

**Stateflow®**

Getting Started Guide



**MATLAB® & SIMULINK®**

R2023a



## How to Contact MathWorks



Latest news: [www.mathworks.com](http://www.mathworks.com)  
Sales and services: [www.mathworks.com/sales\\_and\\_services](http://www.mathworks.com/sales_and_services)  
User community: [www.mathworks.com/matlabcentral](http://www.mathworks.com/matlabcentral)  
Technical support: [www.mathworks.com/support/contact\\_us](http://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 2004–2023 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](http://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](http://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)
September 2019	Online only	Revised for Version 10.1 (Release 2019b)
March 2020	Online only	Revised for Version 10.2 (Release 2020a)
September 2020	Online only	Revised for Version 10.3 (Release 2020b)
March 2021	Online only	Revised for Version 10.4 (Release 2021a)
September 2021	Online only	Revised for Version 10.5 (Release 2021b)
March 2022	Online only	Revised for Version 10.6 (Release 2022a)
September 2022	Online only	Revised for Version 10.7 (Release 2022b)
March 2023	Online only	Revised for Version 10.8 (Release 2023a)



## Introduction to the Stateflow Product

**1**

<b>Stateflow Product Description</b> .....	<b>1-2</b>
--	------------

## Stateflow Fundamentals

**2**

<b>Model Finite State Machines</b> .....	<b>2-2</b>
Example of a Stateflow Chart .....	<b>2-2</b>
Execute Chart as a MATLAB Object .....	<b>2-3</b>
Simulate Chart as a Simulink Block With Local Events .....	<b>2-4</b>
Simulate Chart as a Simulink Block With Temporal Conditions .....	<b>2-7</b>
Next Steps .....	<b>2-9</b>
<b>Construct and Run a Stateflow Chart</b> .....	<b>2-10</b>
Construct the Stateflow Chart .....	<b>2-10</b>
Simulate the Chart as a Simulink Block .....	<b>2-13</b>
Execute the Chart as a MATLAB Object .....	<b>2-15</b>
<b>Define Chart Behavior by Using State and Transition Actions</b> .....	<b>2-17</b>
<b>Create a Hierarchy to Manage System Complexity</b> .....	<b>2-21</b>
<b>Model Synchronous Subsystems by Using Parallel Decomposition</b> ....	<b>2-25</b>
<b>Synchronize Parallel States by Broadcasting Events</b> .....	<b>2-29</b>
<b>Monitor Chart Activity by Using Active State Data</b> .....	<b>2-36</b>
<b>Schedule Chart Actions by Using Temporal Logic</b> .....	<b>2-42</b>

## Additional Learning Tools

**3**



# Introduction to the Stateflow Product

---

## **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.



# Stateflow Fundamentals

---

This chapter describes the fundamental concepts of event-based modeling in Stateflow.

- “Model Finite State Machines” on page 2-2
- “Construct and Run a Stateflow Chart” on page 2-10
- “Define Chart Behavior by Using State and Transition Actions” on page 2-17
- “Create a Hierarchy to Manage System Complexity” on page 2-21
- “Model Synchronous Subsystems by Using Parallel Decomposition” on page 2-25
- “Synchronize Parallel States by Broadcasting Events” on page 2-29
- “Monitor Chart Activity by Using Active State Data” on page 2-36
- “Schedule Chart Actions by Using Temporal Logic” on page 2-42

## 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.

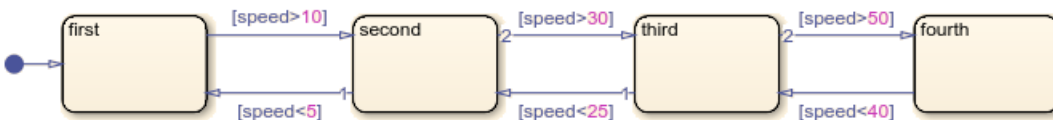
Finite state machines are representations of dynamic systems that transition from one mode of operation (state) to another. State machines:

- 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.

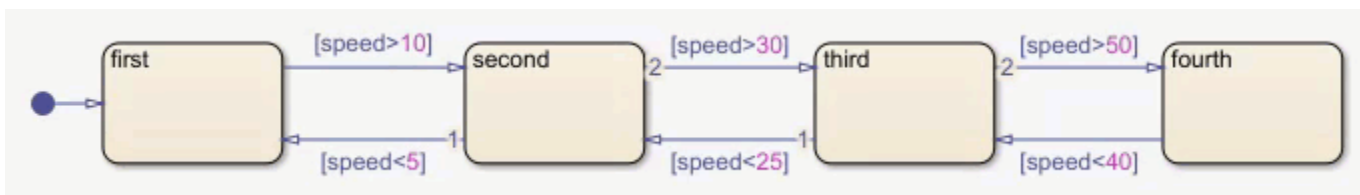
Control systems design relies heavily on state machines to manage complex logic. Applications include designing aircraft, automobiles, and robotics control systems.

### 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.

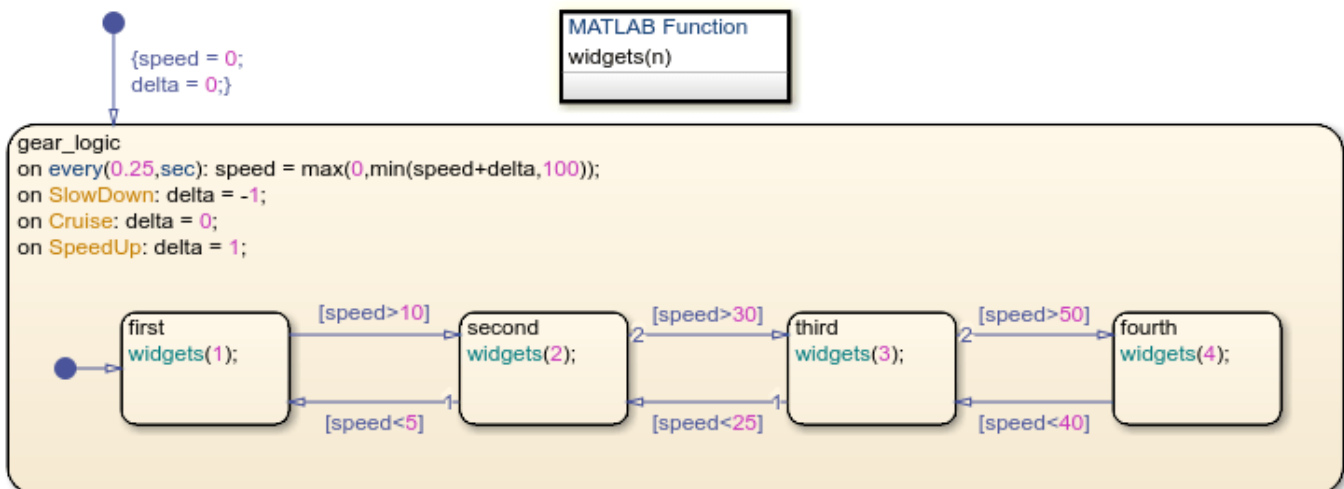


This chart offers a simple design that disregards important factors such as engine speed and torque. You can construct a more comprehensive and realistic model by linking this Stateflow chart with other components in MATLAB or Simulink. The following examples describe three possible approaches.

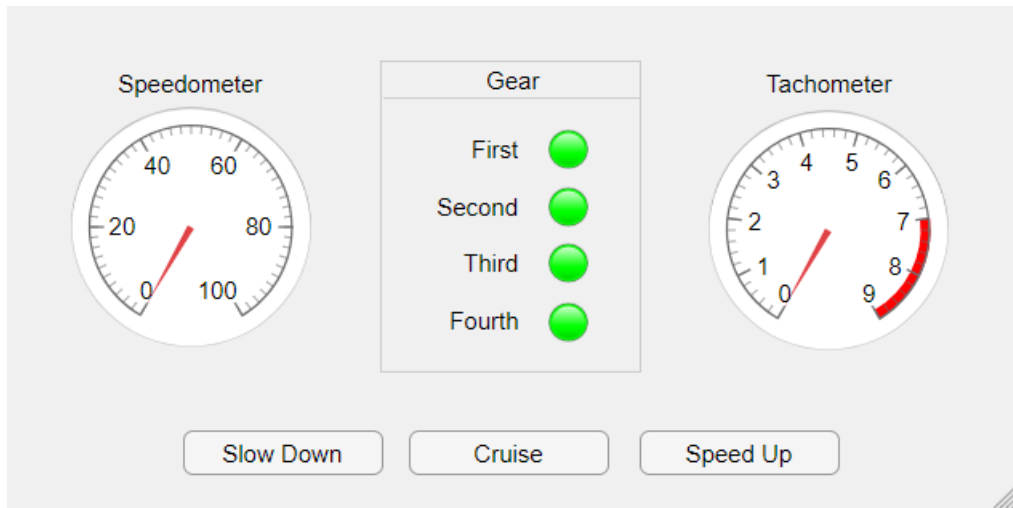
## Execute Chart as a MATLAB Object

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.



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. To start the example, in the App Designer toolstrip, click **Run**. The example continues to run until you close the user interface window.

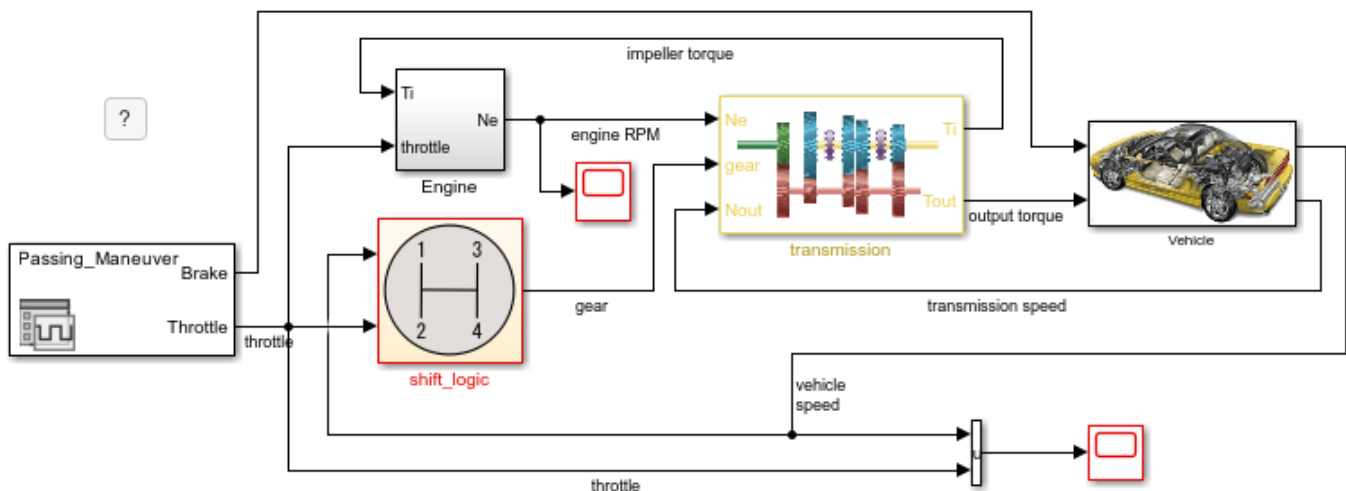


Alternatively, in the Stateflow Editor, in the **State Chart** Tab, click **Run**. To control the speed of the car, in the **Symbols** pane, use the **SpeedUp**, **SlowDown**, and **Cruise** buttons. To stop the example, click **Stop**.

For more information about executing Stateflow charts as MATLAB objects, see “Execution in MATLAB”.

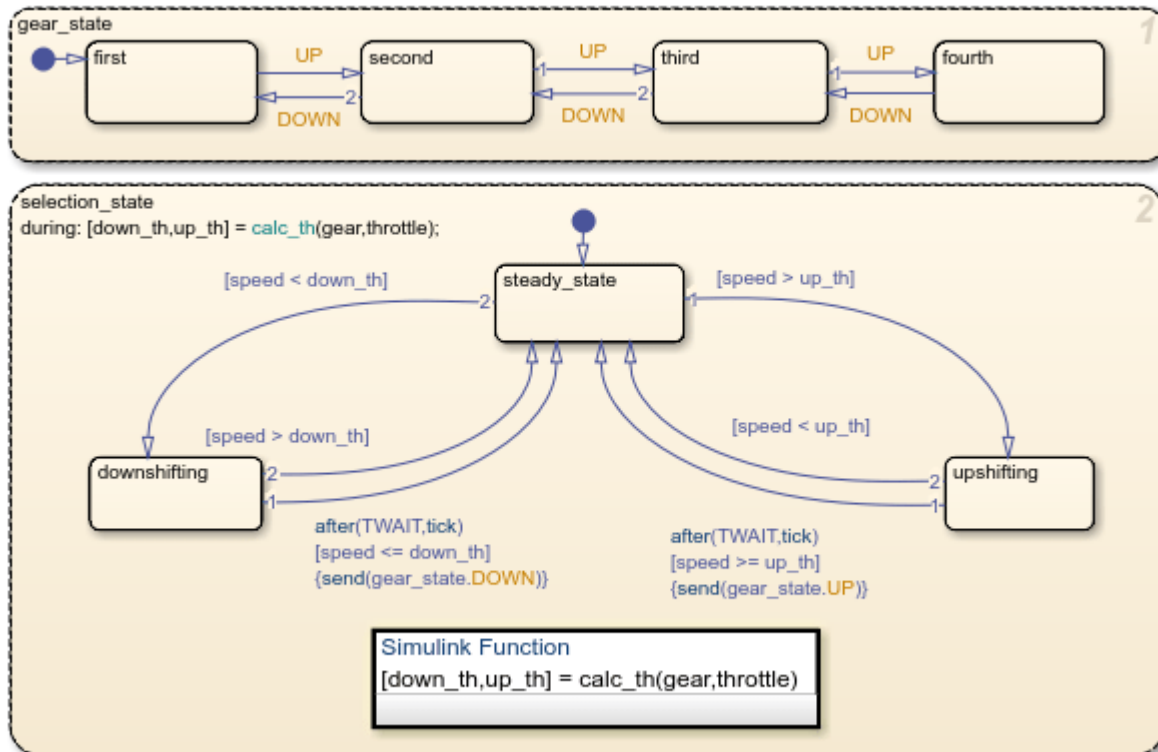
## Simulate Chart as a Simulink Block With Local Events

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.

