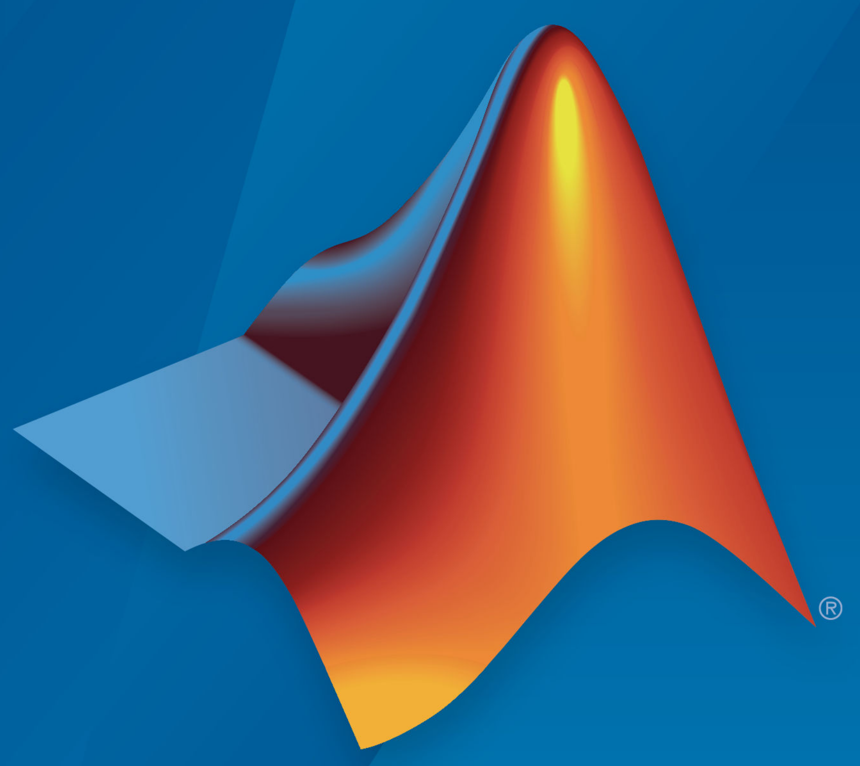
Stateflow®

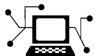
Getting Started Guide



MATLAB® & SIMULINK®

R2019a

How to Contact MathWorks



Latest news: www.mathworks.com

Sales and services: www.mathworks.com/sales_and_services

User community: www.mathworks.com/matlabcentral

Technical support: www.mathworks.com/support/contact_us

Phone: 508-647-7000

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

Stateflow® Getting Started Guide

© COPYRIGHT 2004–2019 by The MathWorks, Inc.

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

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

Trademarks

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

Patents

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

Revision History

June 2004	First printing	New for Version 6.0 (Release 14)
October 2004	Online only	Revised for Version 6.1 (Release 14SP1)
March 2005	Online only	Revised for Version 6.2 (Release 14SP2)
September 2005	Online only	Revised for Version 6.3 (Release 14SP3)
October 2005	Reprint	Version 6.0
March 2006	Second printing	Revised for Version 6.4 (Release 2006a)
September 2006	Reprint	Version 6.5 (Release 2006b)
March 2007	Online only	Rereleased for Version 6.6 (Release 2007a)
September 2007	Third printing	Rereleased for Version 7.0 (Release 2007b)
March 2008	Fourth printing	Revised for Version 7.1 (Release 2008a)
October 2008	Fifth printing	Revised for Version 7.2 (Release 2008b)
March 2009	Sixth printing	Revised for Version 7.3 (Release 2009a)
September 2009	Online only	Revised for Version 7.4 (Release 2009b)
March 2010	Online only	Revised for Version 7.5 (Release 2010a)
September 2010	Online only	Revised for Version 7.6 (Release 2010b)
April 2011	Seventh printing	Revised for Version 7.7 (Release 2011a)
September 2011	Online only	Revised for Version 7.8 (Release 2011b)
March 2012	Online only	Revised for Version 7.9 (Release 2012a)
September 2012	Online only	Revised for Version 8.0 (Release 2012b)
March 2013	Online only	Revised for Version 8.1 (Release 2013a)
September 2013	Online only	Revised for Version 8.2 (Release 2013b)
March 2014	Online only	Revised for Version 8.3 (Release 2014a)
October 2014	Online only	Revised for Version 8.4 (Release 2014b)
March 2015	Online only	Revised for Version 8.5 (Release 2015a)
September 2015	Online only	Revised for Version 8.6 (Release 2015b)
October 2015	Online only	Rereleased for Version 8.5.1 (Release 2015aSP1)
March 2016	Online only	Revised for Version 8.7 (Release 2016a)
September 2016	Online only	Revised for Version 8.8 (Release 2016b)
March 2017	Online only	Revised for Version 8.9 (Release 2017a)
September 2017	Online only	Revised for Version 9.0 (Release 2017b)
March 2018	Online only	Revised for Version 9.1 (Release 2018a)
September 2018	Online only	Revised for Version 9.2 (Release 2018b)
March 2019	Online only	Revised for Version 10.0 (Release 2019a)

Contents

1 Introduction to the Stateflow Product

Stateflow Product Description **1-2**

Model Finite State Machines **1-3**
Example of a Stateflow Chart **1-3**
Execute Chart as a MATLAB Object **1-4**
Simulate Chart as a Simulink Block With Local Events **1-6**
Simulate Chart as a Simulink Block With Temporal Conditions **1-9**

Construct and Run a Stateflow Chart **1-14**
Construct the Stateflow Chart **1-14**
Simulate the Chart as a Simulink Block **1-17**
Execute the Chart as a MATLAB Object **1-19**

Define Chart Behavior by Using Actions **1-23**
State Action Types **1-24**
Transition Action Types **1-25**
Examine Chart Behavior **1-26**

Create a Hierarchy to Manage System Complexity **1-29**
State Hierarchy **1-29**
Example of Hierarchy **1-29**
Simplify Chart Appearance by Using Subcharts **1-32**
Explore the Example **1-34**

Model Synchronous Subsystems by Using Parallelism **1-36**
State Decomposition **1-36**
Example of Parallel Decomposition **1-36**
Order of Execution for Parallel States **1-38**
Explore the Example **1-39**

Synchronize Parallel States by Broadcasting Events **1-41**
Broadcasting Local Events **1-41**
Example of Event Broadcasting **1-41**
Coordinate with Other Simulink Blocks **1-43**
Explore the Example **1-45**

Monitor Chart Activity by Using Active State Data **1-49**
Active State Data **1-49**
Example of Active State Data **1-50**
Behavior of Traffic Controller Subcharts **1-52**
Explore the Example **1-55**

Schedule Chart Actions by Using Temporal Logic **1-60**
Temporal Logic Operators **1-60**
Example of Temporal Logic **1-61**
Timing of Bang-Bang Cycle **1-62**
Timing of Status LED **1-64**
Explore the Example **1-67**

Installing Stateflow Software **1-70**
Installation Instructions **1-70**
Prerequisite Software **1-70**
Product Dependencies **1-70**
Set Up Your Own Target Compiler **1-71**
Using Stateflow Software on a Laptop Computer **1-71**

2 The Stateflow Chart You Will Build

The Stateflow Chart **2-2**

How the Stateflow Chart Works with the Simulink Model **2-6**

A Look at the Physical Plant **2-7**

Running the Model **2-9**

3 Defining the Interface to the Simulink Model

Implementing the Interface with Simulink **3-2**
Build It Yourself or Use the Supplied Model **3-2**
Design Considerations for Defining the Interface **3-2**
Adding a Stateflow Block to a Simulink Model **3-3**
Defining the Inputs and Outputs **3-8**
Connecting the Stateflow Block to the Simulink Subsystem . . **3-15**

4 Defining the States for Modeling Each Mode of Operation

Implementing the States to Represent Operating Modes **4-2**
Build It Yourself or Use the Supplied Model **4-2**
Design Considerations for Defining the States **4-2**
Adding the Power On and Power Off States **4-6**
Adding and Configuring Parallel States **4-8**
Adding the On and Off States for the Fans **4-13**

5 Defining Transitions Between States

Adding the Transitions **5-2**
Build It Yourself or Use the Supplied Model **5-2**
Design Considerations for Defining Transitions Between States **5-2**
Drawing the Transitions Between States **5-4**
Adding Default Transitions **5-7**
Adding Conditions to Guard Transitions **5-10**
Adding Events to Guard Transitions **5-11**

6 Triggering a Stateflow Chart

Implementing the Triggers **6-2**
Build It Yourself or Use the Supplied Model **6-2**
Design Considerations for Triggering Stateflow Charts **6-2**
Defining the CLOCK Event **6-3**
Connecting the Edge-Triggered Events to the Input Signals . . **6-4**

7 Simulating the Chart

Setting Simulation Parameters and Breakpoints **7-2**
Prepare the Chart Yourself or Use the Supplied Model **7-2**
Checking That Your Chart Conforms to Best Practices **7-2**
Setting the Length of the Simulation **7-3**
Configuring Animation for the Chart **7-4**
Setting Breakpoints to Observe Chart Behavior **7-5**
Simulating the Air Controller Chart **7-5**

8 Debugging the Chart

Debugging Common Modeling Errors **8-2**
Debugging State Inconsistencies **8-2**
Debugging Data Range Violations **8-4**

Introduction to the Stateflow Product

This chapter describes the basics of Stateflow event-based modeling software and its components.

- “Stateflow Product Description” on page 1-2
- “Model Finite State Machines” on page 1-3
- “Construct and Run a Stateflow Chart” on page 1-14
- “Define Chart Behavior by Using Actions” on page 1-23
- “Create a Hierarchy to Manage System Complexity” on page 1-29
- “Model Synchronous Subsystems by Using Parallelism” on page 1-36
- “Synchronize Parallel States by Broadcasting Events” on page 1-41
- “Monitor Chart Activity by Using Active State Data” on page 1-49
- “Schedule Chart Actions by Using Temporal Logic” on page 1-60
- “Installing Stateflow Software” on page 1-70

Stateflow Product Description

Model and simulate decision logic using state machines and flow charts

Stateflow provides a graphical language that includes state transition diagrams, flow charts, state transition tables, and truth tables. You can use Stateflow to describe how MATLAB® algorithms and Simulink® models react to input signals, events, and time-based conditions.

Stateflow enables you to design and develop supervisory control, task scheduling, fault management, communication protocols, user interfaces, and hybrid systems.

With Stateflow, you model combinatorial and sequential decision logic that can be simulated as a block within a Simulink model or executed as an object in MATLAB. Graphical animation enables you to analyze and debug your logic while it is executing. Edit-time and run-time checks ensure design consistency and completeness before implementation.

Model Finite State Machines

Stateflow is a graphical programming environment based on finite state machines. With Stateflow, you can test and debug your design, consider different simulation scenarios, and generate code from your state machine.

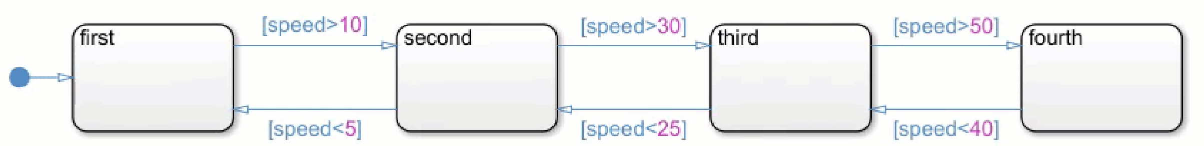
Finite state machines are representations of dynamic systems that transition from one mode of operation (state) to another. State machines:

- Serve as a high-level starting point for a complex software design process.
- Enable you to focus on the operating modes and the conditions required to pass from one mode to the next mode.
- Help you to design models that remain clear and concise even as the level of model complexity increases.

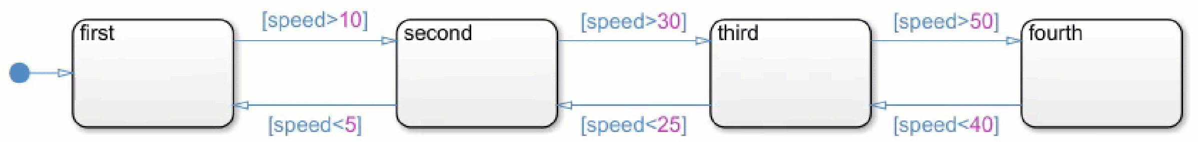
Control systems design relies heavily on state machines to manage complex logic. Applications include designing aircraft, automobiles, and robotics control systems.

Example of a Stateflow Chart

In a Stateflow chart, you combine states, transitions, and data to implement a finite state machine. This Stateflow chart presents a simplified model of the logic to shift gears in a four-speed automatic transmission system of a car. The chart represents each gear position by a state, shown as a rectangle labeled `first`, `second`, `third`, or `fourth`. Like the gears they represent, these states are exclusive, so only one state is active at a time.



The arrow on the left of the diagram represents the default transition and indicates the first state to become active. When you execute the chart, this state is highlighted on the canvas. The other arrows indicate the possible transitions between the states. To define the dynamics of the state machine, you associate each transition with a Boolean condition or a trigger event. For example, this chart monitors the speed of the car and shifts to a different gear when the speed crosses a fixed threshold. During simulation, the highlighting in the chart changes as different states become active.



This chart offers a simple design that disregards important factors such as engine speed and torque. You can construct a more comprehensive and realistic model by linking this Stateflow chart with other components in MATLAB or Simulink. Following are three possible approaches.

Execute Chart as a MATLAB Object

This example presents a modified version of an automatic transmission system that incorporates state hierarchy, temporal logic, and input events.

- **Hierarchy:** The chart consists of a superstate `gear_logic` that surrounds the four-speed automatic transmission chart in the previous example. This superstate controls the speed and acceleration of the car. During execution, `gear_logic` is always active.
- **Temporal Logic:** In the state `gear_logic`, the action `on every(0.25,sec)` determines the speed of the car. The operator `every` creates a MATLAB timer that executes the chart and updates the chart data `speed` every 0.25 seconds.
- **Input Events:** The input events `SpeedUp`, `Cruise`, and `SlowDown` reset the value of the chart data `delta`. This data determines whether the car accelerates or maintains its speed at each execution step.

{speed = 0;
delta = 0;}

gear_logic
on every(0.25,sec): speed = max(0,min(speed+delta,100));
on SlowDown: delta = -1;
on Cruise: delta = 0;
on SpeedUp: delta = 1;

first
widgets(1);

[speed>10]

[speed<5]

second
widgets(2);

[speed>30]

[speed<25]

third
widgets(3);

[speed>50]

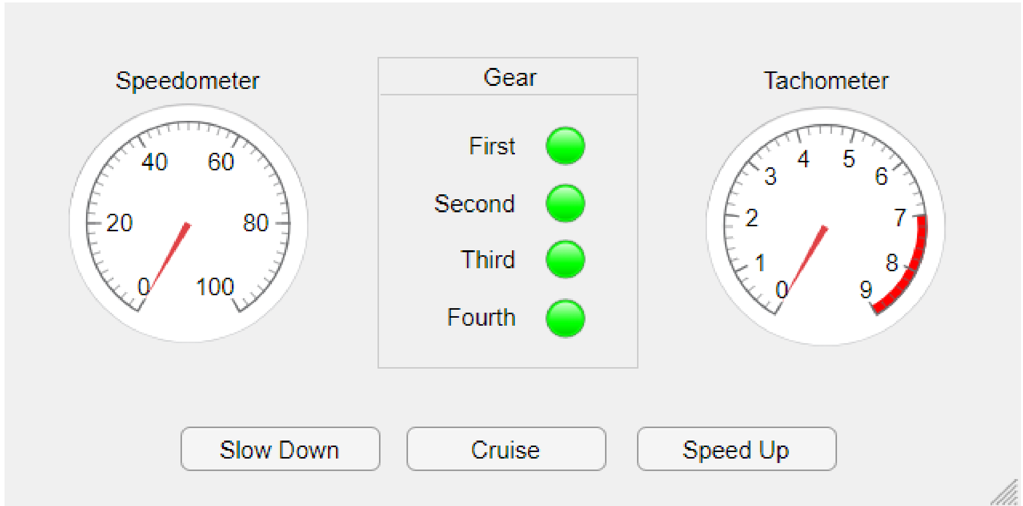
[speed<40]

fourth
widgets(4);

MATLAB Function
widgets(n)

Copyright 2018-2019 The MathWorks, Inc.

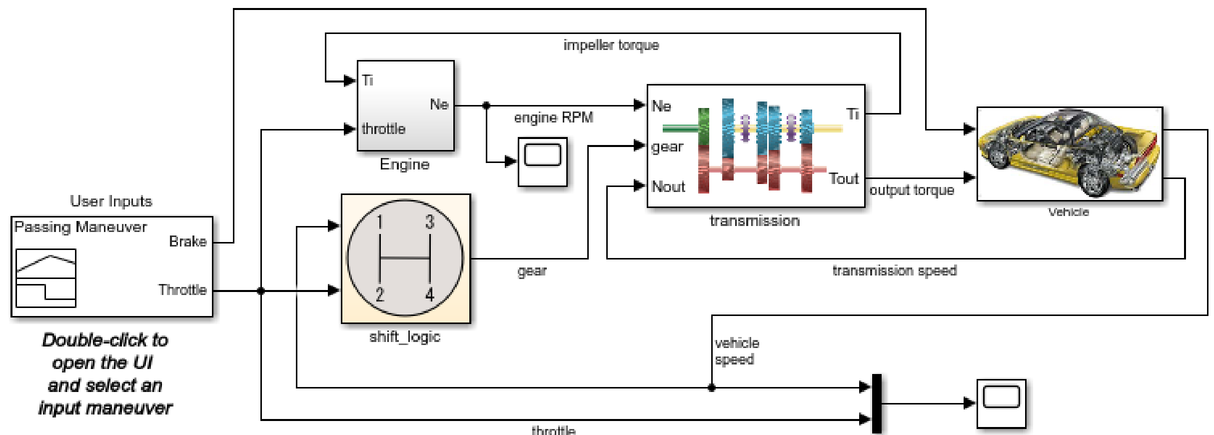
You can execute this chart as an object in MATLAB directly through the Command Window or by using a script. You can also program a MATLAB app that controls the state of the chart through a graphical user interface. For example, this user interface sends an input event to the chart when you click a button. In the chart, the MATLAB function `widgets` controls the values of the gauges and lamps on the interface.



The chart continues to run until you close the user interface window. For more information about executing Stateflow charts as MATLAB objects, see “Execution in MATLAB”.

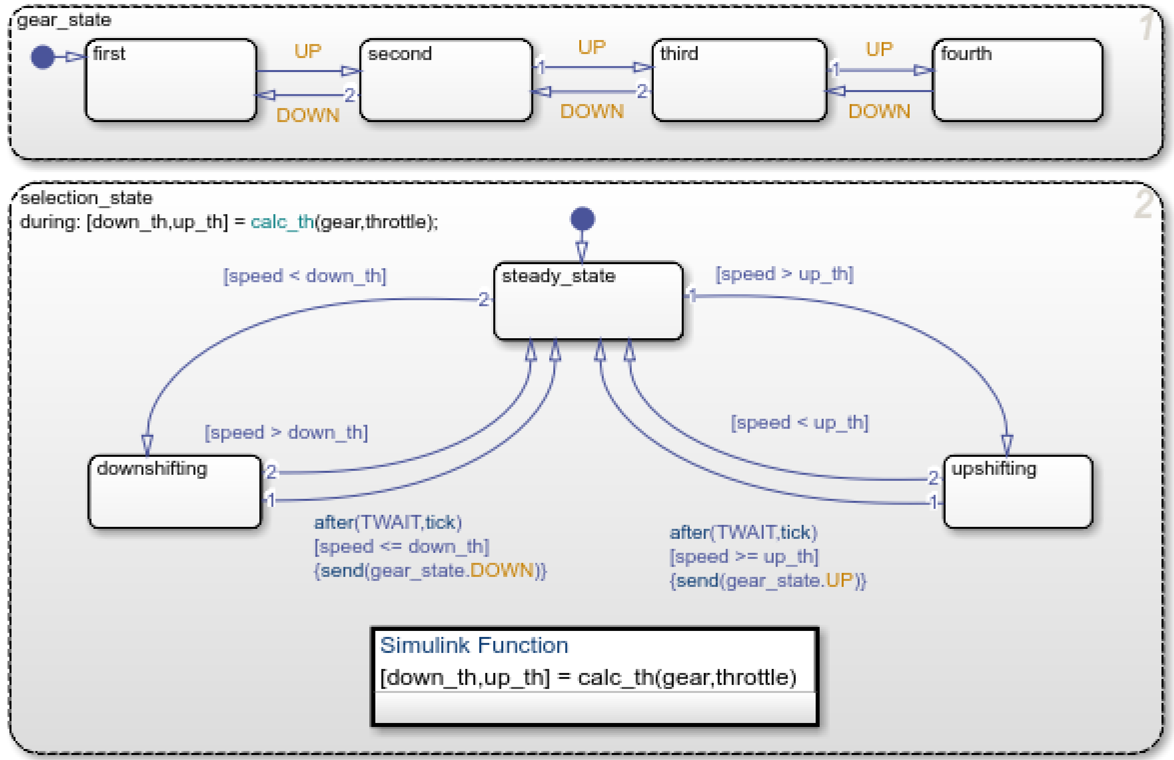
Simulate Chart as a Simulink Block With Local Events

This example provides a more complex design for an automatic transmission system. The Stateflow chart appears as a block in a Simulink model. The other blocks in the model represent related automotive components. The chart interfaces with the other blocks by sharing data through input and output connections. To open the chart, click the arrow in the bottom left corner of the `shift_logic` block.



This chart combines state hierarchy, parallelism, active state data, local events, and temporal logic.

- **Hierarchy:** The state `gear_state` contains a modified version of the four-speed automatic transmission chart. The state `selection_state` contains substates that represent the steady state, upshifting, and downshifting modes of operation. When circumstances require a shift to a higher or lower gear, these states become active.
- **Parallelism:** The parallel states `gear_state` and `selection_state` appear as rectangles with a dashed border. These states operate simultaneously, even as the substates inside them turn on and off.
- **Active State Data:** The output value `gear` reflects the choice of gears during simulation. The chart generates this value from the active substate in `gear_state`.
- **Local Events:** In place of Boolean conditions, this chart uses the local events `UP` and `DOWN` to trigger the transitions between gears. These events originate from the `send` commands in `selection_state` when the speed of the car goes outside the range of operation for the selected gear. The Simulink function `calc_th` determines the boundary values for the range of operation based on the selected gear and the engine speed.
- **Temporal Logic:** To prevent a rapid succession of gear changes, `selection_state` uses the temporal logic operator `after` to delay the broadcasting of the `UP` and `DOWN` events. The state broadcasts one of these events only if a change of gears is required for longer than some predetermined time `TWAIT`.

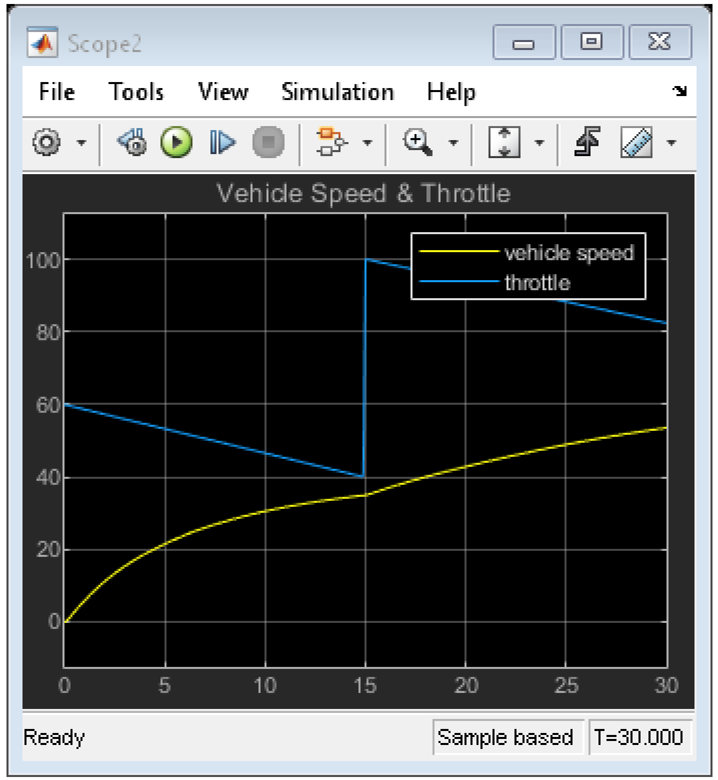


To run a simulation of the model:

1 Double-click the **User Inputs** block. In the Signal Builder dialog box, you can select a predefined brake-to-throttle profile to simulate or create a custom profile. The default profile is `Passing Maneuver`.

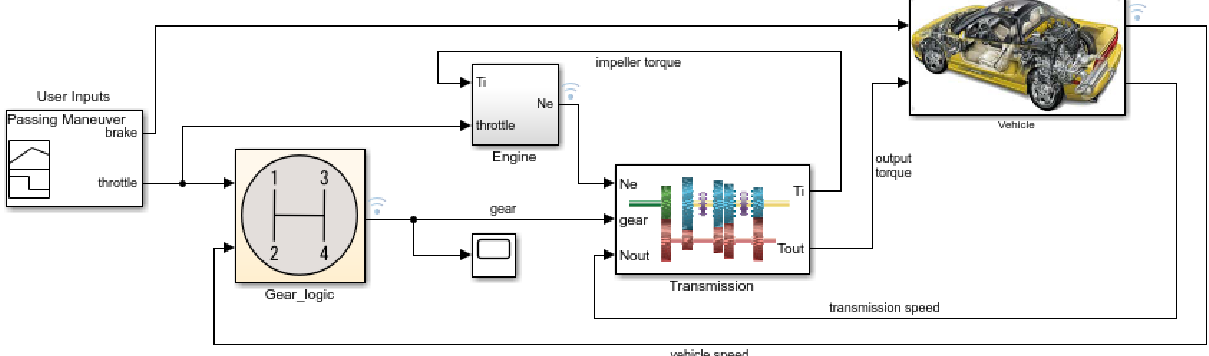
2 Click the **Run** icon. In the Stateflow Editor, chart animation highlights the active states during the simulation. You can slow down the animation speed by selecting **Display** > **Stateflow Animation** > **Slow**.

3 In the Scope blocks, examine the results of the simulation. Each scope displays a graph of its input signals during simulation.



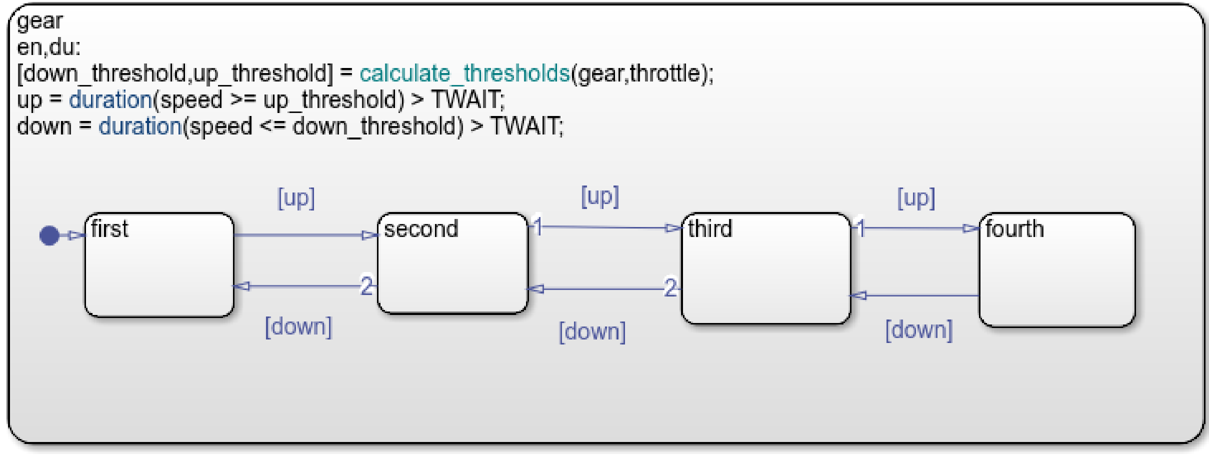
Simulate Chart as a Simulink Block With Temporal Conditions

This example provides another alternative for modeling the transmission system in a car. The Stateflow chart appears as a block in a Simulink model. The other blocks in the model represent related automotive components. The chart interfaces with the other blocks by sharing data through input and output connections. To open the chart, click the arrow in the bottom left corner of the `Gear_logic` block.



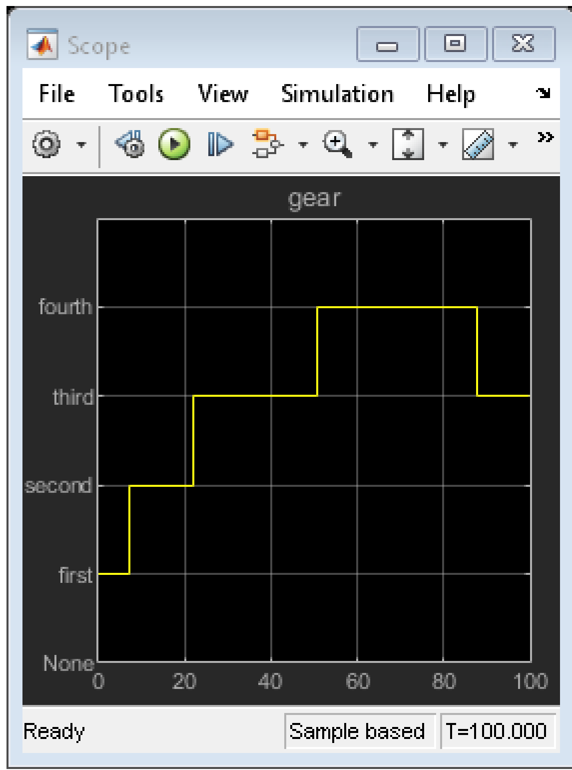
This chart combines state hierarchy, active state data, and temporal logic.

- **Hierarchy:** This model places the four-speed automatic transmission chart inside a superstate `gear`. The superstate monitors the vehicle and engine speeds and triggers gear changes. The actions listed on the upper left corner of the state gear determine the operating thresholds for the selected gear and the values of the Boolean conditions `up` and `down`. The label `en,du` indicates that the state actions are executed when the state first becomes active (`en` = `entry`) and at every subsequent time step while the state is active (`du` = `during`).
- **Active State Data:** The output value `gear` reflects the choice of gears during simulation. The chart generates this value from the active substate in `gear`.
- **Temporal Logic:** To prevent a rapid succession of gear changes, the Boolean conditions `up` and `down` use the temporal logic operator `duration` to control the transition between gears. The conditions are valid when the speed of the car remains outside the range of operation for the selected gear longer than some predetermined time `TWAIT` (measured in seconds).



To run a simulation of the model:

1. Double-click the **User Inputs** block. In the Signal Builder dialog box, you can select a predefined brake-to-throttle profile to simulate or create a custom profile. The default profile is `Passing Maneuver`.
2. Click the **Run** icon. In the Stateflow Editor, chart animation highlights the active states during the simulation. You can slow down the animation speed by selecting **Display** > **Stateflow Animation** > **Slow**.
3. In the Scope block, examine the results of the simulation. The scope displays a graph of the gear selected during simulation.



See Also

after | duration | every | send

More About

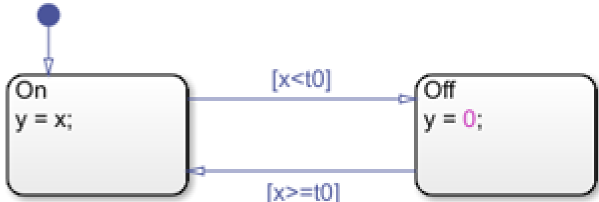
- "Construct and Run a Stateflow Chart" on page 1-14
- "Define Chart Behavior by Using Actions" on page 1-23
- "Create a Hierarchy to Manage System Complexity" on page 1-29
- "Model Synchronous Subsystems by Using Parallelism" on page 1-36
- "Synchronize Parallel States by Broadcasting Events" on page 1-41
- "Monitor Chart Activity by Using Active State Data" on page 1-49

- "Schedule Chart Actions by Using Temporal Logic" on page 1-60

Construct and Run a Stateflow Chart

A Stateflow chart is a graphical representation of a finite state machine consisting of states, transitions, and data. You can create a Stateflow chart to define how a MATLAB algorithm or a Simulink model reacts to external input signals, events, and time-based conditions. For more information, see “Model Finite State Machines” on page 1-3.

For instance, this Stateflow chart presents the logic underlying a half-wave rectifier. The chart contains two states labeled `On` and `Off`. In the `On` state, the chart output signal `y` is equal to the input `x`. In the `Off` state, the output signal is set to zero. When the input signal crosses some threshold `t0`, the chart transitions between these states. The actions in each state update the value of `y` at each time step of the simulation.



This example shows how to create this Stateflow chart for simulation in Simulink and execution in MATLAB.

Construct the Stateflow Chart

Open the Stateflow Editor

The Stateflow Editor is a graphical environment for designing state transition diagrams, flow charts, state transition tables, and truth tables. The main components of the Stateflow Editor are the object palette, the chart canvas, and the Symbols window.

- The chart canvas is a drawing area where you create a chart by combining states, transitions, and other graphical elements.
- On the left side of the canvas, the object palette displays a set of tools for adding graphical elements to your chart.
- On the right side of the canvas, in the Symbols window, you add new data to the chart and resolve any undefined or unused symbols.

To create a Stateflow chart that you can simulate as a block in a Simulink model, at the MATLAB command prompt, enter:

```
sfnew rectify     % create chart for simulation in a Simulink model
```

After a few moments, Simulink opens a model called `rectify` that contains an empty Stateflow Chart block. To open the Stateflow Editor, double-click the chart block.

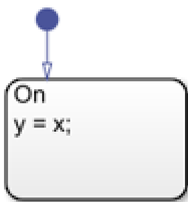
To create a standalone Stateflow chart that you can execute as a MATLAB object, at the MATLAB command prompt, enter:

```
edit rectify.sfx  % create chart for execution as a MATLAB object
```

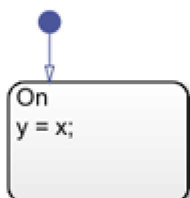
If the file `rectify.sfx` does not exist, the Stateflow Editor opens an empty chart with the name `rectify`.

Add States and Transitions

1. From the object palette, click the **State** icon and move the pointer to the chart canvas. A state with its default transition appears. To place the state, click a location on the canvas. At the text prompt, enter the state name `On` and the state action `y = x`.

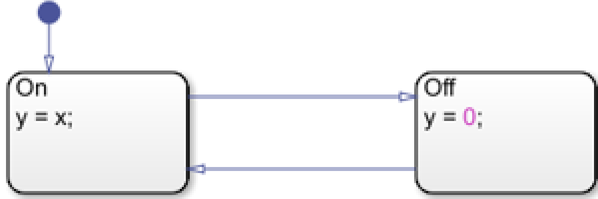


2. Add another state. Right-click and drag the `On` state. Blue graphical cues help you to align your states horizontally or vertically. The name of the new state changes to `Off`. Double-click the state and modify the state action to `y = 0`.

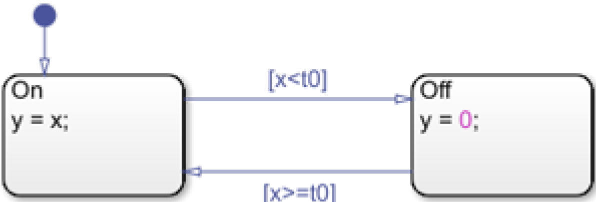


3 Realign the two states and pause on the space between the two states. Blue transition cues indicate several ways in which you can connect the states. To add transitions, click the appropriate cue.

Alternatively, to draw a transition, click and drag from the edge of one state to the edge of the other state.



4 Double-click each transition and type the appropriate transition condition `x<t0` or `x>=t0`. The conditions appear inside square brackets.



5 Clean up the chart:

- To improve clarity, move each transition label to a convenient location above or below its corresponding transition.
- To align and resize the graphical elements of your chart, select **Chart > Arrange > Arrange Automatically** or press **Ctrl+Shift+A**.
- To resize the chart to fit the canvas, press the space bar or click the **Fit To View** icon .

Resolve Undefined Symbols

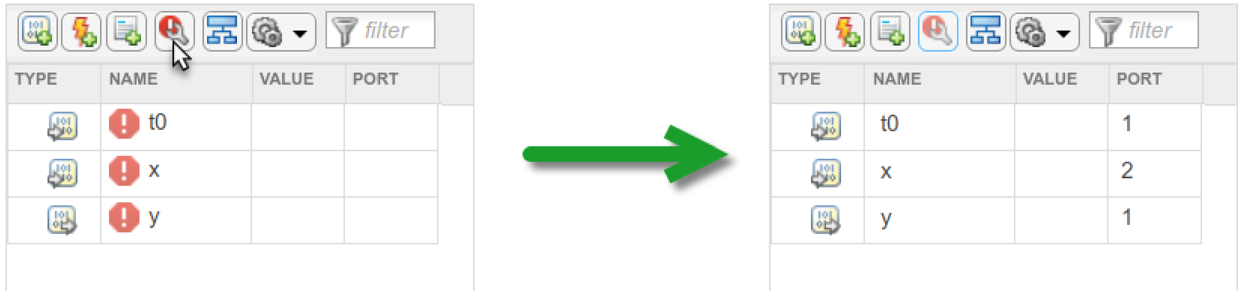
Before you can execute your chart, you must define each symbol that you use in the chart and specify its scope (for example, input data, output data, or local data). In the Symbols

window, undefined symbols are marked with a red error icon. The **Type** column displays the suggested scope for each undefined symbol based on its usage in the chart.

