Stateflow®

Getting Started Guide

MATLAB&SIMULINK®

MathWorks®

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

Revision History

Contents

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
- "Installing Stateflow Software" on page 1-3

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.

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 1-3 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 1-3.

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**.

2

Stateflow Fundamentals

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

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.
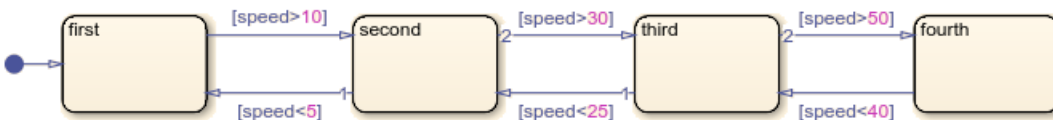
The arrow on the left of the diagram represents the default transition and indicates the first state to become active. When you execute the chart, this state is highlighted on the canvas. The other arrows indicate the possible transitions between the states. To define the dynamics of the state machine, 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.
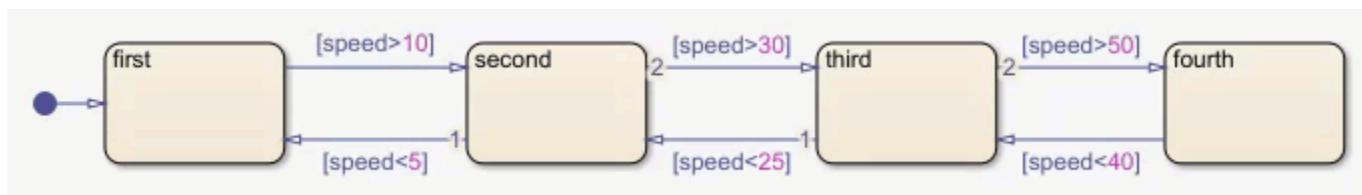
This chart offers a simple design that disregards important factors such as engine speed and torque. You can construct a more comprehensive and realistic model by linking this Stateflow chart with other components in MATLAB or Simulink. Following are three possible approaches.

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.
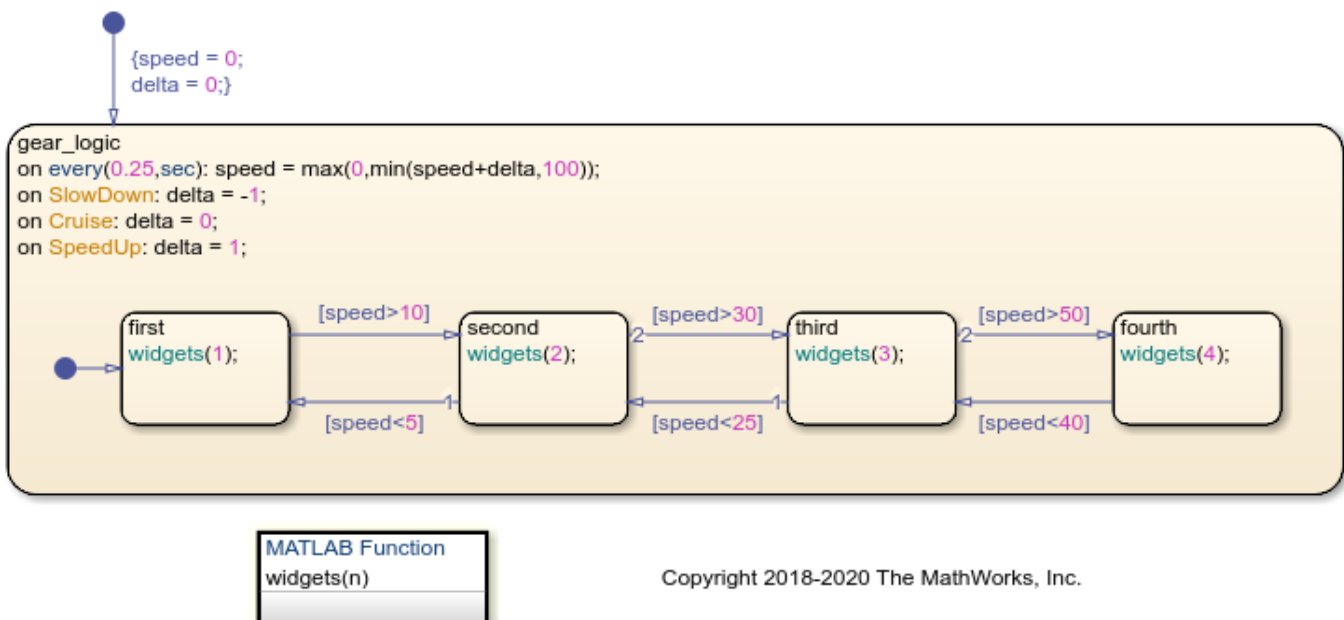
You can execute this chart as an object in MATLAB directly through the Command Window or by using a script. You can also program a MATLAB app that controls the state of the chart through a graphical user interface. For example, this user interface sends an input event to the chart when you click a button. In the chart, the MATLAB function `widgets` controls the values of the gauges and lamps on the interface.
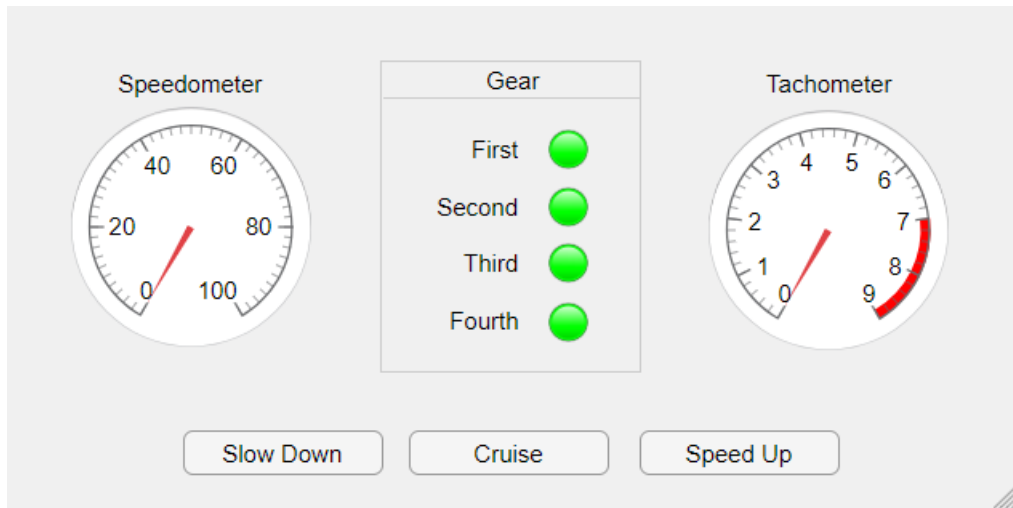
The chart continues to run until you close the user interface window. For more information about executing Stateflow charts as MATLAB objects, see "Execution in MATLAB".

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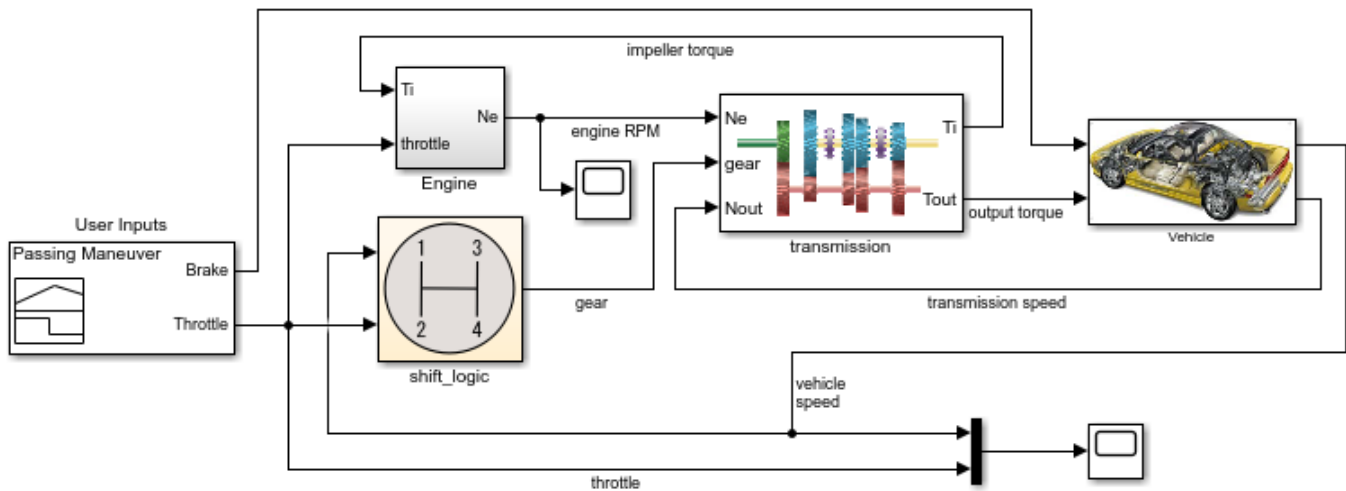
This chart combines state hierarchy, parallelism, active state data, local events, and temporal logic.

- **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.
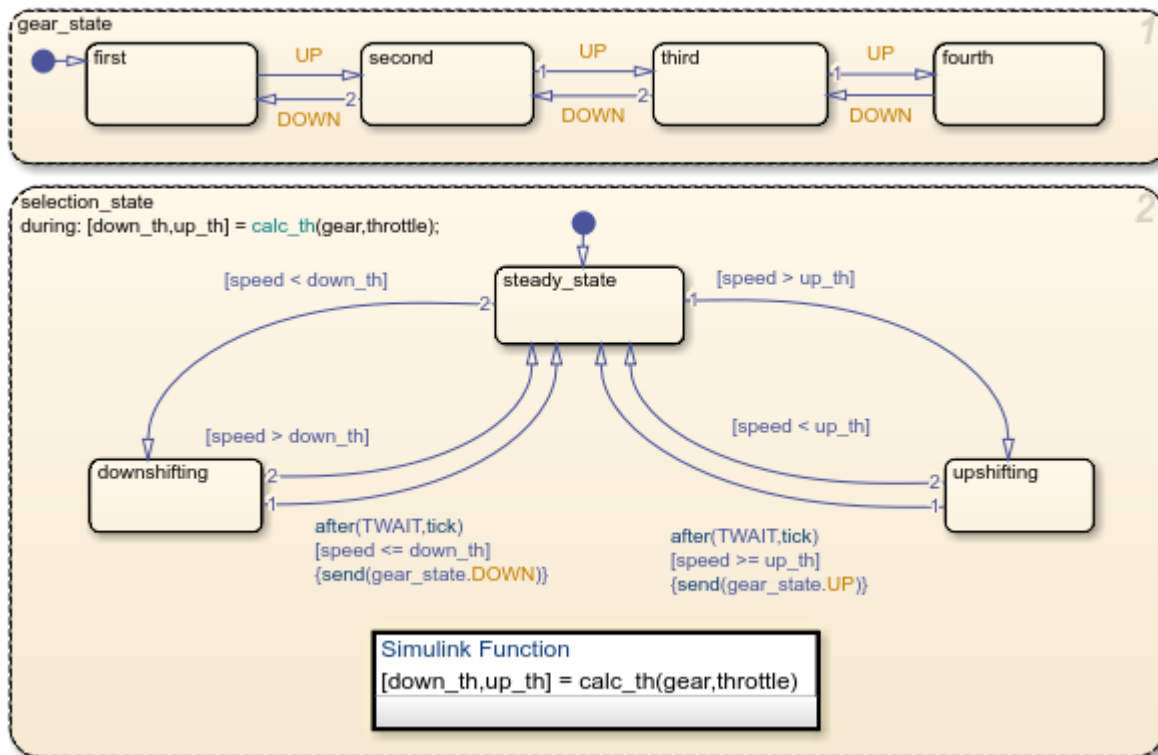
To run a simulation of the model:

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 **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.
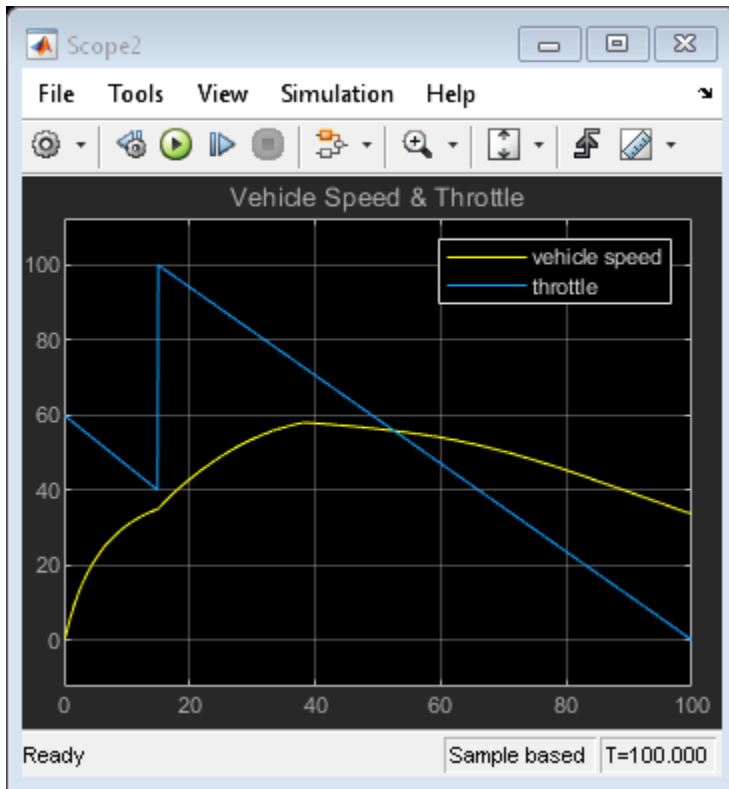
Simulate Chart as a Simulink Block With Temporal Conditions

