Stateflow®

User's 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

Contents

Create Stateflow Charts

4

Model Logic Patterns and Iterative Loops Using Flow Charts

5

Simulink Subsystems as Stateflow States

6

7

8

Define Events

10

<div align="right">

Messages
</div>

11

Use Actions in Charts

12

MATLAB Syntax Support for States and Transitions

13

Tabular Expression of Modal Logic

14

Save and Restore Simulations with Operating Point

19

20

21

22

23

Define Interfaces to Simulink Models and the MATLAB Workspace

24

Structures and Bus Signals in Stateflow Charts

25

Stateflow Design Patterns

26

Truth Table Functions for Decision-Making Logic

27

28

MATLAB Functions in Stateflow Charts

Code Generation

31

Debug and Test Stateflow Charts

32

33

34

Semantic Rules Summary

A

Semantic Examples

B

Glossary

1

Stateflow Chart Concepts

Finite State Machine Concepts

Stateflow charts can contain sequential decision logic based on state machines. A finite state machine is a representation of an event-driven (reactive) system. In an event-driven system, the system makes a transition from one state (mode) to another, if the condition defining the change is true.

For example, you can 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.

Finite State Machine Representations

Traditionally, designers used truth tables to represent relationships among the inputs, outputs, and states of a finite state machine. The resulting table describes the logic necessary to control the behavior of the system under study. Another approach to designing event-driven systems is to model the behavior of the system by describing it in terms of transitions among states. The occurrence of events under certain conditions determine the state that is active. State-transition charts and bubble charts are graphical representations based on this approach.

Stateflow Chart Representations

A Stateflow chart can contain sequential and combinatorial logic in the form of state transition diagrams, flow charts, state transition tables, and truth tables. A state transition diagram is a graphical representation of a finite state machine. States and transitions form the basic building blocks of a sequential logic system. Another way to represent sequential logic is a state transition table, which allows you to enter the state logic in tabular form. You can also represent combinatorial logic in a chart with flow charts and truth tables.

You can include Stateflow charts as blocks in a Simulink® model. The collection of these blocks in a Simulink model is the Stateflow machine.

A Stateflow chart enables the representation of hierarchy, parallelism, and history. You can organize complex systems by defining a parent and offspring object structure. For example, you can organize states within other higher-level states. A system with parallelism can have two or more orthogonal states active at the same time. You can also specify the destination state of a transition based on historical information.

Notation

Notation defines a set of objects and the rules that govern the relationships between those objects. Stateflow chart notation provides a way to communicate the design information in a Stateflow chart.

Stateflow chart notation consists of these elements:

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

Semantics

Semantics describe how to interpret chart notation. A typical Stateflow chart contains actions associated with transitions and states. The semantics describe the sequence of these actions during chart execution.

See Also

Chart | State Transition Table | Truth Table

More About

- "Overview of Stateflow Objects" on page 2-2
- "Model Reactive Systems in Stateflow" on page 4-2
- "How Stateflow Objects Interact During Execution" on page 3-5
- "Specify Properties for Stateflow Charts" on page 24-3

Stateflow Charts and Simulink Models

The Simulink Model and the Stateflow Machine

A Stateflow chart functions as a finite state machine within a Simulink model. The Stateflow machine is the collection of Stateflow blocks in a Simulink model. The Simulink model and the Stateflow machine work seamlessly together. Running a simulation automatically executes both the Simulink blocks and the Stateflow charts of the model.

A Simulink model can consist of combinations of Simulink blocks, toolbox blocks, and Stateflow blocks (charts). A chart consists of graphical objects (states, boxes, functions, notes, transitions, connective junctions, and history junctions) and nongraphical objects (events, messages, and data).

There is a one-to-one correspondence between the Simulink model and the Stateflow machine. Each Stateflow block in the Simulink model appears as a single Stateflow chart. Each Stateflow machine has its own object hierarchy. The Stateflow machine is the highest level in the Stateflow hierarchy. The object hierarchy beneath the Stateflow machine consists of combinations of graphical and nongraphical objects. See "Hierarchy of Stateflow Objects" on page 1-7.

Overview of Defining Stateflow Block Interfaces to Simulink Models

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

Stateflow charts are event-driven. Events can be local to the Stateflow block or can propagate to and from the Simulink model. Data can be local to the Stateflow block or can pass to and from the Simulink model and external code sources.

Defining the interface for a Stateflow block can involve some or all these tasks:

- Defining the Stateflow block update method
- Defining output events
- Adding and defining nonlocal events and nonlocal data within the Stateflow chart
- Defining relationships with any external sources

In the following example, the Simulink model consists of a Sine Wave block, a Scope block, and a single Stateflow block, titled On_off.
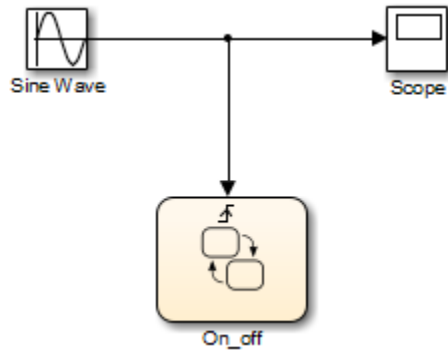
Stateflow Chart Objects

Stateflow charts consist of graphical and nongraphical objects:
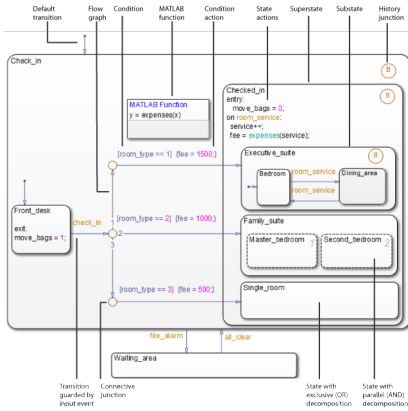
To learn how these objects interact, see "How Stateflow Objects Interact During Execution" on page 3-5.

Hierarchy of Stateflow Objects

Stateflow machines arrange Stateflow objects in a hierarchy based on containment. That is, one Stateflow object can contain other Stateflow objects.

Stateflow Hierarchy
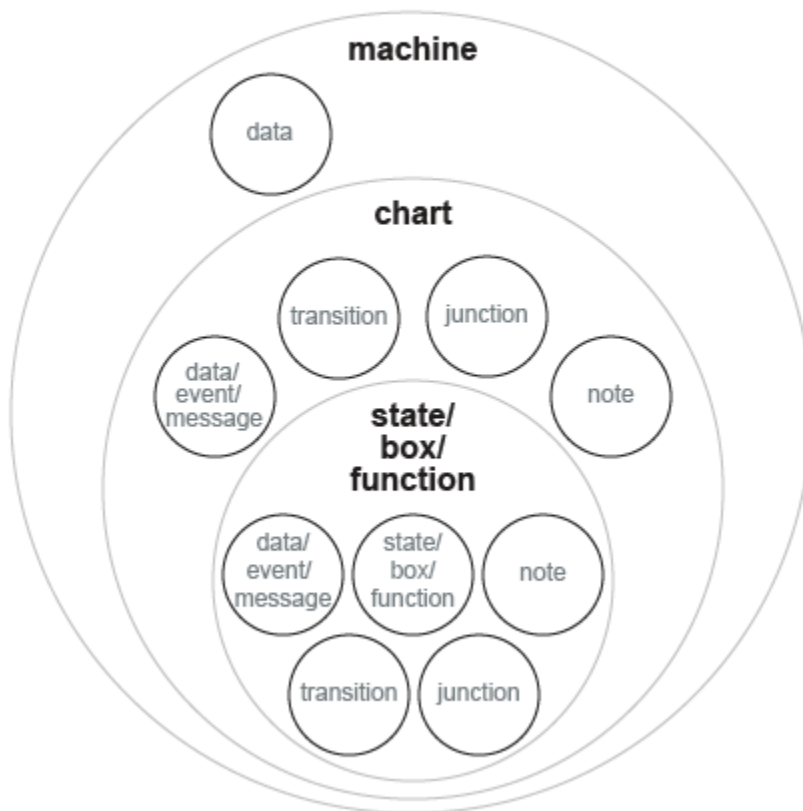
The highest object in Stateflow hierarchy is the Stateflow machine. This object contains all other Stateflow objects in a Simulink model. The Stateflow machine contains all the charts in a model. In addition, the Stateflow machine for a model can contain its own data.

Similarly, charts can contain state, box, function, data, event, message, transition, junction, and note objects. Continuing with the Stateflow hierarchy, states can contain all these objects as well, including other states. You can represent state hierarchy with superstates and substates.

A transition out of a superstate implies transitions out of any of its active substates. Transitions can cross superstate boundaries to specify a substate destination. If a substate becomes active, its parent superstate also becomes active.

You can organize complex charts by defining a containment structure. A hierarchical design usually reduces the number of transitions and produces neat, manageable charts.

- To manage graphical objects, use the Stateflow Editor.
- To manage nongraphical objects, use the Symbols window or Model Explorer.

Bibliography

[1] Hatley, D. J. and I. A. Pirbhai. *Strategies for Real-Time System Specification.* New York, NY: Dorset House Publishing, 1988.

2

Stateflow Chart Notation

Overview of Stateflow Objects

Graphical Objects

The following table lists each type of graphical object you can draw in a chart and the toolbar icon to use for drawing the object.

Type of Graphical Object	Toolbar Icon
State	
Transition	Not applicable. Point and drag from the source state to the destination state.
History junction	
Default transition	
Connective junction	
Truth table function	
Graphical function	
MATLAB® function	
Box	
Simulink based state	
Simulink function	

Nongraphical Objects

You can define data, event, and message objects that do not appear graphically in the Stateflow Editor. However, you can see them in the Symbols window and the Model Explorer. See "Use the Model Explorer with Stateflow Objects" on page 33-13.

Data Objects

A Stateflow chart stores and retrieves data that it uses to control its execution. Stateflow data resides in its own workspace, but you can also access data that resides externally in the Simulink model or application that embeds the Stateflow machine. You must define any internal or external data that you use in a Stateflow chart.

Event Objects

An event is a Stateflow object that can trigger a whole Stateflow chart or individual actions in a chart. Because Stateflow charts execute by reacting to events, you specify and program events into your charts to control their execution. You can broadcast events to every object in the scope of the object sending the event, or you can send an event to a specific object. You can define explicit events that you specify directly, or you can define implicit events to take place when certain actions are performed, such as entering a state. For more information, see "Synchronize Model Components by Broadcasting Events" on page 10-2.

Message Objects

Stateflow message objects are queued objects that can carry data. You can send a message from one Stateflow chart to another to communicate between charts. You can also send local messages within a chart. You define the type of message data. You can view the lifeline of a message in the Sequence Viewer block. For more information, see "Communicate with Stateflow Charts by Sending Messages" on page 11-10.

Rules for Naming Stateflow Objects

Characters You Can Use

You can name Stateflow objects with any combination of alphanumeric and underscore characters. Names cannot begin with a numeric character or contain embedded spaces.

Restriction on Name Length

Name length should comply with the maximum identifier length enforced by Simulink Coder™ software. You can set the **Maximum identifier length** parameter. The default is 31 characters and the maximum length you can specify is 256 characters.

Keywords to Avoid When Naming Chart Objects

You cannot use reserved keywords to name chart objects. These keywords are part of the action language syntax.

Usage in Action Language Syntax	Keywords	Syntax References
Boolean symbols	• `true` • `false`	"Boolean Symbols, true and false" on page 12-22
Change detection	• `hasChanged` • `hasChangedFrom` • `hasChangedTo`	"Detect Changes in Data Values" on page 12-69
Complex data	• `complex` • `imag` • `real`	"Supported Operations for Complex Data" on page 23-4

Usage in Action Language Syntax	Keywords	Syntax References
Data types	• boolean • double • int32 • int16 • int8 • single • uint32 • uint16 • uint8	"Specify Type of Stateflow Data" on page 9-33
Data type operations	• cast • fixdt • type	"Type Cast Operations" on page 12-19
Explicit events	• send	"Broadcast Local Events to Synchronize Parallel States" on page 12-45
Implicit events	• change • chg • tick • wakeup	"Define Chart Behavior by Using Implicit Events" on page 10-33
Messages	• send • forward • discard • isvalid • length • receive	"Communicate with Stateflow Charts by Sending Messages" on page 11-10
Literal symbols	• inf • t (C charts only)	"Supported Symbols in Actions" on page 12-22

Usage in Action Language Syntax	Keywords	Syntax References
MATLAB functions and data	• matlab • ml	"ml Namespace Operator" on page 12-32
State actions	• bind • du • during • en • entry • ex • exit • on	"State Action Types" on page 12-2
State activity	• in	"Check State Activity by Using the in Operator" on page 12-77
Temporal logic	• after • at • before • every • sec • msec • usec • temporalCount • elapsed • t • duration • count	"Control Chart Execution by Using Temporal Logic" on page 12-49

States

A state describes an operating mode of a reactive system. In a Stateflow chart, states are used for sequential design to create state transition diagrams.

States can be active or inactive. The activity or inactivity of a state can change depending on events and conditions. The occurrence of an event drives the execution of the state transition diagram by making states become active or inactive. At any point during execution, active and inactive states exist.

State Hierarchy

To manage multilevel state complexity, use hierarchy in your Stateflow chart. With hierarchy, you can represent multiple levels of subcomponents in a system.

State Hierarchy Example

In the following example, three levels of hierarchy appear in the chart. Drawing one state within the boundaries of another state indicates that the inner state is a substate (or child) of the outer state (or superstate). The outer state is the parent of the inner state.

In this example, the chart is the parent of the state `Car_done`. The state `Car_done` is the parent state of the `Car_made` and `Car_shipped` states. The state `Car_made` is also the parent of the `Parts_assembled` and `Painted` states. You can also say that the states `Parts_assembled` and `Painted` are children of the `Car_made` state.

To represent the Stateflow hierarchy textually, use a slash character (`/`) to represent the chart and use a period (`.`) to separate each level in the hierarchy of states. The following list is a textual representation of the hierarchy of objects in the preceding example:

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

Objects That a State Can Contain

States can contain all other Stateflow objects. Stateflow chart notation supports the representation of graphical object hierarchy in Stateflow charts with containment. A state is a superstate if it contains other states. A state is a substate if it is contained by another state. A state that is neither a superstate nor a substate of another state is a state whose parent is the Stateflow chart itself.

States can also contain nongraphical data, event, and message objects. The hierarchy of this containment appears in the Model Explorer. You define data, event, and message containment by specifying the parent object.

State Decomposition

Every state (or chart) has a decomposition that dictates what type of substates the state (or chart) can contain. All substates of a superstate must be of the same type as the superstate decomposition. State decomposition can be exclusive (OR) or parallel (AND).

Exclusive (OR) State Decomposition

Substates with solid borders indicate exclusive (OR) state decomposition. Use this decomposition to describe operating modes that are mutually exclusive. When a state has exclusive (OR) decomposition, only one substate can be active at a time.

In the following example, either state A or state B can be active. If state A is active, either state A1 or state A2 can be active at a given time.

Parallel (AND) State Decomposition

Substates with dashed borders indicate parallel (AND) decomposition. Use this decomposition to describe concurrent operating modes. When a state has parallel (AND) decomposition, all substates are active at the same time.

In the following example, when state A is active, A1 and A2 are both active at the same time.

The activity within parallel states is essentially independent, as demonstrated in the following example.

In the following example, when state A becomes active, both states B and C become active at the same time. When state C becomes active, either state C1 or state C2 can be active.

State Labels

The label for a state appears on the top left corner of the state rectangle with the following general format:

name/
entry:*entry actions*
during:*during actions*
exit:*exit actions*
on *event_name*:*on event_name actions*
on *message_name*:*on message_name actions*
bind:*events*

The following example demonstrates the components of a state label.
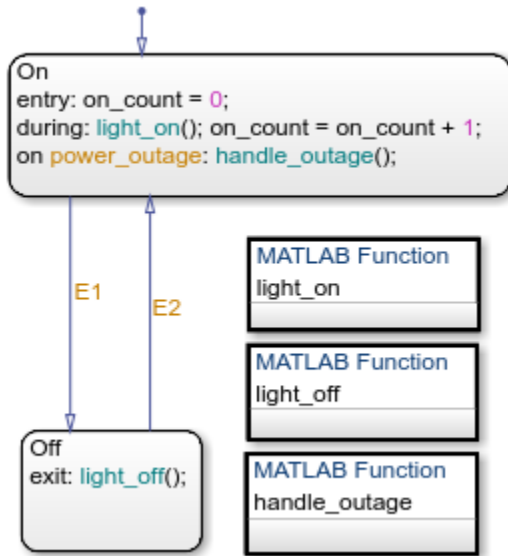
Each action in the state label appears in the subtopics that follow. For more information on state actions, see:

- "Execution of a Stateflow Chart" on page 3-44 — Describes how and when chart execution occurs.
- "State Action Types" on page 12-2 — Gives more detailed descriptions of each type of state action, `entry`, `during`, `exit`.

State Name

A state label starts with the name of the state followed by an optional / character. In the preceding example, the state names are `On` and `Off`. Valid state names consist of alphanumeric characters and can include the underscore (_) character. For more information, see "Rules for Naming Stateflow Objects" on page 2-4.

Hierarchy provides some flexibility in naming states. The name that you enter on the state label must be unique when preceded by ancestor states. The name in the Stateflow hierarchy is the text you enter as the label on the state, preceded by the names of parent states separated by periods. Each state can have the same name appear in the label, as long as their full names within the hierarchy are unique. Otherwise, the parser indicates an error.

The following example shows how unique naming of states works.
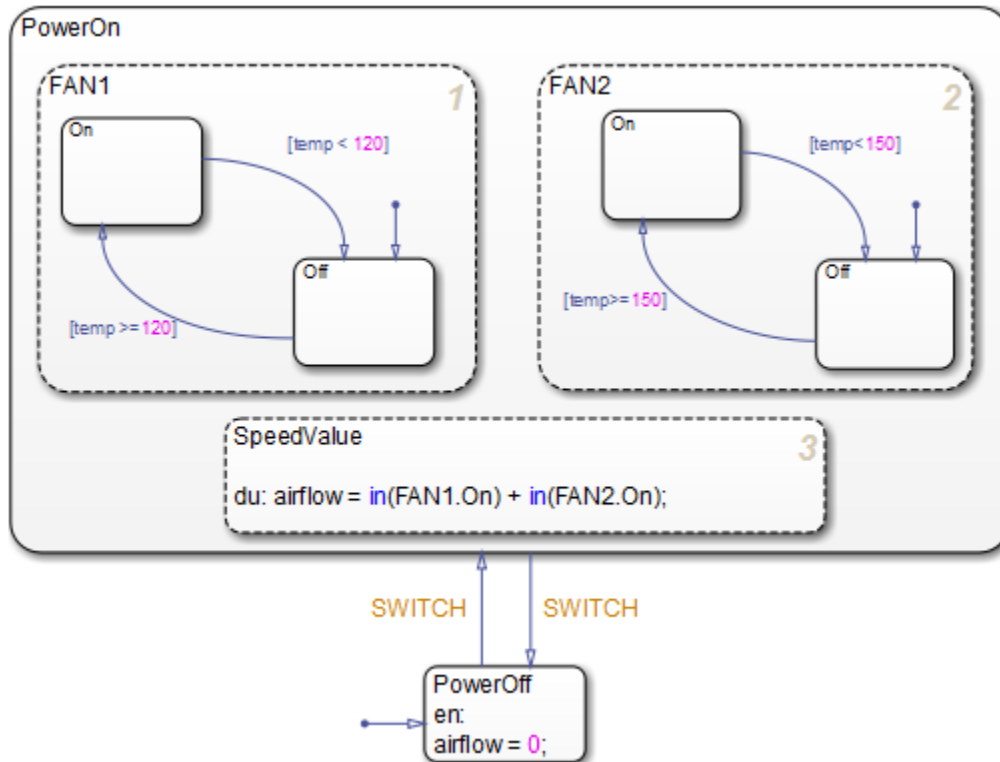
Each of these states has a unique name because of its location in the chart. The full names for the states in FAN1 and FAN2 are:

- PowerOn.FAN1.On
- PowerOn.FAN1.Off
- PowerOn.FAN2.On
- PowerOn.FAN2.Off

State Actions

After the name, you enter optional action statements for the state with a keyword label that identifies the type of action. You can specify none, some, or all of them. The colon

after each keyword is required. The slash following the state name is optional as long as it is followed by a carriage return.

For each type of action, you can enter more than one action by separating each action with a carriage return, semicolon, or a comma. You can specify actions for more than one event or message by adding additional on *event_name* or on *message_name* lines.

If you enter the name and slash followed directly by actions, the actions are interpreted as `entry` action(s). This shorthand is useful if you are specifying only `entry` actions.

Entry Action

Preceded by the prefix `entry` or `en` for short. In the preceding example, state `On` has `entry` action `on_count=0`. This means that the value of `on_count` is reset to 0 whenever state `On` becomes active (entered).

During Action

Preceded by the prefix `during` or `du` for short. In the preceding label example, state `On` has two `during` actions, `light_on()` and `on_count++`. These actions are executed whenever state `On` is already active and any event occurs.

Exit Action

Preceded by the prefix `exit` or `ex` for short. In the preceding label example, state `Off` has the `exit` action `light_off()`. If the state `Off` is active, but becomes inactive (exited), this action is executed.

On Action

Preceded by the prefix `on` *event_name*, or `on` *message_name*. In the preceding label example, state `On` has an `on power_outage` action. If state `On` is active and the event `power_outage` occurs, the action `handle_outage()` is executed.

Bind Action

Preceded by the prefix `bind`. Events bound to a state can only be broadcast by that state or its children.

State Hierarchy

To manage multilevel state complexity, use hierarchy in your Stateflow chart. With hierarchy, you can represent multiple levels of subcomponents in a system.

State Hierarchy Example

In the following example, three levels of hierarchy appear in the chart. Drawing one state within the boundaries of another state indicates that the inner state is a substate (or child) of the outer state (or superstate). The outer state is the parent of the inner state.

In this example, the chart is the parent of the state `Car_done`. The state `Car_done` is the parent state of the `Car_made` and `Car_shipped` states. The state `Car_made` is also the parent of the `Parts_assembled` and `Painted` states. You can also say that the states `Parts_assembled` and `Painted` are children of the `Car_made` state.

To represent the Stateflow hierarchy textually, use a slash character (`/`) to represent the chart and use a period (`.`) to separate each level in the hierarchy of states. The following list is a textual representation of the hierarchy of objects in the preceding example:

- `/Car_done`
- `/Car_done.Car_made`
- `/Car_done.Car_shipped`

- `/Car_done.Car_made.Parts_assembled`
- `/Car_done.Car_made.Painted`

Objects That a State Can Contain

States can contain all other Stateflow objects. Stateflow chart notation supports the representation of graphical object hierarchy in Stateflow charts with containment. A state is a superstate if it contains other states. A state is a substate if it is contained by another state. A state that is neither a superstate nor a substate of another state is a state whose parent is the Stateflow chart itself.

States can also contain nongraphical data, event, and message objects. The hierarchy of this containment appears in the Model Explorer. You define data, event, and message containment by specifying the parent object.

State Decomposition

Every state (or chart) has a decomposition that dictates what type of substates the state (or chart) can contain. All substates of a superstate must be of the same type as the superstate decomposition. State decomposition can be exclusive (OR) or parallel (AND).

Exclusive (OR) State Decomposition

Substates with solid borders indicate exclusive (OR) state decomposition. Use this decomposition to describe operating modes that are mutually exclusive. When a state has exclusive (OR) decomposition, only one substate can be active at a time.

In the following example, either state A or state B can be active. If state A is active, either state A1 or state A2 can be active at a given time.

Parallel (AND) State Decomposition

Substates with dashed borders indicate parallel (AND) decomposition. Use this decomposition to describe concurrent operating modes. When a state has parallel (AND) decomposition, all substates are active at the same time.

In the following example, when state A is active, A1 and A2 are both active at the same time.

The activity within parallel states is essentially independent, as demonstrated in the following example.

In the following example, when state A becomes active, both states B and C become active at the same time. When state C becomes active, either state C1 or state C2 can be active.

Transitions

A transition is a line with an arrowhead that links one graphical object to another. In most cases, a transition represents the passage of the system from one mode (state) to another. A transition typically connects a source and a destination object. The source object is where the transition begins and the destination object is where the transition ends. The following chart shows a transition from a source state, B, to a destination state, A.
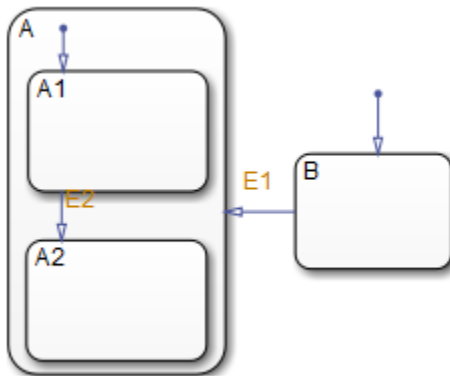
Junctions divide a transition into transition segments. In this case, a full transition consists of the segments taken from the origin to the destination state. Each segment is evaluated in the process of determining the validity of a full transition.

The following example has two segmented transitions: one from state On to state Off, and the other from state On to itself:
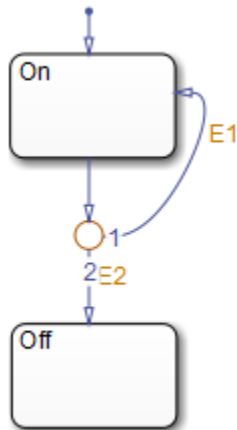
A default transition is a special type of transition that has no source object. See "Default Transitions" on page 2-33 for details.

Transition Hierarchy

Transitions cannot contain other objects the way that states can. However, transitions are contained by states. The hierarchy for a transition is described in terms of its parent, source, and destination states. The parent is the lowest level that contains the source and destination of the transition. Consider the parents for the transitions in the following example:
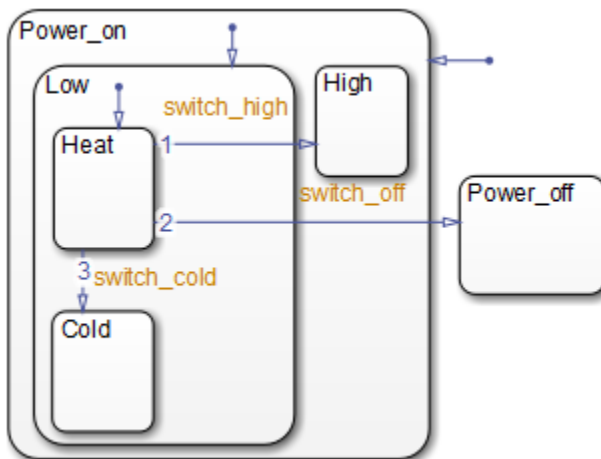
The following table resolves the parentage of each transition in the preceding example. The / character represents the chart. Each level in the hierarchy of states is separated by the period (.) character.

Transition Label	Transition Parent	Transition Source	Transition Destination
switch_off	/	/Power_on.Low.Heat	/Power_off
switch_high	/Power_on	/Power_on.Low.Heat	/Power_on.High
switch_cold	/Power_on.Low	/Power_on.Low.Heat	/Power_on.Low.Cold

Transition Label Notation

A transition label can consist of an event or message, a condition, a condition action, and a transition action. Each part of the label is optional. The ? character is the default transition label. Transition labels have this overall format:

event_or_message[*condition*]{*condition_action*}/*transition_action*

This example illustrates the parts of a transition label.
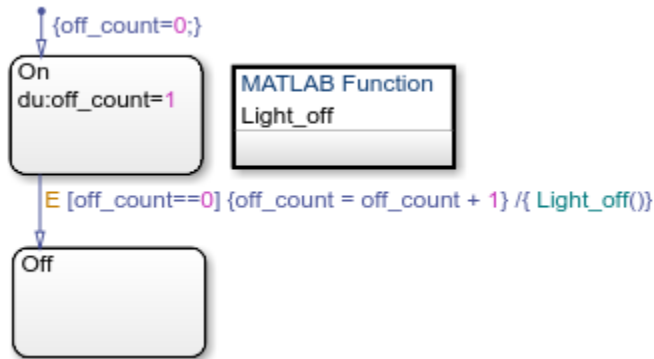
Event or Message Trigger

Specifies an event or message that causes the transition to occur when the condition is true. Specify multiple events using the OR logical operator (|). Specifying an event or message is optional. The absence of an event or message indicates that the transition takes place on the occurrence of any event. For more information, see "Synchronize Model Components by Broadcasting Events" on page 10-2 and "Communicate with Stateflow Charts by Sending Messages" on page 11-10.

In the previous example, the broadcast of event E triggers the transition from On to Off if the condition [off_count==0] is true.

Condition

Specifies a Boolean expression that, when true, validates a transition for the specified event or message trigger. Enclose the condition in square brackets ([]). If no condition is specified, an implied condition evaluates to true. For more information, see "Conditions" on page 12-8.

In the previous example, when the event E occurs, the condition [off_count==0] must evaluate as true for the transition from On to Off to be valid.

Condition Action

Executes after the condition for the transition is evaluated as true, but before the transition to the destination is determined to be valid. Enclose the condition action in curly braces ({}) following the condition. For more information, see "Condition Action Behavior" on page B-10.

In the previous example, if the event E occurs and the condition `[off_count==0]` is true, then the condition action `{off_count = off_count + 1}` is immediately executed.

Transition Action

Executes after the transition to the destination is determined to be valid. If the transition consists of multiple segments, then the transition action is executed when the entire transition path to the final destination is determined to be valid. Transition actions occur after the exit actions of the source state and before the entry actions of the destination state. Precede the transition action with a `/`. For more information, see "Condition and Transition Action Behavior" on page B-11.

In the preceding example, if the event E occurs and the condition `[off_count==0]` is true, then the transition action `{Light_off()}` is executed when the transition from On to Off is determined to be valid. The transition action occurs after On becomes inactive, but before Off becomes active.

Transition actions are supported only in Stateflow charts in Simulink models.

Valid Transitions

Usually, a transition is valid when the source state of the transition is active and the transition label is valid. Default transitions are different because there is no source state. Validity of a default transition to a substate is evaluated when there is a transition to its superstate, assuming the superstate is active. This labeling criterion applies to both default transitions and general case transitions. The following table lists possible combinations of valid transition labels.

Transition Label	Is Valid If...
Event only	That event occurs
Event and condition	That event occurs and the condition is true
Message only	That message occurs
Message and condition	That message occurs and the condition is true
Condition only	Any event occurs and the condition is true
Action only	Any event occurs
Not specified	Any event occurs

Transition Connections

Transitions to and from Exclusive (OR) States

This example shows simple transitions to and from exclusive (OR) states.
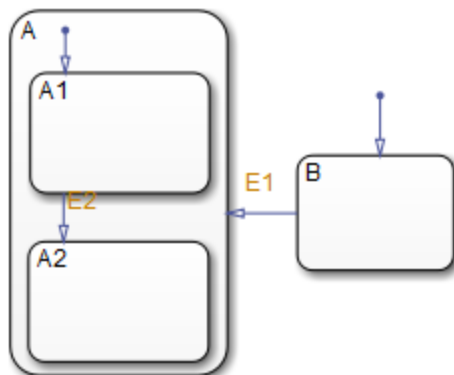
The following transition...	Is valid when...
B to A	State B is active and the event E1 occurs.
A1 to A2	State A1 is active and event E2 occurs.

See "Transition Between Exclusive States" on page B-4 for more information on the semantics of this notation.

Transitions to and from Junctions

The following chart shows transitions to and from connective junctions.
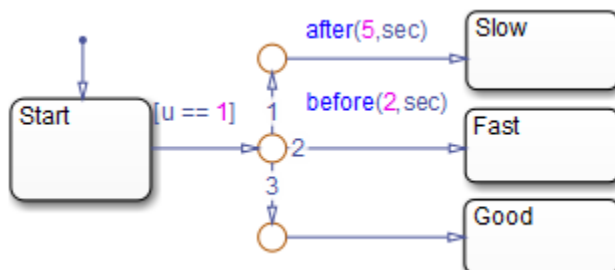
The chart uses temporal logic to determine when the input u equals 1.

If the input equals 1...	A transition occurs from...
Before t = 2	`Start` to `Fast`
Between t = 2 and t = 5	`Start` to `Good`
After t = 5	`Start` to `Slow`

For more information about temporal logic, see "Control Chart Execution by Using Temporal Logic" on page 12-49. For more information on the semantics of this notation, see "Transition from a Common Source to Multiple Destinations" on page B-34.

Transitions to and from Exclusive (OR) Superstates

This example shows transitions to and from an exclusive (OR) superstate and the use of a default transition.
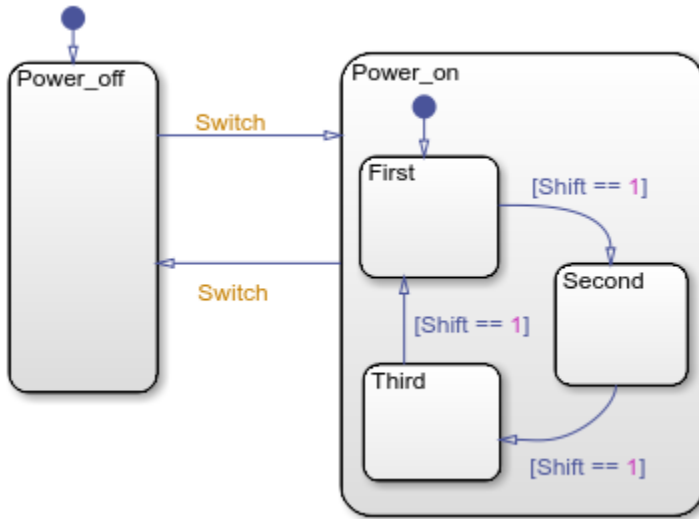
The chart has two states at the highest level in the hierarchy, `Power_off` and `Power_on`. By default, `Power_off` is active. The event `Switch` toggles the system between the `Power_off` and `Power_on` states. `Power_on` has three substates: `First`, `Second`, and `Third`. By default, when `Power_on` becomes active, `First` also becomes active. When `Shift` equals 1, the system transitions from `First` to `Second`, `Second` to `Third`, `Third` to `First`, for each occurrence of the event `Switch`, and then the pattern repeats.

For more information on the semantics of this notation, see "Control Chart Execution by Using Default Transitions" on page B-16.

Transitions to and from Substates

The following example shows transitions to and from exclusive (OR) substates.
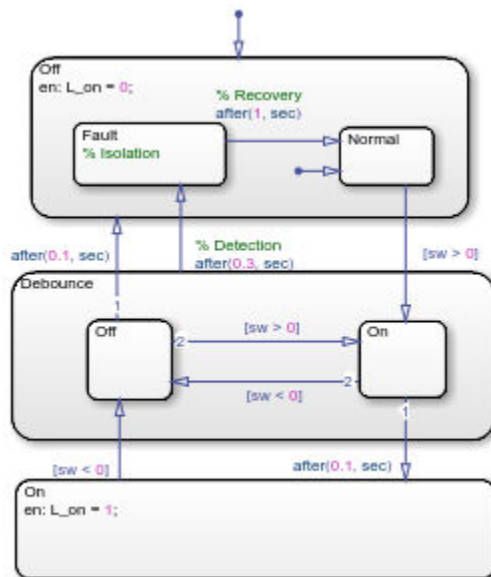
For details on how this chart works, see "Debounce Signals with Fault Detection" on page 26-7. For information on the semantics of this notation, see "Transition from a Substate to a Substate with Events" on page B-8.

See Also

More About
- "Default Transitions" on page 2-33
- "Transition Action Types" on page 12-7
- "Control Chart Execution by Using Condition Actions" on page B-10

- "Synchronize Model Components by Broadcasting Events" on page 10-2
- "Communicate with Stateflow Charts by Sending Messages" on page 11-10

Self-Loop Transitions

A self-loop transition is a transition that originates from and terminates on the same state. The following chart contains four self-loop transitions:

```
MATLAB Function
initBalls
```

```
/initBalls();
```

```
TotalDynamics
du:
    // derivatives
    p_dot = v;
    v_dot = frictionForce() + interactionForce();

    // chart outputs
    p_out = p;
    v_out = v;
    pocketed_out = pocketed;
```

1 [isAnyBallGoingToStop()]
 /updateStopFlags();

2 [hasBallInteractionChanged()]
 /ball_interaction = getBallInteraction();
 updateStopFlags();

3 [isAnyBallNewlyPocketed()]
 /pocketNewBalls();
 updateStopFlags();

4 [isAnyBallOutOfBounds()]
 /resetBallsPosAndVel();
 updateStopFlags();

```
MATLAB Function
yn = isAnyBallOutOfBounds
```

```
MATLAB Function  f = interactionForce
```

```
MATLAB Function
balli = getBallInteraction
```

```
MATLAB Function  resetBallsPosAndVel
```

```
MATLAB Function
yn = hasBallInteractionChanged
```

```
MATLAB Function
yn = isAnyBallNewlyPocketed
```

```
MATLAB Function  pocketNewBalls
```

```
MATLAB Function
yn = nearHole(pp)
```

```
MATLAB Function
yn = isAnyBallGoingToStop
```

```
MATLAB Function  updateStopFlags
```

```
MATLAB Function
f = frictionForce
```

See these sections for more information about the semantics of this notation:

- "Self-Loop Transition" on page B-30
- "For-Loop Construct" on page B-31

Inner Transitions

An inner transition is a transition that does not exit the source state. Inner transitions are powerful when defined for superstates with exclusive (OR) decomposition. Use of inner transitions can greatly simplify a Stateflow chart, as shown by the following examples:

Before Using an Inner Transition

This chart is an example of how you can simplify logic using an inner transition.
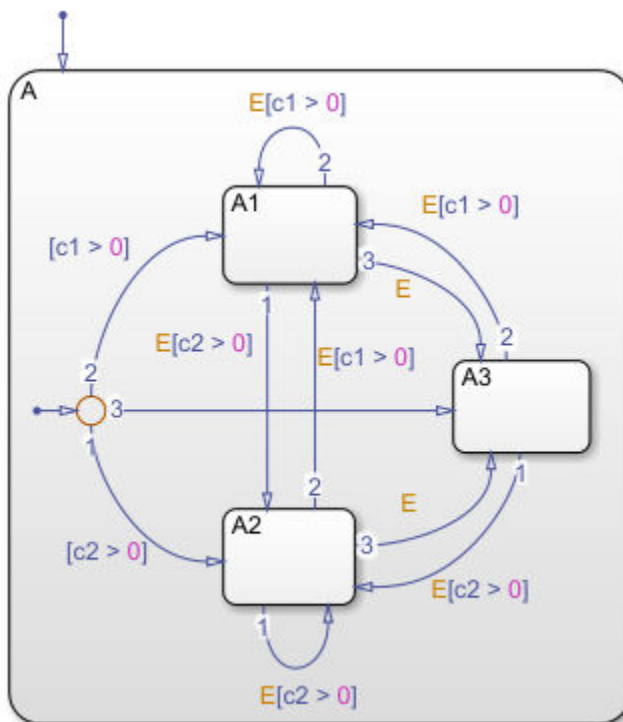
Any event occurs and awakens the Stateflow chart. The default transition to the connective junction is valid. The destination of the transition is determined by [c1 > 0] and [c2 > 0]. If [c1 > 0] is true, the transition to A1 is true. If [c2 > 0] is true, the transition to A2 is valid. If neither [c1 > 0] nor [c2 > 0] is true, the transition to A3 is valid. The transitions among A1, A2, and A3 are determined by E, [c1 > 0], and [c2 > 0].

After Using an Inner Transition to a Connective Junction

This example simplifies the preceding example using an inner transition to a connective junction.
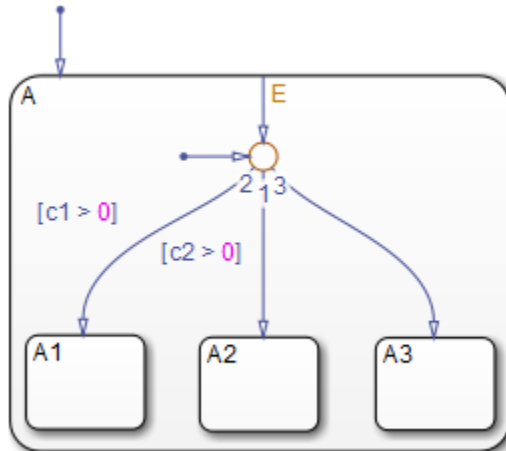
An event occurs and awakens the chart. The default transition to the connective junction is valid. The destination of the transitions is determined by [c1 > 0] and [c2 > 0].

You can simplify the chart by using an inner transition in place of the transitions among all the states in the original example. If state A is already active, the inner transition is used to reevaluate which of the substates of state A is to be active. When event E occurs, the inner transition is potentially valid. If [c1 > 0] is true, the transition to A1 is valid. If [c2 > 0] is true, the transition to A2 is valid. If neither [c1 > 0] nor [c2 > 0] is true, the transition to A3 is valid. This chart design is simpler than the previous one.

Note When you use an inner transition to a connective junction, an active substate can exit and reenter when the transition condition for that substate is valid. For example, if substate A1 is active and [c1 > 0] is true, the transition to A1 is valid. In this case:

1   Exit actions for A1 execute and complete.

2   A1 becomes inactive.

3   A1 becomes active.

4   Entry actions for A1 execute and complete.

See "Process the First Event with an Inner Transition to a Connective Junction" on page B-24 for more information on the semantics of this notation.

Using an Inner Transition to a History Junction

This example shows an inner transition to a history junction.
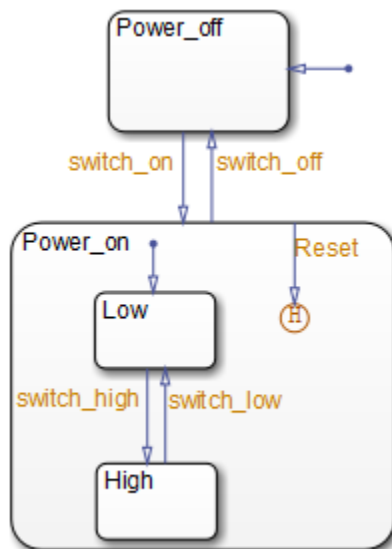
State Power_on.High is initially active. When event Reset occurs, the inner transition to the history junction is valid. Because the inner transition is valid, the currently active state, Power_on.High, is exited. When the inner transition to the history junction is

2-31

processed, the last active state, `Power_on.High`, becomes active (is reentered). If `Power_on.Low` was active under the same circumstances, `Power_on.Low` would be exited and reentered as a result. The inner transition in this example is equivalent to drawing an outer self-loop transition on both `Power_on.Low` and `Power_on.High`.

See "Example of History Junctions" on page 2-44 for another example using a history junction.

See "Inner Transition to a History Junction" on page B-26 for more information on the semantics of this notation.

Default Transitions

A default transition specifies which exclusive (OR) state to enter when there is ambiguity among two or more neighboring exclusive (OR) states. A default transition has a destination but no source object. For example, a default transition specifies which substate of a superstate with exclusive (OR) decomposition the system enters by default, in the absence of any other information, such as a history junction. A default transition can also specify that a junction should be entered by default.

Drawing Default Transitions

Click the **Default transition** button in the toolbar, and click a location in the drawing area close to the state or junction you want to be the destination for the default transition. Drag the mouse to the destination object to attach the default transition. In some cases, it is useful to label default transitions.

A common programming mistake is to create multiple exclusive (OR) states without a default transition. In the absence of the default transition, there is no indication of which state becomes active by default. Note that this error is flagged when you simulate the model with the **State Inconsistencies** option enabled.

Label Default Transitions

You can label default transitions as you would other transitions. For example, you might want to specify that one state or another should become active depending upon the event that has occurred. In another situation, you might want to have specific actions take place that are dependent upon the destination of the transition.

Tip When labeling default transitions, ensure that there is at least one valid default transition. Otherwise, a chart can transition into an inconsistent state.

Default Transition Examples

The following examples show the use of default transitions in Stateflow charts:

- "Default Transition to a State Example" on page 2-34
- "Default Transition to a Junction Example" on page 2-34

- "Default Transition with a Label Example" on page 2-35

Default Transition to a State Example

This example shows a default transition to a state.
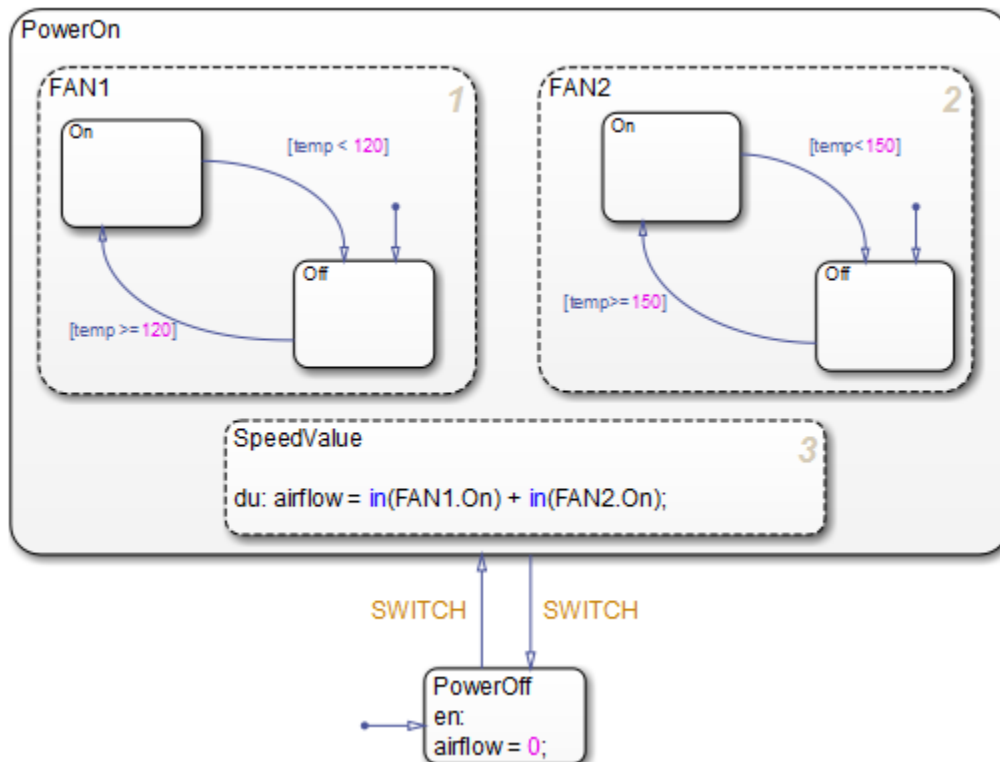
Without the default transition to state `PowerOff`, when the Stateflow chart wakes up, none of the states becomes active. A state inconsistency error is reported at run time.

See "Control Chart Execution by Using Default Transitions" on page B-16 for information on the semantics of this notation.

Default Transition to a Junction Example

This example shows a default transition to a connective junction.
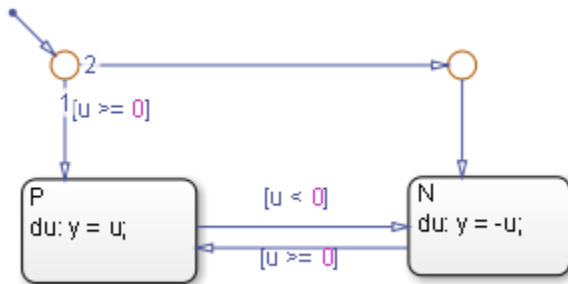
The default transition to the connective junction defines that upon entering the chart, the destination depends on the condition of each transition segment.

See "Default Transition to a Junction" on page B-17 for information on the semantics of this notation.

Default Transition with a Label Example

This example shows a default transition with a label.
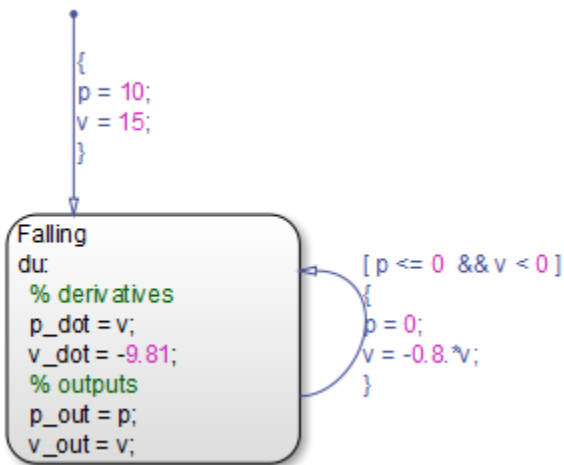
When the chart wakes up, the data p and v initialize to 10 and 15, respectively.

See "Labeled Default Transitions" on page B-19 for more information on the semantics of this notation.

Connective Junctions

A connective junction enables representation of different possible transition paths for a single transition. Connective junctions are used to help represent the following:

- Variations of an `if-then-else` decision construct, by specifying conditions on some or all of the outgoing transitions from the connective junction
- A self-loop transition back to the source state if none of the outgoing transitions is valid
- Variations of a `for` loop construct, by having a self-loop transition from the connective junction back to itself
- Transitions from a common source to multiple destinations
- Transitions from multiple sources to a common destination
- Transitions from a source to a destination based on common events

> **Note** An event cannot trigger a transition from a connective junction to a destination state.

See "Represent Multiple Paths by Using Connective Junctions" on page B-28 for a summary of the semantics of connective junctions.

Flow Chart Notation with Connective Junctions

Flow chart notation uses connective junctions to represent common code structures like `for` loops and `if-then-else` constructs without the use of states. And by reducing the number of states in your Stateflow charts, flow chart notation produces efficient simulation and generated code that helps optimize memory use.

Flow chart notation uses combinations of the following:

- Transitions to and from connective junctions
- Self-loops to connective junctions
- Inner transitions to connective junctions

Flow chart notation, states, and state-to-state transitions coexist in the same Stateflow chart. The key to representing flow chart notation is in the labeling of transitions, as shown in the following examples.

Connective Junction with All Conditions Specified Example
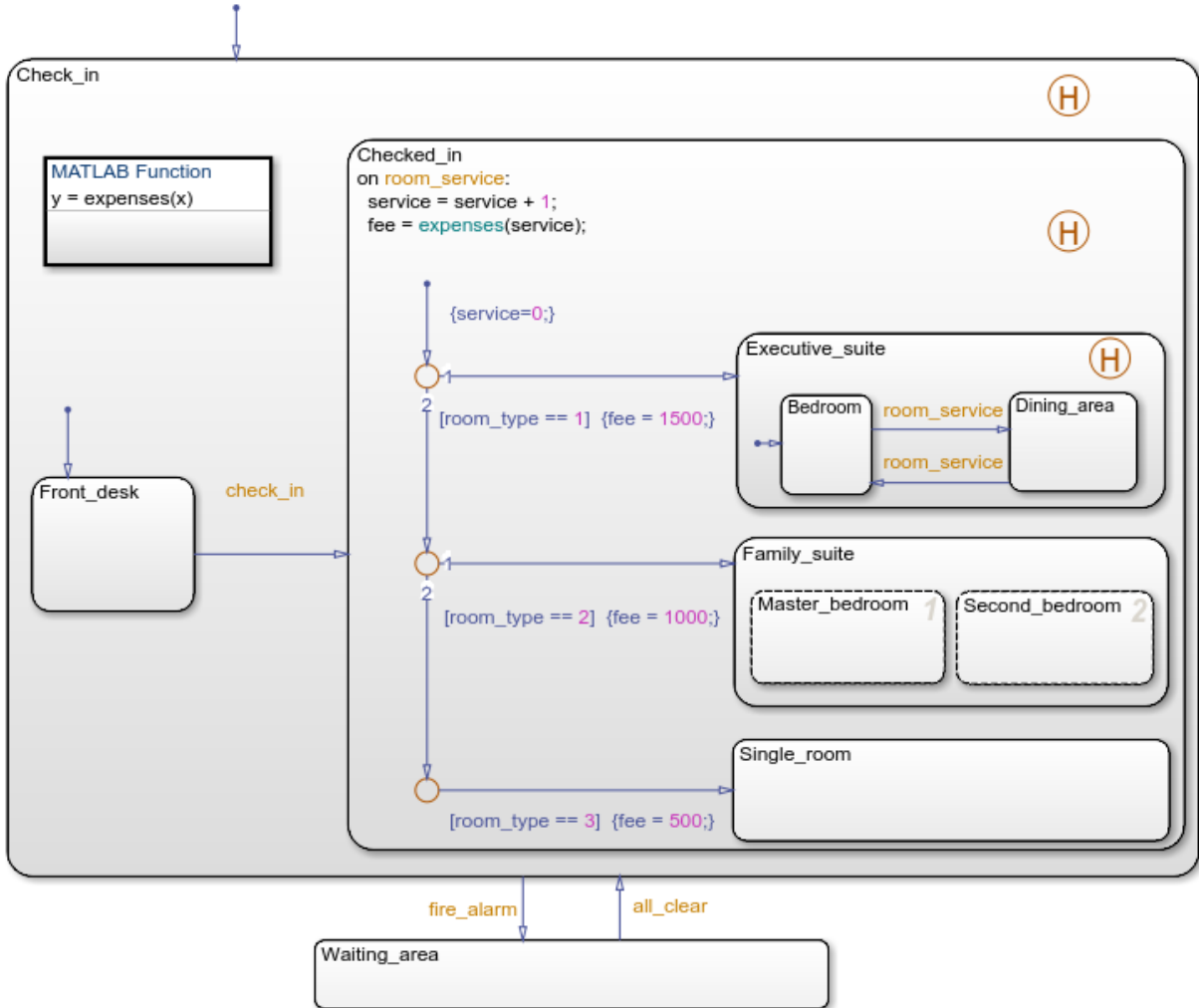
A transition from the `Front_desk` state to a connective junction is labeled by the `check_in` event. Transitions from the connective junction to the destination states are labeled with conditions. If `Front_desk` is active when `check_in` occurs, the transition from `Front_desk` to the connective junction occurs first. The transition from the

connective junction to a destination state depends on which of the `room_type` conditions is true. If none of the conditions is true, no transition occurs and `Front_desk` remains active.

For more information about this chart, see "Phases of Chart Execution" on page 3-9. For more information on the semantics of this notation, see "If-Then-Else Decision Construct" on page B-29.

Connective Junction with One Unconditional Transition Example
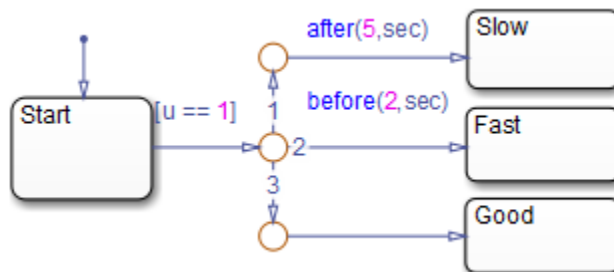
The chart uses temporal logic to determine when the input u equals 1.

If the input equals 1...	A transition occurs from...
Before t = 2	`Start` to `Fast`
Between t = 2 and t = 5	`Start` to `Good`
After t = 5	`Start` to `Slow`

For more information about temporal logic, see "Control Chart Execution by Using Temporal Logic" on page 12-49. For more information on the semantics of this notation, see "If-Then-Else Decision Construct" on page B-29.

Connective Junction and For Loops Example

This example shows a combination of flow chart notation and state transition notation. Self-loop transitions to connective junctions can represent `for` loop constructs. The chart uses implicit ordering of outgoing transitions (see "Implicit Ordering" on page 3-65).
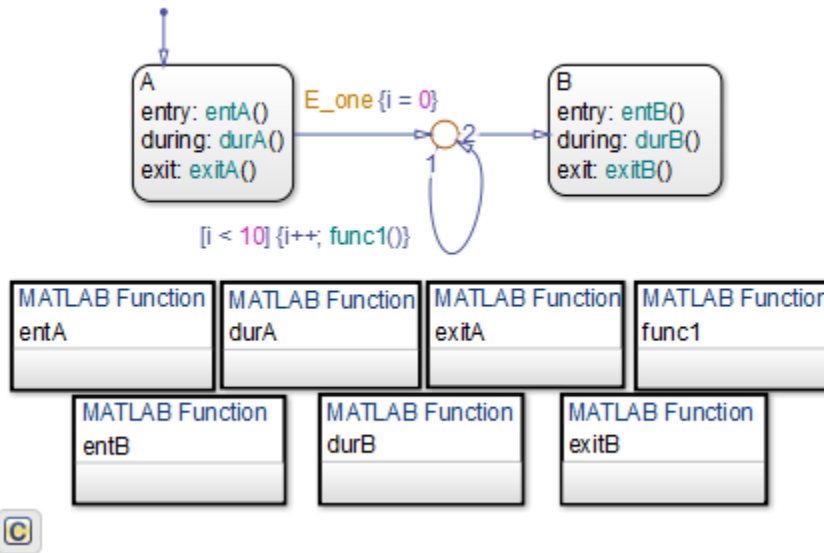
See "For-Loop Construct" on page B-31 for information on the semantics of this notation.

Flow Chart Notation Example

This example shows the use of flow chart notation. The chart uses implicit ordering of outgoing transitions (see "Implicit Ordering" on page 3-65).
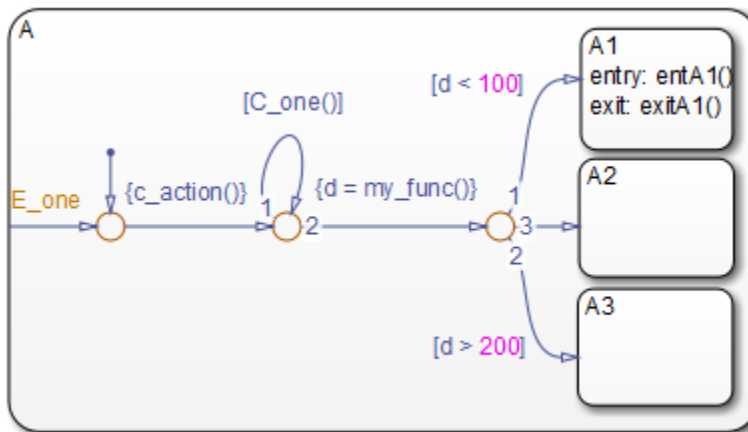
See "Flow Chart Notation" on page B-32 for information on the semantics of this notation.

Connective Junction from a Common Source to Multiple Destinations Example

This example shows transition segments from a common source to multiple conditional destinations using a connective junction. The chart uses implicit ordering of outgoing transitions (see "Implicit Ordering" on page 3-65).
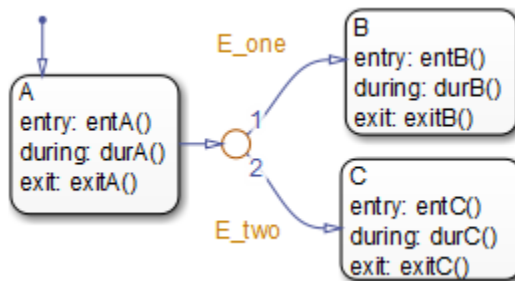
See "Transition from a Common Source to Multiple Destinations" on page B-34 for information on the semantics of this notation.

Connective Junction Common Events Example

This example shows transition segments from multiple sources to a single destination based on the same event using a connective junction.
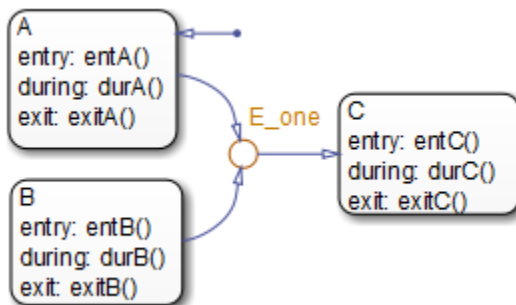
See "Transition from a Source to a Destination Based on a Common Event" on page B-37 for information on the semantics of this notation.

Change Connective Junction Size

1 Select one or more connective junctions.

2 Right-click one of the selected junctions and select **Junction Size** from the drop-down menu.

3 From the menu, select a junction size.

Modify Connective Junction Properties

1 Right-click a connective junction and select **Properties** from the drop-down menu.

2 In the Connective Junction dialog box, edit the fields in the dialog box according to your requirements.

Field	Description
Parent	Parent of the connective junction (read-only). To bring the parent to the foreground, click the hypertext link.
Description	Textual description or comment.
Document link	Link to other information. Enter a URL address or a general MATLAB command. Examples are `www.mathworks.com`, `mailto:`*email_address*, and `edit/spec/data/speed.txt`.

3   To save changes, click **Apply**.

History Junctions

A history junction represents historical decision points in the Stateflow chart. The decision points are based on historical data relative to state activity. Placing a history junction in a superstate indicates that historical state activity information is used to determine the next state to become active. The history junction applies only to the level of the hierarchy in which it appears.

Example of History Junctions

The following example uses a history junction:
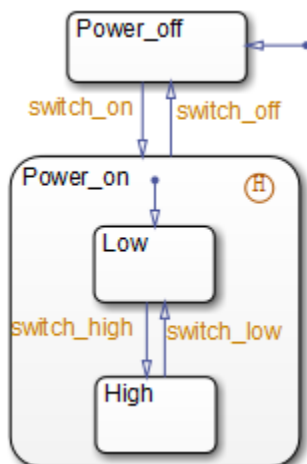
Superstate `Power_on` has a history junction and contains two substates. If state `Power_off` is active and event `switch_on` occurs, the system can enter `Power_on.Low` or `Power_on.High`. The first time superstate `Power_on` is entered, substate `Power_on.Low` is entered because it has a default transition. At some point afterward, if state `Power_on.High` is active and event `switch_off` occurs, superstate `Power_on` is exited and state `Power_off` becomes active. Then event `switch_on` occurs. Because `Power_on.High` was the last active substate, it becomes active again. After the first time `Power_on` becomes active, the history junction determines whether to enter `Power_on.Low` or `Power_on.High`.

See "Default Transition and a History Junction" on page B-17 for more information on the semantics of this notation.

History Junctions and Inner Transitions

By specifying an inner transition to a history junction, you can specify that, based on a specified event or condition, the active state is to be exited and then immediately reentered.

See "Using an Inner Transition to a History Junction" on page 2-31 for an example of this notation.

See "Inner Transition to a History Junction" on page B-26 for more information on the semantics of this notation.

Reusable Functions in Charts

State actions and transition conditions can be complicated enough that defining them inline on the state or transition is not feasible. In this case, express the conditions or actions using one of the following types of Stateflow functions:

- Flow chart — Encapsulate flow charts containing if-then-else, switch-case, for, while, or do-while patterns. For more information, see "Reuse Logic Patterns by Defining Graphical Functions" on page 8-16.

- MATLAB — Write matrix-oriented algorithms; call MATLAB functions for data analysis and visualization. For more information, see "Reuse MATLAB Code by Defining MATLAB Functions" on page 28-2.

- Simulink — Call Simulink function-call subsystems directly to streamline design and improve readability. For more information, see "Reuse Simulink Components in Stateflow Charts" on page 29-2.

- Truth table — Represent combinational logic for decision-making applications such as fault detection and mode switching. For more information, see "Reuse Combinatorial Logic by Defining Truth Tables" on page 27-2.

Use the function format that is most natural for the type of calculation required in the state action or transition condition.

If the four standard types of Stateflow functions do not work, you can write your own C or C++ code for integration with your chart. For more information about custom code integration, see "Reuse Custom Code in Stateflow Charts" on page 30-26.

See Also

More About

- "Reuse Logic Patterns by Defining Graphical Functions" on page 8-16
- "Reuse MATLAB Code by Defining MATLAB Functions" on page 28-2
- "Reuse Combinatorial Logic by Defining Truth Tables" on page 27-2
- "Reuse Simulink Components in Stateflow Charts" on page 29-2
- "Reuse Custom Code in Stateflow Charts" on page 30-26

3

Stateflow Semantics

Stateflow Semantics

In Stateflow, semantics describe the execution behavior of your Stateflow chart. Various factors can affect how your chart executes, including:

- Explicit or implicit ordering of states
- Transition ordering between states
- Events sent by parallel or superstates

As you build your chart, you expect it to behave in a certain way. By knowing how these factors affect your chart, you can create a chart that behaves with intentional interaction of the graphical and nongraphical objects. Graphical and nongraphical objects are the building blocks for all Stateflow charts.

Stateflow Objects

Stateflow objects are the building blocks of Stateflow charts. These objects can be categorized as either graphical or nongraphical. Graphical objects consist of objects that appear graphically in a chart. Nongraphical objects appear textually in a chart and often refer to data, events, and messages. This chart shows a variety of both graphical and nongraphical objects.
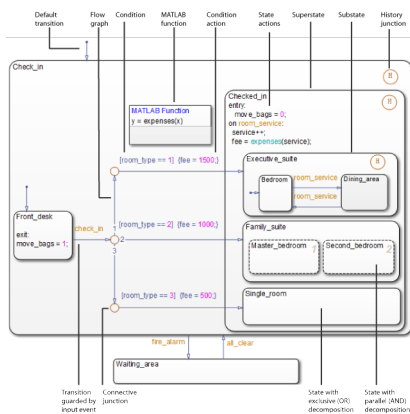
Graphical Objects

To build graphical objects, use the object palette in the Stateflow Editor (see "Stateflow Editor Operations" on page 4-28).

Graphical Objects	Types	References
Flow charts	Decision logic patterns	"Flow Charts in Stateflow" on page 5-2
	Loop logic patterns	
Functions	Graphical functions	"Reuse Logic Patterns by Defining Graphical Functions" on page 8-16
	MATLAB functions	"Reuse MATLAB Code by Defining MATLAB Functions" on page 28-2
	Truth table functions	"Reuse Combinatorial Logic by Defining Truth Tables" on page 27-2
	Simulink functions	"Reuse Simulink Components in Stateflow Charts" on page 29-2
Junctions	Connective junctions	"Connective Junctions" on page 2-37
	History junctions	"History Junctions" on page 2-44
States	States with exclusive (OR) decomposition	"Exclusive (OR) State Decomposition" on page 2-8
	States with parallel (AND) decomposition	"Parallel (AND) State Decomposition" on page 2-9
	Substates and superstates	"Create Substates and Superstates" on page 4-6
Transitions	Default transitions	"Default Transitions" on page 2-33
	Object-to-object transitions	"Transition Connections" on page 2-23
	Inner transitions	
	Self-loop transitions	

Nongraphical Objects

You create nongraphical objects textually in your chart. See "Add Stateflow Data" on page 9-2, "Define Events in a Chart" on page 10-3, and "Define Messages in a Chart" on page 11-10 for details. Examples of nongraphical objects include:

Nongraphical Object	Description	Reference
Condition	Boolean expression that specifies that a transition path is valid if the expression is true; part of a transition label	"Transition Label Notation" on page 2-20 and "Conditions" on page 12-8
Condition action	Action that executes as soon as the condition evaluates to true; part of a transition label	"Transition Label Notation" on page 2-20 and "Condition Actions" on page 12-9
State actions	Expressions that specify actions to take when a state is active, such as initializing or updating data; part of a state label	"State Labels" on page 2-10 and "State Action Types" on page 12-2
Function calls	Expression used to activate a specific function within a chart.	"Reuse MATLAB Code by Defining MATLAB Functions" on page 28-2 and "Reuse Simulink Components in Stateflow Charts" on page 29-2
Temporal logic statements	Operators that are used to control chart actions.	"Control Chart Execution by Using Temporal Logic" on page 12-49

See Also

More About

- "Graphical Objects" on page 2-2
- "Nongraphical Objects" on page 2-2
- "Types of Chart Execution" on page 3-41

How Stateflow Objects Interact During Execution

During execution, Stateflow objects interact with each other to simulate real-world behavior. In the following model, different Stateflow objects are shown and their interactions are explained.

Overview of the Example Model

The example model shows how common graphical and nongraphical objects in a chart interact during execution. These objects include:

- Conditions and condition actions
- Exclusive (OR) states
- Flow charts
- Function calls
- History junctions
- Parallel (AND) states
- State actions
- Transitions guarded by input events

For details of the chart semantics, see "Phases of Chart Execution" on page 3-9.

Model of the Check-In Process for a Hotel

This example uses the hotel check-in process to explain Stateflow chart semantics.

The `sf_semantics_hotel_checkin` model consists of four Manual Switch blocks, one Mux block, one Multiport Switch block, a Hotel chart, and a Display block.

The model uses this block...	To...	Because...
Manual Switch	Enable toggling between two settings during simulation without having to pause or restart.	During simulation, you can interactively trigger the chart by sending one of these input events: • Checking in to a hotel • Calling room service • Triggering a fire alarm • Sending an all-clear signal after a fire alarm

The model uses this block...	To...	Because...
Mux	Combine multiple input signals into a vector.	A chart can support multiple input events only if they connect to the trigger port of a chart as a vector of inputs.
Multiport Switch	Enable selection among more than two inputs.	This block provides a value for the chart input data `room_type`, where each room type corresponds to a number (1, 2, or 3). A Manual Switch block cannot support more than two inputs, but a Multiport Switch block can.
Display	Show up-to-date numerical value for input signal.	During simulation, any change to the chart output data `fee` appears in the display.

The Hotel chart contains graphical objects, such as states and history junctions, and nongraphical objects, such as conditions and condition actions.
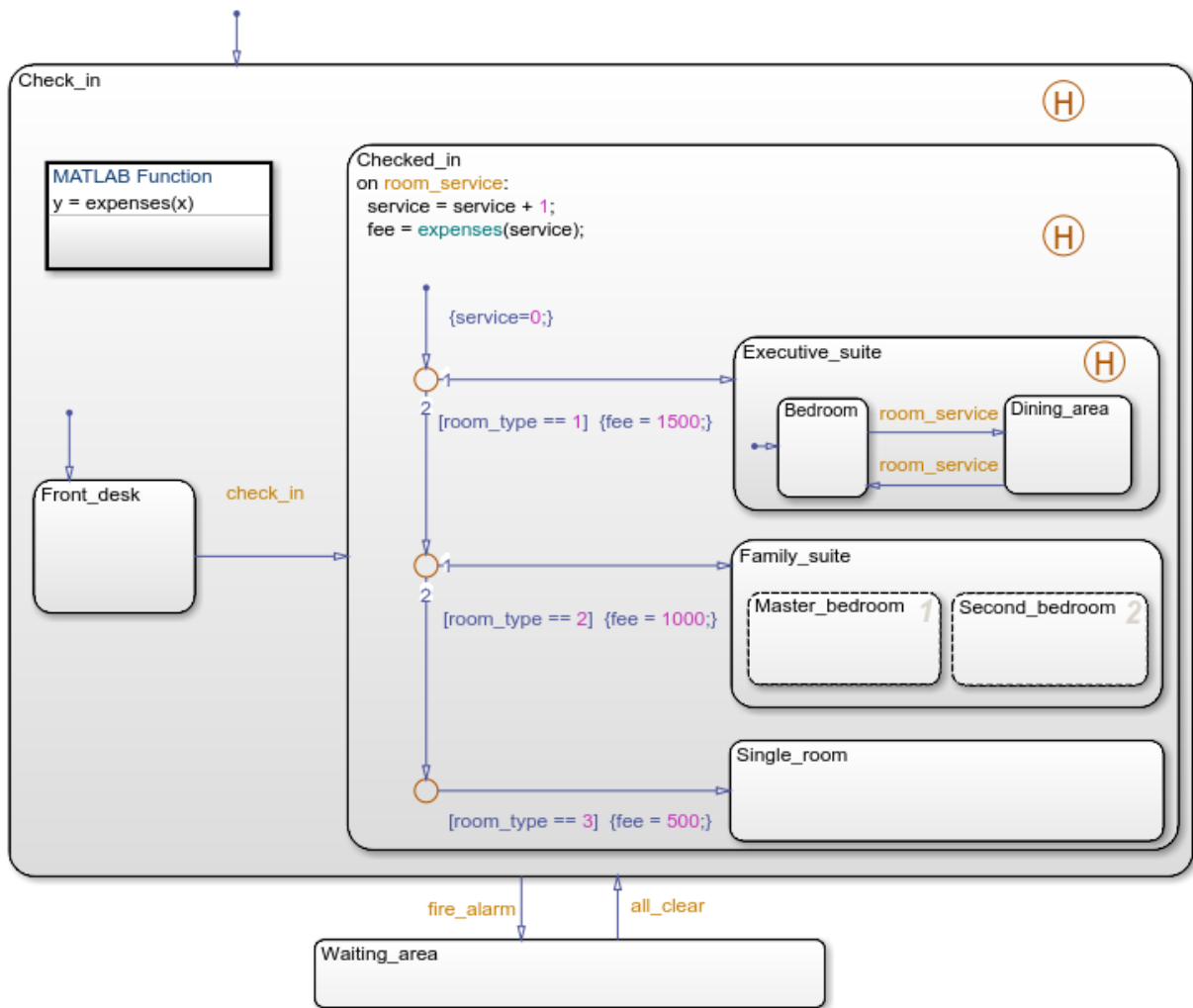
For a mapping of objects to their locations in the chart, see "Stateflow Objects" on page 3-2.