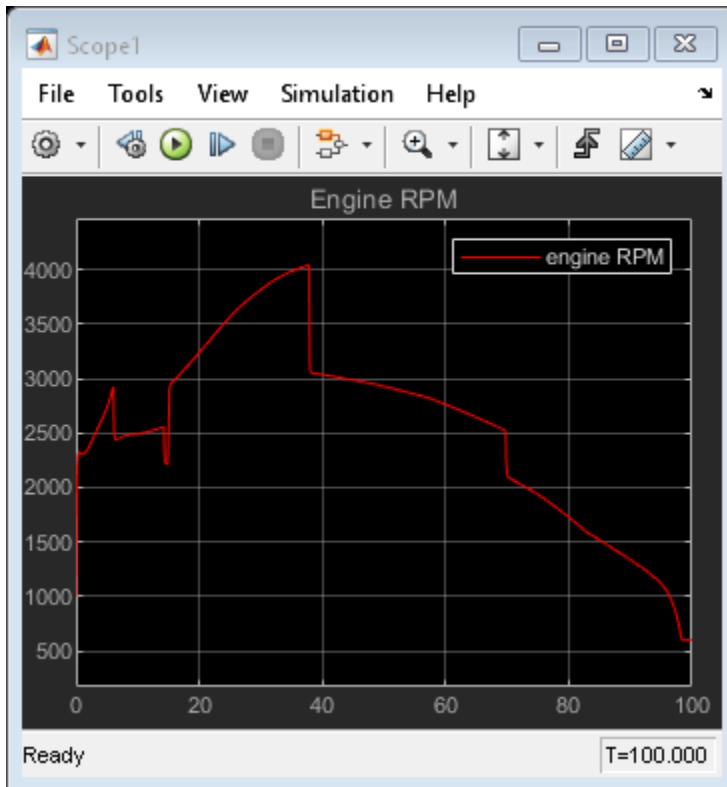
- **Hierarchy:** The state `gear_state` contains a modified version of the four-speed automatic transmission chart. The state `selection_state` contains substates that represent the steady state, upshifting, and downshifting modes of operation. When circumstances require a shift to a higher or lower 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`.

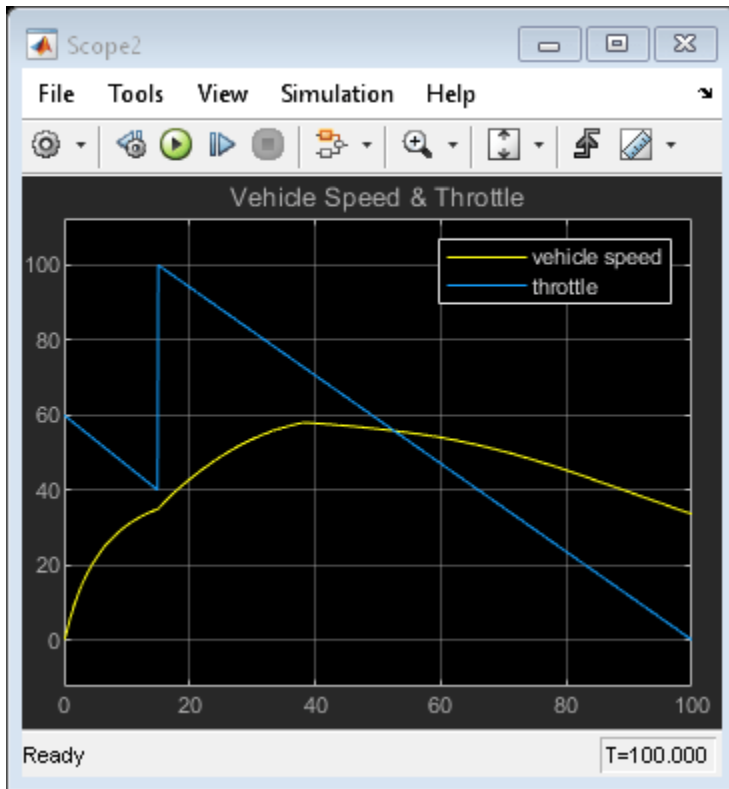


To run a simulation of the model:

- 1 Double-click the **User Inputs** block. In the Signal Editor dialog box, select a predefined brake-to-throttle profile from the **Active Scenario** list. The default profile is **Passing Maneuver**.

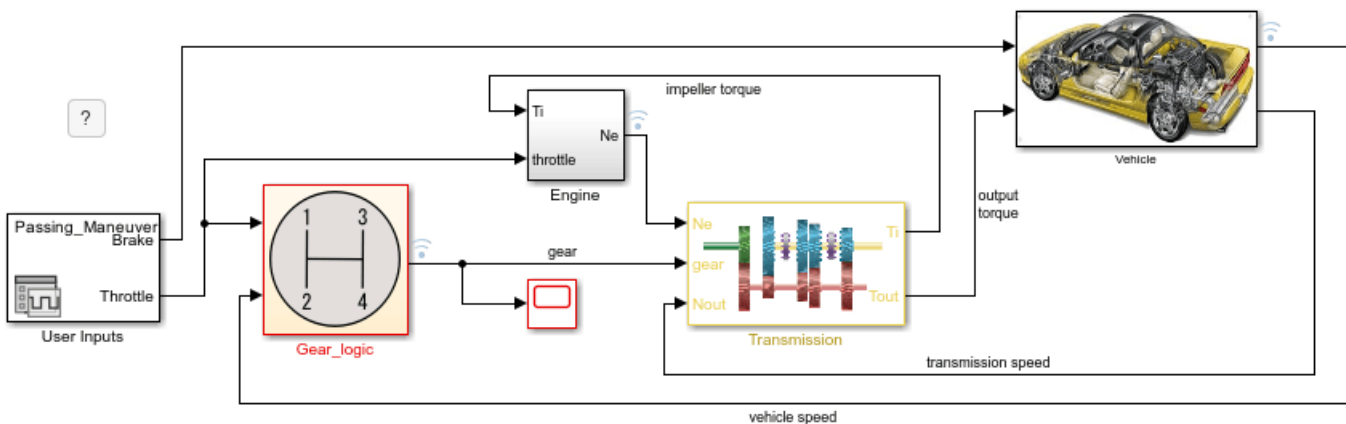
- 2 Click **Run**. In the Stateflow Editor, chart animation highlights the active states during the simulation. To slow down the animation, in the **Debug** tab, select **Slow** from the **Animation Speed** drop-down list.
- 3 In the Scope blocks, examine the results of the simulation. Each scope displays a graph of its input signals during simulation.





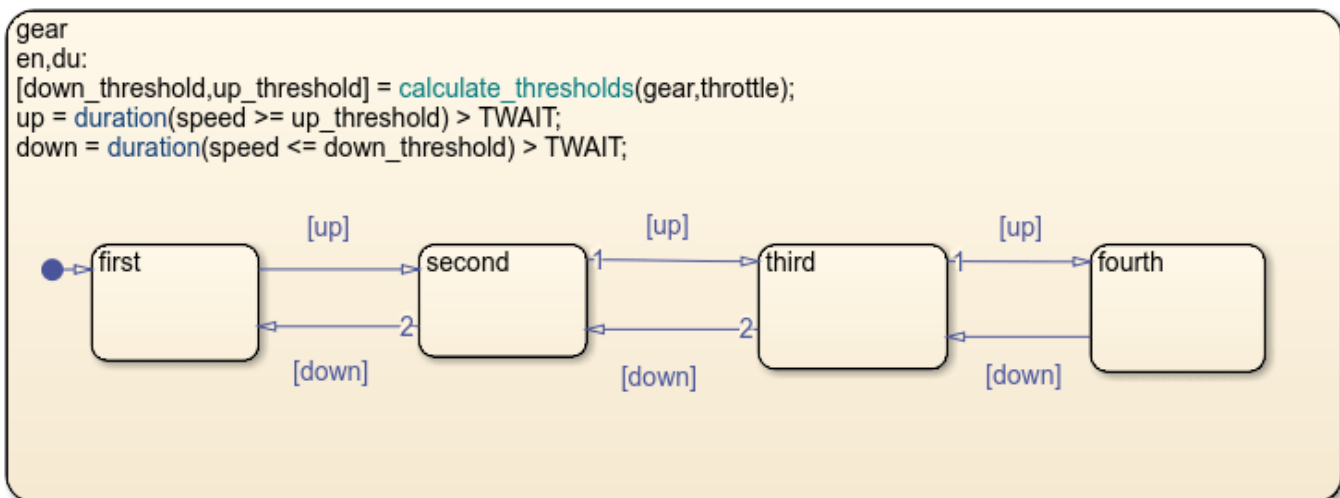
### Simulate Chart as a Simulink Block With Temporal Conditions

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 block 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).



```

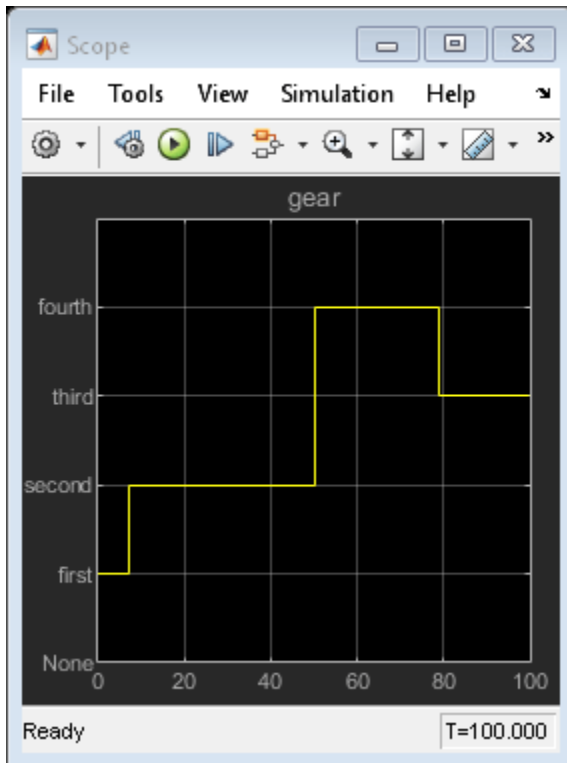
Simulink Function
[down_th,up_th] = calculate_thresholds(gear,throttle)

```

To run a simulation of the model:

- 1 Double-click the **User Inputs** block. In the Signal Editor dialog box, select a predefined brake-to-throttle profile from the **Active Scenario** list. The default profile is **Passing Maneuver**.
- 2 Click **Run**. In the Stateflow Editor, chart animation highlights the active states during the simulation. To slow down the animation, in the **Debug** tab, select **Slow** from the **Animation Speed** drop-down list.
- 3 In the Scope block, examine the results of the simulation. The scope displays a graph of the gear selected during simulation.





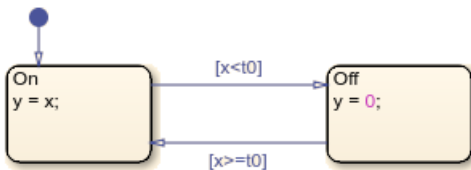
## Next Steps

- 1 "Construct and Run a Stateflow Chart" on page 2-10
- 2 "Define Chart Behavior by Using State and Transition Actions" on page 2-17
- 3 "Create a Hierarchy to Manage System Complexity" on page 2-21
- 4 "Model Synchronous Subsystems by Using Parallel Decomposition" on page 2-25
- 5 "Synchronize Parallel States by Broadcasting Events" on page 2-29
- 6 "Monitor Chart Activity by Using Active State Data" on page 2-36
- 7 "Schedule Chart Actions by Using Temporal Logic" on page 2-42

## 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 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  $t_0$ , 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. Before opening the Stateflow Editor, decide on the chart execution mode that best meets your needs.

