Break On Controls	10-6
Debugger Action Control Buttons	10-7
Animation Controls	10-8
Display Controls	10-8
MATLAB Command Field	10-9
Debugging Runtime Errors	10-10
Example Stateflow Diagram	10-10
Typical Scenario to Debug Runtime Errors	10-11
Create the Model and Stateflow Diagram	10-11
Define the sfun Target	10-12
Invoke the Debugger and Choose Debugging Options	10-12
Start the Simulation	10-12
Debug the Simulation Execution	10-12
Resolve Runtime Error and Repeat	10-13
Solution Stateflow Diagram	10-13
Debugging State Inconsistencies	10-14
What Is a State Inconsistency?	10-14

Causes of State Inconsistency	10-14
Detecting State Inconsistency	10-14
Debugging Conflicting Transitions	10-16
What Is a Conflicting Transition?	10-16
Detecting Conflicting Transitions	10-16
Debugging Data Range Violations	10-18
What Is a Data Range Violation?	10-18
Detecting Data Range Violations	10-18
Debugging Cyclic Behavior	10-19
What Is Cyclic Behavior?	10-19
Detecting Cyclic Behavior	10-19

Reference

11 |

Glossary

A |

Before You Begin

System Requirements	2
Using Stateflow on a Laptop Computer	3
Related Products	4
MATLAB	4
Simulink	5
Real-Time Workshop	6
Using This Guide	7
Chapter Quick Reference.	7
Typographical Conventions.	8
Installing Stateflow	9

System Requirements

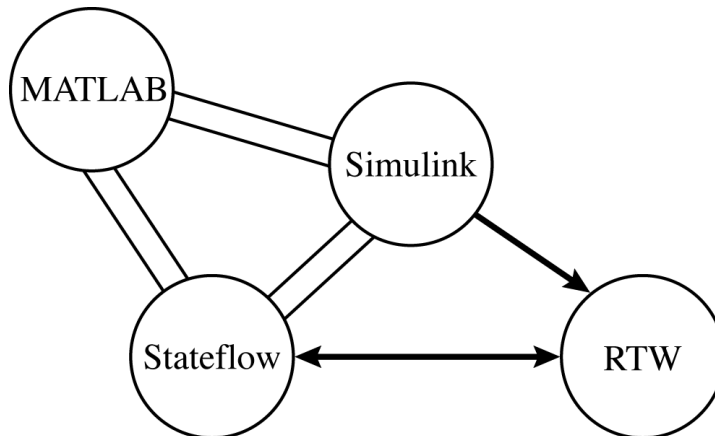
Stateflow[®] is a multiplatform product, running on Microsoft Windows 95, Windows NT, and UNIX systems.

Stateflow requires:

- MATLAB[®] 5.3 (Release11)
- Simulink[®] 3

The UNIX version of Stateflow require a C or C++ compiler for generating code from a Stateflow model. See “Setting Up Target Build Tools” on page 9-5 for more information.

Generating code for the Simulink elements of a Stateflow model requires Real-Time Workshop[®] .



Using Stateflow on a Laptop Computer

If you plan to run the Microsoft Windows version of Stateflow on a laptop computer, you should configure the Windows color palette to use more than 256 colors. Otherwise, you may experience unacceptably slow performance.

To set the Windows graphics palette:

- 1** Click the right mouse button on the Windows desktop to display the desktop menu.
- 2** Select **Properties** from the desktop menu to display the Windows **Display Properties** dialog.
- 3** Select the **Settings** panel on the **Display Properties** dialog.
- 4** Choose a setting that is more than 256 colors from the **Color Palette** colors list.
- 5** Select **OK** to apply the new setting and dismiss the **Display Properties** dialog.

Related Products

MATLAB

MATLAB is a high-performance language for technical computing. It integrates computation, visualization, and programming in an easy-to-use environment where problems and solutions are expressed in familiar mathematical notation. Typical uses include:

- Math and computation
- Algorithm development
- Modeling, simulation, and prototyping
- Data analysis, exploration, and visualization
- Scientific and engineering graphics
- Application development, including graphical user interface building

MATLAB is an interactive system whose basic data element is an array that does not require dimensioning. This allows you to solve many technical computing problems, especially those with matrix and vector formulations, in a fraction of the time it would take to write a program in a scalar noninteractive language such as C or Fortran.

The name MATLAB stands for *matrix laboratory*. MATLAB was originally written to provide easy access to matrix software developed by the LINPACK and EISPACK projects, which together represent the state of the art in software for matrix computation.

MATLAB has evolved over a period of years with input from many users. In university environments, it is the standard instructional tool for introductory and advanced courses in mathematics, engineering, and science. In industry, MATLAB is the tool of choice for high-productivity research, development, and analysis.

MATLAB features a family of application-specific solutions called *toolboxes*. Very important to most users of MATLAB, toolboxes allow you to *learn* and *apply* specialized technology. Toolboxes are comprehensive collections of MATLAB functions (M-files) that extend the MATLAB environment to solve particular classes of problems. Areas in which toolboxes are available include signal processing, control systems, neural networks, fuzzy logic, wavelets, simulation, and many others.

See the MATLAB documentation set for more information.

Simulink

Simulink is a 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.

For modeling, Simulink provides a graphical user interface (GUI) for building models as block diagrams, using click-and-drag mouse operations. With this interface, you can draw the models just as you would with pencil and paper (or as most textbooks depict them). This is a far cry from previous simulation packages that require you to formulate differential equations and difference equations in a language or program. Simulink includes a comprehensive block library of sinks, sources, linear and nonlinear components, and connectors. You can also customize and create your own blocks.

Models are hierarchical, so you can build models using both top-down and bottom-up approaches. You can view the system at a high-level, then double-click on blocks to go down through the levels to see increasing levels of model detail. This approach provides insight into how a model is organized and how its parts interact.

After you define a model, you can simulate it, using a choice of integration methods, either from the Simulink menus or by entering commands in MATLAB's command window. The menus are particularly convenient for interactive work, while the command-line approach is very useful for running a batch of simulations (for example, if you are doing Monte Carlo simulations or want to sweep a parameter across a range of values). Using scopes and other display blocks, you can see the simulation results while the simulation is running. In addition, you can change parameters and immediately see what happens, for "what if" exploration. The simulation results can be put in the MATLAB workspace for postprocessing and visualization.

Model analysis tools include linearization and trimming tools, which can be accessed from the MATLAB command line, plus the many tools in MATLAB and its application toolboxes. And because MATLAB and Simulink are integrated, you can simulate, analyze, and revise your models in either environment at any point.

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 control flow by combining Stateflow blocks with the standard Simulink blocksets. These combined models are simulated using Simulink.

The *Using Simulink* document 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.

Real-Time Workshop

Real-Time Workshop (RTW) is an 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.

Stateflow works in conjunction with Real-Time Workshop in these ways:

- Stateflow allows you to create and simulate Simulink models containing state machines.
- Stateflow generates stand-alone executables defined by state machines.
- Real-Time Workshop generates stand-alone executables from Simulink models, including those containing state machines defined by Stateflow.

Using This Guide

Chapter Quick Reference

If you are new to the Stateflow environment, go to Chapter 1, “Introduction,” to get an overview and a quick start.

If you want to get started right away, go to Chapter 8, “Tutorial: Entering a Stateflow Diagram” for step by step instruction on creating a Stateflow diagram.

For information on creating charts, refer to Chapter 2, “Creating Charts.”

Chapter 3, “Defining Events and Data,” describes the nongraphical objects that are essential to completing and defining interfaces to the Stateflow diagram.

Chapter 4, “Defining Stateflow Interfaces,” describes the types of interfaces and how to define them.

For information on using the Stateflow Explorer and the Stateflow Finder, see Chapter 5, “Exploring and Searching.”

Use Chapter 6, “Notations,” Chapter 7, “Semantics,” and Chapter 9, “Building Targets” together to understand the language used to communicate Stateflow diagram design information, how that notation is interpreted and implemented behind the scenes, and how to generate code.

See Chapter 10, “Debugging,” for information on debugging your simulation.

See Chapter 11, “Reference,” for information on specific functions and their syntax.

See the “Glossary” for definitions of key terms and concepts.

Typographical Conventions

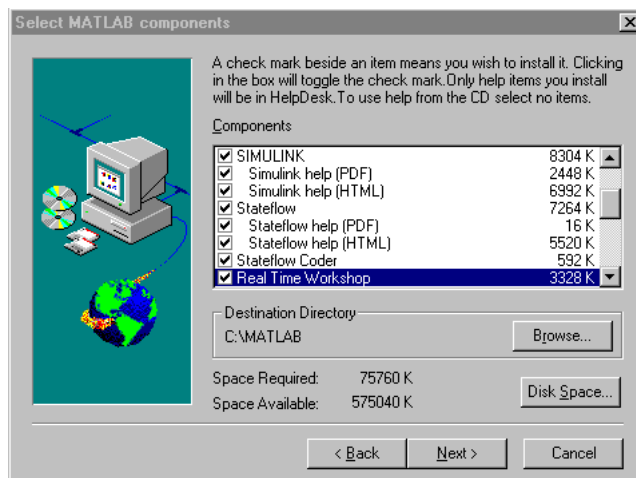
To Indicate...	This Guide Uses...	Example
New terms	<i>Italics</i>	An <i>array</i> is an ordered collection of information.
Keys, menu names, items, and GUI controls	Boldface with an initial capital letter	Choose the File menu.
Function names, variables, example code, user-entered text	Monospace type	The <code>sfoopen</code> command opens a model.

Installing Stateflow

Your platform-specific MATLAB *Installation Guide* provides essentially all of the information you need to install Stateflow.

Prior to installing Stateflow, you must obtain a License File or Personal License Password from The MathWorks. The License File or Personal License Password identifies the products you are permitted to install and use.

As the installation process proceeds, a screen similar to this, where you can indicate which products to install, is displayed.



Stateflow and Stateflow Coder have certain product prerequisites that must be met for proper installation and execution.

Licensed Product	Prerequisite Products	Additional Information
Simulink 3	MATLAB 5.3 (Release 11)	Allows installation of Simulink and Stateflow in Demo mode.
Stateflow	Simulink 3	
Stateflow Coder	Stateflow	Same as Stateflow.

If you experience installation difficulties and have Web access, connect to The MathWorks home page (<http://www.mathworks.com>). Look for the license manager and installation information under the Tech Notes/FAQ link under Tech Support Info.

Introduction

Overview.	1-2
What Is Stateflow?	1-2
Examples of Stateflow Applications	1-2
Stateflow Components	1-3
Design Approaches	1-3
Finite State Machine Concepts.	1-5
What Is a Finite State Machine?	1-5
FSM Representations	1-5
Stateflow Representations	1-5
Notations	1-6
Semantics	1-6
References	1-6
Anatomy of a Model and Machine	1-7
The Simulink Model and Stateflow Machine	1-7
Defining Stateflow Interfaces	1-9
Stateflow Diagram Objects	1-10
Quick Start.	1-21
The Power Switch Model	1-21
Creating a Simulink Model	1-22
Creating a Stateflow Diagram	1-25
Defining Input Events	1-28
Defining the Stateflow Interface	1-28
Define Simulink Parameters	1-29
Parsing the Stateflow Diagram	1-30
Running a Simulation	1-31
Debugging	1-32
Generating Code	1-34

Overview

What Is Stateflow?

Stateflow is a powerful graphical design and development tool for complex control and supervisory logic problems. Using Stateflow you can:

- Visually model and simulate complex reactive systems based on *finite state machine* theory.
- Design and develop deterministic, supervisory control systems.
- Easily modify your design, evaluate the results, and verify the system's behavior at any stage of your design.
- Automatically generate integer or floating-point code directly from your design.
- Take advantage of the integration with the MATLAB and Simulink environments to model, simulate, and analyze your system.

Stateflow allows you to use flow diagram notation and state transition notation seamlessly in the same Stateflow diagram. Flow diagram notation is essentially logic represented without the use of states. In some cases, using flow diagram notation is a closer representation of the system's logic and avoids the use of unnecessary states. Flow diagram notation is an effective way to represent common code structures like for loops and if-then-else constructs.

Stateflow also provides clear, concise descriptions of complex system behavior using finite state machine theory, flow diagram notations, and state-transition diagrams. Stateflow brings system specification and design closer together. It is easy to create designs, consider various scenarios, and iterate until the Stateflow diagram models the desired behavior.

Examples of Stateflow Applications

A few of the types of applications that benefit from using the capabilities of Stateflow are:

- Embedded systems
 - Avionics (planes)
 - Automotive (cars)

-
- Telecommunications (e.g., routing algorithms)
 - Commercial (computer peripherals, appliances, etc.)
 - Programmable logic controllers (PLCs) (process control)
 - Industrial (machinery)
 - Man-machine interface (MMI)
 - Graphical user interface (GUI)
 - Hybrid systems
 - Air traffic control systems (Digital Signal Processing (DSP) + Control + MMI)

Stateflow Components

Stateflow consists of these primary components:

- Stateflow graphics editor (see Chapter 2, “Creating Charts”)
- Stateflow Explorer (see Chapter 5, “Exploring and Searching”)
- Stateflow Finder (see Chapter 5, “Exploring and Searching”)
- Stateflow simulation code generator (see Chapter 9, “Building Targets”)
- Stateflow Debugger (see Chapter 10, “Debugging”)

Stateflow Coder is a separately available product and generates code for nonsimulation targets. (See Chapter 9, “Building Targets” for information relevant to Stateflow Coder.)

Stateflow Dynamic Checker supports runtime checking for conditions such as cyclic behavior and data range violations. The Dynamic Checker is currently available if you have a Stateflow license.

Design Approaches

Stateflow is used together with Simulink and optionally with the Real-Time Workshop (RTW), all running on top of MATLAB. MATLAB provides access to data, high-level programming, and visualization tools. The control behavior that Stateflow models complements the algorithmic behavior modeled in Simulink. Simulink supports development of continuous-time and discrete-time dynamic systems in a graphical block diagram environment. Stateflow diagrams are incorporated into Simulink models to enhance the new

event-driven capabilities in Simulink (such as conditionally executed subsystems and event detection).

You can design a model starting with a Stateflow (control) perspective and then later build the Simulink model. You can also design a model starting from a Simulink (algorithmic) perspective and then later add Stateflow diagrams. You may have an existing Simulink model that would benefit by replacing Simulink logic blocks with Stateflow diagrams. The approach you use determines how, and in what sequence, you develop various parts of the model.

The collection of all Stateflow blocks in the Simulink model is a machine. When using Simulink together with Stateflow for simulation, Stateflow generates an S-function (MEX-file) for each Stateflow machine to support model simulation. This generated code is a simulation target and is called the `sfun` target within Stateflow.

Stateflow Coder generates integer or floating-point code based on the Stateflow machine. Real-Time Workshop generates code from the Simulink portion of the model and provides a framework for running generated Stateflow code in real-time. The code generated by Stateflow Coder is seamlessly incorporated into code generated by Real-Time Workshop. You may want to design a solution that targets code generated from both products for a specific platform. This generated code is specifically a RTW target and within Stateflow is called the `rtw` target.

Using Stateflow and Stateflow Coder you can generate code exclusively for the Stateflow machine portion of the Simulink model. This generated code is for stand-alone (nonsimulation) targets. You uniquely name this target within Stateflow.

In summary, the primary design approach options are:

- Use Stateflow together with Simulink for simulation.
- Use Stateflow, Stateflow Coder, Simulink, and Real-Time Workshop to generate target code for the complete model.
- Use Stateflow and Stateflow Coder to generate target code for a machine.

Finite State Machine Concepts

What Is a Finite State Machine?

A *finite state machine* (FSM) is a representation of an event-driven (reactive) system. In an event-driven system, the system transitions 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, neutral, drive, reverse, and so on. The system transitions from one state to another when a driver shifts the stick from one position to another, for example, from park to neutral.

FSM Representations

Traditionally, designers used truth tables to represent relationships among the inputs, outputs, and states of an FSM. 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 (STDs) 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 STDs and bubble diagrams provide.

Notations

A 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:

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

See Chapter 6, “Notations,” 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 7, “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.

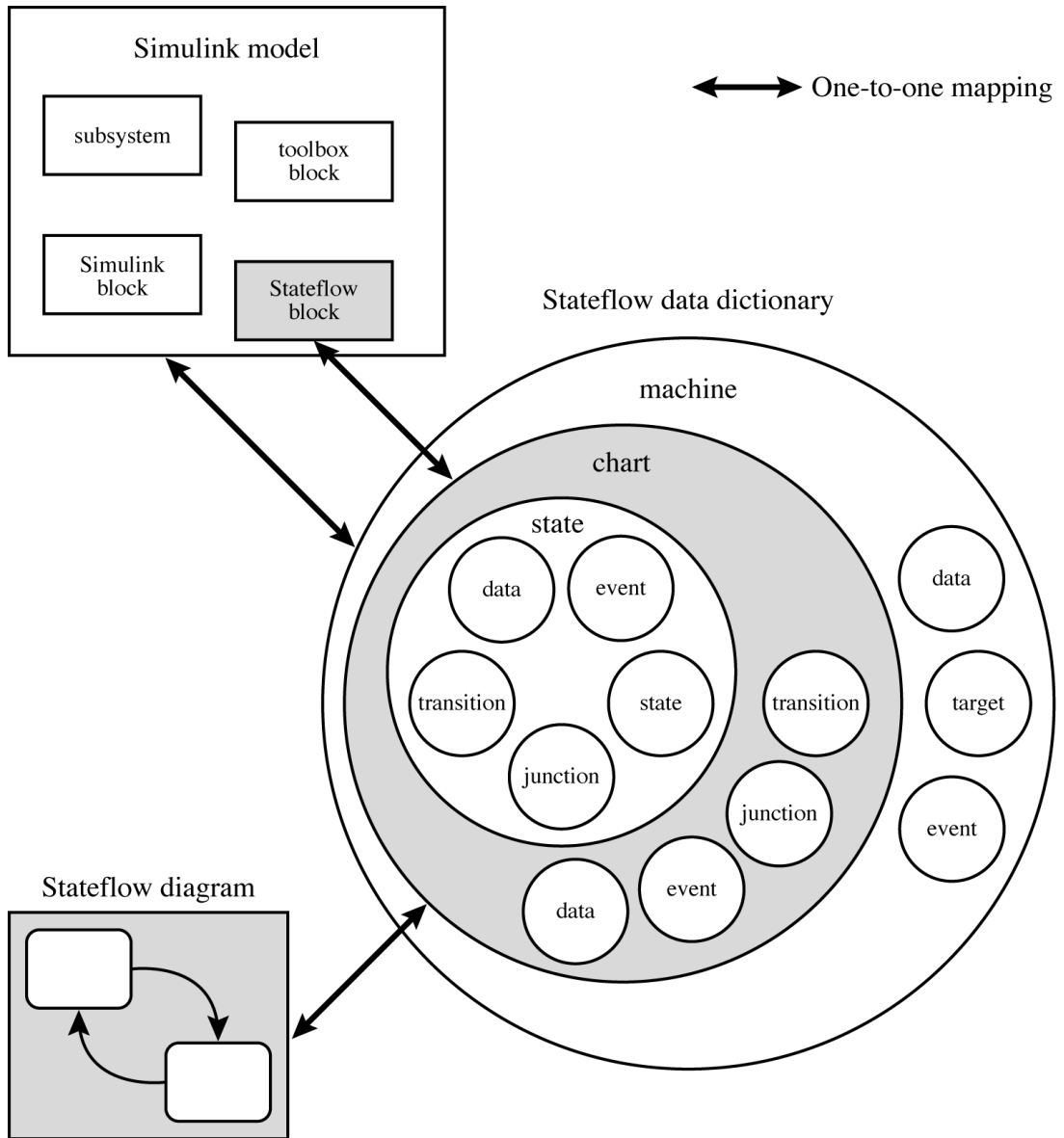
Anatomy of a Model and Machine

The Simulink Model and 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 (states, transitions, connective junctions, and history junctions) and nongraphical (event, data, and target) objects.

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 the graphical and nongraphical objects. The data dictionary is the repository for all Stateflow objects.



Stateflow scoping rules dictate where the types of 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 downwards (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.

Defining Stateflow Interfaces

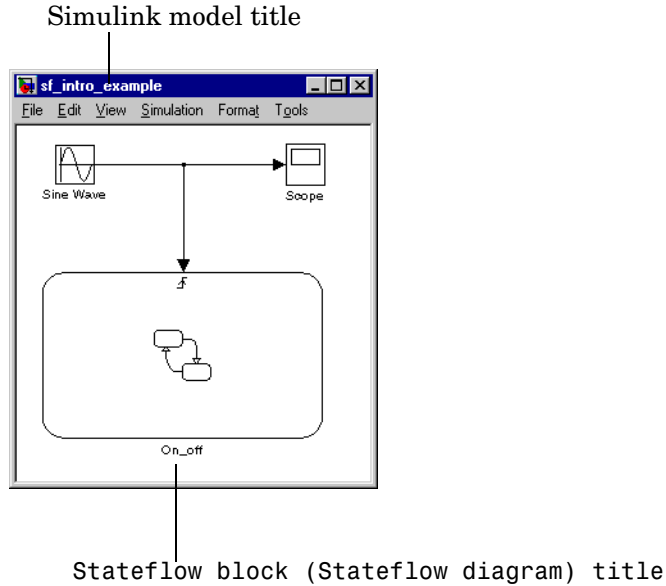
Each Stateflow block corresponds to a single Stateflow diagram. The Stateflow block interfaces to its Simulink model. The Stateflow block may interface to code sources external to the Simulink model (data, events, custom code).

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 this 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 Example.



See “Defining Input Events” on page 1-28 and Chapter 4, “Defining Stateflow Interfaces” for more information.

Stateflow Diagram Objects

This sample Stateflow diagram highlights some key graphical components. The sections that follow describe these graphical components as well as some nongraphical objects and related concepts in greater detail.

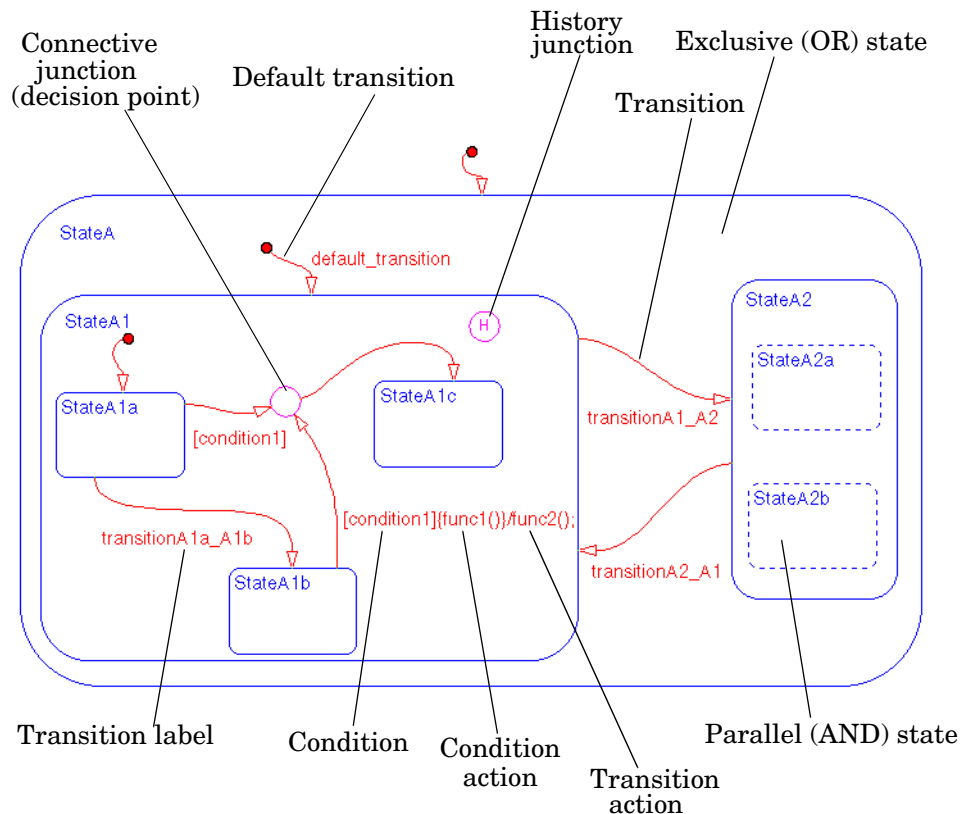


Figure 1-1: Graphical Components

States

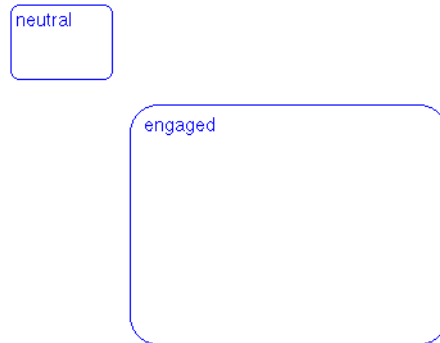
A *state* describes a mode of an event-driven system. The activity or inactivity of the states dynamically changes based on events and conditions.

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 figure, StateA1 is a child in the hierarchy to StateA.

A state also has history. History provides an efficient means of basing future activity on past activity.

States have labels that can specify actions executed in a sequence based upon action type. The action types are: entry, during, exit, and on.

In an automatic transmission example, the transmission can either be in neutral or engaged in a gear. Two states of the transmission system are neutral and engaged.

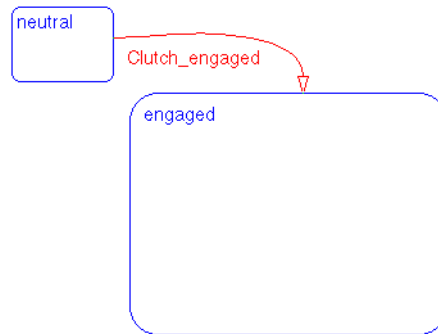


Stateflow provides two types of states: parallel (AND) and exclusive (OR) states. You represent parallelism with AND (parallel) states. The transmission example shows exclusive (OR) states. Exclusive (OR) states are used to describe modes that are mutually exclusive. The system is either in the neutral state or the engaged state at any one time.

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 figure, 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`.



Events

Events drive the Stateflow diagram execution. Events are nongraphical objects and are thus not represented directly in the figure. 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. Data are nongraphical objects and are thus not represented directly in the figure.

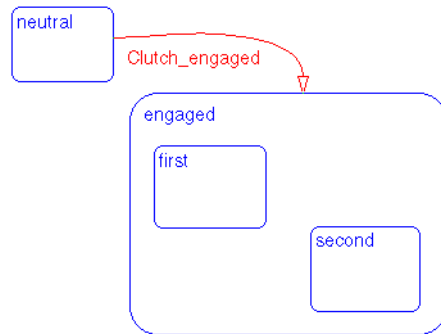
Data objects are created and modified using the Stateflow Explorer. Data objects can be created at any level in the hierarchy. Data objects have properties such as a scope. The scope defines whether the data object is:

- 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
- Non-persistent 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

Hierarchy

Hierarchy enables you to organize complex systems by defining a parent and offspring object structure. A hierarchical design usually reduces the number of transitions and produces neat, manageable diagrams. Stateflow supports a hierarchical organization of states such that states can exist within other states. Stateflow represents state hierarchy with superstates and substates.

For example, this Stateflow diagram has a superstate that contains two substates.



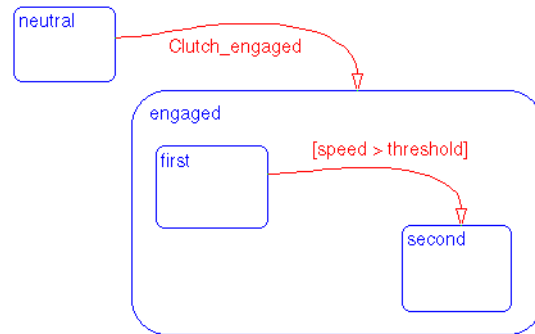
The engaged superstate contains the first and second substates. The engaged superstate is the parent in the hierarchy to the states first and second. When the event `clutch_engaged` occurs, the system transitions out of the neutral state to the engaged superstate. Transitions within the engaged superstate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

A transition out of a higher level, or *superstate*, also implies transitions out of any active substates of the superstate. Transitions can cross superstate boundaries to specify a substate destination. If a substate is active its parent superstate is also active.

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, `[condition]` 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 Boolean condition `[speed > threshold]` is true.

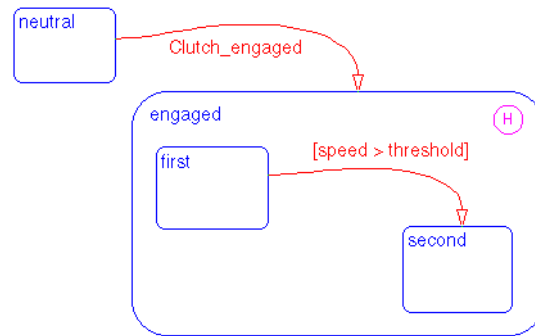


History Junction

History provides the means to specify the destination substate of a transition based on historical information. If a superstate with exclusive (OR) decomposition has a history junction, the transition to 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,

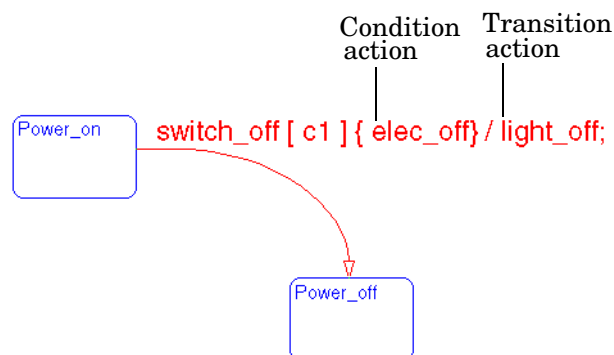
the substate that becomes active, either first or second, is based on which of those substates was most recently active.



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. In the figure, the transition segment from StateA1b to the connective junction is labeled with a condition action (`func1()`) and a transition action (`func2()`). The semantics of how and why actions take place are discussed throughout the examples in Chapter 7, “Semantics.”

Transitions can have *condition* actions and *transition* actions, as shown in this example.



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();
```

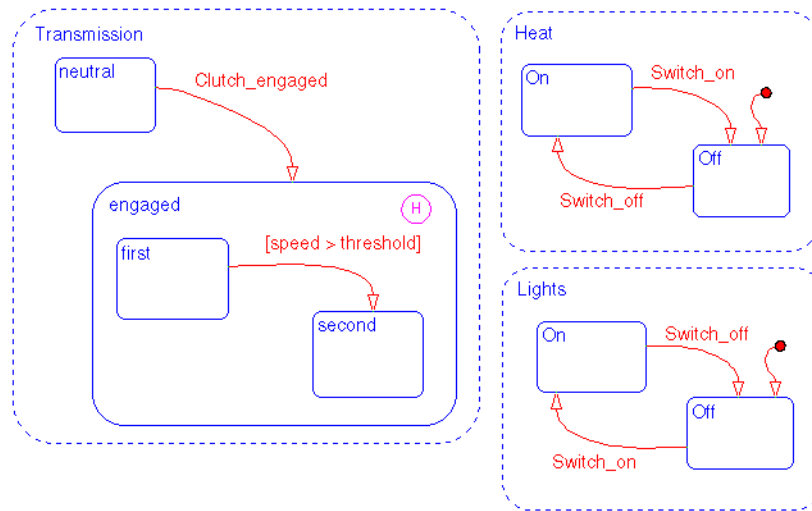
The *action language* defines the types of actions you can specify and their associated notations. An action can be a function call, an event to be broadcast, a variable to be assigned a value, etc.

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 section titled “What Is an Action Language?” on page 6-41.

Parallelism

A system with *parallelism* has two or more states that can be active at the same time. The activity of each parallel state is essentially independent of other states. In the figure, StateA2a and StateA2b are parallel (AND) states. StateA2 has parallel (AND) state decomposition.

For example, this Stateflow diagram has parallel superstate decomposition.



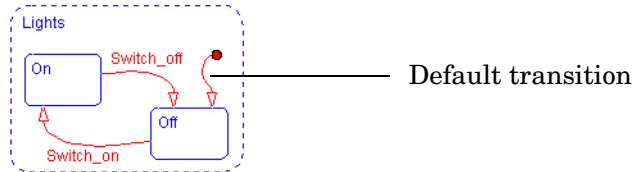
The transmission, heating, and light systems are parallel subsystems in a car. They exist in parallel and are physically independent of each other. There are many other parallel components in a car, such as the braking and windshield wiper subsystems.

You represent parallelism in Stateflow by specifying parallel (AND) state decomposition. Parallel (AND) states are displayed as dashed rectangles.

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. In the figure, when StateA is active, by default StateA1 is also active. Without the default transition to StateA1, there is ambiguity in whether StateA1 or StateA2 should be active.

In the 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.

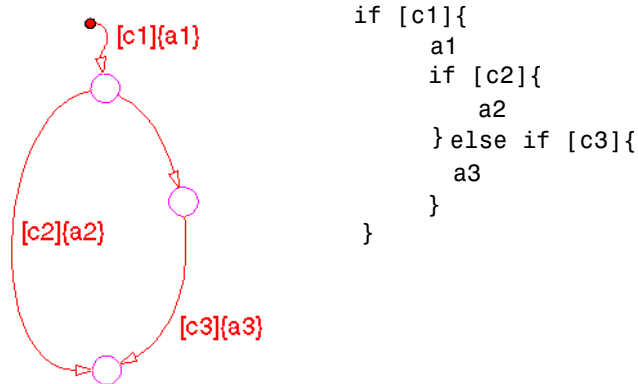


Default transitions specify which exclusive (OR) substate in a superstate the system enters by default, in the absence of any information. History junctions override default transition paths in superstates with exclusive (OR) decomposition.

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 figure, the connective junction is used as a decision point for two transition segments that complete at StateA1c.

This example shows how connective junctions (displayed as small circles) are used to represent the flow of an if code structure.

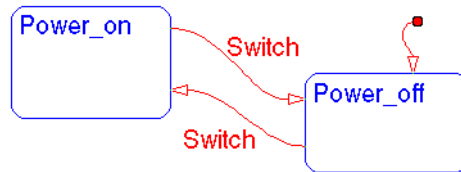


Quick Start

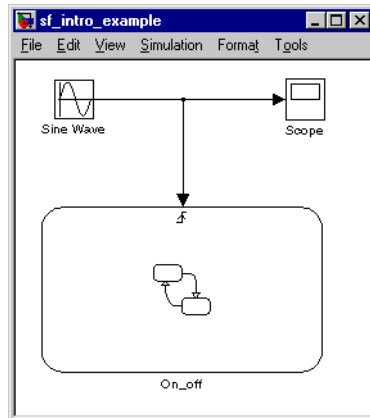
This section provides you with a quick introduction to using Stateflow. In this section, you will use Stateflow to create, run, and debug a model of a simple power switch.

The Power Switch Model

The following figure shows a Stateflow diagram that represents the power switch we intend to model.



Here is a sample of the completed Simulink model.



When you simulate this model, the generation of the input event from Simulink, Switch, will toggle the activity of the states between **Power_on** and **Power_off**.

Creating a Simulink Model

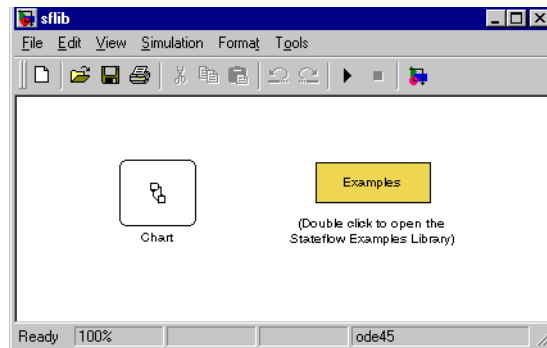
Opening the Stateflow model window is the first step toward creating a Simulink model with a Stateflow block. By default, an untitled Simulink model with an untitled, empty Stateflow block is created for you when you open the Stateflow model window. You can either start with the default empty model or copy the untitled Stateflow block into any Simulink model to include a Stateflow diagram in an existing Simulink model.

These steps describe how to create a Simulink model with a Stateflow block, label the Stateflow block, and save the model:

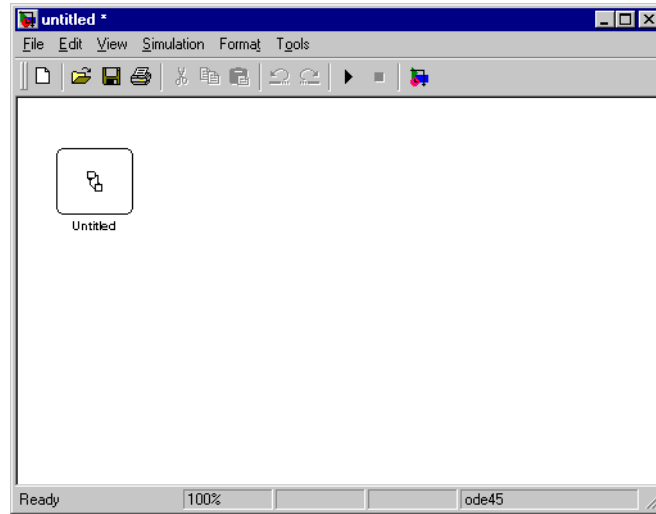
- 1 Display the Stateflow model window.

At the MATLAB prompt enter `stateflow`.

MATLAB displays the Stateflow block library and a Stateflow model window. Here is the library window.

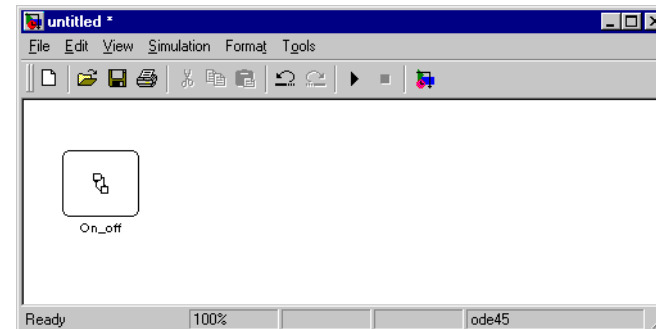


The window consists of an untitled Stateflow block icon, an Examples block, and a manual switch. This is the untitled Simulink model window with an untitled Stateflow block.



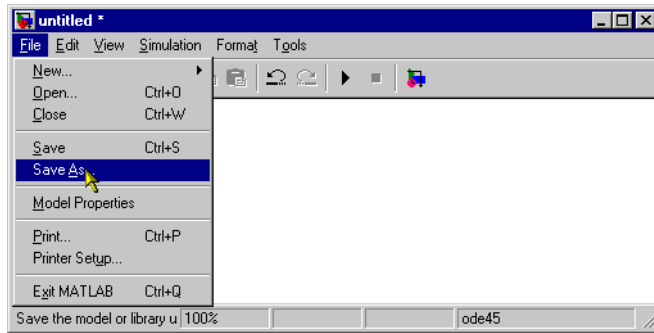
2 Label the Stateflow block.

Label the Stateflow block in the new untitled model by clicking in the text area and replacing the text "Untitled" with the text On_off.



3 Save the model.

Choose **Save As** from the **File** menu of the Simulink model window. Enter a model title.



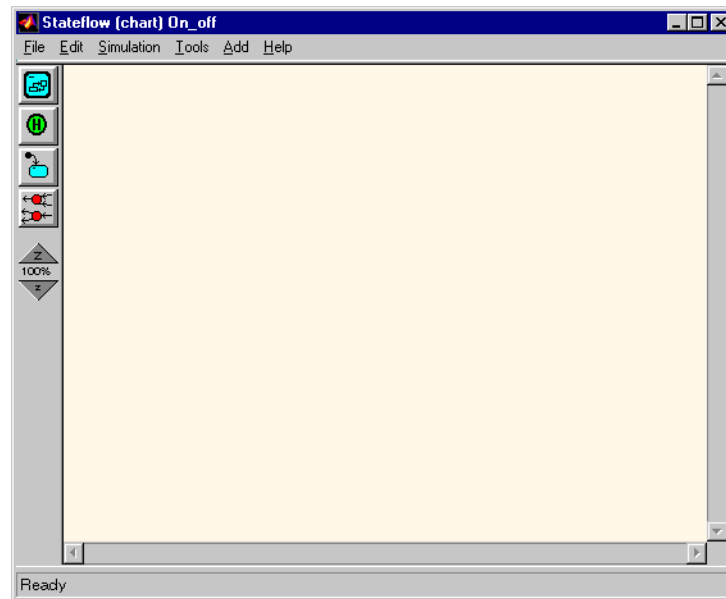
You can also save the model by choosing **Save** or **Save As** from the Stateflow graphics editor **File** menu. Saving the model either from Simulink or from the graphics editor saves all contents of the Simulink model.

Creating a Stateflow Diagram


These steps describe how to create a simple Stateflow diagram using the graphics editor.

- 1 Invoke the graphics editor.

Double-click on the Stateflow block in the Simulink model window to invoke the graphics editor window.

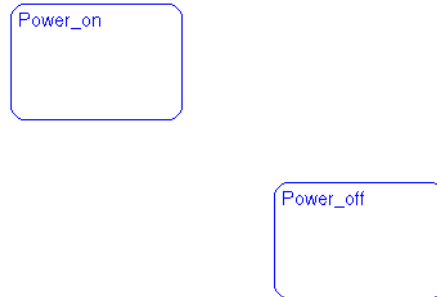


- 2 Create states.

Click on the **State** button  in the toolbar. Click in the drawing area to place the state in the drawing area. Position the cursor over that state, click the right mouse button, and drag to make a copy of the state. Release the right mouse button to drop the state at that location.

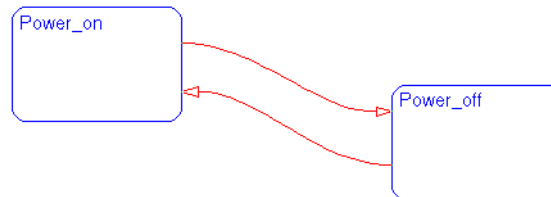
3 Label states.

Click on the ? character within each state to enter each state label. Label the states with the titles `Power_on` and `Power_off`. Deselect the state to exit the edit. Your Stateflow diagram should look similar to this sample.

**4** Create transitions.

Draw a transition starting from `Power_on` and ending at `Power_off`. Place the cursor at a straight portion of the border of the `Power_on` state. Click the border when the cursor changes to a crosshair. Without releasing the mouse button, drag the mouse to a straight portion on the border of the `Power_off` state. When the transition snaps to the border of the `Power_off` state, release the mouse button. (The crosshair will not appear if you place the cursor on a corner, since corners are used for resizing.)

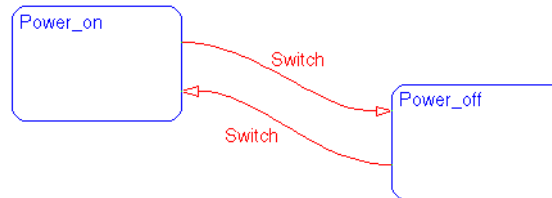
Draw another transition starting from `Power_off` and ending on `Power_on`. Your Stateflow diagram should look similar to this sample.




5 Label the transitions.

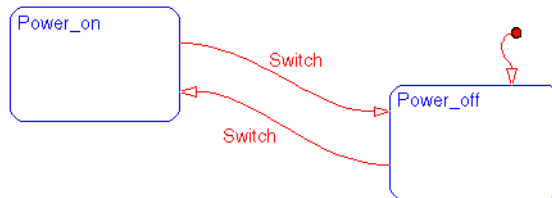
Click on the transition from Power_on to Power_off to select it. Click on the ? alongside the transition and enter the label Switch. Press the **Escape** key to deselect the transition label and exit the edit.

Label the transition from Power_off to Power_on with the same text, Switch. Your Stateflow diagram should look similar to this sample.



6 Add a default transition.

Click and release the mouse on the **Default Transition** button  in the toolbar. Drag the mouse to a straight portion on the border of the Power_off state. Click and release the mouse when the arrowhead snaps to the border of the Power_off state. Your Stateflow diagram should look similar to this sample.



For More Information

For more information on creating Stateflow diagrams using the graphics editor see Chapter 2, “Using the Stateflow Editor.”

Defining Input Events

Add and define input events within the Stateflow diagram:

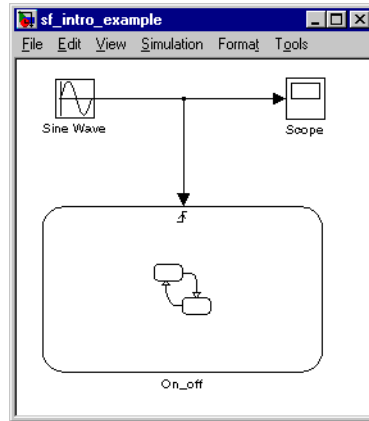
- 1 Choose **Explore** from the graphics editor **Tools** menu to invoke the Explorer.
- 2 Double-click on the machine name (same as the Simulink model name) in the **Object Hierarchy** list.
- 3 Click on the On_off chart entry in the Object Hierarchy list.
- 4 Select **Event** from the **Add** menu.
- 5 Double-click the entry for the event to display its properties dialog.
- 6 Enter Switch in the **Name** field of the **Event properties** dialog box.
- 7 Select **Input from Simulink** as the **Scope** value.
- 8 Select **Rising Edge** as the **Trigger** type.
- 9 Click on the **OK** button to apply the changes and close the window.
- 10 Choose **Close** from the Explorer **File** menu to close the Explorer.

Defining the Stateflow Interface

Make connections in the Simulink model between other blocks and the Stateflow block:

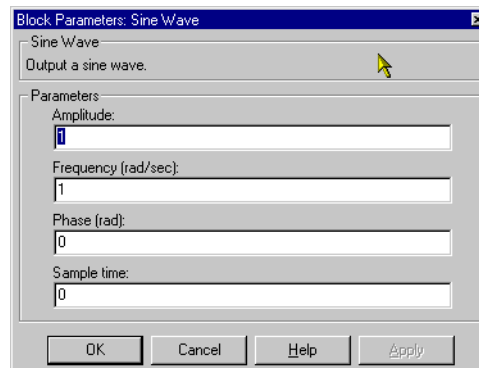
- 1 Enter `simulink` in the MATLAB command window to invoke Simulink.
- 2 Add a Sine Wave block (located in the Simulink Sources block library) and connect it to the input trigger port of the Stateflow block.

- 3 Add a Scope block (located in the Simulink Sinks block library) and connect it to the Sine Wave block output as well. Your model should look similar to this.



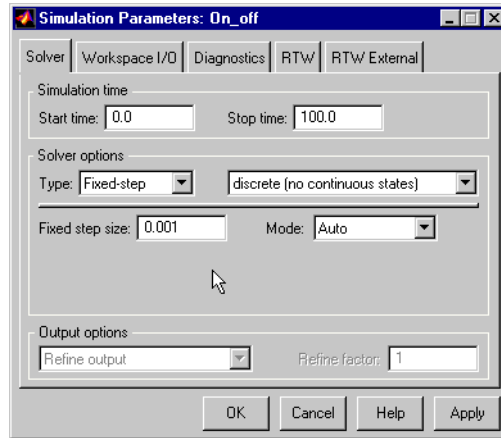
Defining Simulink Parameters

- 1 Double-click on the Sine Wave block and edit the parameters as shown in this example dialog box.



Click on the **OK** button to apply the changes and close the dialog box.

- 2 Choose **Parameters** from the **Simulation** menu of the Simulink model window and edit the values to match the values in this dialog box.



For More Information

See Chapter 4, “Defining Stateflow Interfaces.”

Parsing the Stateflow Diagram

Parsing the Stateflow diagram ensures that the notations you have specified are valid and correct. To parse the Stateflow diagram, choose **Parse Diagram** from the **Tools** menu of the graphics editor. Informational messages are displayed in the MATLAB command window. Any error messages are displayed in red. If no red error messages appear, the parse operation is successful and the text Done is displayed.

For More Information

See “How Stateflow Builds Targets” on page 9-3.

Running a Simulation

Note Running a simulation may require setting up the tools used to build Stateflow targets. See “Setting Up Target Build Tools” on page 9-5 for more information.

These steps show how to run a simulation:

- 1 Ensure that the Stateflow diagram and the Scope block are open.

Double-click on the `On_off` Stateflow block to display the Stateflow diagram. Double-click on the Scope block to display the output of the Sine wave.

- 2 Select **Open Simulation Target** from the graphics editor **Tools** menu.

The **Simulation Target Builder** dialog box appears.

- 3 Select **Coder Options** on the Simulation Target Builder dialog box.

The **Simulation Coder Options** dialog box appears.

- 4 Ensure that the checkbox to **Enable Debugging/Animation** is checked. Click on the **OK** button to apply the change. Close the **Simulation Coder Options** and the **Simulation Target Builder** dialog boxes.

- 5 Select **Debug** from the graphics editor **Tools** menu. Ensure that the **Enabled** radio button under **Animation** is checked to enable Stateflow diagram animation. Click on the **Close** button to apply the change and close the window.

- 6 Choose **Start** from the graphics editor **Simulation** menu to start a simulation of the model.

By default the S-function is the simulation target for any Stateflow blocks. Informational and error messages related to the S-function code generation for Stateflow blocks are displayed in the MATLAB command window. Once the simulation is started and the target is built, the graphics editor becomes read-only (frozen).

The input from the Sine block is defined as the **Input from Simulink** event Switch. When the simulation starts the Stateflow diagram is animated reflecting the state changes triggered by the input sine wave. Each input event of Switch toggles the model between the Power_off and Power_on state.

- 7** Choose **Stop** from the graphics editor **Simulation** menu to stop a simulation. Once the simulation is stopped the graphics editor becomes writeable (unfrozen).

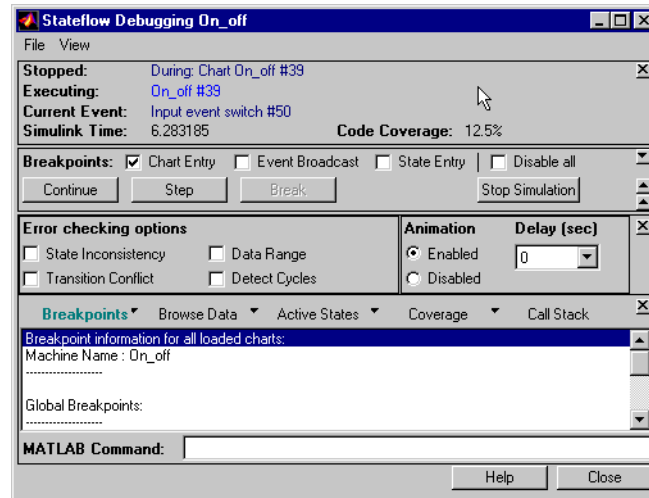
Debugging

The Stateflow Debugger supports functions like single stepping, animating, and running up to a designated breakpoint and then stopping.

These steps show how to step through the simulation using the Debugger:

- 1** Display the Debugger by choosing **Debug** from the **Tools** menu of the graphics editor.
- 2** Click on the **Breakpoints: Chart Entry** checkbox to specify you want the Debugger to stop the simulation execution when the chart is entered.

- 3 Click on the **Start** button to start the simulation. Informational and error messages related to the S-function code generation for Stateflow blocks are displayed in the MATLAB command window. When the target is built, the graphics editor becomes read-only (frozen) and the Debugger window will be updated and look similar to this.



- 4 Click on the **Step** button to proceed one step at a time through the simulation. Take note of the top portion of the Debugger window where this information is displayed:
- Where the simulation is stopped
 - What is executing
 - The current event
 - The simulation time
 - The current code coverage percentage

Watch the graphics editor window as you click on the **Step** button to see each transition and state animated when it is executing. After both `Power_off` and `Power_on` have become active by stepping through the simulation, the code coverage indicates 100%.

- 5** Choose **Stop** from the graphics editor **Simulation** menu to stop a simulation. Once the simulation is stopped the graphics editor becomes writeable (unfrozen).
- 6** Click on the **Close** button in the Debugger window.
- 7** Choose **Close** from the **File** menu in the Simulink model window.

For More Information

See Chapter 10, “Debugging” for more information beyond the debugging topics in this section.

Generating Code

If you have the Stateflow Coder product, you can generate stand-alone code suitable to target for a particular processor. Stateflow generates simulation code (sfun target) for the On_off Stateflow block.

The On_off Stateflow block and its Simulink model is not designed or intended to generate stand-alone code for a specific target processor.

For More Information

See Chapter 9, “Building Targets” for more information on code generation.

Creating Charts

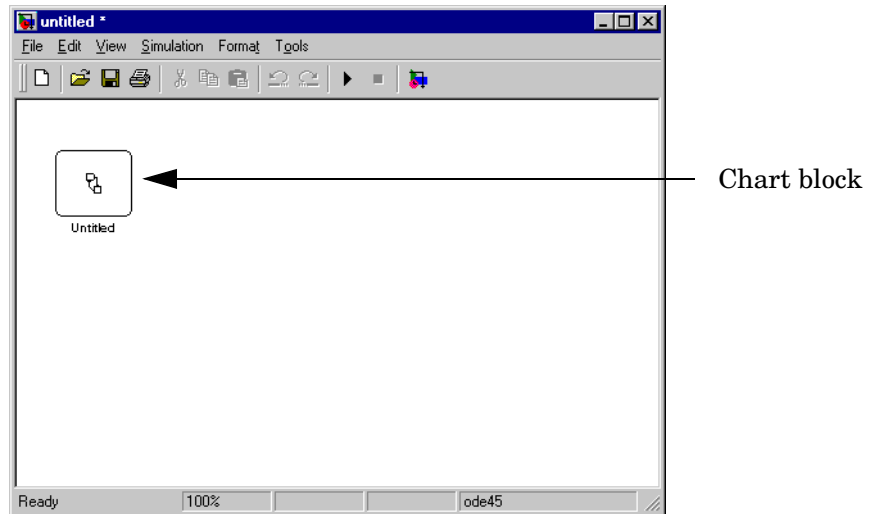
Creating a Chart	2-2
Using the Stateflow Editor	2-5
Creating States	2-13
Creating Boxes	2-20
Creating Transitions	2-21
Creating Junctions	2-26
Specifying Chart Properties	2-29
Waking Up Charts	2-31
Creating Chart Libraries	2-32
Stateflow Printing Options	2-33

Creating a Chart

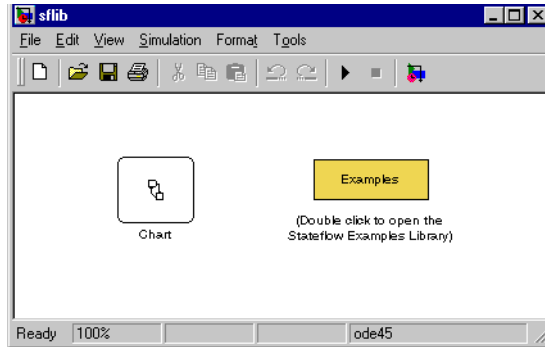
To create a Stateflow chart:

- 1 Create a new model with an empty chart block or copy an empty chart from the Stateflow block library into your model.

To create a new model with an empty chart, enter `sfnew` or `stateflow` at the MATLAB command prompt. The first command creates a new model.



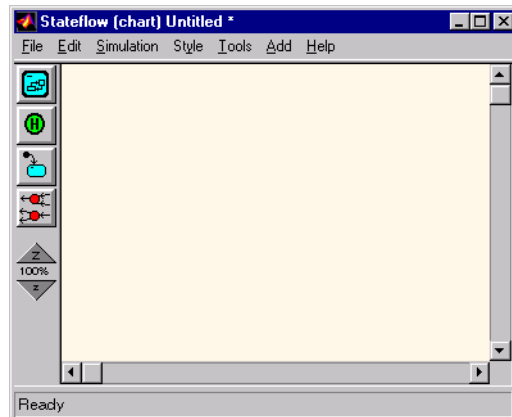
The second command also displays the Stateflow block library in case you want to create multiple charts in your model.



For information on creating your own chart libraries, see “Creating Chart Libraries” on page 2-32.

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

Stateflow opens the empty chart in a Stateflow editor window.



- 3 Use the Stateflow editor to draw and connect states representing the desired state machine or a component of the desired state machine.

See “Using the Stateflow Editor” on page 2-5 for more information.

- 4 Specify a wakeup method for the chart.

See “Specifying Chart Properties” on page 2-29.

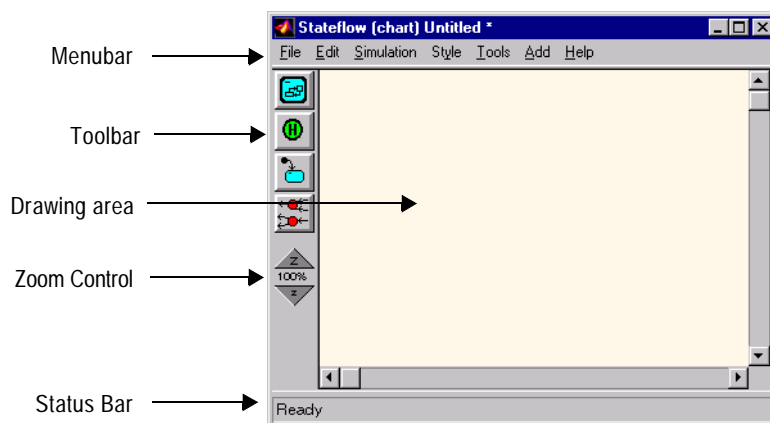
- 5 Interface the chart to other charts and blocks in your Stateflow model, using events and data.

See Chapter 3, “Defining Events and Data” and Chapter 4, “Defining Stateflow Interfaces” 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.

Using the Stateflow Editor

The Stateflow Editor consists of a window for displaying a state diagram and a set of commands that allow you to draw, zoom, modify, print, and save a state diagram displayed in the window.



The editor window includes the following elements:

- **Menubar**
Most editor commands are available from the menubar.
- **Shortcut menus**
These menus pop-up from the drawing area when you press the right mouse button. These menus display commands that apply only to the currently selected object or to the chart as a whole, if no object is selected. See “Displaying Shortcut Menus” on page 2-6 for more information.
- **Toolbar**
Displays a set of tools for creating state diagrams. See “Displaying Shortcut Menus” on page 2-6 for more information.
- **Drawing area**
Displays an editable copy of a state diagram.
- **Titlebar**
Displays the name of the state diagram being edited followed by an asterisk if the diagram needs to be saved.

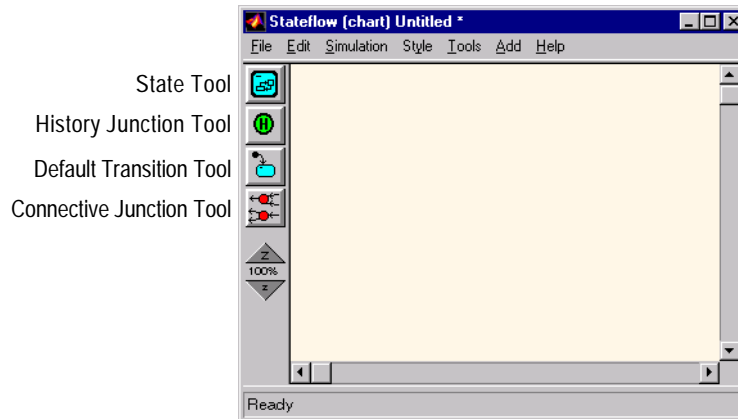
- Zoom control
See “Exploring Objects in the Editor Window” on page 2-12 for information on using the zoom control.
- Status bar
Displays tooltips and status information.

Displaying Shortcut Menus

Every object in a state diagram has a shortcut menu. To display the shortcut menu, move the cursor over the object and press the right mouse button. Stateflow then pops up a menu of operations that apply to the object. You can similarly display a shortcut menu for the chart as a whole. To display the chart shortcut menu, move the cursor to an unoccupied location in the diagram and press the right mouse button.

Drawing Objects

A state diagram comprises five types of objects: states, transitions, default transitions, history junctions, and connective junctions. Stateflow provides tools for creating instances of each of these types of objects. You must enable these tools to use them. The Transition tool is available by default. You select and deselect the other tools by clicking their icons in the Stateflow editor’s toolbar.



You use the tools by clicking and dragging the cursor in the editor's drawing area. For more information, see the following topics:

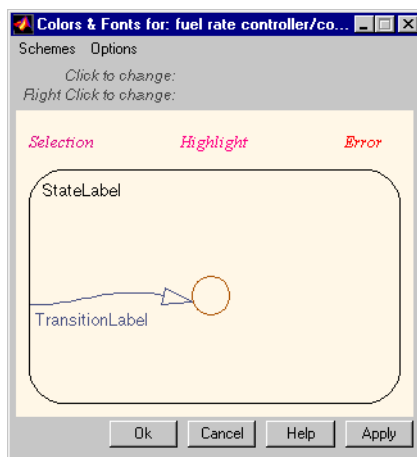
- “Creating States” on page 2-13
- “Creating Transitions” on page 2-21
- “Creating Junctions” on page 2-26

Specifying Object Styles

An object's style consists of its color and the size of its label font. The Stateflow **Colors & Fonts** dialog allows you to specify a color scheme for a chart as a whole or colors and label fonts for various types of objects in a chart. To display the dialog, select **Style...** from the Stateflow editor's **Edit** menu. Stateflow displays the **Colors & Fonts** dialog. To specify the label font size of a particular object, select the object and choose the size from the **Set Font Size** submenu of the editor's **Edit** menu.

Colors & Fonts Dialog

The Colors & Fonts Dialog allows you to specify colors and label fonts for items in a chart or for the chart as a whole.



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. Choose **Apply** to apply the selected scheme to the chart or **Ok** to apply the scheme and dismiss the dialog.

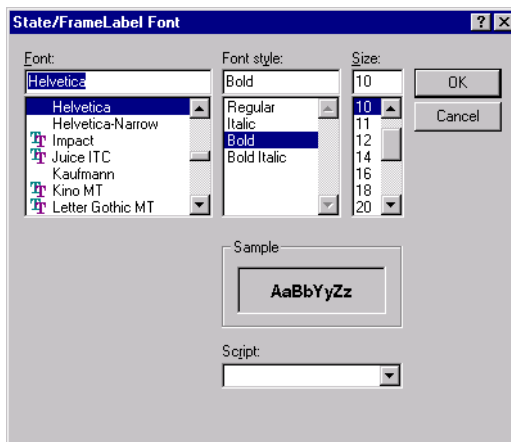
To make the selected scheme the default scheme for all Stateflow charts, select **Make this the 'Default' scheme** from the dialog's **Options** menu.

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 click the left mouse button to change the object's color or the right mouse button to change the object's font. If you click the left mouse button, 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 click the right mouse button, 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. To select an object, click on it. When an object is selected, it is highlighted in the color set as the selection color (blue by default; see “Specifying Object Styles” on page 2-7 for more information).

To select multiple objects, click the left mouse button and drag the selection rubberband so that the rubberband box encompasses or touches the objects you want to select. Once all objects are within the rubberband, release the left mouse button. All objects or portions of objects within the rubberband are selected.

Simultaneously pressing the **Shift** key and clicking on an object either adds that object to the selection list if it was deselected or deselects the object if it is on the selection list. This is useful to select objects within a state without selecting the state itself.

To select all objects in the Stateflow diagram, choose **Select All** from the **Edit** menu or the right mouse button shortcut menu.

Simultaneously, pressing the **Shift** key and doing a rubberband selection selects objects touched by the rubberband if they are deselected and deselects objects touched by the rubberband if they are selected.

Pressing the **Escape** key deselects all selected objects. Pressing the **Escape** key again brings the Simulink model (the parent of the Stateflow diagram) forward.

Cutting and Pasting Objects

You can cut one or more objects from the drawing area or cut and then paste the object(s) as many times as you like. You can cut and paste objects from one Stateflow diagram to another. The Stateflow clipboard contains the most recently cut selection list of objects. The object(s) are pasted in the drawing area location closest to the current mouse location.

To cut an object, select the object and choose **Cut** from either:

- The **Edit** menu on the main window
- The right mouse button 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 list of objects, choose **Paste** from either:

- The **Edit** menu on the main window
- The right mouse button 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 object(s), click and hold the right mouse button down, and drag 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:

- The **Edit** menu on the Stateflow graphics editor window
- Any right mouse button 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 (e.g., 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.

Changing a Label's Font Size

You can adjust font size through the shortcut menus using a virtual scrollbar. To adjust font size with the virtual scrollbar:

- 1 Select the state(s) whose label font size you want to change.
- 2 Right-click on a selected state whose font you want to change 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.

The font size of all labels on any selected states is adjusted.

Exploring Objects in the Editor Window

To view or modify events and data defined by any state visible in the Stateflow editor window (see Chapter 3, “Defining Events and Data”), position the editor cursor over the state, display the state’s context menu (by pressing the right mouse button), and select **Explore** from the context menu. Stateflow opens the Stateflow Explorer (if not already open) and expands its object hierarchy view (see “Explorer Main Window” on page 5-4) to show any events or data defined by the state.

To view events and data defined by a transition or junction’s parent state, select **Explore** from the transition or junction’s context menu.

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”).
- **Zoom In** button. Zooms in by the current zoom factor.
- **Zoom Out** button. Zooms out by the current zoom factor.

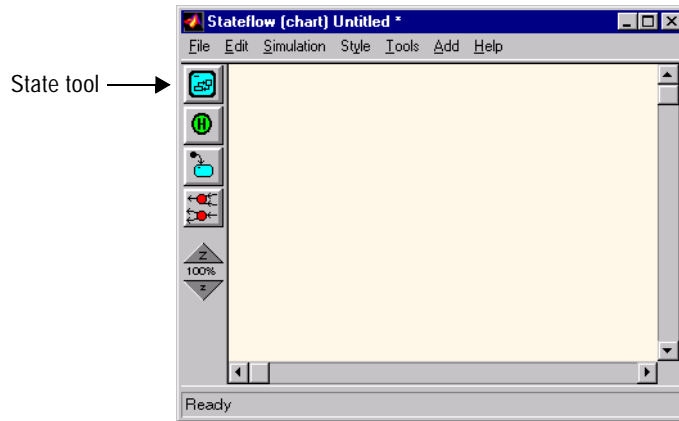
Using the Zoom Factor Selector

The **Zoom Factor Selector** allows you to specify the zoom factor by:

- Choosing a value from a menu.
Click on the selector to display the menu.
- Double-clicking on 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 any shortcut menu or by pressing the **F** key to apply the maximum zoom that includes all selected objects. Press the spacebar to fit all objects to the view.
- Clicking on 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.

Creating States

You create states by drawing them in the Stateflow editor's drawing area, using the Stateflow's State tool.



To activate the State tool, click or double-click on the **State** button in the Stateflow toolbar. Single-clicking on the button puts the State tool in single-creation mode. In this mode, you create a state by clicking the tool in the drawing area. Stateflow creates the state at the specified location and returns to edit mode.

Double-clicking on the **State** button puts the State tool in multiple-creation mode. This mode works the same way as single-creation mode except that the State tool remains active after you create a state. You can thus create as many states as you like in this mode without having to reactivate the State tool. To return to edit mode, click on the **State** button, or right click in the drawing area, or press the **Escape** key.

To delete a state, 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, select, drag, and release it in a new position. To resize a state, drag one of the state's corners. When the cursor is over a corner, it appears as a double-ended arrow (PC only; cursor appearance will vary on other platforms).

Creating Substates

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 large enough to accommodate all its substates. You may therefore need to resize a parent state before dragging a new substate into it.

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 on it or its border or select **Grouped** from the **Make Contents** submenu on the state or box' shortcut menu. Stateflow thickens the state's border and grays its contents to indicate that it is grouped. To ungroup a state, double-click it or its border or select **Ungrouped** from the **Make Contents** submenu on its shortcut menu. You need to ungroup a superstate to select objects within the superstate.

Hiding a State's Contents

To hide the contents of a superstate, move the cursor over the superstate, press the right-most mouse button to display the **Make Contents** shortcut menu and choose **Hidden**. Hiding the contents of an ungrouped state causes the state to be grouped.

Specifying State Decomposition

Stateflow allows you to specify whether activating a state activates all or only one of its substates. 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. You can alter a state's decomposition

only if it contains substates. To alter a state's decomposition, select the state, press the right mouse button 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, press the right mouse button 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.

Labeling States

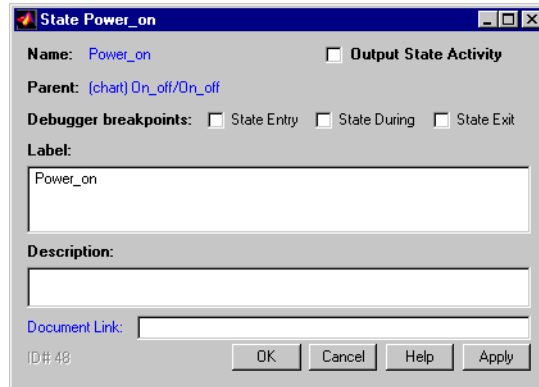
Every state has a label. A state's label specifies its name and actions that a state machine takes when entering or exiting the state or while the state is active. Initially, a state's label is empty. Stateflow indicates this by displaying a ? in the state's label position (upper left corner). Click on the label or display the state's properties dialog (see "Using the State Properties Dialog") to add to

or change its contents. For more information on labeling states, see the following topics:

- “Naming States” on page 2-17
- “Defining State Actions” on page 2-17

Using the State Properties Dialog

The **State Properties Dialog** lets you view and change a state’s properties. To display the dialog for a particular state, choose **Properties** from the state’s shortcut menu or click on the state’s entry in the Explorer content pane. Stateflow displays the **State** properties dialog box.



The dialog includes the following fields:

Name. Stateflow diagram name; read-only; click on this hypertext link to bring the state to the foreground.

Output State Activity. Check this 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 2-19 for more information.

Parent. Parent of this state; a / character indicates the Stateflow diagram is the parent; read-only; click on this hypertext link to bring the parent to the foreground.

Debugger breakpoints. Click on the check boxes to set debugging breakpoints on state entry, during, or exit actions. See Chapter 10, “Debugging” for more information.

Label. The state’s label. See the section titled “Labeling States” on page 2-15 for more information.

Description. Textual description/comment.

Document Link. Enter a web URL address or a general MATLAB command. Examples are: `www.mathworks.com`, `mailto:email_address`, `edit/spec/data/speed.txt`.

Click on the dialog **Apply** button to save the changes. Click on the **Revert** button to cancel any changes and return to the previous settings. Click on the **Close** button to save the changes and close the dialog box. Click on the **Help** button to display the Stateflow online help in an HTML browser window.

Naming States

Naming a state allows a state machine to reference the state. To name a state, 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.

Defining State Actions

Stateflow allows you to specify actions that occur when a state machine enters a state, exits a state, and while a state is active.

Defining Entry Actions

An entry action is an action executed by a state machine when it enters a particular state as the result of taking a transition to that state. To specify the entry action to be taken for a given state, add an entry block to the state’s label. An entry block begins on a new line and consists of the entry action keyword, `entry` or `en`, followed by a colon, followed by one or more action statements on one or more lines. You must separate statements on the same line by a comma or semicolon. See “Action Language” on page 6-41 for information on writing action statements.

Note You can also begin a state’s entry action 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.

Defining Exit Actions

An exit action is an action executed by a state machine when it exits a state as the result of taking a transition away from the state or the occurrence of an event (see “Defining On-Event Actions” below). To specify an exit action for a state, add an exit block to the state’s label. The format of an exit block is the same as that of an entry block except that the exit block begins with the keyword `exit` or `ex`.

Defining During Actions

A during action is an action that a state machine executes while a state is active, that is, after the state machine has entered the state and while there is no valid transition away from the state. To specify a during action, add a during block to the state’s label. A during block has the same format as an entry block except that it begins with the keyword `during` or `dur`.

Defining On-Event Actions

An on-event action is an action that a state machine takes when a state is active and one or more events of a specific type occur. (See “Defining Events” on page 3-2 for information on defining and using events to drive a state machine.) To specify an event handler for a state, add an on-event block to the state. An on-event block has the same format as an entry action block except that it begins with the keyword, `on`, followed by the name of the event, followed by a colon, for example

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

A state machine can respond to multiple events, with either the same or different actions, when a state is active. If you want more than one type of event to trigger the same action, specify the keyword as `on events`, where `events` is a comma-separated list of the events that trigger the actions, for example,