How the Chart Interacts with Simulink Blocks

Chart Initialization

When simulation starts, the chart wakes up and executes its default transitions because the **Execute (enter) Chart At Initialization** option is on (see "Execution of a Chart at Initialization" on page 3-42). Then the chart goes to sleep.

Note If this option is off, the chart does not wake up until you toggle one of the Manual Switch blocks. You can verify the setting for this option in the Chart properties dialog box. Right-click inside the top level of the chart and select **Properties** from the context menu.

Chart Interaction with Other Blocks

The chart wakes up again only when an *edge-triggered* input event occurs: `check_in`, `room_service`, `fire_alarm`, or `all_clear`. When you toggle a Manual Switch block for an input event during simulation, the chart detects a rising or falling edge and wakes up. While the chart is awake:

- The Multiport Switch block provides a value for the chart input data `room_type`.
- The Display block shows any change in value for the chart output data `fee`.

Chart Inactivity

After completing all possible phases of execution, the chart goes back to sleep.

Phases of Chart Execution

The following sections explain chart execution for each shaded region of the Hotel chart.
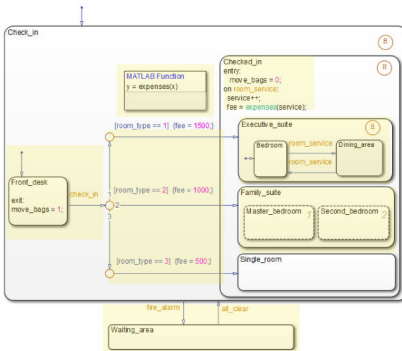
Phase: Chart Initialization

This section describes what happens in the Front_desk state just after the chart wakes up.
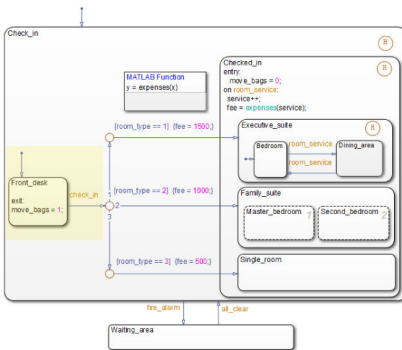
Stage	Hotel Scenario	Chart Behavior
1	Your first stop is at the front desk of the hotel.	At the chart level, the default transition to Check_in occurs, making that state active. Then, the default transition to Front_desk occurs, making that state active. For reference, see "Enter a Chart or State" on page 3-50.

Stage	Hotel Scenario	Chart Behavior
2	You leave the front desk after checking in to the hotel.	The `check_in` event guards the outgoing transition from `Front_desk`. When the chart receives an event broadcast for `check_in`, the transition becomes valid. For reference, see "How Charts Process Events" on page 3-82.
3	Just before you leave the front desk, you pick up your bags to move to your room.	Just before the transition occurs, the `exit` action of `Front_desk` sets the `move_bags` local data to 1. Then, `Front_desk` becomes inactive. For reference, see "Exit a State" on page 3-58.

Modeling Guidelines for Chart Initialization.   The following guidelines apply to chart initialization.

Modeling Guideline	Why This Guideline Applies	Reference
Use exclusive (OR) decomposition when no two states at a level of the hierarchy can be active at the same time.	This guideline ensures proper chart execution. For example, `Check_in` and `Waiting_area` are exclusive (OR) states, because you cannot be inside and outside the hotel at the same time.	• "State Decomposition" on page 2-8 • "Specify Substate Decomposition" on page 4-8
Use a default transition to mark the first state to become active among exclusive (OR) states.	This guideline prevents state inconsistency errors during chart execution.	• "Default Transitions" on page 2-33 • "State Inconsistencies" on page 32-28
Use events, instead of conditions, to guard transitions that depend on occurrences without inherent numerical value.	Since you cannot easily quantify the numerical value of checking into a hotel, model such an occurrence as an event.	• "Activate a Stateflow Chart by Sending Input Events" on page 10-9

Modeling Guideline	Why This Guideline Applies	Reference
Use an `exit` action to execute a statement once, just before a state becomes inactive.	Other types of state actions execute differently and do not apply: • `Entry` actions execute once, just after a state becomes active. • `During` actions execute at every time step (except the first time step after a state becomes active). Execution continues as long as the chart remains in that state and no valid outgoing transitions exist. • `On` *event_name* actions execute only after receiving an event broadcast.	• "State Action Types" on page 12-2

Phase: Evaluation of Outgoing Transitions from a Single Junction

This section describes what happens after exiting the `Front_desk` state: the evaluation of a group of outgoing transitions from a single junction.

Stage	Hotel Scenario	Chart Behavior
1	You can move to one of three types of rooms.	After the `check_in` event triggers a transition out of `Front_desk`, three transition paths are available based on the type of room you select with the Multiport Switch block. Transition testing occurs based on the priority you assign to each path. For reference, see "Order of Execution for a Set of Flow Charts" on page 3-84.
2	If you choose an executive suite, the base fee is 1500.	If the `room_type` input data equals 1, the top transition is valid. If this condition is true, the condition action executes by setting the `fee` output data to 1500. **Note** If the top transition is not valid, control flow backtracks to the central junction so that testing of the next transition can occur. This type of backtracking is intentional. To learn about *unintentional* backtracking and how to avoid it, see "Backtrack in Flow Charts" on page B-38 and "Best Practices for Creating Flow Charts" on page 5-3.
3	If you choose a family suite, the base fee is 1000.	If `room_type` equals 2, the middle transition is valid. If this condition is true, the condition action executes by setting `fee` to 1000.
4	If you choose a single room, the base fee is 500.	If `room_type` equals 3, the bottom transition is valid. If this condition is true, the condition action executes by setting `fee` to 500.

What happens if `room_type` has a value other than 1, 2, or 3?

Because the Multiport Switch block outputs only 1, 2, or 3, `room_type` cannot have any other values. However, if `room_type` has a value other than 1, 2, or 3, the chart stays in the `Front_desk` state. This behavior applies because no transition path out of that state is valid.

Modeling Guidelines for Evaluation of Outgoing Transitions.   The following guidelines apply to transition syntax.

Modeling Guideline	Why This Guideline Applies	Reference
Use conditions, instead of events, to guard transitions that depend on occurrences with numerical value.	Because you can quantify a type of hotel room numerically, express the choice of room type as a condition.	"Flow Charts in Stateflow" on page 5-2
Use condition actions instead of transition actions whenever possible.	Condition actions execute as soon as the condition evaluates to true. Transition actions do not execute until after the transition path is complete, to a terminating junction or a state. Unless an execution delay is necessary, use condition actions instead of transition actions.	"Transition Action Types" on page 12-7
Use explicit ordering to control the testing order of a group of outgoing transitions.	You can specify *explicit* or *implicit* ordering of transitions. By default, a chart uses explicit ordering. If you switch to implicit ordering, the transition testing order can change when graphical objects move.	"Transition Evaluation Order" on page 3-65

Phase: Execution of State Actions for a Superstate

This section describes what happens after you enter the `Checked_in` state, regardless of which substate becomes active.

Stage	Hotel Scenario	Chart Behavior
1	After reaching your desired room, you finish moving your bags.	The `entry` action executes by setting the `move_bags` local data to 0.
2	If you order room service, your hotel bill increases by a constant amount.	If the chart receives an event broadcast for `room_service`, these actions occur: **1** The counter for the `service` local data increments by 1. **2** A function call to `expenses` occurs, which returns the value of the hotel bill stored by the `fee` output data. For reference, see "How Charts Process Events" on page 3-82.

Modeling Guidelines for Execution of State Actions. The following guidelines apply to state actions.

Modeling Guideline	Why This Guideline Applies	Reference
Use an `entry` action to execute a statement once, right after a state becomes active.	Other types of state actions execute differently and do not apply: • `During` actions execute at every time step until there is a valid transition out of the state.	"State Action Types" on page 12-2

Modeling Guideline	Why This Guideline Applies	Reference
Use an On *event_name* or On *message_name* action to execute a statement only after receiving an event broadcast or a message.	• Exit actions execute once, just before a state becomes inactive.	
Use a superstate to enclose multiple substates that share the same state actions.	This guideline enables reuse of state actions that apply to multiple substates. You write the state actions only *once*, instead of writing them separately in each substate.	"Create Substates and Superstates" on page 4-6

Phase: Function Call from a State Action

This part of the chart describes how you can perform function calls while a state is active.

Stage	Hotel Scenario	Chart Behavior

1	Based on your room type and the total number of room service requests, you can track your hotel bill.	`expenses` is a MATLAB function that takes the total number of room service requests as an input and returns the current hotel bill as an output.
		
		If you double-click the function box, you see this script in the function editor:
		
		`function y = expenses(x)`
		
		`if (room_type == 1)`
		`    y = 1500 + (x*50);`
		`else`
		`    if (room_type == 2)`
		`        y = 1000 + (x*25);`
		`    else`
		`        y = 500 + (x*5);`
		`    end`
		`end`

Modeling Guidelines for Function Calls.   The following guidelines apply to function calls.

Modeling Guideline	Why This Guideline Applies	Reference
Use MATLAB functions for performing numerical computations in a chart.	MATLAB functions are better at handling numerical computations than graphical functions, truth tables, or Simulink functions.	"Reuse MATLAB Code by Defining MATLAB Functions" on page 28-2
Use descriptive names in function signatures.	Descriptive function names enhance readability of chart objects.	

Phase: Execution of State with Exclusive Substates

This part of the chart shows how a state with exclusive (OR) decomposition executes.

Stage	Hotel Scenario	Chart Behavior
1	When you reach the executive suite, you enter the bedroom first. **Note** The executive suite has separate bedroom and dining areas. Therefore, you can be in only one area of the suite at any time.	When the condition `room_type == 1` is true, the condition action `fee = 1500` executes. Completion of that transition path triggers these state initialization actions: **1** `Checked_in` becomes active and executes its entry action. **2** `Executive_suite` becomes active. **3** The default transition to `Bedroom` occurs, making that state active. For reference, see "Enter a Chart or State" on page 3-50.
2	When you order room service, you enter the dining area to eat.	When the `room_service` event occurs, the transition from `Bedroom` to `Dining_area` occurs.
3	When you want the food removed from the dining area, you order room service again and then return to the bedroom.	When the `room_service` event occurs, the transition from `Dining_area` to `Bedroom` occurs.

Stage	Hotel Scenario	Chart Behavior
4	If you leave the executive suite because of a fire alarm, you return to your previous room after the all-clear signal.	If a transition out of `Executive_suite` occurs, the history junction records the last active substate, `Bedroom` or `Dining_area`. For details on how this transition can occur, see "Phase: Events Guard Transitions Between States" on page 3-22.

Modeling Guidelines for Execution of Exclusive (OR) States. The following guidelines apply to exclusive (OR) states.

Modeling Guideline	Why This Guideline Applies	Reference
Use exclusive (OR) decomposition when no two states at that level of the hierarchy can be active at the same time.	This guideline ensures proper chart execution. For example, `Bedroom` and `Dining_area` are exclusive (OR) states, because you cannot be in both places at the same time.	• "State Decomposition" on page 2-8 • "Specify Substate Decomposition" on page 4-8
If reentry to a state with exclusive (OR) decomposition depends on the previously active substate, use a history junction. This type of junction records the active substate when the chart exits the state.	If you do not record the previously active substate, the default transition occurs and the wrong substate can become active upon state reentry. For example, if you were eating when a fire alarm sounded, you would return to the bedroom instead of the dining room.	• "History Junctions" on page 2-44

Phase: Execution of State with Parallel Substates

This part of the chart shows how a state with parallel (AND) decomposition executes.

Stage	Hotel Scenario	Chart Behavior
1	When your family reaches the suite, family members can be in both bedrooms (for example, parents in the master bedroom and children in the second bedroom). A default room choice does not apply.	When the condition `room_type == 2` is true, the condition action `fee = 1000` executes. Completion of that transition path triggers these state initialization actions: **1** `Checked_in` becomes active and executes its entry action. **2** `Family_suite` becomes active. **3** The parallel states wake up in the order given by the number in the upper right corner of each state: `Master_bedroom`, then `Second_bedroom`. **How do I specify the order?** To specify the order: **a** Verify that the chart uses explicit ordering. In the Chart properties dialog box, select the **User specified state/ transition execution order** check box. **b** Right-click in a parallel state and select a number from the **Execution Order** menu. For reference, see "Enter a Chart or State" on page 3-50.
2	You can occupy both rooms at the same time.	`Master_bedroom` and `Second_bedroom` remain active at the same time.

Modeling Guidelines for Execution of Parallel (AND) States.   The following guidelines apply to parallel (AND) states.

Modeling Guideline	Why This Guideline Applies	Reference
Use parallel (AND) decomposition when all states at that level of the hierarchy can be active at the same time.	This guideline ensures proper chart execution. For example, `Master_bedroom` and `Second_bedroom` are parallel states, because you can occupy both rooms at the same time.	• "State Decomposition" on page 2-8 • "Specify Substate Decomposition" on page 4-8
Use *no* history junctions in states with parallel (AND) decomposition.	This guideline prevents parsing errors. Since all parallel states at a level of hierarchy are active at the same time, history junctions have no meaning.	• "History Junctions" on page 2-44
Use explicit ordering to control the execution order of parallel (AND) states.	You can specify *explicit* or *implicit* ordering of parallel states. By default, a chart uses explicit ordering. If you switch to implicit ordering, the execution order can change when parallel states move.	• "Execution Order for Parallel States" on page 3-86

Phase: Events Guard Transitions Between States

This part of the chart describes how events can guard transitions between exclusive (OR) states.

Stage	Hotel Scenario	Chart Behavior

1	If a fire alarm sounds, you leave the hotel and move to a waiting area outside.	When the chart receives an event broadcast for `fire_alarm`, a transition occurs from a substate of `Check_in` to `Waiting_area`.
		
		How does this transition occur?
		
		Suppose that `Check_in`, `Checked_in`, `Executive_suite`, and `Dining_area` are active when the chart receives `fire_alarm`.
		
		1 States become inactive in *ascending* order of hierarchy:
		
		  a `Dining_area`
		  b `Executive_suite`
		  c `Checked_in`
		  d `Check_in`
		
		2 `Waiting_area` becomes active.

2	If an all-clear signal occurs, you can leave the waiting area and return to your previous location inside the hotel.	When the chart receives an event broadcast for `all_clear`, a transition from `Waiting_area` to the previously active substate of `Check_in` occurs.
		
		The history junction at each level of hierarchy in `Check_in` enables the chart to remember which substate was previously active before the transition to `Waiting_area` occurred.
		
		How does this transition occur?
		
		Suppose that `Check_in`, `Checked_in`, `Executive_suite`, and `Dining_area` were active when the chart received `fire_alarm`.
		
		1 `Waiting_area` becomes inactive.
		2 States become active in *descending* order of hierarchy:
		
		  a `Check_in`
		  b `Checked_in` (The default transition does not apply.)
		  c `Executive_suite`
		  d `Dining_area` (The default transition does not apply.)

Modeling Guidelines for Guarding Transitions. The following guideline discusses the use of events versus conditions.

Modeling Guideline	Why This Guideline Applies	Reference
Use events, instead of conditions, to guard transitions that depend on occurrences without numerical value.	Because you cannot easily quantify the numerical value of a fire alarm or an all-clear signal, model such an occurrence as an event.	"Activate a Stateflow Chart by Sending Input Events" on page 10-9

Modeling Guidelines for Stateflow Charts

Use these guidelines to efficiently model charts with events, states, and transitions.

Use signals of the same data type for input events

When you use multiple input events to trigger a chart, verify that all input signals use the same data type. Otherwise, simulation stops and an error message appears. For more information, see "Data Types Allowed for Input Events" on page 10-12.

Use a default transition to mark the first state to become active among exclusive (OR) states

This guideline prevents state inconsistency errors during chart execution.

Use condition actions instead of transition actions whenever possible

Condition actions execute as soon as the condition evaluates to true. Transition actions do not execute until after the transition path is complete, to a terminating junction or a state.

Unless an execution delay is necessary, use condition actions instead of transition actions.

Use explicit ordering to control the testing order of a group of outgoing transitions

You can specify *explicit* or *implicit* ordering of transitions. By default, a chart uses explicit ordering. If you switch to implicit ordering, the transition testing order can change when graphical objects move.

Verify intended backtracking behavior in flow charts

If your chart contains unintended backtracking behavior, a warning message appears with instructions on how to avoid that problem. For more information, see "Best Practices for Creating Flow Charts" on page 5-3.

Use a superstate to enclose substates that share the same state actions

When you have multiple exclusive (OR) states that perform the same state actions, group these states in a superstate and define state actions at that level.

This guideline enables reuse of state actions that apply to multiple substates. You write the state actions only once, instead of writing them separately in each substate.

Note You cannot use boxes for this purpose because boxes do not support state actions.

Use MATLAB functions for performing numerical computations in a chart

MATLAB functions are better at handling numerical computations than graphical functions, truth tables, or Simulink functions.

Use descriptive names in function signatures

Descriptive function names enhance readability of chart objects.

Use history junctions to record state history

If reentry to a state with exclusive (OR) decomposition depends on the previously active substate, use a history junction. This type of junction records the active substate when the chart exits the state. If you do not record the previously active substate, the default transition occurs and the wrong substate can become active upon state reentry.

Do not use history junctions in states with parallel (AND) decomposition

This guideline prevents parsing errors. Since all parallel states at a level of hierarchy are active at the same time, history junctions have no meaning.

Use explicit ordering to control the execution order of parallel (AND) states

You can specify *explicit* or *implicit* ordering of parallel states. By default, a chart uses explicit ordering. If you switch to implicit ordering, the execution order can change when parallel states move.

Detect Modeling Errors During Edit Time

The Stateflow editor displays potential errors and warnings by highlighting objects in red or orange. The editor underlines syntax errors with a red wavy line. To see details and possible fixes, place your cursor over the object that is highlighted and click the error or warning badge. By fixing these issues when you design your charts, you avoid potential compile or run-time warnings and errors.

Enable the edit-time checking and syntax error highlighting through the **Display > Error & Warnings** check box.

Many edit-time checks have an associated diagnostic configuration parameter. To control the level of diagnostic action, open the Model Configuration Parameters dialog box. In the **Diagnostics > Stateflow** pane, you can select `error`, `warning`, or `none`. When you change the diagnostic level for a configuration parameter, the diagnostic level for the corresponding edit-time check also changes. For example, if you set the **Unreachable execution path** configuration parameter to `none`, then Stateflow does not highlight transition shadowing in the editor. Not all edit-time checks have corresponding configuration parameters.

Edit-Time Check Issue	Diagnostic Configuration Parameter
"Object Contains a Syntax Error" on page 3-29	No associated diagnostic. Always an error.
"Dangling Transition" on page 3-29	"Unreachable execution path" (Simulink)
"Unreachable State" on page 3-30	"Unreachable execution path" (Simulink)
"Transition Shadowing" on page 3-30	"Unreachable execution path" (Simulink)
"Invalid Default Transition Path" on page 3-31	No associated diagnostic. Always an error.
"Unconditional Path out of State with During Actions or Child States" on page 3-32	"Transition outside natural parent" (Simulink)
"Graphical Function Contains a State" on page 3-32	No associated diagnostic. Always an error.
"Default Transition Is Missing" on page 3-33	No associated diagnostic. Always an error.

Edit-Time Check Issue	Diagnostic Configuration Parameter
"Default Transition Path Does Not Terminate in a State" on page 3-34	"No unconditional default transitions" (Simulink)
"Unexpected Backtracking" on page 3-35	"Unexpected backtracking" (Simulink)
"Transition Loops Outside Natural Parent" on page 3-36	"Transition outside natural parent" (Simulink)
"Transition Action Precedes a Condition Action Along This Path" on page 3-37	"Transition action specified before condition action" (Simulink)
"Transition Begins or Ends in a Parallel State" on page 3-39	No associated diagnostic. Always a warning.
"Monitoring Leaf or Child State Activity of Parallel States" on page 3-39	No associated diagnostic. Always a warning.
"Invalid Transitions Crossing into Graphical Function" on page 3-40	No associated diagnostic. Always an error.
"Invalid Transitions Crossing out of Graphical Function" on page 3-40	No associated diagnostic. Always an error.

Object Contains a Syntax Error

In a state or transition, the notation for an action or condition does not follow the syntax rules. Violations are highlighted as errors. See "Transition Label Notation" on page 2-20 and "State Labels" on page 2-10.

Note Subcharts with syntax errors appear red in the parent chart with a badge indicating a syntax issue. In the subchart editor, the object is highlighted in red, however there is no badge indicating the issue.

Dangling Transition

A dangling transition is not connected to a destination object. Transitions must have a valid source state or junction and a valid destination state or junction. See "Transitions" on page 2-18.

Control the level of diagnostic action by setting the **Unreachable execution path** diagnostic configuration parameter.

Unreachable State

A state is unreachable when there is no valid execution path leading to the state. To make the state a reachable destination, connecting it with a transition from a reachable state or junction.
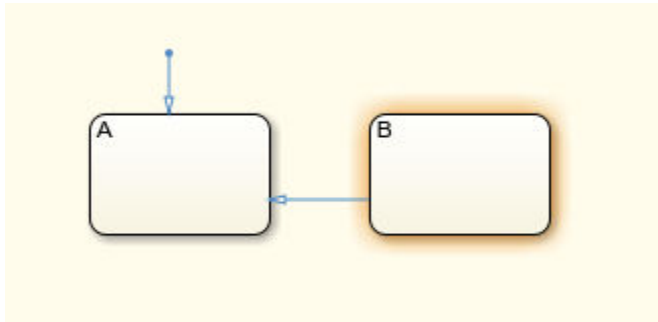
Control the level of diagnostic action by setting the **Unreachable execution path** diagnostic configuration parameter.

Transition Shadowing

Transition shadowing occurs when a chart contains an unconditional transition originating from a source that prevents other transitions from the same source from executing. To avoid transition shadowing:

- Create no more than one unconditional transition for each group of outgoing transitions from a state or junction.
- Explicitly specify an unconditional transition as a lower evaluation order than any transitions with conditions. See "Transition Evaluation Order" on page 3-65.
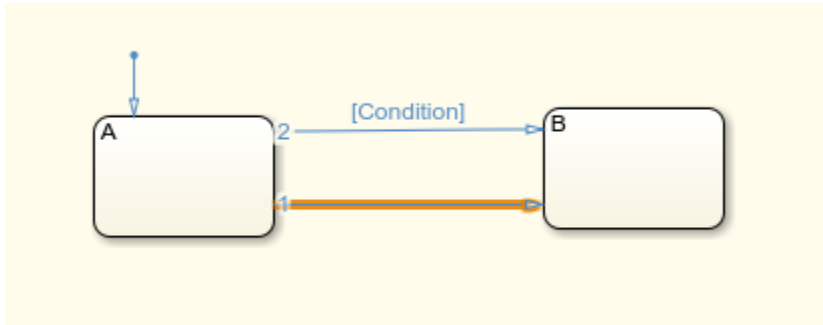
Control the level of diagnostic action by setting the **Unreachable execution path** diagnostic configuration parameter.

Invalid Default Transition Path

A default transition path is invalid when the execution path for the default transition exits the parent state. Violations are highlighted as errors. To resolve the error, create default transition paths that stay within the parent state.
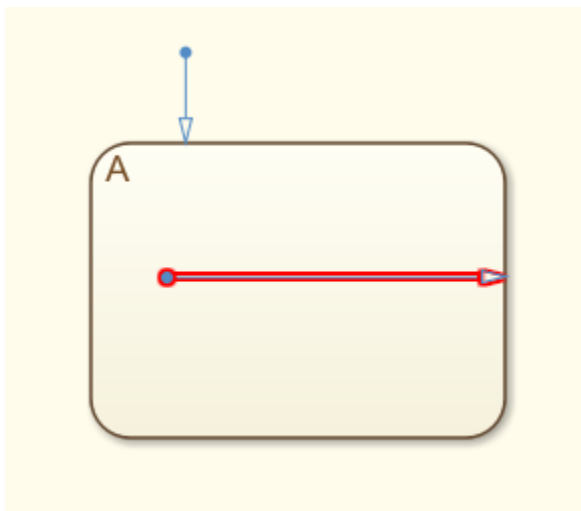
Unconditional Path out of State with During Actions or Child States

Unconditional transitions leading out of a state inhibit:

- The execution of the `during` actions in the state.
- The default transitions to child states.
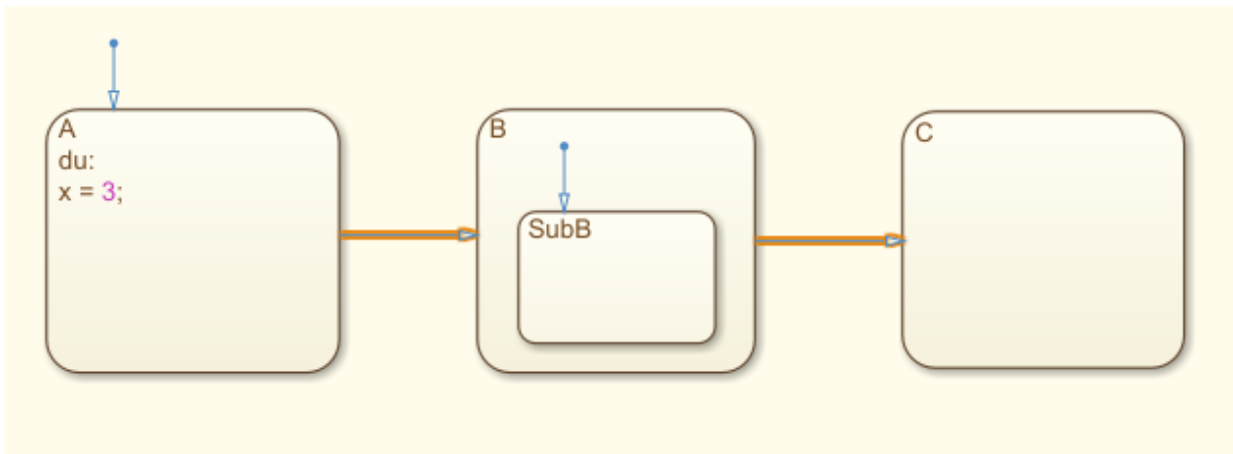
Control the level of diagnostic action by setting the **Transition outside natural parent** diagnostic configuration parameter.

Graphical Function Contains a State

When called, graphical functions must execute completely so they cannot contain any states. Violations are highlighted as errors. To resolve the error, delete the state from the graphical function.
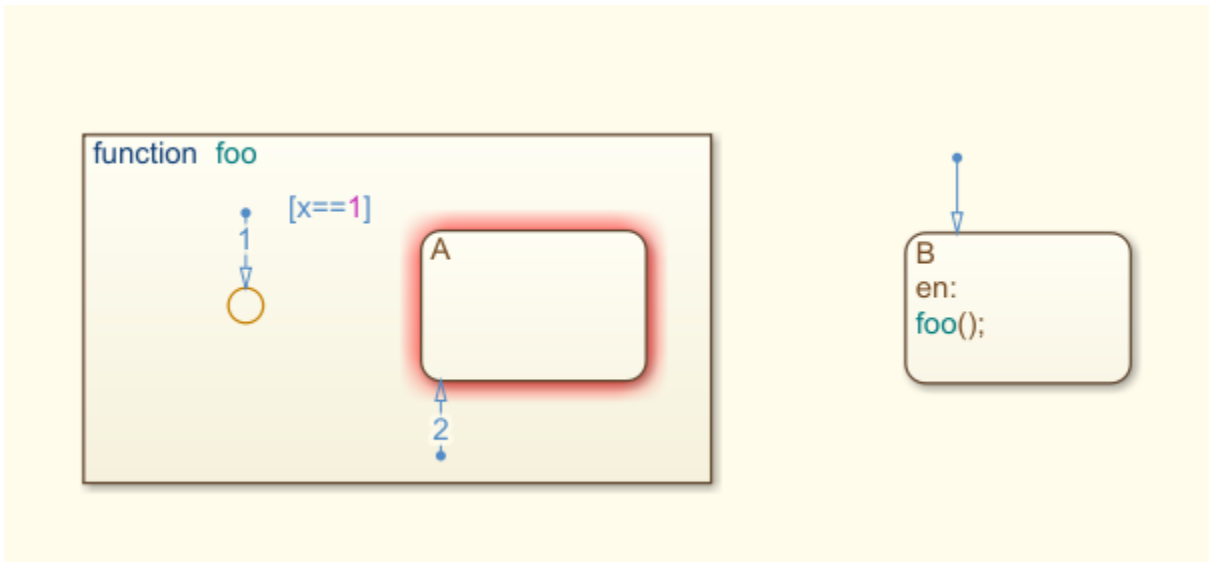
Default Transition Is Missing

Charts or states with exclusive (OR) decomposition require a default transition to indicate the child state or junction where execution begins. See "Default Transitions" on page 2-33. Violations are highlighted as errors. To resolve the error, add a default transition to a child state or junction.
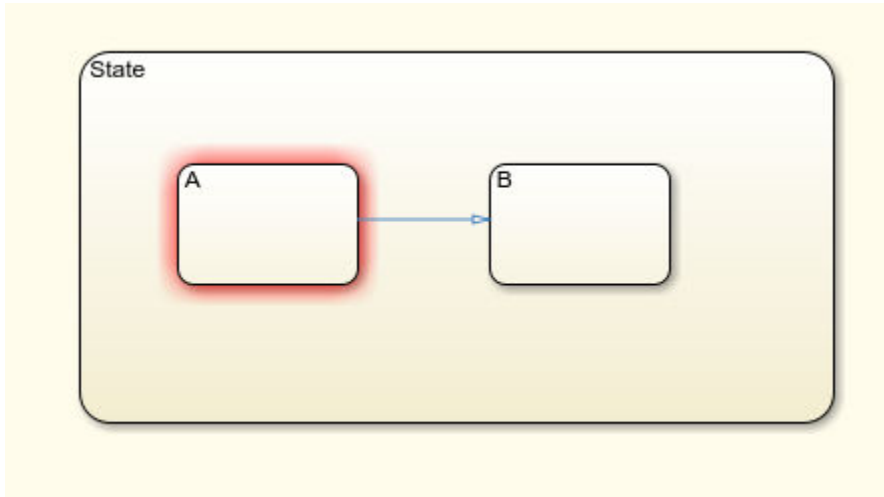
Default Transition Path Does Not Terminate in a State

In charts or states with exclusive (OR) decomposition and at least one substate:

- Every path along the default transition must lead to a substate.
- There must be one default transition path that is not guarded by a condition or triggered by an event.

If one path along the default transition remains unconditional, you can include multiple junctions and transitions.
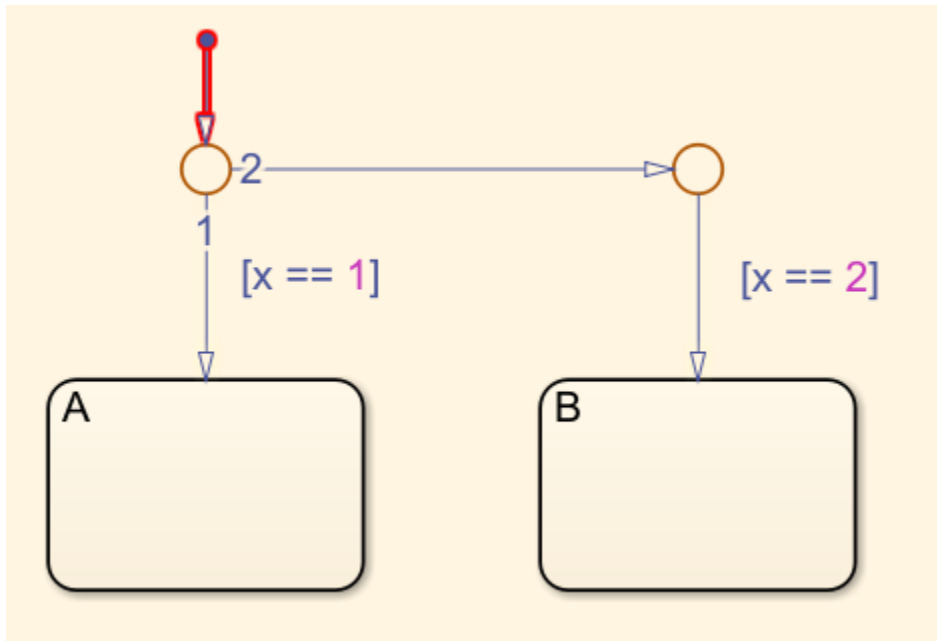
Control the level of diagnostic action by setting the **No unconditional default transitions** diagnostic configuration parameter.

Unexpected Backtracking

Unintended backtracking of control flow can occur at a junction when these conditions are true:

- The junction does not have an unconditional transition path to a state or terminating junction.
- Multiple transition paths that share a source lead to the junction.

See "Backtrack in Flow Charts" on page B-38.

3-35
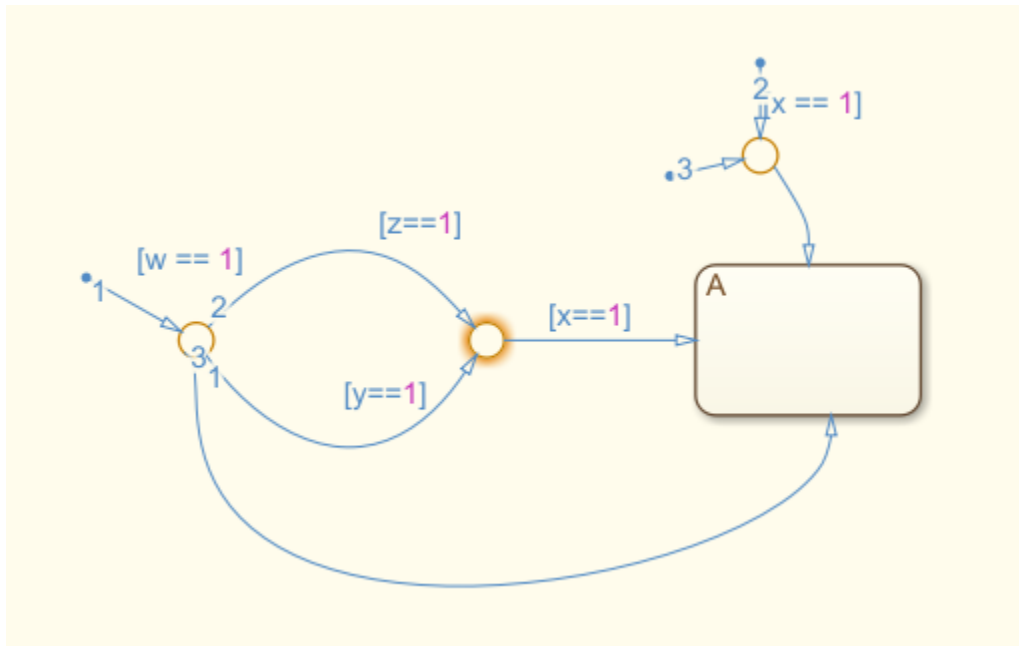
Control the level of diagnostic action by setting the **Unexpected backtracking** diagnostic configuration parameter.

Transition Loops Outside Natural Parent

If a transition loops outside of the parent state between the source and destination, the exit and entry actions of the parent state execute before the destination state becomes active. For example, when the highlighted transition in this chart becomes active, the state actions are executed in this order:

- `exit` actions in state B
- `exit` actions in state A
- `entry` actions in state A
- `entry` actions in state C
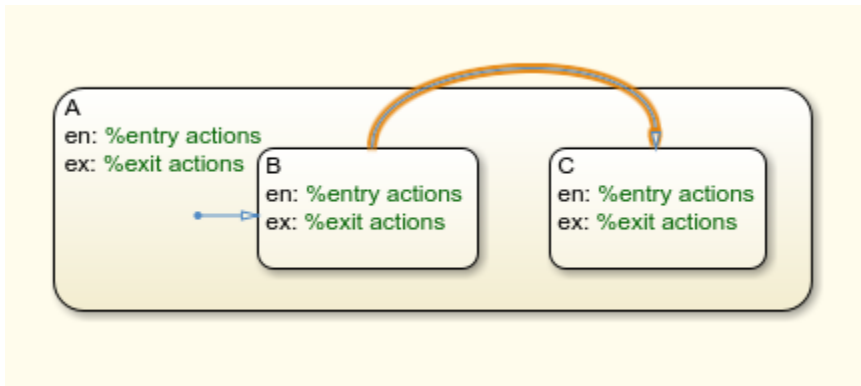
To prevent the chart from exiting and reentering state A, move the transition so that it is contained within state A.
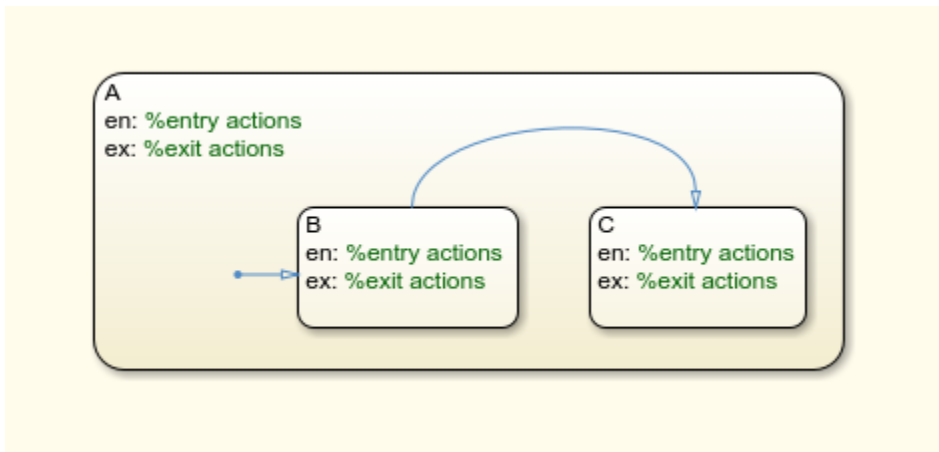
Control the level of diagnostic action by setting the **Transition outside natural parent** diagnostic configuration parameter.

Transition Action Precedes a Condition Action Along This Path

When there is a transition action on a path with a condition action on a following transition, the actions are not executed in the order of the transitions. Stateflow charts execute condition actions when the associated condition is evaluated as true. In contrast, charts execute transition actions only when the transition path is fully executed.

For example, in this chart, if `ConditionA` and `ConditionB` are true, then `ConditionAction2` occurs before `TransitionAction1`. The transition path from state A to state B follows this order:

- State A is active.
- Chart evaluates `ConditionA`.
- Chart evaluates `ConditionB`.
- Chart executes `ConditionAction2`.
- State A becomes inactive.
- Chart executes `TransitionAction1`.
- State B becomes active.
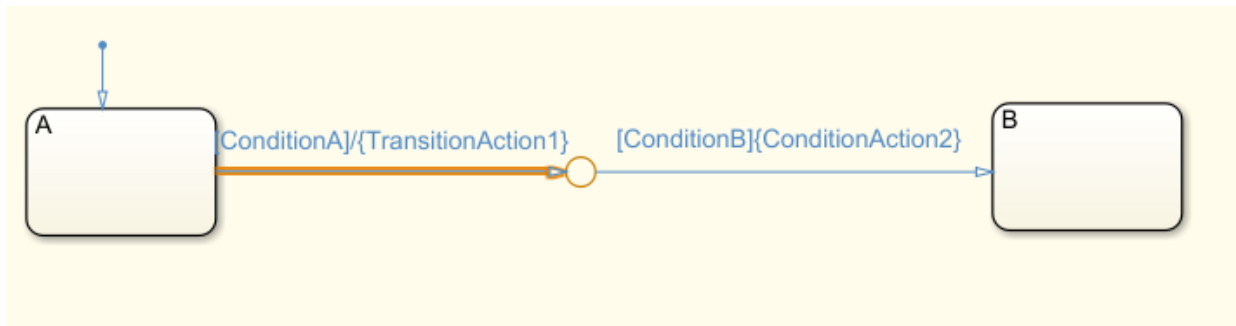
To improve the clarity of the chart, place the transition action after the last condition action on the path.
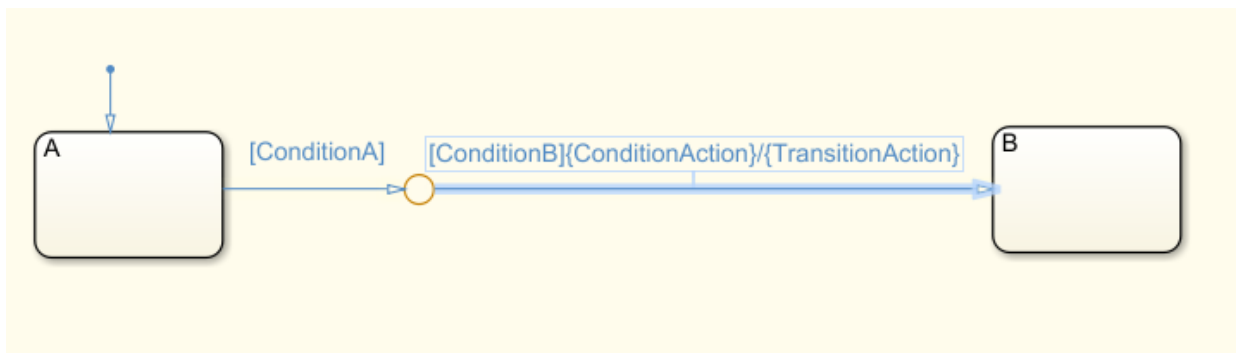
Control the level of diagnostic action by setting the **Transition action specified before condition action** diagnostic configuration parameter.

Transition Begins or Ends in a Parallel State

Transitions leading to or from a parallel state cause all sibling parallel states to become active or inactive. Violations are highlighted as warnings. To resolve the warning and avoid unintentional behavior, delete transitions into or out of parallel states.
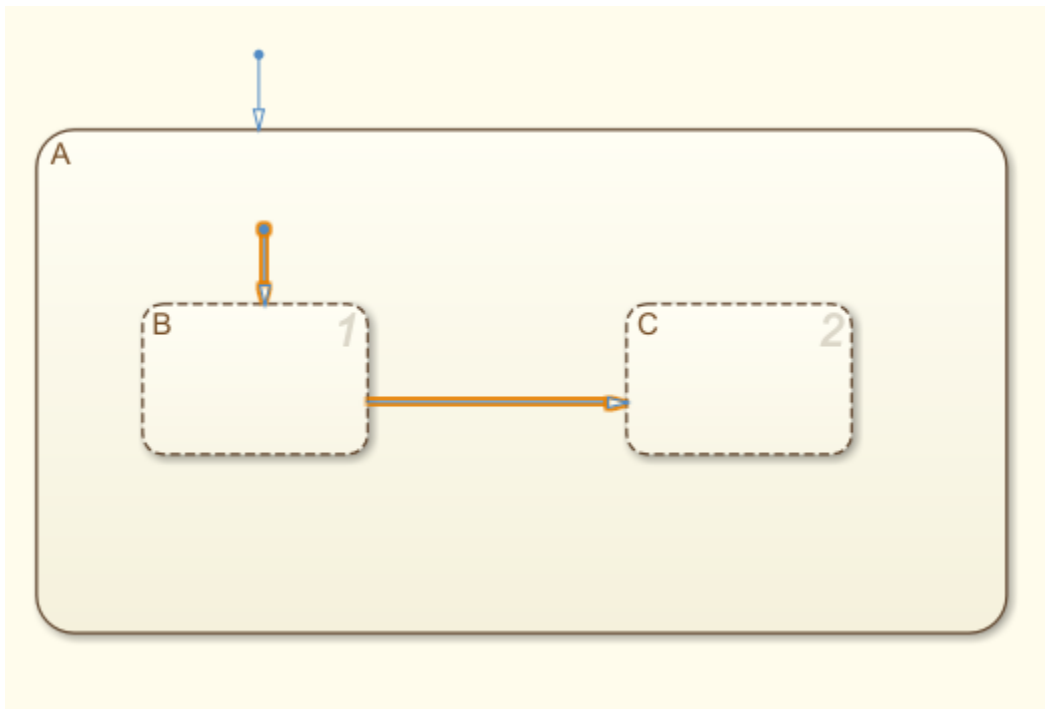
Monitoring Leaf or Child State Activity of Parallel States

Leaf or child state activity is not supported for parallel states. Violations are highlighted as warnings. To resolve the warning, in the Property Inspector or the Model Explorer, clear the **Create output for monitoring** check box or select `Self activity` from the drop down list. For more information, see "Monitor State Activity Through Active State Data" on page 24-26.

Invalid Transitions Crossing into Graphical Function

Transitions cannot have a graphical function as the destination. Call graphical functions from state actions or transitions. Violations are highlighted as errors. To resolve the error, remove the transition crossing into the graphical function.

Invalid Transitions Crossing out of Graphical Function

Transitions cannot have a graphical function as the destination. Call graphical functions from state actions or transitions. Violations are highlighted as errors. To resolve the error, remove the transition crossing out of the graphical function.

See Also

More About

- "Detect Common Modeling Errors During Chart Simulation" on page 32-28
- "Modeling Guidelines for Stateflow Charts" on page 3-25
- "Stateflow Semantics" on page 3-2
- "Model Configuration Parameters: Stateflow Diagnostics" (Simulink)

Types of Chart Execution

Life Cycle of a Stateflow Chart

Stateflow charts go through several stages of execution:

Stage	Description
Inactive	Chart has no active states
Active	Chart has active states
Sleeping	Chart has active states, but no events to process

When a Simulink model first triggers a Stateflow chart, the chart is inactive and has no active states. After the chart executes and completely processes its initial trigger event from the Simulink model, it transfers control back to the model and goes to sleep. At the next Simulink trigger event, the chart changes from the sleeping to active stage.

See "How Events Drive Chart Execution" on page 3-81.

Execution of an Inactive Chart

When a chart is inactive and first triggered by an event from a Simulink model, it first executes its set of default flow charts (see "Order of Execution for a Set of Flow Charts" on page 3-84). If this action does not cause an entry into a state and the chart has parallel decomposition, then each parallel state becomes active (see "Enter a Chart or State" on page 3-50).

If executing the default flow paths does not cause state entry, a state inconsistency error occurs.

Execution of an Active Chart

After a chart has been triggered the first time by the Simulink model, it is an active chart. When the chart receives another event from the model, it executes again as an active chart. If the chart has no states, each execution is equivalent to initializing a chart. Otherwise, the active substates execute. Parallel states execute in the same order that they become active.

Execution of a Chart at Initialization

By default, the first time a chart wakes up, it executes the default transition paths. At this time, the chart can access inputs, write to outputs, and broadcast events.

If you want your chart to begin executing from a known configuration, you can enable the **Execute (enter) Chart At Initialization** chart property. When you turn on this option, the state configuration of a chart initializes at time 0 instead of the first occurrence of an input event. The default transition paths of the chart execute during the model initialization phase at time 0, corresponding to the mdlInitializeConditions() phase for S-functions. For more information, see "Execute (Enter) Chart at Initialization" on page 24-7.

Note If an output of this chart connects to a SimEvents® block, do not select this check box. To learn more about using Stateflow charts and SimEvents blocks together in a model, see the SimEvents documentation.

Due to the transient nature of the initialization phase, do not perform certain actions in the default transition paths of the chart — and associated state entry actions — which execute at initialization. Follow these guidelines:

- Do not access chart input data, because blocks connected to chart input ports might not have initialized their outputs yet.
- Do not call exported graphical functions from other charts, because those charts might not have initialized yet.
- Do not broadcast function-call output events, because the triggered subsystems might not have initialized yet.

You can control the level of diagnostic action for invalid access to chart input data in the **Diagnostics** > **Stateflow** pane of the Configuration Parameters dialog box. For more information, see the documentation for the "Invalid input data access in chart initialization" (Simulink) diagnostic.

Execute at initialization is ignored in Stateflow charts that do not contain states.

See Also

More About

- "Execution of a Stateflow Chart" on page 3-44
- "Exit a State" on page 3-58
- "Evaluate Transitions" on page 3-63

Execution of a Stateflow Chart

When a Stateflow chart wakes up, the chart follows a workflow and executes actions. A Stateflow chart wakes up:

- At each time step according to the Simulink solver.
- When the Stateflow chart receives an event.

When a chart wakes up for the first time, the chart is initialized and becomes active. See "Chart Entry" on page 3-52. Once the chart is active but with no more actions to take, the chart goes to sleep until it is triggered by a new time step or an event.

Workflow for Stateflow Chart Execution

This flow chart shows the progression of events that Stateflow takes for executing a chart or state. In this flow chart, the current state refers to the state in which a decision or a process is taking place.
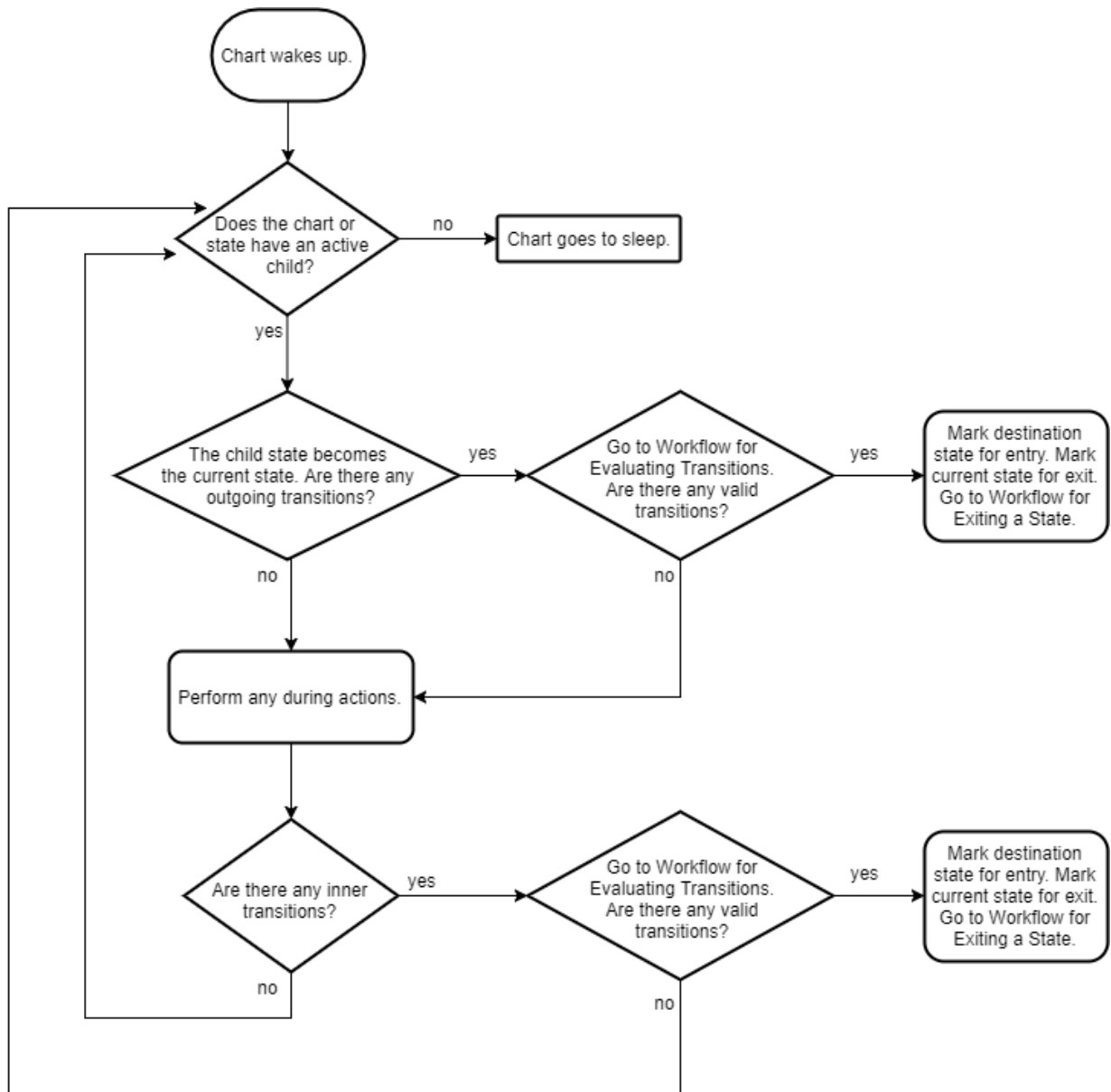
During Actions

During actions for a state execute when:

- The state is active, a new time step occurs, and no valid transition to another state is available.
- The state is active, an event occurs, and no valid transition to another state is available.

During actions are preceded by the prefix `during` or `du`, and then followed by a required colon (`:`), followed by one or more actions. Separate multiple actions with a carriage return, semicolon (`;`), or a comma (`,`). If you do not specify the state action type explicitly for a statement, the chart treats that statement as an `entry,during` action.

A state performs its `during` actions (if specified) when the chart wakes up. The preceding flow chart depicts the process of state execution and shows when `during` actions occur.

If your Stateflow chart does not contain states, each time the chart is executed, Stateflow always evaluates the default transition path.

Outgoing Transition

Stateflow marks outgoing transitions for evaluation as a part of the execution of a Stateflow chart. Once an outgoing transition is marked for evaluation, follow the "Workflow for Evaluating Transitions" on page 3-64. For more information about how Stateflow evaluates outgoing transitions, see "Evaluate Transitions" on page 3-63.

Inner Transitions

Stateflow marks inner transitions for evaluation as a part of the execution of a Stateflow chart. Once an inner transition is marked for evaluation, follow the "Workflow for Evaluating Transitions" on page 3-64. For more information about how Stateflow evaluates inner transitions, see "Evaluate Transitions" on page 3-63.

Chart Execution with a Valid Transition

In this example, the Stateflow chart is initialized and the `entry` actions are performed for `StateA` and `StateA1`. A new time step occurs and the chart wakes up.

At this time step, $x = 5$, $y = 2$, and $z = 0$.
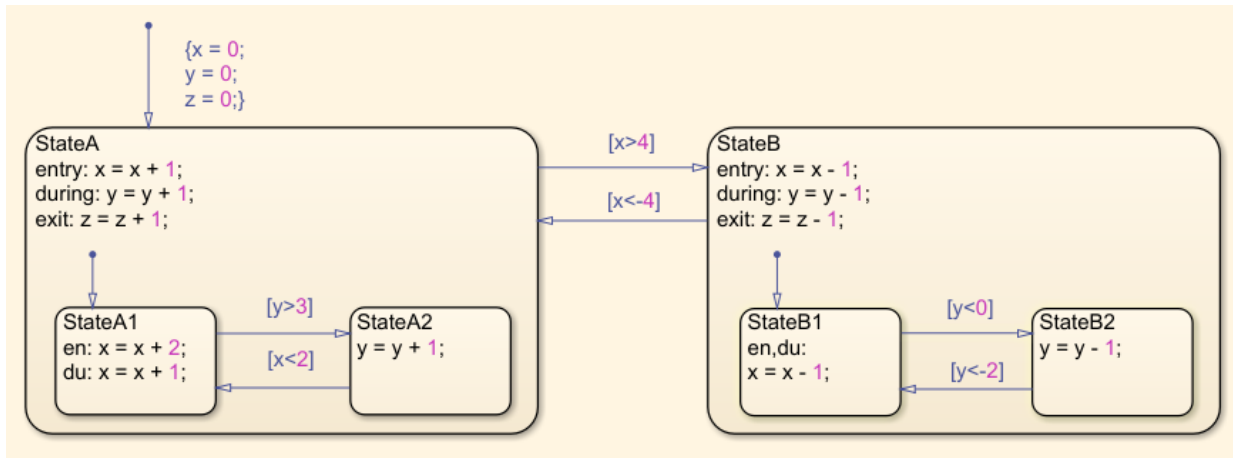
By following the "Workflow for Stateflow Chart Execution" on page 3-44, the execution steps for chart execution are in this order:

1 The chart has an active substate, StateA.
2 StateA has an outgoing transition to StateB. By following the "Workflow for Evaluating Transitions" on page 3-64, the transition is determined to be valid.
3 StateB is marked for entry and StateA is marked for exit.

To complete the time step, follow the "Workflow for Exiting a State" on page 3-58 for StateA and the "Workflow for Entering a Chart or State" on page 3-50 for StateB.

Chart Execution Without a Valid Transition

In this example, the Stateflow chart is initialized and the entry actions are performed for StateA and StateA1. A new time step occurs and the chart wakes up.

At this time step, x = 3, y = 0, and z = 0.
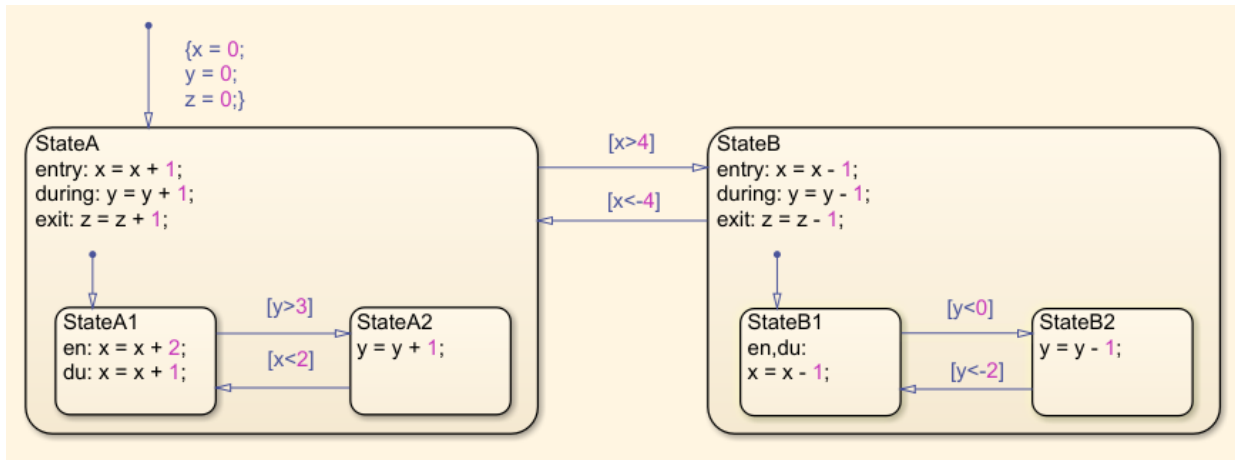
By following the "Workflow for Stateflow Chart Execution" on page 3-44 until the chart goes to sleep, the execution steps for chart execution are in this order:

1 The chart has an active substate, `StateA`.

2 `StateA` has an outgoing transition to `StateB`. By following the "Workflow for Evaluating Transitions" on page 3-64, the transition is determined to be invalid.

3 Perform the `during` actions for `StateA`. Now `y = 1`.

4 `StateA` does not have any inner transitions.

5 The active substate of `StateA` is `StateA1`.

6 `StateA1` has an outgoing transition to `StateA2`. By following the "Workflow for Evaluating Transitions" on page 3-64, the transition is determined to be invalid.

7 Perform the `during` actions for `StateA1`. Now `x = 4`.

8 `StateA1` does not have any active substates.

9 The chart goes to sleep.

Steps 1 through 9 take place in the second time step.

See Also

More About

- "Enter a Chart or State" on page 3-50
- "Exit a State" on page 3-58
- "Evaluate Transitions" on page 3-63

Enter a Chart or State

Chart and state entry occurs when:

- A chart is activated for the first time. This is called chart initialization.
- A valid transition into a state exists. See "Evaluate Transitions" on page 3-63.

Workflow for Entering a Chart or State

This flow chart shows the progression of events that Stateflow takes for entry into a chart or a state. In this flow chart, the current state refers to the state in which a decision or a process is taking place.

Chart or state has been marked for entry and as the current chart or state.

Is the current state the child of an inactive parent?

no → Perform any entry actions for the current state.

yes → Mark parent as active and as the current state.

Is this the state that was initially marked for entry?

no → Active child of this state becomes the current state.

yes

Are there any history junctions?

yes → Previously active child becomes the current state. Perform any entry actions.

no

Is there a single child state without a default transition?

Are there any default transitions?

no → Chart goes to sleep.

yes → Mark child state as active and as the current state.

no

yes → Go to Workflow for Evaluating a Transition.

3-51

Chart Entry

The first time that your Stateflow chart becomes active is called initialization. When initialization of your chart occurs, the chart is entered and Stateflow executes any default transitions for exclusive (OR) states. If the states at the top level of your chart are parallel (AND), they become active based on their order number.

If you want your chart to take any default transitions at time $t = 0$, in the Chart Properties dialog box, select the **Execute (enter) Chart At Initialization** check box. The default transition paths of the chart then execute during the model initialization phase.

State Entry

When a state is marked for entry, entry actions for a state execute. Once your chart is active and has gone through initialization, the top-level state becomes active. A state is marked for entry in one of these ways:

- An incoming transition crosses state boundaries.
- An incoming transition ends at the state boundary.
- The state is a parallel state child of an active state.

Entry Actions

Entry actions are preceded by the prefix `entry` or `en` for short, followed by a required colon (`:`), and then followed by one or more actions. You separate multiple actions by using a carriage return, a semicolon (`;`), or a comma (`,`). If you do not specify the state action type explicitly for a statement, the chart treats that statement as an `entry,during` action.

Enter a Stateflow Chart

In this example, the first time the chart becomes active, chart initialization occurs.
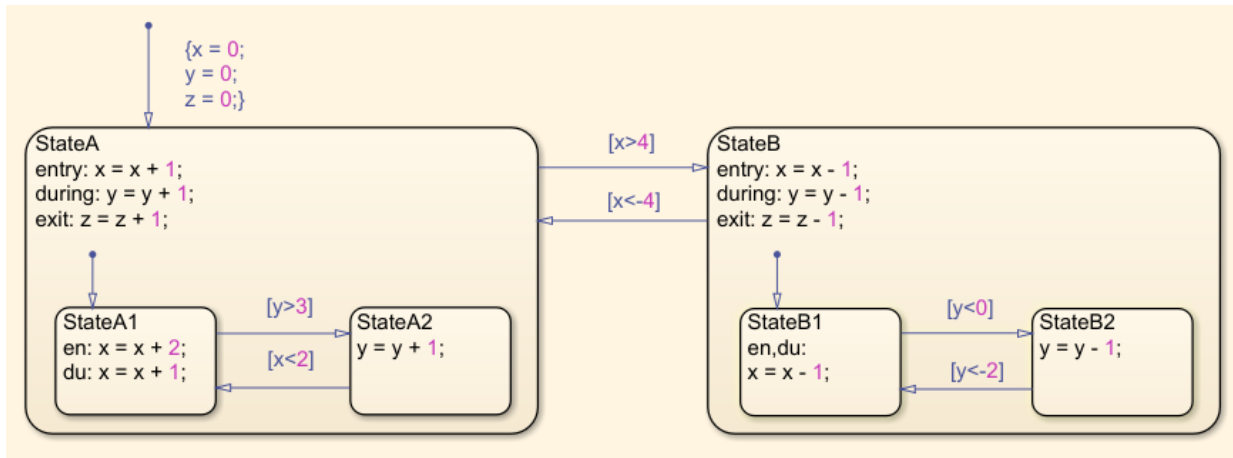
By following the "Workflow for Entering a Chart or State" on page 3-50 until the chart goes to sleep, the steps for chart initialization are in this order:

1   The default transition actions are executed, and `x = 0`, `y = 0`, and `z = 0`.

2   `StateA` is marked for entry.

3   `StateA` is not a substate of an inactive parent. Perform the `entry` actions for `StateA`. Now `x = 1`.

4   `StateA` is the state that was initially marked for entry.

5   `StateA` does not contain any history junctions.

6   There is a default transition to the substate, `StateA1`. Go to the Evaluate Transitions flow chart.

7   By following the Evaluate Transitions flow chart, mark `StateA1` for `entry`. Go to the Exit Actions flow chart.

8   The current state, `StateA`, is a superstate of the destination state, `StateA1`. Return to the Entry Actions flow chart.

9   `StateA1` is not a substate of an inactive parent. Perform `entry` actions for `StateA1`. Now `x = 3`.

10 `StateA1` is the state that was initially marked for entry.

11 `StateA1` does not contain any history junctions.

12 `StateA1` does not contain any default transitions.

13 `StateA1` does not contain any single substates.

14   The chart goes to sleep.

Steps 1 through 14 take place in the initial time step. This completes the chart initialization process.

Entering a State by Using History Junctions

If you want your Stateflow chart to remember and return to a substate that was previously active, regardless of a default transition, use a history junction. Placing a history junction within a state overrides the default transition leading to exclusive (OR) substates. After placing a history junction within a state, upon entry, your Stateflow chart remembers and enters the previously active substate. The history junction applies only to the level of the hierarchy in which it appears.

In this example, a light can be on or off. These options are indicated by the states `Power_on` and `Power_off`. The options are controlled by the input events `switch_on` and `switch_off`. When the light is on, it can be dim or bright. These options are indicated by the states `Low` and `High` and are controlled by the input events `switch_low` and `switch_high`.

Initially, the chart is asleep. The state `Power_off` is active. When the state `Power_on` was last active, `High` was the previously active substate. The event `switch_on` occurs and the state `Power_on` is marked for entry. At this time `p = 0`.
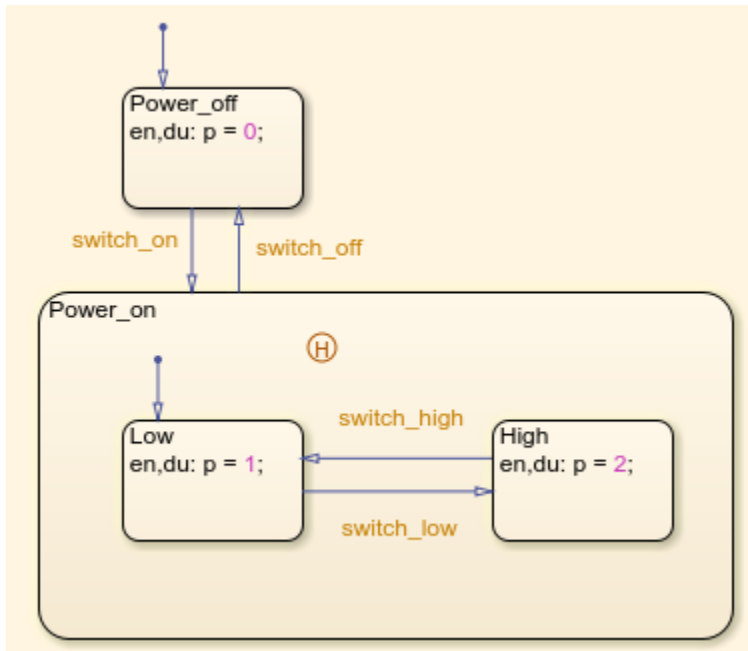
By following the "Workflow for Entering a Chart or State" on page 3-50 until the chart goes to sleep, the execution steps for entering the state Power_on are in this order:

1   Power_on is not the child of an inactive parent.

2   There are no entry actions for Power_on.

3   Power_on is the state that was initially marked for entry.

4   There are history junctions in Power_on.

5   High was the previously active substate. Now p = 2.

6   High does not contain any history junctions.

7   High does not contain any default transitions.

8   High does not contain any single substates.

9   The chart goes to sleep.

This completes the entry actions for Power_on and High.

Entering a State by Using Supertransitions

A supertransition is a transition between different levels in a chart. A supertransition can be between a state in a top-level chart and a state in one of its subcharts, or between states residing in different subcharts at the same or different levels in a chart. You can create supertransitions that span any number of levels in your chart.

When a state is entered through a supertransition, before the entry actions for the final destination are executed, its superstates must be marked active and their entry actions must be executed. In this example, StateB1 has been marked for entry from StateA2. At this point, x = 5, y = 5, and z = 1.
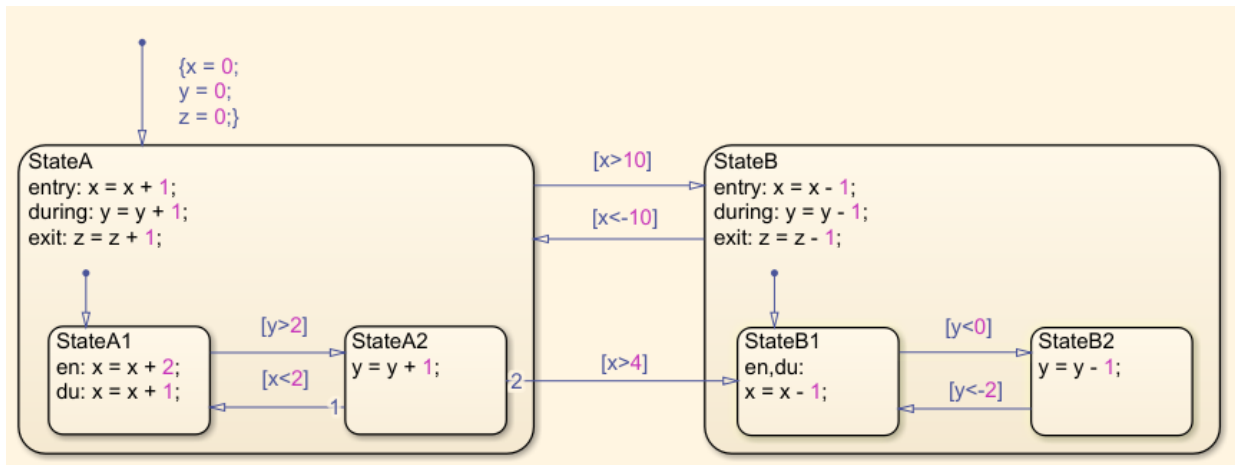
By following the "Workflow for Entering a Chart or State" on page 3-50 until the chart goes to sleep, the execution steps for entering the state StateB1 are in this order:

1   StateB1 is the substate of an inactive parent (StateB).

2   StateB is marked as active.

3   StateB is not the substate of an inactive parent.

4   Perform the entry actions for StateB. Now x = 4.

5   StateB is not the state that was initially marked for entry.

6   Perform the entry actions for StateB1. Now x = 3.

7   StateB1 is the state that was initially marked for entry.

8   `StateB1` has no history junctions.