- To model conditional, event-based, and time-based logic for periodic or continuous-time Simulink algorithms, use the `sfnew` function 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
```

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

- To design reusable state machine and timing logic for MATLAB applications, use the `edit` function to create a standalone Stateflow chart that you can execute as a MATLAB object. At the MATLAB command prompt, enter:

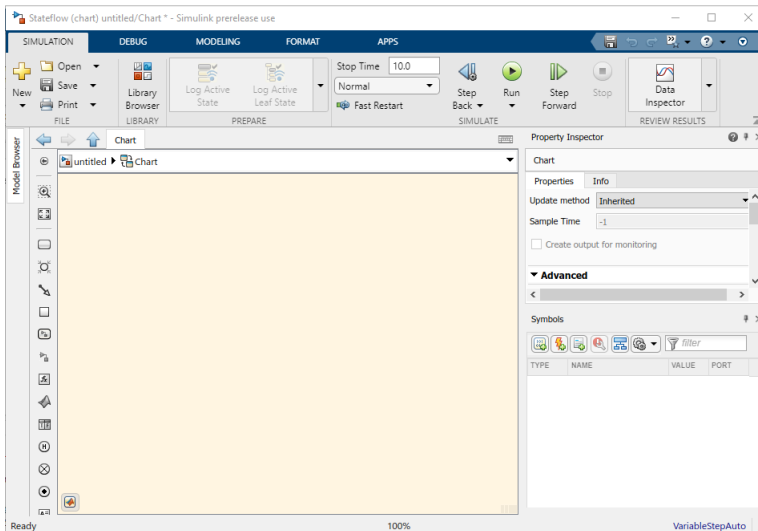
```
edit rectify.sfx % create chart for execution as a MATLAB object
```

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

The main components of the Stateflow Editor are the chart canvas, the object palette, and the **Symbols** pane.

- 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** pane, you add new data, events, and messages to the chart and resolve any undefined or unused symbols.




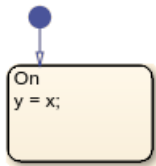

---

**Tip** After you construct your Stateflow chart, you can copy its contents to another chart with a different execution mode. For example, you can construct a chart for execution in MATLAB and copy its contents into a chart for simulation in Simulink.

---

### 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;`.

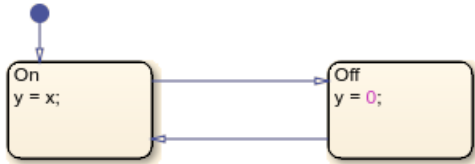


- 2 Add another state. Right-click and drag the `On` state. Blue graphical cues help you to align your states horizontally or vertically. The name of the new state changes to `Off`. Double-click the state and modify the state action to `y = 0;`.

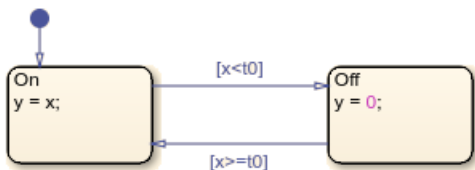



- 3 Realign the two states and pause on the space between the two states. Blue transition cues indicate several ways in which you can connect the states. To add transitions, click the appropriate cue.

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  $x < t\theta$  or  $x \geq t\theta$ . 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, in the **Format** tab, click **Auto Arrange** 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** pane, undefined symbols are marked with a red error badge . The **Type** column displays the suggested scope for each undefined symbol based on its usage in the chart.

- 1 Open the **Symbols** pane.
  - If you are building a chart in a Simulink model, in the **Modeling** tab, under **Design Data**, select **Symbols Pane**.
  - If you are building a standalone chart for execution in MATLAB, in the **State Chart** tab, select **Add Data > Symbols Pane**.
- 2 In the **Symbols** pane, click **Resolve Undefined Symbols** .
  - If you are building a chart in a Simulink model, the Stateflow Editor resolves the symbols  $x$  and  $t\theta$  as input data  and  $y$  as output data .
  - If you are building a standalone chart for execution in MATLAB, the Stateflow Editor resolves  $t\theta$ ,  $x$ , and  $y$  as local data .



TYPE	NAME	VALUE	PORT
	t0		
	x		
	y		

TYPE	NAME	VALUE	PORT
	t0		1
	x		2
	y		1

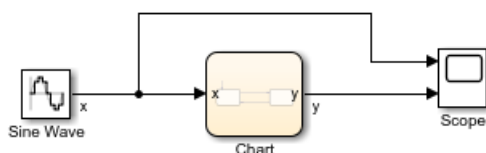
- Because the threshold  $t_0$  does not change during simulation, change its scope to constant data. In the **Type** column, click the data type icon next to  $t_0$  and select Constant Data.
- Set the value for the threshold  $t_0$ . In the **Value** column, click the blank entry next to  $t_0$  and enter a value of 0.
- Save your Stateflow chart.


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

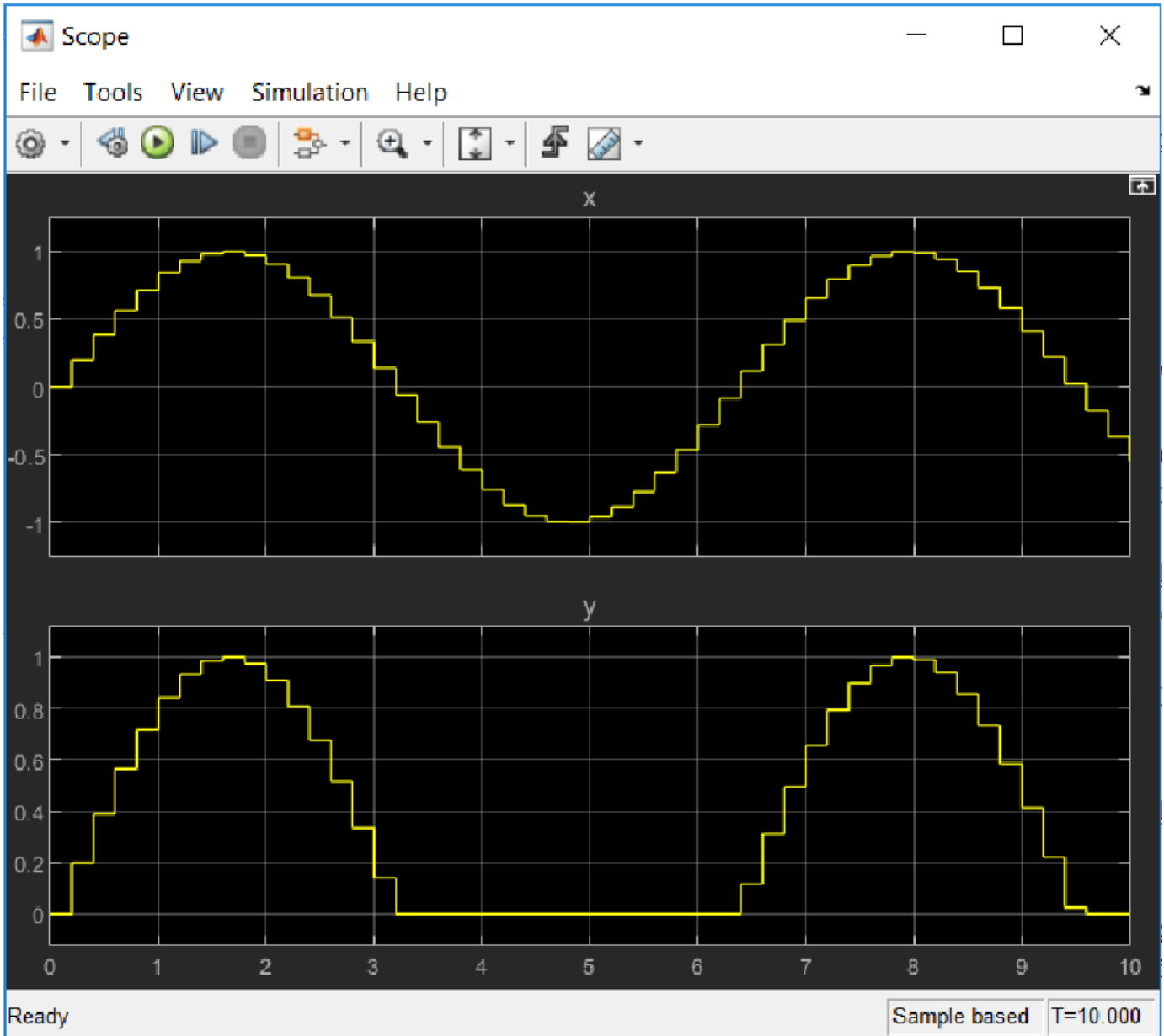
## Simulate the Chart as a Simulink Block

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 2-15.

- 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.
- 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  $x$ .
- 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  $y$ .
- Save the Simulink model.



- 5 To simulate the model, click **Run** . 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 2-13.

- 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
```

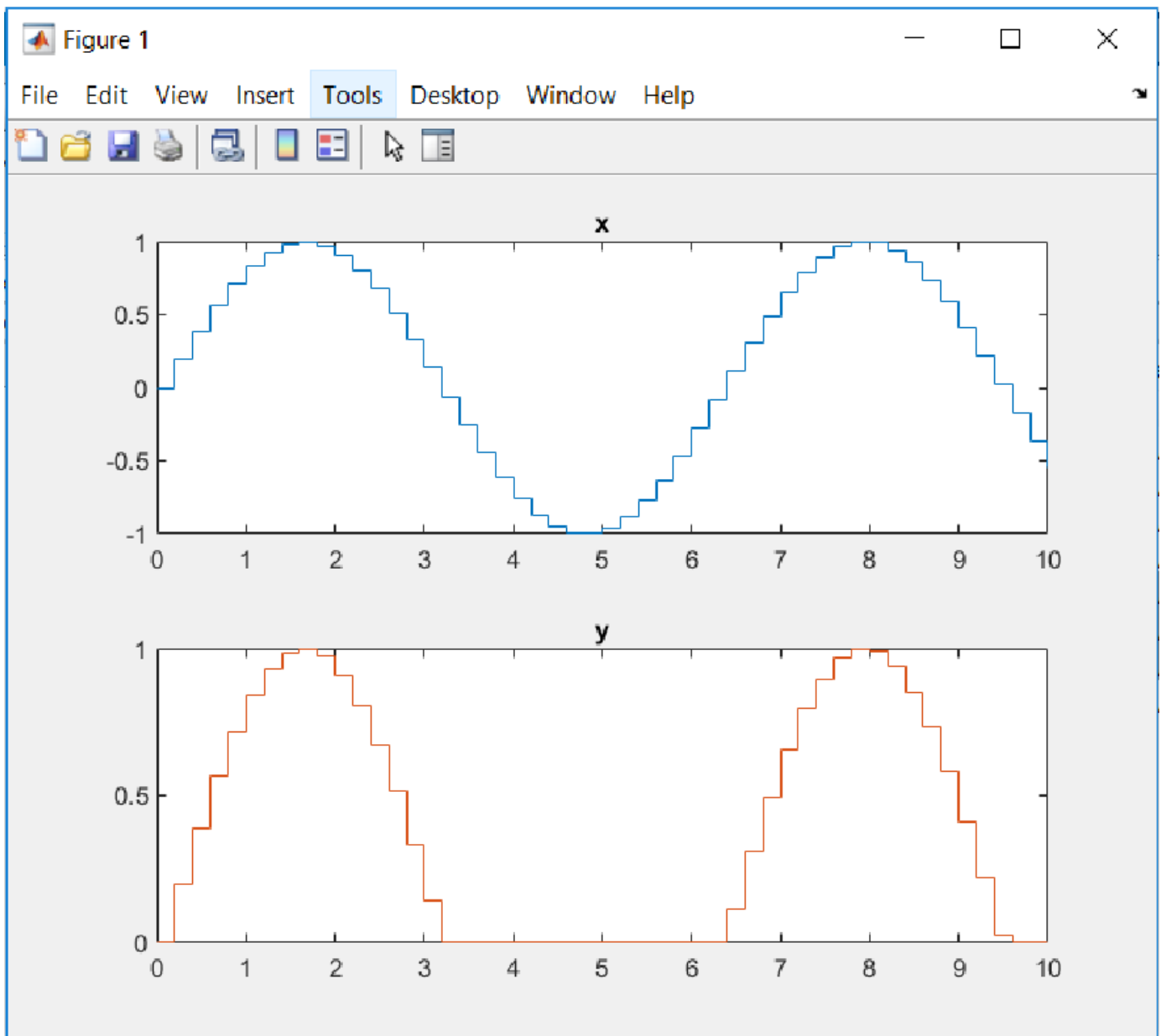
- 4 Delete the chart object `r` from the MATLAB workspace.

```
delete(r)
```

- 5 Examine the results of the chart execution. For example, you can call the `stairs` function to create a staircase 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

### More About

- "Stateflow Editor Operations"
- "Manage Symbols in the Stateflow Editor"
- "Create Stateflow Charts for Execution as MATLAB Objects"

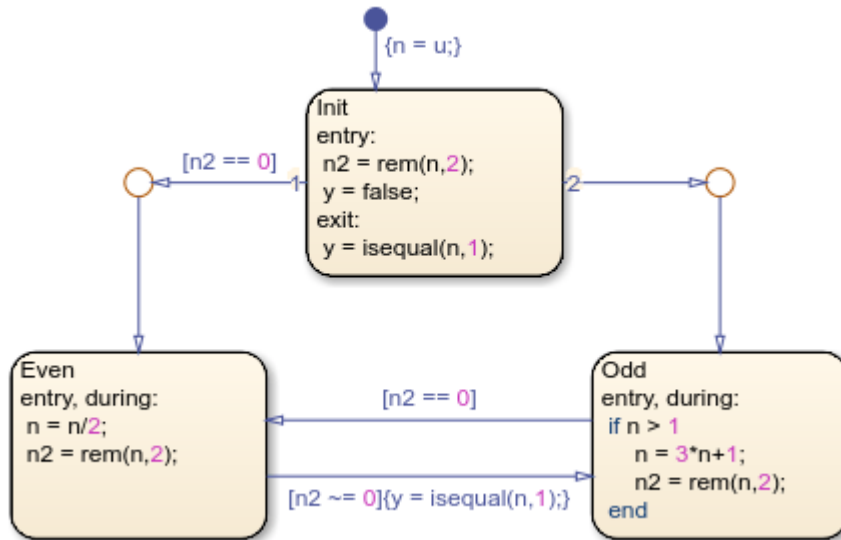


## Define Chart Behavior by Using State and Transition Actions

*State actions* and *transition actions* are instructions that you write inside a state or on a transition, respectively, to define how a Stateflow® chart behaves during simulation. For example, the actions in this chart define a state machine that empirically verifies one instance of the Collatz conjecture. For a given numeric input  $u$ , the chart computes the hailstone sequence  $n_0 = u, n_1, n_2, n_3, \dots$  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 one.



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

- Local data `n` to the value of the input `u`.
- Local data `n2` to the remainder when `n` is divided by two.
- Output data `y` 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 (Simulink)` 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:

- `entry` actions occur when the state becomes active.
- `during` actions occur on a time step when the state is already active and the chart does not transition out of the state.
- `exit` actions occur when the chart transitions out of the state.

You can specify the type of a state action by using a complete keyword (`entry`, `during`, `exit`) or an 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.

The hailstone chart contains actions in these states:

- `Init` — When this state becomes active at the start of the simulation, the `entry` action determines the parity of `n` and sets `y` to `false`. When the chart transitions out of `Init` after one time step, the `exit` action determines whether `n` is equal to one.
- `Even` — When this state becomes active, and on every subsequent time step that the state is active, the combined `entry, during` action computes the value and parity for next number of the hailstone sequence,  $n/2$ .
- `Odd` — When this state becomes active, and on every subsequent time step that the state is active, the combined `entry, during` action checks whether `n` is greater than one and, if it is, computes the value and parity for next number of the hailstone sequence,  $3*n+1$ .

### Transition Action Types

Transition actions define what a Stateflow chart does when the active state changes. The most common types of transition actions are conditions and condition actions. To specify transition actions, use a label with this syntax:

```
[Condition]{ConditionAction}
```

`Condition` is a Boolean expression that determines whether the transition occurs. If you do not specify a condition, the transition occurs one time step after the source state becomes active.

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

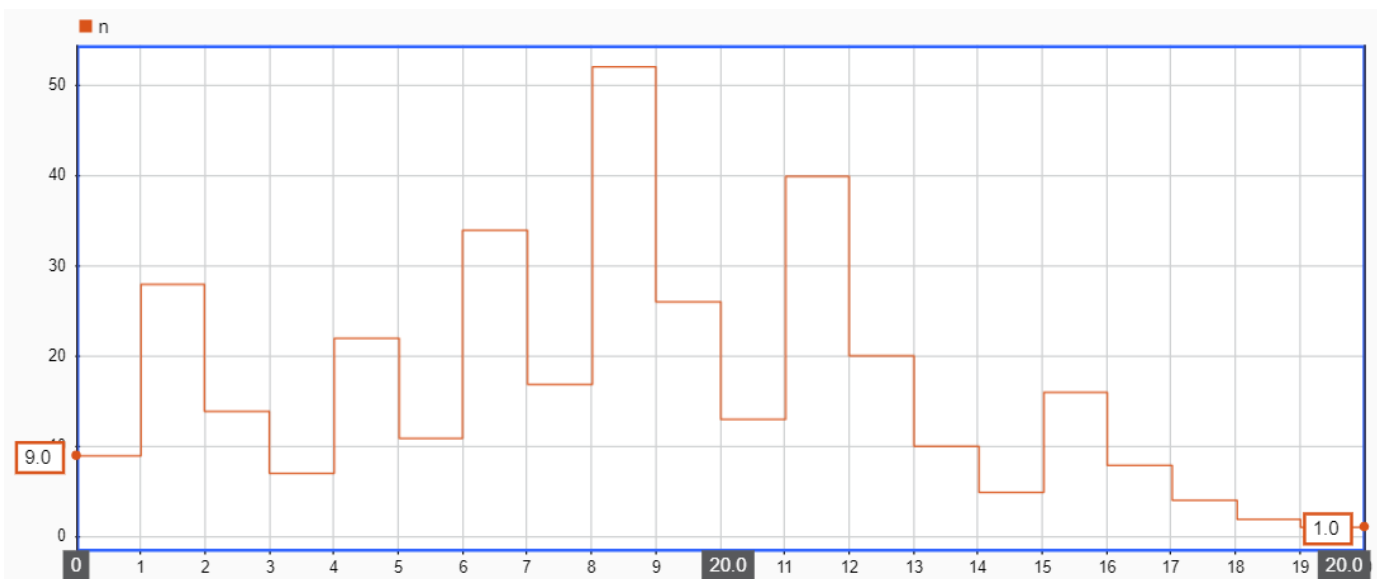
The hailstone chart contains actions on these transitions:

- Default transition into `Init` — At the start of the simulation, the condition action `n = u` assigns the input value `u` to the local data `n`.
- Transition from `Init` to `Even` — The condition `n2 == 0` determines that the transition occurs when `n` is even. The number 1 at the source of this transition indicates that this transition is evaluated before the transition `Init` to `Odd`.
- Transition from `Odd` to `Even` — The condition `n2 == 0` determines that the transition occurs when `n` is even.
- Transition from `Even` to `Odd` — The condition `n2 ~= 0` determines that the transition occurs when `n` is odd. In this case, the condition action `y = isequal(n, 1)` determines whether `n` is equal to one.