```
on ev1, ev2: exit();
```

If you want different events to trigger different actions, enter multiple event 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();  
on ev3, ev4: exit();
```

Note Use a during block to specify actions that you want a state machine to take in response to any visible event that occurs while the machine is in a particular state (see “Defining During Actions” on page 2-18).

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 “Naming States” on page 2-17), if unnamed, then check the Output State Activity checkbox on the state's property dialog (see “Using the State Properties Dialog” on page 2-16). 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 “Defining Output Data” on page 3-22 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 Boxes

Stateflow allows you to use graphical entities called boxes to organize your diagram visually. To create a box, first create a superstate containing the objects to be boxed. Then, select **Box** from the superstate's **Type** shortcut menu. Stateflow converts the superstate to a box, redrawing its border with sharp corners to indicate its changed status.

Boxes are primarily graphical entities. They do not correspond to any real element of a state machine. However, boxes do affect the activation order of a diagram's parallel states. In particular, a box wakes up before any parallel states or boxes that are lower down or to the right of the box in the diagram. Within a box, parallel states still wake up in top down, right-to-left order.

You can group and ungroup boxes and hide or show them in the same way you hide or show states. See "Grouping States" on page 2-14 and "Hiding a State's Contents" on page 2-14 for more information.

Creating Transitions

Place the cursor at a straight portion of the border of the source state. Click the border when the cursor changes to a cross-hair. Drag and release on a straight portion of the border of the destination state when the transition changes from a dotted line to a solid line. The solid line indicates the transition is in position to be attached. The cross-hair will not appear if you place the cursor on a corner, since corners are used for resizing.

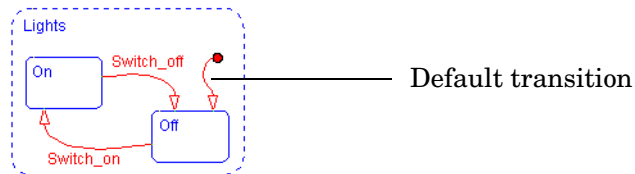
Use the same procedure to create transitions to and from junctions. The only difference is that transitions can be connected to any point on the junction. (Junctions are resized through the **Junction** shortcut menu.)

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

What Is a Default Transition?


The default transition object is a transition with a destination, but no source object. 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. Default transitions also specify that a junction should be entered by default.

In the `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.



Default transitions specify which exclusive (OR) substate in a superstate the system enters by default, in the absence of any information. History junctions override default transition paths in superstates with exclusive (OR) decomposition.

Creating Default Transitions

Click on the **Default transition** button in the toolbar , and click on 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 end point of the default transition is proportional to the arrowhead size. Default transitions can be labeled just like other transitions. See the section titled “Labeling of Default Transitions” on page 6-23 for an example.

Editing Transition Attach Points

Place the cursor over an attach point until it changes to a small circle. Click and drag the mouse to move the attach point; release to drop the attach point. You can edit both the source and destination attach points of a transition.

The appearance of the transition changes from a solid to a dashed line when editing 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 editing a source attach point.

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

Labeling Transitions

The ? character is the default empty label for transitions. Transitions and default transitions follow the same labeling format. Select the transition to display the ? character. Click on the ? to edit the label.

Transition labels have this general format:

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

Specify, as appropriate, relevant names for event, condition, condition_action, and transition_action. Transitions do not have to have labels. You can specify some, all, or none of the parts of the label.

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	After a valid destination is found and the transition is taken, this action executes and completes.

To apply and exit the edit, deselect the object.

See these sections in Chapter 6, “Notations” for more information:

- “Transitions” on page 6-16
- “Action Language” on page 6-41

Valid Labels

Labels can consist of any alphanumeric and special character combination, with the exception of embedded spaces. Labels cannot begin with a numeric character. The length of a label is not restricted. Carriage returns are allowed in most cases. Within a condition, you must specify an ellipsis (...) to continue on the next line.

Changing 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 transition(s) whose arrowhead size you want to change.
- 2 Place the cursor over a selected transition, click the right mouse button to display the shortcut menu.

A menu of arrowhead sizes appears.

- 3 Select an arrowhead size from the menu.

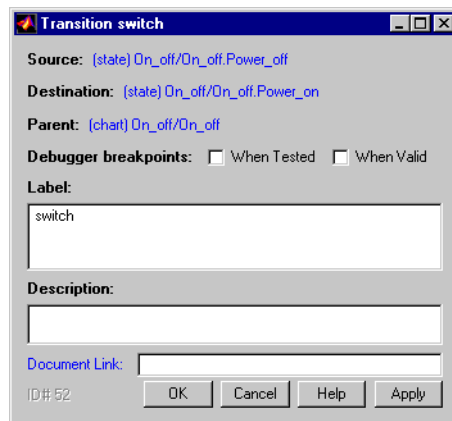
Moving Transition Labels

You can move transition labels to make the Stateflow diagram more readable. To move a transition label, click on and drag the label to the new location and then release the mouse button.

If you mistakenly click and release the left mouse button on the label, you will be in edit mode. Press the **Escape** key to deselect the label and try again.

Using the Transition Properties Dialog

Select a transition and click the right mouse button on that transition's border to display the **Transition** shortcut menu. Choose **Properties** to display the **Transition properties** dialog box.



This table lists and describes the transition object fields.



Field	Description
Source	Source of the transition; read-only; click on the hypertext link to bring the transition source to the foreground.
Destination	Destination of the transition; read-only; click on the hypertext link to bring the transition destination to the foreground.
Parent	Parent of this state; read-only; click on the hypertext link to bring the parent to the foreground.
Debugger breakpoints	Click on the checkboxes to set debugging breakpoints either when the transition is tested for validity or when it is valid.
Label	The transition's label. See the section titled "Labeling a Transition" on page 6-17 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: www.mathworks.com, mailto:email_address, edit/spec/data/speed.txt.

Click on the **Apply** button to save the changes. Click on the **OK** button to save the changes and close the dialog box. Click on the **Cancel** button to close the dialog without applying any changes made since the last time you clicked the **Apply** button. Click on the **Help** button to display the Stateflow online help in an HTML browser window.

Creating Junctions

To create one junction at a time, click on a **Junction** button in the toolbar and click on the desired location for the junction in the drawing area. To create multiple junctions, double-click on the **Junction** button in the toolbar. The button is now in multiple object mode. Click anywhere in the drawing area to place a junction in the drawing area. Additional clicks in the drawing area create additional junctions. Click on the **Junction** button or press the **Escape** key to cancel the operation.

You can choose from these types of junctions.

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.
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.

Changing Size

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

- 1 Select the junction(s) whose size you want to change. The size of any selected junctions is adjusted.
- 2 Place the cursor over a selected junction, click the right mouse button to display the shortcut menu and place the cursor over **Junction Size**.
A menu of junction sizes appears.
- 3 Select a junction size from the menu.

Changing Arrowhead Size

The arrowhead size is a property of the destination object. Changing one of the incoming arrowheads of a junction causes all incoming arrowheads to that junction to be adjusted to the same size. The arrowhead size of any selected junctions is adjusted.

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

- 1 Select the junction(s) whose incoming arrowhead size you want to change.
- 2 Place the cursor over a selected junction, click the right mouse button to display the shortcut menu. Place the cursor over **Arrowhead Size**.

A menu of arrowhead sizes appears

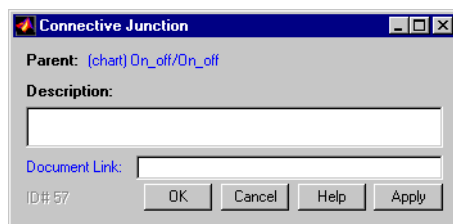
- 3 Select a size from the menu.

Moving a Junction

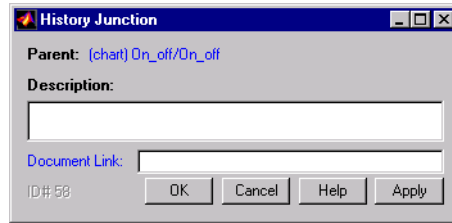
To move a junction, select, drag, and release it in a new position.

Editing Junction Properties

Select a junction, click the right mouse button on that junction to display the **Junction** shortcut menu. Choose **Properties** to display the **Connective Junction properties** dialog box.



This is the history junction properties dialog box.



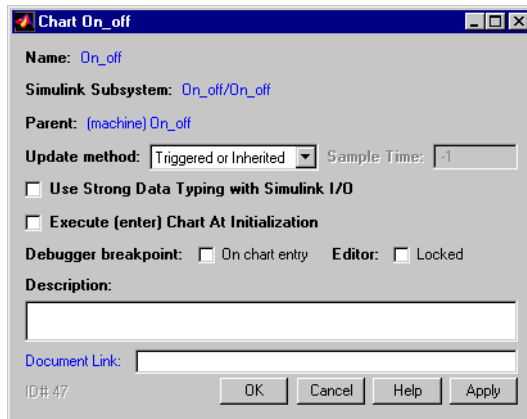
This table describes the junction object fields.

Field	Description
Parent	Parent of this state; read-only; click on the hypertext link to bring the parent to the foreground.
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> , <code>edit/spec/data/speed.txt</code> .

Click on the **Apply** button to save the changes. Click on the **Cancel** button to cancel any changes since the last apply. Click on the **OK** button to save the changes and close the dialog box. Click on the **Help** button to display the Stateflow online help in an HTML browser window.

Specifying Chart Properties

Click the right mouse button in an open area of the Stateflow diagram to display the **General** shortcut menu. This is the **Chart properties** dialog box.



This table lists and describes the chart object fields.

Field	Description
Name	Stateflow diagram name; read-only; click on this hypertext link to bring the chart to the foreground.
Simulink Subsystem	Simulink subsystem name; read-only; click on this hypertext link to bring the Simulink subsystem to the foreground.
Parent	Parent name; read-only; click on this hypertext link to display the parent's property dialog box.
Update method	Choose from Triggered or Inherited , Sampled , or Continuous .
Sample Time	If Update method is Sampled , enter a sample time.
Debugger breakpoint	Click on the check box to set a debugging breakpoint On chart entry .
Editor	Click on the Locked check box to mark the Stateflow diagram as read-only and prohibit any write operations.
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> , <code>edit/spec/data/speed.txt</code> .

Click on the **Apply** button to save the changes. Click on the **Cancel** button to cancel any changes since the last apply. Click on the **OK** button to save the changes and close the dialog box. Click on the **Help** button to display the Stateflow online help in an HTML browser window.

Waking Up Charts

Stateflow lets you specify the method by which a simulation updates (wakes up) a chart. To specify a wake up method for a chart, set the chart's Update method property (see "Specifying Chart Properties" on page 2-29) to one of the following options:

- Triggered or Inherited

This is the default update method. Specifying this method causes inputs from the Simulink model to determine when the chart wakes up during a simulation. If you define input events for the chart (see "Defining Input Events" on page 3-7), the chart awakens when trigger signals appear on the chart's trigger port. If you define data inputs (see "Defining Input Data" on page 3-21) but no event inputs, the chart awakens at the rate of the fastest data input. If you do not define any inputs for the chart, the chart wakes up at the model's solver sample rate.

- Sampled

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 may have different sample times.

- Continuous

The Stateflow block wakes up at each step in the simulation, as well as at intermediate time points that may be requested by the Simulink solver.

See "Defining the Interface to External Sources" on page 4-25 and *Using Simulink* for more information.

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 state machine are invalid. Stateflow allows you to define such events but flags them as errors when parsing a model.

Stateflow Printing Options

The following options are available for printing Stateflow models

- You can print a block diagram of the Stateflow model, using the Simulink **Print** command.
See the *Using Simulink* manual for more information.
- You can print a model's state diagram, using the Stateflow **Print** command.
See “Printing a Stateflow Diagram” below.
- You can generate a report that documents the Stateflow component of a Stateflow model, using the Stateflow **Print Book** command.
See “Printing a Stateflow Book” below.
- You can generate a report that documents an entire Stateflow model, including both Simulink and Stateflow components, using the Simulink Report Generator.

The Simulink Report Generator is available as a separate product. See the *Report Generator User's Guide* for more information.

Printing a Stateflow Diagram

To print a Stateflow diagram, open the chart containing the diagram and select **Current View** or **Entire Chart** from the **Print...** submenu of the Stateflow editor's **File** menu or enter `sfprint` at the MATLAB command prompt.

See `sfprint` in Chapter 11, “Reference” for more information about printing from the command line.

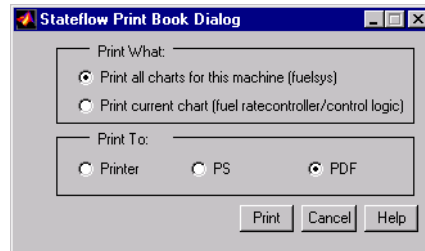
Printing a Stateflow Book

A Stateflow book is a report that 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 Stateflow book:

- 1 Select and open one of the charts you want to document.
- 2 Select **Print Book** from the Stateflow editor's **File** menu.

Stateflow displays the **Print Book** dialog.



- 3 Check the desired print options on the dialog.
- 4 Select the **Print** button to generate the report.

Defining Events and Data

Defining Events	3-2
Adding Events to the Data Dictionary	3-2
Setting Event Properties	3-4
Event Properties	3-5
Naming Events	3-7
Defining Local Events	3-7
Defining Input Events	3-7
Defining Output Events	3-8
Exporting Events	3-9
Importing Events	3-9
Specifying Trigger Types	3-10
Describing Events	3-11
Documenting Events	3-11
Implicit Events	3-11
Defining Data	3-14
Adding Data to the Data Dictionary	3-14
Setting Data Properties	3-16
Data Properties	3-18
Defining Data Arrays	3-20
Defining Input Data	3-21
Defining Output Data	3-22
Associating Ports with Data	3-23
Defining Temporary Data	3-23
Exporting Data	3-24
Importing Data	3-25
Documenting Data	3-25

Defining Events

An event is a Stateflow object that triggers actions in a state machine or its environment. Stateflow defines a set of events that typically occur whenever a state machine executes (see “Implicit Events” on page 3-11). You can define other types of events that occur only during execution of a specific state machine or its environment.

To define an event:

- 1 Add a default definition of the event to the Stateflow data dictionary (see “Adding Events to the Data Dictionary”).
- 2 Set the new event’s properties to values that reflect its intended usage (see “Setting Event Properties” on page 3-4).

Adding Events to the Data Dictionary

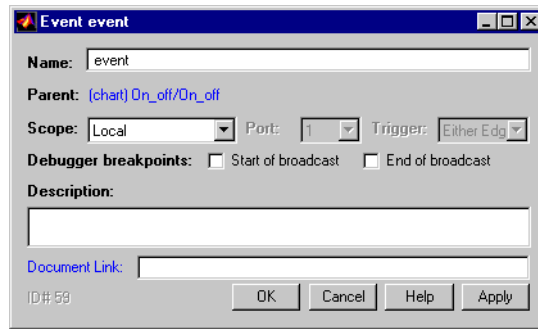
You can use either the Stateflow editor or Explorer to add events that are visible everywhere in a chart. You must use the Stateflow Explorer to add events that are visible everywhere in a state machine or only in a particular state.

Using the Stateflow Editor

To use the Stateflow editor to add an event:

- 1 Select the event's scope (see “Event Properties” on page 3-5) from the **Event** submenu of the Stateflow editor's **Add** menu.

Stateflow adds a default definition of the new event to the Stateflow data dictionary and displays the **Event** dialog.



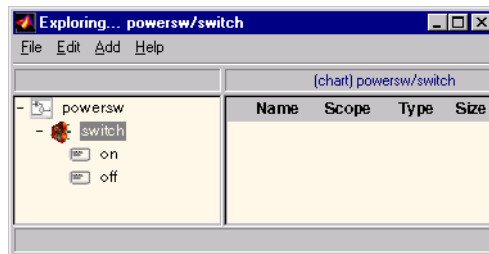
- 2 Use the **Event** dialog to set the new event's properties to reflect its intended usage (see “Setting Event Properties” on page 3-4).

Using the Explorer

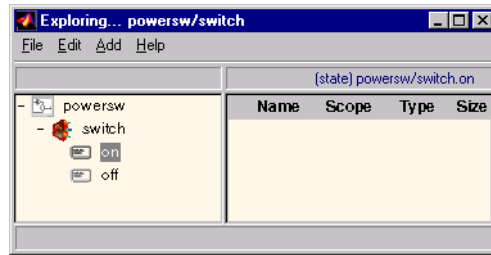
To use the Stateflow Explorer to define an event:

- 1 Select **Explore** from the Stateflow editor's **Tools** menu.

Stateflow opens the Explorer.

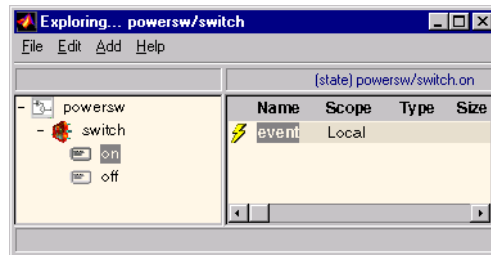


- 2 Select the object (state machine, chart, or state) in the Explorer's object hierarchy pane where you want the new event to be visible.



- 3 Select **Event** from the Explorer's **Add** menu.

Stateflow adds a default definition for the new event in the data dictionary and displays an entry for the new event in the Explorer's content pane.



- 4 Set the new event's properties to values that reflect its intended usage (see "Setting Event Properties").

Setting Event Properties

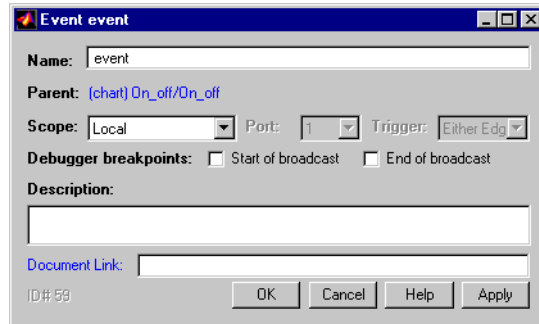
An event is defined by its properties. You can redefine an event by resetting its properties.

To set an event's properties:

- 1 Select Explorer from the Stateflow editor's **Tools** menu.
- 2 Select the event in the Explorer's object hierarchy pane.

-
- 3 Double-click on the event's entry in the Explorer's contents pane.

Stateflow displays the **Event** dialog for the selected event.



- 4 Edit the properties displayed in the dialog to suit the event's intended usage.

See “Event Properties” on page 3-5 for a description of the various properties that events can have and information on how to set the properties.

- 5 Select **OK** to apply your changes and dismiss the **Event** dialog.

Note You can also set an event's Scope (see “Defining Local Events” on page 3-7) and Trigger properties by editing the corresponding fields in the event's entry in the Explorer's contents pane. If you want to set only these properties, you do not need to open the **Event** dialog for the event.

Event Properties

Stateflow events are defined by their properties. Properties that events can have include:

Event Parent

A property's parent is the object in which it is visible. When an event is triggered, Stateflow broadcasts the event to the parent and all the parent's descendants. An event's parent can be a state machine, a chart, or a state. You

specify an event's parent when you add it to the data dictionary (see “Adding Events to the Data Dictionary” on page 3-2).

Event Name

Event names enable Stateflow actions to identify events unambiguously. See “Naming Events” on page 3-7 for more information.

Event Scope

An event's scope specifies where an event originates relative to its parent. You can specify the following event scopes:

Local. A local event is an event that originates in a state machine and is parented by the state machine or one of its charts or states. See “Defining Local Events” on page 3-7 for more information.

Input from Simulink. This type of event originates in one Simulink block and is broadcast in another. The first block may be any type of Simulink block. The second block must be a chart block. See “Defining Input Events” on page 3-7 for more information.

Output to Simulink. This type of event originates in one Simulink block and is broadcast in another. The first block is a chart block. The second block may be any type of Simulink block. See “Defining Output Events” on page 3-8 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” on page 3-9 for more information.

Imported. An imported event is an externally defined event that can be broadcast by a state machine embedded in the external code. See “Importing Events” on page 3-9 for more information.

Event Trigger

Specifies the type of signal that triggers an input or output event. See “Specifying Trigger Types” on page 3-10 for more information.

Event Index

This property applies only to input events. It associates an input event with a control signal that can be used to trigger it. See “Associating Input Events with Control Signals” on page 3-8 for more information.

Event Port

This property applies only to output events. It associates an output event with a port that outputs a control signal used to trigger the event. See “Associating an Output Event with an Output Port” on page 3-8 for more information.

Event Description

This property enables you to store brief descriptions of events in the data dictionary. See “Describing Events” on page 3-11 for more information.

Event Documentation

This property allows a model to incorporate links to online documentation for user-defined events. See “Documenting Events” on page 3-11 for more information.

Naming Events

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.

Defining Local Events

A local event is an event that can occur anywhere in a state machine but is visible only in its parent (and its parent’s descendants). To define an event as local, set its Scope property to Local.

Defining Input Events

An input event occurs outside of 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.

You can define multiple input events for a chart. The first time you define an input event for a chart, Stateflow adds a trigger port to the chart's block. External blocks can trigger the chart's input events via a signal or vector of signals connected to the chart's trigger port. See "Associating Input Events with Control Signals" on page 3-8. When defining input events for a chart, you must specify how control signals connected to the chart trigger the input events (see "Specifying Trigger Types" on page 3-10).

Associating Input Events with Control Signals

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 "Waking Up Charts" on page 2-31). 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 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 "Event Port" on page 3-7). Your model can use the output ports to trigger the output events in other Simulink blocks in the same model.

Associating an Output Event with an Output Port

An output event's Port property associates the event with an output port on the chart block that parents the event. The property specifies the position of the

port relative to other event ports on the chart block. Event ports appear below data ports on the right side of a chart block. Stateflow numbers ports sequentially from top to bottom, starting with port 1. Stateflow assigns port 1 to the first output event that you define for a chart, port 2 to the second output event, and so on. You can change the default port assignment of an event by resetting its Port property.

Exporting Events

Stateflow allows a state machine to export events. Exporting events enables external code to trigger events in the state machine. To export an event, first add the event to the data dictionary as a child of the state machine (see “Adding Events to the Data Dictionary” on page 3-2). Then set the new event’s Scope property to Exported.

Note External events can be parented only by a state machine. This means that you must use the Explorer to add external events to the data dictionary. It also means that external events are visible everywhere in a state machine.

When encoding a state machine that parents exported events, 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 the target containing the state machine 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 4-25 for an example of how to trigger an exported event.

Importing Events

A state machine can import events defined by external code. Importing an event allows a state machine built into a stand-alone or Real-Time Workshop target to trigger the event in external code. To import an event, first add the event to the data dictionary as a child of the state machine that needs to trigger

the event (see “Adding Events to the Data Dictionary” on page 3-2). Then set the new event’s Scope property to Imported.

Note The state machine serves as a surrogate parent for imported events. This means that you must use the Explorer to add imported events to the data dictionary.

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 a state machine imports an external event named `switch_on`. Then Stateflow assumes that external code defines a function named `external_broadcast_switch_on` that broadcasts the event to external code. When encoding the state 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.

Specifying Trigger Types

A trigger type defines how control signals trigger input and output events associated with a chart. Trigger types fall into two categories: function call and edge. The basic difference between these two types is when receiving blocks are notified of their occurrence. Receiving blocks are notified of edge-triggered events only at the beginning of the next simulation time step, regardless of when the events occurred during the previous time step. By contrast, receiving blocks are notified of function-call-triggered events the moment the events occur, even if they occur in mid-step.

You set a chart’s trigger type by setting the Trigger property of any of the input or output events defined for the chart. If you want a chart to notify other blocks the moment an output event occurs, set the **Trigger** property of the output event to **Function Call**. The output event’s trigger type must be **Either Edge**. If a chart is connected to a block that outputs function-call events, you must specify the **Trigger** property of the receiving chart’s input events to **Function Call**, Stateflow changes all of the chart’s other input events to **Function Call**.

If it is not critical that blocks be notified of events the moment they occur, you can define the events as edge-triggered. You can specify any of the falling types of edge triggers:

Rising Edge. A rising level on the control signal triggers the corresponding event.

Falling Edge. A falling level on the control signal triggers the event.

Either Edge. A change in the signal level triggers the event.

In all cases, the signal must cross 0 to constitute a valid trigger. For example, a change from -1 to 1 constitutes a valid rising edge, but not a change from 1 to 2.

If you specify an edge trigger type that differs from the edge type previously defined for a chart, Stateflow changes the **Trigger** type of the chart's input events to **Either Edge**.

Describing Events

Stateflow allows you to store brief descriptions of events in the data dictionary. To describe a particular event, set its Description property to the description.

Documenting Events

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 on the event's documentation link (the blue text that reads "Documentation" displayed at the bottom of the event's **Event** dialog box).

Implicit Events

Stateflow defines and triggers the following events that typically occur whenever a chart executes:

- Entry into a state
- Exit from a state
- Value assigned to an internal (non-input) 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.

Referencing Implicit Events

Action expressions can use the following syntax to reference implicit events.

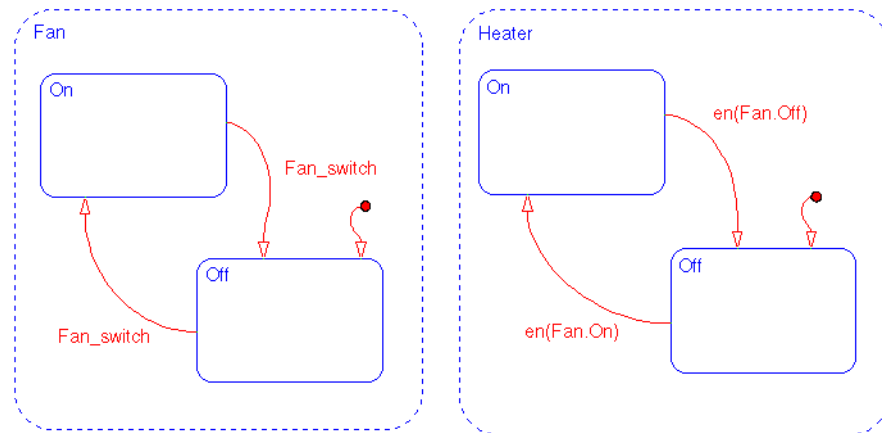
```
event(object)
```

where `event` is the name of the implicit event and `object` is the state or datum in which the event occurred. Valid implicit event names (and their shortcuts) are `enter` (`en`), `exit` (`ex`), and `change` (`chg`). 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)
```

Example

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.

Defining Data

A state machine 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 state machine. When creating a Stateflow model, you must define any internal or external data referenced by the state machine's actions.

To define an item of data:

- 1 Add the item to the data dictionary (see “Adding Data to the Data Dictionary”).
- 2 Set the new item's properties (see “Setting Data Properties” on page 3-16)

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 state machine or only in a specific state.

Using the Stateflow Editor

To use the Stateflow editor to add data:

- 1 Select the data's scope (see "Data Properties" on page 3-18) from the **Data** submenu of the Stateflow editor's **Add** menu.

Stateflow adds a default definition of the new item to the Stateflow data dictionary and displays a **Data** dialog that displays the new item's default properties.



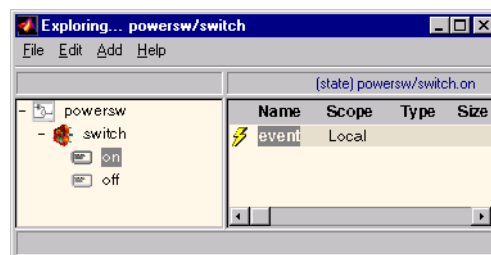
- 2 Use the **Data** dialog to set the new item's properties to reflect its intended usage (see "Setting Data Properties" on page 3-16).

Using the Explorer

To use the Stateflow Explorer to define a data item:

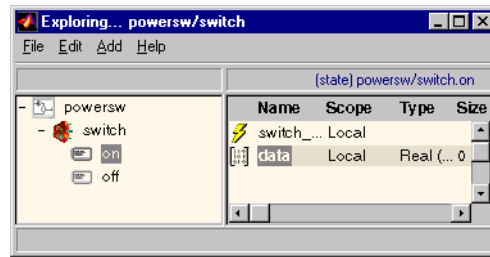
- 1 Select **Explore** from the Stateflow editor's **Tools** menu.

Stateflow opens the Explorer.



- 2 Select the object (state machine, chart, or state) in the Explorer's object hierarchy pane where you want the new item to be accessible.
- 3 Select **Data** from the Explorer's **Add** menu.

Stateflow adds a default definition for the new item in the data dictionary and displays an entry for the item in the Explorer's content pane.



- 4 Set the new item's properties to values that reflect its intended usage (see "Setting Event Properties").

Setting Data Properties

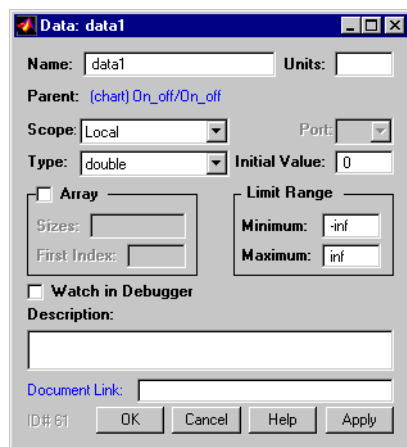
You define a data item by setting its properties.

To set a data item's properties:

- 1 Select Explorer from the Stateflow editor's **Tools** menu.
- 2 Select the item in the Explorer's object hierarchy pane.

- 3 Double-click on the item's entry in the Explorer's contents pane.

Stateflow displays the **Data** dialog for the selected item.



- 4 Edit the properties displayed in the dialog to suit the item's intended usage.

See "Data Properties" on page 3-18 for a description of the various properties that data items can have and information on how to set them.

- 5 Select **OK** to apply your changes and dismiss the **Data** dialog.

Note You can also set a data item's Scope, Type, Sizes, Initial Value, Minimum, and Maximum properties (see "Data Properties") by editing the corresponding fields in the item's entry in the Explorer's contents pane. If you want to set only these properties, you do not need to open the **Data** dialog for the event.

Data Properties

Items of data can have the following properties:

Data Name

A data name can consist of any alphanumeric and special character combination, with the exception of embedded spaces. The name cannot begin with a numeric character. The length of a name is not restricted.

Data Parent

The parent of a data item determines the objects that can access it. Specifically, only the item's parent and descendants of that parent can access the item. You specify the parent of a datum when you add it to the data dictionary.

Data Scope

A data object's scope specifies where it resides in memory relative to its parent. These are the options for the **Scope** property:

Local. A local data object resides and is accessible only in a machine, chart, or state.

Input from Simulink. This is a data item that is accessible in a Simulink chart block but resides in another Simulink block that may or may not be a chart block. The receiving chart block reads the value of the data item from an input port associated with the data item. See "Importing Data" on page 3-25 for more information.

Output to Simulink. This is a data item that resides in a chart block and is accessible in another block that may or may not be a chart block. The chart block outputs the value of the datum to an output port associated with the data item. See "Defining Output Data" on page 3-22 for more information.

Temporary. A temporary data item exists only while its parent is executing. See "Defining Temporary Data" on page 3-23 for more information.

Workspace. A Workspace data object is a value that typically does not change during a simulation. Workspace data objects are passed in from MATLAB and/or Simulink. Workspace data objects can be defined at the MATLAB command line or through Simulink mask initialization. Stateflow will attempt

to resolve a data name first locally, then in Simulink, and lastly in MATLAB. See the *Using Simulink* document for more information on mask initialization.

Constant. A Constant data object is read-only and retains the initial value set in its data properties dialog box.

Exported. An exported data item is state machine data that can be accessed by external code that embeds the state machine. See “Exporting Data” on page 3-24 for more information.

Imported. Imported data is data defined by external code that can be accessed by a state machine embedded in the external code. See “Importing Data” on page 3-25 for more information.

Data Port

This property applies only to input and output data. It specifies the index of the port associated with the data item (see “Associating Ports with Data” on page 3-23).

Data Type

This property specifies the type of the data, e.g., integer, double, etc.

Data Sizes

This property applies to arrays of data (see “Data Initial Value” on page 3-19). The value of this property may be a scalar or a MATLAB vector. If it is a scalar, it specifies the size of a one-dimensional array (i.e., a vector). If a MATLAB vector, it indicates the size of each dimension of a multidimensional array whose number of dimensions corresponds to the length of the vector.

Data First Index

This property applies only to arrays of data. It specifies the index of the first element of this array. For example, the first index of a zero-based array is 0.

Data Initial Value

This property specifies a data item’s value at the start of a state machine’s execution. If the item is an array, Simulink sets each element of the array to the specified initial value.

Data Minimum

The minimum value that a data item can have. A state machine uses this property to check that an item is in range.

Data Maximum

The maximum value that a data item can have. A state machine uses this property to check that an item is in range.

Data Description

The description can consist of any ASCII text you want to enter to describe the object.

Data Documentation

This property allows a model to incorporate links to online documentation for user-defined data. See “Documenting Data” on page 3-25 for more information.

Defining Data Arrays

Stateflow allows you to define arrays of data.

To define an array:

- 1 Add a default data item to the data dictionary as a child of the state, chart, or machine that needs to access the data (see “Adding Data to the Data Dictionary” on page 3-14).
- 2 Open the **Data** properties dialog. Check the **Array** checkbox on the dialog. Set the item’s Sizes property to the size of each of the array’s dimensions (“Setting Data Properties” on page 3-16).

For example, to define a 100-element vector, set the Sizes property to 100. To define a 2-by-4 array, set the sizes property to [2 4].

- 3 Set the item’s Initial Index property to the index of the array’s first element.

For example, to define a zero-based array, set the Initial Index property to 0.

- 4 Set the item's Initial Value property to an appropriate value.

For example, to specify that an array's elements be initialized to zero, set the Initial Value property to 0.

- 5 Set the item's other properties (e.g., Name, Type, and so on) to reflect its intended usage.

Example

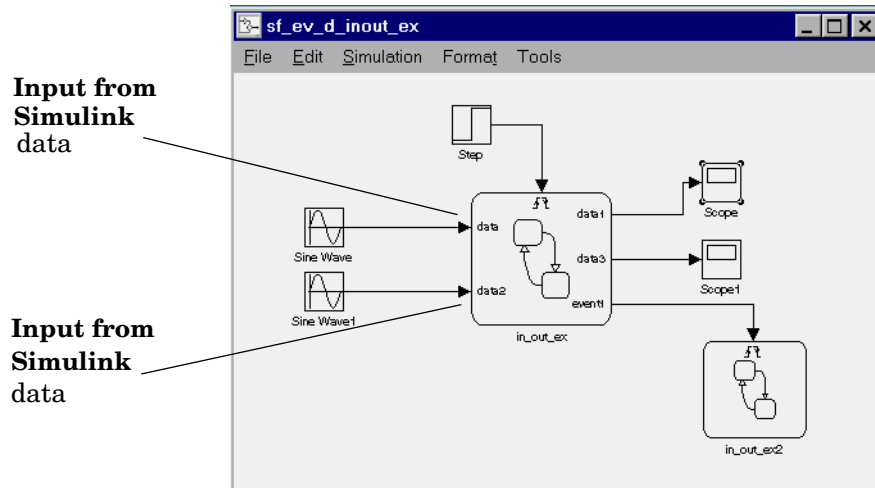
Suppose that you want to define a local, 4-by-4, zero-based array of type Integer named `rotary_switches`. Further, suppose that each element of the array was initially 1 and could have no values less than 1 or greater than 10. The following table shows the data property settings for such an array.

Property	Value
Name	<code>rotary_switches</code>
Scope	Local
Type	Integer
Sizes	[4 4]
First Index	0
Initial Value	1
Minimum	1
Maximum	10

Defining Input Data

Stateflow allows a model to supply data to a chart via input ports on the chart's block. Such data is called input data. To define an item of input data, add a default item to the Stateflow data dictionary as a child of the chart that will input the data. Set the new item's Scope to **Input from Simulink**. Stateflow

adds an input port to a chart for each item of input data that you define for the chart.

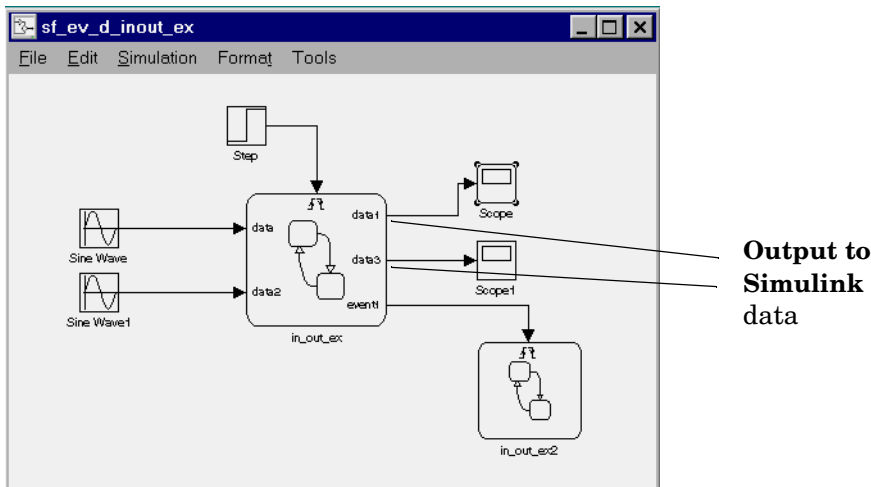


Set the item's other properties (e.g., Name, Type, etc.) to appropriate values. Simulink blocks can input only scalar or vector data of type Double. You can set an input item's data type to any Stateflow-supported type. If necessary, Stateflow converts the input value to the specified type. If the input item is a vector, the model must supply the data via a signal vector connected to the corresponding input port on the chart.

Defining Output Data

Output data is data that a chart supplies to other blocks via its output ports. To define an item of output data, add a default data item to the data dictionary as a child of the chart that supplies the item. Then, set the new item's Scope

property to **Output to Simulink**. Stateflow adds an output port to the chart for each item that it outputs.



Stateflow blocks can output only scalar or vector data of type Double. You can set an output item's type to any supported Stateflow data type (for example, Integer). However, Stateflow will convert the item to Double on output.

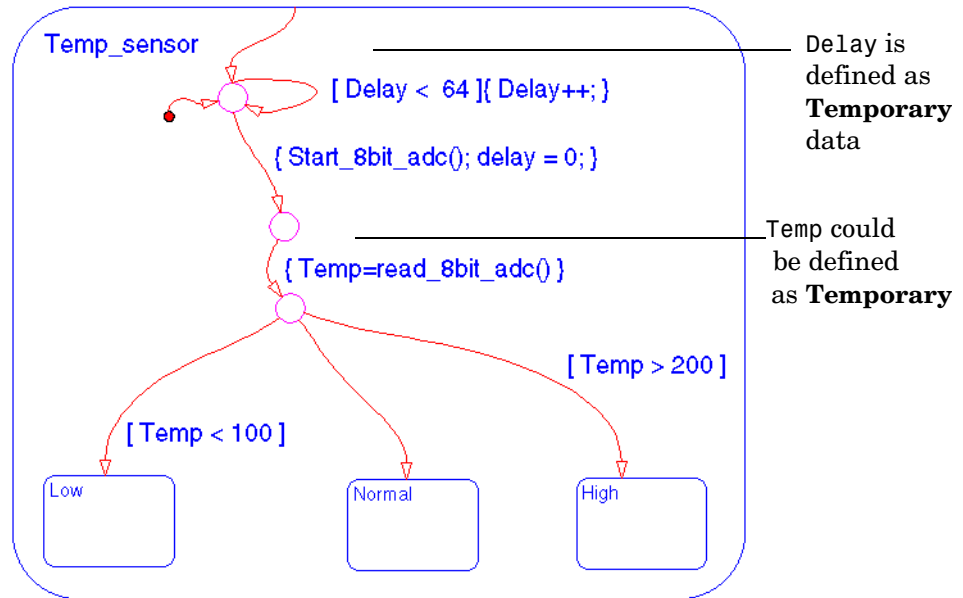
Associating Ports with Data

Stateflow creates and associates an input port with each input data item that you define for a chart and an output port for each output data item. By default, Stateflow associates the first input port with the first input item you define, the first output port with the first output item, the second input port with the second input item, and so on. The **Data** dialog for each item shows its current port assignment in the Port field. You can alter the assignment by editing the value displayed in the Port field.

Defining Temporary Data

Temporary data is non-persistent. **Temporary** data is accessible only by its parent. Defining a loop counter to be **Temporary** is a good use of this **Scope** since the value is used only as a counter and the value does not need to persist. The **Temporary** data remains defined and accessible only while the parent is executing. For a state, the **Temporary** data is defined when that state is active and processing state actions. For a chart, the **Temporary** data is defined and

accessible when the chart is active and processing condition and/or transition actions. This is an example Stateflow diagram of an Analog to Digital converter that uses **Temporary** data.



Delay is used only as a counter in the loop. Once the delay loop is complete, the value of Delay is no longer needed. If you were interested only in whether the Low, Normal, or High state is active, the value of Temp could be defined as **Temporary** as well.

Exporting Data

Stateflow can export definitions of state machine data to external code that embeds the state machine. Exporting data enables external code, as well as the state machine, to access the data. To export a data item, first add it to the data dictionary as the child of the state machine in which it is defined. Then set its Scope property to Exported and its other properties (e.g., Name and Type) to appropriate values.

The Stateflow code generator generates a C declaration for each exported data item of the form