9   `StateB1` does not contain any default transitions.

10 `StateB1` does not contain any single substates.

11 The chart goes to sleep.

This completes the `entry` actions for `StateB` and `StateB1`.

See Also

More About

- "Execution of a Stateflow Chart" on page 3-44
- "Exit a State" on page 3-58
- "Evaluate Transitions" on page 3-63

Exit a State

When there is a valid transition out of a state, that state is marked for exit. A state is marked for exit in one of these ways:

- The outgoing transition originates at the state boundary.
- The outgoing transition crosses the state boundary.
- The destination state is a parallel state child of an activated state.

Workflow for Exiting a State

This flow chart shows the progression of events in Stateflow for exiting a state. In this flow chart, the current state refers to the state in which a decision or a process is taking place.
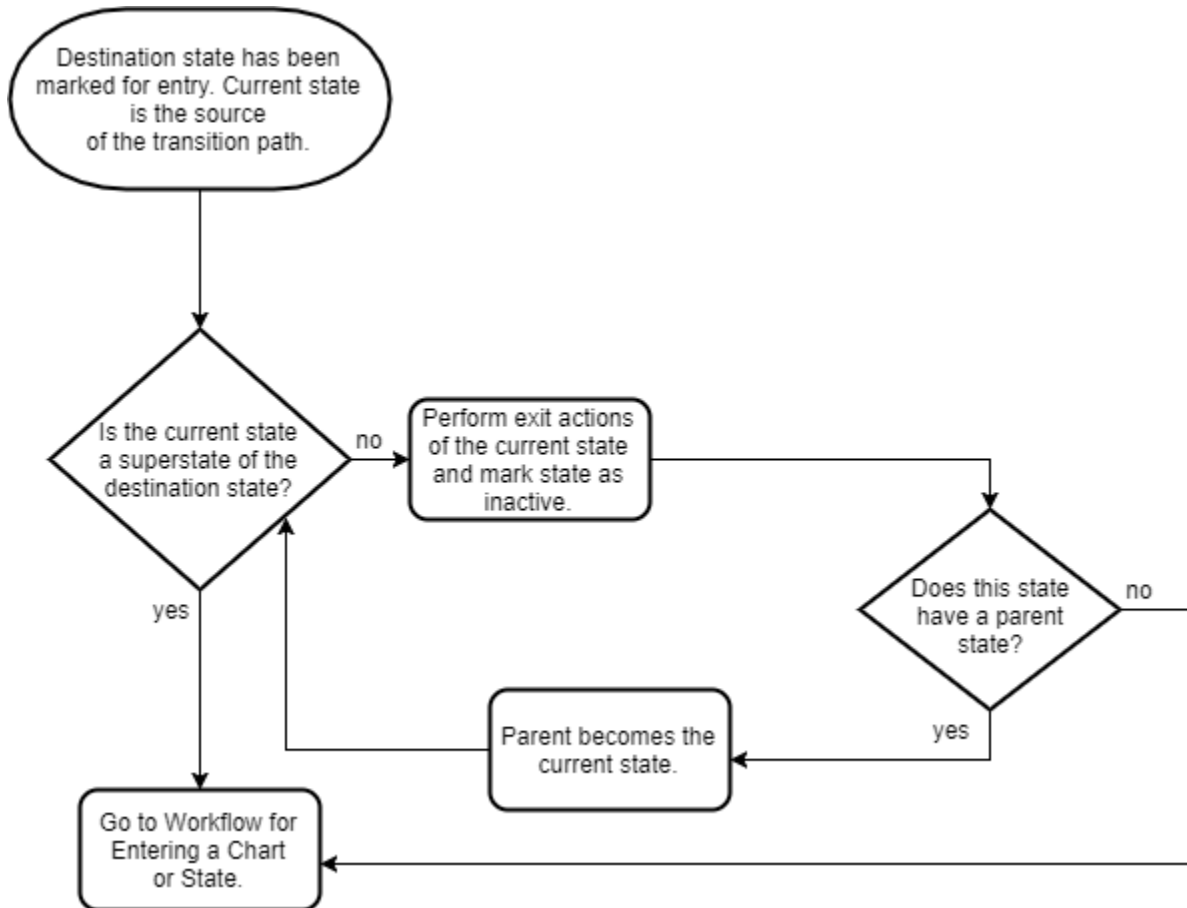
Exit Actions

`Exit` actions for a state execute when the state is active and a valid transition from the state exists. A state performs its `exit` actions before becoming inactive.

Exit actions are preceded by the prefix `exit` or `ex`, followed by a required colon (`:`), and then followed by one or more actions. Separate multiple actions with a carriage return, semicolon (`;`), or a comma (`,`).

Exit a State Example

In this example, the Stateflow chart is initialized and the `entry` actions are performed for `StateA` and `StateA1`. For this chart, the `during` actions for this chart have occurred twice. A new time step occurs, and then the chart wakes up.

By following the "Workflow for Stateflow Chart Execution" on page 3-44 and the "Workflow for Evaluating Transitions" on page 3-64, `StateB` has been marked for entry. `StateA` is the source of the transition. At this time step `x = 5`, `y = 2`, and `z = 0`.
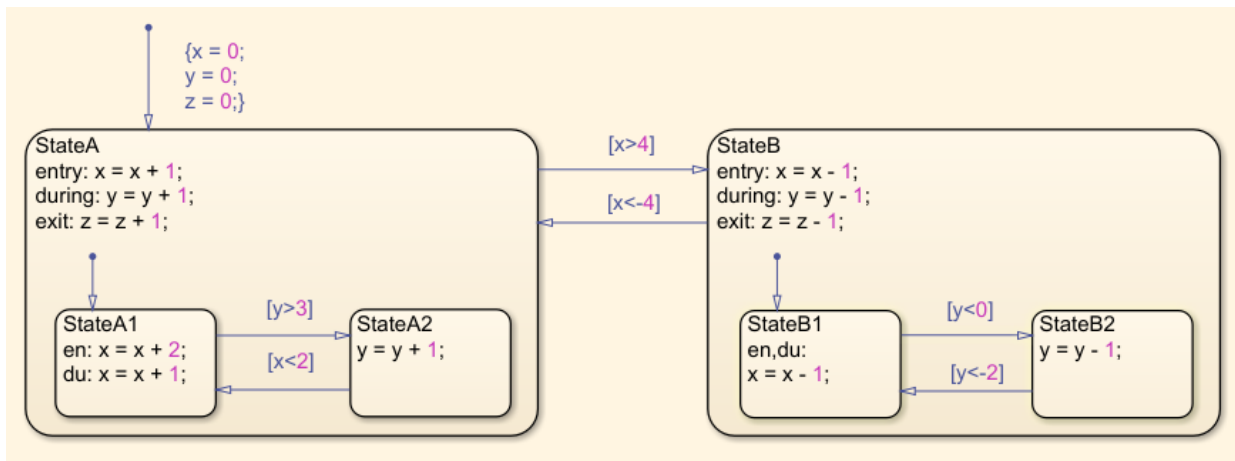
By following the flow chart for state exit actions until the chart goes to sleep, the execution steps for this chart are in this order:

1   `StateA` is not a superstate of `StateB`.

2   Perform the exit actions of `StateA` and mark `StateA` as inactive. Now `z = 1`.

3   `StateA` does not have a parent state.

4   Go to "Entry Actions" on page 3-52.

These steps complete the `exit` workflow for `StateA`. However, the chart is not yet asleep.

Perform the "Workflow for Entering a Chart or State" on page 3-50 for `StateB` to complete the time step.

Exit a State by Using Supertransitions

A supertransition is a transition between different levels in a chart. A supertransition can be between a state in a top-level chart and a state in one of its subcharts, or between states residing in different subcharts at the same or different levels in a chart. You can create supertransitions that span any number of levels in your chart.

When a state is exited through a supertransition, after the exit actions for the source of the transition are executed, its superstates are marked inactive and exit actions of the superstates are executed. In this example, `StateA2` is marked for `exit` and `StateB1` is marked for `entry`. At this point, `x = 5`, `y = 5`, and `z = 0`.
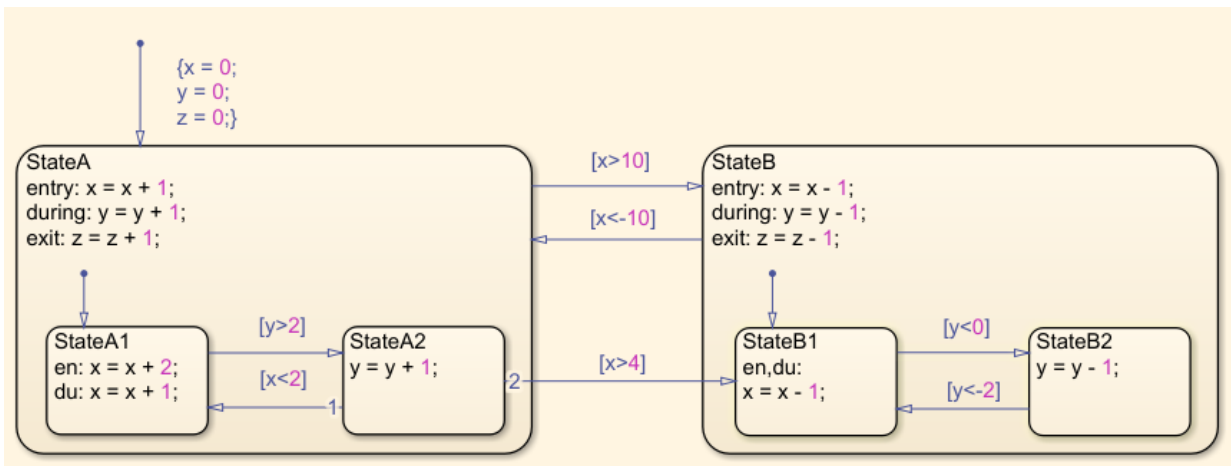
By following the "Workflow for Entering a Chart or State" on page 3-50 until the chart goes to sleep, the execution steps for exiting the state `StateA2` are in this order:

1   `StateA2` is not a superstate of the destination state (`StateB1`).

2   Perform the `exit` actions for `StateA2` and mark `StateA2` as inactive.

3   `StateA2` does have a parent state, `StateA`.

4   `StateA` is not a superstate of the destination state (`StateB1`).

5   Perform the `exit` actions for `StateA`, and mark `StateA` as inactive.

6   `StateA` does not have a parent state.

These actions complete the `exit` workflow for `StateA2` and `StateA`. However, the chart is not yet asleep.

Perform the "Workflow for Entering a Chart or State" on page 3-50 for `StateB` and `StateB1` to complete the time step.

See Also

More About

- "Execution of a Stateflow Chart" on page 3-44
- "Enter a Chart or State" on page 3-50
- "Evaluate Transitions" on page 3-63

Evaluate Transitions

Stateflow uses transitions in charts to move from one exclusive (OR) state to another exclusive (OR) state. For the `entry` and `execution` workflows of chart execution, Stateflow evaluates transitions to determine if they are valid. A valid transition is a transition whose condition labels are true and whose path ends at a state. If a transition is valid, Stateflow exits from the source state and enters the destination state. To learn about when evaluation occurs during the `execution` and `entry` workflows, see "Execution of a Stateflow Chart" on page 3-44 and "Enter a Chart or State" on page 3-50.

Workflow for Evaluating Transitions

Transition is marked for evaluation.

Does the transition have a condition? — no

yes

Is the condition true? — no

yes

Execute any condition actions.

Is the destination a state? — no

yes

Valid transition path found. Execute all transition actions for the valid path and return to previous flow chart.

Does the junction have any outgoing transitions? — no

No valid transitions. Return to previous flow chart.

yes — Identify the first transition in the execution order.

Next transition in execution order becomes the current transition.

no

Is this transition the last transition in execution order?

yes

Is this transition the initial transition marked for evaluation? — yes — No valid transitions. Return to previous flow chart.

no

Return to the previous junction or state on the transition path.

Identify the last transition executed. This transition is the current transition.

Transition Evaluation Order

When multiple transitions originate from a single source, such as a state or junction, Stateflow uses evaluation order to determine when to test each transition. Depending on which action language your chart uses, you can create the order of your transitions explicitly or implicitly. Whether explicitly or implicitly ordered, transitions show a number near the source of the transition that designates the transition order.

Note Use explicit ordering to avoid your transitions from changing order while you are editing a chart.

Explicit Ordering

When you open a new Stateflow chart, all outgoing transitions from a source are automatically numbered in the order in which you create them. The order starts with 1 and continues to the next available number for the source.

To change the execution order of a transition, right-click the transition, place your cursor over Execution Order, and select the order in which you want your transition to execute. When you change a transition number, the Stateflow chart automatically renumbers the other outgoing transitions for the source by preserving their relative order.

Implicit Ordering

For C charts in implicit ordering mode, a Stateflow chart evaluates a group of outgoing transitions from a single source based on:

- Hierarchy.

  A chart evaluates a group of outgoing transitions in an order based on the hierarchical level of the parent of each transition.

- Label.

  A chart evaluates a group of outgoing transitions with equal hierarchical priority based on the labels, in the following order of precedence:

  1 Labels with events and conditions
  2 Labels with events

    3   Labels with conditions

    4   No label

- Angular surface position of transition source.

  A chart evaluates a group of outgoing transitions with equal hierarchical and label priority based on angular position on the surface of the source object. The transition with the smallest clock position has the highest priority. For example, a transition with a 2 o'clock source position has a higher priority than a transition with a 4 o'clock source position. A transition with a 12 o'clock source position has the lowest priority.

Outgoing Transition Example

In this example, the Stateflow chart is initialized and the `entry` actions are performed for `StateA`. A new time step occurs and the chart wakes up. By following the "Workflow for Stateflow Chart Execution" on page 3-44, Stateflow finds multiple outgoing transitions from `StateA`. At this time step `x = 1`, `y = 1`, and `z = 1`.
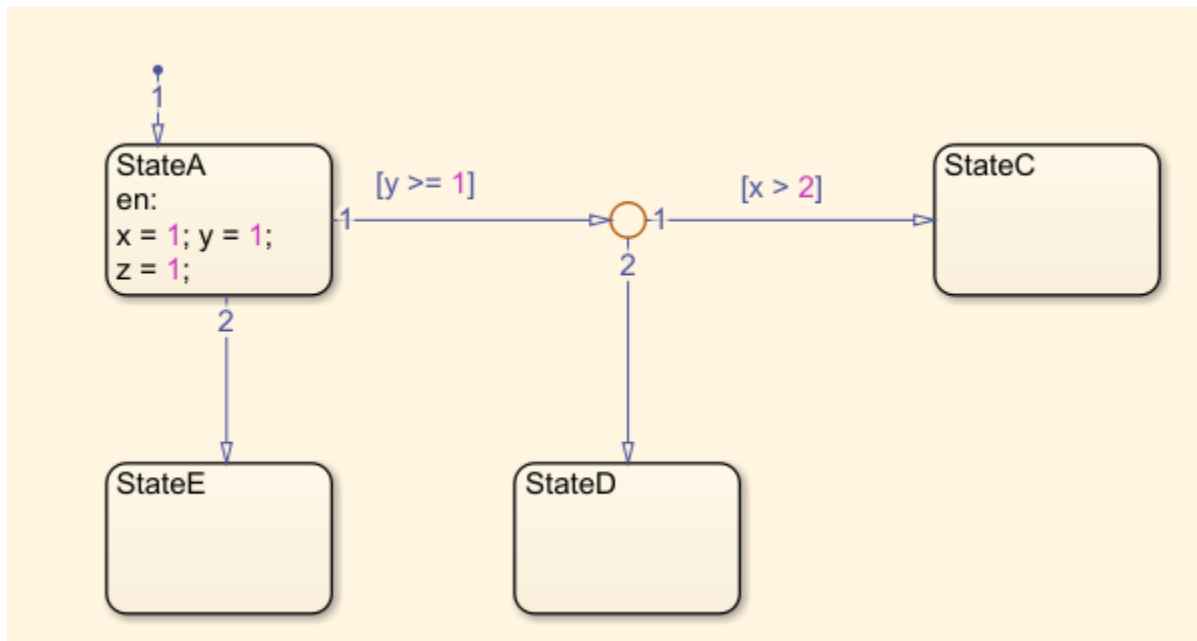
Evaluate Outgoing Transitions

By following the "Workflow for Evaluating Transitions" on page 3-64, the steps for evaluating the transitions of this chart are in this order:

1   Transition 1 from `StateA` is marked for evaluation.

2   Transition 1 from `StateA` has a condition.

3   The condition is true.

4   The destination of transition 1 from `StateA` is not a state.

5   The junction does have outgoing transitions.

6   Transition 1 from the junction is marked for evaluation.

7   Transition 1 from the junction has a condition.

8   The condition is false.

9   Transition 2 from the junction is marked for evaluation.

10   Transition 2 from the junction does not have a condition.

11   The destination of transition 2 from the junction is a state (`StateD`).

12   `StateD` is marked for `entry`, and `StateA` is marked for `exit`.

To complete the time step, follow the "Workflow for Exiting a State" on page 3-58 for `StateA` and the "Workflow for Entering a Chart or State" on page 3-50 for `StateE`.

Outgoing Transition Example with Backtracking

When all outgoing transitions from a source are invalid or do not end with a terminal junction, but there are previously unevaluated transitions, Stateflow returns to the previous state or junction to evaluate all possible paths.

In this example, the Stateflow chart is initialized and the `entry` actions are performed for `StateA`. A new time step occurs, and the chart wakes up. By following the "Workflow for Stateflow Chart Execution" on page 3-44, Stateflow finds multiple outgoing transitions from `StateA`. At this time step `x = 1`, `y = 1`, and `z = 1`.
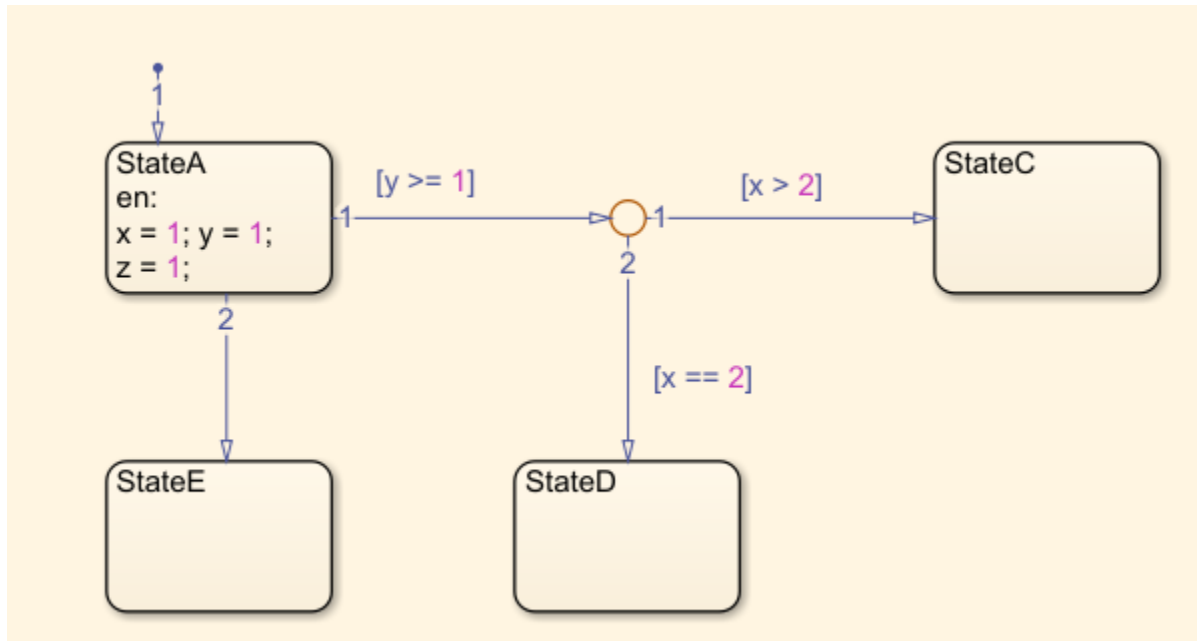
Evaluate Outgoing Transitions with Backtracking

By following the "Workflow for Evaluating Transitions" on page 3-64, the steps for evaluating the transitions of this chart are in this order:

1   Transition 1 from `StateA` is marked for evaluation.

2   Transition 1 from `StateA` has a condition.

3   The condition is true.

4   The destination of transition 1 from `StateA` is not a state.

5   The junction does have outgoing transitions.

6   Transition 1 from the junction is marked for evaluation.

7   Transition 1 from the junction has a condition.

8   The condition is false.

9   Transition 2 from the junction is marked for evaluation.

10 Transition 2 from the junction has a condition.

11 The condition is false.

12 Transition 2 from `StateA` is marked for evaluation.

13 Transition 2 from `StateA` does not have a condition.

14 The destination of transition 2 from `StateA` is a state (`StateE`).

15 `StateE` is marked for `entry`, and `StateA` is marked for `exit`.

To complete the time step, follow the "Workflow for Exiting a State" on page 3-58 for `StateA` and the "Workflow for Entering a Chart or State" on page 3-50 for `StateE`.

Prevent Backtracking

In this example, a terminal junction prevents backtracking. The Stateflow chart is initialized and the `entry` actions are performed for `StateA`. A new time step occurs and the chart wakes up. By following the "Workflow for Stateflow Chart Execution" on page 3-44, Stateflow finds multiple outgoing transitions from `StateA`. At this time step $x = 1$, $y = 1$, and $z = 1$.

By following the "Workflow for Evaluating Transitions" on page 3-64, the steps for evaluating the transitions of this chart are in this order:

1 Transition 1 from `StateA` is marked for evaluation.

2 Transition 1 from `StateA` has a condition.

3 The condition is true.

4 The destination of transition 1 from `StateA` is not a state.

5 The junction does have outgoing transitions.

6 Transition 1 from the junction is marked for evaluation.

7 Transition 1 from the junction has a condition.

8 The condition is false.

9 Transition 2 from the junction is marked for evaluation.

10 Transition 2 from the junction has a condition.

11 The condition is false.

12 Transition 3 from the junction is marked for evaluation.

13 Transition 3 from the junction does not have a condition.

14 The destination is not a state and does not have any outgoing transitions.

15 Return to "Workflow for Stateflow Chart Execution" on page 3-44.

To complete the time step, follow the "Workflow for Stateflow Chart Execution" on page 3-44 for `StateA`, starting where you left off.

Condition and Transition Actions

Condition Actions

In transition label syntax, condition actions follow the transition condition and are enclosed in curly braces (`{}`). Condition actions are executed when the condition is evaluated as true but before the transition path has been determined to be valid.

Transition Actions

In transition label syntax, transition actions are preceded with a forward slash (`/`) and are enclosed in curly braces (`{}`). Transition actions execute only after the transition path is determined to be valid.

In this example, both condition actions and transition actions exist. The Stateflow chart is initialized and the `entry` actions are performed for `StateA`. A new time step occurs and the chart wakes up. There are multiple outgoing transitions from `StateA`. At this time step $x = 1$, $y = 1$, and $z = 1$.
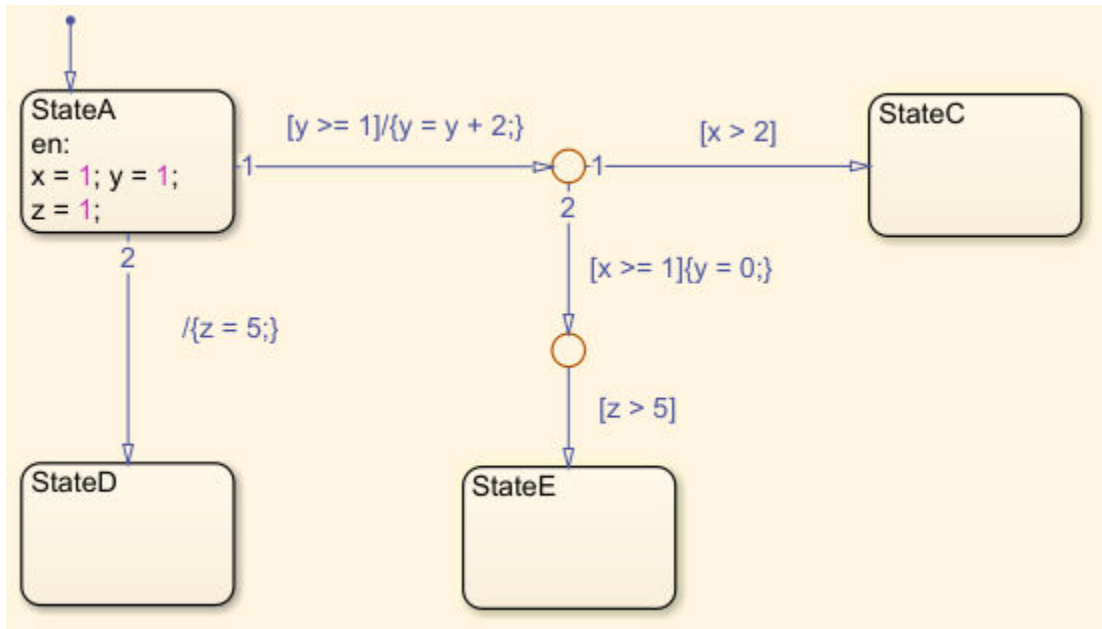
Evaluate Outgoing Transitions with Condition and Transition Actions

By following the "Workflow for Evaluating Transitions" on page 3-64, the steps for evaluating the transitions of this chart are in this order:

1   Transition 1 from `StateA` is marked for evaluation.

2   Transition 1 from `StateA` has a condition (`[y >= 1]`).

3   The condition is true.

4   There are no condition actions.

5   The destination of transition 1 from `StateA` is not a state.

6   The junction does have outgoing transitions.

7   Transition 1 from the junction is marked for evaluation.

8   Transition 1 from the junction has a condition (`[x > 2]`).

9   The condition is false.

10   Transition 2 from the junction is marked for evaluation.

11   Transition 2 from the junction has a condition (`[x >= 1]`).

12 The condition is true.

13 There is a condition action ($\{y = 0;\}$). Now $y = 0$.

14 The junction does have outgoing transitions.

15 The transition from the junction is marked for evaluation.

16 Transition 1 from the junction has a condition ([z >= 5]).

17 The condition is false.

18 Transition 2 from `StateA` is marked for evaluation.

19 Transition 2 from `StateA` does not have a condition.

20 The destination of transition 2 from `StateA` is a state (`StateD`).

21 `StateD` is marked for `entry`, and `StateA` is marked for `exit`. Execute the transition action for this valid path (/{z = 5}). Now $z = 5$.

To complete the time step, follow the "Workflow for Exiting a State" on page 3-58 for `StateA` and the "Workflow for Entering a Chart or State" on page 3-50 for `StateE`.

See Also

More About

- "Execution of a Stateflow Chart" on page 3-44
- "Enter a Chart or State" on page 3-50
- "Exit a State" on page 3-58

Super Step Semantics

By default, Stateflow charts execute once for each input event or time step. If you are modeling a system that must react quickly to inputs, you can enable super step semantics. Super step semantics are supported only in Stateflow charts in a Simulink model.

When you enable super step semantics, a Stateflow chart executes multiple times for every active input event or for every time step when the chart has no input events. The chart takes valid transitions until *either* of these conditions occurs:

- No more valid transitions exist, that is, the chart is in a stable active state configuration.
- The number of transitions taken exceeds a user-specified maximum number of iterations.

In a super step, your chart responds faster to inputs but performs more computations in each time step. Therefore, when generating code for an embedded target, make sure that the chart can finish the computation in a single time step. To achieve this behavior, fine-tune super step parameters by setting an upper limit on the number of transitions that the chart takes per time step (see "Maximum Number of Iterations" on page 3-73).

For simulation targets, specify whether the chart goes to the next time step or generates an error if it reaches the maximum number of transitions prematurely. However, in generated code for embedded targets, the chart always goes to the next time step after taking the maximum number of transitions.

Maximum Number of Iterations

In a super step, your chart always takes at least one transition. Therefore, when you set a maximum number of iterations in each super step, the chart takes that number of transitions plus 1. For example, if you specify 10 as the maximum number of iterations, your chart takes 11 transitions in each super step. To set maximum number of iterations in each super step, see "Enable Super Step Semantics" on page 3-73.

Enable Super Step Semantics

To enable super step semantics:

1 Right-click inside the top level of a chart and select **Properties** from the context menu.

2   In the Chart properties dialog box, select the **Enable Super Step Semantics** check box.

Two additional fields appear below that check box.
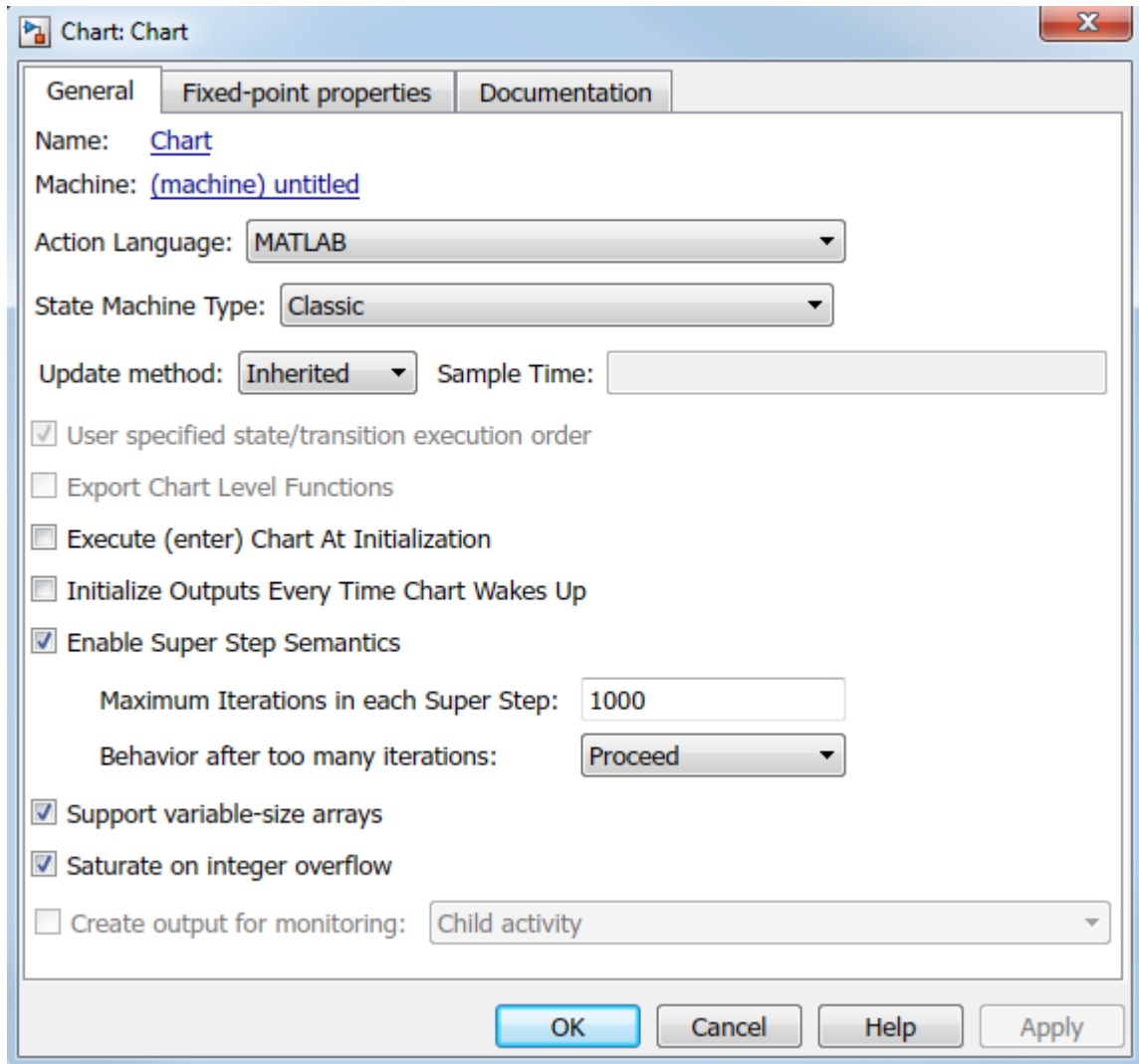
3   Enter a value in the field **Maximum Iterations in each Super Step**.

The chart always takes one transition during a super step, so the value N that you specify represents the maximum number of *additional* transitions (for a total of N+1). Try to choose a number that allows the chart to reach a stable state within the time step, based on the mode logic of your chart. For more information, see "Maximum Number of Iterations" on page 3-73

4 Select an action from the drop-down menu in the field **Behavior after too many iterations**.

Your selection determines how the chart behaves during simulation if it exceeds the maximum number of iterations in the super step before reaching a stable state.

Behavior	Description
Proceed	The chart goes back to sleep with the last active state configuration, that is, after updating local data at the last valid transition in the super step.
Throw Error	Simulation stops and the chart generates an error, indicating that too many iterations occurred while trying to reach a stable state.
	Note Selecting **Throw Error** can help detect infinite loops in transition cycles (see "Detection of Infinite Loops in Transition Cycles" on page 3-79.

Note This option is relevant only for simulation targets. For embedded targets, code generation goes to the next time step rather than generating an error.

Super Step Example

The following model shows how super step semantics differs from default semantics:
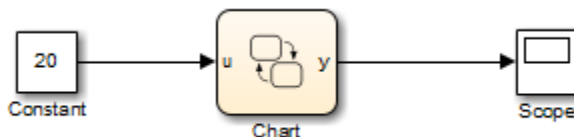
In this model, a Constant block outputs a constant value of 20 to input u in a Stateflow chart. Because the value of u is always 20, each transition in the chart is valid:
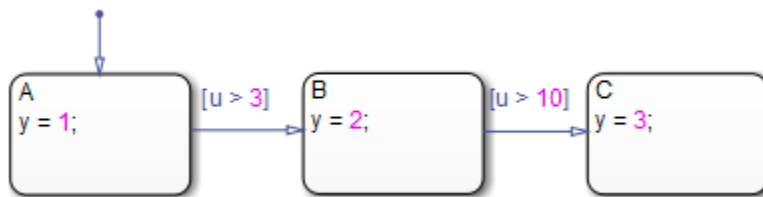
By default, the chart takes only one transition in each simulation step, incrementing y each time.
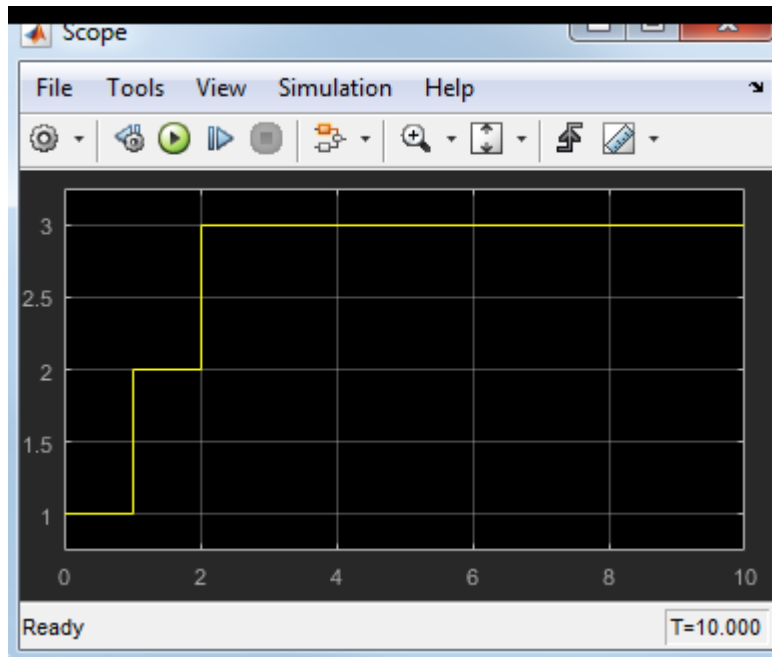
When you enable super step semantics, the chart takes all valid transitions in each time step, stopping at state C with y = 3.
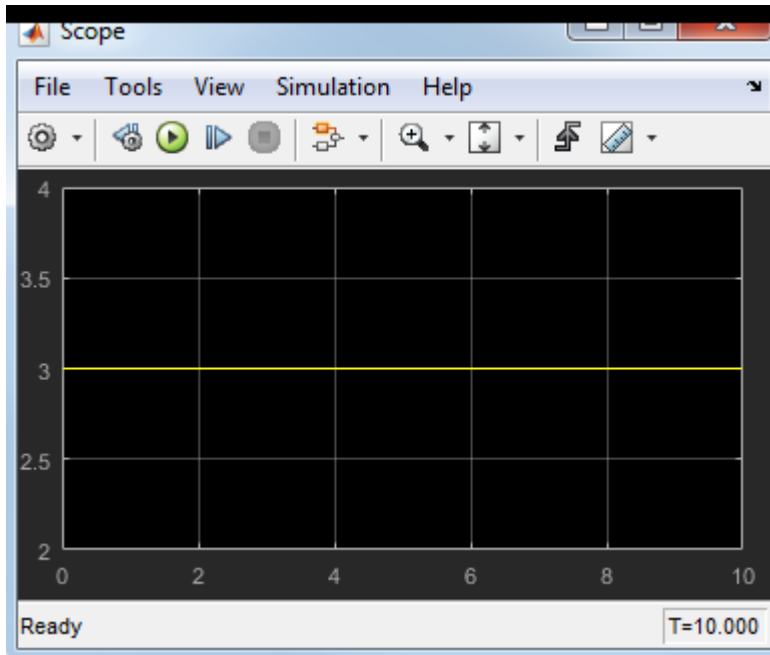
How Super Step Semantics Works with Multiple Input Events

When you enable super step semantics for a chart with multiple active input events, the chart takes all valid transitions for the first active event before it begins processing the next active event. For example, consider the following model:
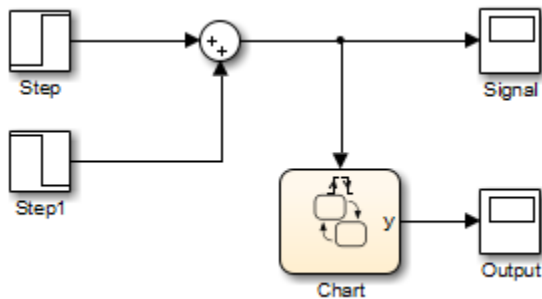
In this model, the Sum block produces a 2-by-1 vector signal that goes from [0,0] to [1,1] at time t = 1. As a result, when the model wakes up the chart, events E1 and E2 are both active:
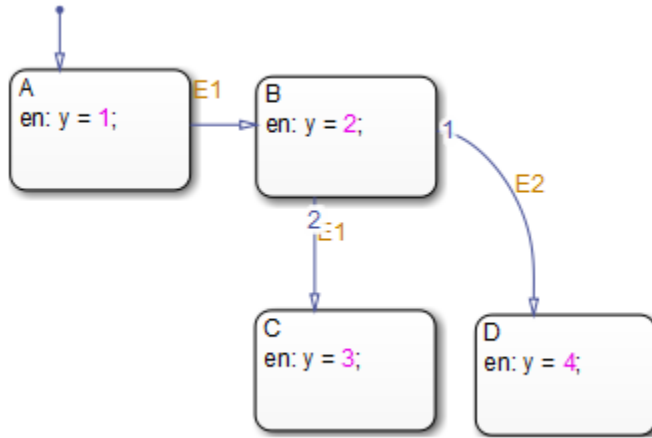
If you enable super step semantics, the chart takes all valid transitions for event E1. The chart takes transitions from state A to B and then from state B to C in a single super step. The scope shows that y = 3 at the end of the super step:
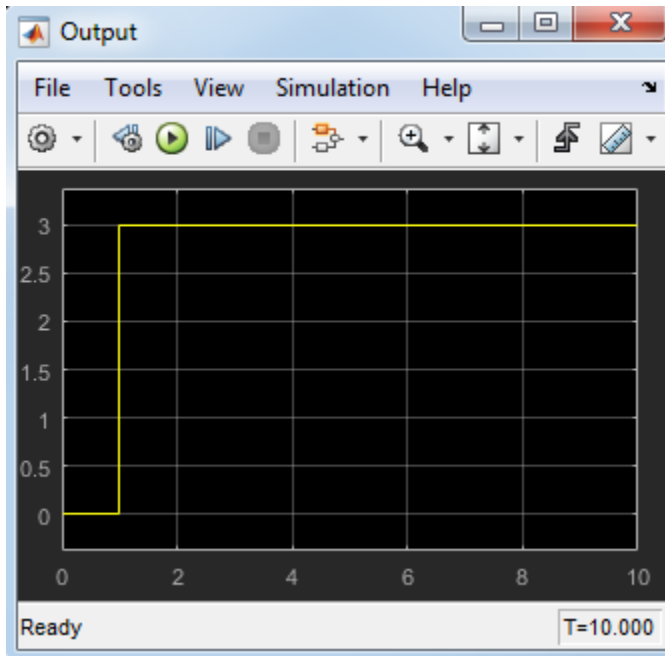
In a super step, this chart never transitions to state D because there is no path from state C to state D.

Detection of Infinite Loops in Transition Cycles

If your chart contains transition cycles, taking multiple transitions in a single time step can cause infinite loops. Consider the following example:
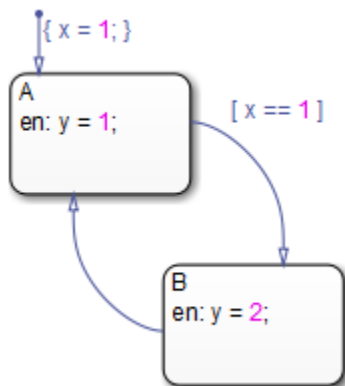
In this example, the transitions between states A and B cycle and produce an infinite loop because the value of x remains constant at 1. One way to detect infinite loops is to configure your chart to generate an error if it reaches a maximum number of iterations in a super step. See "Enable Super Step Semantics" on page 3-73.

How Events Drive Chart Execution

How Stateflow Charts Respond to Events

Stateflow charts in Simulink models execute in response to an event in a cyclical manner.
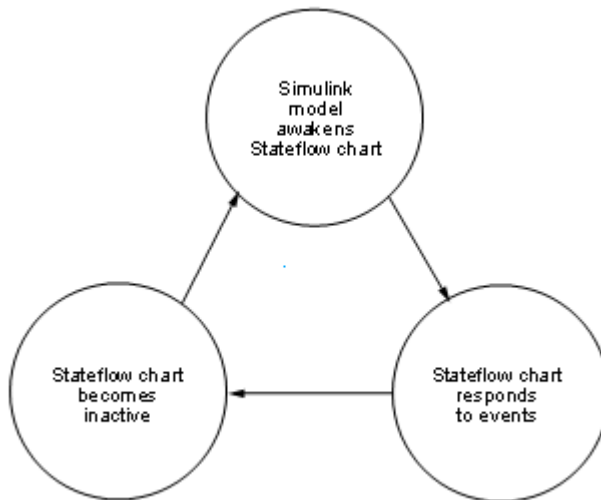
Because a chart runs on a single thread, actions that take place based on an event are atomic to that event. All activity caused by the event in the chart finishes before execution returns to the activity that was taking place before receiving the event. Once an event initiates an action, the action completes unless interrupted by an early return.

Sources for Stateflow Events

Simulink events awaken Stateflow charts. You can use events to control the processing of your charts by broadcasting events, as described in "Broadcast Local Events to Synchronize Parallel States" on page 12-45. For examples using event broadcasting and directed event broadcasting, see:

- Directed Event Broadcasting on page 12-45
- "Broadcast Events to Parallel (AND) States Using Condition Actions" on page B-12

- "Avoid Cyclic Behavior" on page B-14
- "Broadcast Events in State Actions" on page B-41
- "Broadcast Events in Transition Actions" on page B-43
- "Broadcast Events in Condition Actions" on page B-46

Events have hierarchy (a parent) and scope. The parent and scope together define a range of access to events. The parent of an event usually determines who can trigger on the event (has receive rights). See the **Name** and **Parent** fields for an event in "Set Properties for an Event" on page 10-6 for more information.

How Charts Process Events

Stateflow charts process events from the top down through the chart hierarchy:

1 Executes `during` and on *event_name* actions for the active state

2 Checks for valid transitions in substates

All events, except for the output edge trigger to a Simulink block (see the following note), have the following execution in a chart:

1 If the *receiver* of the event is active, then it executes (see "Execution of a Stateflow Chart" on page 3-44). (The event *receiver* is the parent of the event unless a directed event broadcast occurs using the `send()` function.)

2 If the receiver of the event is not active, nothing happens.

3 After broadcasting the event, the broadcaster performs early return logic based on the type of action statement that caused the event.

To learn about early return logic, see "Early Return Logic for Event Broadcasts" on page 3-93.

Note Output edge-trigger event execution in a Simulink model is equivalent to toggling the value of an output data value between 1 and 0. It is not treated as a Stateflow event. See "Define Edge-Triggered Output Events" on page 24-15.

Group and Execute Transitions

Transition Flow Chart Types

Before executing transitions for an active state or chart, Stateflow software groups transitions by the following types:

- Default flow charts are all default transition segments that start with the same parent.
- Inner flow charts are all transition segments that originate on a state and reside entirely within that state.
- Outer flow charts are all transition segments that originate on the respective state but reside at least partially outside that state.

Each set of flow charts includes other transition segments connected to a qualifying transition segment through junctions and transitions. Consider the following example:
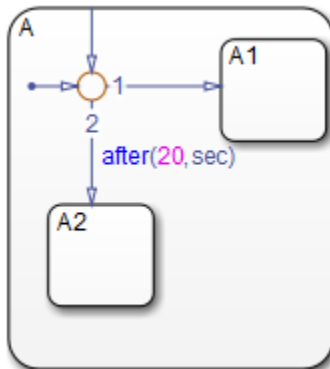
In this example, state A has both an inner and a default transition that connect to a junction with outgoing transitions to states A.A1 and A.A2. If state A is active, its set of inner flow charts includes:

- The inner transition
- The outgoing transitions from the junction to state A.A1 and A.A2

In addition, the set of default flow charts for state A includes:

- The default transition to the junction

- The two outgoing transitions from the junction to state `A.A1` and `A.A2`

In this case, the two outgoing transition segments from the junction are members of more than one flow chart type.

Order of Execution for a Set of Flow Charts

Each flow chart group executes in the order of group priority until a valid transition appears. The default transition group executes first, followed by the outer transitions group and then the inner transitions group. Each flow chart group executes as follows:

1   Order the group's transition segments for the active state.

An active state can have several possible outgoing transitions. The chart orders these transitions before checking them for validity. See "Transition Evaluation Order" on page 3-65.

2   Select the next transition segment in the set of ordered transitions.

3   Test the transition segment for validity.

4   If the segment is invalid, go to step 2.

5   If the destination of the transition segment is a state, do the following:

   a   Testing of transition segments stops and a transition path forms by backing up and including the transition segment from each preceding junction back to the starting transition.

   b   The states that are the immediate substates of the parent of the transition path exit (see "Exit a State" on page 3-58).

   c   The transition action from the final transition segment of the full transition path executes.

   d   The destination state becomes active (see "Enter a Chart or State" on page 3-50).

6   If the destination is a junction with no outgoing transition segments, do the following:

   a   Testing stops without any state exits or entries.

7   If the destination is a junction with outgoing transition segments, repeat step 1 for the set of outgoing segments.

8   After testing all outgoing transition segments at a junction, take the following actions:

a Backtrack the incoming transition segment that brought you to the junction.

b Continue at step 2, starting with the next transition segment after the backup segment.

The set of flow charts completes execution when all starting transitions have been tested.

Execution Order for Parallel States

Ordering for Parallel States

Although multiple parallel (AND) states in the same chart execute concurrently, the Stateflow chart must determine when to activate each one during simulation. This ordering determines when each parallel state performs the actions that take it through all stages of execution.

Unlike exclusive (OR) states, parallel states do not typically use transitions. Instead, order of execution depends on:

- Explicit ordering

  Specify explicitly the execution order of parallel states on a state-by-state basis (see "Explicit Ordering of Parallel States" on page 3-86).

- Implicit ordering

  Override explicit ordering by letting a Stateflow chart use internal rules to order parallel states (see "Implicit Ordering of Parallel States" on page 3-88).

Parallel states are assigned priority numbers based on order of execution. The lower the number, the higher the priority. The priority number of each state appears in the upper right corner.

Because execution order is a chart property, all parallel states in the chart inherit the property setting. You cannot mix explicit and implicit ordering in the same Stateflow chart. However, you can mix charts with different ordering modes in the same Simulink model.

Explicit Ordering of Parallel States

By default, a Stateflow chart orders parallel states explicitly based on execution priorities you set.

How Explicit Ordering Works

When you open a new Stateflow chart — or one that does not yet contain any parallel states — the chart automatically assigns priority numbers to parallel states in the order

you create them. Numbering starts with the next available number within the parent container.

When you enable explicit ordering in a chart that contains implicitly ordered parallel states, the implicit priorities are preserved for the existing parallel states. When you add new parallel states, execution order is assigned in the same way as for new Stateflow charts — in order of creation.

You can reset execution order assignments at any time on a state-by-state basis, as described in "Set Execution Order for Parallel States Individually" on page 3-88. When you change execution order for a parallel state, the Stateflow chart automatically renumbers the other parallel states to preserve their relative execution order. For details, see "Order Maintenance for Parallel States" on page 3-89.

Order Parallel States Explicitly

To use explicit ordering for parallel states, perform these tasks:

1   "Enable Explicit Ordering at the Chart Level" on page 3-87
2   "Set Execution Order for Parallel States Individually" on page 3-88

Enable Explicit Ordering at the Chart Level

To enable explicit ordering for parallel states, follow these steps:

1   Right-click inside the top level of the chart and select **Properties** from the context menu.

The Chart properties dialog box appears.
2   Select the **User specified state/transition execution order** check box.
3   Click **OK**.

If your chart already contains parallel states that have been ordered implicitly, the existing priorities are preserved until you explicitly change them. When you add new parallel states in explicit mode, your chart automatically assigns priorities based on order of creation (see "How Explicit Ordering Works" on page 3-86). However you can now explicitly change execution order on a state-by-state basis, as described in "Set Execution Order for Parallel States Individually" on page 3-88.

Set Execution Order for Parallel States Individually

In explicit ordering mode, you can change the execution order of individual parallel states. Right-click the parallel state of interest and select a new priority from the **Execution Order** menu.

Implicit Ordering of Parallel States

Rules of Implicit Ordering for Parallel States

In implicit ordering mode, a Stateflow chart orders parallel states implicitly based on location. Priority goes from top to bottom and then left to right, based on these rules:

- The higher the vertical position of a parallel state in the chart, the higher the execution priority for that state.
- Among parallel states with the same vertical position, the leftmost state receives highest priority.

The following example shows how these rules apply to top-level parallel states and parallel substates.

Note Implicit ordering creates a dependency between design layout and execution priority. When you rearrange parallel states in your chart, you can accidentally change order of execution and affect simulation results. For more control over your designs, use the default explicit ordering mode to set execution priorities.

Order Parallel States Implicitly

To use implicit ordering for parallel states, follow these steps:

1 Right-click inside the top level of the chart and select **Properties** from the context menu.
2 In the Chart properties dialog box, clear the **User specified state/transition execution order** check box.
3 Click **OK**.

Order Maintenance for Parallel States

Whether you use explicit or implicit ordering, a chart tries to reconcile execution priorities when you remove, renumber, or add parallel states. In these cases, a chart reprioritizes the parallel states to:

- Fill in gaps in the sequence so that ordering is contiguous

- Ensure that no two states have the same priority

- Preserve the intended relative priority of execution

How a Chart Preserves Relative Priorities in Explicit Mode

For explicit ordering, a chart preserves the user-specified priorities. Consider this example of explicit ordering:
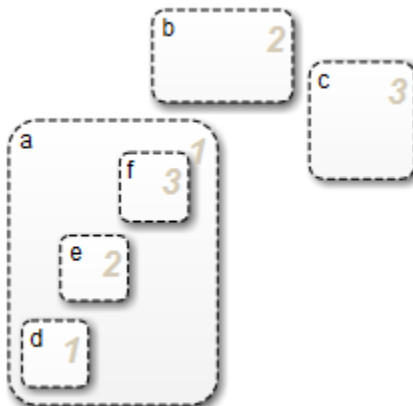
Because of explicit ordering, the priority of each state and substate matches the order of creation in the chart. The chart reprioritizes the parallel states and substates when you perform these actions:

1   Change the priority of top-level state b to *3*.

2   Add a top-level state g.

3   Remove substate e.
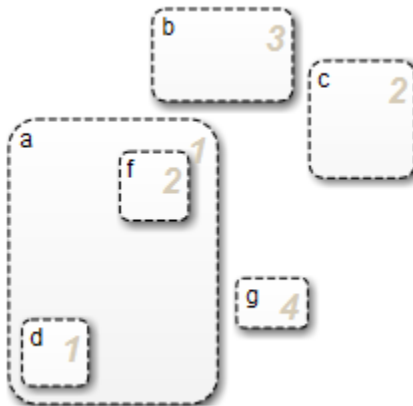
The chart preserves the priority set explicitly for top-level state b, but renumbers all other parallel states to preserve their prior relative order.

How a Chart Preserves Relative Priorities in Implicit Mode

For implicit ordering, a chart preserves the intended relative priority based on geometry. Consider this example of implicit ordering:
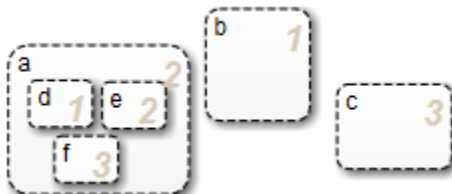
If you remove top-level state b and substate e, the chart automatically reprioritizes the remaining parallel states and substates to preserve implicit geometric order:

Execution Priorities in Restored States

There are situations in which you need to restore a parallel state after you remove it from a Stateflow chart. In implicit ordering mode, a chart reassigns the execution priority based on where you restore the state. If you return the state to its original location in the chart, you restore its original priority.

However, in explicit ordering mode, a chart cannot always reinstate the original execution priority to a restored state. It depends on *how* you restore the state.

If you remove a state by...	And restore the state by...	What is the priority?
Deleting, cutting, dragging outside the boundaries of the parent state, or dragging so its boundaries overlap the parent state	Using the undo command	The original priority is restored.
Dragging outside the boundaries of the parent state or so its boundaries overlap the parent state *and* releasing the mouse button	Dragging it back into the parent state	The original priority is lost. The Stateflow chart treats the restored state as the last created and assigns it the lowest execution priority.
Dragging outside the boundaries of the parent state or so its boundaries overlap the parent state *without* releasing the mouse button	Dragging it back into the parent state	The original priority is restored.

If you remove a state by...	And restore the state by...	What is the priority?
Dragging so its boundaries overlap one or more sibling states	Dragging it to a location with no overlapping boundaries inside the same parent state	The original priority is restored.
Cutting	Pasting	The original priority is lost. The Stateflow chart treats the restored state as the last created and assigns it the lowest execution priority.

Switching Between Explicit and Implicit Ordering

If you switch to implicit mode after explicitly ordering parallel states, the Stateflow chart resets execution order to follow implicit rules of geometry. However, if you switch from implicit to explicit mode, the chart does not restore the original explicit execution order.

Execution Order of Parallel States in Boxes and Subcharts

When you group parallel states inside a box, the states retain their relative execution order. In addition, the Stateflow chart assigns the box its own priority based on the explicit or implicit ordering rules that apply. This priority determines when the chart activates the parallel states inside the box.

When you convert a state with parallel decomposition into a subchart, its substates retain their relative execution order based on the prevailing explicit or implicit rules.

Early Return Logic for Event Broadcasts

Guidelines for Proper Chart Behavior

These guidelines ensure proper chart behavior in event-driven systems:

- When a state is active, its parent should also be active.
- A state (or chart) with exclusive (OR) decomposition must never have more than one active substate.
- If a parallel state is active, siblings with higher priority must also be active.

How Early Return Logic Works

Stateflow charts run on a single thread. Therefore, charts must interrupt current activity to process events. Activity based on an event broadcast from a state or transition action can conflict with the current activity. Charts resolve these conflicts by using early return logic for event broadcasts as follows:

Action Type	Early Return Logic
Entry	If the state is no longer active at the end of the event broadcast, the chart does not perform remaining steps for entering a state.
Exit	If the state is no longer active at the end of the event broadcast, the chart does not perform remaining `exit` actions or transitions from state to state.
During	If the state is no longer active at the end of the event broadcast, the chart does not perform remaining steps for executing an active state.
Condition	If the origin state of the inner or outer flow chart — or parent state of the default flow chart — are no longer active at the end of the event broadcast, the chart does not perform remaining steps for executing the flow chart.
Transition	If the parent of the transition path is not active — or if that parent has an active substate — the chart does not perform remaining transition actions and state `entry` actions.

Example of Early Return Logic
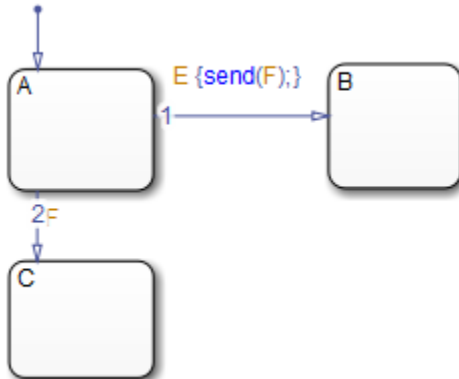
In this example, assume that state A is initially active. Event E occurs, causing the following behavior:

1 The chart root checks to see if there is a valid transition out of the active state A as a result of event E.

2 A valid transition to state B exists.

3 The condition action of the valid transition executes and broadcasts event F.

   Event F interrupts the transition from A to B.

4 The chart root checks to see if there is a valid transition out of the active state A as a result of event F.

5 A valid transition to state C exists.

6 State A executes its `exit` action.

7 State A becomes inactive.

8 State C becomes active.

9 State C executes and completes its `entry` action.

State C is now the only active substate of its chart. The Stateflow chart cannot return to the transition from state A to state B and continue after the condition action that broadcast event F (step 3). First, its source, state A, is no longer active. Second, if the chart allowed the transition, state B would become the second active substate of the chart. This behavior violates the guideline that a state (or chart) with exclusive (OR)

decomposition can never have more than one active substate. Therefore, the chart uses early return logic and halts the transition from state A to state B.

Tip Avoid using undirected local event broadcasts, which can cause unwanted recursive behavior in your chart. Use the `send` operator for directed local event broadcasts. For more information, see "Broadcast Local Events to Synchronize Parallel States" on page 12-45.

You can set the diagnostic level for detecting undirected local event broadcasts. In the Model Configuration Parameters dialog box, go to the **Diagnostics > Stateflow** pane and set the **Undirected event broadcasts** diagnostic to `none`, `warning`, or `error`. The default setting is `warning`.

4

Create Stateflow Charts

Model Reactive Systems in Stateflow

Identify System Attributes

Before you build a Stateflow chart in a Simulink model, identify your system attributes by answering these questions:

1 What are your interfaces?

    a What are the event triggers to which your system reacts?

    b What are the inputs to your system?

    c What are the outputs from your system?

2 Does your system have any operating modes?

    a If the answer is yes, what are the operating modes?

    b Between which modes can you transition? Are there any operating modes that run in parallel?

If your system has no operating modes, the system is stateless. If your system has operating modes, the system is modal.

Select a State Machine Type

After identifying your system attributes, the first step is to create a new chart. For more information, see `sfnew`. Select one of the following state machine types:

- Classic — The default machine type. Provides the full set of semantics for MATLAB charts and C charts.
- Mealy — Machine type in which output is a function of inputs *and* state.
- Moore — Machine type in which output is a function of state.

For more information, see "How Stateflow Objects Interact During Execution" on page 3-5, "Differences Between MATLAB and C as Action Language Syntax" on page 13-6, and "Overview of Mealy and Moore Machines" on page 7-2.

Specify State Actions and Transition Conditions

After you create an empty chart, answer the following questions:

1   For each state, what are the actions you want to perform?

2   What are the rules for transitioning between your states? If your chart has no states, what are the rules for transitioning between branches of your flow logic?

Using your answers to those questions, specify state actions and transition conditions:

1   Draw states to represent your operating modes, if any. See "Represent Operating Modes by Using States" on page 4-5.

2   Implement the state actions by adding state labels that use the appropriate syntax. See "State Action Types" on page 12-2.

3   Draw transitions to represent the direction of flow logic, between states or between branches of your flow chart. See "Transition Between Operating Modes" on page 4-19.

4   Implement the transition conditions by adding transition labels that use the appropriate syntax. See "Transition Action Types" on page 12-7.

Define Persistent Data to Store State Variables

After adding state actions and transition conditions to your chart, determine if the chart requires any local or persistent data to store state variables. If so, follow these steps:

1   Add local data to the appropriate level of the chart hierarchy. See "Add Stateflow Data" on page 9-2.

    You can also use the Symbol Wizard to add data to your chart. See "Resolve Symbols Through the Symbol Wizard" on page 30-36.

2   Specify the type, size, complexity, and other data properties. See "Set Data Properties" on page 9-5.

Simplify State Actions and Transition Conditions with Function Calls

State actions and transition conditions can be complex enough that defining them inline on the state or transition is not feasible. In this case, express the actions or conditions using one of the following types of Stateflow functions:

- Flow chart — Encapsulate flow charts containing if-then-else, switch-case, for, while, or do-while patterns.

- MATLAB — Write matrix-oriented algorithms; call MATLAB functions for data analysis and visualization.
- Simulink — Call Simulink function-call subsystems directly to streamline design and improve readability.
- Truth table — Represent combinational logic for decision-making applications such as fault detection and mode switching.

Use the function format that is most natural for the type of calculation in the state action or transition condition. For more information on the four types of functions, see:

- "Reuse Logic Patterns by Defining Graphical Functions" on page 8-16
- "Reuse MATLAB Code by Defining MATLAB Functions" on page 28-2
- "Reuse Simulink Components in Stateflow Charts" on page 29-2
- "Reuse Combinatorial Logic by Defining Truth Tables" on page 27-2

If the four types of Stateflow functions do not work, you can write your own C or C++ code for integration with your chart. For more information about custom code integration, see "Reuse Custom Code in Stateflow Charts" on page 30-26.

Check That Your System Representation Is Complete

Does your Stateflow chart fully express the logical or event-driven components of your system?

- If the answer is yes, you are done.
- If the answer is no, you can create a separate chart or add hierarchy to your current chart.

  - To create a new chart, repeat all the steps in this basic workflow.
  - To add hierarchy, repeat the previous three steps on lower levels of the current chart.

Represent Operating Modes by Using States

A state describes an operating mode of a reactive system. In a Stateflow chart, states are used for sequential design to create state transition diagrams.

States can be active or inactive. The activity or inactivity of a state can change depending on events and conditions. The occurrence of an event drives the execution of the state transition diagram by making states become active or inactive. For more information, see "States" on page 2-7.

Create a State

You create states by drawing them in the editor for a particular chart (block). Follow these steps:

1   Select the State tool:

2   Move your pointer into the drawing area.

In the drawing area, the pointer becomes state-shaped (rectangular with oval corners).

3   Click in a particular location to create a state.

The created state appears with a question mark (`?`) label in its upper left-hand corner.

4   Click the question mark.

A text cursor appears in place of the question mark.

5   Enter a name for the state and click outside of the state when finished.

The label for a state specifies its required name and optional actions. See "Label States" on page 4-15 for more detail.

Move and Resize States

To move a state, do the following:

1   Click and drag the state.

2   Release it in a new position.

To resize a state, do the following:

1   Place your pointer over a corner of the state.

When your pointer is over a corner, it appears as a double-ended arrow (PC only; pointer appearance varies with other platforms).

2   Click and drag the state's corner to resize the state and release the left mouse button.

Create Substates and Superstates

A substate is a state that can be active only when another state, called its parent, is active. States that have substates are known as superstates. To create a substate, click the State tool and drag a new state into the state you want to be the superstate. A Stateflow chart creates the substate in the specified parent state. You can nest states in this way to any depth. To change the parentage of a substate, drag it from its current parent in the chart and drop it in its new parent.

Note A parent state must be graphically large enough to accommodate all its substates. You might need to resize a parent state before dragging a new substate into it. You can bypass the need for a state of large graphical size by declaring a superstate to be a subchart. See "Encapsulate Modal Logic by Using Subcharts" on page 8-5 for details.

Group States

When to Group a State

Group a state to move all graphical objects inside a state together. When you group a state, the chart treats the state and its contents as a single graphical unit. This behavior simplifies editing of a chart. For example, moving a grouped state moves all substates and functions inside that state.

How to Group a State

You can group a state by right-clicking it and then selecting **Group & Subchart** > **Group** in the context menu. The state appears shaded in gray to indicate that it is now grouped.
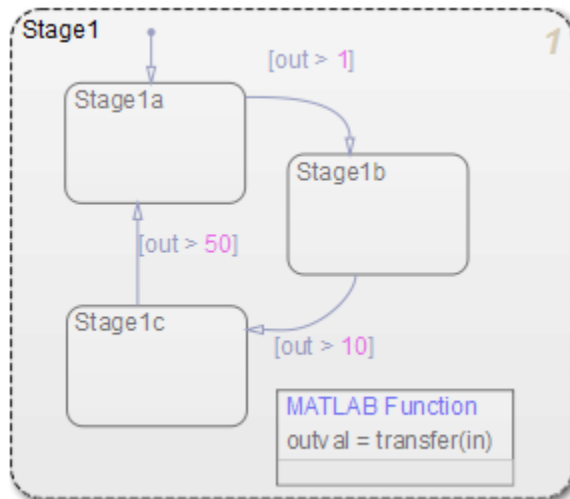
When to Ungroup a State

You must ungroup a state before performing these actions:

- Selecting objects inside the state
- Moving other graphical objects into the state

  If you try to move objects such as states and graphical functions into a grouped state, you see an invalid intersection error message. Also, the objects with an invalid intersection have a red border.

How to Ungroup a State

You can ungroup a state by right-clicking it and then clearing **Group & Subchart** > **Group** in the context menu. The background of the state no longer appears gray.
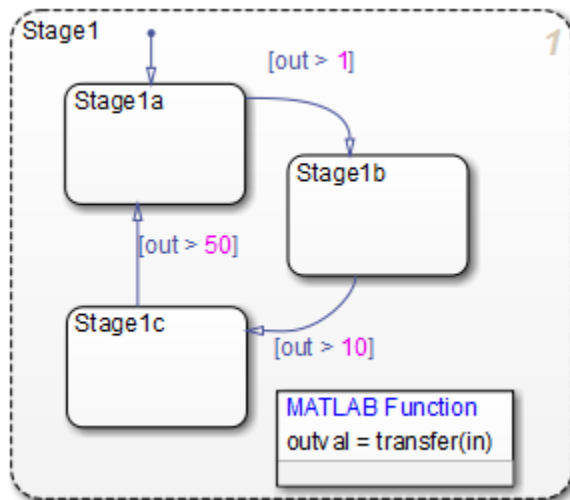
Specify Substate Decomposition

You specify whether a superstate contains parallel (AND) states or exclusive (OR) states by setting its decomposition. A state whose substates are all active when it is active has parallel (AND) decomposition. A state in which only one substate is active when it is active has exclusive (OR) decomposition. An empty state's decomposition is exclusive.

To alter a state's decomposition, select the state, right-click to display the state's **Decomposition** context menu, and select **OR (Exclusive)** or **AND (Parallel)** from the menu.

You can also specify the state decomposition of a chart. In this case, the Stateflow chart treats its top-level states as substates. The chart creates states with exclusive decomposition. To specify a chart's decomposition, deselect any selected objects, right-click to display the chart's **Decomposition** context menu, and select **OR (Exclusive)** or **AND (Parallel)** from the menu.

The appearance of the substates indicates the decomposition of their superstate. Exclusive substates have solid borders, parallel substates, dashed borders. A parallel substate also contains a number in its upper right corner. The number indicates the activation order of the substate relative to its sibling substates.

Specify Activation Order for Parallel States

You can specify activation order by using one of two methods: explicit or implicit ordering.

- By default, when you create a new Stateflow chart, explicit ordering applies. In this case, you specify the activation order on a state-by-state basis.
- You can also override explicit ordering by letting the chart order parallel states based on location. This mode is known as implicit ordering.

For more information, see "Explicit Ordering of Parallel States" on page 3-86 and "Implicit Ordering of Parallel States" on page 3-88.

Note The activation order of a parallel state appears in its upper right corner.

Change State Properties

Use the State dialog box to view and change the properties for a state. To access the State dialog box:

1 Right-click the state and select **Properties**.

The State properties dialog box appears. For descriptions of properties, see "Properties You Can Set in the General Pane" on page 4-9 and "Properties You Can Set in the Logging Pane" on page 4-11.

2 Modify property settings and then click one of these buttons:

- **Apply** to save the changes and keep the State dialog box open
- **Cancel** to return to the previous settings
- **OK** to save the changes and close the dialog box
- **Help** to display the documentation in an HTML browser window

Properties You Can Set in the General Pane

The **General** pane of the State properties dialog box appears as shown.
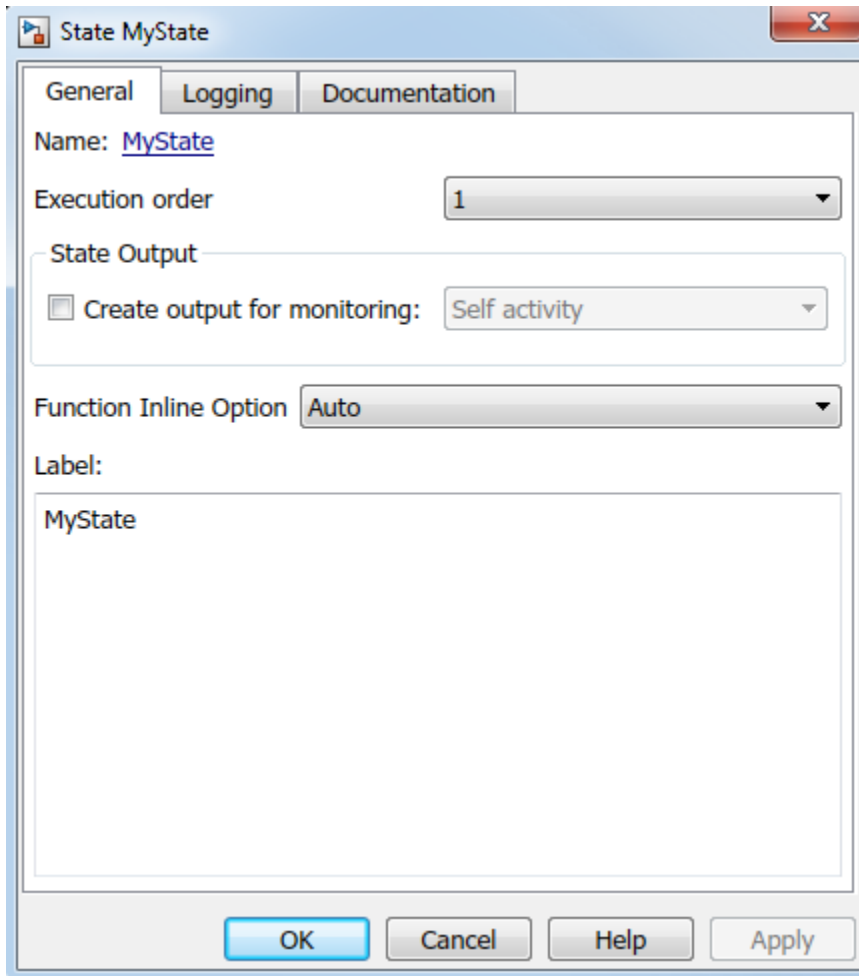
You can set these properties in the **General** pane.

