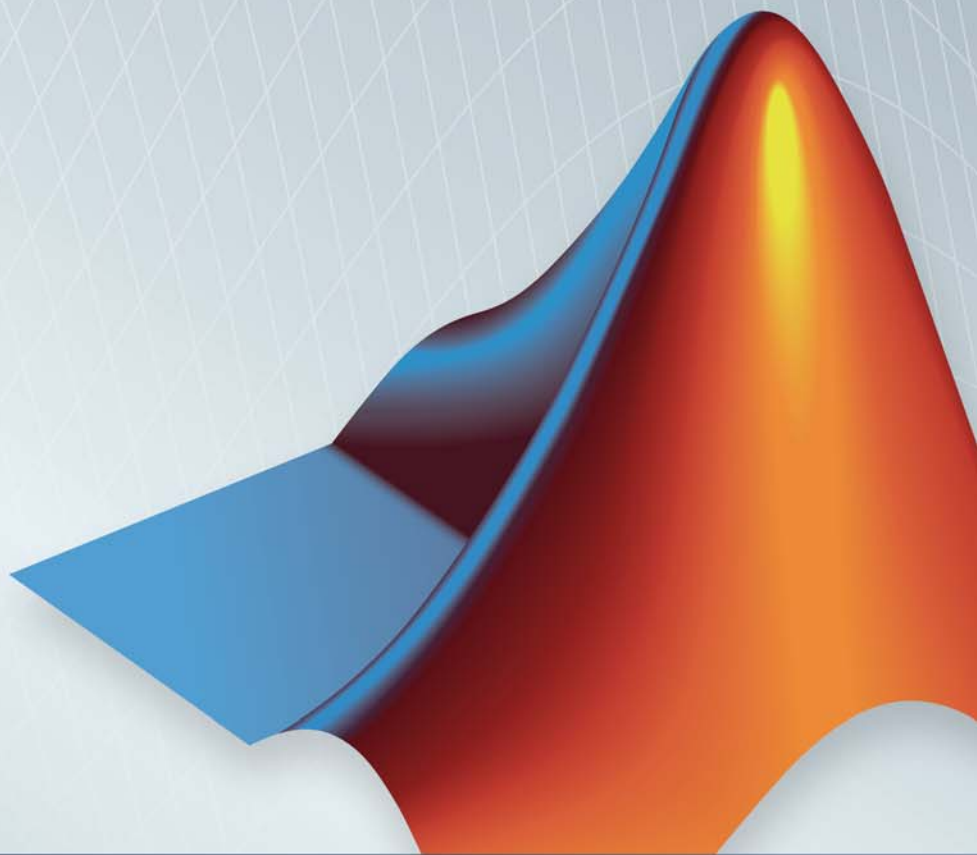


**Stateflow<sup>®</sup>**  
Reference

**R2014a**



**MATLAB<sup>®</sup>&SIMULINK<sup>®</sup>**



## How to Contact MathWorks



[www.mathworks.com](http://www.mathworks.com) Web  
[comp.soft-sys.matlab](mailto:comp.soft-sys.matlab) Newsgroup  
[www.mathworks.com/contact\\_TS.html](http://www.mathworks.com/contact_TS.html) Technical Support



[suggest@mathworks.com](mailto:suggest@mathworks.com) Product enhancement suggestions  
[bugs@mathworks.com](mailto:bugs@mathworks.com) Bug reports  
[doc@mathworks.com](mailto:doc@mathworks.com) Documentation error reports  
[service@mathworks.com](mailto:service@mathworks.com) Order status, license renewals, passcodes  
[info@mathworks.com](mailto:info@mathworks.com) Sales, pricing, and general information



508-647-7000 (Phone)



508-647-7001 (Fax)



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

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

*Stateflow*® Reference

© COPYRIGHT 2006–2014 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

March 2006	Online only	New for Version 6.4 (Release 2006a)
September 2006	Online only	Revised for Version 6.5 (Release R2006b)
September 2007	Online only	Rereleased for Version 7.0 (Release 2007b)
March 2008	Online only	Revised for Version 7.1 (Release 2008a)
October 2008	Online only	Revised for Version 7.2 (Release 2008b)
March 2009	Online only	Rereleased for Version 7.3 (Release 2009a)
September 2009	Online only	Revised for Version 7.4 (Release 2009b)
March 2010	Online only	Rereleased for Version 7.5 (Release 2010a)
September 2010	Online only	Rereleased for Version 7.6 (Release 2010b)
April 2011	Online only	Rereleased for Version 7.7 (Release 2011a)
September 2011	Online only	Rereleased 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)



## Functions — Alphabetical List

**1**

## Block Reference

**2**



# Functions — Alphabetical List

---

# sfclipboard

---

<b>Purpose</b>	Stateflow clipboard object
<b>Syntax</b>	<code>object = sfclipboard</code>
<b>Description</b>	<code>object = sfclipboard</code> returns a handle to the Stateflow® clipboard object, which you use to copy objects from one chart or state to another.