1

   In the Symbols window, click the **Resolve Undefined Symbols** icon.

   - If you are building a chart in a Simulink model, the Stateflow Editor resolves the symbols `x` and `t0` as input data and `y` as output data.
   - If you are building a standalone chart for execution in MATLAB, the Stateflow Editor resolves `t0`, `x`, and `y` as local data.



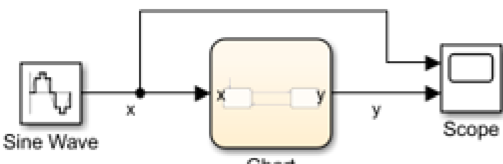
2. Because the threshold `t0` does not change during simulation, change its scope to constant data. In the **Type** column, click the data type icon next to `t0` and select `Constant Data`.
3. Set the value for the threshold `t0`. In the **Value** column, click the blank entry next to `t0` and enter a value of 0.
4. To save your Stateflow chart, click the **Save** icon.

Your chart is now ready for simulation in Simulink or execution in MATLAB.

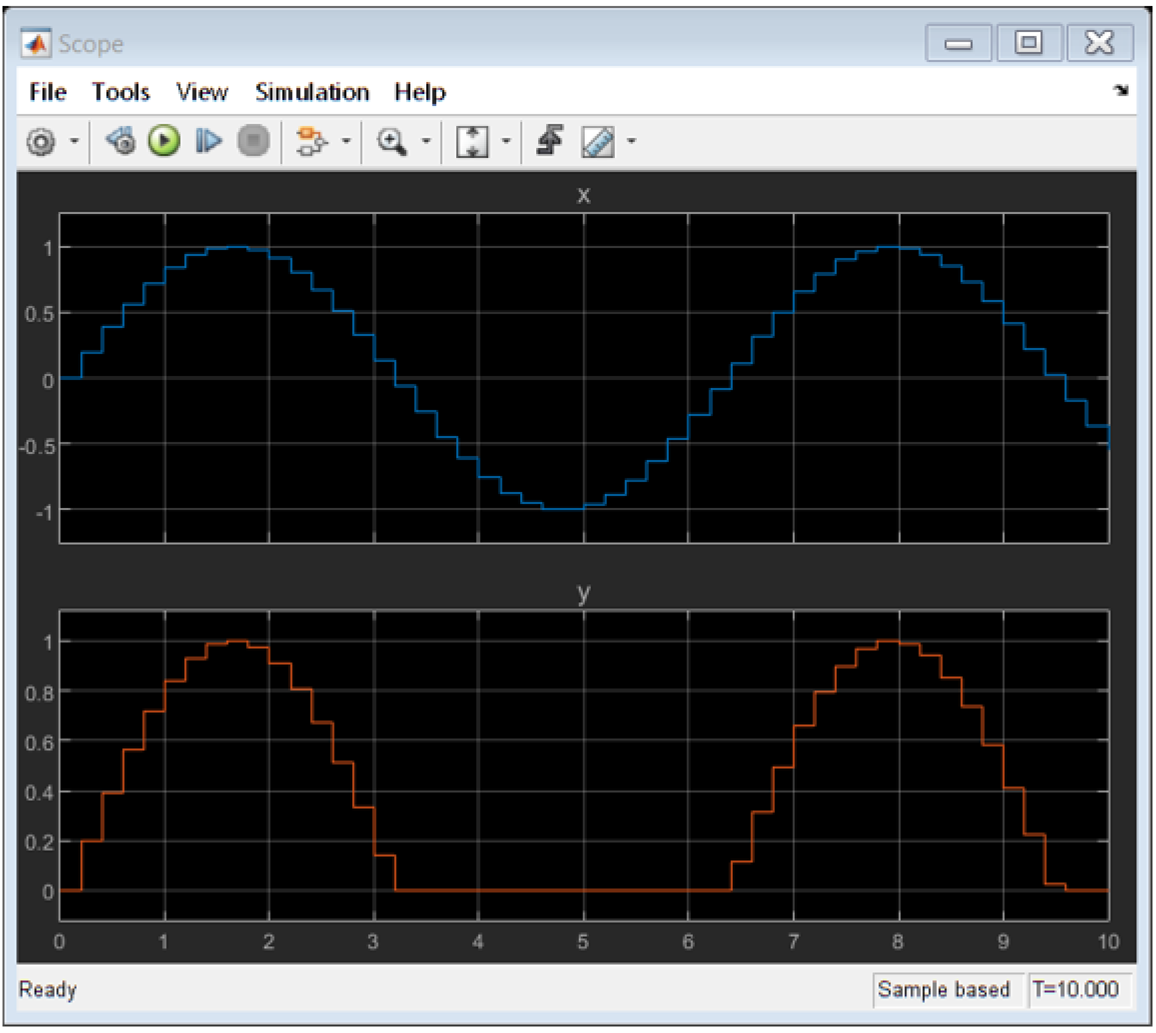
Simulate the Chart as a Simulink Block

To simulate the chart inside a Simulink model, connect the chart block to other blocks in the model through input and output ports. If you want to execute the chart from the MATLAB Command Window, see “Execute the Chart as a MATLAB Object” on page 1-19.

1. To return to the Simulink Editor, on the explorer bar at the top of the canvas, click the name of the Simulink model: `rectify`. If the explorer bar is not visible, click the **Hide/Show Explorer Bar** icon at the top of the object palette.
2. Add a source to the model:
   - From the Simulink Sources library, add a Sine Wave block.
   - Double-click the Sine Wave block and set the **Sample time** to 0.2.
   - Connect the output of the Sine Wave block to the input of the Stateflow chart.
   - Label the signal as `x`.
3. Add a sink to the model:
   - From the Simulink Sinks library, add a Scope block with two input ports.
   - Connect the output of the Sine Wave block to the first input of the Scope block.
   - Connect the output of the Stateflow chart to the second input of the Scope block.
   - Label the signal as `y`.
4. Save the Simulink model.



5. To simulate the model, click the **Run** icon. During the simulation, the Stateflow editor highlights active states and transitions through chart animation.
6. After you simulate the model, double-click the Scope block. The scope displays the graphs of the input and output signals to the charts.



The simulation results show that the rectifier filters out negative input values.

Execute the Chart as a MATLAB Object

To execute the chart in the MATLAB Command Window, create a chart object and call its `step` function. If you want to simulate the chart inside a Simulink model, see “Simulate the Chart as a Simulink Block” on page 1-17.

1. Create a chart object `r` by using the name of the `sfx` file that contains the chart definition as a function. Specify the initial value for the chart data `x` as a name-value pair.

   ```
   r = rectify('x',0);
   ```

2. Initialize input and output data for chart execution. The vector `X` contains input values from a sine wave. The vector `Y` is an empty accumulator.

   ```
   T = [0:0.2:10];
   X = sin(T);
   Y = [];
   ```

3. Execute the chart object by calling the `step` function multiple times. Pass individual values from the vector `X` as chart data `x`. Collect the resulting values of `y` in the vector `Y`. During the execution, the Stateflow editor highlights active states and transitions through chart animation.

   ```
   for i = 1:51
       step(r,'x',X(i));
       Y(i) = r.y;
   end
   ```

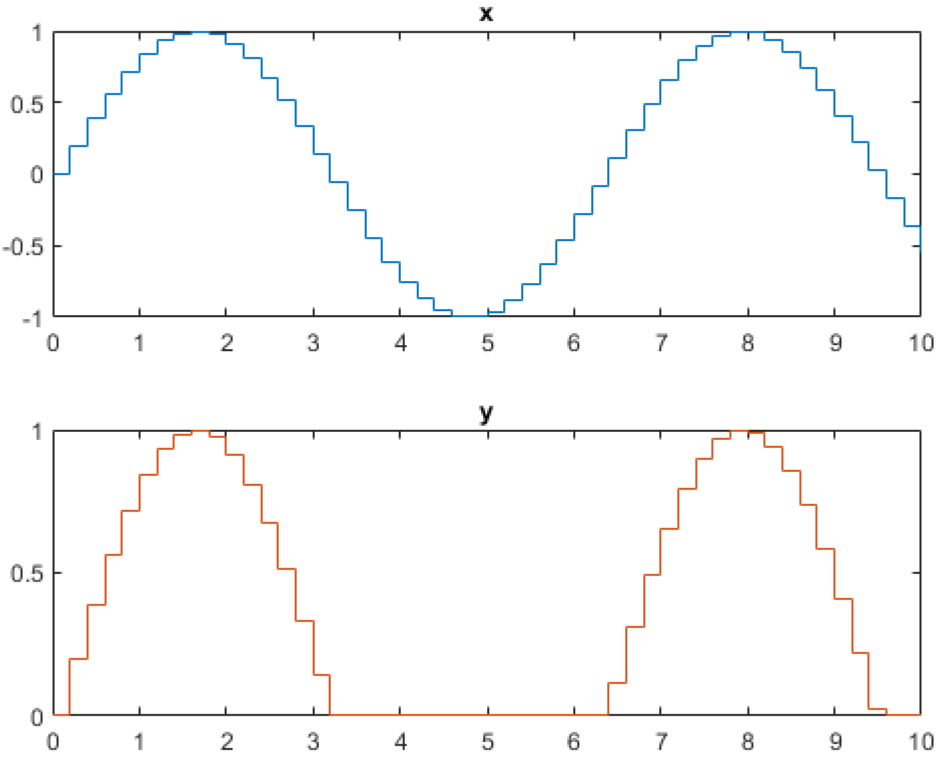
4. Delete the chart object `r` from the MATLAB workspace.

   ```
   delete(r)
   ```

5. Examine the results of the chart execution. For example, you can call the `stairs` function to create a stairstep graph that compares the values of `X` and `Y`.

   ```
   ax1 = subplot(2,1,1);
   stairs(ax1,T,X,'color','#0072BD')
   title(ax1,'x')

   ax2 = subplot(2,1,2);
   stairs(ax2,T,Y,'color','#D95319')
   title(ax2,'y')
   ```



The execution results show that the rectifier filters out negative input values.

See Also

Chart | Scope | Sine Wave | `sfnew` | `stairs`

More About

- "Model Finite State Machines" on page 1-3
- "Define Chart Behavior by Using Actions" on page 1-23
- "Stateflow Editor Operations"

- “Resolve Undefined Symbols in Your Chart”
- “Create Stateflow Charts for Execution as MATLAB Objects”

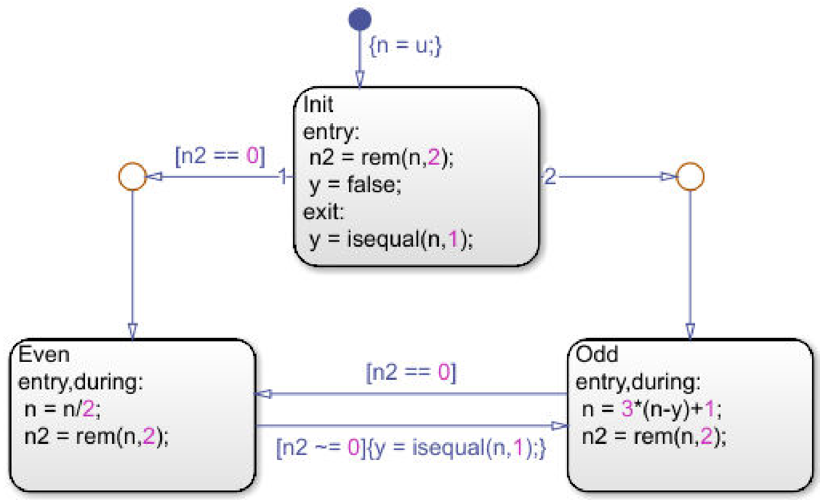
Define Chart Behavior by Using Actions

State and transition actions are instructions that you write inside a state or next to a transition to define how a Stateflow chart behaves during simulation. For more information, see “Model Finite State Machines” on page 1-3.

For example, the actions in this chart define a state machine that empirically verifies one instance of the Collatz conjecture. For a given numerical input `u`, the chart computes the hailstone sequence n_0 = `u`, n_1, n_2, n_3, … by iterating this rule:

- If n_i is even, then $n_{i+1} = n_i / 2$.
- If n_i is odd, then $n_{i+1} = 3n_i+1$.

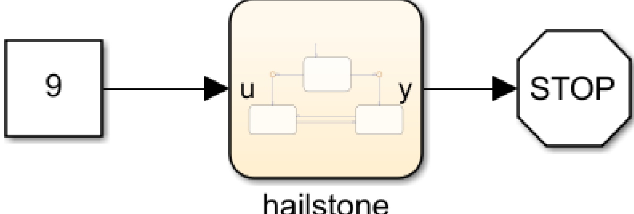
The Collatz conjecture states that every positive integer has a hailstone sequence that eventually reaches a value of one.



The chart consists of three states. At the start of simulation, the `Init` state initializes the chart data:

- The local data `n` is set to the value of the input `u`.
- The local data `n2` is set to the remainder when `n` is divided by two.
- The output data `y` is set to `false`.

Depending on the parity of the input, the chart transitions to either the `Even` or `Odd` state. As the state activity shifts between the `Even` and `Odd` states, the chart computes the numbers in the hailstone sequence. When the sequence reaches a value of one, the output data `y` becomes `true` and triggers a Stop Simulation block in the Simulink model.



State Action Types

State actions define what a Stateflow chart does while a state is active. The most common types of state actions are `entry`, `during`, and `exit` actions.

Type of State Action	Abbreviation	Description
`entry`	`en`	Action occurs on a time step when the state becomes active.
`during`	`du`	Action occurs on a time step when the state is already active and the chart does not transition out of the state.
`exit`	`ex`	Action occurs on a time step when the chart transitions out of the state.

You can specify the type of a state action by its complete keyword (`entry`, `during`, `exit`) or by its abbreviation (`en`, `du`, `ex`). You can also combine state action types by using commas. For instance, an action with the combined type `entry,during` occurs on the time step when the state becomes active and on every subsequent time step while the state remains active.

This table lists the result of each state action in the hailstone chart.

<table>
<tr><th>State</th><th>Action</th><th>Result</th></tr>
<tr><td rowspan="2"><code>Init</code></td><td><code>entry:
 n2 = rem(n,2);
 y = false;</code></td><td>When <code>Init</code> becomes active at the start of the simulation, determines the parity of <code>n</code> and sets <code>y</code> to <code>false</code>.</td></tr>
<tr><td><code>exit:
 y = isequal(n,1);</code></td><td>When transitioning out of <code>Init</code> after one time step, determines whether <code>n</code> is equal to one.</td></tr>
<tr><td><code>Even</code></td><td><code>entry,during:
 n = n/2;
 n2 = rem(n,2);</code></td><td>Computes the next number of the hailstone sequence (<code>n</code> / 2) and updates its parity on:
• The time step when <code>Even</code> first becomes active.
• Every subsequent time step that <code>Even</code> is active.</td></tr>
<tr><td><code>Odd</code></td><td><code>entry,during:
 n = 3*(n-y)+1;
 n2 = rem(n,2);</code></td><td>Computes the next number of the hailstone sequence (3<code>n</code> +1) and updates its parity on:
• The time step when <code>Odd</code> first becomes active.
• Every subsequent time step that <code>Odd</code> is active.

Throughout most of the simulation, <code>y</code> evaluates to zero. On the last time step, when <code>n</code> = 1, <code>y</code> evaluates to one so this action does not modify <code>n</code> or <code>n2</code> before the simulation stops.</td></tr>
</table>

Transition Action Types

Transition actions define what a Stateflow chart does when a transition leads away from an active state. The most common types of transition actions are conditions and conditional actions. To specify transition actions, use a label with this syntax:

```
[condition]{conditional_action}
```

`condition` is a Boolean expression that determines whether the transition occurs. If you do not specify a condition, an implied condition evaluating to true is assumed.

`conditional_action` is an instruction that executes when the condition guarding the transition is true. The conditional action takes place after the condition but before any `exit` or `entry` state actions.

This table lists the result of each transition action in the hailstone chart.

Transition	Action	Action Type	Result
Default transition into `Init`	`n = u`	Conditional action	At the start of the simulation, assigns the input value `u` to the local data `n`.
Transition from `Init` to `Even`	`n2 == 0`	Condition	When `n` is even, transition occurs. The number 1 at the source of this transition indicates that it is evaluated before the transition to `Odd`.
Transition from `Init` to `Odd`		None	When `n` is odd, transition occurs. The number 2 at the source of this transition indicates that it is evaluated after the transition to `Even`.
Transition from `Odd` to `Even`	`n2 == 0`	Condition	When `n` is even, transition occurs.
Transition from `Even` to `Odd`	`n2 ~= 0`	Condition	When `n` is odd, transition occurs.
	`y = isequal(n,1)`	Conditional action	When transition occurs, determines whether `n` is equal to one.

Examine Chart Behavior

Suppose that you want to compute the hailstone sequence starting with a value of nine.

1. In the Model Configuration Parameters dialog box, under **Solver**, select these options:

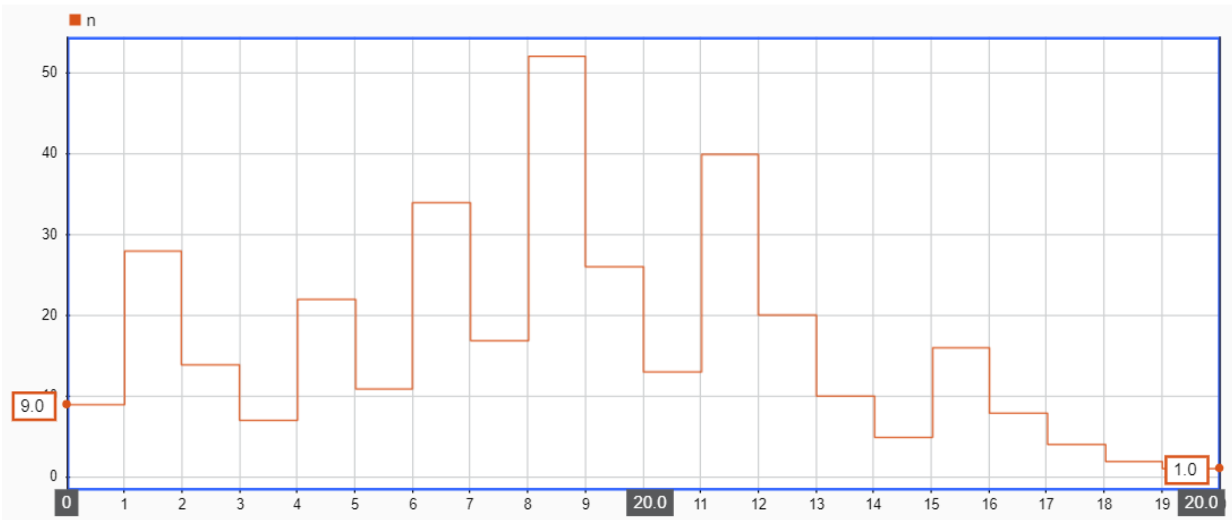
   - **Start time**: `0.0`
   - **Stop time**: `inf`
   - **Type**: `Fixed-step`
   - **Fixed-step size**: `1`

2. In the Property Inspector, select the symbol `n` for logging.
3. In the Constant block, enter an input of `u` = 9.
4. Click the **Run** icon .

The chart responds with these actions:

- At time *t* = 0, the default transition to `Init` occurs.
  - The transition action sets the value of `n` to 9.
  - The `Init` state becomes active.
  - The `entry` actions in `Init` set `n2` to 1 and `y` to `false`.
- At time *t* = 1, the condition `n2 == 0` is false so the chart prepares to transition to `Odd`.
  - The `exit` action in `Init` sets `y` to `false`.
  - The `Init` state becomes inactive.
  - The `Odd` state becomes active.
  - The `entry` actions in `Odd` set `n` to 28 and `n2` to 0.
- At time *t* = 2, the condition `n2 == 0` is true so the chart prepares to transition to `Even`.
  - The `Odd` state becomes inactive.
  - The `Even` state becomes active.
  - The `entry` actions in `Even` set `n` to 14 and `n2` to 0.
- At time *t* = 3, the condition `n2 ~= 0` is false so the chart does not take a transition.
  - The `Even` state remains active.
  - The `during` actions in `Even` set `n` to 7 and `n2` to 1.
- At time *t* = 4, the condition `n2 ~= 0` is true so the chart prepares to transition to `Odd`.
  - The transition action sets `y` to `false`.
  - The `Even` state becomes inactive.
  - The `Odd` state becomes active.
  - The `entry` actions in `Odd` set `n` to 22 and `n2` to 0.
- The chart continues to compute the hailstone sequence until it arrives at a value of `n` = 1 at time *t* = 19.
- At time *t* = 20, the chart prepares to transition from `Even` to `Odd`.
  - Before the `Even` state becomes inactive, the transition action sets `y` to `true`.
  - The `Odd` state becomes active.
  - The `entry` actions in `Odd` do not modify `n` or `n2`.

- The Stop Simulation block connected to the output signal y stops the simulation.



See Also

More About

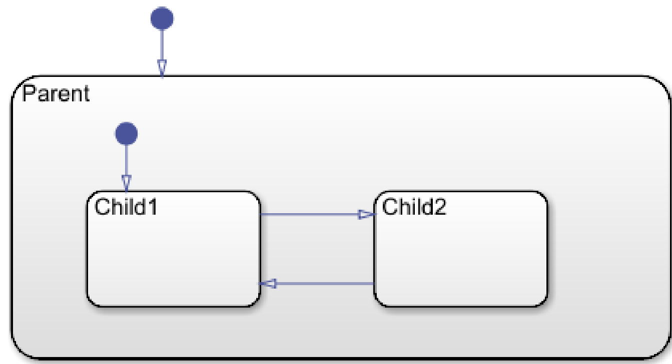
- “Model Finite State Machines” on page 1-3
- “State Action Types”
- “Transition Action Types”
- “Synchronize Parallel States by Broadcasting Events” on page 1-41
- “Schedule Chart Actions by Using Temporal Logic” on page 1-60

Create a Hierarchy to Manage System Complexity

Add structure to your model one subcomponent at a time by creating a *hierarchy* of nested states. You can then control multiple levels of complexity in your Stateflow chart. For more information, see “Model Finite State Machines” on page 1-3.

State Hierarchy

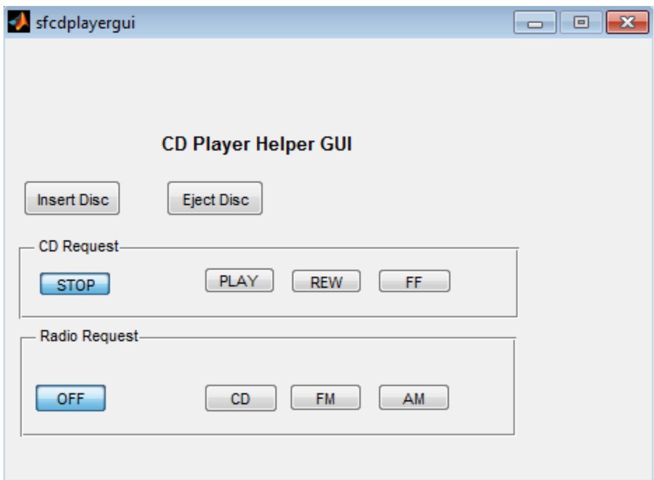
To create a hierarchy of states, place one or more states within the boundaries of another state. The inner states are child states (or substates) of the outer state. The outer state is the parent (or superstate) of the inner states.



The contents of a parent state behave like a smaller chart. When a parent state becomes active, one of its child states also becomes active. When the parent state becomes inactive, all of its child states become inactive.

Example of Hierarchy

The Stateflow example `sf_cdplayer` models a stereo system consisting of an AM radio, an FM radio, and a CD player. During simulation, you control the stereo system by clicking buttons on the CD Player Helper.



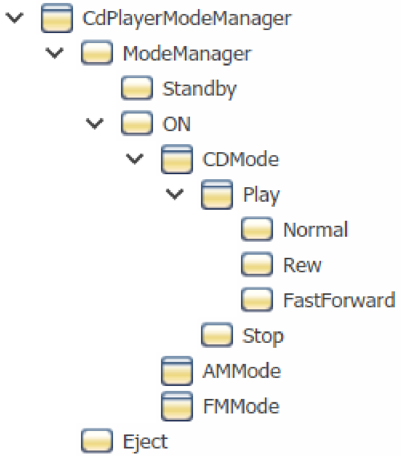
The stereo is initially in standby mode (OFF). When you select one of the Radio Request buttons, the stereo turns on the corresponding subcomponent. If you select the CD player, you can click one of the CD Request buttons to choose Play, Rewind, Fast-Forward, or Stop. You can insert or eject a disc at any point during the simulation.

Initially, the complete implementation of this stereo system appears rather complicated. However, by focusing on a single level of activity at a time, you can design the overall system design incrementally. For example, these conditions are necessary for the CD player to enter Fast-Forward play mode:

1. You turned on the stereo.
2. You turned on the CD player.
3. You are playing a disc.
4. You clicked the FF button in the UI.

You can construct a hierarchical model that considers each of these conditions one at a time. For instance, the outermost level can define the transitions between the stereo turning on and off. The middle levels define the transition between the different stereo subcomponents, and between the stop and play modes of the CD player. The bottommost level defines the response to the CD Request buttons if you have met all the other conditions for playing a disc.

To implement the behavior of the stereo system, `sf_cdplayer` uses the hierarchy of nested states listed by the Model Explorer under the `CdPlayerModeManager` chart. To open the Model Explorer, select **View** > **Model Explorer** > **Model Explorer**.

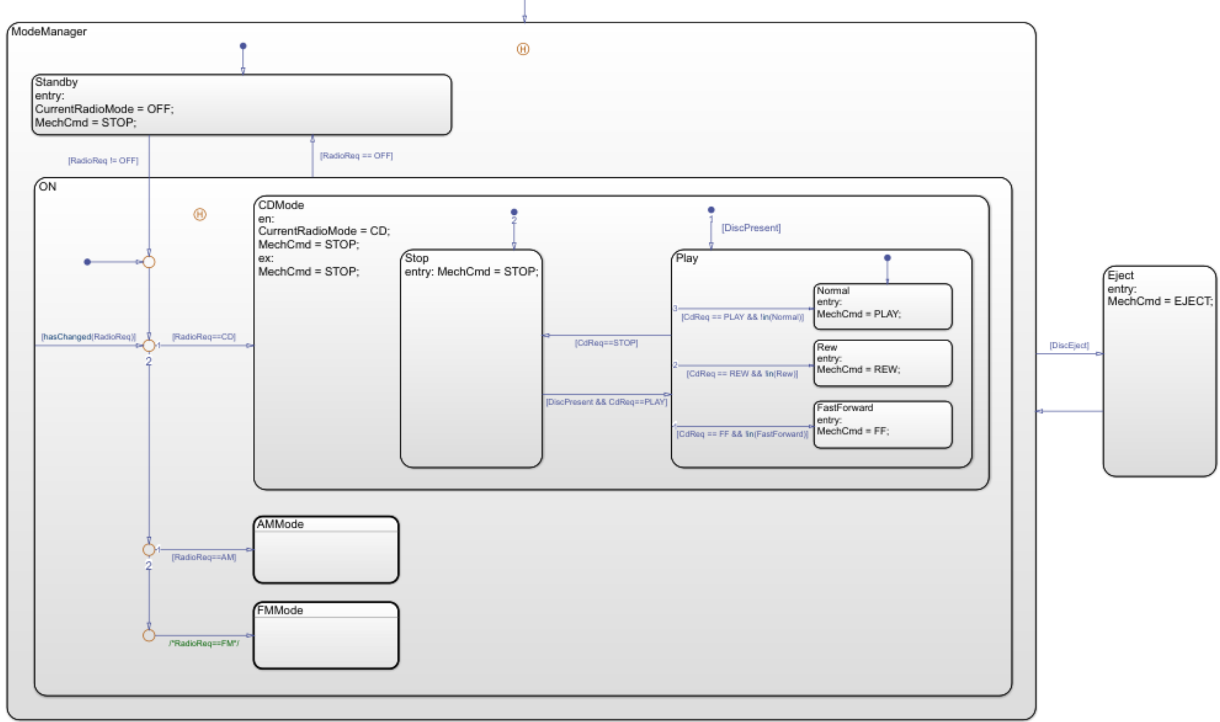


This table lists the role of each state in the hierarchy.

Hierarchy Level	State	Description
Top level (Stateflow chart `CdPlayerModeManager`)	`ModeManager`	Normal operating mode for stereo system
	`Eject`	Disc ejection mode (interrupts all other stereo functions)
Stereo system activity (child states of `ModeManager`)	`Standby`	Stereo system is in standby mode (OFF)
	`ON`	Stereo system is active (ON)
Stereo subcomponents (child states of `On`)	`AMMode`	AM radio subcomponent is active
	`FMMode`	FM radio subcomponent is active
	`CDMode`	CD player subcomponent is active
CD player activity (child states of `CDMode`)	`Stop`	CD player is stopped
	`Play`	CD player is playing disc
Disc play modes (child states of `Play`)	`Normal`	Normal play mode

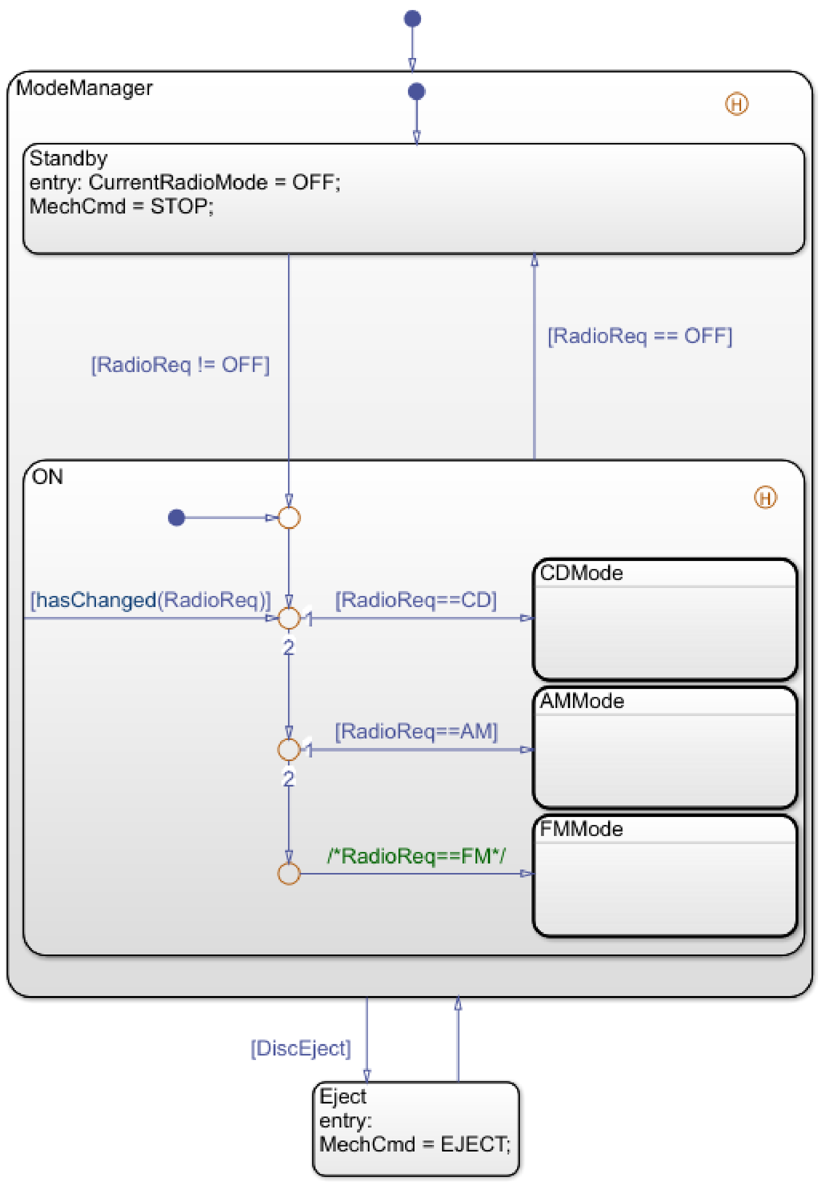
Hierarchy Level	State	Description
	Rew	Reverse play mode
	FastForward	Fast-Forward play mode

This figure shows the complete layout of the states in the chart.



Simplify Chart Appearance by Using Subcharts

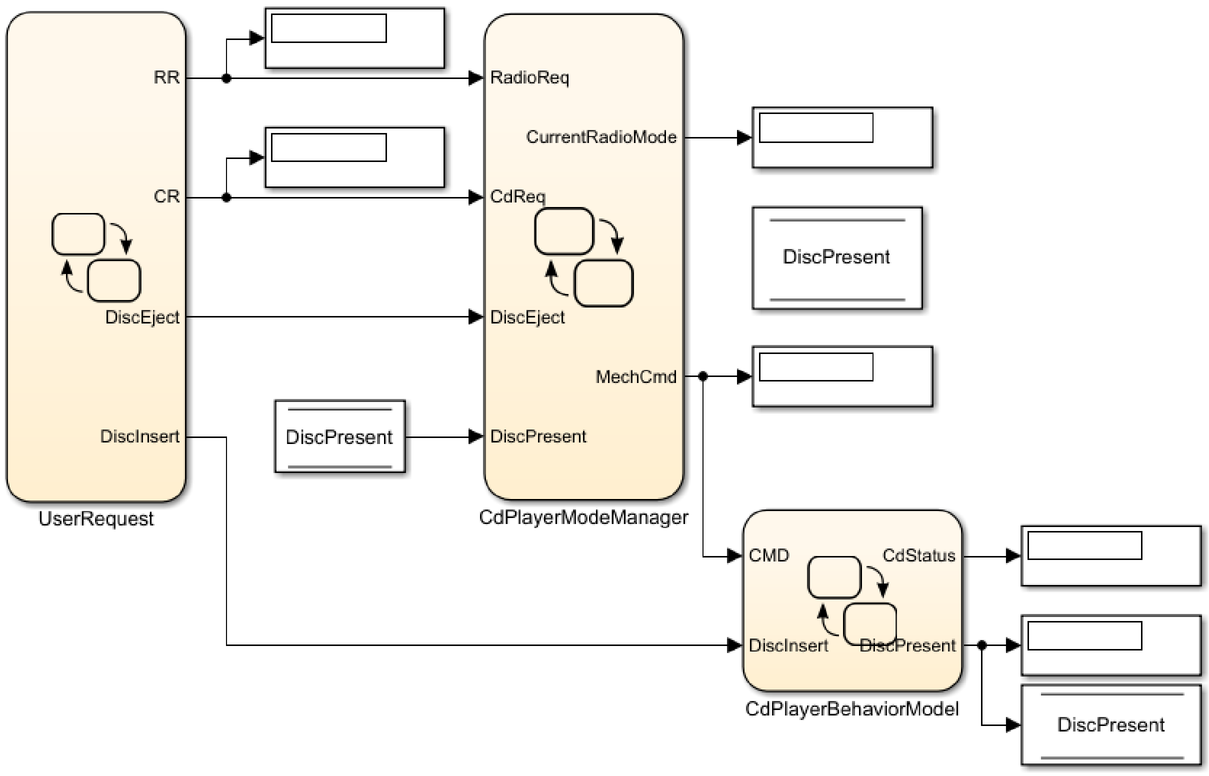
You can simplify the overall appearance of a chart with a complex hierarchy by hiding low-level details inside subcharts, which appear as opaque boxes. The use of subcharts does not change the behavior of the chart. For instance, in `sf_cdplayer`, the stereo subcomponents `AMMode`, `FMMode`, and `CDMode` are implemented as subcharts. When you open the chart `CdPlayerModeManager`, you see only three levels of the state hierarchy. To see the details inside one of the subcharts, double-click the subchart.


ModeManager
Standby
entry: CurrentRadioMode = OFF;
MechCmd = STOP;
[RadioReq != OFF]
[RadioReq == OFF]
ON
[hasChanged(RadioReq)]
[RadioReq==CD]
CDMode
1
2
[RadioReq==AM]
AMMode
1
2
/*RadioReq==FM*/
FMMode
[DiscEject]
Eject
entry:
MechCmd = EJECT;

Explore the Example

The example `sf_cdplayer` contains two other Stateflow charts:

- `UserRequest` manages the interface with the UI and passes inputs to the `CdPlayerModeManager` chart.
- `CdPlayerBehaviorModel` receives the output from the `CdPlayerModeManager` and mimics the mechanical behavior of the CD player.



During simulation, you can investigate how each chart responds to interactions with the CD Player Helper. To switch quickly between charts, use the tabs at the top of the Stateflow Editor.

See Also

Related Examples

- “Modeling a CD Player/Radio by Using Enumerated Data Types”
- “Modeling a CD Player/Radio Using State Transition Tables”
- “Simulate a Media Player by Using Strings”

More About

- “Model Finite State Machines” on page 1-3
- “State Hierarchy”
- “Encapsulate Modal Logic by Using Subcharts”
- “Model Media Player by Using Enumerated Data”

Model Synchronous Subsystems by Using Parallelism

To implement operating modes that run concurrently, use *parallelism* in your Stateflow chart. For example, as part of a complex system design, you can employ parallel states to model independent components or subsystems that are active simultaneously. For more information, see “Model Finite State Machines” on page 1-3.