Property	Description
Name	Stateflow chart name; read-only; click this hypertext link to bring the state to the foreground.

Property	Description
Execution order	Set the execution order of a parallel (AND) state. This property does not appear for exclusive (OR) states. See "Execution Order for Parallel States" on page 3-86.
Create data for monitoring	Select this option to create state activity data. See "Monitor State Activity Through Active State Data" on page 24-26.

Function Inline Option	Select one of these options to control the inlining of state functions in generated code:
	
	• **Auto**
	
	  Inlines state functions based on an internal heuristic.
	
	• **Inline**
	
	  Always inlines state functions in the parent function, as long as the function is not part of a recursion. See "Inline State Functions in Generated Code" (Simulink Coder)
	
	• **Function**
	
	  Creates separate static functions for each state.

Label	The label for the state, which includes the name of the state and its associated actions. See "Label States" on page 4-15.

Properties You Can Set in the Logging Pane

The **Logging** pane of the State properties dialog box appears as shown.
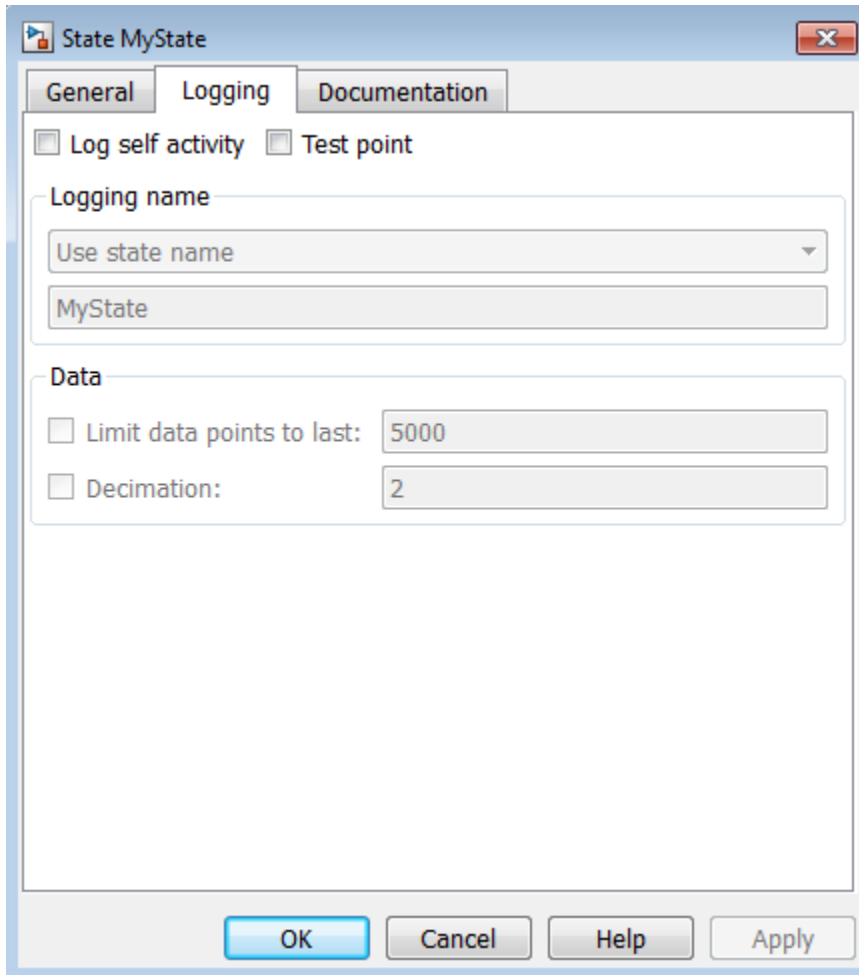
You can set these properties in the **Logging** pane.

Property	Description
Log self activity	Saves the self activity value to the MATLAB workspace during simulation.

Property	Description
Test point	Designates the state as a test point that can be monitored with a floating scope during model simulation. You can also log test point values into MATLAB workspace objects. See "Monitor Test Points in Stateflow Charts" on page 32-46.
Logging name	Specifies the name associated with the logged self activity. Simulink software uses the signal name as its logging name by default. To specify a custom logging name, select `Custom` from the list box and enter the new name in the adjacent edit field.
Limit data points to last	Limits the self activity logged to the most recent samples.
Decimation	Limits self activity logged by skipping samples. For example, a decimation factor of 2 saves every other sample.

Properties You Can Set in the Documentation Pane

The **Documentation** pane of the State properties dialog box appears as shown.
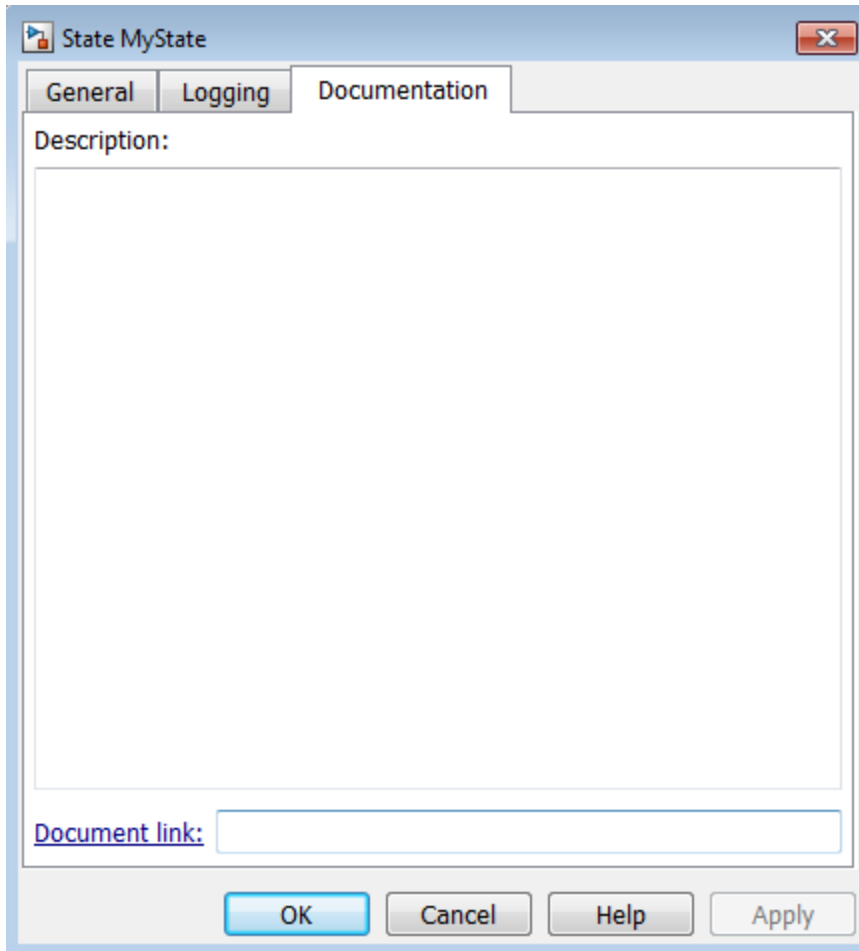
You can set these properties in the **Documentation** pane.

Property	Description
Description	Textual description or comment.
Document link	Enter a URL address or a general MATLAB command. Examples are `www.mathworks.com`, `mailto:email_address`, and `edit /spec/data/ speed.txt`.

Label States

The label for a state specifies its required name for the state and the optional actions executed when the state is entered, exited, or receives an event while it is active.

State labels have the following general format.

```
name/
entry:entry actions
during:during actions
exit:exit actions
bind:data and events
on event_or_message_name:on event_or_message_name actions
```

The italicized entries in this format have the following meanings:

Keyword	Entry	Description
Not applicable	*name*	A unique reference to the state with optional slash
`entry` or `en`	*entry actions*	Actions executed when a particular state is entered as the result of a transition taken to that state
`during` or `du`	*during actions*	Actions that are executed when a state receives an event while it is active with no valid transition away from the state
`exit` or `ex`	*exit actions*	Actions executed when a state is exited as the result of a transition taken away from the state
`bind`	*data* or *events*	Binds the specified data or events to this state. Bound data can be changed only by this state or its children, but can be read by other states. Bound events can be broadcast only by this state or its children.

Keyword	Entry	Description
on	*event_or_messag e_name* and *on event_name actions*	A specified event or message and Actions executed when a state is active and the specified event occurs or message is present. For more information, see "Synchronize Model Components by Broadcasting Events" on page 10-2 and "Communicate with Stateflow Charts by Sending Messages" on page 11-10.

Enter the Name

Initially, a state's label is empty. The Stateflow chart indicates this by displaying a ? in the state's label position (upper left corner). Begin labeling the state by entering a name for the state with the following steps:

1 Click the state.

    The state turns to its highlight color and a question mark character appears in the upper left-hand corner of the state.

2 Click the ? to edit the label.

    An editing cursor appears. You are now free to type a label.

    Enter the state's name in the first line of the state's label. Names are case sensitive. To avoid naming conflicts, do not assign the same name to sibling states. However, you can assign the same name to states that do not share the same parent.

    After labeling the state, click outside it. Otherwise, continue entering actions. To reedit the label, click the label text near the character position you want to edit.

Enter Actions

After entering the name of the state in the label, you can enter actions for any of the following action types:

- **Entry Actions** — begin on a new line with the keyword `entry` or `en`, followed by a colon, followed by one or more action statements on one or more lines. To separate multiple actions on the same line, use a comma or a semicolon.

You can begin entry actions on the same line as the state's name. In this case, begin the entry action with a forward slash (`/`) instead of the entry keyword.

- **Exit Actions** — begin on a new line with the keyword `exit` or `ex`, followed by a colon, followed by one or more action statements on one or more lines. To separate multiple actions on the same line, use a comma or a semicolon.

- **During Actions** — begin on a new line with the keyword `during` or `du`, followed by a colon, followed by one or more action statements on one or more lines. To separate multiple actions on the same line, use a comma or a semicolon.

- **Bind Actions** — begin on a new line with the keyword `bind` followed by a colon, followed by one or more data or events on one or more lines. To separate multiple actions on the same line, use a comma or a semicolon.

- **On Actions** — begin with the keyword `on`, followed by a space and the name of an event or message, followed by a colon, followed by one or more action statements on one or more lines, for example

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

To separate multiple actions on the same line, use a comma or a semicolon. If you want different events to trigger different actions, enter multiple `on` blocks in the state label. Each block specifies the action for a specific event or message, for example:

```
on ev1: action1(); on ev2: action2();
```

The execution of the actions you enter for a state is dependent only on their action type, and not the order in which you enter actions in the label. If you do *not* specify the action type explicitly for a statement, the chart treats that statement as an entry action.

Tip You can also edit the label in the properties dialog box for the state. See "Change State Properties" on page 4-9.

See Also

States

More About

- "State Action Types" on page 12-2
- "Eliminate Redundant Code by Combining State Actions" on page 12-11

- "Record State Activity by Using History Junctions" on page 8-2
- "Monitor State Activity Through Active State Data" on page 24-26

Transition Between Operating Modes

A transition is a line with an arrowhead that links one graphical object to another. In most cases, a transition represents the passage of the system from one mode (state) to another. A transition typically connects a source and a destination object. The source object is where the transition begins and the destination object is where the transition ends. For more information, see "Transitions" on page 2-18.

Create a Transition

Follow these steps to create transitions between states and junctions:

1   Place your pointer on or close to the border of a source state or junction.

    The pointer changes to crosshairs.

2   Click and drag a transition to a destination state or junction.

3   Release on the border of the destination state or junction.

Attached transitions obey the following rules:

- Transitions do not attach to the corners of states. Corners are used exclusively for resizing.

- Transitions exit a source and enter a destination at angles perpendicular to the source or destination surface.

- All transitions have smart behavior.

To delete a transition, click it and select **Edit > Cut**, or press the **Delete** key.

See the following sections for help with creating self-loop and default transitions:

- "Create Self-Loop Transitions" on page 4-23
- "Create Default Transitions" on page 4-23

Label Transitions

Transition labels contain syntax that accompanies the execution of a transition. The following topics discuss creating and editing transition labels:

- "Edit Transition Labels" on page 4-20
- "Transition Label Format" on page 4-20

For more information on transition concepts, see "Transition Label Notation" on page 2-20.

For more information on transition label contents, see "Transition Action Types" on page 12-7.

Edit Transition Labels

Label unlabeled transitions as follows:

1   Select (left-click) the transition.

The transition changes to its highlight color and a question mark (`?`) appears on the transition. The ? character is the default empty label for transitions.

2   Left-click the ? to edit the label.

You can now type a label.

To apply and exit the edit, deselect the object. To reedit the label, simply left-click the label text near the character position you want to edit.

Transition Label Format

Transition labels have the following general format:

`event_or_message [condition]{condition_action}/transition_action`

Specify, as appropriate, relevant names for `event_or_message`, `condition`, `condition_action`, and `transition_action`.

Label Field	Description
`event_or_message`	The event or message that causes the transition to be evaluated.
`condition`	Defines what, if anything, has to be true for the condition action and transition to take place.
`condition_action`	If the condition is true, the action specified executes and completes.

Label Field	Description
transition_action	This action executes after the source state for the transition is exited but before the destination state is entered. Transition actions are supported only in Stateflow charts in Simulink models.

Transitions do not need labels. You can specify some, all, or none of the parts of the label. Rules for writing valid transition labels include:

- Can have any alphanumeric and special character combination, with the exception of embedded spaces
- Cannot begin with a numeric character
- Can have any length
- Can have carriage returns in most cases
- Must have an ellipsis (...) to continue on the next line

Move Transitions

You can move transition lines with a combination of several individual movements. These movements are described in the following topics:

- "Bow the Transition Line" on page 4-21
- "Move Transition Attach Points" on page 4-22
- "Move Transition Labels" on page 4-22

In addition, transitions move along with the movements of states and junctions.

Bow the Transition Line

You can move or "bow" transition lines with the following procedure:

1 Place your pointer on the transition at any point along the transition except the arrow or attach points.

2 Click and drag your pointer to move the transition point to another location.

   Only the transition line moves. The arrow and attachment points do not move.

3 Release the mouse button to specify the transition point location.

The result is a bowed transition line. Repeat the preceding steps to move the transition back into its original shape or into another shape.

Move Transition Attach Points

You can move the source or end points of a transition to place them in exact locations as follows:

1   Place your pointer over an attach point until it changes to a small circle.
2   Click and drag your pointer to move the attach point to another location.
3   Release the mouse button to specify the new attach point.

The appearance of the transition changes from a solid to a dashed line when you detach and release a destination attach point. Once you attach the transition to a destination, the dashed line changes to a solid line.

The appearance of the transition changes to a default transition when you detach and release a source attach point. Once you attach the transition to a source, the appearance returns to normal.

Move Transition Labels

You can move transition labels to make the Stateflow chart more readable. To move a transition label, do the following:

1   Click and drag the label to a new location.
2   Release the left mouse button.

If you mistakenly click and then immediately release the left mouse button on the label, you will be in edit mode for the label. Press the **Esc** key to deselect the label and try again. You can also click the mouse on an empty location in the chart to deselect the label.

Change Transition Arrowhead Size

The arrowhead size is a property of the destination object. Changing one of the incoming arrowheads of an object causes all incoming arrowheads to that object to be adjusted to the same size. The arrowhead size of any selected transitions, and any other transitions ending at the same object, is adjusted.

To adjust arrowhead size:

1 Select the transitions whose arrowhead size you want to change.

2 Place your pointer over a selected transition and right-click to select **Arrowhead Size**.

3 Select an arrowhead size from the menu.

Create Self-Loop Transitions

A self-loop transition is a transition whose source and destination are the same state or junction. To create a self-loop transition:

1 Create the transition by clicking and dragging from the source state or junction.

2 Press the **S** key or right-click your mouse to enable a curved transition.

3 Continue dragging the transition tip back to a location on the source state or junction.

For the semantics of self-loops, see "Self-Loop Transitions" on page 2-27.

Create Default Transitions

A default transition is a transition with a destination (a state or a junction), but no apparent source object.

Click the **Default Transition** button in the toolbar and click a location in the drawing area close to the state or junction you want to be the destination for the default transition. Drag your pointer to the destination object to attach the default transition.

The size of the endpoint of the default transition is proportional to the arrowhead size. See "Change Transition Arrowhead Size" on page 4-22.

Default transitions can be labeled just like other transitions. See "Label Default Transitions" on page 2-33 for an example.

Change Transition Properties

Use the Transition properties dialog box to view and change the properties for a transition. To access the dialog box for a particular transition:

1 Right-click the transition and select **Properties**.

The Transition properties dialog box appears.

Transition [temp < 150]

Source:	(state) Stage7Debug/Air Controller.On
Destination:	(state) Stage7Debug/Air Controller.Off
Parent:	(chart) Stage7Debug/Air Controller

Execution order: 1

Label:

[temp < 150]

Description:

Document link:

OK   Cancel   Help   Apply

The following transition properties appear in the dialog box:

Field	Description
Source	Source of the transition; read-only; click the hypertext link to bring the transition source to the foreground.
Destination	Destination of the transition; read-only; click the hypertext link to bring the transition destination to the foreground.
Parent	Parent of this state; read-only; click the hypertext link to bring the parent to the foreground.
Execution order	The order in which the chart executes the transition.
Label	The transition's label. See "Transition Label Notation" on page 2-20 for more information on valid label formats.
Description	Textual description or comment.
Document link	Enter a Web URL address or a general MATLAB command. Examples are `www.mathworks.com`, `mailto:email_address`, and `edit/spec/data/speed.txt`.

2 After making changes, click one of these buttons:

- **Apply** to save the changes and keep the Transition dialog box open.
- **Cancel** to return to the previous settings for the dialog box.
- **OK** to save the changes and close the dialog box.
- **Help** to display Stateflow online help in an HTML browser window.

See Also

Transitions

More About

- "Transition Action Types" on page 12-7
- "Default Transitions" on page 2-33

- "Inner Transitions" on page 2-29
- "Self-Loop Transitions" on page 2-27

Stateflow Editor Operations

Stateflow Editor

Use the Stateflow Editor to draw, zoom, modify, print, and save a chart shown in the window.

Opening a Stateflow chart displays the chart in the Stateflow Editor.

To open a new Stateflow chart in the Stateflow Editor:

1   At the MATLAB command prompt, enter:

Command	Result
`sfnew`	Creates a chart with the default action language. For more information, see `sfnew`.
`sfnew -matlab`	Creates an empty chart with MATLAB as the action language.
`sfnew -C`	Creates an empty C chart.
`stateflow`	Creates an empty chart with the default action language and displays the Stateflow block library.

The Simulink Editor opens, with an empty chart.

2   Double-click the chart object.

The Stateflow Editor opens.
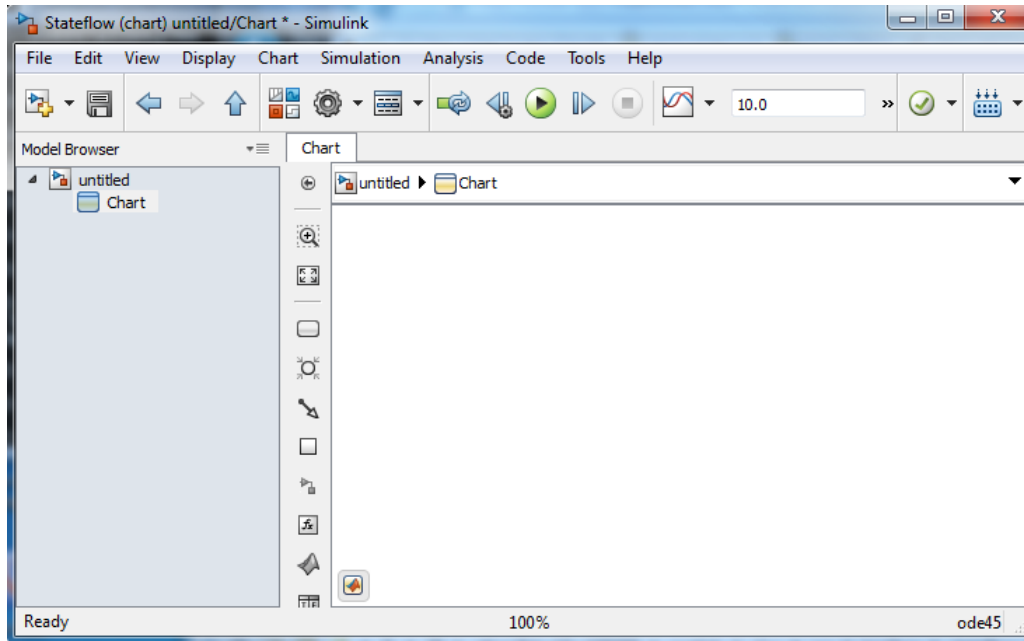
The Stateflow Editor window includes the following sections:

- **Title bar**
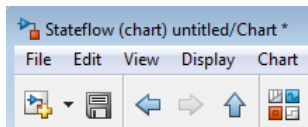
The full chart name appears here in *model name/chart name** format. The * character appears on the end of the chart name for a newly created chart or for an existing chart that has been edited but not saved yet.

- **Menu bar** and **toolbar**
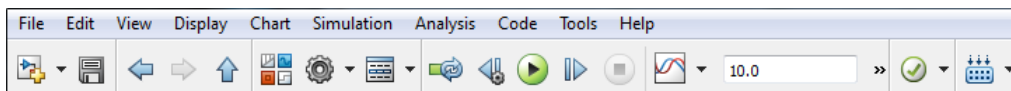
Most editor commands are available from the menu bar. The toolbar contains buttons for cut, copy, paste, and other commonly used editor commands. You can identify each

tool of the toolbar by placing your pointer over it until an identifying tool tip appears. The toolbar also contains buttons for navigating the chart hierarchy (see "Navigate Subcharts" on page 8-8).

- **Object palette**
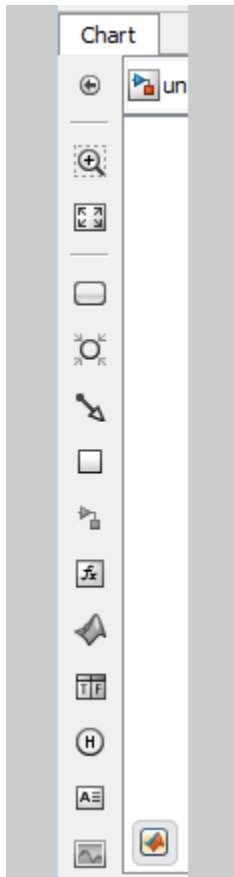
Displays a set of tools for drawing states, transitions, and other chart objects. To add an object, you can use the palette to:

- Click the icon for the object and move the cursor to the spot in the drawing area where you want to place the object.

- Drag the icon for the object into the drawing area.

- Double-click the icon and then click multiple times in the drawing area to make copies of the object.

- **Explorer bar**

  The breadcrumb shows the systems that you have open in the editor. Click a system in the breadcrumb to display that system.

- **Model Browser**
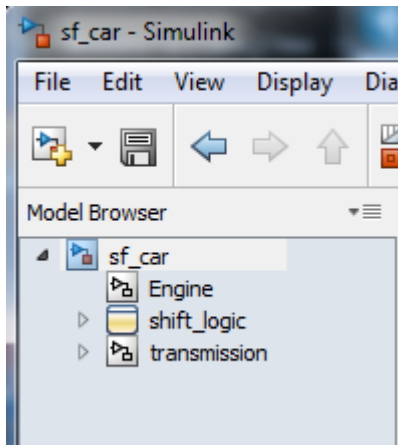
  Click the double arrows in the bottom left corner to open or close a tree-structured view of the model in the editor.

- **Drawing area** — This area displays an editable copy of a chart.

- **Context menus** (shortcut menus) — These menus pop up from the drawing area when you right-click an object. They display commands that apply only to that object. If you right-click an empty area of the chart, the context menu applies to the chart object.

- **Status information** — Near the top of the editor, you can see (and reset) the simulation time and the simulation mode. The bottom status bar displays the status of the Stateflow processing, tool tips, the zoom factor, and the solver.

Undo and Redo Editor Operations

You can undo and redo operations that you perform in a chart. When you undo an operation, you reverse the last edit operation that you performed. After you undo operations in the chart, you can also redo them one at a time.

- To undo an operation in the chart, select **Edit > Undo**.
- To redo an operation in the chart, select **Edit > Redo**.

You can undo and redo many operations you complete on Stateflow objects in the Symbols or Property Inspector windows.

Exceptions for Undo

You can undo or redo all editor operations, with the following exceptions:

- You cannot undo the operation of turning subcharting off for a state previously subcharted.

  To understand subcharting, see "Encapsulate Modal Logic by Using Subcharts" on page 8-5.
- You cannot undo the drawing of a supertransition or the splitting of an existing transition.

  Splitting of an existing transition refers to the redirection of the source or destination of a transition segment that is part of a supertransition. For a description of supertransitions, see "Draw a Supertransition Into a Subchart" on page 8-11 and "Draw a Supertransition Out of a Subchart" on page 8-13.
- You cannot undo any changes made to the chart using the Stateflow API.

  For a description of the Stateflow API, see "Stateflow Programmatic Interface".

Note When you perform one of the preceding operations, the undo and redo buttons are disabled from undoing and redoing any prior operations.

Specify Colors and Fonts in a Chart

You can change the way Stateflow displays an individual element of a chart or specify the global display options used throughout the entire chart.

Change Size of a Single Element

To change the display size for a single element in the chart, right-click the element, and then select a new **Format** option from the context menu. The options available depend on the element that you select.

Option	States	Transitions	Junctions	Annotations	Other Elements
Font Size	Available	Available	Not Available	Available	Available
Arrowhead Size	Available	Available	Available	Not Available	Not Available
Junction Size	Not Available	Not Available	Available	Not Available	Not Available
Font Style	Not Available	Not Available	Not Available	Available	Not Available
Shadow	Not Available	Not Available	Not Available	Available	Not Available
Text Alignment	Not Available	Not Available	Not Available	Available	Not Available

Change Global Display Options

Through the **Colors & Fonts** dialog box, you can specify a color scheme for the chart or specify colors and label fonts for different types of objects in a chart. To open the **Colors & Fonts** dialog box, select **File** > **Stateflow Preferences** > **Style**.
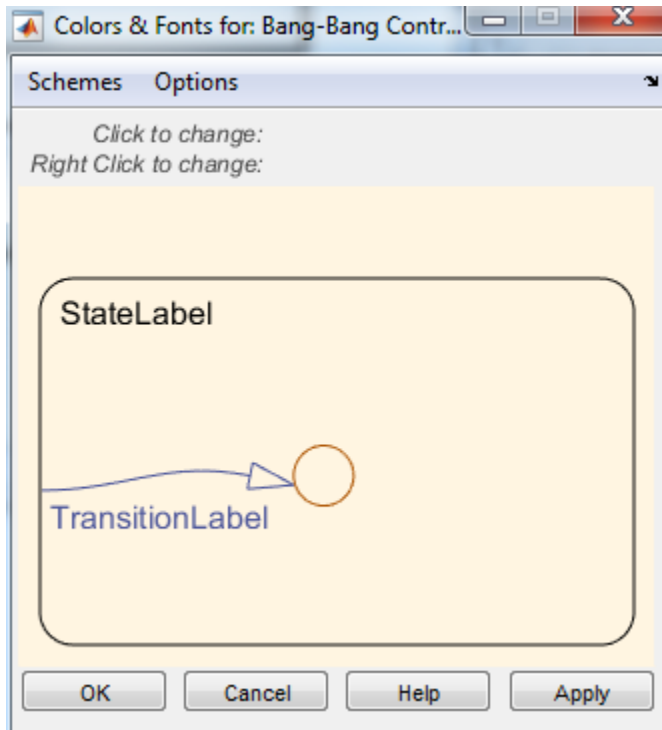
In the **Colors & Fonts** dialog box, the drawing area displays examples of the colors and label fonts specified by the current color scheme for the chart. You can choose a different color scheme from the **Schemes** menu. To modify the display options for a single type of chart element, position your pointer over the sample object.

- To change the color of the element, click the sample object and select a new color in the dialog box.
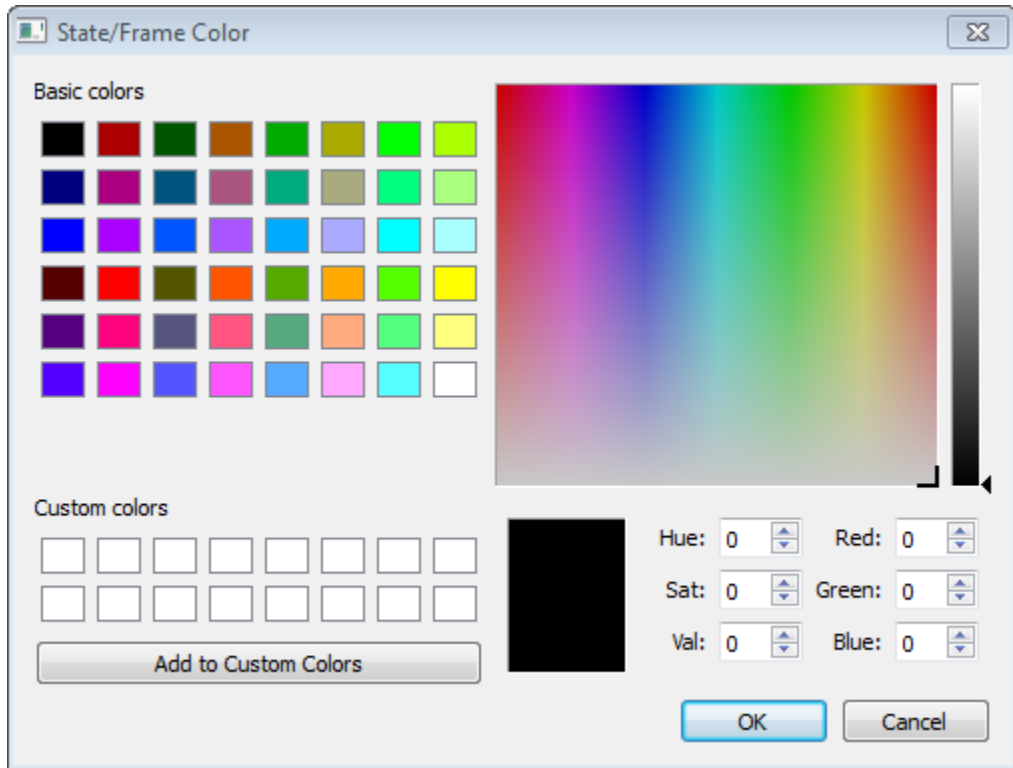
- To change the font of the element, right-click the sample object and select a new font, style, or size in the dialog box.
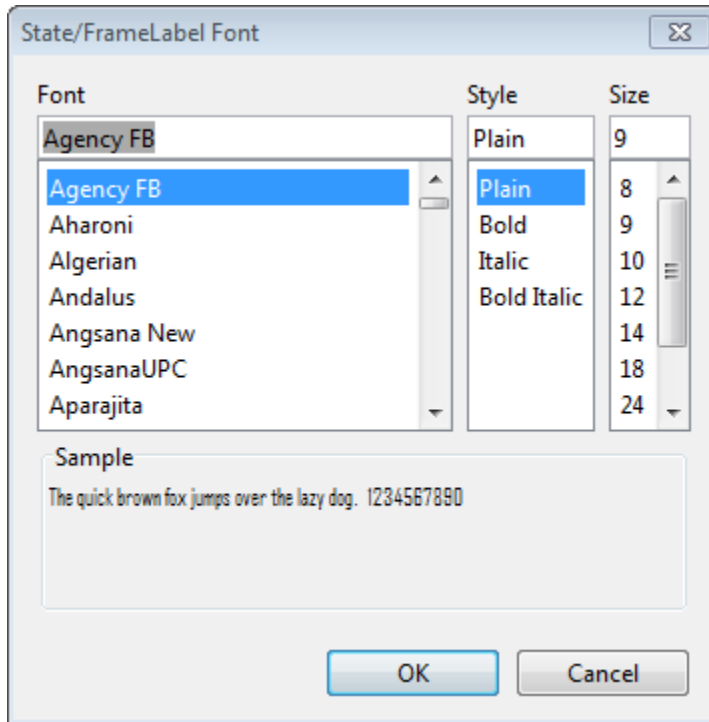
To apply the scheme to the chart, click **Apply**. To apply the scheme and close the dialog box, click **OK**.

To make the scheme the default scheme for all charts, select **Options > Make this the 'Default' scheme**.

To save changes to the default color scheme, select **Options > Save defaults to disk**. If the modified scheme is not the default scheme, choosing **Save defaults to disk** has no effect.

Content Preview for Stateflow Objects

When a chart is closed, you can preview the content of Stateflow charts in Simulink. You can see an outline of the contents of a chart. During simulation, you can see chart animation. When a chart is open, you can preview the content of subcharts and Simulink functions.

For example, the Temporal Logic chart uses content preview. The chart Without Temporal Logic does not.
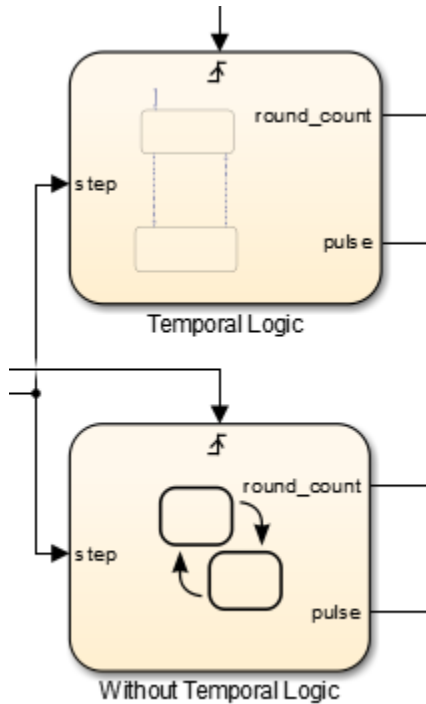
To turn on content preview for Stateflow charts and subcharts, right-click the chart and select **Format > Content Preview**. For Simulink functions, right-click the function and select **Content Preview**. For details on content preview in Simulink, see "Preview Content of Model Components" (Simulink).

Note In order to see the content preview, you may need to enlarge the Stateflow chart or object.

Intelligent Tab Completion for Stateflow Charts

Stateflow tab completion provides context-sensitive editing assistance. Tab completion helps you avoid typographical errors. It also helps you quickly select syntax-appropriate

options for keywords, data, event, messages, and function names, without having to navigate the Model Explorer. In a Stateflow chart, to complete entries:

1. Type the first few characters of the word that you want.
2. Press **Tab** to see the list of possible matches.
3. Use the arrow keys to select a word.
4. Press **Tab** to make the selection.

Additionally, you can:

- Close the list without selecting anything by pressing the **Esc** key.
- Type additional characters onto your original term to narrow the list of possible matches.

If you press **Tab** and no words are listed, then the current word is the only possible match.

Differentiate Elements of Action Language Syntax

You can use color highlighting to differentiate the following syntax elements:

- Keyword
- Comment
- Event
- Message
- Function
- String
- Number

Syntax highlighting is a user preference, not a model preference.

Default Syntax Highlighting

The following chart illustrates the default highlighting for language elements.
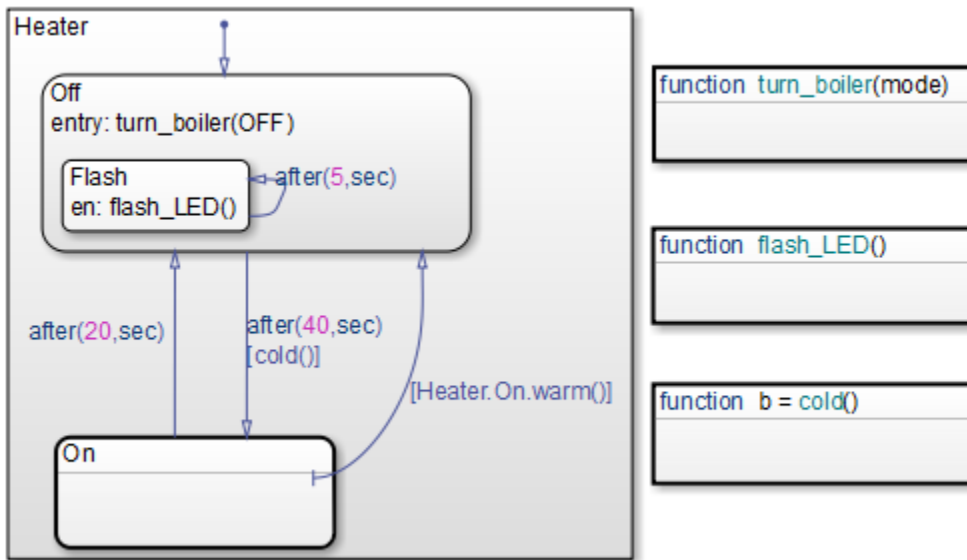
If the parser cannot resolve a syntax element, the chart displays the element in the default text color.

To modify color assignments, see "Edit Syntax Highlighting" on page 4-39. To disable syntax highlighting, see "Enable and Disable Syntax Highlighting" on page 4-39.

Edit Syntax Highlighting

1 In the Stateflow Editor, select **File** > **Stateflow Preferences** > **Syntax Highlighting**.

   The Syntax Highlight Preferences dialog box appears.

2 Click the color that you want to change, choose an alternative from the color palette, and click **Apply**.

3 Click **OK** to close the Syntax Highlight Preferences dialog box.

Enable and Disable Syntax Highlighting

1 In the Stateflow Editor, select **File** > **Stateflow Preferences** > **Syntax Highlighting**.

   The Syntax Highlight Preferences dialog box appears.

2   Select or clear **Enable syntax highlighting** and click **OK**.

Select and Deselect Graphical Objects

Once an object is in the drawing area, to make any changes or additions to that object, select it.

- To select an object, click anywhere inside of the object.
- To select multiple adjacent objects, click and drag a selection box so that the box encompasses or touches the objects that you want to select, and then release the mouse button.

    All objects or portions of objects within the box become selected.

- To select multiple separate objects, simultaneously press the **Shift** key and click an object or box a group of objects.

    This step adds objects to the list of already selected objects unless an object was already selected, in which case, the object is deselected. This type of multiple object selection is useful for selecting objects within a state without selecting the state itself.

- To deselect all selected objects, click in the drawing area, but not on an object.

When an object is selected, it appears highlighted in the color set as the selection color (blue by default; see "Specify Colors and Fonts in a Chart" on page 4-32 for more information).

Cut and Paste Graphical Objects

You can cut objects from the drawing area or cut and then paste them as many times as you like. You can cut and paste objects from one chart to another. The chart retains a selection list of the most recently cut objects. The objects are pasted in the drawing area location closest to the current pointer location.

- To cut an object, right-click the object and select **Cut** from the context menu.
- To paste the most recently cut selection of objects, right-click in the chart and select **Paste** from the context menu.

Copy Graphical Objects

To copy and paste an object in the drawing area, select the objects and right-click and drag them to the desired location in the drawing area. This operation also updates the chart clipboard.

Note If you copy and paste a state in the chart, these rules apply:

- If the original state uses the default ? label, then the new state retains that label.
- If the original state does not use the default ? label, then a unique name is generated for the new state.

Alternatively, to copy from one chart to another, select **Copy** and then **Paste** from the right-click context menu.

Comment Out Objects

To Comment Out a Stateflow object, right-click the selected object and select **Comment Out**. For more information, see "Commenting Stateflow Objects in a Chart" on page 32-64.

Format Chart Objects

To enhance readability of objects in a chart, in the Stateflow Editor you can use commands in the **Chart > Arrange** menu. These commands include options for:

- Alignment
- Distribution
- Resizing

You can align, distribute, or resize these chart objects:

- States
- Functions
- Boxes

- Junctions

Align, Distribute, and Resize Chart Objects

The basic steps to align, distribute, or resize chart objects are similar.

1   If the chart includes parallel states or outgoing transitions from a single source, make sure that the chart uses explicit ordering.

To set explicit ordering, in the Chart properties dialog box, select **User specified state/transition execution order**.

Note If a chart uses implicit ordering to determine execution order of parallel states or evaluation order of outgoing transitions, the order can change after you align, distribute, or resize chart objects. Using explicit ordering prevents this change from occurring. For more information, see "Execution Order for Parallel States" on page 3-86 and "Transition Evaluation Order" on page 3-65.

---

2   Select the chart objects that you want to align, distribute, or resize.

You can select objects in any order, one-by-one or by drawing a box around them.

3   Decide which object to use as the anchor for aligning, distributing, or resizing other chart objects. This object is the reference object.

To set an object as the reference, right-click the object. Brackets appear around the reference object. In the following example, the `Door` and `Motion` states are selected, and the `Door` state is the reference.
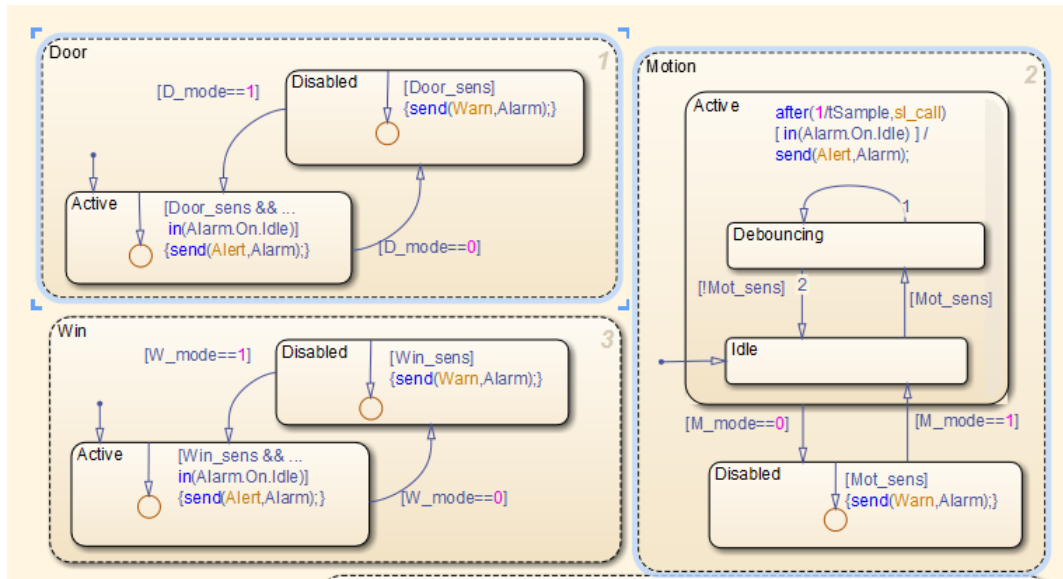
Note If you select objects one-by-one, the last object that you select acts as the reference.

4 Select an option from the **Chart > Arrange** menu to align, distribute, or resize your chosen objects.

For more information about chart object distribution options, see "Options for Distributing Chart Objects" on page 4-44

Options for Distributing Chart Objects

Option	Description
Distribute Horizontally	The center-to-center horizontal distance between any two objects is the same.
	Note The horizontal space for distribution is the distance between the left edge of the leftmost object and the right edge of the rightmost object. If the total width of the objects you select exceeds the horizontal space available, objects can overlap after distribution.
Distribute Vertically	The center-to-center vertical distance between any two objects is the same.
	Note The vertical space for distribution is the distance between the top edge of the highest object and the bottom edge of the lowest object. If the total height of the objects you select exceeds the vertical space available, objects can overlap after distribution.
Even Horizontal Gaps	The horizontal white space between any two objects is the same.
	Note The space restriction for **Distribute Horizontally** applies.
Even Vertical Gaps	The vertical white space between any two objects is the same.
	Note The space restriction for **Distribute Vertically** applies.

Example of Aligning Chart Objects

Suppose that you open the `sf_pool` model and see a chart with multiple MATLAB functions.
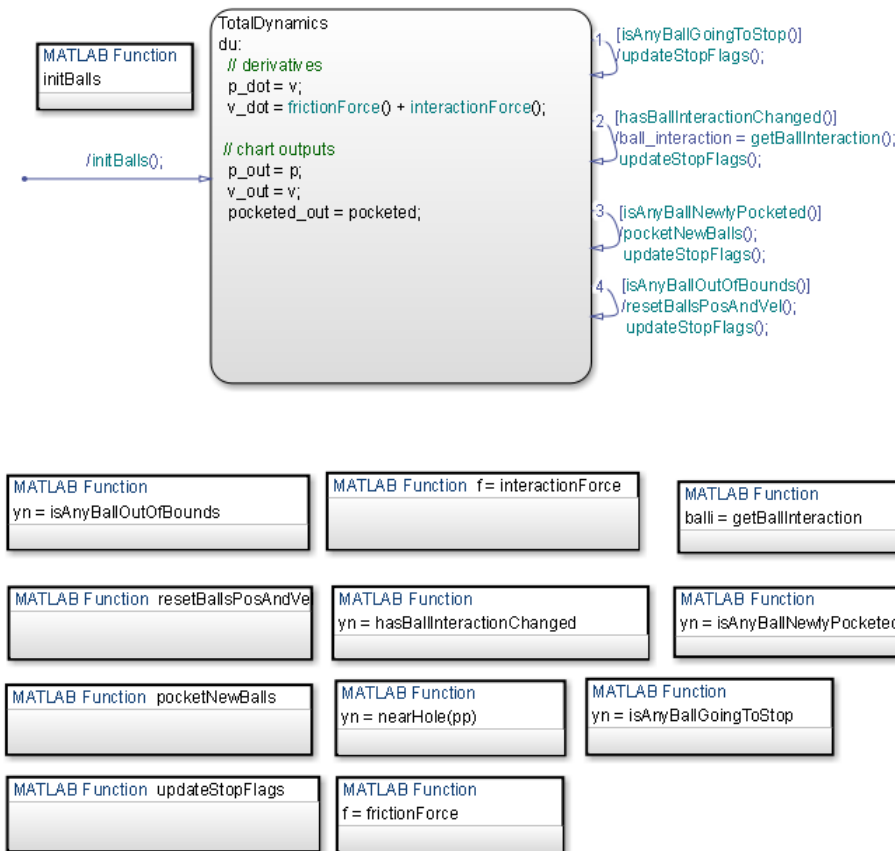
To align the three MATLAB functions on the right:

1 Open the `sf_pool` model. Double-click the Pool block to open the chart.

Tip Expand the Stateflow Editor to see the entire chart.

---

2 Click the function `isAnyBallGoingToStop`.

3 Shift-click the function `isAnyBallNewlyPocketed`.

4 Shift-click the function `getBallInteraction`.

This object is the reference (or anchor) for aligning the three functions. Brackets appear around the function.
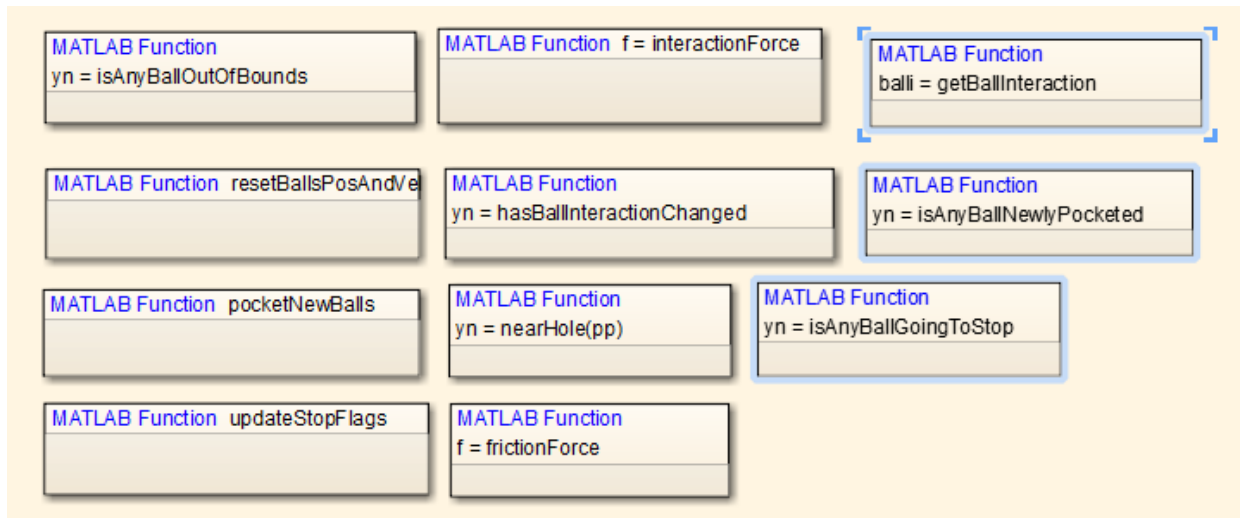
5   Select **Chart** > **Arrange** > **Align Right**.

This step aligns the right edges of the three functions based on the right edge of `getBallInteraction`.
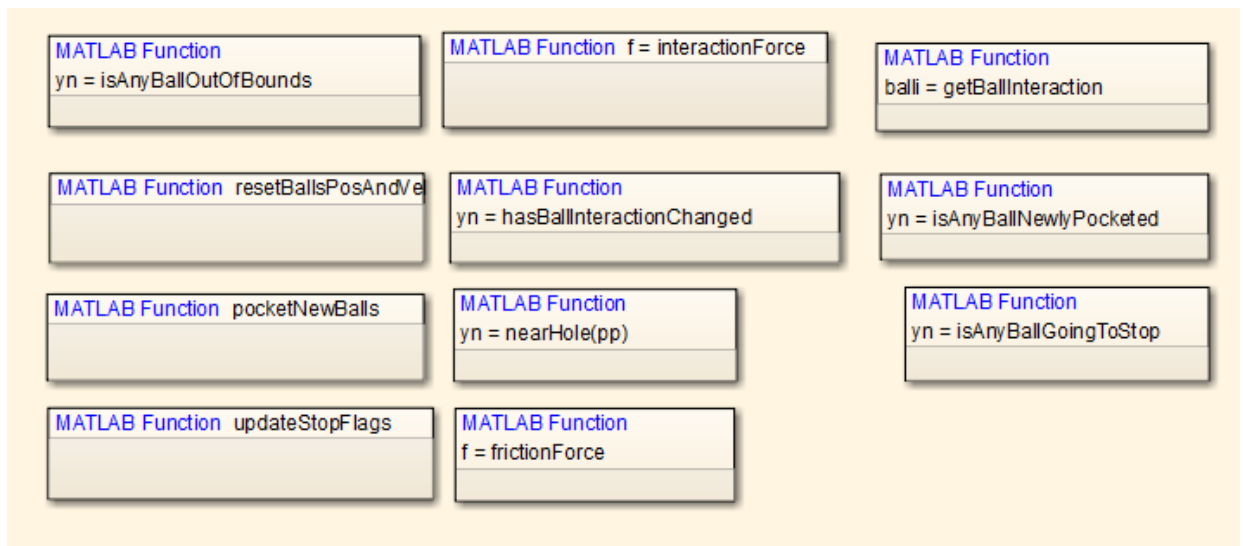
Example of Distributing Chart Objects

Suppose that you open the `sf_frame_sync_controller` model and see a chart with three states.
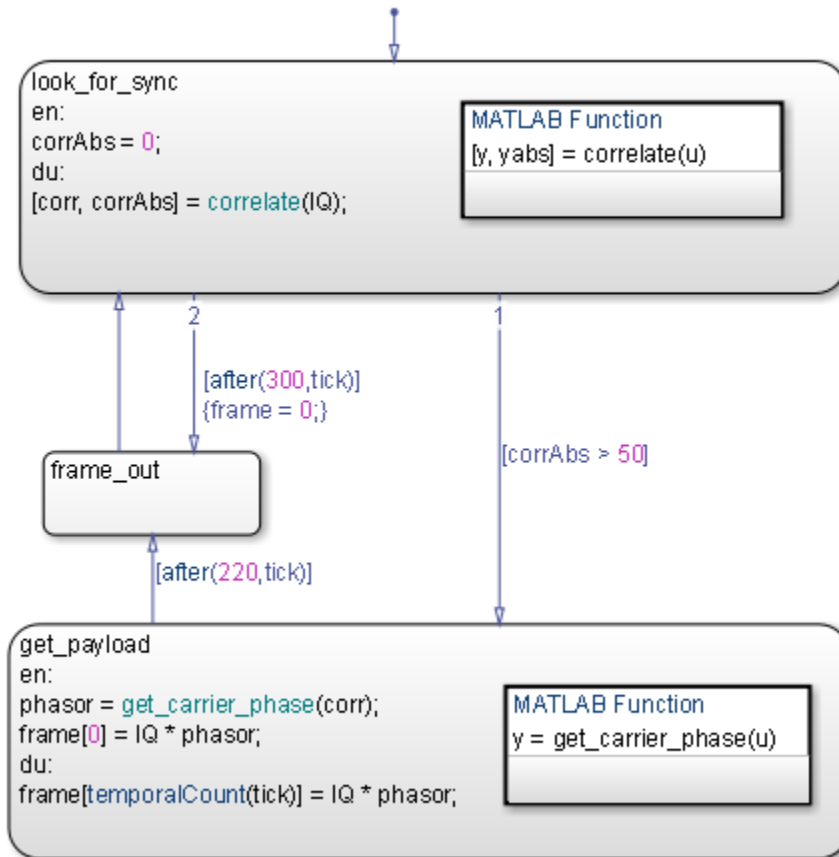
To distribute the three states vertically:

1 Open the `sf_frame_sync_controller` model.

Tip Double-click the Frame Sync Controller block to open the chart.

---

2 Select the three states in any order.

Shift-click to select more than one state.
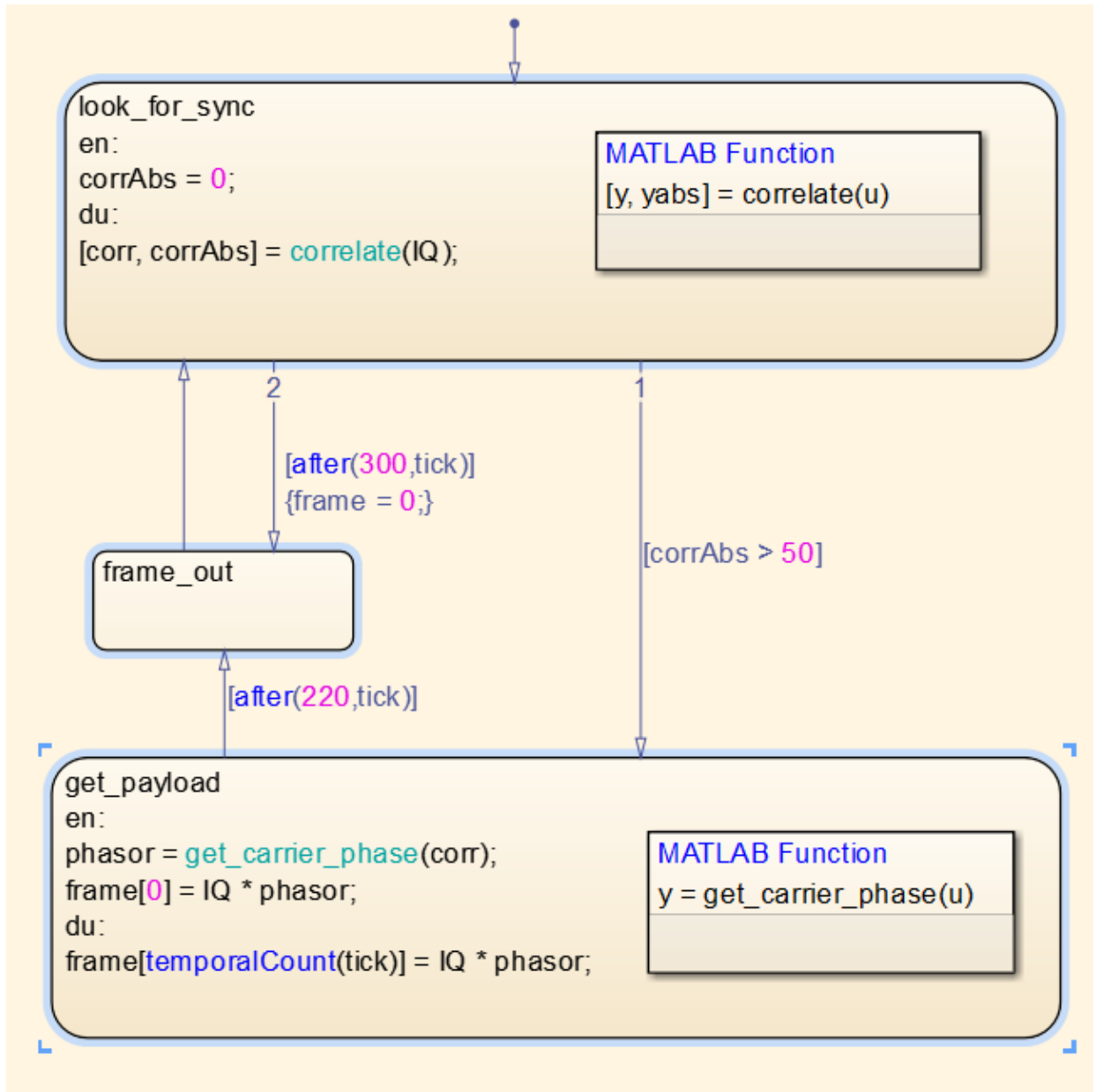
> **Note** When you select the three states in any order, your reference object might differ from the one shown. This difference does not affect distribution of vertical white space.

3 Select **Chart** > **Arrange** > **Even Vertical Gaps**.

This step ensures that the vertical white space between any two states is the same.

Example of Resizing Chart Objects

Suppose that you open the `sf_clutch_enabled_subsystems` model and see a chart with graphical functions of different sizes.
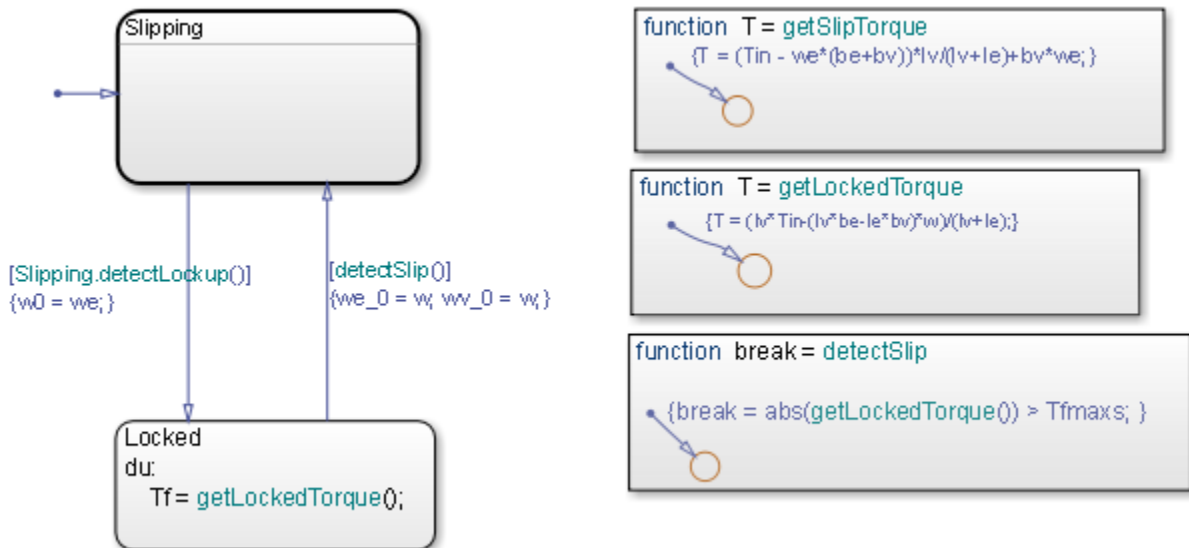
To resize the graphical functions so that they all match the size of `detectSlip`:

1 Open the `sf_clutch_enabled_subsystems` model.

2 In the Friction Mode chart, select the three graphical functions by drawing a box around them.

3 Set `detectSlip` as the reference object to use for resizing.

Right-click the function to mark it with brackets.
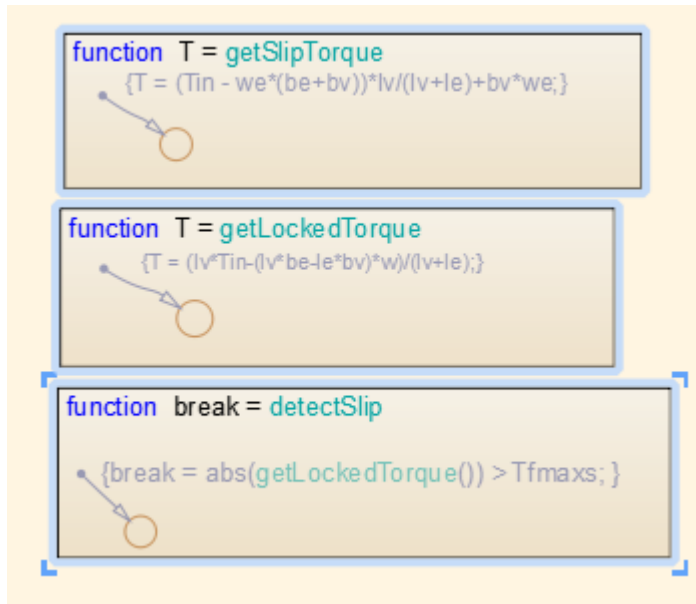
4 Select **Chart** > **Arrange** > **Match Size**.

This step ensures that the three functions are the same size.
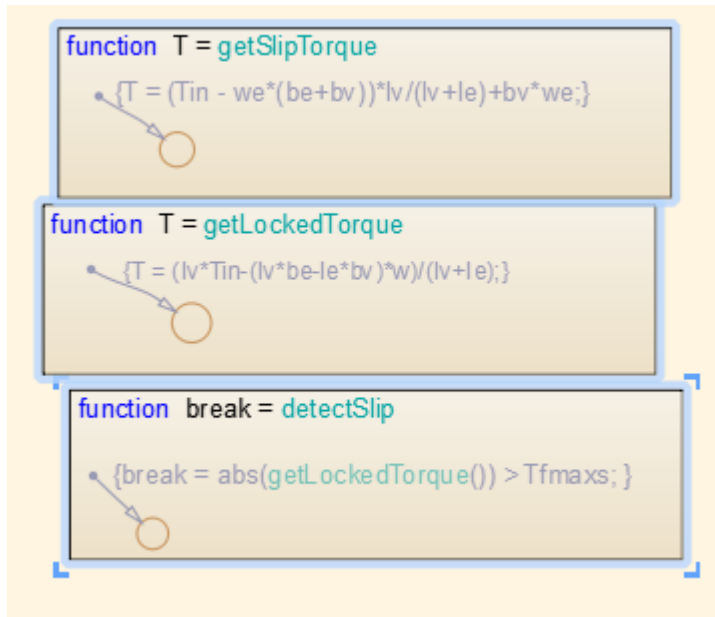
5   Adjust the function boxes to correct the format:

    a   To align the functions, select **Chart > Arrange > Align Left**.
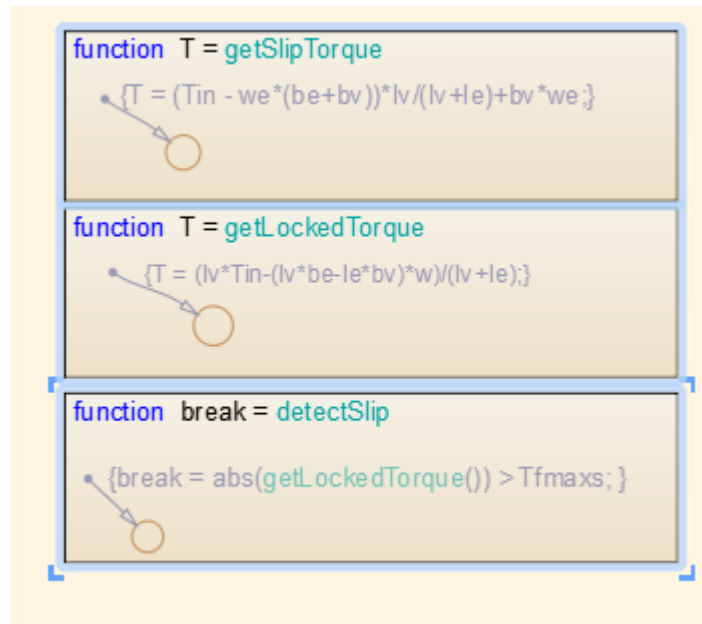
b To distribute the functions evenly in terms of vertical spacing, select **Chart > Arrange > Even Vertical Gaps**.
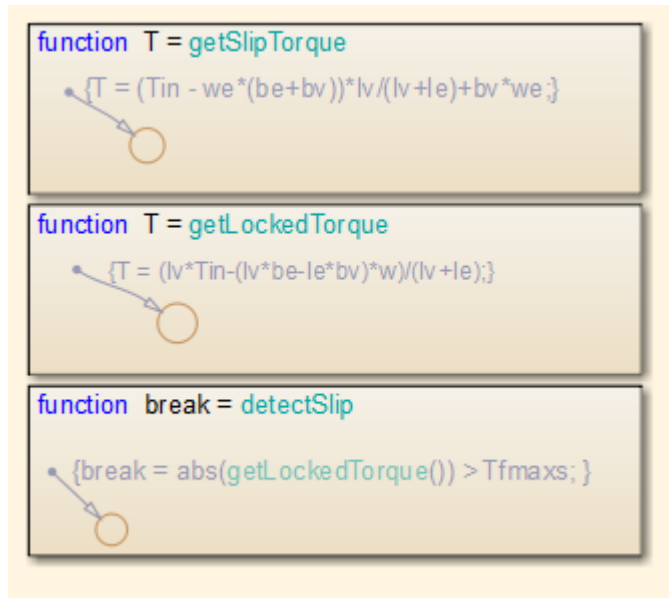
Automatic Chart Formatting

With Arrange Automatically, Stateflow arranges your charts to:

- Expand states and transitions to fit their label strings.
- Resize similar states to be the same size.
- Align states if they were slightly misaligned.
- Straightens transitions.
- Repositions horizontal transition labels to the midpoint.

To format your chart, select **Chart** > **Arrange** > **Arrange Automatically**.

In this example, the chart has:

1 State actions that are outside of the boundary for state A.
2 A transition condition that overlaps state B.
3 A transition that is not horizontal.
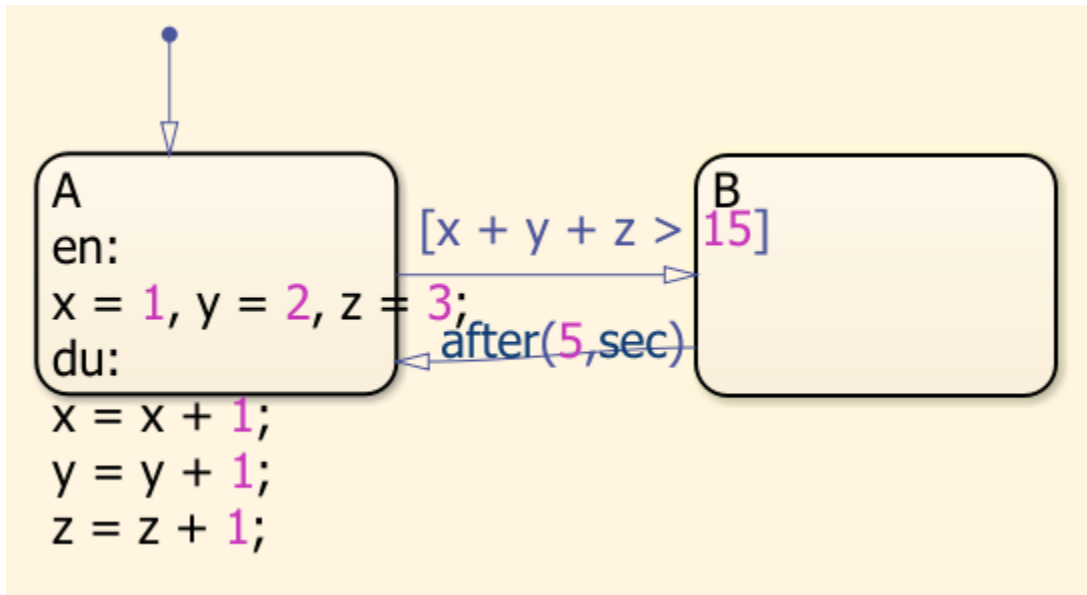
After the layout has been automatically arranged:

1   The state actions are contained within state A.

2   The transition condition does not overlap into state B.

3   The lower transition is horizontal.
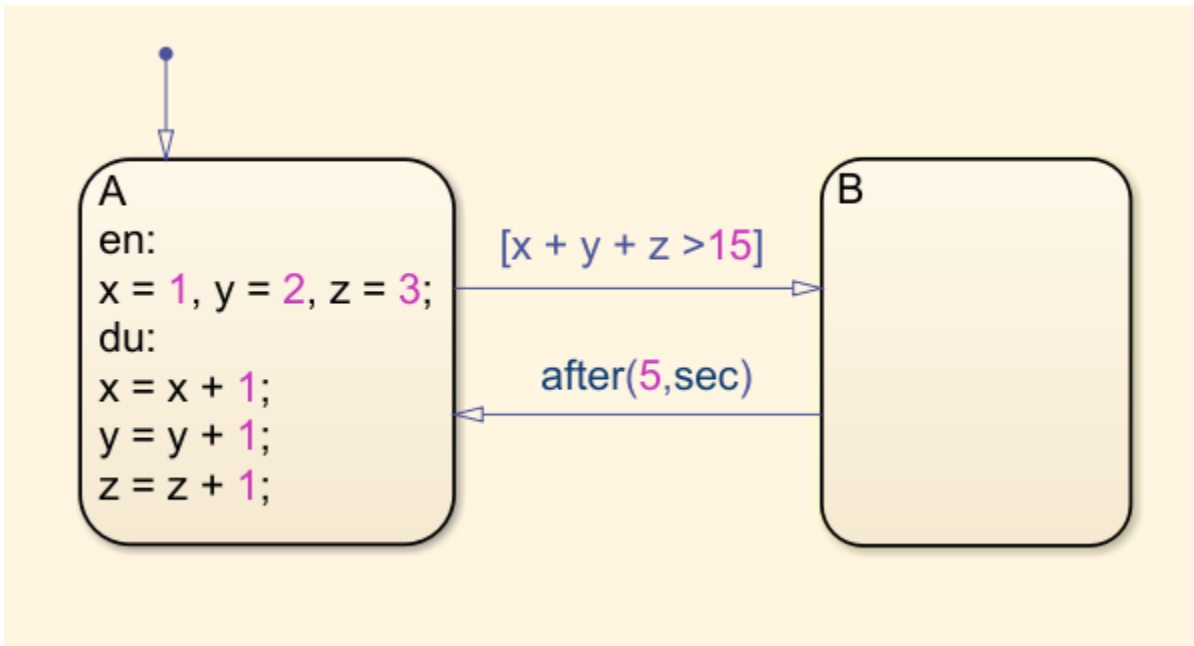
Generate a Model Report

The **Print Details** report is an extension of the **Print Details** report in the Simulink model window. It provides a report of Stateflow and Simulink objects relative to the chart currently in view from which you select the report.