**Examples** Copy the `init` function from the `Init` chart to the `Pool` chart in the `sf_pool` model:

```
sf_pool;
% Get handle to the root object
rt = sfroot;
% Get handle to 'init' function in Init chart
f1 = rt.find('-isa','Stateflow.EMFunction','Name','init');
% Get handle to Pool chart
chP = rt.find('-isa','Stateflow.Chart','Name','Pool');
% Get handle to the clipboard object
cb = sfclipboard;
% Copy 'init' function to the clipboard
cb.copy(f1);
% Paste 'init' function to the Pool chart
cb.pasteTo(chP);
% Get handle to newly pasted function
f2 = chP.find('-isa','Stateflow.EMFunction','Name','init');
% Reset position of new function in the Pool chart
f2.Position = [90 180 90 60];
```

**See Also** `sfgco` | `sfnew` | `sfroot` | `stateflow`

**Tutorials**

- “Copy Objects”
- “Create and Access Charts Using the Stateflow API”

**How To**

- “Getting a Handle on Stateflow API Objects”
- “Access the Chart Object”



**Purpose** Close chart

**Syntax**  
sfclose  
sfclose('chart\_name')  
sfclose('all')

**Description** sfclose closes the current chart.  
sfclose('chart\_name') closes the chart called 'chart\_name'.  
sfclose('all') closes all open or minimized charts. 'all' is a literal string.

**See Also** sfnew | sfoopen | stateflow

# sfdebugger

---

<b>Purpose</b>	Open Stateflow Debugger
<b>Syntax</b>	<code>sfdebugger</code> <code>sfdebugger('model_name')</code>
<b>Description</b>	<code>sfdebugger</code> opens the Stateflow Debugger for the current model. <code>sfdebugger('model_name')</code> opens the debugger for the Simulink® model called ' <i>model_name</i> '. Use this input argument to specify which model to debug when you have multiple models open.
<b>See Also</b>	<code>sfexplr</code>   <code>sfhelp</code>   <code>sflib</code>
<b>How To</b>	<ul style="list-style-type: none"><li>• “Debug Run-Time Errors in a Chart”</li></ul>

<b>Purpose</b>	Open Model Explorer
<b>Syntax</b>	<code>sfexplr</code>
<b>Description</b>	<code>sfexplr</code> opens the Model Explorer. A model does not need to be open.
<b>See Also</b>	<code>sfdebugger</code>   <code>sfhelp</code>   <code>sflib</code>
<b>How To</b>	<ul style="list-style-type: none"><li>• “Use the Model Explorer with Stateflow Objects”</li></ul>



- “Zoom a Chart Object Using the API”

# sfhelp

---

<b>Purpose</b>	Open Stateflow online help
<b>Syntax</b>	<code>sfhelp</code>
<b>Description</b>	<code>sfhelp</code> opens the Stateflow online help in the MATLAB® Help browser.
<b>See Also</b>	<code>sfdebugger</code>   <code>sfexplr</code>   <code>sfnew</code>   <code>stateflow</code>

**Purpose** Open Stateflow library window

**Syntax** `sflib`

**Description** `sflib` opens the Stateflow block library. From this library, you can drag Stateflow blocks into Simulink models and access the Stateflow Examples Library.

**See Also** `sfdebugger` | `sfexplr` | `sfhelp` | `sfnew`

**Purpose** Create model containing empty Stateflow block

**Syntax**

```
sfnew
sfnew('chart_type')
sfnew('model_name')
sfnew('chart_type','model_name')
```

**Description** `sfnew` creates an untitled model with an empty chart. Stateflow sets the default action language for new charts to MATLAB. To change the default action language to C, use the command `sfpref('ActionLanguage','C')`. For more information, see “Modify the Action Language for a Chart”.

`sfnew('chart_type')` creates an untitled model that contains an empty block of type `chart_type`.

`sfnew('model_name')` creates a model called `model_name` with an empty chart with the default action language.

`sfnew('chart_type','model_name')` creates a model called `model_name` with an empty block of type `chart_type`.

## Input Arguments

### **chart\_type**

Empty block to add to an empty model:

- |         |   |
|---------|---|
| -MATLAB | Use a chart that supports MATLAB expressions in Stateflow actions |
| -C      | Use a chart that supports C expressions in Stateflow actions      |
| -Mealy  | Use a chart that supports only Mealy state machine semantics      |
| -Moore  | Use a chart that supports only Moore state machine semantics      |



---

-TT	Use a truth table
-STT	Use a state transition table

**model\_name**

Name of the model.

**Examples**

Create a untitled model with an empty chart that uses MATLAB as the action language:

```
sfnew()
```

---

Create a model called `MyModel` with an empty chart that uses only Mealy semantics:

```
sfnew(' -Mealy', 'MyModel')
```

---

Create a model called `MyModel` with an empty chart that uses only Moore semantics:

```
sfnew(' -Moore', 'MyModel')
```

**See Also**

`sfhelp` | `sfprint` | `sfrout` | `sfsave` | `stateflow`

**How To**

- “Model Event-Driven System”
- “Create Mealy and Moore Charts”
- “Build Model with Stateflow Truth Table”
- “Syntax for States and Transitions”

# sfopen

---

<b>Purpose</b>	Open existing model
<b>Syntax</b>	<code>sfopen</code>
<b>Description</b>	<code>sfopen</code> prompts you for a model file and opens the model that you select from your file system.
<b>See Also</b>	<code>sfclose</code>   <code>sfdebugger</code>   <code>sfexplr</code>   <code>sflib</code>   <code>sfnew</code>   <code>stateflow</code>

**Purpose**

Print graphical view of charts

**Syntax**

```
sfprint
sfprint(objects)
sfprint(objects,format)
sfprint(objects,format,output_option)
sfprint(objects,format,output_option,print_entire_chart)
```

**Description**

`sfprint` prints a PostScript file of the current chart to the default printer.

`sfprint(objects)` prints a PostScript file of all charts specified by `objects` to the default printer.

`sfprint(objects,format)` prints all charts specified by `objects` in the specified `format` to output files. Each output file matches the name of the chart and the file extension matches the `format`.