State Decomposition

Stateflow charts can combine exclusive (OR) states and parallel (AND) states:

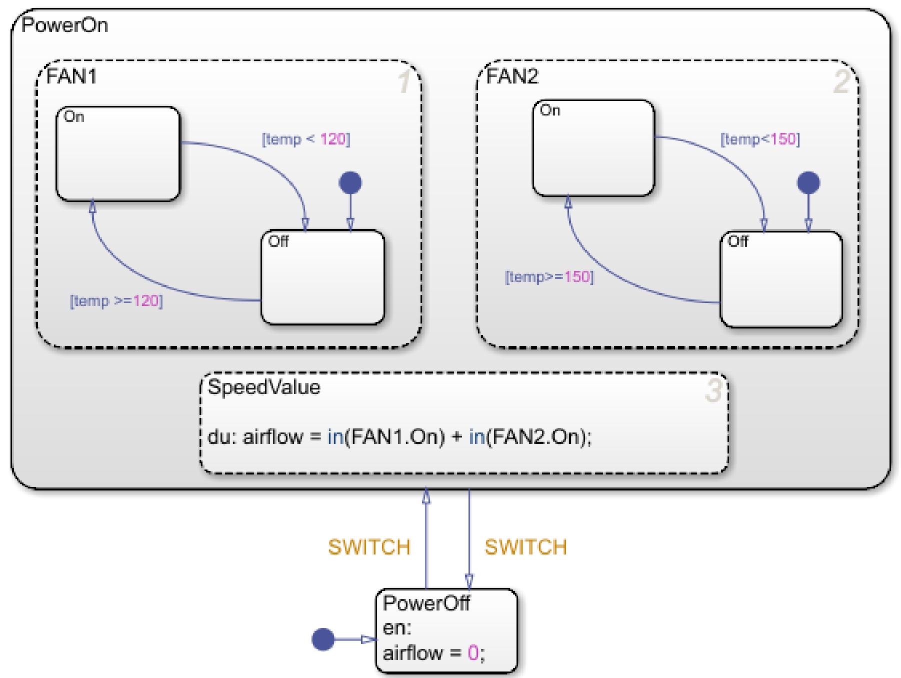
- **Exclusive (OR) states** represent mutually exclusive modes of operation. No two exclusive states at the same hierarchical level can be active or execute at the same time. Stateflow represents each exclusive state by a solid rectangle.

- **Parallel (AND) states** represent independent modes of operation. Two or more parallel states can be active at the same time, although they execute in a serial fashion. Stateflow represents each parallel state by a dashed rectangle with a number indicating its execution order.

All states at a given hierarchical level must be of the same type. The parent state, or in the case of top-level states, the chart itself, has OR (exclusive) or AND (parallel) decomposition. The default state decomposition type is OR (exclusive). To change the decomposition type, right-click the parent state and select **Decomposition > AND (Parallel)**.

Example of Parallel Decomposition

The Stateflow example `sf_aircontrol` employs parallelism to implement an air controller that maintains air temperature at 120 degrees in a physical plant.



The controller operates two fans. The first fan turns on when the air temperature rises above 120 degrees. The second fan provides additional cooling when the air temperature rises above 150 degrees. The chart models these fans as parallel states `FAN1` and `FAN2`, both of which are active when the controller is turned on. Except for their operating thresholds, the fans have an identical configuration of states and transitions that reflects the two modes of fan operation (`On` and `Off`).

A third parallel state `SpeedValue` calculates the value of the output data `airflow` based on how many fans have cycled on at each time step. The Boolean expression `in(FAN1.On)` has a value of 1 when the `On` state of `FAN1` is active. Otherwise, `in(FAN1.On)` equals 0. The value of `in(FAN2.On)` represents whether `FAN2` has cycled on or off. The sum of these expressions indicates the number of fans that are turned on during each time step.

Note To give objects unique identifiers when they have the same name in different parts of the chart hierarchy, use dot notation such as `Fan1.On` and `Fan2.On`. For more information, see "Identify Data by Using Dot Notation".

This table lists the rationale for using exclusive (OR) and parallel (AND) states in the air controller chart.

State	Decomposition	Rationale
`PowerOff`, `PowerOn`	Exclusive (OR) states	The controller cannot be on and off at the same time.
`FAN1`, `FAN2`	Parallel (AND) states	The fans operate as independent components that turn on or off depending on how much cooling is required.
`FAN1.On`, `FAN1.Off`	Exclusive (OR) states	Fan 1 cannot be on and off at the same time.
`FAN2.On`, `FAN2.Off`	Exclusive (OR) states	Fan 2 cannot be on and off at the same time.
`SpeedValue`	Parallel (AND) state	`SpeedValue` represents an independent subsystem that monitors the status of the fans at each time step.

Order of Execution for Parallel States

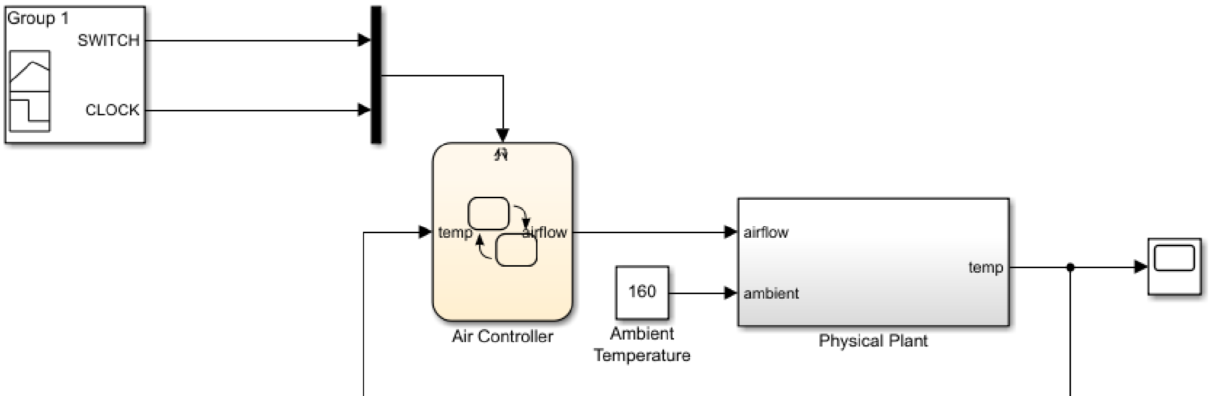
Although `FAN1`, `FAN2`, and `SpeedValue` are active concurrently, these states execute in serial fashion during simulation. The numbers in the upper-right corners of the states specify the order of execution. The rationale for this order of execution is:

- `FAN1` executes first because it cycles on at a lower temperature than `FAN2`. It can turn on regardless of whether `FAN2` is on or off.
- `FAN2` executes second because it cycles on at a higher temperature than `FAN1`. It can turn on only if `FAN1` is already on.
- `SpeedValue` executes last so it can observe the most up-to-date status of `FAN1` and `FAN2`.

By default, Stateflow assigns the execution order of parallel states based on their order of creation in the chart. To change the execution order of a parallel state, right-click the state and select a value from the **Execution Order** drop-down list.

Explore the Example

The Stateflow example contains a Stateflow chart and a Simulink subsystem.



Based on the air temperature `temp`, the Air Controller chart turns on the fans and passes the value of `airflow` to the Physical Plant subsystem. This output value determines the amount of cooling activity, as indicated by this table.

Value of `airflow`	Description	Cooling Activity Factor k_{Cool}
0	No fans are running. The value of `temp` does not decrease.	0
1	One fan is running. The value of `temp` decreases according to the cooling activity factor.	0.05
2	Two fans are running. The value of `temp` decreases according to the cooling activity factor.	0.1

The Physical Plant block updates the air temperature inside the plant based on the equations

$$\texttt{temp}(0) = T_{\text{Initial}}$$

$$\texttt{temp}'(t) = (T_{\text{Ambient}} - \texttt{temp}(t))\cdot(k_{\text{Heat}} - k_{\text{Cool}}),$$

where:

- $T_{Initial}$ is the initial temperature (default = 70º)
- $T_{Ambient}$ is the ambient temperature (default = 160º)
- k_{Heat} is the heat transfer factor for the plant (default = 0.01)
- k_{Cool} is the cooling activity factor corresponding to `airflow`

The new value of `temp` determines the amount of cooling at the next time step of the simulation.

See Also

More About

- “Model Finite State Machines” on page 1-3
- “State Decomposition”
- “Execution Order for Parallel States”
- “Check State Activity by Using the in Operator”
- “Synchronize Parallel States by Broadcasting Events” on page 1-41

Synchronize Parallel States by Broadcasting Events

Events help parallel states to coordinate with one another, allowing one state to trigger an action in another state. To synchronize parallel states in the same Stateflow chart, broadcast events directly from one state to another. For more information on parallel states, see “Model Synchronous Subsystems by Using Parallelism” on page 1-36.

Broadcasting Local Events

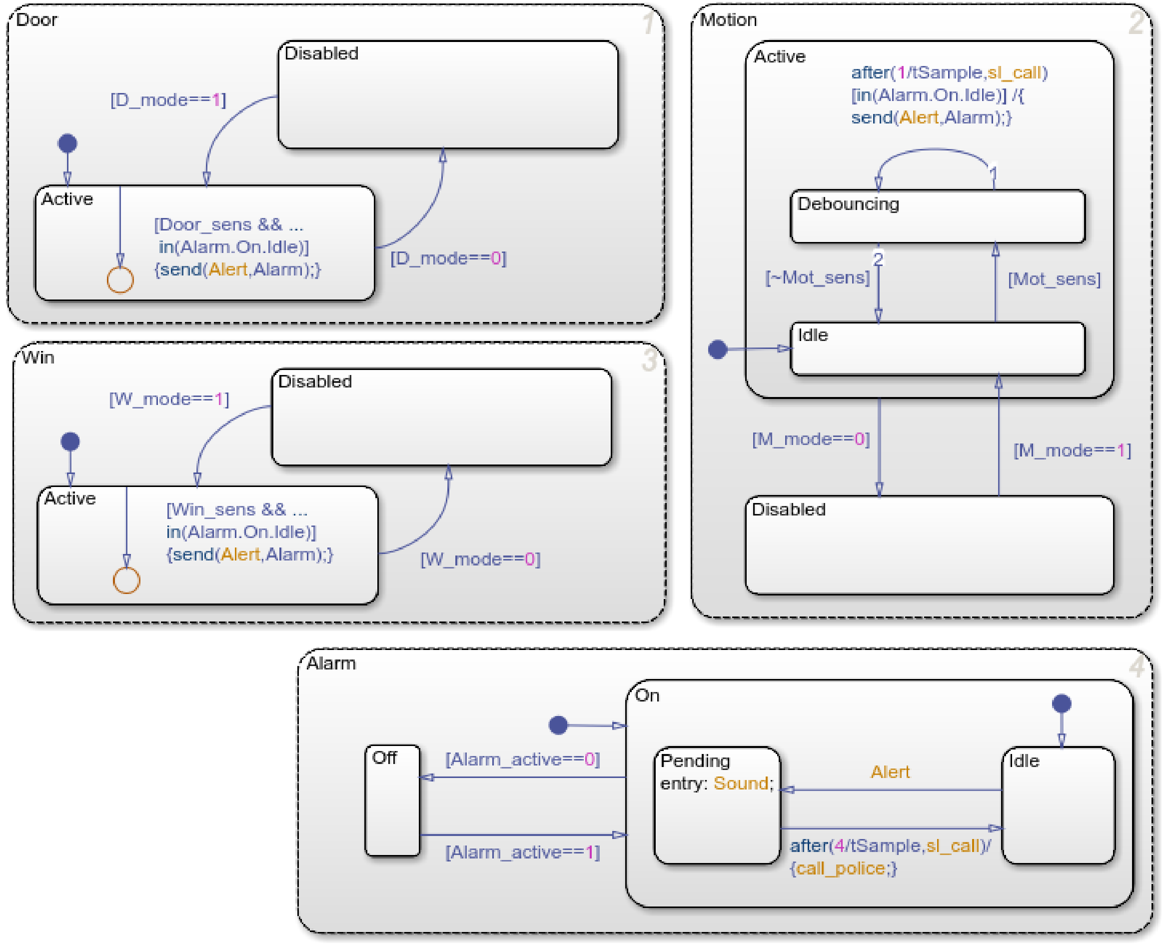
A *local event* is a nongraphical object that can trigger transitions or actions in a parallel state of a Stateflow chart. When you broadcast an event to a state, the event takes effect in the receiving state and in any substates in the hierarchy of that state. To broadcast an event, use the `send` operator:

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

`event_name` is the name of the event to be broadcast. *`state_name`* is an active state during the broadcast.

Example of Event Broadcasting

The Stateflow example `sf_security` uses local events as part of the design of a home security system.



The security system consists of an alarm and three anti-intrusion sensors (a window sensor, a door sensor, and a motion detector). After the system detects an intrusion, you have a small amount of time to disable the alarm. Otherwise, the system calls the police.

The Security System chart models each subsystem with a separate parallel state. An enabling input signal selects between the `On` and `Off` modes for the alarm, or between the `Active` and `Disabled` modes for each sensor. When enabled, each sensor monitors a triggering input signal that indicates a possible intrusion.

Subsystem	State	Enabling Signal	Triggering Signal
Door sensor	`Door`	`D_mode`	`Door_sens`
Window sensor	`Win`	`W_mode`	`Win_sens`
Motion detector	`Motion`	`M_mode`	`Mot_sens`
Alarm	`Alarm`	`Alarm_active`	

If a sensor detects an intrusion while the alarm subsystem is on, then it broadcasts an `Alert` event with this command:

```
send(Alert,Alarm)
```

To mitigate the effect of sporadic false positives, the motion detector incorporates a debouncing design, so that only a sustained positive trigger signal produces an alarm. In contrast, the door and window sensors interpret a single positive trigger signal as an intrusion and issue an immediate alarm.

In the alarm subsystem, the `Alert` event causes a transition from the `Idle` substate to the `Pending` substate. When this state becomes active, a warning sound alerts occupants to the possible intrusion. If there is an accidental alarm, the occupants have a short time to disable the security system. If not disabled within that time period, the system calls the police for help.

Coordinate with Other Simulink Blocks

Stateflow charts can use events to communicate with other blocks in a Simulink model. For instance, in the `sf_security` example:

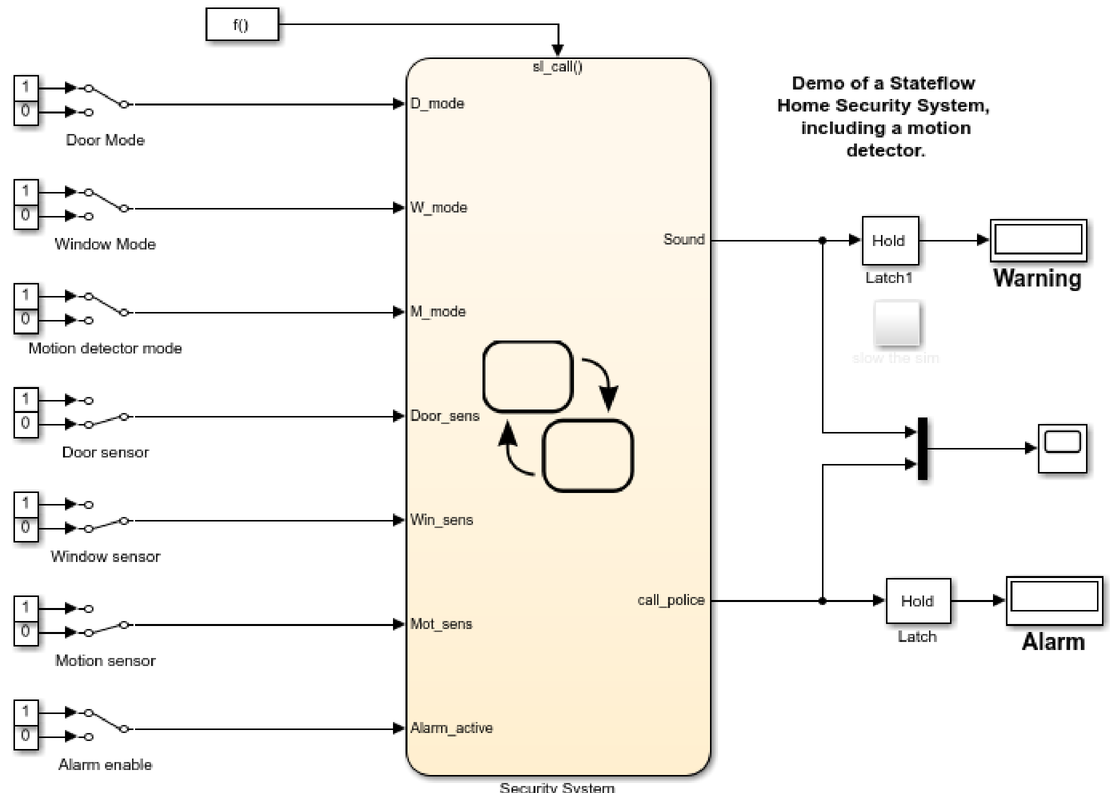
- The output events `Sound` and `call_police` drive external blocks that handle the warning sound and the call to the police. The commands for broadcasting these events occur in the `Alarm.On` state:

  - The command for `Sound` occurs as an entry action in the `Pending` substate.
  - The command for `call_police` occurs as an action in the transition between the `Pending` and `Idle` substates.

  In each case, the command to issue the output event is the name of the event.

- The input event `sl_call` controls the timing of the motion detector debouncer and the short delay before the call to the police. In each instance, the event occurs inside a

call to the temporal operator `after`, which results in a transition after the chart receives the event some number of times.



Output Events

An *output event* occurs in a Stateflow chart but is visible in Simulink blocks outside the chart. This type of event enables a chart to notify other blocks in a model about events that occur in the chart.

Each output event maps to an output port on the right side of the chart. Depending on its configuration, the corresponding signal can control a Triggered Subsystem or a Function-Call Subsystem. To configure an output event, in the Property Inspector, set the **Trigger** field to one of these options.

Type of Trigger	Description
`Either Edge`	Output event broadcast causes the outgoing signal to toggle between zero and one.
`Function call`	Output event broadcast causes a Simulink function-call event.

In the `sf_security` example, the output events `Sound` and `call_police` use edge triggers to activate a pair of latch subsystems in the Simulink model. When each latch detects a change of value in its input signal, it briefly outputs a value of one before returning to an output of zero.

Input Events

An *input event* occurs in a Simulink block but is visible in a Stateflow chart. This type of event enables other Simulink blocks, including other Stateflow charts, to notify a specific chart of events that occur outside it.

An external Simulink block sends an input event through a signal connected to the trigger port on the top of the Stateflow chart. Depending on its configuration, an input event results from a change in signal value or through a function call from a Simulink block. To configure an input event, in the Property Inspector, set the **Trigger** field to one of these options.

Type of Trigger	Description
`Rising`	Chart is activated when the input signal changes from either zero or a negative value to a positive value.
`Falling`	Chart is activated when the input signal changes from a positive value to either zero or a negative value.
`Either`	Chart is activated when the input signal crosses zero as it changes in either direction.
`Function call`	Chart is activated with a function call from a Simulink block.

In the `sf_security` example, a Simulink Function-Call Generator block controls the timing of the security system by triggering the input event `sl_call` through periodic function calls.

Explore the Example

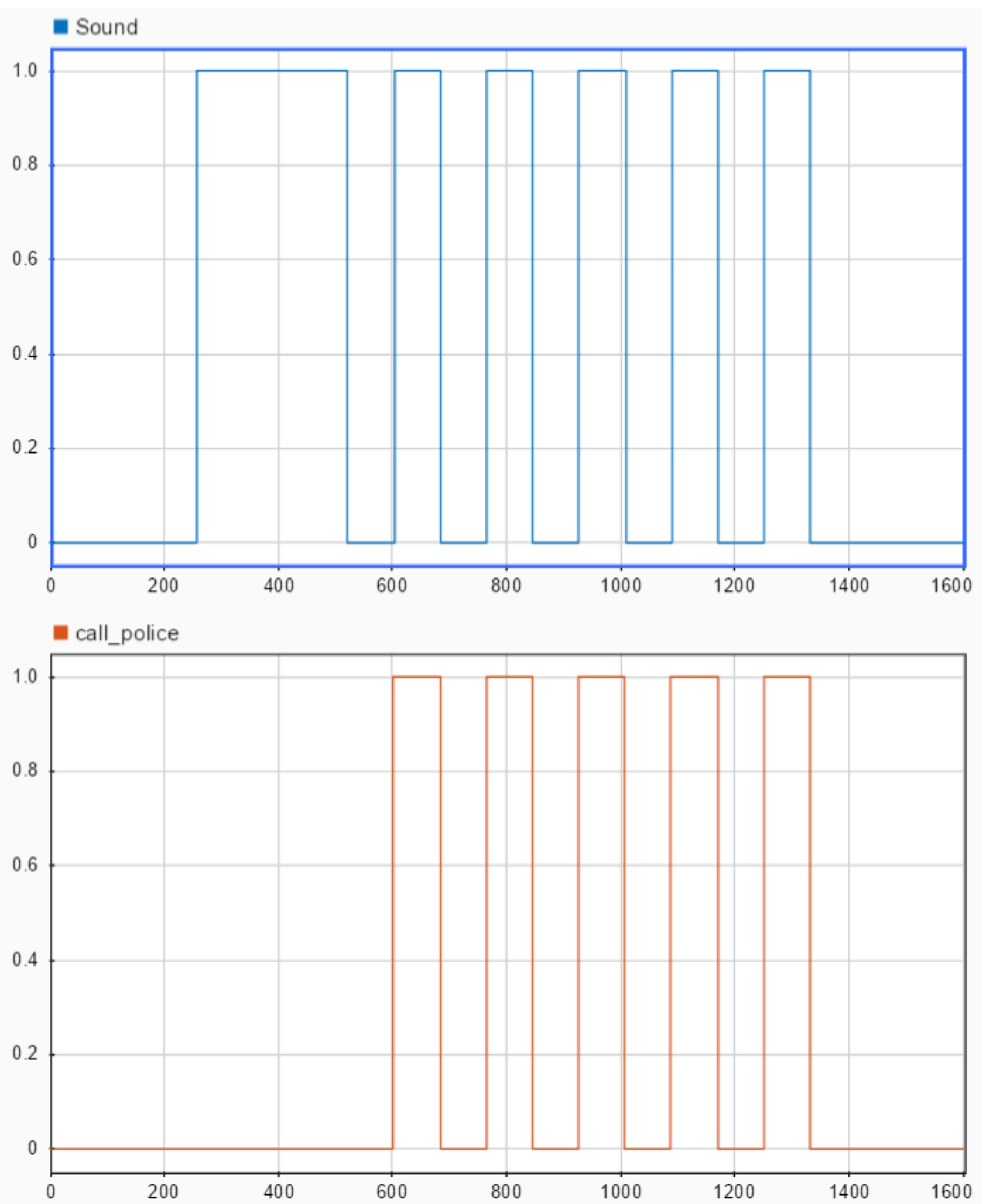
The Security System chart has inputs from several Manual Switch blocks and outputs to a pair of latch subsystems that connect to Display blocks. During simulation, you can:

- Enable the alarm and sensor subsystems and trigger intrusion detections by clicking the Switch blocks.
- Watch the chart animation highlight the various active states in the chart.
- View the output signals in the Scope block and in the Simulation Data Inspector.

To adjust the timing of the simulation, double-click the Function-Call Generator block and, in the dialog box, modify the **Sample time** field. For example, suppose that you set the sample time to 1 and start the simulation with all subsystems switched on and all sensor triggers switched off. During the simulation, you perform these actions:

1. At time *t* = 250 seconds, you trigger the door sensor. The alarm begins to sound (`Sound` = 1) so you immediately disable the alarm system. You switch off the trigger and turn the alarm back on.
2. At time *t* = 520 seconds, you trigger the window sensor and the alarm begins to sound (`Sound` = 0). This time, you do not disable the alarm. At around time *t* = 600, the security system calls the police (`call_police` = 1). The `Sound` and `call_police` signals continue to toggle between zero and one every 80 seconds.
3. At time *t* = 1400 seconds, you disable the alarm. The `Sound` and `call_police` signals stop toggling.

The Simulation Data Inspector shows the response of the `Sound` and `call_police` signals to your actions.



Sound
1.0
0.8
0.6
0.4
0.2
0
0
200
400
600
800
1000
1200
1400
1600
call_police
1.0
0.8
0.6
0.4
0.2
0
0
200
400
600
800
1000
1200
1400
1600

See Also

Related Examples

- “Model a Security System”

More About

- “Model Synchronous Subsystems by Using Parallelism” on page 1-36
- “Activate a Simulink Block by Sending Output Events”
- “Activate a Stateflow Chart by Sending Input Events”
- “Using Triggered Subsystems” (Simulink)
- “Using Function-Call Subsystems” (Simulink)
- “Schedule Chart Actions by Using Temporal Logic” on page 1-60

Monitor Chart Activity by Using Active State Data

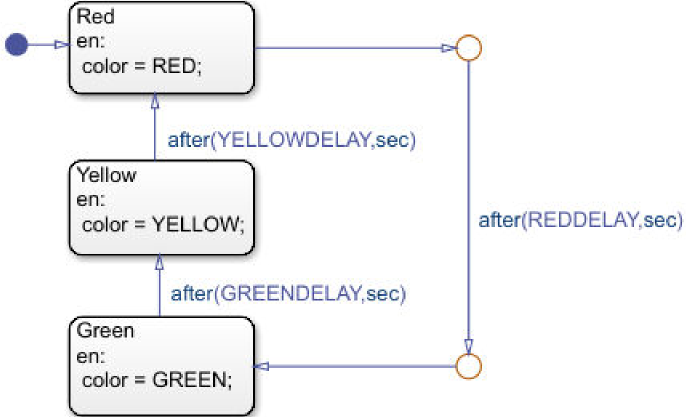
If your Stateflow chart includes data that is highly correlated to the chart hierarchy, you can simplify your design by using *active state data*. By enabling active state data, you can:

- Avoid manual data updates reflecting chart activity.
- Log and monitor chart activity in the Simulation Data Inspector.
- Use chart activity data to control other subsystems.
- Export chart activity data to other Simulink blocks.

For more information, see “Create a Hierarchy to Manage System Complexity” on page 1-29.

Active State Data

Using active state data output can simplify the design of some Stateflow charts. For example, in this model of a traffic signal, the state that is active determines the value of the symbol `color`. When you enable active state data, Stateflow can provide the color of the traffic signal by tracking state activity. Explicitly updating `color` is no longer necessary, so you can delete this symbol and simplify the design of the chart.



Stateflow provides active state data through an output port to Simulink or as local data to your chart. This table lists the different modes of active state data available.

Activity Mode	Data Type	Description
Self activity	Boolean	Is the state active?
Child activity	Enumeration	Which child state is active?
Leaf state activity	Enumeration	Which leaf state is active?

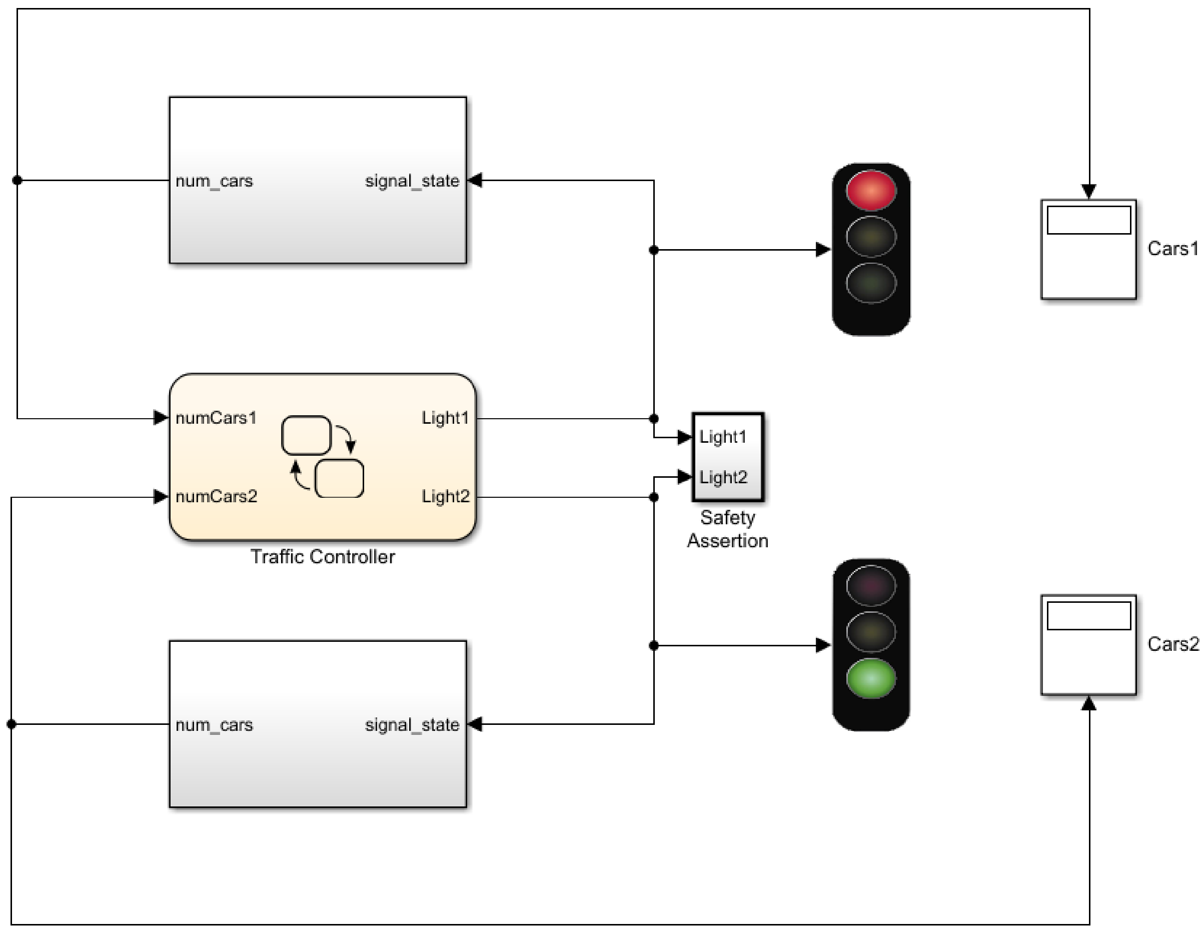
To enable active state data, use the Property Inspector.

1. Select the **Create output for monitoring** check box.
2. Select an activity mode from the drop-down list.
3. Enter the **Data name** for the active state data symbol.
4. (Optional) For Child or Leaf state activity, enter the **Enum name** for the active state data type.

By default, Stateflow reports state activity as output data. To change the scope of an active state data symbol to local data, use the Symbols window.

Example of Active State Data

The Stateflow example `sf_traffic_light` uses active state data to implement the controller system for a pair of traffic lights.



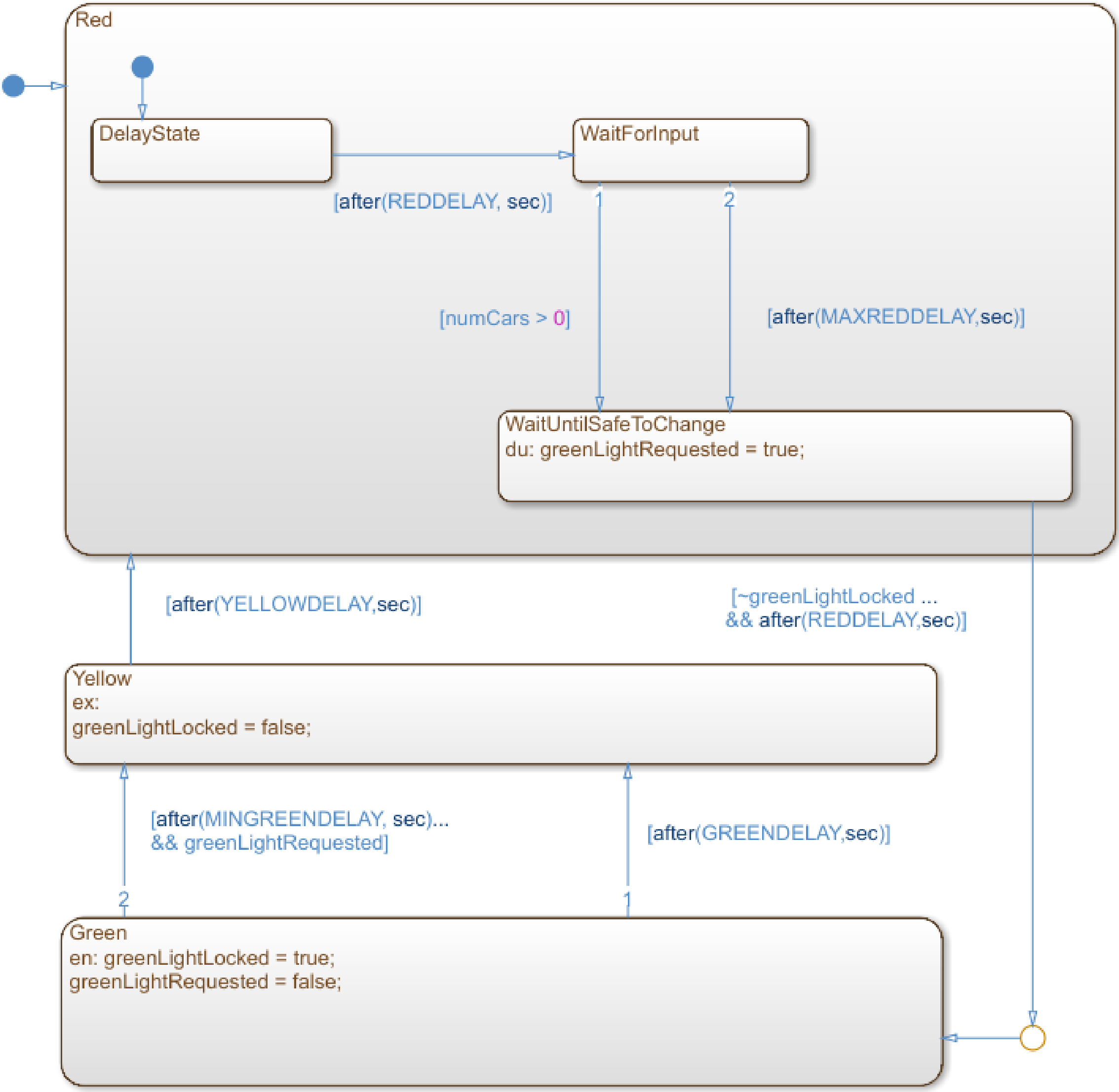
Inside the Traffic Controller chart, a pair of parallel subcharts manages the logic controlling the traffic lights. The subcharts have an identical hierarchy consisting of three child states: `Red`, `Yellow`, and `Green`. The output data `Light1` and `Light2` correspond to the active child states in the subcharts. These signals:

- Determine the phase of the animated traffic lights.
- Contribute to the number of cars waiting at each light.
- Drive a Safety Assertion subsystem verifying that the two traffic lights are never simultaneously green.

To see the subcharts inside the Traffic Controller chart, click the arrow at the bottom left corner of the chart.

Behavior of Traffic Controller Subcharts

Each traffic controller cycles through its child states, from `Red` to `Green` to `Yellow` and back to `Red`. Each state corresponds to a phase in the traffic light cycle. The output signals `Light1` and `Light2` indicate which state is active at any given time.



Red Light

When the Red state becomes active, the traffic light cycle begins. After a short delay, the controller checks for cars waiting at the intersection. If it detects at least one car, or if a

fixed length of time elapses, then the controller requests a green light by setting `greenLightRequest` to `true`. After making the request, the controller remains in the `Red` state for a short length of time until it detects that the other traffic signal is red. The controller then makes the transition to `Green`.

Green Light

When the `Green` state becomes active, the controller cancels its green light request by setting `greenLightRequest` to `false`. The controller sets `greenLightLocked` to `true`, preventing the other traffic signal from turning green. After some time, the controller checks for a green light request from the other controller. If it receives a request, or if a fixed length of time elapses, then the controller transitions to the `Yellow` state.

Yellow Light

Before transitioning to the `Red` state, the controller remains in the `Yellow` state for a fixed amount of time. When the `Yellow` state becomes inactive, the controller sets `greenLightLocked` to `false`, indicating that the other traffic light can safely turn green. The traffic light cycle then begins again.

Timing of Traffic Lights

Several parameters define the timing of the traffic light cycle. To change the values of these parameters, double-click the Traffic Controller chart and enter the new values in the Block Parameters dialog box.