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 connections. To open the chart, click the arrow in the bottom left corner of the `Gear_logic` block.
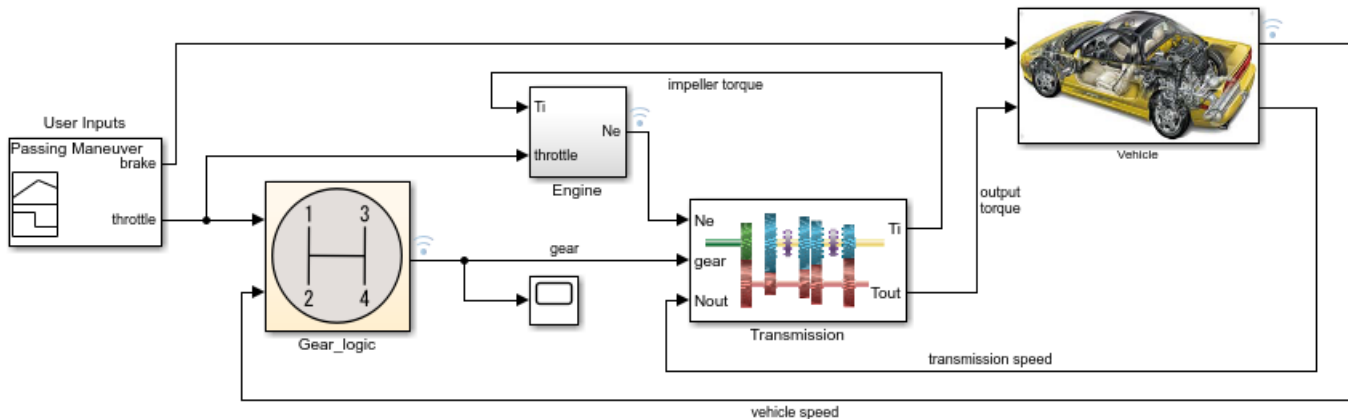
This chart combines state hierarchy, active state data, and temporal logic.

- **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).
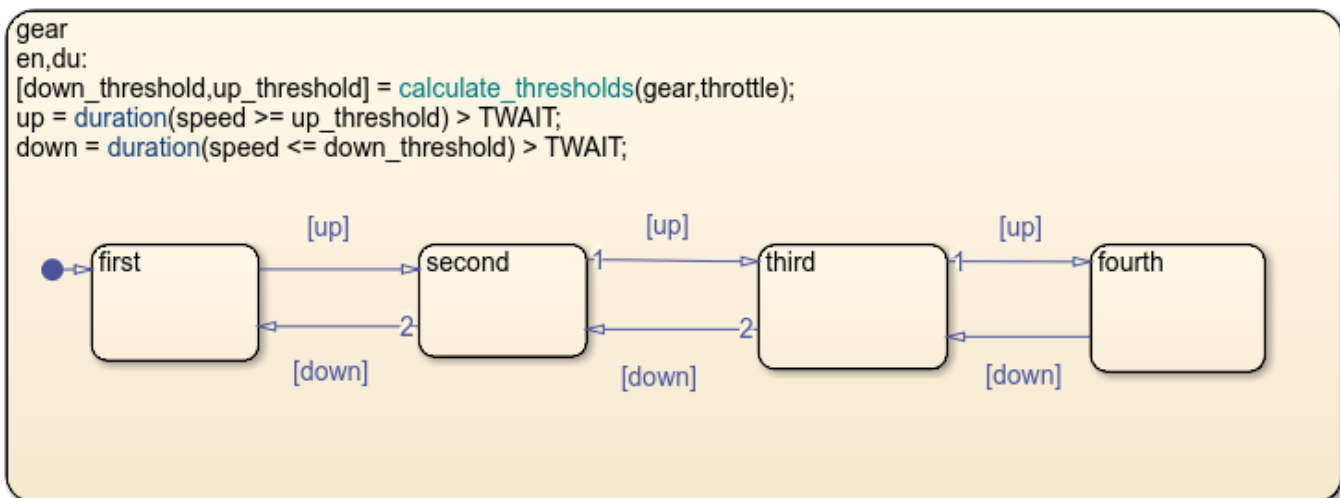
To run a simulation of the model:

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 **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.
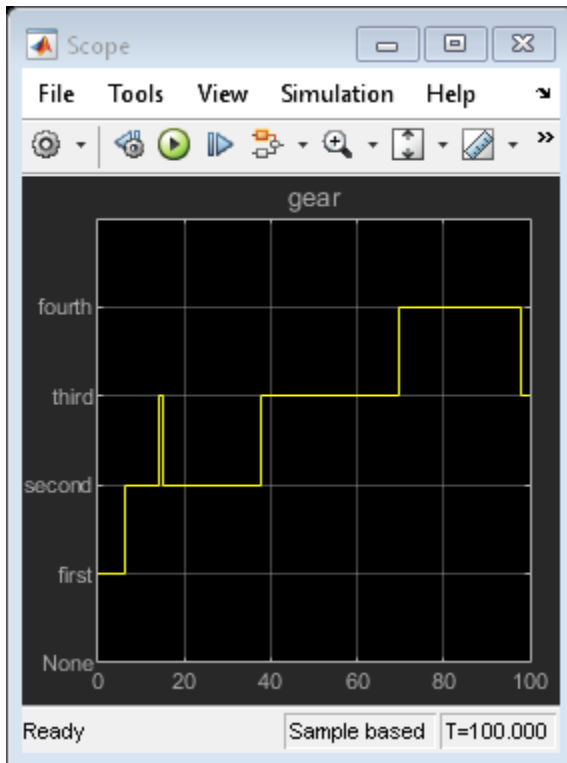
See Also
after | duration | every | send

More About

- "Construct and Run a Stateflow Chart" on page 2-9
- "Define Chart Behavior by Using Actions" on page 2-18
- "Create a Hierarchy to Manage System Complexity" on page 2-23
- "Model Synchronous Subsystems by Using Parallelism" on page 2-29
- "Synchronize Parallel States by Broadcasting Events" on page 2-33
- "Monitor Chart Activity by Using Active State Data" on page 2-40
- "Schedule Chart Actions by Using Temporal Logic" on page 2-47

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 2-2.

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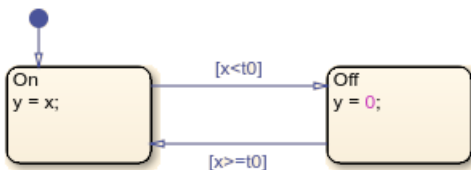
This example shows how to create this Stateflow chart for simulation in Simulink and execution in MATLAB.

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. 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, 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, 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 object palette, the chart canvas, 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 to the chart and resolve any undefined or unused symbols.
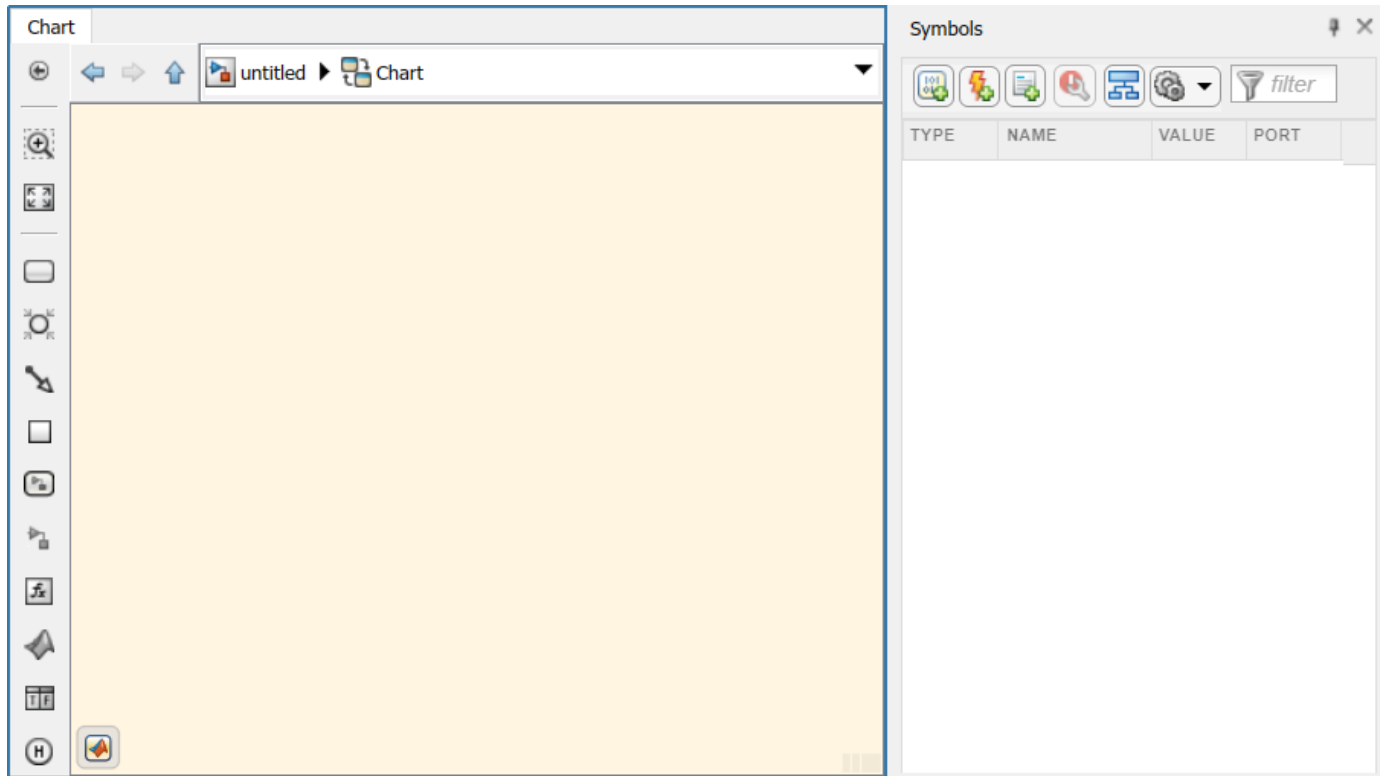
Tip After you construct your Stateflow chart, you can copy its contents to another chart with a different execution mode. For example, you can construct a chart for execution in MATLAB and copy its contents into a chart for simulation in Simulink.

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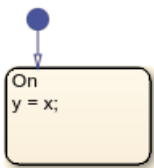
2   Add another state. Right-click and drag the On state. Blue graphical cues help you to align your states horizontally or vertically. The name of the new state changes to Off. Double-click the state and modify the state action to y = 0.

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<t0 or x>=t0. The conditions appear inside square brackets.
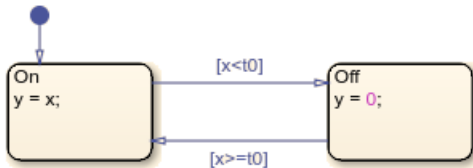
5 Clean up the chart:

- 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, 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 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 🔢.
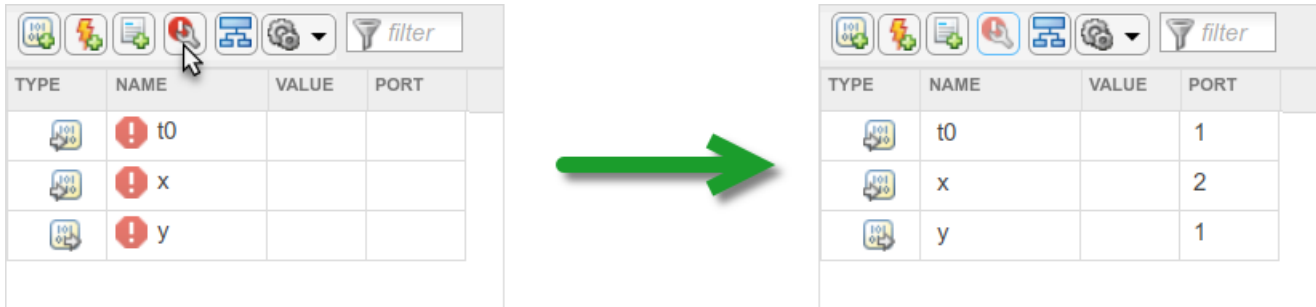
3 Because the threshold t0 does not change during simulation, change its scope to constant data.