`sfprint(objects,format,output_option)` prints all charts specified by `objects` in the specified `format` to the file or printer specified in `output_option`.

`sfprint(objects,format,output_option,print_entire_chart)` prints all charts specified by `objects` in the specified `format` to the file or printer specified in `output_option`. As specified in `print_entire_chart`, prints either a complete or current view.

**Input Arguments****objects - Identifier of charts to print**

`gcb` (default) | `gcs` | `string`

Identifier of charts to print, specified as a command or a string. Use:

- `gcb` to specify the current block of the model.
- `gcs` to specify the current system of the model.

- a string to specify the path of a chart, model, subsystem, or block.

**Example:** `sfprint(gcs)`

Prints all the charts in the current system to the default printer.

**Example:** `sfprint('sf_pool/Pool')`

Prints the complete chart with the path 'sf\_pool/Pool' to the default printer.

### **format - Output format of printed charts specified as a string**

``bitmap' | 'eps' | `epsc' | `jpg' | `meta' | `png' | `ps' | `psc'  
| `tif'`

Output format of the printed charts specified as one of these string values:

<code>'bitmap'</code>	Save the chart image to the clipboard as a bitmap (for Windows® operating systems only)
<code>'eps'</code>	Generate an encapsulated PostScript file
<code>'epsc'</code>	Generate a color encapsulated PostScript file
<code>'jpg'</code>	Generate a JPEG file
<code>'meta'</code>	Save the chart image to the clipboard as a metafile (for Windows operating systems only)
<code>'png'</code>	Generate a PNG file
<code>'ps'</code>	Generate a PostScript file
<code>'psc'</code>	Generate a color PostScript file
<code>'tif'</code>	Generate a TIFF file

**Example:** `sfprint('sf_car/shift_logic','jpg')`

Prints the complete chart with the path 'sf\_car/shift\_logic' in a JPEG format to a file in the current folder named 'sf\_car\_shift\_logic.jpg'.

### Data Types

char

### output\_option - Name of the printer or output file

`'file'` (default) | string | `'clipboard'` | `'promptForFile'` | `'printer'`

Name of the output file or printer specified as one of these values:

<code>'file'</code>	Send output to a default file with the name <i>chart_name.file_extension</i> . The file name is the name of the chart, with an extension that matches the output format.
string	Specify the name of the output file with a string.
<code>'clipboard'</code>	Copy output to the clipboard
<code>'promptForFile'</code>	Prompts the user interactively for path and file name.
<code>'printer'</code>	Send output to the default printer (use only with 'ps', or 'eps' formats)

**Example:** `sfprint('sf_car/shift_logic','png','myFile')`

Prints the complete chart whose path is 'sf\_car/shift\_logic' in the PNG format to a file in the current folder with the name 'myFile'.png.

**Example:** `sfprint('sf_car/shift_logic','ps','promptForFile')`

Prints all charts in the current block of the model in PostScript format. An interactive window opens for each chart to prompt you for the path and name of the output file.

## Data Types

char

## **print\_entire\_chart - View of charts to print**

1 (default) | 0

View of charts to print specified as a integer of value 0 or 1. A value of 1 prints the complete views of all the charts, whereas a value of 0 prints the current views of all the charts.

**Example:** `sfprint(gcs,'png','file',0)`

Prints the current view of all charts in the current system in PNG format using default file names.

## Examples

### **Print open chart**

```
sfprint
```

Prints current chart to the default printer.

### **Print PostScript file of all charts specified in path**

```
sfprint('sf_car/shift_logic');
```

Prints the chart with the path 'sf\_car/shift\_logic' in PostScript format to the default printer.

### **Print chart specified in path to a JPG file format.**

```
sfprint('sf_car/shift_logic','jpg')
```

Prints a copy of the chart 'sf\_car/shift\_logic' in JPG format to the file 'sf\_car\_shift\_logic.jpg'.

**Print chart in TIFF format to the clipboard.**

```
sfprint(gcs, 'tif', 'clipboard')
```

Prints the chart in the current system to the clipboard in TIFF format.

**Print the current view of a chart.**

```
sfprint('sf_car/shift_logic', 'png', 'file', 0)
```

Prints the current view of 'sf\_car/shift\_logic' in a PNG format to the file 'sf\_car\_shift\_logic.png'.

**See Also**

`gcb` | `gcs` | `sfhelp` | `sfnew` | `sfsave` | `stateflow`

# sfroot

---

**Purpose** Root object

**Syntax** `object = sfroot`

**Description** `object = sfroot` returns a handle to the top-level object in the Stateflow hierarchy of objects. Use the root object to access all other objects in your charts when using the API.

**Examples** Zoom in on a state in your chart:

```
old_sf_car;  
% Get handle to the root object  
rt = sfroot;  
% Find the state with the name 'first'  
myState = rt.find('-isa','Stateflow.State','Name','first');  
% Zoom in on that state in the chart  
myState.fitToView;
```

**See Also** `sfclipboard` | `sfgco`

**Tutorials**

- “Create and Access Charts Using the Stateflow API”

**How To**

- “Getting a Handle on Stateflow API Objects”
- “Access the Chart Object”