### Examine Chart Behavior

To compute the hailstone sequence starting with a value of nine:

1. In the Constant block, enter a value of 9.
2. In the **Simulation** tab, click **Run**. 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 state **Init** 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 state **Init** becomes inactive and the state **Odd** becomes active. The entry action in **Odd** sets  $n$  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 state **Odd** becomes inactive and the state **Even** becomes active. The entry action in **Even** sets  $n$  to 14 and  $n2$  to 0.
  - At time  $t = 3$ , the condition  $n2 \sim= 0$  is false so the chart does not take a transition. The state **Even** remains active. The during action in **Even** sets  $n$  to 7 and  $n2$  to 1.
  - At time  $t = 4$ , the condition  $n2 \sim= 0$  is true so the chart prepares to transition to **Odd**. The transition action sets  $y$  to **false**. The state **Even** becomes inactive and the state **Odd** 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**. The transition action sets  $y$  to **true**. The state **Even** becomes inactive and the state **Odd** 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.
3. In the **Simulation** tab, under **Review Results**, click **Data Inspector**.
4. To see the values of the hailstone sequence, in the Simulation Data Inspector, select the logged signal  $n$ .



The hailstone sequence reaches a value of one after 19 iterations.

### **See Also**

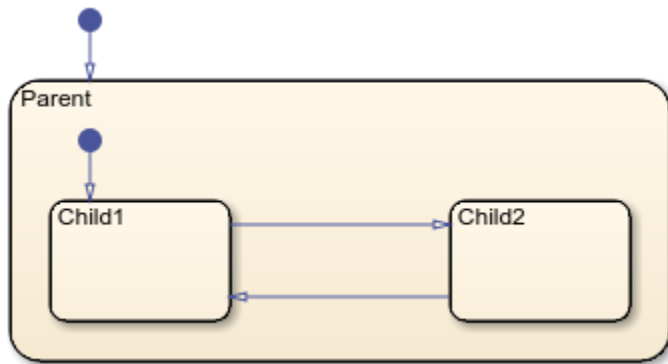
### **More About**

- “Represent Operating Modes by Using States”
- “Define Actions in a Transition”

## 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.

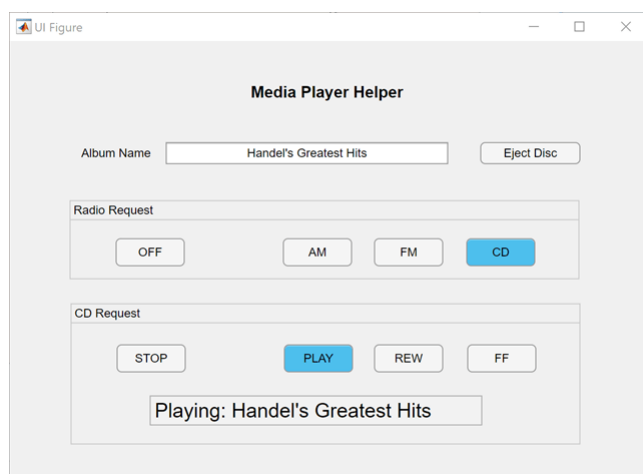
To create a hierarchy of states, place one or more states inside the boundaries of another state. The inner states are child states, or *substates*, of the outer state. The outer state is the parent, or *superstate*, of the inner states.



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.

### Model a Media Player

This example models a media system consisting of an AM radio, an FM radio, and a CD player. During simulation, you control the media player by clicking buttons on the Media Player Helper user interface.



The media player is initially off. When you select one of the **Radio Request** buttons, the media player 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.

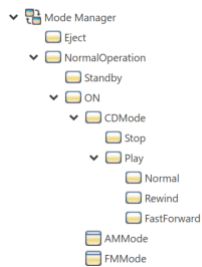
## Implement Behavior by Using State Hierarchy

This example implements the media player incrementally by focusing on a single level of activity at a time. For example, these conditions are necessary for the CD player to enter fast-forward play mode:

- 1 You turn on the media player.
- 2 You turn on the CD player.
- 3 You start playing a disc.
- 4 You click the FF button.

The model uses a hierarchy of states to consider each one of these conditions individually. For instance, the top level of the hierarchy defines the operating modes as the media player turns on and off. The middle levels define the transitions between the different media player subcomponents, and between the stop and play modes of the CD player. The bottommost level of the hierarchy defines the response to the **CD Request** buttons when you meet all the other conditions for playing a disc.

In the Model Explorer, under the **Mode Manager** chart, you can see the hierarchy of nested states that implement the behavior of the media player. To open the Model Explorer, in the **Modeling** tab, select **Model Explorer**.



At the top level of the hierarchy, the **Mode Manager** chart has two states:

- **Eject** corresponds to the disc ejection mode, which interrupts all other media player functions.
- **NormalOperation** corresponds to the normal operating mode for the media player.

The child states of **NormalOperation** control the activity of the media player:

- **Standby** is active when the media player is off.
- **ON** is active when the media player is on.

The child states of **On** control the media player subcomponents:

- **CDMode** is active when the CD player subcomponent is on.
- **AMMode** is active when the AM radio subcomponent is on.
- **FMMode** is active when the FM radio subcomponent is on.

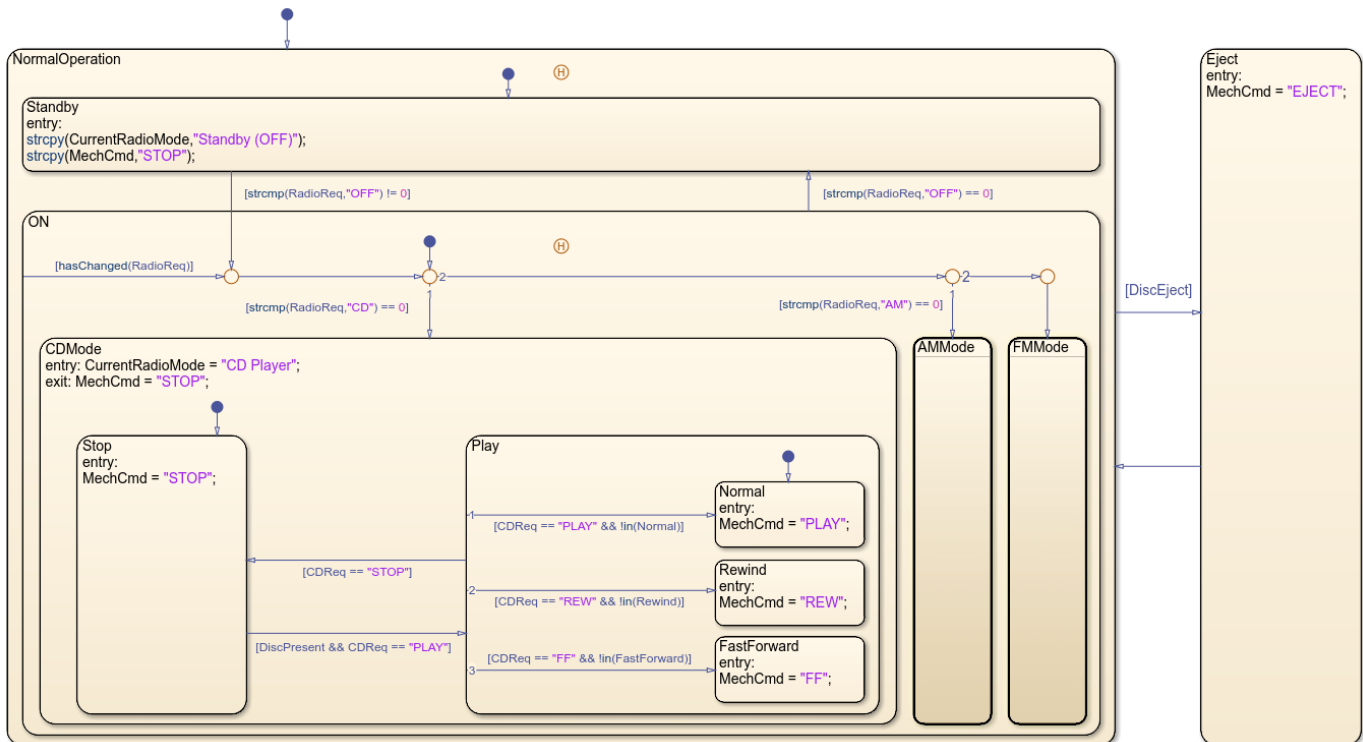
The child states of **CDMode** control the activity of the CD player:

- **Stop** is active when the CD player is stopped.
- **Play** is active when the CD player is playing a disc.

The child states of **Play** control the play modes for the CD player:

- Normal is active during normal play mode.
- Rewind is active during reverse play mode
- FastForward is active during fast-forward play mode

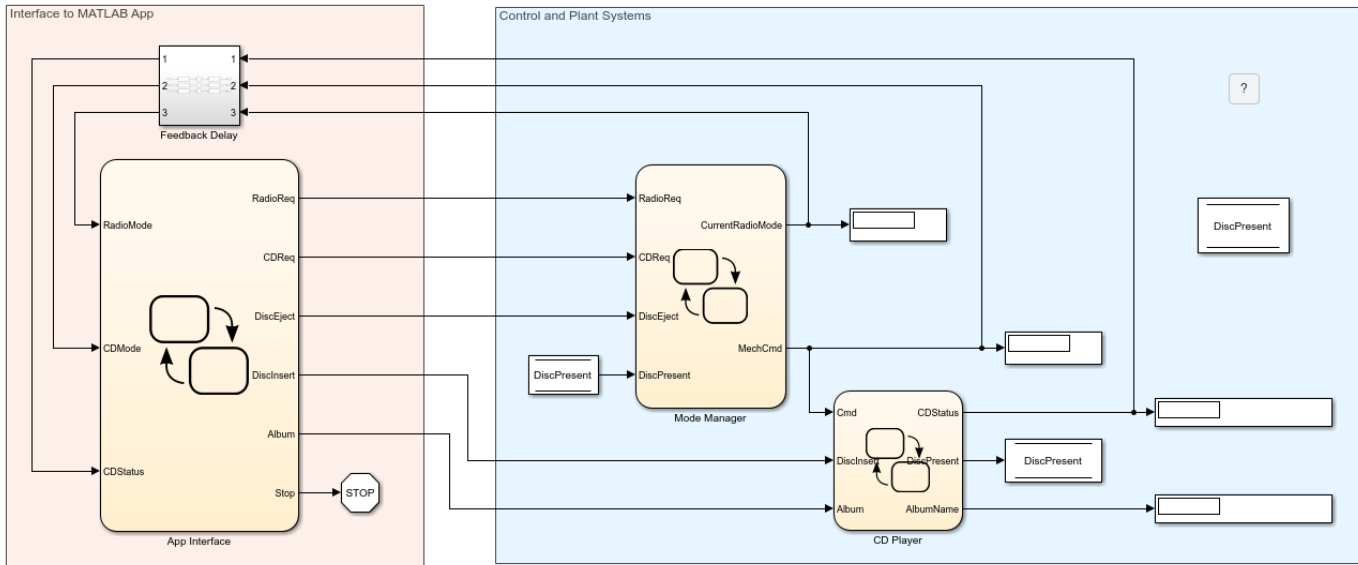
This figure shows the complete layout of the states in the chart.



### Explore the Example

The model in this example contains two other Stateflow charts:

- App Interface manages the interface with the UI and passes inputs to the Mode Manager and CD Player charts.
- CD Player receives the output from the App Interface and Mode Manager charts and mimics the mechanical behavior of the CD player.



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

### See Also

### More About

- “Use State Hierarchy to Design Multilevel State Complexity”



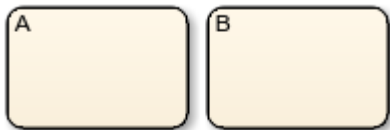
## Model Synchronous Subsystems by Using Parallel Decomposition

To implement operating modes that run concurrently, use parallel states 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.

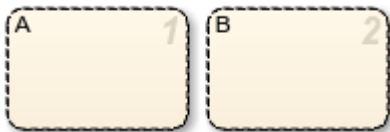
### State Decomposition

The decomposition type of a chart or state specifies whether the chart or state contains exclusive states or parallel states:

- Exclusive 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 charts represent each exclusive state by a solid rectangle.



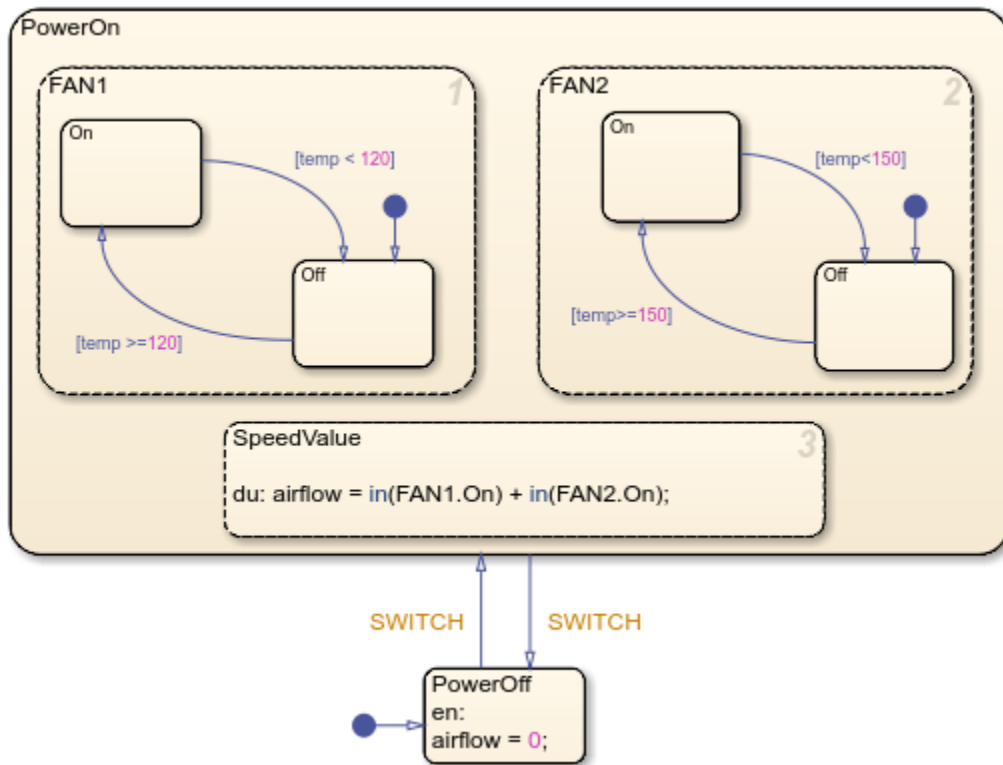
- Parallel states represent independent modes of operation. Two or more parallel states can be active at the same time, although they execute in a serial fashion. Stateflow charts represent each parallel state by a dashed rectangle with a number that indicates the execution order.