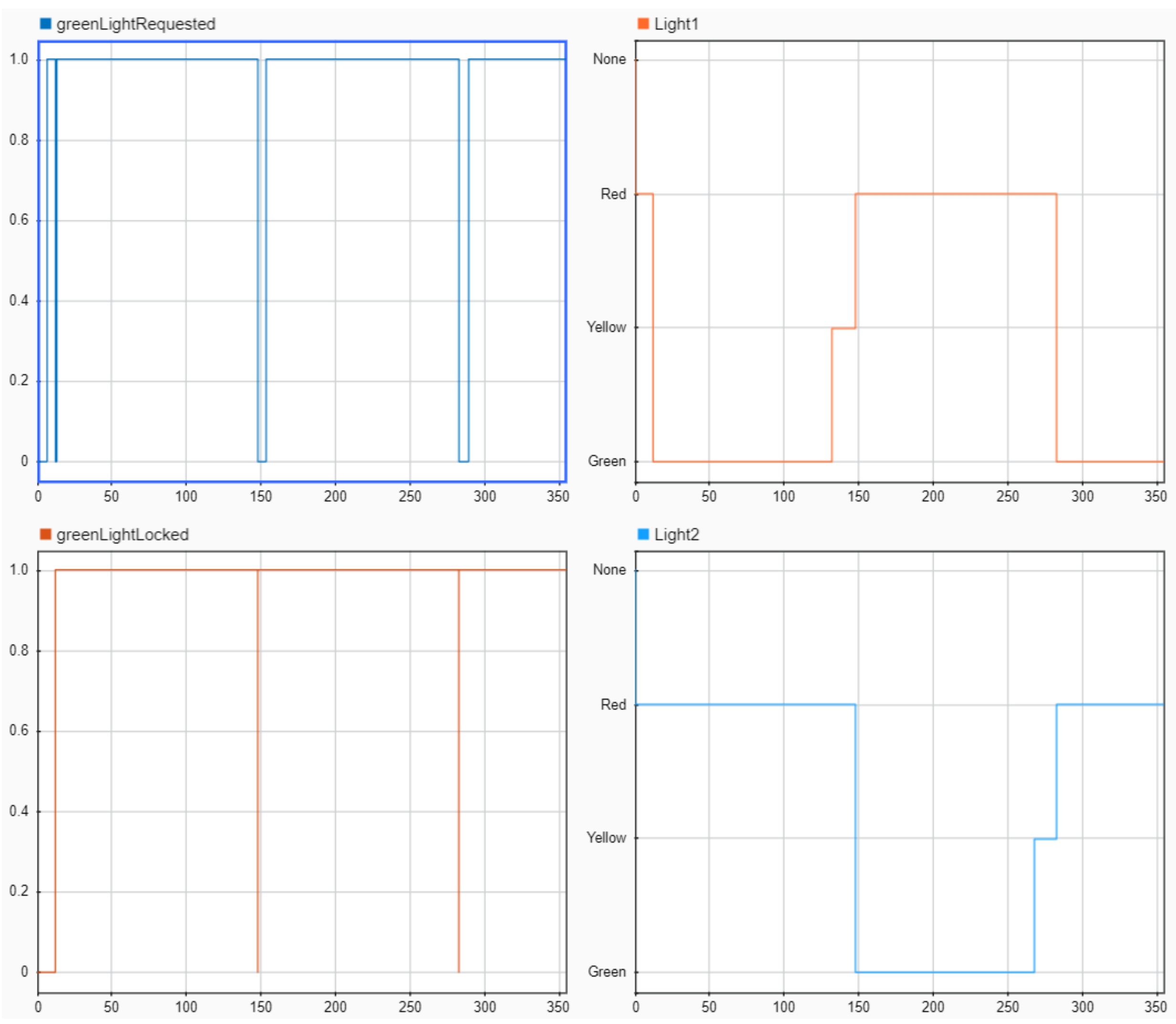
Parameter	Preset Value	Description
`REDDELAY`	6 seconds	Length of time before the controller begins to check for cars at the intersection. Also, minimum length of time before the traffic light can turn green after the controller requests a green light.
`MAXREDDELAY`	360 seconds	Maximum length of time that the controller checks for cars before requesting a green light.
`GREENDELAY`	180 seconds	Maximum length of time that the traffic light remains green.

Parameter	Preset Value	Description
MINGREENDELAY	120 seconds	Minimum length of time that the traffic light remains green.
YELLOWDELAY	15 seconds	Length of time that the traffic light remains yellow.

Explore the Example

1. In the Property Inspector, enable logging for these symbols:

   - greenLightRequested
   - greenLightLocked
   - Light1
   - Light2
2. Run the simulation.
3. In the Simulation Data Inspector, display the logged signals in separate axes. The Boolean signals greenLightRequested and greenLightLocked appear as numeric values of zero or one. The state activity signals Light1 and Light2 are shown as enumerated data with values of Green, Yellow, Red, and None.



To trace the chart activity during the simulation, you can use the zoom and cursor buttons in the Simulation Data Inspector. For example, this table details the activity during the first 300 seconds of the simulation.

Time	Description	Light 1	Light2	greenLight Requested	greenLight Locked
t = 0	At the start of the simulation, both traffic lights are red.	Red	Red	false	false
t = 6	After 6 seconds (REDDELAY), there are cars waiting in both streets. Both traffic lights request a green light by setting greenLightRequested = true.	Red	Red	true	false
t = 12	After another 6 seconds (REDDELAY): • Light 1 turns green, setting greenLightLocked = true and greenLightRequested = false. • Light 2 requests a green light by setting greenLightRequested = true.	Green	Red	false, then true	true
t = 132	After 120 seconds (MINGREENDELAY), Light 1 turns yellow.	Yellow	Red	true	true

Time	Description	Light 1	Light2	greenLight Requested	greenLight Locked
t = 147	After 15 seconds (YELLOWDELAY): • Light 1 turns red, setting greenLightLocked = false. • Light 2 turns green, setting greenLightLocked = true and greenLightRequested = false.	Red	Green	false	false, then true
t = 153	After 6 seconds (REDDELAY), Light 1 requests a green light by setting greenLightRequested = true.	Red	Green	true	true
t = 267	Light 2 turns yellow 120 seconds (MINGREENDELAY) after turning green.	Red	Yellow	true	true
t = 282	After 15 seconds (YELLOWDELAY): • Light 2 turns red, setting greenLightLocked = false. • Light 1 turns green, setting greenLightLocked = true and greenLightRequested = false.	Green	Red	false	false, then true

Time	Description	Light 1	Light2	greenLight Requested	greenLight Locked
t = 288	After 6 seconds (REDDELAY), Light 2 requests a green light by setting greenLightRequested = true.	Green	Red	true	true

The cycle repeats until the simulation ends at t = 1000 seconds.

See Also

Related Examples

- “Model An Intersection Of One-Way Streets”

More About

- “Create a Hierarchy to Manage System Complexity” on page 1-29
- “Schedule Chart Actions by Using Temporal Logic” on page 1-60
- “Monitor State Activity Through Active State Data”
- “Simplify Stateflow Charts by Incorporating Active State Output”
- “Check State Activity by Using the in Operator”
- “View State Activity by Using the Simulation Data Inspector”

Schedule Chart Actions by Using Temporal Logic

To define the behavior of a Stateflow chart in terms of simulation time, include *temporal logic operators* in the state and transition actions of the chart. Temporal logic operators are built-in functions that can tell you the length of time that a state remains active or that a Boolean condition remains true. With temporal logic, you can control the timing of:

- Transitions between states
- Function calls
- Changes in variable values

For more information, see “Define Chart Behavior by Using Actions” on page 1-23.

Temporal Logic Operators

The most common operators for absolute-time temporal logic are `after`, `elapsed`, and `duration`.

Operator	Syntax	Description
`after`	`after(n, sec)`	Returns `true` if `n` seconds of simulation time have elapsed since the activation of the associated state. Otherwise, the operator returns `false`.
`elapsed`	`elapsed(sec)`	Returns the number of seconds of simulation time that have elapsed since the activation of the associated state.
`duration`	`duration(C)`	Returns the number of seconds of simulation time that have elapsed since the Boolean condition `C` becomes `true`.

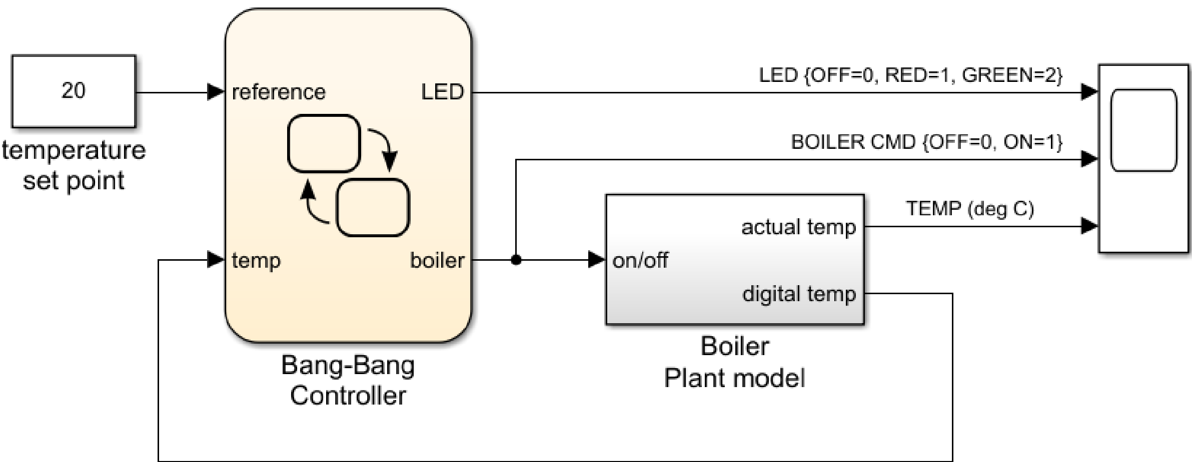
Each operator resets its associated timer to zero every time that:

- The state containing the operator reactivates.
- The source state for the transition containing the operator reactivates.
- The Boolean condition in a `duration` operator becomes `false`.

Note Some operators, such as `after`, support event-based temporal logic and absolute-time temporal logic in seconds (`sec`), milliseconds (`msec`), and microseconds (`usec`). For more information, see “Control Chart Execution by Using Temporal Logic”.

Example of Temporal Logic

The Stateflow example `sf_boiler` uses temporal logic to model a bang-bang controller that regulates the internal temperature of a boiler.



The example consists of a Stateflow chart and a Simulink subsystem. The Bang-Bang Controller chart compares the current boiler temperature to a reference set point and determines whether to turn on the boiler. The Boiler Plant subsystem models the dynamics inside the boiler, increasing or decreasing its temperature according to the status of the controller. The boiler temperature then goes back into the controller chart for the next step in the simulation.

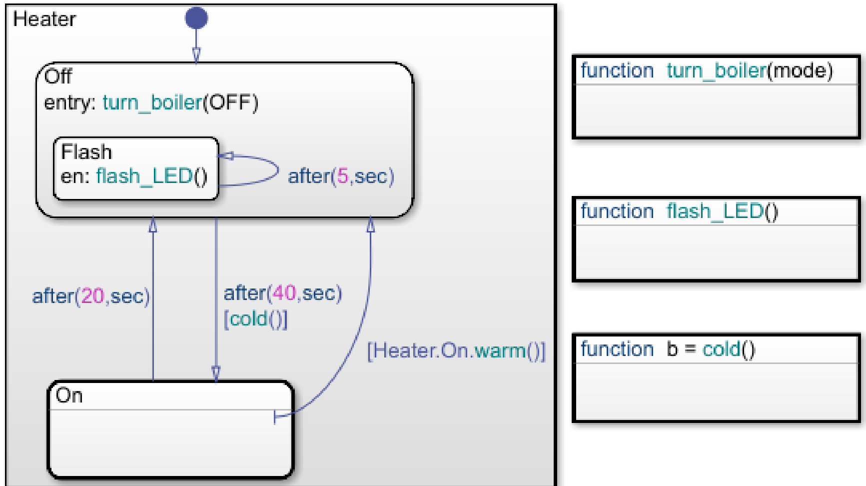
The Bang-Bang Controller chart uses the temporal logic operator `after` to:

- Regulate the timing of the bang-bang cycle as the boiler alternates between on and off.
- Control a status LED that flashes at different rates depending on the operating mode of the boiler.

The timers defining the behavior of the boiler and LED subsystems operate independently of one another without blocking or disrupting the simulation of the controller.

Timing of Bang-Bang Cycle

The Bang-Bang Controller chart contains a pair of substates representing the two operating modes of the boiler: `On` and `Off`. The graphical function `turn_boiler` updates the output data `boiler` to indicate which one of substates is active.

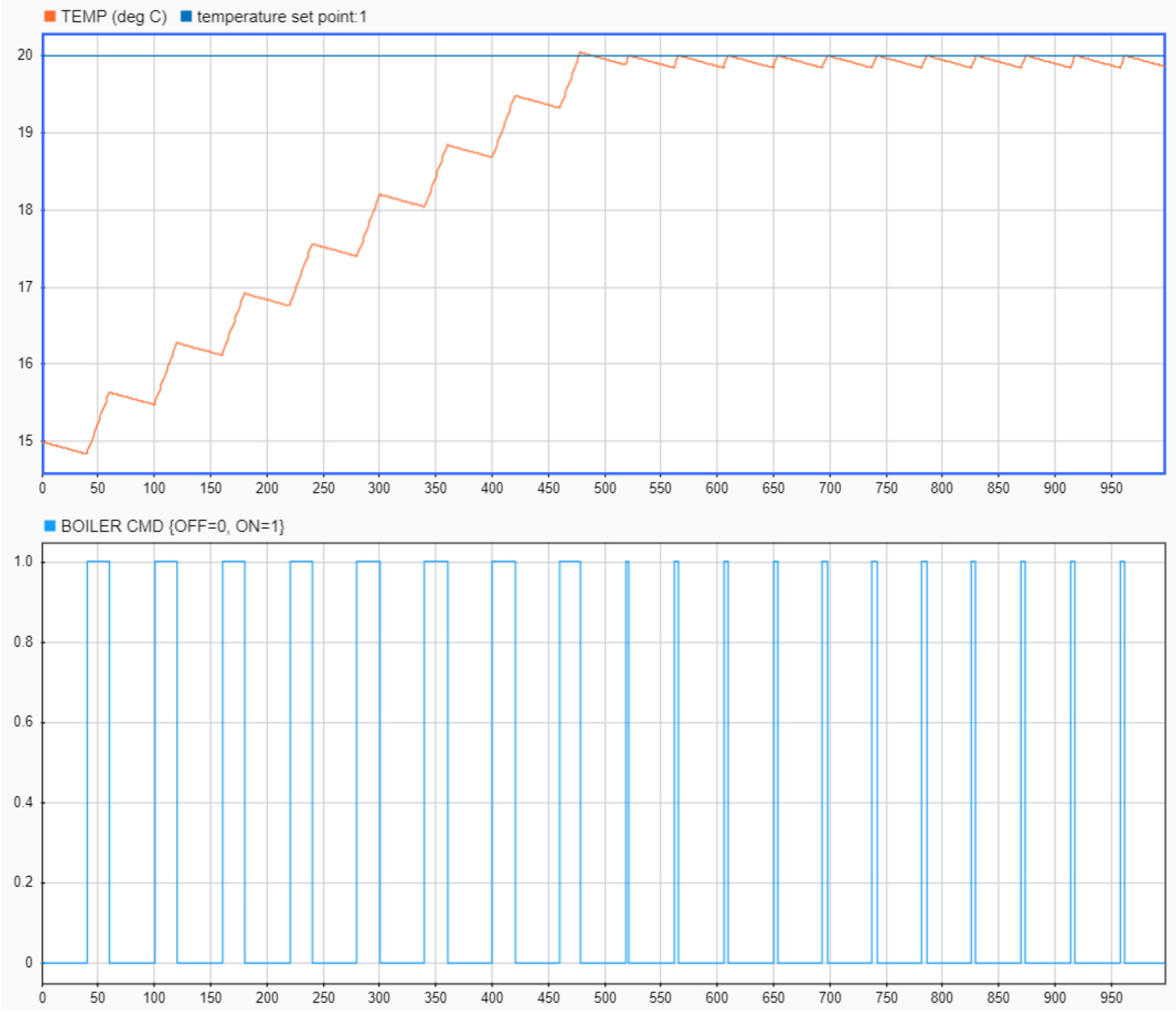


The actions guarding the transitions between the `On` and `Off` substates define the behavior of the bang-bang controller.

Transition	Action	Description
From `On` to `Off`	`after(20,sec)`	Transition to the `Off` state after spending 20 seconds in the `On` state.
From `Off` to `On`	`after(40,sec)` `[cold()]`	When the boiler temperature is below the reference set point (when the graphical function `cold()` returns `true`), transition to the `On` state after spending at least 40 seconds in the `Off` state.

Transition	Action	Description
From `On` to `Off`	`[Heater.On.warm()]`	When the boiler temperature is at or above the reference set point (when the graphical function `Heater.On.warm()` returns `true`), transition to the `Off` state.

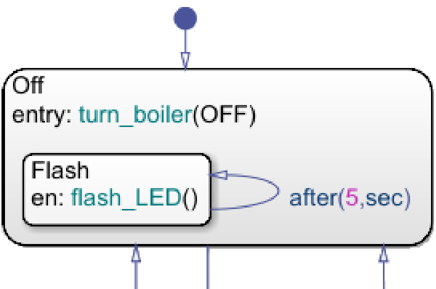
As a result of these transition actions, the timing of the bang-bang cycle depends on the current temperature of the boiler. At the start of the simulation, when the boiler is cold, the controller spends 40 seconds in the `Off` state and 20 seconds in the `On` state. At time $t = 478$ seconds, the temperature of the boiler reaches the reference point. From that point on, the boiler has to compensate only for the heat lost while in the `Off` state. The controller then spends 40 seconds in the `Off` state and 4 seconds in the `On` state.



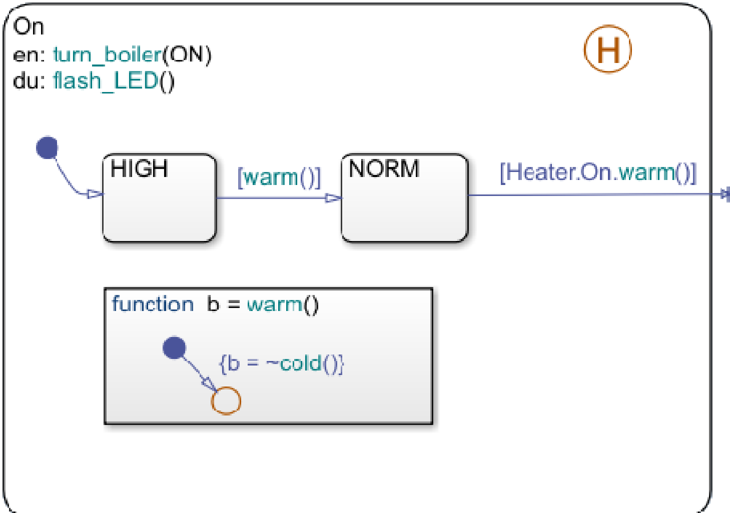
Timing of Status LED

The `Off` state contains a substate `Flash` with a self-loop transition guarded by the action `after(5,sec)`. Because of this transition, when the `Off` state is active, the substate

executes its entry action and calls the graphical function `flash_LED` every 5 seconds. The function toggles the value of the output symbol `LED` between 0 and 1.



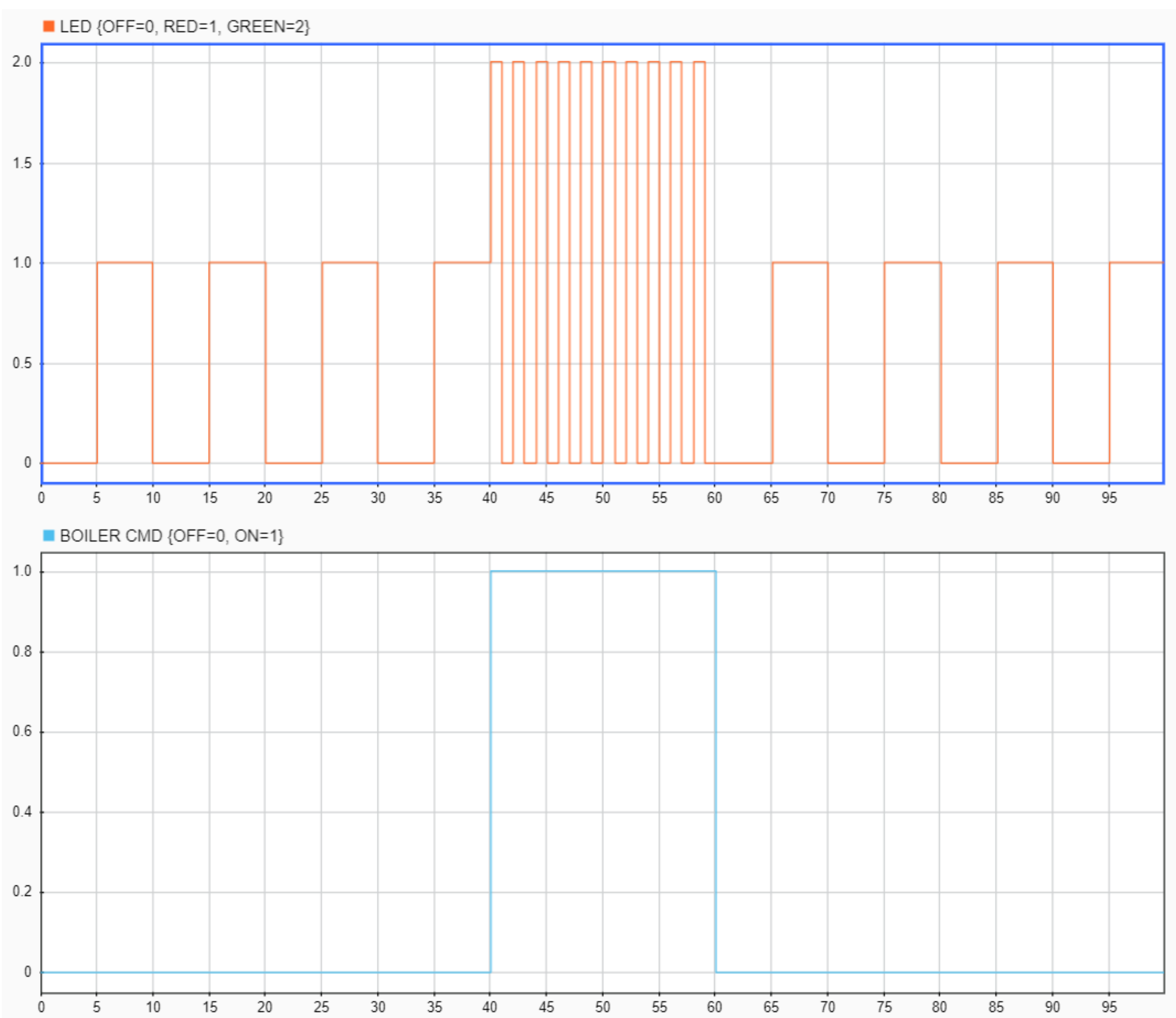
The `On` state calls the graphical function `flash_LED` as a state action of type `during`. When the `On` state is active, it calls the function at every time step of the simulation (in this case, every second), toggling the value of the output symbol `LED` between 0 and 2.



As a result, the timing of the status LED depends on the operating mode of the boiler. For example:

- From t = 0 to t = 40 seconds, the boiler is off and the `LED` signal alternates between 0 and 1 every 5 seconds.

- From t = 40 to t = 60 seconds, the boiler is on and the LED signal alternates between 0 and 2 every second.
- From t = 60 to t = 100 seconds, the boiler is once again off and the LED signal alternates between 0 and 1 every 5 seconds.



Explore the Example

Use additional temporal logic to investigate how the timing of the bang-bang cycle changes as the temperature of the boiler approaches the reference set point.

1. Enter new state actions that call the operators `elapsed` and `duration`.

   - In the `On` state, let `Timer1` be the length of time that the `On` state is active:

     ```
     en,du,ex: Timer1 = elapsed(sec)
     ```
   - In the `Off` state, let `Timer2` be the length of time that the boiler temperature is at or above the reference set point:

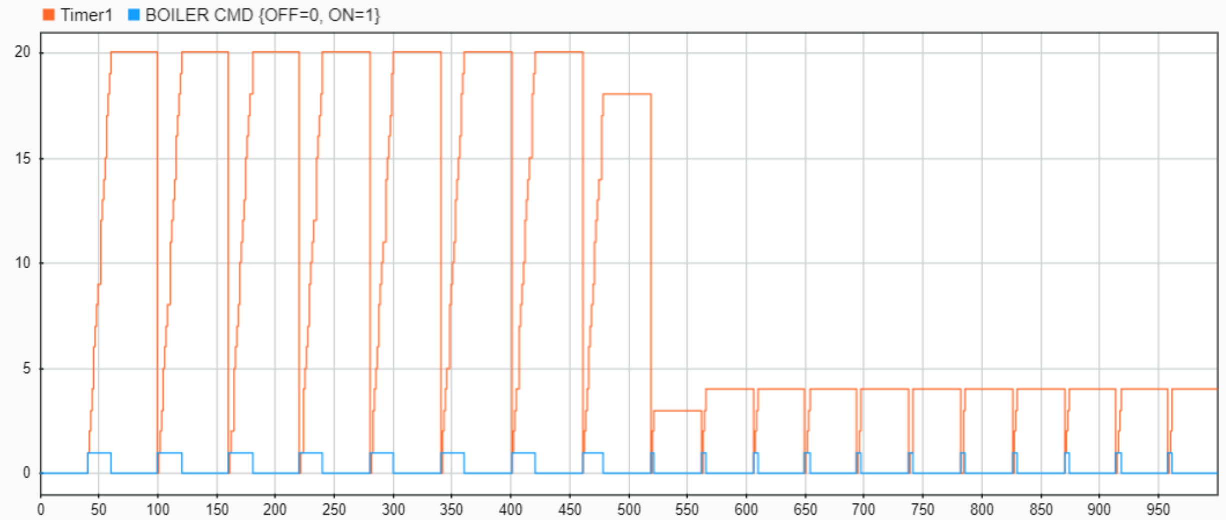
     ```
     en,du,ex: Timer2 = duration(temp>=reference)
     ```

   The label `en,du,ex` indicates that these actions take place whenever the corresponding state is active.

2. In the Symbols window, click the **Resolve Undefined Symbols** icon. The Stateflow editor resolves the symbols `Timer1` and `Timer2` as output data.
3. In the Property Inspector, enable logging for these symbols:

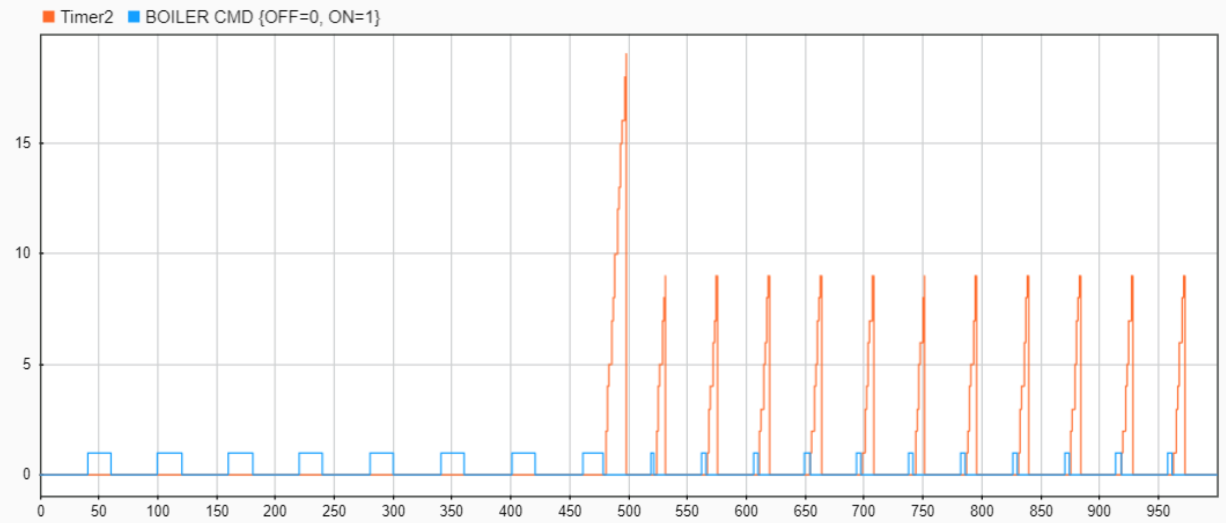
   - `boiler`
   - `Timer1`
   - `Timer2`
4. Run the simulation.
5. In the Simulation Data Inspector, display the signals `boiler` and `Timer1` in the same set of axes. The plot shows that:

   - The `On` phase of the bang-bang cycle typically lasts 20 seconds when the boiler is cold and 4 seconds when the boiler is warm.
   - The first time that the boiler reaches the reference temperature, the cycle is interrupted prematurely and the controller stays in the `On` state for only 18 seconds.
   - When the boiler is warm, the first cycle is slightly shorter than the subsequent cycles, as the controller stays in the `On` state for only 3 seconds.



6 In the Simulation Data Inspector, display the signals `boiler` and `Timer2` in the same set of axes. The plot shows that:

- Once the boiler is warm, it typically takes 9 seconds to cool in the `Off` phase of the bang-bang cycle.
- The first time that the boiler reaches the reference temperature, it takes more than twice as long to cool (19 seconds).



The shorter cycle and longer cooling time are a consequence of the substate hierarchy inside the `On` state. When the boiler reaches the reference temperature for the first time, the transition from `HIGH` to `NORM` keeps the controller on for an extra time step, resulting in a warmer-than-normal boiler. In later cycles, the history junction (H) causes the `On` phase to start with an active `NORM` substate. The controller then turns off immediately after the boiler reaches the reference temperature, resulting in a cooler boiler.

See Also

Related Examples

- “Bang-Bang Control Using Temporal Logic”

More About

- “Define Chart Behavior by Using Actions” on page 1-23
- “Control Chart Execution by Using Temporal Logic”
- “Reuse Logic Patterns by Defining Graphical Functions”
- “History Junctions”

Installing Stateflow Software

Installation Instructions

Stateflow software runs on Windows® and UNIX® operating systems. Your MATLAB installation documentation provides all the information you need to install Stateflow software. Before installing the product, you must obtain and activate a license (see instructions in your MATLAB installation documentation) and install prerequisite software (see “Prerequisite Software” on page 1-70 for a complete list).

Prerequisite Software

Before installing Stateflow software, you need the following products:

- MATLAB
- Simulink
- C or C++ compiler supported by the MATLAB technical computing environment

  The compiler is required for compiling code generated by Stateflow software for simulation.

  The 64–bit Windows version of the Stateflow product comes with a default C compiler, `LCC-win64`. `LCC-win64` is used for simulation and acceleration. `LCC-win64` is only used when another compiler has not been configured in MATLAB.

  Note The `LCC-win64` compiler is not available as a general compiler for use with the command line `MEX` in MATLAB. It is a C compiler only, and cannot be used for `SIL/PIL` modes.

  For platforms other than Microsoft® Windows or to install a different compiler, see “Set Up Your Own Target Compiler” on page 1-71.

Product Dependencies

For information about product dependencies and requirements, see System Requirements.

Set Up Your Own Target Compiler

If you have multiple compilers that MATLAB supports on your system, MATLAB selects one as your default compiler. You can change the default compiler by calling the `mex –setup` command, and following the instructions. For a list of supported compilers, see `www.mathworks.com/support/compilers/current_release/`.

Note If you are using Microsoft Visual C++® 2010 Professional (or earlier), the generated C code cannot contain any C structure greater than 2 GB. In a single chart, do not use data with an aggregate size greater than 2 GB or 400 MB with debugging enabled.

Using Stateflow Software on a Laptop Computer

If you plan to run the Microsoft Windows version of the Stateflow product 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 box.
3. Select the **Settings** panel on the **Display Properties** dialog box.
4. Choose a setting that is more than 256 colors and click **OK**.

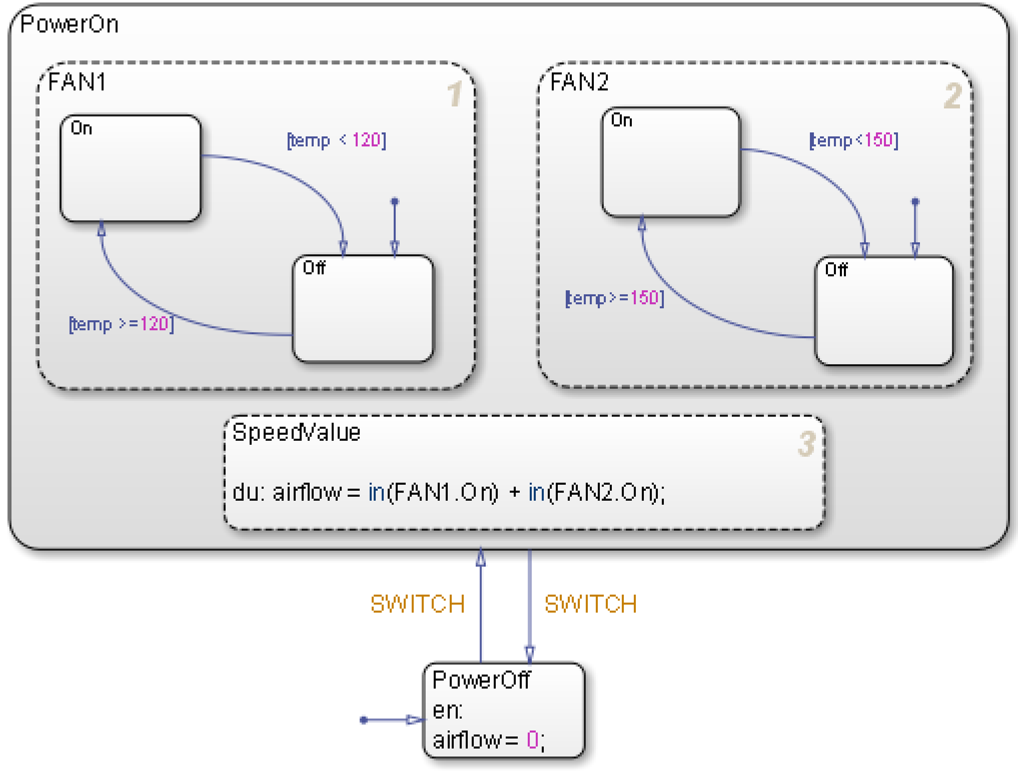
The Stateflow Chart You Will Build

To get hands-on experience using Stateflow software, you will build a Stateflow chart in incremental steps that follow the basic workflow described in “Use C Chart to Model Event-Driven System”. To give you a context for your development efforts, this chapter describes the purpose and function of the chart you will build and explains how it interfaces with a Simulink model. You will also learn how to run a completed version of the model from the MATLAB command line.

- “The Stateflow Chart” on page 2-2
- “How the Stateflow Chart Works with the Simulink Model” on page 2-6
- “A Look at the Physical Plant” on page 2-7
- “Running the Model” on page 2-9

The Stateflow Chart

You will build a Stateflow chart that maintains air temperature at 120 degrees in a physical plant. The Stateflow controller operates two fans. The first fan turns on if the air temperature rises above 120 degrees and the second fan provides additional cooling if the air temperature rises above 150 degrees. When completed, your Stateflow chart should look something like this:



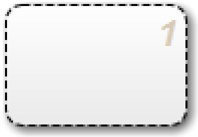
As you can see from the title bar, the chart is called Air Controller and is part of a Simulink model called `sf_aircontrol`. When you build this chart, you will learn how to work with the following elements of state-transition charts:

Exclusive (OR) states. States that represent mutually exclusive modes of operation. No two exclusive (OR) states can ever be active or execute at the same time. Exclusive (OR) states are represented graphically by a solid rectangle:

The Air Controller chart contains six exclusive (OR) states:

- `PowerOn`
- `PowerOff`
- `FAN1.On`
- `FAN1.Off`
- `FAN2.On`
- `FAN2.Off`

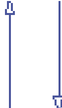
Parallel (AND) states. States that represent independent modes of operation. Two or more parallel (AND) states at the same hierarchical level can be active concurrently, although they execute in a serial fashion. Parallel (AND) states are represented graphically by a dashed rectangle with a number indicating execution order:



The Air Controller chart contains three parallel (AND) states:

- `FAN1`
- `FAN2`
- `SpeedValue`

Transitions. Graphical objects that link one state to another and specify a direction of flow. Transitions are represented by unidirectional arrows:



The Air Controller chart contains six transitions, from

- `PowerOn` to `PowerOff`
- `PowerOff` to `PowerOn`
- `FAN1.On` to `FAN1.Off`
- `FAN1.Off` to `FAN1.On`
- `FAN2.On` to `FAN2.Off`
- `FAN2.Off` to `FAN2.On`

Default transitions. Graphical objects that 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 are represented by arrows with a closed tail:

The Air Controller chart contains default transitions:

- At the chart level, the default transition indicates that the state `PowerOff` is activated (wakes up) first when the chart is activated.
- In the `FAN1` and `FAN2` states, the default transitions specify that the fans be powered off when the states are activated.

State actions. Actions executed based on the status of a state.

The Air Controller chart contains two types of state actions:

- `entry` (`en`) action in the `PowerOff` state. Entry actions are executed when the state is entered (becomes active).
- `during` (`du`) action in the `SpeedValue` state. During actions are executed for a state while it is active and no valid transition to another state is available.

Other types of state actions

There are other types of state actions besides `entry` and `during`, but they involve concepts that go beyond the scope of this guide. For more information, see "Syntax for States and Transitions".

Conditions. Boolean expressions that allow a transition to occur when the expression is true. Conditions appear as labels for the transition, enclosed in square brackets (`[ ]`).

The Air Controller chart provides conditions on the transitions between `FAN1.On` and `FAN1.Off`, and between `FAN2.On` and `FAN2.Off`, based on the air temperature of the physical plant at each time step.

Events. Objects that can trigger a variety of activities, including:

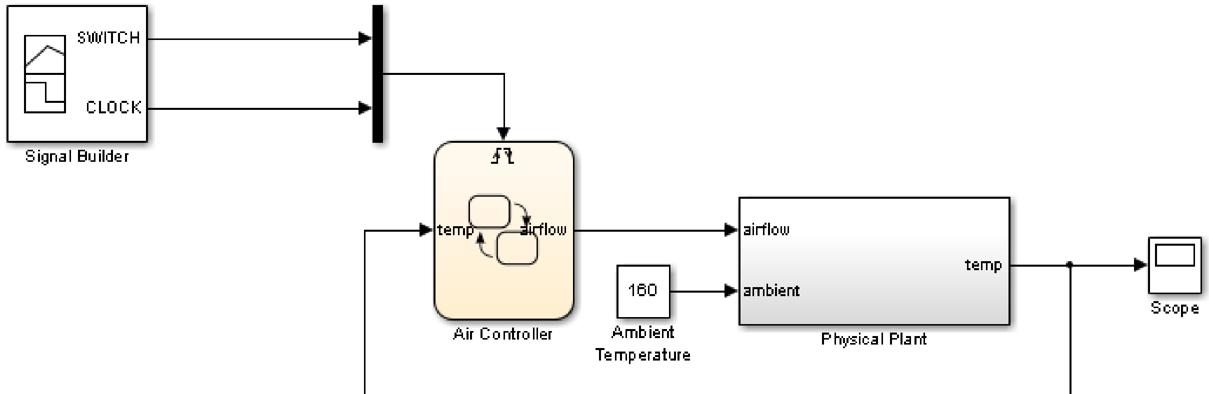
- Waking up a Stateflow chart
- Causing transitions to occur from one state to another (optionally in conjunction with a condition)
- Executing actions

The Air Controller chart contains two edge-triggered events:

- `CLOCK` wakes up the Stateflow chart at each rising or falling edge of a square wave signal.
- `SWITCH` allows transitions to occur between `PowerOff` and `PowerOn` at each rising or falling edge of a pulse signal.