---

<b>Purpose</b>	Save chart in current folder
<b>Syntax</b>	<pre>sfsave sfsave('model_name') sfsave('model_name','new_model_name') sfsave('Defaults')</pre>
<b>Description</b>	<p>sfsave saves the chart in the current model.</p> <p>sfsave('model_name') saves the chart in the model called 'model_name'.</p> <p>sfsave('model_name','new_model_name') saves the chart in 'model_name' to 'new_model_name'.</p> <p>sfsave('Defaults') saves the settings of the current model as defaults. 'Defaults' is a literal string.</p> <p>The model must be open and the current folder must be writable.</p>
<b>Examples</b>	<p>Develop a script to create a baseline chart and save it in a new model:</p> <pre>bdclose('all');  % Create an empty chart in a new model sfnew;  % Get root object rt = sfroot;  % Get model m = rt.find('-isa','Simulink.BlockDiagram');  % Get chart chart1 = m.find('-isa','Stateflow.Chart');  % Create two states, A and B, in the chart sA = Stateflow.State(chart1);</pre>

```
sA.Name = 'A';
sA.Position = [50 50 100 60];
sB = Stateflow.State(chart1);
sB.Name = 'B';
sB.Position = [200 50 100 60];

% Add a transition from state A to state B
tAB = Stateflow.Transition(chart1);
tAB.Source = sA;
tAB.Destination = sB;
tAB.SourceOClock = 3;
tAB.DestinationOClock = 9;

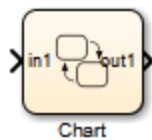
% Add a default transition to state A
dtA = Stateflow.Transition(chart1);
dtA.Destination = sA;
dtA.DestinationOClock = 0;
x = sA.Position(1)+sA.Position(3)/2;
y = sA.Position(2)-30;
dtA.SourceEndPoint = [x y];

% Add an input in1
d1 = Stateflow.Data(chart1);
d1.Scope = 'Input';
d1.Name = 'in1';

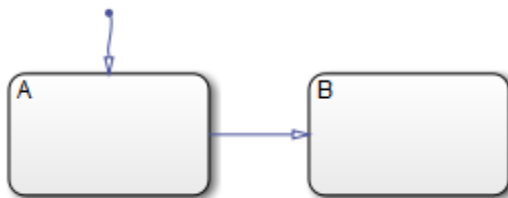
% Add an output out1
d2 = Stateflow.Data(chart1);
d2.Scope = 'Output';
d2.Name = 'out1';

% Save the chart in a model called "NewModel"
% in current folder
sfsave('untitled', 'NewModel');
```

Here is the resulting model:



Here is the resulting chart:



### See Also

`sfoopen` | `sfclose` | `sfrroot` | `sfnew` | `find`

### Tutorials

- “Create and Access Charts Using the Stateflow API”

### How To

- “Create a MATLAB Script of API Commands”

# stateflow

---

**Purpose** Create empty chart

**Syntax** `stateflow`

**Description** `stateflow` creates an untitled model that contains an empty chart. The function also opens the Stateflow block library. From this library, you can drag Stateflow blocks into models or access the Stateflow Examples Library.

**See Also** `sflib` | `sfnew`

# Block Reference

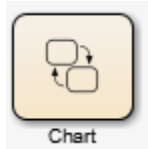
---

# Chart

---

**Purpose** Implement control logic with finite state machine

**Library** Stateflow



## Description

A *finite state machine* is a representation of an event-driven (reactive) system. In an event-driven system, the system responds to an event by making a transition from one state (mode) to another. This action occurs as long as the condition defining the change is true.

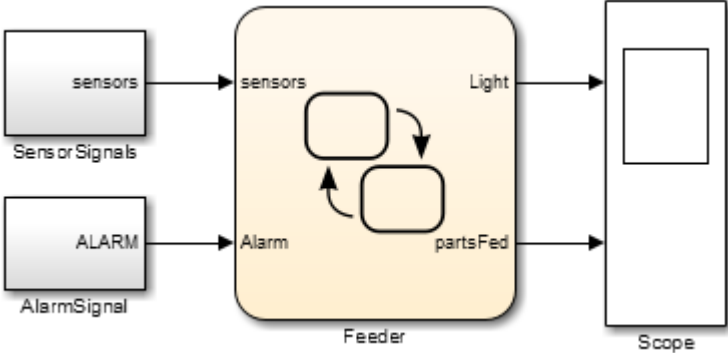
A Stateflow chart is a graphical representation of a finite state machine. *States* and *transitions* form the basic elements of the system. You can also represent stateless flow charts.

For example, you can use Stateflow charts to control a physical plant in response to events such as a temperature and pressure sensors, clocks, and user-driven events.

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

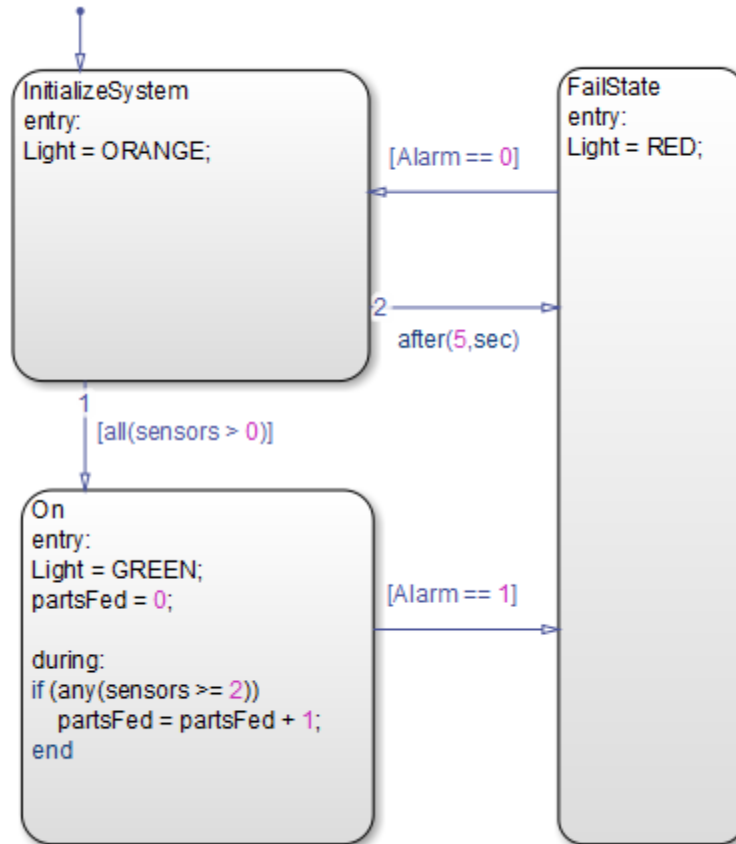
A Stateflow Chart can use MATLAB or C as the action language to implement control logic.