To generate a model report on chart objects:

1   Open the chart or subchart for which you want a report.
2   In the editor, select **File** > **Print** > **Print Details**.

   The Print Details dialog box appears.
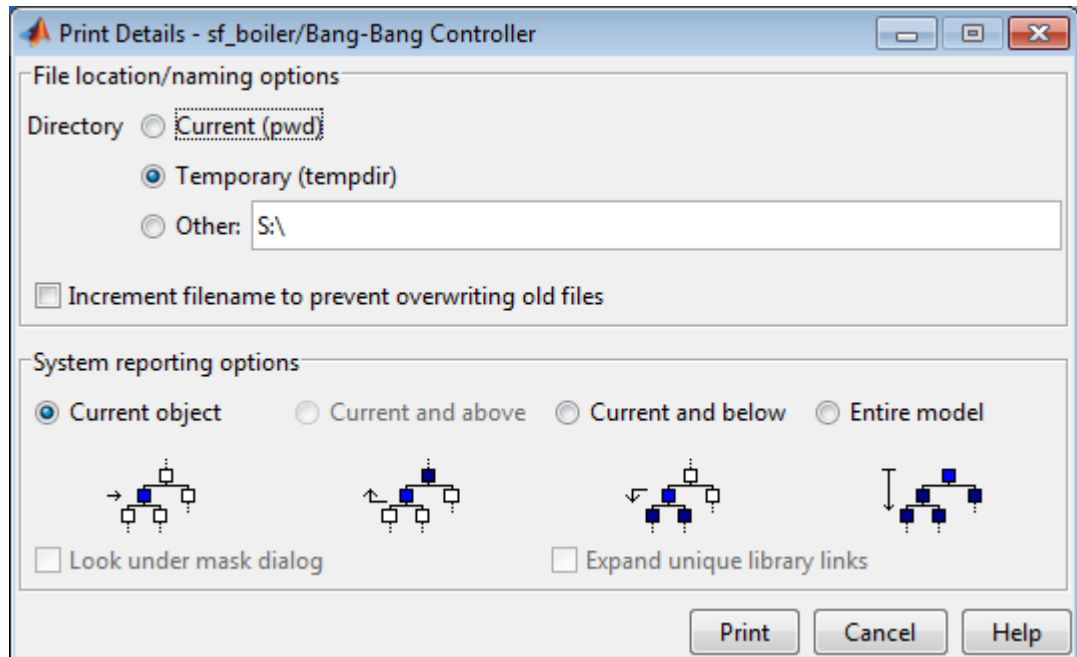
3 Enter the destination directory of the report file and select options to specify what objects appear in the report.

For details on setting the fields in the **File locations/naming options** section, see "Print Model Reports" (Simulink). For details on the report you receive, see "System Report Options" on page 4-58 and "Report Format" on page 4-58.

4 Click **Print**.

The Print Details dialog box appears and tracks the report generation. See "Print Model Reports" (Simulink) for more details on this window.

The HTML report appears in your default browser.

Tip If you have the Simulink Report Generator™ installed, you can generate a detailed report about a system. To do so, in the Simulink Editor, select **File > Reports > System Design Description**. For more information, see "System Design Description" (Simulink Report Generator).

System Report Options

Reports for the current Stateflow chart vary with your choice of one of the **System reporting options** fields:

- **Current** — Reports on the chart or subchart in the current editor window and its immediate parent Simulink system.
- **Current and above** — This option is grayed out and unavailable for printing chart details.
- **Current and below** — Reports on the chart or subchart in the current editor window and all contents at lower levels of the hierarchy, along with the immediate Simulink system.
- **Entire model** — Reports on the entire model including all charts and all Simulink systems.

    If you select this option, you can modify the report as follows:

    - **Look under mask dialog** – Includes the contents of masked subsystems in the report.
    - **Expand unique library links** – Includes the contents of library blocks that are subsystems in the report.

        The report includes a library subsystem only once even if it occurs in more than one place in the model.

Report Format

The general top-down format of the **Print Details** report is as follows:

- The report shows the title of the system in the Simulink model containing the chart or subchart in current view.
- A representation of Simulink hierarchy for the containing system and its subsystems follows. Each subsystem in the hierarchy links to the report of its Stateflow charts.
- The report section for the Stateflow charts of each system or subsystem begins with a small report on the system or subsystem, followed by a report of each contained chart.
- Each chart report includes a reproduction of its chart with links for subcharted states that have reports of their own.
- An appendix tabulates the covered Stateflow and Simulink objects in the report.

5

Model Logic Patterns and Iterative Loops Using Flow Charts

- "Flow Charts in Stateflow" on page 5-2
- "Create Flow Charts by Using Pattern Wizard" on page 5-6

Flow Charts in Stateflow

A Stateflow flow chart is a graphical construct that models logic patterns such as decision trees and iterative loops. Flow charts represent combinatorial logic in which one result does not depend on prior results. You build flow charts by combining connective junctions and transitions without using any states. The junctions provide decision branches between different transition paths. Executing a flow chart begins at a default transition and ends at a terminating junction (a junction that has no valid outgoing transitions).

A best practice is to encapsulate flow charts in graphical functions to create modular and reusable logic that you can call anywhere in a chart. For more information about graphical functions, see "Reuse Logic Patterns by Defining Graphical Functions" on page 8-16.

An example of a flow chart that models simple `if-else` logic:
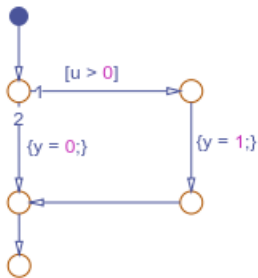
The flow chart models this code:

```
if u > 0
    y = 1;
else
    y = 0;
end
```

Draw a Flow Chart

You can draw and customize flow charts manually by using connective junctions as branch points between alternate transition paths:

1   Open a chart.

2   From the editor toolbar, drag one or more connective junctions into the chart with the **Connective Junction** tool:

3   Add transition paths between junctions.

4   Label the transitions.

5   Add a default transition to the junction where the flow chart execution starts.

Best Practices for Creating Flow Charts

Follow these best practices to create efficient, accurate flow charts:

Use only one default transition

Flows charts have a single entry point.

Provide only one terminating junction

Multiple terminating junctions reduce readability of a flow chart.

Converge all transition paths to the terminating junction

Execution of a flow chart always reaches the termination point.

Provide an unconditional transition from every junction except the terminating junction

If unintended backtracking occurs during simulation, a warning message appears.

You can control the level of diagnostic action for unintended backtracking in the **Diagnostics > Stateflow** pane of the Model Configuration Parameters dialog box. For more information, see the documentation for the "Unexpected backtracking" (Simulink) diagnostic.

Unintended backtracking can occur at a junction under these conditions:

- The junction does not have an unconditional transition path to a state or terminating junction.
- Multiple transition paths lead to that junction.

Use condition actions to process updates, not transition actions

Flow charts test transitions, but do not execute them (and, therefore, never execute transition actions).

An example that illustrates these best practices:
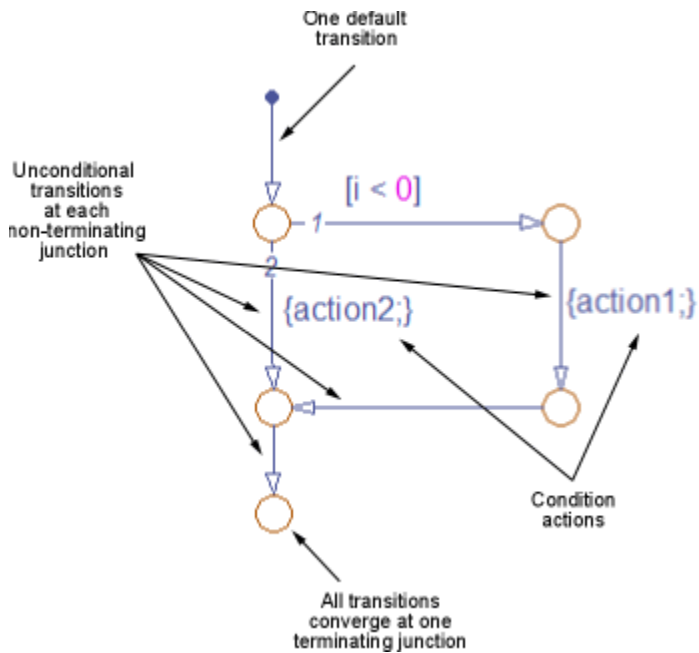
See Also

Related Examples

- "Create Flow Charts by Using Pattern Wizard" on page 5-6
- "How Stateflow Objects Interact During Execution" on page 3-5

More About

- "States" on page 2-7

- "Transitions" on page 2-18

Create Flow Charts by Using Pattern Wizard

The Pattern Wizard is a utility that generates common flow chart patterns for use in graphical functions and charts. The Pattern Wizard offers several advantages over manually creating flow charts. The Pattern Wizard:

- Generates common logic and iterative loop patterns.
- Promotes consistency in geometry and layout across patterns.
- Facilitates storing and reusing patterns from a central location.
- Allows inserting patterns in an existing flow chart.

The Pattern Wizard generates flow charts whose geometry and layout comply with the guidelines from the MathWorks Automotive Advisory Board (MAAB). You can customize your flow chart by modifying the conditions and actions or by inserting additional logic patterns. You can also save your flow chart as a custom pattern in the Pattern Wizard for later reuse.

For example, suppose that you want to use the Pattern Wizard to create a graphical function for iterating over the upper triangle of a two-dimensional matrix. The function consists of two nested `for` loops in which the row index i is always less than or equal to the column index j. By using the Pattern Wizard, you can:

1. Create a flow chart for the outer loop that iterates over the row index i.
2. Extend the flow chart by inserting an inner loop that iterates over the column index j.
3. Save the flow chart as a custom pattern in the Pattern Wizard.
4. Reuse the custom pattern in a graphical function.

Create Reusable Flow Charts
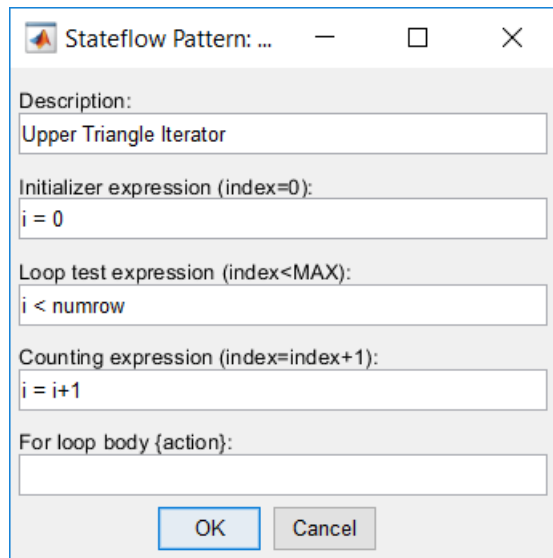
To create a flow chart, select a pattern from the **Chart > Add Pattern in Chart** menu. Pattern selections include:

- **Decision**: `if`, `if-else`, and nested `if` decision patterns.
- **Loop**: `for`, `while`, and `do-while` loop patterns.
- **Switch**: `switch` patterns with up to four cases.
- **Custom**: custom patterns that you saved for later reuse.

The Stateflow Patterns dialog box prompts you for conditions and actions specific to the pattern that you select. For more information on flow chart patterns, see "MAAB-Compliant Patterns from the Pattern Wizard" on page 5-14.

For example, to create the outer `for` loop in the upper triangle iterator pattern:

1 In the Stateflow editor, select **Chart > Add Pattern in Chart > Loop > For**.

2 In the Stateflow Patterns dialog box, specify the initializer, loop test, and counting expressions for iterating through the first dimension of the matrix:

3 Click **OK**. The Pattern Wizard generates this flow chart.
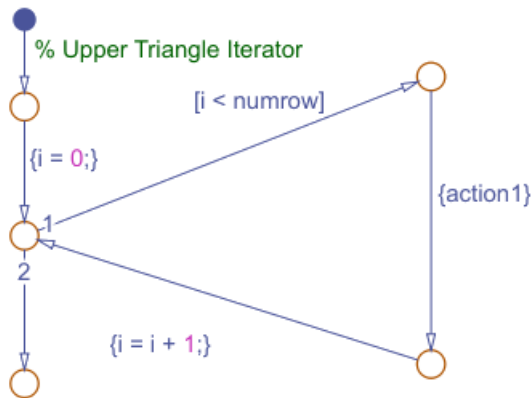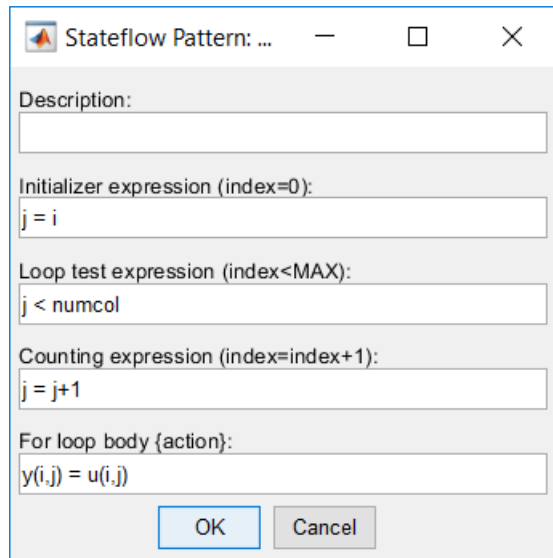
To complete the upper triangle iterator pattern, insert a second `for` loop along the vertical transition in this flow chart.

Insert Logic Patterns in Existing Flow Charts

Use the Pattern Wizard to add loop or decision logic extensions to an existing flow chart. Select an eligible vertical transition and choose a pattern from the **Chart** > **Insert Pattern on Selection** menu. Available patterns include **Decision**, **Loop**, and **Switch**. The Stateflow Patterns dialog box prompts you for conditions and actions specific to the pattern that you select.

For example, to add the second loop in the upper triangle iterator pattern:

1 In the Stateflow editor, from the outer `for` loop pattern, select the vertical transition labeled `{action1}`.

2 Select **Chart** > **Insert Pattern on Selection** > **Loop** > **For**.

3 In the Stateflow Patterns dialog box, specify the initializer, loop test, and counting expressions for iterating through the second dimension of the matrix. The initializer expression ensures that *i* never exceeds *j*. Also enter an action that retrieves each element in the upper triangle of the matrix.

4 Click **OK**. The Pattern Wizard adds the second loop to the flow chart.
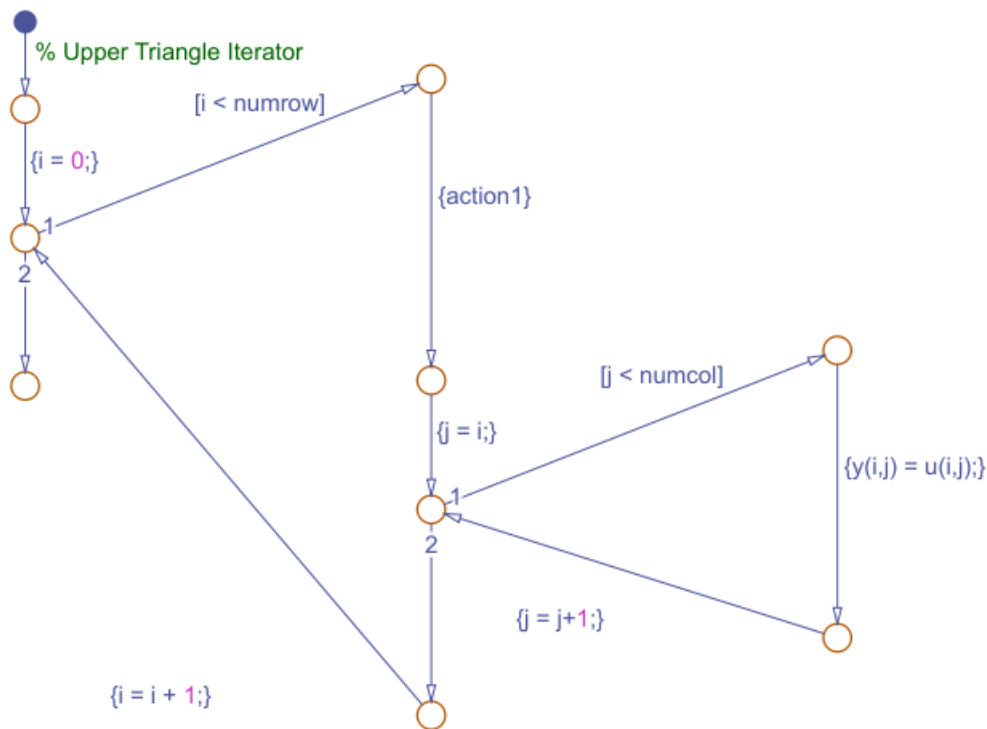
5 Save the model containing the pattern.

Guidelines for Inserting Logic Patterns

When you create logic extensions:

- You can select only one transition to extend at a time. The selected transition must be exactly vertical and have a destination junction.

- You can extend only flow charts created by the Pattern Wizard.

- The Stateflow chart containing the flow chart can contain only junctions and transitions. The chart cannot contain other objects, such as states, functions, or truth tables.

- You cannot extend a pattern that has been custom-created or modified.

- You cannot choose a custom pattern as the extension.

If your selection is not eligible for insertion, when you select **Chart > Insert Pattern on Selection**, you see a message instead of pattern options.

Message	Issue
Select a vertical transition	You have not selected a vertical transition.
Selected transition must be exactly vertical	You selected a transition, but it is not vertical.
Select only one vertical transition	You have selected more than one transition.
Editor must contain only transitions and junctions	There are other objects, such as states, functions, or truth tables, in the chart.

Save Custom Flow Chart Patterns

Use the Pattern Wizard to save flow chart patterns in a central location, and then easily retrieve them for reuse in graphical functions and charts. Select the flow chart with the pattern that you want to save and select **Chart > Save Pattern**.

For example, suppose that you want to save the upper triangle iterator pattern for later reuse:

1   Create a folder for storing your custom patterns. See "Guidelines for Creating a Custom Pattern Folder" on page 5-12.
2   In the Stateflow editor, select the upper triangle iterator flow chart.
3   Select **Chart > Save Pattern**.
4   If you have not designated the custom pattern folder, the Pattern Wizard prompts you to select a folder. Choose the folder that you created and click **Select Folder**.
5   At the prompt, name your pattern `UpperTriangleIterator` and click **Save**. The Pattern Wizard saves your pattern as a model file `UpperTriangleIterator.slx` in the custom pattern folder.

Note You can use the Pattern Wizard to reuse only flow charts. To reuse states and subcharts, create an atomic subchart. For more information, see "Create Reusable Subcomponents by Using Atomic Subcharts" on page 15-2.

Guidelines for Creating a Custom Pattern Folder

The Pattern Wizard uses a single, flat folder for saving and retrieving flow chart patterns.

- Store all flow charts at the top level of the custom pattern folder. Do not create subfolders.
- Make sure that all flow chart files have an `.mdl` or `.slx` extension.

Change Your Custom Pattern Folder

The Pattern Wizard remembers your choice of custom pattern folder for future sessions. To choose a different folder, rename your existing custom pattern folder and do one of the following:

- Save a new custom pattern to the Pattern Wizard.
- Reuse an existing custom pattern from the Pattern Wizard.

The Pattern Wizard prompts you to choose a new folder.

Reuse Custom Flow Chart Patterns

The Pattern Wizard stores your flow charts as model files in the custom pattern folder. The patterns that you save in this folder appear in a drop-down list when you select **Chart > Add Pattern in Chart > Custom**. You can add a custom pattern directly to a chart or to a subcharted graphical function in your chart.

For example, to add the upper triangle iterator custom pattern to a graphical function:

1 From the object palette, add a graphical function to your chart as described in "Define a Graphical Function" on page 8-16.

2 Enter this function signature:
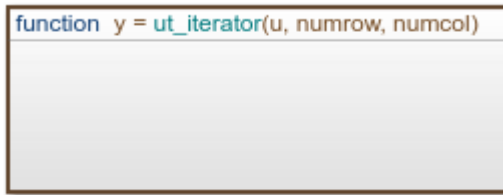
```
function y = ut_iterator(u, numrow, numcol)
```

The function takes three inputs.

Input	Description
u	2-D matrix
numrow	Number of rows in the matrix

Input	Description
numcol	Number of columns in the matrix

3 Right-click inside the function and select **Group & Subchart** > **Subchart**. The function appears as an opaque box.

4 Double-click the subcharted graphical function to open it.

5 Select **Chart** > **Add Pattern in Function** > **Custom**. A dialog box opens, listing all the patterns that you have saved in your custom pattern folder.
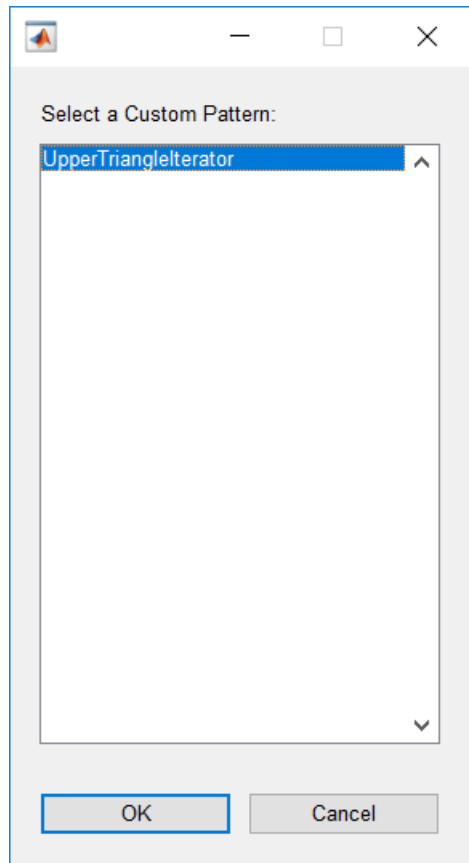
6. Select the upper triangle iterator pattern and click **OK**. The Pattern Wizard adds your custom pattern to the graphical function.

7. In the graphical function, in place of `action1`, substitute an appropriate action. This action repeats once for every row of the matrix.

MAAB-Compliant Patterns from the Pattern Wizard

The Pattern Wizard generates flow charts whose geometry and layout comply with the guidelines from the MathWorks Automotive Advisory Board (MAAB).

Decision Logic Patterns

`if`
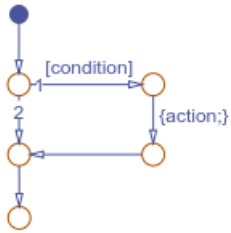
`if-else`
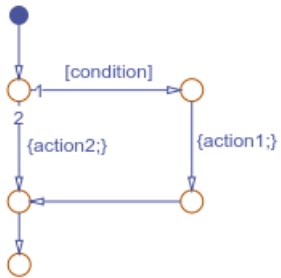
if-elseif
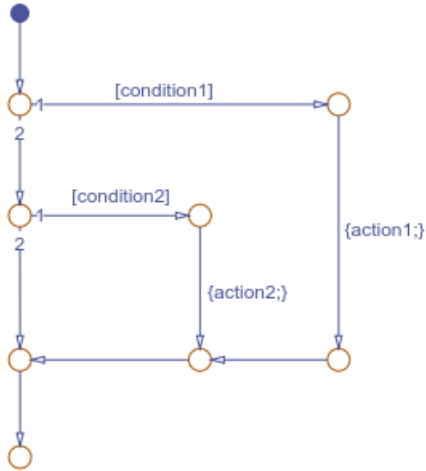
if-elseif-else
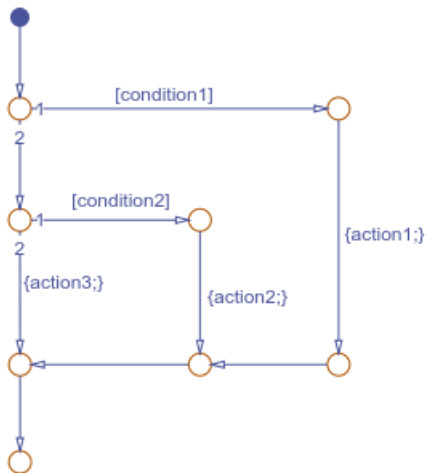
if-elseif-elseif-else
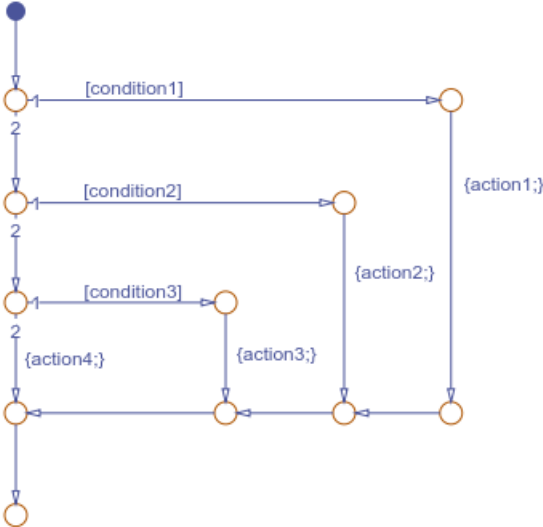
Nested if
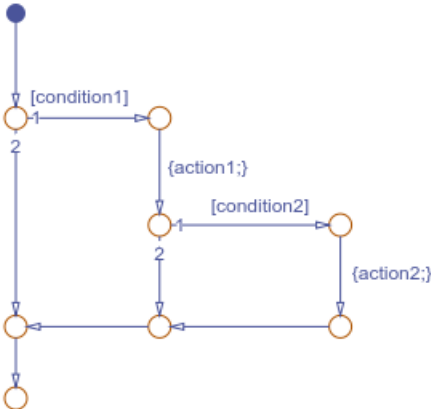
Iterative Loop Patterns

for

while

do-while

{action;}

[condition]
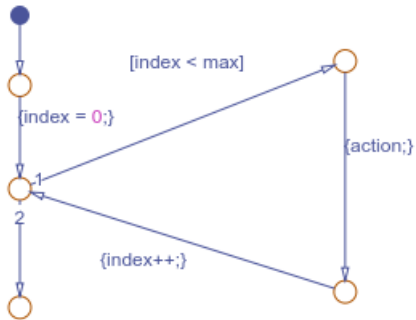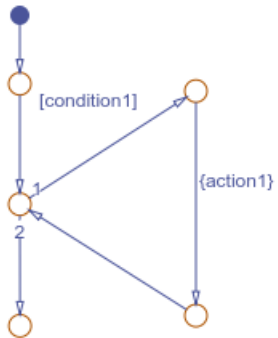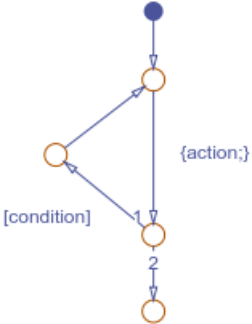
Switch Patterns

switch with Two Cases and Default

[exp == const1]

{action1;}

[exp == const2]

{action2;}

{action 3;}
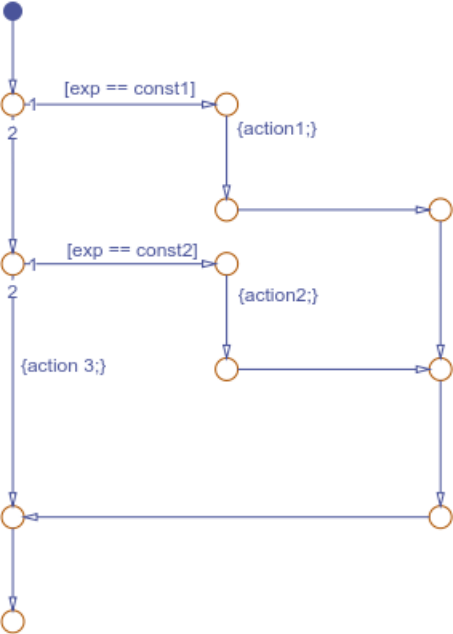
switch with Three Cases and Default
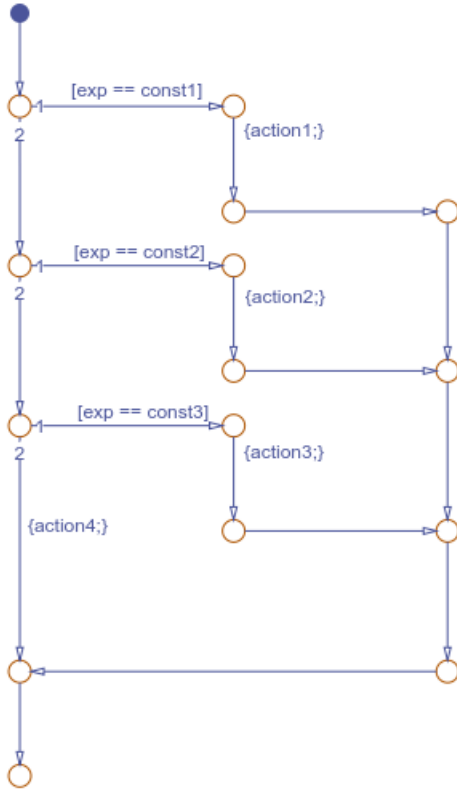
`switch` with Four Cases and Default
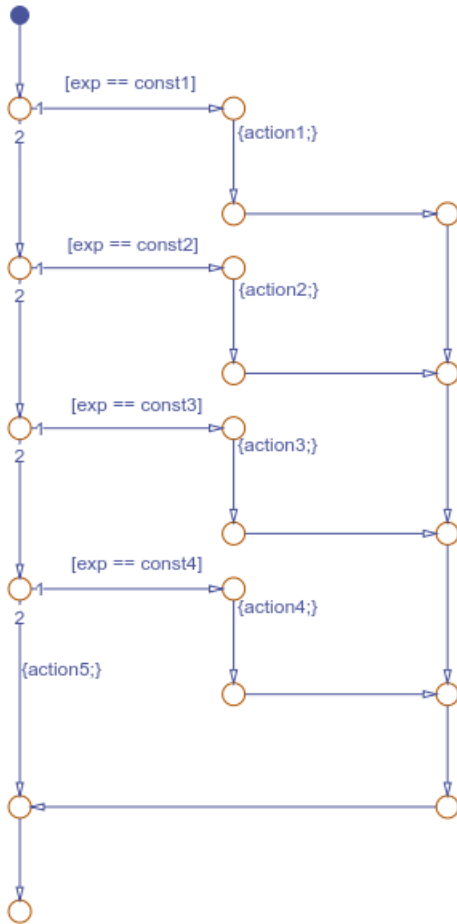
See Also

More About

- "Flow Charts in Stateflow" on page 5-2

- "Reuse Logic Patterns by Defining Graphical Functions" on page 8-16

Simulink Subsystems as Stateflow States

Simulink Subsystems as States

By using a Simulink subsystem within a Stateflow state, you can model hybrid dynamic systems or systems that switch between periodic or continuous time dynamics. In your Stateflow chart, you can use Simulink based states to model a periodic or continuous dynamic system combined with switching logic that uses transitions. You can access inputs and outputs from your chart within each Simulink based state. Simulink based states are supported only in Stateflow charts in a Simulink model.

To initialize Simulink blocks when switching between Simulink based states, use Stateflow textual notation or Simulink State Reader and State Writer blocks.

To create linked Simulink based states, use libraries to save action subsystems. When you copy an action subsystem from a library model into a Stateflow chart, it appears as a linked Simulink based state. When you update the library block, the changes are reflected in all Stateflow charts containing the block.

Using Simulink based states means that you do not have to use complex textual syntax in Stateflow to model hybrid systems.

When to Use Simulink Based States

Use Simulink based states when:

- You want to model hybrid dynamic systems that include continuous or periodic dynamics.
- The structure of the system dynamics change substantially between the various modes of operation, for example, modeling PID controllers.

For systems where you call logic intermittently, use Simulink functions.

When the structure of the Simulink algorithm remains substantially unchanged, but certain gains or parameters switch between various models, use Simulink logic outside of Stateflow. An example of this type of algorithm is gain scheduling. See "Model Gain-Scheduled Control Systems in Simulink" (Simulink Control Design).
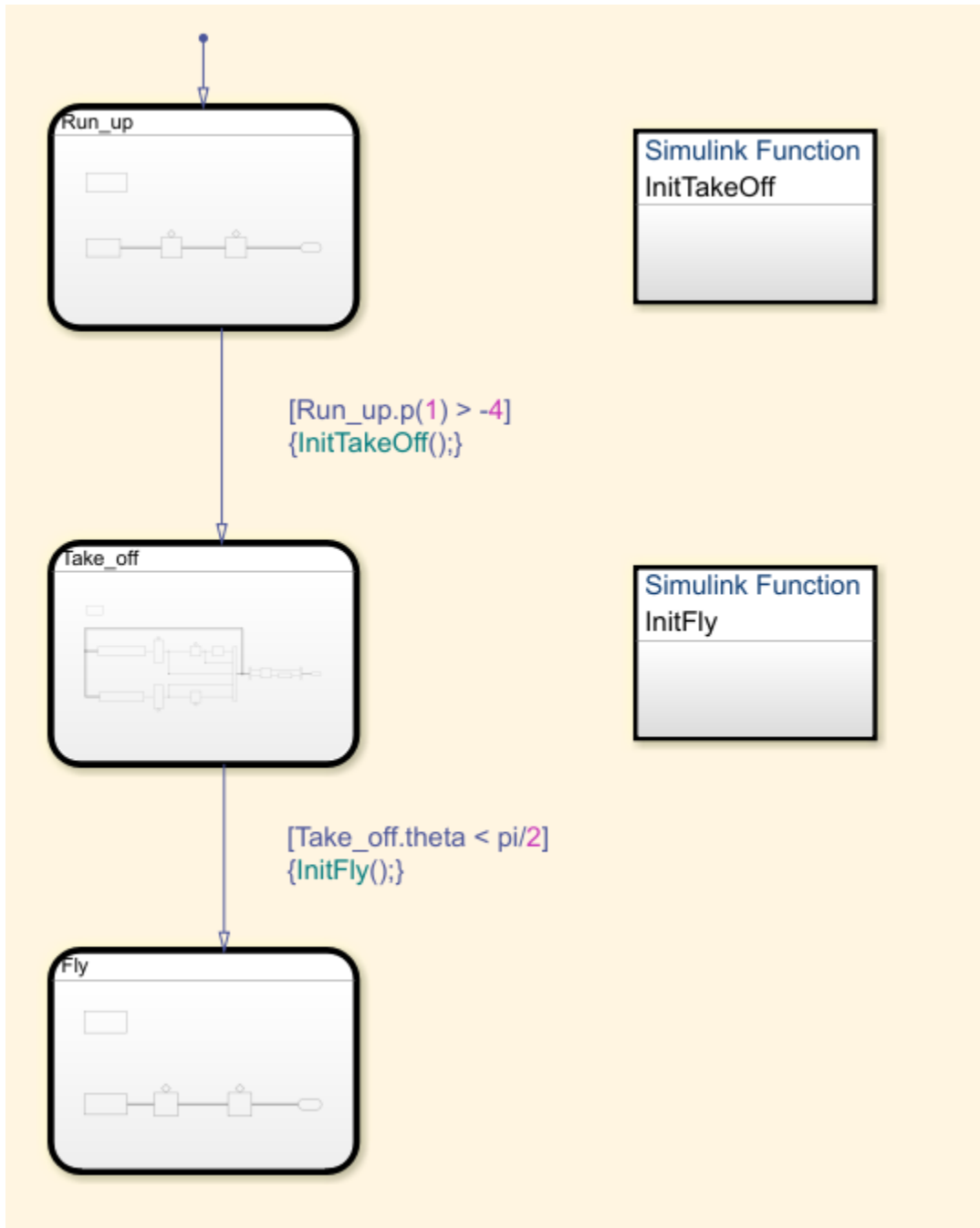
Model a Pole Vaulter by Using Simulink Based States

This Stateflow chart models a person moving through the stages of pole vaulting by using Simulink based states. The first stage is the approach run of the vaulter, which is modeled

in the Simulink based state Run_up. In the second stage, the vaulter plants the pole and takes off, which is modeled by the Simulink based state Take_off. The final stage happens when the vaulter clears the bar and releases the pole, which is modeled by the Simulink based state Fly.

PoleVaulter
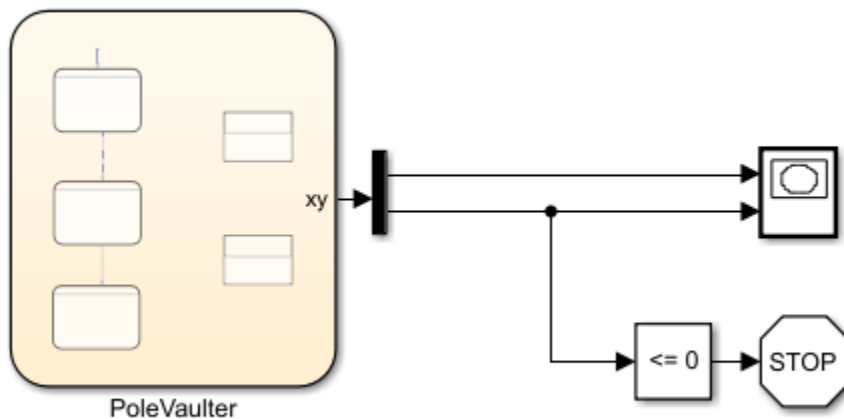
The Stateflow chart contains this logic:

The states `Run_up` and `Fly` are easier to model by using Cartesian coordinates. The state `Take_off` is easier to model by using polar coordinates. To switch from one coordinate system to another, use Simulink functions `InitTakeOff` and `InitFly`.

Model the Approach of the Pole Vaulter

The Simulink based state `Run_up` contains this logic:
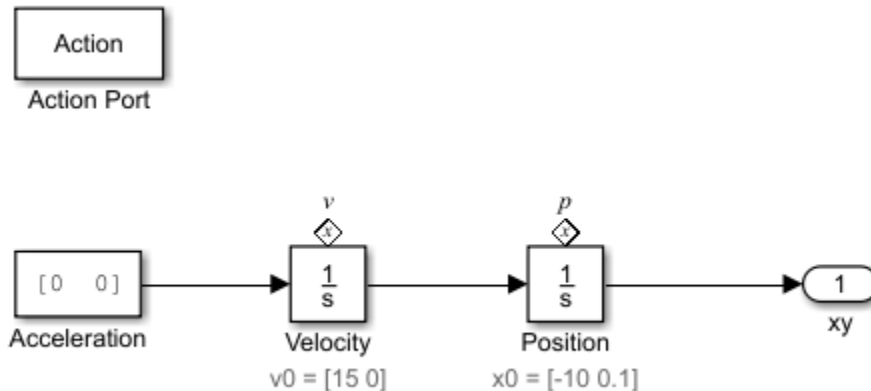
The default state in the chart `PoleVaulter` is `Run_up`. This state models the pole vaulter traveling along the ground toward the jump. The pole vaulter starts at -10 on the x-axis and runs toward zero. As the pole vaulter moves along the ground, the position of the pole vaulter in the xy-plane is continuously changing, but the state of running remains the same. In this model, the integrator blocks `Position` and `Velocity` are state owner blocks for State Reader blocks in the Simulink function `InitTakeOff`. This subsystem outputs the Cartesian coordinates of the pole vaulter.

Convert Cartesian Coordinates to Polar Coordinates

The Simulink function `InitTakeOff` contains this logic:
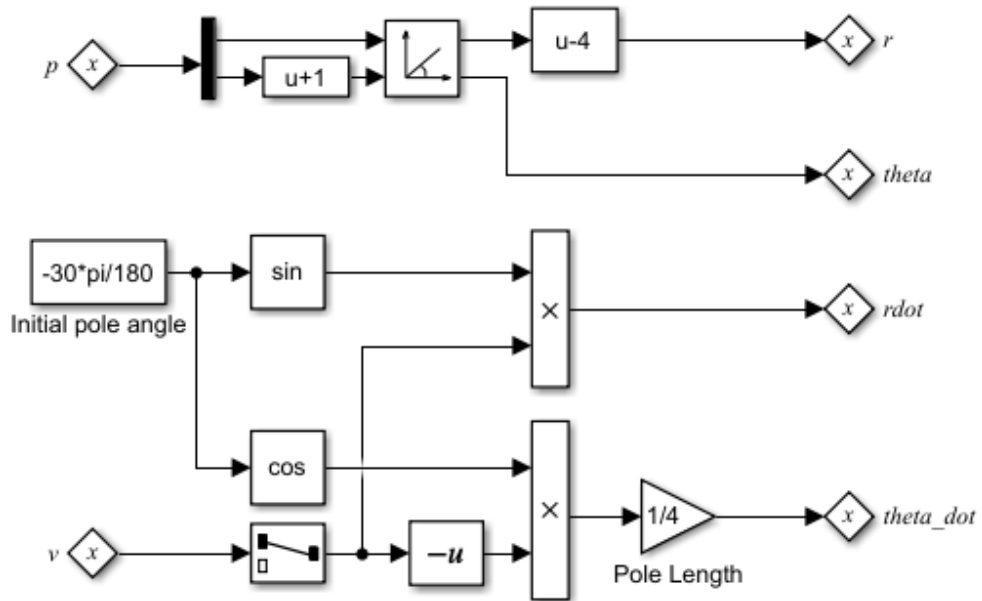
Once the position of the pole vaulter along the x-axis, `Run_up.p(1)`, becomes greater than `-4`, the transition from `Run_up` to `Take_off` occurs. During the transition `InitTakeOff` is initialized, the State Reader block connects to its owner block, and the function is executed. This function converts the Cartesian coordinates from `Position` and `Velocity` to polar coordinates, `r`, `theta`, `rdot`, and `theta_dot`. These coordinates are output as State Writer blocks, which are connected to owner blocks in the state `Take_off`.

Model the Take Off of the Pole Vaulter

The Simulink based state `Take_off` contains this logic:
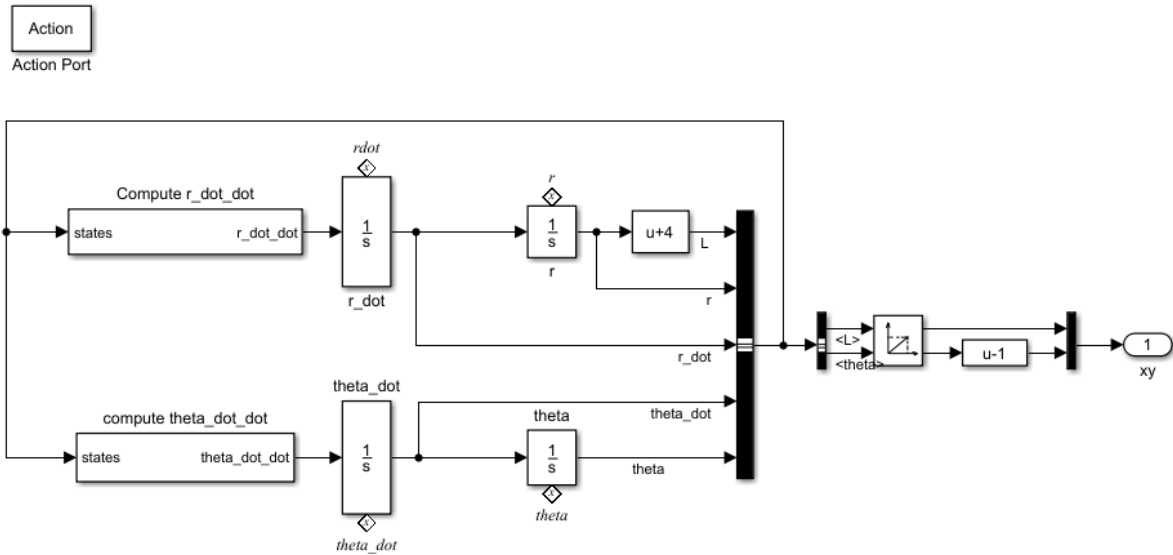
Once the position of the pole vaulter along the x-axis, Run_up.p(1), becomes greater than -4, the active state becomes Take_off. This Simulink subsystem models the pole vaulter during the take off phase of the jump. The subsystem outputs the Cartesian coordinates of the pole vaulter.

Convert Polar Coordinates to Cartesian Coordinates

The Simulink function InitFly contains this logic:
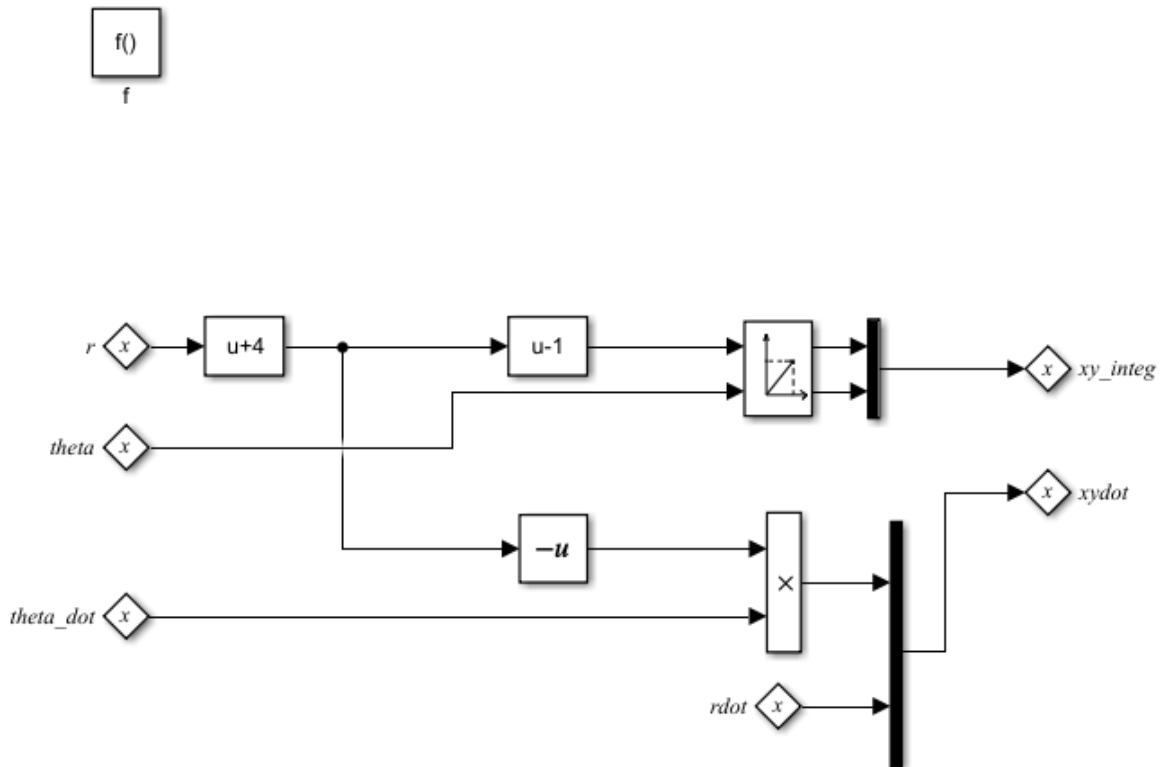
Once the angle of the pole vaulter, theta, becomes less than pi/2, the transition from Take_off to Fly occurs. During the transition InitFly is initialized, the State Reader block connects to its owner block, and the function is executed. This function converts the polar coordinates from r, theta, and theta_dot to Cartesian coordinates, xy_integ and xydot. These coordinates are output as State Writer blocks, which are connected to owner blocks in the state Fly.

Model the Free Fall of the Pole Vaulter

The Simulink based state Fly contains this logic:
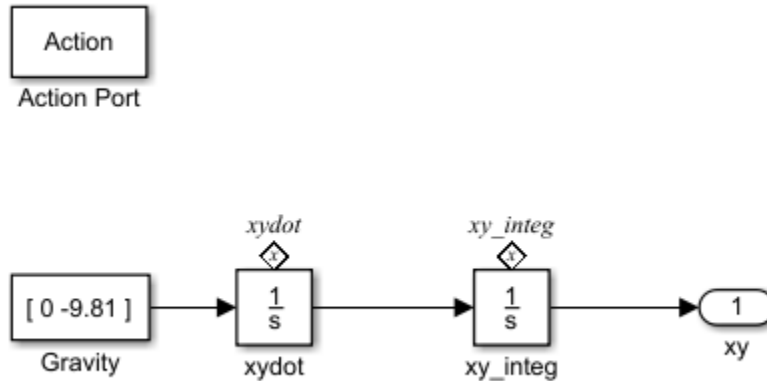
Once the angle of the pole vaulter, `theta`, is less than `pi/2`, the active state becomes `Fly`. This state models the pole vaulter after the jump has cleared and the pole vaulter is falling to the ground. As the pole vaulter falls, the position of the pole vaulter in the x-y plane is continuously changing, but the state of falling remains the same. In this model, the integrator blocks `xydot` and `xy_integ` are state owner blocks for State Writer blocks in the Simulink function `InitFly`. This subsystem outputs the Cartesian coordinates of the pole vaulter.

The results of this simulation are seen in the XY Graph.
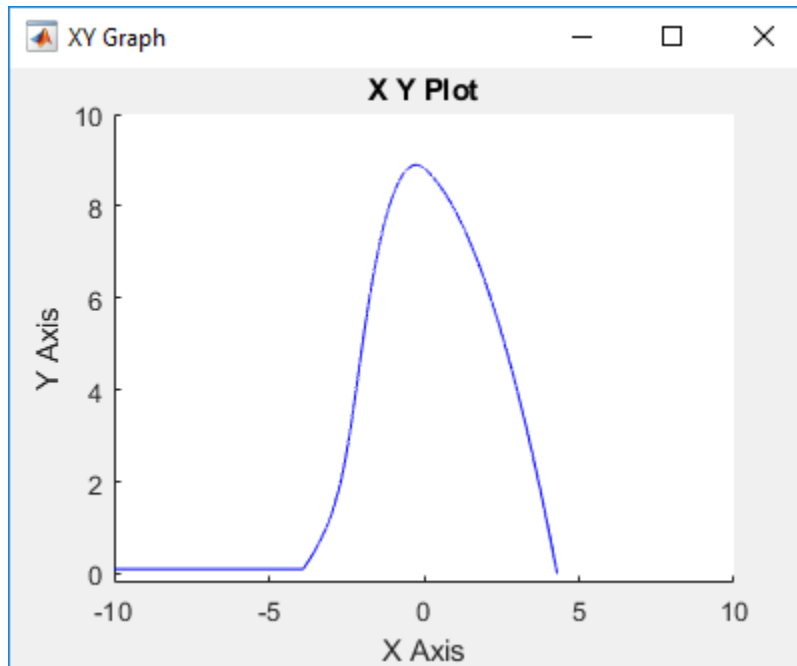
Limitations

You cannot use Simulink based states with:

- Moore charts
- Discrete Event charts
- HDL Coder
- PLC Coder
- Simulink Code Inspector
- Super step transitions

Simulink based states do not support debugging.

See Also

More About

- "Create and Edit Simulink Based States" on page 6-12
- "Reuse Charts in Models with Chart Libraries" on page 24-17
- "Custom Libraries and Linked Blocks" (Simulink)

Create and Edit Simulink Based States

To model systems that switch between periodic or continuous time dynamics, use Simulink based states. Simulink based states are supported only in Stateflow charts in a Simulink model. For more information, see "Simulink Subsystems as States" on page 6-2.

You can create Simulink based state by using the drawing tool. To reuse systems from separate Simulink models, copy and paste enabled subsystems. To reuse subsystems in multiple Stateflow charts, copy and paste action subsystems that are saved in a library.

Create a Simulink Based State

To create a Simulink based state, do one of the following:

- Create an empty Simulink based state by using the Simulink based state drawing tool.
- Create a Simulink based state from another model by copying an enabled subsystem or an action subsystem to your Stateflow chart.
- Create a linked Simulink based state by copying an action subsystem from a library to your Stateflow chart.

Create an Empty Simulink Based State

1 Add a Stateflow chart block to a Simulink model. To open the Stateflow editor, double-click the block.

2 On the object palette, click the Simulink based state drawing tool ⬚. Move your cursor onto your chart.

3 To place the new Simulink based state, click the Stateflow canvas. A shaded state appears.
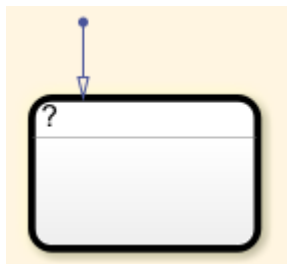
4 Enter the state label. In this example, the state models a pole vaulter running along a flat surface, so the state label is Run_up. Simulink based states are action subsystems, so an Action Port appears with your new state.
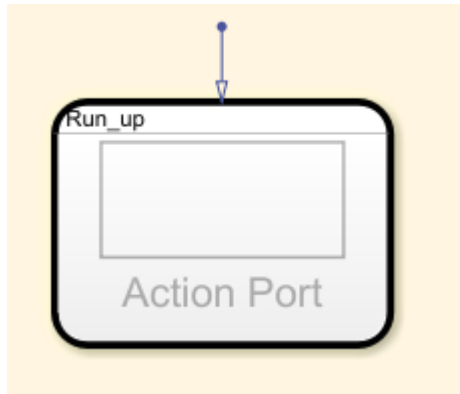
5 Build your Simulink subsystem. This subsystem outputs the Cartesian coordinates of the pole vaulter. For more information about this model, see "Access Block State Data" on page 6-19.
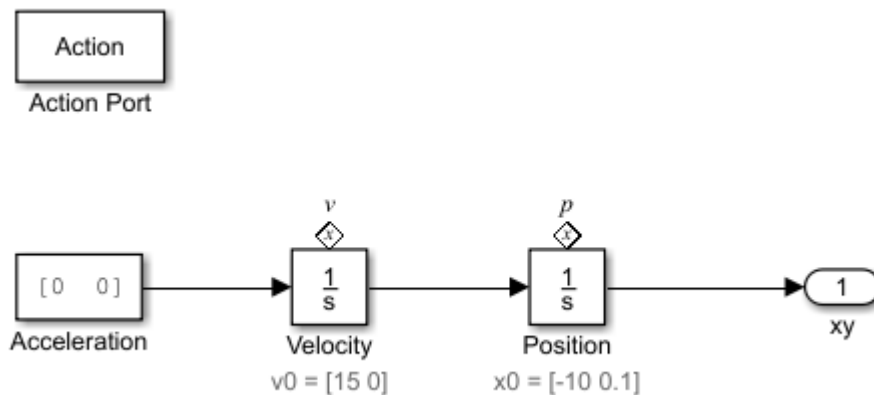
Create a Simulink Based State from an Enabled Subsystem

To create a Simulink based state in your Stateflow chart, copy enabled subsystems from separate Simulink models. You can reuse components from Simulink models in a Stateflow chart without creating a brand new Simulink based state.

1   Open the `sf_clutch_enabled_subsystems` model.

**Building a Clutch Lock-Up Model
Using Stateflow(R) and Simulink(R)**
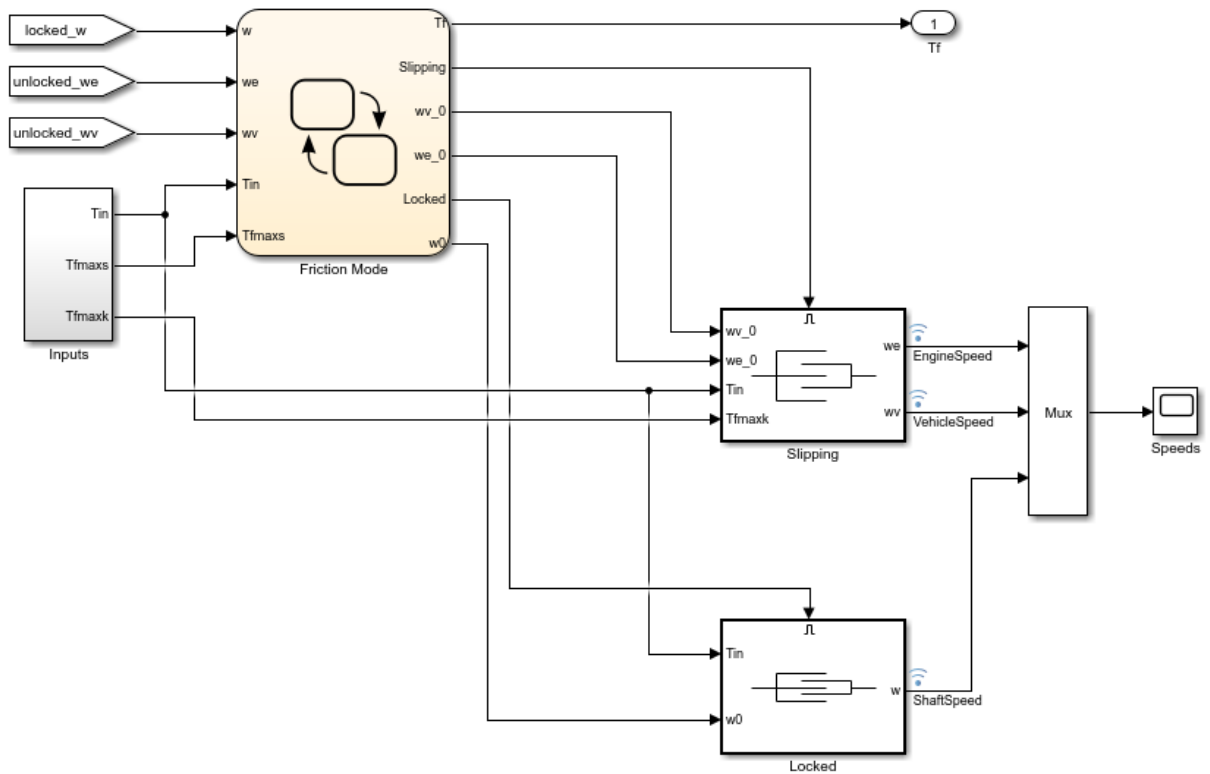
Copyright 2007-2012 The MathWorks, Inc.

2   From the model, copy the block `Slipping` to your Stateflow chart.
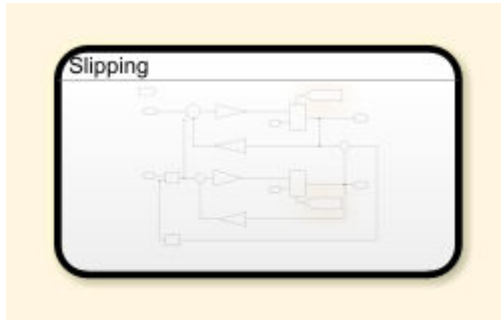
3   The inports and outports of your Simulink subsystem appear as undefined symbols in your Stateflow chart. To add corresponding input and output data to your Stateflow chart, click the **Resolve undefined symbols** button .

Create a Linked Simulink Based State

To create a linked Simulink based state in your Stateflow chart, copy an action subsystem from a library to Stateflow. When the library block is updated, the changes are reflected in all Stateflow charts containing the block.

1   Open the library model.

2   Copy and paste the library block Run_up to your Stateflow chart.
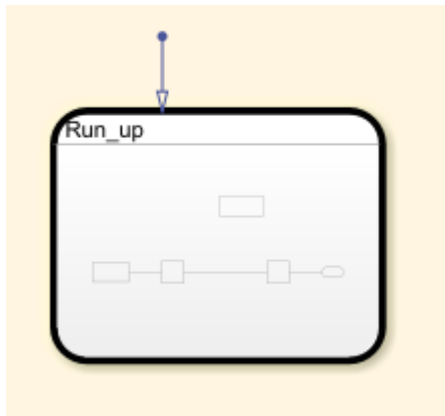
3   To display a link in the bottom leftmost corner on a linked subsystem, select **Display > Library Links > All**.
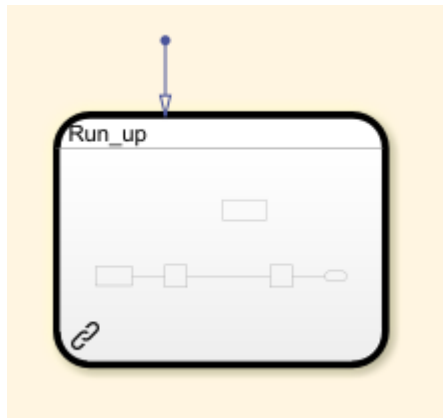
4 The outports of this Simulink subsystem, `xy`, appears as an undefined symbol in your Stateflow chart. To add a corresponding output data to your Stateflow chart, click the

Resolve undefined symbols button .

Create Inports and Outports

When using Simulink based states, inports and outports for your Simulink subsystem connect to input and output data at the Stateflow chart level. This connection allows the top-level Simulink model to read data from the subsystem contained within your Simulink based state.

When you create an empty Simulink based state, Stateflow creates inputs and outputs in your Simulink subsystem that correspond to inputs and outputs that exist in the parent Stateflow chart. However, if you add inports and outports to your Simulink based state after it is created, you must create corresponding input and output data for your Stateflow chart.

To create additional inports or outports for a Simulink based state:

1 Open your Simulink based state.

2 Click the Simulink canvas, type `in1`, and press **Enter**. An undefined inport is created.

3 The undefined symbol `in1` appears in the Symbols window of your Stateflow chart. To resolve the undefined symbol, click the **Resolve undefined symbols** button .

4 A chart inport named `In1` is created.

Create an Additional Outport
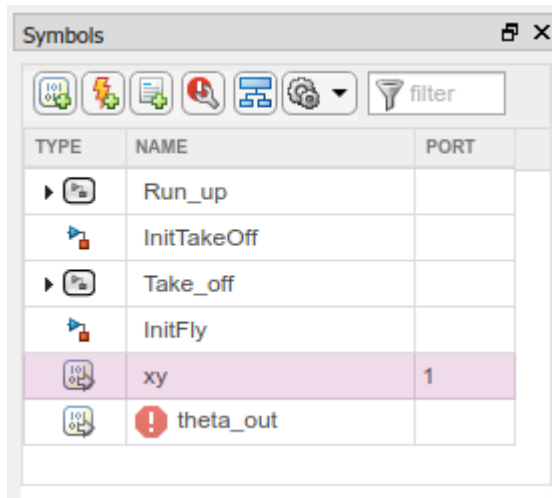
In this example, you create an additional outport for the model `sf_pole_vault`:

1   To open the model.

2   Open the chart `PoleVaulter` and double-click Simulink based state `Take_off`.

3   Click the Simulink based state canvas and type `out1` and press **Enter**. An undefined outport is created. Rename the outport `theta_out` and connect it to the signal for theta.
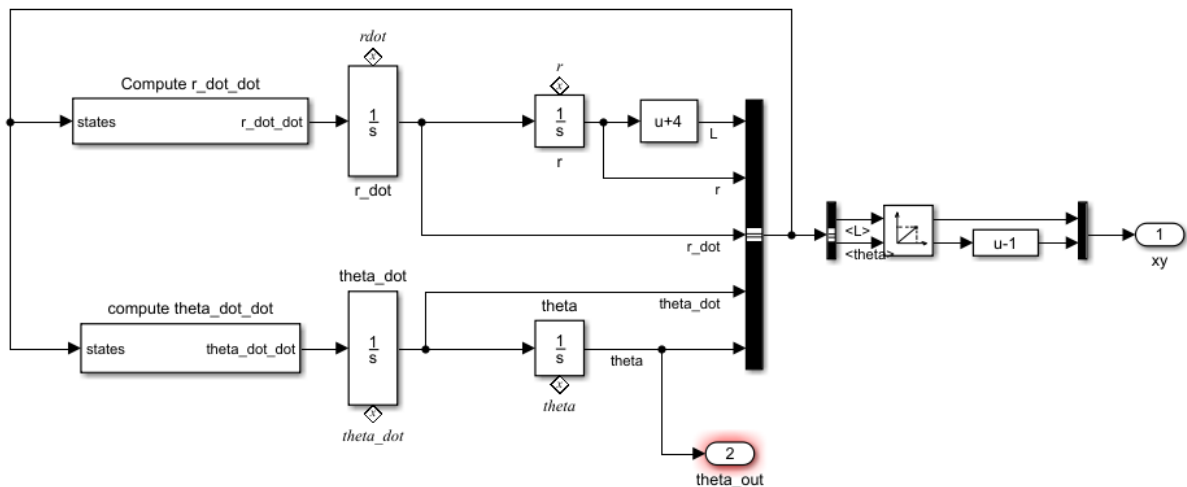
4   In the Symbols window of `PoleVaulter`, an undefined symbol for `theta_out` appears. To resolve the undefined symbol, click the **Resolve undefined symbols** button .

5   Stateflow creates an output in the chart called `theta_out` that corresponds to the
    outport `theta_out`.

For more information about editing data, see "Add and Modify Data, Events, and
Messages" on page 33-3.

See Also

More About

Access Block State Data

To model systems that switch between periodic or continuous time dynamics, use Simulink based states. Simulink based states are supported only in Stateflow charts in a Simulink model. For more information, see "Simulink Subsystems as States" on page 6-2.

You can read and write the state of blocks within your Simulink based states in transition actions of your Stateflow chart. You can read and write the state of blocks textually on the chart transitions or by using Simulink State Reader and State Writer blocks.

This Stateflow chart models a person moving through the stages of pole vaulting. The first stage is the approach run of the vaulter, which is modeled by the Simulink based state Run_up. In the second stage, the vaulter plants the pole and takes off, which is modeled by the Simulink based state Take_off. The final stage happens when the vaulter clears the bar and releases the pole, which is modeled by the Simulink based state Fly.

The states Run_up and Fly are easier to model by using Cartesian coordinates. The state Take_off is easer to model by using polar coordinates. The Simulink functions InitTakeOff and InitFly are used to switch from one coordinate system to another. For more details on this chart, see "Model a Pole Vaulter by Using Simulink Based States" on page 6-2.

Textual Access

This subsystem is contained within the Simulink based state Run_up. For the transition from Run_up to Take_off to occur, the position of the pole vaulter along the x-axis, p(1), must be greater than -4.
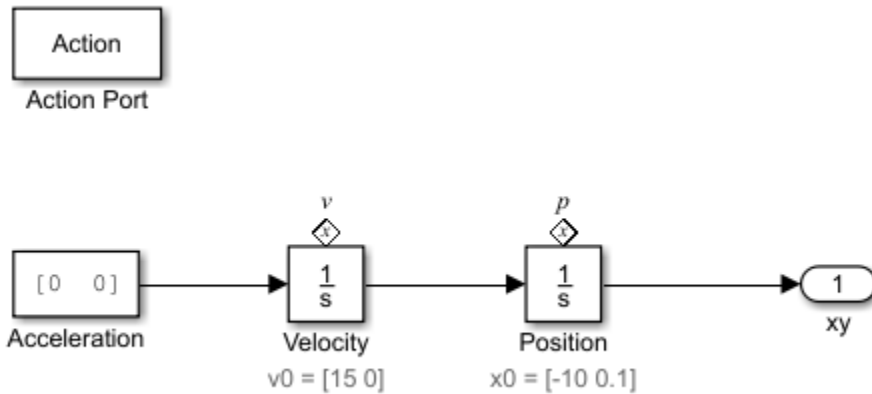
By setting the State Name of the integrator block Position to 'p', you can textually access the state of this block from your Stateflow chart. To access the state of the integrator block in the transition condition, type [Run_up.p(1)> -4]. When this condition becomes true, the transition is taken and the active state becomes Take_off.
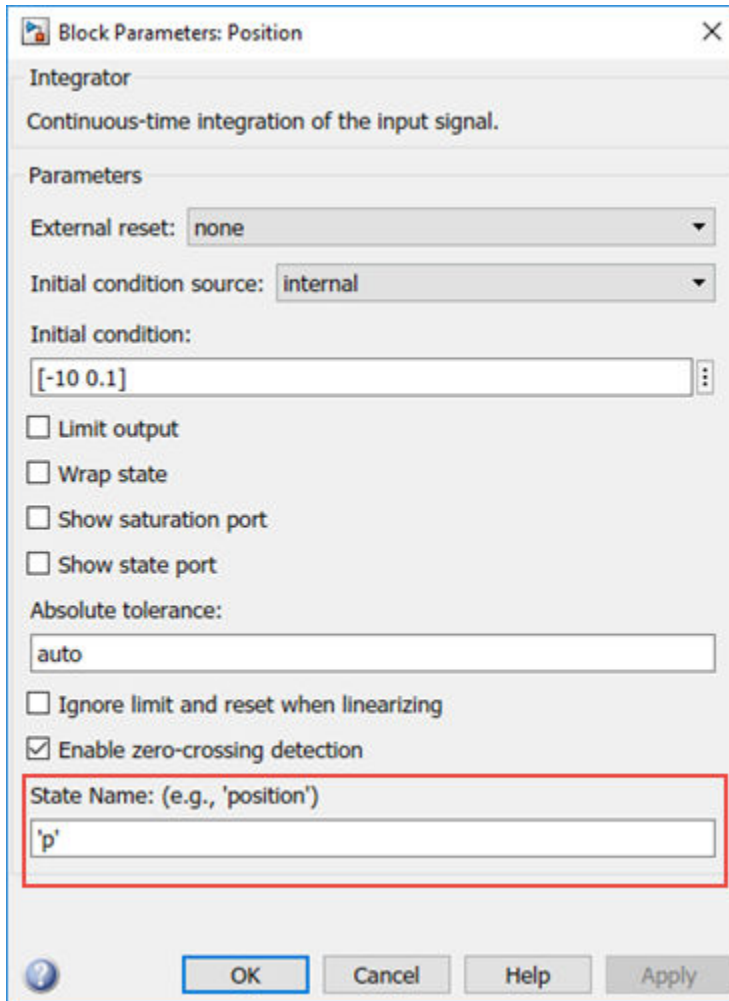
In the Symbols Window, you can see that the state `'p'` appears under the state Run_up.
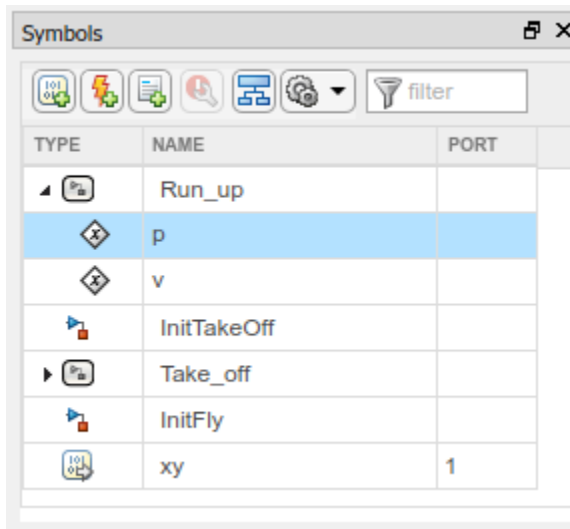
Graphical Access

Stateflow uses State Reader and State Writer blocks to connect the subsystems within a Simulink based state to other Simulink subsystems in your model. State Reader and State Writer blocks display the name of the state owner block that they are connected to.