```
type ext_data;
```

where `type` is the C type of the exported item (e.g., `int`, `double`) and `data` is the item's Stateflow name. For example, suppose that your state machine defines an exported integer item named `counter`. The Stateflow code generator exports the item as the C declaration

```
int ext_counter;
```

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.

Importing Data

A state machine can import definitions of data defined by external code that embeds the state machine. Importing externally defined data enables a state machine to access data defined by the system in which it is embedded. To import an externally defined data item into a state machine, add a default item to the data dictionary as a child of the state machine. Then set the new item's Scope property to **Imported**, its Name property to the name used by the machine's actions to reference the item, and its other properties (i.e., Type, Initial Value, etc.) to appropriate values.

The Stateflow code generator assumes that external code provides a prototype for each imported item of the form

```
type ext_data;
```

where `type` is the C data type corresponding to the Stateflow data type of the imported item (e.g., `int` for `Integer`, `double` for `Double`, etc.) and `data` is the item's Stateflow name. For example, suppose that your state machine defines an imported integer item named `counter`. The Stateflow code generator expects the item to be defined in the external C code as

```
int ext_counter;
```

Documenting Data

Stateflow allows you to provide online documentation for data defined by a model. To document a particular item of data, 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 on the item's documentation link (the blue text that reads Documentation displayed at the bottom of the event's **Data** dialog box).

Defining Stateflow Interfaces

Overview	4-2
Defining the Stateflow Block Update Method	4-4
Defining Output to Simulink Event Triggers	4-10
Event Inputs from Simulink	4-16
Data Inputs from Simulink	4-18
Event Outputs to Simulink	4-20
Data Outputs to Simulink	4-22
MATLAB Workspace	4-24
Defining the Interface to External Sources	4-25

Overview

Interfaces to Stateflow

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 block interface includes:

- Physical connections between Simulink blocks and the Stateflow block
- Event and data information exchanged between the Stateflow block and either its Simulink model or external sources
- MATLAB workspace
- Definitions in external code sources

Typical Tasks to Define Stateflow Interfaces

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

The tasks are presented in this section in the order of appearance in this list. This could be a typical sequence. You may find a particular sequence complements your model development process better than another.

Where to Find More Information on Events and Data

See these sections for conceptual information on data and events: “Defining Events” on page 3-2 and “Defining Data” on page 3-14. These references in particular are relevant to defining the interface:

- “Defining Input Events” on page 3-7
- “Defining Output Events” on page 3-8
- “Importing Events” on page 3-9
- “Exporting Events” on page 3-9
- “Defining Input Data” on page 3-21
- “Defining Output Data” on page 3-22

Defining the Stateflow Block Update Method

Stateflow Block Update Methods

Stateflow blocks are Simulink subsystems. You have some flexibility in defining the type of Simulink subsystem of a particular Stateflow block. The chart is awakened when an event occurs. You can choose from these methods of having the chart awakened, entered, and executed:

- Triggered/Inherited

This is the default update method.

- Triggered

The Stateflow block is explicitly triggered by a signal originating from a connected Simulink block. The edge trigger can be set to **Rising**, **Falling**, **Either**, or **Function Call**.

- Inherited

The Stateflow block inherits (implicitly) triggers from the Simulink model. These implicit events are the sample times (discrete-time or continuous) of the Simulink signals providing inputs to the chart. The sample times are determined by Simulink to be consistent with various rates of all the incoming signals.

- Sampled

Simulink will awaken (sample) the Stateflow block at the rate you specify. 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 may have different sample times.

- Continuous

Simulink will awaken (sample) 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 *Using Simulink* for more information on these types of Simulink subsystems.

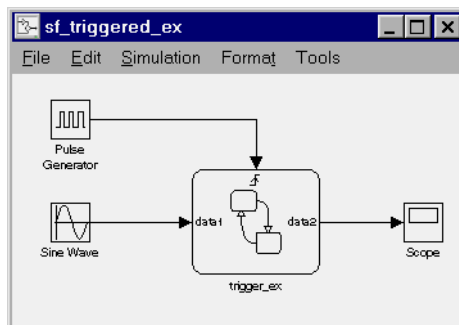
Defining a Triggered Stateflow Block

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 2-29.)
- The chart has an **Input from Simulink** event defined and an edge-trigger type specified. (See “Defining Input Events” on page 3-7.)

Example: Triggered Stateflow Block

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

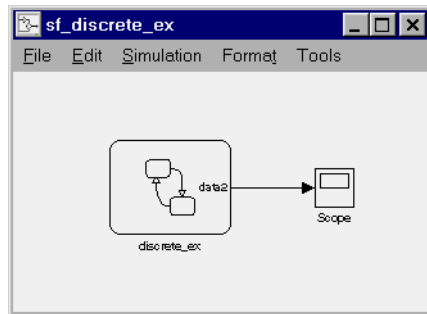
There are two ways you can define a sampled Stateflow block. Setting the chart **Update method** (set in the **Chart Properties** dialog box) to **Sampled** and entering a **Sample Time** value defines a sampled Stateflow block. (See “Specifying Chart Properties” on page 2-29.)

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 “Defining Input Data” on page 3-21.) 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.

Example: Sampled Stateflow Block

A Stateflow block that is not explicitly triggered via the trigger port can be triggered by Simulink by specifying a discrete sample rate. 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

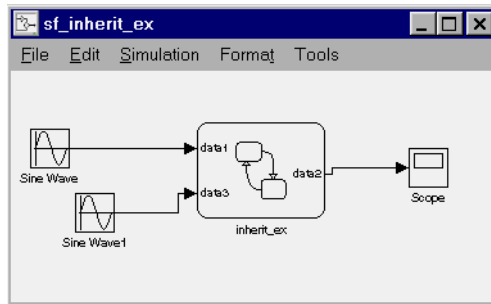
These are essential conditions that define an inherited trigger Stateflow block:

- The chart **Update method** (set in the **Chart Properties** dialog box) is set to **Triggered or Inherited**. (See “Specifying Chart Properties” on page 2-29)
- The chart has an **Input from Simulink** data object defined (added and defined using either the graphics editor **Add** menu or the Explorer). (See “Defining Input Data” on page 3-21.) The chart sample time is determined by Simulink to be consistent with the rate of the incoming data signal.

Example: Inherited Stateflow Block

A Stateflow block that is not explicitly triggered via the trigger port nor is a discrete sample time specified can be triggered by Simulink. 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, the chart **Update method** (set in the **Chart Properties** dialog box) is set to **Continuous**. (See “Specifying Chart Properties” on page 2-29)

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 *Using Simulink* 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.

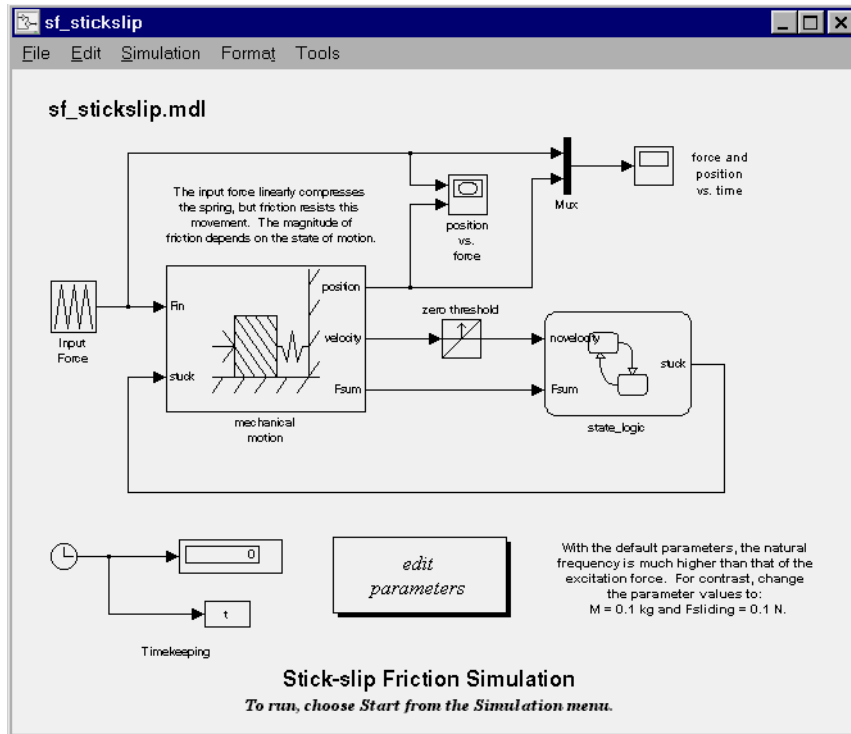
There are situations when changes within the Stateflow block must be felt immediately by the plant and a **Continuous** update is needed:

- Data **Output to Simulink** that is a direct function of data **Input from Simulink** and the data is 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. This is in contrast to states that are assigned, for example, as modes of control strategy.

Example: Continuous Stateflow Block

Simulink will awaken (sample) the Stateflow block at each step in the simulation, as well as at intermediate time points that may be requested by the Simulink solver. This method is consistent with the continuous method in Simulink.

In this 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 Output to Simulink Event Triggers

Overview

Stateflow block output events connect to other Simulink blocks or Stateflow blocks. There are two main options for trigger type:

- Edge-triggered
- Function call

Simulink controls the execution of edge-triggered subsystems. The function call mechanism is a means by which Stateflow executes a subsystem essentially outside of Simulink's direct control. Use a function call trigger to have the Stateflow block control the execution of the connected Simulink block. Function call subsystems are never executed directly by Simulink.

See these examples for more information:

- “Example: Using Function Call Output Events” on page 4-10
- “Example: Function Call Semantics” on page 4-11
- “Example: Edge-Triggered Semantics” on page 4-13

Defining Function Call Output Events

These are essential conditions that define the use of function call output events:

- The chart has an **Output to Simulink** event with a **Function Call** trigger type defined (added and defined using either the graphics editor **Add** menu or the Explorer. See “Defining Output Events” on page 3-8.)
- The Simulink block connected to the function call event **Output to Simulink** has the **Trigger** type field set to **function-call**.
- Stateflow blocks that have feedback loops from a block triggered by a function call should avoid placing any other blocks in the connection lines between the two blocks.

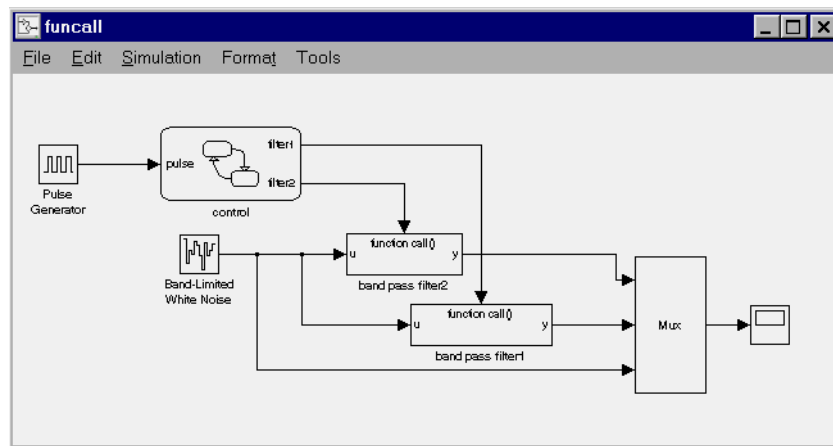
Example: Using Function Call Output Events

A function call trigger operates essentially like a programming subroutine call. When the system executes the step where the function call is specified, the

triggered subsystem executes and then returns to the next statement in the execution sequence. Using function call triggers, the Stateflow block can control the execution of other Simulink blocks in the model.

Use a function call event output when you want a Stateflow block (logic portion/control flow) to control one or more Simulink blocks (algorithmic portion/data flow).

This example shows a use of function call output events.



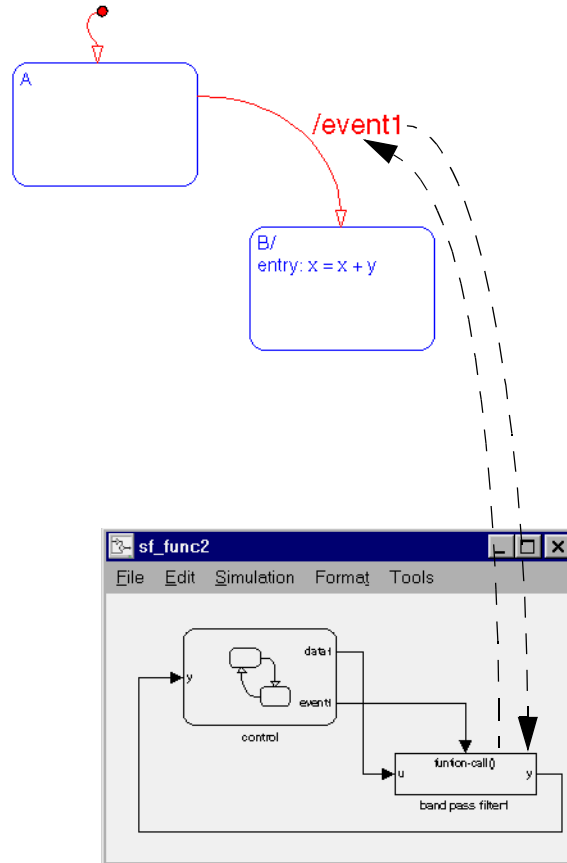
The control block is a Stateflow block that has one data input called pulse and two event **Function Call** outputs called filter1 and filter2. A pulse generator provides input data to the control block. Within the control block, a determination is made whether to make a function call to filter1 or filter2. If, for example, the **Output to Simulink** event **Function Call** filter1 is broadcast, the band pass filter1 block executes and then returns to the next execution step in the control block. As part of its execution, band pass filter1 receives unfiltered input data and outputs filtered data for display on a scope.

The Stateflow block controls the execution of band pass filter1 and band pass filter2.

Example: Function Call Semantics

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

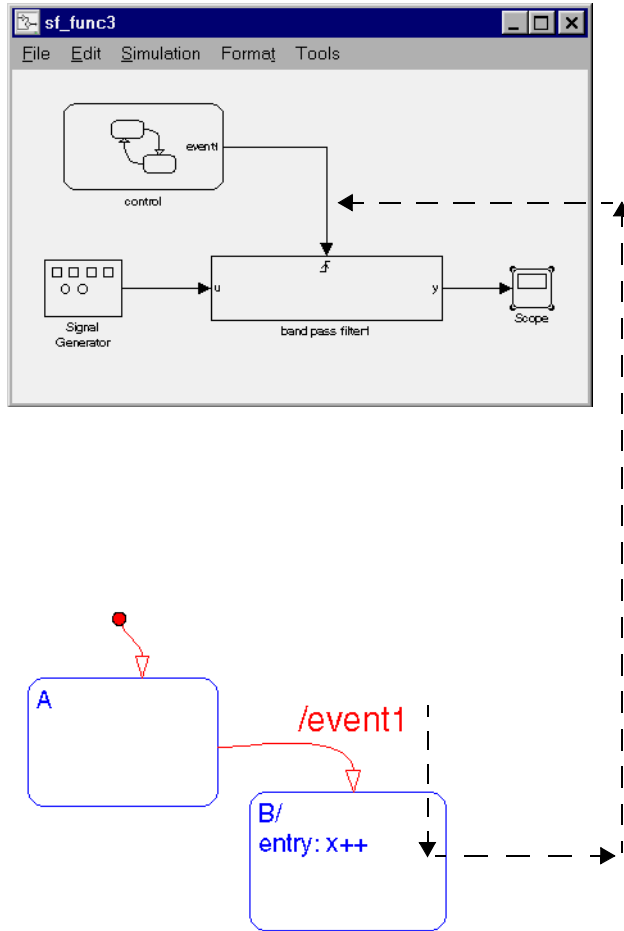
These are essential conditions that define the use of triggered output events:

- The chart has an **Output to Simulink** event with a trigger type: **Either Edge**. (See “Defining Output Events” on page 3-8 .)
- The Simulink block connected to the edge triggered event **Output to Simulink** has the **Trigger** type field set to the equivalent edge triggering type.

Example: Edge-Triggered Semantics

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 **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, 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.

Event Inputs from Simulink

These tasks describe how to add and define the necessary fields for an event input from Simulink:

- Add an event choosing a chart as the parent of the event
- Choose **Input from Simulink** as the **Scope**
- Specify the **Trigger** type
- Apply and save the changes

Add an Event Choosing a Chart as the Parent

These steps describe how to add an event:

- 1 Choose **Explore** from the graphics editor **Tools** menu to invoke the Explorer.
- 2 Select the chart object in the hierarchy that you want to be the event's parent.

You must explicitly choose a parent to create an event. Choosing the chart to be the parent of the event enables receive rights to Simulink, to the chart, and all its offspring.

- 3 Choose **Event** from the Explorer **Add** menu. The **Event properties** dialog box is displayed.
- 4 Enter a name in the **Name** field.

Choose Input from Simulink as the Scope

Once you have chosen the chart as the parent, the choice of valid scopes includes: **Local**, **Input from Simulink**, or **Output to Simulink**.

Choose **Input from Simulink** as the **Scope** to enable send rights to Simulink and any offspring of the chart and to enable receive rights to the chart and all of its offspring.

When you add an event input, a single Simulink trigger port is added to the top of the Stateflow block.

Select the Trigger

The trigger defines how the Stateflow block's input events are handled in the context of their Simulink model. The **Trigger** type indicates what kind of signal has meaning for the input event. The **Trigger** can have these values.

Keyword	Description
Either Edge	Either rising or falling edge trigger, where the control signal changes either from 0 to a nonzero value, from a negative value to a positive value, or from a positive value to a negative value.
Rising Edge	Rising edge trigger, where the control signal changes from either 0 to a positive value or from a negative value to a positive value.
Falling Edge	Falling edge trigger, where the control signal changes from either 0 to a negative value or from a positive value to a negative value.
Function Call	Function call triggered.

Each Stateflow block can only have one overall trigger type. See “Specifying Trigger Types” on page 3-10 for more information.

Apply the Changes

Click on the **Apply** button to save the properties. Click on the **OK** button to save the properties and close the dialog box.

Data Inputs from Simulink

These tasks describe how to add and define the necessary fields for a data input from Simulink:

- Add a data object choosing a chart as the parent of the data
- Choose **Input from Simulink** as the **Scope**
- Specify data attributes
- Apply and save the changes

Add a Data Object Choosing a Chart as the Parent

These steps describe how to add a data object:

- 1 Choose **Explore** from the graphics editor **Tools** menu to invoke the Explorer.
- 2 Select a chart object in the hierarchy that you want to be the data object's parent.

You must explicitly choose a parent to create a data object. Choosing the Chart to be the parent determines that the data resides within the chart.

- 3 Choose **Data** from the Explorer **Add** menu. The **Data properties** dialog box is displayed.
- 4 Enter a name in the **Name** field.

Choose Input from Simulink as the Scope

Once you have chosen the chart as the parent, the choice of valid scopes includes: **Local**, **Input from Simulink**, **Output to Simulink**, **Temporary**, or **Workspace**.

Choose **Input from Simulink** as the **Scope** to enable access rights to Simulink and any offspring of the chart.

When you add a data input, each data input is represented on the Stateflow block by a Simulink input port. Multiple data inputs to the Stateflow block must be scalar (they cannot be vectorized).

Specify Data Attributes

If you want to change the defaults, you can specify data **Units**, **Type**, **Initial**, **Minimum**, and **Maximum** values.

Apply and Save the Changes

Click on the **Apply** button to save the properties. Click on the **OK** button to save the properties and close the dialog box.

Event Outputs to Simulink

These tasks describe how to add and define the necessary fields for an event output to Simulink:

- Add an event choosing a Chart as the parent of the data
- Choose **Output to Simulink** as the Scope
- Specify the **Trigger** type
- Apply and save the changes

Add an Event Choosing a Chart as the Parent

These steps describe how to add an event:

- 1 Choose **Explore** from the graphics editor **Tools** menu to invoke the Explorer.
- 2 Select a chart object in the hierarchy that you want to be the event's parent.

You must explicitly choose a parent to create an event. Choosing the Chart to be the parent of the event enables receive rights to Simulink and to the chart and all its offspring.

- 3 Choose **Event** from the Explorer **Add** menu. The **Event properties** dialog box is displayed.
- 4 Enter a name in the **Name** field.

Choose Output to Simulink as the Scope

Once you have chosen the chart as the parent, the choice of valid scopes includes: **Local**, **Input from Simulink**, or **Output to Simulink**.

Choose **Output to Simulink** as the **Scope** to enable send rights to Simulink and any offspring of the chart and to enable receive rights to the Simulink model.

When you define an event to be an **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.

Apply the Changes

Click on the **Apply** button to save the properties. Click on the **OK** button to save the properties and close the dialog box.

Data Outputs to Simulink

These tasks describe how to add and define the necessary fields for a data output to Simulink:

- Add a data object choosing a chart as the parent of the data
- Choose **Output to Simulink** as the **Scope**
- Specify data attributes
- Apply and save the changes

Add a Data Object Choosing a Chart as the Parent

These steps describe how to add a data object:

- 1 Choose **Explore** from the graphics editor **Tools** menu to invoke the Explorer.
- 2 Select a chart object in the hierarchy that you want to be the data object's parent.

You must explicitly choose a parent to create a data object. Choosing the Chart to be the parent determines that the data resides within the chart.

- 3 Choose **Data** from the Explorer **Add** menu. The **Data properties** dialog box is displayed.
- 4 Enter a name in the **Name** field.

Choose Output to Simulink as the Scope

Once you have chosen the chart as the parent, the choice of valid scopes includes: **Local**, **Input from Simulink**, or **Output to Simulink**.

Choose **Output to Simulink** as the **Scope** to enable access rights to Simulink and any offspring of the chart.

When you define a data object to be an **Output to Simulink**, a Simulink output port is added to the Stateflow block. Output data objects from the Stateflow block to the Simulink model are scalar.

Specify Data Attributes

If you want to change the defaults, you can specify data **Units**, **Type**, **Initial**, **Minimum**, and **Maximum** values.

Apply the Changes

Click on the **Apply** button to save the properties. Click on the **OK** button to save the properties and close the dialog box.

MATLAB Workspace

What Is the MATLAB Workspace?

The MATLAB workspace is the area of memory accessible from the MATLAB command line. The workspace maintains the set of variables built up during a MATLAB session.

See *Getting Started with MATLAB* and *Using MATLAB* for more information.

Using the MATLAB Workspace

You can define data objects to have a **Workspace Scope**. Any data objects defined with **Workspace Scope** must have value assigned in the MATLAB command window prior to starting a simulation.

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.

Using the **Workspace data Scope** is only meaningful for `sfun` simulation and RTW targets.

Defining the Interface to External Sources

What Are External Sources?

Any code that is not part of a Stateflow diagram, the Stateflow machine, nor the Simulink model is considered external. You can include external source code in the **Target Options** section of the **Target Builder** dialog box. (See “Building Custom Code into the Target” on page 9-3.)

See Chapter 3, “Defining Events and Data” for information on defining events and data.

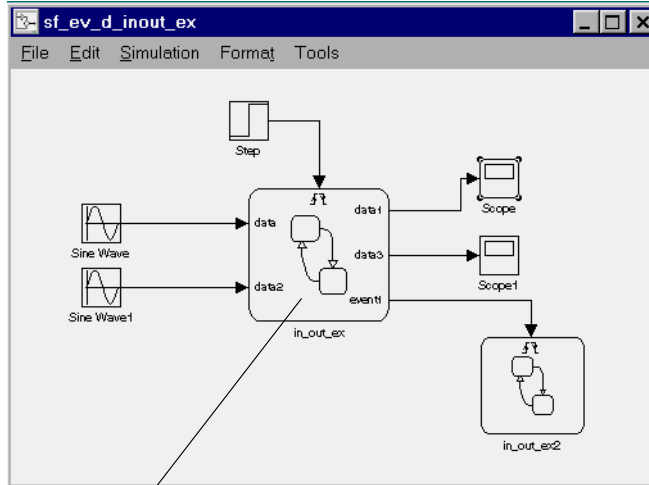
Exported Events

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 whatever you might be doing. 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.

Similarly, you may want an external source (outside the Stateflow diagram, the machine, and the Simulink model) to be able to broadcast an event. By defining an event’s scope to be **Exported**, that event is made available to external sources for broadcast purposes. **Exported** events must be parented by the machine because the machine is the (highest) level in the Stateflow hierarchy that can interface to external sources. The 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 is another machine, then one machine defines an **Exported** event and the other machine defines the same event to be **Imported**. Stateflow generates the appropriate export and import event code for both machines.

This example shows the format required in the external code source (custom code) to take advantage of an **Exported** event.



e is added and defined as an **Exported** event.

Stateflow generates this code:

```
void external_broadcast_e (void)
{
/* code based on the event
definition
*/
...
{
```

External code source

```
void func_example(void)
{
extern void external_broadcast_e (void);
...
external_broadcast_e();
...
}
```

e is imported in the external code source.

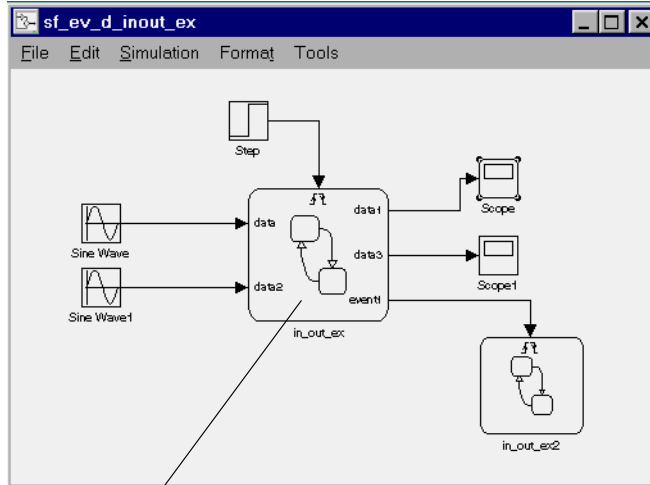
Imported Events

Consider the same pager example discussed for **Exported** events to clarify the use of **Imported** events. Someone buys a pager and indicates you may want to use this number to page them in the future. They tell you the pager number and you take note of the number by writing it down. You can then use the number to page that person. You cannot receive the page because you do not own the pager.

Similarly, you may want to broadcast an event that is externally (outside the Stateflow diagram, the machine, and the Simulink model) defined. By defining an event's scope to be **Imported**, the event can be broadcast 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 machine because the machine is the (highest) level in the Stateflow hierarchy that can interface to external sources. Importing the event does not imply anything about what the external source does with the information. It is the responsibility of the external source to make the **Imported** event available (in the manner appropriate to the source) to make use of the right to broadcast the event.

If the external source is another machine then one machine defines an **Imported** event and the other machine defines the same event to be **Exported**. Stateflow generates the appropriate import and export event code for both machines.

This example shows the format required in the external code source (custom code) to make the event available.



e is exported in the external code source.

e is added and defined as an **Imported** event.

Stateflow generates this code for the **Imported** event:

```
extern void external_broadcast_e (void);
```

External code source

```
void external_broadcast_e (void)
{
...
}
```

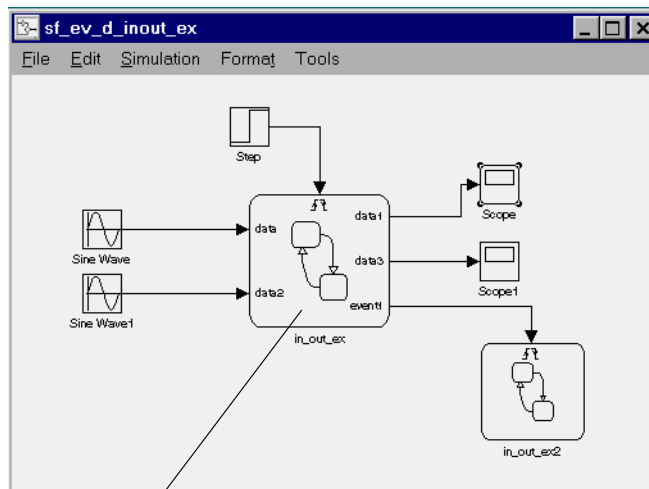
Exported Data

You may want an external source (outside the Stateflow diagram, the machine, and the Simulink model) to be able to access a data object. By defining a data object's scope to be **Exported**, that data is made accessible to external sources. **Exported** data must be parented by the machine because the machine is the (highest) level in the Stateflow hierarchy that can interface to external sources. The 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 machine, then one machine defines an **Exported** data object and the other machine defines the same data object to be **Imported**. Stateflow generates the appropriate export and import data code for both machines.

This example shows the format required in the external code source (custom code) to import an **Exported** data object.



`ext_data` added and defined as an **Exported** data.

Stateflow generates this code:

```
int ext_data;
```

`ext_data` is defined as imported in the external code source

External codesource

```
extern int ext_data;
void func_example(void)
{
...
ext_data = 123;
...
}
```

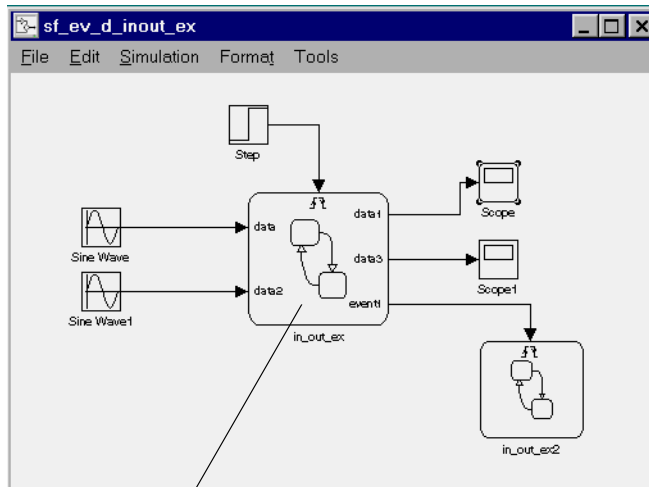
Imported Data

Similarly, you may want to access a data object that is externally (outside the Stateflow diagram, the machine, and the Simulink model) defined. By defining

a data's scope to be **Imported**, the data can be accessed anywhere within the hierarchy of that 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 machine because the machine is the (highest) level in the Stateflow hierarchy that can interface to external sources. Importing the data object does not imply anything about what the external source does with the data object. It is the responsibility of the external source to make the **Imported** data object available (in the manner appropriate to the source) to make use of the right to access the data object.

If the external source is another machine, then one machine defines an **Imported** data object and the other machine defines the same data object to be **Exported**. Stateflow generates the appropriate import and export data code for both machines.

This example shows the format required if the data is **Imported** from an external code source (custom code).



ext_data added and defined as an **Imported** data.

Stateflow generates this code:

```
extern int ext_data;
```

ext_data is defined as exported in the external code source.

External code source