In the **Type** column, click the data type icon next to t0 and select 🔢 Constant Data.

4 Set the value for the threshold t0. In the **Value** column, click the blank entry next to t0 and enter a value of 0.

5 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 2-14.

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.
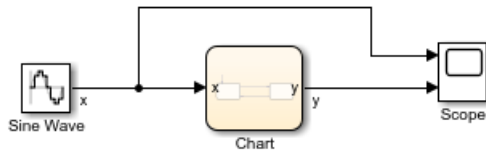
5

To simulate the model, click **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.
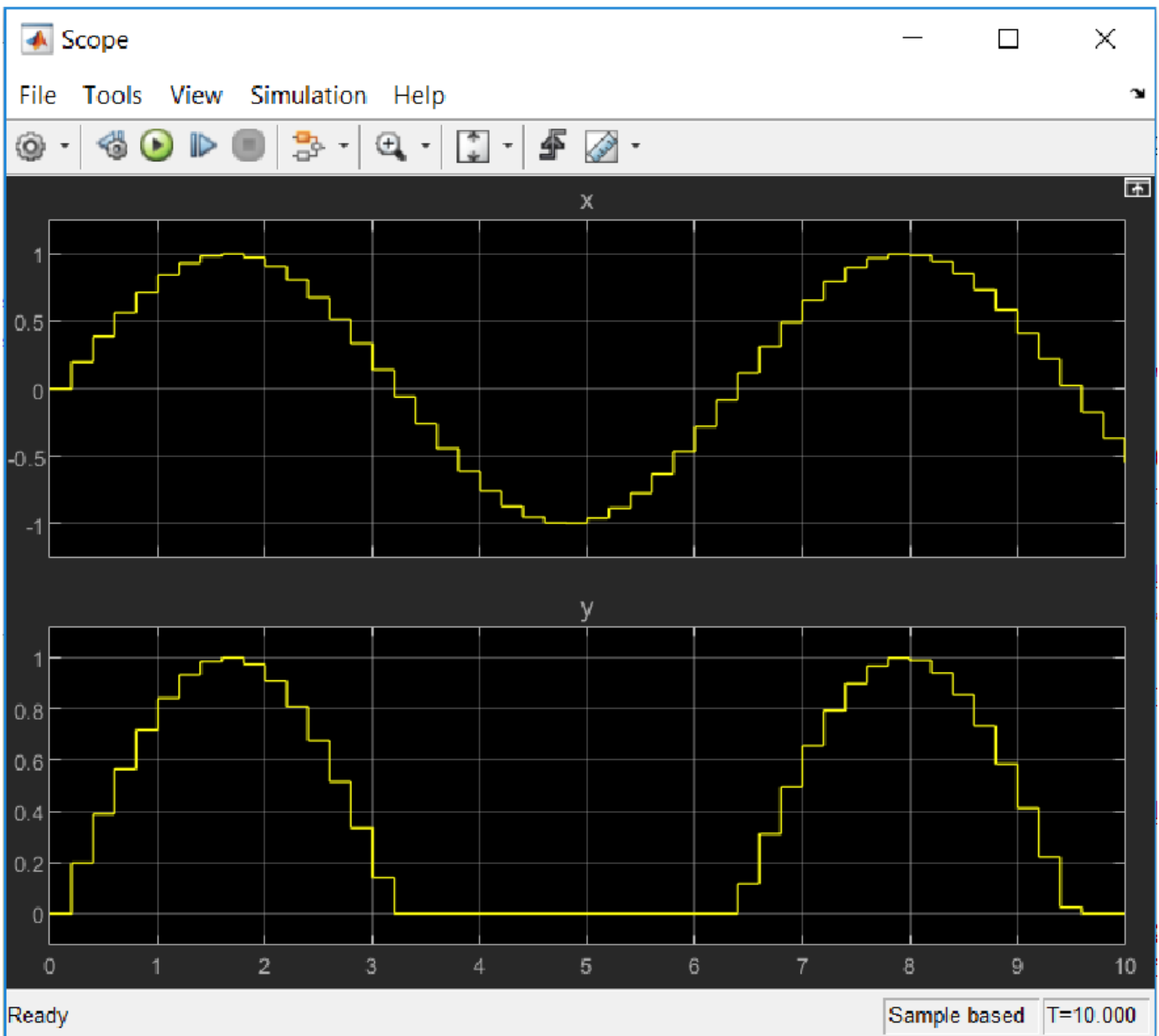
The simulation results show that the rectifier filters out negative input values.

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 2-12.

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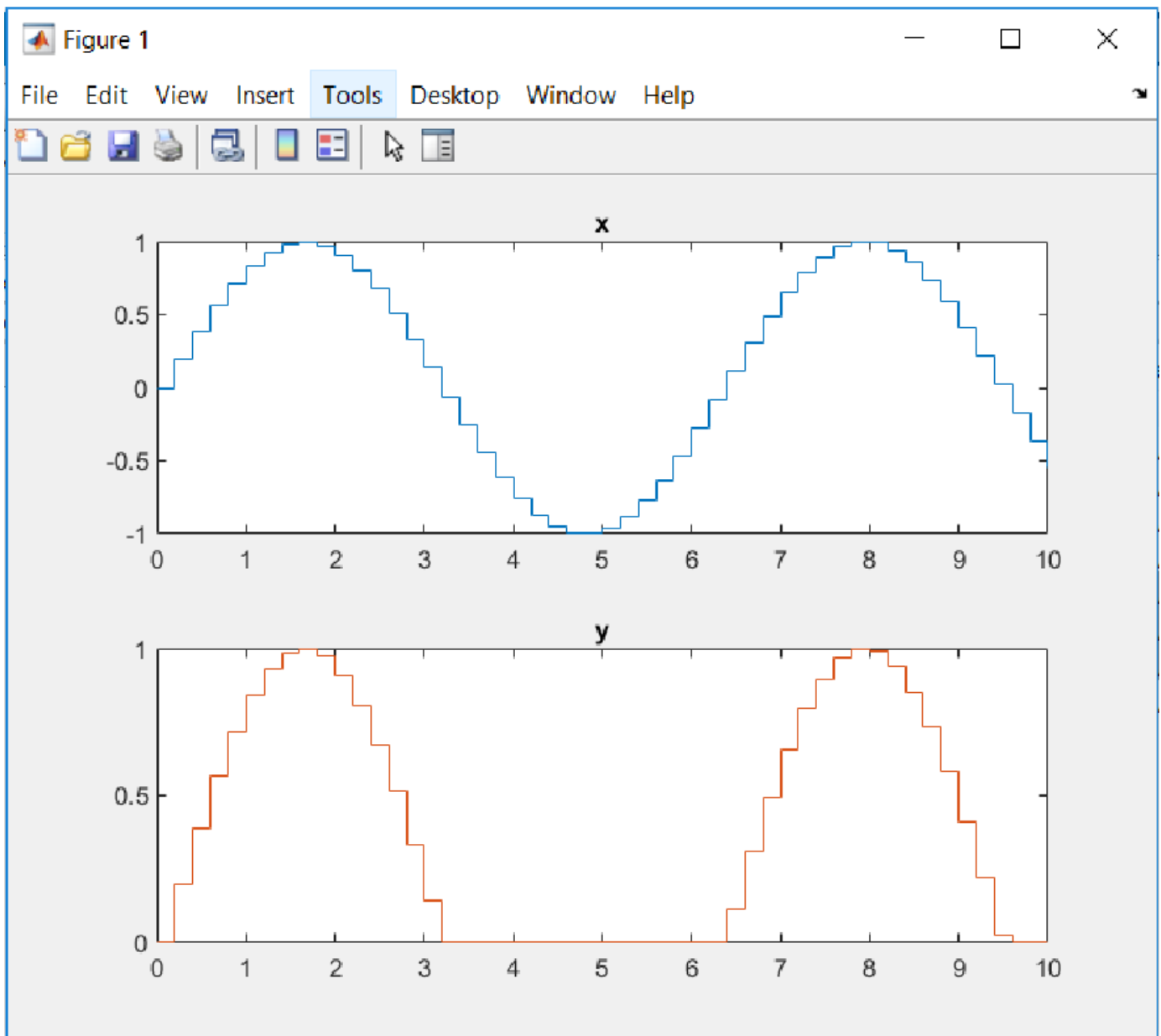
The execution results show that the rectifier filters out negative input values.

See Also

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

More About

- "Model Finite State Machines" on page 2-2
- "Define Chart Behavior by Using Actions" on page 2-18
- "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 2-2.

Example of State and Transition Actions

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 $n_0 = u, n_1, n_2, n_3, \ldots$ 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.
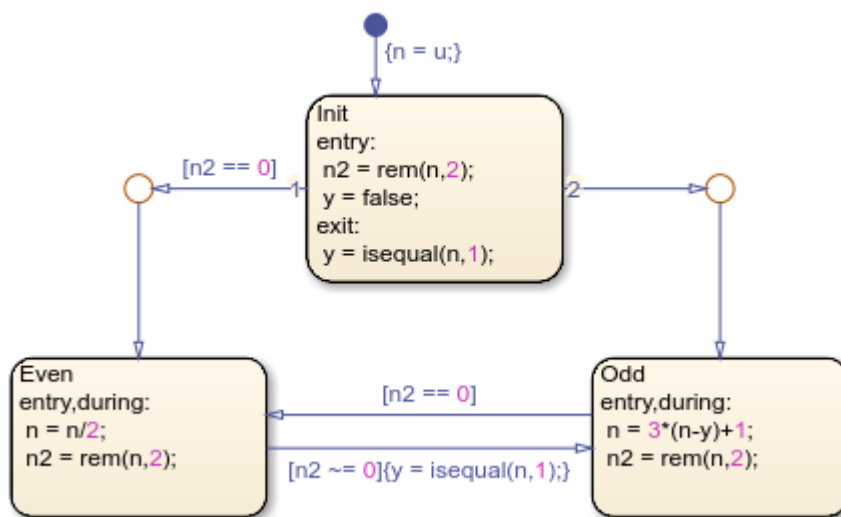
The chart consists of three states. At the start of simulation, the `Init` state initializes the chart data:

- 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.

State	Action	Result

`Init`	`entry:`
	`  n2 = rem(n,2);`
	`  y = false;`
		When `Init` becomes active at the start of the simulation, determines the parity of `n` and sets `y` to `false`.

	`exit:` `  y = isequal(n,1);`	When transitioning out of `Init` after one time step, determines whether `n` is equal to one.

`Even`	`entry,during:`
	`  n = n/2;`
	`  n2 = rem(n,2);`
		Computes the next number of the hailstone sequence (`n / 2`) and updates its parity on:
		
		• The time step when `Even` first becomes active.
		
		• Every subsequent time step that `Even` is active.

`Odd`	`entry,during:`
	`  n = 3*(n-y)+1;`
	`  n2 = rem(n,2);`
		Computes the next number of the hailstone sequence (3n+1) and updates its parity on:
		
		• The time step when `Odd` first becomes active.
		
		• Every subsequent time step that `Odd` is active.
		
		Throughout most of the simulation, `y` evaluates to zero. On the last time step, when `n = 1`, `y` evaluates to one so this action does not modify `n` or `n2` before the simulation stops.

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 Symbols pane, select the local data `n`. In the Property Inspector, under **Logging**, select **Log signal data**.

3 In the Constant block, enter an input of `u = 9`.

4
   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 `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.

5   In the **Simulation** tab, under **Review Results**, click **Data Inspector** ![icon].

6   To see the values of the hailstone sequence, in the Simulation Data Inspector, select the logged signal n.
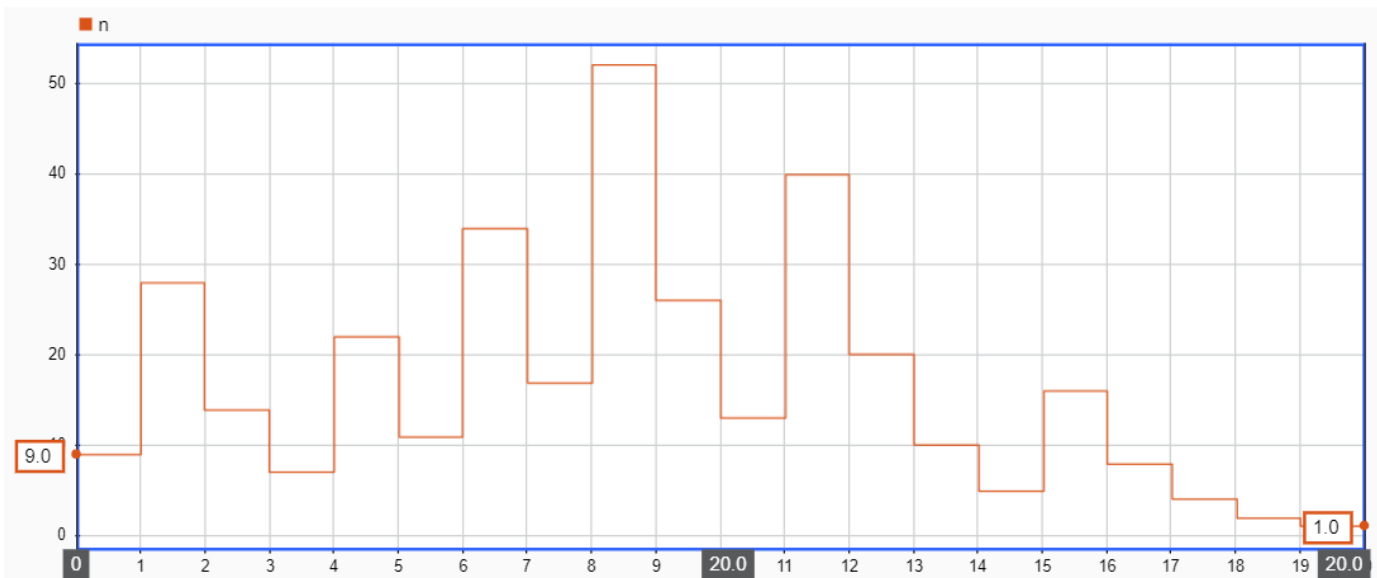
See Also

Stop Simulation

See Also

More About

- "Model Finite State Machines" on page 2-2
- "State Action Types"
- "Transition Action Types"
- "Synchronize Parallel States by Broadcasting Events" on page 2-33
- "Schedule Chart Actions by Using Temporal Logic" on page 2-47

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 2-2.