Conversely, the state owner block displays a tag ⬦ indicating a link to a State Reader or a State Writer block. If you click the label above the tag, a list opens with a link for navigating to the State Writer block.

Connect a State Reader Block to an Owner Block

The following subsystem is contained within the Simulink function `InitTakeOff`. The function uses State Reader blocks to connect to the state `Run_up` and reads `p` and `v`. The function then converts the Cartesian values for the position of the pole vaulter and velocity into polar coordinates, `r` and `theta` and `rdot` and `theta_dot`, respectively. These polar coordinates are then accessed by using state owner blocks in the state `Take_off`.

When the transition action occurs, the State Reader blocks in `InitTakeOff` read the state of their state owner blocks. Once the Simulink function finishes executing, the State Writer blocks write to the state owner blocks in the Simulink based state `Take_off`.
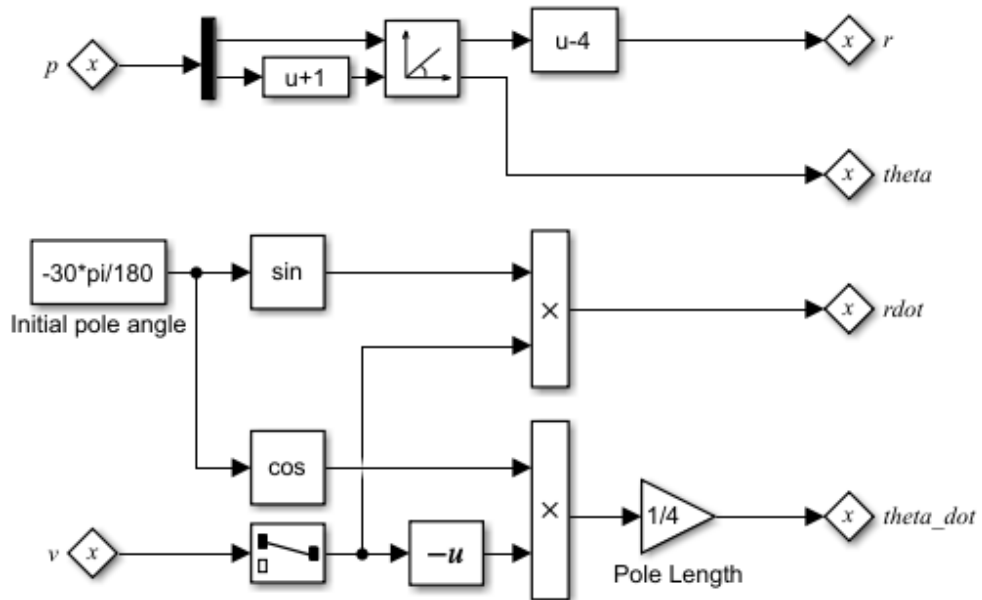
To connect a State Reader or a State Writer block to an owner block within a Simulink subsystem:

1   To open the properties, double-click the State Reader.

2   In the **State Owner Selector Tree**, navigate to the block that you want to be the state owner block. In this example, by choosing `Position`, you connect the State Reader block to the integrator `Position` in the state `Run_up`.
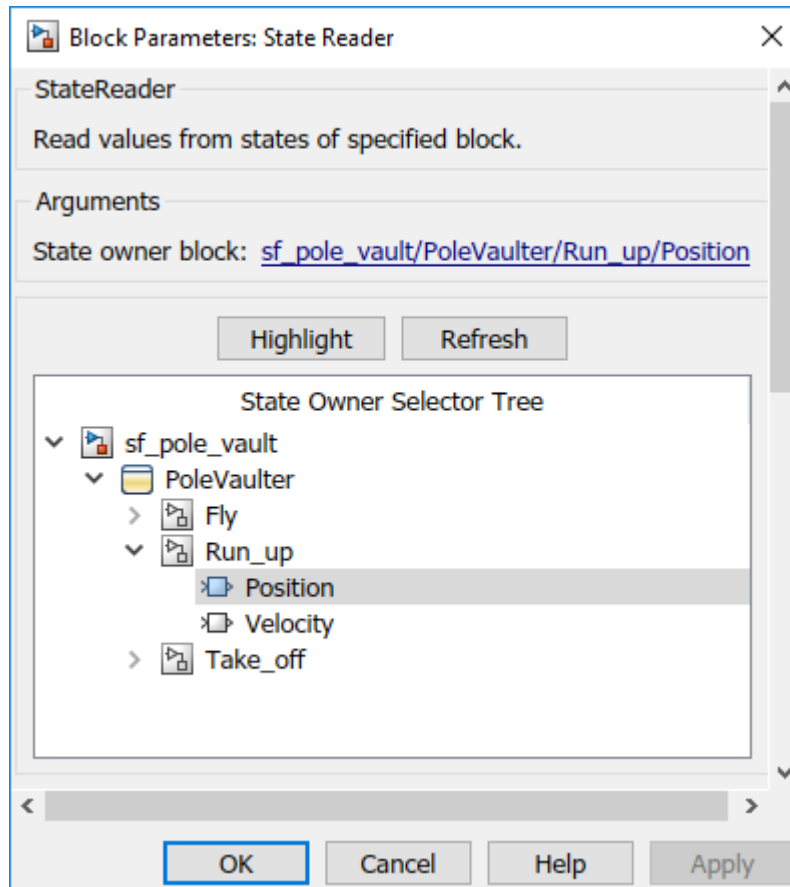
3 By connecting the State Reader block to the `Position` integrator block, this Simulink function can use the state of the integrator `Position` to execute.

See Also

More About

- "Simulink Subsystems as States" on page 6-2
- State Reader

- State Writer

Map Variables for Simulink Based States

To model systems that switch between periodic or continuous time dynamics, use Simulink based states. Simulink based states are supported only in Stateflow charts in a Simulink model. For more information, see "Simulink Subsystems as States" on page 6-2.

You can access inports or outports of a subsystem within a Simulink based state by using inputs and outputs in Stateflow that have the same name as your inports and outports. For Simulink based states that are created by copying and pasting enabled subsystems and action subsystems from a library, click the **Resolve undefined symbols** button to map your Simulink inports and outports to Stateflow inputs and outputs automatically. See "Create Inports and Outports" on page 6-16.

If you are using a linked Simulink based state where the name of the inport or outport differs from the Stateflow chart input or output, you must ensure that your variables are mapped correctly. You can change your mappings from the Property Inspector or in the Mappings dialog box.

Map Variables in a Simulink Based State

To open the mappings dialog box, select **Chart > Mappings**.
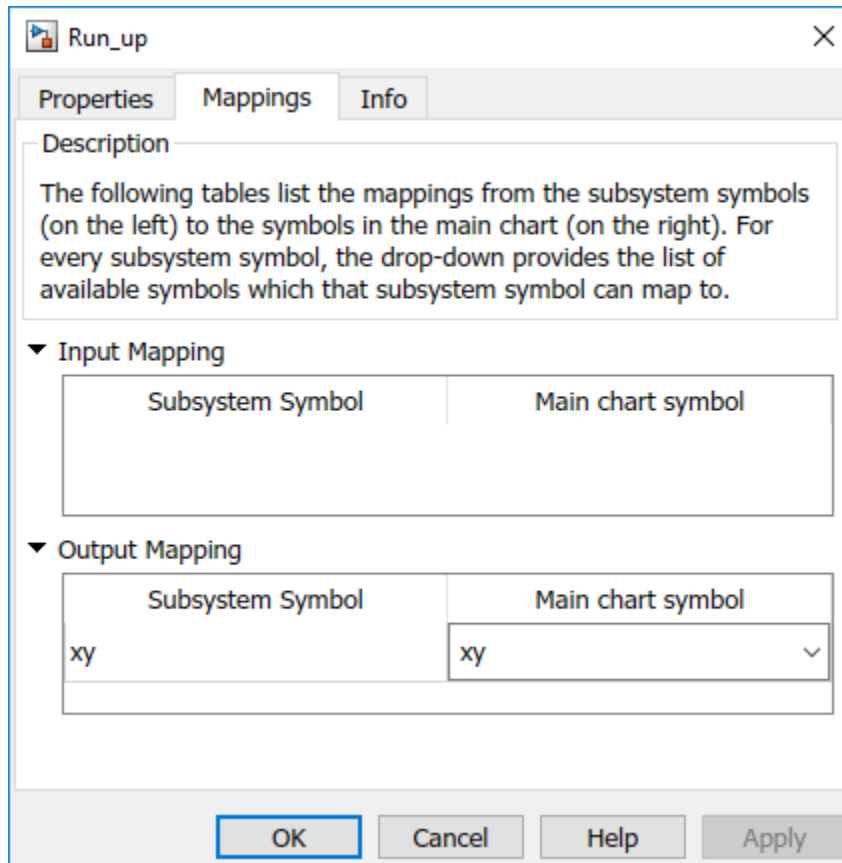
Under **Input Mapping**, you can specify which parent chart input maps to an inport in your Simulink subsystem.

Under **Output Mapping**, you can specify which parent chart output maps to an outport in your Simulink subsystem.

See Also

More About

- "Simulink Subsystems as States" on page 6-2
- "Create and Edit Simulink Based States" on page 6-12
- "Access Block State Data" on page 6-19
- "Resolve Symbols Through the Symbols Window" on page 30-35

Set Simulink Based State Properties

To model systems that switch between periodic or continuous time dynamics, use Simulink based states. Simulink based states are supported only in Stateflow charts in a Simulink model. For more information, see "Simulink Subsystems as States" on page 6-2.

Set and modify the properties of a Simulink based state in the Property Inspector or in a Simulink based state properties dialog box.

For a Simulink based state, you can set these properties.

Simulink Based State Properties	Description
Create data for monitoring self activity	Creates a data output port on the Stateflow block for this self-activity of the state.
Log self activity	Logs the state self-activity. View the activity of the state in the Simulation Data Inspector.
Logging name	Specify the signal logging name. To create a signal logging name that is different from the state name, choose `Custom`, and add the name.
Limit data points to last	Maximum number of data points to log. Default value is 5000, which means the chart logs the last 5000 data points generated by the simulation.
Decimation	Decimation interval limits the amount of data logged by skipping samples. Default value is 2, which means the chart logs every other sample.
Test point	Sets the Simulink based state as a Stateflow test point. See "Monitor Test Points in Stateflow Charts" on page 32-46.

Simulink Based State Properties	Description
Function packaging	Specify the code format generated for a Simulink based state. You can set the format to one of these options: • **Auto** — Simulink Coder software chooses the optimal format for you based on the type and number of instances of the subsystem that exist in the model. • **Inline** — Simulink Coder software inlines the subsystem unconditionally. • **Nonreusable function** — Simulink Coder software explicitly generates a separate function in a separate file. • **Reusable function** — Simulink Coder software generates a function with arguments that allows reuse of Simulink based state code when a model includes multiple instances of the Simulink based state. See "Function packaging" (Simulink).

See Also

More About

- "Rules for Naming Stateflow Objects" on page 2-4
- "Watch Stateflow Data Values" on page 32-36
- "Specify Size of Stateflow Data" on page 9-40
- "Specify Type of Stateflow Data" on page 9-33

7

Build Mealy and Moore Charts

Overview of Mealy and Moore Machines

In a finite state machine, *state* is a combination of local data and chart activity. "Computing state" means updating local data and making transitions from a currently active state to a new state. In state machine models, the next state is a function of the current state and its inputs:

$$X(n + 1) = f(X(n), u)$$

In this equation:

- $X(n)$ represents the state at time step n.
- $X(n+1)$ represents the state at the next time step $n+1$.
- u represents inputs.

State persists from one time step to the next time step.

Semantics of Mealy and Moore Machines

Mealy and Moore machines are often considered the basic, industry-standard paradigms for modeling finite-state machines. You can create charts that implement pure Mealy or Moore semantics as a subset of Stateflow chart semantics. You can use Mealy and Moore charts in simulation and code generation with Embedded Coder®, Simulink Coder, and HDL Coder™ software. Mealy and Moore semantics are supported only in Stateflow charts in Simulink models.

Semantics of Mealy Charts

Mealy machines are finite state machines in which transitions occur on clock edges. The output of a Mealy chart is a function of inputs and state:

$$y = g(X, u)$$

At every time step, a Mealy chart wakes up, evaluates its input, and then transitions to a new configuration of active states, also called its *next state*. The chart computes its output as it transitions to the next state.

To ensure that output is a function of input and state, Mealy state machines enforce these semantics:

- Outputs do not depend on the next state.
- The chart computes outputs only in transitions, not in states.
- The chart wakes up periodically based on a system clock.

Mealy machines compute their output on transitions. Therefore, Mealy charts can compute their first output at the time that the default path for the chart executes. If you enable the chart property **Execute (enter) Chart At Initialization** for a Mealy chart, this computation occurs at $t = 0$ (first time step). Otherwise, it occurs at $t = 1$ (next time step). For more information, see "Execute (Enter) Chart at Initialization" on page 24-7.

Semantics of Moore Charts

Moore machines are finite state machines in which output is modified at clock edges. The output of a Moore chart is a function only of state:

$$y = g(X)$$

At every time step, a Moore chart wakes up, computes its output, and then evaluates its input to reconfigure itself for the next time step. For example, after evaluating its input, the chart can transition to a new configuration of active states. The chart computes its output before evaluating its input and updating its state.

To ensure that output is a function only of the current state, Moore state machines enforce these semantics:

- Outputs do not depend on inputs.
- Outputs do not depend on previous outputs.
- Outputs do not depend on temporal logic.

Moore machines compute their output in states. Therefore, Moore machines can compute outputs only *after* the default path executes. Until then, outputs take the default values.

Create Mealy and Moore Charts

When you create a Stateflow chart, the default type is a hybrid state machine model called a *Classic* chart. Classic charts combine the semantics of Mealy and Moore charts with the extended Stateflow chart semantics.

To create a Mealy chart, at the MATLAB command prompt, enter:

```
sfnew -mealy
```

To create a Moore chart, at the MATLAB command prompt, enter:

```
sfnew -moore
```


Alternatively, after adding a Stateflow chart block to a Simulink model, you can choose the type of semantics for the chart by setting the **State Machine Type** chart property. For more information, see "State Machine Type" on page 24-4.

Advantages of Mealy and Moore Charts

Mealy and Moore charts offer these advantages over Classic Stateflow charts:

- You can verify that the Mealy and Moore charts you create conform to their formal definitions and semantic rules. Error messages appear at compile time (not at design time).

- Moore charts provide a more efficient implementation than Classic charts for C/C++ and HDL targets.

- You can use a Moore chart to model a feedback loop. In Moore charts, inputs do not have direct feedthrough. You can design a loop with feedback from the output port to the input port without introducing an algebraic loop. Mealy and Classic charts have direct feedthrough and produce an error in the presence of an algebraic loop.
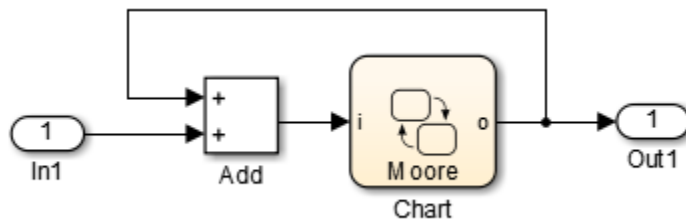
See Also

`sfnew`

More About

- "Design Considerations for Mealy Charts" on page 7-6
- "Design Considerations for Moore Charts" on page 7-11
- "Sequence Recognition by Using Mealy and Moore Charts"
- "Model a Vending Machine by Using Mealy Semantics" on page 7-9
- "Model a Traffic Light by Using Moore Semantics" on page 7-15
- "Convert Charts Between Mealy and Moore Semantics" on page 7-18

Design Considerations for Mealy Charts

Mealy machines are finite state machines in which transitions occur on clock edges. In Mealy charts, output is a function of input and state. At every time step, a Mealy chart wakes up, evaluates its input, and then transitions to a new configuration of active states, also called its *next state*. The chart computes its output as it transitions to the next state. Mealy semantics are supported only in Stateflow charts in Simulink models.

Mealy Semantics

To ensure that output is a function of input and state, Mealy state machines enforce these semantics:

- Outputs do not depend on the next state.
- The chart computes outputs only in transitions, not in states.
- The chart wakes up periodically based on a system clock.

> **Note** A chart provides one time base for input and clock (see "Calculate Output and State by Using One Time Base" on page 7-8).

Design Rules for Mealy Charts

To conform to the Mealy definition of a state machine, ensure that every time there is a change on the input port, the chart computes outputs.

Compute Outputs in Condition Actions Only

You can compute outputs only in the condition actions of outer and inner transitions. A common modeling style for Mealy machines is to test inputs in conditions and compute outputs in the associated action.

Do Not Use State Actions or Transition Actions

You cannot use state actions or transition actions in Mealy charts. This restriction enforces Mealy semantics by:

- Preventing the chart from computing output without considering changes on the input port.

- Ensuring that output depends on the current state and not the next state.

Restrict Data Scope

In Mealy charts, these data restrictions apply:

- **Restrict Machine-Parented Data.** Machine-parented data is data that you define for a Stateflow machine. The Stateflow machine is the highest level of the Stateflow hierarchy. When you define data at this level, every chart in the machine can read and modify the data. To ensure that Mealy charts do not access data that can be modified unpredictably outside the chart, do not use machine-parented data.

- **Do Not Define Data Store Memory.** You cannot define data store memory (DSM) in Mealy charts because objects external to the chart can modify DSM. A Stateflow chart uses data store memory to share data with a Simulink model. Data store memory acts as global data. In the Simulink hierarchy that contains the chart, other blocks and models can modify DSM. Mealy charts must not access data that can change unpredictably.

Restrict Use of Events

Limit the use of events in Mealy charts:

- **Valid Uses:**

  - Use input events to trigger the chart.
  - Use event-based temporal logic to guard transitions.

    The change in value of a temporal logic condition behaves like an event that the Mealy chart schedules internally. At each time step, the number of ticks before the temporal event executes depends only on the state of the chart. For more information, see "Operators for Event-Based Temporal Logic" on page 12-50.

    Note In Mealy charts, the base event for temporal logic operators must be a predefined event such as `tick` or `wakeup` (see "Keywords for Implicit Events" on page 10-33).

- **Invalid Uses:**

  - You cannot broadcast an event of any type.

- You cannot use local events to guard transitions. Local events violate Mealy semantics because they are not deterministic and can occur while the chart computes its outputs.

- You cannot use implicit events such as `chg(data_name)`, `en(state_name)`, or `ex(state_name)`.

Calculate Output and State by Using One Time Base

You can use one time base for clock and input, as determined by the Simulink solver. The Simulink solver sets the clock rate to be fast enough to capture input changes. As a result, a Mealy chart commonly computes outputs and changes states in the same time step. For more information, see "Solvers" (Simulink).

See Also

More About

- "Overview of Mealy and Moore Machines" on page 7-2
- "Model a Vending Machine by Using Mealy Semantics" on page 7-9
- "Convert Charts Between Mealy and Moore Semantics" on page 7-18

Model a Vending Machine by Using Mealy Semantics

This example shows how to use Mealy semantics to model a vending machine.
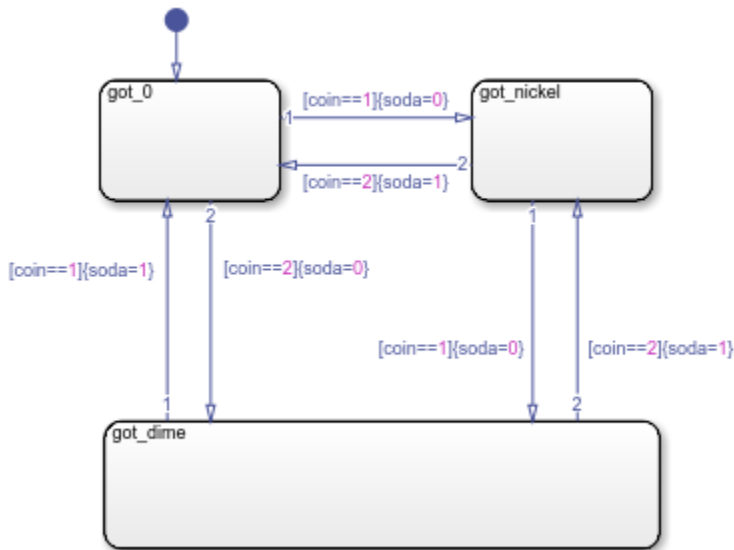
Logic of the Mealy Vending Machine

In this example, the vending machine requires 15 cents to release a can of soda. The purchaser can insert a nickel or a dime, one at a time, to purchase the soda. The chart behaves like a Mealy machine because its output `soda` depends on both the input coin and current state:

When initial state `got_0` is active. No coin has been received or no coins are left.

- If a nickel is received (`coin == 1`), output `soda` remains 0, but state `got_nickel` becomes active.
- If a dime is received (`coin == 2`), output `soda` remains 0, but state `got_dime` becomes active.
- If input coin is not a dime or a nickel, state `got_0` stays active and no soda is released (output `soda = 0`).

In active state `got_nickel`. A nickel was received.

- If another nickel is received (`coin == 1`), state `got_dime` becomes active, but no can is released (`soda` remains at 0).
- If a dime is received (`coin == 2`), a can is released (`soda = 1`), the coins are banked, and the active state becomes `got_0` because no coins are left.
- If input coin is not a dime or a nickel, state `got_nickel` stays active and no can is released (output `soda = 0`).

In active state `got_dime`. A dime was received.

- If a nickel is received (`coin == 1`), a can is released (`soda = 1`), the coins are banked, and the active state becomes `got_0` because no coins are left.
- If a dime is received (`coin == 2`), a can is released (`soda = 1`), 15 cents are banked, and the active state becomes `got_nickel` because a nickel (change) is left.
- If input coin is not a dime or a nickel, state `got_dime` stays active and no can is released (output `soda = 0`).

Design Rules in Mealy Vending Machine

This example of a Mealy vending machine illustrates these Mealy design rules:

- The chart computes outputs in condition actions.
- There are no state actions or transition actions.
- The chart defines chart inputs (`coin`) and outputs (`soda`).
- The value of the input `coin` determines the output: whether or not soda is released.

See Also

More About
- "Overview of Mealy and Moore Machines" on page 7-2
- "Design Considerations for Mealy Charts" on page 7-6
- "Convert Charts Between Mealy and Moore Semantics" on page 7-18

Design Considerations for Moore Charts

Moore machines are finite state machines in which output is modified at clock edges. In Moore charts, output is a function of the current state only. At every time step, a Moore chart wakes up, computes its output, and then evaluates its input to reconfigure itself for the next time step. For example, after evaluating its input, the chart can transition to a new configuration of active states. The chart computes its output before evaluating its input and updating its state. Moore semantics are supported only in Stateflow charts in Simulink models.

Moore Semantics

To ensure that output is a function *only* of the current state, Moore state machines enforce these semantics:

- Outputs do not depend on inputs.
- Outputs do not depend on previous outputs.
- Outputs do not depend on temporal logic.

Design Rules for Moore Charts

To conform to the Moore definition of a state machine, ensure that every time that a Moore chart wakes up, it computes outputs from the current set of active states without regard to inputs.

Restrictions on State Actions

To ensure that outputs depend solely on the current state, you can compute outputs in state actions, subject to these restrictions:

- **Combine Actions.** In Moore charts, you can include only one action per state. The action can consist of multiple command statements. Stateflow evaluates states in Moore charts from the top level down. Active states in Moore charts execute the state action before evaluating the transitions. Therefore, outputs are computed at each time step whether an outer transition is valid or not.

- **Do Not Label State Actions.** Do not label state actions in Moore charts with any keywords, such as `entry`,`during`, or `exit`. By default, Moore charts execute the actions in the active states before evaluating inputs and updating state.

Restrictions on Transition Actions

Transitions in Moore charts can contain condition and transition actions if these actions do not introduce a dependency between output values and input values. For example, in this chart, each transition tests the input u in a condition and modifies the output y in a condition action. Because the output value depends on the value of the input, this construct violates Moore semantics and triggers an error.
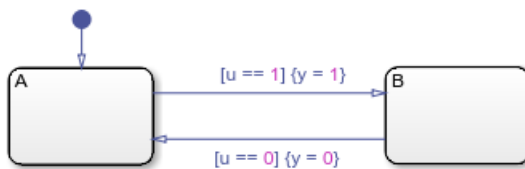
Do Not Use Inputs to Compute Outputs

In Moore charts, outputs cannot depend on inputs. Using an input to contribute directly or indirectly to the computation of an output triggers an error.

Restrict Data Scope

In Moore charts, these data restrictions apply:

- **Restrict Machine-Parented Data.** Machine-parented data is data that you define for a Stateflow machine. The Stateflow machine is the highest level of the Stateflow hierarchy. When you define data at this level, every chart in the machine can read and modify the data. To ensure that Moore charts do not access data that can be modified unpredictably outside the chart, do not use machine-parented data.

- **Do Not Define Data Store Memory.** You cannot define data store memory (DSM) in Moore charts because objects external to the chart can modify DSM objects. A Stateflow chart uses data store memory to share data with a Simulink model. Data store memory acts as global data. In the Simulink hierarchy that contains the chart, other blocks and models can modify DSM. Moore charts must not access data that can change unpredictably.

Do Not Use coder.extrinsic to Call Extrinsic Functions

When calling extrinsic functions with `coder.extrinsic`, it is not possible to enforce that the outputs depend only on the current state. Calling an extrinsic function with `coder.extrinsic` in a Moore chart triggers an error.

Do Not Use Simulink Functions

You cannot use Simulink functions in Moore charts. This restriction prevents violations of Moore semantics during chart execution.

Do Not Export Functions

You cannot export functions in a Moore chart.

Do Not Disable Inlining

Moore chart semantics require inlining.

Do Not Enable Super Step Semantics

You cannot use super step semantics in a Moore chart.

Do Not Use Messages

You cannot use messages in a Moore chart.

Restrict Use of Events

Limit the use of events in Moore charts:

- **Valid Uses:**

  - Use only one input event to trigger the chart.
  - Use event-based temporal logic to guard transitions.

    The change in value of a temporal logic condition behaves like an event that the Moore chart schedules internally. At each time step, the number of ticks before the temporal event executes depends only on the state of the chart. For more information, see "Operators for Event-Based Temporal Logic" on page 12-50.

    Note In Moore charts, the base event for temporal logic operators must be a predefined event such as `tick` or `wakeup` (see "Keywords for Implicit Events" on page 10-33).

- **Invalid Uses:**

  - You cannot broadcast an event of any type.

- You cannot use local events to guard transitions. Local events violate Moore semantics because they are not deterministic and can occur while the chart computes its outputs.
- You cannot use implicit events such as `chg(data_name)`, `en(state_name)`, or `ex(state_name)`.

Do Not Use Moore Charts for Modeling Continuous-Time Systems

In Moore charts, you cannot set the update method to `Continuous`. For modeling systems with continuous time in Stateflow, use Classic or Mealy charts.

See Also

More About

- "Overview of Mealy and Moore Machines" on page 7-2
- "Model a Traffic Light by Using Moore Semantics" on page 7-15
- "Convert Charts Between Mealy and Moore Semantics" on page 7-18

Model a Traffic Light by Using Moore Semantics

This example shows how to use Moore semantics to model a traffic light.
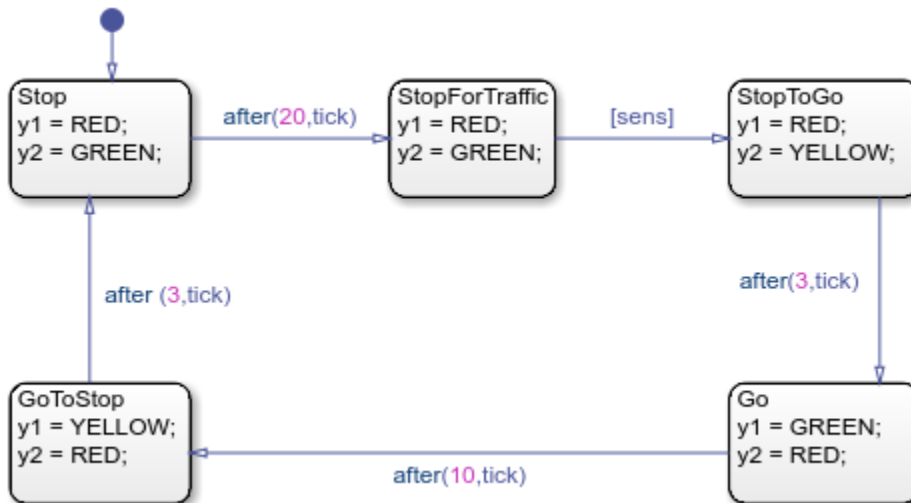
Logic of the Moore Traffic Light

In this example, the traffic light model contains a Moore chart called Light_Controller, which operates in five traffic states. Each state represents the color of the traffic light in two opposite directions, North-South and East-West, and the duration of the current color. The name of each state represents the operation of the light viewed from the North-South direction.

This chart uses temporal logic to regulate state transitions. The `after` operator implements a countdown timer, which initializes when the source state is entered. By default, the timer provides a longer green light in the East-West direction than in the North-South direction because the volume of traffic is greater on the East-West road. The green light in the East-West direction stays on for at least 20 clock ticks, but it can remain green as long as no traffic arrives in the North-South direction. A sensor detects whether cars are waiting at the red light in the North-South direction. If so, the light turns green in the North-South direction to keep traffic moving.

The Light_Controller chart behaves like a Moore machine because it updates its outputs based on current state before transitioning to a new state:

When initial state `Stop` is active. Traffic light is red for North-South, green for East-West.

- Sets output `y1` = RED (North-South) based on current state.
- Sets output `y2` = GREEN (East-West) based on current state.
- After 20 clock ticks, active state becomes `StopForTraffic`.

In active state `StopForTraffic`. Traffic light has been red for North-South, green for East-West for at least 20 clock ticks.

- Sets output `y1` = RED (North-South) based on current state.
- Sets output `y2` = GREEN (East-West) based on current state.
- Checks sensor.
- If sensor indicates cars are waiting (`[sens]` is true) in the North-South direction, active state becomes `StopToGo`.

In active state `StopToGo`. Traffic light must reverse traffic flow in response to sensor.

- Sets output `y1` = RED (North-South) based on current state.
- Sets output `y2` = YELLOW (East-West) based on current state.
- After 3 clock ticks, active state becomes `Go`.

In active state `Go`. Traffic light has been red for North-South, yellow for East-West for 3 clock ticks.

- Sets output `y1` = GREEN (North-South) based on current state.
- Sets output `y2` = RED (East-West) based on current state.
- After 10 clock ticks, active state becomes `GoToStop`.

In active state `GoToStop`. Traffic light has been green for North-South, red for East-West for 10 clock ticks.

- Sets output `y1` = YELLOW (North-South) based on current state.
- Sets output `y2` = RED (East-West) based on current state.
- After 3 clock ticks, active state becomes `Stop`.

Design Rules in Moore Traffic Light

This example of a Moore traffic light illustrates these Moore design rules:

- The chart computes outputs in state actions.
- The chart tests inputs in conditions on transitions.
- The chart uses temporal logic, but no asynchronous events.
- The chart defines chart inputs (`sens`) and outputs (`y1` and `y2`).

See Also

More About
- "Overview of Mealy and Moore Machines" on page 7-2
- "Design Considerations for Moore Charts" on page 7-11
- "Convert Charts Between Mealy and Moore Semantics" on page 7-18
- "Model An Intersection Of One-Way Streets"
- "Model a Distributed Traffic Control System by Using Messages"

Convert Charts Between Mealy and Moore Semantics

Mealy and Moore machines are often considered the basic, industry-standard paradigms for modeling finite-state machines. You can create charts that implement pure Mealy or Moore semantics as a subset of Stateflow chart semantics. Mealy and Moore semantics are supported only in Stateflow charts in Simulink models. For more information, see "Overview of Mealy and Moore Machines" on page 7-2.

A best practice is to not change from one Stateflow chart type to another in the middle of development. You cannot automatically convert the semantics of the original chart to conform to the design rules of the new chart type. Changing chart type usually requires you to redesign your chart to achieve the equivalent behavior of the original chart, so that both charts produce the same sequence of outputs given the same sequence of inputs. Sometimes, however, there is no way to translate specific behaviors without violating chart definitions.

This table lists a summary of what happens when you change chart types mid-design.

From	To	Result
Mealy	Classic	Mealy charts retain their semantics when changed to Classic type.
Classic	Mealy	If the Classic chart defines its output at every time step and conforms to Mealy semantic rules, the Mealy chart exhibits behavior equivalent to the original Classic chart.
Moore	Classic	State actions in the Moore chart behave as `entry` and `during` actions because they are not labeled. The Classic chart exhibits behavior that is not equivalent to the original Moore chart. Requires redesign.
Classic	Moore	Actions that are unlabeled in the Classic chart (`entry` and `during` actions by default) behave as `during` and `exit` actions. The Moore chart exhibits behavior that is not equivalent to the original Classic chart. Requires redesign.
Mealy	Moore	Mealy and Moore rules about placement of actions are mutually exclusive. Converting chart types between Mealy and Moore semantics does not produce equivalent behavior. Requires redesign.
Moore	Mealy	

Transform Chart from Mealy to Moore Semantics

This example shows how to use a Mealy chart to model a vending machine, as described in "Model a Vending Machine by Using Mealy Semantics" on page 7-9.
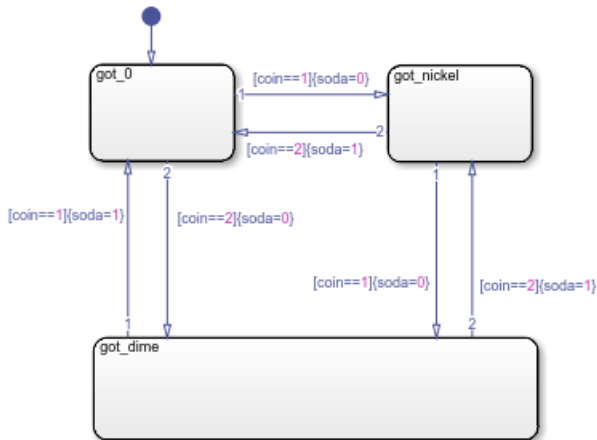
In the Mealy chart, each state represents one of the three possible coin inputs: nickel, dime, or no coin. Each condition action computes output (whether soda is released) based on input (the coin received) as the chart transitions to the next state. If you change the chart type to `Moore`, you get a compile-time diagnostic message indicating that the chart violates Moore chart semantics. According to Moore semantics, the chart output `soda` cannot depend on the value of the input `coin`.

To convert the chart to valid Moore semantics, redesign your chart by moving the logic that computes the output out of the transitions and into the states. In the Moore chart, each state must represent both coins received and the soda release condition (`soda = 0` or `soda = 1`). As a result, the redesigned chart requires more states.
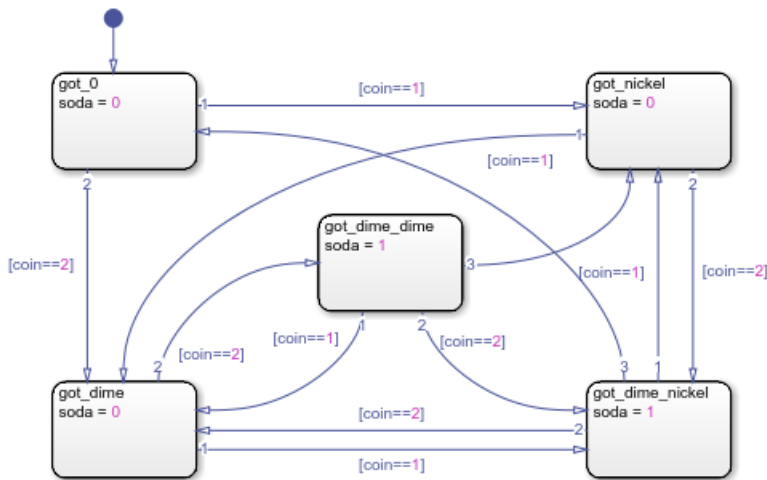
Before considering the value of `coin`, the Moore chart must compute the value of `soda` according to the active state. As a result, there is a delay in releasing soda, even if the machine receives enough money to cover the cost.

Compare Semantics of Vending Machines

This table compares the semantics of the charts before and after the transformation.

Mealy Vending Machine	Moore Vending Machine
Uses three states	Uses five states
Computes outputs in condition actions	Computes outputs in state actions
Updates output based on input	Updates output before evaluating input, requiring an extra time step to produce the soda

For this vending machine, Mealy is a better modeling paradigm because there is no delay in releasing the soda once sufficient coins are received. By contrast, the Moore vending machine requires an extra time step before producing the soda. Therefore, it is possible for the Moore vending machine to produce a can of soda at the same time step that it accepts a coin toward the next purchase. In this situation, the delivery of a soda can appear to be in response to this coin, but actually occurs because the vending machine received the purchase price in previous time steps.

Transform Chart from Moore to Mealy Semantics

This example shows how to use a Moore chart to model a traffic light, as described in "Model a Traffic Light by Using Moore Semantics" on page 7-15.
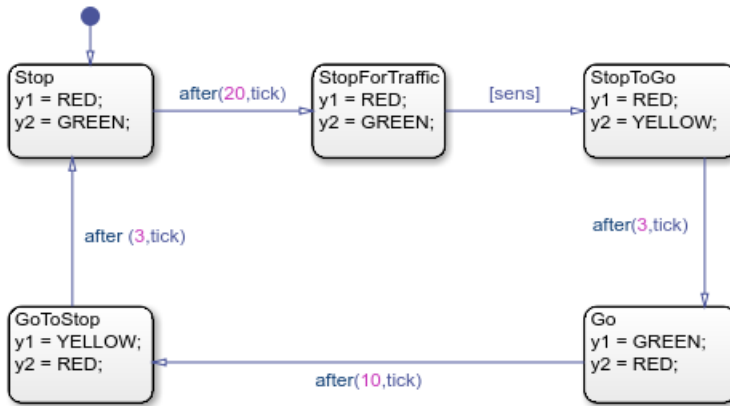
In the Moore chart, each state represents the colors of the traffic light in two opposite directions and the duration of the current color. Each state action computes output (the colors of the traffic light) regardless of input (if there are cars waiting at a sensor). If you change the chart type to `Mealy`, you get a compile-time diagnostic message indicating that the chart violates Mealy chart semantics. According to Mealy semantics, the chart cannot compute its outputs in state actions.

To convert the chart to valid Mealy semantics, redesign your chart by moving the logic that computes the output out of the states and into the transitions. The redesigned Mealy chart consists of five states, just like the Moore chart. In most transitions, a temporal logic expression or the Boolean input `sens` guards a condition action computing the outputs `y1` and `y2`. The only exceptions are:

- The default transition, which computes the initial outputs without a guarding condition.
- The transition from the `Stop` state to the `StopForTraffic` state, which does not compute new outputs.
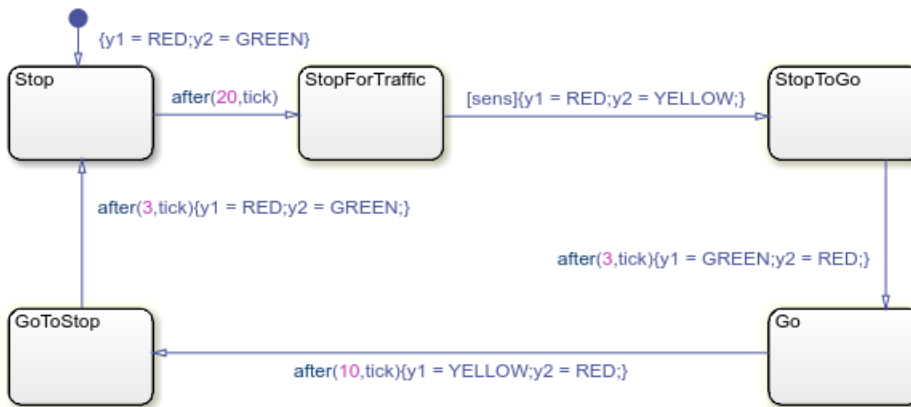
In the same time step, the Mealy chart evaluates the temporal logic expressions and the input signal `sens`, and computes the value of the outputs `y1` and `y2`. As a result, in the Mealy chart, the output changes one time step before the corresponding change occurs in the original Moore chart. In the Simulink model, you can compensate for the anticipated changes in output by adding a Delay block to each output signal.

Compare Semantics of Traffic Light Controllers

This table compares the semantics of the charts before and after the transformation.

Moore Traffic Light Controller	Mealy Traffic Light Controller
Uses five states	Uses five states
Computes outputs in state actions	Computes outputs in condition actions
Updates output before evaluating input	Updates output based on input, requiring a Delay block in each output signal

See Also

More About

- "Overview of Mealy and Moore Machines" on page 7-2
- "Design Considerations for Mealy Charts" on page 7-6

- "Design Considerations for Moore Charts" on page 7-11
- "Model a Vending Machine by Using Mealy Semantics" on page 7-9
- "Model a Traffic Light by Using Moore Semantics" on page 7-15

Initialize Persistent Variables in MATLAB Functions

A persistent variable is a local variable in a MATLAB function that retains its value in memory between calls to the function. For code generation, functions must initialize a persistent variable if it is empty. For more information, see `persistent`.

When programming MATLAB functions in these situations:

- MATLAB Function blocks with no direct feedthrough
- MATLAB Function blocks in models that contain State Control blocks in `Synchronous` mode
- MATLAB functions in Stateflow charts that implement Moore machine semantics

the specialized semantics impact how a function initializes its persistent data. Because the initialization must be independent of the input to the function, follow these guidelines:

- The function initializes its persistent variables only by accessing constants.
- The control flow of the function does not depend on whether the initialization occurs.

For example, this function has a persistent variable n.

```
function y = fcn(u)
    persistent n

    if isempty(n)
        n = u;
        y = 1;
        return
    end

    y = n;
    n = n + u;
end
```

This type of initialization results in an error because the initial value of n depends on the input u and the `return` statement interrupts the normal control flow of the function.

To correct the error, initialize the persistent variable by setting it to a constant value and remove the `return` statement. For example, this function initializes the persistent variable without producing an error.

```
function y = fcn(u)
    persistent n
```

```
    if isempty(n)
        n = 1;
    end

    y = n;
    n = n + u;
end
```

MATLAB Function Block With No Direct Feedthrough

This model contains a MATLAB function block that defines the function `fcn`, described previously. The input `u` is a square wave with values of 1 and -1.
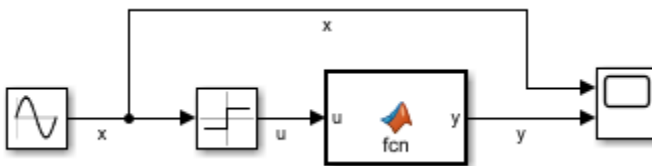
In the MATLAB function block:

- The initial value of the persistent variable `n` depends on the input `u`.
- The `return` statement interrupts the normal control flow of the function.

Because the `Allow direct feeedthrough` check box is cleared, the initialization results in an error.

If you modify the function so it initializes `n` independently of the input, then you can simulate an error-free model.
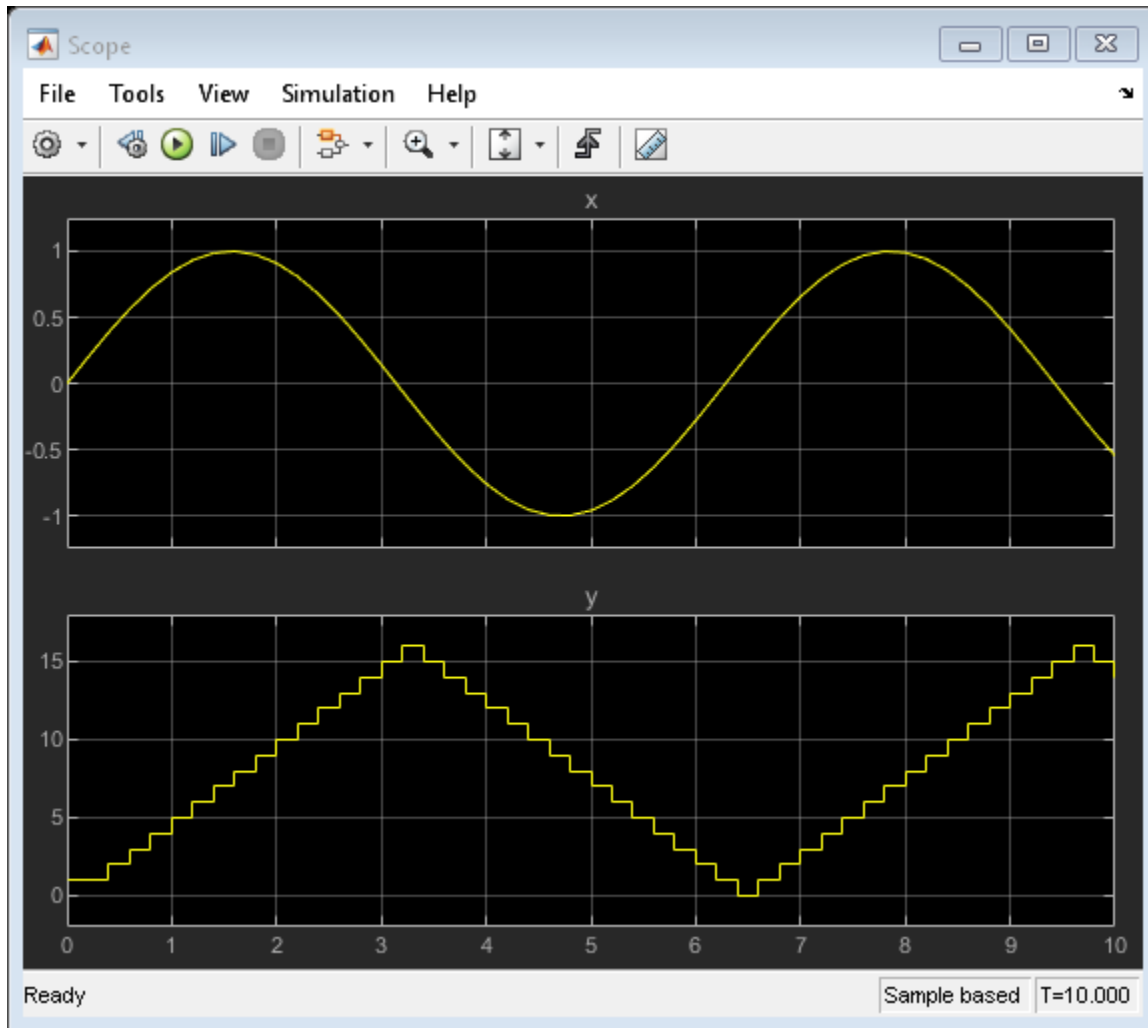
State Control Block in Synchronous Mode

This model contains a MATLAB function block that defines the function `fcn`, described previously. The input `u` is a square wave with values of 1 and -1.
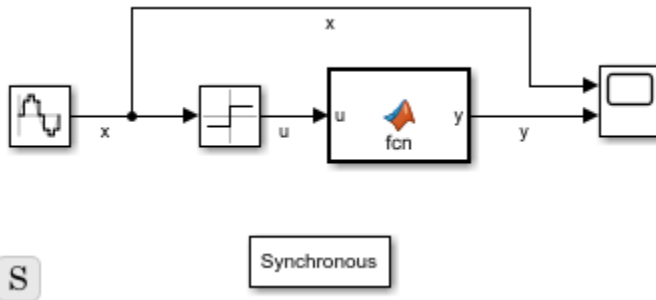
In the MATLAB function block:

- The initial value of the persistent variable n depends on the input u.
- The return statement interrupts the normal control flow of the function.

Because the model contains a State Control block in Synchronous mode, the initialization results in an error.

If you modify the function so it initializes n independently of the input, then you can simulate an error-free model.
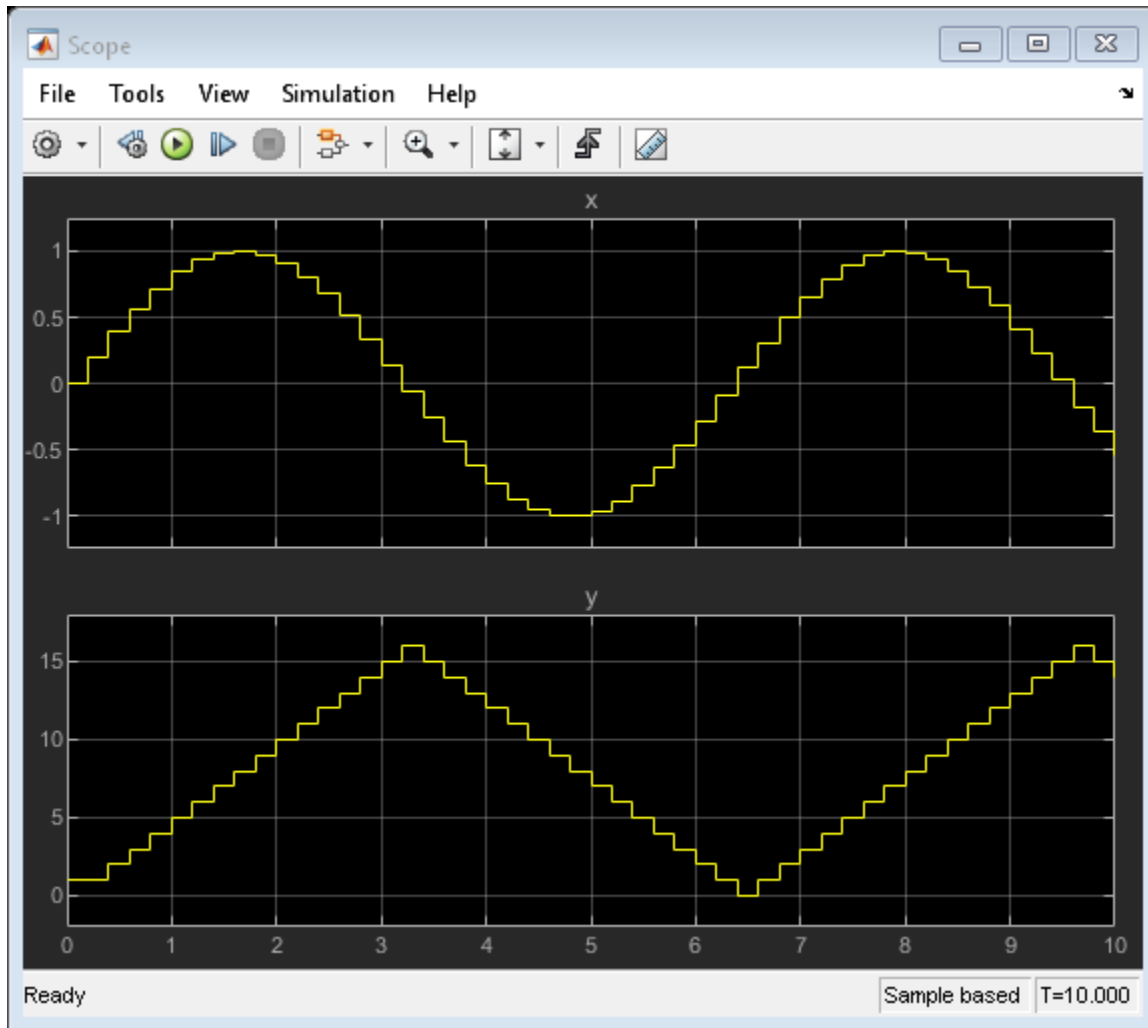
Stateflow Chart Implementing Moore Semantics

This model contains a Stateflow chart that implements Moore machine semantics. The chart contains a MATLAB function that defines the function fcn, described previously. The input u has values of 1 and -1 that depend on the state of the chart.
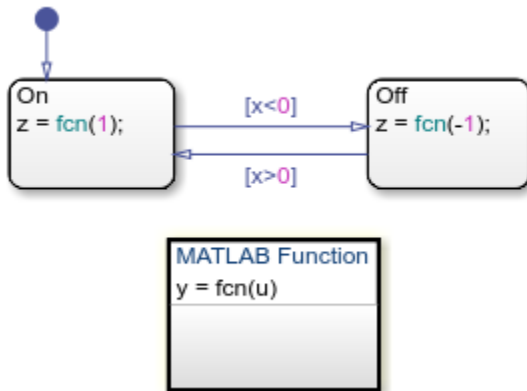
In the MATLAB function:

- The initial value of the persistent variable n depends on the input u.
- The return statement interrupts the normal control flow of the function.

Because the chart implements Moore semantics, the initialization results in an error.

If you modify the function so it initializes n independently of the input, then you can simulate an error-free model.
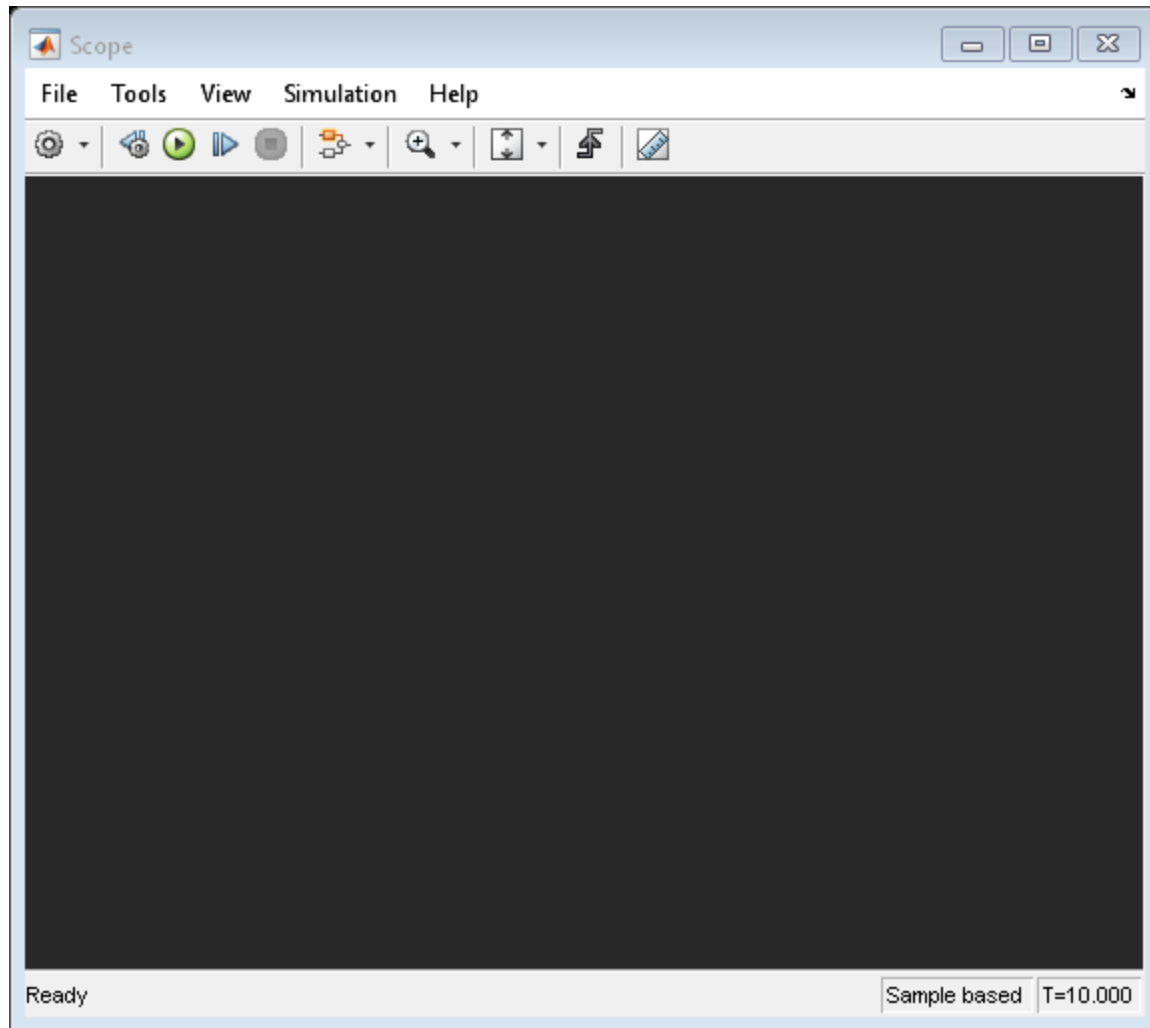
See Also

Chart | MATLAB Function | State Control | `persistent`

More About

- "Use Nondirect Feedthrough in a MATLAB Function Block" (Simulink)
- "Synchronous Subsystem Behavior with the State Control Block" (HDL Coder)
- "Design Considerations for Moore Charts" on page 7-11

8

Techniques for Streamlining Chart Design

Record State Activity by Using History Junctions

A history junction records the activity of substates inside superstates. Use a history junction in a chart or superstate to indicate that its last active substate becomes active when the chart or superstate becomes active. For more information, see "History Junctions" on page 2-44.

Create a History Junction

To create a history junction, do the following:

1 In the editor toolbar, click the History Junction icon:

2 Move your pointer into the chart.

3 Click to place a history junction inside the state whose last active substate it records.

To create multiple history junctions, do the following:

1 In the editor toolbar, double-click the History Junction icon.

The button is now in multiple-object mode.

2 Click anywhere in the drawing area to place a history junction.

3 Move to and click another location to create an additional history junction.

4 Click the History Junction icon or press the **Esc** key to cancel the operation.

To move a history junction to a new location, click and drag it to the new position.

Change History Junction Size

To change the size of junctions:

1 Select the history junctions whose size you want to change.

2 Right-click one of the junctions and select **Junction Size**.

3 Select a size from the list of junction sizes.

Change History Junction Properties

To edit the properties for a junction:

1 Right-click a junction and select **Properties**.

The History Junction dialog box appears.
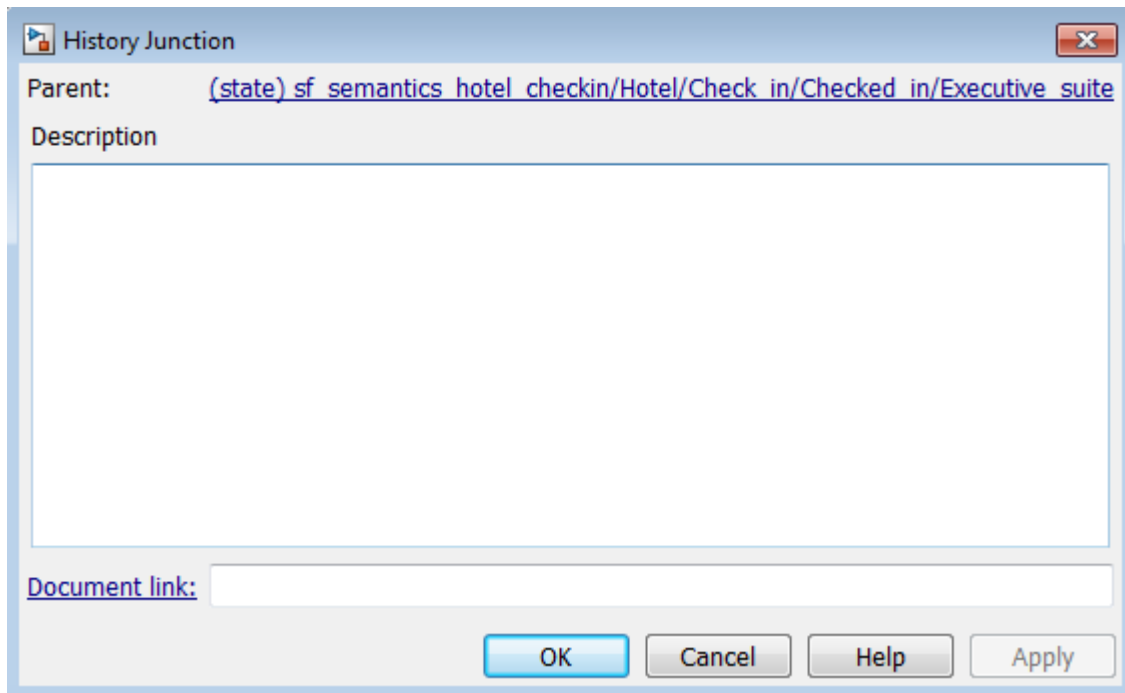
2 Edit the fields in the properties dialog box.

Field	Description
Parent	Parent of this history junction; read-only; click the hypertext link to bring the parent to the foreground.
Description	Textual description/comment.

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

3 When finished editing, click one of the following buttons:

- **Apply** to save the changes
- **Cancel** to cancel any changes
- **OK** to save the changes and close the dialog box
- **Help** to display the Stateflow online help in an HTML browser window

Encapsulate Modal Logic by Using Subcharts

A subchart is a graphical object that can contain anything that a top-level chart can, including other subcharts. A subchart, or a subcharted state, is a superstate of the states that it contains. You can nest subcharts to any level in your chart design.

Using subcharts, you can reduce a complex chart to a set of simpler, hierarchically organized units. This design makes the chart easier to understand and maintain, without changing the chart behavior. Subchart boundaries do not apply during simulation and code generation.

The subchart appears as a block with its name in the block center. However, you can define actions and default transitions for subcharts just as you can for superstates. You can also create transitions to and from subcharts just as you can create transitions to and from superstates. You can create transitions between states residing outside a subchart and any state within a subchart. The term *supertransition* refers to a transition that crosses subchart boundaries in this way. See "Move Between Levels of Hierarchy by Using Supertransitions" on page 8-9 for more information.

Subcharts define a containment hierarchy within a top-level chart. A subchart or top-level chart is the *parent* of the states it contains at the first level and an *ancestor* of all the subcharts contained by its children and their descendants at lower levels.

Some subcharts can become *atomic* units if they meet certain modeling requirements. For more information, see "Restrictions for Converting to Atomic Subcharts" on page 15-32.

Create a Subchart

You create a subchart by converting an existing state, box, or graphical function into the subchart. The object to convert can be one that you create for making a subchart or an existing object whose contents you want to turn into a subchart.

To convert a new or existing state, box, or graphical function to a subchart:

1   Right-click the object and select **Group & Subchart** > **Subchart**.
2   Confirm that the object now appears as a subchart.

To convert the subchart back to its original form, right-click the subchart. In the context menu, select **Group & Subchart** > **Subchart**.

Rules of Subchart Conversion

When you convert a box to a subchart, the subchart retains the attributes of a box. For example, the position of the resulting subchart determines its activation order in the chart if implicit ordering is enabled (see "Group Chart Objects by Using Boxes" on page 8-29 for more information).

You cannot undo the operation of converting a subchart back to its original form. When you perform this operation, the undo and redo buttons are disabled from undoing and redoing any prior operations.

Convert a State to a Subchart

Suppose that you have the following chart:
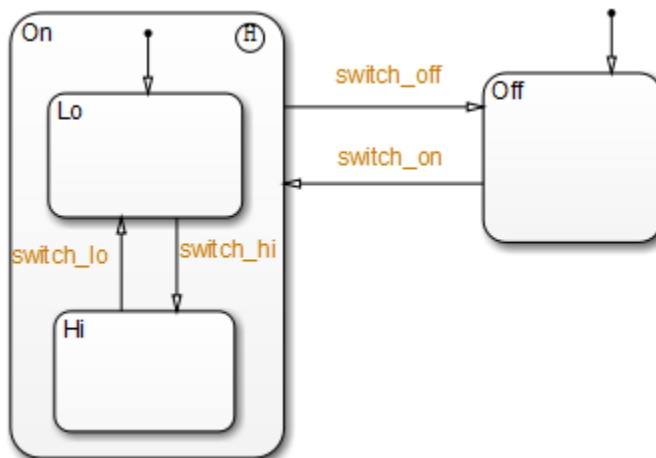
1   To convert the On state to a subchart, right-click the state and select **Group & Subchart** > **Subchart**.

2   Confirm that the On state now appears as a subchart.
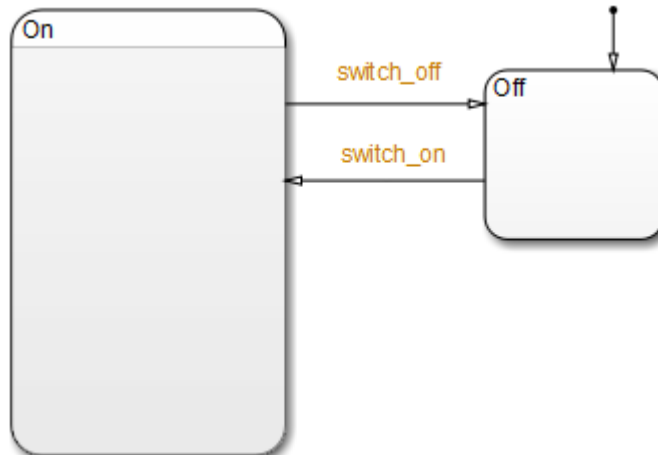
Manipulate Subcharts as Objects

Subcharts also act as individual objects. You can move, copy, cut, paste, relabel, and resize subcharts as you would states and boxes. You can also draw transitions to and from a subchart and any other state or subchart at the same or different levels in the chart hierarchy (see "Move Between Levels of Hierarchy by Using Supertransitions" on page 8-9).

Open a Subchart

Opening a subchart allows you to view and change its contents. To open a subchart, do one of the following:

- Double-click anywhere in the box that represents the subchart.
- Select the box representing the subchart and press the **Enter** key.

Edit a Subchart

After you open a subchart (see "Open a Subchart" on page 8-7), you can perform any editing operation on its contents that you can perform on a top-level chart. This means that you can create, copy, paste, cut, relabel, and resize the states, transitions, and subcharts in a subchart. You can also group states, boxes, and graphical functions inside subcharts.

You can also cut and paste objects between different levels in your chart. For example, to copy objects from a top-level chart to one of its subcharts, first open the top-level chart and copy the objects. Then open the subchart and paste the objects into the subchart.

Transitions from outside subcharts to states or junctions inside subcharts are called *supertransitions*. You create supertransitions differently than you do ordinary transitions. See "Move Between Levels of Hierarchy by Using Supertransitions" on page 8-9 for information on creating supertransitions.

Navigate Subcharts

The Stateflow Editor toolbar contains a set of buttons for navigating the subchart hierarchy of a chart.

Tool	Description
⬆	If the Stateflow Editor is displaying a subchart, clicking this button replaces the subchart with the parent of the subchart in the Stateflow Editor. If the Stateflow Editor is displaying a top-level chart, clicking this button replaces the chart with the Simulink model window containing that chart.
⬅	Clicking this button shows the chart that you visited before the current chart, so that you can navigate up the hierarchy.
➡	Clicking this button shows the chart that you visited after visiting the current chart, so that you can navigate down the hierarchy.

Note You can also use the **Escape** key to navigate up to the parent object for a subcharted state, box, or function.

Move Between Levels of Hierarchy by Using Supertransitions

A supertransition is a transition between different levels in a chart, for example, between a state in a top-level chart and a state in one of its subcharts, or between states residing in different subcharts at the same or different levels in a chart. You can create supertransitions that span any number of levels in your chart, for example, from a state at the top level to a state that resides in a subchart several layers deep in the chart.

The point where a supertransition enters or exits a subchart is called a *slit*. Slits divide a supertransition into graphical segments. For example, the `sf_boiler` model shows a supertransition leaving the `On` subchart:
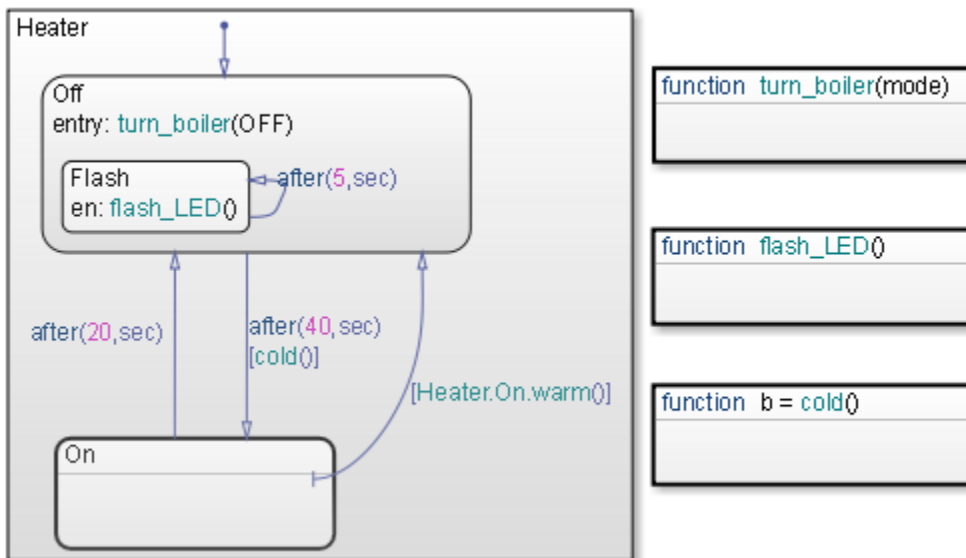
The same supertransition appears inside the subchart as follows:
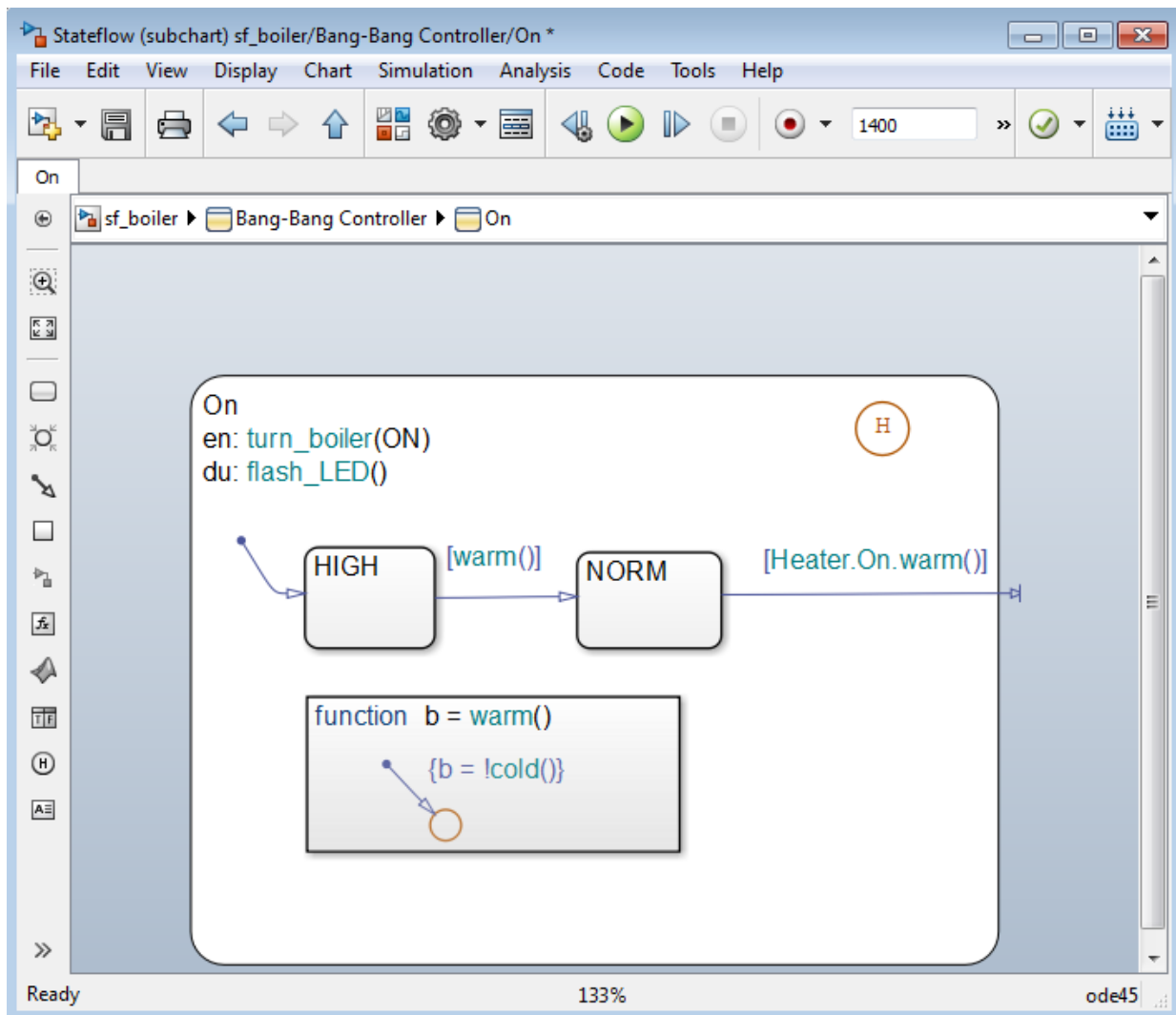
In this example, supertransition `[Heater.On.warm()]` goes from NORM in the On subchart to the Off state in the parent chart. Both segments of the supertransition have the same label.