You can combine exclusive and parallel states in your Stateflow chart by setting the state decomposition at different levels of your state hierarchy. The default state decomposition type is **Exclusive (OR)**. To change the decomposition type to **AND (Parallel)**, right-click the parent state and select **Decomposition > AND (Parallel)**.

### Model an Air Temperature Controller

This example uses parallel decomposition to model an air controller that maintains air temperature at 120 degrees in a physical plant.



At the top level, the air controller chart has two exclusive states, `PowerOff` and `PowerOn`. The chart uses exclusive (OR) decomposition because controller cannot be on and off at the same time.

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 substates, `FAN1` and `FAN2`, of the top-level state `PowerOn`. Because the fans operate as independent components that turn on or off depending on how much cooling is required, `PowerOn` uses parallel (AND) decomposition to ensure that both substates are active when the controller is turned on.

Except for the operating thresholds, the fans are modeled by states with an identical configuration of substates and transitions that reflects the two modes of fan operation, `On` and `Off`. Because neither fan can be on and off at the same time, `FAN1` and `FAN2` have exclusive (OR) decomposition.

In `PowerOn`, a third parallel state called `SpeedValue` represents an independent subsystem that calculates the number of fans that 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. Similarly, 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.

### Specify 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:

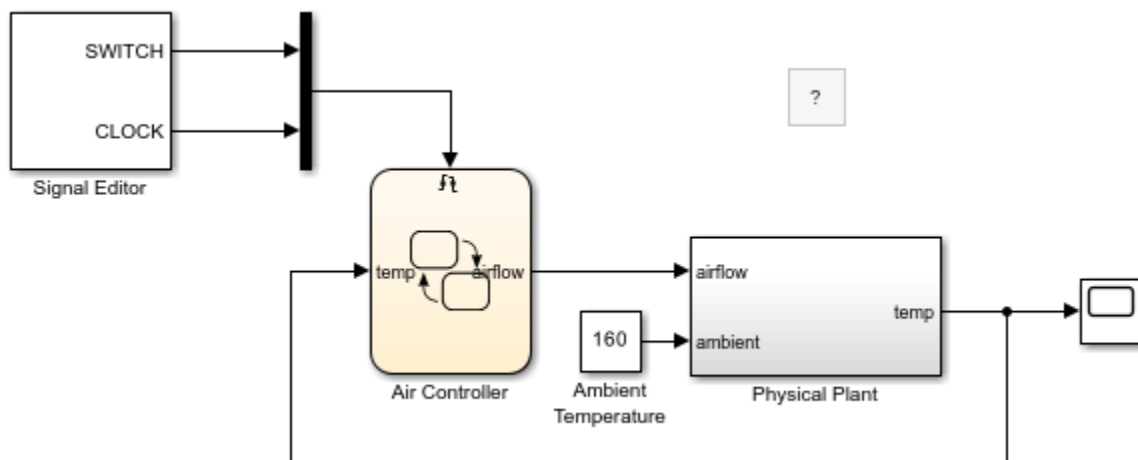
- `FAN1` executes first because it cycles on at a lower temperature than `FAN2`. It can turn on regardless of whether `FAN2` is on or off.

- 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 the order you add them to 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

This example contains a Stateflow chart called Air Controller and a Simulink® subsystem called Physical Plant.



Based on the air temperature of the physical plant, the chart turns on the fans and outputs the number of running fans, *airflow*, to the subsystem. This value determines the cooling activity factor,  $k_{Cool}$ , according to these rules:

- $airflow = 0$  — No fans are running. The air temperature does not decrease because  $k_{Cool} = 0$ .
- $airflow = 1$  — One fan is running. The air temperature decreases according to the cooling activity factor  $k_{Cool} = 0.05$ .
- $airflow = 2$  — Two fans are running. The air temperature decreases according to the cooling activity factor  $k_{Cool} = 0.1$ .

The Physical Plant subsystem updates the air temperature, *temp*, inside the plant based on the equations

$$temp(0) = T_{Initial}$$

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

where:

- $T_{Initial}$  is the initial temperature. The default value is 70°.
- $T_{Ambient}$  is the ambient temperature. The default value is 160°.

- $k_{\text{Heat}}$  is the heat transfer factor for the plant. The default value is 0.01.
- $k_{\text{Cool}}$  is the cooling activity factor that corresponds to `airflow`.

The new temperature determines the amount of cooling at the next time step of the simulation.

### See Also

#### More About

- “Define Exclusive and Parallel Modes by Using State Decomposition”
- “Execution Order for Parallel States”
- “Check State Activity by Using the `in` Operator”

## Synchronize Parallel States by Broadcasting Events

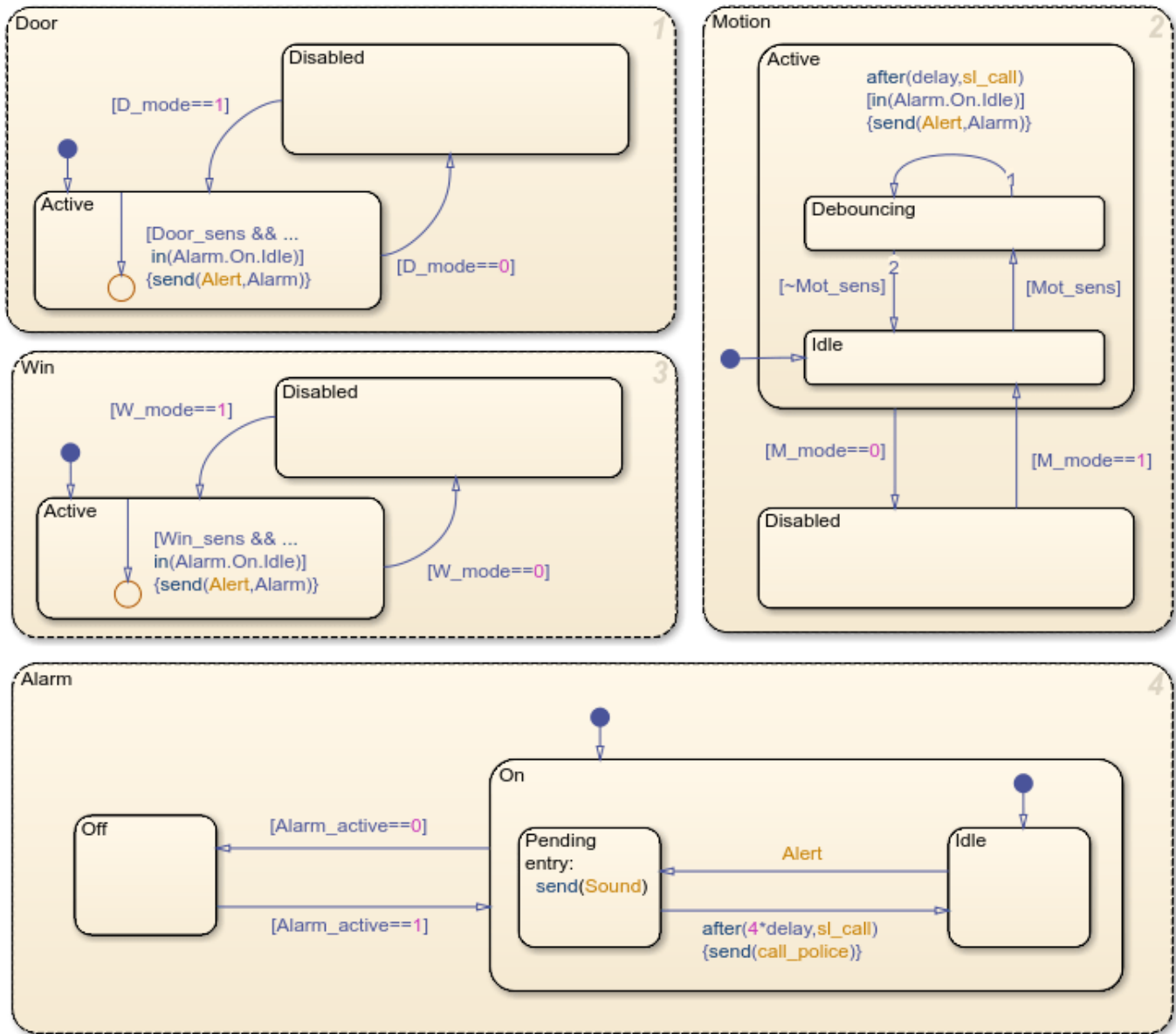
Local events enable you to coordinate parallel states by allowing one state to trigger a transition or an action in another state in the same Stateflow® chart. To broadcast an event from one state to another state, use the `send` operator with the name of the event and the name of an active state:

```
send(eventName, stateName)
```

When you broadcast an event, the event takes effect in the receiving state and in any substates in the hierarchy of that state.

### **Model a Home Security System**

This example 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 door sensor, a window 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 chart models each sensor by using one of these parallel states:

- The parallel state **Door** models the door sensor. The input signal **D\_mode** selects between the **Active** and **Disabled** modes for this sensor. When the sensor is active, the input signal **Door\_sens** indicates a possible intrusion.
- The parallel state **Win** models the window sensor. The input signal **W\_mode** selects between the **Active** and **Disabled** modes for this sensor. When the sensor is active, the input signal **Win\_sens** indicates a possible intrusion.

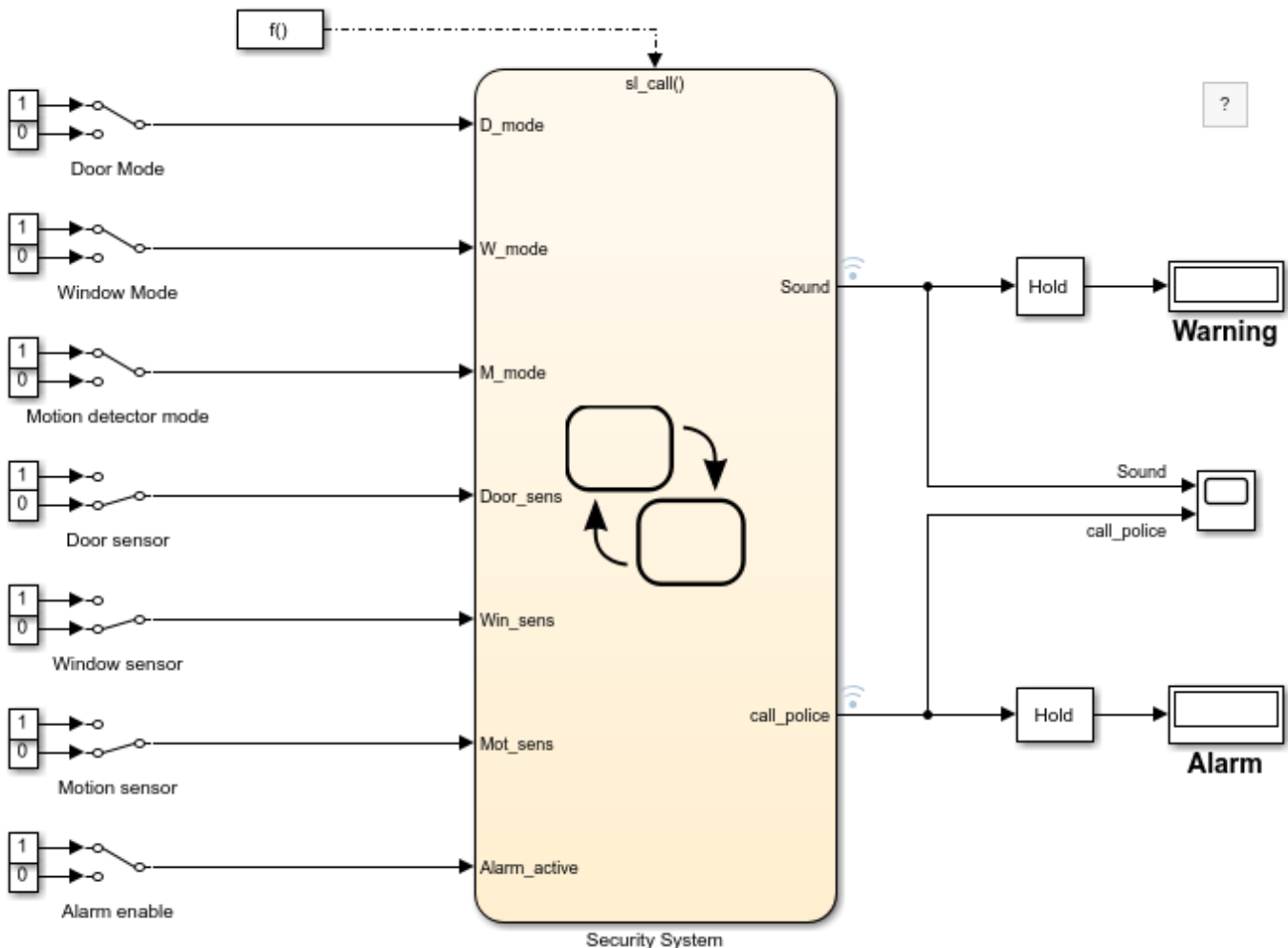
- The parallel state `Motion` models the motion detector. The input signal `M_mode` selects between the `Active` and `Disabled` modes for this sensor. When the sensor is active, the input signal `Mot_sens` indicates a possible intrusion.

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 alert. In contrast, the door and window sensors interpret a single positive trigger signal as an intrusion and issue an immediate alert.

A fourth parallel state called `Alarm` models the operating modes of the alarm system. The input signal `Alarm_active` selects between the `On` and `Off` modes for the alarm. If a sensor detects an intrusion while the alarm subsystem is on, the sensor broadcasts the local event `Alert` to the `Alarm` state. In the `On` substate of the state `Alarm`, the event triggers the transition from the `Idle` substate to the `Pending` substate. When `Pending` 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 before returning to the `Idle` mode.

### Coordinate with Other Simulink Blocks

Stateflow charts can also use events to communicate with other blocks in a Simulink® model.



## Output Events

An output event is an event that 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. For instance, in this example, the output events `Sound` and `call_police` drive external blocks that handle the warning sound and the call to the police. The chart broadcasts these events when the local event `Alert` triggers the transition to the `Pending` substate of the state `Alarm`. In particular, in the `Pending` substate, the entry action broadcasts the `Sound` event. Similarly, the condition action on the transition from `Pending` to `Idle` broadcasts the `call_police` event. In each case, the action that broadcasts the output event uses the `send` operator with the name of the event:

```
send(eventName)
```

Each output event maps to an output port on the chart. Depending on the configuration, the corresponding signal can control a Triggered Subsystem or a Function-Call Subsystem. To configure an output event:

- 1 In the **Modeling** tab, under **Design Data**, select **Symbols Pane** and **Property Inspector**.
- 2 In the **Symbols** pane, select the output event.
- 3 In the **Property Inspector**, set **Trigger** to one of these options:
  - `Either edge` — The output event broadcast causes the outgoing signal to toggle between zero and one.
  - `Function call` — The output event broadcast causes a Simulink function-call event.

In this example, the output events 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 is an event that 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. For instance, in this example, 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 triggers a transition after the chart receives the event some number of times.