How the Stateflow Chart Works with the Simulink Model

The Stateflow chart you will build appears as a block named Air Controller that is connected to the model of a physical plant in the Simulink `sf_aircontrol` model. Here is the top-level view of the model:

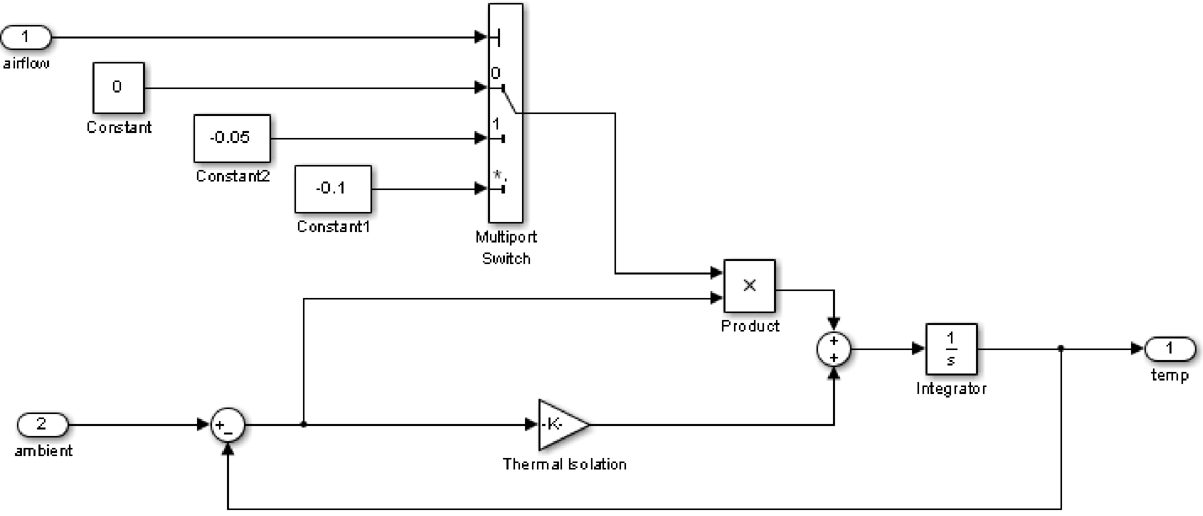


The Simulink model passes the temperature of the plant as an input `temp` to the Stateflow Air Controller block. Based on the temperature of the plant, the controller activates zero, one, or two fans, and passes back to the model an output value `airflow` that indicates how fast the air is flowing. The amount of cooling activity depends on the speed of the fans. As air flows faster, cooling activity increases. The model uses the value of `airflow` to simulate the effect of cooling when it computes the air temperature in the plant over time.

The Signal Builder block in the Simulink model sends a square wave signal (`CLOCK`) to wake up the Stateflow chart at regular intervals and a pulse signal (`SWITCH`) to cycle the power on and off for the control system modeled by the Stateflow chart. You will learn more about these design elements in “Implementing the Triggers” on page 6-2.

A Look at the Physical Plant

Simulink software models the plant using a subsystem called Physical Plant, which contains its own group of Simulink blocks. The subsystem provides a graphical hierarchy for the blocks that define the behavior of the Simulink model. The inputs, airflow speed and ambient temperature, model the effects of the controller activity on plant temperature. Here is a look inside the Physical Plant subsystem:



In this model, the internal temperature of the plant attempts to rise to achieve steady state with the ambient air temperature, set at a constant 160 degrees (as shown in “How the Stateflow Chart Works with the Simulink Model” on page 2-6). The rate at which the internal temperature rises depends in part on the degree of thermal isolation in the plant and the amount of cooling activity.

Thermal isolation measures how much heat flows into a closed structure, based on whether the structure is constructed of materials with insulation or conduction properties. Here, thermal isolation is represented by a Gain block, labeled `Thermal Isolation`. The Gain block provides a constant multiplier that is used in calculating the temperature in the plant over time.

Cooling activity is modeled using a constant multiplier, derived from the value of `airflow`, an output from the Stateflow chart. The chart assigns `airflow` one of three

cooling factors, each a value that serves as an index into a multiport switch. Using this index, the multiport switch selects a cooling activity multiplier that is directly proportional to the cooling factor, as follows:

Cooling Factor (Value of Airflow)	What It Means	Cooling Activity
0	No fans are running. The value of `temp` is not lowered.	0
1	One fan is running. The value of `temp` is lowered by the cooling activity multiplier.	-0.05
2	Two fans are running. The value of `temp` is lowered by the cooling activity multiplier.	-0.1

Over time, the subsystem calculates the cooling effect inside the plant, taking into account thermal isolation and cooling activity. The cooling effect is the time-derivative of the temperature and is the input to the Integrator block in the Physical Plant subsystem. Let the variable `temp_change` represent the time derivative of temperature. Note that `temp_change` can be a warming or cooling effect, depending on whether it is positive or negative, based on this equation:

$$temp_change = ((ambient - temp) * (thermalisolationmultiplier)) + ((ambient - temp) * (coolingfactor))$$

The Integrator block computes its output `temp` from the input `temp_change`, as follows:

$$temp(t) = \int_{t0}^{t} temp_change(t)dt + 70$$

Note In this model, the initial condition of the Integrator block is 70 degrees.

`temp` is passed back to the Stateflow Air Controller chart to determine how much cooling is required to maintain the ideal plant temperature.

Running the Model

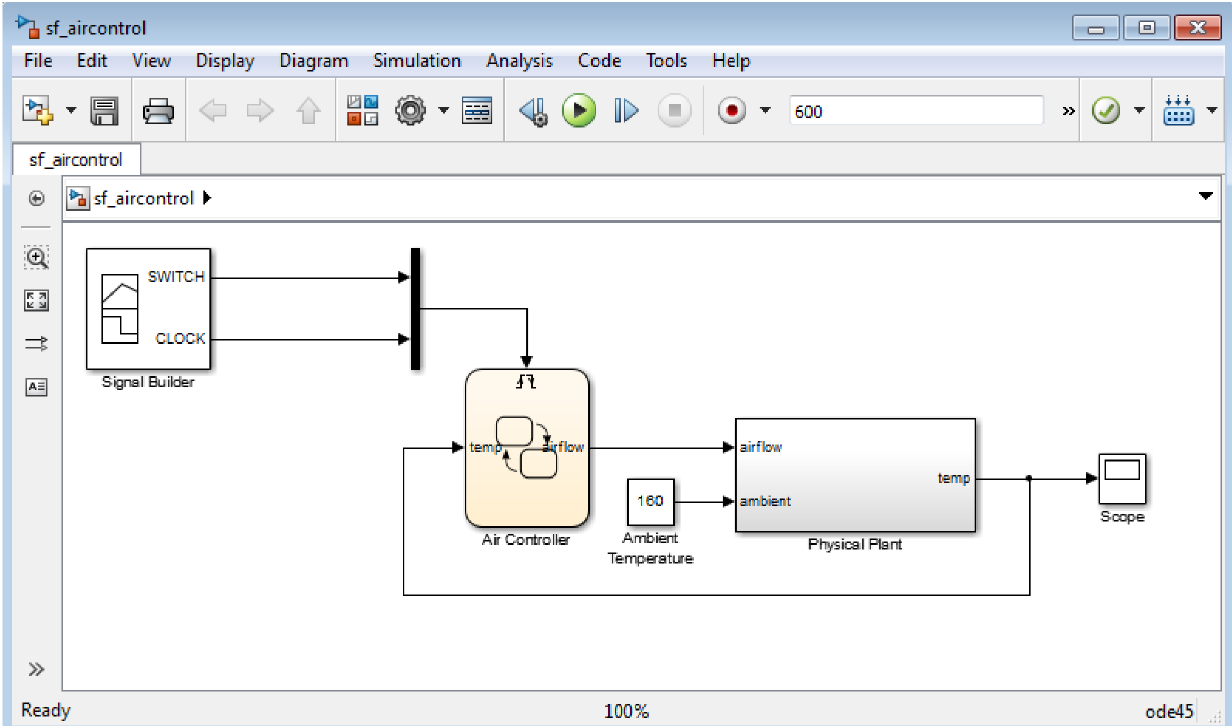
To see how the sf_aircontrol model works, you can run a completed, tested version, which includes the Stateflow chart you will build. Here's how to do it:

1. Start MATLAB software.

   If you need instructions, consult your MATLAB documentation.

2. Type sf_aircontrol at the command line.

   This command starts Simulink software and opens the sf_aircontrol model:



3. Double-click the Air Controller block to open the Stateflow chart.
4. Double-click the Scope block to display the changes in temperature over time as the model runs.

Tip Position the Air Controller chart and the Scope window so they are both visible on your desktop.

---

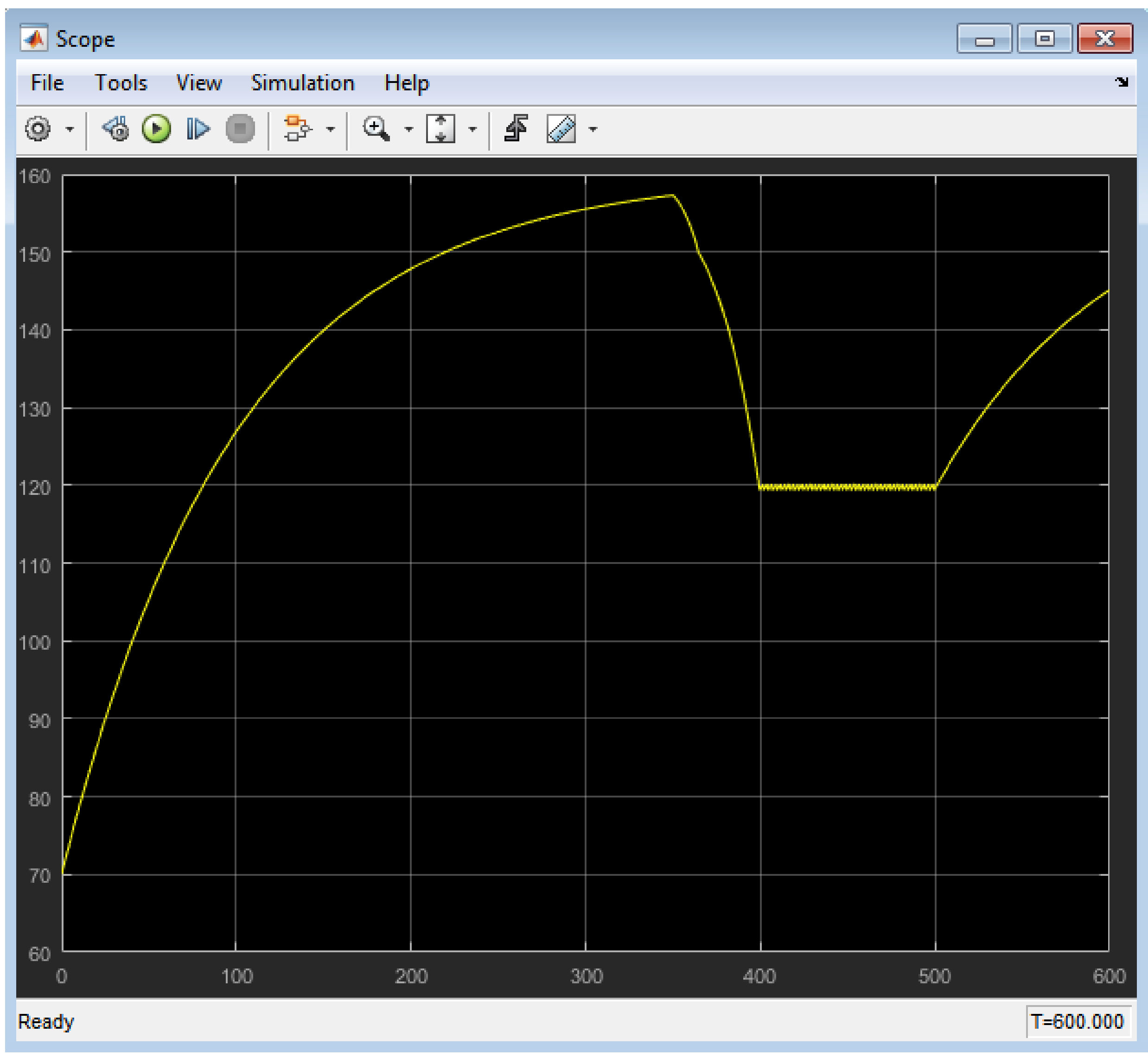
5 Start simulation in the Air Controller chart by selecting **Simulation** > **Run**.

As the simulation runs, the chart becomes active (wakes up) in the `PowerOff` state. Notice in the Scope that until `PowerOn` becomes active, the temperature rises unchecked. After approximately 350 seconds into the simulation, a rising edge signal switches power on and the fans become active.

Note Simulation time can be faster than elapsed time.

When the temperature rises above 120 degrees, `FAN1` cycles on. When the temperature exceeds 150 degrees, `FAN2` cycles on to provide additional cooling. Ultimately, `FAN1` succeeds in maintaining the temperature at 120 degrees until a falling edge signal switches power off again at 500 seconds. Then, the temperature begins to rise again.

The Scope captures the temperature fluctuations:



Stopping or pausing simulation

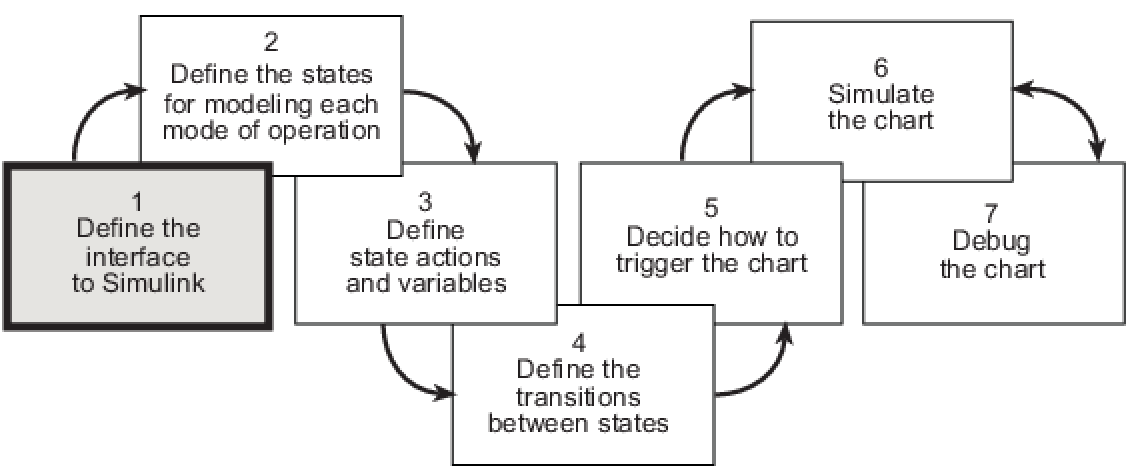
You can stop or pause simulation at any time.

- To stop simulation, select **Simulation** > **Stop**.
- To pause simulation, select **Simulation** > **Pause**.

6 Close the model.

Where to go next. Now you are ready to start building the Air Controller chart. Begin at phase 1 of the workflow: "Implementing the Interface with Simulink" on page 3-2.

Defining the Interface to the Simulink Model



In phase 1 of this workflow, you *define the interface to the Simulink model*.

Implementing the Interface with Simulink

In this section...

"Build It Yourself or Use the Supplied Model" on page 3-2
"Design Considerations for Defining the Interface" on page 3-2
"Adding a Stateflow Block to a Simulink Model" on page 3-3
"Defining the Inputs and Outputs" on page 3-8
"Connecting the Stateflow Block to the Simulink Subsystem" on page 3-15

Build It Yourself or Use the Supplied Model

To implement the interface yourself, work through the exercises in this section. Otherwise, open the supplied model by entering this command at the MATLAB prompt:

```
addpath(fullfile(docroot, 'toolbox', 'stateflow', 'gs', 'examples'))
Stage1Interface
```

Design Considerations for Defining the Interface

The following sections describe the rationale for the input and output of the Stateflow chart.

Inputs Required from Simulink Model

Type of Input. Temperature of the physical plant

Rationale. The purpose of the chart is to control the air temperature in a physical plant. The goal is to maintain an ideal temperature of 120 degrees by activating one or two cooling fans if necessary. The chart must check the plant temperature over time to determine the amount of cooling required.

Properties of Input. The properties of the temperature input are as follows:

Property	Value
Name	`temp`
Scope	Input
Size	Inherit from Simulink input signal for compatibility

Property	Value
Data type	Inherit from Simulink input signal for compatibility
Port	1

Outputs Required from Stateflow Chart

Type of Output. Speed of airflow, based on how many fans are operating

Rationale. When the Simulink subsystem determines the temperature of the physical plant over time, it needs to account for the speed of the airflow. Airflow speed is directly related to the amount of cooling activity generated by the fans. As more fans are activated, cooling activity increases and air flows faster. To convey this information, the Stateflow chart outputs a value that indicates whether 0, 1, or 2 fans are running. The Simulink subsystem uses this value as an index into a multiport switch, which outputs a cooling activity value, as described in "A Look at the Physical Plant" on page 2-7.

Properties of Output. The properties of the airflow output are as follows:

Property	Value
Name	`airflow`
Scope	Output
Data type	8-bit unsigned integer (`uint8`) (The values can be only 0, 1, or 2.)
Port	1

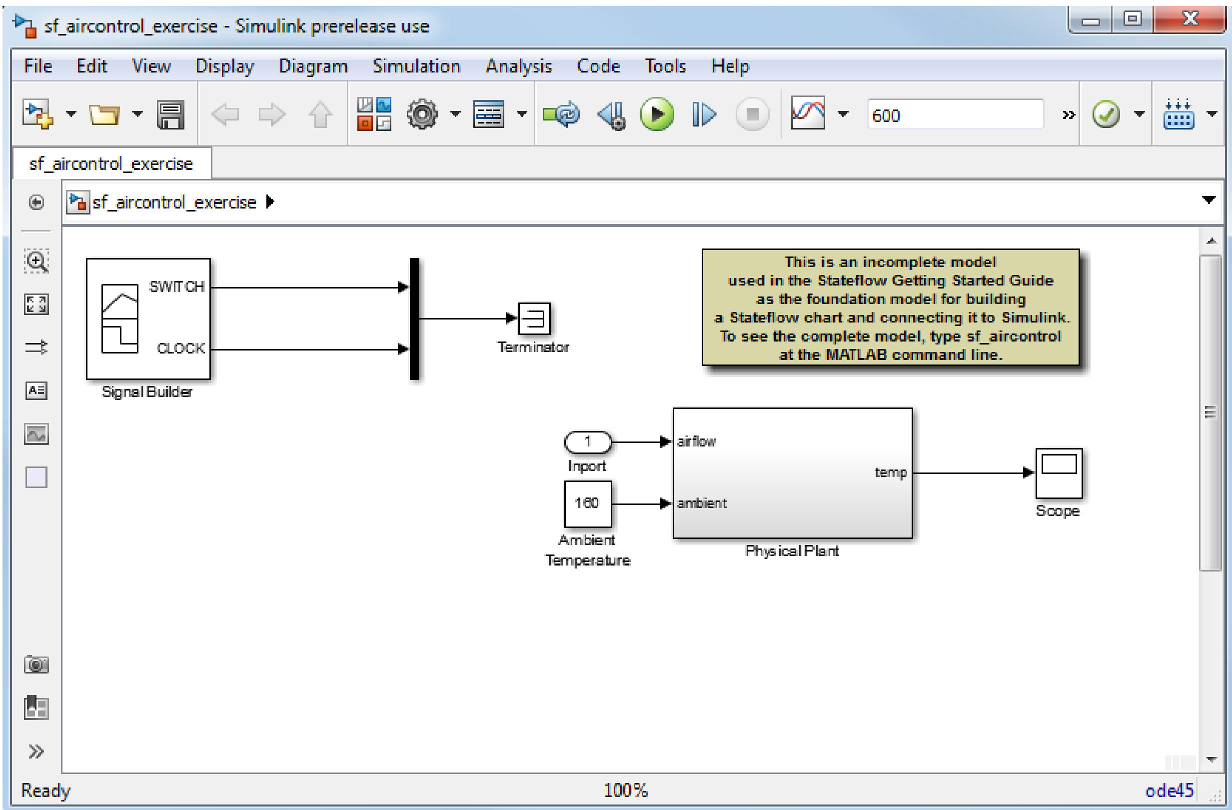
Adding a Stateflow Block to a Simulink Model

To begin building your Stateflow chart, you will add a Stateflow block to a partially built Simulink model called `sf_aircontrol_exercise`, which contains the Physical Plant subsystem, described in "A Look at the Physical Plant" on page 2-7.

To add a Stateflow block to an existing Simulink model:

1. Open the Simulink model by typing `sf_aircontrol_exercise` at the MATLAB command prompt.

   The model opens on your desktop:



The model is incomplete because it does not include the Stateflow chart that you will build as you work through the exercises in this guide. Instead, the model contains several nonfunctional blocks: the Terminator, Inport, and Annotation blocks.

2 Delete the nonfunctional blocks and their connectors.

Tip Hold down the **Shift** key to select multiple objects, and then press **Delete**.

Your model should now look like this:

3 Save the model as **Stage1Interface**:

a Create a *new* local folder for storing your working model.

b In the Simulink model window, select **File > Save As**.

c Navigate to the new folder.

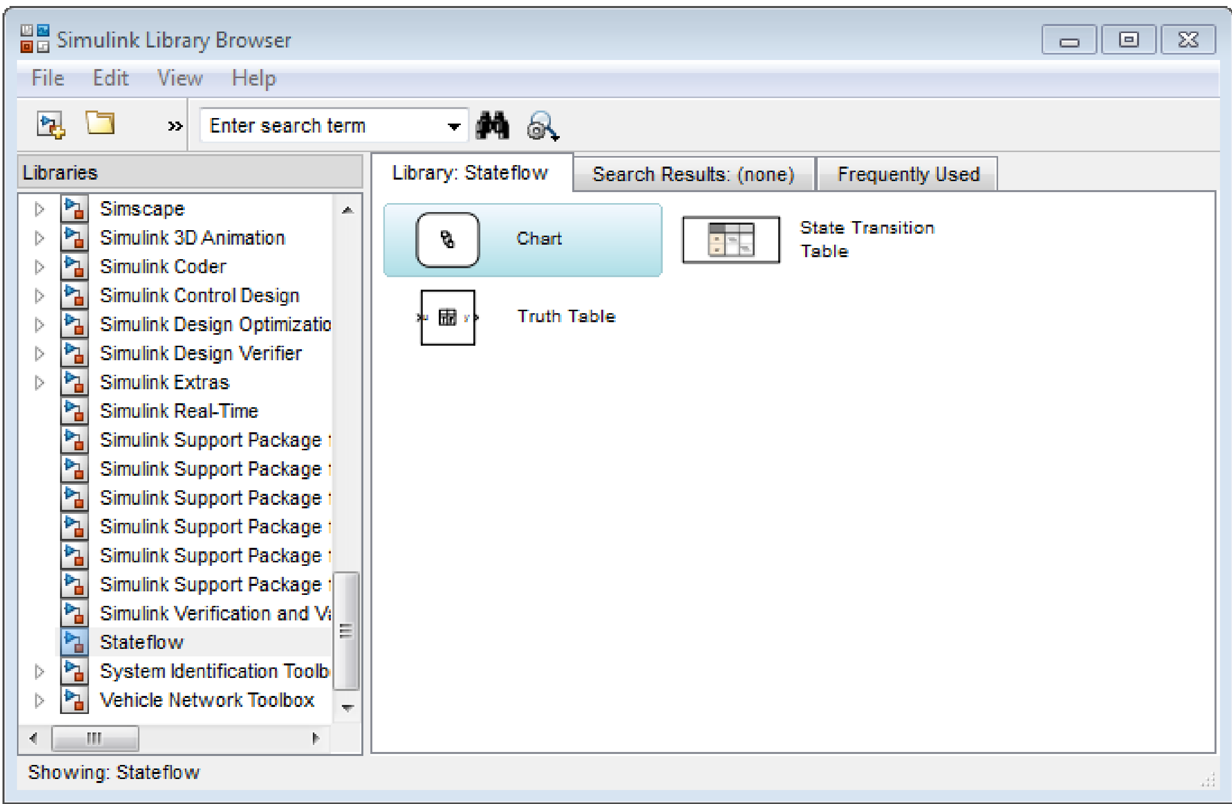
d Enter `Stage1Interface` as the file name.

e Leave the default type as **Simulink Models**.

f Click **Save**.

4 On the toolbar of the Simulink model, click the Library Browser icon:

The Simulink Library Browser opens on your desktop:

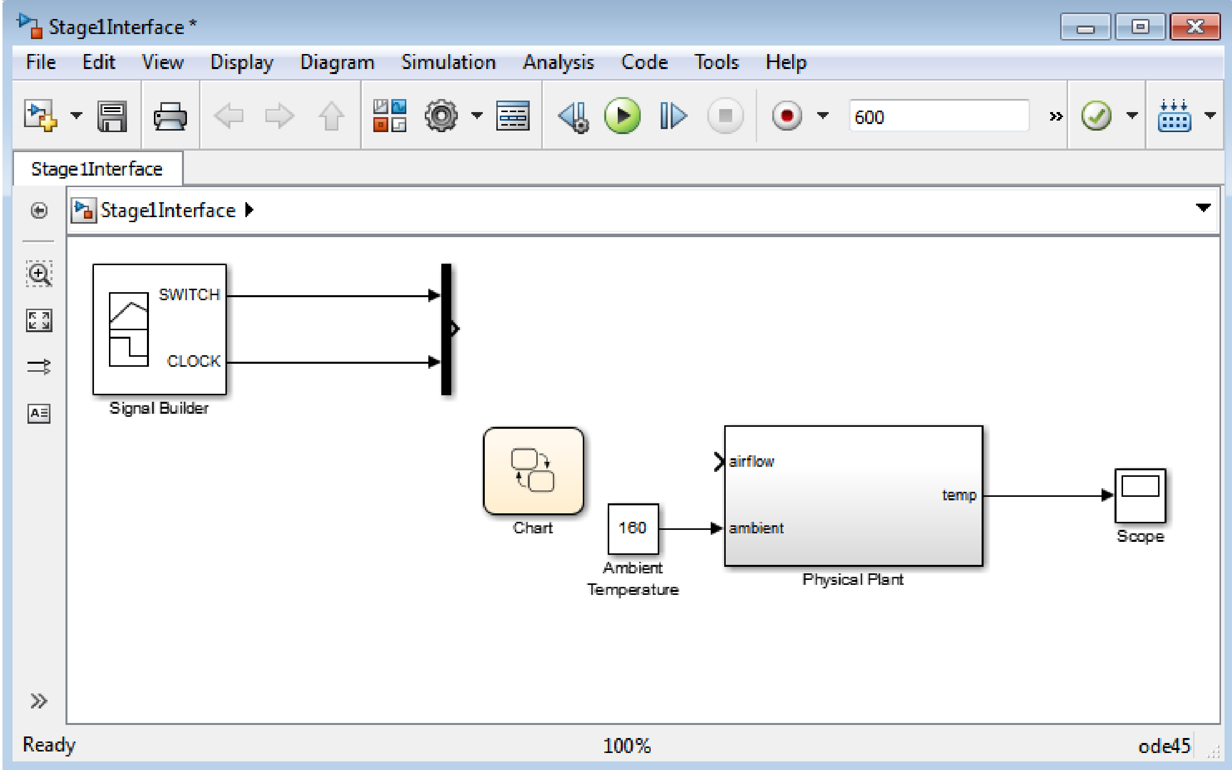


5 Add the Stateflow Chart block to the Simulink model:

a In the left scroll pane of the Library Browser, select **Stateflow**.

b Drag the first block, called **Chart**, into your model.

The model should now look like this:



6 Click the label `Chart` under the Stateflow block and rename it `Air Controller`.

7 Change the action language of the chart to C:

  a Double-click the block to open the chart.

  b Right click in an empty area of the chart and select **Properties**.

  c From the **Action Language** box, select `C`.

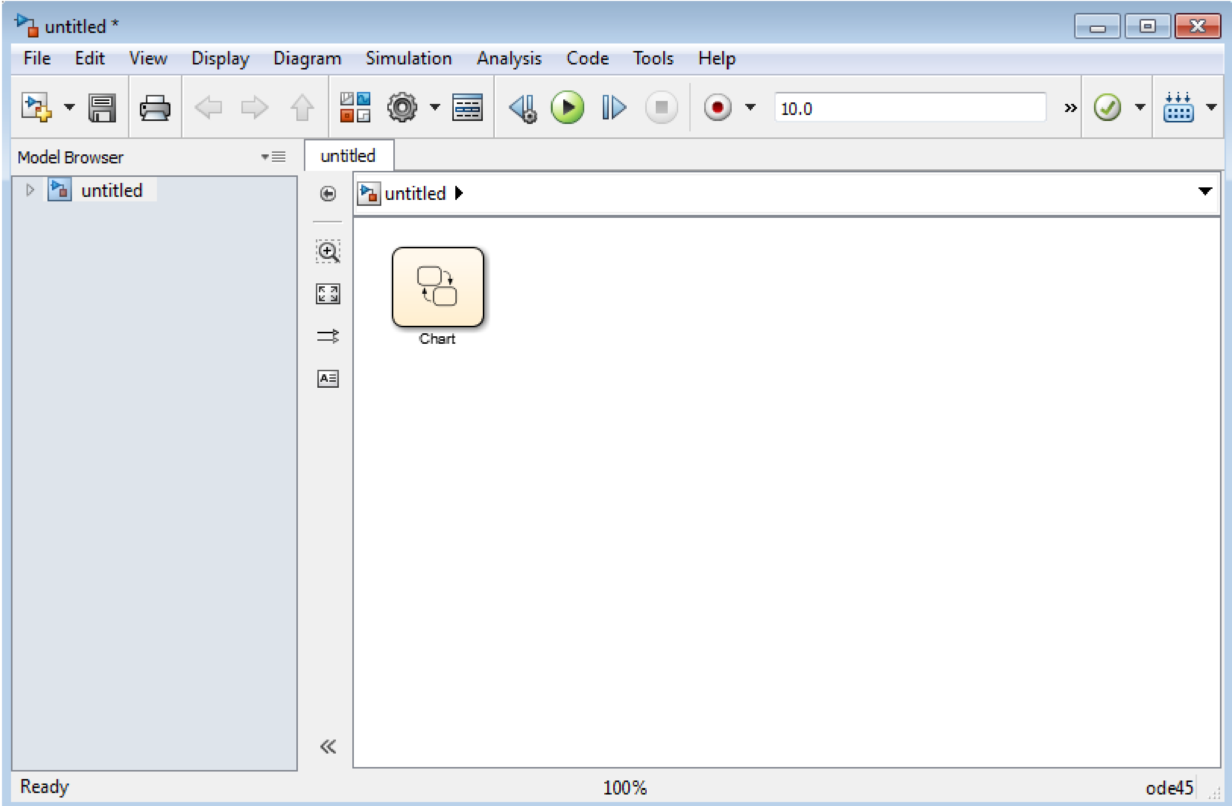
  d Select OK.

Shortcut for adding a Stateflow block to a new Simulink model

At the MATLAB command prompt, enter this command:

```
sfnew
```

A new, untitled Simulink model opens on your desktop, automatically configured with a Stateflow chart. For more information, see `sfnew`.

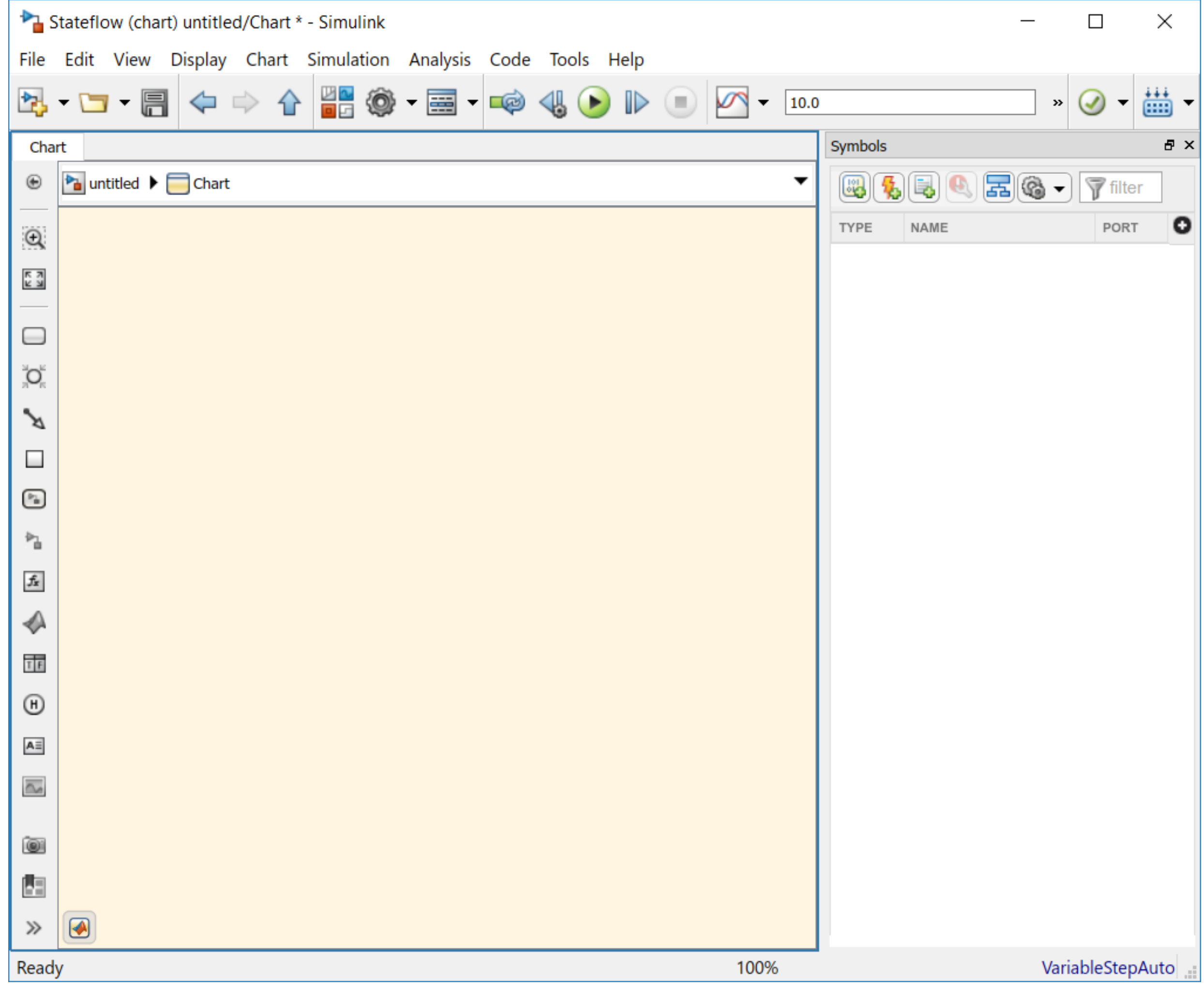


Defining the Inputs and Outputs

Inputs and outputs are data elements in a Stateflow chart that interact with the parent Simulink model. To define inputs and outputs for your chart, follow these steps:

1. Double-click the Air Controller block in the Simulink model `Stage1Interface` to open the Stateflow chart.

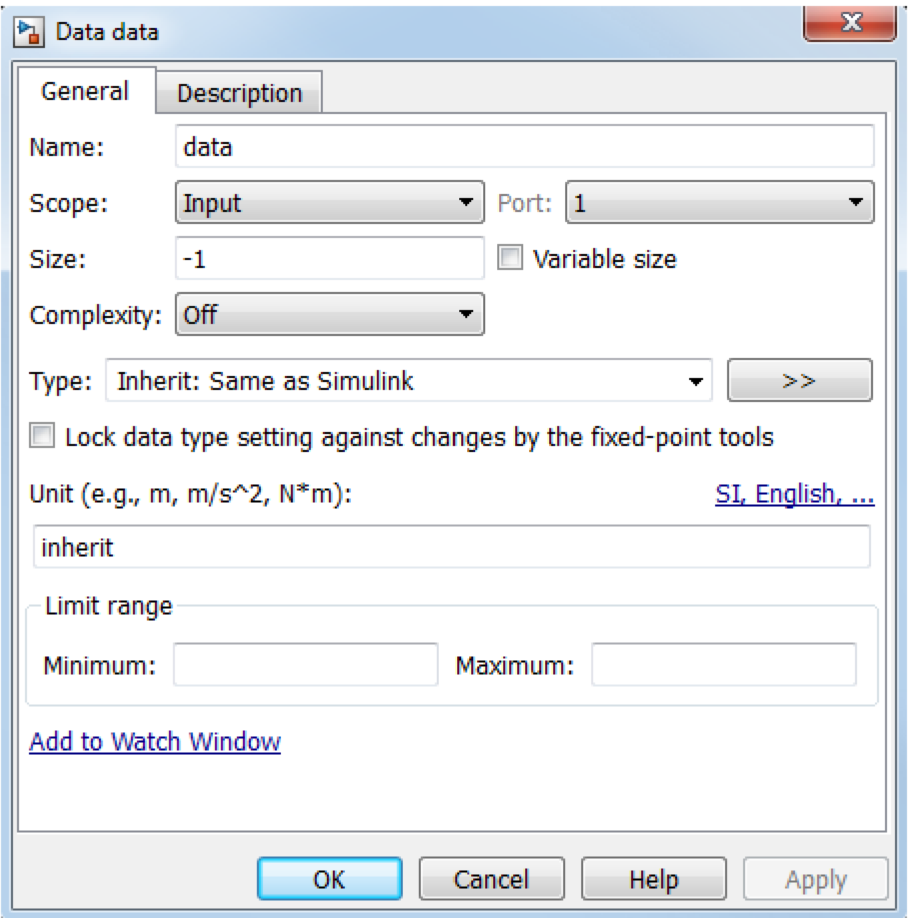
   The Stateflow Editor opens on your desktop:



2 Add a data element to hold the value of the temperature input from the Simulink model:

a In the editor menu, select **Chart > Add Inputs & Outputs > Data Input From Simulink**.