Draw a Supertransition Into a Subchart

Use the following steps to draw a supertransition from an object outside a subchart to an object inside the subchart.

Note You cannot undo the operation of drawing a supertransition. When you perform this operation, the undo and redo buttons are disabled from undoing and redoing any prior operations.

---

1 Position your cursor over the border of the state.

The cursor assumes the crosshairs shape.
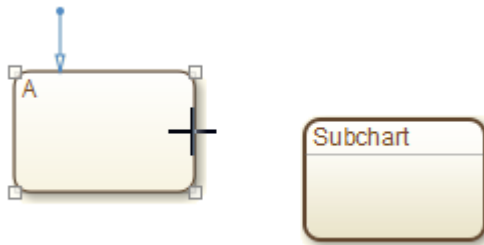
2 Drag the mouse just inside the border of the subchart.

A supertransition appears, extending from the source state into the subchart with its arrowhead penetrating a slit in the subchart.
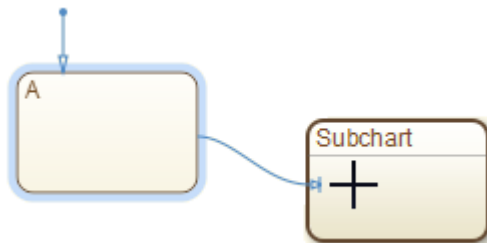
If you are not happy with the initial position of the slit, you can continue to drag the slit around the inside edge of the subchart to the desired location.

3 Double-click the subchart to open it.

The tip of the arrowhead of the supertransition appears highlighted in red, entering the subchart.
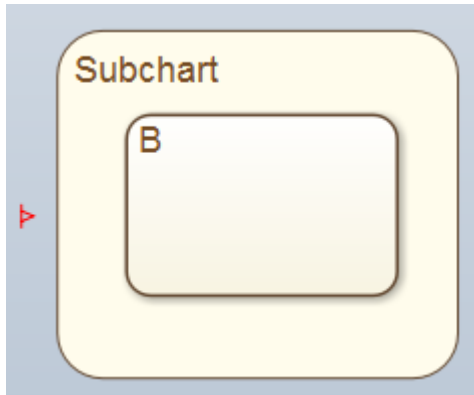
4 Position your cursor over the arrowhead.

The cursor becomes an arrow.

5 Drag the cursor to the desired position in the subchart.
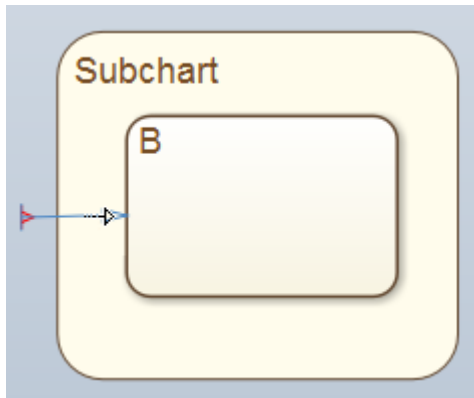
6 Release the cursor.

The supertransition terminates in the desired location.
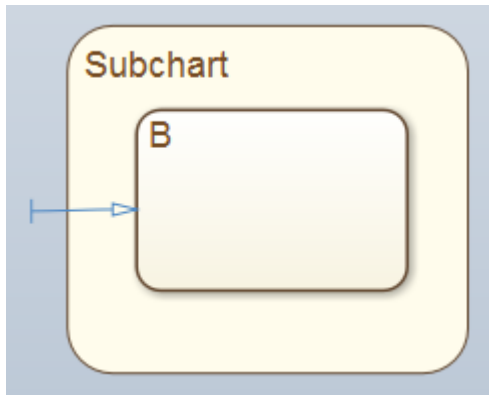
Draw a Supertransition Out of a Subchart

Use the following steps to draw a supertransition out of a subchart.

1   Draw an inner transition segment from the source object anywhere just outside the border of the subchart

A slit appears as shown.
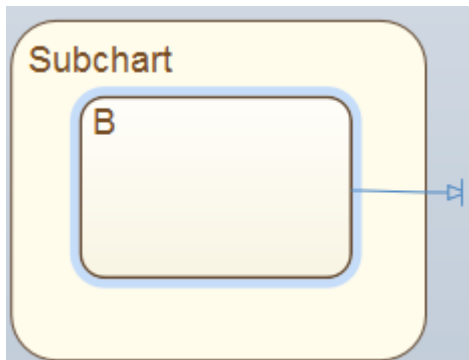
2   Navigate up to the parent object by selecting **View** > **Navigate** > **Up to Parent**.

The tip of the arrowhead of the supertransition appears highlighted in red, exiting the subchart.
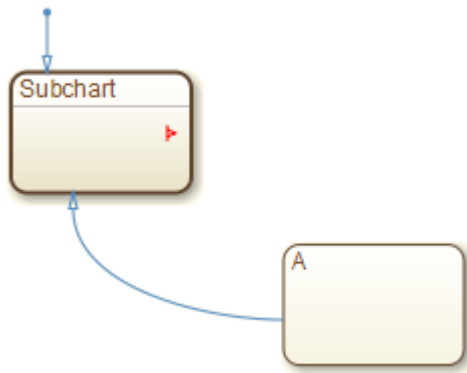
3 Position your cursor over the arrowhead.

   The cursor becomes an arrow.

4 Drag the cursor to the desired position in the chart.

   The parent of the subchart appears.
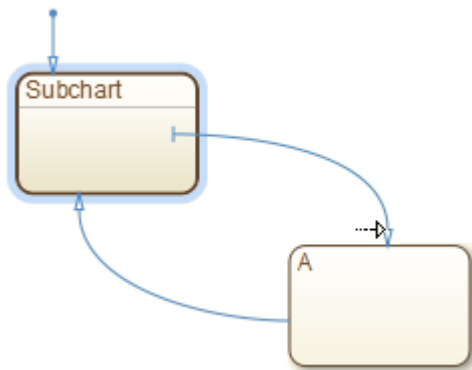
5 Release the cursor to complete the connection.
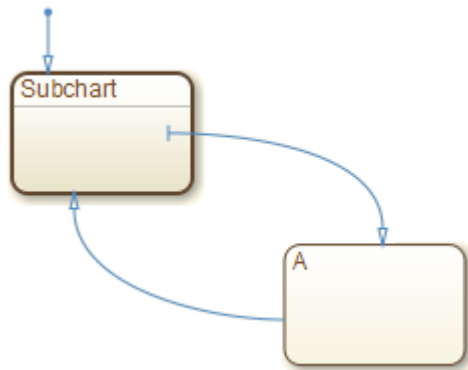
> **Note** If the parent chart is itself a subchart and the terminating object resides at a higher level in the subchart hierarchy, repeat these steps until you reach the desired parent. In this way, you can connect objects separated by any number of layers in the subchart hierarchy.

Label Supertransitions

A supertransition is displayed with multiple resulting transition segments for each layer of containment traversed. For example, if you create a transition between a state outside a subchart and a state inside a subchart of that subchart, you create a supertransition with three segments, each displayed at a different containment level.

You can label any one of the transition segments constituting a supertransition using the same procedure used to label a regular transition (see "Label Transitions" on page 4-19). The resulting label appears on all the segments that constitute the supertransition. Also, if you change the label on any one of the segments, the change appears on all segments.

Reuse Logic Patterns by Defining Graphical Functions

A *graphical function* in a Stateflow chart is a graphical element that helps you reuse control-flow logic and iterative loops. You create graphical functions with flow charts that use connective junctions and transitions. You can call a graphical function in the actions of states and transitions. With graphical functions, you can:

- Create modular, reusable logic that you can call anywhere in your chart.
- Track simulation behavior visually during chart animation.

A graphical function can reside anywhere in a chart, state, or subchart. The location of the function determines the set of states and transitions that can call the function.

- If you want to call the function only within one state or subchart and its substates, put your graphical function in that state or subchart. That function overrides any other functions of the same name in the parents and ancestors of that state or subchart.
- If you want to call the function anywhere in that chart, put your graphical function at the chart level.
- If you want to call the function from any chart in your model, put your graphical function at the chart level and enable exporting of chart-level functions. For more information, see "Export Stateflow Functions for Reuse" on page 8-22.

For example, this graphical function has the name f1. It takes three arguments (a, b, and c) and returns three output values (x, y, and z). The function contains a flow chart that computes three different products of the arguments.
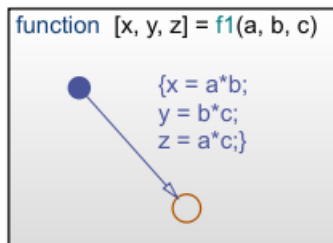
Define a Graphical Function

1   In the object palette, click the graphical function icon . Move your pointer to the location for the new graphical function in your chart.

2 Enter the signature label for the function, as described in "Declare Function Arguments and Return Values" on page 8-18.

3 To program the function, construct a flow chart inside the function box, as described in "Flow Charts in Stateflow" on page 5-2.

Because a graphical function must execute completely when you call it, you cannot use states. Connective junctions and transitions are the only graphical elements that you can use in a graphical function.

4 In the Model Explorer, expand the chart object and select the graphical function. The arguments and return values of the function signature appear as data items that belong to your function. Arguments have the scope `Input`. Return values have the scope `Output`.

5 In the Data properties dialog box for each argument and return value, specify the data properties, as described in "Set Data Properties" on page 9-5.

6 Create any additional data items required by your function. For more information, see "Add Data Through the Model Explorer" on page 9-3.

Your function can access its own data or data belonging to parent states or the chart. The data items in the function can have one of these scopes:

- `Local` — Local data persists from one function call to the next function call. Valid for C charts only.
- `Constant` — Constant data retains its initial value through all function calls.
- `Parameter` — Parameter data retains its initial value through all function calls.
- `Temporary` — Temporary data initializes at the start of every function call. Valid for C charts only.

In charts that use C as the action language, define temporary data when you want to use data that is only valid while a function executes. For example, you can designate a loop counter to have `Temporary` scope if the counter value does not need to persist after the function completes.

In charts that use MATLAB as the action language, you do not need to define temporary function data. If you use an undefined variable, Stateflow creates a temporary variable. The variable is available to the rest of the function.

You can initialize your function data (other than arguments and return values) from the MATLAB workspace. For more information, see "Initialize Data from the MATLAB Base Workspace" on page 9-24.

Declare Function Arguments and Return Values

The function signature label specifies a name for your function and the formal names for its arguments and return values. A signature label has this syntax:

[*return_val1*, *return_val2*,...] = *function_name*(*arg1*, *arg2*,...)

You can specify multiple return values and multiple input arguments. Each return value and input argument can be a scalar, vector, or matrix of values. For functions with only one return value, omit the brackets in the signature label.

You can use the same variable name for both arguments and return values. For example, a function with this signature label uses the variables y1 and y2 as both inputs and outputs:

[y1, y2, y3] = f(y1, u, y2)

If you export this function to C code, y1 and y2 are passed by reference (as pointers), and u is passed by value. Passing inputs by reference reduces the number of times that the generated code copies intermediate data, resulting in more optimal code.

Call Graphical Functions in States and Transitions

You can call graphical functions from the actions of any state or transition. You can also call graphical functions from other functions. If you export a graphical function, you can call it from any chart in the model.

The syntax for a call to a graphical function is the same as the function signature, with actual arguments replacing the formal ones specified in a signature. If the data types of an actual and formal argument differ, a function casts the actual argument to the type of the formal argument.

Tip If the formal arguments of a function signature are scalars, verify that inputs and outputs of function calls follow the rules of scalar expansion. For more information, see "How Scalar Expansion Works for Functions" on page 17-9.

Specify Graphical Function Properties

You can set general properties for your graphical function through its properties dialog box. To open the function properties dialog box, right-click the graphical function box and select **Properties** from the context menu.

Name

Function name. Click the function name link to bring your function to the foreground in its native chart.

Function Inline Option

Controls the inlining of your function in generated code:

- `Auto` — Determines whether to inline your function based on an internal calculation.
- `Inline` — Inlines your function if you do not export it to other charts and it is not part of a recursion. (A recursion exists if your function calls itself directly or indirectly through another function call.)
- `Function` — Does not inline your function.

Label

Signature label for your function. For more information, see "Declare Function Arguments and Return Values" on page 8-18.

Description

Function description. You can enter brief descriptions of functions in the hierarchy.

Document Link

Link to online documentation for the function. You can enter a web URL address or a MATLAB command that displays documentation in a suitable online format, such as an HTML file or text in the MATLAB Command Window. When you click the **Document link** hyperlink, Stateflow displays the documentation.

Manage Large Graphical Functions

You can choose to make your graphical function as large as you want. If your function grows too large, you can hide its contents by right-clicking inside the function box and selecting **Group & Subchart** > **Subchart** from the context menu.
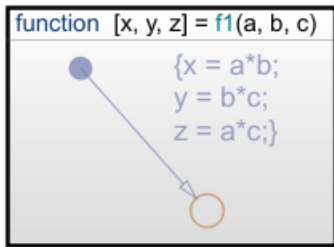
To make the graphical function box opaque, right-click the function and clear the **Content Preview** property from the context menu.
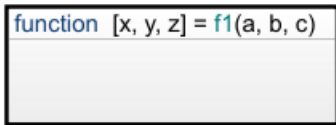
To dedicate the entire chart window to programming your function, access the flow chart in your subcharted graphical function by double-clicking the function box.

Event Broadcasts in Graphical Functions

In a graphical function, do not broadcast events that can cause the active state to change. In a graphical function, the behavior of an event broadcast that causes an exit from the active state is unpredictable.

See Also

More About

- "Flow Charts in Stateflow" on page 5-2

- "Create Flow Charts by Using Pattern Wizard" on page 5-6
- "Reusable Functions in Charts" on page 2-46
- "Export Stateflow Functions for Reuse" on page 8-22
- "Reuse Functions by Using Atomic Boxes" on page 8-35

Export Stateflow Functions for Reuse

When export chart-level functions, you extend the scope of your functions to other blocks in the Simulink model. For an example, see "Share Functions Across Simulink and Stateflow" on page 29-7. You can export these functions:

- Graphical functions
- MATLAB functions
- Truth tables

How to Export Chart-Level Functions

1 Open the chart where your function resides.
2 In the Property Inspector, open the **Advanced** section.
3 Select **Export Chart Level Functions**.
4 If your function resides in a library chart, link that chart to your main model.

When you select **Export Chart Level Functions**, you can call exported functions by using Simulink Caller blocks with dot notation, *chartName.functionName*. To call the exported functions throughout the model from any Stateflow or Simulink Caller block, select **Treat Exported Functions as Globally Visible**. Do not use dot notation to call these functions. You cannot export functions with the same name.

Simulink functions can also be defined directly in the Simulink canvas. For more information, see Simulink Function.

Rules for Exporting Chart-Level Functions

Link library charts to your main model to export chart-level functions from libraries

You must perform this step to export functions from library charts. Otherwise, a simulation error occurs.

Do not export chart-level functions that contain unsupported inputs or outputs

You cannot export a chart-level function when inputs or outputs have any of the following properties:

- Fixed-point data type with word length greater than 32 bits
- Variable size

Do not export Simulink functions

If you try to export Simulink functions, an error appears when you simulate your model. To avoid this behavior, clear the **Export Chart Level Functions** check box in the Chart properties dialog box.

Do not export functions across model reference boundaries

You cannot export functions from a referenced model and call the functions from a parent model.

Export Chart-Level Functions

This example describes how to export functions in library charts to your main model.

1   Create these objects:

- Add a model named `main_model`, with a chart named `modChart`.
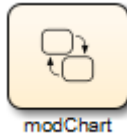
- Add a library model named `lib1`, with a chart named `lib1Chart`.

- Add a library model named `lib2`, with a chart named `lib2Chart`.

2   Create these graphical functions in the library charts:

   • For `lib1Chart`, add this graphical function.
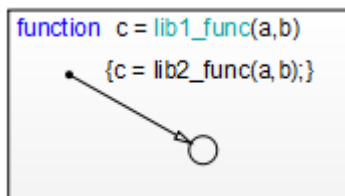
   • For `lib2Chart`, add this graphical function.
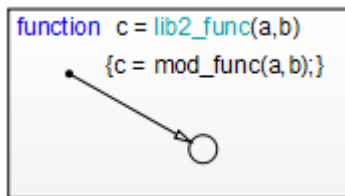
   • In the Model Explorer, for each of the function inputs and outputs, `a`, `b`, and `c`, set these properties:.

      • `Size` to `1`
      • `Complexity` to `Off`
      • `Type` to `double`

3   For `modChart`, add a graphical function and a default transition with a `lib1_func` action.
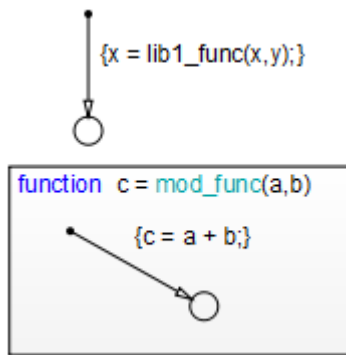
4   For each chart, follow these steps:

    a   In the Model Explorer, for each of the function inputs and outputs, `a`, `b`, and `c`, set:

       • `Size` to `1`

       • `Complexity` to `Off`

       • `Type` to `double`

    b   Open the Chart properties dialog box.

    c   In the Chart properties dialog box, select **Export Chart Level Functions** and **Treat Exported Functions as Globally Visible**.

    d   Click **OK**.

5   Drag `lib1Chart` and `lib2Chart` into `main_model` from `lib1` and `lib2`, respectively. Your main model should look something like this:
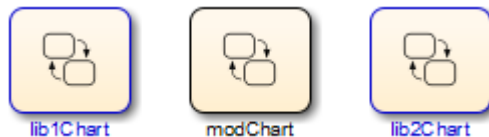
Each chart now defines a graphical function that any chart in `main_model` can call.

6   Open the Model Explorer.

7   In the **Model Hierarchy** pane of the Model Explorer, navigate to `main_model`.

8   Add the data `x` and `y` to the Stateflow machine:

    a   Select **Add > Data**.

    b   In the **Name** column, enter x.

    c   In the **Initial Value** column, enter 0.

    d   Use the default settings for other properties of x.

    e   Select **Add > Data**.

    f   In the **Name** column, enter y.

    g   In the **Initial Value** column, enter 1.

    h   Use the default settings for other properties of y.

This step ensures that input and output data are defined globally to support exported graphical functions.

9   Open the Model Configuration Parameters dialog box.

10   In the Model Configuration Parameters dialog box, go to the **Solver** pane.

11   In the **Solver selection** section, make these changes:

    a   For **Type**, select Fixed-step.

    b   For **Solver**, select Discrete (no continuous states).

12   In the **Solver details** section, make these changes:

    a   For **Fixed-step size**, enter 1.

    b   Click **OK**.

This step ensures that when you simulate your model, a discrete solver is used. For more information, see "Solvers" (Simulink).

What Happens During Simulation

When you simulate the model, these actions take place during each time step.

Phase	The object...	Calls the graphical function...	Which...
1	modChart	lib1_func	Reads two input arguments x and y
2	lib1_func	lib2_func	Passes the two input arguments

Phase	The object...	Calls the graphical function...	Which...
3	lib2_func	mod_func	Adds x and y and assigns the sum to x

How to View the Simulation Results

To view the simulation results, add a scope to your model. Follow these steps:

1 Open the Simulink Library Browser.

2 From the Simulink/Sinks Library, select the Scope block and add it to `main_model`.

3 Open the Model Explorer.

4 In the **Model Hierarchy** pane, navigate to `modChart`.

5 Add the output data `z` to the chart:

    a Select **Add > Data**.

    b In the **Name** column, enter `z`.

    c In the **Scope** column, select `Output`.

    d Use the default settings for other properties.

6 For `modChart`, update the default transition action to read as follows:

`{x = lib1_func(x,y); z = x;}`

7 In the model, connect the outport from `modChart` to the inport of the Scope block.
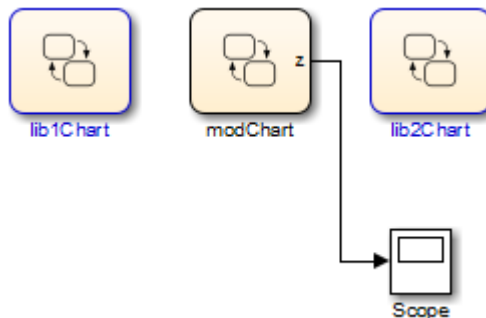
8 Double-click the Scope block to open the display.

9 Start simulation.

10 After the simulation ends, right-click in the scope display and select **Autoscale**.

The results look something like this:
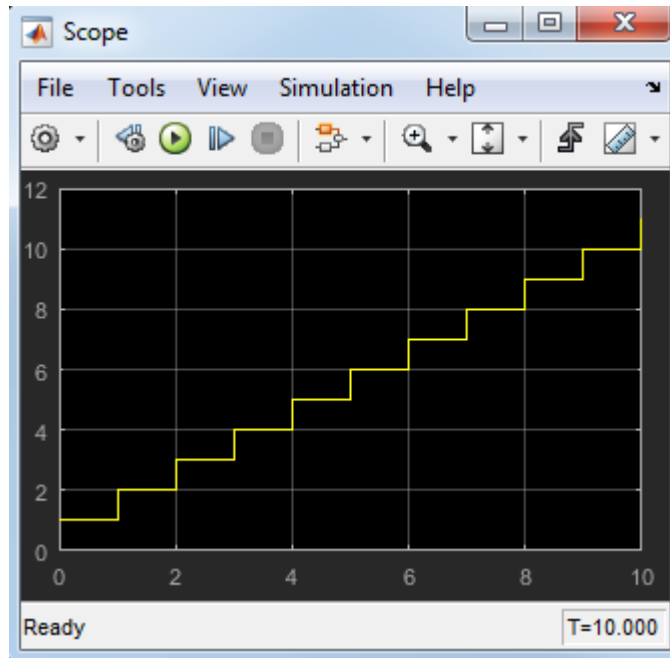
Group Chart Objects by Using Boxes

A box is a graphical object that organizes other objects in your chart, such as functions and states. You can use a box to encapsulate states and functions in a separate namespace. Boxes are supported only in Stateflow charts in Simulink models.

For example, in this chart, the box `Heater` groups together related states `Off` and `On`.
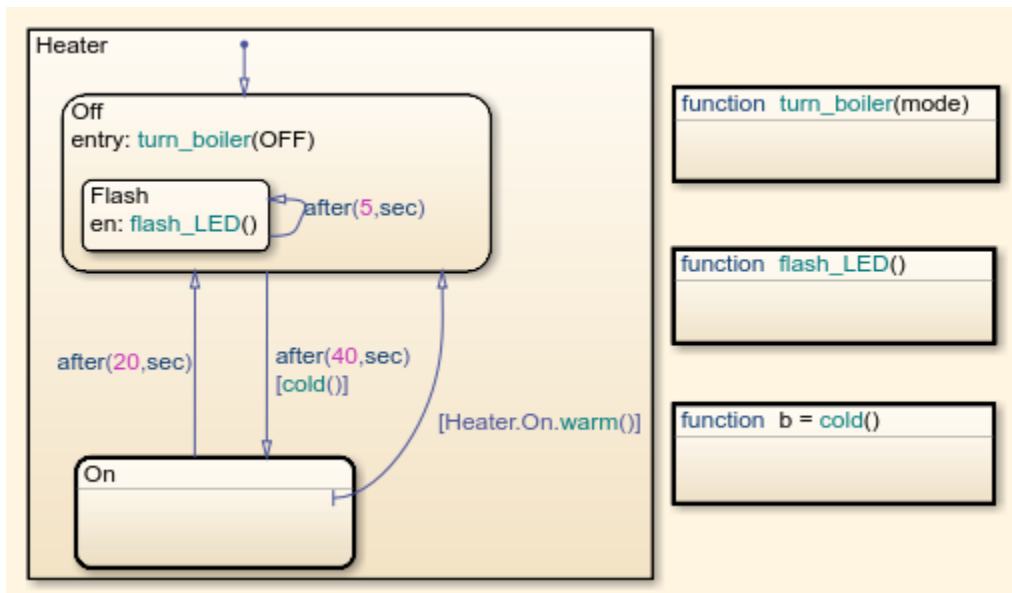
Semantics of Stateflow Boxes

Visibility of Graphical Objects in Boxes

Boxes add a level of hierarchy to Stateflow charts. This property affects visibility of functions and states inside a box to objects that reside outside of the box. If you refer to a box-parented function or state from a location outside of the box, you must include the box name in the path. See "Group Functions Using a Box" on page 8-32.

Activation Order of Parallel States

Boxes affect the implicit activation order of parallel states in a chart. If your chart uses implicit ordering, parallel states within a box wake up before other parallel states that are

lower or to the right in that chart. Within a box, parallel states wake up in top-down, left-right order. See "Group States Using a Box" on page 8-33.

Note To specify activation order explicitly on a state-by-state basis, select **User specified state/transition execution order** in the Chart properties dialog box. This option is selected by default when you create a new chart. For details, see "Explicit Ordering of Parallel States" on page 3-86.

Rules for Using Boxes

When you use a box, these rules apply:

- Include the box name in the path when you use dot notation to refer to a box-parented function or state from a location outside of the box.
- You can move or draw graphical objects inside a box, such as functions and states.
- You can add data to a box so that all the elements in the box can share the same data.
- You can group a box and its contents into a single graphical element. See "Group States" on page 4-6.
- You can subchart a box to hide its elements. See "Encapsulate Modal Logic by Using Subcharts" on page 8-5.
- You cannot define action statements for a box, such as `entry`, `during`, and `exit` actions.
- You cannot define a transition to or from a box. However, you can define a transition to or from a state within a box.

Draw and Edit a Box

Create a Box

You create boxes in your chart by using the box tool shown below.
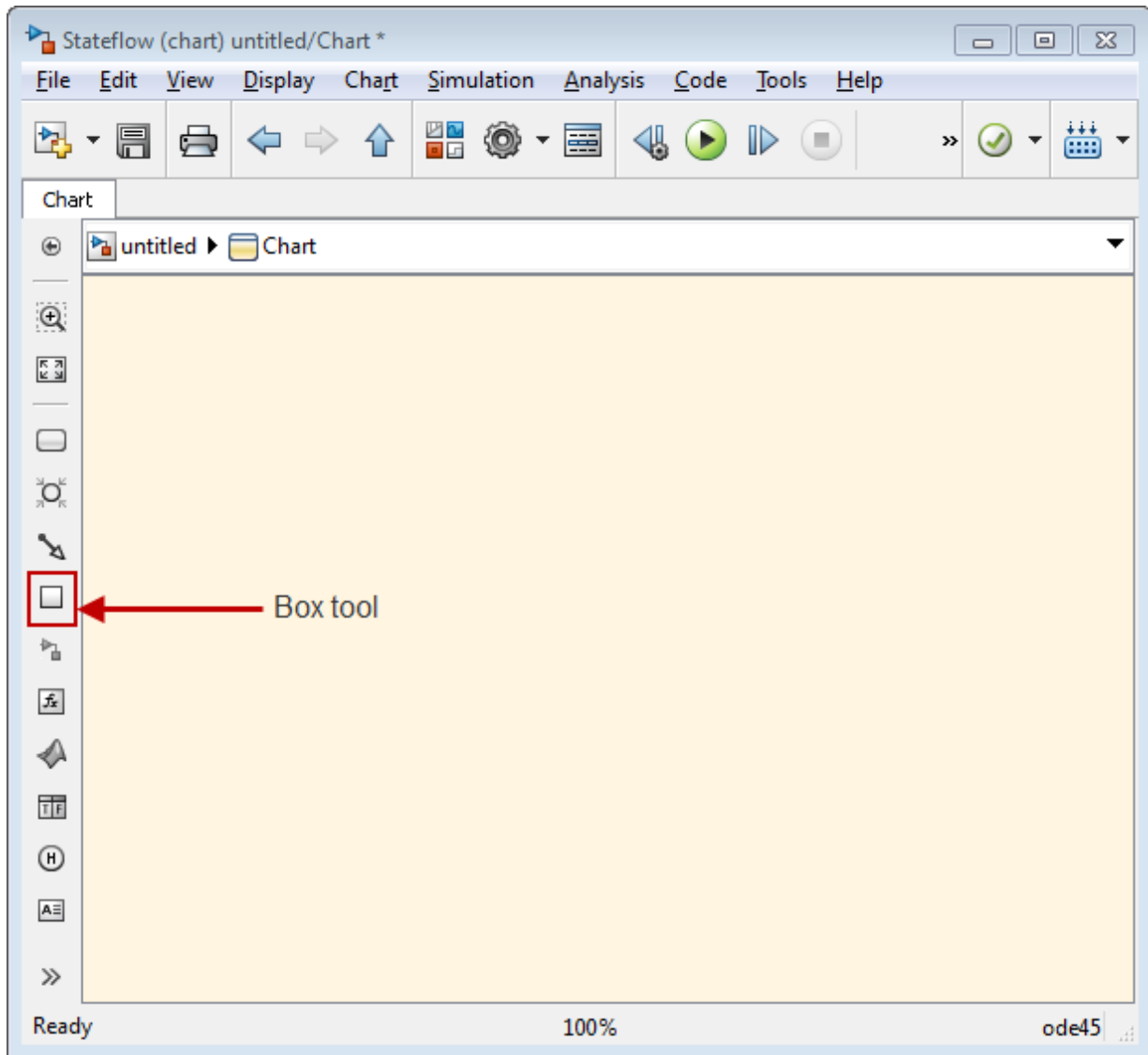
1 Click the Box tool.
2 Move your pointer into the drawing area.
3 Click in any location to create a box.

The new box appears with a question mark (**?**) name in its upper left corner.

4 Click the question mark label.

5 Enter a name for the box and then click outside of the box.

Delete a Box

To delete a box, click to select it and press the **Delete** key.

Examples of Using Boxes

Group Functions Using a Box

This chart shows a box named `Status` that groups together MATLAB functions.
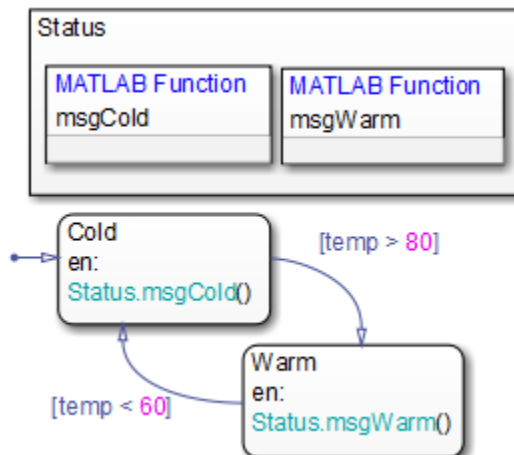
Chart execution takes place as follows:

1 The state `Cold` activates first.

2 Upon entry, the state `Cold` invokes the function `Status.msgCold`.

This function displays a status message that the temperature is cold.

Note Because the MATLAB function resides inside a box, the path of the function call must include the box name `Status`. If you omit this prefix, an error message appears.

3 If the value of the input data `temp` exceeds 80, a transition to the state `Warm` occurs.

4 Upon entry, the state `Warm` invokes the function `Status.msgWarm`.

This function displays a status message that the temperature is warm.

Note Because the MATLAB function resides inside a box, the path of the function call must include the box name `Status`. If you omit this prefix, an error message appears.

---

5 If the value of the input data `temp` drops below 60, a transition to the state `Cold` occurs.

6 Steps 2 through 5 repeat until the simulation ends.

Group States Using a Box

This chart shows a box named `Status` that groups together related states. The chart uses implicit ordering for parallel states, instead of the default explicit mode. (For details, see "Implicit Ordering of Parallel States" on page 3-88.)
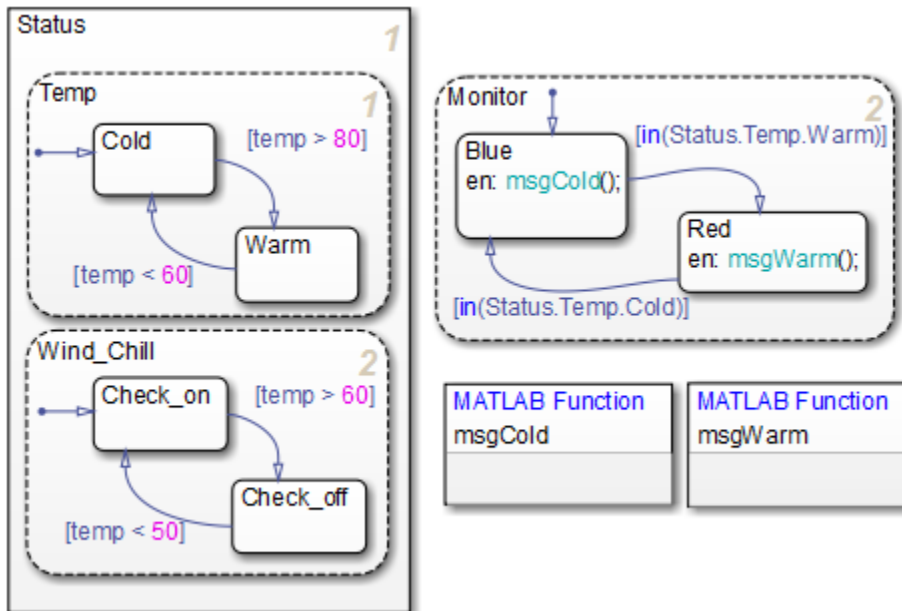
The main ideas of this chart are:

- The state `Temp` wakes up first, followed by the state `Wind_Chill`. Then, the state `Monitor` wakes up.

Note This implicit activation order occurs because `Temp` and `Wind_Chill` reside in a box. If you remove the box, the implicit activation order changes, as shown, to: `Temp`, `Monitor`, `Wind_Chill`.
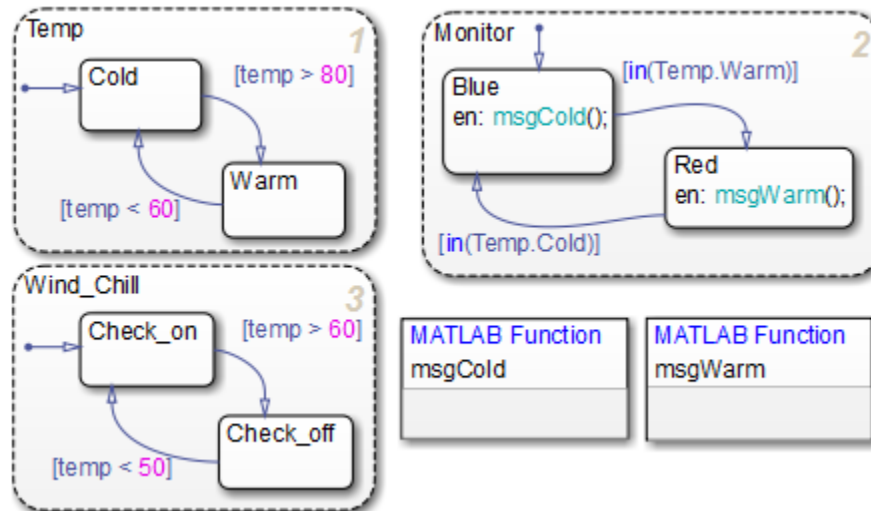
- Based on the input data `temp`, transitions between substates occur in the parallel states `Status.Temp` and `Status.Wind_Chill`.
- When the transition from `Status.Temp.Cold` to `Status.Temp.Warm` occurs, the transition condition `in(Status.Temp.Warm)` becomes true.
- When the transition from `Status.Temp.Warm` to `Status.Temp.Cold` occurs, the transition condition `in(Status.Temp.Cold)` becomes true.

Note Because the substates `Status.Temp.Cold` and `Status.Temp.Warm` reside inside a box, the argument of the `in` operator must include the box name `Status`. If you omit this prefix, an error message appears. For information about the `in` operator, see "Check State Activity by Using the in Operator" on page 12-77.

Reuse Functions by Using Atomic Boxes

An atomic box is a graphical object that helps you encapsulate graphical, truth table, MATLAB, and Simulink functions in a separate namespace. Atomic boxes are supported only in Stateflow charts in Simulink models. Atomic boxes allow for:

- Faster simulation after making small changes to a function in a chart with many states or levels of hierarchy.
- Reuse of the same functions across multiple charts and models.
- Ease of team development for people working on different parts of the same chart.
- Manual inspection of generated code for a specific function in a chart.

An atomic box looks opaque and includes the label **Atomic** in the upper left corner. If you use a linked atomic box from a library, the label **Link** appears in the upper left corner.

Example of an Atomic Box

This example shows how to use a linked atomic box to reuse a graphical function across multiple charts and models.

The function `GetTime` is defined in a chart in the library model `sf_timer_utils_lib`. The graphical function returns the simulation time in C charts where the equivalent MATLAB® function `getSimulationTime` is not available.
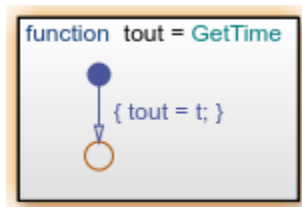
The model `sf_timer_function_calls` consists of two charts with a similar structure. Each chart contains a pair of states (`A` and `B`) and an atomic box (`Time`) linked from the library chart. The entry action in state `A` calls the function `GetTime` and stores its value as `t0`. The condition guarding the transition from `A` to `B` calls the function again and compares its output with the parameter `T`.
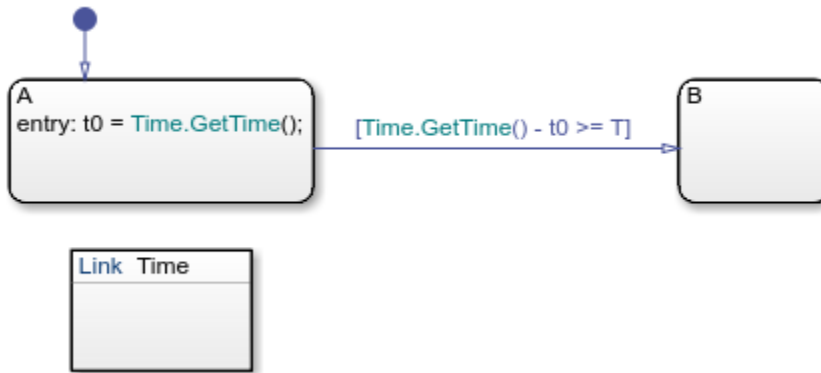
The top model `sf_timer_modelref` reuses the timer function in multiple referenced blocks. Because there are no exported functions, you can use more than one instance of the referenced block in the top model.
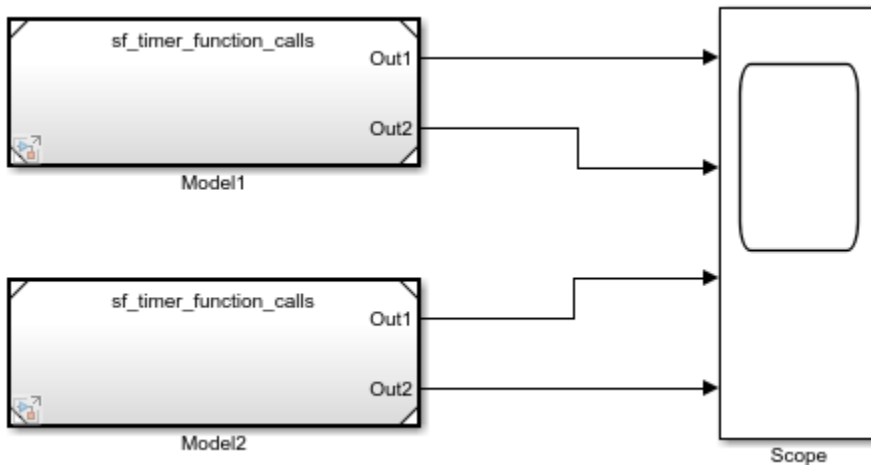
Benefits of Using Atomic Boxes

Atomic boxes combine the functionality of normal boxes and atomic subcharts. Atomic boxes:

- Improve the organization and clarity of complex charts.
- Support usage as library links.
- Support the generation of reusable code.
- Allow mapping of inputs, outputs, parameters, data store memory, and input events.

Atomic boxes contain only functions. They cannot contain states. Adding a state to an atomic box results in a compilation-time error.

To call a function that resides in an atomic box from a location outside the atomic box, use dot notation to specify its full path:

atomic_box_name.function_name

Using the full path for the function call:

- Makes clear the dependency on the function in the linked atomic box.
- Avoids pollution of the global namespace.
- Does not affect the efficiency of generated code.

Create an Atomic Box

You can create an atomic box by converting an existing box or by linking a chart from a library model. After creating the atomic box, update the mapping of variables by right-clicking the atomic box and selecting **Subchart Mappings**. For more information, see "Map Variables for Atomic Subcharts and Boxes" on page 15-9.

Convert a Normal Box to an Atomic Box

To create a container for your functions that allows for faster debugging and code generation workflows, convert an existing box into an atomic box. In your chart, right-click a normal box and select **Group & Subchart > Atomic Subchart**. The label **Atomic** appears in the upper left corner of the box.

The conversion process gives the atomic box its own copy of every data object that the box accesses in the chart. Local data is copied as data store memory. The scope of other data, including input and output data, does not change.

Note If a box contains any states or messages, you cannot convert it to an atomic box.

Link an Atomic Box from a Library

To create a collection of functions for reuse across multiple charts and models, create a link from a library model. Copy a chart in a library model and paste it to a chart in another model. If the library chart contains only functions and no states, it appears as a linked atomic box with the label **Link** in the upper left corner.

This modeling method minimizes maintenance of reusable functions. When you modify the atomic box in the library, your changes propagate to the links in all charts and models.

If the library chart contains any states, then it appears as a linked atomic subchart in the chart. For more information, see "Create Reusable Subcomponents by Using Atomic Subcharts" on page 15-2.

Convert an Atomic Box to a Normal Box

Converting an atomic box back to a normal box removes all of its variable mappings by merging subchart-parented data objects with the chart-parented data to which they map.

1   If the atomic box is a library link, right-click the atomic box and select **Library Link** > **Disable Link**.

2   To convert an atomic box to a subcharted box, right-click the atomic box and clear the **Group & Subchart** > **Atomic Subchart** check box.

3   To convert the subcharted box back to a normal box, right-click the subchart and clear the **Group & Subchart** > **Subchart** check box.

4   If necessary, rearrange graphical objects in your chart.

You cannot convert an atomic box to a normal box if:

*   The atomic box maps a parameter to an expression other than a single variable name. For example, mapping a parameter `data1` to one of these expressions prevents the conversion of an atomic box to a normal box:

    *   `3`
    *   `data2(3)`
    *   `data2 + 3`

*   Both of these conditions are true:

    *   The atomic box contains MATLAB functions or truth table functions that use MATLAB as the action language.

- The atomic box does not map each variable to a variable of the same name in the main chart.

When to Use Atomic Boxes

Debug Functions Incrementally

Suppose that you want to test a sequence of changes to a library of functions. The functions are part of a chart that contains many states or several levels of hierarchy, so recompiling the entire chart can take a long time. If you define the functions in an atomic box, recompilation occurs for only the box and not for the entire chart. For more information, see "Reduce the Compilation Time of a Chart" on page 15-43.

Reuse Functions

Suppose that you have a set of functions for use in multiple charts and models. The functions reside in the library model to enable easier configuration management. To use the functions in another model, you can either:

- Configure the library chart to export functions and create a link to the library chart in the model.
- Link the library chart as an atomic box in each chart of the model.

Models that use these functions can appear as referenced blocks in a top model. When the functions are exported, you can use only one instance of that referenced block for each top model. For more information, see "Model Reference Requirements and Limitations" (Simulink).

With atomic boxes, you can avoid this limitation. Because there are no exported functions in the charts, you can use more than one instance of the referenced block in the top model.

Develop Charts Used by Multiple People

Suppose that multiple people are working on different parts of a chart. If you store each library of functions in a linked atomic box, different people can work on different libraries without affecting the other parts of the chart. For more information, see "Divide a Chart into Separate Units" on page 15-45.

Inspect Generated Code

Suppose that you want to inspect code generated by Simulink Coder or Embedded Coder manually for a specific function. You can specify that the code for an atomic box appears

in a separate file to avoid searching through unrelated code. For more information, see "Generate Reusable Code for Unit Testing" on page 15-48.

See Also

More About

- "Group Chart Objects by Using Boxes" on page 8-29
- "Create Reusable Subcomponents by Using Atomic Subcharts" on page 15-2
- "Map Variables for Atomic Subcharts and Boxes" on page 15-9

Add Descriptive Comments in a Chart

Create Notes

You can enter comments or notes in any location on the chart.

1   Double-click in the desired location of the chart and start typing your comments.

2   Press the **Enter** key to start a new line.

3   After you finish typing, click outside the note text.

Change Note Properties

You can use the Note properties dialog box to edit note properties.

You can specify the layout of the note, including:

- Borders
- Text alignment and word wrap
- Text color and background color
- Margins between the text and the borders of the note

Change Note Font and Color

To change font and color for your chart notes, follow the procedures described in the section "Specify Colors and Fonts in a Chart" on page 4-32.

You can also change your note text to bold or italic:

1   Right-click the note text and select **Font Style**.

2   In the submenu, select **Bold** or **Italic**.

TeX Instructions

In your notes, you can embed a subset of TeX commands to produce special characters. For example, you can embed Greek letters and mathematical symbols.

1   Right-click the text of a note and select **Enable TeX Commands**.

2 Click the note text.

3 Replace the existing note text with the following expression.

`\it{\omega_N = e^{(-2\pii)/N}}`

4 Click outside the note.

The note in your chart looks something like this:

$$\omega_N = e^{(-2\pi i)/N}$$

9

Define Data

Add Stateflow Data

When you want to store values that are visible at a specific level of the Stateflow hierarchy, add data to your chart.

Data defined in a Stateflow chart is visible by multiple Stateflow objects in the chart, including states, transitions, MATLAB functions, and truth tables. To determine what data is used in a state or transition, right-click the state or transition and select **Explore**. A context menu lists the names and scopes of all resolved symbols in the state or transition. Selecting a symbol from the context menu displays its properties in the Model Explorer. Selecting an output event from the context menu opens the Simulink subsystem or Stateflow chart associated with the event.

Note Stateflow data is not available to Simulink functions within a Stateflow chart.

You can add data through the Symbols window, the **Chart** menu in the Stateflow Editor, or the Model Explorer.

Add Data Through the Symbols Window

1 To open the Symbols window, select **View** > **Symbols**.

2 Click the **Create Data** icon ![icon].

3 In the row for the new data, under **TYPE**, click the icon and choose:

- `Input Data`
- `Local Data`
- `Output Data`
- `Constant`
- `Data Store Memory`
- `Parameter`
- `Temporary`

4 Edit the name of the data.

5 For input and output data, click the **PORT** field and choose a port number.

6   To specify properties for data, open the Property Inspector. In the Symbols window, right-click the row for the symbol and select **Explore**. For more information, see "Stateflow Data Properties" on page 9-5.

Add Data by Using the Stateflow Editor Menu

1   In a Stateflow chart in a Simulink model, select the menu option corresponding to the scope of the data that you want to add.

Scope	Menu Option
Input	**Chart > Add Inputs & Outputs > Data Input From Simulink**
Output	**Chart > Add Inputs & Outputs > Data Output To Simulink**
Local	**Chart > Add Other Elements > Local Data**
Constant	**Chart > Add Other Elements > Constant**
Parameter	**Chart > Add Other Elements > Parameter**
Data Store Memory	**Chart > Add Other Elements > Data Store Memory**

2   In the Data dialog box, specify data properties. For more information, see "Stateflow Data Properties" on page 9-5.

Add Data Through the Model Explorer

To add machine or state-parented data to Stateflow charts in Simulink models, use the Model Explorer:

1   In the Stateflow Editor, select **View > Model Explorer**.

2   In the **Model Hierarchy** pane, select the object in the Stateflow hierarchy where you want to make the new data visible. The object that you select becomes the parent of the new data.

3   In the Model Explorer menu, select **Add > Data**. The new data with a default definition appears in the **Contents** pane of the Model Explorer.

4   In the **Data** pane, specify the properties of the data. For more information, see "Stateflow Data Properties" on page 9-5.

Best Practices for Using Data in Charts

Avoid inheriting output data properties from Simulink blocks

Stateflow output data should not inherit properties from output signals, because the values back propagate from Simulink blocks and can be unpredictable.

Restrict use of machine-parented data

Avoid using machine-parented data. The presence of machine-parented data in a model prevents reuse of generated code and other code optimizations. This type of data is also incompatible with many Simulink and Stateflow features.

For example, the following features do not support machine-parented data:

- Enumerated data
- Simulink functions
- Chart operating point
- Implicit change events
- Detection of unused data
- Model referencing (see "Model Reference Requirements and Limitations" (Simulink))
- Analysis by Simulink Design Verifier™ software
- Code generation by Simulink PLC Coder™ software
- Parameters binding to a `Simulink.Parameter` object in the base workspace

To make Stateflow data accessible to other charts and blocks in a model, use data store memory. For details, see "Access Data Store Memory from a Chart" on page 9-28.

See Also

More About

Set Data Properties

When you create Stateflow charts in Simulink, you can specify data properties in either the Property Inspector or the Model Explorer.

- Property Inspector

  1   Open the Symbols window by selecting **View** > **Symbols**.

  2   Open the Property Inspector by selecting **View** > **Property Inspector**.

  3   In the Symbols window, select the data object.

  4   In the Property Inspector, edit the data properties.

- Model Explorer

  1   Open the Model Explorer by selecting **View** > **Model Explorer**.

  2   In the **Contents** pane, select the data object.

  3   In the **Data** pane, edit the data properties.

Properties vary according to the scope and type of the data object. For many data properties, you can enter expressions or parameter values. Using parameters to set properties for many data objects simplifies maintenance of your model because you can update multiple properties by changing a single parameter.

Stateflow Data Properties

You can set these data properties in:

- The main and **Advanced** sections of the Property Inspector.
- The **General** tab of the Model Explorer.

Name

Name of the data object. For more information, see "Rules for Naming Stateflow Objects" on page 2-4.

Scope

Location where data resides in memory, relative to its parent.

Setting	Description
Local	Data defined in the current chart only.
Constant	Read-only constant value that is visible to the parent Stateflow object and its children.
Parameter	Constant whose value is defined in the MATLAB base workspace or derived from a Simulink block parameter that you define and initialize in the parent masked subsystem. The Stateflow data object must have the same name as the MATLAB variable or the Simulink parameter. For more information, see "Share Parameters with Simulink and the MATLAB Workspace" on page 9-26.
Input	Input argument to a function if the parent is a graphical function, truth table, or MATLAB function. Otherwise, the Simulink model provides the data to the chart through an input port on the Stateflow block. For more information, see "Share Input and Output Data with Simulink" on page 9-23.
Output	Return value of a function if the parent is a graphical function, truth table, or MATLAB function. Otherwise, the chart provides the data to the Simulink model through an output port on the Stateflow block. For more information, see "Share Input and Output Data with Simulink" on page 9-23.
Data Store Memory	Data object that binds to a Simulink data store, which is a signal that functions like a global variable. All blocks in a model can access that signal. This binding allows the chart to read and write to the Simulink data store, sharing global data with the model. The Stateflow object must have the same name as the Simulink data store. For more information, see "Access Data Store Memory from a Chart" on page 9-28.
Temporary	Data that persists during only the execution of a function. For C charts, you can define temporary data only for a graphical function, truth table, or MATLAB function.
Exported	Data from the Simulink model that is made available to external code defined in the Stateflow hierarchy. You can define exported data only for a Stateflow machine.
Imported	Data parented by the Simulink model that you define in external code embedded in the Stateflow machine. You can define imported data only for a Stateflow machine.

Port

Index of the port associated with the data object. This property applies only to input and output data. See "Share Input and Output Data with Simulink" on page 9-23.

Update Method

Specifies whether a variable updates in discrete or continuous time. This property applies only when the chart is configured for continuous-time simulation. See "Continuous-Time Modeling in Stateflow" on page 21-2.

Data Must Resolve to Signal Object

Specifies that output or local data explicitly inherits properties from `Simulink.Signal` objects of the same name in the MATLAB base workspace or the Simulink model workspace. The data can inherit these properties:

- Size
- Complexity
- Type
- Unit
- Minimum value
- Maximum value
- Initial value
- Storage class
- Sampling mode (for Truth Table block output data)

This option is available only when you set the model configuration parameter **Signal resolution** to a value other than `None`. For more information, see "Resolve Data Properties from Simulink Signal Objects" on page 9-55.

Size

Size of the data object. The size can be a scalar value or a MATLAB vector of values. To specify a scalar, set the **Size** property to 1 or leave it blank. To specify a MATLAB vector, use a multidimensional array. The number of dimensions equals the length of the vector and the size of each dimension that corresponds to the value of each vector element.

The scope of the data object determines what sizes you can specify. Stateflow data store memory inherits all its properties, including its size, from the Simulink data store to

which it is bound. For all other scopes, size can be scalar, vector, or a matrix of n-dimensions.

For more information, see "Specify Size of Stateflow Data" on page 9-40.

Variable Size

Specifies that the data object changes dimensions during simulation. This option is available for input and output data only when you enable the chart property **Support variable-size arrays**. For more information, see "Declare Variable-Size Data in Stateflow Charts" on page 18-2.

Complexity

Specifies whether the data object accepts complex values.

Setting	Description
Off	Data object does not accept complex values.
On	Data object accepts complex values.
Inherited	Data object inherits the complexity setting from a Simulink block.

The default value is Off. For more information, see "Complex Data in Stateflow Charts" on page 23-2.

First Index

Index of the first element of the data array. The first index can be any non-negative integer. The default value is 0. This property is available only for C charts.

Type

Type of data object. To specify the data type:

• From the **Type** drop-down list, select a built-in type.

• In the **Type** field, enter an expression that evaluates to a data type.

• Use the Data Type Assistant to specify a data **Mode**, and then specify the data type based on that mode. To display the Data Type Assistant, in the Model Explorer, click the **Show data type assistant** button ⬚⬚＞＞⬚⬚. The Data Type Assistant is available only in the Model Explorer.

For more information, see "Specify Type of Stateflow Data" on page 9-33.

Note If you enter an expression for a fixed-point data type, you must specify scaling explicitly. For example, you cannot enter an incomplete specification such as `fixdt(1,16)` in the **Type** field. If you do not specify scaling explicitly, an error appears when you try to simulate your model.

Lock Data Type Against Fixed-Point Tools

Prevents replacement of the current fixed-point type with an autoscaled type chosen by the "Fixed-Point Tool" (Fixed-Point Designer). For more information, see "Autoscaling Using the Fixed-Point Tool" (Fixed-Point Designer).

Unit (e.g., m, m/s^2, N*m)

Specifies physical units for input and output data. For more information, see "Specify Units for Stateflow Data" on page 24-41.

Initial Value

Initial value of the data object. The options for initializing values depend on the scope of the data object.

Scope	Specify for Initial Value
Local	Expression or parameter defined in the Stateflow hierarchy, MATLAB base workspace, or Simulink masked subsystem
Constant	Constant value or expression. The expression is evaluated when you update the chart. The resulting value is used as a constant for running the chart.
Parameter	You cannot enter a value. The chart inherits the initial value from the parameter.
Input	You cannot enter a value. The chart inherits the initial value from the Simulink input signal on the designated port.
Output	Expression or parameter defined in the Stateflow hierarchy, MATLAB base workspace, or Simulink masked subsystem.
Data Store Memory	You cannot enter a value. The chart inherits the initial value from the Simulink data store to which it resolves.

If you do not specify a value, the default value for numeric data is 0. For enumerated data, the default value is the first one listed in the `enumeration` section of the definition,

unless you specify otherwise in the `methods` section of the definition. For more information, see "Enter Expressions and Parameters for Data Properties" on page 9-20.

Allow Initial Value to Resolve to a Parameter

Specifies that local, constant, or output data inherits its initial value from a Simulink parameter in the MATLAB base workspace. For more information, see "Initialize Data from the MATLAB Base Workspace" on page 9-24.

Limit Range Properties

Range of acceptable values for this data object. Stateflow charts use this range to validate the data object during simulation.

- **Minimum** — The smallest value allowed for the data item during simulation. You can enter an expression or parameter that evaluates to a numeric scalar value.
- **Maximum** — The largest value allowed for the data item during simulation. You can enter an expression or parameter that evaluates to a numeric scalar value.

The smallest value that you can set for **Minimum** is `-inf`. The largest value that you can set for **Maximum** is `inf`. For more information, see "Enter Expressions and Parameters for Data Properties" on page 9-20.

Note A Simulink model uses the **Limit range** properties to calculate best-precision scaling for fixed-point data types. Before you select **Calculate Best-Precision Scaling**, specify a minimum or maximum value. For more information, see "Calculate Best-Precision Scaling" on page 9-14.

Add to Watch Window

Enables watching the data values in the Stateflow Breakpoints and Watch window. For more information, see "Watch Stateflow Data Values" on page 32-36.

Fixed-Point Data Properties

In the Model Explorer, when you set the Data Type Assistant **Mode** to `Fixed point`, the Data Type Assistant displays fields for specifying additional information about your fixed-point data.

Signedness

Specifies whether the fixed-point data is `Signed` or `Unsigned`. Signed data can represent positive and negative values. Unsigned data represents positive values only. The default setting is `Signed`.
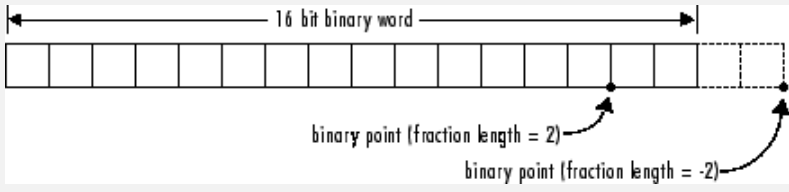
Word Length

Specifies the bit size of the word that holds the quantized integer. Large word sizes represent large values with greater precision than small word sizes. The default value is 16.

- Word length can be any integer from 0 through 128 for chart-level data of these scopes:

  - `Input`
  - `Output`
  - `Parameter`
  - `Data Store Memory`

- For other Stateflow data, word length can be any integer from 0 through 32.

Scaling

Specifies the method for scaling your fixed-point data to avoid overflow conditions and minimize quantization errors. The default method is `Binary point` scaling.

Setting	Description
Binary point	If you select this mode, the Data Type Assistant displays the **Fraction length** field, which specifies the binary point location. **Fraction length** can be any integer. The default value is 0. A positive integer moves the binary point left of the rightmost bit by that amount. A negative integer moves the binary point farther right of the rightmost bit.
Slope and bias	If you select this mode, the Data Type Assistant displays fields for entering the **Slope** and **Bias** for the fixed-point encoding scheme. **Slope** can be any positive real number. The default value is 1.0. **Bias** can be any real number. The default value is 0.0. You can enter slope and bias as expressions that contain parameters you define in the MATLAB base workspace.

Whenever possible, use binary-point scaling to simplify the implementation of fixed-point data in generated code. Operations with fixed-point data that use binary-point scaling are performed with simple bit shifts and eliminate expensive code implementations required for separate slope and bias values. For more information about fixed-point scaling, see "Scaling" (Fixed-Point Designer).

Data Type Override

Specifies whether to inherit the data type override setting of the Fixed-Point Tool that applies to this model. If the data does not inherit the model-wide setting, the specified data type applies. For more information about the Fixed-Point Tool, see `fxptdlg`.

Calculate Best-Precision Scaling

Specifies whether to calculate the best-precision values for `Binary point` and `Slope and bias` scaling, based on the values in the **Minimum** and **Maximum** fields in the **Limit range** section.

To calculate best-precision scaling values:

1 Specify **Limit range** properties.

2 Click **Calculate Best-Precision Scaling**.

Simulink software calculates the scaling values and displays them in the **Fraction length** field or the **Slope** and **Bias** fields. For more information, see "Constant Scaling for Best Precision" (Fixed-Point Designer).

Note The **Limit range** properties do not apply to `Constant` and `Parameter` scopes. For `Constant`, Simulink software calculates the scaling values based on the **Initial value** setting. The software cannot calculate best-precision scaling for data of `Parameter` scope.

Show Fixed-Point Details

Displays information about the fixed-point data type that is defined in the Data Type Assistant:

- `Minimum` and `Maximum` show the same values that appear in the corresponding **Minimum** and **Maximum** fields in the **Limit range** section.
- `Representable minimum`, `Representable maximum`, and `Precision` show the minimum value, maximum value, and precision that the fixed-point data type can represent.

If the value of a field cannot be determined without first compiling the model, the **Fixed-point details** subpane shows the value as `Unknown`.

Data data

General | Logging | Description

Name: data

Scope: Local ▾

☐ Data must resolve to signal object

Size:

Complexity: Off ▾

Type: fixdt(1,16,0) ▾ | << |

Data Type Assistant

Mode: Fixed point ▾ | Signedness: Signed ▾ | Word length: 16

Scaling: Binary point ▾ | Fraction length: 0

Data type override: Inherit ▾ | Calculate Best-Precision Scaling

⊟ Fixed-point details

Representable maximum: 32767
Maximum: 10000
Minimum: -20
Representable minimum: -32768

Precision: 1 | Refresh Details

☐ Lock data type against Fixed-Point tools

Initial value: Expression ▾ |

Limit range

Minimum: -20 | Maximum: 10000

Add to Watch Window

| OK | Cancel | Help | Apply |

The values displayed by the **Fixed-point details** subpane *do not* automatically update if you change the values that define the fixed-point data type. To update the values shown in the **Fixed-point details** subpane, click **Refresh Details**.

Clicking **Refresh Details** does not modify the model. It changes only the display. To apply the displayed values, click **Apply** or **OK**.

The **Fixed-point details** subpane indicates any error resulting from the fixed-point data type specification. For example, this figure shows two errors.

The row labeled `Maximum` indicates that the value specified in the **Maximum** field of the **Limit range** section is not representable by the fixed-point data type. To correct the

error, make one of these modifications so the fixed-point data type can represent the maximum value:

- Decrease the value in the **Maximum** field of the **Limit range** section.
- Increase **Word length**.
- Decrease **Fraction length**.

The row labeled `Minimum` shows the error `Cannot evaluate` because evaluating the expression `MySymbol`, specified in the **Minimum** field of the **Limit range** section, does not return a numeric value. When an expression does not evaluate successfully, the **Fixed-point details** subpane shows the unevaluated expression (truncating to 10 characters as needed) in place of the unavailable value. To correct this error, define `MySymbol` in the base workspace to provide a numeric value. If you click **Refresh Details**, the error indicator and description are removed and the value of `MySymbol` appears in place of the unevaluated text.

Logging Properties

You can set logging properties for data in:

- The **Logging** section of the Property Inspector.
- The **Logging** tab of the Model Explorer.

Log Signal Data

Saves the data value to the MATLAB base workspace during simulation. For more information, see "Log Simulation Output for States and Data" on page 32-51.

Test Point

Designates the data as a test point. A test point is a signal that you can observe in a Floating Scope block in a model. Data objects can be test points if:

- Scope is `Local`.
- Parent is not a Stateflow machine.
- Data type is not `ml`.

For more information, see "Monitor Test Points in Stateflow Charts" on page 32-46.

Logging Name

Specifies the name associated with logged signal data. Simulink software uses the signal name as its logging name by default. To specify a custom logging name, select `Custom` from the list box and enter the new name in the adjacent edit field.

Limit Data Points to Last

Limits the amount of data logged to the most recent samples.

Decimation

Limits the amount of data logged by skipping samples. For example, a decimation factor of 2 saves every other sample.

Additional Properties

You can set additional data properties in:

- The **Info** tab of the Property Inspector.
- The **Description** tab of the Model Explorer.

Save Final Value to Base Workspace

Assigns the value of the data item to a variable of the same name in the MATLAB base workspace at the end of simulation. This option is available only in the Model Explorer for C charts. For more information, see "Model Workspaces" (Simulink).

Units

Units of measurement associated with the data object. The unit in this field resides with the data object in the Stateflow hierarchy. This property is available only in the Model Explorer for C charts.

Description

Description of the data object. You can enter brief descriptions of data in the hierarchy.

Document Link

Link to online documentation for the data object. You can enter a web URL address or a MATLAB command that displays documentation in a suitable online format, such as an

HTML file or text in the MATLAB Command Window. When you click the **Document link** hyperlink, Stateflow evaluates the link and displays the documentation.

Enter Expressions and Parameters for Data Properties

In the Property Inspector and Model Explorer, you can enter expressions as values for these properties:

- "Size" on page 9-7
- "Type" on page 9-8
- "Initial Value" on page 9-9
- Minimum and Maximum (see "Limit Range Properties" on page 9-10)
- "Fixed-Point Data Properties" on page 9-10

Expressions can contain a mix of parameters, constants, arithmetic operators, and calls to MATLAB functions.

Default Data Property Values

When you leave an expression or parameter field blank, Stateflow assumes a default value.

Field	Default
Initial value	0.0
Maximum	`inf`
Minimum	`–inf`
Word length	16
Slope	1.0
Bias	0.0
Binary point	0
First index	0
Size	`–1` (inherited), for inputs, parameters, and function outputs `1` (scalar), for other data objects

Use Parameters in Expressions

You can include parameters in expressions. A parameter is a constant value that you can:

- Define in the MATLAB base workspace.
- Derive from a Simulink block parameter that you define and initialize in the parent masked subsystem.

You can mix both types of parameters in an expression. For more information, see "Share Parameters with Simulink and the MATLAB Workspace" on page 9-26.

Use Constants in Expressions

For expressions in the Data properties dialog box, you can use numeric constants of the appropriate type and size. Do not use Stateflow constants in these expressions.

Use Arithmetic Operators in Expressions

In the Data properties dialog box, you can use these arithmetic operators in expressions:

- +
- –
- *
- /

Call Functions and Operators in Expressions

In fields that accept expressions, you can call functions that return property values of other variables defined in the Stateflow hierarchy, MATLAB base workspace, or Simulink masked subsystem. For example, these functions can return appropriate values for specified fields in the Data properties dialog box.

Function	Returns	For Field
Stateflow operator `type` on page 12-20	Type of input data	Data type

Function	Returns	For Field
Simulink function `fixdt`	`Simulink.NumericType` object that describes a fixed-point or floating-point data type	Data type
MATLAB function `min`	Smallest element or elements of input array	Minimum
MATLAB function `max`	Largest element or elements of input array	Maximum

See Also

More About

- "Add Stateflow Data" on page 9-2
- "Rules for Naming Stateflow Objects" on page 2-4
- "Specify Type of Stateflow Data" on page 9-33
- "Identify Data by Using Dot Notation" on page 9-50

Share Data with Simulink and the MATLAB Workspace

Stateflow charts interface with the other blocks in a Simulink model by:

- Sharing data through input and output connections.
- Importing initial data values from the MATLAB base workspace.
- Saving final data values to the MATLAB base workspace.

Charts also can access Simulink parameters and data stores. For more information, see "Share Parameters with Simulink and the MATLAB Workspace" on page 9-26 and "Access Data Store Memory from a Chart" on page 9-28.

Share Input and Output Data with Simulink

Data flows from Simulink into a Stateflow chart through input ports. Data flows from a Stateflow chart into Simulink through output ports.

To define input or output data in a chart:

1 Add a data object to the chart, as described in "Add Stateflow Data" on page 9-2.

2 Set the **Scope** property for the data object.

- To define input data, set **Scope** to `Input Data`. An input port appears on the left side of the chart block.
- To define output data, set **Scope** to `Output Data`. An output port appears on the right side of the chart block.

By default, **Port** values appear in the order in which you add data objects. You can change these assignments by modifying the **Port** property of the data. When you change the **Port** property for an input or output data object, the **Port** values for the remaining input or output data objects automatically renumber.

3 Set the data type of the data object, as described in "Specify Type of Stateflow Data" on page 9-33.

4 Set the size of the data object, as described in "Specify Size of Stateflow Data" on page 9-40.

Note You cannot set the type or size of Stateflow input data to accept frame-based data from Simulink.

Initialize Data from the MATLAB Base Workspace

You can import the initial value of a data symbol by defining it in the MATLAB base workspace and in the Stateflow hierarchy.

1 Define and initialize a variable in the MATLAB base workspace.

2 In the Stateflow hierarchy, define a data object with the same name as the MATLAB variable.

3 Select the **Allow initial value to resolve to a parameter** property for the data object.

When the simulation starts, data resolution occurs. During this process, the Stateflow data object gets its initial value from the associated MATLAB variable.

One-dimensional Stateflow arrays are compatible with MATLAB row and column vectors of the same size. For example, a Stateflow vector of size 5 is compatible with a MATLAB row vector of size `[1,5]` or column vector of size `[5,1]`. Each element of the Stateflow array initializes to the same value as the corresponding element of the array in the MATLAB base workspace.