This block diagram represents a machine on an assembly line that feeds raw material to other parts of the line. It contains a chart, Feeder, with MATLAB as the action language.



# Chart

If you double-click the Feeder block in the model, the chart appears.



For a tutorial on this model, see “Model Event-Driven System”.

## Data Type Support

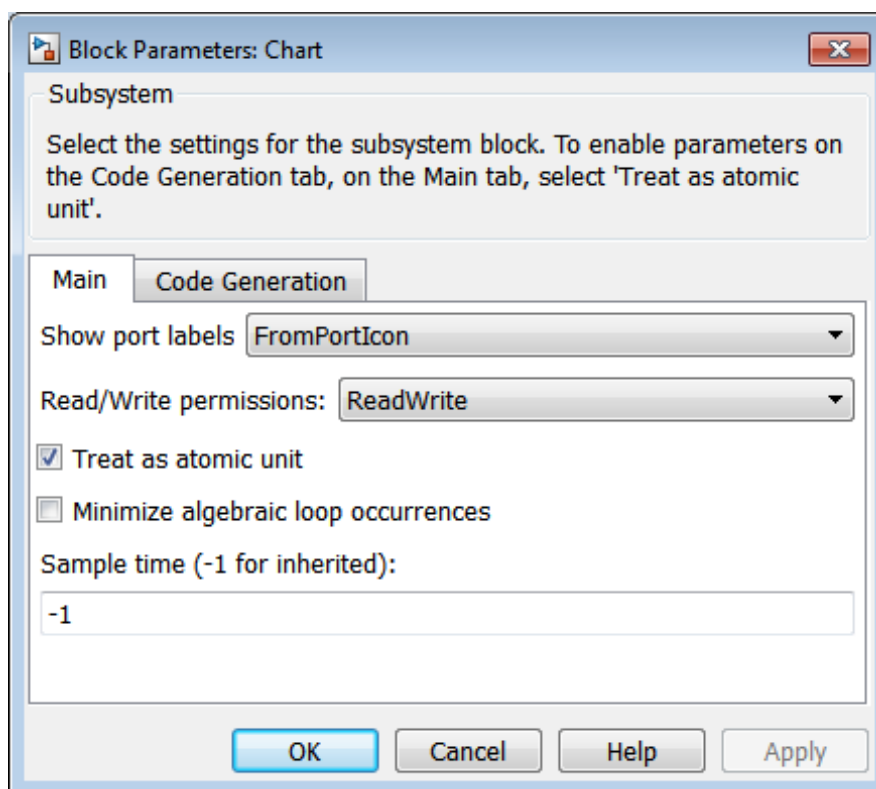
The Chart block accepts input signals of any data type that Simulink supports, including fixed-point data and enumerated data types. For a description of data types that Simulink supports, refer to the Simulink documentation.



Floating-point inputs pass through the block unchanged. Boolean inputs to charts that use MATLAB as the action language pass directly as Boolean outputs. Boolean inputs to charts that use C as the action language are treated as double type.

You can declare local data of any type or size.

## Parameters and Dialog Box



For a description of the block parameters, see the Subsystem block reference page in the Simulink documentation.

# Chart

---

## Characteristics

Direct Feedthrough	Yes
Sample Time	Specified in the <b>Sample time</b> parameter
Scalar Expansion	N/A
Dimensionalized	Yes
Zero-Crossing Detection	Yes, if enabled for continuous-time systems.  For more information, see “When to Enable Zero-Crossing Detection”.

# State Transition Table

**Purpose** Represent modal logic in tabular format

**Library** Stateflow



## Description

Use this block when you want to represent modal logic in tabular format. The State Transition Table block uses only MATLAB as the action language.

## State Transition Table Editor

If you double-click the State Transition Table block in `sf1ib`, the State Transition Table Editor shows the default layout of state-to-state transitions.

STATES	TRANSITIONS (Condition / Action / Desti...	
☐	if	else-if(1)
<div style="border: 2px solid black; border-radius: 10px; padding: 5px; width: fit-content;">           state1           <div style="position: absolute; left: -20px; top: 50%; transform: translateY(-50%);">             ● ↙           </div> </div>	<code>[x &gt; 0]</code>	
	<code>{x = x + 1;}</code>	
	<code>\$NEXT</code> ▼	▼
<div style="border: 2px solid black; border-radius: 10px; padding: 5px; width: fit-content;">           state2         </div>		
	▼	▼

Using the State Transition Table Editor, you can:

# State Transition Table

---

- Add states and enter state actions
- Add hierarchy among your states
- Enter conditions and actions for state-to-state transitions
- Specify default transitions, inner transitions, and self-loop transitions
- Add input or output data and events
- Set breakpoints for debugging
- Run diagnostics to detect parser errors
- View auto-generated content as you edit the table

For more information about the State Transition Table Editor, see “State Transition Table Editor Operations” in the Stateflow documentation.