```
int ext_data;
void func_example(void)
{
...
}
```


Exploring and Searching

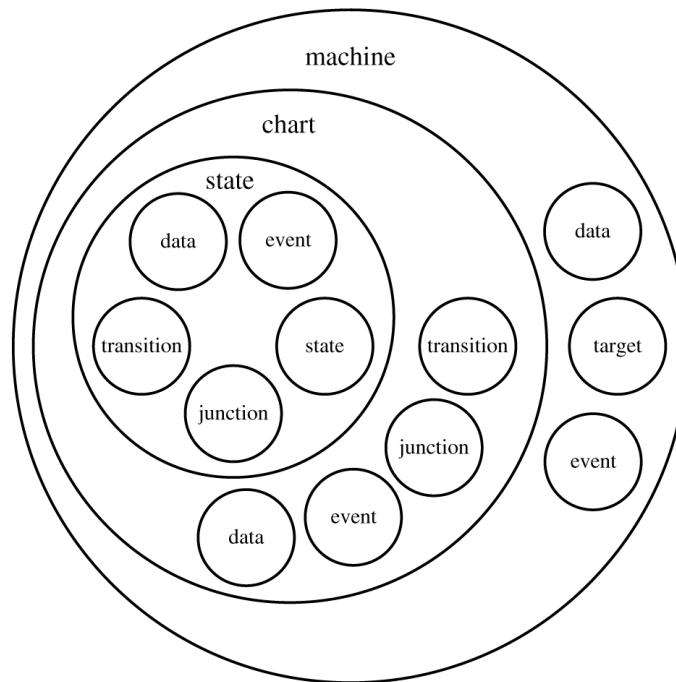
Overview	5-2
Stateflow Explorer User Interface	5-4
Stateflow Finder User Interface	5- 8
Searching for a Wildcard String	5-14
Searching for an Exact String Match	5-16
Searching for All Objects of a Specified Object Type . .	5-18
Searching Based on Multiple Conditions	5-20
Searching for an Object to Edit Properties	5-22

Overview

Exploring and Searching

The Stateflow machine is the highest level in the Stateflow hierarchy. The object hierarchy beneath the Stateflow machine consists of combinations of the graphical and nongraphical objects. The data dictionary is the repository for all Stateflow objects. You can use the Stateflow Finder and Stateflow Explorer together to browse and make changes to data dictionary objects.

Stateflow data dictionary



The Explorer displays any defined events, data, and targets within an object hierarchy where machines, charts, and states are potential parents.

You can create, modify, and delete events, data, and target objects using the Explorer. You can also add events, data, and targets using the graphics editor **Add** menu. (See “Defining Events” on page 3-2 for more information.) If you

add data or events via the **Add** menu, the chart is automatically defined as the parent. If you add a target, the machine is defined as the parent. Targets can only be parented by the machine. If you want to change the parent of a data or event object, you must use the Explorer to do so. Similarly you must use the Explorer if you want to delete an event, data, or target object.

See “Configuring a Target” on page 9-9 for information on customizing the simulation target. See “Adding a Target to a State Machine’s Target List” on page 9-9 for information on creating targets to generate code using the Stateflow Coder product.

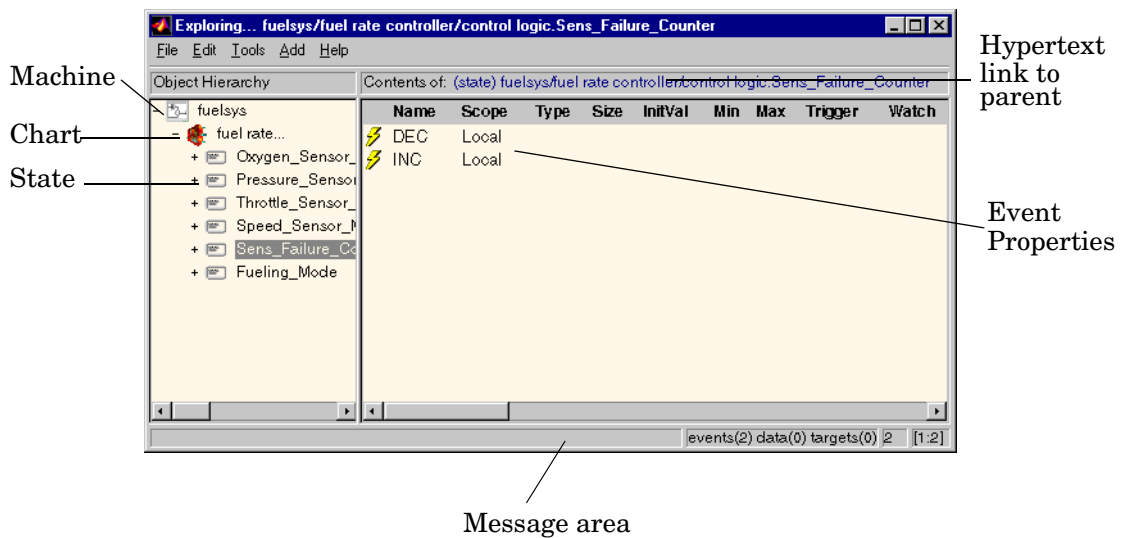
The Finder displays a list of objects based on search criterion you specify. You can directly access the properties dialog box of any object in the search output display by double-clicking on that object.

The Finder, the Explorer, and the **Add** menu on the graphics editor provide access to the event, data, and target properties dialog boxes.

Stateflow Explorer User Interface

Explorer Main Window

This is the Explorer main window showing the object hierarchy of an example chart (explore_ex).



Object Hierarchy

The **Object Hierarchy** (machines, charts, and states) is displayed in the left-hand pane. A '+' character indicates that the hierarchy can be expanded by double-clicking on that entry (or by clicking on the '+' character). A '-' character indicates there is nothing to expand. Clicking on an entry in the **Object Hierarchy** selects that entry.

Contents Pane

Data, and target objects parented by the currently selected object in the **Object Hierarchy** are displayed in the **Contents** pane. Each type of object has an icon. The entry for a data object displays selected properties of the object.

These are the possible parent and object combinations.

	Machine	Chart	State
Event	yes	yes	yes
Data	yes	yes	yes
Target	yes	no	no

Targets are parented exclusively by machines. (Although all other combinations are valid, there are guidelines describing how **Scope** affects choice of parent and vice-versa.) The default `sfun` simulation target is automatically defined for every machine. If you have a Real-Time Workshop license, a Real-Time Workshop target is also automatically added:

- When you select **Open RTW Target** from the graphics editor **Tools** menu
- If you build a target that includes a Stateflow machine using Real-Time Workshop

For convenience, a hypertext link to the parent of the currently selected object in the **Object Hierarchy** is included following the **Contents of:** label. Click on the hypertext link to bring that object to the forefront.

File Menu

Select **Close (Ctrl+W)** from the **File** menu to close the Explorer window.

Edit Menu

Select **Cut (Ctrl+X)** from the **Edit** menu to delete the selected data, event, or target object in the **Contents of:** pane.

Add Menu

The **Add** menu provides these options.

Menu Choice	Description
Event (Ctrl+E)	Creates a new event at the selected level in the object hierarchy; displays an Event properties dialog box.
Data (Ctrl+D)	Creates a new data object at the selected level in the object hierarchy; displays a Data properties dialog box.
Target (Ctrl+T)	Creates a new untitled standalone target at the selected machine level in the object hierarchy (a target's parent must be a machine; this menu item is disabled if a chart or state parent type is selected); displays an untitled Target Configuration properties dialog box.

Help Menu

The Help menu provides these options.

Menu Choice	Description
Topics (HTML)	Displays Stateflow online help in an HTML browser. An online version of the <i>Stateflow User's Guide</i> is one of the options available from the online help.
Help Desk (HTML)	Displays the MATLAB Help Desk in an HTML browser.
About Stateflow	Displays a pop-up window with software version information.

Moving Objects/Changing Parent

To create desired behavior you may need to change the parent of an event, data, or target object.

Objects in the **Contents of:** pane can be moved in the hierarchy to change an object's parent. Click and drag an object from the **Contents of:** pane to a new location in the **Object Hierarchy** pane to change its parent. If the object is the current parent, an X with a circle around it is displayed (indicating this is an invalid operation). If you move an object to a level in the hierarchy that does not support that object's current **Scope** property, the **Scope** is changed to **Local**.

Moving Objects/Changing Index and Port Order

To ensure proper ordering of event and/or data **Input from** or **Output to Simulink** you may need to move some of these objects in the Explorer.

Click and drag a data object with **Input from** or **Output to Simulink Scope** to a new position in the **Contents of:** pane **Data list** to change its port number. Click and drag an event **Input from** or **Output to Simulink Scope** to a new position in the **Contents of:** pane **Event list** to change its index number.

Deleting Objects

Select the object in the **Contents of:** pane and press the **Delete** key or select **Cut (Ctrl+X)** from the **Edit** menu to delete an object.

Editing Objects

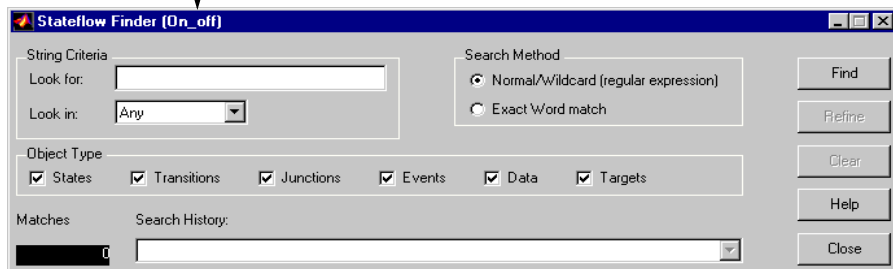
To edit a state or chart displayed in the Explorer's **Object Hierarchy** pane, select the object, display its context menu by clicking the right mouse button, and select **Edit** from the context menu. Stateflow displays the selected object in the Stateflow editor.

Stateflow Finder User Interface

Finder Dialog Box

The Finder operates on a machine. This is the Finder dialog box.

Stateflow Machine name



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 drop down **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 on 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 defines 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.

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 type(s) to search by toggling the radio 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:

- States
- Transitions
- Junctions
- Events
- Data
- Targets

Find Button

Click on 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 on the **Refine** button to narrow the previously entered search criteria. An ampersand(&) is prepended 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 on the pull-down list to display search refinements. An ampersand is prepended 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 the **Clear** button to clear any previously specified search criteria. Results are removed and the search criteria is reset to the default settings.

Close Button

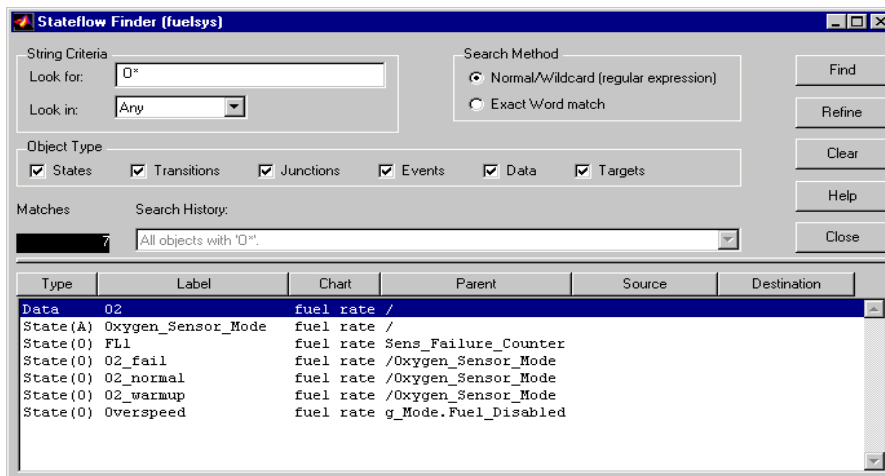
Click the **Close** button to close the Finder.

Help Button

Click the **Help** button to display the Stateflow online help in an HTML browser window.

Finder Display Area

The Finder display area looks like this.



The display area is divided into these fields.

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.

Field	Description
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, is used for resorting.

Each field label is also a button. Click on 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.

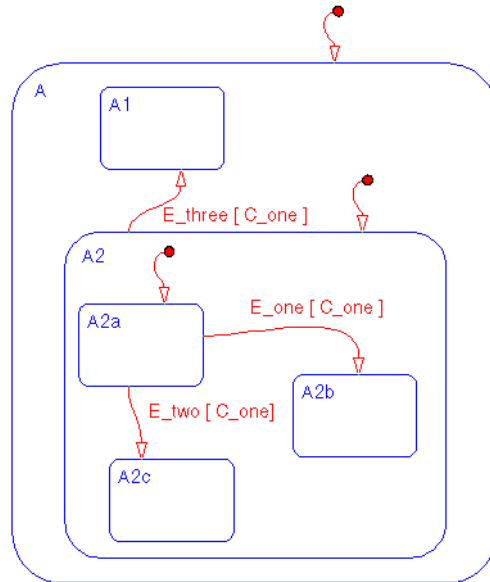
The Finder can be resized vertically to display more output rows, but cannot be expanded horizontally.

Click on a graphical entry to highlight that object in the graphical editor window. Double-click on an entry to invoke the **Property** dialog box for that object. Right-click the entry to display a pop-up 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 . character. The **Source** and **Destination** fields use the combination of the ~ and the . characters to denote that the state listed is relative to the **Parent** hierarchy.

Using this Stateflow diagram as an example,



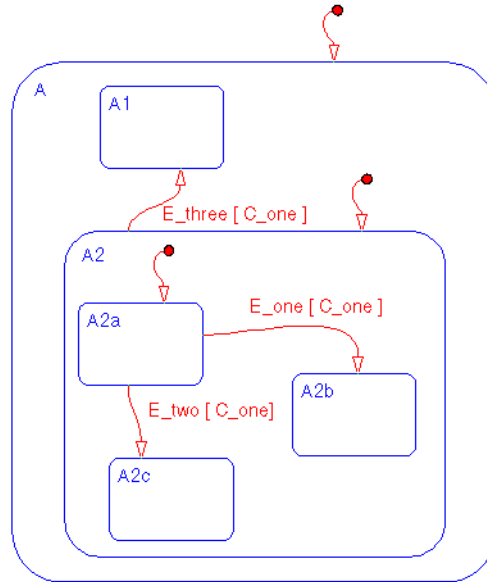
what are the values for the **Parent**, **Source**, and **Destination** fields for the transition from A2a to A2b?

The transition is within state A2. State A2's parent is state A and state A's parent is the Stateflow diagram itself. /A.A2 is the notation for state A2a's parent. State A2a is the transition source and state A2b is the destination. These states are at the same level in the hierarchy. ~.A2a is the relative hierarchy notation for the source of the transition. 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.

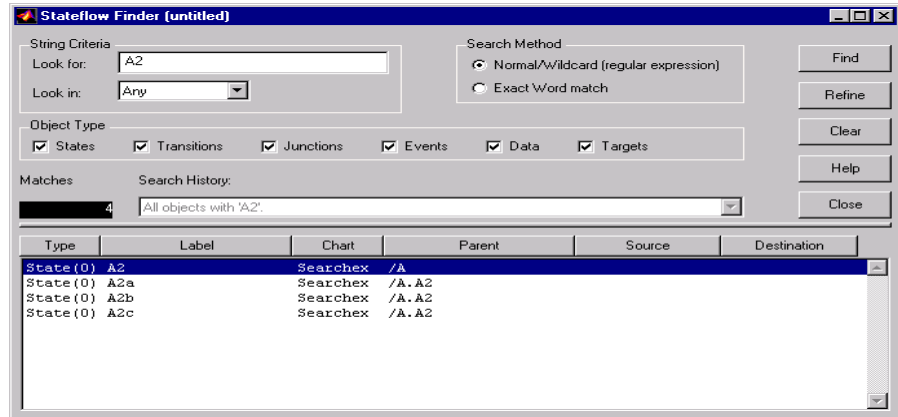
Searching for a Wildcard String

Example

This example Stateflow diagram is used to show how to find all object labels containing a specified wildcard string.



- 1 Select **Find...** from the **Tools** menu of the Stateflow graphics editor window to invoke the Finder.
- 2 Click in the **Look for** text field and enter A2.
- 3 Click on and enable the **Normal/Wildcard** radio box.

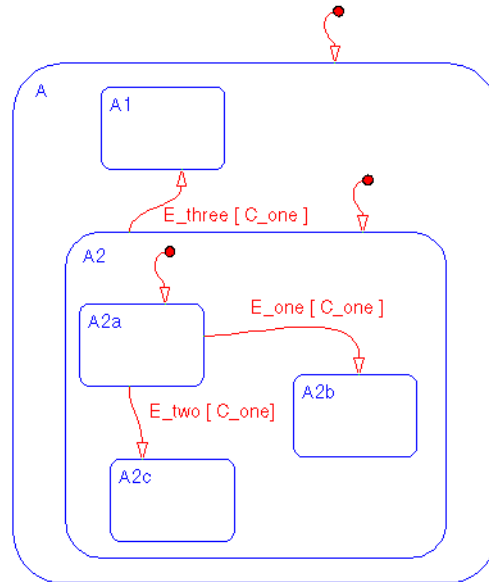
4 Click on the **Find** button. These are the results of the search.

There are four matches. The **Search History** shows that the current search criterion is All objects with 'A2'. The display shows the matches are all exclusive OR (O) states.

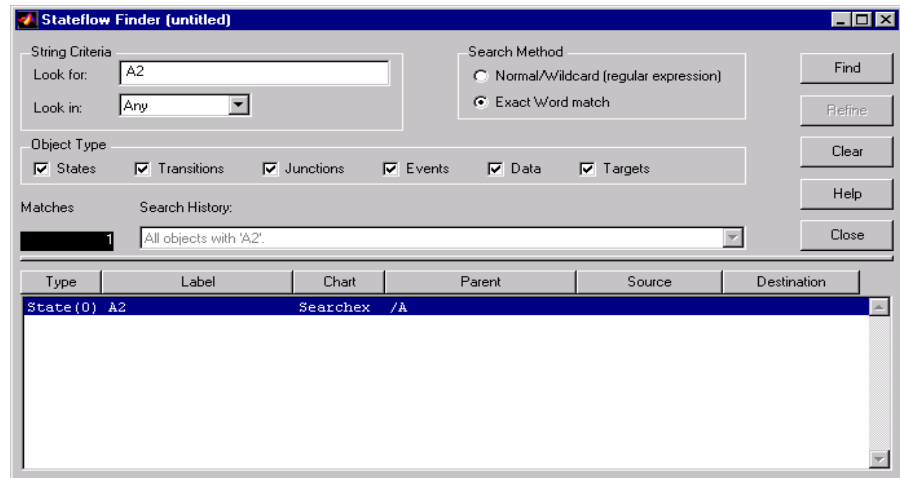
Searching for an Exact String Match

Example

This example Stateflow diagram is used to show how to find all objects with an exact string match in the label.



- 1 Select **Find...** from the **Tools** menu of the Stateflow graphics editor window to invoke the Finder.
- 2 Click in the **Look for** text field and enter A2.
- 3 Click on and enable the **Exact word match** radio button.

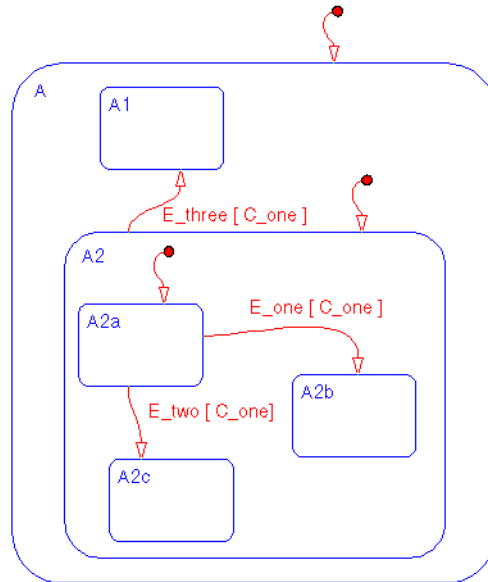
4 Click on the **Find** button. These are the results of the search.

There is one match. The **Search History** shows that the current search criterion is All objects with 'A2'. The display shows the match is an exclusive (O) state labeled A2 in a Stateflow diagram titled search_ex. The parentage of that state is /A.

Searching for All Objects of a Specified Object Type

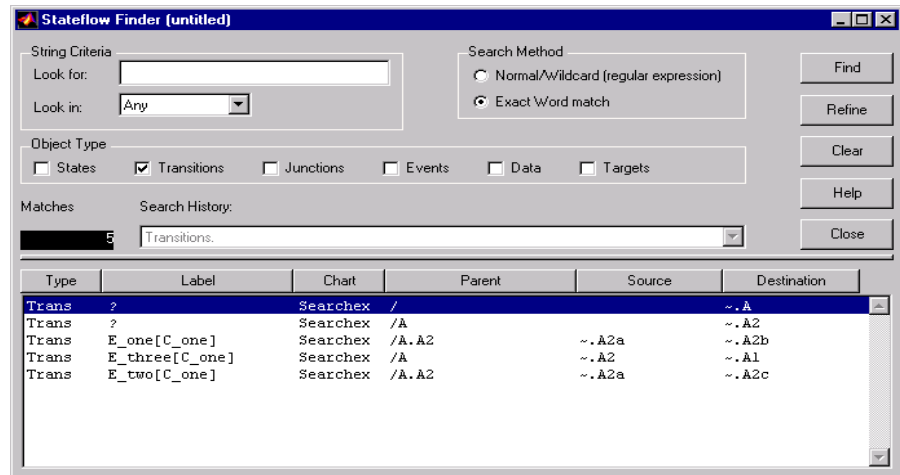
Example

This Stateflow diagram is used to show how to search for all objects of a specified type.



- 1 Select **Find...** from the **Tools** menu of the Stateflow graphics editor window to invoke the Finder.
- 2 Ensure that the **Transitions** object type is checked. A checkmark indicates that the object is included in the search criteria. By default, all object types are included in the search criteria. Deselect all other object types.

3 Click on the **Find** button. These are the results of the search.

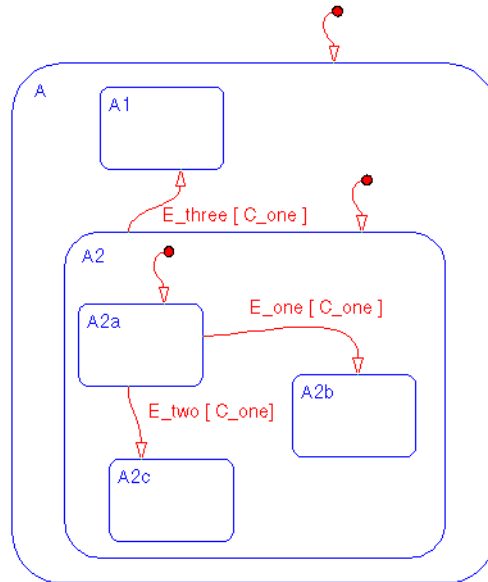


There are six matches. The **Search History** shows that the current search criterion is Transitions. The display shows the matches.

Searching Based on Multiple Conditions

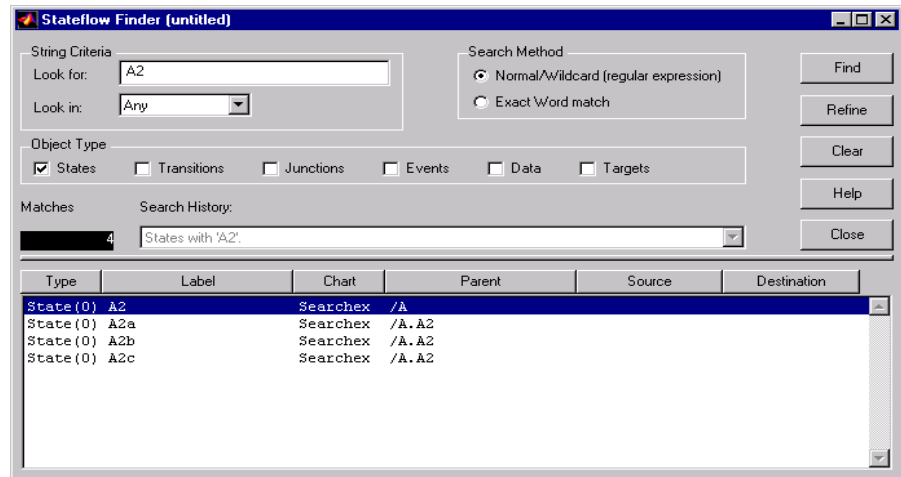
Example

This Stateflow diagram is used to show how to search based on multiple conditions.



- 1 Select **Find...** from the **Tools** menu of the Stateflow graphics editor window to invoke the Finder.
- 2 Click in the **Look for** text field and enter A2.
- 3 Click on and enable the **Normal/Wildcard** radio box.
- 4 Ensure that the **States** object type is checked. A checkmark indicates that the object is included in the search criteria. By default, all object types are included in the search criteria. Deselect all other object types.

- 5 Click on the **Search** button. These are the results of the search.

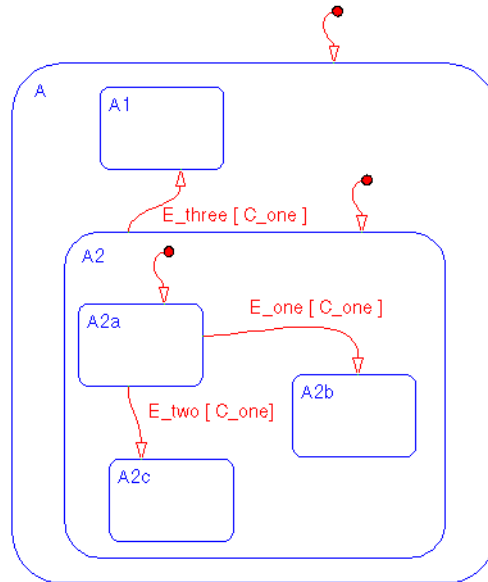


There are four matches. The **Search History** shows that the current search criterion is States with 'A2'. The display shows the matches.

Searching for an Object to Edit Properties

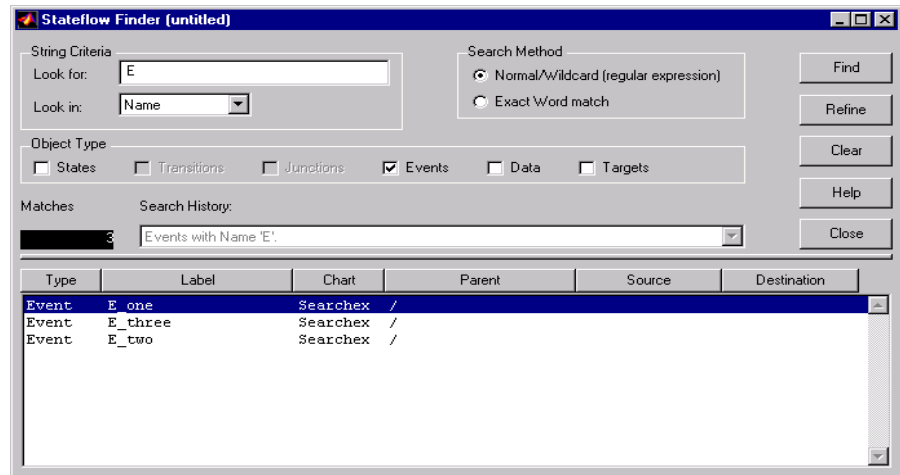
Example

This Stateflow diagram is used to show how to search for an object and edit its properties.



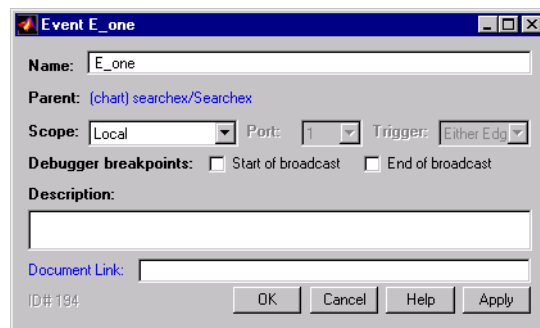
- 1 Select **Find...** from the **Tools** menu of the Stateflow graphics editor window to invoke the Finder.
- 2 Click in the **Look for** text field and enter E.
- 3 Choose **Name** from the **Look In** field pull-down menu.
- 4 Click on and enable the **Normal/Wildcard** radio box.
- 5 Ensure that the **Events** object type is checked. Event and data objects must already be defined using either the Explorer or the graphics editor **Add** menu. A checkmark indicates that the object is included in the search criteria. By default, all object types are included in the search criteria. Deselect all other objects.

- 6 Click on the **Search** button. These are the results of the search.



There are three matches. The **Search History** shows that the current search criterion is Events with name 'E'. The display shows the matches.

- 7 Double-click on event E_one to display the **Event properties** dialog box.



- 8 Add a description. Click on the **Close** button to save the changes and close the **Event properties** dialog box. An event's parent can be changed only through the Explorer.

Notations

Overview	6-2
States	6-9
Transitions	6-16
Connective Junctions	6-31
History	6-39
Action Language	6-41
Using the Notation Efficiently	6-63

Overview

What Is Meant by Notation?

A 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:

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

Motivation Behind the Notation

Chapter 2, “Creating Charts,” and Chapter 3, “Defining Events and Data,” discuss how to use the product to create the various objects. Knowing how to create the objects is the first step to designing and implementing a Stateflow diagram. The next step is understanding and using the notation to create a well-designed and efficient Stateflow diagram.

This chapter focuses on the notation: the supported relationships amongst the graphical objects and the action language that dictates the actions that can be associated with states and transitions. The Stateflow notation supports many different ways of representing desired system behavior. The representation you choose directly affects the efficiency of the generated code. You can use the notation to optimize for these code attributes:

- Speed
- Code size
- Memory use

See “Using the Notation Efficiently” on page 6-63 for more information.

How Is the Notation Checked?

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” on page 9-17 for more information.

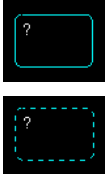



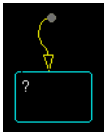



Some aspects of the notation are verified at runtime. Using the Debugger you can detect runtime errors such as:

- State inconsistencies
- Conflicting transitions
- Data range violations
- Cyclic behavior

You can modify the notation to resolve runtime errors. See Chapter 10, “Debugging,” for more information on debugging runtime errors.

What Are the Graphical Objects?

These are the graphical objects in the notation that are on the toolbar.

Name	Notation	Toolbar Icon
State		
History junction		
Default transition		
Connective junction		

A transition is a curved line with an arrowhead that links one graphical object 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.

What Are the Nongraphical Objects?

Event and data objects do not have graphical representations. These objects are defined using the Stateflow Explorer. See Chapter 3, “Defining Events and Data.”

What Is the Data Dictionary?

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 Explorer. The parser evaluates entries and relationships between entries in the data dictionary to verify the notation is correct.

Keywords

These Stateflow keywords have special meaning in the notation.

Keyword	Shorthand	Meaning
<code>change(data_name)</code>	<code>chg(data_name)</code>	Generates a local event when the value of <code>data_name</code> changes.
<code>during:</code>	<code>du</code>	Actions that follow are executed as part of a state's during action.
<code>entry:</code>	<code>en</code>	Actions that follow are executed as part of a state's entry action.
<code>entry(state_name)</code>	<code>en(state_name)</code>	Generates a local event when the specified <code>state_name</code> is entered.
<code>exit</code>	<code>ex</code>	Actions that follow are executed as part of a state's exit action.
<code>exit(state_name)</code>	<code>ex(state_name)</code>	Generates a local event when the specified <code>state_name</code> is exited.
<code>in(state_name)</code>	<code>none</code>	A condition function that is evaluated as true when the <code>state_name</code> specified as the argument is active.
<code>on event_name</code>	<code>none</code>	Actions that follow are executed when the <code>event_name</code> specified as an argument to the <code>on</code> keyword is broadcast.

Keyword	Shorthand	Meaning
<code>send(event_name, state_name)</code>	none	Send the event_name specified to the state_name specified (directed event broadcasting).
<code>matlab(evalString, arg1, arg2, ...)</code>	<code>m1()</code>	Action specifies a call using MATLAB function notation.
<code>matlab.MATLAB_workspace_data</code>	<code>m1.</code>	Action specifies a call using the m1 name space notation.

Use of these keywords in any way other than their intended meaning within the rules of the notation will cause unpredictable results.

How Is Hierarchy Represented?

The notation supports the representation of object hierarchy in Stateflow diagrams. Some of the objects are graphical while others are nongraphical.

An example of a graphical hierarchy is the ability to draw one state within the boundaries of another state. Such a representation indicates that the inner state is a substate or child of the outer state or superstate. The outer state is the parent of the inner state. In the simple case of a Stateflow diagram with a single state, the Stateflow diagram is that state's parent. Transitions are another example of graphical hierarchy. A transition's hierarchy is represented by determining its parent, source, and destination. In a Stateflow diagram you can see a transition's parent, source, and destination.

Data and event object are nongraphical and their hierarchy is represented differently (using the Explorer) from the graphical object hierarchy (using the graphics editor).

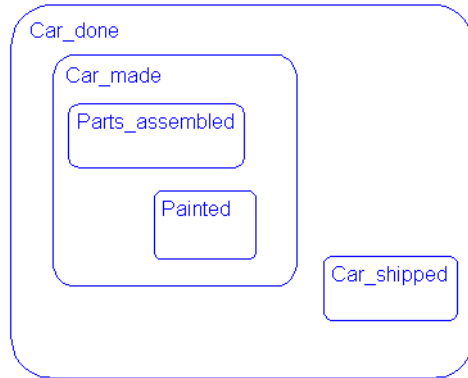
All of the objects in the notation support the representation of hierarchy.

See Chapter 3, "Defining Events and Data," and Chapter 4, "Defining Stateflow Interfaces," for information and examples of representing data and event objects.

For more information on how the hierarchy representations are interpreted, see Chapter 7, "Semantics."

Example: Representing State Hierarchy

This is an example of how state hierarchy is represented.



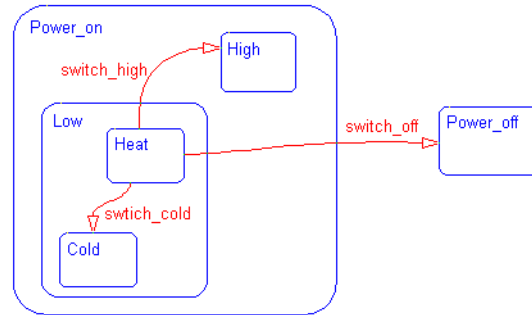
The Stateflow diagram is the parent of Car_done. Car_done is the parent state of the Car_made and Car_shipped states. Car_made is also a parent to the Parts_assembled and Car_painted states. Parts_assembled and Car_painted are children of the Car_made state.

The machine is the root of the Stateflow hierarchy. The Stateflow diagram is represented by the / character. Each level in the hierarchy of states is separated by the . character. The full hierarchy representation of the state names in this example is:

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

Example: Representing Transition Hierarchy

This is an example of how transition hierarchy is represented.



A transition's hierarchy is described in terms of the transition's parent, source, and destination. The parent is the lowest level that the transition (source and destination) is contained within. The machine is the root of the hierarchy. The Stateflow diagram is represented by the / character. Each level in the hierarchy of states is separated by the . character. The three transitions in the example are represented as:

Transition Label	Parent	Source	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

Example: Representing Event Hierarchy

Event hierarchy is defined by specifying the parent of an event when you create it. Events are nongraphical and are created using either the graphics editor **Add** menu or the Explorer. Using hierarchy you can optimize event processing through directed event broadcasting. Directed event broadcasting is the ability to qualify who can send and receive event broadcasts.

See “Defining Events” on page 3-2 for more information.

See “Action Language” on page 6-41 for more information on the notation for directed event broadcasting.


States

Overview

A *state* describes a mode of a reactive system. States in a Stateflow diagram represent these modes. The activity or inactivity of the states dynamically changes based on 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.

This table shows the button icon and a short description of a state.

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

Superstate

A state is a superstate if it contains other states, called substates.

Substate

A state is a substate if it is contained by a superstate.

What Is a State 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. When looking at any one point in the hierarchy, all substates of a superstate must be of the same type.

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. The children of parallel (AND) decomposition parents are AND states.

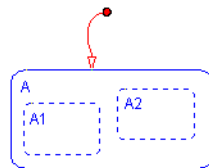
Exclusive (OR) State Decomposition

Exclusive (OR) state decomposition is represented by states with solid borders. Exclusive (OR) decomposition is used to describe system modes that are mutually exclusive. 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.

Active and Inactive States

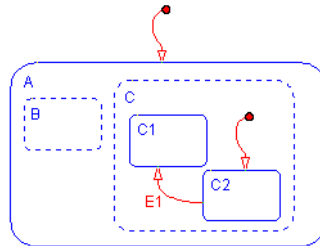
States have the Boolean characteristic of being active or inactive. The occurrence of events drives the execution of the Stateflow diagram. At any point in the execution of a Stateflow diagram, there will be some combination of active and inactive states. These are some possible combinations:

- Multiple active states with parallel (AND) decomposition
In this example, when state A is active, A1 and A2 are active.



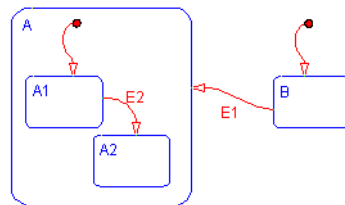
- An active state with parallel (AND) decomposition and an active state with exclusive (OR) decomposition

In this example, state B and C.C2 or state B and C.C1 are active at the same time.



- One active state with exclusive (OR) decomposition

In this example, state B or state A.A1 or state A.A2 is active at any one time.



When a given state is active, all of its ancestor states are also active. See the section titled “Semantics of Active and Inactive States” on page 7-5 for more information.

What Are Combination States?

When a Stateflow diagram has states with parallel (AND) decomposition, multiple states can be active simultaneously. A combination state is a notational representation of those multiple states. For example, a Stateflow diagram could have two active states with parallel (AND) decomposition, A.B and X.Y. Using combination state notation, the activity of the Stateflow diagram is denoted by (A.B, X.Y).

A state is characterized by its label. The label consists of the name of the state optionally followed by a / character and additional keywords defined below. The label appears on the top left-hand corner of the state rectangle.

Labeling a State

The ? character is the default state label. State labels have this general format:

```
name/  
entry:  
during:  
exit:  
on event_name:
```

The keywords `entry` (shorthand `en`), `during` (shorthand `du`), `exit` (shorthand `ex`), and `on` identify actions associated with the state. You can specify multiple actions by separating them by any of these:

- Carriage return
- Semicolon
- Comma

Specify multiple `on event_name` actions for different events by adding multiple `on event_name` lines specifying unique values for `event_name`.

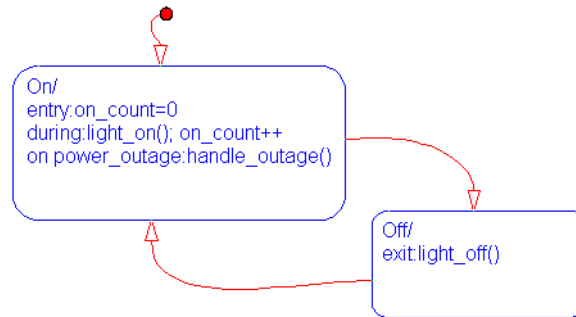
Each keyword is optional and positionally independent. You can specify none, some, or all of them. The colon after each keyword is required. The slash following the state name is optional as long as it is followed by a carriage return.

If you enter the name and slash followed directly by an action or actions (without the `entry` keyword), the action(s) is interpreted as entry action(s). This shorthand is useful if you are only specifying entry actions.

See the section titled “What Is an Action Language?” on page 6-41 for more information on the action language.

Example: Labeling a State

This example shows the state labeling formats and explains the components of the label.



Name. The *name* of the state forms the first part of the state label. Valid state names consist of alphanumeric characters and can include the `_` character, e.g., `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 as stored in the data dictionary consists of the text you entered as the label on the state, preceded by the hierarchy of its ancestor states separated by periods. States can have the same name appear on the graphical representation of the state, as long as the full names within the data dictionary are unique. The parser indicates an error if a state does not have a unique name entry in the data dictionary for that Stateflow diagram.

See the section titled “Example: Unique State Names” on page 6-14 for an example of uniquely named states.

In this example, the state names are `On` and `Off`.

Entry Action. In the example, state `On` has entry action `on_count=0`. The value of `on_count` is reset to 0 whenever state `On`’s entry action is executed.

See “Semantics of State Actions” on page 7-7 for information on how and when entry actions are executed.

During Action. In the example, state On has two during actions `light_on()` and `on_count++`. These actions are executed whenever state On's during action is executed.

See “Semantics of State Actions” on page 7-7 for information on how and when during actions are executed.

Exit Action. In the example, state Off has exit action `light_off()`. This action is executed whenever state Off's exit action is executed.

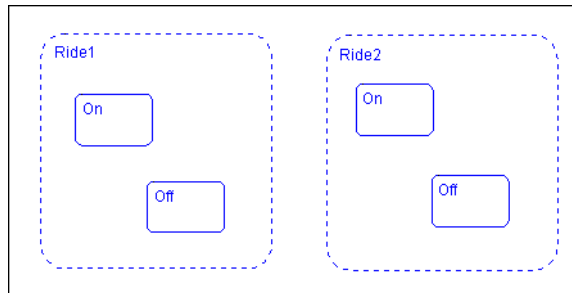
See “Semantics of State Actions” on page 7-7 for information on how and when exit actions are taken.

On Event_name Action. In the example, state Off has the on *event_name*, `power_outage`. When the event `power_outage` occurs, the action `handle_outage()` is executed.

See “Semantics of State Actions” on page 7-7 for information on how and when on *event_name* actions are taken.

Example: Unique State Names

This example shows how hierarchy supports unique naming of states.



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

- Ride1.On
- Ride1.Off

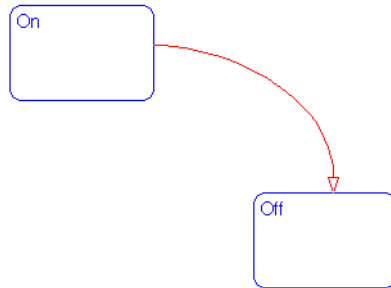
- Ride2.On
- Ride2.Off

Although the names On and Off are duplicated, the full names are unique because of the hierarchy of the Stateflow diagram. The example intentionally contains only states for simplification purposes.

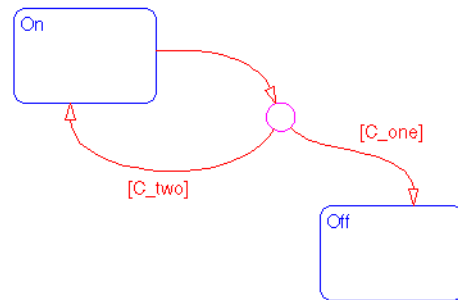
Transitions

What Is a Transition?

In most cases, a *transition* represents the passage of the system from a source object to a destination object. There are transitions between states. There are also transitions between junctions and states. A transition is represented by a line segment ending with an arrow drawn from a source object to the destination object. This is an example of a transition from a source state, On, to a destination state, Off.



Junctions divide a transition into segments. Each segment is evaluated in the process of determining the validity of the transition from a source to a destination. This is an example of a transition with segments.



A default transition is one special type of transition that has no source object.

Labeling a Transition

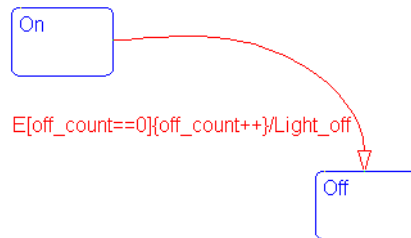
A transition is characterized by its *label*. The label can consist of an event, a condition, a condition action, and/or a transition action. The ? character is the default transition label. Transition labels have this general format.

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

Replace, as appropriate, your names for event, condition, condition action, and transition action. Each part of the label is optional.

Example: Transition Label

This example shows the format of a transition label.



Event. The specified *event* is what causes the transition to be taken, provided the condition, if specified, is true. Specifying an event is optional. 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 this example, the broadcast of event E, triggers the transition from On to Off, provided the condition, [off_count==0], is true.

Condition. A *condition* is a Boolean expression to specify that a transition occurs given that the specified expression is true. Enclose the condition in square brackets. See the section titled “Conditions” on page 6-60 for information on the condition notation.

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

Condition Action. The *condition action* is executed as soon as the condition, if specified, is evaluated as true and before the transition destination has been determined to be valid.

If the transition consists of multiple segments, the condition action is executed as soon as the condition, if specified, is evaluated as true and before the entire transition is determined as valid. Enclose the condition action in curly brackets. See the section titled “Action Language” on page 6-41 for more information on the action language.

If no condition is specified, the implied condition is always evaluated as true.

In this 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 as valid. Precede the transition action with a backslash. See the section titled “Action Language” on page 6-41 for more information on the action language.

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

What Is a Valid Transition?

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. These 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

Transition Label	Is Valid If:
Action only	Any event occurs
Not specified	Any event occurs

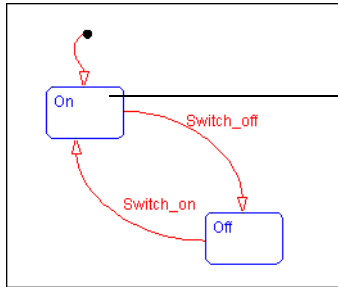
Types of Transitions

The notation supports these transition types:

- Transitions to and from exclusive (OR) states
See the sections titled “Example: Transitions to and from Exclusive (OR) States” on page 6-20 for an example of this type of transition.
- Transitions to and from junctions
See the section titled “Example: Transitions to and from Junctions” on page 6-20 for an example of this type of transition.
- Transitions to and from exclusive (OR) superstates
See the section titled “Example: Transitions to and from Exclusive OR Superstates” on page 6-21 for an example of this type of transition.
- Transitions from no source to an exclusive (OR) state (default transitions)
See the section titled “What Is a Default Transition?” on page 6-23 for examples of this type of transition.
- Inner state transitions
See the section titled “What Is an Inner Transition?” on page 6-26 for examples of this type of transition.
- Self loop transitions
See the section titled “What Is a Self Loop Transition?” on page 6-30 for examples of this type of transition.

Example: Transitions to and from Exclusive (OR) States

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



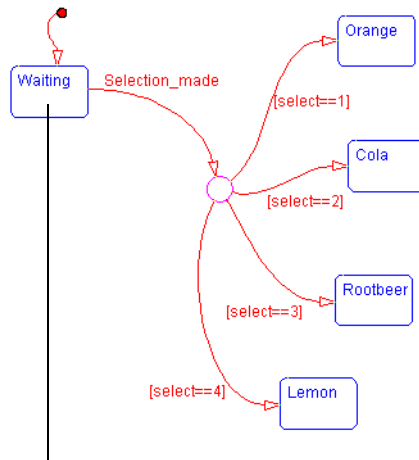
State On is initially active.

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 the section titled “Transitions to and from Exclusive (OR) States” on page 7-8 for more information on the semantics of this notation.

Example: Transitions to and from Junctions

This example shows transitions to and from a connective junction.



State Waiting is initially active.

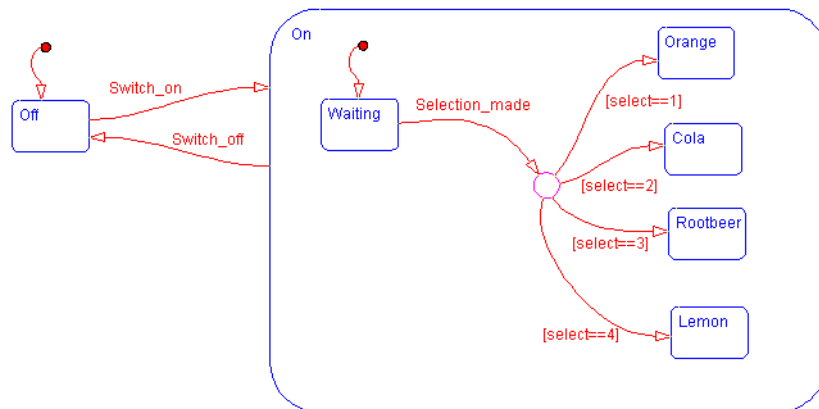
This is a Stateflow diagram of a soda machine. The Stateflow diagram is called when the external event Selection_made occurs. The Stateflow diagram

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 the section titled “Example: Transitions from a Common Source to Multiple Destinations” on page 7-36 for more information on the semantics of this notation.

Example: 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.

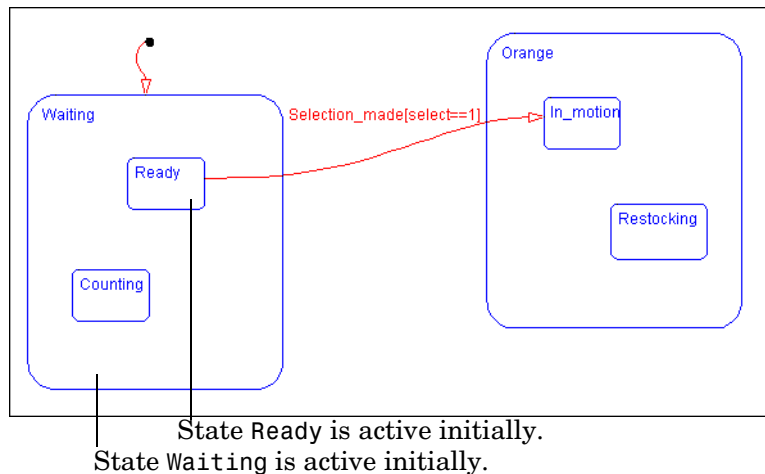
To be a valid transition to a superstate, the destination substate must be implicitly defined. By defining that the `Waiting` substate has a default transition, the destination substate is implicitly defined. This notation defines that the resultant transition is `Off`→`On.Waiting`.

The transition `On→Off` is valid when state `On` is active and event `Switch_off` occurs. When the `Switch_off` event occurs, no matter which of the substates of `On` is active, we want to transition to the `Off` state. This top-down approach supports the ability to simplify 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 the section titled “Default Transitions” on page 7-18 for more information on the semantics of this notation.

Example: Transitions to and from Substates

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



Two of the substates of the `On` superstate are further defined to be superstates of their own. The Stateflow diagram shows a transition from one OR substate to another OR substate. The transition `Waiting.Ready→Orange.In_motion` is valid when state `Waiting.Ready` is active and event `Selection_made` occurs, providing that the `select` variable equals one. 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 the section titled “Example: Transition from a Substate to a Substate” on page 7-11 for more information on the semantics of this notation.


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. 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. The default transition object is a transition with a destination but no source object.

Click on the **Default transition** button in the toolbar, and click on 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.

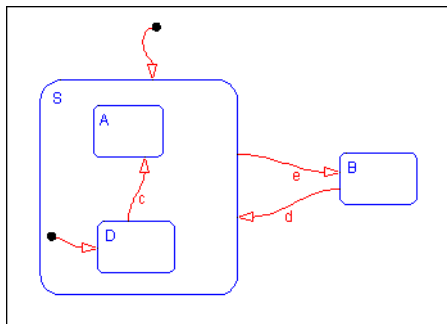
Labeling of Default Transitions

In some circumstances, you may want to label default transitions. You can label default transitions as you would other transitions. For example, you may want to specify that one state or another should become active depending upon the event that has occurred. In another situation, you may 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 will always be at least one valid default transition. Otherwise, the state machine can transition into an inconsistent state.

Example: Use of Default Transitions

This example shows a use of default transitions.



When the Stateflow diagram is first awakened, the default transition to superstate **S** defines that of states **S** and **B**; the transition to state **S** is valid. State **S** has two substates, **A** and **D**. Which substate does the system transfer to? It cannot transfer to both of them since **A** and **D** are not parallel (AND) states. Again, this kind of ambiguity is cleared up by defining a default transition to substate **D**.

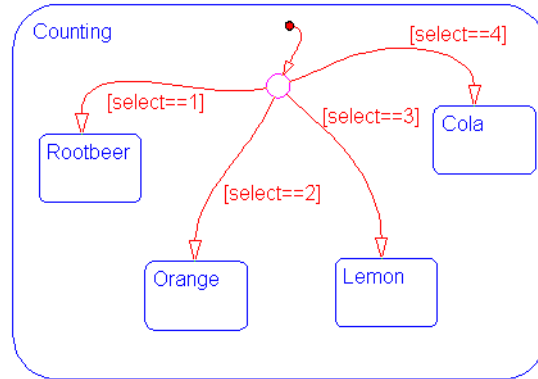
Suppose at a different execution point, the Stateflow diagram is awakened by the occurrence of event **d** and state **B** is active. The transition **B**→**S** is valid. When the system enters state **S**, it enters substate **D** because the default transition is defined.

See the section titled “Default Transitions” on page 7-18 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 become active. You can detect this situation at runtime by checking for state inconsistencies. See “Animation Controls” on page 10-8 for more information.

Example: Default Transition to a Junction

This example shows a default transition to a connective junction.

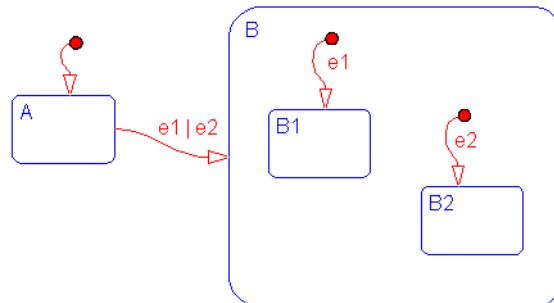


In this example, the default transition to the connective junction defines that upon entering the Counting state, the destination is determined by the condition on each transition segment.

See the section titled “Example: Default Transition to a Junction” on page 7-19 for more information on the semantics of this notation.

Example: Default Transition with a Label

This example shows a use of labeling 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→B1 is valid. If event e2 occurs, the transition A→B2 is valid.

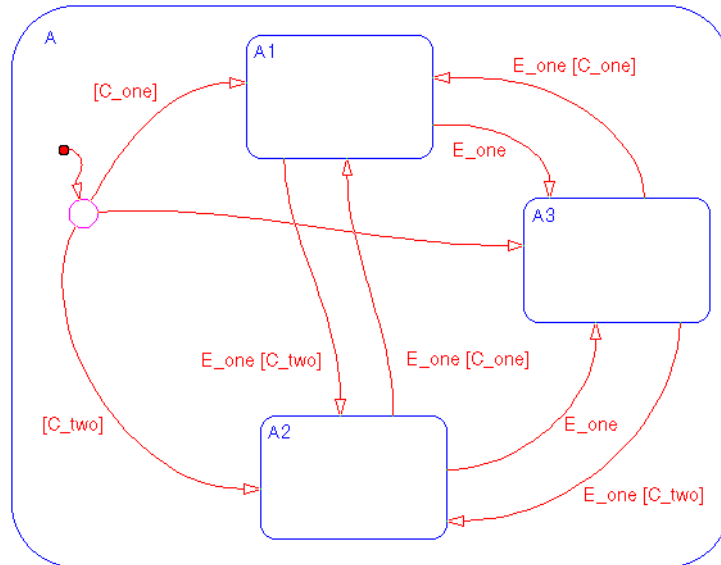
See the section titled “Example: Labeled Default Transitions” on page 7-21 for more information on the semantics of this notation.

What Is an Inner Transition?

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.

Example One: Before Using an Inner Transition

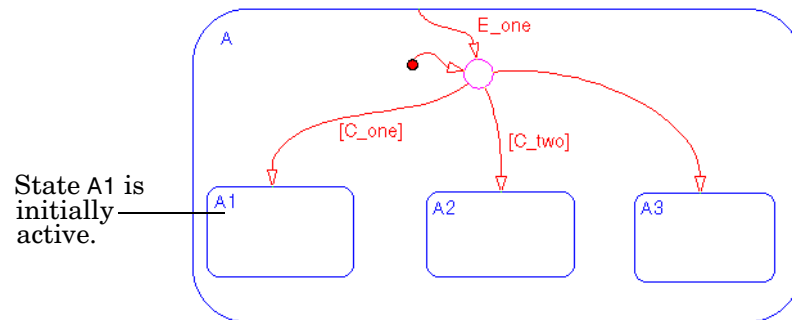
This is an example of a Stateflow diagram that could be simplified by using an inner transition.



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].

Example One: Inner Transition to a Connective Junction

This example shows a solution to the same problem (Example One) using an inner transition to a connective junction.



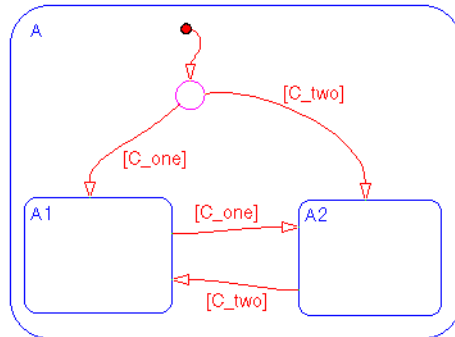
Any event occurs and awakens the Stateflow diagram. The default transition to the connective junction is valid. The destination of the transitions is determined by [C_one] and [C_two].

The Stateflow diagram is simplified by using an inner transition in place of the many transitions amongst all the states in the original example. If state A is already active, the inner transition is used to re-evaluate 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 the section titled “Example: Processing One Event with an Inner Transition to a Connective Junction” on page 7-26 for more information on the semantics of this notation.

Example Two: Before Using an Inner Transition

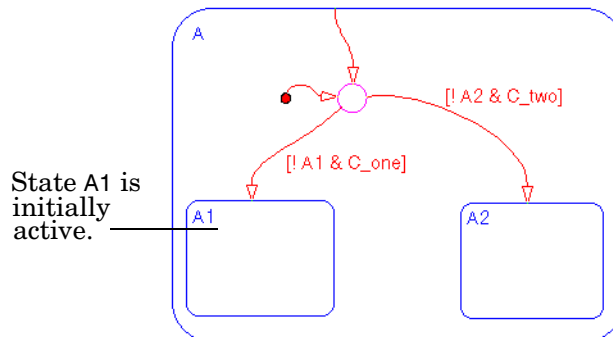
This is an example of a Stateflow diagram that could be simplified by using an inner transition.



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 the conditions [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. The transitions between A1 and A2 are also determined by [C_one] and [C_two].

Example Two: Inner Transition to a Connective Junction

This example shows a solution to the same problem (Example Two) 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

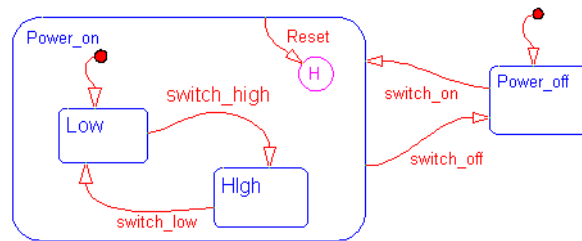
determined by [C_one] and [C_two]. (When the default transition path is followed, neither state A1 nor A2 is active; thus the truth of condition is based exclusively on [C_one] and [C_two].)

The Stateflow diagram is simplified by using an inner transition in place of the transitions between A.A1 and A.A2 in the original example. If state A is already active, the inner transition is used to re-evaluate which of the substates of state A is to be active. If state A1 is not active and [C_one] is true, the transition to A1 is valid. If state A2 is not active and [C_two] is true, the transition to A2 is valid. This solution is much simpler than the previous one.

See the section titled “Example: Processing One Event with an Inner Transition to a Connective Junction” on page 7-26 for more information on the semantics of this notation.

Example: Inner Transition to a History Junction

This example shows an inner transition to a history junction.



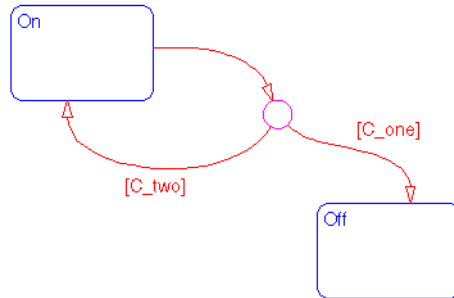
State `Power_on.High` is initially active. When event `Reset` occurs, the inner transition to the history junction is valid. Because the inner transition is valid, the currently active state, `Power_on.High`, will be exited. When the inner transition to the history junction is processed, the last active state, `Power_on.High`, becomes active (is re-entered). If `Power_on.Low` was active under the same circumstances, `Power_on.Low` would be exited and re-entered 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 the section titled “Example: Use of History Junctions” on page 6-39 for another example using a history junction.

See the section titled “Example: Inner Transition to a History Junction” on page 7-29 for more information on the semantics of this notation.

What Is a Self Loop Transition?

A transition segment from a state to a connective junction that has an outgoing transition segment from the connective junction back to itself is a self loop. This is an example of a self loop.



See these sections for examples of self loops:

- “Example: Connective Junction Special Case - Self Loop” on page 6-33
See the section titled “Example: Self Loop” on page 7-32 for information on the semantics of this notation.
- “Example: Connective Junction and For Loops” on page 6-34
See the section titled “Example: For Loop Construct” on page 7-33 for information on the semantics of this notation.

Connective Junctions

What Is a Connective Junction?

A connective junction is used to represent a decision point in the Stateflow diagram. The connective junction enables representation of different transition paths. Connective junctions are used to help represent:

- 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 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

See the section titled “Connective Junctions” on page 7-31 for a summary of the semantics of connective junctions.

What Is Flow Diagram Notation?

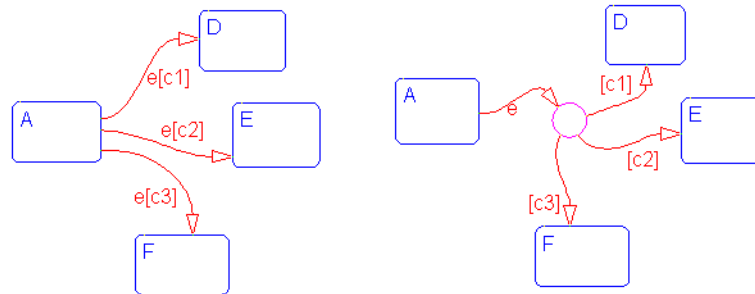
Flow diagram notation is essentially logic represented without the use of states. In some cases, using flow diagram notation is a closer representation of the system’s logic and avoids the use of unnecessary states. Flow diagram notation is an effective way to represent common code structures like for loops and if - then - else constructs. The use of flow diagram notation in a Stateflow diagram can produce more efficient code optimized for memory use. Reducing the number of states optimizes memory use.

Flow diagram notation is represented through combinations of self-loops to connective junctions, transitions to and from connective junctions, and inner transitions to connective junctions. The key to representing flow diagram notation is in the labeling of the transitions (specifically the use of action language).

Flow diagram notation and state-to-state transition notation seamlessly coexist in the same Stateflow diagram.

Example: Connective Junction with All Conditions Specified

When event e occurs, state A transfers to D , E , or F depending on which of the conditions $[c1]$, $[c2]$, or $[c3]$ is met. With the alternative representation, using a connective junction, the transition from A to the connective junction occurs first, provided the event has occurred. A destination state is then determined based on which of the conditions $[c1]$, $[c2]$, or $[c3]$ is satisfied. The transition from the source state to the connective junction is labeled by the event, and those from the connective junction to the destination states by the conditions. No event is applicable in a transition from a connective junction to a destination state.

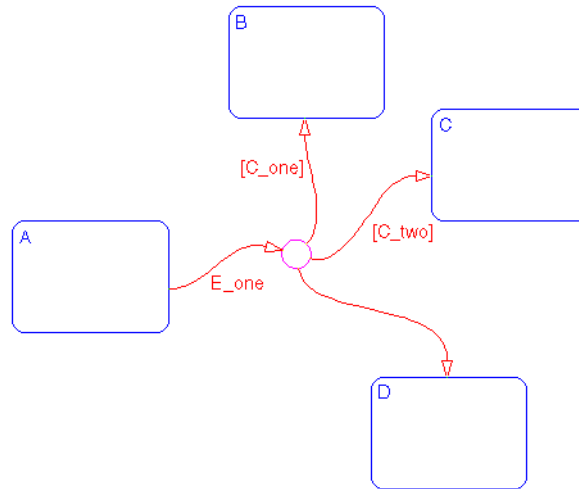


See the section titled “Example: If-Then-Else Decision Construct” on page 7-31 for information on the semantics of this notation.

Example: Connective Junction with One Unconditional Transition

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



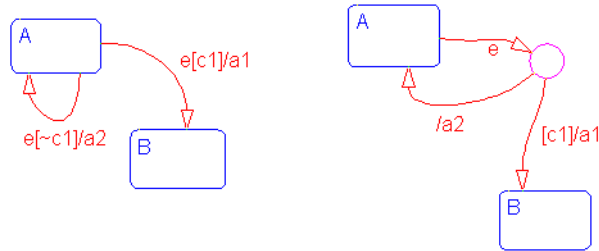
See the section titled “Example: If-Then-Else Decision Construct” on page 7-31 for information on the semantics of this notation.

Example: Connective Junction Special Case - Self Loop

In some situations, the transition event occurs, but the condition is not met. The transition cannot be taken, but an action is generated. You can represent this situation by using a connective junction or a self loop (transition from state to itself).

In state A, event e occurs. If condition $[c1]$ is met, transition $A \rightarrow B$ is taken, generating action $a1$. The transition $A \rightarrow A$ is valid if event e occurs and $[c1]$ is not true. In this self loop, the system exits and re-enters state A, and executes action $a2$. An alternative representation using a connective junction is shown.

The two representations are equivalent; in the one that uses a connective junction, it is not necessary to specify condition $[\sim c1]$ explicitly, as it is implied.



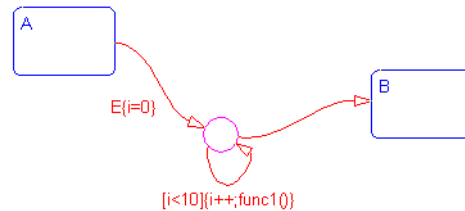
See the section titled “Example: Self Loop” on page 7-32 for information on the semantics of this notation.

Example: Connective Junction and For Loops

This example shows a combination of flow diagram notation and state transition notation. Self loops 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 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 to the connective junction effectively execute a for loop (for i values 0 to 9 execute $func1()$). The for loop is executed outside of the context of a state. The remainder of the path is

evaluated. Since there are no conditions, the transition completes at the destination, state B.



See the section titled “Example: For Loop Construct” on page 7-33 for information on the semantics of this notation.

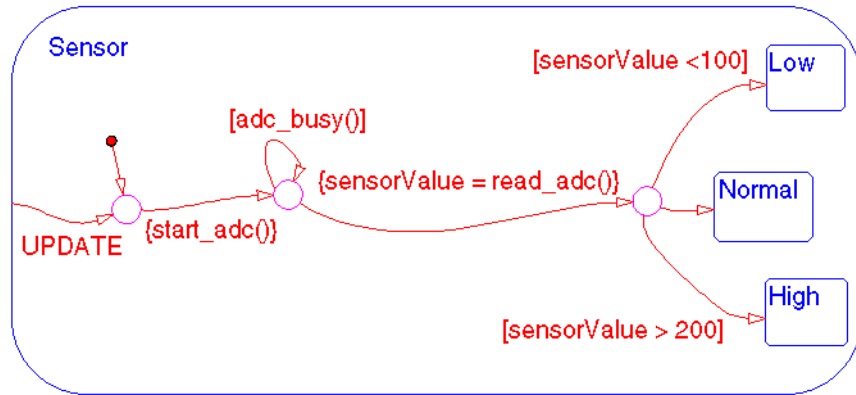
Example: Flow Diagram Notation

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 is used to introduce the delay needed for the ADC reading to settle. The delay could have been represented by using 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

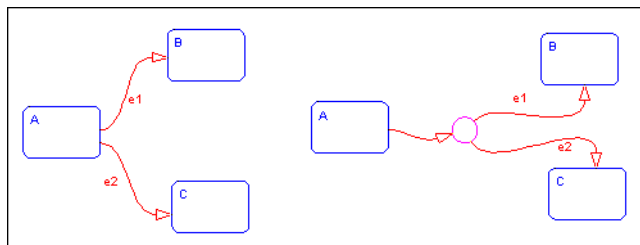
[sensorValue >200] is true, the state Sensor.High is the destination. Otherwise, state Sensor.Normal is the destination state.



See the section titled “Example: Flow Diagram Notation” on page 7-34 for information on the semantics of this notation.

Example: Connective Junction from a Common Source to Multiple Destinations

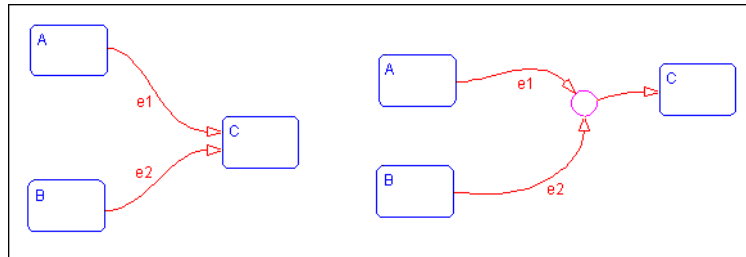
Transitions A→B and A→C share a common source state A. An alternative representation uses one arrow from A to a connective junction, and multiple arrows labeled by events from the junction to the destination states B and C.



See the section titled “Example: Transitions from a Common Source to Multiple Destinations” on page 7-36 for information on the semantics of this notation.

Example: Connective Junction Common Destination

Transitions $A \rightarrow C$ and $B \rightarrow C$ share a common destination state C . An alternative representation uses one transition from A to a connective junction, one transition from B to the same connective junction, and one transition from the connective junction to the destination state, C .



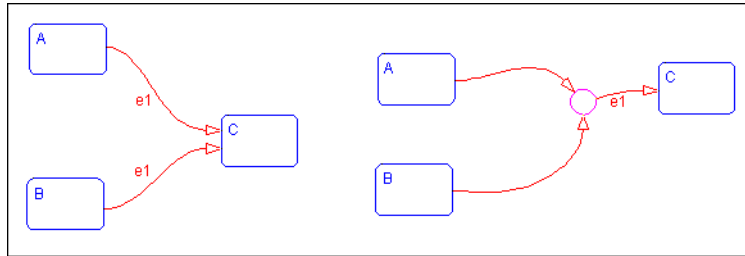
See the section titled “Example: Transitions from a Common Source to Multiple Destinations” on page 7-36 for information on the semantics of this notation.

Example: Connective Junction Common Events

Suppose, for example, that when event e_1 occurs, the system, whether it is in state A or B , will transfer to state C . Suppose that transitions $A \rightarrow C$ and $B \rightarrow C$ are triggered by the same event e_1 , so that both destination state and trigger event are common between 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 the section titled “Example: Transitions from a Source to a Destination Based on a Common Event” on page 7-38 for information on the semantics of this notation.

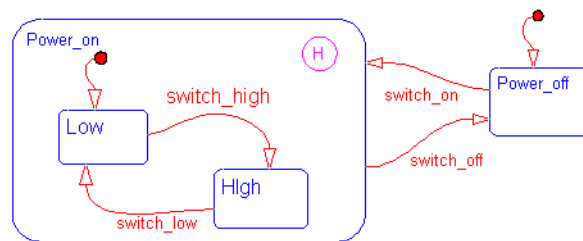
History

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.

Example: Use of History Junctions

This example shows a use of history junctions.



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` will be entered because it has a default transition. At some point afterwards, 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 the section titled “Example: Default Transition and a History Junction” on page 7-20 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 re-entered.

See the section titled “Example: Inner Transition to a History Junction” on page 6-29 for an example of this notation.

See the section titled “Example: Inner Transition to a History Junction” on page 7-29 for more information on the semantics of this notation.

Action Language

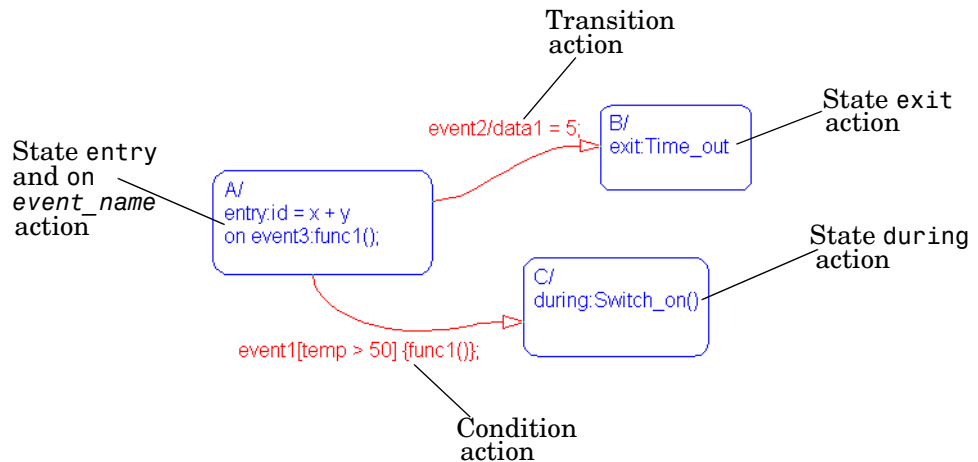
What Is an Action Language?

You sometimes want actions to take place as part of Stateflow diagram execution. The action can be executed as part of a transition from one state to another, or it can depend on the activity status of a state. Transitions can have condition actions and transition actions. States can have entry, during, exit, and, on *event_name* actions.

An action can be a function call, an event to be broadcast, a variable to be assigned a value, etc. The *action language* defines the categories of actions you can specify and their associated notations. Violations of the action language notation are flagged as errors by the parser. This section describes the action language notation rules.

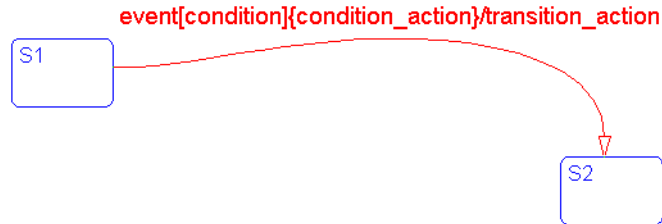
Objects with Actions

This Stateflow diagram shows examples of the possible transition and state actions.



Transition Action Notation

Actions can be associated with transitions via the transition's label. The general format of a transition label is shown below.



When the event occurs, the transition is evaluated. The condition action is executed as soon as the condition is evaluated as true and before the transition destination has been determined to be valid. Enclose the condition action in curly brackets. Specifying a transition action means that the action is executed when the transition is taken, provided the condition, if specified, is true.

State Action Notation

Actions can be associated with states via the state's label by defining entry, during, exit, and on *event_name* keywords. The general format of a state label is shown below.



The / following the state name is optional. See the section “Semantics of State Actions” on page 7-7 for information on the semantics of state actions. See the examples in Chapter 7, “Semantics,” for examples of the semantics of state actions.

Action Language Components

The action language defines usage rules for these components:

- Bit operations
- Binary operations
- Unary operations
- Assignment operations
- User-written function calls
- `m1()` function
- `m1.` name space operator
- Data and event arguments
- Arrays
- Event broadcasting
- Directed event broadcasting
- Conditions
- Time, `t`
- Literals
- Continuation symbols
- Comments
- Semicolon

Bit Operations

You can enable C-like bit operations. See “Enable C-Like Bit Operations” on page 9-13 for more information. If you have `bitops` enabled, some of the logical binary operators and unary operators are interpreted as bitwise operators. See “Binary Operations” on page 6-43 and “Unary Operations” on page 6-46 for specific interpretations.

Binary Operations

Binary operations fall into these categories:

- Numerical

Example	Description
<code>a + b</code>	Addition of two operands
<code>a - b</code>	Subtraction of one operand from the other
<code>a * b</code>	Multiplication of two operands
<code>a / b</code>	Division of one operand by the other
<code>a %% b</code>	Modulus

- Logical (the default setting; bit operations not enabled)

Example	Description
<code>a == b</code>	Comparison of equality of two operands
<code>a & b</code>	Logical AND of two operands
<code>a && b</code>	
<code>a b</code>	Logical OR of two operands
<code>a b</code>	
<code>a ~= b</code>	Comparison of inequality of two operands
<code>a != b</code>	
<code>a > b</code>	Comparison of the first operand greater than the second operand
<code>a < b</code>	Comparison of the first operand less than the second operand
<code>a >= b</code>	Comparison of the first operand greater than or equal to the second operand
<code>a <= b</code>	Comparison of the first operand less than or equal to the second operand

- Logical (bit operations enabled)

Example	Description
<code>a == b</code>	Comparison of equality of two operands
<code>a && b</code>	Logical AND of two operands
<code>a & b</code>	Bitwise AND of two operands
<code>a b</code>	Logical OR of two operands
<code>a b</code>	Bitwise OR of two operands
<code>a != b</code>	Comparison of inequality of two operands
<code>a != b</code>	
<code>a <> b</code>	
<code>a > b</code>	Comparison of the first operand greater than the second operand
<code>a < b</code>	Comparison of the first operand less than the second operand
<code>a >= b</code>	Comparison of the first operand greater than or equal to the second operand
<code>a <= b</code>	Comparison of the first operand less than or equal to the second operand
<code>a ^ b</code>	Bitwise XOR of two operands

Unary Operations

These unary operations are supported: \sim , $!$, $-$.

Example	Description
$\sim a$	Logical not of a Complement of a (if bitops is enabled)
$!a$	Logical not of a
$-a$	Negative of a

Unary Actions

These unary actions are supported.

Example	Description
$a++$	Increment a
$a--$	Decrement a

Assignment Operations

These assignment operations are supported.

Example	Description
$a = \text{expression}$	Simple assignment
$a += \text{expression}$	Equivalent to $a = a + \text{expression}$
$a -= \text{expression}$	Equivalent to $a = a - \text{expression}$
$a *= \text{expression}$	Equivalent to $a = a * \text{expression}$
$a /= \text{expression}$	Equivalent to $a = a / \text{expression}$

These additional assignment operations are supported when bit operations are enabled.

Example	Description
<code>a = expression</code>	Equivalent to <code>a = a expression</code> (bit operation)
<code>a &= expression</code>	Equivalent to <code>a = a & expression</code> (bit operation)
<code>a ^= expression</code>	Equivalent to <code>a = a ^ expression</code> (bit operation)

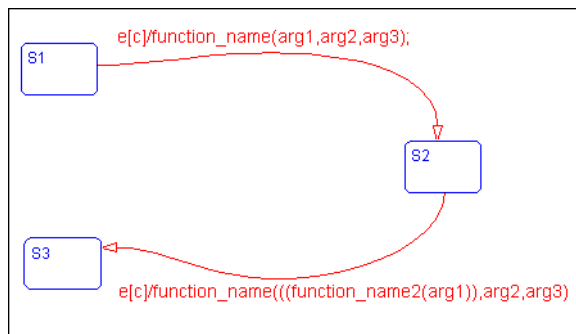
User-Written Functions

You can specify calls to user-written functions in the action language. These guidelines apply to user-written function calls:

- Define a function by its name, any arguments in parenthesis, and an optional semicolon.
- String parameters to user-written functions are passed between single quotes. For example, `func('string')`.
- You can nest function calls.

Example: Function Call Transition Action

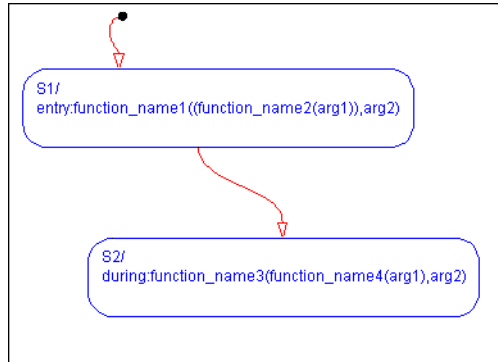
These are example formats of function calls using transition action notation.



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.

Example: Function Call State Action

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.

ml() Functions

You can specify calls to MATLAB functions that return scalars (of type double) in the action language.

ml() Function Format

The format of the `ml()` function is