The Data properties dialog box opens on your desktop with the **General** tab selected:



The default values in the dialog box depend on the scope — in this case, a data input.

b In the **Name** field, change the name of the data element to `temp`.

c Leave the following fields at their default values in the **General** tab because they meet the design requirements:

Field	Default Value	What It Means
Scope	**Input**	Input from Simulink model. The data element gets its value from the Simulink signal on the same input port.
Size	`-1`	The data element inherits its size from the Simulink signal on the same port.
Complexity	`Off`	The data element does not contain any complex values.
Type	**Inherit: Same as Simulink**	The data element inherits its data type from the Simulink signal on the same output port.

Note Ports are assigned to inputs and outputs in the order they are created. Because `temp` is the first *input* you created, it is assigned to *input port* 1.

---

d In the **General** tab, select **Add to watch window**.

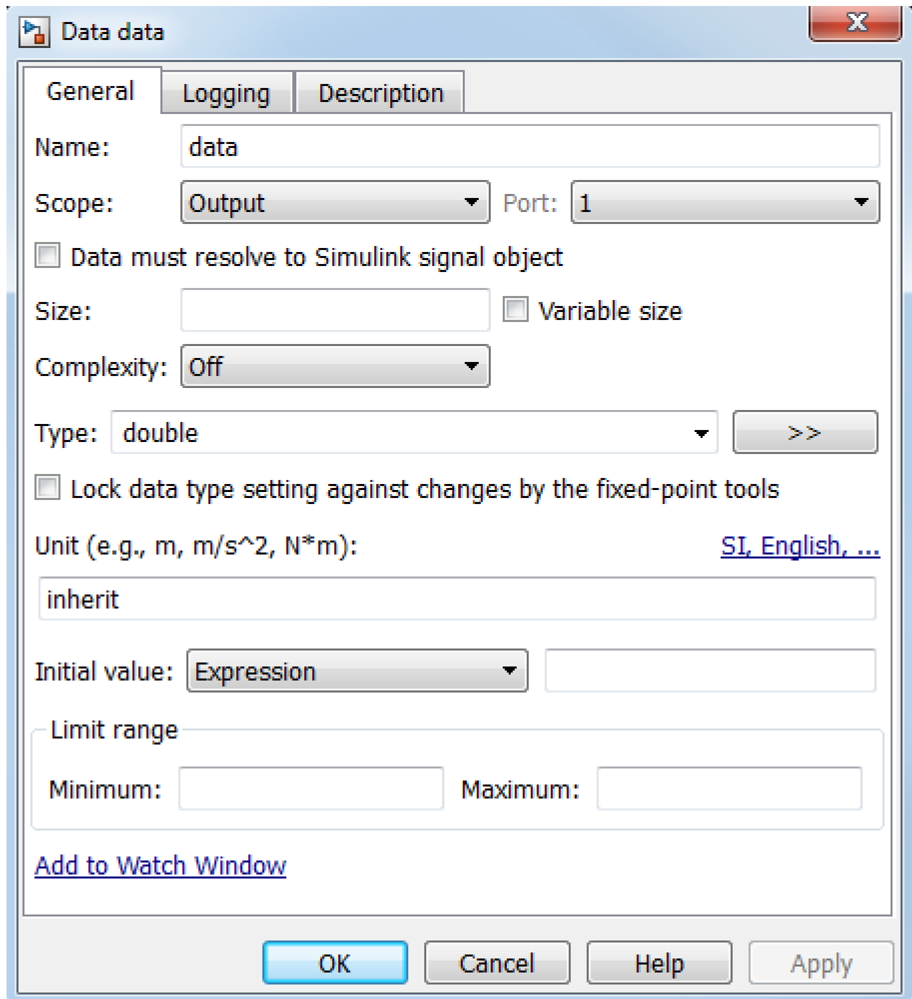
The Stateflow Breakpoints and Watch window lets you examine the value of `temp` during breakpoints in simulation. You will try this in “Setting Simulation Parameters and Breakpoints” on page 7-2.

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

3 Add a data element to hold the value of the airflow output from the Air Controller chart:

a In the editor menu, select **Chart > Add Inputs & Outputs > Data Output To Simulink**.

The Data properties dialog box opens on your desktop, this time with different default values, associated with the scope **Output**:



Note Because `airflow` is the first *output* you created, it is assigned to *output port* 1.

---

b In the **Name** field of the Data properties dialog box, change the name of the data element to `airflow`.

c In the Type field, select `uint8` (8-bit unsigned integer) from the submenu.

d Look at the **Initial value** field.

The initial value is a blank expression, which indicates a default value of zero, based on the data type. This value is consistent with the model design, which specifies that no fans are running when the chart wakes up for the first time.

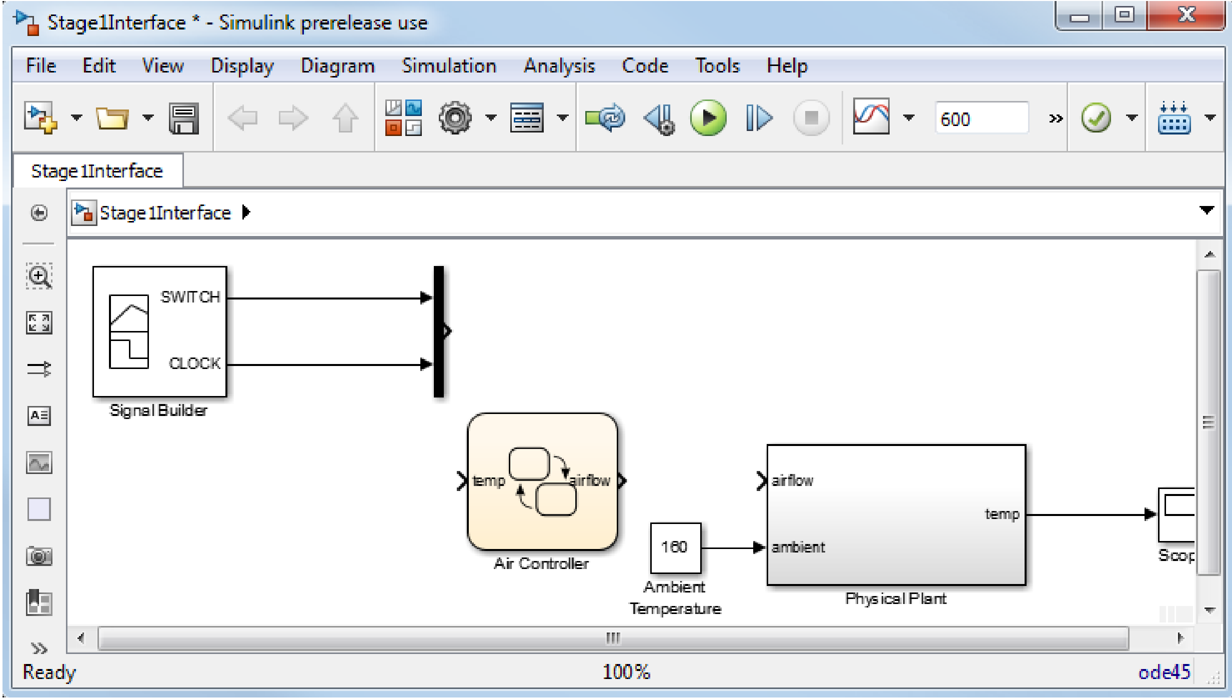
e Make the following changes for other properties in the **General** tab:

Property	What to Specify
Limit range	Enter **0** for **Minimum** and **2** for **Maximum**.
Add to watch window	Select this to add `airflow` to the Watch tab of the Stateflow Breakpoints and Watch window.

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

4 Go back to the Simulink model by clicking the up-arrow button in the Stateflow Editor toolbar.

Notice that the input `temp` and output `airflow` have been added to the Stateflow block:



Tip You might need to enlarge the Air Controller block to see the input and output clearly. To change the size of the block:

a Select the block and move your pointer over one of the corners until it changes to this shape:

b Hold down the left mouse button and drag the block to the desired size.

5 Save `Stage1Interface`.

Tip There are several ways to add data objects to Stateflow charts. You used the Stateflow Editor, which lets you add data elements to the Stateflow chart that is open and has focus. However, to add data objects not just to a chart, but anywhere in the Stateflow design hierarchy, you can use a tool called the Model Explorer. This tool also lets you view

and modify the data objects you have already added to a chart. For more information, see "Hierarchy of Stateflow Objects" and "Add Data Through the Model Explorer" in the Stateflow User's Guide. You can also add data objects programmatically using the Stateflow API, as described in "Create Stateflow Objects" in the Stateflow API Guide.

Connecting the Stateflow Block to the Simulink Subsystem

Now that you have defined the inputs and outputs for the Stateflow Air Controller block, you need to connect them to the corresponding signals of the Simulink Physical Plant subsystem. Follow these steps:

1. In the model `Stage1Interface`, connect the output `airflow` from Air Controller to the corresponding input in Physical Plant:

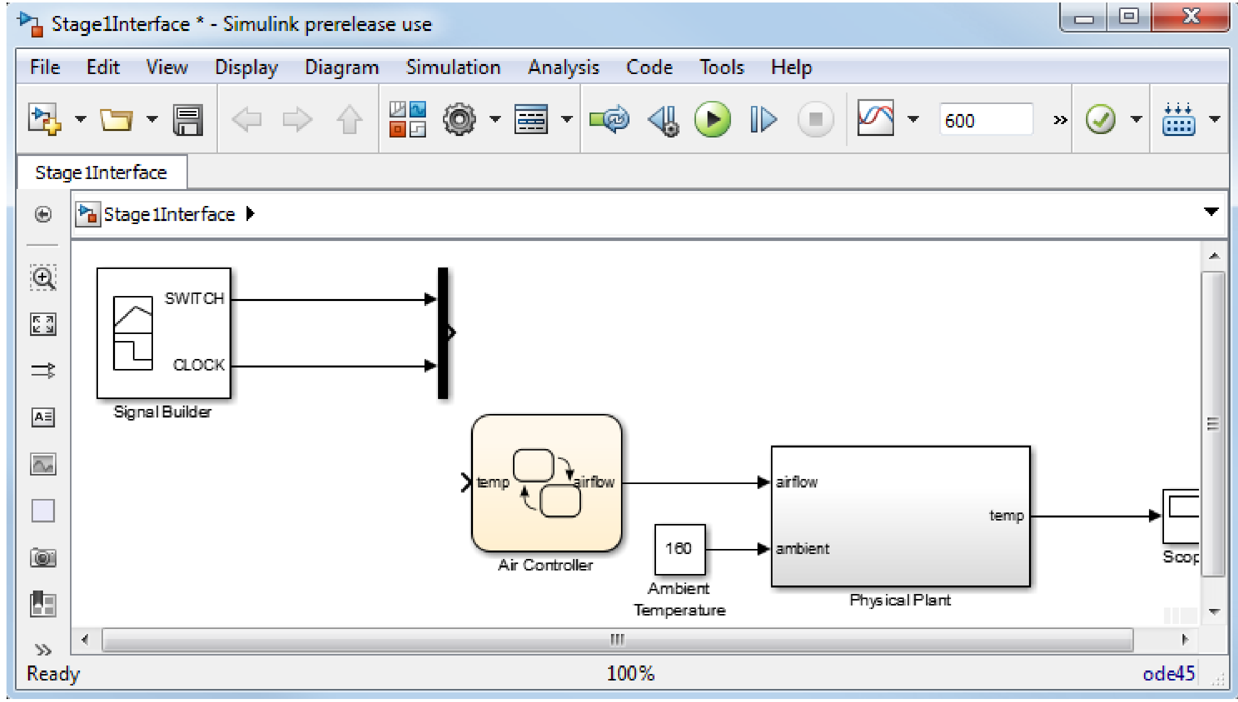
   a. Place your pointer over the output port for `airflow` on the right side of the Air Controller block.

      The pointer changes in shape to crosshairs.

   b. Hold down the left mouse button and move the pointer to the input port for `airflow` on the left side of the Physical Plant block.

   c. Release the mouse.

      The connection should look something like this:



Tip You can use a shortcut for automatically connecting blocks. Select the source block, and then hold down the **Ctrl** key and left-click the destination block.

2 Connect the output `temp` from the Physical Plant to the corresponding input in Air Controller by drawing a branch line from the line that connects `temp` to the Scope:

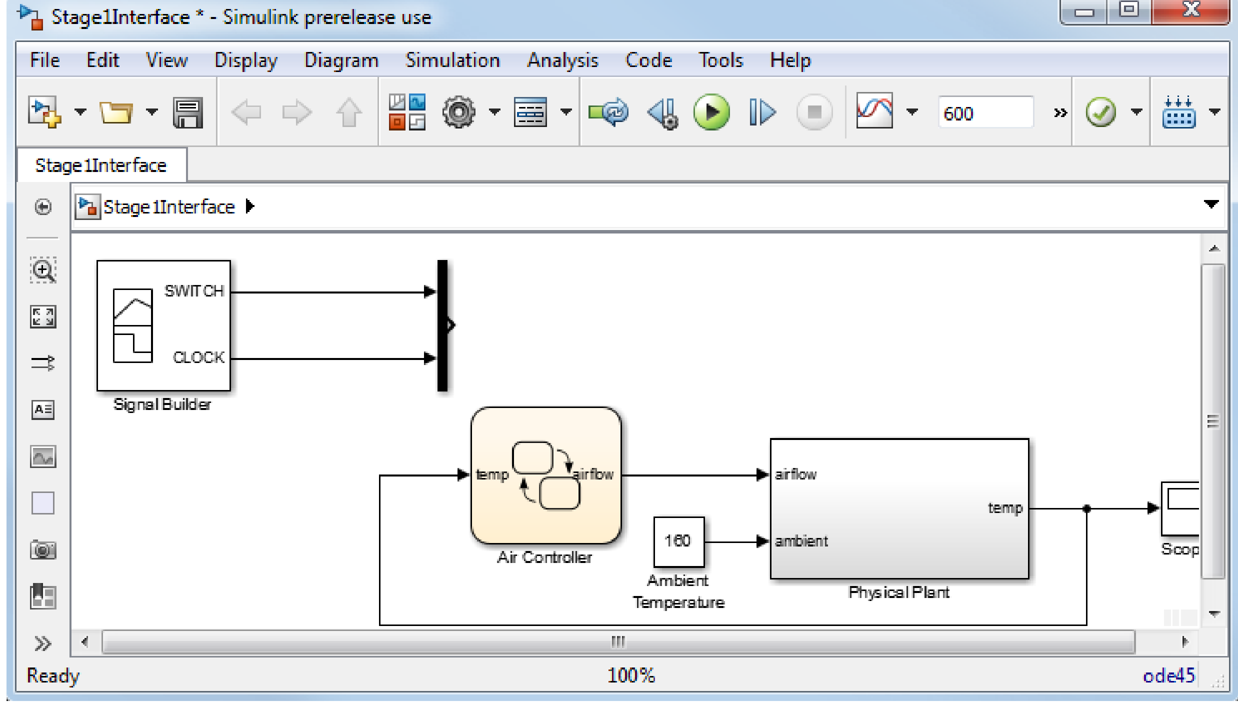
a Place your pointer on the line where you want the branch line to start.

b While holding down the **Ctrl** key, press and hold down the left mouse button.

c Drag your pointer to the input port for `temp` on the left side of the Air Controller block.

d Release the mouse button and the **Ctrl** key.

e Reposition the connection so that it looks like this:

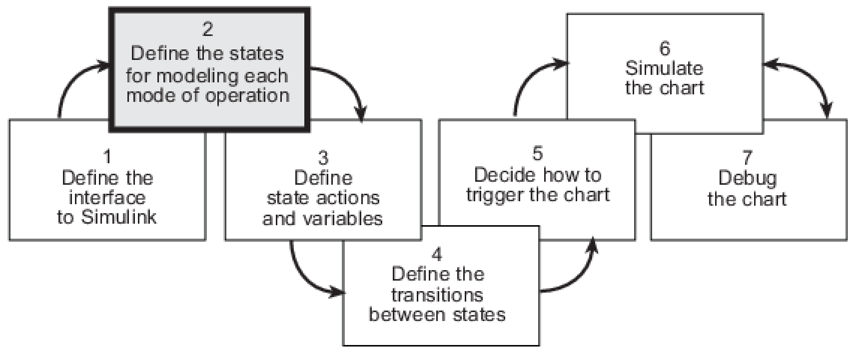


Tip To reposition connections, move your cursor over the end of the line. When the cursor changes to a circle, select the end of the line with the left mouse button and drag the line to a new location.

3 Save `Stage1Interface`.

Where to go next. Now you are ready to model the operating modes with states. See “Implementing the States to Represent Operating Modes” on page 4-2.

Defining the States for Modeling Each Mode of Operation



In phase 2 of this workflow, you *define the states for modeling each mode of operation*.

Implementing the States to Represent Operating Modes

In this section...

"Build It Yourself or Use the Supplied Model" on page 4-2
"Design Considerations for Defining the States" on page 4-2
"Adding the Power On and Power Off States" on page 4-6
"Adding and Configuring Parallel States" on page 4-8
"Adding the On and Off States for the Fans" on page 4-13

Build It Yourself or Use the Supplied Model

To implement the states yourself, work through the exercises in this section. Otherwise, open the supplied model by entering this command at the MATLAB prompt:

```
addpath(fullfile(docroot, 'toolbox', 'stateflow', 'gs', 'examples'))
Stage2States
```

Design Considerations for Defining the States

The following sections describe the rationale for the hierarchy and decomposition of states in the chart.

When to Use States

Whether or not to use states depends on the control logic you want to implement. You can model two types of control logic: *finite state machines* and *stateless flow charts*. Each type is optimized for different applications, as follows:

Control Logic	Optimized for Modeling
Finite state machines	Physical systems that transition between a finite number of operating modes. In Stateflow charts, you represent each mode as a state.
Stateless flow charts	Abstract logic patterns — such as `if`, `if-else`, and `case` statements — and iterative loops — such as `for`, `while`, and `do` loops. You represent these logic constructs with connective junctions and transitions in Stateflow charts. No states are required.

The Air Controller chart is a system that cools a physical plant by transitioning between several modes of operation and, therefore, can be modeled as a finite state machine. In the following sections, you will design the states that model each mode of operation.

Determining the States to Define

States model modes of operation in a physical system. To determine the number and type of states required for your Air Controller chart, you must identify each mode in which the system can operate. Often, a table or grid is helpful for analyzing each mode and determining dependencies between modes.

Analysis of Operating Modes

For Air Controller, the modes of operation are

Operating Mode	Description	Dependencies
Power Off	Turns off all power in the control system	No fan can operate when power is off.
Power On	Turns on all power in the control system	Zero, one, or two fans can operate when power is on.
Fan 1	Activates Fan 1	Fan 1 can be active at the same time as Fan 2. When activated, Fan 1 can turn on or off.
Fan 1 On	Cycles on Fan 1	Fan 1 On can be active if Fan 1 is active and power is on.
Fan 1 Off	Cycles off Fan 1	Fan 1 Off can be active if Fan 1 is active, and power is on.
Fan 2	Activates Fan 2	Fan 2 can be active at the same time as Fan 1. When activated, Fan 2 can turn on or off.
Fan 2 On	Cycles on Fan 2	Fan 2 On can be active if Fan 2 is active and power is on.
Fan 2 Off	Cycles off Fan 2	Fan 2 Off can be active if Fan 2 is active and power is on.

Operating Mode	Description	Dependencies
Calculate airflow	Calculates a constant value of 0, 1, or 2 to indicate how fast air is flowing. Outputs this value to the Simulink subsystem for selecting a cooling factor.	Calculates the constant value, based on how many fans have cycled on at each time step.

Number of States to Define

The number of states depends on the number of operating modes to be represented. In "Analysis of Operating Modes" on page 4-3, you learned that the Air Controller chart has nine operating modes. Therefore, you need to define nine states to model each mode. Here are the names you will assign to the states that represent each operating mode in "Implementing the States to Represent Operating Modes" on page 4-2:

State Name	Operating Mode
`PowerOff`	Power Off
`PowerOn`	Power On
`FAN1`	Fan 1
`FAN2`	Fan 2
`SpeedValue`	Calculate airflow
`FAN1.On`	Fan 1 On
`FAN1.Off`	Fan 1 Off
`FAN2.On`	Fan 2 On
`FAN2.Off`	Fan 2 Off

Note Notice the use of dot notation to refer to the On and Off states for `FAN1` and `FAN2`. You use namespace dot notation to give objects unique identifiers when they have the same name in different parts of the chart hierarchy.

Determining the Hierarchy of States

Stateflow objects can exist in a hierarchy. For example, states can *contain* other states — referred to as *substates* — and, in turn, can *be contained by* other states — referred to as

superstates. You need to determine the hierarchical structure of states you will define for the Air Controller chart. Often, dependencies among states imply a hierarchical relationship — such as parent to child — between the states.

Based on the dependencies described in “Analysis of Operating Modes” on page 4-3, here is an analysis of state hierarchy for the Air Controller chart:

Dependent States	Implied Hierarchy
`FAN1` and `FAN2` depend on `PowerOn`. No fan can operate unless `PowerOn` is active.	`FAN1` and `FAN2` should be substates of a `PowerOn` state.
`FAN1.On` and `FAN1.Off` depend on `Fan1` and `PowerOn`. `FAN1` must be active before it can be cycled on or off.	`FAN1` should have two substates, `On` and `Off`. In this hierarchical relationship, `On` and `Off` will inherit from `FAN1` the dependency on `PowerOn`.
`FAN2.On` and `FAN2.Off` depend on `FAN2` and `PowerOn`. `FAN2` must be active before it can be cycled on or off.	`FAN2` should have two substates, `On` and `Off`. In this hierarchical relationship, `On` and `Off` will inherit from `FAN2` the dependency on `PowerOn`.
The state that calculates airflow needs to know how many fans are running at each time step.	The state that calculates airflow should be a substate of `PowerOn` so it can check the status of `FAN1` and `FAN2` at the same level of hierarchy.

Determining the Decomposition of States

The *decomposition* of a state dictates whether its substates execute exclusively of each other — as *exclusive (OR) states* — or can be activated at the same time — as *parallel (AND) states*. No two exclusive (OR) states can ever be active at the same time, while any number of parallel (AND) states can be activated concurrently.

The Air Controller chart requires both types of states. Here is a breakdown of the exclusive (OR) and parallel (AND) states required for the Stateflow chart:

State	Decomposition	Rationale
`PowerOff`, `PowerOn`	Exclusive (OR) states	The power can never be on and off at the same time.

State	Decomposition	Rationale
FAN1, FAN2	Parallel (AND) states	Zero, one, or two fans can operate at the same time, depending on how much cooling is required.
FAN1.On, FAN1.Off	Exclusive (OR) states	Fan 1 can never be on and off at the same time.
FAN2.On, FAN2.Off	Exclusive (OR) states	Fan 2 can never be on and off at the same time.
SpeedValue	Parallel (AND) state	SpeedValue is an observer state that monitors the status of Fan 1 and Fan 2, updating its output based on how many fans are operating at each time step. SpeedValue must be activated at the same time as Fan 1 and Fan 2, but execute last so it can capture the most current status of the fans.

Adding the Power On and Power Off States

When you add states to the Air Controller chart, you will work from the top down in the Stateflow hierarchy. As you learned in “Determining the Decomposition of States” on page 4-5, the PowerOff and PowerOn states are exclusive (OR) states that turn power off and on in the control system. These states are never active at the same time. By default, states are exclusive (OR) states, represented graphically as rectangles with solid borders.

To add PowerOn and PowerOff to your chart, follow these steps:

1. Open the model Stage1Interface — either the one you created in the previous exercise or the supplied model for stage 1.

   To open the supplied model, enter the following command at the MATLAB prompt:

   ```
   addpath(fullfile(docroot, 'toolbox', 'stateflow', 'gs', 'examples'))
   Stage1Interface
   ```

2. Save the model as Stage2States in your local work folder.
3. In Stage2States, double-click the Air Controller block to open the Stateflow chart.

   The Stateflow Editor for Air Controller opens on your desktop. Notice the object palette on the left side of the editor window. This palette displays a set of tools for drawing graphical chart objects, including states:

4 Left-click the state tool icon:

5 Move your pointer into the drawing area.

The pointer changes to a rectangle, the graphical representation of a state.

6 Click in the upper-left corner of the drawing area to place the state.

The new state appears with a blinking text cursor in its upper-left corner.

7 At the text cursor, type `PowerOn` to name the state.

Tip If you click away from the text cursor before typing the new name, the cursor changes to a question mark. Click the question mark to restore the text cursor.

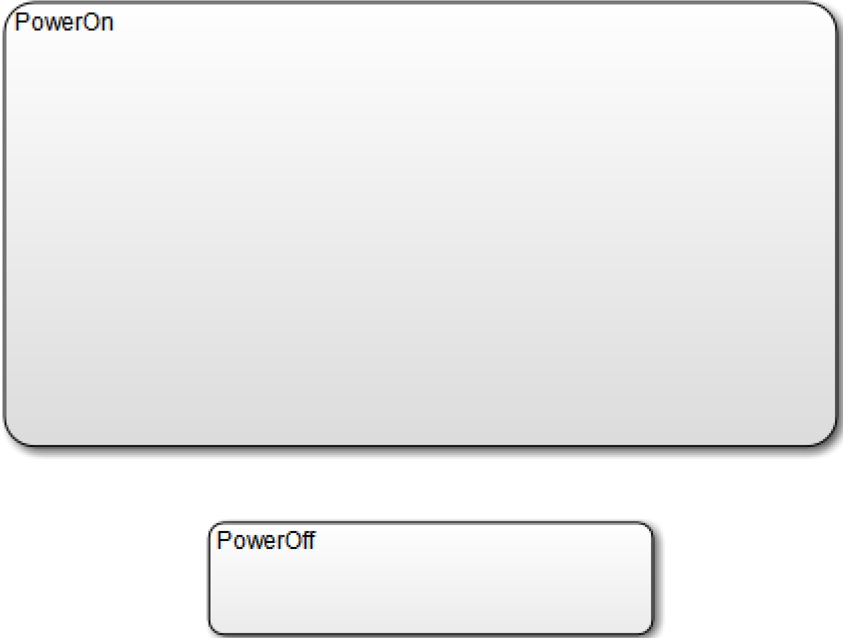
---

8 Move your pointer to the lower-right corner of the rectangle so it changes to this symbol:

9 Drag the lower-right corner to enlarge the state as shown:

PowerOn

10 Click the state tool icon again and draw a smaller state named `PowerOff` at the bottom of the drawing area, like this:



11 Save the chart by selecting **File** > **Save** in the Stateflow Editor, but leave the chart open for the next exercise.

Adding and Configuring Parallel States

In "Determining the States to Define" on page 4-3, you learned that `FAN1`, `FAN2`, and `SpeedValue` will be represented by parallel (AND) substates of the `PowerOn` state. Parallel states appear graphically as rectangles with dashed borders.

In this set of exercises, you will learn how to:

- Assign parallel decomposition to `PowerOn` so its substates can be activated concurrently.

  Recall that the decomposition of a state determines whether its substates will be exclusive or parallel.
- Add parallel substates to a state in the chart.

- Set the order of execution for the parallel substates.

  Even though parallel states can be activated concurrently, they execute in a sequential order.

Setting Parallel Decomposition

Follow these steps:

1. In the Air Controller chart, right-click inside `PowerOn`.

   A submenu opens, presenting tasks you can perform and properties you can set for the selected state.
2. In the submenu, select **Decomposition** > **AND (Parallel)**.
3. Save the model `Stage2States`, but leave the chart open for the next exercise.

Adding the Fan States

Follow these steps:

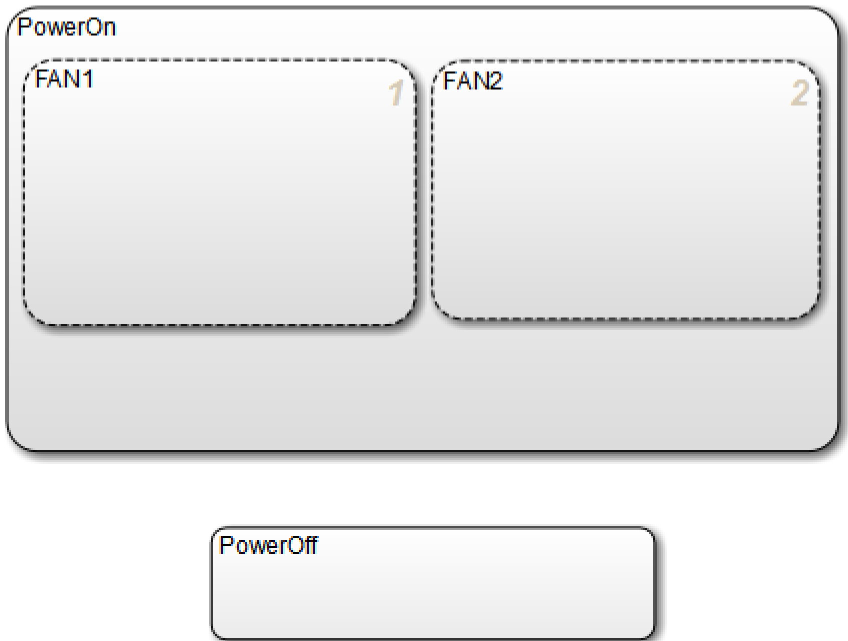
1. Left-click the state tool icon in the Stateflow Editor and place two states inside the `PowerOn` state.

   Tip Instead of using the state tool icon to add multiple states, you can right-click inside an existing state and drag a copy to a new position in the chart. This shortcut is convenient when you need to create states of the same size and shape, such as the fan states.

   ---
2. Notice the appearance of the states you just added.

   The borders of the two states appear as dashed lines, indicating that they are parallel states. Note also that the substates display numbers in their upper-right corners. These numbers specify the order of execution. Although multiple parallel (AND) states in the same chart are activated concurrently, the chart must determine when to execute each one during simulation.
3. Name the new substates `FAN1` and `FAN2`.

   You have created hierarchy in the Air Controller chart. `PowerOn` is now a superstate while `FAN1` and `FAN2` are substates. Your chart should look something like this:



Note Your chart might not show the same execution order for parallel substates FAN1 and FAN2. The reason is that, by default, Stateflow software orders parallel states based on order of creation. If you add FAN2 before FAN1 in your chart, FAN2 moves to the top of the order. You will fine-tune order of activation in a later exercise, "Setting Explicit Ordering of Parallel States" on page 4-11.

Tip If you want to move a state together with its substates — and any other graphical objects it contains — double-click the state. It turns gray, indicating that the state is grouped with the objects inside it and that they can be moved as a unit. To ungroup the objects, double-click the state again.

4. Save the model `Stage2States`, but leave the chart open for the next exercise.

Adding the SpeedValue State

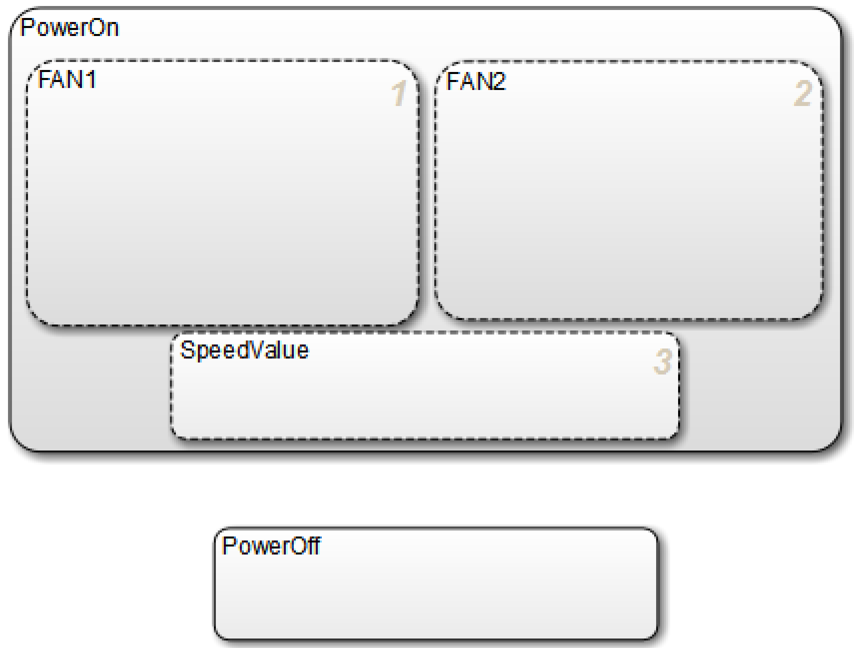
Recall that `SpeedValue` acts as an observer state, which monitors the status of the `FAN1` and `FAN2` states. To add the `SpeedValue` state, follow these steps:

1. Add another substate to `PowerOn` under `FAN1` and `FAN2`, either by using the state tool icon or copying an existing state in the chart.

   You might need to resize the substate to prevent overlap with other substates, but remain within the borders of `PowerOn`.

2. Name the state `SpeedValue`.

   Like `FAN1` and `FAN2`, `SpeedValue` appears as a parallel substate because its parent, the superstate `PowerOn`, has parallel decomposition.

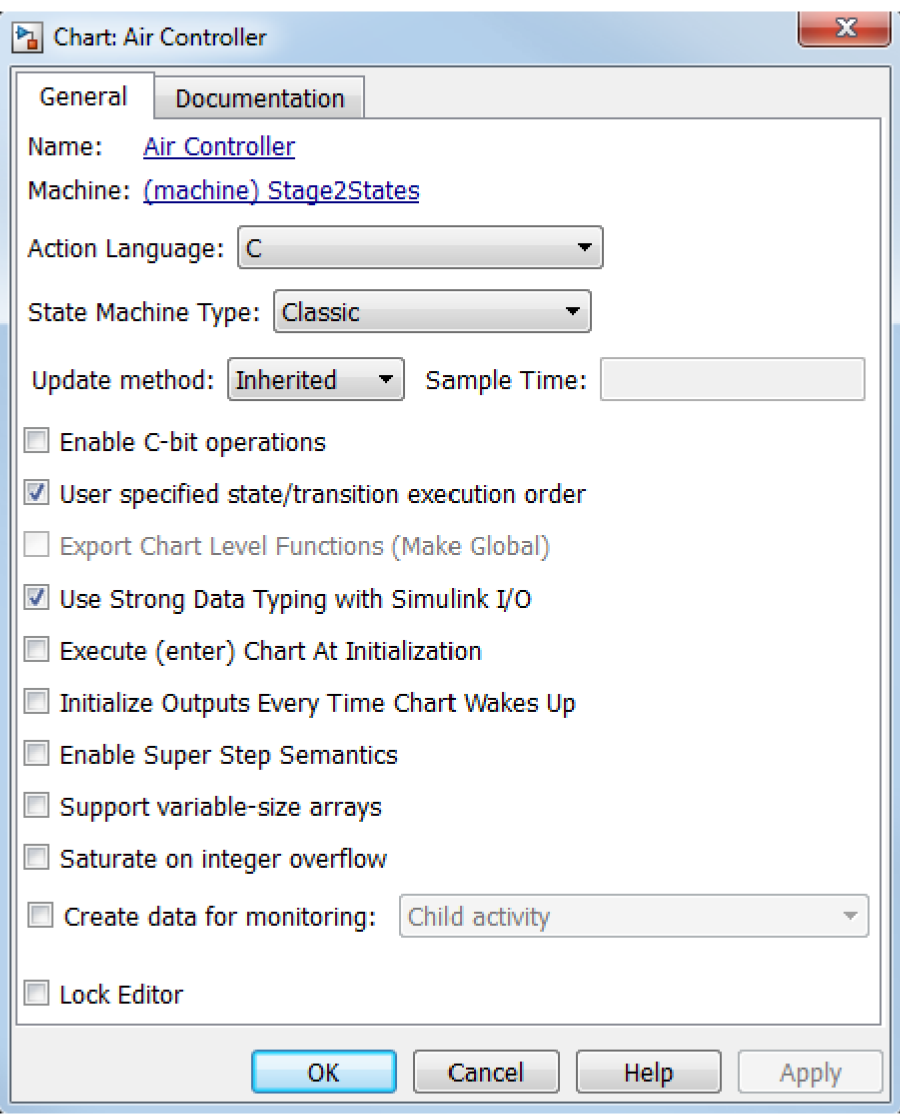


3. Save the model `Stage2States`, but leave the chart open for the next exercise, "Setting Explicit Ordering of Parallel States" on page 4-11.