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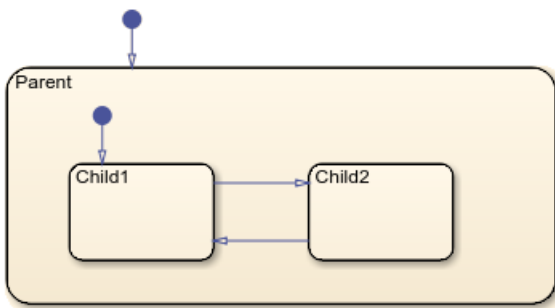
The contents of a parent state behave like a smaller chart. When a parent state becomes active, one of its child states also becomes active. When the parent state becomes inactive, all of its child states become inactive.

Example of Hierarchy

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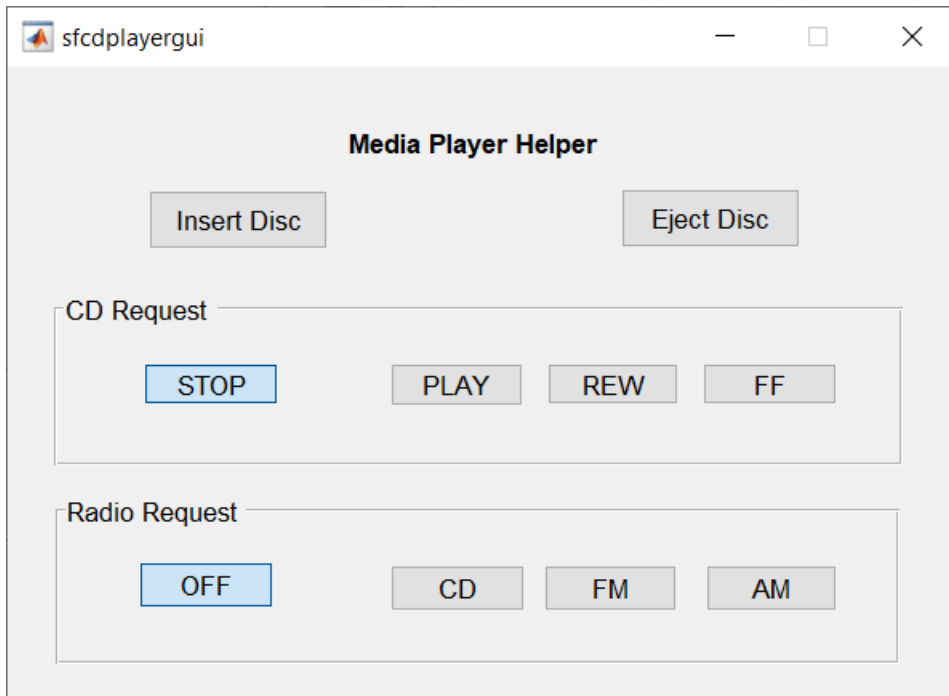
The stereo is initially in standby mode (OFF). When you select one of the Radio Request buttons, the stereo turns on the corresponding subcomponent. If you select the CD player, you can click one of the CD Request buttons to choose Play, Rewind, Fast-Forward, or Stop. You can insert or eject a disc at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

Implement Behavior by Using State Hierarchy

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 turn on the stereo.
2 You turn on the CD player.
3 You playing a disc.
4 You click 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 when you meet 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 Media Player Mode Manager chart. To open the Model Explorer, in the **Modeling** tab, select **Model Explorer**.

```
∨ ⊟ Media Player Mode Manager
   ∨ ▢ ModeManager
        ▢ Standby
      ∨ ▢ ON
         ∨ ▤ CDMode
            ∨ ▤ Play
                 ▢ Normal
                 ▢ Rew
                 ▢ FastForward
              ▢ Stop
            ▤ AMMode
            ▤ FMMode
   ▢ Eject
```
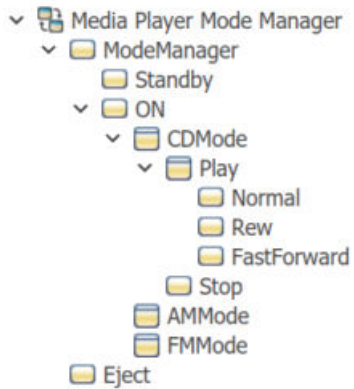
This table lists the role of each state in the hierarchy.

Hierarchy Level	State	Description
Top level (Media Player Mode Manager chart)	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
	Rew	Reverse play mode
	FastForward	Fast-Forward play mode

This figure shows the complete layout of the states in the chart.
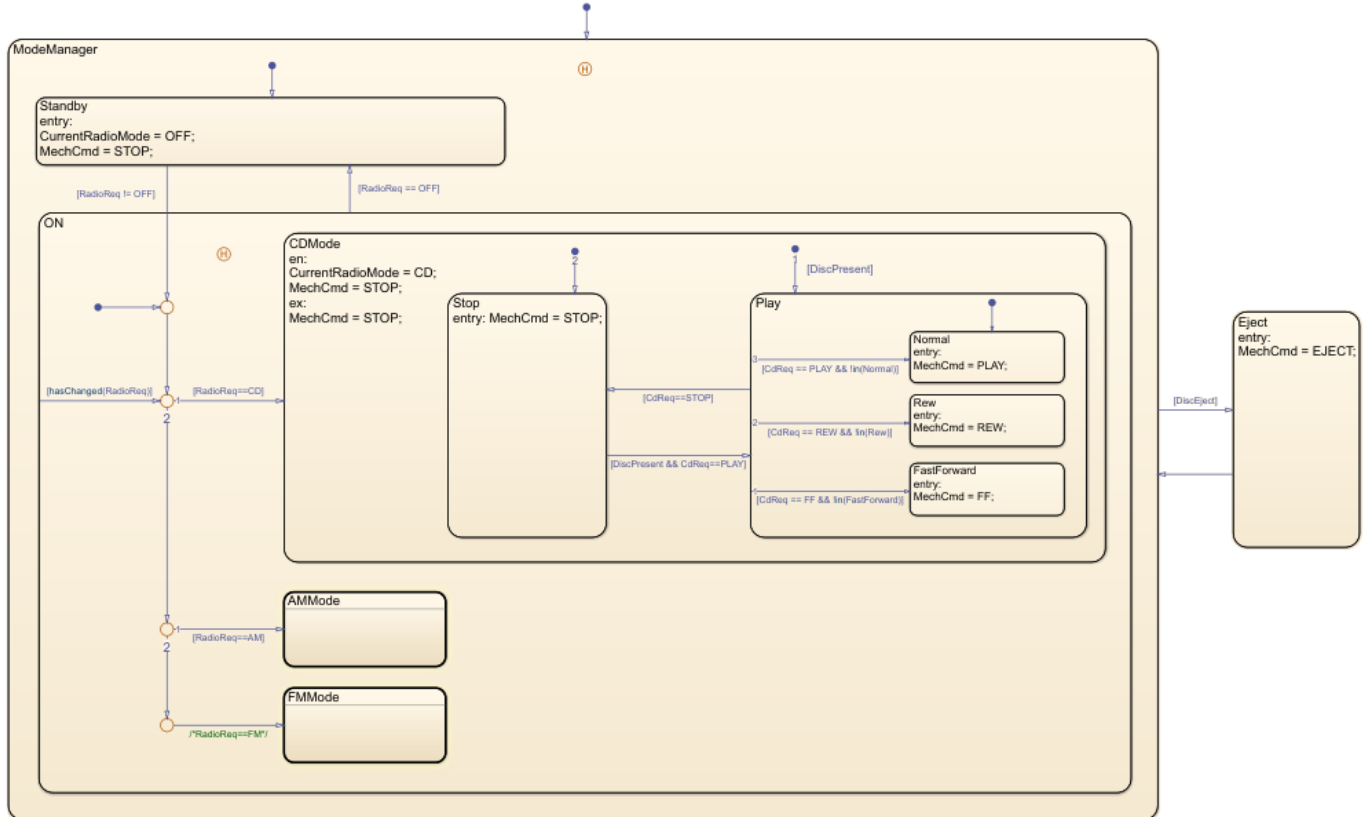
Simplify Chart Appearance by Using Subcharts

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 Media Player Mode Manager chart, you see only three levels of the state hierarchy. To see the details inside one of the subcharts, double-click the subchart.
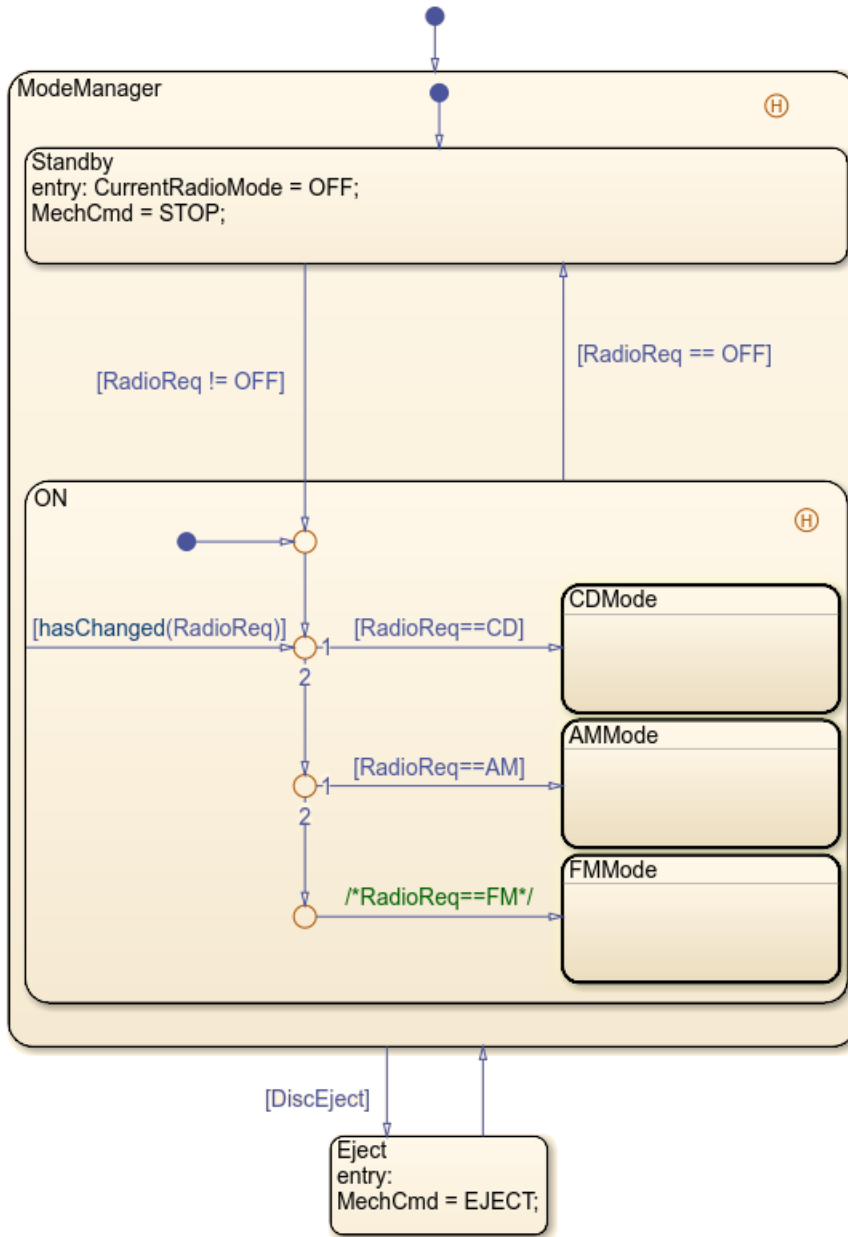
Explore the Example

The example `sf_cdplayer` contains two other Stateflow charts:

- User Request manages the interface with the UI and passes inputs to the Media Player Mode Manager and CD Player Behavior Model charts.
- CD Player Behavior Model receives the output from the User Request and Media Player Mode Manager charts and mimics the mechanical behavior of the CD player.
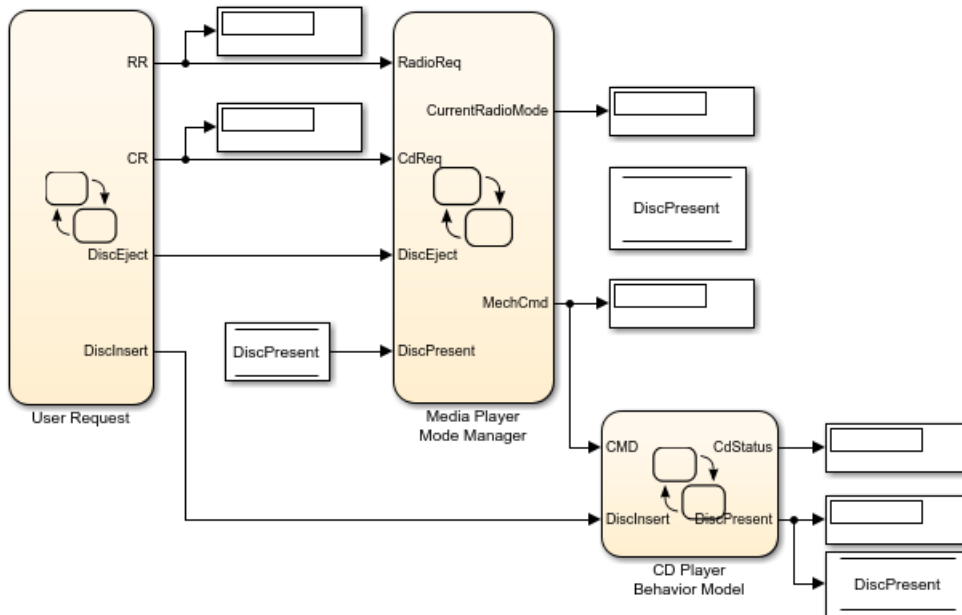
During simulation, you can investigate how each chart responds to interactions with the Media Player Helper. To switch quickly between charts, use the tabs at the top of the Stateflow Editor.

See Also

Related Examples
- "Model Media Player by Using Enumerated Data"
- "Modeling a CD Player/Radio Using State Transition Tables"
- "Simulate a Media Player by Using Strings"

More About
- "Model Finite State Machines" on page 2-2
- "State Hierarchy"
- "Encapsulate Modal Logic by Using Subcharts"

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 2-2.

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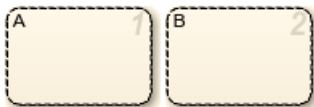
All states at a given hierarchical level must be of the same type. The parent state, or in the case of top-level states, the chart itself, has OR (exclusive) or AND (parallel) decomposition. The default state decomposition type is OR (exclusive). To change the decomposition type, right-click the parent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

Example of Parallel Decomposition

This example employs parallelism to implement an air controller that maintains air temperature at 120 degrees in a physical plant.
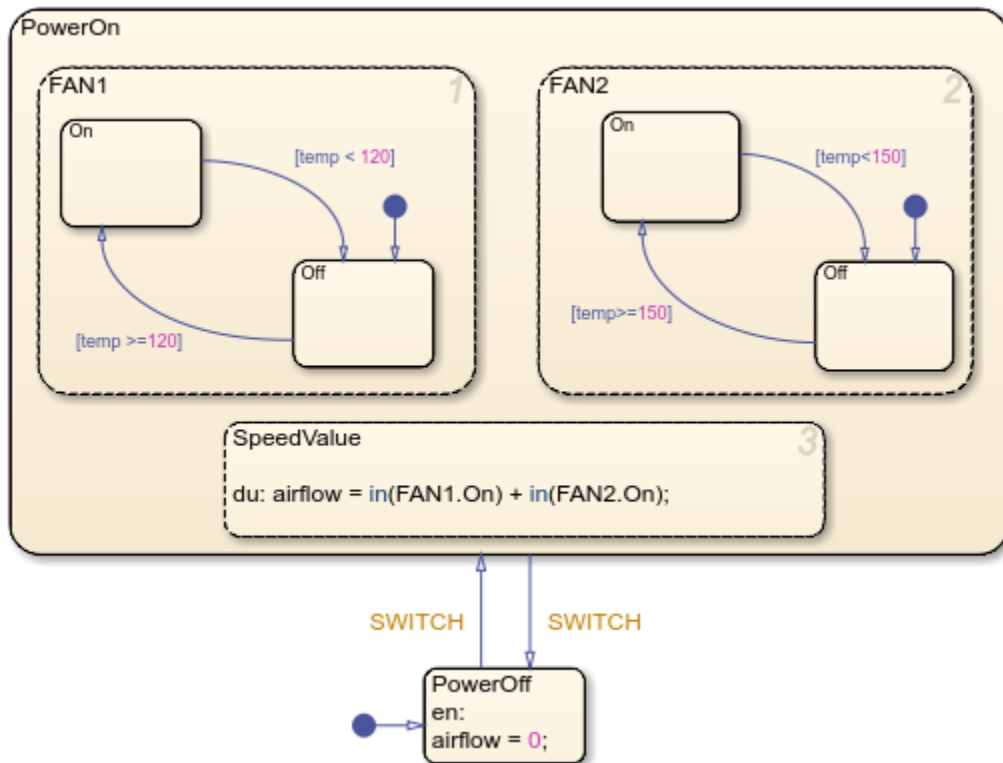
The controller operates two fans. The first fan turns on when the air temperature rises above 120 degrees. The second fan provides additional cooling when the air temperature rises above 150 degrees. The chart models these fans as parallel states FAN1 and FAN2, both of which are active when the controller 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. 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.

Combine Exclusive (OR) and Parallel (AND) States

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.

State	Decomposition	Rationale
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.

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".

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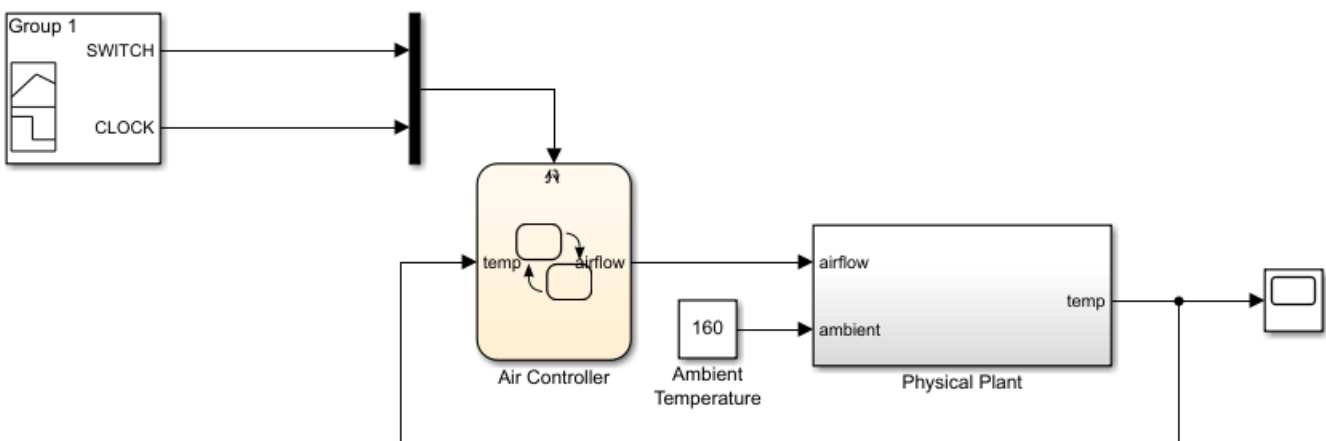
Based on the air temperature temp, the Air Controller chart turns on the fans and passes the value of airflow to the Physical Plant subsystem. This output value determines the amount of cooling activity, as indicated by this table.

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

$$\texttt{temp}(0) = T_{Initial}$$
$$\texttt{temp}'(t) = (T_{Ambient} - \texttt{temp}(t)) \cdot (k_{Heat} - k_{Cool}),$$

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

in

More About

- "Model Finite State Machines" on page 2-2
- "State Decomposition"
- "Execution Order for Parallel States"
- "Check State Activity by Using the in Operator"
- "Synchronize Parallel States by Broadcasting Events" on page 2-33

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 2-29.

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

This example uses local events as part of the design of a home security system.

The Door and Win states model anti-intrusion sensors at a window and a door. When these sensors are active, the corresponding states check the value of D_mode and W_mode. If one of these variables is true, the states broadcast a local event to the Alarm state.

Input from the motion sensor can be sporadic, so a debouncing state is used. The debouncing state remains active while Mot_sens is true. After a delay period, the debouncing state broadcasts a local event to the Alert state.

Door 1

Disabled

[D_mode==1]

Active

[Door_sens && ...
in(Alarm.On.Idle)]
{send(Alert,Alarm);}

[D_mode==0]

Motion 2

Active

after(1/tSample,sl_call)
[in(Alarm.On.Idle)] /{
send(Alert,Alarm);}

Debouncing 1

[~Mot_sens] 2 [Mot_sens]

Idle

[M_mode==0] [M_mode==1]

Disabled

Win 3

Disabled

[W_mode==1]

Active

[Win_sens && ...
in(Alarm.On.Idle)]
{send(Alert,Alarm);}

[W_mode==0]
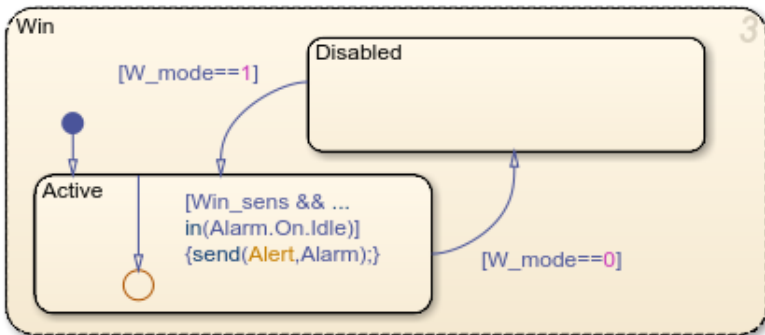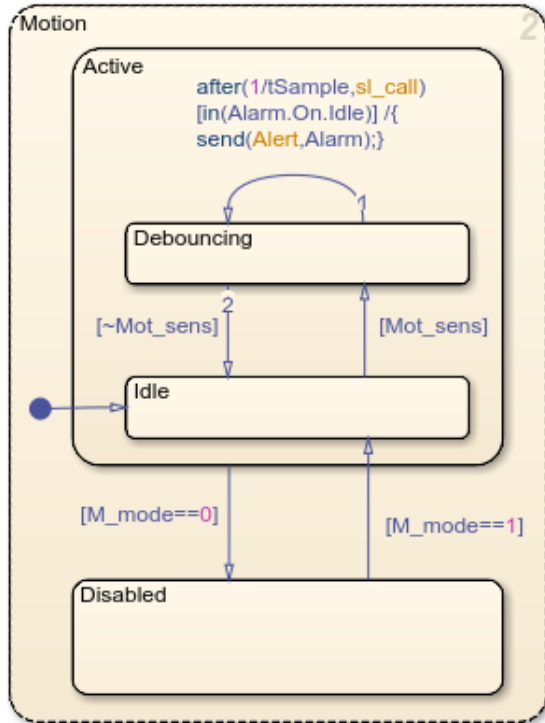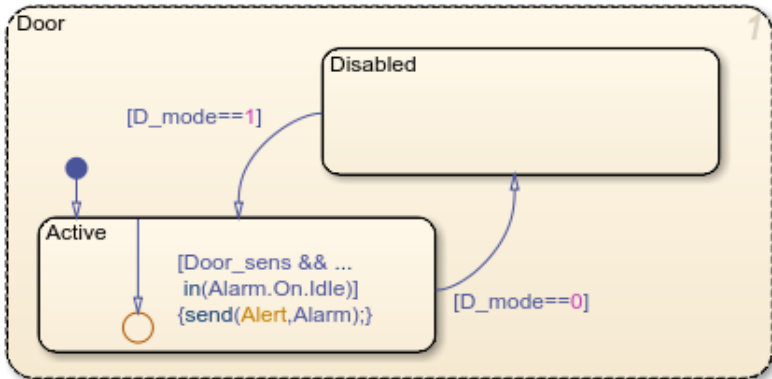
Alarm 4

When an Alert event is broadcast to the Alarm state, the Pending state becomes activate for a delay period. If the alarm is not disabled during the delay period, the police are contacted by the output event call_police.