```
ml(evalString, arg1, arg2, arg3, ...);
```

where the return value is scalar (of type double).

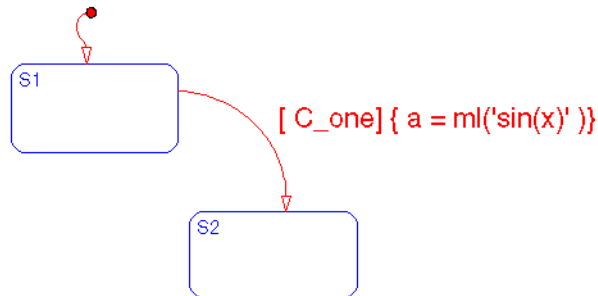
If the result returned is:

- A vector, then the first element is returned.
- A void, then an appropriate format must be used (an assignment statement cannot be used).
- A string, a structure, or a cell array, then the behavior is undefined.

`evalString` is a string that is evaluated in the MATLAB workspace with formatted substitutions of `arg1`, `arg2`, `arg3`, etc.

Example One: `ml()` Function Call

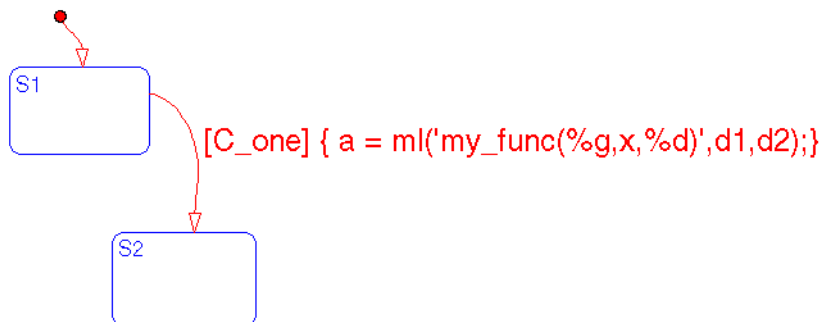
This is an example of an `ml()` function call as part of a condition action.



If S1 is active, an event occurs, and if `[c_one]` is true, the expression `sin(x)` is evaluated in the MATLAB workspace and the return value assigned to `a`. (`x` must be a variable in the MATLAB workspace and `a` is a data object in the Stateflow diagram). The result of the evaluation must be a scalar. If `x` is not defined in the MATLAB workspace, a runtime error is generated.

Example Two: `ml()` Function Call

This is an example of a `ml()` function call that passes Stateflow data as arguments. Notice the use of format specifiers `%g` and `%d` as are used in the C language function `printf`.



Use Guidelines

These guidelines apply to `m1()` functions:

- The first argument must be a string.
- If there are multiple arguments, ensure that the number and types of format specifiers (`%g`, `%d`, etc.) match the actual number and types of the arguments. These format specifiers are the same as those used in the C function `printf`.
- A scalar (of type double) is returned.
- `m1()` function calls can be nested.
- Calls to `m1()` functions should be avoided if you plan to build an RTW target that includes code from Stateflow Coder.

MATLAB Name Space Operator

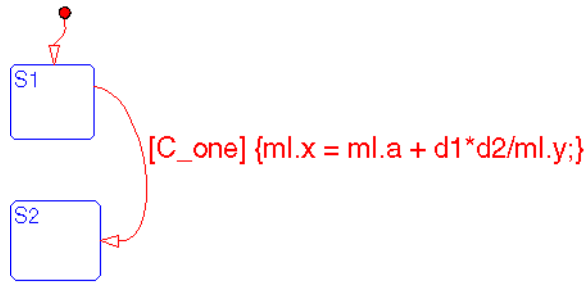
The MATLAB name space operator, `m1`, is used to get and set variables in the MATLAB workspace. The `m1` operator can also be used to access MATLAB functions that operate on scalars in a convenient format.

Use the notation, `a = m1.func_name();`, to call a MATLAB function that does not accept any arguments. Omission of the empty brackets causes a search for a variable of the name specified. The variable will not be found and a runtime error is encountered during simulation.

Use of the `m1` name space operator should be avoided if you plan to build a Real-Time Workshop target that includes code from Stateflow Coder.

Example: Using the ml Operator to Access MATLAB Workspace Variables

This is an example of using the ml operator to get and set variables in the MATLAB workspace.



These data objects are defined in the Stateflow diagram:

- d1 and d2 are **Local** data objects
- a, x, and y must be defined in the MATLAB workspace prior to starting the simulation; otherwise a runtime error is generated at the execution time of the transition

The values of a and y are accessed in the MATLAB workspace and used in the expression with the **Local** data objects d1 and d2. The result of the expression is assigned to the MATLAB workspace variable x. If x does not exist, it is automatically created in the MATLAB workspace.

Example: Using the ml Operator to Access MATLAB Functions

This is an example of using the ml operator to access MATLAB functions.



These data objects are defined:

- d1 and d2 are **Local** data objects defined in the Stateflow diagram
- x is assumed to be a two dimensional array in the MATLAB workspace
- y is assumed to be a MATLAB workspace vector.
- z is assumed to be a MATLAB workspace scalar variable.

x, y, and z must be defined in the MATLAB workspace prior to starting the simulation; otherwise a runtime error is generated at the execution time of the transition.

A MATLAB function named `my_func` is called with these arguments:

- 1 `x(1,3)`
- 2 `y(3)`
- 3 `z`
- 4 `d1`
- 5 `d2`
- 6 `string 'abcdefgh'`

The result of `my_func()` (if it is a scalar) is assigned to element (5, 6, 7) of a multidimensional matrix `v` in the MATLAB workspace. If `v` does not exist prior to the execution of this statement, then it is automatically created by MATLAB workspace.

If `my_func()` returns a vector, the first element is assigned to `v(5,6,7)`. If it is a structure, a cell array, or a string, the result is undefined.

Choosing Between the `m1()` Function and the `m1` Name Space Operator

It is recommended to use the `m1` name space operator wherever possible. The `m1` name space operator is faster and more robust than the `m1()` function. If you need to work with MATLAB matrices instead of scalars, then use the `m1()` function.

In this example, the `m1()` function must be used to specify an array argument.

```
a = m1('my_function([1:4],%g)',d1);
```

`x` is a MATLAB workspace matrix. `my_function` is a MATLAB function that expects a vector as its first argument and a scalar as a second argument.

Data and Event Arguments

Unqualified data and event objects are assumed to be defined at the same level in the hierarchy as the reference to them in the action language. Stateflow will attempt to resolve the object name by searching up the hierarchy. If the data or event object is parented elsewhere in the hierarchy, you need to define the hierarchy path explicitly.

Arrays

You can use arrays in the action language.

Examples of Array Assignments

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 of 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;
```

Using Arrays with Simulink

Array data objects that have a scope of **Input from Simulink** or **Output to Simulink** are constrained to one dimension. Use a single scalar value for the **Sizes** property of these arrays.

Arrays and Custom Code

The action language provides the same syntax for Stateflow arrays and custom code arrays. 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.

Pointer and Address Operators

The Stateflow action language includes address and pointer operators. 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.

Syntax Examples

These 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;
```

These examples show syntax that is valid for use with both custom code variables and Stateflow variables.

```
varPtr = &var;  
  
ptr = &varArray[<expression>];  
  
*(&var) = <expression>;  
  
function(&varA, &varB, &varC);  
  
function(&sf.varArray[<expr>]);
```

Syntax Error Detection

The action language parser uses a relaxed set of restrictions. As a result, many syntax errors will not be trapped until compilation.

Hexadecimal Notation

The action language supports C style hexadecimal notation (for example, 0xFF). You can use hexadecimal values wherever you can use decimal values.

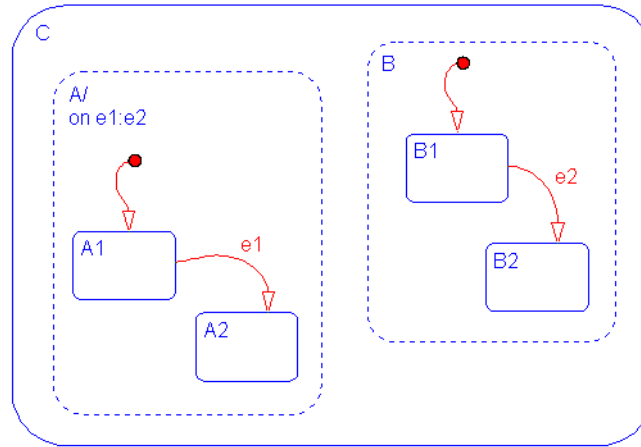
Event Broadcasting

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 "Event Parent" on page 3-5 for more information.

Broadcasting an event in the action language is most useful as a means of synchronization amongst AND (parallel) states. Recursive event broadcasts can lead to definition of cyclic behavior. Cyclic behavior can be detected only during simulation.

Example: Event Broadcast State Action

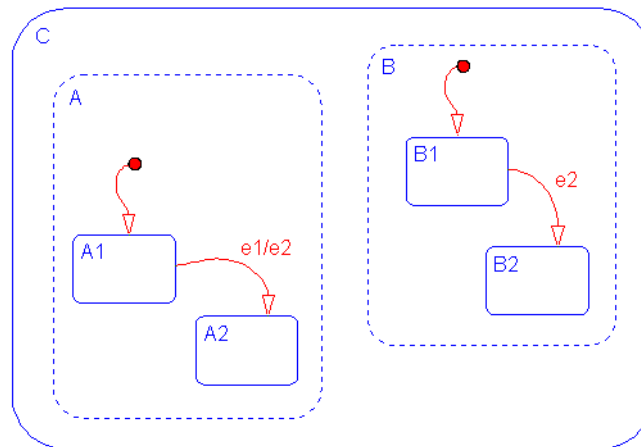
This is an example of the event broadcast state action notation.



See the section titled “Example: Event Broadcast State Action” on page 7-42 for information on the semantics of this notation.

Example: Event Broadcast Transition Action

This is an example of the event broadcast transition action notation.



See the section titled “Example: Event Broadcast Transition Action (Nested Event Broadcast)” on page 7-46 for information on the semantics of this notation.

Directed Event Broadcasting

You can specify a directed event broadcast in the action language. Using a directed event broadcast, you can broadcast a specific event to a specific receiver state. Directed event broadcasting is a more efficient means of synchronization amongst AND (parallel) 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.

The format of the directed broadcast is

```
send(event_name, state_name)
```

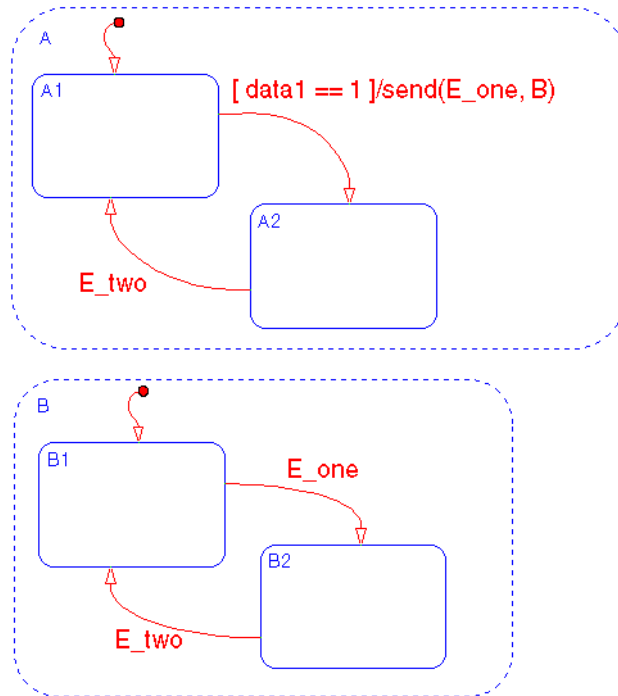
where `event_name` is broadcast to `state_name` (and any offspring of that state in the hierarchy). The `state_name` argument can include a full hierarchy path. For example,