## Adding Data and Events

You can add data and events from the State Transition Table Editor:

Element	Menu	Description
Inputs and outputs	<b>Table &gt; Add Inputs &amp; Outputs &gt; Data Input from Simulink</b> <b>Table &gt; Add Inputs &amp; Outputs &gt; Data Output to Simulink</b>	You can add inputs from the model and outputs to the model.
Data	<b>Table &gt; Add Other Elements</b>	You can add these types of data: <ul style="list-style-type: none"><li>• Local</li><li>• Constant</li><li>• Parameter</li><li>• Data store memory</li></ul>

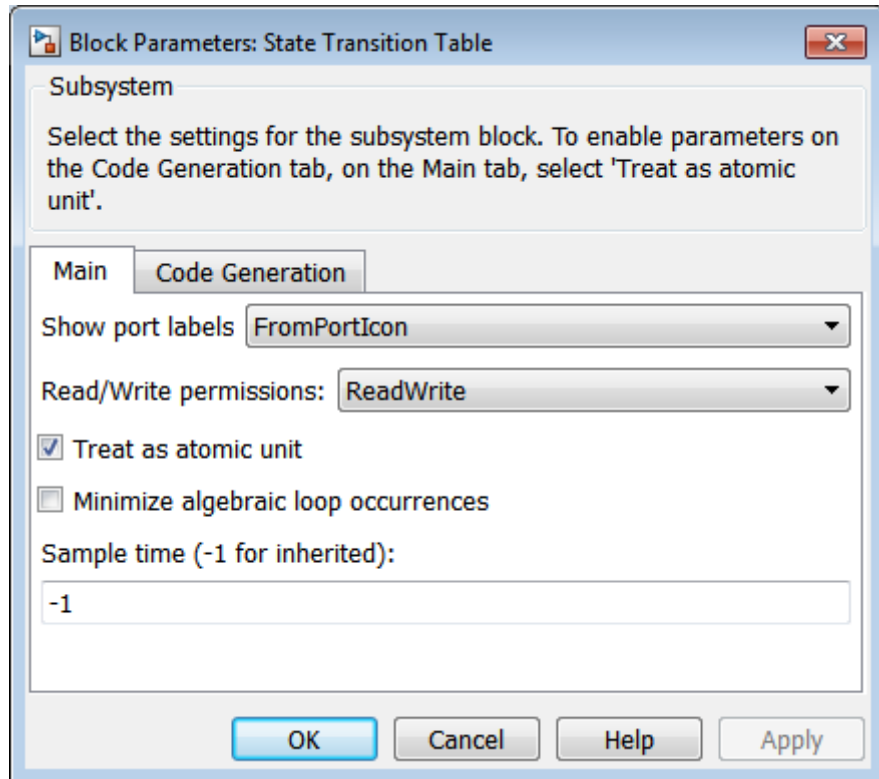
Element	Menu	Description
Input events	<b>Table &gt; Add Inputs &amp; Outputs &gt; Event Input from Simulink</b>	<p>An <i>input event</i> causes a State Transition Table block to execute when a Simulink control signal changes or through a Simulink block that outputs function-call events. You can use one of these input triggers:</p> <ul style="list-style-type: none"> <li>• Rising edge</li> <li>• Falling edge</li> <li>• Either rising or falling edge</li> <li>• Function call</li> </ul>
Output events	<b>Table &gt; Add Inputs &amp; Outputs &gt; Event Output to Simulink</b>	<p>A <i>output event</i> triggers a function call to a subsystem. You can use one of these output triggers:</p> <ul style="list-style-type: none"> <li>• Function call</li> <li>• Either rising or falling edge</li> </ul> <p>For more information, see “Create a Function-Call Subsystem”</p>

## Data Type Support

The State Transition Table block accepts input signals of any data type that Simulink supports, including fixed-point and enumerated data types.

# State Transition Table

## Parameters and Dialog Box



For a description of the block parameters, see the Subsystem block reference page.

## Characteristics

Direct Feedthrough	Yes
Sample Time	Specified in the <b>Sample time</b> parameter
Scalar Expansion	N/A

## State Transition Table

---

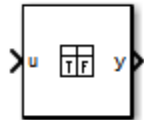
Dimensionalized	Yes
Zero-Crossing Detection	Yes, if enabled for continuous-time systems For more information, see “When to Enable Zero-Crossing Detection”.

# Truth Table

---

**Purpose** Represent logical decision-making behavior with conditions, decisions, and actions

**Library** Stateflow



## Description

The Truth Table block is a truth table function that uses MATLAB as the action language. Use this block when you want to use truth table logic directly in a Simulink model. This block requires a Stateflow license.

When you add a Truth Table block directly to a model instead of calling truth table functions from a Stateflow chart, these advantages apply:

- It is a more direct approach, especially if your model requires only a single truth table.
- You can define truth table inputs and outputs to have inherited types and sizes.

The Truth Table block works with a subset of the MATLAB language that is optimized for generating embeddable C code. This block generates content as MATLAB code. As a result, you can take advantage of other tools to debug your Truth Table block during simulation.