An external Simulink block sends an input event through a signal connected to the trigger port on the Stateflow chart. Depending on the 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:

- 1 In the **Modeling** tab, under **Design Data**, select **Symbols Pane** and **Property Inspector**.
- 2 In the **Symbols** pane, select the input event.
- 3 In the **Property Inspector**, set **Trigger** to one of these options:
  - `Rising` — The chart activates when the input signal changes from either zero or a negative value to a positive value.
  - `Falling` — The chart activates when the input signal changes from a positive value to either zero or a negative value.
  - `Either` — The chart activates when the input signal crosses zero as it changes in either direction.



- **Function call** — The chart activates with a function call from a Simulink block.

In this 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

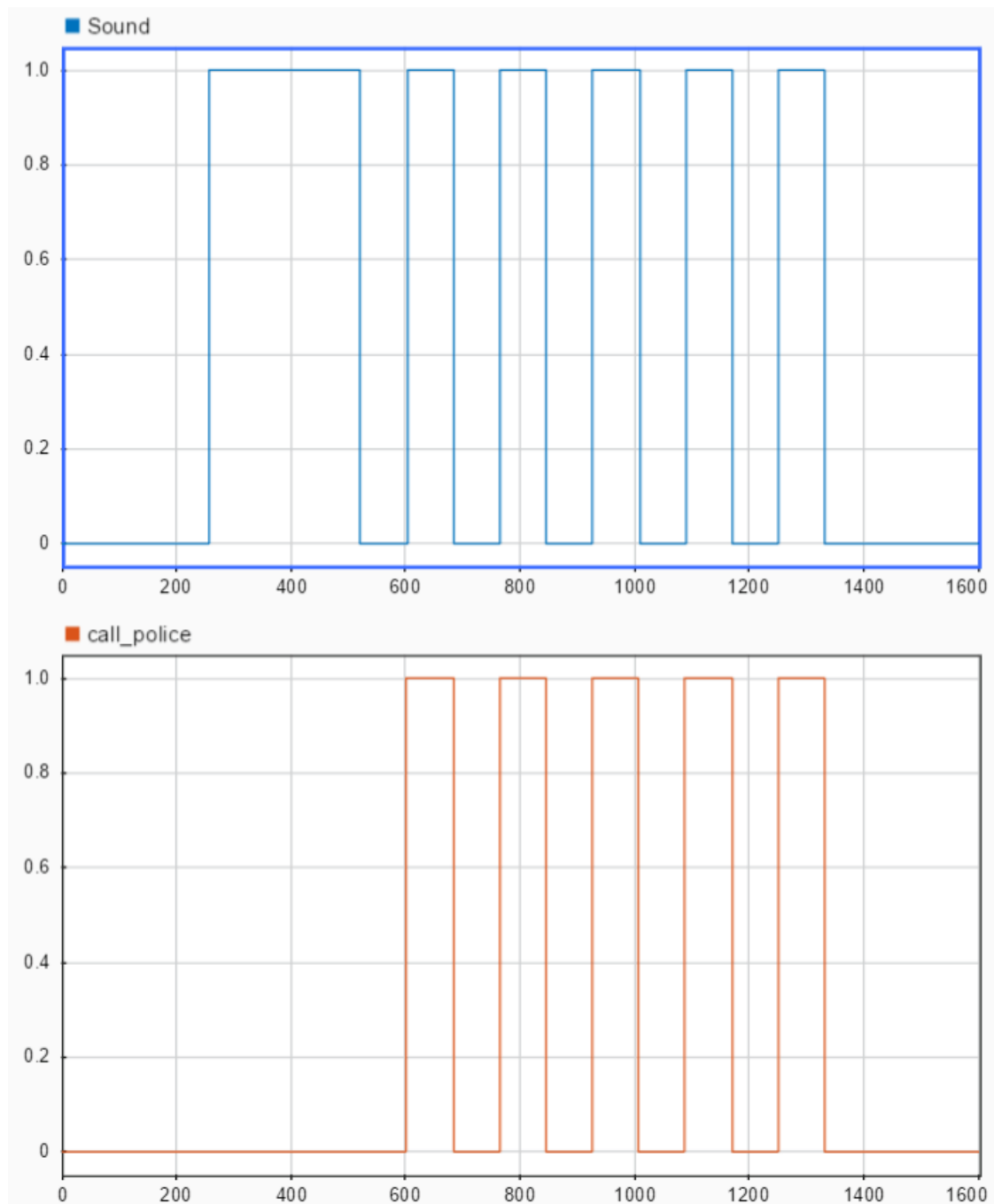
In this example, the Stateflow chart has inputs from several Manual Switch blocks and outputs to a pair of latch subsystems that connect to Display blocks. During simulation, you can:

- 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.

For example, suppose that you switch the alarm and sensor subsystems on, switch the sensor triggers off, and start the simulation. 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 the door sensor trigger off 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.



### See Also

send

### More About

- “Model Synchronous Subsystems by Using Parallel Decomposition” on page 2-25
- “Activate a Stateflow Chart by Sending Input Events”

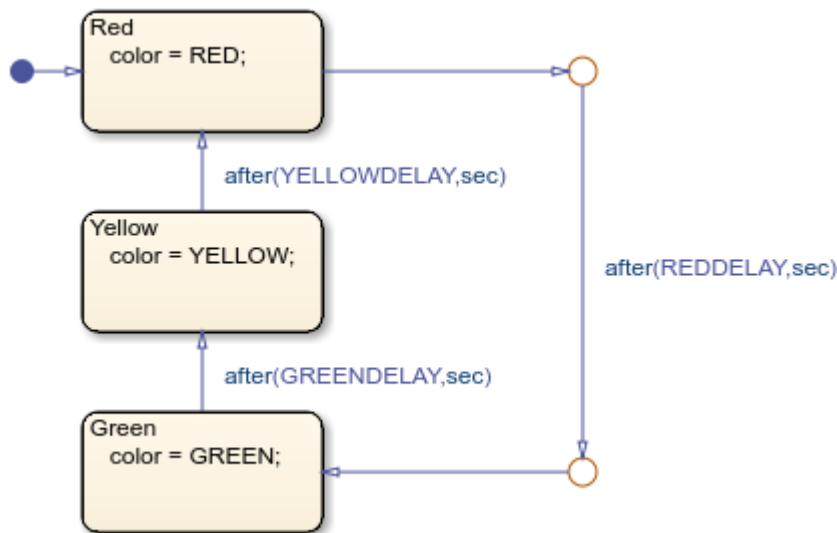
- “Activate a Simulink Block by Sending Output Events”

## Monitor Chart Activity by Using Active State Data

If your Stateflow® chart includes data that is 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 example, in this model of a traffic signal, the state that is active determines the value of the output signal `color`. You can simplify the design of the chart by enabling active state data. In this case, the Stateflow chart can provide the color of the traffic signal by tracking state activity, so you do not have to explicitly update the value of `color`.



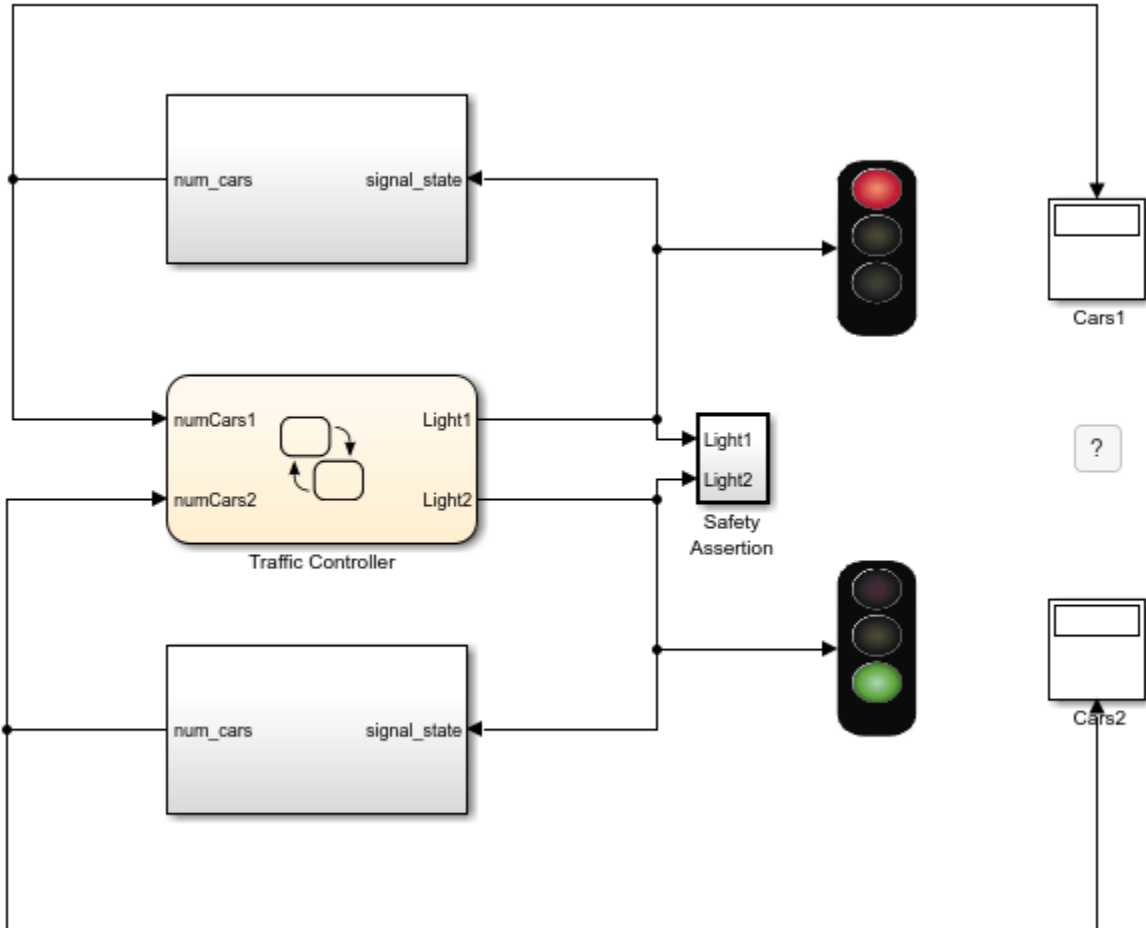
To enable active state data, select a state to monitor. Then, in the **Property Inspector**:

1. Select **Create output for monitoring**.
2. Select one of these activity types:
  - **Self activity** — Boolean value that indicates whether the state is active
  - **Child activity** — Enumerated value that indicates which child state is active
  - **Leaf state activity** — Enumerated value that indicates which leaf state is active
3. Enter the **Data name** for the active state data symbol.
4. Optionally, for **Child activity** or **Leaf state activity**, enter the **Enum name** for the active state data type.

By default, Stateflow charts report state activity as output data to the Simulink model. To change the scope of an active state data symbol to local data, use the **Symbols** pane.

## Model a Traffic Signal Controller

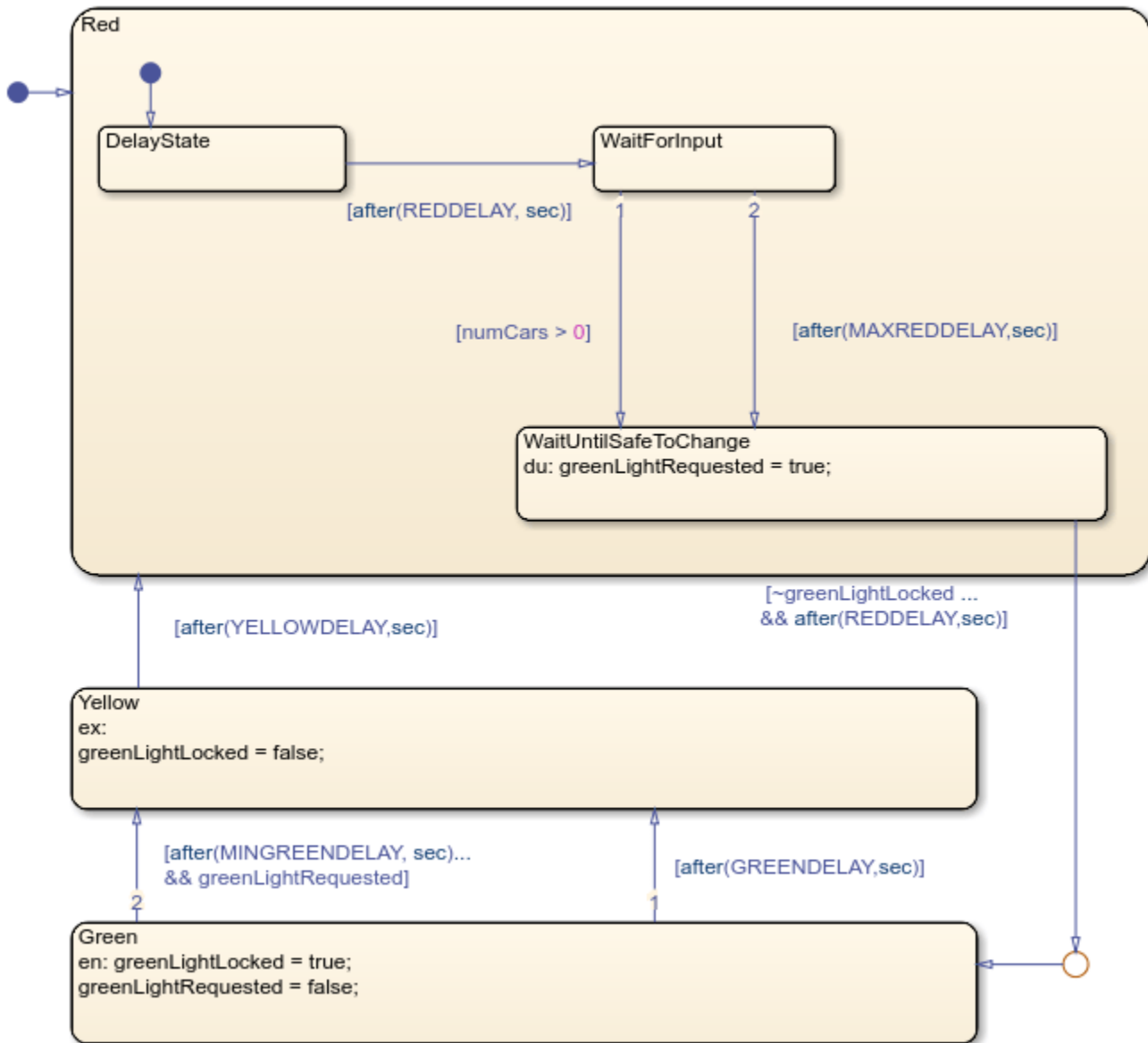
This example uses active state data to model the controller system for a pair of traffic lights.



Inside the **Traffic Controller** chart, two parallel subcharts manage the logic that controls the traffic lights. The subcharts have an identical hierarchy that consists 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 that verifies that the two traffic lights are never green simultaneously.