```
send(event_name, chart_name.state_name1.state_name2)
```

The `state_name` specified must be active at the time the `send` is executed for the `state_name` to receive and potentially act on the directed event broadcast.

Example: Directed Event Broadcast Using send

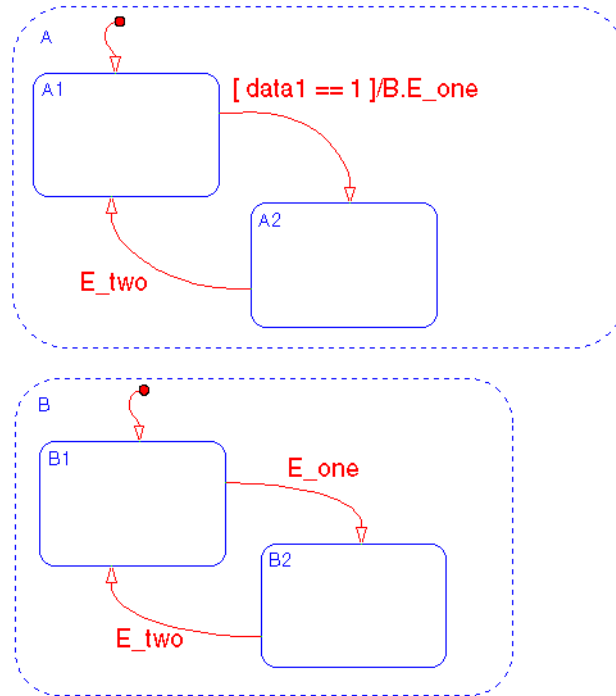
This is an example of a directed event broadcast using the `send(event_name, state_name)` transition action as a transition action.



See “Example: Directed Event Broadcasting Using Qualified Event Names” on page 7-56 for information on the semantics of this notation.

Example: Directed Event Broadcast Using Qualified Event Names

This example shows an alternate way to specify the same directed event broadcast as the previous example using a qualified event name.



See “Example: Directed Event Broadcasting Using Qualified Event Names” on page 7-56 for information on the semantics of this notation.

Conditions

You sometimes want transitions or actions associated with transitions to take place only if a certain condition is true. Conditions are placed within `[]`. These are some guidelines for defining conditions:

- The expression must be a Boolean expression of some kind. The condition must evaluate to either true (1) or false(0).

- The expression can consist of:
 - Boolean operators, which make comparisons between data and numeric values
 - Any 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.
- 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.

Time Symbol

You can use the letter `t` to represent absolute time in simulation targets. This simulation time is inherited from Simulink.

For example, the condition `[t - On_time > 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`.

The meaning of `t` for nonsimulation targets is undefined since it is dependent upon the specific application and target hardware.

Literals

Place action language you want the parser to ignore but you want to appear as entered in the generated code within `$` characters. 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.

Continuation Symbols

Enter the characters ... at the end of a line to indicate the expression continues on the next line.

Comments

These comment formats are supported:

- % MATLAB comment line
- // C++ comment line
- /* C comment line */

Use of the Semicolon

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.

Using the Notation Efficiently

Designing for Efficiency

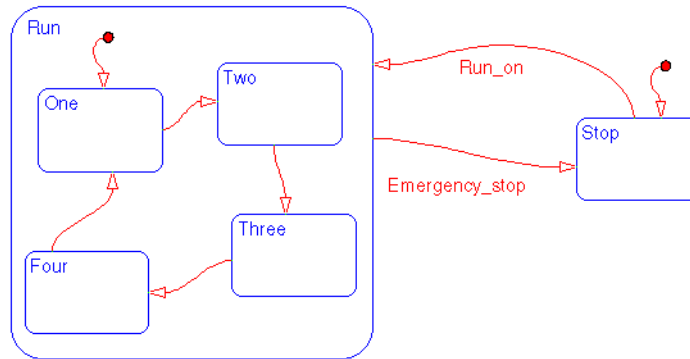
The Stateflow notation supports many different ways of representing desired system behavior. The representation you choose directly affects the efficiency of the generated code. You can use the notation to optimize for these code attributes:

- Speed
- Code size
- Memory use

Optimizing for Speed

If execution speed is the most important efficiency criterion for the generated code, take advantage of the top-down processing of events. Designing the Stateflow diagram so that execution of speed-critical actions occurs at the highest hierarchy level possible will produce speed-efficient code. Taking advantage of the ability to specify an event's parent (within the hierarchy) provides speed optimizations. You can make sure that an event's broadcast scope is as specific as possible to avoid unnecessary event broadcasting. See "Defining Events" on page 3-2 for more information. You can also use the send action language function to further qualify the receiver of an event broadcast (directed event broadcasting). See "Directed Event Broadcasting" on page 6-58 for more information.

This example Stateflow diagram shows the use of top-down event processing to optimize the event-handling speed.



If the emergency stop button is pushed, the event `Emergency_stop` occurs and the system should transition to the `Stop` state as quickly as possible. By drawing the `Emergency_stop` transition from the `Run` superstate border to the `Stop` state, the transition is evaluated at the highest level in the hierarchy. Using this notation, if any of `Run` substates are active and the `Emergency_stop` event occurs, the system transitions to the `Stop` state. The alternative to this notation is to have individual transitions (originating from a lower level in the hierarchy) from each of `Run`'s substates. The event handling of the alternative is less speed efficient.

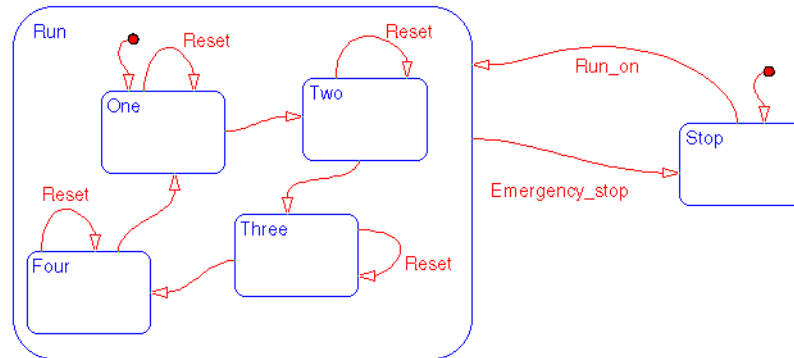
See “Example: Directed Event Broadcast Using `send`” on page 6-59 and “Example: Directed Event Broadcast Using Qualified Event Names” on page 6-60 for examples using the `send` action language function.

Optimizing for Code Size

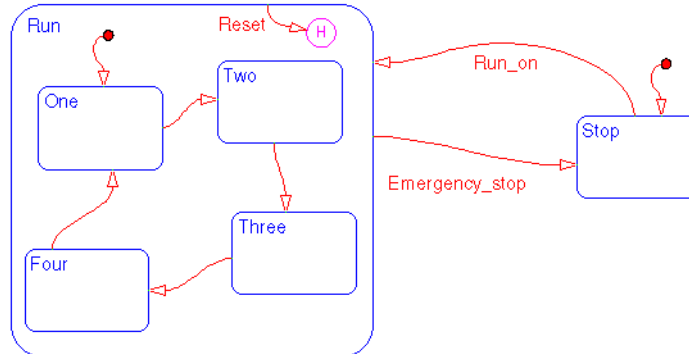
Simplifying Stateflow diagrams results in less generated code. The fewer transitions, the simpler the Stateflow diagram is. Use of hierarchy, history junctions, and inner transitions reduces the number of transitions.

In this example, if event `Reset` occurs and any substate of `Run` is active, you want to exit and re-enter that same substate. For example, when the substate

is re-entered, its entry action is executed. This example uses many transitions to represent the logic.



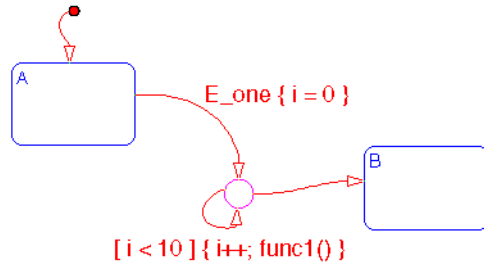
This is an equivalent, simpler Stateflow diagram that uses an inner transition to a history junction to optimize for code size.



Optimizing for Memory Use

The use of flow diagram notation in a Stateflow diagram can produce efficient code optimized for memory use. In some cases, specifying actions using flow diagram notation is a closer representation of the system's logic and avoids the use of unnecessary states. Reducing the number of states optimizes for memory use.

This example shows the use of flow diagram notation to represent a for loop.



The transition from state A to state B is divided into segments. Two of the segments are labeled with condition actions { i = 0 } and { i++; func1() }. Using the Connective junction and condition action notation in this manner is a closer representation of a for loop than could be otherwise represented using states. The elimination of unnecessary states also optimizes the use of memory.

For data that does not need to be persistent, use of **Temporary** data objects provides a memory-use optimization. Defining i to be a **Temporary** data object (parented by the chart) also improves memory use. See “Data Scope” on page 3-18 for more information on **Temporary** data objects.

Semantics

Overview	7-2
Event-Driven Effects on Semantics	7-5
Transitions to and from Exclusive (OR) States	7-8
Condition Actions	7-13
Default Transitions	7-18
Inner Transitions	7-23
Connective Junctions	7-31
Event Actions	7-40
Parallel (AND) States	7-42
Directed Event Broadcasting	7-54
Execution Order	7-58
Semantic Rules Summary	7-62

Overview

What Are Semantics?

Semantics describe how the notation is interpreted and implemented. 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.

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.

Stateflow semantics consist of rules for:

- Event broadcasting
- Processing states
- Processing transitions
- Taking transition paths

The details of Stateflow semantics are described largely by examples in this chapter. The examples cover a range of various notations and combinations of state and transition actions.

See “Semantic Rules Summary” on page 7-62 for a summary of the semantics.

List of Semantic Examples

This is a list of the semantic examples provided in this chapter.

Contents of Section “Transitions to and from Exclusive (OR) States” on page 7-8

- “Example: Processing of One Event” on page 7-8
- “Example: Processing of a Second Event” on page 7-9
- “Example: Processing of a Third Event” on page 7-10
- “Example: Transition from a Substate to a Substate” on page 7-11

Contents of Section “Condition Actions” on page 7-13

- “Example: Actions Specified as Condition Actions” on page 7-13
- “Example: Actions Specified as Condition and Transition Actions” on page 7-14
- “Example: Using Condition Actions in For Loop Construct” on page 7-15
- “Example: Using Condition Actions to Broadcast Events to Parallel (AND) States” on page 7-16
- “Example: Cyclic Behavior to Avoid When Using Condition Actions” on page 7-17

Contents of Section “Default Transitions” on page 7-18

- “Example: Default Transition in an Exclusive (OR) Decomposition” on page 7-18
- “Example: Default Transition to a Junction” on page 7-19
- “Example: Default Transition and a History Junction” on page 7-20
- “Example: Labeled Default Transitions” on page 7-21

Contents of Section “Inner Transitions” on page 7-23

- “Example: Processing One Event Within an Exclusive (OR) State” on page 7-23
- “Example: Processing a Second Event Within an Exclusive (OR) State” on page 7-24
- “Example: Processing a Third Event Within an Exclusive (OR) State” on page 7-25
- “Example: Processing One Event with an Inner Transition to a Connective Junction” on page 7-26
- “Example: Processing a Second Event with an Inner Transition to a Connective Junction” on page 7-27
- “Example: Inner Transition to a History Junction” on page 7-29

Contents of Section “Connective Junctions” on page 7-31

- “Example: If-Then-Else Decision Construct” on page 7-31
- “Example: Self Loop” on page 7-32
- “Example: For Loop Construct” on page 7-33
- “Example: Flow Diagram Notation” on page 7-34
- “Example: Transitions from a Common Source to Multiple Destinations” on page 7-36
- “Example: Transitions from Multiple Sources to a Common Destination” on page 7-37
- “Example: Transitions from a Source to a Destination Based on a Common Event” on page 7-38

Contents of Section “Event Actions” on page 7-40

- “Example: Event Actions and Superstates” on page 7-40

Contents of Section “Parallel (AND) States” on page 7-42

- “Example: Event Broadcast State Action” on page 7-42
- “Example: Event Broadcast Transition Action (Nested Event Broadcast)” on page 7-46
- “Example: Event Broadcast Condition Action” on page 7-50

Contents of Section “Directed Event Broadcasting” on page 7-54

- “Example: Directed Event Broadcast Using send” on page 7-54
- “Example: Directed Event Broadcasting Using Qualified Event Names” on page 7-56

Event-Driven Effects on Semantics

What Does Event-Driven Mean?

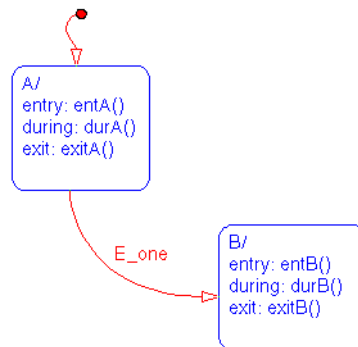
The Stateflow diagram executes only when an event occurs; an event occurs and the Stateflow diagram is awakened to respond to the event. Exactly what executes depends on the circumstances when the event occurs. Actions that are to take place based on an event are atomic to that event. Once an action is initiated, it is completed unless interrupted by an early return.

Top-Down Processing of Events

When an event occurs, it is processed from the top or root of the Stateflow diagram down through the hierarchy of the Stateflow 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.

Semantics of Active and Inactive States

This example shows the semantics of active and inactive states.

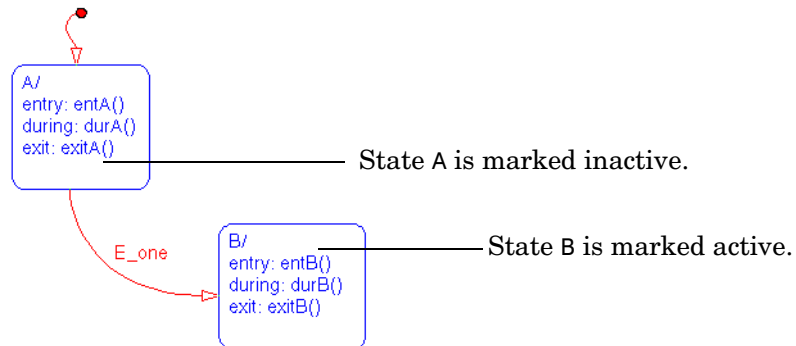


Initially the Stateflow diagram is asleep and both states are inactive. An event occurs and the Stateflow diagram is awakened. This is the semantic sequence:

- 1 The Stateflow diagram root checks to see if there is a valid transition as a result of the event. A valid default transition to state A is detected.
- 2 State A is marked active.
- 3 State A entry actions execute and complete (`entA()`).
- 4 The Stateflow diagram goes back to sleep waiting to be awakened by another event.

Event `E_one` occurs and the Stateflow diagram is awakened. State A is active. This is the semantic sequence:

- 1 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.
- 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 The Stateflow diagram goes back to sleep, to be awakened by the next event.



Semantics of State Actions

An entry action is executed as a result of any transition into the state. The state is marked active before its entry action is executed and completed.

A during action executes to completion when that state is active and an event occurs that does not result in an exit from 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.

An exit action is executed as a result of any transition out of the state. The state is marked inactive after the exit action has executed and completed.

Semantics of Transitions

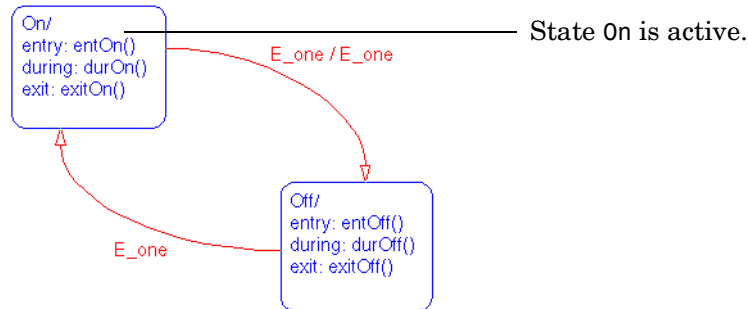
Transitions play a large role in defining the animation or execution of a system. Transitions have sources and destinations; thus any actions associated with the sources or destinations are related to the transition that joins them. The type of the source and destination is equally important to define the semantics.

The examples provided in this chapter show how the semantics are defined.

Transitions to and from Exclusive (OR) States

Example: Processing of One Event

This example shows the semantics of a simple transition focusing on the implications of states being active or inactive.



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:

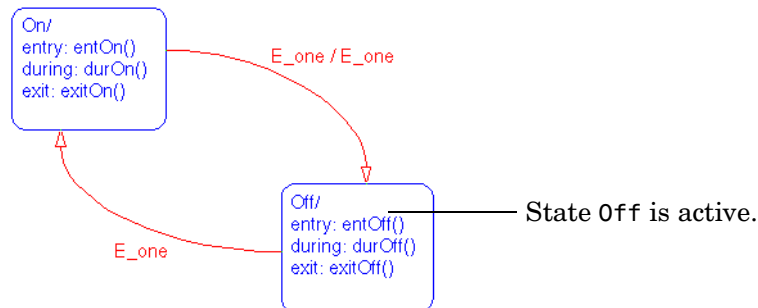
- 1 The Stateflow diagram root checks to see if there is a valid transition as a result of **E_one**. A valid transition from state On to state Off is detected.
- 2 State On exit actions execute and complete (**ExitOn()**).
- 3 State On is marked inactive.
- 4 The event **E_one** is broadcast as the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

- 6 State Off entry actions execute and complete (`entOff()`).
- 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` was active.

Example: Processing of a Second Event

Using the same example, what happens when the next event, `E_one`, occurs?



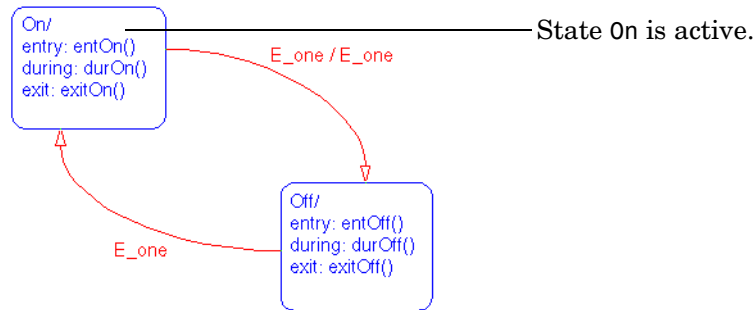
Again, initially the Stateflow diagram is asleep. State `Off` is active. Event `E_one` occurs and awakens the Stateflow diagram. Event `E_one` is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram:

- 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 execute and complete (`exitOff()`).
- 3 State `Off` is marked inactive.
- 4 State `On` is marked active.
- 5 State `On` entry actions execute and complete (`entOn()`).
- 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` was active.

Example: Processing of a Third Event

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



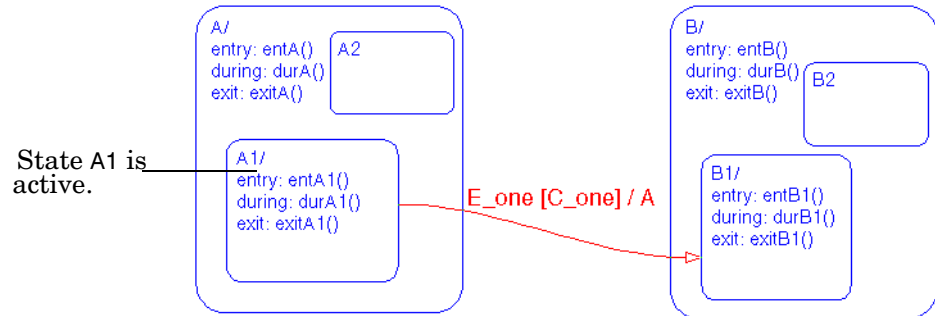
Again, initially the Stateflow diagram is asleep. State `On` is active. Event `E_two` occurs and awakens the Stateflow diagram. Event `E_two` is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram:

- 1 The Stateflow diagram root checks to see if there is a valid transition as a result of `E_two`. There is none.
- 2 State `On` during actions execute and complete (`durOn()`).
- 3 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` was active.

Example: Transition from a Substate to a Substate

This example shows the semantics 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 executes and completes during actions (`durA()`).
- 3 State A.A1 executes and completes exit actions (`exitA1()`).
- 4 State A.A1 is marked inactive.
- 5 State A executes and completes exit actions (`exitA()`).
- 6 State A is marked inactive.
- 7 The transition action, A, is executed and completed.
- 8 State B is marked active.
- 9 State B executes and completes entry actions (`entB()`).
- 10 State B.B1 is marked active.

11 State B.B1 executes and completes entry actions (entB1()).

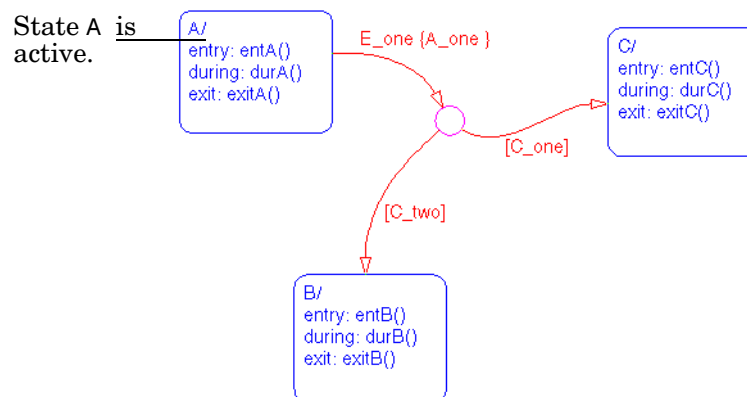
12 The Stateflow diagram goes back to sleep waiting to be awakened by another event.

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

Condition Actions

Example: Actions Specified as Condition Actions

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



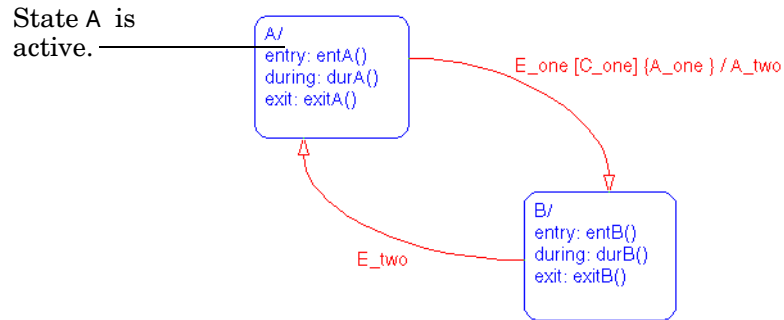
Initially the Stateflow diagram is asleep. State A is active. Event `E_one` occurs and awakens the Stateflow diagram. Conditions `C_one` and `C_two` are false. Event `E_one` is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram:

- 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 Since the conditions on the transition segments to possible destinations are false, none of the complete transitions is valid.
- 3 State A remains active. State A during action executes and completes (`durA()`).
- 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_one when state A was active.

Example: Actions Specified as Condition and Transition Actions

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



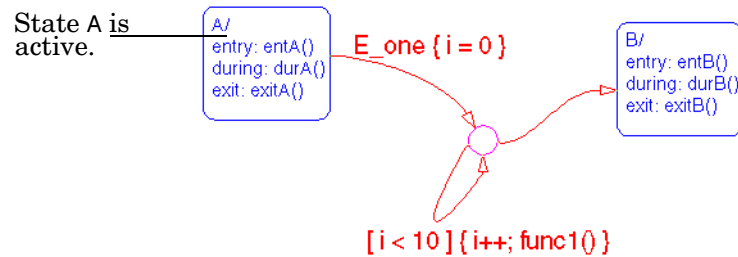
Initially the Stateflow diagram is asleep. State A is active. Event E_one occurs and awakens the Stateflow diagram. Condition C_one is true. Event E_one is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram:

- 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 execute and complete (ExitA()).
- 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 execute and complete (entB()).
- 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 A was active.

Example: Using Condition Actions in For Loop Construct

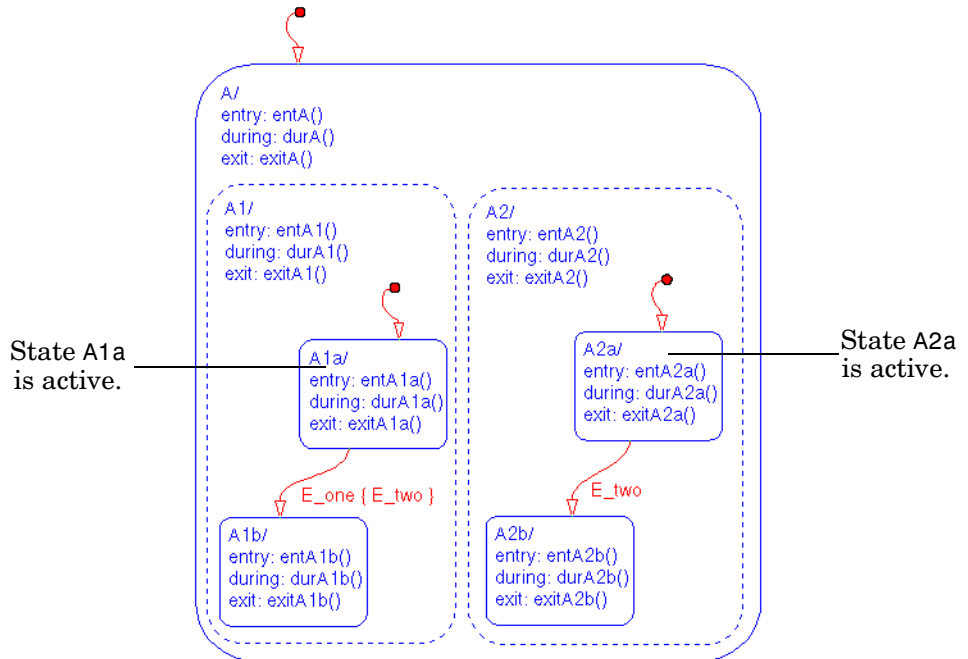
Condition actions and connective junctions are used to design a for loop construct. This example shows the use of a condition action and connective junction to create a for loop construct.



See “Example: For Loop Construct” on page 7-33 to see the semantics of this example.

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

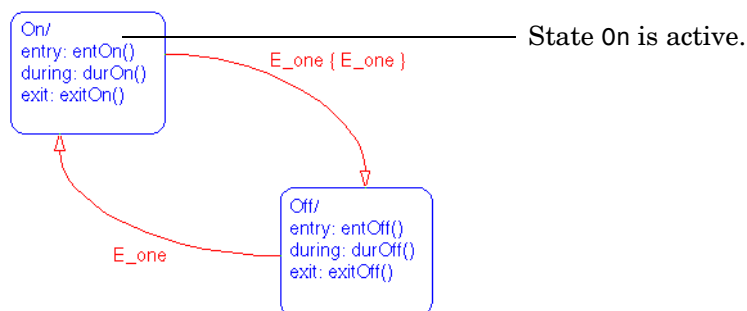
Condition actions can be used to broadcast events immediately to parallel (AND) states. This example shows this use.



See “Example: Event Broadcast Condition Action” on page 7-50 to see the semantics of this example.

Example: Cyclic Behavior to Avoid When Using Condition Actions

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:

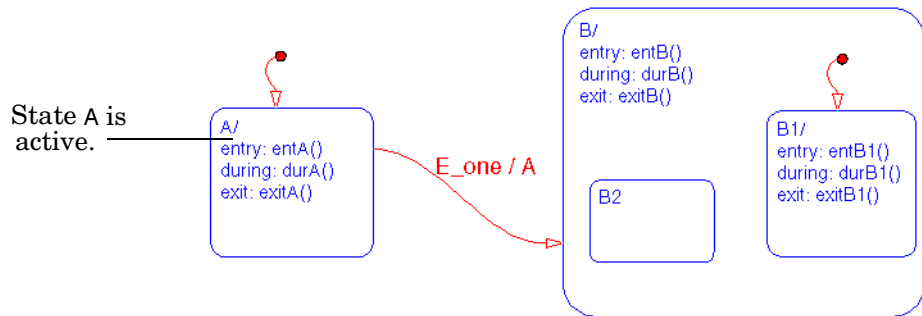
- 1 The Stateflow diagram root checks to see if there is a valid transition as a result of **E_one**. A valid transition from state On to state Off is detected. A condition action, broadcast of event **E_one**, is detected on the valid transition and is immediately executed. State On is still active.
 - The broadcast of event **E_one** 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_one**. The transition from state On to state Off is still valid. The condition action, broadcast of event **E_one**, is immediately executed again.
- 2 Step 1a continues 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 Transitions

Example: Default Transition in an Exclusive (OR) Decomposition

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:

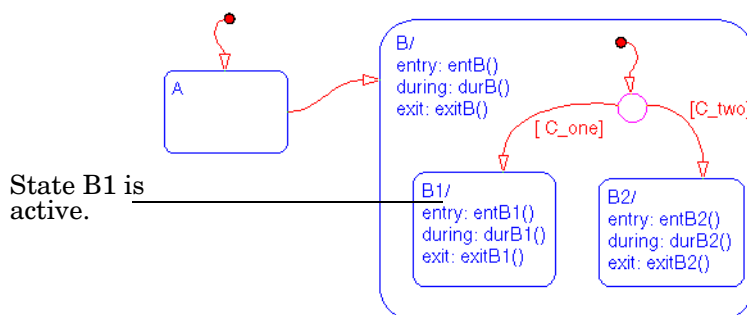
- 1 The Stateflow diagram root checks to see if there is a valid transition as a result of E_one. There is a valid transition from state A to superstate B.
- 2 State A exit actions execute and complete (exitA()).
- 3 State A is marked inactive.
- 4 The transition action, A, is executed and completed.
- 5 State B is marked active.
- 6 State B entry actions execute and complete (entB()).
- 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 execute and complete (entB1()).
- 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.

Example: Default Transition to a Junction

This example shows the semantics of a default transition to a connective junction.



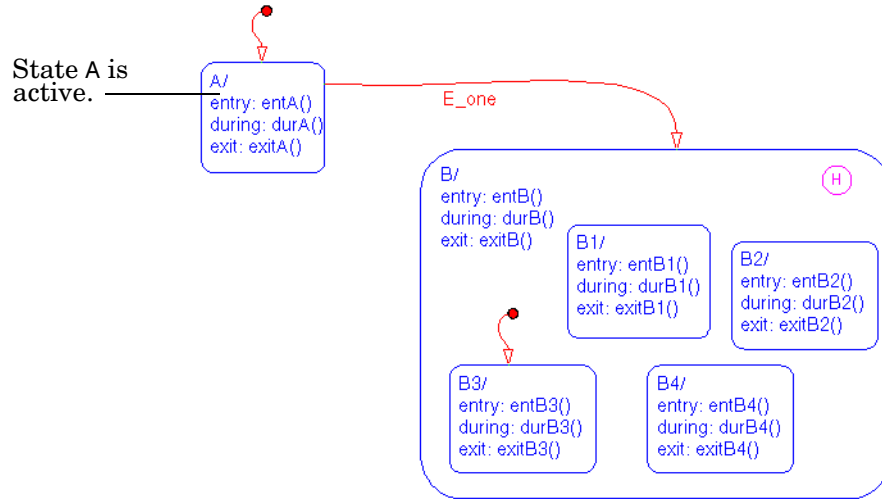
Initially the Stateflow diagram is asleep. State B.B1 is active. An event occurs and awakens the Stateflow diagram. Condition [C_two] is true. The event is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram:

- 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 execute and complete (durB1()).

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

Example: Default Transition and a History Junction

This example shows the semantics of a superstate 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:

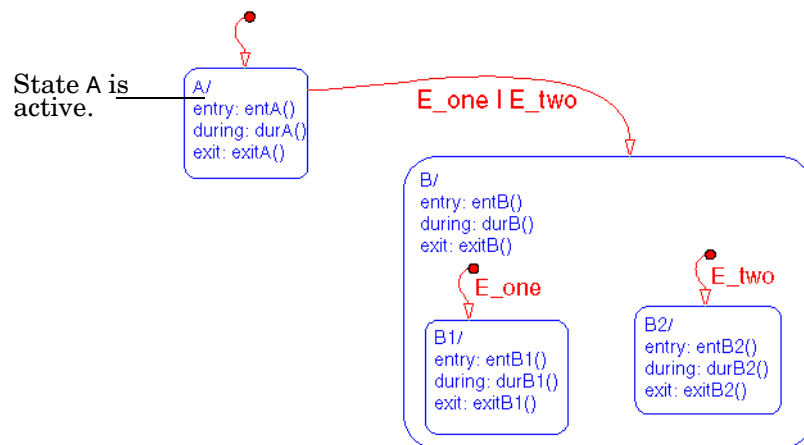
- 1 The Stateflow diagram root checks to see if there is a valid transition as a result of E_one. There is valid transition from state A to superstate B.
- 2 State A exit actions execute and complete (exitA()).
- 3 State A is marked inactive.
- 4 State B is marked active.
- 5 State B entry actions execute and complete (entB()).
- 6 State B detects and uses the history junction to determine which substate is the destination of the transition into the superstate. The history junction indicates substate B.B4 was the last active substate, and thus the destination of the transition.

- 7 State B.B4 is marked active.
- 8 State B.B4 entry actions execute and complete (entB4()).
- 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.

Example: Labeled Default Transitions

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:

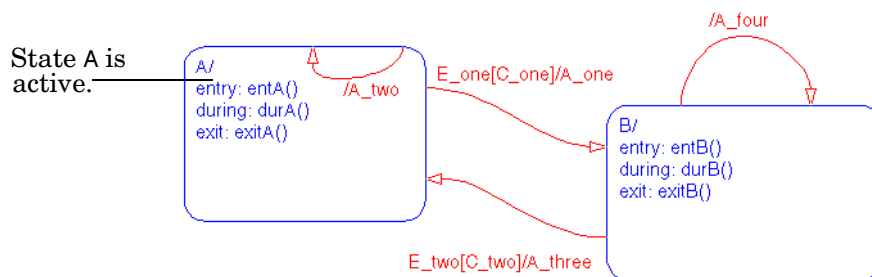
- 1** The Stateflow diagram root checks to see if there is a valid transition as a result of E_one. There is a valid transition from state A to superstate B. A pipe is used to represent that the transition is valid if event E_one or E_two occurs.
- 2** State A exit actions execute and complete (exitA()).
- 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.

Inner Transitions

Example: Processing One Event Within an Exclusive (OR) State

This example shows the semantics of an inner transition.



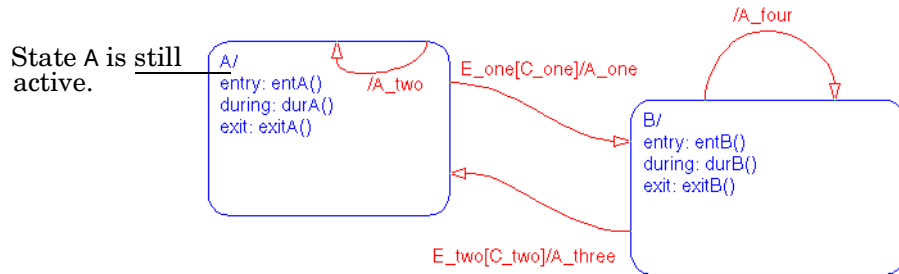
Initially the Stateflow diagram is asleep. State A is active. Event `E_one` occurs and awakens the Stateflow diagram. Condition `[C_one]` is false. Event `E_one` is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram:

- 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 execute and complete (`durA()`).
- 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`.

Example: Processing a Second Event Within an Exclusive (OR) State

Using the same example, what happens when a second event, E_one, occurs?



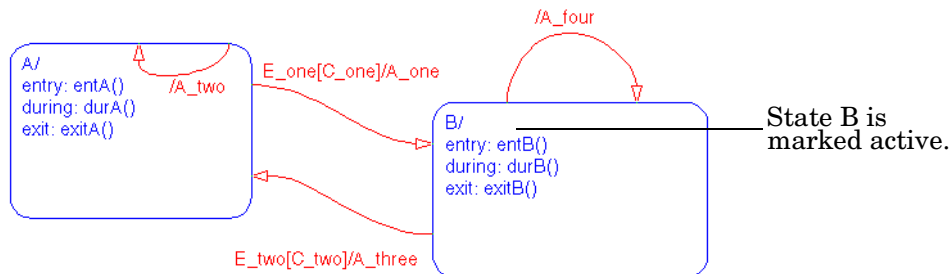
Initially the Stateflow diagram is asleep. State A is still active. Event E_one occurs and awakens the Stateflow diagram. Condition [C_one] is true. Event E_one is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram:

- 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 execute and complete (exitA()).
- 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 execute and complete (entB()).
- 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.

Example: Processing a Third Event Within an Exclusive (OR) State

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



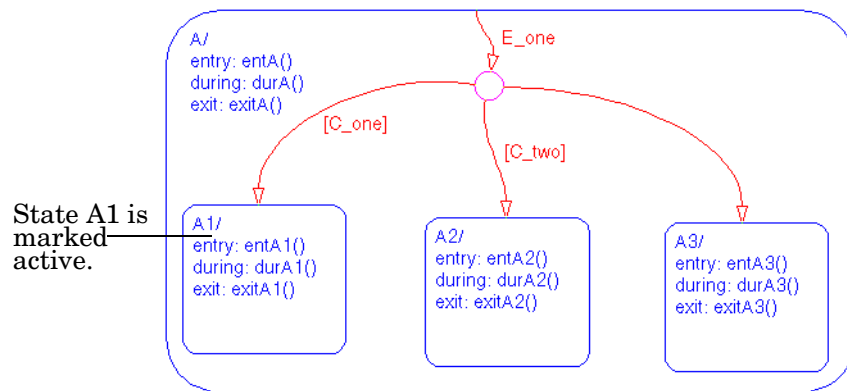
Initially the Stateflow diagram is asleep. State B is now active. Event E_two occurs and awakens the Stateflow diagram. Condition [C_two] is false. Event E_two is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram:

- 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 execute and complete (exitB()).
- 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 execute and complete (entB()).
- 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 semantics between inner transitions and self loop transitions.

Example: Processing One Event with an Inner Transition to a Connective Junction

This example shows the semantics of an inner transition to a connective junction.



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:

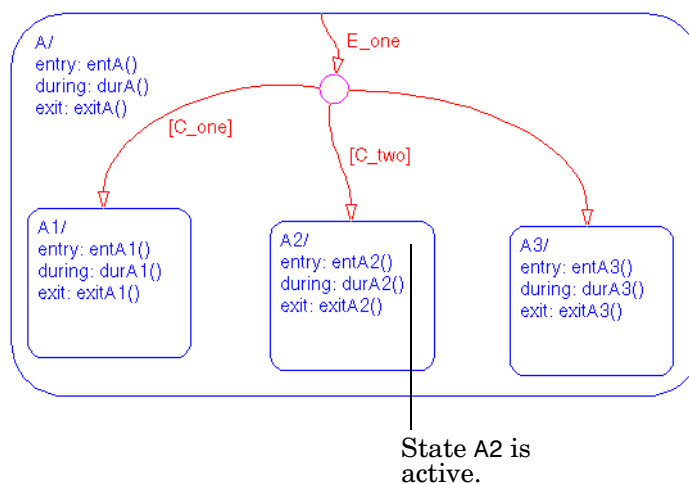
- 1 The Stateflow diagram root checks to see if there is a valid transition at the root level, as a result of E_one. There is no valid transition.
- 2 State A during actions execute and complete (durA()).
- 3 State A checks itself for valid transitions and detects there is a valid inner transition to a connective junction. The conditions are evaluated to determine if 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. Since [C_two] is true, the inner transition to the junction and then to state A.A2 is valid.

- 4 State A.A1 exit actions execute and complete (exitA1()).
- 5 State A.A1 is marked inactive.
- 6 State A.A2 is marked active.
- 7 State A.A2 entry actions execute and complete (entA2()).
- 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.

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

This example shows the semantics of an inner transition to a junction when a second event, E_one, occurs.



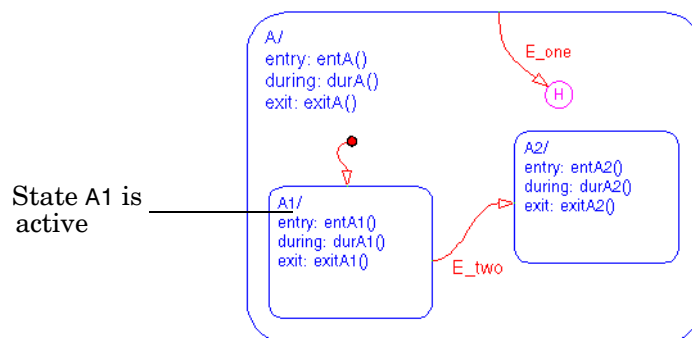
Initially the Stateflow diagram is asleep. State A2 is active. Event E_one occurs and awakens the Stateflow diagram. Neither [C_one] nor [C_two] is true. Event E_one is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram:

- 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 execute and complete (durA()).
- 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. Since 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 execute and complete (exitA2()).
- 5** State A.A2 is marked inactive.
- 6** State A.A3 is marked active.
- 7** State A.A3 entry actions execute and complete (entA3()).
- 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.

Example: Inner Transition to a History Junction

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



Initially the Stateflow diagram is asleep. State A.A1 is active. There is history information since 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:

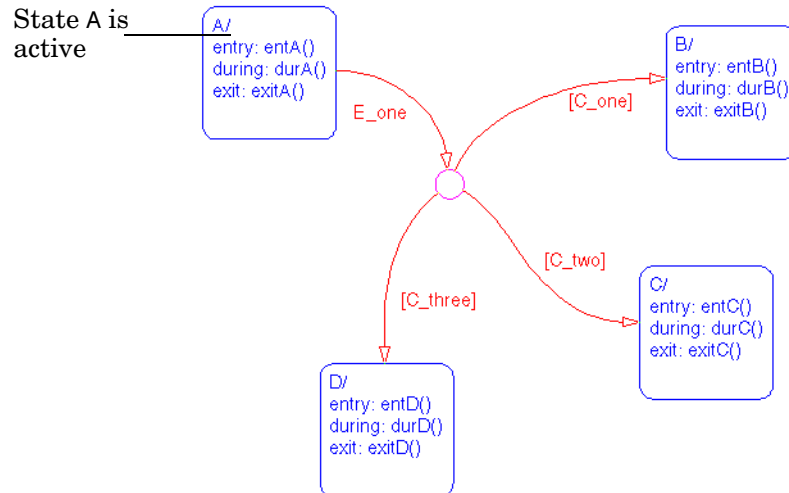
- 1 The Stateflow diagram root checks to see if there is a valid transition as a result of `E_one`. There is no valid transition.
- 2 State A during actions execute and complete (`durA()`).
- 3 State A checks itself for valid transitions and detects there is a valid inner transition to a history junction. According to the semantics of history junctions, the last active state, A.A1, is the destination state.
- 4 State A.A1 exit actions execute and complete (`exitA1()`).
- 5 State A.A1 is marked inactive.
- 6 State A.A1 is marked active.
- 7 State A.A1 entry actions execute and complete (`entA1()`).
- 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 Junctions

Example: If-Then-Else Decision Construct

This example shows the semantics 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:

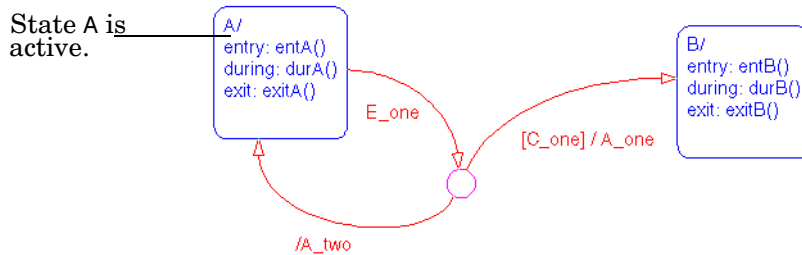
- 1 The Stateflow diagram root checks to see if there is a valid transition as a result of `E_one`. There is a valid transition segment from state A to the connective junction. The transition segments beginning from a twelve o'clock position on the connective junction are evaluated for validity. The 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 executes and completes exit actions (`exitA()`).
- 3 State A is marked inactive.
- 4 State C is marked active.

- 5 State C executes and completes entry actions (entC()).
- 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.

Example: Self Loop

This example shows the semantics of a self loop 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:

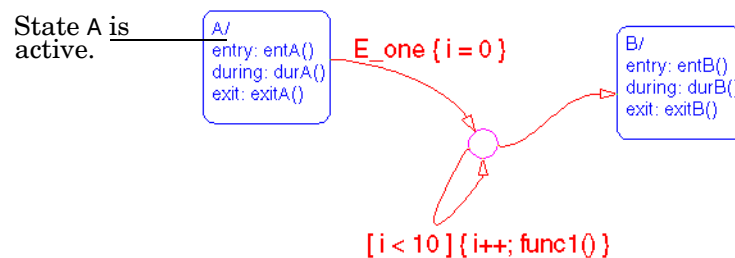
- 1 The Stateflow diagram root checks to see if there is a valid transition as a result of E_one. There is a valid transition segment from state A to the connective junction. The transition segment labeled with a condition and action is evaluated for validity. Since the condition [C_one] is not valid, 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 executes and completes exit actions (exitA()).
- 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 executes and completes entry actions (entA()).
- 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.

Example: For Loop Construct

This example shows the semantics of a for loop.



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.

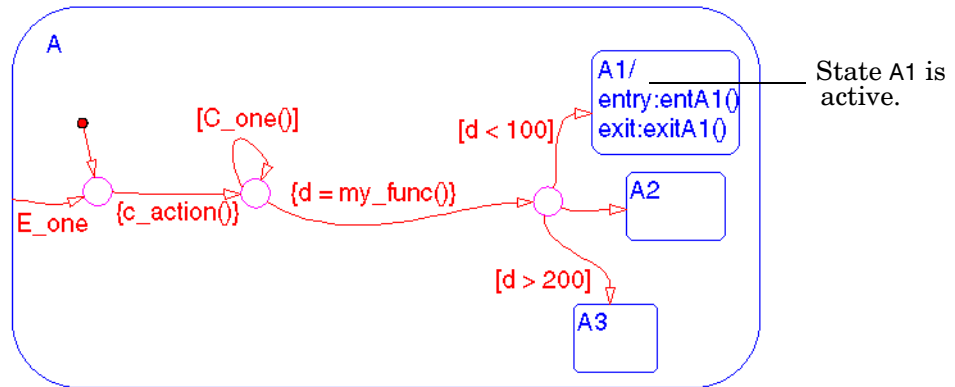
- 1 The Stateflow diagram root checks to see if there is a valid transition as a result of E_one. There is a valid transition segment from state A to the connective junction. The transition segment condition action, $i = 0$, is executed and completed. Of the two transition segments leaving the connective 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 executes and completes exit actions (exitA()).
- 5 State A is marked inactive.
- 6 State B is marked active.
- 7 State B executes and completes entry actions (entB()).
- 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.

Example: Flow Diagram Notation

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



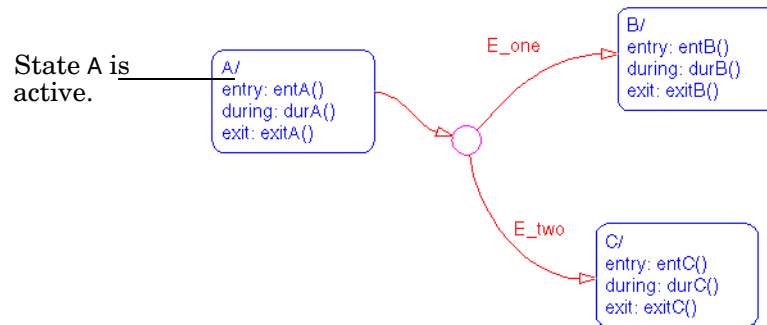
Initially the Stateflow diagram is asleep. State A.A1 is active. The condition [C_one()] is initially true. Event E_one occurs and awakens the Stateflow diagram. Event E_one is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram:

- 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 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 is taken. Since a final transition destination has not been reached, this self loop continues until [C_one()] is false. Assume that after five loops [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. Since 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 execute and complete (exitA1()).
- 8** State A.A1 is marked inactive.
- 9** State A.A1 is marked active.
- 10** State A.A1 entry actions execute and complete (entA1()).
- 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.

Example: Transitions from a Common Source to Multiple Destinations

This example shows the semantics of transitions from a common source to multiple destinations.



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:

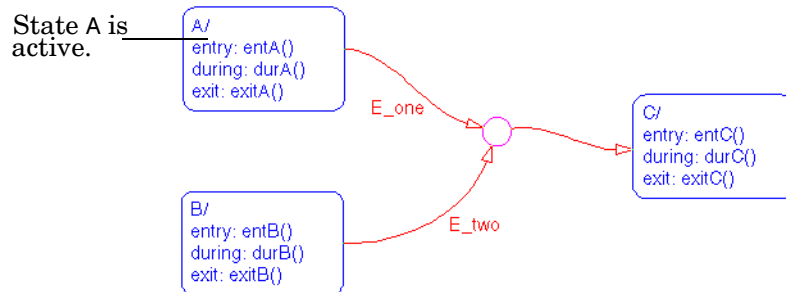
- 1 The Stateflow diagram root checks to see if there is a valid transition as a result of E_two. There is a valid transition segment from state A to the connective junction. Given that the transition segments are equivalently labeled, evaluation begins from a twelve o'clock position on the connective 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 executes and completes exit actions (exitA()).
- 3 State A is marked inactive.
- 4 State C is marked active.

- 5 State C executes and completes entry actions (`entC()`).
- 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`.

Example: Transitions from Multiple Sources to a Common Destination

This example shows the semantics of transitions from multiple sources to a single destination.



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.

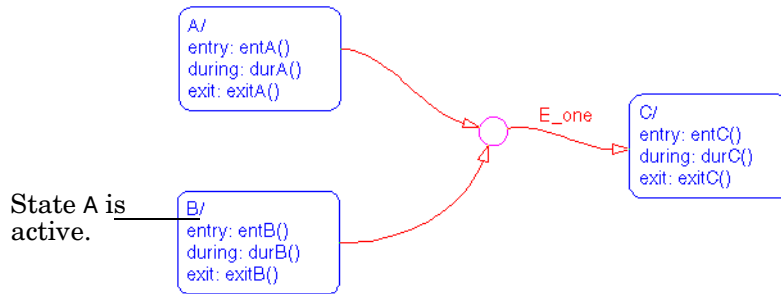
- 1 The Stateflow diagram root checks to see if there is a valid transition as a result of `E_one`. There is a valid transition segment from state A to the connective junction and from the junction to state C.
- 2 State A executes and completes exit actions (`exitA()`).
- 3 State A is marked inactive.
- 4 State C is marked active.

- 5 State C executes and completes entry actions (entC()).
- 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.

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

This example shows the semantics of transitions from multiple sources to a single destination based on the same event.



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 executes and completes exit actions (exitB()).
- 3 State B is marked inactive.
- 4 State C is marked active.

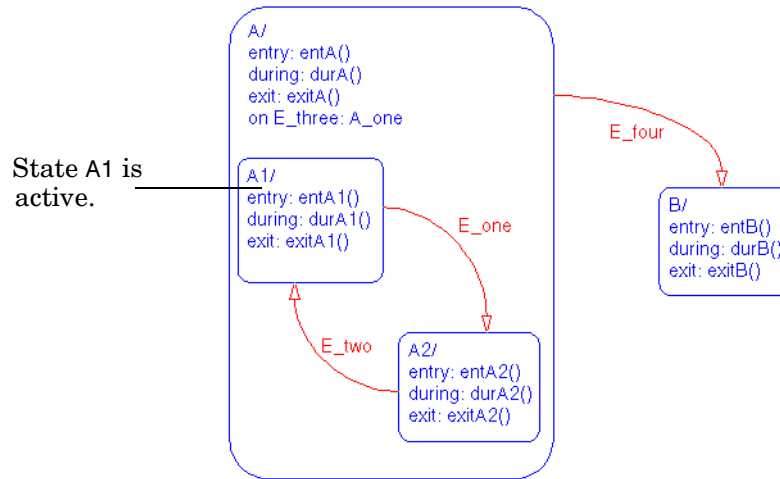
- 5** State C executes and completes entry actions (entC()).
- 6** The Stateflow diagram goes back to sleep waiting to be awakened by another event.

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

Event Actions

Example: Event Actions and Superstates

This example shows the semantics of event actions within superstates.



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:

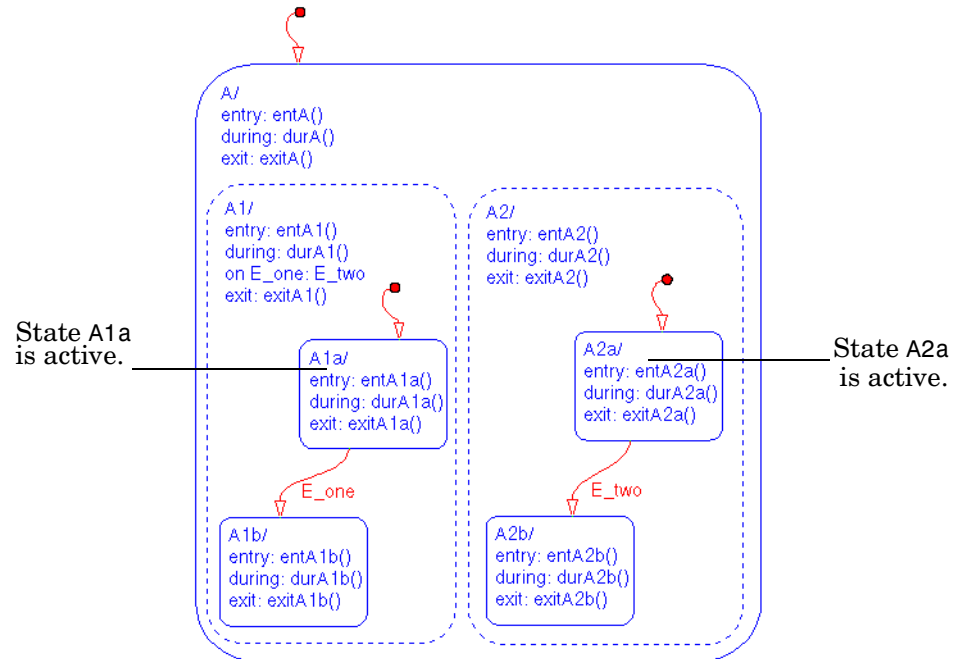
- 1 The Stateflow diagram root checks to see if there is a valid transition as a result of E_three. There is no valid transition.
- 2 State A executes and completes during actions (durA()).
- 3 State A executes and completes the on event E_three action (A_one).
- 4 State A checks its children for valid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

Example: Event Broadcast State Action

This example shows the semantics of event broadcast state actions.



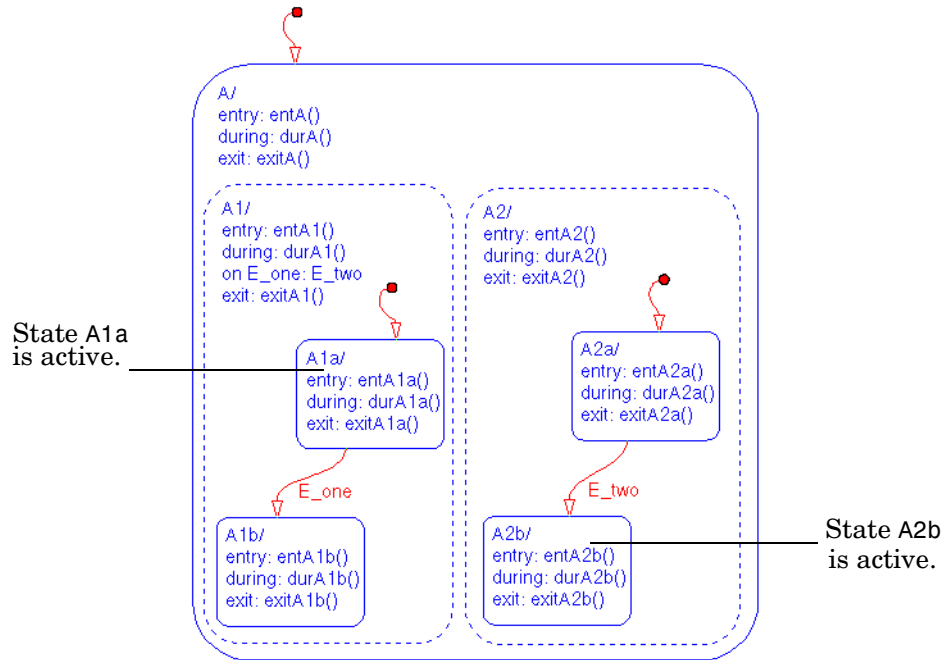
Initially the Stateflow diagram is asleep. Parallel substates A.A1.A1a and A.A2.A2a are active. Event E_one occurs and awakens the Stateflow diagram. Event E_one is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram:

- 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 executes and completes during actions (durA()).
- 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 executes and completes during actions (durA1()). 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 executes and completes during actions (durA()).
- 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 executes and completes during actions (durA1()). 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 executes and completes during actions (durA2()). 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 execute and complete (exitA2a()).
- g** State A.A2.A2a is marked inactive.

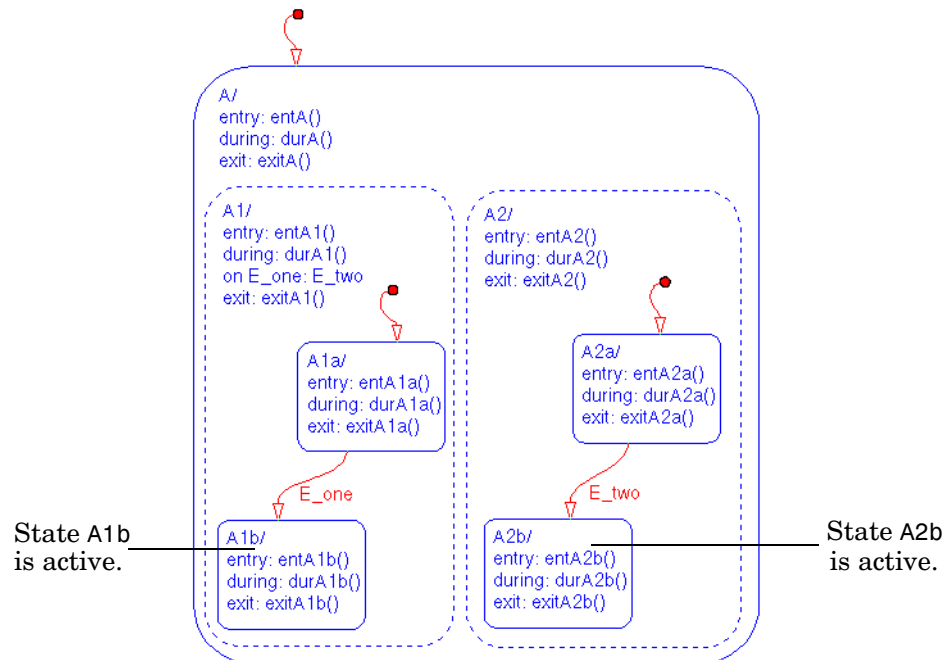
- h** State A.A2.A2b is marked active.
- i** State A.A2.A2b entry actions execute and complete (entA2b()). The Stateflow diagram activity now looks like this.



- 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 executes and completes entry actions (entA1b()).
- 8** State A.A1.A1b is marked active.

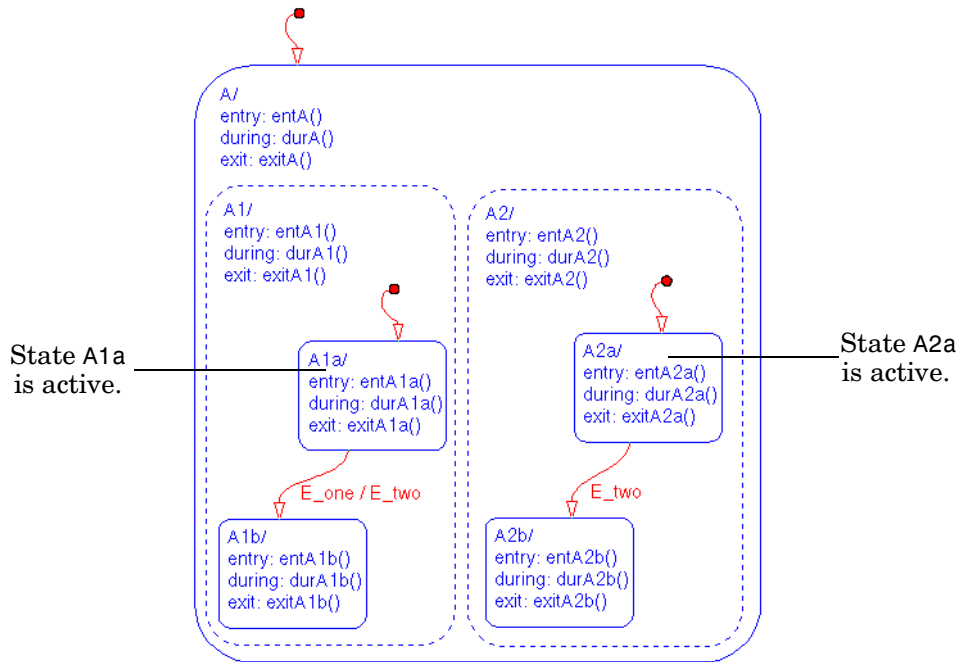
- 9 Parallel state A.A2 is evaluated next. State A.A2 during actions execute and complete (`durA2()`). There are no valid transitions as a result of `E_one`.
- 10 State A.A2.A2b, now active as a result of the processing of the on event broadcast of `E_two`, executes and completes during actions (`durA2b()`).
- 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.



Example: Event Broadcast Transition Action (Nested Event Broadcast)

This example shows the semantics of an event broadcast transition action that includes nested event broadcasts.



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 down through the hierarchy of the Stateflow diagram:

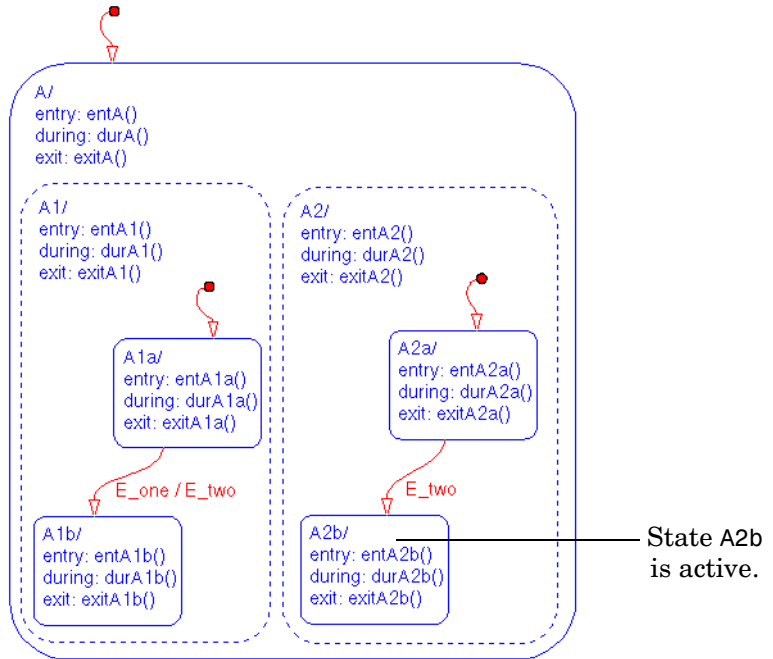
- 1 The Stateflow diagram root checks to see if there is a valid transition as a result of E_one. There is no valid transition.
- 2 State A executes and completes during actions (`durA()`).
- 3 State A's children are parallel (AND) states. They are evaluated and executed from left to right and top to bottom. State A.A1 is evaluated first. State A.A1 executes and completes during actions (`durA1()`).

Start of event
E_two
_____ processing
(E_one processing
is preempted)

- 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.
- 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 executes and completes during actions (durA()).
 - d** State A's children are evaluated starting with state A.A1. State A.A1 executes and completes during actions (durA1()). 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 executes and completes during actions (durA2()). 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 execute and complete (exitA2a()).
 - g** State A.A2.A2a is marked inactive.

End of event
E_two
processing

- h State A.A2.A2b is marked active.
- i State A.A2.A2b entry actions execute and complete (entA2b()). The Stateflow diagram activity now looks like this.

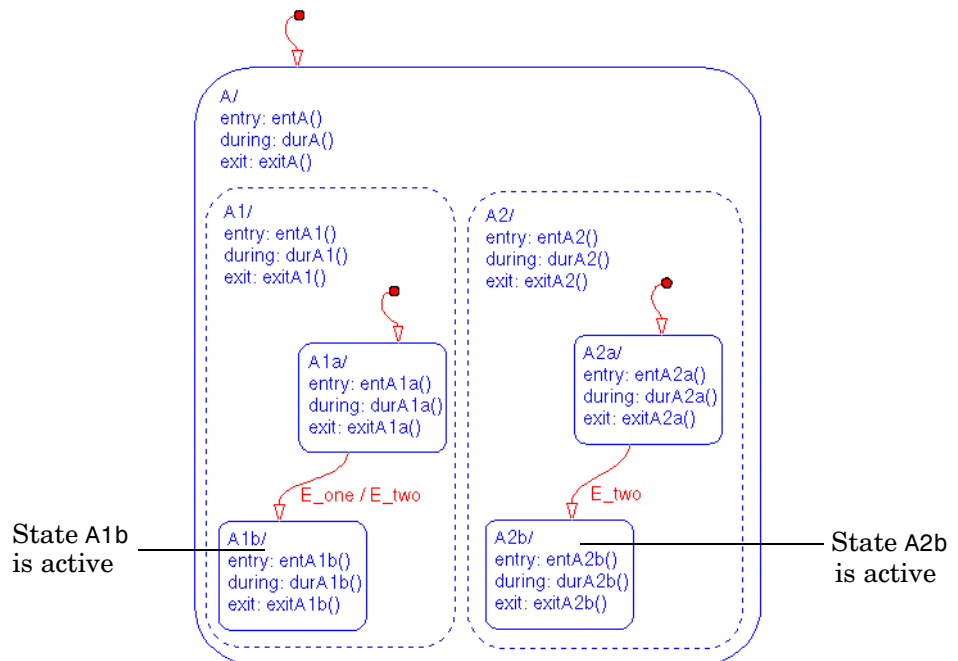


- 8 State A.A1.A1b is marked active.

Continuation
of event
E_one
processing

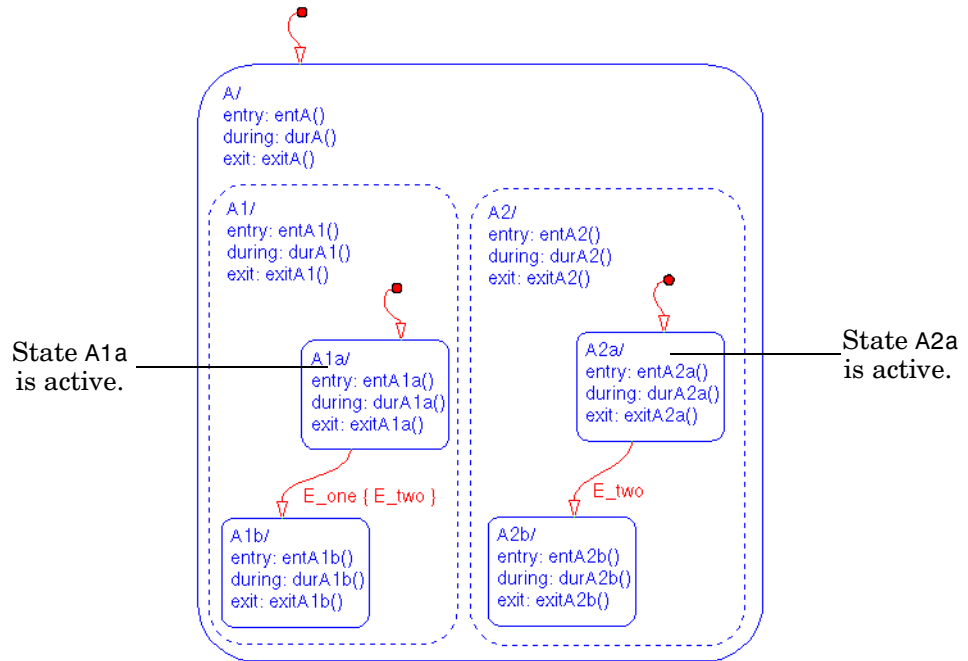
- 9 State A.A1.A1b executes and completes entry actions (entA1b()).
- 10 Parallel state A.A2 is evaluated next. State A.A2 during actions execute and complete (durA2()). There are no valid transitions as a result of E_one.
- 11 State A.A2.A2b, now active as a result of the processing of the transition action event broadcast of E_two, executes and completes during actions (durA2b()).
- 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.



Example: Event Broadcast Condition Action

This example shows the semantics of condition action event broadcast in parallel (AND) states.

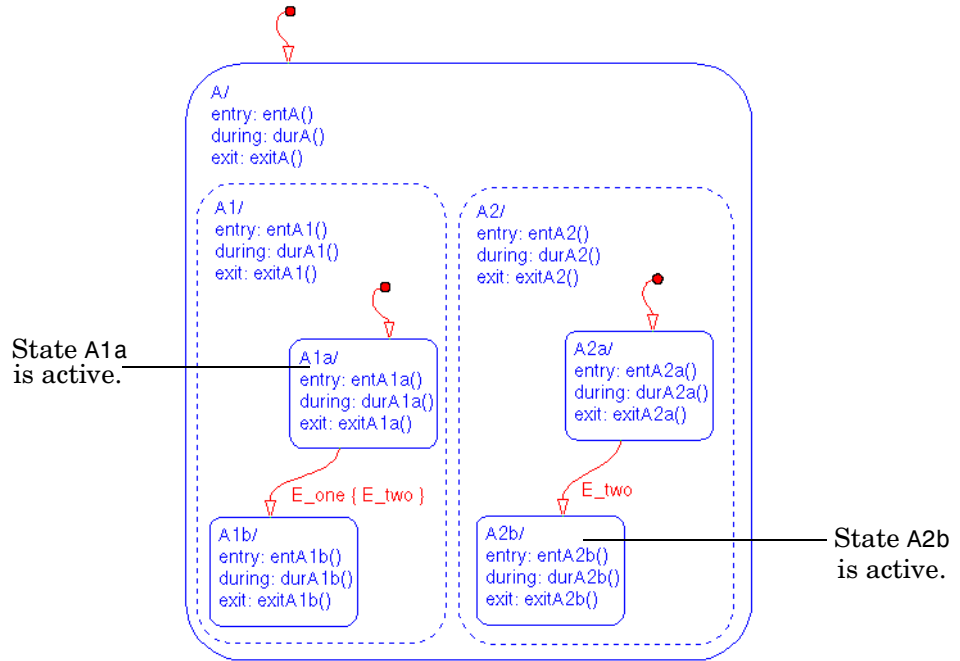


Initially the Stateflow diagram is asleep. Parallel substates A.A1.A1a and A.A2.A2a are active. Event E_one occurs and awakens the Stateflow diagram. Event E_one is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram:

- 1 The Stateflow diagram root checks to see if there is a valid transition as a result of E_one. There is no valid transition.
- 2 State A executes and completes during actions (durA()).
- 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 executes and completes during actions (durA1()).

- 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 executes and completes during actions (durA()).
 - c** State A's children are evaluated starting with state A.A1. State A.A1 executes and completes during actions (durA1()). 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 executes and completes during actions (durA2()). 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 execute and complete (exitA2a()).
 - f** State A.A2.A2a is marked inactive.

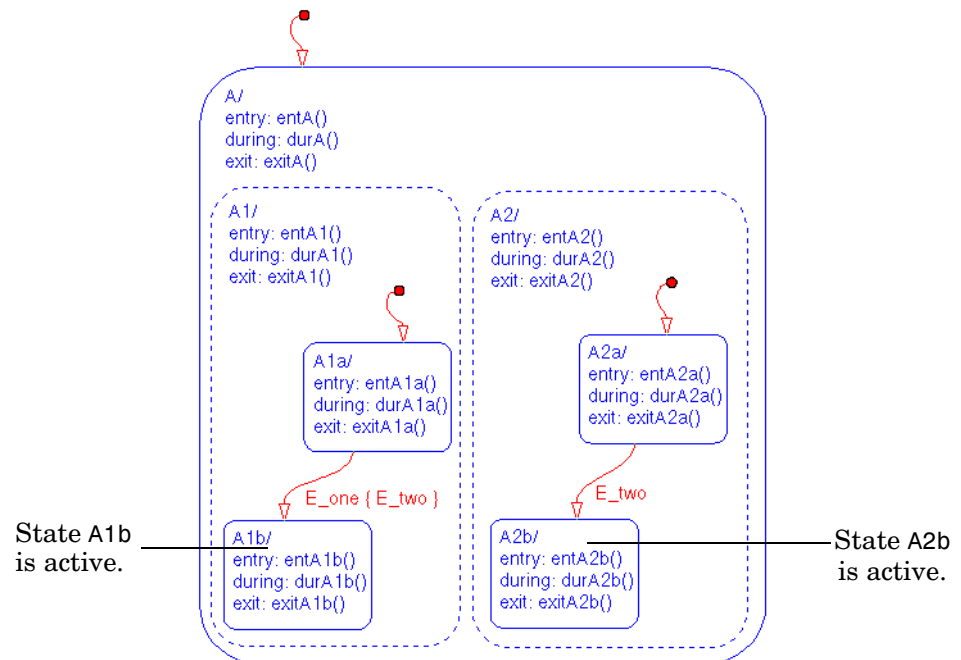
- g State A.A2.A2b is marked active.
- h State A.A2.A2b entry actions execute and complete (entA2b()). The Stateflow diagram activity now looks like this.



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

- 10** State A.A2.A2b, now active as a result of the processing of the condition action event broadcast of E_two, executes and completes during actions (durA2b()).
- 11** The Stateflow diagram goes back to sleep waiting to be awakened by another event.

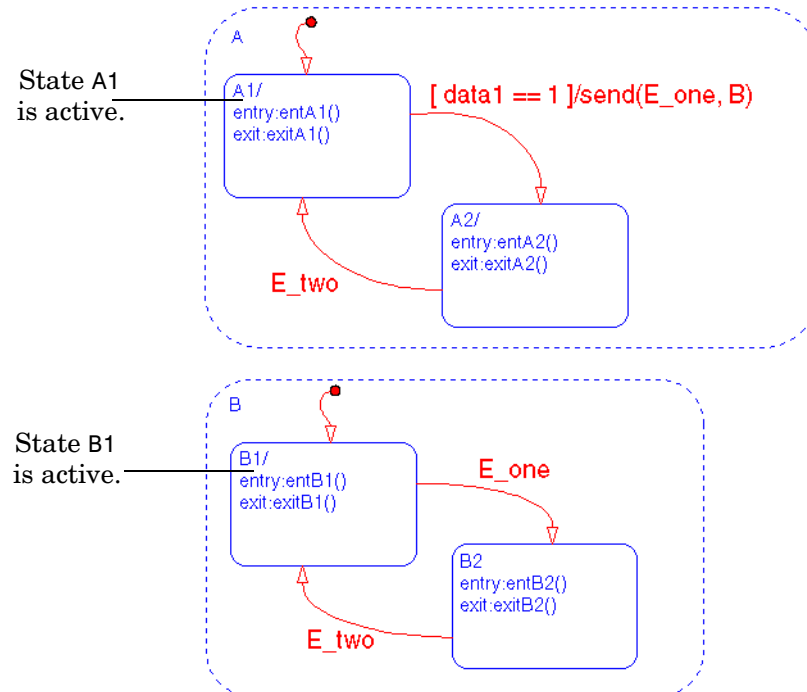
This sequence completes the execution of this Stateflow diagram associated with event E_one and the condition action event broadcast to a parallel state of event E_two. The final Stateflow diagram activity now looks like this.



Directed Event Broadcasting

Example: Directed Event Broadcast Using send

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

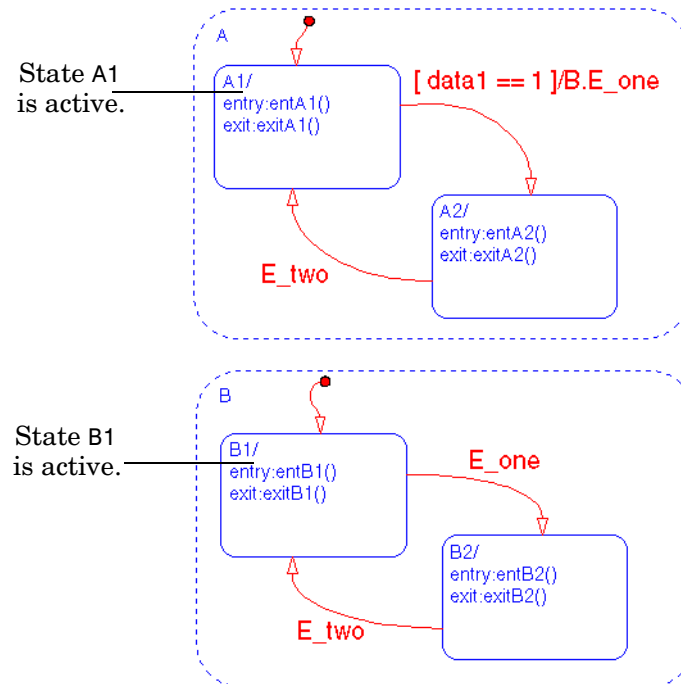


Initially the Stateflow diagram is asleep. Parallel substates A.A1 and B.B1 are active. By definition, this implies parallel (AND) superstates A and B are active. An event occurs and awakens the Stateflow diagram. The condition `[data1==1]` is true. The event is processed from the root of the Stateflow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

- Start of event E_one processing. _____
- 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. Since 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 execute and complete (exitA1()).
 - 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.) Since 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 executes and completes exit actions (exitB1()).
 - c** State B.B1 is marked inactive.
 - d** State B.B2 is marked active.
 - e** State B.B2 executes and completes entry actions (entB2()).
 - 6** State A.A2 is marked active.
 - 7** State A.A2 entry actions execute and complete (entA2()).
- End of event E_one processing. _____
- 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.

Example: Directed Event Broadcasting Using Qualified Event Names

This example shows the semantics 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 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. Since 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 execute and complete (`exitA1()`).
- 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.) Since 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` executes and completes exit actions (`exitB1()`).
 - c** State `B.B1` is marked inactive.
 - d** State `B.B2` is marked active.
 - e** State `B.B2` executes and completes entry actions (`entB2()`).
- 6** State A.A2 is marked active.
- 7** State A.A2 entry actions execute and complete (`entA2()`).

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

Execution Order

Overview

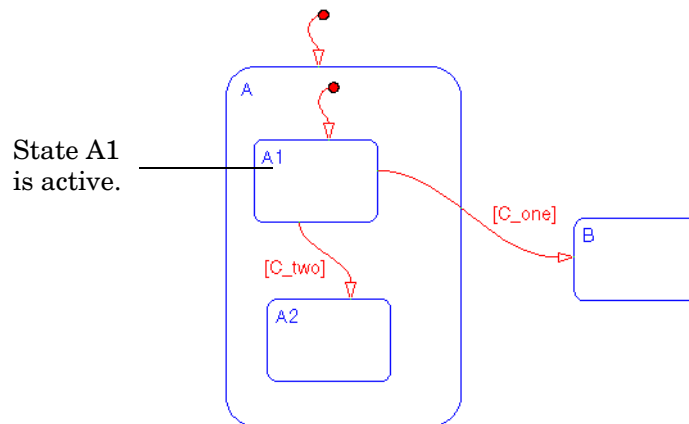
In a single processor environment, sequential execution order is the only option. In this case, it may be necessary for you to know the implicit ordering determined by a Stateflow diagram. The ordering is specific to transitions originating from the same source. Knowing the order of execution for Stateflow diagrams with more than one parallel (AND) state may be important.

We recommend that you do not design your Stateflow diagram based on an expected execution order.

Execution Order Guidelines

Execution order of transitions originating from the same source is based on these guidelines. The guidelines appear in order of their precedence:

- 1 Transitions are evaluated, based on hierarchy, in a top-down manner. In this example, when an event occurs and state A.A1 is active, the transition from state A.A1 to state B is valid and takes precedence over the transition from state A.A1 to state A.A2 based on the hierarchy.

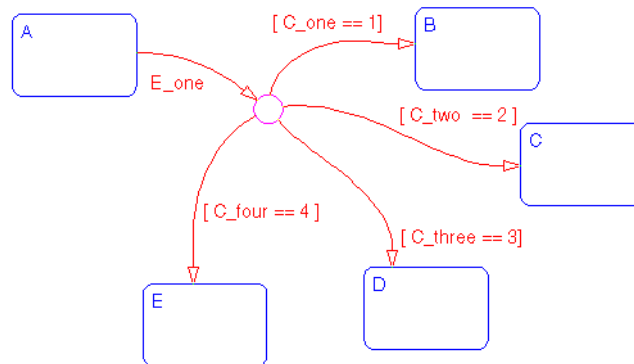


- 2 Transitions are evaluated based on their labels.

- a Labels with events and conditions
 - b Labels with events
 - c Labels with conditions
 - d No label
- 3 Equivalent transitions (based on their labels) are evaluated based on the geometry of the outgoing transitions. The geometry of junctions and states is considered separately.

Junctions

Multiple outgoing transitions from junctions that are of equivalent label priority are evaluated in a clockwise progression starting from a twelve o'clock position on the junction.



In this example, the transitions are of equivalent label priority. The conditions $[C_three == 3]$ and $[C_four == 4]$ are both true. Given that, the outgoing transitions from the junction are evaluated in this order:

- 1 A → B

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

- 2 A → C

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

3 $A \rightarrow D$

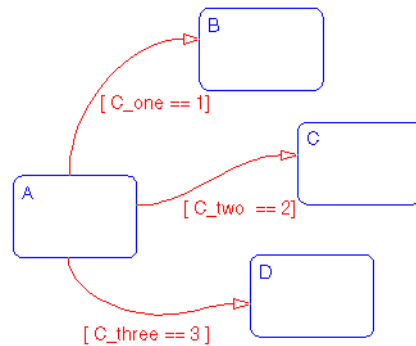
Since the condition $[C_three == 3]$ is true, this transition is valid and is taken.

4 $A \rightarrow E$

This transition, even though it too is valid, is not evaluated since the previous transition evaluated was valid.

States

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



In this example, the transitions are of equivalent label priority. The conditions $[C_two == 2]$ and $[C_three == 3]$ are both true and $[C_one == 1]$ is false. Given that, the outgoing transitions from the state are evaluated in this order:

1 $A \rightarrow B$

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

2 A → C

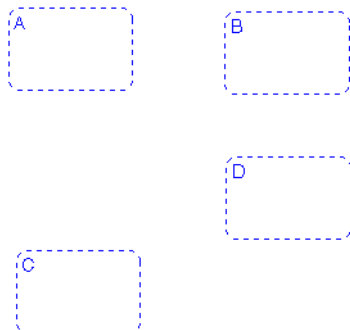
Since the condition `[C_two == 2]` is true, this transition is valid and is taken.

3 A → D

This transition, even though it too is valid, is not evaluated since the previous transition evaluated was valid.

Parallel (AND) States

Parallel (AND) states are evaluated and executed first from top to bottom and then from left to right in the case of a tie. In this example, assuming that A and B, and C and D are exactly equivalent from top-down, the parallel (AND) states are executed in this order: A, B, D, C.



Semantic Rules Summary

Entering a Chart

The set of default flow paths is executed (see “Executing a Set of Flow Graphs”). If this does not cause a state entry and the chart has parallel decomposition, then each parallel state is entered (see “Entering a State”).

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 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 Perform explicit entry actions, 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 this state. 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 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

- 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 these 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 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 actions from the entire transition path are executed in the order the respective segments were validated.
- 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” and “Executing an Active State”). (The event *receiver* is the

parent of the event unless the event was explicitly directed to a *receiver* using the `send()` function.)

- 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.

Tutorial: Entering a Stateflow Diagram

Designing the Carousel Stateflow Diagram	8-4
Procedure	8-8
Run the Simulation for the Provided Solution	8-22
Run the Carousel Demo	8-25

Overview

Objectives

After completing this tutorial you should be able to:

- Understand the Stateflow diagram design process for the carousel.
- Understand the notation and methods to create a model and Stateflow diagram using Stateflow.
- Create an initial model and Stateflow diagram of the carousel.
- Run the simulation using the provided solution.
- Run the demo of the solution.

Provided Models for This Chapter

The files associated with the tutorial example, the running solution, and the demo are included and accessible from the Examples block.

Carousel Tutorial Example

A carousel is a familiar reactive system that you can model with a Stateflow diagram. The system reacts based upon the operator's actions.

This tutorial focuses on learning the notation and the methods to create an initial Stateflow diagram for the carousel. The completed tutorial example cannot be simulated. The provided solution includes a Simulink model and can be simulated. The demo is more elaborate and shows how the Simulink model can be combined with a graphical user interface (GUI).

Basic Operation

The carousel is divided into these main physical components:

- Main platform of the ride
- Audio
- Video

The carousel can either be on or off. When the carousel is on, the platform can be stopped or moving, depending on whether a ride is in progress. You have the option of having music play and displaying a video image of the carousel that

rotates on the operator's console. You enable audio and video before starting the ride. When the operator of the ride pushes the start button, the platform turns (music plays and video rotates if enabled). The ride lasts the amount of time the operator specifies.

If the carousel exceeds a maximum speed, it stops automatically. There is an emergency stop button on the carousel operator's console if, for any reason, the operator decides the ride needs to be stopped.

Designing the Carousel Stateflow Diagram

Overview

A Stateflow diagram is only as good as the up-front analysis and design work for the desired system behavior. Before you can create your Stateflow diagram, you need to interpret the verbal description of the system behavior and translate that description into Stateflow notation. For the purposes of this tutorial, some aspects of the operation are simplified.

The Design Questions

Addressing these questions helps in creating a Stateflow diagram that is representative of the carousel system:

- What are the states?
- What is the decomposition of the states?
- What should the state labels be?
- Are there junctions?
- What are the transitions?
- Are there default transitions?
- Are there events?
- Is there data?

What Are the States?

Begin by listing possible states in the carousel system. A good place to begin is by considering the physical components of the carousel. There are likely to be states associated in some way with some of the physical components.

Audio

The audio components are a subset of the carousel system as a whole. If the audio system is enabled on the operator's console, music plays when the ride is in progress. In this tutorial, the substates of the audio component are not discussed in any detail.

Video

The video components are a subset of the carousel system as a whole. If the video system is enabled on the operator's console, an image of a rotating platform is displayed on the operator's console. In this tutorial, the substates of the video component are not discussed in any detail.

Ride

The platform movement is the ride. There is a state associated with the ride as a whole. In the simplest sense the ride is either in progress or not. There is one state, Ride, that consists of two substates, On, and Stopped.

Timer

A state for a timer is somewhat less obvious than the others. The duration of the ride is set by the operator. It is useful to define a separate state for the timer that will generate the event indicating when the ride is done. The timer is either counting or stopped. There are three states associated with the timer: Counting, Stopped, and Timer.

The Carousel System

The carousel system is either on or off. There are two states associated with the carousel at the system level: Power_on and Power_off.

What Is a State Decomposition?

The Stateflow diagram level of the hierarchy has exclusive (OR) decomposition and consists of two OR states: Power_on and Power_off.

The Power_on state has parallel (AND) decomposition and consists of these AND substates: Timer, Audio, Video, and Ride. When the Power_on state is active, the Timer, Audio, Video, and Ride states are active. They are essentially subsystems of the carousel system.

The Timer, Ride, Audio, and Video states have exclusive (OR) decomposition.

See "What Is a State Decomposition?" on page 6-9 for more information.

What Are the State Labels?

Part of the process of defining each state is defining the state label. The state name is the part of the label that describes the state in some recognizable way. The state labels in the carousel consist of the name of the state.

See the section titled “Labeling a State” on page 6-12 for information on valid state labels.

Will Junctions Simplify the Design?

The carousel system as described and designed does not have junctions.

See the sections titled “Connective Junctions” on page 6-31 and “History” on page 6-39 for information on the use and notation for Connective and History junctions.

What Are the Transitions?

In the carousel system, these are the transitions between OR states:

- Power_off→Power_on triggered by event Switch_on
- Power_on→Power_off triggered by event Switch_off
- Power_on.Ride.On→Power_on.Ride.Stopped triggered by any event when the condition [Speed > Max_speed] is true
- Power_on.Ride.On→Power_on.Ride.Stopped triggered by event Emergency_stop
- Power_on.Ride.On→Power_on.Ride.Stopped triggered by event Ride_done
- Power_on.Ride.Stopped→Power_on.Ride.On triggered by event Start_button
- Power_on.Timer.Stopped→Power_on.Timer.Counting triggered by event Start_button
- Power_on.Timer.Counting→Power_on.Timer.Stopped triggered by any event when the condition [Time > Duration] is true. The event Ride_done is broadcast as a transition action.

See the section titled “Transitions” on page 6-16 for more information.

Are There Default Transitions?

These are the default transitions in the example:

- The `Power_off` substate is the default state to become active upon a transition to the carousel Stateflow diagram.
- The `Power_on.Ride.Stopped` substate is the default state to become active upon a transition to the `Ride` superstate.
- The `Power_on.Timer.Stopped` substate is the default state to become active upon a transition to the `Timer` superstate.

See the section titled “What Is a Default Transition?” on page 6-23 for more information.

Are There Events?

These events are defined: `Ride_done`, `Switch_on`, `Switch_off`, `Start_button`, and `Emergency_stop`.

See the sections titled “Defining Events” on page 3-2 for more information on events.

Is There Data?

The carousel system has these data objects: `Time`, `Speed`, `Duration`, and `Max_speed`.

See the section titled “Defining Data” on page 3-14 for more information on data.

Procedure

Overview

A carousel is a familiar reactive system that you can model with a Stateflow diagram. The system reacts based upon actions the ride operator takes to use it. For the purposes of this tutorial, some aspects of the operation are simplified.