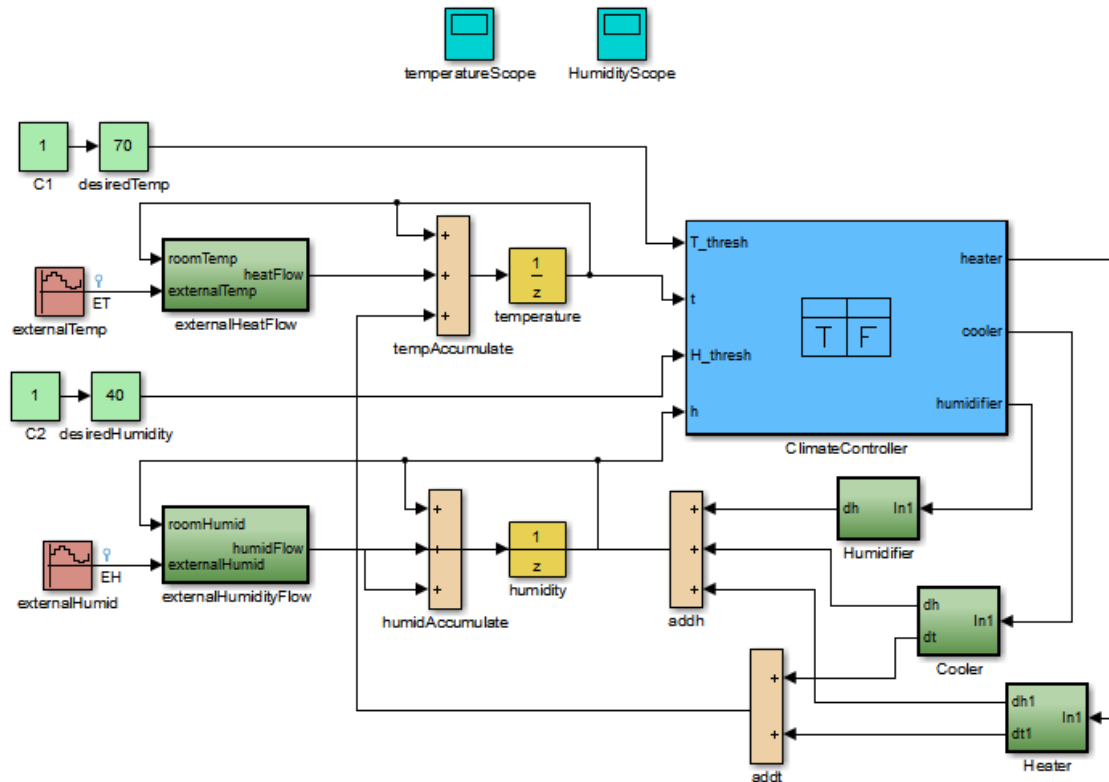
For purely logical behavior, truth tables are easier to program and maintain than graphical functions. Truth tables also provide diagnostics that indicate whether you have too few (underspecified) or too many (overspecified) decisions for the conditions you specify.

The following model, `sf_climate_control`, shows a home environment controller that attempts to maintain a selected temperature and humidity. The model has a Truth Table block, `ClimateController`,



that responds to changes in room temperature (input t) and humidity (input h).

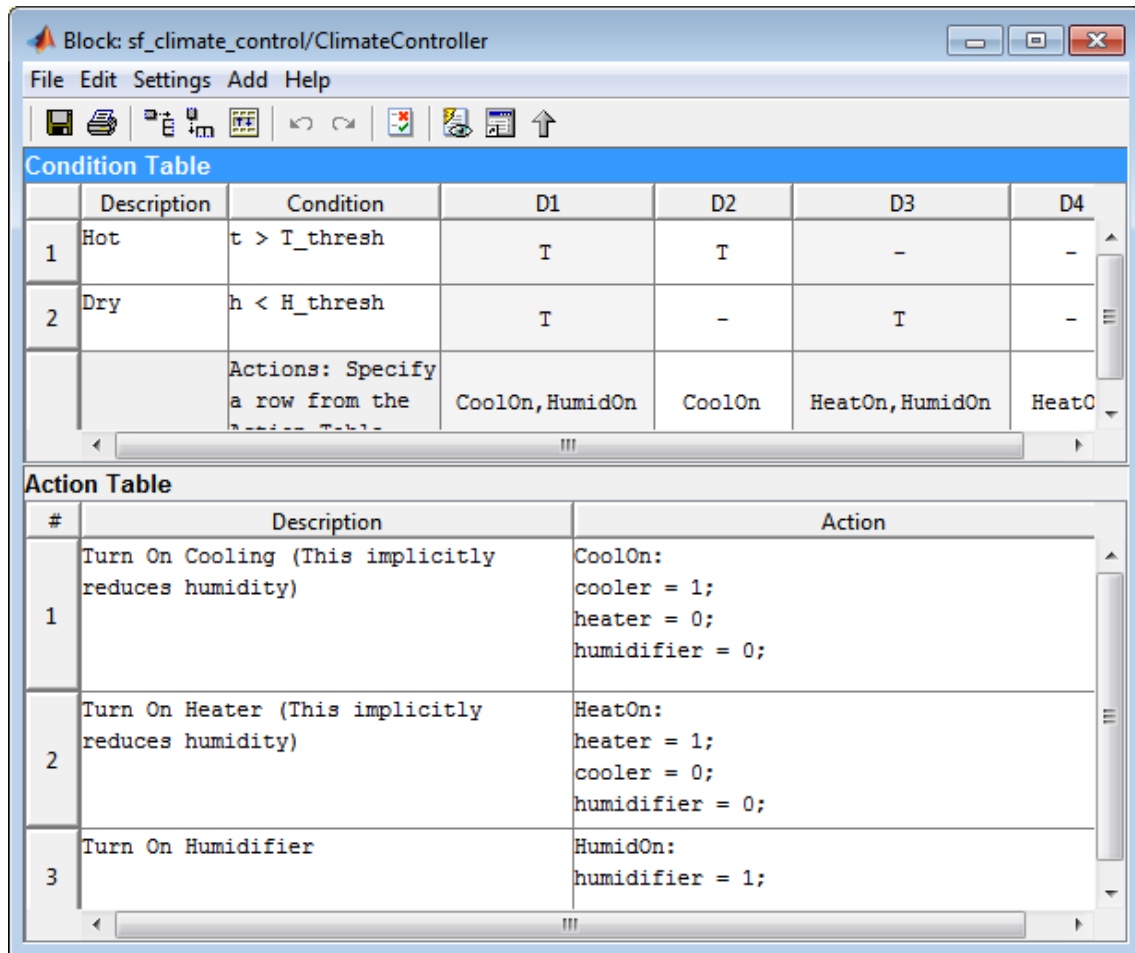
Home climate (temperature and humidity) controller using Truth Table