On

Off

[Alarm_active==0]

Pending
entry: Sound;

Alert

Idle

[Alarm_active==1]

after(4/tSample,sl_call)/
{call_police;}
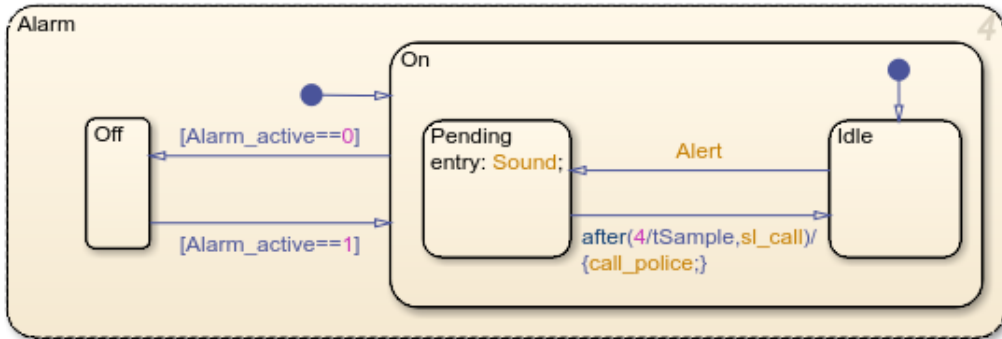
The security system consists of an alarm and three anti-intrusion sensors (a window sensor, a door sensor, and a motion detector). After the system detects an intrusion, you have a small amount of time to disable the alarm. Otherwise, the system calls the police.

Model Subsystems as Parallel States

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.
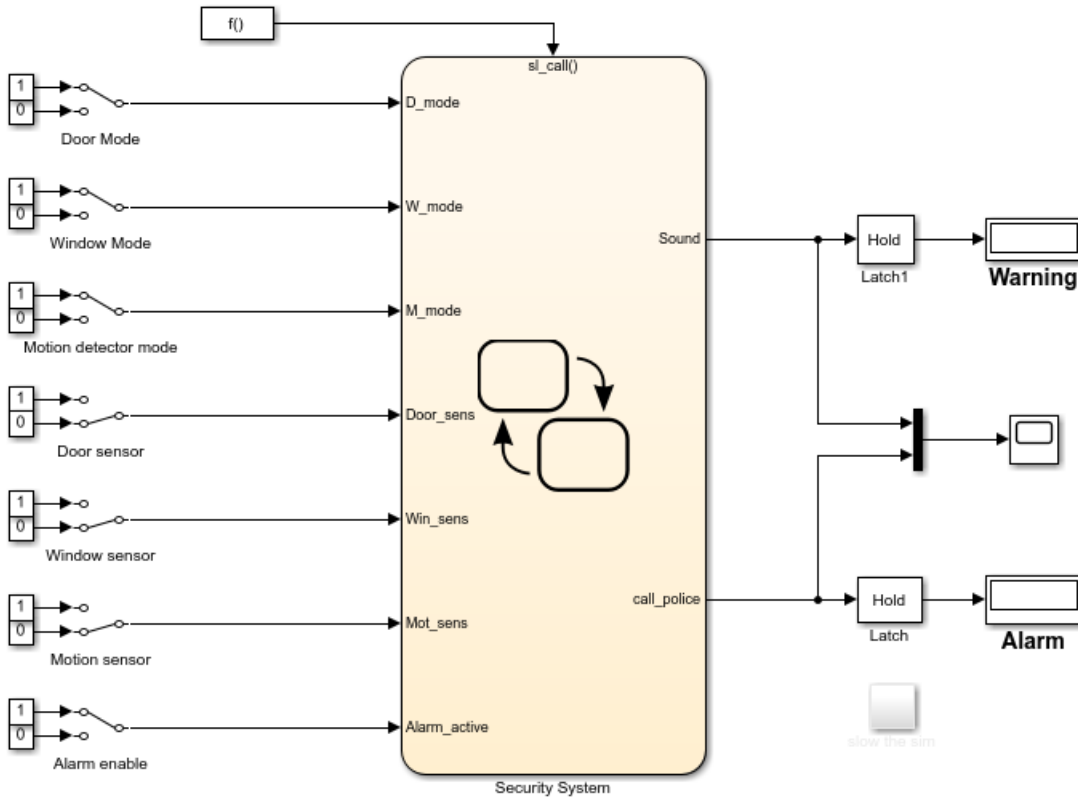
Output Events

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.
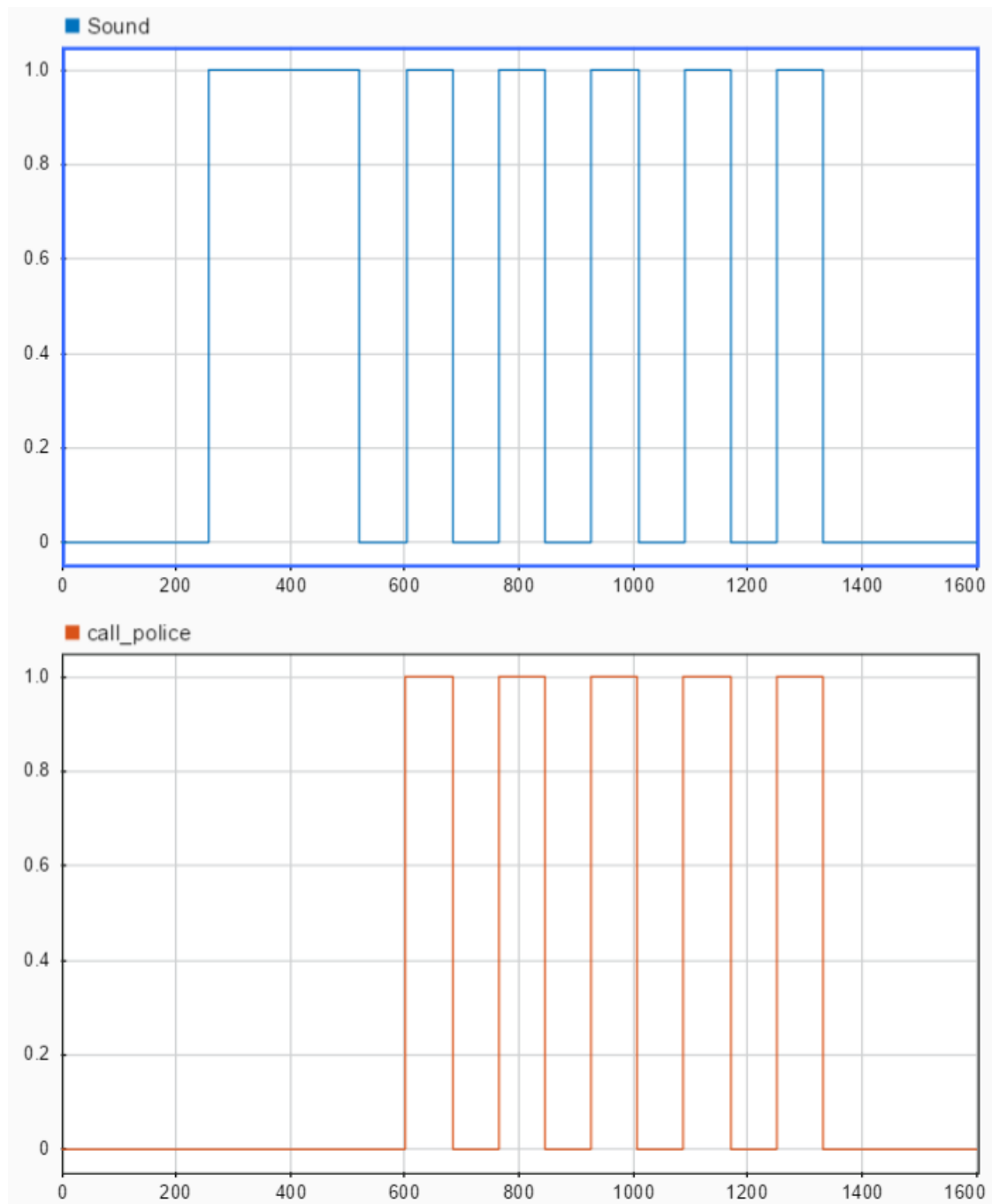
See Also

Related Examples

- "Model a Security System"

More About

- "Model Synchronous Subsystems by Using Parallelism" on page 2-29
- "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 2-47

# 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 2-23.

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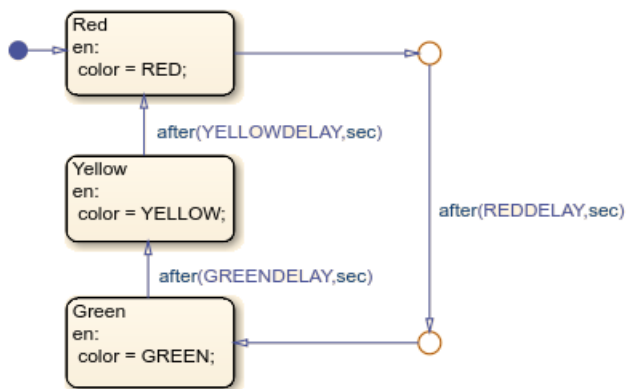
Stateflow provides active state data through an output port to Simulink or as local data to your chart. This table lists the different modes of active state data available.

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 pane.

Example of Active State Data

This example uses active state data to model the controller system for a pair of traffic lights.
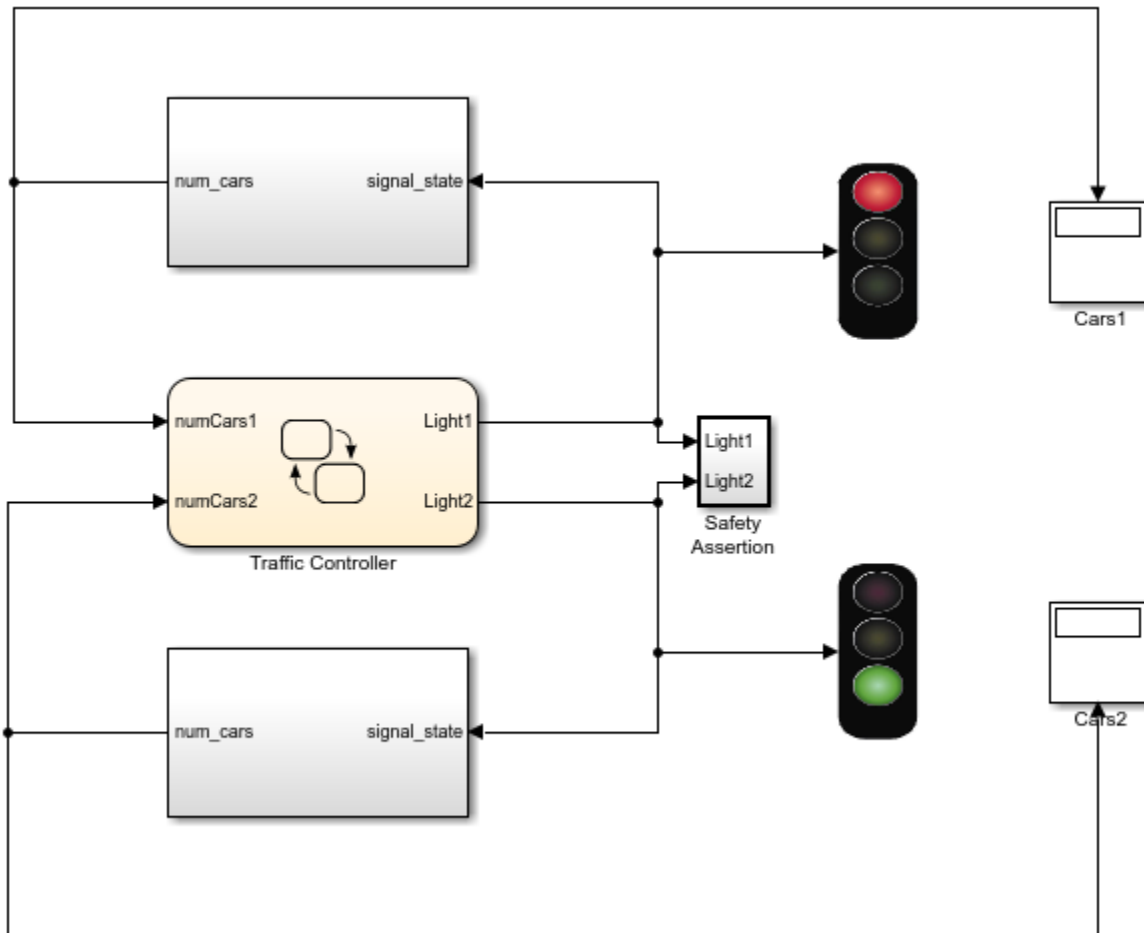
Inside the Traffic Controller chart, two parallel subcharts manage the logic controlling the traffic lights. The subcharts have an identical hierarchy consisting of three child states: `Red`, `Yellow`, and `Green`. The output data `Light1` and `Light2` correspond to the active child states in the subcharts. 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.
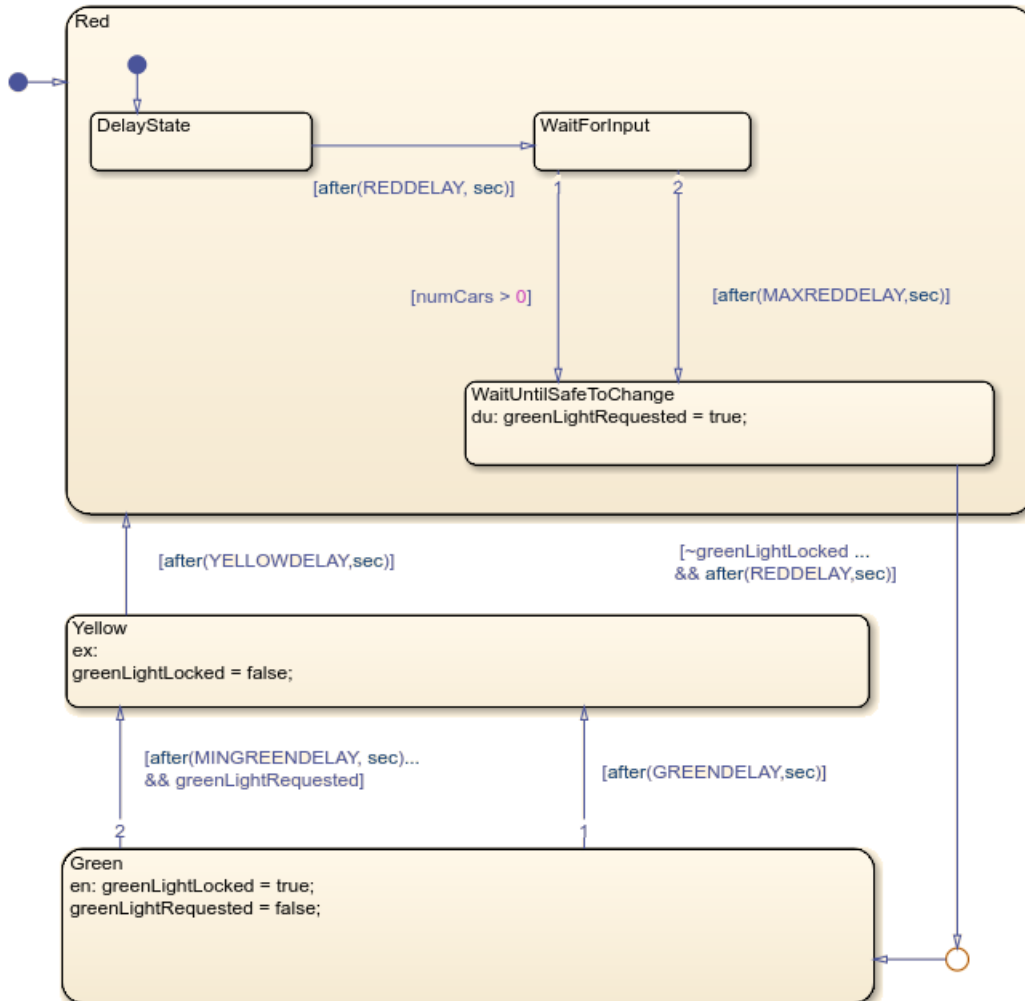
Red Light