Use the sample solution in this section as a guide while following these steps. In this section you learn how to:

- Create a simple Simulink model with a Stateflow block.
- Save an untitled model and Stateflow diagram.
- Create and label the OR states.
- Create and label the AND states.
- Create and label the superstates.
- Create the transitions.
- Label the transitions.
- Create default transitions.
- Add and define event objects.
- Add and define data objects.
- Save the Stateflow diagram and exit.

Create a Simulink Model

These steps describe how to invoke Stateflow and save the untitled model and Stateflow diagram:

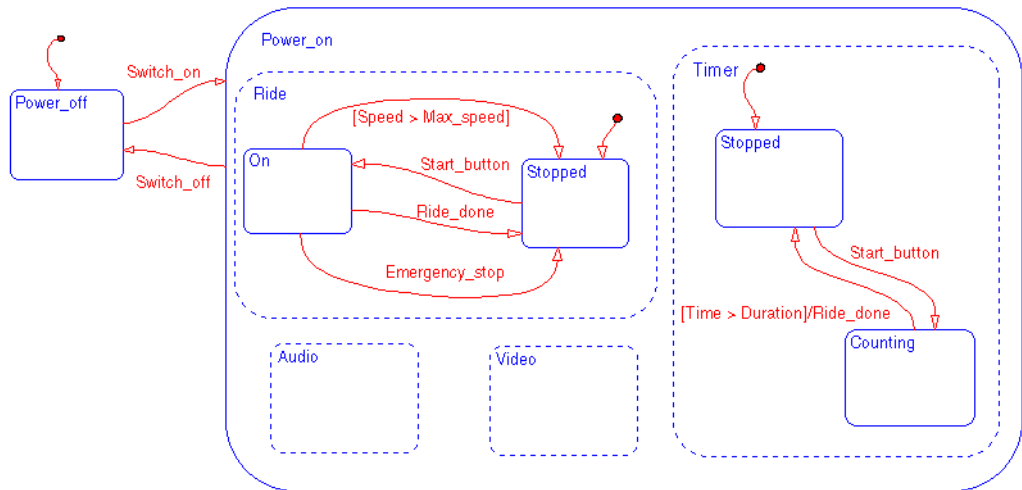
- 1 Double-click on the MATLAB icon to invoke MATLAB.
- 2 At the MATLAB prompt, type `sfnew` to display an untitled model window.

Choose **Save As...** from the **File** menu of the model window. Enter a unique model name, modify the saved location if desired, and click on the **OK** button.


- 3 Click in the text area of the Stateflow block and enter a unique Stateflow block name. Double-click on the Stateflow block to invoke the Stateflow graphics editor.

Sample Stateflow Diagram

This is a sample of what the completed Stateflow diagram should look like.



Create and Label the OR States

1 Double-click on the **State** button  in the toolbar. The button appears depressed and is now in multiple object mode. Using the sample solution Stateflow diagram as a guide, click in the drawing area to place the states in the drawing area. The Stateflow diagram is exclusive (OR) decomposition by default, so you do not need to change the decomposition. The states correspond to these states in the solution:

- `Power_off`
- `Power_on.Ride.On`
- `Power_on.Ride.Stopped`
- `Power_on.Timer.Stopped`
- `Power_on.Timer.Counting`

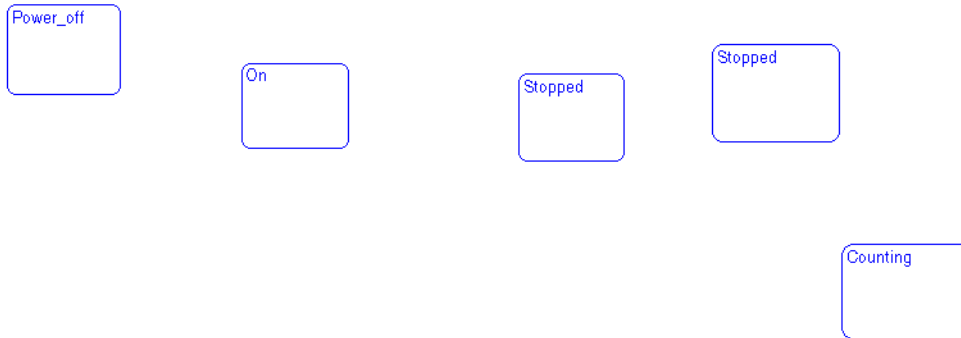
Press the **Escape** key or click on the **State** button to toggle the button out of multiple object mode.

2 Click on the ? character within each state to enter each state label. The final text after the last period is the individual state's name. There are some states that have the same text as their label. This is ok since the hierarchy of the state is used to create a unique label. For example, both the `Ride` and `Timer` superstates have a `Stopped` substate. Reading the Stateflow diagram, you can see that these states are uniquely described by the text `Power_on.Ride.Stopped` and `Power_on.Timer.Stopped`.

Use the sample solution to label the states based on their placement in the drawing area. Deselect the state (press the **Escape** key) to exit the edit.



Sample Snapshot


Your Stateflow diagram should look similar to this Stateflow diagram.



Create and Label the Superstates

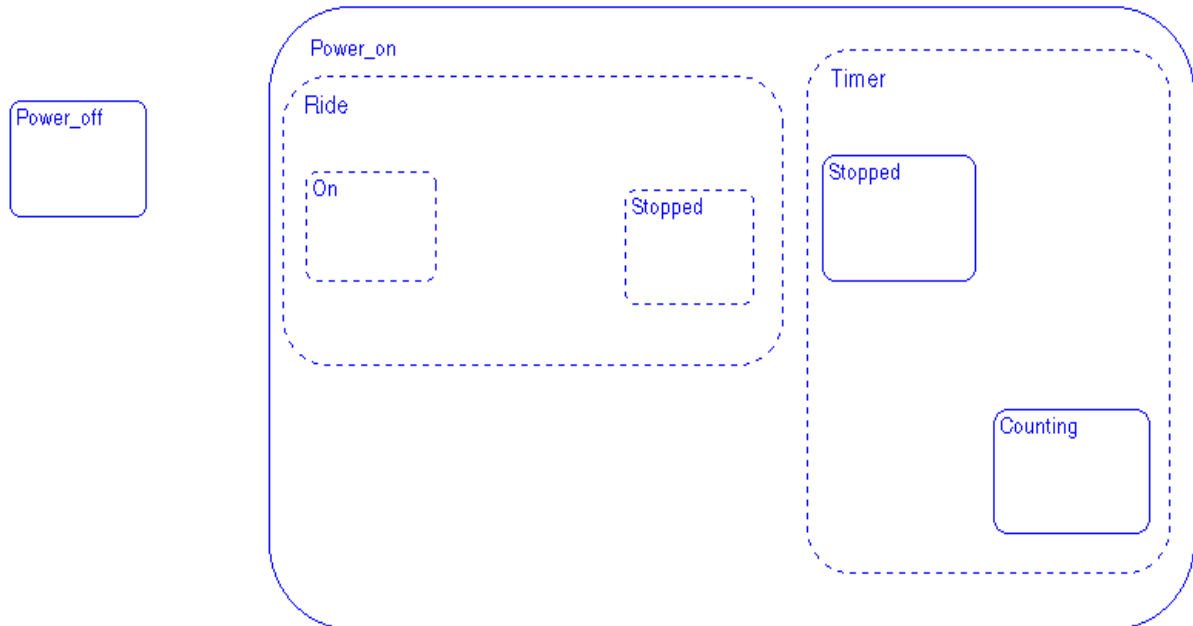
To create a superstate, create a state and resize so that it includes or surrounds the desired substates. These steps show how to create and label the superstates.

- 1 Click on the **State** button  from the toolbar. Using the sample solution as a guide, place the Ride superstate in the drawing area.
- 2 Resize the state to create the superstate. Dragging any one of the state's corners resizes the state. You know you are at the resizing location on the state border when the cursor appears as a double-ended arrow.
- 3 Click on the ? within the superstate and enter the text Ride. Deselect the superstate to exit the edit.
- 4 Click on the **State** button  and place the Power_on state in the drawing area.
- 5 Resize the Power_on state to create a superstate. Dragging any one of the state's corners resizes the state. You know you are at the resizing location on the state border when the cursor appears as a double-ended arrow.


- 6 Click on the ? within the OR superstate and enter the text `Power_on`. Deselect the OR superstate to exit the edit.
- 7 Click on the **State** button  and place the `Timer` state in the drawing area.
- 8 Resize the `Timer` state to create a superstate. Dragging any one of the state's corners resizes the state. You know you are at the resizing location on the state border when the cursor appears as a double-ended arrow.
- 9 Click on the ? within the OR superstate and enter the text `Timer`. Deselect the OR superstate to exit the edit.
- 10 Display the state shortcut menu by placing the cursor inside the borders of the `Power_on` state but outside the borders of any of `Power_on`'s substates and clicking the right mouse button. Choose the **Parallel (AND) decomposition** menu item.

Sample Snapshot

Your Stateflow diagram should now look similar to this Stateflow diagram.

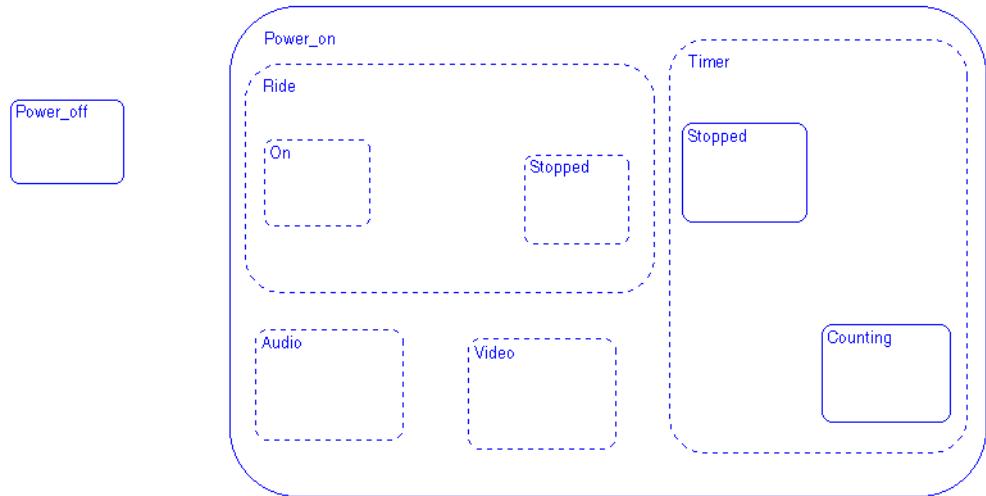


Create and Label the AND States

- 1 Double-click on the **State** button  in the toolbar. The button appears depressed and is now in multiple object mode. Using the sample solution Stateflow diagram as a guide, click in the drawing area to place the Audio and Video states in the drawing area. The states are created as AND states since the decomposition of Ride is already set. Press the **Escape** key or click on the **State** button to toggle the button out of multiple object mode.
- 2 Click on the ? character within each state to enter each state label. Use the sample solution to label the states based on their placement in the drawing area. Deselect the state (press the **Escape** key) to exit the edit.

Sample Snapshot

Your Stateflow diagram should look similar to this Stateflow diagram.



Create the Transitions

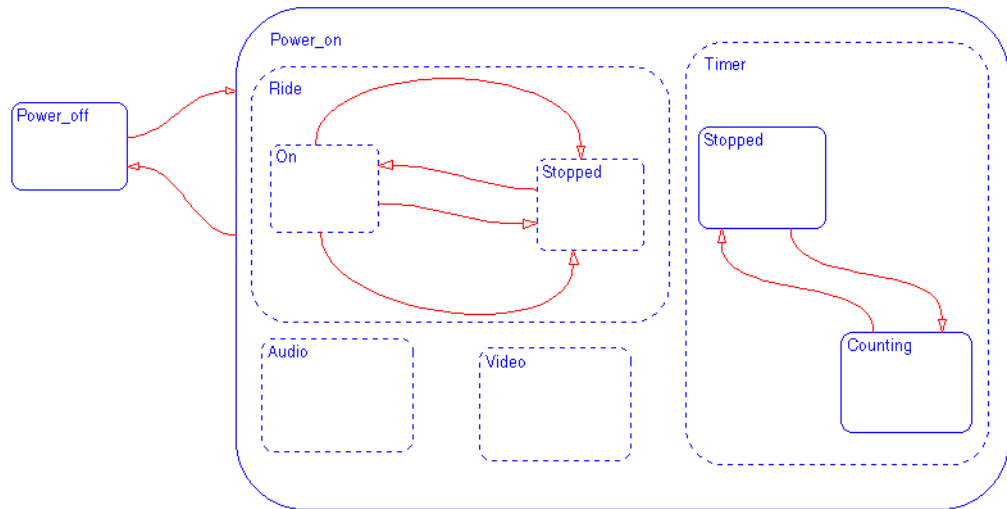
These steps show how to create the transitions:

- 1 Start with the transitions within the Ride superstate. Using the solution as a guide, place the cursor at a straight portion of the side border of the Power_on.Ride.Stopped substate. Click the border when the cursor changes to a crosshair. Without releasing the mouse button, drag the mouse to a straight portion on the side border of the Power_on.Ride.On substate. When the transition snaps to the border, release the mouse button. Transitions cannot be attached on corners since corners are used for resizing. You create one transition in this step.
- 2 Create the rest of the transitions within the Ride superstate. You create three transitions in this step.
- 3 Create the transitions within the Timer superstate. You create two transitions in this step.

- 4 Create the transitions to and from the `Power_off` state. You create two transitions in this step.

Sample Snapshot

Your Stateflow diagram should now look similar to this Stateflow diagram.



Label the Transitions

The general format of a transition label is

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

In this example, there are eight transitions to label. Use the transition labels in the solution Stateflow diagram as a guide. These steps show how to label the transitions:

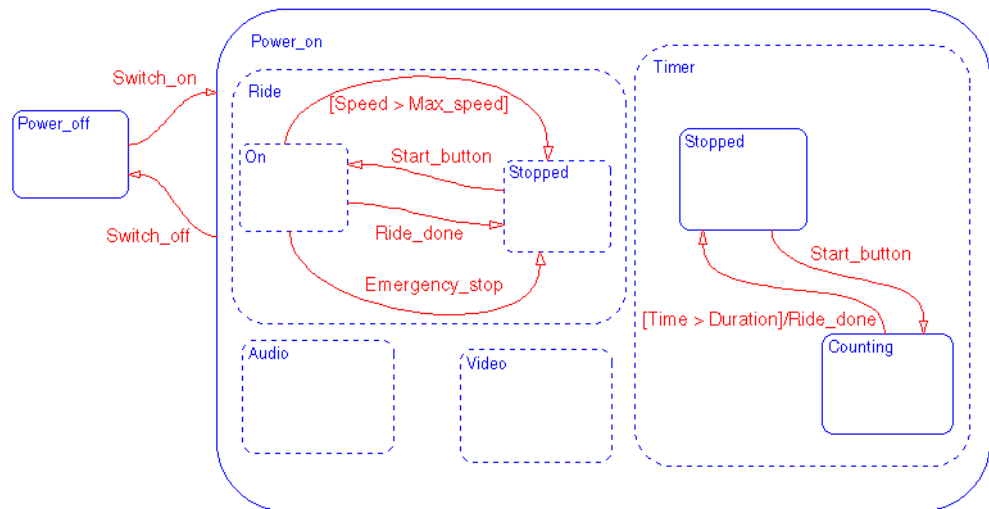
- 1 Start with the transition from the substate `Power_on.Ride.Stopped` to the substate `Power_on.Ride.On`. Select the transition. Click on the ? alongside the transition and enter the label `Start_button`. Deselect the transition label (press the **Escape** key) to exit the edit. You label one transition in this step. To move a transition label, click and drag the left mouse button to the new desired location and then release the mouse button. If you mistakenly

click and release the left mouse button, you will be in label edit mode. Deselect the label and try again.

- 2 Label the remaining transitions in the Power_on.Ride superstate. You label three transitions in this step.
- 3 Label the transitions in the Timer superstates. You label two transitions in this step.
- 4 Label the transitions to and from the Power_off substate. You label two transitions in this step.


Sample Snapshot

Your Stateflow diagram should now look similar to this Stateflow diagram.



Create Default Transitions

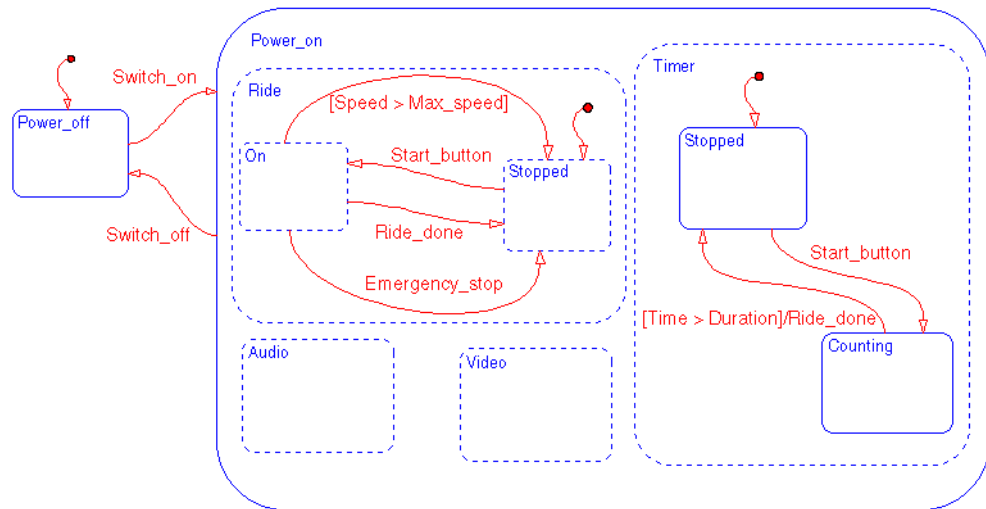
There are three Default transitions in the solution Stateflow diagram.

Double-click on the **Default transition** button  in the toolbar. The button appears depressed and is now in multiple object mode. Using the sample solution as a guide, place the default transitions in the drawing area.

This is the final step in creating the graphic objects in the Stateflow diagram.

Sample Snapshot

Your Stateflow diagram should now look similar to this Stateflow diagram.



Add and Define Event Objects

Events are added, removed, and edited using the Explorer. These events must be defined: Ride_done, Switch_on, Switch_off, Start_button, Emergency_stop.

Add and Define a Local Event

Ride_done is the only local event in the carousel example:

- 1 Choose **Explore** from the **Tools** menu of the graphics editor window to display the Explorer.
- 2 Select the parent for the new event.

Double-click on the machine name (the name you specified when you saved the Simulink model) in the **Object Hierarchy** list. Double-click on the chart

name (the name you specified on the Stateflow block). The chart will be the parent for all the events in the carousel example.

- 3 Choose **Event** from the Explorer **Add** menu. An event appears in the **Contents** pane. Position the cursor over the item's **Name** field, press the right mouse button to display the context menu, and select **Rename** from the menu.
- 4 Enter Ride_done in the **Name** field.
- 5 Position the cursor over the **Scope** field and press the left mouse button to display a list of scopes. Select **Local** in the **Scope** field.
- 6 Click on the **OK** button to save the changes and close the dialog box.

Add and Define Input from Simulink Events

There are four **Input from Simulink** events to add and define: Switch_on, Switch_off, Start_button, and Emergency_stop:

- 1 Choose **Explore** from the **Tools** menu of the graphics editor window to display the Explorer.
- 2 Select the parent for the new event.

Double-click on the machine name (the name you specified when you saved the Simulink model) in the **Object Hierarchy** list. Double-click on the chart name (the name you specified on the Stateflow block). The chart will be the parent for all the events in the carousel example.

- 3 Choose **Event** from the Explorer **Add** menu. An event appears in the **Contents** pane.
- 4 Enter Switch_on in the name field.
- 5 Select **Input from Simulink** in the **Scope** field.
- 6 Select **Either Edge** in the **Trigger** field.
- 7 Click on the **OK** button to save the changes and close the dialog box.

- 8 Repeat these steps for each **Input from Simulink** event, entering the appropriate text in the **Name** field.

Add and Define Data Objects

Data are added, removed, and edited using the Explorer. These data objects must be defined: Time, Speed, Duration, and Max_speed.

Add and Define Local Data

The Time data object is local.

- 1 Choose **Explore...** from the **Tools** menu of the graphics editor.
- 2 Select the parent for the new data object.

Double-click on the machine name (the name you specified when you saved the Simulink model) in the **Object Hierarchy** list. Double-click on the chart name (the name you specified on the Stateflow block). The chart will be the parent for all the data in the carousel example.

- 3 Choose **Data** from the Explorer **Add** menu. A data item appears in the **Contents** pane.
- 4 Enter Time in the item's **Name** field.
- 5 Select **Local** in the **Scope** field.
- 6 Click on the **OK** button to save the changes and close the dialog box.

Add and Define Input from Simulink Data

The Speed data object is **Input from Simulink**:

- 1 Choose **Explore...** from the **Tools** menu of the graphics editor.
- 2 Select the parent for the new data object.

Double-click on the machine name (the name you specified when you saved the Simulink model) in the **Object Hierarchy** list. Double-click on the chart name (the name you specified on the Stateflow block). The chart will be the parent for all the data in the carousel example.

- 3 Choose **Data** from the Explorer **Add** menu. A data item appears in the **Contents** pane.
- 4 Enter `Speed` in the item's **Name** field.
- 5 Select **Input from Simulink** in the **Scope** field.
- 6 Click on the **OK** button to save the changes and close the dialog box.

Add and Define Data Output to Simulink

`Ride_on` is the only data **Output to Simulink**:

- 1 Choose **Explore...** from the **Tools** menu of the graphics editor.
- 2 Select the parent for the new data object.

Double-click on the machine name (the name you specified when you saved the Simulink model) in the Object Hierarchy list. Double-click on the chart name (the name you specified on the Stateflow block). The chart will be the parent for all the data in the carousel example.

- 3 Choose **Data** from the Explorer **Add** menu. A data item appears in the **Contents** pane.
- 4 Enter `Ride_on` in the **Name** field.
- 5 Select **Output to Simulink** in the **Scope** field.
- 6 Since the **Scope** is **Output to Simulink**, a port number is displayed. You do not need to change the port number.
- 7 Click on the **OK** button to save the changes and close the dialog box.

Add and Define Workspace Data

`Duration` is the only **Workspace** data object:

- 1 Choose **Explore...** from the **Tools** menu of the graphics editor.
- 2 Select the parent for the new data object.

Double-click on the machine name (the name you specified when you saved the Simulink model) in the Object Hierarchy list. Double-click on the chart name (the name you specified on the Stateflow block). The chart will be the parent for all the data in the carousel example.
- 3 Choose **Data** from the Explorer **Add** menu. The new data object appears in the Explorer contents pane.
- 4 Enter `Duration` in the **Name** field.
- 5 Select **Workspace** in the **Scope** field.

Add and Define Constant Data

`Max_speed` is the only **Constant** data object:

- 1 Choose **Explore...** from the **Tools** menu of the graphics editor.
- 2 Select the parent for the new data object.

Double-click on the machine name (the name you specified when you saved the Simulink model) in the Object Hierarchy list. Double-click on the chart name (the name you specified on the Stateflow block). The chart will be the parent for all the data in the carousel example.
- 3 Choose **Data** from the Explorer **Add** menu. The new data object appears in the Explorer contents pane.
- 4 Enter `Max_speed` in the item's **Name** field.
- 5 Select **Constant** in the **Scope** field.

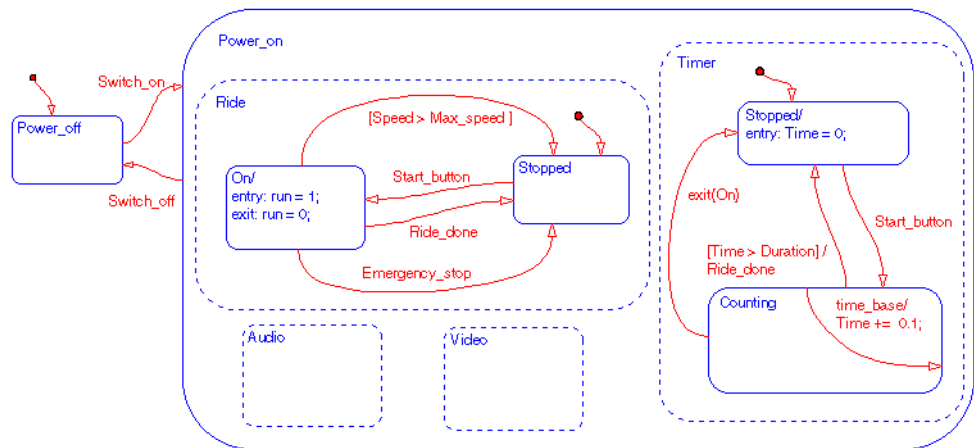
Save the Model

- 1 Choose **Save** from the graphics editor **File** menu to save the entire model.
- 2 If you want to exit from Stateflow and MATLAB, choose **Exit MATLAB** from the **File** menu of the Stateflow graphics editor.

- Stop the ride in case of emergency
- Specify whether the speed is in the normal or overspeed range

Carousel Solution Stateflow Diagram

This is the provided carousel Stateflow diagram solution.



Procedure

Follow these steps to run the simulation:

- 1 Double-click on the Examples block in the Stateflow model window. Double-click on the Carousel and then the Basic Functional Carousel block.
- 2 Choose **Start** from the graphics editor **Simulation** menu.

The first time you run the simulation, and any time you have modified the Stateflow diagram, choosing **Start** will parse the machine, generate simulation code and generate the simulation target. If the target build takes

place, informational messages are displayed in pop-up windows and in the MATLAB command window.

In any case, the graphics editor is “iced” and modifications to the Stateflow diagram are prohibited during a simulation.

- 3 The **power** switch has an off (down) and an on (up) position. Turn the power on by clicking the **power** switch from off to on. (This may require switching off, then switching on again.)
- 4 Ensure that the **running speed** switch is in the normal position.
- 5 If it is not already visible, double-click on the Scope block.
- 6 The **start** switch is a toggle. Click the **start** switch from up to down or from down to up to start the carousel. Observe the speed in the Scope block. The ride runs for `Duration` seconds but can be stopped prematurely by changing the **running speed** switch to overspeed or by clicking the **emergency** switch.
- 7 The **emergency** switch is also a toggle. Click the **emergency** switch from up to down or from down to up to bring the carousel to an emergency stop.
- 8 Choose **Stop** from the graphics editor **Simulation** menu to stop the simulation (and unfreeze the graphics editor). The simulation must be stopped to be able to close the model.
- 9 Choose **Close** from the **File** menu of the carousel model solution window.

Run the Carousel Demo

Overview

Audio and video have been added to the carousel demo. The demo also uses many advanced features that are left to your experimentation and investigation.

On UNIX systems, the audio portion of the demo is disabled.

Carousel Operator's Console

As in the carousel solution example, the switches in the Simulink model are used to act as a carousel operator's console. The switches allow you to:

- Turn the carousel power on and off
- Start the ride
- Stop the ride in case of emergency
- Specify whether the speed is in the normal or overspeed range

Procedure

Follow these steps to run the demo:

- 1 Double-click on the Examples block in the Stateflow model window. Double-click on the Carousel and then Multi-Media Carousel block.
- 2 Choose **Start** from the graphics editor **Simulation** menu.

The first time you run the simulation, and any time you have modified the Stateflow diagram, choosing **Start** will parse the machine, generate simulation code and generate the simulation target. If the target build takes place, informational messages are displayed in pop-up windows and in the MATLAB command window.

In any case, the graphics editor is “iced” and modifications to the Stateflow diagram are prohibited during a simulation.

- 3 Turn the power on by clicking the **power** switch from off to on.
- 4 Ensure that the **running speed** switch is in the normal position.

- 5 Click the **start** switch on to start the carousel. The ride runs for **Duration** seconds but can be stopped prematurely by changing the **running speed** switch to overspeed or by clicking the **emergency** switch.

If you have sound support (sound board, driver, and speakers), you should hear the music. You should also see a representation of the carousel platform rotating in XY Graph figure window. The Stateflow diagram has animation enabled, so you can see the state transitions occur as the simulation executes.

- 6 Click the **emergency** switch to bring the carousel to an emergency stop.
- 7 Choose **Stop** from the graphics editor **Simulation** menu to stop the simulation (and unfreeze the graphics editor). The simulation must be stopped to be able to close the model.
- 8 Choose **Close** from the **File** menu of the carousel demo model window.

Building Targets

Overview.	9-2
Target Types.	9-2
Building a Target.	9-2
How Stateflow Builds Targets.	9-3
Setting Up Target Build Tools	9-5
Setting Up Build Tools on UNIX.	9-5
Setting Up Build Tools on Windows	9-5
Starting a Build.	9-7
Starting from a Target Builder Dialog	9-8
Configuring a Target	9-9
Specifying Code Generation Options	9-11
Specifying Custom Code Options	9-14
Parsing	9-16
Parser	9-16
Parse the Machine or the Stateflow Diagram	9-17
Error Messages	9-21
Parser Error Messages	9-21
Code Generation Error Messages	9-22
Compilation Error Messages	9-22

Overview

A target is a program that executes a Stateflow model or a Simulink model containing a Stateflow state machine. Stateflow and companion tools can build targets for virtually any computer.

Target Types

Simulink and its companion tools can build the following types of targets:

- Simulation target

A simulation target is a compiled Simulink S-function (MEX file) that enables Simulink to simulate a Stateflow model. See “Parsing” on page 9-17 for more information.

- Stand-alone target

A stand-alone target is an executable program that enables a specific type of computer (the target machine) to execute a Stateflow model. The target can be built to run on computers that do not have a floating-point instruction set, a feature useful in embedded control applications. A stand-alone target does not require Simulink to run. Building a stand-alone target requires Stateflow Coder.

- RTW target

An RTW target is an executable program that implements 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.

Building a Target

Building a target involves the following steps:

- 1 Configure the target.

See “Configuring a Target” on page 9-9 for more information. You need to perform this step only if you are building a stand-alone or RTW target or are including custom code in the target (see “Building Custom Code into the Target” on page 9-3).

2 Start the build process.

Stateflow automatically builds or rebuilds simulation targets, when you initiate simulation of a state machine. You must explicitly initiate the build process for other types of targets. See “Starting a Build” on page 9-7 for more information.

Configuring and building a target requires a basic understanding of how Stateflow builds targets, in the case of simulation and stand-alone targets, and how Real-Time Workshop builds targets, in the case of RTW targets. See “How Stateflow Builds Targets” on page 9-3 for information on how Stateflow builds targets. Real-Time Workshop uses basically the same process for building targets that contain state machines as it uses for building targets that do not. See the *Real-Time Workshop User’s Guide* for information on how Real-Time Workshop builds targets.

Rebuilding a Target

You can rebuild a target at any time by repeating step 2. When rebuilding a target, Stateflow rebuilds only those parts corresponding to charts that have changed logically since the last build. When rebuilding a target, you need to perform step 1 only if you want to change the target’s custom code or configuration.

Building Custom Code into the Target

You can configure the target build process to include to build custom code, that is, C code supplied by you, into the target (see “Specifying Custom Code Options” on page 9-14). This capability facilitates creation of applications that integrate Stateflow state machines. In particular, it allows you to use Stateflow or Real-Time Workshop to build the entire application, including both the portions that you supply and the state machine target code generated by Stateflow (or by Real-Time Workshop and Stateflow, when building applications that include other types of Simulink blocks).

How Stateflow Builds Targets

Stateflow builds a target for a particular state machine as follows. It begins by parsing the charts that represent the state machine to ensure that the machine’s logic is valid. If any errors occur, Stateflow displays the errors in the MATLAB command window (see “Parsing” on page 9-17) and halts.

If the charts parse, Stateflow next invokes a code generator to convert the state machine into C source code. The code generator is a MATLAB M-file, named `sfc.m`, that translates a Stateflow state diagram into an equivalent C source code representation. The code generator accepts various command-line options that control the code generation process. You can specify these options via the Stateflow user interface (see “Adding a Target to a State Machine’s Target List” on page 9-9).

In the case of stand-alone targets, the code generator optionally generates a main program that exercises each of the state machine’s charts. If you do not select this option, you must provide the main program for the stand-alone target. See “Creating a Target Main Program” for more information. (Simulink serves as the main program for a simulation target. Hence, you do not need to provide a main program for a simulation target.)

The code generator also 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 “Specifying Custom Code Options” on page 9-14).

Finally Stateflow builds the target, using a C compiler and make utility that you specify (see “Setting Up Target Build Tools” on page 9-5 for more information).

Setting Up Target Build Tools

Building Simulink targets may require some initial build tool setup, depending on the platform you are using and the tools you want to use. Typically you need to perform the setup only once.

Setting Up Build Tools on UNIX

To build targets on UNIX:

- 1 Install the C compiler you want Stateflow to use to build targets on your system.

You can use any compiler supported by MATLAB for building MATLAB extension (MEX) files. See the *MATLAB Application Program Interface Guide* for information on C compilers supported by MATLAB. To access the online version of this guide, choose **Help Desk** from the MATLAB **Help** window.

- 2 Set up MATLAB to build MEX files, using the compiler installed in step 1.

See “System Setup” in the *MATLAB Application Program Interface Guide* for information on setting up MATLAB to build MEX files. Stateflow uses the compiler that you specify to build MEX files to build Stateflow targets.

Setting Up Build Tools on Windows

The Microsoft Windows version of Stateflow comes with a C compiler (`lcc.exe`) and make utility (`lccmake`). Both tools are installed in the directory `matlabroot\sys\lcc`. If you have not configured MATLAB to use any other compiler, Stateflow uses `lcc` to build targets. Thus, you do not have to perform any tool setup to build targets with the Windows version of Stateflow. If you want to use a compiler other than `lcc`, however, you must do some initial setup.

To use a compiler other than `lcc`:

- 1 Install the compiler on your system.

You can use any compiler supported by MATLAB for building MATLAB extension (MEX) files. See the *MATLAB Application Program Interface Guide* for information on C compilers supported by MATLAB. To access the

online version of this guide, choose **Help Desk** from the **MATLAB Help** window.

- 2** Set up MATLAB to build MEX files, using the compiler installed in step 1.

See “System Setup” in the *MATLAB Application Program Interface Guide* for information on setting up MATLAB to build MEX files. Stateflow uses the compiler that you specify to build MEX files to build Stateflow targets.

If you want to use a compiler that you supply to build some targets and `lcc` to build other targets, first set up MATLAB to use the compiler you supply. Then, check the **Use lcc compiler** option on the Coder dialog (see “Coder Options” on page 9-12) for each target that you want to be built with `lcc`.

Starting a Build

You can start a target build in the following ways:

- By selecting **Start** from the Stateflow or Simulink editor's **Simulation** menu or **Debug** from the Stateflow editor's **Tools** menu.

This option lets you use a single command to build and run a simulation target. Use the next 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.

- By selecting the **Build** or **Build RTW** (for RTW targets) button on the **Target Builder** dialog box for the target

You must use this option to build stand-alone targets. You can also use this option to build simulation targets and RTW targets. Using the target builder to launch the build allows you to choose between full build, incremental build, and code generation only options. See “Starting from a Target Builder Dialog Box” on page 9-8 for more information.

- By selecting the **Build** button on the RTW panel of Simulink's **Simulation Parameters** dialog box

While building a target, Stateflow displays a stream of progress messages in the MATLAB command window. You can determine the success or failure of the build by examining these messages (see “Parsing” on page 9-17).

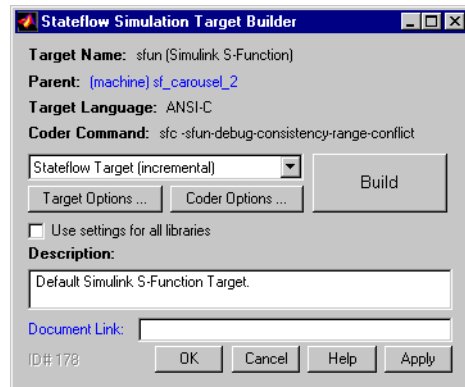
Starting from a Target Builder Dialog Box

To build a target from the **Target Builder** dialog box:

- 1 Open the **Target Builder** dialog box for the target you want to build.

You can do this by selecting the appropriate item, for example, **Open Simulation Target**, from the Stateflow editor's **Tools** menu or by clicking on the simulation target in the Stateflow Explorer.

The dialog box for the selected target appears, for example,



- 2 Select a build option from the drop-down list next to the **Build** button.
 - Choose “incremental” to rebuild only those portions of the target corresponding to charts that have changed logically since the last build.
 - Choose “all” to rebuild the target, including chart libraries, from scratch.
 - Choose “code” to regenerate code corresponding to charts that have changed logically since the rebuild.
- 3 Select the **Build** button to begin the build process.

Configuring a Target

Configuring a target entails some or all of the following steps:

- 1 Add the target, if necessary, to the state machine's target list.

See “Adding a Target to a State Machine's Target List” on page 9-9 for instructions on how to add targets to a state machine's target list.

- 2 Specify code generation options.

See “Specifying Code Generation Options” on page 9-11 for more information.

- 3 Specify custom code options.

See “Specifying Custom Code Options” on page 9-14 for more information.

- 4 Check “Apply to all Libraries” on the **Target Builder** dialog box if you want the selected options to apply to the code generated for charts imported from chart libraries.

Configuring an RTW target may require additional steps. See the *Real-Time Workshop User's Guide* for more information.

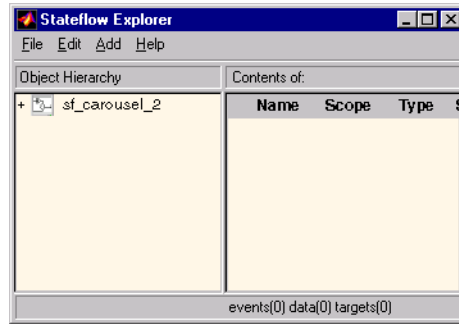
Adding a Target to a State Machine's Target List

Building a stand-alone target requires that you first add the target to the list of potential targets maintained by Stateflow for a particular model.

To add a target:

- 1 Select **Explore** from the Stateflow editor's **Tools** menu.

The Stateflow Explorer appears.



The Explorer object hierarchy shows the state machines currently loaded in memory.

- 2 Select the state machine to which you want to add the stand-alone target.

The Explorer displays the selected state machine's data, events, and targets in the contents pane.

- 3 Select **Target** from the Explorer's **Add** menu to add a target with the default name "untitled" to the selected machine.
- 4 Rename the target.

You can rename the target to any name that differs from the names of other targets targeted by the state machine. If you are adding an RTW target, you must name the target `rtw`. (A state machine can have only one RTW target.)

Renaming the Target

To rename the target:

- 1 Select the target in the Explorer's content pane and press the right mouse button.

A pop-up menu appears.

- 2 Select **Rename** from the pop-up menu.

The Explorer redisplay the selected target's name in an edit box.

- 3 Change the target's name in the edit box.

- 4 Click outside the edit box to close it.

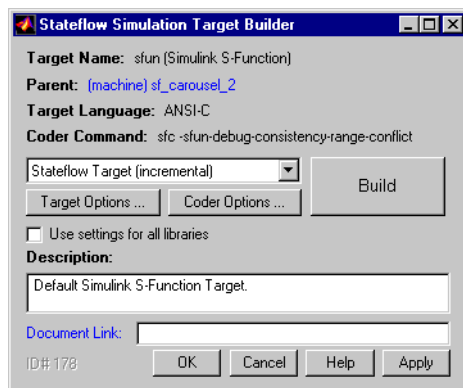
Specifying Code Generation Options

To specify code generation options:

- 1 Open the target builder dialog for the target.

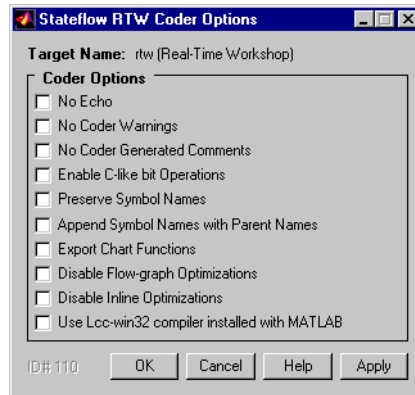
You can do this by selecting the appropriate open target item (for example, **Open Simulation Target**) from the graphics editor's **Tools** menu or by clicking on the target in the Stateflow Explorer.

The **Target Builder** dialog box for the target appears, for example,



2 Select Coder Options...

The **RTW Coder Options** dialog box for the target appears.



3 Check the desired options.

4 Select **Apply to apply the selected options or **Close** to apply the options and close the dialog.**

Coder Options

The Stateflow simulation coder provides 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.)

No Echo. Suppress display of runtime output in the MATLAB command window, specifically actions that are not terminated by a semicolon.

No Coder Warnings. Suppress display of coder warnings in the MATLAB command window.

No Coder Generated Comments. Causes the coder to omit comments from generated code.

Enable C-Like Bit Operations. Causes the coder to recognize C bit-wise operators (~, &, |, ^, >>, etc.) in action language statements and encode these operators as C bit-wise operations.

Preserve Symbol Names. Causes the coder to preserve symbol names (names of states and data) when generating code. This is useful when the target contains custom code that accesses state machine data. Note that 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. Generates a state or data name by appending the name of the item's parent to the item's name.

Export Chart Functions. Exports the names of generated functions so that they can be invoked by user-written C code.

Disable Flow-Graph Optimizations. Disables detection and elimination of redundant "goto" statements and dead code to facilitate debugging.

Disable Inline Optimizations. Disables inlining of state entry and exit functions for leaf states and optimization of state entry functions for all states. Disabling such optimizations can facilitate debugging.

Note Inline optimizations are disabled when you select **Enable Debugging/Animation** regardless of whether the **Disable Inline Optimizations** option is checked or unchecked.

Use lcc-win32 compiler installed with MATLAB. Checking this option causes Stateflow to use the lcc compiler to build this target. See "Setting Up Build Tools on Windows" on page 9-5 for more information.

Specifying Custom Code Options

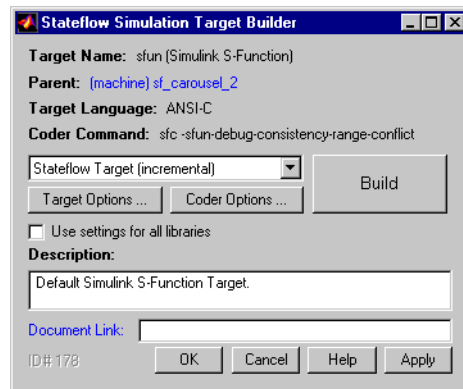
You must specify various configuration options (see “Custom Code Options” on page 9-15) to build custom code into a simulation target.

To specify the custom code options:

- 1 Open the **Target Builder** dialog box for the target in which you want to include custom code.

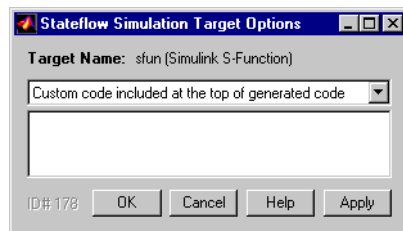
You can do this by selecting the appropriate open target item (e.g., **Open Simulation Target**) from the Stateflow editor’s **Tools** menu or by clicking on the simulation target in the Stateflow Explorer.

The **Target Builder** dialog box appears, for example,



- 2 Select **Target Options** from the dialog.

The **Target Options** dialog box appears.



The dialog box contains a drop-down list listing various options for specifying what code to include in the target and where the code is located. The edit box below the list displays the setting for the current option.

- 3 Select the options required to specify your code and enter the specifications in the edit box.

See “Custom Code Options” on page 9-15 for information on how to use these options to specify your custom code.

- 4 Select **Apply** to apply the specification to the target or **OK** to apply the specifications and close the dialog.

Custom Code Options

The target options dialog provides the following options for specifying custom code to be built into a simulation target:

Custom code included at the top of generate code. Custom C code to be included at the top of a generated header file that is included at the top of all generated source code files. In other words, all generated code sees code specified by this option. Use this option to include header files that declare custom functions and data used by generated code.

Custom include directory paths. Space-separated list of paths of directories containing custom header files to be included either directly (see first option above) or indirectly in the compiled target.

Note Stateflow ignores this option when building RTW targets. This means that all header files required for building custom code into an RTW target must reside in MATLAB’s working directory.

Custom source files. Space separated list of source files to be compiled and linked into the target.

Note Stateflow ignores this option when building RTW targets. This means that all source files required for building custom code into an RTW target must reside in MATLAB's working directory.

Custom libraries. Space-separated list of libraries containing custom object code to be linked into the target.

Custom make files. Space-separated list of custom makefiles. The Stateflow code generator includes these makefiles at the head of the makefile it generates to build the simulation target. You can use this option to include makefiles for building custom code required by the target.

Build command. The MATLAB command used to build the target.

Code command. The MATLAB command used to invoke the code generator (`sfc`, by default). You can add command-line arguments for `sfc` options not reflected on the **Coder Options** dialog box for the target.

Custom 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.

Custom termination code. Code statements that are executed at the end of simulation. You can use this code to invoke functions that free memory allocated by custom code or perform other cleanup tasks.

Parsing

Parser

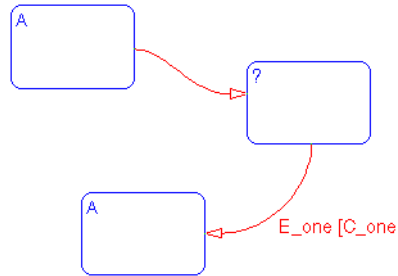
The parser evaluates the graphical and nongraphical objects in each Stateflow machine against the supported Stateflow notation and the action language syntax.

Parse the Machine or the Stateflow Diagram

Explicitly parse each Stateflow diagram in the machine by choosing **Parse** from the graphics editor **Tools** menu. Explicitly parse the current Stateflow diagram by choosing **Parse Diagram** from the graphics editor **Tools** menu. The machine is implicitly parsed when you simulate a model, build a target, or generate code.

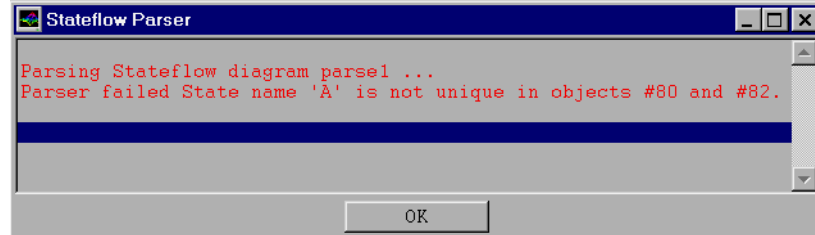
In all cases, a pop-up information window is displayed when the parsing is complete. If the parsing is unsuccessful, one error at a time is displayed (in red) in the informational window. The Stateflow diagram automatically selects and pans to the object containing the parse error. Double-click on the error in the information window to bring the Stateflow diagram to the forefront, zoom (fit to view), and select the object containing the parse error. Press the space bar to zoom back out. Fix the error and reparse the Stateflow diagram. Informational messages are also displayed in the MATLAB command window.

These steps describe parsing, assuming this Stateflow diagram.



1 Parse the Stateflow diagram.

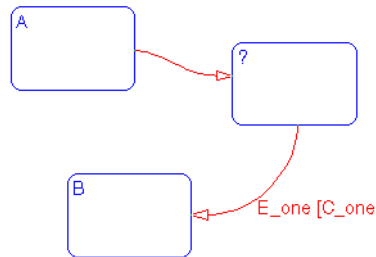
Choose **Parse Diagram** from the graphics editor **Tools** menu to parse the Stateflow diagram. State A in the upper left-hand corner is selected and this message is displayed in the pop-up window and the MATLAB command window.



2 Fix the parse error.

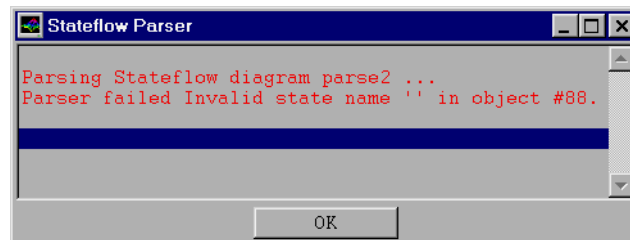
In this example, there are two states with the name A. Edit the Stateflow diagram and label the duplicate state with the text B.

The Stateflow diagram should look similar to this.



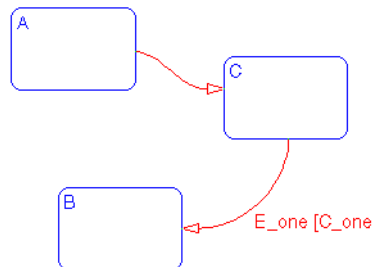
3 Reparse.

Choose **Parse Diagram** from the graphics editor **Tools** menu. This message is displayed in the pop-up menu and the MATLAB command window.



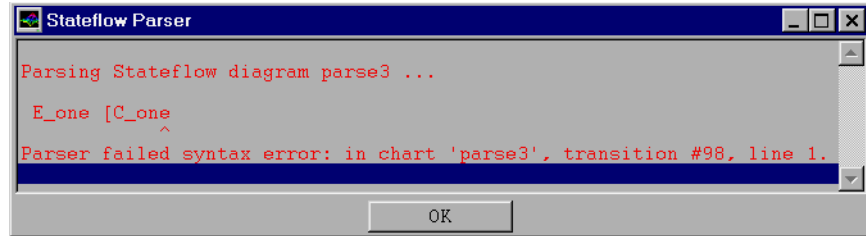
4 Fix the parse error.

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.

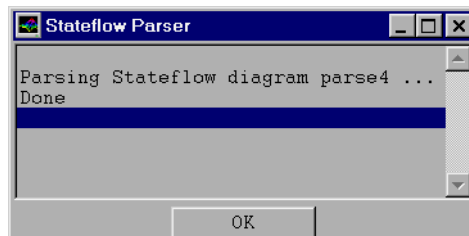
Choose **Parse Diagram** from the graphics editor **Tools** menu. This message is displayed in the pop-up window and the MATLAB command window.

**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 has no parse errors.

Error Messages

When building a target, you may see error messages from any of the following sources: the parser, the code generator, or from external build tools (make utility, C compiler, linker). Stateflow displays errors in a dialog box and in the MATLAB command window. Double-clicking on a message in the error dialog zooms the Stateflow diagram to the object that caused the error.

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:

- "Invalid state name xxx" or "Invalid event name yyy" or "Invalid data name zzz"
A state, data, or event name contains a non-alphanumeric 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:

- "Failed to create file: modelName_sfuns.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.

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.

If the code generator assumes that undefined data is defined in custom code, it generates a warning for each undefined item.

Debugging

Overview	10-2
Stateflow Debugger User Interface	10-5
Debugging Runtime Errors	10-10
Debugging State Inconsistencies	10-14
Debugging Conflicting Transitions	10-16
Debugging Data Range Violations	10-18
Debugging Cyclic Behavior	10-19

Overview

Debugging

Use the Stateflow Debugger to debug and animate the Stateflow diagrams in a particular machine.

It is a good idea to include debugging options in preliminary simulation target builds to ensure that the model is behaving as you expect, to evaluate code coverage, and to perform dynamic checking.

When you save the Stateflow diagram, all of the debugger settings (including breakpoints) are saved.

Generally speaking, debugging options should be disabled for Real-Time Workshop and stand-alone targets. The Debugger does not interact with Real-Time Workshop or stand-alone targets and the overhead incurred from the added instrumented code is undesirable.

Typical Debugging Tasks

These are some typical debugging tasks you might want to accomplish:

- Animate Stateflow diagrams, set breakpoints, and debug runtime errors
- If you are licensed for the Stateflow Dynamic Checker you can detect these conditions:
 - Evaluate code coverage
 - State inconsistencies
 - Conflicting transitions
 - Data range violations
 - Cyclic behavior

Including Debugging in the Target Build

These debugging options require supporting code additions to the target code generated:

- State inconsistency
- Transition conflict
- Data range violations

To include the supporting code for these debugging options, you must check **Enable debugging and animation** in the **Coder Options** dialog box. See “Specifying Code Generation Options” on page 9-11. You must rebuild the target for any changes made to the settings in the **Target Builder** properties dialog box to take effect. See “Target Types” on page 9-2, and “Configuring a Target” on page 9-9 for more information.

Breakpoints

A breakpoint indicates where and when the Debugger should break execution of a Stateflow diagram. The Debugger 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 instance. When simulation execution is halted at a breakpoint, you can:

- Examine the current status of the Stateflow diagram
- Step through the execution of the Stateflow diagram
- Specify display of one of these options at a time: the call stack, code coverage, data values, or active states

The breakpoints can be changed during runtime and are immediately enforced. When you save the Stateflow diagram, all of the debugger settings (including breakpoints) are saved so that the next time you open the model, the breakpoints remain as you left them.

Runtime Debugging

Once the target is built with the debugging code, you can then optionally enable or disable the associated runtime options in the Debugger. Enabling or disabling the options in the Debugger window affects the Debugger output display results. Enabling/disabling the options in the Debugger window affects the target code and can cause the target to be rebuilt when you start the simulation from the debugger.

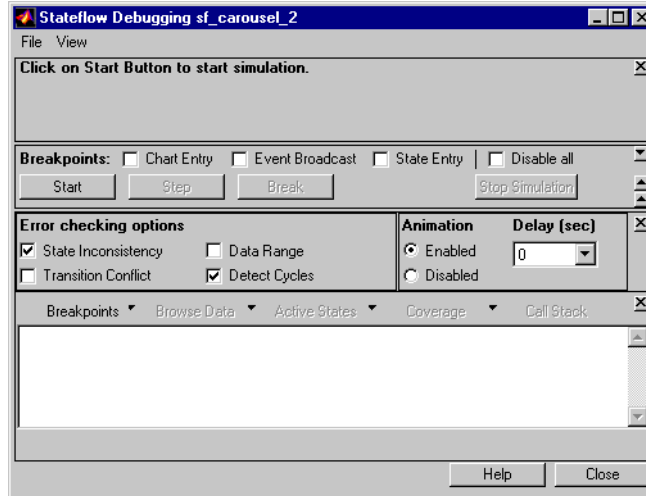
There are also some runtime debugging options that do not require supporting code in the target. These options can be dynamically set:

- Enable/disable cycle detection in the Debugger window
- Set global breakpoints at:
 - Any chart entry
 - Any event broadcast
 - Any state entry
- Enable/disable local Debugger breakpoints at specific chart or state action execution points in these appropriate property dialog boxes:
 - Chart (see “Specifying Chart Properties” on page 2-29)
 - State (see “Setting Event Properties” on page 3-4)
- Enable/disable local Debugger breakpoints at a specific transition (either when the transition is tested or when it is determined to be valid) in the **Transition property** dialog box (see “Using the Transition Properties Dialog” on page 2-24)
- Enable/disable local Debugger breakpoints based on a specific event broadcast (see “Event Properties” on page 3-5)

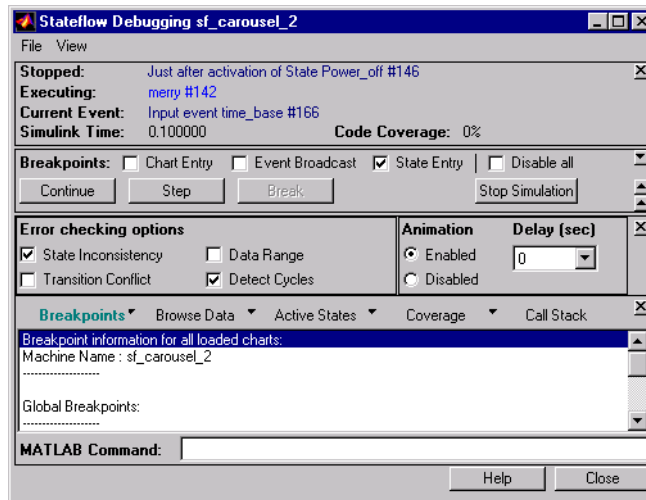
Stateflow Debugger User Interface

Debugger Main Window

This is the Debugger main window as it appears when first invoked.



This is the Debugger main window as it appears when a debug session is active.



Status Display Area

Once a debugging session is in progress, these status items are displayed in the upper portion of the Debugger window:

- The currently executing model is displayed in the **Executing** field.
- The execution point that the Debugger is halted at is displayed in the **Stopped** field. Consecutive displays of this field show each semantic step being executed.
- The event being processed is displayed in the **Current Event** field.
- The current simulation time is displayed in the **Simulink Time** field.
- The percentage of code that has been covered thus far in the simulation is displayed in the **Code Coverage** field.

Break On Controls

Use the **Break On** controls to specify global breakpoints. When a global breakpoint is encountered normal simulation execution stops and the Debugger takes control on any:

- Chart entry
Click on the **Chart Entry** check box (check is displayed when enabled) to enable this type of breakpoint.
- Event broadcast
Click on the **Event Broadcast** check box (check is displayed when enabled) to enable this type of breakpoint.
- State entry
Click on the **State Entry** check box (check is displayed when enabled) to enable this type of breakpoint.

The breakpoints can be changed during runtime and are immediately enforced. When you save the Stateflow diagram, the breakpoint settings are saved.

Debugger Action Control Buttons

Use these buttons when debugging a Stateflow machine to control the Debugger's actions:

- Go
Click on the **Go** button to have simulation execution proceed until a breakpoint (global or local) is reached. Once the **Go** button has been clicked, the Stateflow diagram is marked read-only. The appearance of the graphics editor toolbar and menus changes so that object creation is not possible. When the graphics editor is in this read-only mode, it is called “iced.”
- Step
Click on the **Step** button to single step through the simulation execution.
- Break
Click on the **Break** button to insert a breakpoint at that point in the simulation execution.
- Stop Simulation
Click on the **Stop Simulation** button to stop the simulation execution and relinquish debugging control. Once the debug session is stopped, the graphics editor toolbar and menus return to their normal appearance and operation so that object creation is again possible.

Animation Controls

Activating animation causes visual color changes (objects are highlighted in the selection color) in the Stateflow diagram based on the simulation execution.

Activate animation by turning on the **Enabled** check box. Deactivate animation by turning on the **Disabled** check box. You can specify the animation speed from a range of 0 (fast; the default) to 1 (slow) seconds.

Display Controls

Use these buttons to control the output display:

- **Call Stack**
Click on the **Call Stack** button to display a sequential list of the **Stopped** and **Current Event** status items that occur when single stepping through the simulation.
- **Coverage**
The **Coverage** button displays the current percentage of unprocessed transitions, states, etc. at that point in the simulation. The drop-down list button on the **Coverage** button lets you specify the extent of the display: coverage for the current chart only, for all loaded charts, or for all charts in the model. Click on the button's drop down list icon to display a list of coverage options: coverage for the current chart only, for all loaded charts, or for all charts in the model.
- **Browse Data**
Click on the **Browse** button to display the current value of any defined data objects.
- **Active States**
The **Active States** button displays a list of active states in the display area. Double-clicking on any state causes the graphics editor to display that state. The drop-down list button on the **Active States** button lets you specify the extent of the display: active states in the current chart only, in all loaded charts, or for all charts in the model.
- **Breakpoints**
Click on the **Breakpoints** button to display a list of the set breakpoints. The drop-down list button on the **Breakpoints** button lets you specify the extent of the display: breakpoints in the current chart only or in all loaded charts.

Once you have selected 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 debugger's **File** menu.

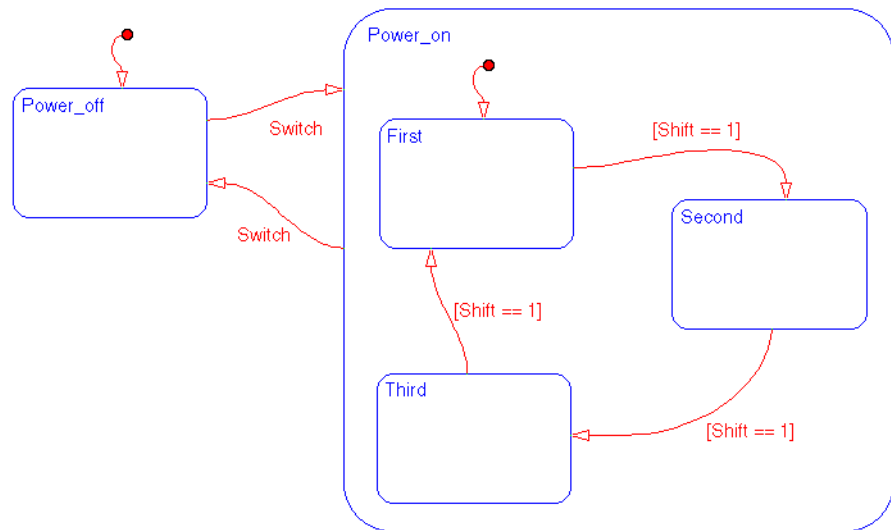
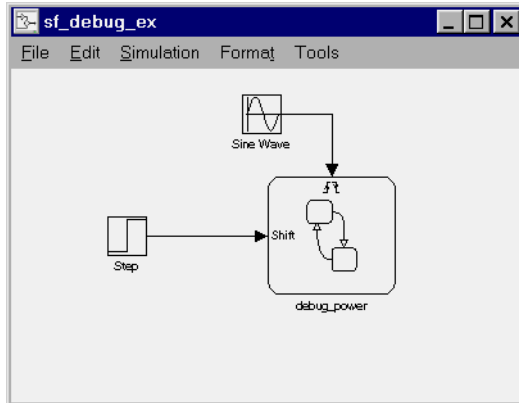
MATLAB Command Field

Direct access to the MATLAB command window is not possible while the Debugger is stopped at a breakpoint. If you need to enter any MATLAB commands during a debugging session, enter them into the **MATLAB Command** field and press the **Return** key.

Debugging Runtime Errors

Example Stateflow Diagram

This example Simulink model and Stateflow diagram is used to show how to debug some typical runtime errors.



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 being in `Power_off` and `Power_on`. `Switch` is defined as an **Input from Simulink** event. `Power_on` has three substates, `First`, `Second`, and `Third`. By default, when `Power_on` becomes active, `First` also becomes active. `Shift` is defined as an **Input from Simulink** data object. When `Shift` equals 1, the system transitions from `First` to `Second`, `Second` to `Third`, `Third` to `First`, 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`.

Typical Scenario to Debug Runtime Errors

These steps describe a typical debugging scenario to resolve runtime errors in the example model:

- 1 Create the Simulink model and Stateflow diagram (including defining the event and data objects).
- 2 Ensure the `sfun` target includes debugging options.
- 3 Invoke the Debugger and choose debugging options.
- 4 Start the simulation.
- 5 Debug the simulation execution.
- 6 Resolve runtime error, and repeat from step 3.

Create the Model and Stateflow Diagram

Using the sample (see “Example Stateflow Diagram” on page 10-10) as a guide, create the Simulink model and Stateflow diagram. Using the graphics editor **Add** menu, add the `Switch` **Input from Simulink** event and the `Shift` **Input from Simulink** data object.

Define the sfun Target

Choose **Open Simulation Target** from the **Tools** menu of the graphics editor. Ensure that the checkbox to **Enable Debugging/Animation** is enabled (checked). Click on the **Close** button to apply the changes and close the dialog box.

Invoke the Debugger and Choose Debugging Options

Choose **Debug** from the **Tools** menu of the graphics editor. Click on the **Chart entry** option under the **Break Controls** border. When the simulation begins, it will break on the entry into the chart. Click on the **Enabled** radio button under the **Animation** border to turn animation on.

Start the Simulation

Click on the **Go** 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 have specified a breakpoint on chart entry, the execution stops at that point and the Debugger display status indicates