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



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

The traffic light cycle begins when the Red state becomes active. 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` to prevent the other traffic signal from turning green. After a short delay, 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

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

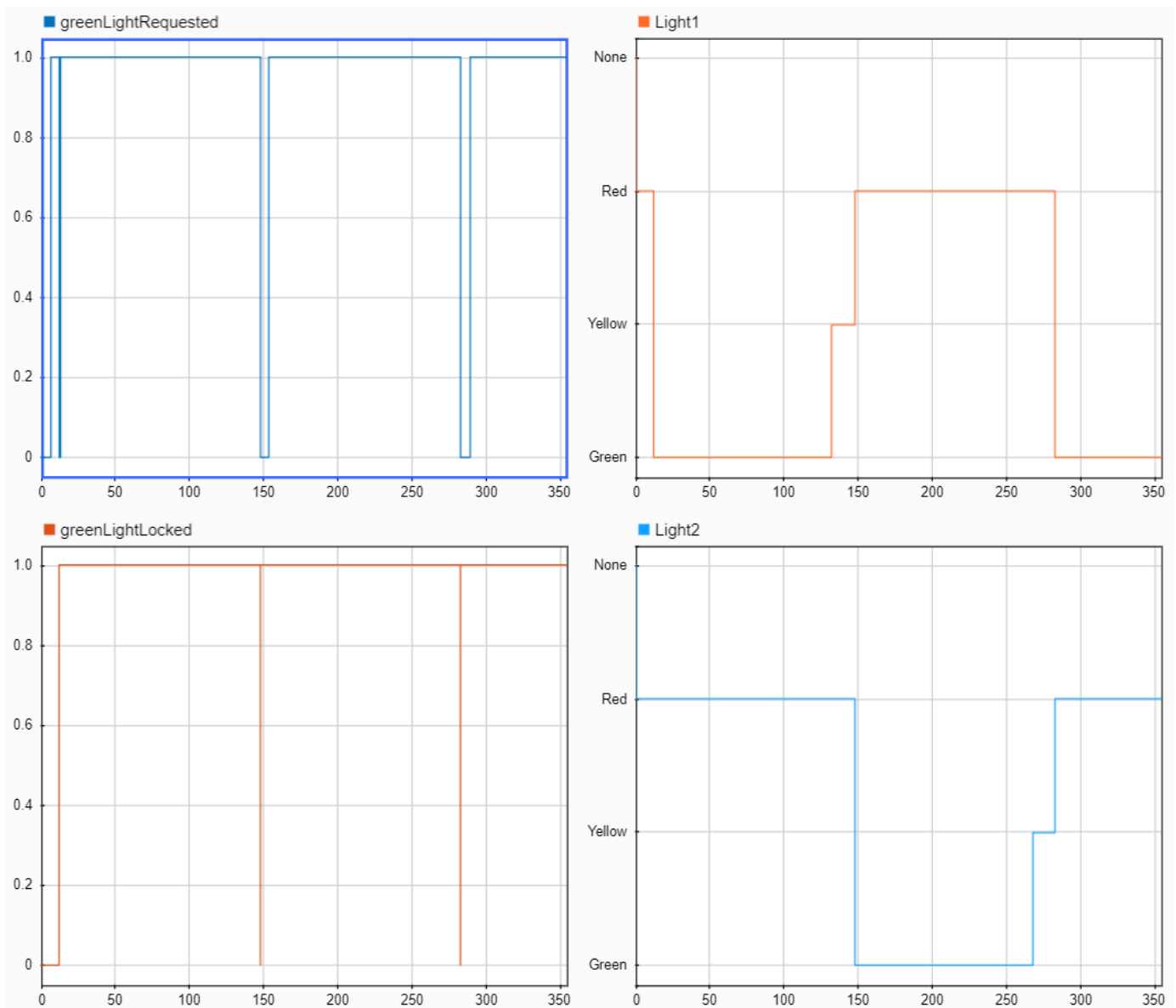
### Timing of Traffic Lights

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

- **REDDELAY** — Length of time before the controller checks for cars at the intersection. This value is also the minimum length of time before the traffic light can turn green after the controller requests a green light. The default value is 6 seconds.
- **MAXREDDelay** — Maximum length of time that the controller checks for cars before requesting a green light. The default value is 360 seconds.
- **GREENDELAY** — Maximum length of time that the traffic light remains green. The default value is 180 seconds.
- **MINGREENDELAY** — Minimum length of time that the traffic light remains green. The default value is 120 seconds.
- **YELLOWDELAY** — Length of time that the traffic light remains yellow. The default value is 15 seconds.

### Explore the Example

- 1 Open the chart by clicking the arrow in the bottom-left corner.
- 2 In the **Symbols** pane, select `greenLightRequested`. Then, in the **Property Inspector**, under **Logging**, select **Log signal data**.
- 3 Repeat the previous step for `greenLightLocked`, `Light1`, and `Light2`.
- 4 In the **Simulation** tab, click **Run**.
- 5 In the **Simulation** tab, under **Review Results**, click **Data Inspector**.
- 6 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 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, these are the key moments during the first 300 seconds of the simulation:

- $t = 0$  — At the start of the simulation, both traffic lights are red. Light1 and Light2 are Red, greenLightRequested is false, and greenLightLocked is false.
- $t = 6$  — After 6 seconds, the default value of REDDELAY, there are cars waiting in both streets, so both traffic lights request a green light. Light1 and Light2 are still Red, greenLightRequested is true, and greenLightLocked is false.
- $t = 12$  — After another 6 seconds, the default value of REDDELAY, Light 1 becomes green, cancels the green light request, and sets greenLightLocked to true. Then, Light 2 requests a green light. Light1 is Green, Light2 is Red, greenLightRequested becomes false and then true, and greenLightLocked is true.



- $t = 132$  — After 120 seconds, the default value of MINGREENDELAY, Light 1 turns yellow. Light1 is Yellow, Light2 is Red, greenLightRequested is true, and greenLightLocked is true.
- $t = 147$  — After 15 seconds, the default value of YELLOWDELAY, Light 1 turns red and sets greenLightLocked to false. Then, Light 2 turns green, cancels the green light request, and sets greenLightLocked to true. Light1 is Red, Light2 is Green, greenLightRequested is false, and greenLightLocked becomes false and then true.
- $t = 153$  — After 6 seconds, the default value of REDDELAY, Light 1 requests a green light. Light1 is Red, Light2 is Green, greenLightRequested is true, and greenLightLocked is true.
- $t = 267$  — After Light 2 is green for 120 seconds, the default value of MINGREENDELAY, Light 2 turns yellow. Light1 is Red, Light2 is Yellow, greenLightRequested is true, and greenLightLocked is true.
- $t = 282$  — After 15 seconds, the default value of YELLOWDELAY, Light 2 turns red and sets greenLightLocked to false. Then, Light 1 turns green, cancels the green light request, and sets greenLightLocked to true. Light1 is Green, Light2 is Red, greenLightRequested is false, and greenLightLocked becomes false and then true.
- $t = 288$  — After 6 seconds, the default value of REDDELAY, Light 2 requests a green light. Light1 is Green, Light2 is Red, greenLightRequested is true, and greenLightLocked is true.

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

## See Also

### More About

- “Create a Hierarchy to Manage System Complexity” on page 2-21
- “Monitor State Activity Through Active State Data”
- “Simplify Stateflow Charts by Incorporating Active State Output”
- “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 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:

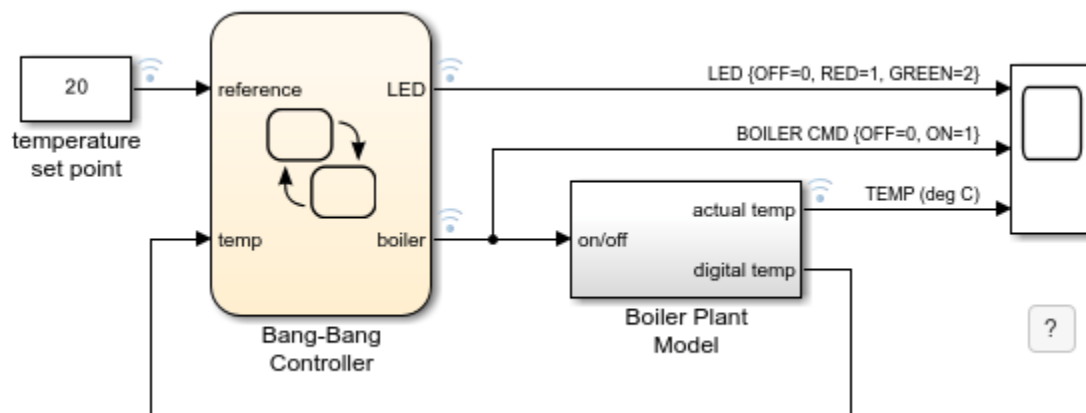
- Transitions between states
- Function calls
- Changes in variable values

These are the most common operators for absolute-time temporal logic:

- **after** — `after(n, sec)` returns `true` if `n` seconds of simulation time have elapsed since the state that contains the operator or the source state of the transition that contains the operator became active. Otherwise, the operator returns `false`. This operator supports event-based temporal logic and absolute-time temporal logic in seconds (`sec`), milliseconds (`msec`), and microseconds (`usec`).
- **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.

### Model a Bang-Bang Temperature Controller

This example 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 Model subsystem models the dynamics inside the boiler by increasing or decreasing its temperature according to the status of the controller. Then, the chart uses the boiler temperature for the next step in the simulation.

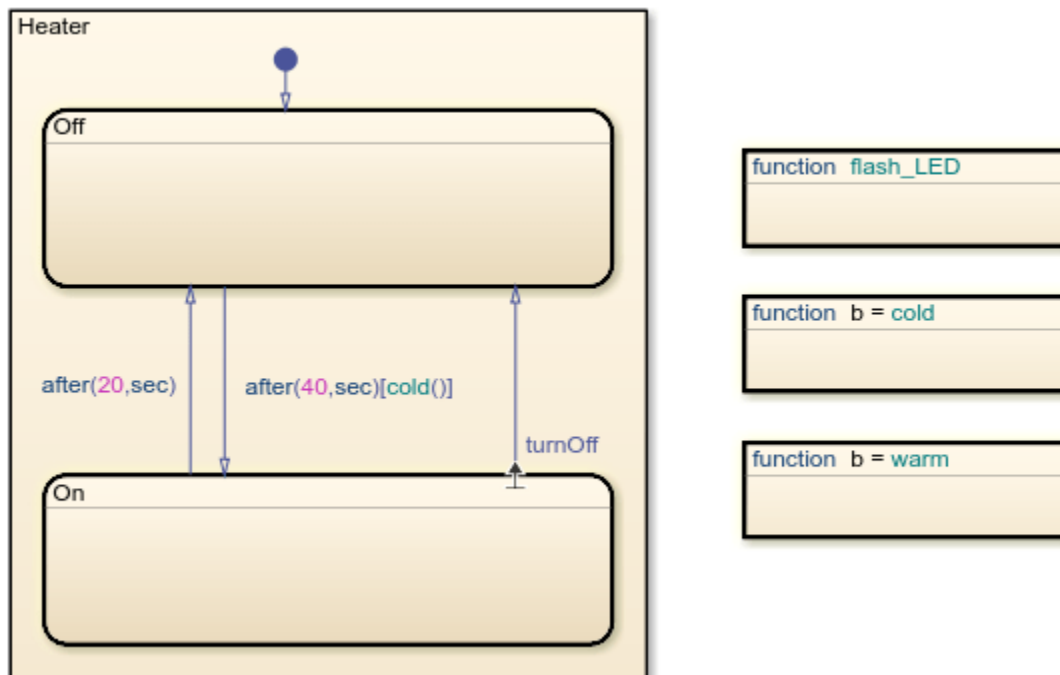
The Bang-Bang Controller chart uses the temporal logic operator `after` to:

- 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

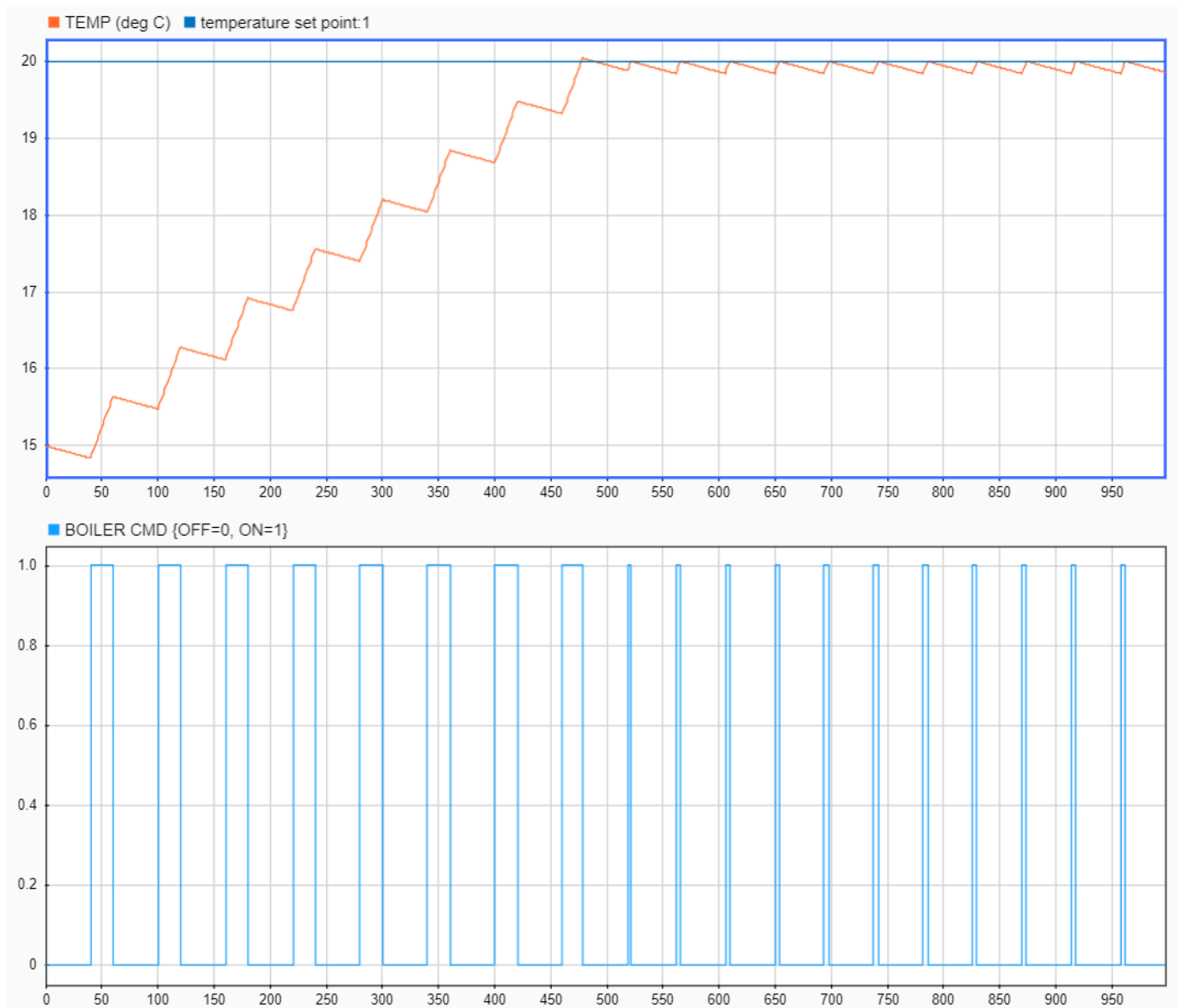
The Bang-Bang Controller chart contains a pair of substates that represent the two operating modes of the boiler, On and Off. The chart uses the active state output data `boiler` to indicate which substate is active.