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.
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   Enable logging for each of these symbols. In the Symbols pane, select each symbol. In the Property Inspector, under **Logging**, select **Log signal data**.

   - `greenLightRequested`
   - `greenLightLocked`
   - `Light1`
   - `Light2`

2   In the **Simulation** tab, click **Run** ▶.

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

4   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`.
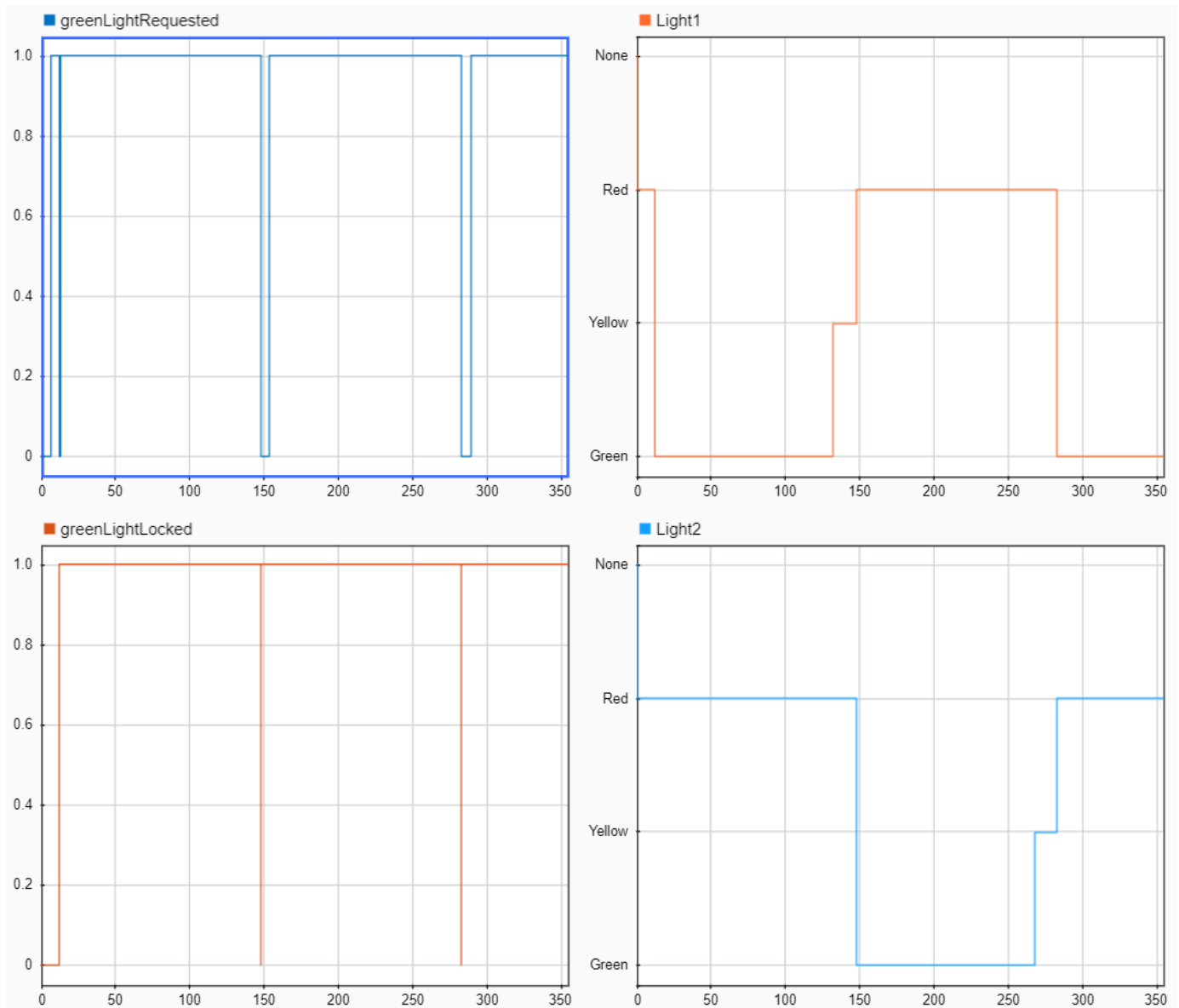
To trace the chart activity during the simulation, you can use the zoom and cursor buttons in the Simulation Data Inspector. For example, this table details the activity during the first 300 seconds of the simulation.

Time	Description	Light 1	Light2	greenLightRequested	greenLightLocked
$t = 0$	At the start of the simulation, both traffic lights are red.	Red	Red	false	false

Time	Description	Light 1	Light2	greenLightRequested	greenLightLocked
$t = 6$	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`
$t = 147$	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`

Time	Description	Light 1	Light2	greenLightRequested	greenLightLocked
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`
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 2-23
• "Schedule Chart Actions by Using Temporal Logic" on page 2-47
• "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 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 2-18.

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

This example uses temporal logic to model a bang-bang controller that regulates the internal temperature of a boiler.
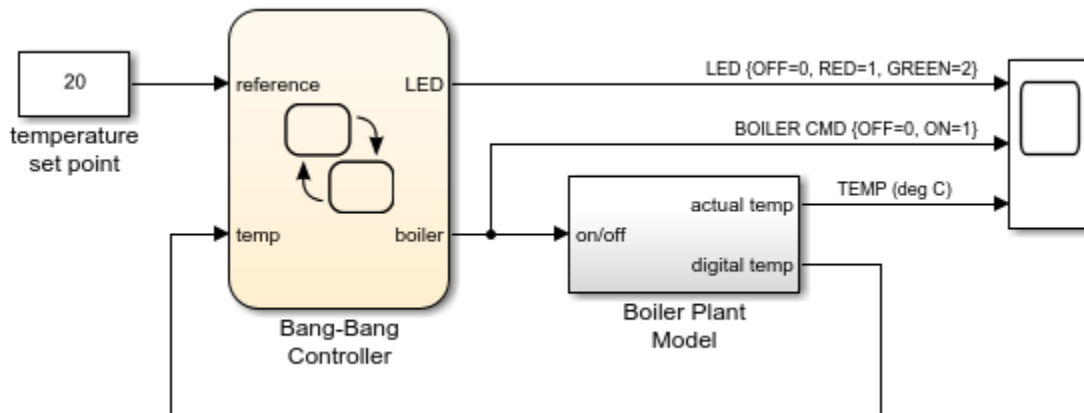
The example consists of a Stateflow chart and a Simulink® subsystem. The Bang-Bang Controller chart compares the current boiler temperature to a reference set point and determines whether to turn on the boiler. The Boiler Plant Model subsystem models the dynamics inside the boiler, increasing or decreasing 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.
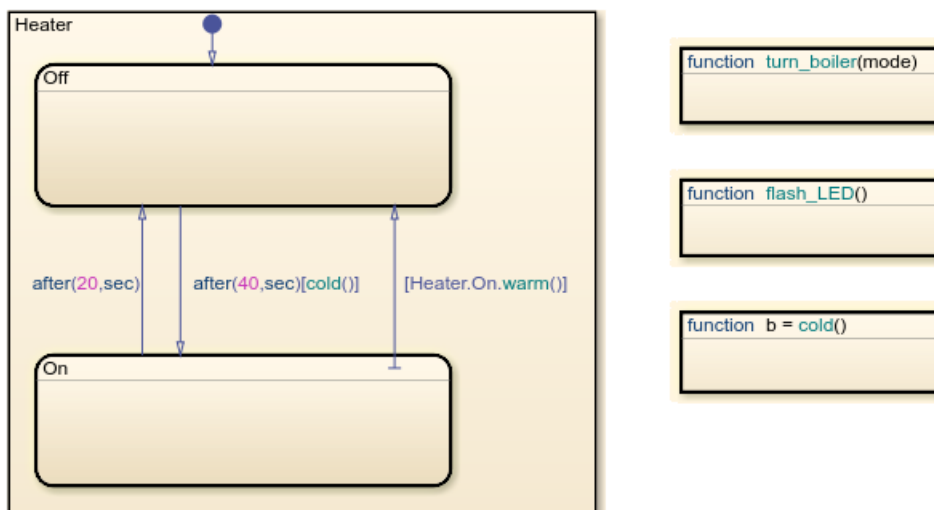
The actions guarding the transitions between the `On` and `Off` substates define the behavior of the bang-bang controller.