## Truth Table Editor

If you double-click the Truth Table block in `sf_climate_control`, the Truth Table Editor opens to display its conditions, actions, and decisions. Here is the display for the Truth Table block named `ClimateController`.

# Truth Table



The inputs  $t$  and  $h$  define the conditions, and the outputs `heater`, `cooler`, and `humidifier` define the actions for this Truth Table block.

Using the Truth Table Editor, you can:

- Enter and edit conditions, actions, and decisions



- Add or modify Stateflow data and ports using the Ports and Data Manager
- Run diagnostics to detect parser errors
- View generated content after simulation

For more information about the Truth Table Editor, see “Truth Table Editor Operations”.

## Ports and Data Manager

To add or edit data in a Truth Table block, open the Ports and Data Manager by selecting **Add > Edit Data/Ports** in the Truth Table Editor.

Using the Ports and Data Manager, you can add these elements to a Truth Table block.

Element	Tool	Description
Data		<p>You can add these types of data:</p> <ul style="list-style-type: none"> <li>• Local</li> <li>• Constant</li> <li>• Parameter</li> <li>• Data store memory</li> </ul>
Input trigger		<p>An <i>input trigger</i> causes a Truth Table block to execute when a Simulink control signal changes or through a Simulink block that outputs function-call events. You can use one of these input triggers:</p> <ul style="list-style-type: none"> <li>• Rising edge</li> <li>• Falling edge</li> <li>• Either rising or falling edge</li> </ul>

# Truth Table

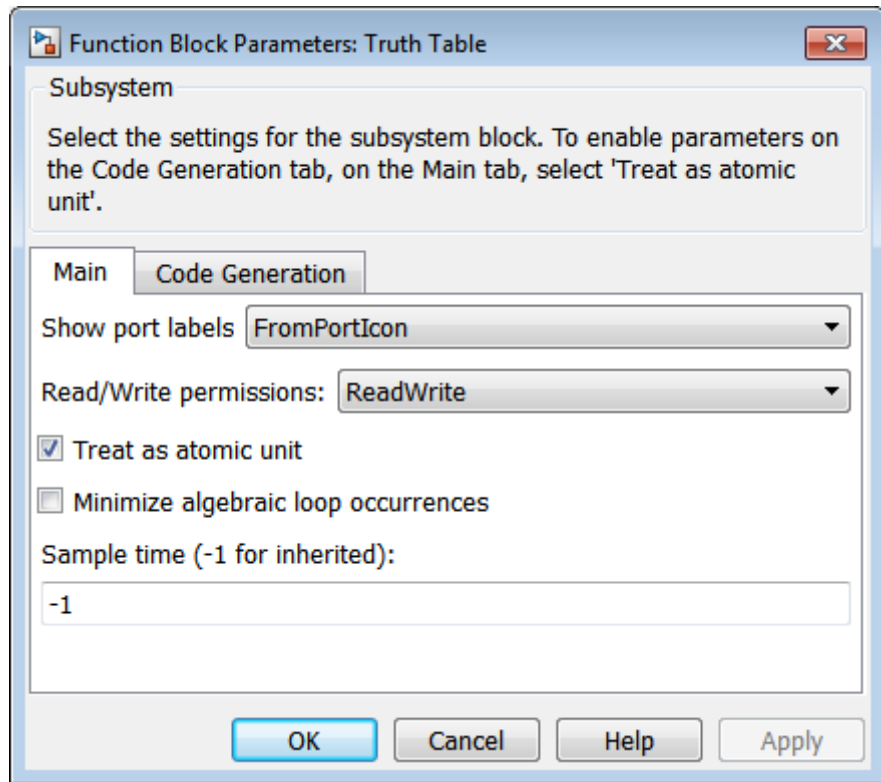
---

Element	Tool	Description
		<ul style="list-style-type: none"><li>Function call</li></ul> <p>For more information, see “Define Events”.</p>
Function-call output	<i>fx</i>	A <i>function-call output</i> triggers a function call to a subsystem. For more information, see “Create a Function-Call Subsystem” in the Simulink documentation.

## Data Type Support

The Truth Table block accepts signals of any data type that Simulink supports, including fixed-point and enumerated data types. The block also accepts frame-based signals. Truth Table blocks work with frame-based signals in the same way as MATLAB Function blocks (see “Add Frame-Based Signals” in the Simulink documentation).

For a discussion of data types that Simulink supports, refer to the Simulink documentation.



## Parameters and Dialog Box

For a description of the block parameters, see the Subsystem block reference page in the Simulink documentation.

## Characteristics

Direct Feedthrough	Yes
Sample Time	Specified in the <b>Sample time</b> parameter
Scalar Expansion	N/A

# Truth Table

---

Dimensionalized	Yes
Zero-Crossing Detection	No