Setting Explicit Ordering of Parallel States

Recall that, by default, Stateflow software assigns execution order of parallel states based on order of creation in the chart. This behavior is called *explicit ordering*. In this exercise, you will set the execution order explicitly for each parallel state in your chart.

1 In the Stateflow Editor, select **File > Model Properties > Chart Properties**.
2 In the Chart properties dialog box, verify that the check box **User specified state/transition execution order** is selected and click **OK**.



Note This option also lets you explicitly specify the order in which transitions execute when there is a choice of transitions to take from one state to another. This behavior does not apply to the Air Controller chart because it is deterministic: for each exclusive (OR) state, there is one and only one transition to a next exclusive (OR) state. You will learn more about transitions in "Drawing the Transitions Between States" on page 5-4.

---

3. Assign order of execution for each parallel state in the Air Controller chart:

   a. Right-click inside each parallel state to bring up its state properties submenu.

   b. From the submenu, select **Execution Order** and make these assignments:

For State:	Assign:
FAN1	1
FAN2	2
SpeedValue	3

   Here is the rationale for this order of execution:

   - FAN1 should execute first because it cycles on at a lower temperature than FAN2.
   - SpeedValue should execute last so it can observe the most current status of FAN1 and FAN2.

4. Save the model Stage2States, but leave the chart open for the next exercise, "Adding the On and Off States for the Fans" on page 4-13.

Adding the On and Off States for the Fans

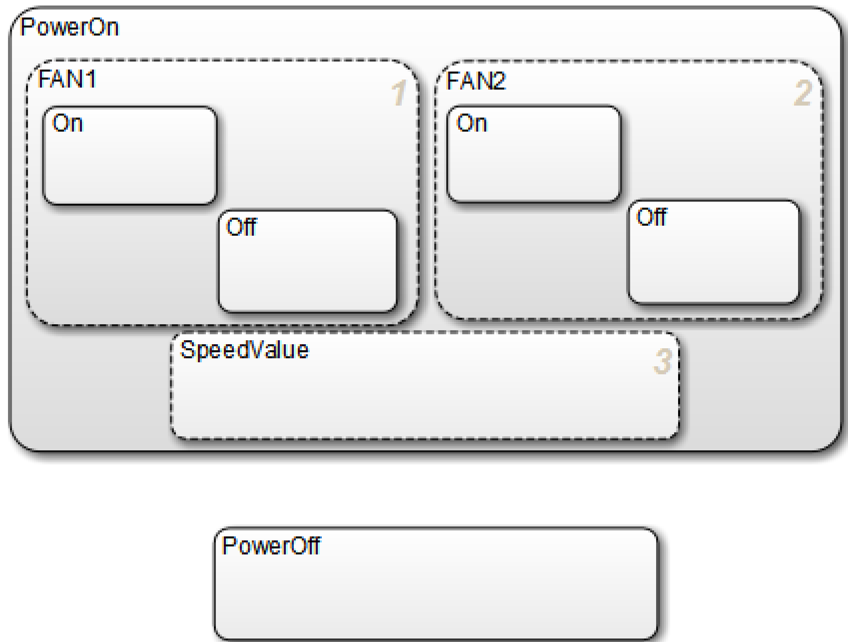
In this exercise, you will enter the on and off substates for each fan. Because fans cannot cycle on and off at the same time, these states must be exclusive, not parallel. Even though FAN1 and FAN2 are parallel states, their *decomposition* is exclusive (OR) by default. As a result, any substate that you add to FAN1 or FAN2 will be an exclusive (OR) state.

Follow these steps:

1. Add two substates inside FAN1 and FAN2.
2. Resize the substates to fit within the borders of FAN1 and FAN2.

3 In each fan state, name one substate `On` and name the other `Off`.

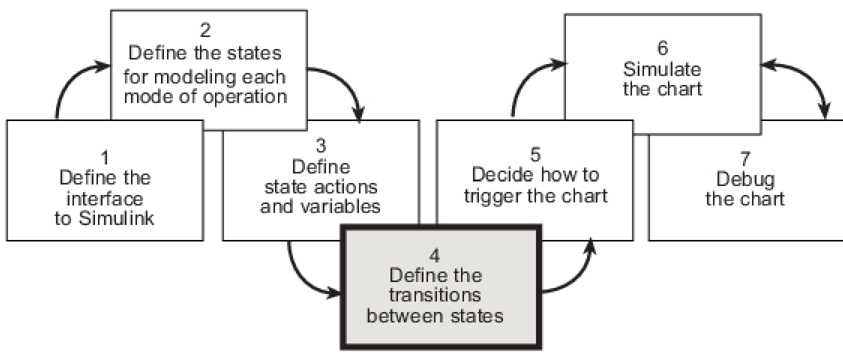
Your Air Controller chart should now look something like this:



4 Save the model `Stage2States`.

Where to go next. Now you are ready to specify the actions that execute when a state is active.

Defining Transitions Between States



In phase 4 of this workflow, you *define the transitions between states*.

Adding the Transitions

In this section...
“Build It Yourself or Use the Supplied Model” on page 5-2
“Design Considerations for Defining Transitions Between States” on page 5-2
“Drawing the Transitions Between States” on page 5-4
“Adding Default Transitions” on page 5-7
“Adding Conditions to Guard Transitions” on page 5-10
“Adding Events to Guard Transitions” on page 5-11

Build It Yourself or Use the Supplied Model

To add the transitions yourself, work through the exercises in this section. Otherwise, open the supplied model to see how the transitions should appear in the chart. Enter this command at the MATLAB prompt:

```
addpath(fullfile(docroot, 'toolbox', 'stateflow', 'gs', 'examples'))
Stage4Transitions
```

Design Considerations for Defining Transitions Between States

The following sections describe the decisions you make for defining state transitions.

Deciding How and When to Transition Between Operating Modes

Transitions create paths for the logic flow of a system from one state to another. When a transition is taken from state A to state B, state A becomes inactive and state B becomes active.

Transitions have direction and are represented in a Stateflow chart by lines with arrowheads. Transitions are unidirectional, not bidirectional. You must add a transition for each direction of flow between two states.

Exclusive (OR) states require transitions. Recall that no two exclusive states can be active at the same time. Therefore, you need to add transitions to specify when and where control flows from one exclusive state to another.

Typically, parallel (AND) states do not require transitions because they execute concurrently.

The Air Controller chart models a system in which power can cycle on and off and, while power is on, fans can cycle on and off. Six exclusive (OR) states represent these operating modes. To model this activity, you need to add the following transitions between exclusive (OR) states:

- `PowerOff` to `PowerOn`
- `PowerOn` to `PowerOff`
- `FAN1.Off` to `FAN1.On`
- `FAN1.On` to `FAN1.Off`
- `FAN2.Off` to `FAN2.On`
- `FAN2.On` to `FAN2.Off`

Deciding Where to Place Default Transitions

Good design practice requires that you specify default transitions for exclusive (OR) states at each level of hierarchy. *Default transitions* indicate 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 Stateflow hierarchy. There are three such areas of ambiguity in the Air Controller chart:

- When the chart wakes up, should power be on or off?
- When `FAN1` becomes active, should it be on or off?
- When `FAN2` becomes active, should it be on or off?

In each case, the initial state should be off so you will add default transitions to the states `PowerOff`, `FAN1.Off`, and `FAN2.Off`.

Deciding How to Guard the Transitions

Guarding a transition means specifying a condition, action, or event that allows the transition to be taken from one state to another. Based on the design of the Air Controller chart, here are the requirements for guarding the transitions from one exclusive operating mode to another:

Transition	When Should It Occur?	How to Guard It
PowerOff to PowerOn	At regular time intervals	Specify an edge-triggered event
PowerOn to PowerOff		
FAN1.Off to FAN1.On	When the temperature of the physical plant rises above 120 degrees	Specify a condition based on temperature value
FAN1.On to FAN1.Off	When the temperature of the physical plant falls below 120 degrees	
FAN2.Off to FAN2.On	When the temperature rises above 150 degrees, a threshold indicating that first fan is not providing the required amount of cooling	
FAN2.On to FAN2.Off	When the temperature falls below 150 degrees	

Drawing the Transitions Between States

In “Design Considerations for Defining Transitions Between States” on page 5-2, you learned that the following transitions occur in the Air Controller chart:

- Power for the control system can cycle on and off.
- Each fan can cycle on and off.

You will model this activity by drawing transitions between the `PowerOn` and `PowerOff` states and between the `On` and `Off` states for each fan. Follow these steps:

1. Open the model `Stage3Actions` — either the one you created in the previous exercises or the supplied model for stage 3.

   To open the supplied model, enter the following command at the MATLAB prompt:

   ```
   addpath(fullfile(docroot, 'toolbox', 'stateflow', 'gs', 'examples'))
   Stage3Actions
   ```

2. Save the model as `Stage4Transitions` in your local work folder.
3. In `Stage4Transitions`, double-click the Air Controller block to open the Stateflow chart.

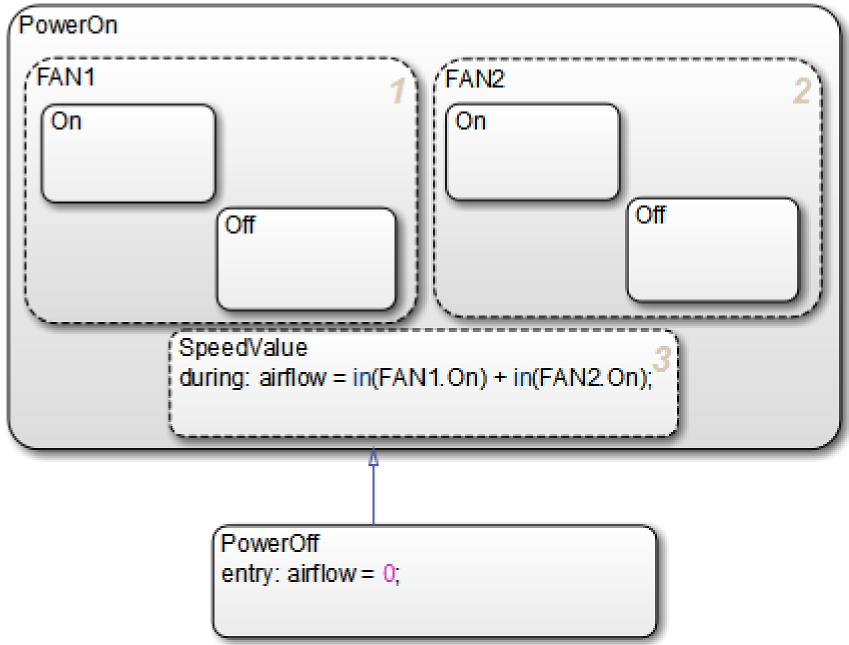
The chart opens on your desktop.

4 Draw transitions between the `PowerOff` to `PowerOn` states:

a Move your pointer over the top edge of `PowerOff` until the pointer shape changes to crosshairs.

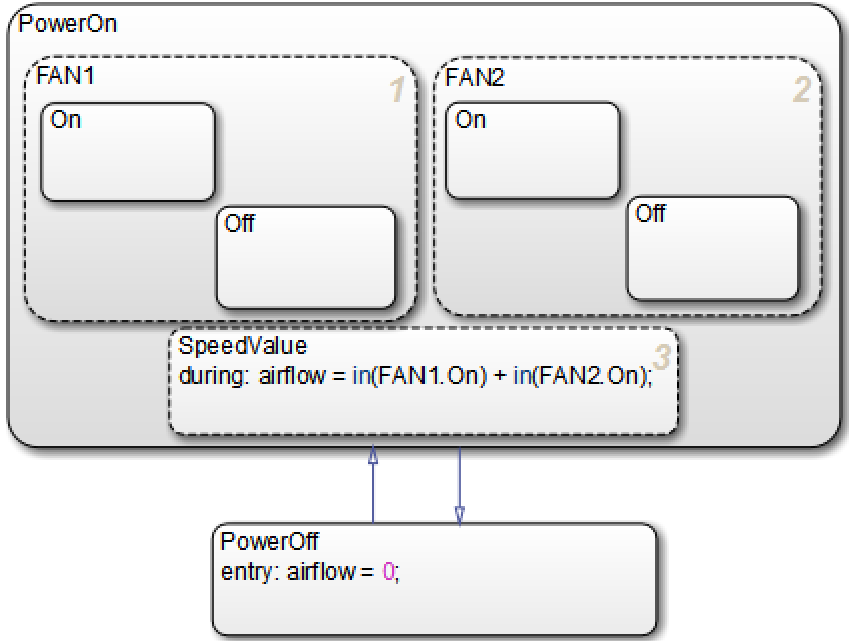
b Hold down the left mouse button, drag your pointer to the bottom edge of `PowerOn`, and release the mouse.

You should see a transition pointing from `PowerOff` to `PowerOn`:



c Follow the same procedure to draw a transition from `PowerOn` to `PowerOff`.

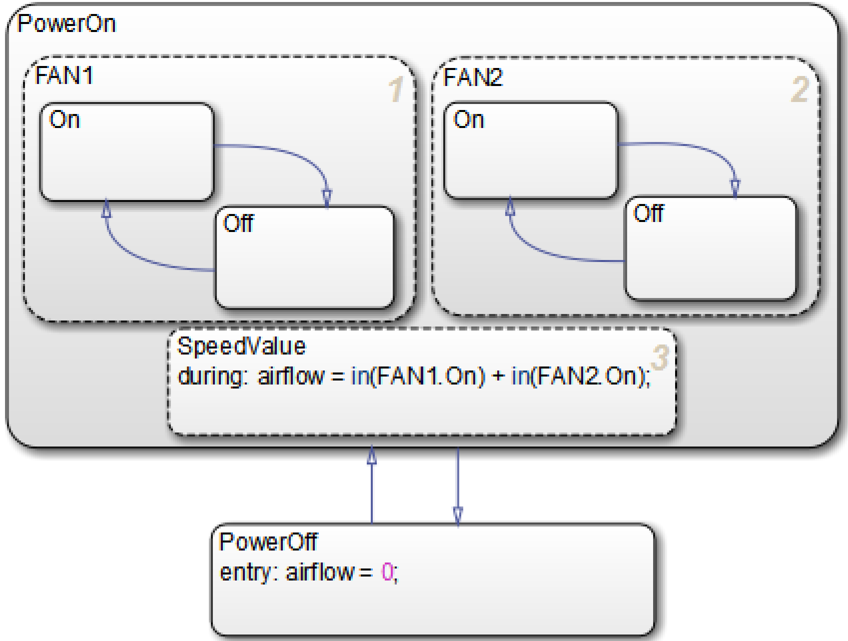
Your chart should now look like this:



5 Follow the procedure described in step 3 to draw the following transitions between the `Off` and `On` states for each fan:

- Transition from `Off` to `On` in `FAN1`
- Transition from `On` to `Off` in `FAN1`
- Transition from `Off` to `On` in `FAN2`
- Transition from `On` to `Off` in `FAN2`

Your chart should now look like this:



6 Save `Stage4Transitions`, but leave the chart open for the next exercise.

Adding Default Transitions

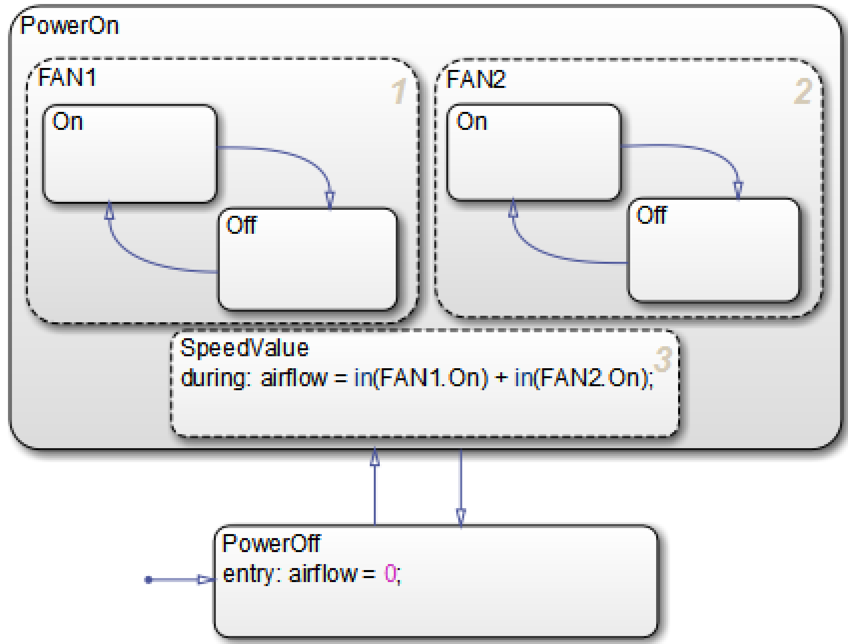
In “Deciding Where to Place Default Transitions” on page 5-3, you learned that you need to add default transitions to `PowerOff`, `FAN1.Off`, and `FAN2.Off`. Follow these steps:

1. In the Stateflow Editor, left-click the default transition icon in the object palette:

2. Move your pointer into the drawing area.

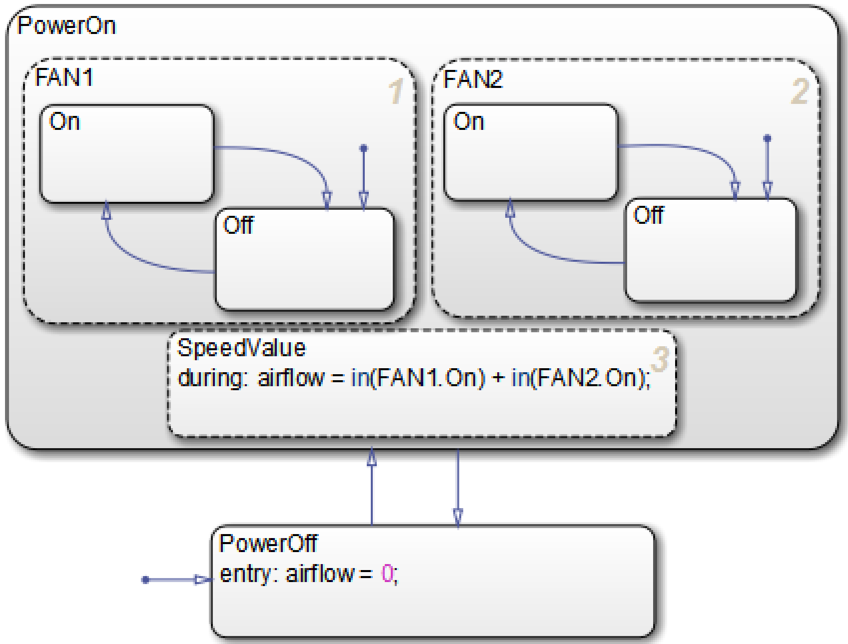
   The pointer changes to a diagonal arrow.
3. Place your pointer at the left edge of the `PowerOff` state.
4. When the arrow becomes orthogonal to the edge, release the mouse button.

   The default transition attaches to the `PowerOff` state. It appears as a directed line with an arrow at its head and a closed tail:

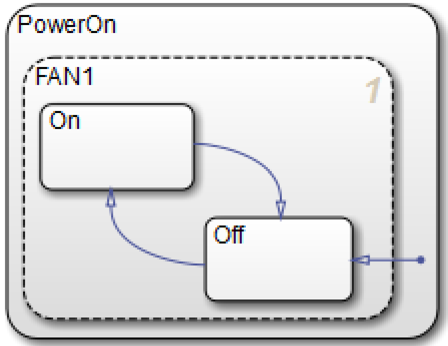


5 Repeat the same procedure to add default transitions at the top edges of `FAN1.Off` and `FAN2.Off`.

Your chart should now look like this:



Tip The location of the tail of a default transition determines the state it activates. Therefore, make sure that your default transition fits completely inside the parent of the state that it activates. In the Air Controller chart pictured above, notice that the default transition for `FAN1.Off` correctly resides inside the parent state, `FAN1`. Now consider this chart:



In this example, the tail of the default transition resides in `PowerOn`, not in `FAN1`. Therefore, it will activate `FAN1` instead of `FAN1.Off`.

6 Save `Stage4Transitions`, but leave the chart open for the next exercise.

Adding Conditions to Guard Transitions

Conditions are expressions enclosed in square brackets that evaluate to true or false. When the condition is true, the transition is taken to the destination state; when the condition is false, the transition is not taken and the state of origin remains active.

As you learned in "Deciding How to Guard the Transitions" on page 5-3, the fans cycle on and off depending on the air temperature. In this exercise, you will add conditions to the transitions in `FAN1` and `FAN2` that model this behavior.

Follow these steps:

1 Click the transition from `FAN1.Off` to `FAN1.On`.

The transition appears highlighted and displays a question mark (`?`).

2 Click next to the question mark to display a blinking text cursor.

3 Type the following expression:

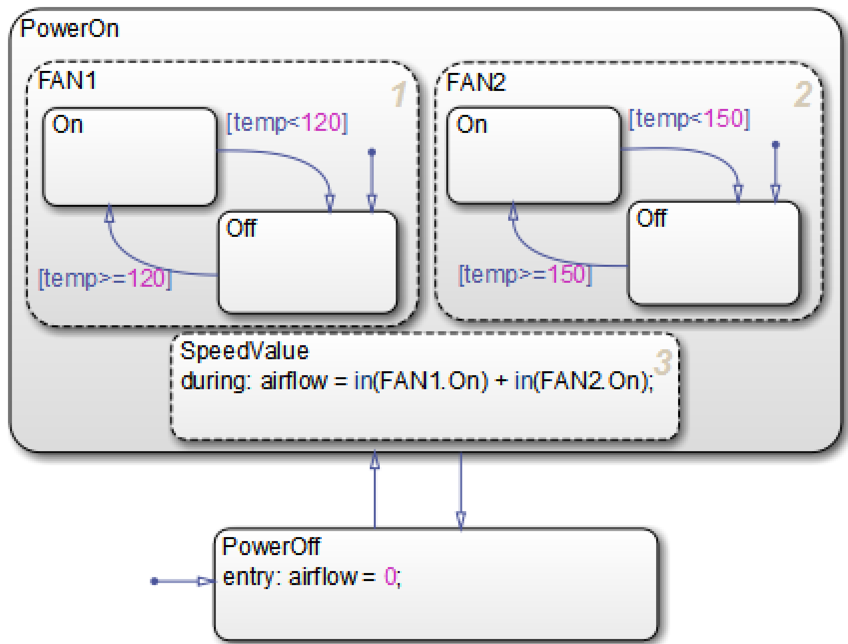
```
[temp >= 120]
```

You may need to reposition the condition for readability. Click outside the condition, then left-click and drag the condition expression to a new location.

4 Repeat these steps to add the following conditions to the other transitions in `FAN1` and `FAN2`:

Transition	Condition
`FAN1.On` to `FAN1.Off`	[temp < 120]
`FAN2.Off` to `FAN2.On`	[temp >= 150]
`FAN2.On` to `FAN2.Off`	[temp < 150]

Your chart should look like this:



5 Save `Stage4Transitions`, but leave the chart open for the next exercise.

Adding Events to Guard Transitions

Events are nongraphical objects that trigger activities during the execution of a Stateflow chart. Depending on where and how you define events, they can trigger a transition to

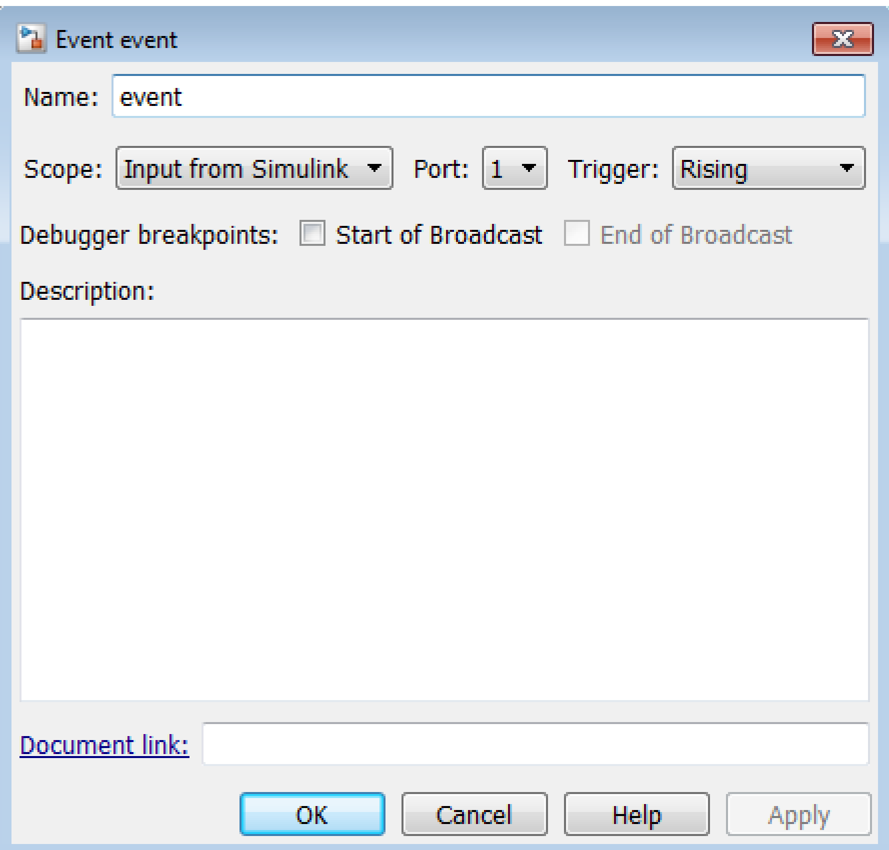
occur, an action to be executed, and state status to be evaluated. In this exercise, you will define an event that triggers transitions.

As you learned in "Deciding How to Guard the Transitions" on page 5-3, the control system should power on and off at regular intervals. You model this behavior by first defining an event that occurs at the rising or falling edge of an input signal, and then associating that event with the transitions between the `PowerOn` and `PowerOff` states.

Follow these steps to define an edge-triggered event and associate it with the transitions:

1. In the Stateflow Editor, add an input event by selecting **Chart** > **Add Inputs & Outputs** > **Event Input From Simulink**.

   The Event properties dialog box opens on your desktop:



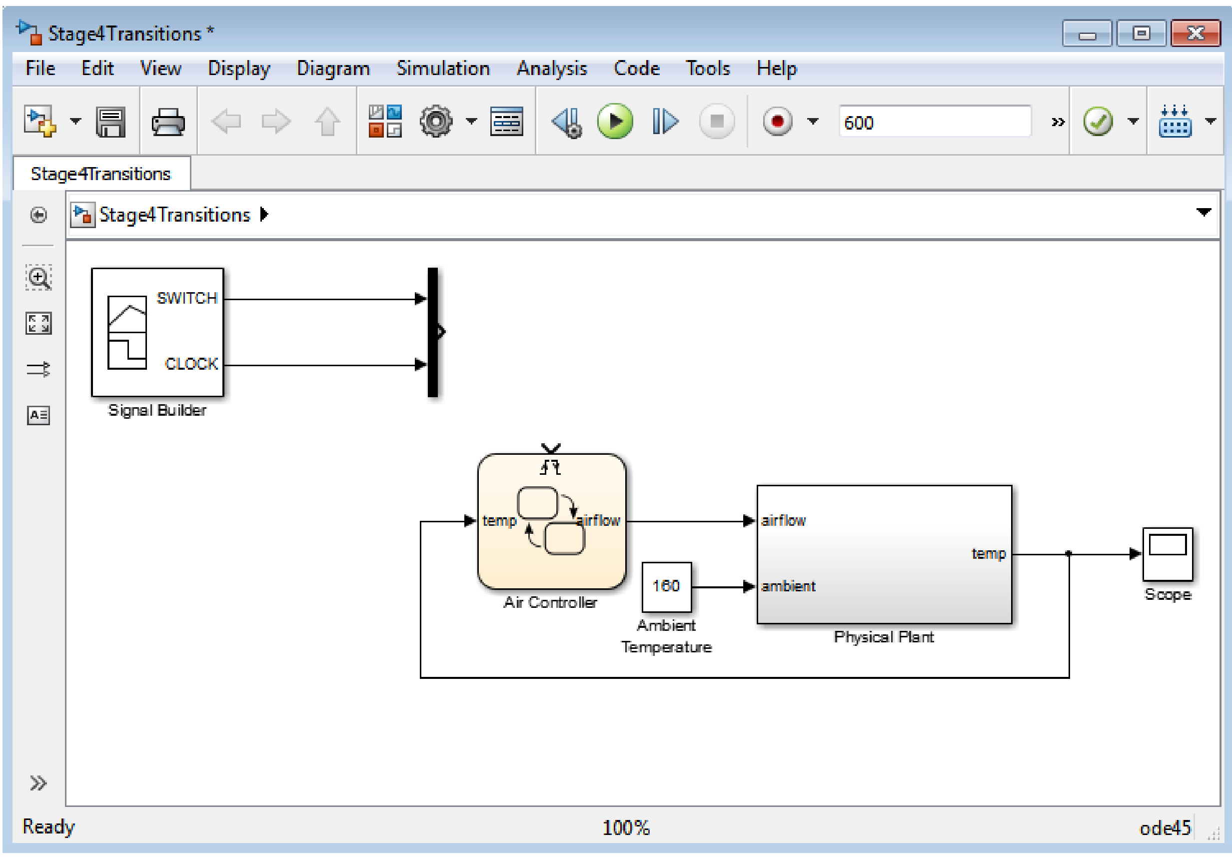
Note that the event is assigned to trigger port 1.

2 Edit the following properties:

Property	What to Specify
Name	Change the name to `SWITCH`.
Trigger	Select `Either` from the drop-down menu so the event can be triggered by either the rising edge or falling edge of a signal.

3 Click **OK** to record the changes and close the dialog box.

4 Look back at the model and notice that a trigger port appears at the top of the Stateflow block:



When you define one or more input events for a chart, Stateflow software adds a single trigger port to the block. External Simulink blocks can trigger the input events via a signal or vector of signals connected to the trigger port.

5 Back in the Stateflow Editor, associate the input event `SWITCH` with the transitions:

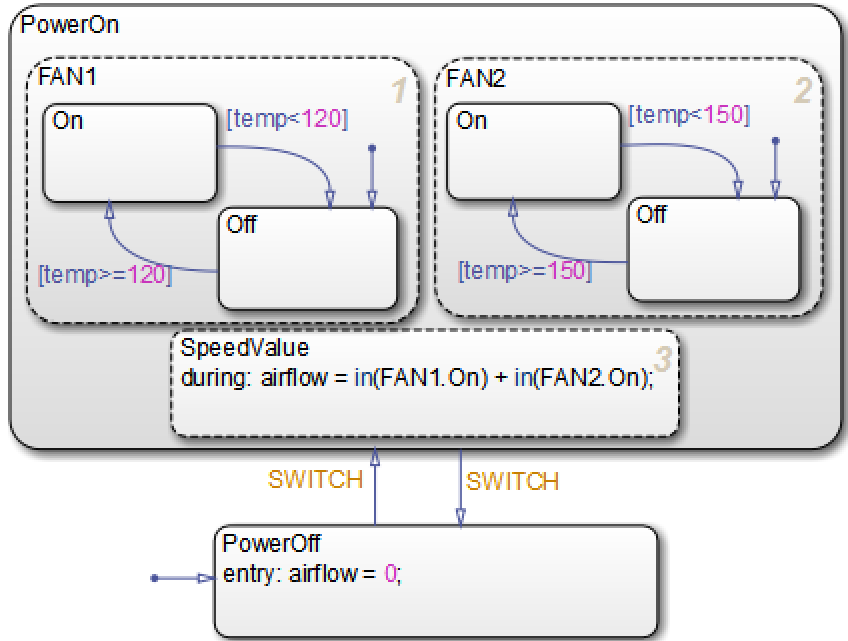
a Select the transition from `PowerOff` to `PowerOn` and click the question mark to get a text cursor.

b Type the name of the event you just defined, `SWITCH`.

You might need to reposition the event text for readability. If so, click outside the text, left-click the text, and drag it to the desired location.

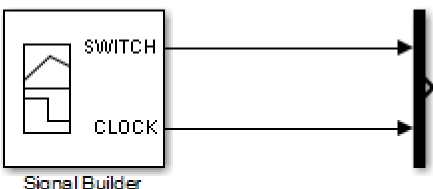
c Repeat these steps to add the same event, `SWITCH`, to the transition from `PowerOn` to `PowerOff`.

Your chart should now look something like this:



Now that you have associated these transitions with the event `SWITCH`, the control system will alternately power on and off every time `SWITCH` occurs — that is, every time the chart detects a rising or falling signal edge.

Note that the `sf_aircontrol` model has already defined the pulse signal `SWITCH` in the Signal Builder block at the top level of the model hierarchy:

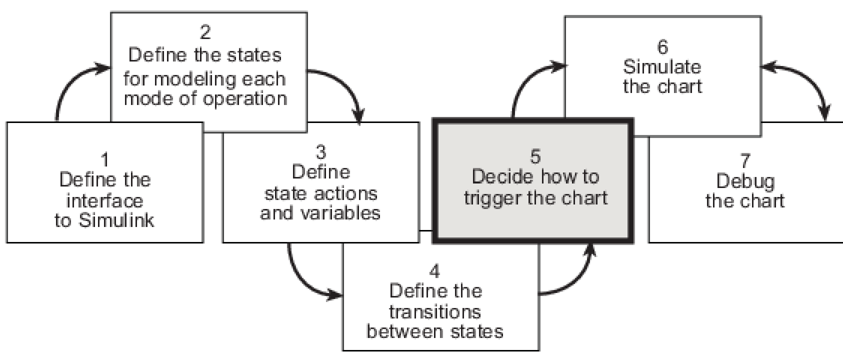


In the next phase of the workflow, you will connect your Stateflow chart to the `SWITCH` signal to trigger the transitions between power on and power off.

6 Save `Stage4Transitions`.

Where to go next. Now you are ready to implement an edge-triggered event to wake up the chart at regular intervals. See “Implementing the Triggers” on page 6-2.

Triggering a Stateflow Chart



In phase 5 of this workflow, you *decide how to trigger the chart*.

Implementing the Triggers

In this section...

"Build It Yourself or Use the Supplied Model" on page 6-2
"Design Considerations for Triggering Stateflow Charts" on page 6-2
"Defining the CLOCK Event" on page 6-3
"Connecting the Edge-Triggered Events to the Input Signals" on page 6-4

Build It Yourself or Use the Supplied Model

To implement the triggers yourself, work through the exercises in this section. Otherwise, open the supplied model to see how the triggers should appear in the chart. Enter this command at the MATLAB prompt:

```
addpath(fullfile(docroot, 'toolbox', 'stateflow', 'gs', 'examples'))
Stage5Trigger
```

Design Considerations for Triggering Stateflow Charts

A Simulink model can wake up a Stateflow chart by

- Sampling the chart at a specified or inherited rate
- Using a signal as a trigger
- Using one Stateflow chart to drive the activity of another

A signal trigger works best for the Air Controller chart because it needs to monitor the temperature of the physical plant at regular intervals. To meet this requirement, you will use a periodic signal to trigger the chart. The source is a square wave signal called `CLOCK`, provided by a Signal Builder block in the Simulink model, described in "How the Stateflow Chart Works with the Simulink Model" on page 2-6. To harness the signal, you will set up an edge trigger event that wakes the chart at the rising or falling edge of `CLOCK`.

The rationale for using an edge trigger in this case is that it uses the regularity and frequency of the signal to wake up the chart. When using edge triggers, keep in mind that there can be a delay from the time the trigger occurs to the time the chart begins executing. This is because an edge trigger causes the chart to execute at the beginning of the next simulation time step, regardless of when the edge trigger actually occurred

during the previous time step. The Air Controller can tolerate this delay, as long as the edge occurs frequently enough. (For more information about triggering Stateflow charts, see “Implement Interfaces to Simulink Models” in the Stateflow User's Guide.)

Recall that you already defined one edge-triggered event, `SWITCH`, to guard the transitions between `PowerOff` and `PowerOn`. You will now define a second edge-triggered event, `CLOCK`, to wake up the chart.

Defining the CLOCK Event

To define the `CLOCK` event, follow these steps:

1. Open the model `Stage4Transitions` — either the one you created in the previous exercises or the supplied model for stage 4.

   To open the supplied model, enter the following command at the MATLAB prompt:

   ```
   addpath(fullfile(docroot, 'toolbox', 'stateflow', 'gs', 'examples'))
   Stage4Transitions
   ```

2. Save the model as `Stage5Trigger` in your local work folder.
3. In `Stage5Trigger`, double-click the Air Controller block to open the Stateflow chart.
4. In the Stateflow Editor, add an input event by selecting **Chart** > **Add Inputs & Outputs** > **Event Input From Simulink**.
5. In the Event properties dialog box, edit the following fields:

Property	What to Specify
Name	Change the name to `CLOCK`.
Trigger	Select `Either` from the drop-down menu so that the rising or falling edge of a signal can trigger the event.

   Because the `SWITCH` event you created in “Adding Events to Guard Transitions” on page 5-11 was assigned to trigger port 1, the `CLOCK` event is assigned to trigger port 2. Nevertheless, only one trigger port appears at the top of the Air Controller block to receive trigger signals. This means that each signal must be indexed into an array, as described in “Connecting the Edge-Triggered Events to the Input Signals” on page 6-4.

6. Click **OK** to record the changes and close the dialog box.
7. Save `Stage5Trigger`, but leave it open for the next exercise.