The conditions on the transitions between the On and Off substates define the behavior of the bang-bang controller:

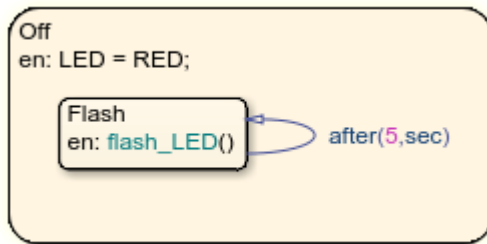
- On the first transition from On to Off, the condition `after(20,sec)` turns the boiler off after it is on for 20 seconds.
- On the transition from Off to On, the condition `after(40,sec)[cold()]` turns the boiler on when the graphical function `cold()` indicates that the boiler temperature is below the reference set point for at least 40 seconds.
- On the second transition from On to Off, the trivial condition turns the boiler off when the internal transition logic in the On state determines that the boiler temperature is at or above the reference set point.

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 set point. From that point, 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.

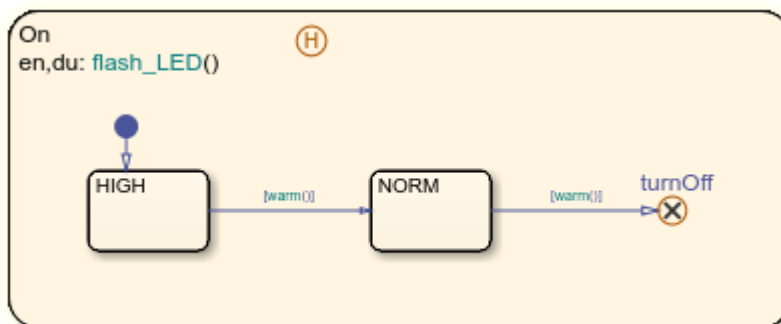


### Timing of Status LED

The `Off` state contains a substate `Flash` with a self-loop transition triggered 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.

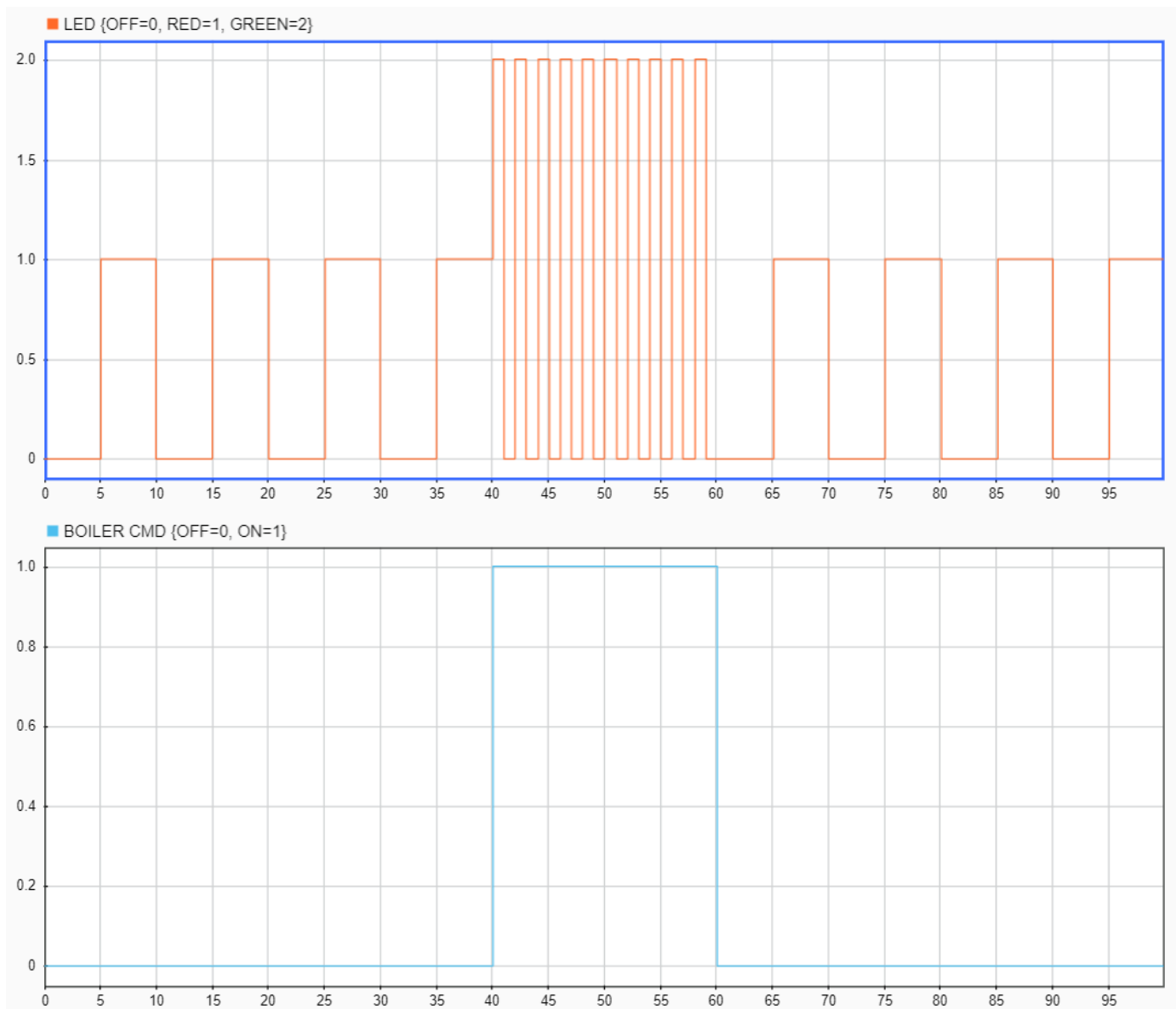


The On state calls the graphical function `flash_LED` as a combined entry, during state action. When the On state is active, this action calls the function at every time step of the simulation to toggle the value of the output symbol LED between 0 and 2.



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 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 `elapsed` and `duration` operators:

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

2. In the **Symbols** pane, click **Resolve Undefined Symbols**. The Stateflow Editor resolves the symbols Timer1 and Timer2 as output data.

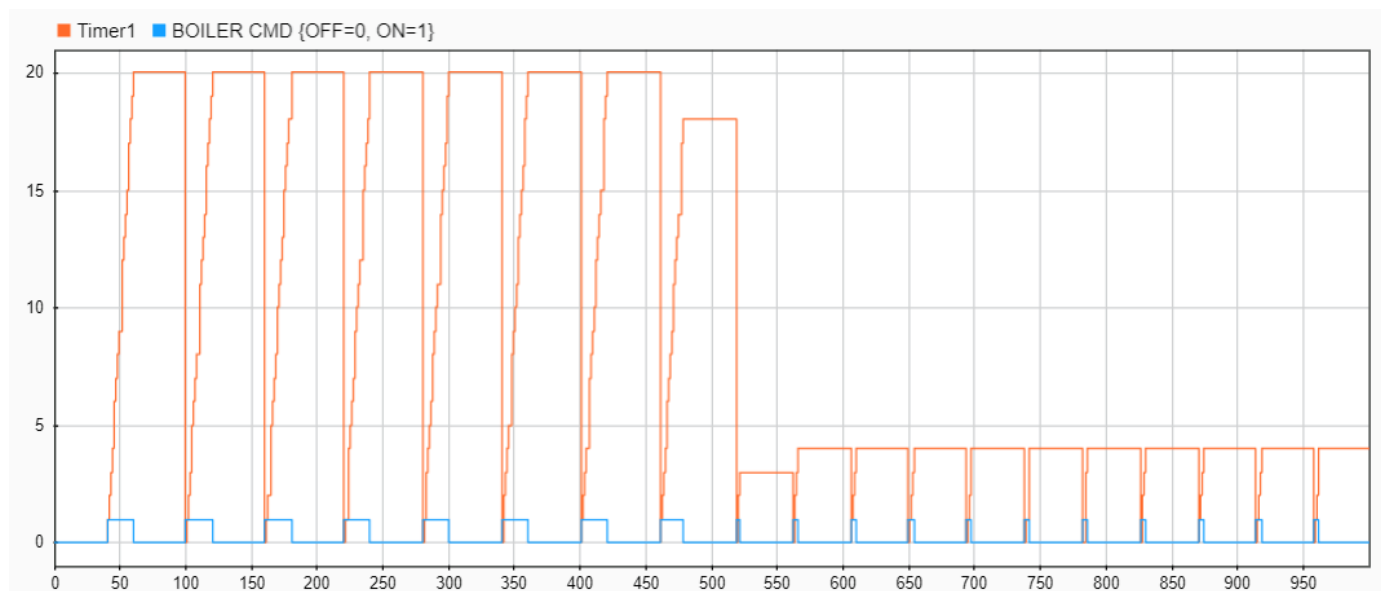
3. Enable logging for Timer1 and Timer2. In the **Symbols** pane, select each symbol. Then, in the **Property Inspector**, under **Logging**, select **Log signal data**.

4. In the **Simulation** tab, click **Run**.

5. In the **Simulation** tab, under **Review Results**, click **Data Inspector**.

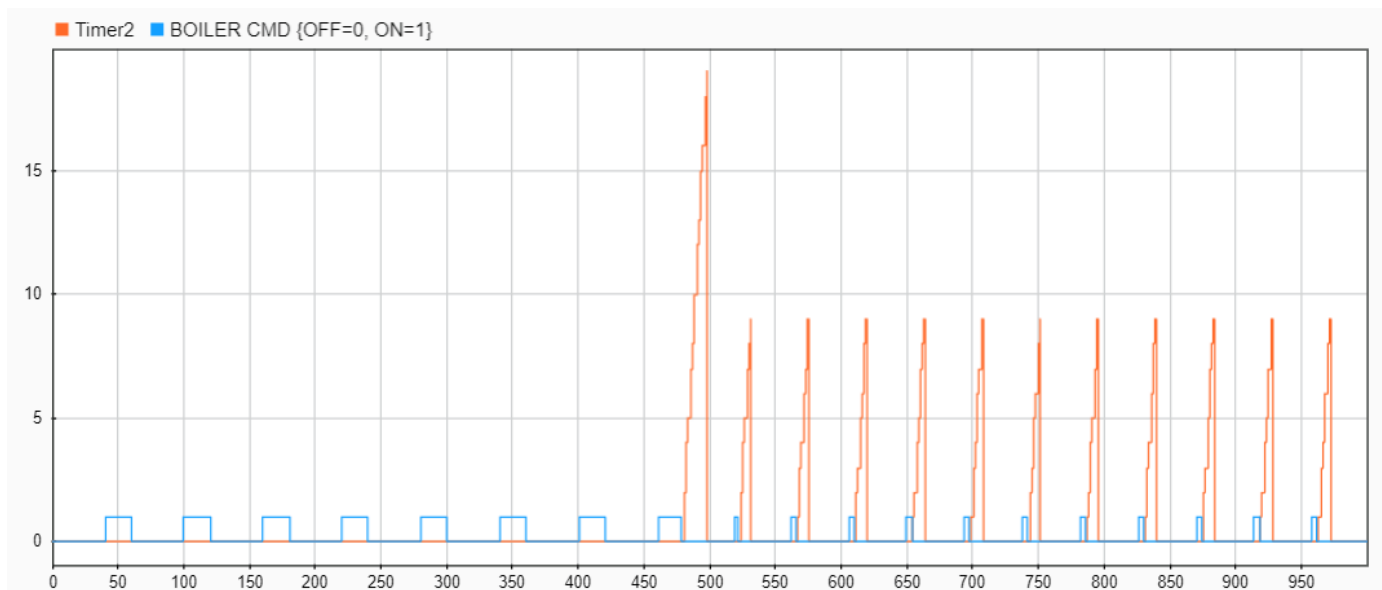
6. In the Simulation Data Inspector, display the signals boiler and Timer1 in the same set of axes. The plot shows that:

- 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.



7. 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 19 seconds to cool, which is more than twice as long as the other cycles.



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, which results in a warmer-than-normal boiler. In later cycles, the history junction causes the On phase to start with an active NORM substate. The controller then turns off immediately after the boiler reaches the reference temperature, which results in a cooler boiler.

### See Also

#### More About

- “Define Chart Behavior by Using State and Transition Actions” on page 2-17
- “Control Chart Execution by Using Temporal Logic”
- “Reuse Logic Patterns by Defining Graphical Functions”
- “Resume Prior Substate Activity by Using History Junctions”



# Additional Learning Tools

---

# Stateflow Onramp

Free, self-paced, interactive Stateflow course

## Description

Stateflow Onramp is a free, self-paced, interactive course that helps you get started with Stateflow.

After completing Stateflow Onramp, you will be able to use the Stateflow environment and build Stateflow charts based on real-life examples.

Stateflow Onramp teaches you to:

- Model state machines with Stateflow charts.
- Use Stateflow symbols and data.
- Control chart execution with actions.
- Simulate Stateflow charts with Simulink.
- Create flow charts to model common logic patterns.
- Improve chart readability and reuse code with functions.
- Organize charts using hierarchy.

Stateflow Onramp uses tasks to teach concepts incrementally, such as through a real-life example with a robotic vacuum. You receive automated assessments and feedback after submitting tasks. Your progress is saved when you exit the course, so you can complete the course in multiple sessions.

The screenshot displays the Stateflow Onramp interface. On the left, a task description for 'Task 1' explains the traffic light model. The main area shows a Stateflow chart with two modes: 'Normal' and 'Fault'. The 'Normal' mode includes states for 'Stop', 'PrepareToStop', and 'Go', with transitions based on timers. The 'Fault' mode includes states for 'BlinkOn' and 'BlinkOff', triggered by a fault condition. On the right, a 'Symbols' table lists the variables used in the chart.

TYPE	NAME	VALUE	PORT
	red	1	1
	yellow		2
	fault		1
	green	0	3

Below the symbols table, a 'Training - Assessment' section shows a green checkmark and the text: 'Requirements: Are the chart states, transitions, and Symbols all correct?'.

## Open the Stateflow Onramp

- Simulink Start Page: On the **Learn** tab, click the **Launch** button that appears when you pause on **Stateflow Onramp**.

- Stateflow chart: On the quick access toolbar, click **Help > Learn Stateflow**.
- MATLAB Command Window: Enter `learning.simulink.launchOnramp("stateflow")`.

---

**Note** If you do not have a Stateflow license, you can take the course at Self-Paced Online Courses.

---

## Version History

Introduced in R2019b

### See Also

`learning.simulink.launchOnramp`

### Topics

“Model Finite State Machines” on page 2-2

“Construct and Run a Stateflow Chart” on page 2-10

“Define Chart Behavior by Using State and Transition Actions” on page 2-17

“Create a Hierarchy to Manage System Complexity” on page 2-21