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.
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.
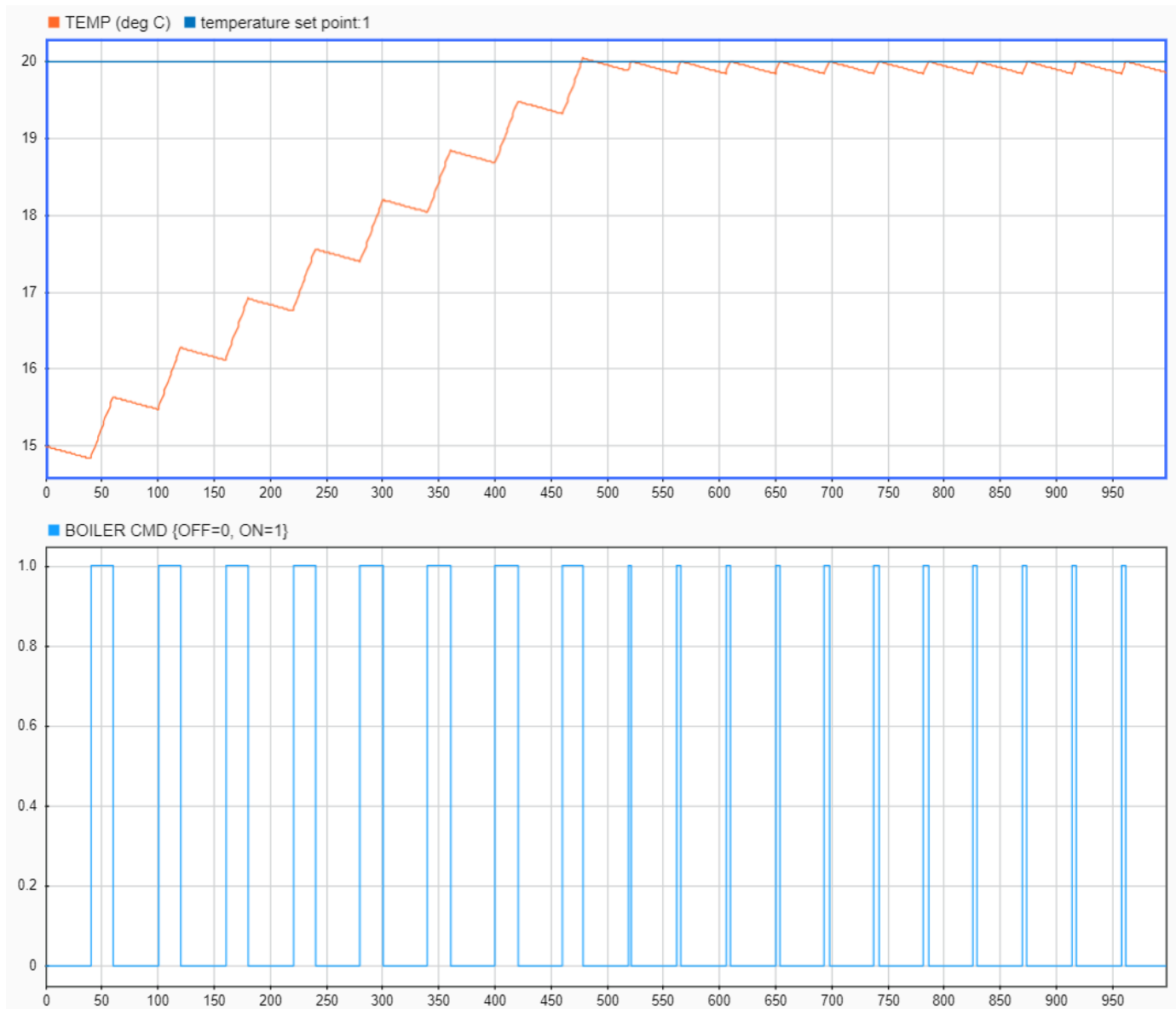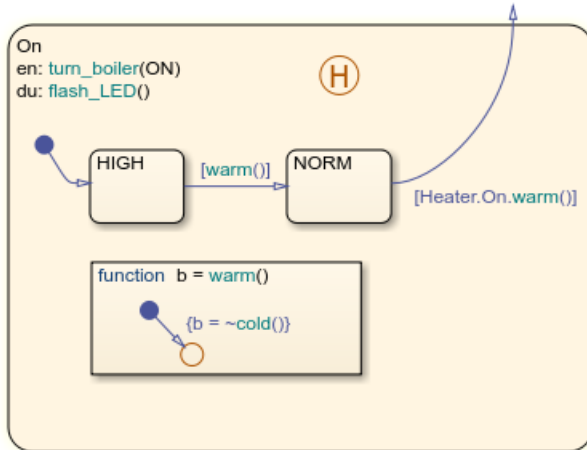
Timing of Status LED

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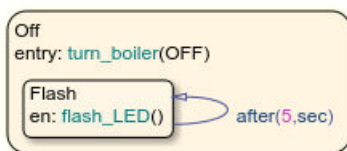
The `On` state calls the graphical function `flash_LED` as a state action of type `during`. When the `On` state is active, it calls the function at every time step of the simulation (in this case, every second), toggling the value of the output symbol LED between 0 and 2.

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.
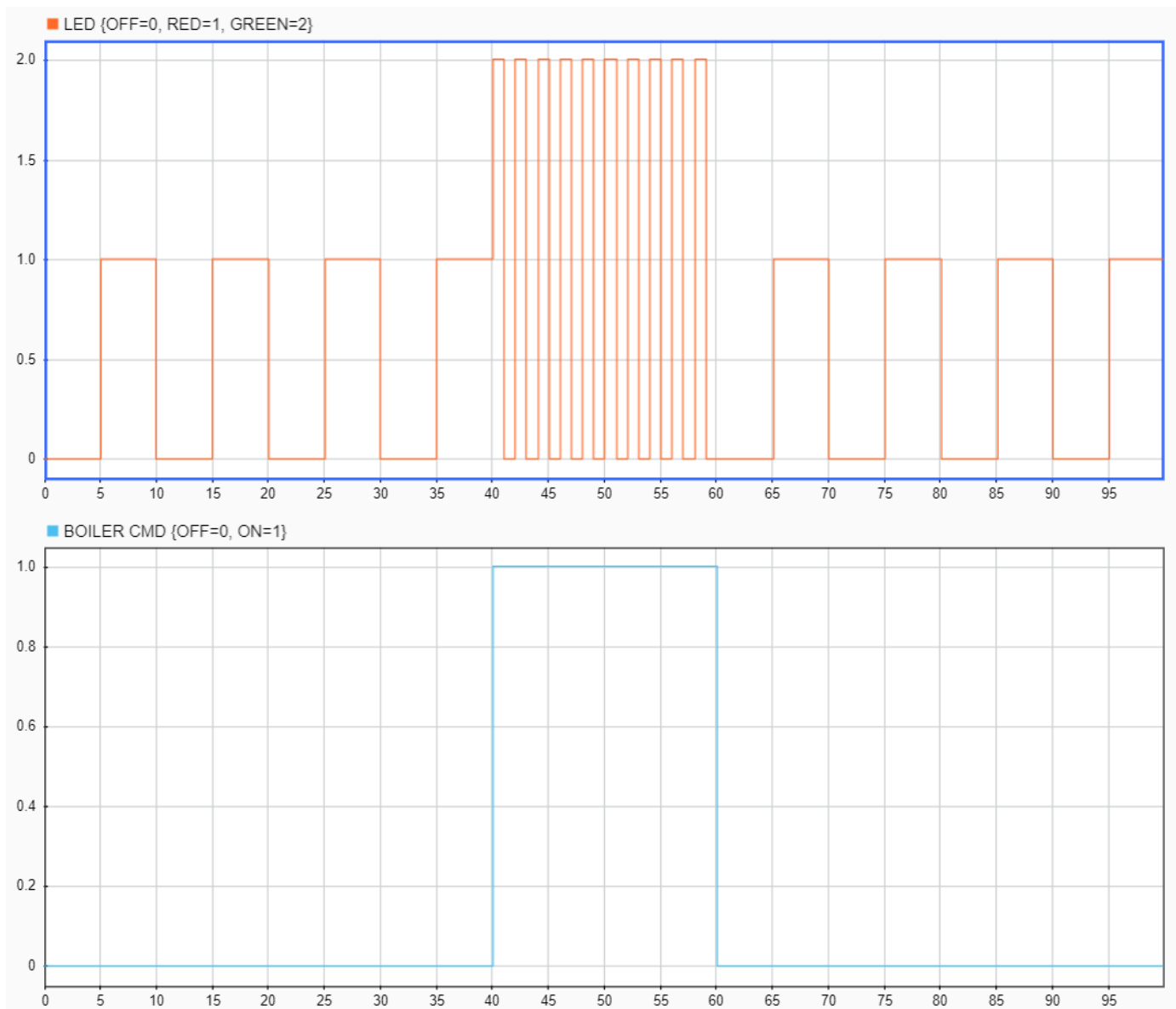
Explore the Example

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 `elapsed` and `duration` operators.

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

3   Enable logging for each of these symbols. In the Symbols pane, select each symbol. In the Property Inspector, under **Logging**, select **Log signal data**.

- `boiler`
- `Timer1`
- `Timer2`

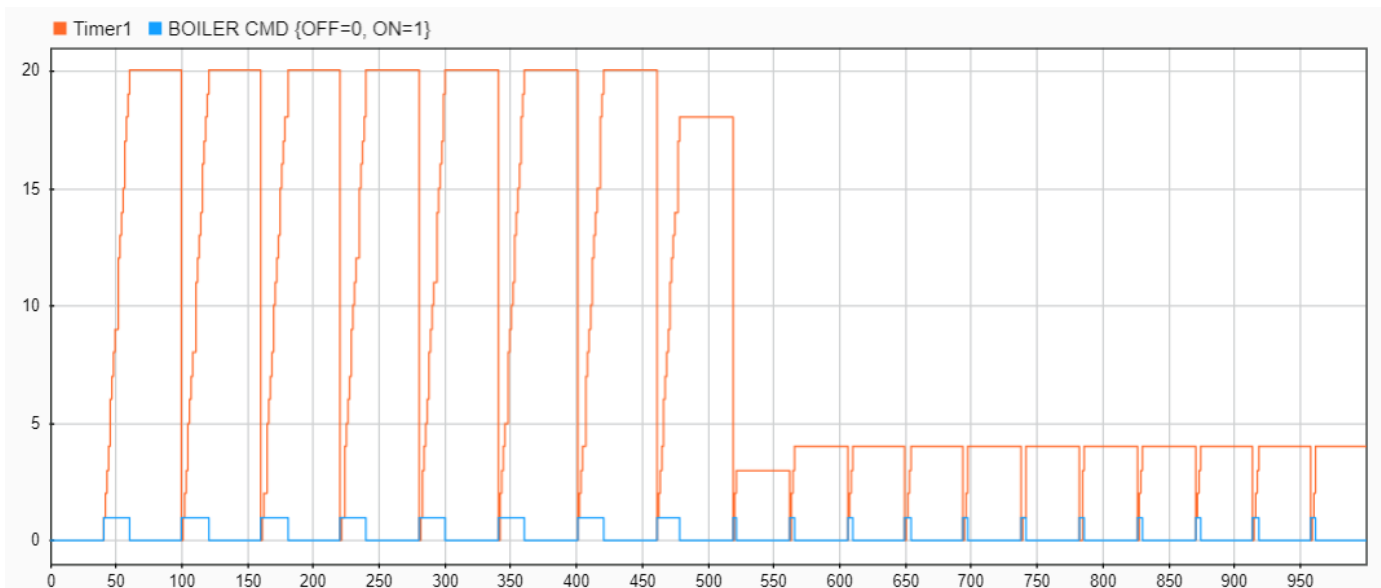**4**   In the **Simulation** tab, click **Run** .

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.
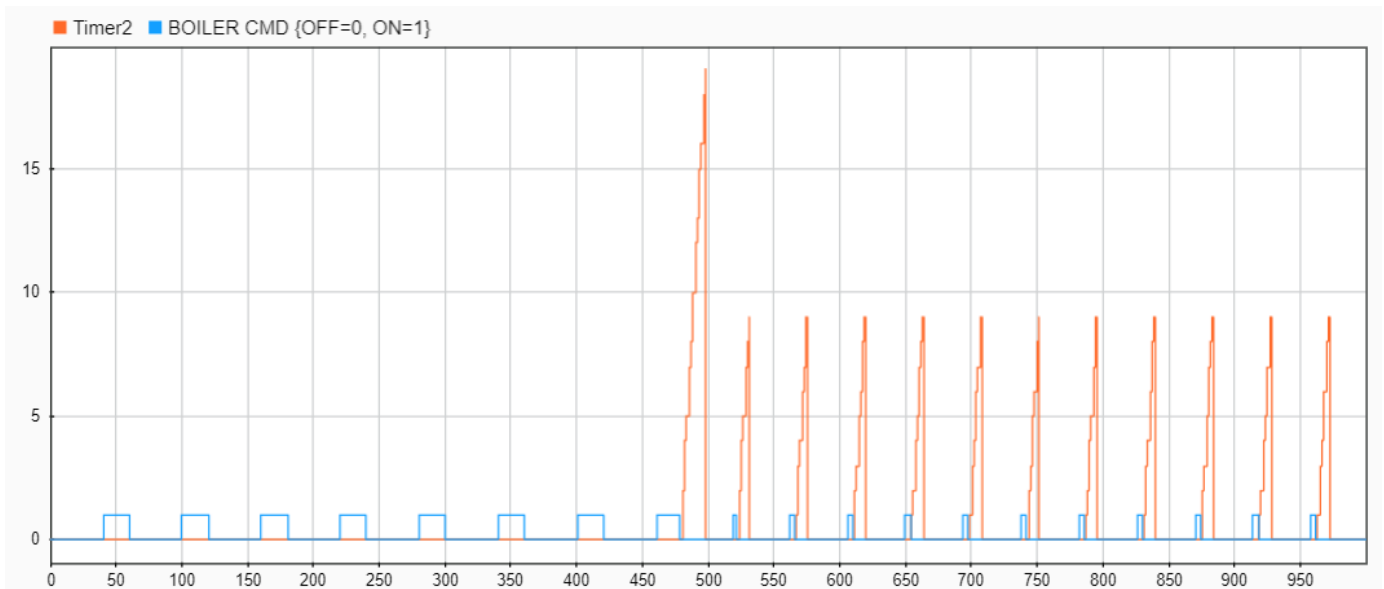

7   In the Simulation Data Inspector, display the signals `boiler` and `Timer2` in the same set of axes. The plot shows that:

- 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 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 Ⓗ 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

after | duration | elapsed

Related Examples

- "Model Bang-Bang Temperature Control System"

More About

- "Define Chart Behavior by Using Actions" on page 2-18
- "Control Chart Execution by Using Temporal Logic"
- "Reuse Logic Patterns by Defining Graphical Functions"
- "History Junctions"