Connecting the Edge-Triggered Events to the Input Signals

You need to connect the edge-triggered events to the Simulink input signals in a way that

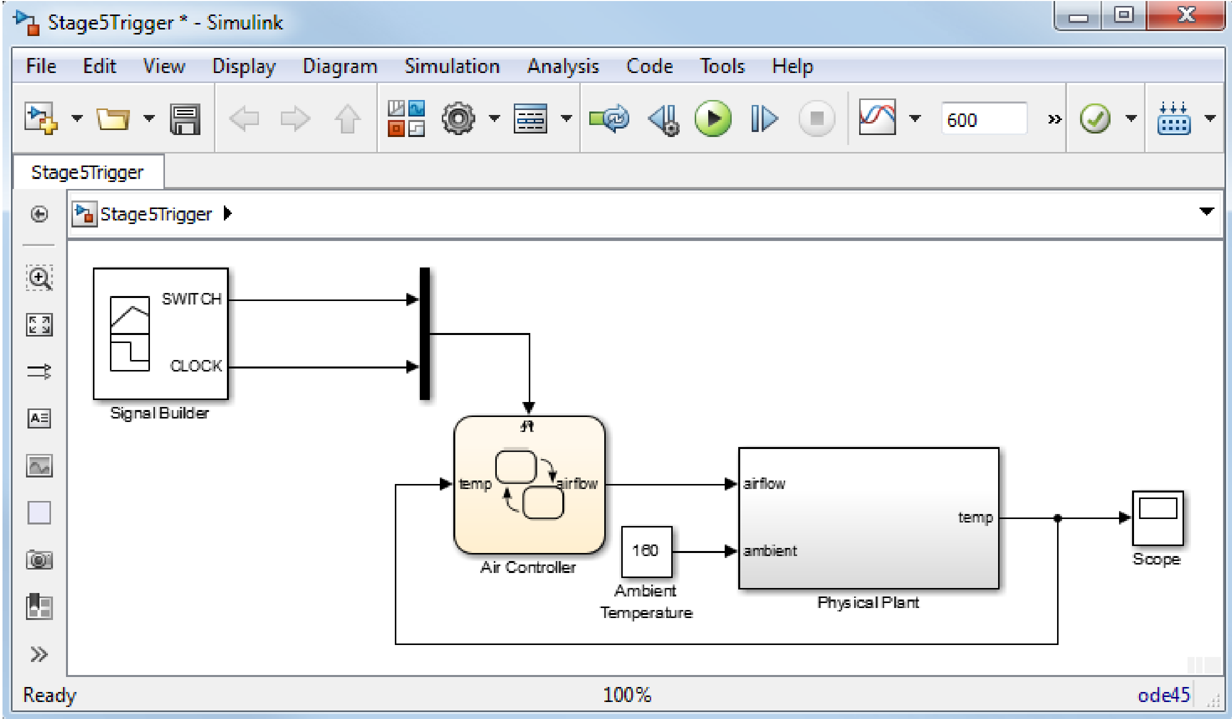
- Associates each event with the correct signal
- Indexes each signal into an array that can be received by the Air Controller trigger port

In `Stage5Trigger`, notice that the two input signals `SWITCH` and `CLOCK` feed into a Mux block where they are joined in an array to a single output. `SWITCH` is a pulse signal and `CLOCK` is a square wave. When you connect the Mux to the trigger port, the index of the signals in the array are associated with the like-numbered ports. Therefore, the `SWITCH` signal at the top input port of the Mux triggers the event `SWITCH` on trigger port 1. Likewise, the `CLOCK` signal at the second input port of the Mux triggers the event `CLOCK` on trigger port 2.

To connect the Mux to the trigger port, follow these steps:

1. Click the Mux block, hold down the **Ctrl** key, and click the Air Controller block.

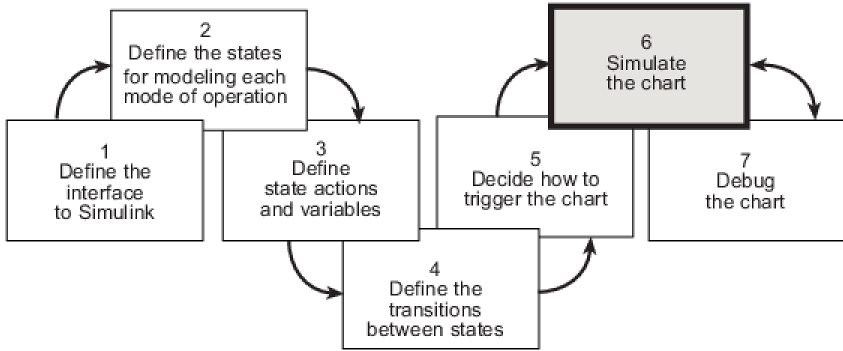
   The output signal of the Mux block connects to the input trigger port of the Stateflow block. Your model should look like this:



2 Save `Stage5Trigger`.

Where to go next. Now you are ready to simulate your chart. See "Setting Simulation Parameters and Breakpoints" on page 7-2.

Simulating the Chart



In phase 6 of this workflow, you *simulate the chart* to test its behavior. During simulation, you can *animate* Stateflow charts to highlight states and transitions as they execute.

Setting Simulation Parameters and Breakpoints

In this section...

"Prepare the Chart Yourself or Use the Supplied Model" on page 7-2
"Checking That Your Chart Conforms to Best Practices" on page 7-2
"Setting the Length of the Simulation" on page 7-3
"Configuring Animation for the Chart" on page 7-4
"Setting Breakpoints to Observe Chart Behavior" on page 7-5
"Simulating the Air Controller Chart" on page 7-5

Prepare the Chart Yourself or Use the Supplied Model

To prepare the chart for simulation yourself, work through the exercises in this section. Otherwise, open the supplied model to see how simulation parameters should appear. Enter this command at the MATLAB prompt:

```
addpath(fullfile(docroot, 'toolbox', 'stateflow', 'gs', 'examples'))
Stage6Simulate
```

Checking That Your Chart Conforms to Best Practices

Before starting a simulation session, you should examine your chart to see if it conforms to recommended design practices:

- A default transition must exist at every level of the Stateflow hierarchy that contains exclusive (OR) states (has exclusive [OR] decomposition). (See "Deciding Where to Place Default Transitions" on page 5-3.)
- Whenever possible, input data objects *should* inherit properties from the associated Simulink input signal to ensure consistency, minimize data entry, and simplify maintenance of your model. Recall that in "Defining the Inputs and Outputs" on page 3-8, you defined the input `temp` to inherit its size and type from the Simulink output port `temp`, which provides the input value to the Air Controller chart.
- Output data objects *should not* inherit types and sizes because the values are back propagated from Simulink blocks and may, therefore, be unpredictable. Recall that in "Defining the Inputs and Outputs" on page 3-8, you specified the data type as `uint8` and the size as scalar (the default). (See "Avoid inheriting output data properties from Simulink blocks" in the Stateflow User's Guide.)

Tip You can specify data types and sizes as expressions in which you call functions that return property values of other variables already defined in Stateflow, MATLAB, or Simulink software. Such functions include type and `fixdt`. For more information, see “Enter Expressions and Parameters for Data Properties” in the Stateflow User's Guide.

Setting the Length of the Simulation

To specify the length of the simulation, follow these steps:

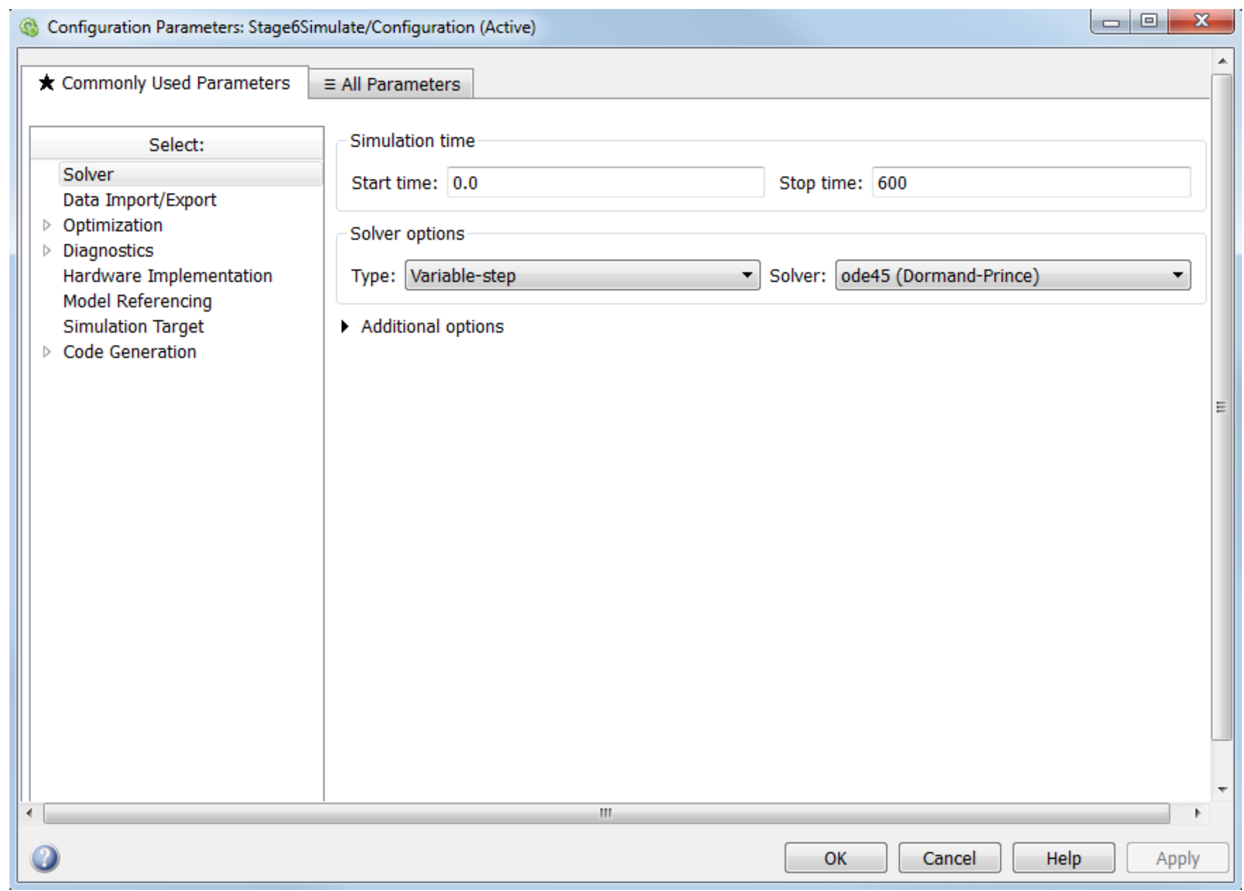
1. Open the model `Stage5Trigger` — either the one you created in the previous exercises or the supplied model for stage 5.

   To open the supplied model, enter the following command at the MATLAB prompt:

   ```
   addpath(fullfile(docroot, 'toolbox', 'stateflow', 'gs', 'examples'))
   Stage5Trigger
   ```
2. Save the model as `Stage6Simulate` in your local work folder.
3. Double-click Air Controller to open the chart.
4. Check the settings for simulation time:

   a. In the Stateflow Editor, select **Simulation** > **Model Configuration Parameters**.

      The following dialog box opens:



- **b** Click **Solver** in the left **Select** pane if it is not already selected.

  Under **Simulation time** on the right, note that the start and stop times have been preset for you. You can adjust these times later as you become more familiar with the run-time behavior of the chart.

- **c** Keep the preset values for now and click **OK** to close the dialog box.

5 Leave the chart open for the next exercise.

Configuring Animation for the Chart

When you simulate a Simulink model, Stateflow animates charts to highlight states and transitions as they execute. Animation provides visual verification that your chart behaves as you expect. Animation is enabled by default to Fast. Slowing it down gives you more time to view the execution order of objects. To configure animation for your simulation session, follow these steps:

1. Set the speed of animation by selecting **Simulation > Stateflow Animation > Medium**. This slows the animation down.
2. Leave the Air Controller chart open for the next exercise.

Setting Breakpoints to Observe Chart Behavior

In this exercise, you will learn how to set breakpoints to pause simulation during key run-time activities so you can observe the behavior of your chart in slow motion. You will set the following breakpoints:

Breakpoint	Description
Chart Entry	Simulation halts when the Stateflow chart wakes up.
State Entry	Simulation halts when a state becomes active.

You will also learn how to examine data values when simulation pauses.

Follow these steps:

1. Right click in the chart, and select **Set Breakpoint on Chart Entry**.
2. For each state `PowerOn` and `PowerOff`, right click in the state, and select **Set Breakpoints > On State Entry**.

Simulating the Air Controller Chart

In this exercise, you will simulate the Air Controller chart. During simulation, you will change breakpoints and observe data values when execution pauses. Follow these steps.

1. In `Stage6Simulate`, open the Scope block. Position the Scope block and the Air Controller chart so they are visible on your desktop.
2. Start simulation by selecting **Simulation > Run**.

After the simulation target is built, the chart appears with a gray background, indicating that simulation has begun. Simulation continues until it reaches the first breakpoint, when the Air Controller chart wakes up.

3. Right click a transition in the state `FAN1`, and select **Add to watch** > **(Input) temp**. This adds the variable `temp` to the Stateflow Breakpoints and Watch window.
4. Right click in the state `SpeedValue`, and select **Add to watch** > **(Output) airflow**. This adds the variable `airflow` to the Stateflow Breakpoints and Watch window.

Tip You can also view data values from the MATLAB command line at simulation breakpoints. Here's how to do it:

a. When simulation pauses at a breakpoint, click in the MATLAB command line and press the **Enter** key.

   The MATLAB Command Window displays a `debug>>` prompt.
b. At the prompt, type the name of the data object.

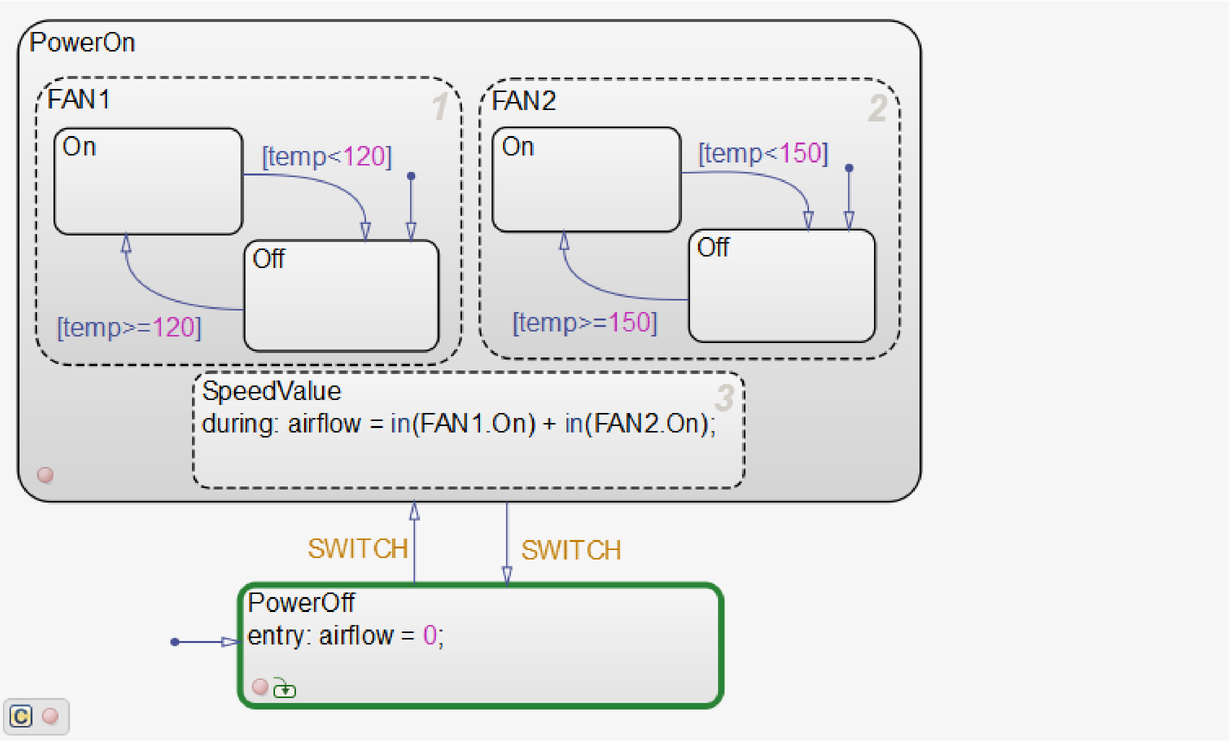
   The MATLAB Command Window displays the value of the data object.

---

5. View the values of `temp` and `airflow`.

   Note that `temp` is 70 (below the threshold for turning on `FAN1`) and `airflow` is 0 (indicating that no fans are running).
6. Resume simulation by clicking the Continue button.

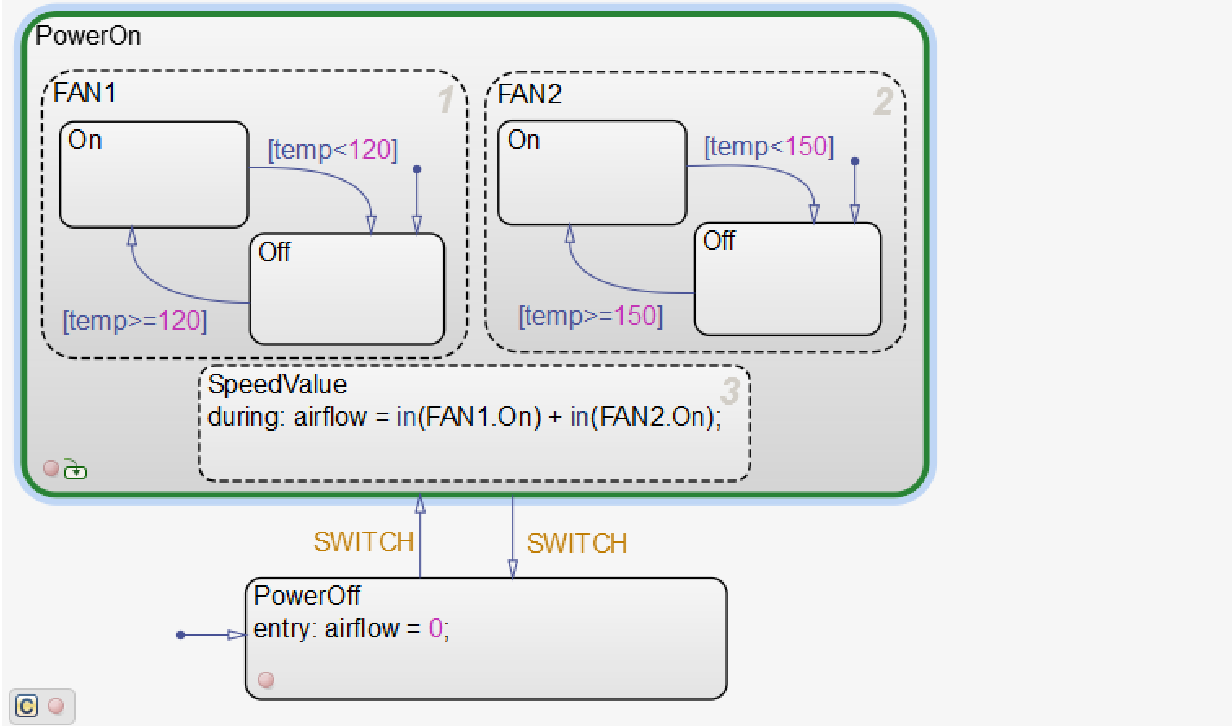
   Simulation continues until the next breakpoint, activation of the `PowerOff` state, which appears highlighted in the chart (as part of animation).



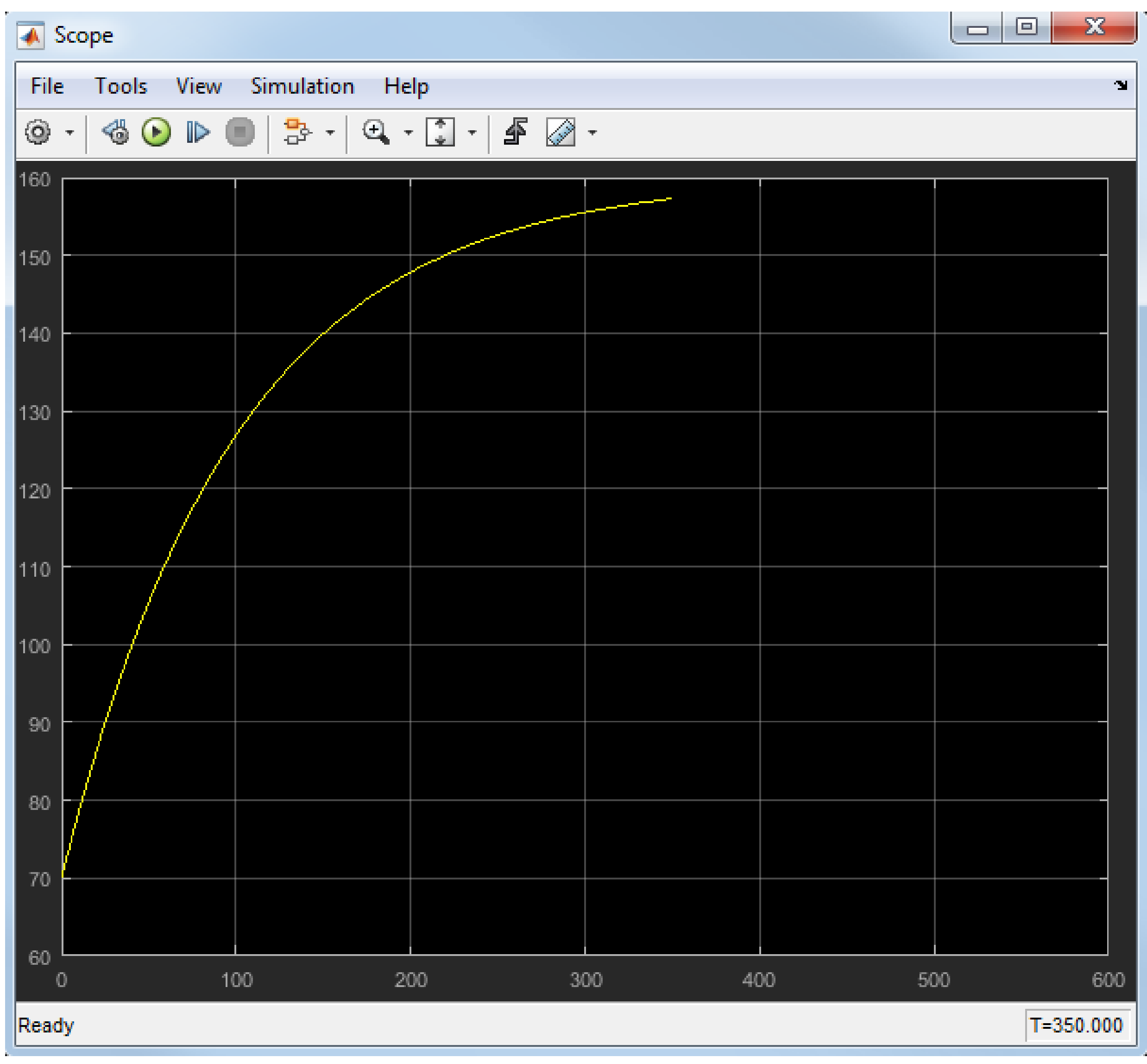
The default transition activates `PowerOff` after the chart wakes up.

7 In the **Breakpoints** tab of the Stateflow Breakpoints and Watch Data window, clear the breakpoint on Chart Entry. Hover the cursor over the name of the breakpoint, and select the delete button, ⊗. Continue simulation.

Simulation continues to the next breakpoint, the activation of the `PowerOn` state:



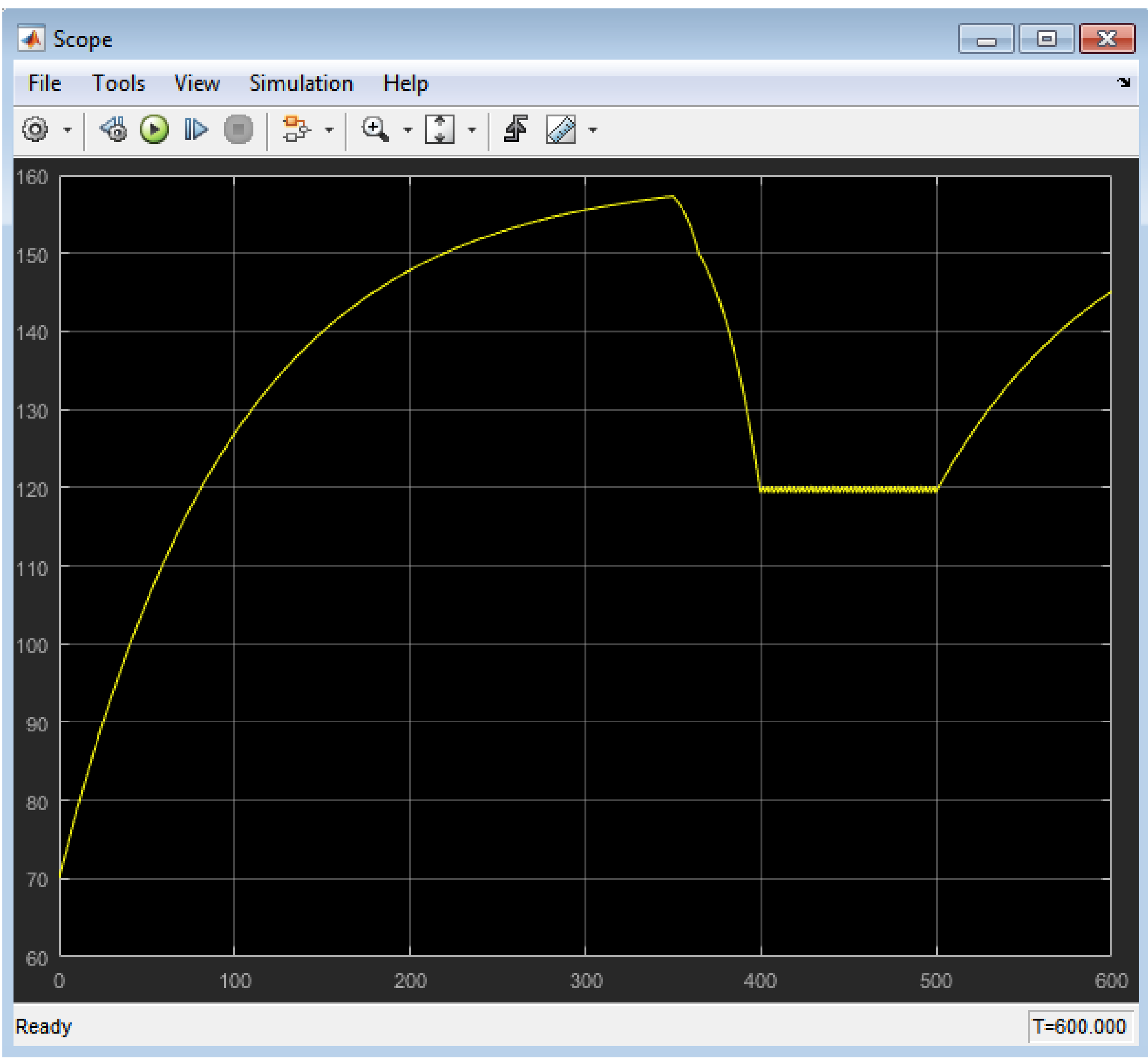
Note that `temp` has risen to over 157 degrees. The Scope displays the temperature pattern:



8 To speed through the rest of the simulation, clear all breakpoints, and continue simulation.

Notice that `FAN1` continues to cycle on and off as `temp` fluctuates between 119 and 120 degrees until power cycles off at 500 seconds. After power cycles off, the fans stop running and `temp` begins to rise unchecked until simulation reaches stop time at 600 seconds.

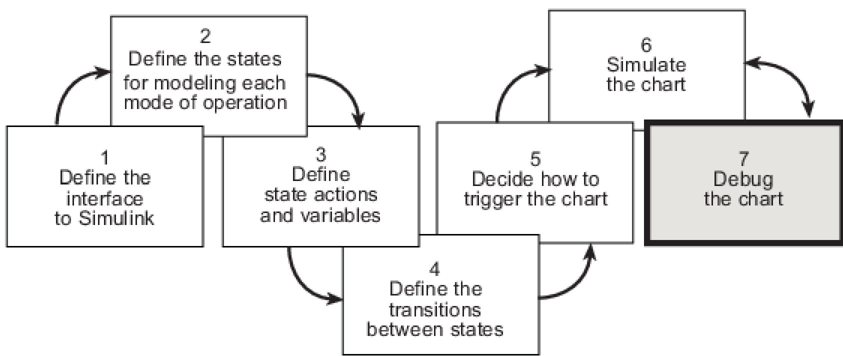
The Scope captures this activity:



Note This display should look the same as the Scope after running the prebuilt model in “Running the Model” on page 2-9.

9 Save `Stage6Simulate`, and close all other windows and dialog boxes.

Debugging the Chart



In phase 7 of this workflow, you *debug the chart*. In "Setting Simulation Parameters and Breakpoints" on page 7-2, you learned how to set breakpoints and watch data. In this chapter, you will learn how Stateflow software detects errors and provides diagnostic assistance.

Debugging Common Modeling Errors

In this section...

"Debugging State Inconsistencies" on page 8-2
"Debugging Data Range Violations" on page 8-4

Debugging State Inconsistencies

In this exercise, you will introduce a state inconsistency error in your chart and troubleshoot the problem. Follow these steps:

1. Open the model `Stage6Simulate` — either the one you created in the previous exercises or the supplied model for stage 6.

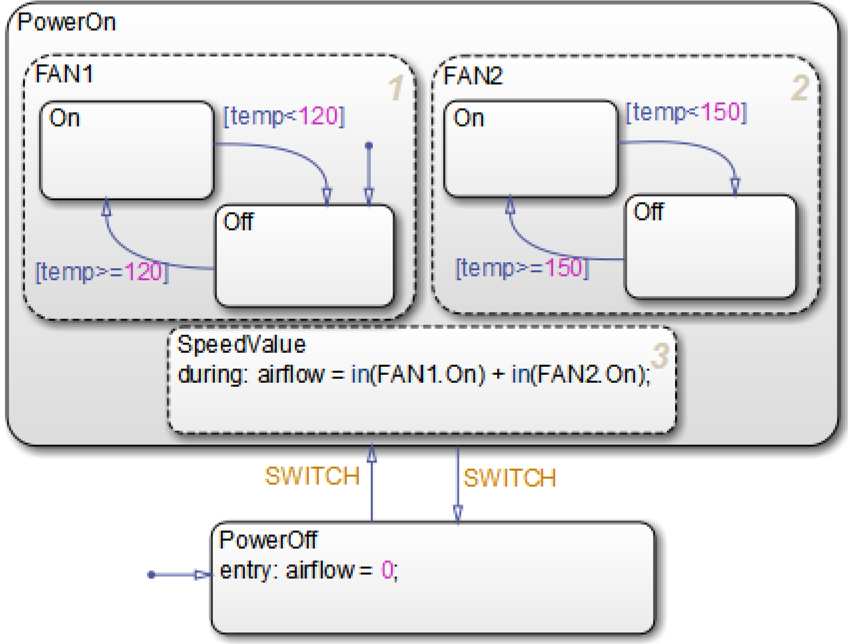
   To open the supplied model, enter the following command at the MATLAB prompt:

   ```
   addpath(fullfile(docroot, 'toolbox', 'stateflow', 'gs', 'examples'))
   Stage6Simulate
   ```

2. Save the model as `Stage7Debug` in your local work folder.
3. Double-click Air Controller to open the chart.
4. Delete the default transition to `FAN2.Off` by selecting it and pressing the **Delete** key.

   Removing the default transition will cause a state inconsistency error. (Recall from "Checking That Your Chart Conforms to Best Practices" on page 7-2 that there must be a default transition at every level of the Stateflow hierarchy that has exclusive [OR] decomposition.)

   Your chart should look like this:



5 Save the chart, and start simulation.

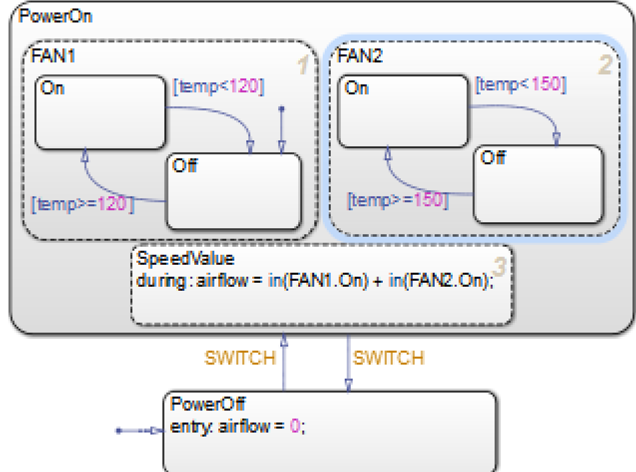
An error appears in the Diagnostic Viewer. The error indicates that the state `FAN2` has no default paths to a substate.

Note The state number in your dialog display can differ from the one pictured above.

---

6 Locate the offending state in the Air Controller chart, by clicking the link to the state name.

`FAN2` appears highlighted in the chart:



7 Add back the default transition to `FAN2.Off`.

The default transition provides an unconditional default path to one of the substates of `FAN2`.

8 Simulate the model again.

This time, simulation proceeds without any errors.

9 Save `Stage7Debug`, and leave the chart open for the next exercise.

Debugging Data Range Violations

In this exercise, you will introduce a data range violation in your chart and troubleshoot the problem. To enable data range violation checking, set **Simulation range checking** in the **Diagnostics: Data Validity** pane of the Configuration Parameters dialog box to `error`.

Follow these steps:

1 In the Air Controller chart, modify the `during` action in the `SpeedValue` state by adding 1 to the computed value, as follows:

```
during: airflow = in(FAN1.On) + in(FAN2.On) + 1;
```

Recall that in “Defining the Inputs and Outputs” on page 3-8, you set a limit range of 0 to 2 for `airflow`. By adding 1 to the computation, the value of `airflow` will exceed the upper limit of this range when two fans are running.

2 Start simulation.

Simulation pauses because of an out-of-range data error:

As expected, the error occurs in the `during` action of `SpeedValue` because the value of `airflow` is out of range.

3 To isolate the problem, double-click the last line in the error message:

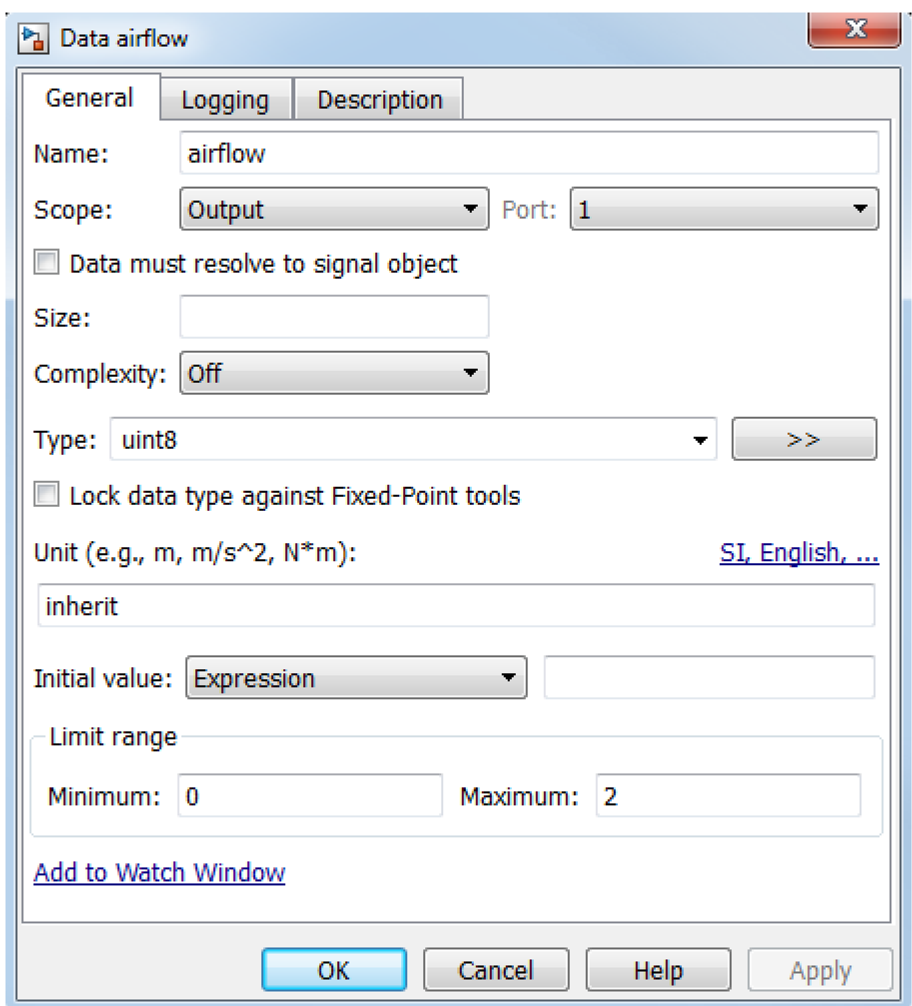
```
Data '#439 (0:0:0)': 'airflow'
```

The Model Explorer opens on your desktop, allowing you to view the properties of `airflow` in the right, read-only pane (read-only because simulation is running).

Note The ID number of the data that appears in the error message can vary from the value shown.

---

4 Check the limit range for `airflow`:



5. Hover your cursor over `airflow` to view the value.

   `airflow` = 3

   This value exceeds the upper limit of 2.
6. Stop simulation.
7. Restore the `during` action to its previous code, and then restart simulation for the model.

The model should simulate with no errors or warnings.

See Also

Related Examples

- “Set Breakpoints to Debug Charts”
- “Watch Stateflow Data Values”