```
Stopped: Just after entering during function of Chart debug__power
```

```
Executing: sf_debug_ex_debug_power
```

```
Current Event: Input event Switch
```

Debug the Simulation Execution

At this point, you can single step through the simulation and see whether the behavior is what you expect. Click on the **Step** button and watch the Stateflow diagram animation and the Debugger status area to see the sequence of execution.

Single stepping shows that the desired behavior is not occurring. The transitions from `Power_on.First` to `Power_on.Second`, etc., are not occurring because the transition from `Power_on` to `Power_off` takes priority. The output display of code coverage also confirms this observation.

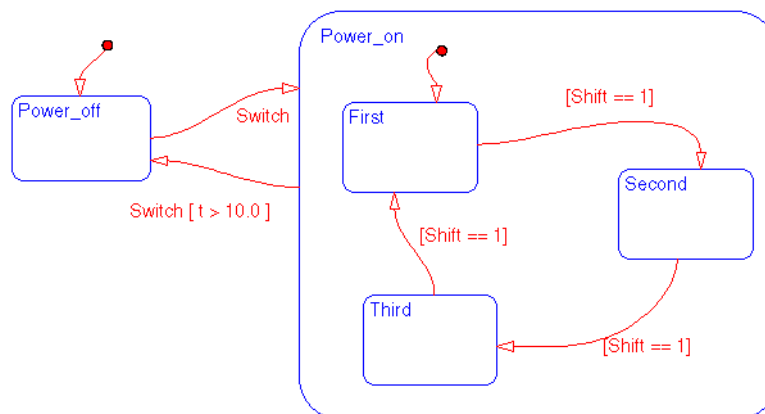
Resolve Runtime Error and Repeat

Choose **Stop** from the **Simulation** menu of the graphics editor. The Stateflow diagram is now writeable. The generation of event `Switch` is driving the simulation and the simulation time is passing too quickly for the input data object `Shift` to have an effect. The model may need to be completely rethought.

In the meantime, there is a test that verifies the conclusion. Modify the transition from `Power_on` to `Power_off` to include a condition. The transition is not to be taken until simulation time is greater than 10.0. Make this modification and click on the **Go** button to start the simulation again. Repeat the debugging single stepping and observe the behavior.

Solution Stateflow Diagram

This is the corrected Stateflow diagram with the condition added to the transition from `Power_on` to `Power_off`.



Debugging State Inconsistencies

What Is a State Inconsistency?

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 these 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 may only be detectable using the Debugger at runtime.

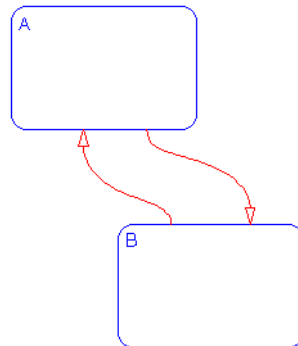
Detecting State Inconsistency

To detect the state inconsistency during a simulation:

- Build the target with debugging enabled.
- Invoke the Debugger and enable **State Inconsistency** checking
- Start the simulation

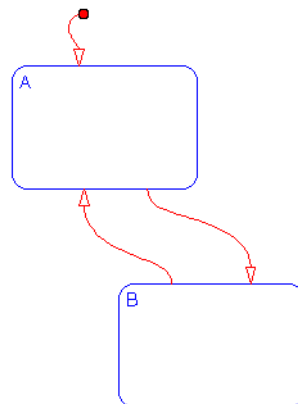
Example: State Inconsistency

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 runtime state inconsistency error.

Adding a default transition to one of the substates resolves the state inconsistency.



Debugging Conflicting Transitions

What Is a Conflicting Transition?

A transition conflict exists if, at any step in the simulation, there are two equally valid transition paths from a source to a destination. In the case of a conflict, equivalent transitions (based on their labels) are evaluated based on the geometry of the outgoing transitions. See “Execution Order” on page 7-58 for more information.

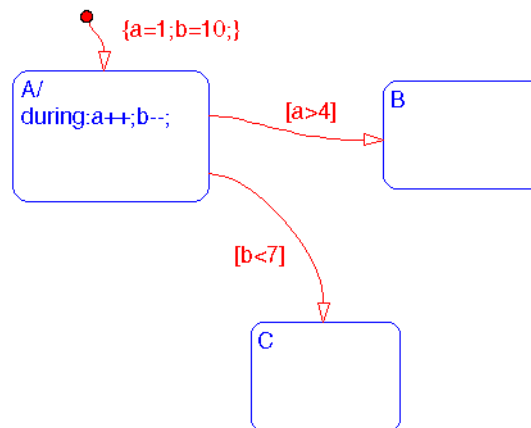
Detecting Conflicting Transitions

To detect conflicting transitions during a simulation:

- Build the target with the debugging enabled.
- Invoke the Debugger and enable **Transition Conflict** checking
- Start the simulation

Example: Conflicting Transition

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 to design your Stateflow diagram based on an expected execution order.

Debugging Data Range Violations

What Is a Data Range Violation?

Each Data property dialog box has fields for an **Initial**, **Minimum**, and **Maximum** value. If the data object equals a value outside of this range, enabling data range checking will detect the error.

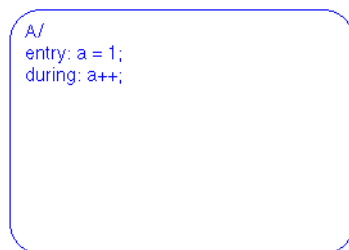
Detecting Data Range Violations

To detect data range violations during a simulation:

- Build the target with debugging enabled
- Invoke the Debugger and enable **Data Range** checking
- Start the simulation

Example: Data Range Violation

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

What Is 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.

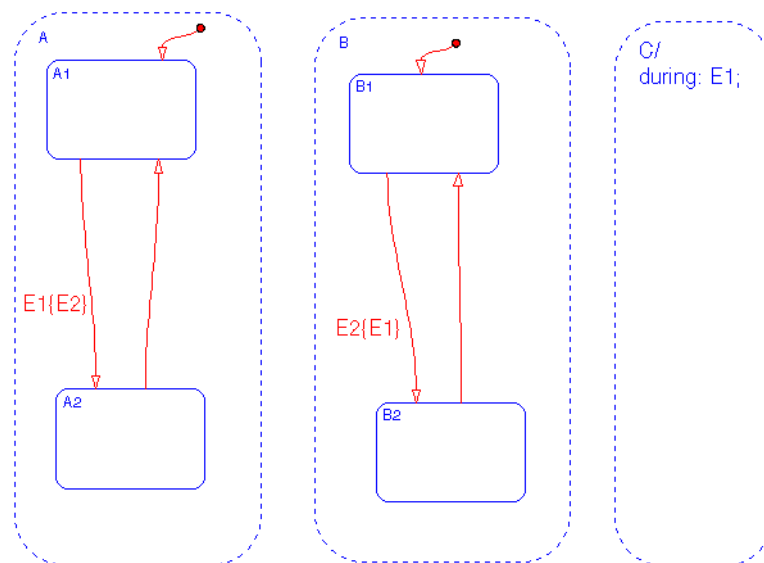
Detecting Cyclic Behavior

To detect cyclic behavior during a simulation:

- Build the target with debugging enabled
- Invoke the Debugger and enable **Detect Cycles**
- Start the simulation

Example: Cyclic Behavior

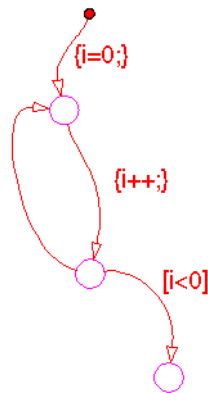
This Stateflow diagram shows a typical example of a cycle created by infinite recursions caused by an event broadcast.



When 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.

Example: Flow Cyclic Behavior Not Detected

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 zero 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 are involved with data values is not currently supported.

Example: Noncyclic Behavior Flagged as a Cycle

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 zero. When the transition is tested, the condition $[i < 5]$ is true. The condition actions, increment i and broadcast 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.

Reference

This chapter contains detailed descriptions of Stateflow functions. These functions operate on the machine.

Functions	
<code>sfnew</code>	Creates and displays a new Simulink model containing an empty Stateflow block. See <code>sfnew</code> .
<code>sfexit</code>	Closes all Stateflow diagrams, Simulink models containing Stateflow diagrams, and exits the Stateflow environment. See <code>sfexit</code> .
<code>sfsave</code>	Saves the current machine and Simulink model. See <code>sfsave</code> .
<code>stateflow</code>	Opens the Stateflow model window. See <code>stateflow</code> .

This function operates on a Stateflow diagram.

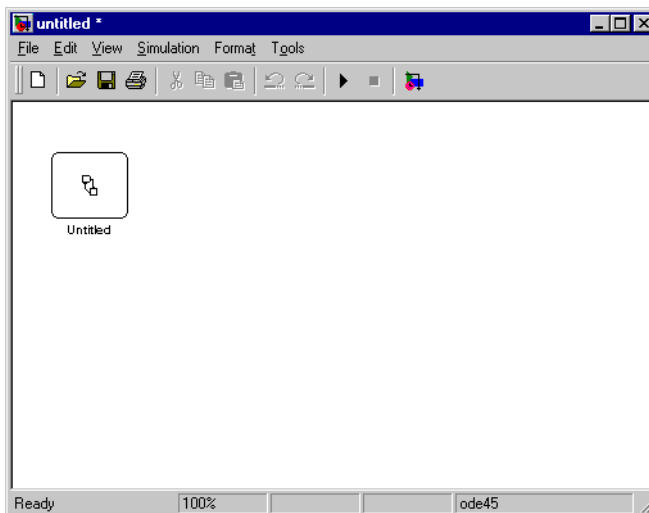
Functions	
<code>sfprint</code>	Prints the visible portion of a Stateflow diagram. See <code>sfprint</code> .

This function is independent of models and Stateflow diagrams.

Functions	
<code>sfhelp</code>	Displays Stateflow online help in an HTML browser. See <code>sfhelp</code> .

The Stateflow model window contains an empty Stateflow block, an Examples block, and a manual switch. A Stateflow block is equivalent to a Stateflow diagram.

- Purpose** Create a new Simulink model containing an empty Stateflow block.
- Syntax** `sfnew`
`sfnew modelName`
- Description** `sfnew` creates and displays a new untitled Simulink model containing an empty Stateflow block.
- `sfnew modelName` creates and displays a new Simulink model with the title specified.
- Example** Create and display a new titled Simulink model, which contains an empty Stateflow block:
- ```
sfnew example
```
- A new Simulink model, titled example, which contains an empty Stateflow block, is displayed.



# sfexit

---

**Purpose** Close all Stateflow diagrams, Simulink models containing Stateflow diagrams, and exit the Stateflow environment.

**Syntax** `sfexit`

**Description** `sfexit` closes all Stateflow diagrams, all Simulink models containing Stateflow diagrams, and exits the Stateflow environment.

`sfexit` unlocks the Stateflow MEX-file and clears it from the workspace.

**Purpose** Save the current machine and Simulink model.

**Syntax**

```
sfsave
sfsave ('machinename')
sfsave ('machine', 'saveasname')
sfsave ('defaults')
```

**Description**

sfsave saves the current machine and Simulink model.

sfsave ('machinename') saves the specified machine and its Simulink model.

sfsave ('machine', 'saveasname') saves the specified machine and its Simulink model with the specified name.

sfsave ('defaults') saves the current environment default settings in the defaults file.

# stateflow

---

**Purpose** Open the Stateflow model window.

**Syntax** `stateflow`

**Description** `stateflow` opens the Stateflow model window. The model contains an untitled Stateflow block, an Examples block, and a manual switch. The Stateflow block is a masked Simulink model and is equivalent to an empty, untitled Stateflow diagram. Use the Stateflow block to include a Stateflow diagram in a Simulink model.

Every Stateflow block has a corresponding S-function. This S-function is the agent Simulink interacts with for simulation and analysis.

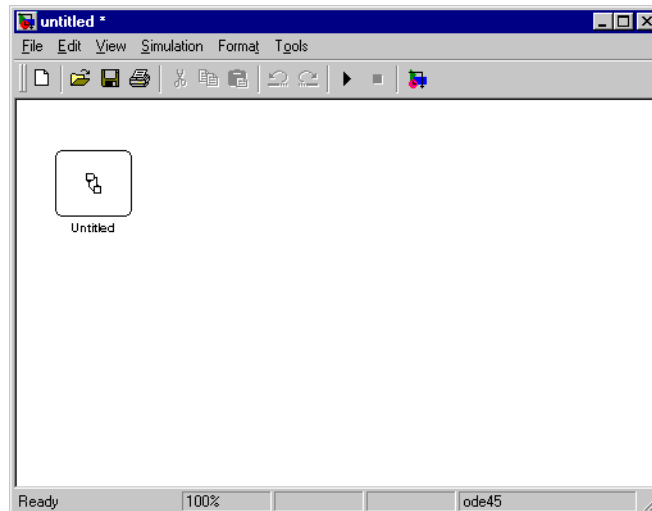
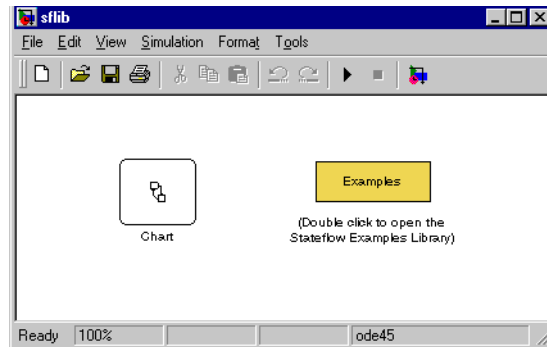
The control behavior that Stateflow models complements the algorithmic behavior modeled in Simulink block diagrams. By incorporating Stateflow blocks into Simulink models, you can add event-driven behavior to Simulink simulations. You create models that represent both data and control flow by combining Stateflow blocks with the standard Simulink and toolbox blocksets. These combined models are simulated using Simulink.

**Example** This example shows how to open the Stateflow model window and use a Stateflow block to create a Simulink model:

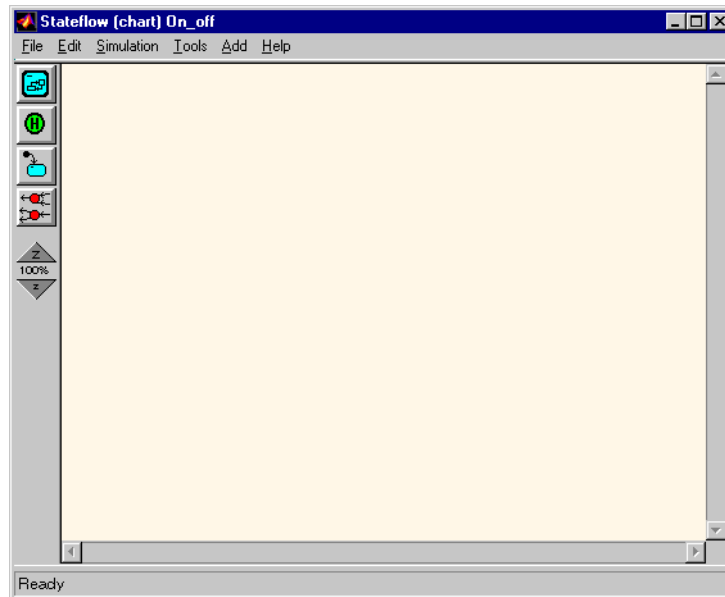
## 1 Invoke Stateflow:

```
stateflow
```

The Stateflow model window and an untitled Simulink model containing a Stateflow block are displayed.



- 2 Double-click on the untitled Stateflow block in the untitled Simulink model to invoke a Stateflow editor window.



- 3 You are ready to create the underlying Stateflow diagram.

**Purpose** Print the visible portion of the Stateflow diagram.

**Syntax**

```
sfprint
sfprint ('diagram_name','ps')
sfprint ('diagram_name','psc')
sfprint ('diagram_name','tif')
sfprint ('diagram_name','clipboard')
```

**Description** sfprint prints the visible portion of the current Stateflow diagram. A read-only preview window appears while the print operation is in progress. An informational box appears indicating the printing operation is starting.

See “Printing a Stateflow Diagram” on page 2-33, for information on printing Stateflow diagrams that are larger than the editor display area.

sfprint ('diagram\_name','ps') prints the visible portion of the specified Stateflow diagram to a postscript file.

sfprint ('diagram\_name','psc') prints the visible portion of the specified Stateflow diagram to a color postscript file.

sfprint ('diagram\_name','tif') prints the visible portion of the specified Stateflow diagram to a .tif file.

sfprint ('diagram\_name','clipboard') prints the visible portion of the specified Stateflow diagram to a clipboard bitmap (PC version only).

# sfhelp

---

**Purpose**                Display Stateflow online help in an HTML browser.

**Syntax**                sfhelp

**Description**        sfhelp invokes your HTML browser (Netscape Navigator, Internet Explorer, etc.) displaying the Stateflow online help. An online version of the *Stateflow User's Guide* is also accessible from the online help.

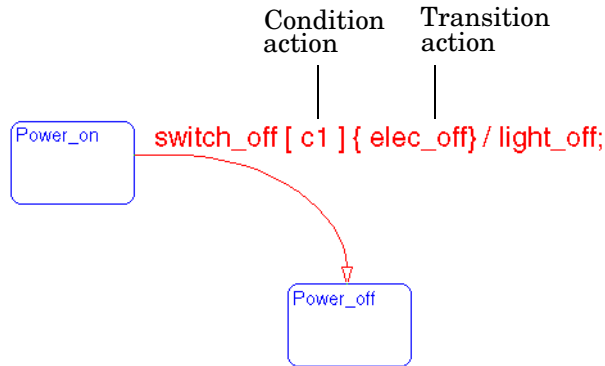
Your HTML browser must support HTML frames.

# Glossary

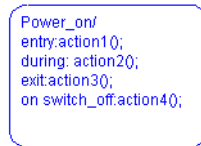
---

## Actions

*Actions* take place as 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,



States can have entry, during, exit, and, on *event\_name* actions. For example,



If you enter the name and backslash followed directly by an action or actions (without the entry keyword), the action(s) are interpreted as entry action(s). This shorthand is useful if you are only specifying entry actions.

The *action language* defines the categories of actions you can specify and their associated notations. An action can be a function call, an event to be broadcast, a variable to be assigned a value, etc. For more information, see the section titled “Action Language” on page 6-41.

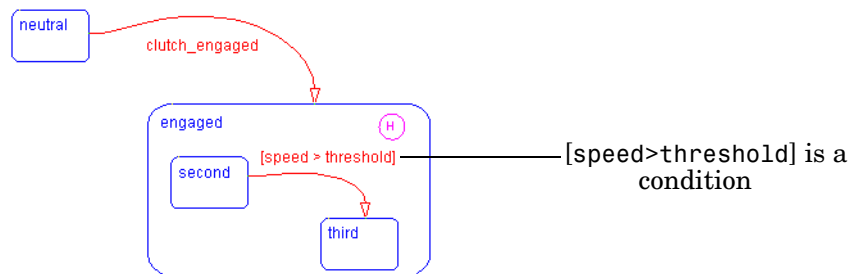
---

## Chart Instance

A *chart instance* is a 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

A *condition* is a Boolean expression to specify that a transition occurs given that the specified expression is true. For example,

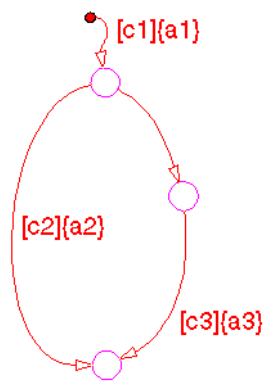


The action language defines the notation to define conditions associated with transitions. See the section titled “Action Language” on page 6-41 for more information.

## Connective Junction

*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.


This example shows how connective junctions (displayed as small circles) are used to represent the 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 the section titled “Connective Junctions” on page 6-31 for more information.

**Data**

*Data* objects store numerical values for reference in the Stateflow diagram.

See “Defining Data” on page 3-14 for more information on representing data objects.

**Data Dictionary**

The *data dictionary* is a 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 A-11.

## **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**

*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. 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 the section titled “What Is a Default Transition?” on page 6-23 for more information.

### Events

*Events* 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.

Events are added, removed and edited through the Stateflow Explorer. See the section titled “Defining Events” on page 3-2 for more information.

### Explorer

See “Stateflow Explorer” on page A-12.

### Finder

See “Stateflow Finder” on page A-12.

### Finite State Machine

A *finite state machine* (FSM) is a 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.

---

## Flow Graph

A *flow graph* is the set of flow paths that start from a transition segment that, in turn, starts from a state or a default transition segment.

## Flow Path

A *flow path* is an ordered sequence of transition segments and junctions where each succeeding segment starts on the junction that terminated the previous segment.

## Flow Subgraph


A *flow subgraph* is the set of flow paths that that start on the same transition segment.

## Hierarchy

*Hierarchy* 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 the section titled “Hierarchy” on page 1-14 for more information.

## History Junction

A *History junction* 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 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:

- “What Is a History Junction?” on page 6-39
- “Example: Default Transition and a History Junction” on page 7-20
- “Example: Labeled Default Transitions” on page 7-21
- “Example: Inner Transition to a History Junction” on page 7-29

### **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 XOR decomposition. Use of inner transitions can greatly simplify a Stateflow diagram.

See the sections titled “What Is an Inner Transition?” on page 6-26 and “Example: Inner Transition to a History Junction” on page 7-29 for more information.

### **Library Link**

A *library link* is a link to a chart that is stored in a library model in a Simulink block library.

### **Library Model**

A Stateflow *library model* is a 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 state machine. Thus, a Stateflow model that includes links to library charts has multiple state 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**

A *machine* is the collection of all Stateflow blocks defined by a Simulink model exclusive of chart instances (library links). If a model includes any library

---

links, it also includes the state machines defined by the models from which the links originate.

### **Notation**

A *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:

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

### **Parallelism**

A system with *parallelism* 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.

See the section titled “What Is a State Decomposition?” on page 6-9 for more information.

### **Real-Time Workshop**

The Real-Time Workshop is an 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 User's Guide* for more information.

### **RTW Target**

An executable built from code generated by the Real-Time Workshop. See Chapter 9, “Building Targets” for more information.

### **S-Function**

When using Simulink together with Stateflow for simulation, Stateflow generates an *S-function* (MEX-file) for each Stateflow machine to support model simulation. This generated code is a simulation target and is called the *sfun* target within Stateflow.

For more information, see the *Using Simulink* document.

### **Semantics**

*Semantics* 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.

### **Simulink**

Simulink is a 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 control flow by combining Stateflow blocks with the standard Simulink blocksets. These combined models are simulated using Simulink.


The *Using Simulink* document 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 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

The *Stateflow block* is a masked Simulink model and 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 control flow by combining Stateflow blocks with the standard Simulink and toolbox block libraries. These combined models are simulated using Simulink.

### Stateflow Debugger

Use the *Stateflow Debugger* 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 the section titled “Anatomy of a Model and Machine” on page 1-7 for more information on Stateflow diagrams.

### **Stateflow Explorer**

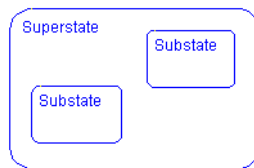
Use the *Explorer* to add, remove, and modify data, event, and target objects. See, “Stateflow Explorer User Interface” on page 5-4 for more information.

### **Stateflow Finder**

Use the *Finder* 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 on that object. See “Stateflow Finder User Interface” on page 5-8 for more information.

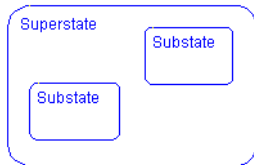
### **Substate**

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.



### **Target**

An executable program built from code generated by Stateflow or the Real-Time Workshop. See Chapter 9, “Building Targets” for more information.

---

## Topdown Processing

Topdown processing refers to the way in which Stateflow processes states. In particular, Stateflow processes superstates before states. Stateflow processes a state only if its superstate is activated first.

## Transition

A *transition* describes 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

A *transition path* is a flow path that starts and ends on a state.

## Transition Segment

A *transition segment* is a single directed edge on a Stateflow diagram. Transition segments are sometimes loosely referred to as transitions.

## Virtual Scrollbar

A *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.

See the section titled “Exploring Objects in the Editor Window” on page 2-12 for more information.



## A

- action language
  - array arguments 6-54
  - assignment operations 6-46
  - binary operations 6-43
  - bit operations 6-43
  - choosing between `m1()` and `m1` 6-53
  - comments 6-62
  - components 6-43
  - continuation symbols 6-62
  - data and event arguments 6-54
  - definition 1-18
  - directed event broadcasting 6-58
  - event broadcasting 6-56
  - glossary definition A-2
  - literals 6-61
  - objects with actions 6-41
  - pointer and address operators 6-55
  - state action notation 6-42
  - time symbol 6-61
  - transition action notation 6-42
  - unary operations 6-46
  - use of the semicolon 6-62
  - user-written functions 6-47
- actions
  - definition 1-17
  - glossary definition A-2
- address operators 6-55
- animation
  - debugger control 10-8
- array arguments 6-54
- assignment operations 6-46

## B

- binary operations 6-43
- bit operations 6-43

- breakpoints
  - global 10-3
    - chart entry 10-7
    - event broadcast 10-7
    - state entry 10-7
  - local 10-3
  - overview 10-3

## C

- carousel
  - run the demo 8-25
  - run the provided solution 8-22
  - Simulink model solution 8-22
  - solution
    - procedure to run the simulation 8-23
    - Stateflow diagram 8-23
  - tutorial tasks 8-8
- chapter quick reference 7
- chart (Stateflow diagram)
  - debugger breakpoint property 2-30
  - description property 2-30
  - document link 2-30
  - editor property 2-30
  - name property 2-30
  - parent property 2-30
  - printing, large 2-33
  - sample time property 2-30
  - Simulink subsystem property 2-30
  - update method property 2-30
  - update methods for defining interface 4-4
- code generation
  - error messages 9-22
  - notations to optimize code size 6-64
  - notations to optimize memory use 6-65
  - notations to optimize speed 6-63

- using notation efficiently 6-63
- code generation related error messages 9-21
- combination state
  - definition 6-11
  - notation 6-11
- compilation error messages 9-22
- condition
  - definition 1-15
  - glossary definition A-3
  - notation overview 6-60
- condition action
  - actions specified as condition actions semantics example 7-13
  - actions specified as condition and transition actions semantics example 7-14
  - cyclic behavior to avoid semantics example 7-17
  - using condition actions in for loop construct semantics example 7-15
- conflicting transitions 10-16
- connective junction
  - common destination notation example 6-37
  - common events notation example 6-37
  - definition 1-20
  - flow diagram semantics example 7-34
  - for loop construct semantics example 7-33
  - for loops notation example 6-34
  - from a common source notation example 6-36
  - glossary definition A-3
  - if - then - else decision construct semantics example 7-31
  - notation overview 6-31
  - self loop notation example 6-33
  - self loop semantics example 7-32
  - transitions based on a common event semantics example 7-38

- transitions from a common source to multiple destinations semantics example 7-36
- transitions from multiple sources semantics example 7-37
- with all conditions specified notation example 6-32
- with one unconditional transition notation example 6-32

- continuation symbols 6-62
- cycle detection 10-19

## D

- data
  - constant 3-19
  - definition 1-14
  - description property 3-20
  - exported 4-28
  - exported to an external code source 3-19
  - glossary definition A-4
  - imported 4-29
  - imported from an external code source 3-19
  - input from Simulink 3-18
  - local 3-18
  - output to Simulink 3-18
  - temporary 3-18
  - workspace 3-18
- data dictionary
  - definition 6-4
  - glossary definition A-4
- data input from Simulink
  - add and choose a parent task 4-18
  - apply and save task 4-19
  - choose scope task 4-18
  - specify data attributes task 4-19
  - tasks 4-18
- data output to Simulink

- add and choose a parent task 4-22
- apply and save task 4-23
- choose scope task 4-22
- specify data attributes task 4-23
- tasks 4-22
- data range violation 10-18
- data scope and parent combinations 3-22
- Debugger
  - action control buttons 10-7
  - active states display 10-8
  - animation controls 10-8
  - break button 10-7
  - break on controls 10-6
  - breakpoints 10-3
  - breakpoints display 10-8
  - browse data display 10-8
  - call stack display 10-8
  - clear output display 10-8
  - conflicting transitions
    - definition 10-16
    - detecting 10-16
    - example 10-16
  - coverage display 10-8
  - cyclic behavior
    - definition 10-19
    - detecting 10-19
    - example 10-19
  - data range violation
    - definition 10-18
    - detecting 10-18
    - example 10-18
  - debugging conflicting transitions 10-16
  - debugging cycle detection 10-19
  - debugging data range violations 10-18
  - debugging runtime errors example 10-10
    - choose options task 10-12
    - create model task 10-11
    - debug simulation task 10-12
    - define sfun task 10-12
    - example Stateflow diagram 10-10
    - resolve errors task 10-13
    - solution Stateflow diagram 10-13
    - start simulation task 10-12
    - typical tasks 10-11
  - debugging state inconsistencies 10-14
  - display controls 10-8
  - Dynamic Checker options 10-8
  - flow cyclic behavior
    - example 10-20
  - glossary definition A-11
  - Go button 10-7
  - including in the target build 10-2
  - main window 10-5
  - MATLAB command field 10-9
  - overview 10-2
  - runtime debugging 10-3
  - state inconsistency
    - causes 10-14
    - definition 10-14
    - detecting 10-14
    - example 10-15
  - status display area 10-6
  - Step button 10-7
  - Stop simulation button 10-7
  - typical tasks 10-2
  - user interface 10-5
- debugging
  - overview 10-2
  - runtime 10-3
- decomposition
  - glossary definition A-5

- specifying 2-14
- default transition
  - and a history junction semantics example 7-20
  - creating 2-22
  - definition 1-19
  - glossary definition A-5
  - in an exclusive (OR) decomposition semantics example 7-18
  - labeled default transition semantics example 7-21
  - notation example 6-24
  - notation labeling 6-23
  - notation overview 6-23
  - to a junction notation example 6-25
  - to a junction semantics example 7-19
  - with a label notation example 6-25
- directed event broadcasting
  - send semantics example 7-54
  - using qualified event names notation example 6-60
  - using qualified event names semantics example 7-56
  - using send notation example 6-59

## E

- edge trigger type
  - either edge 3-11
  - falling edge 3-11
  - rising edge 3-11
- edge triggered
  - defining output events 4-13
  - example output event semantics 4-13
- either edge triggered 3-11
- error messages
  - code generation 9-22

- code generation related 9-21
- compilation 9-22
- parsing 9-21
- target building 9-22
- event
  - broadcasting 6-56
  - definition 1-13
  - directed event broadcast using qualified names 6-60
  - directed event broadcasting 6-58
  - exported 4-25
  - imported 4-27
  - index property 3-8
  - representing hierarchy 6-8
  - trigger property 3-8
- event actions and superstates semantics example 7-40
- event broadcast
  - state action notation example 6-57
  - transition action notation example 6-57
- event input from Simulink
  - add and choose parent task 4-16
  - apply the changes task 4-17
  - choose scope task 4-16
  - select the trigger task 4-17
  - tasks 4-16
- event output to Simulink
  - add and choose a parent task 4-20
  - apply and save task 4-21
  - choose scope task 4-20
  - task overview 4-10
  - tasks 4-20
- event triggers
  - defining edge triggered 4-13
  - defining function call 4-10
  - defining output to Simulink 4-8
  - edge triggered semantics example 4-13

- function call example 4-10
- function call semantics 4-11
- events
  - glossary definition A-6
- exclusive (OR) state notation 6-10
- Explorer
  - add menu 5-6
  - contents of list 5-4
  - deleting objects from 5-7
  - edit menu 5-5
  - file menu 5-5
  - help menu 5-6
  - main window 5-4
  - moving objects to change index and port order 5-7
  - moving objects to change parents 5-7
  - object hierarchy list 5-4
  - overview 5-2
  - user interface 5-4
- exploring and searching overview 5-2
- exported
  - data 4-28
  - event 4-25
- external code sources
  - defining interface for 4-25
  - definition 4-25
- F**
- falling edge triggered 3-11
- Finder
  - dialog box 5-8
    - clear button 5-10
    - close button 5-10
    - display area 5-11
    - help button 5-11
    - matches field 5-10
    - object type 5-9
    - refine button 5-10
    - representing hierarchy 5-12
    - search button 5-10
    - search history list 5-10
    - search method 5-9
    - string criteria field 5-8
  - glossary definition A-12
  - overview 5-2
  - user interface 5-8
- finite state machine
  - glossary definition A-6
  - references 1-6
  - representations 1-5
  - what is 1-5
- flow diagram
  - for loop notation example 6-34
  - notation example 6-35
  - notation optimization 6-65
  - notation overview 6-31
  - overview example 6-35
- for loop
  - notation example 6-34
  - semantics example 7-33
- function call
  - defining output event 4-10
  - example output event semantics 4-11
  - output event example 4-10
- functions
  - sfexit 4
  - sfhelp 10

- sfprint 9
- sfsave 5
- stateflow 6
- user-written 6-47

## G

- graphical objects
  - copying 2-11
  - cutting and pasting 2-10
- graphics editor
  - object button modes 2-13

## H

- Hexadecimal 6-56
- hexadecimal notation 6-56
- hierarchy 1-14
  - definition 1-14
  - events 6-8
  - glossary definition A-7
  - state 6-7
  - transition 6-8
- history junction
  - and a default transition semantics example 7-20
  - and an inner transition semantics example 7-29
  - definition 1-16
  - glossary definition A-7
  - notation overview 6-39
  - use of history junctions notation example 6-39

## I

- if-then-else

- another notation example 6-32
- notation example 6-32
- semantics example 7-31
- implicit local events
  - example 3-12
  - overview 3-11
- imported
  - data 4-29
  - event 4-27
- inner transition
  - before using an inner transition(1) notation example 6-26
  - before using an inner transition(2) notation example 6-28
  - glossary definition A-8
  - notation overview 6-26
  - processing a second event with an inner transition to a connective junction semantics example 7-27
  - processing a second event within an exclusive (OR) state semantics example 7-24
  - processing a third event within an exclusive (OR) state semantics example 7-25
  - processing one event with an inner transition to a connective junction semantics example 7-26
  - processing one event within an exclusive (OR) state semantics example 7-23
  - to a connective junction(1) notation example 6-27
  - to a connective junction(2) notation example 6-28
  - to a history junction notation example 6-29
  - to a history junction semantics example 7-29
- installation 9

**J**

## junction

- changing incoming arrowhead size 2-27
- changing size 2-26
- description property 2-28
- document link property 2-28
- editing properties 2-27
- moving 2-27
- parent property 2-28
- properties 2-28

**K**

## keywords

- change(data\_name) 6-5
- during 6-5
- entry 6-5
- entry(state\_name) 6-5
- exit 6-5
- exit(state\_name) 6-5
- in(state\_name) 6-5
- matlab() 6-6
- matlab. 6-6
- on event\_name 6-5
- send(event\_name,state\_name) 6-6
- summary list 6-5

**L**

## literals 6-61

**M**

## machine

- glossary definition A-8

## MATLAB

- requirements for 2

## ml() functions 6-48

## ml. name space operator 6-51

**N**

## notation

## action language

- array arguments 6-54
  - assignment operations 6-46
  - binary operations 6-43
  - bit operations 6-43
  - choosing between ml() and ml. 6-53
  - comments 6-62
  - components 6-43
  - condition overview 6-60
  - continuation symbols 6-62
  - data and event arguments 6-54
  - directed event broadcasting 6-58
  - event broadcasting 6-56
  - literals 6-61
  - objects with actions 6-41
  - state action 6-42
  - time symbol 6-61
  - transition action 6-42
  - unary actions 6-46
  - unary operations 6-46
  - use of semicolon 6-62
  - user-written functions 6-47
- connective junction overview 6-31
- default transition definition 6-23
- default transition example 6-24
- default transition labeling 6-23
- definition 1-6, 6-2
- designing for efficient code 6-63
- flow diagram overview 6-31
- glossary definition A-9
- graphical objects 6-3

- history junction overview 6-39
  - history junctions and inner transitions overview 6-40
  - how is the notation checked 6-3
  - inner transition overview 6-26
  - keywords 6-5
  - motivation for 6-2
  - nongraphical objects 6-4
  - representing hierarchy 6-6
  - self loop transition overview 6-30
  - state
    - during action 6-14
    - entry action 6-13
    - exit action 6-14
    - labeling example 6-13
    - name 6-13
    - on action 6-14
  - transition
    - label example 6-17
    - labeling 6-17
    - valid label 6-18
  - transition definition 6-16
  - transition label
    - condition 6-17
    - condition action 6-18
    - event 6-17
    - transition action 6-18
  - transition types 6-19
- P**
- parallel (AND) state
    - event broadcast condition action semantics example 7-50
    - event broadcast state action semantics example 7-42
    - event broadcast transition action semantics example 7-46
  - notation 6-10
  - parallelism
    - definition 1-18
    - glossary definition A-9
  - parsing
    - error messages 9-21
    - how to 9-17
    - tasks 9-18
  - Pointer 6-55
  - pointer operators 6-55
  - prerequisites 2
- Q**
- quick start
    - creating a Simulink model 1-22
    - creating a Stateflow diagram 1-25
    - debugging the Stateflow diagram 1-32
    - defining the Stateflow interface 1-28
    - generating code 1-34
    - overview 1-21
    - running a simulation 1-30
    - sample solution 1-21
    - Stateflow typical tasks 1-21
- R**
- Real-Time Workshop
    - glossary definition A-9
    - overview 6
  - references 1-6
  - regular expressions 5-9
  - rising edge triggered 3-11

**S**

## searching

- based on multiple conditions example 5-20
- Finder user interface 5-8
- for a wildcard string example 5-14
- for all objects of a specified object type example 5-18
- for an exact string match example 5-16
- for an object to edit properties example 5-22

## self loop

- notation example 6-33
- notation overview 6-30

## semantics

- definition 1-6
- execution order 7-58
- glossary definition A-10

## send

- keyword 6-6
- notations example 6-59
- semantics example 7-54

## sfexit 4

## sfhelp 10

## sfprint 9

## sfsave 5

## S-function glossary definition A-9

## Simulink

- glossary definition A-10
- overview 5
- requirements for 2

## Simulink model and Stateflow machine

- relationship between 1-7

## state

- active and inactive notation 6-10
- active notation 6-10
- changing incoming transition arrowhead size 2-19
- combination notation 6-11

## debugger breakpoint property 2-17

## decomposition notation 6-9

## definition 1-11

## description property 2-17

## document link property 2-17

## editing properties 2-21

## exclusive (OR) decomposition notation 6-10

## glossary definition A-10

## grouping and ungrouping 2-19

## inactive notation 6-10

## label property 2-17

## labeling notation 6-12

## moving 2-21

## name property 2-16

## notation

- decomposition 6-9
- during action 6-14
- entry action 6-13
- exit action 6-14
- labeling example 6-13
- name 6-13
- on action 6-14

## notation overview 6-9

## operations 2-14

## parallel (AND) decomposition notation 6-10

## representing hierarchy 6-7

## resizing 2-14

## showing and hiding 2-14

## specifying decomposition 2-14

## unique name notation example 6-14

## state inconsistencies 10-14

## state label

- changing font size 2-19

## Stateflow

- applications, types of 1-2
- component overview 1-3
- defining interfaces overview 1-9

- design approaches 1-3
  - feature overview 1-2
  - representations 1-5
  - stateflow 6
  - Stateflow block
    - considerations in choosing continuous update 4-7
    - continuous 4-4
    - continuous example 4-8
    - defining a continuous 4-7
    - defining a sampled 4-5
    - defining a triggered 4-5
    - defining an inherited 4-6
    - inherited 4-4
    - inherited example 4-6
    - sampled 4-4
    - sampled example 4-6
    - triggered 4-4
    - triggered example 4-5
    - update methods 4-4
  - Stateflow diagram
    - glossary definition A-11
    - graphical components 1-11
    - objects 1-10
    - update methods 4-4
  - Stateflow interfaces
    - overview 4-2
    - typical tasks to define 4-2
  - substate glossary definition A-12
  - superstate glossary definition A-12
- T**
- target
    - building error messages 9-22
  - time 6-61
  - transition
    - changing arrowhead size 2-23
    - creating and deleting 2-21
    - debugger breakpoint property 2-25
    - debugging conflicting 10-16
    - default 2-21
    - definition 1-12
    - description property 2-25
    - destination property 2-25
    - document link property 2-25
    - editing attach points 2-22
    - editing properties 2-24
    - glossary definition A-13
    - label format 2-22
    - label property 2-25
    - labeling 2-22
    - notation
      - label example 6-17
      - labeling 6-17
      - types 6-19
      - valid label 6-18
    - notation definition 6-16
    - operations 2-21
    - parent property 2-25
    - properties 2-25
    - representing hierarchy 6-8
    - source property 2-25
    - to and from exclusive (OR) states(1) semantics example 7-8
    - to and from exclusive (OR) states(2) semantics example 7-9
    - to and from exclusive (OR) states(3) semantics example 7-10
    - to and from junctions notation example 6-20
    - to and from OR states notation example 6-20
    - to and from OR superstates notation example 6-21
    - to and from substates notation example 6-22

- valid labels 2-23
- transition label
  - changing font size 2-23
  - condition 2-22
  - condition action 2-22
  - event 2-22
  - moving 2-24
  - notation
    - condition 6-17
    - condition action 6-18
    - event 6-17
    - transition action 6-18
  - transition action 2-22
- tutorial
- carousel
  - add and define data task 8-19
  - add and define events task 8-17
  - basic operation 8-2
  - create a Simulink model task 8-8
  - create and label AND states task 8-13
  - create and label OR states task 8-10
  - create and label superstates task 8-11
  - create default transitions task 8-16
  - create the transitions task 8-14
  - design questions 8-4
  - determining data 8-7
  - determining default transitions 8-7
  - determining events 8-7
  - determining state decomposition 8-5
  - determining state labels 8-6
  - determining states 8-4
  - determining transitions 8-6
  - label the transitions task 8-15
  - objectives 8-2
  - overview 8-2
  - provided models 8-2
  - sample Stateflow diagram 8-9
  - save the model task 8-21
  - task overview 8-8
- typographical conventions 8

## U

- unary actions 6-46
- unary operations 6-46
- user-written functions 6-47

## V

- virtual scrollbar
  - glossary definition A-13