The time of initialization depends on the data parent and scope of the Stateflow data object.

Data Parent	Scope	Initialization Time
Machine	Local, Exported	Start of simulation
	Imported	Not applicable
Chart	Input	Not applicable
	Output, Local	Start of simulation or when chart reinitializes as part of an enabled Simulink subsystem
State with History Junction	Local	Start of simulation or when chart reinitializes as part of an enabled Simulink subsystem
State without History Junction	Local	State activation
Function (graphical, truth table, and MATLAB functions)	Input, Output	Function-call invocation

Data Parent	Scope	Initialization Time
	Local	Start of simulation or when chart reinitializes as part of an enabled Simulink subsystem

Save Data to the MATLAB Base Workspace

At the end of simulation, a Stateflow chart that uses C as the action language can save the final value of a data object to the MATLAB base workspace.

1 Open the Model Explorer by selecting **View > Model Explorer**.
2 Double-click the data object in the **Contents** pane.
3 In the **Description** pane of the Data properties dialog box, select **Save final value to base workspace**.

This option is available for data symbols of all scopes except `Constant` and `Parameter`.

See Also

More About

- "Add Stateflow Data" on page 9-2
- "Set Data Properties" on page 9-5
- "Share Parameters with Simulink and the MATLAB Workspace" on page 9-26
- "Access Data Store Memory from a Chart" on page 9-28

Share Parameters with Simulink and the MATLAB Workspace

A parameter is a constant data object that you can:

- Define in the MATLAB base workspace.
- Derive from a Simulink block parameter that you define and initialize in a mask.

Use parameters to avoid hard-coding data values and properties. Share Simulink parameters with charts to maintain consistency with your Simulink model.

You can access parameter values in multiple Stateflow objects in a chart such as states, MATLAB functions, and truth tables. You can include parameters in expressions defining data properties such as:

- Size
- Type
- Initial Value
- Minimum and Maximum
- Fixed-Point Data Properties

For more information, see "Enter Expressions and Parameters for Data Properties" on page 9-20

Initialize Parameters from the MATLAB Base Workspace

You can initialize a parameter by defining it in the MATLAB base workspace and in the Stateflow hierarchy.

1 Define and initialize a variable in the MATLAB base workspace.
2 In the Stateflow hierarchy, define a data object with the same name as the MATLAB variable.
3 Set the scope of the Stateflow data object to **Parameter**.

When the simulation starts, data resolution occurs. During this process, the Stateflow parameter gets its value from the associated MATLAB variable.

Share Simulink Parameters with Charts

You can share a parameter from a Simulink subsystem containing a Stateflow chart by creating a mask for the subsystem.

1 In the Simulink mask editor for the parent subsystem, define and initialize a Simulink parameter.
2 In the Stateflow hierarchy, define a data object with the same name as the Simulink parameter.
3 Set the scope of the Stateflow data object to **Parameter**.

When the simulation starts, Simulink tries to resolve the Stateflow data object to a parameter at the lowest-level masked subsystem. If unsuccessful, Simulink moves up the model hierarchy to resolve the data object to a parameter at higher-level masked subsystems.

See Also

More About
- "Create a Mask to Share Parameters with Simulink" on page 24-22
- "Add Stateflow Data" on page 9-2
- "Set Data Properties" on page 9-5

Access Data Store Memory from a Chart

A Simulink model implements global variables as data stores, either as Data Store Memory blocks or as instances of `Simulink.Signal` objects. You can use data stores to share data between multiple Simulink blocks without explicit input or output connections to pass data from one block to another. Stateflow charts share global data with Simulink models by reading from and writing to data store memory symbolically.
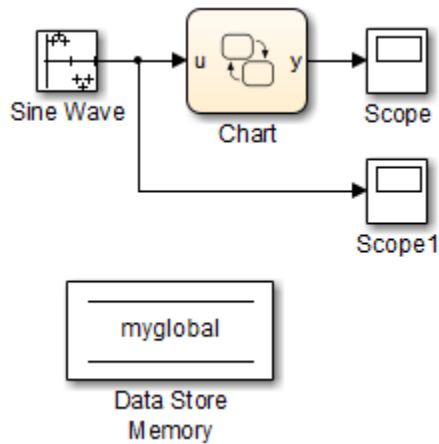
To access global data from a chart, bind a Stateflow data object to a Simulink data store. After you create the binding, the Stateflow data object becomes a symbolic representation of the Simulink data store memory. You can then use this symbolic object to store and retrieve global data.

Local and Global Data Store Memory

Stateflow charts can interface with local and global data stores.

- Local data stores are visible to all blocks in one model. To interact with a local data store, a chart must reside in the model where you define the local data store. You can define a local data store by adding a Data Store Memory block to a model or by creating a Simulink signal object.
- Global data stores have a broader scope that crosses model reference boundaries. To interact with global data stores, a chart must reside in the top model where you define

the global data store or in a model that the top model references. You implement global data stores as Simulink signal objects.

For more information, see "Local and Global Data Stores" (Simulink).

Bind Stateflow Data to Data Stores

1 To define the Simulink data store memory, add a Data Store Memory block to your model or create a Simulink signal object. For more information, see "Data Stores with Data Store Memory Blocks" (Simulink) and "Data Stores with Signal Objects" (Simulink).

2 Add a data object to the Stateflow chart, as described in "Add Stateflow Data" on page 9-2.

3 Set the **Name** property as the name of the Simulink data store memory to which you want to bind the Stateflow data object.

4 Set the **Scope** property to `Data Store Memory`.

The Stateflow data object inherits all additional properties from the data store memory to which you bind the object.

Multiple local and global data stores with the same name can exist in the same model hierarchy. In this situation, the Stateflow data object binds to the data store that is the nearest ancestor.

Store and Retrieve Global Data

After binding a Stateflow data object to a Simulink data store, you can store and retrieve global data in state and transition actions. The data object acts as a global variable that you reference by its symbolic name. When you store numeric values in this variable, you are writing to the Simulink data store memory. When you retrieve numeric values from this variable, you are reading from the data store memory.

For example, in this chart, the state actions read from and write to a Data Store Memory block called `myglobal`.
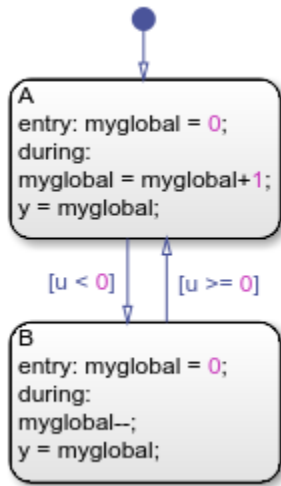
Best Practices for Using Data Stores

Data Store Properties in Charts

When you bind a Stateflow data object to a data store, the Stateflow object inherits all of its properties from the data store. To ensure that properties propagate correctly, when you create the Simulink data stores:

- Specify a data type other than `auto`.
- Minimize the use of automatic-mode properties.

Share Data Store Memory Across Multiple Models

To access a global data store from multiple models:

- Verify that your models do not contain any Data Store Memory blocks. You can include Data Store Read and Data Store Write blocks.
- In the MATLAB base workspace, create a `Simulink.Signal` object with these attributes:
  - Set **Data type** to an explicit data type. The data type cannot be `Auto`.
  - Fully specify **Dimensions**. The signal dimensions cannot be –1 or `Inherited`.

- Fully specify **Complexity**. The complexity cannot be `Auto`.
- Set **Storage class** to `ExportedGlobal`.
- In each chart that shares the data, bind a Stateflow data object to the Simulink data store.

Write to Data Store Memory Before Reading

To avoid algorithm latency, write to data store memory before reading from it. Otherwise, the read actions retrieve the value that was stored in the previous time step, rather than the value computed and stored in the current time step. When unconnected blocks share global data while running at different rates:

- Segregate read actions into separate blocks from write actions.
- Assign priorities to blocks so that your model invokes write blocks before read blocks. For more information, see "Control and Display the Sorted Order" (Simulink).

To avoid situations when multiple reads and writes occur unintentionally in the same time step, enable the Data Store Memory block diagnostics to:

- **Detect Read Before Write**
- **Detect Write After Read**
- **Detect Write After Write**

If you use a data store memory block as a persistent global storage area for accumulating values across time steps, avoid unnecessary warnings by disabling the Data Store Memory block diagnostics. For more information, see "Data Store Diagnostics" (Simulink).
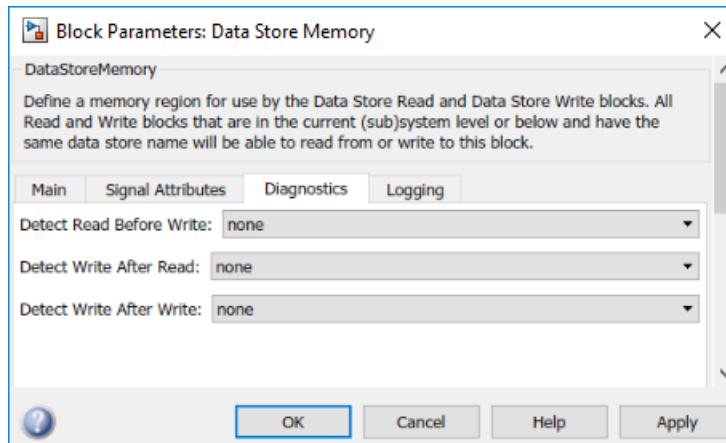
See Also

Data Store Read | Data Store Write | Data Store Memory | `Simulink.Signal`

More About

- "Add Stateflow Data" on page 9-2
- "Data Store Basics" (Simulink)
- "Model Global Data by Creating Data Stores" (Simulink)
- "Data Store Diagnostics" (Simulink)
- "Control and Display the Sorted Order" (Simulink)

Specify Type of Stateflow Data

The term data type refers to how computers represent information in memory. The data type determines the amount of storage allocated to data, the method of encoding a data value as a pattern of binary digits, and the operations available for manipulating the data.

Specify Data Type by Using the Data Type Assistant

You can specify the type of a data object in either the Property Inspector or the Model Explorer. In the **Type** field, select a type from the drop-down list or enter an expression that evaluates to a data type. For more information, see "Set Data Properties" on page 9-5.

Alternatively, use the Data Type Assistant to specify a data **Mode** and select the data type based on that mode:

1 In the Model Explorer, on the **Data** pane, click the **Show data type assistant** button ⟩⟩ .

2 Select a **Mode** from the drop-down list. The list of available modes depends on the scope of the data object.

Scope	Modes
Local	Inherit (available only in charts that use MATLAB as the action language), Built in, Fixed point, Enumerated, Bus object, Expression
Constant	Built in, Fixed point, Expression
Parameter	Inherit, Built in, Fixed point, Enumerated, Bus object, Expression
Input	Inherit, Built in, Fixed point, Enumerated, Bus object, Expression
Output	Inherit, Built in, Fixed point, Enumerated, Bus object, Expression
Data Store Memory	Inherit

3 Specify additional information based on the mode. The Data Type Assistant populates the **Type** field based on your specification.

Mode	Data Types
`Inherit`	You cannot specify a data type. You inherit the data type based on the scope that you select for the data object: • For charts that use MATLAB as the action language, if scope is `Local`, the data infers its type from the context of the MATLAB code in the chart. • If the scope is `Parameter`, the data inherits its type from the associated parameter, which you can define in the Simulink model or in the MATLAB base workspace. See "Share Parameters with Simulink and the MATLAB Workspace" on page 9-26. • If the scope is `Input`, the data inherits its type from the Simulink input signal on the designated input port. See "Share Input and Output Data with Simulink" on page 9-23. • If the scope is `Output`, the data inherits its type from the Simulink output signal on the designated output port. See "Share Input and Output Data with Simulink" on page 9-23. **Note** Avoid inheriting data types from output signals. See "Avoid inheriting output data properties from Simulink blocks" on page 9-4. • If the scope is `Data Store Memory`, the data inherits its type from the Simulink data store to which you bind the data object. See "Access Data Store Memory from a Chart" on page 9-28. For more information, see "Inherit Data Types from Simulink Objects" on page 9-36.

Mode	Data Types
`Built in`	Specify a data type from the drop-down list of supported data types: • `double`: 64-bit double-precision floating point. • `single`: 32-bit single-precision floating point. • `int32`: 32-bit signed integer. • `int16`: 16-bit signed integer. • `int8`: 8-bit signed integer. • `uint32`: 32-bit unsigned integer. • `uint16`: 16-bit unsigned integer. • `uint8`: 8-bit unsigned integer. • `boolean`: Boolean (1 = `true`; 0 = `false`). • `ml`: Typed internally with the MATLAB array `mxArray`. The `ml` data type provides Stateflow data with the benefits of the MATLAB environment, including the ability to assign the Stateflow data object to a MATLAB variable or pass it as an argument to a MATLAB function. `ml` data cannot have a scope outside the Stateflow hierarchy. That is, it cannot have a scope of `Input` or `Output`. For more information, see "ml Data Type" on page 12-37. • `string`: String. Supported only in charts that use C as the action language. For more information, see "Manage Textual Information by Using Strings" on page 20-2.
`Fixed point`	Specify this information about the fixed-point data: • **Signedness**: Whether the data is signed or unsigned • **Word Length**: Bit size of the word that holds the quantized integer. Large word sizes represent large values with greater precision than small word sizes. The default value is 16. • **Scaling**: Method for scaling your fixed-point data to avoid overflow conditions and minimize quantization errors. The default method is `Binary point`. For information, see "Fixed-Point Data Properties" on page 9-10.
`Enumerated`	Specify the class name for the enumerated data type. For more information, see "Define Enumerated Data Types" on page 19-6.

Mode	Data Types
Bus object	Specify the name of a `Simulink.Bus` object to associate with the Stateflow bus object structure. Click **Edit** to create or edit a bus object in the Bus Editor. You can also inherit bus object properties from Simulink signals.
Expression	Specify an expression that evaluates to a data type. You can use one of these expressions: • An alias type from the MATLAB base workspace, as described in "Specify Data Types by Using a Simulink Alias" on page 9-38. • `type` operator to specify the type of previously defined data, as described in "Derive Data Types from Other Data Objects" on page 9-37. • `fixdt` function to create a `Simulink.NumericType` object that describes a fixed-point or floating-point data type. For more information, see "Enter Expressions and Parameters for Data Properties" on page 9-20.

4   To save the data type settings, click **Apply**.

The Data Type Assistant is available only through the Model Explorer.

Inherit Data Types from Simulink Objects

When you select `Inherit: Same as Simulink` from the **Type** drop-down list, data objects of scope `Input`, `Output`, `Parameter`, and `Data Store Memory` inherit their data types from Simulink objects.

Scope	Description
Input	Inherits type from the Simulink input signal connected to corresponding input port in chart.
Output	Inherits type from the Simulink output signal connected to corresponding output port in chart. Avoid inheriting data types from output signals. Values that back-propagate from Simulink blocks can be unpredictable.
Parameter	Inherits type from the corresponding MATLAB base workspace variable or Simulink parameter in a masked subsystem.

Scope	Description
Data Store Memory	Inherits type from the corresponding Simulink data store.

To determine the data types that the objects inherit:

1   Build the Simulink model.
2   Open the Model Explorer.
3   In the **Contents** pane, examine the **CompiledType** column.

Derive Data Types from Other Data Objects

You can use the `type` operator to derive data types from other Stateflow data objects:

`type(data_obj)`

For example, in the model `sf_bus_demo`, the expression `type(inbus)` returns the data type of the input structure `inbus`. Because `inbus` derives its type from the `Simulink.Bus` object `COUNTERBUS`, the data type of the local structure `counterbus_struct` also derives its data type from `COUNTERBUS`.
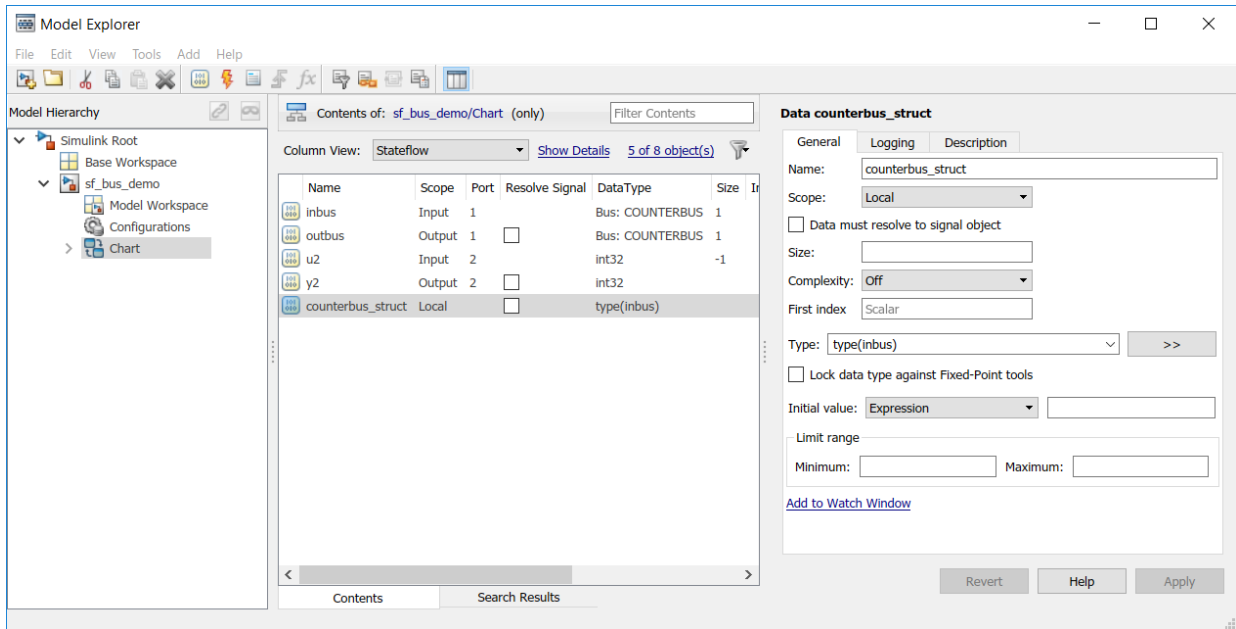
After you build your model, the **CompiledType** column of the Model Explorer shows the type used in the compiled simulation application.

Specify Data Types by Using a Simulink Alias

You can specify the type of Stateflow data by using a Simulink data type alias. For more information, see `Simulink.AliasType`.

For example, suppose that you want to define a data type alias `MyFloat` that corresponds to the built-in data type `single`. At the MATLAB command prompt, enter:

```
MyFloat = Simulink.AliasType;
MyFloat.BaseType = 'single';
```

To use this alias to specify the type of a data object, select the object in the Property Inspector or the Model Explorer. In the **Type** field, enter the alias name `MyFloat`.

After you build your model, the **CompiledType** column of the Model Explorer shows the type used in the compiled simulation application.

Strong Data Typing with Simulink Inputs and Outputs

By default, the **Use Strong Data Typing with Simulink I/O** chart property allows C charts to interface directly with signals from Simulink models. The chart accepts only input signals whose data type matches the type of the corresponding Stateflow data object. Otherwise, a type mismatch error occurs. For example, by selecting **Use Strong Data Typing with Simulink I/O**, you can flag mismatches between input or output fixed-point data in charts and their counterparts in Simulink models. For more information, see "Specify Properties for Stateflow Charts" on page 24-3.

If you clear the **Use Strong Data Typing with Simulink I/O** chart property, the chart converts inputs signals of type `double` to the type of the corresponding input data object in the chart. The chart converts output data objects to type `double` before exporting them as output signals to Simulink models.

Note The **Use Strong Data Typing with Simulink I/O** chart property is provided for backward compatibility. Clearing this check box can produce unpredictable results and is not recommended.

See Also

More About

- "Set Data Properties" on page 9-5
- "Specify Properties for Stateflow Charts" on page 24-3
- "About Data Types in Simulink" (Simulink)

Specify Size of Stateflow Data

Methods for Sizing Stateflow Data

In a Stateflow chart in a Simulink model, you can specify the size of data objects by:

- Inheriting the size from a Simulink signal
- Using numeric values
- Using MATLAB expressions

Support for a sizing method depends on the scope of your data:

Scope of Data	Method for Sizing Data		
	Inherit the Size	**Use Numeric Values**	**Use MATLAB Expressions**
Local	No	Yes	Yes
Constant	No	Yes	Yes
Parameter	No	Yes	Yes
Input	Yes	Yes	Yes
Output	Yes	Yes	Yes
Data store memory	Yes	No	No

Stateflow data store memory inherits all data properties, including size, from the Simulink data store to which it resolves. You cannot specify any properties explicitly for data store memory.

How to Specify Data Size

Specify the size of Stateflow data by setting the **Size** property in the Property Inspector or the Data properties dialog box. To specify the size of Stateflow data using API commands, set the `Props.Array.Size` property to a numeric value or a MATLAB expression that represents a scalar, vector, matrix, or n-dimensional array. For more information on using the API, see "Stateflow.Data Properties".

Inherit Input or Output Size from Simulink Signals

To configure Stateflow input and output data to inherit size from the corresponding Simulink input and output signals, enter –1 in the **Size** field of the data properties. This default setting applies to input and output data that you add to your chart. After you build your model, the **Compiled Size** column of the Model Explorer displays the actual size that the compiled simulation application uses.

The equivalent API command for specifying an inherited data size is:

data_handle.Props.Array.Size = '-1';

Chart actions that store values in the specified output infer the inherited size of output data. If the expected size in the Simulink signal matches the inferred size, inheritance is successful. Otherwise, a mismatch occurs during build time.

Note Charts cannot inherit frame-based data sizes from Simulink signals.

Guidelines for Sizing Data with Numeric Values

When you specify data size using numeric values in the **Size** field of the Data properties dialog box, follow these guidelines:

Dimensionality	What to Specify in the Dialog Box	Equivalent API Command
Scalar	1 (or leave the field blank)	*data_handle*.Props.Array.Size = '1'; *data_handle*.Props.Array.Size = '';
Vector	The number of elements in the row or column vector	*data_handle*.Props.Array.Size = '*number_of_elements*';
Matrix	An expression of the format [*r c*], where: • *r* is the number of rows • *c* is the number of columns	*data_handle*.Props.Array.Size = '[*r c*]';

Dimensionality	What to Specify in the Dialog Box	Equivalent API Command
N-dimensional array	An expression of the format [*Size_of_dim1 Size_of_dim2 ... Size_of_dimN*], where: • *Size_of_dim1* is the size of the first dimension • *Size_of_dim2* is the size of the second dimension • *Size_of_dimN* is the size of the N-th dimension	*data_handle*.Props.Array.Size = '[*Size_of_dim1 Size_of_dim2 ... Size_of_dimN*];

One-dimensional Stateflow vectors are compatible with Simulink row or column vectors of the same size. For example, Stateflow input or output data of size 3 is compatible with a Simulink row vector of size [1 3] or column vector of size [3 1].

Guidelines for Sizing Data with MATLAB Expressions

When you specify data size using MATLAB expressions, follow the same guidelines that apply to sizing with numeric values (see "Guidelines for Sizing Data with Numeric Values" on page 9-41). The following guidelines also apply.

• Expressions that specify the size of a dimension:

  • Can contain a mix of numeric values, variables, arithmetic operators, parameters, and calls to MATLAB functions.

  • Must evaluate to a positive integer value.

• To specify inherited data size, you must enter –1 in the **Size** field or set the Props.Array.Size property for the data to –1. Expressions cannot evaluate to a value of –1.

• If the expression contains an enumerated value, you must include the type prefix for consistency with MATLAB naming rules.

  For example, Colors.Red is valid but Red is not.

• You cannot size Stateflow input data with an expression that accepts frame-based data from Simulink.

Examples of Valid Data Size Expressions

The following examples are valid MATLAB expressions for sizing data in your chart:

- `K+3`, where `K` is a chart-level Stateflow data
- `N/2`, where `N` is a variable in the MATLAB base workspace
- `2*Colors.Red`, where `Red` is an enumerated value of type `Colors`
- `[fi(2,1,16,2) fi(4,1,16,2)]`, which specifies a data size of `[2 4]` using a signed fixed-point type with word length of 16 and fraction length of 2

Name Conflict Resolution for Variables in Size Expressions

When multiple variables with identical names exist in a model, the variable with the highest priority applies:

1. Mask parameters
2. Model workspace
3. MATLAB base workspace
4. Stateflow data

Best Practices for Sizing Stateflow Data

Avoid use of variables that can lead to naming conflicts

For example, if a variable named `off` exists in the MATLAB base workspace and as local chart data, do not use `off` in the **Size** field of the data properties.

Avoid use of size(u) expressions

Instead of using a `size(u)` expression, use a MATLAB expression that evaluates directly to the size of Stateflow data.

See Also

More About

- "Stateflow Data Properties" on page 9-5

Handle Integer Overflow for Chart Data

When Integer Overflow Can Occur

For some arithmetic operations, a processor might need to take an n-bit fixed-point value and store it in m bits, where $m \neq n$. If $m < n$, the reduced range of the value can cause an overflow for an arithmetic operation. Some processors identify this overflow as `Inf` or `NaN`. Other processors, especially digital signal processors (DSPs), handle overflows by saturating or wrapping the value.

For more information about saturation and wrapping for integer overflow, see "Saturation and Wrapping" (Fixed-Point Designer).

Support for Handling Integer Overflow in Charts

For Stateflow charts in Simulink models, you can control whether or not saturation occurs for integer overflow. Use the chart property, **Saturate on integer overflow**, to control overflow handling.
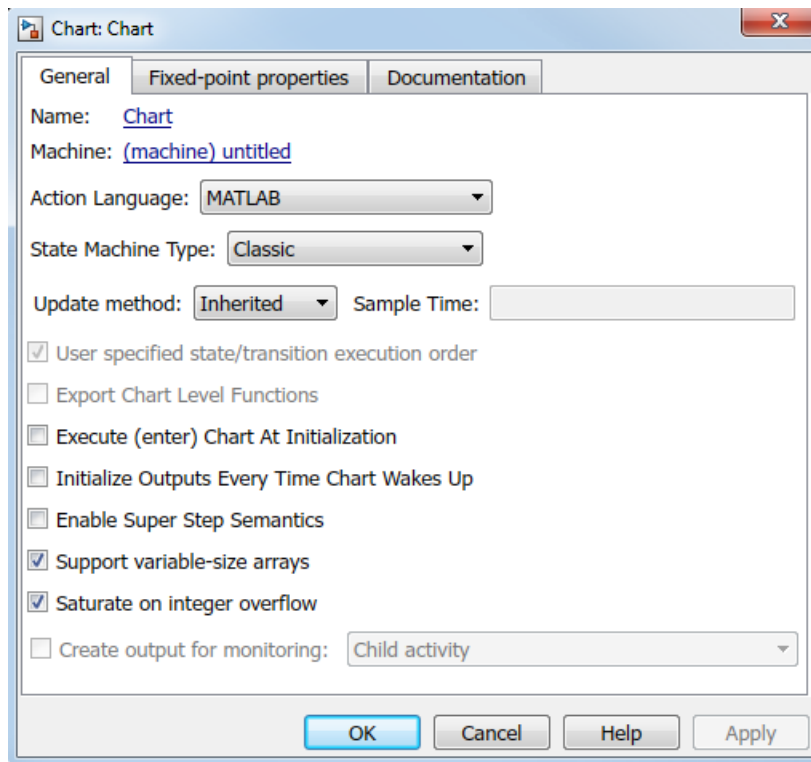
Check Box	When to Use This Setting	Overflow Handling	Example of the Result
Selected	Overflow is possible for data in your chart and you want explicit saturation protection in the generated code.	Overflows saturate to either the minimum or maximum value that the data type can represent.	An overflow associated with a signed 8-bit integer saturates to –128 or +127 in the generated code.
Cleared	You want to optimize efficiency of the generated code.	The handling of overflows depends on the C compiler that you use for generating code.	The number 130 does not fit in a signed 8-bit integer and wraps to –126 in the generated code.

Arithmetic operations for which you can enable saturation protection are:

- Unary minus: `–a`
- Binary operations: `a + b`, `a – b`, `a * b`, `a / b`, `a ^ b`
- Assignment operations: `a += b`, `a –=b`, `a *= b`, `a /= b`
- In C charts, increment and decrement operations: `++`, `--`

When you select **Saturate on integer overflow**, be aware that:

- Saturation applies to all intermediate operations, not just the output or final result.
- The code generator can detect some cases when overflow is not possible. In these cases, the generated code does not include saturation protection.

To determine whether clearing the **Saturate on integer overflow** check box is a safe option, perform a careful analysis of your logic, including simulation if necessary. If saturation is necessary in only some sections of the logic, encapsulate that logic in atomic subcharts or MATLAB functions and define a different set of saturation settings for those units.

Effect of Integer Promotion Rules on Saturation

Charts use ANSI® C rules for integer promotion.

- All arithmetic operations use a data type that has the same word length as the target word size. Therefore, the intermediate data type in a chained arithmetic operation can be different from the data type of the operands or the final result.
- For operands with integer types smaller than the target word size, promotion to a larger type of the same word length as the target size occurs. This implicit cast occurs before any arithmetic operations take place.

  For example, when the target word size is 32 bits, an implicit cast to `int32` occurs for operands with a type of `uint8`, `uint16`, `int8`, or `int16` before any arithmetic operations occur.

Suppose that you have the following expression, where `y`, `u1`, `u2`, and `u3` are of `uint8` type:

```
y = (u1 + u2) - u3;
```

Based on integer promotion rules, that expression is equivalent to the following statements:

```
uint8_T u1, u2, u3, y;
int32_T tmp, result;
tmp = (int32_T) u1 + (int32_T) u2;
result = tmp - (int32_T) u3;
y = (uint8_T) result;
```

For each calculation, the following data types and saturation limits apply.

Calculation	Data Type	Saturation Limits
tmp	int32	(MIN_INT32, MAX_INT32)
result	int32	(MIN_INT32, MAX_INT32)
y	uint8	(MIN_UINT8, MAX_UINT8)

Suppose that u1, u2, and u3 are equal to 200. Because the saturation limits depend on the intermediate data types and not the operand types, you get the following values:

- tmp is 400.
- result is 200.
- y is 200.

Impact of Saturation on Error Checks

Suppose that you set **Wrap on overflow** in the **Diagnostics: Data Validity** pane of the Model Configuration Parameters dialog box to error or warning. When you select **Saturate on integer overflow**, Stateflow does not flag cases of integer overflow during simulation. However, Stateflow continues to flag the following situations:

- Out-of-range data violations based on minimum and maximum range checks
- Division-by-zero operations

Define Temporary Data for Functions in C Charts

In charts that use C as the action language, define temporary data when you want to use data that is only valid while a function executes. You can define temporary data in graphical, truth table, and MATLAB functions in your chart. For example, you can designate a loop counter to have `Temporary` scope if the counter value does not need to persist after the function completes.

In charts that use MATLAB as the action language, you do not need to define temporary function data in the Model Explorer. If a variable is used and not previously defined, then it is automatically created. The variable is available to the rest of the function.

How to Define Temporary Data

To define temporary data for a Stateflow function, follow these steps:

1   Open the Model Explorer.

2   In the Model Explorer, select the graphical, truth table, or MATLAB function that will use temporary data.

3   Select **Add > Data**.

    The Model Explorer adds a default definition for the data in the Stateflow hierarchy, with a scope set to **Temporary** by default.

4   Change other properties of the data if necessary, as described in "Set Data Properties" on page 9-5.

Identify Data by Using Dot Notation

To specify the path from the parent state to a data object, a qualified data name uses dot notation. Dot notation is a way to identify data at a specific level of the Stateflow chart hierarchy. The first part of a qualified data name identifies the parent object. Subsequent parts identify the children along a hierarchical path.

For example, in this chart, the symbol `data` resides in the substate `aa` of the state `a`. The state and transition actions use qualified data names to refer to this symbol.

- In the default transition, the action uses the qualified data name `a.aa.data` to specify a path from the chart to the top-level state `a`, to the substate `aa`, and finally to `data`.

- In state `a`, the entry action uses the qualified data name `aa.data` to specify a path from the substate `aa` to `data`.

- In state `b`, the entry action uses the qualified data name `a.aa.data` to specify a path from the chart to the state `a`, to the substate `aa`, and then to `data`.
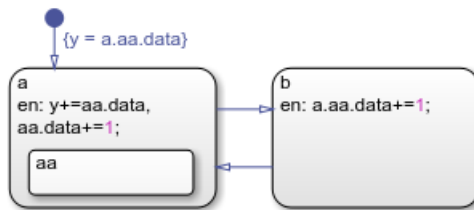
Resolution of Qualified Data Names

During simulation, Stateflow resolves the qualified data name by performing a localized search of the chart hierarchy for a matching data object. The search begins at the hierarchy level where the qualified data name appears:

- For a state action, the starting point is the state containing the action.

- For a transition label, the starting point is the parent of the transition source.

The resolution process searches each level of the chart hierarchy for a path to the data. If a data object matches the path, the process adds that data object to the list of possible matches. Then, the process continues the search one level higher in the hierarchy. The resolution process stops after it searches the chart level of the hierarchy. If a unique

match exists, the qualified data name resolves to the matching path. Otherwise, the resolution process fails. Simulation stops, and you see an error message.

This flow chart illustrates the different stages in the process for resolving qualified data names.
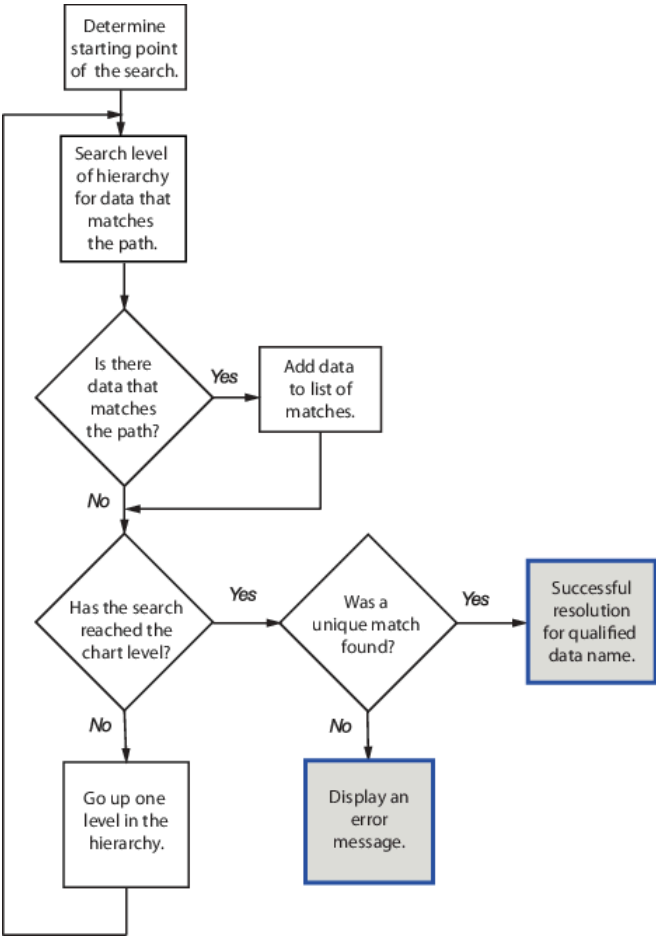
Best Practices for Using Dot Notation

Resolving qualified data names:

- Does not perform an exhaustive search of all data.
- Does not stop after finding the first match.

To improve the chances of finding a unique search result when resolving qualified data names:

- Use specific paths in qualified data names.
- Give states unique names.
- Use states and boxes as enclosures to limit the scope of the path resolution search.

Examples of Qualified Data Name Resolution

Search Produces No Matches

In this chart, the entry action in state b contains the qualified data name `aa.data`. If the symbol `data` resides in state `aa`, then Stateflow cannot resolve the qualified data name.
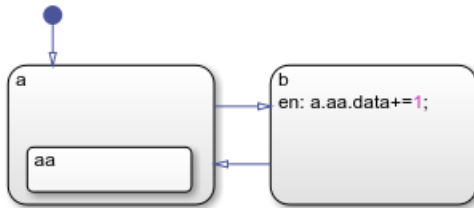
This table lists the different stages in the resolution process for the qualified data name `aa.data`.

Stage	Description	Result
1	Starting in state b, search for an object aa that contains `data`.	No match found.
2	Move up to the next level of the hierarchy (the chart level). Search for an object aa that contains `data`.	No match found.

The search ends at the chart level with no match found for `aa.data`, resulting in an error.

To avoid this error, in the entry action of state b, specify the data with the more specific qualified data name `a.aa.data`.

Search Produces Multiple Matches

In this chart, the entry action in state `a` contains two instances of the qualified data name `aa.data`. If both states named `aa` contain a data object named `data`, then Stateflow cannot resolve the qualified data name.
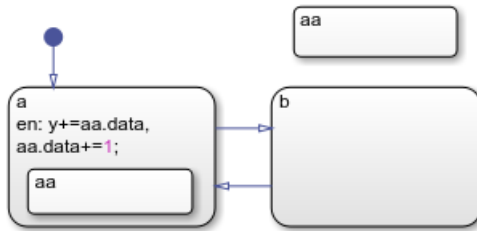
This table lists the different stages in the resolution process for the qualified data name `aa.data`.

Stage	Description	Result
1	Starting in state `a`, search for an object `aa` that contains `data`.	Match found.
2	Move up to the next level of the hierarchy (the chart level). Search for an object `aa` that contains `data`.	Match found.

The search ends at the chart level with two matches found for `aa.data`, resulting in an error.

To avoid this error:

- Use a more specific qualified data name. For instance:

  - To specify the data object in the substate of state `a`, use the qualified data name `a.aa.data`.

  - To specify the data object in the top-level state `aa`, use the qualified data name `/aa.data`.

- Rename one of the states containing `data`.

- Enclose the top-level state `aa` in a box or in another state. Adding an enclosure prevents the search process from detecting data in the top-level state.
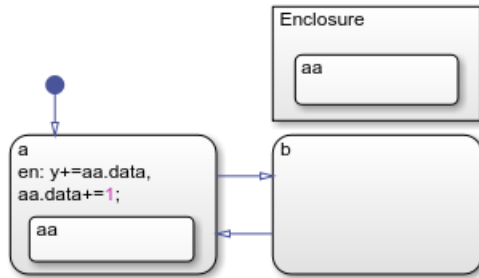
See Also

More About

- "State Action Types" on page 12-2
- "Transition Action Types" on page 12-7
- "State Hierarchy" on page 2-14

Resolve Data Properties from Simulink Signal Objects

Stateflow® local and output data in charts can explicitly inherit properties from `Simulink.Signal` objects in the model workspace or base workspace. This process is called signal resolution and requires that the resolved signal have the same name as the chart output or local data.

For information about Simulink® signal resolution, see "Symbol Resolution" (Simulink) and "Symbol Resolution Process" (Simulink).

Inherited Properties

When Stateflow local or output data resolve to Simulink signal objects, they inherit these properties:

- Size
- Complexity
- Type
- Minimum value
- Maximum value
- Initial value
- Storage class

Storage class controls the appearance of chart data in the generated code. See "Apply Custom Storage Classes to Individual Signal, State, and Parameter Data Elements" (Embedded Coder).

Enable Signal Resolution

To enable explicit signal resolution, follow these steps:

1   Set **Configuration Parameters** > **Diagnostics** > **Data Validity** > **Signal resolution** to a value other than `None`. For more information about the other options, see "Signal resolution" (Simulink).

2   In the model workspace, base workspace, or data dictionary, define a `Simulink.Signal` object with the properties you want your Stateflow data to inherit. For more information about creating Simulink signals, see `Simulink.Signal`.

3 Add output or local data to a chart.

4 Enter a name for your data that matches the name of the `Simulink.Signal` object.

5 In the data properties, select the **Data must resolve to signal object** check box. After you select this check box, the dialog box removes or dims the properties that your data inherits from the signal.

A Simple Example

This model shows how a chart resolves local and output data to `Simulink.Signal` objects.

Resolve Signal Object
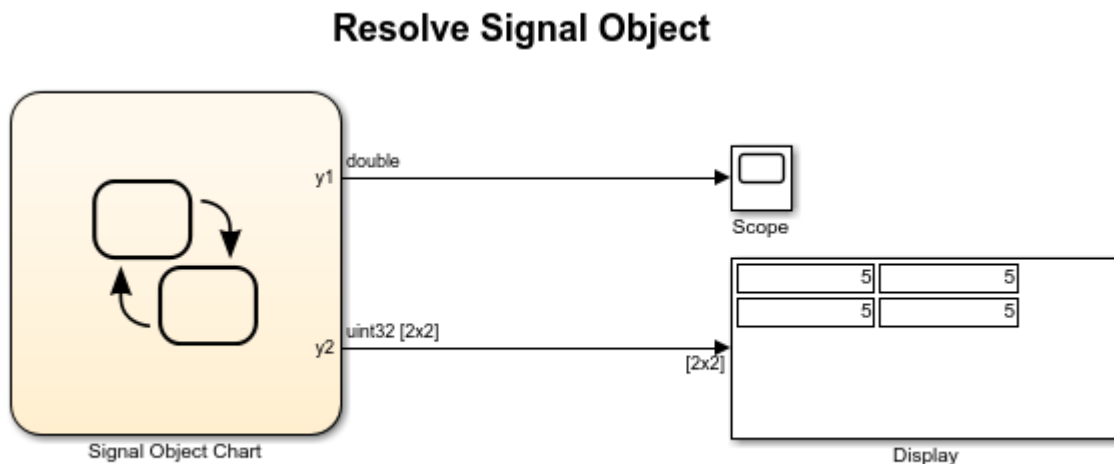
Copyright 2007-2018 The MathWorks, Inc.

In the base workspace, there are three `Simulink.Signal` objects, each with a different set of properties.

- `y1` has these properties: **Type** = `double`, **Dimensions** = 1, and **Storage Class** = `Model default`.

- `y2` has these properties: **Type** = `uint32`, **Dimensions** = [2 2], and **Storage Class** = `Auto`.

- `local` has these properties: **Type** = `single`, **Dimensions** = 1, and **Storage Class** = `ExportedGlobal`.

The chart contains three data objects — two outputs and a local variable — that will resolve to a signal with the same name.
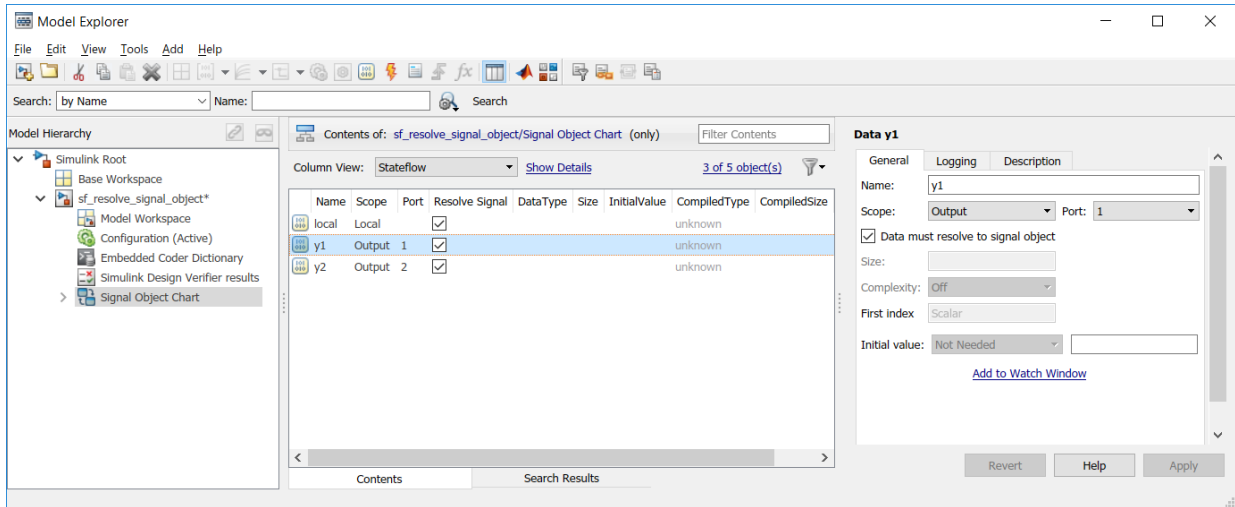
When you build the model, each data object inherits the properties of the identically named signal.
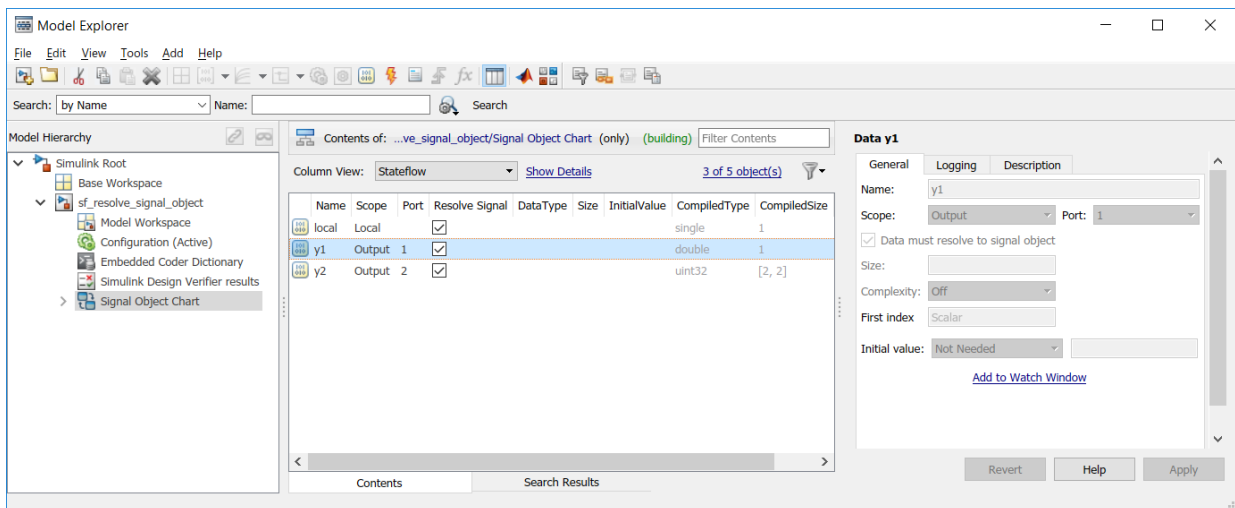
The generated code declares the data based on the storage class that the data inherits from the associated Simulink signal. For example, the header file below declares local to be an exported global variable:

```
/*
 * Exported States
 *
 * Note: Exported states are block states with an exported
 * global storage class designation.
 *
 */
extern real32_T local;                    /* '<Root>/Signal Object Chart' */
```

See Also
`Simulink.Signal`

More About
- "Symbol Resolution" (Simulink)
- "Apply Custom Storage Classes to Individual Signal, State, and Parameter Data Elements" (Embedded Coder)

Transfer Data Across Models

Copy Data Objects

When you copy a chart from one Simulink model to another, all data objects in the chart hierarchy are copied *except* those that the Stateflow machine parents. However, you can use the Model Explorer to transfer individual data objects from machine to machine.

To *copy* a data object, follow these steps:

1. In the **Contents** pane of the Model Explorer, right–click the data object you want to copy and select **Copy** from the context menu.
2. In the **Model Hierarchy** pane, right-click the destination Stateflow machine and select **Paste** from the context menu.

Move Data Objects

To *move* a data object, click the object in the **Contents** pane of the Model Explorer and drag it to the destination Stateflow machine in the **Model Hierarchy** pane.

Define Events

Synchronize Model Components by Broadcasting Events

An event is a Stateflow object that can trigger actions in one of these objects:

- A parallel state in a Stateflow chart.
- Another Stateflow chart.
- A Simulink triggered or function-call subsystem.

For simulation purposes, there is no limit to the number of events in a Stateflow chart. However, for code generation, the underlying C compiler enforces a theoretical limit of 2^{31}-1 events.

Types of Events

An implicit event is a built-in event that is broadcasted during chart execution. These events are implicit because you do not define or trigger them explicitly. For more information, see "Define Chart Behavior by Using Implicit Events" on page 10-33.

An explicit event is an event that you define explicitly. Explicit events can have one of these scopes.

Scope	Description
Input	Event that is broadcast to a Stateflow chart from outside the chart. For more information, see "Activate a Stateflow Chart by Sending Input Events" on page 10-9 and "Design Human-Machine Interface Logic by Using Stateflow Charts" on page 34-28.
Local	Event that can occur anywhere in a Stateflow chart but is visible only in the parent object and its descendants. Local events are supported only in Stateflow charts in Simulink models. For more information, see "Broadcast Local Events to Synchronize Parallel States" on page 12-45.
Output	Event that occurs in a Stateflow chart but is broadcast to a Simulink block. Output events are supported only in Stateflow charts in Simulink models. For more information, see "Activate a Simulink Block by Sending Output Events" on page 10-21.

You can define explicit events at these levels of the Stateflow hierarchy.

Level of Hierarchy	Visibility
Chart	Event is visible to the chart and all its states and substates.
Subchart	Event is visible to the subchart and all its states and substates.
State	Event is visible to the state and all its substates.

Define Events in a Chart

You can add events to a Stateflow chart through the Symbols window, the **Chart** menu in the Stateflow Editor, or the Model Explorer.

Add Events Through the Symbols Window

1   To open the Symbols window, select **View** > **Symbols**.

2   Click the **Create Event** icon .

3   In the row for the new event, under **TYPE**, click the icon and choose:

   - `Input Event`
   - `Local Event`
   - `Output Event`

4   Edit the name of the event.

5   For input and output events, click the **PORT** field and choose a port number.

6   To specify properties for the event, open the Property Inspector. In the Symbols window, right-click the row for the event and select **Explore**. For more information, see "Set Properties for an Event" on page 10-6.

Add Events by Using the Stateflow Editor Menu

1   In a Stateflow chart in a Simulink model, select the menu option corresponding to the scope of the event that you want to add.

Scope	Menu Option
`Input`	**Chart > Add Inputs & Outputs > Event Input From Simulink**
`Output`	**Chart > Add Inputs & Outputs > Event Output To Simulink**

Scope	Menu Option
`Local`	**Chart > Add Other Elements > Local Event**

2 In the Event dialog box, specify data properties. For more information, see "Set Properties for an Event" on page 10-6.

Add Events Through the Model Explorer

1 In a Stateflow chart in a Simulink model, select **View > Model Explorer**.

2 In the **Model Hierarchy** pane, select the object in the Stateflow hierarchy where you want to make the new event visible. The object that you select becomes the parent of the new event.

3 In the Model Explorer menu, select **Add > Event**. The new event with a default definition appears in the **Contents** pane of the Model Explorer.

4 In the **Event** pane, specify the properties of the event. For more information, see "Set Properties for an Event" on page 10-6.

Access Event Information from a Stateflow Chart

You can display the properties of an input or local event, or open the destination of an output event directly from a Stateflow chart. Right-click the state or transition that contains the event of interest and select **Explore**. A context menu lists the names and scopes of all resolved symbols in the state or transition. Selecting an input or local event from the context menu displays its properties in the Model Explorer. Selecting an output event from the context menu opens the Simulink subsystem or Stateflow chart associated with the event.

Best Practices for Using Events in Stateflow Charts

Use the send Command to Broadcast Explicit Events in Actions

To broadcast explicit events in state or transition actions, use the `send` command. Using this command enhances readability of a chart and ensures that explicit events are not mistaken for data. For more information, see "Broadcast Local Events to Synchronize Parallel States" on page 12-45.

Avoid Using Explicit Events to Trigger Conditional Actions

Use conditions on transitions instead of events when you want to:

- Represent conditional statements, for example, [x < 1] or [x == 0].
- Represent a change of data value, for example, [hasChanged(x)].

For more information, see "Transition Action Types" on page 12-7.

Avoid Using the Implicit Event enter to Check State Activity

To check state activity, use the in operator instead of the implicit event enter. For more information, see "Check State Activity by Using the in Operator" on page 12-77.

Do Not Mix Edge-Triggered and Function-Call Input Events in a Chart

Mixing input events that use edge triggers and function calls results in an error during parsing and simulation.

See Also

More About
- "Set Properties for an Event" on page 10-6
- "Broadcast Local Events to Synchronize Parallel States" on page 12-45
- "Activate a Stateflow Chart by Sending Input Events" on page 10-9
- "Activate a Simulink Block by Sending Output Events" on page 10-21
- "Rules for Naming Stateflow Objects" on page 2-4
- "Identify Data by Using Dot Notation" on page 9-50

Set Properties for an Event

When you create Stateflow charts in Simulink models, you can specify event properties in either the Property Inspector or the Model Explorer.

- Property Inspector

    1   Open the Symbols window by selecting **View** > **Symbols**.

    2   Open the Property Inspector by selecting **View** > **Property Inspector**.

    3   In the Symbols window, select the event.

    4   In the Property Inspector window, edit the event properties.

- Model Explorer

    1   Open the Model Explorer by selecting **View** > **Model Explorer**.

    2   In the **Contents** pane, double-click the event.

    3   In the **Event** pane, edit the event properties.

For more information, see "Synchronize Model Components by Broadcasting Events" on page 10-2.

Stateflow Event Properties

Name

Name of the event. Actions reference events by their names. Names must begin with an alphabetic character, cannot include spaces, and cannot be shared by sibling events. For more information, see "Rules for Naming Stateflow Objects" on page 2-4.

Scope

Scope of the event. The scope specifies where the event occurs relative to the parent object.

Scope	Description
Local	Event that can occur anywhere in a Stateflow machine but is visible only in the parent object and its descendants. For more information, see "Directed Event Broadcasting" on page 12-45.

Scope	Description
Input from Simulink	Event that occurs in a Simulink block but is broadcast to a Stateflow chart. For more information, see "Activate a Stateflow Chart by Sending Input Events" on page 10-9.
Output to Simulink	Event that occurs in a Stateflow chart but is broadcast to a Simulink block. For more information, see "Activate a Simulink Block by Sending Output Events" on page 10-21.

Port

Index of the port associated with the event. This property applies only to input and output events.

- For input events, port is the index of the input signal that triggers the event. For more information, see "Association of Input Events with Control Signals" on page 10-12.
- For output events, port is the index of the signal that outputs this event. For more information, see "Association of Output Events with Output Ports" on page 10-31.

Trigger

Type of signal that triggers an input or output event. For more information, see "Activate a Stateflow Chart by Sending Input Events" on page 10-9 and "Activate a Simulink Block by Sending Output Events" on page 10-21.

Debugger Breakpoints

Option for setting debugger breakpoints at the start or end of an event broadcast. Available breakpoints depend on the scope of the event.

Scope of Event	Start of Broadcast	End of Broadcast
Local	Available	Available
Input	Available	Not available
Output	Not available	Not available

For more information, see "Set Breakpoints to Debug Charts" on page 32-5.

Description

Description of the event. You can enter brief descriptions of events in the hierarchy.

Document Link

Link to online documentation for the event. You can enter a web URL address or a MATLAB command that displays documentation in a suitable online format, such as an HTML file or text in the MATLAB Command Window. When you click the **Document link** hyperlink, Stateflow displays the documentation.

See Also

More About

- "Synchronize Model Components by Broadcasting Events" on page 10-2
- "Directed Event Broadcasting" on page 12-45
- "Activate a Stateflow Chart by Sending Input Events" on page 10-9
- "Activate a Simulink Block by Sending Output Events" on page 10-21
- "Rules for Naming Stateflow Objects" on page 2-4
- "Identify Data by Using Dot Notation" on page 9-50

Activate a Stateflow Chart by Sending Input Events

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

1. Add an event to the Stateflow chart, as described in "Define Events in a Chart" on page 10-3.

2. Set the **Scope** property for the event to `Input from Simulink`. A single trigger port appears at the top of the Stateflow block in the Simulink model.

3. An input event can activate a Stateflow chart through a change in a control signal (an edge trigger) or a function call from a Simulink block.

   - To specify an edge-triggered input event, set the **Trigger** property to one of these options:

     - `Rising`
     - `Falling`
     - `Either`

   - To specify a function-call input event, set the **Trigger** property to `Function call`.

   You cannot mix edge-triggered and function-call input events in the same Stateflow chart. Mixing these input events results in an error during parsing and simulation.

For more information, see "Synchronize Model Components by Broadcasting Events" on page 10-2.

Activate a Stateflow Chart by Using Edge Triggers

An edge-triggered input event causes a Stateflow chart to execute during the current time step of simulation. With this type of input event, a change in a control signal acts as a trigger.

Edge Trigger Type	Description
Rising	Rising edge trigger. Chart is activated when the control signal changes from either zero or a negative value to a positive value.

Edge Trigger Type	Description
Falling	Falling edge trigger. Chart is activated when the control signal changes from a positive value to either zero or a negative value.
Either	Either rising or falling edge trigger. Chart is activated when the control signal crosses zero as it changes in either direction.

In all cases, the value of the control signal must cross zero to be a valid edge trigger. For example, a signal that changes from -1 to 1 is a valid rising edge trigger. A signal that changes from 1 to 2 is not a valid rising edge trigger.

When to Use an Edge-Triggered Event

Use an edge-triggered input event to activate a chart when your model requires regular or periodic chart execution. For example, in the model `sf_loop_scheduler`, an edge-triggered input event activates the Edge to Function chart at regular intervals. For more information, see "Schedule A Subsystem Multiple Times in a Single Step" on page 26-18.
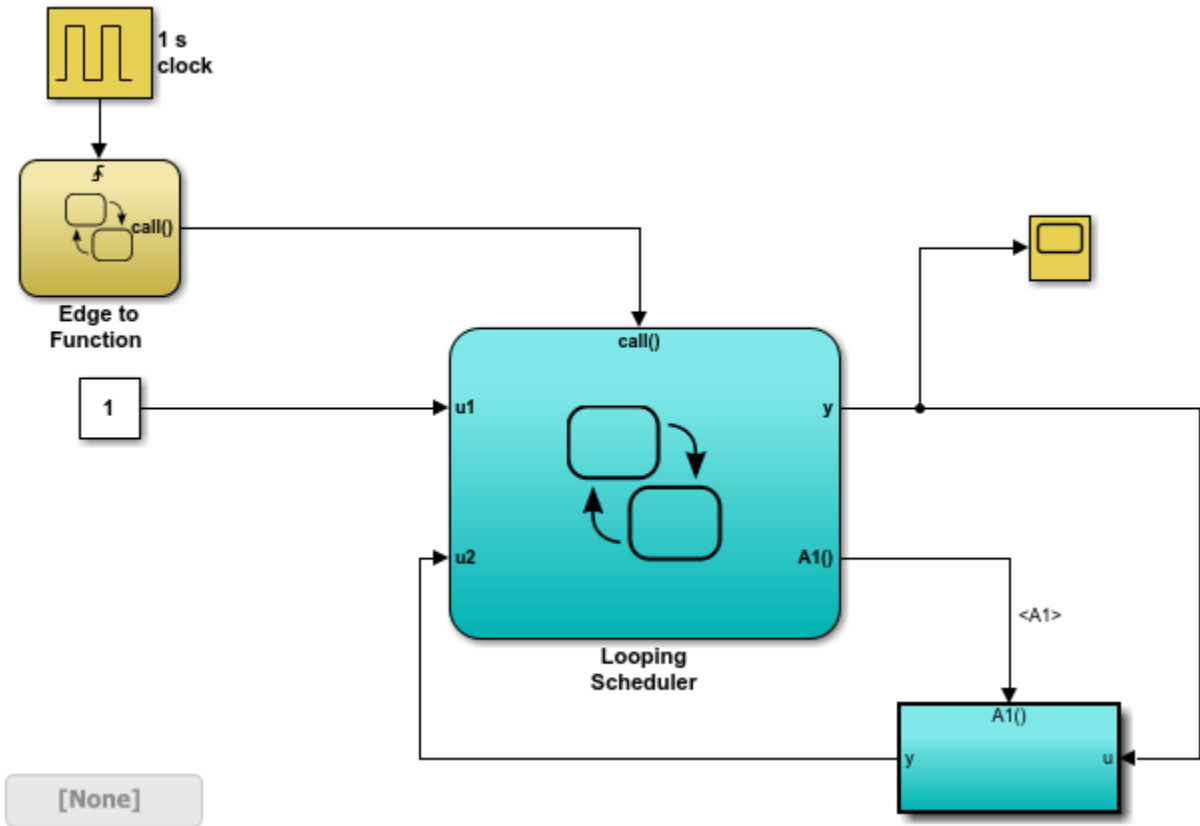
Behavior of Multiple Edge-Triggered Input Events

At any given time step, input events are checked in ascending order based on their port numbers. The chart awakens once for each valid event. For edge-triggered input events, multiple edges can occur in the same time step, waking the chart more than once in that time step. In this situation, the events wake the chart in ascending order based on their port numbers.

Activate a Stateflow Chart by Using Function Calls

A function-call input event causes a Stateflow chart to execute during the current time step of simulation. With this type of input event, you must also define a function-call output event for the block that calls the Stateflow chart.

When to Use a Function-Call Input Event

Use a function-call input event to activate a chart when your model requires access to output data from the chart in the same time step as the function call. For example, in the model `sf_loop_scheduler`, a function-call input event activates the Looping Scheduler chart. For more information, see "Schedule A Subsystem Multiple Times in a Single Step" on page 26-18.

Behavior of Multiple Function-Call Input Events

For function-call input events, only one trigger event exists. The caller of the event explicitly calls and executes the chart. Only one function call is valid in a single time step.

Association of Input Events with Control Signals

When you define one or more input events in a chart, a single trigger port appears on the top side of the chart block. Multiple external Simulink blocks can trigger the input events through a vector of signals connected to the trigger port. The **Port** property of an input event specifies the index into the control signal vector that connects to the trigger port.

By default, **Port** values appear in the order that you add input events. You can change these assignments by modifying the **Port** property of the events. When you change the **Port** property for an input event, the **Port** values for the remaining input events automatically renumber.

Data Types Allowed for Input Events

For multiple input events to a trigger port, all signals must have the same data type. Using signals of different data types as input events results in an error during simulation. For example, you can mux two input signals of type `double` to use as input events to a chart.
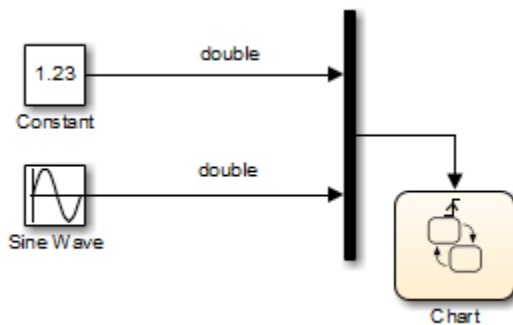
You cannot mux two input signals of different data types, such as `boolean` and `double`.
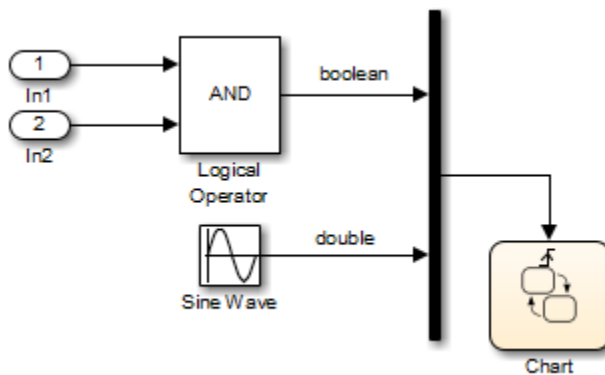
See Also

More About

- "Synchronize Model Components by Broadcasting Events" on page 10-2
- "Set Properties for an Event" on page 10-6
- "Control States in Charts Enabled by Function-Call Input Events" on page 10-14
- "Schedule A Subsystem Multiple Times in a Single Step" on page 26-18
- "View Differences Between Stateflow Messages, Events, and Data" on page 11-2

Control States in Charts Enabled by Function-Call Input Events

In charts enabled by function-call input events in a Simulink model, you control the behavior of states by setting the **States When Enabling** chart property. Depending on the value of this property, when the input event reenables the chart, states either maintain their most recent values or revert to their initial values. To modify the property:

1   Open the Chart properties dialog box.
2   Set the **States When Enabling** property to one of these options.

Setting	Description
Held	Maintain most recent values of the states.
Reset	Revert to the initial values of the states.
Inherit	Inherit setting from the parent subsystem.

For new charts, the default setting is Held. For more information, see "Activate a Stateflow Chart by Sending Input Events" on page 10-9.

Behavior When Parent Is Model Root

If the parent of the chart is the model root, then setting **States When Enabling** to Inherit is equivalent to setting the property to Held. When a function-call input event reenables the chart, the chart maintains the most recent values of its states. For example, in this model, the Caller chart uses the event E to wake up and execute the Callee chart.
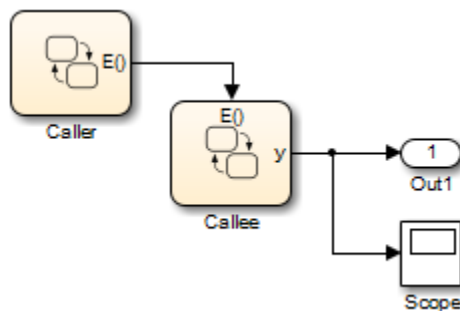
The `Caller` chart contains two states, A and B.
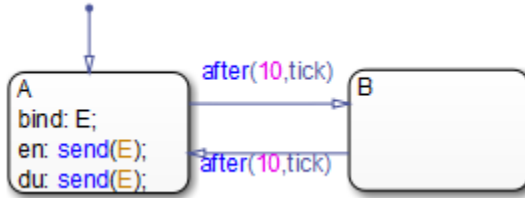
When you bind E to A:

- Entering A enables the `Callee` chart.
- Exiting A disables the `Callee` chart.
- Reentering A reenables the `Callee` chart.

Each time that the `Callee` chart executes, the output data y increments by one.

This table lists the key behaviors of the model.

Time Interval	Caller Chart	Callee Chart
$t = 0$ to $t = 10$	State A is active and enables `Callee`.	State A executes by incrementing y.
$t = 10$ to $t = 20$	State B is active and disables `Callee`.	State A does not execute.
$t = 20$ to $t = 30$	State A is active and reenables `Callee`.	State A executes by incrementing y.
$t = 30$ to $t = 40$	State B is active and disables `Callee`.	State A does not execute.
$t = 40$ to $t = 50$	State A is active and reenables `Callee`.	State A executes by incrementing y.

Set States When Enabling Property to Inherit or Held

In the Chart properties dialog box for `Callee`, **States When Enabling** is `Inherit`. Because the parent of this chart is the model root, the behavior is the same as when **States When Enabling** is `Held`. During simulation, the output y maintains its most recent value when the function-call input event reenables the chart at $t = 20$ and $t = 40$.
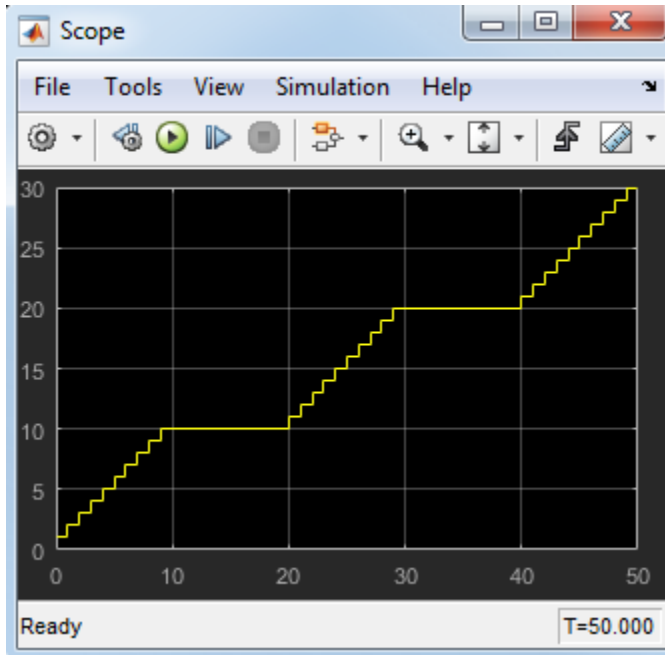
Set States When Enabling Property to Reset

Suppose that you change the **States When Enabling** property for `Callee` to `Reset`. During simulation, y reverts to its initial value of zero when the function-call input event reenables the chart at $t = 20$ and $t = 40$.
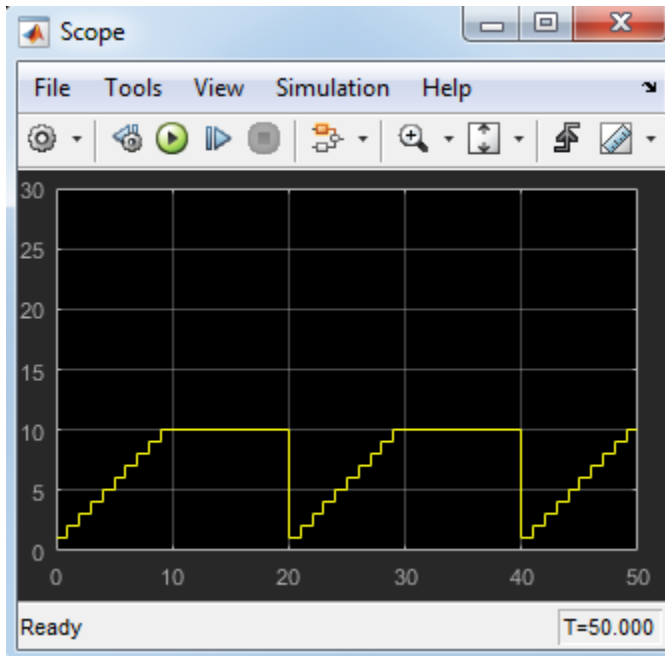
Behavior When Chart Is Inside Model Block

If the chart is inside a Model block, then setting **States When Enabling** to Inherit is equivalent to setting the property to Reset. When a function-call input event reenables the chart, the chart reverts to the initial values of its states. For example, this model contains a Model block.
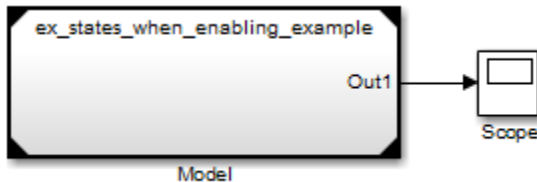
In the Model block, the Caller chart uses the event E to wake up and execute the Callee chart, as in the previous example.

This table lists the key behaviors of the model.

Time Interval	Caller Chart	Callee Chart
$t = 0$ to $t = 10$	State A is active and enables `Callee`.	State A executes by incrementing y.
$t = 10$ to $t = 20$	State B is active and disables `Callee`.	State A does not execute.
$t = 20$ to $t = 30$	State A is active and reenables `Callee`.	State A executes by incrementing y.
$t = 30$ to $t = 40$	State B is active and disables `Callee`.	State A does not execute.
$t = 40$ to $t = 50$	State A is active and reenables `Callee`.	State A executes by incrementing y.

Set States When Enabling Property to Inherit or Reset

In the Chart properties dialog box for `Callee`, **States When Enabling** is `Inherit`. Because this chart is inside a Model block, the behavior is the same as when **States When Enabling** is `Reset`. During simulation, the output y reverts to its initial value of zero when the function-call input event reenables the chart at $t = 20$ and $t = 40$.
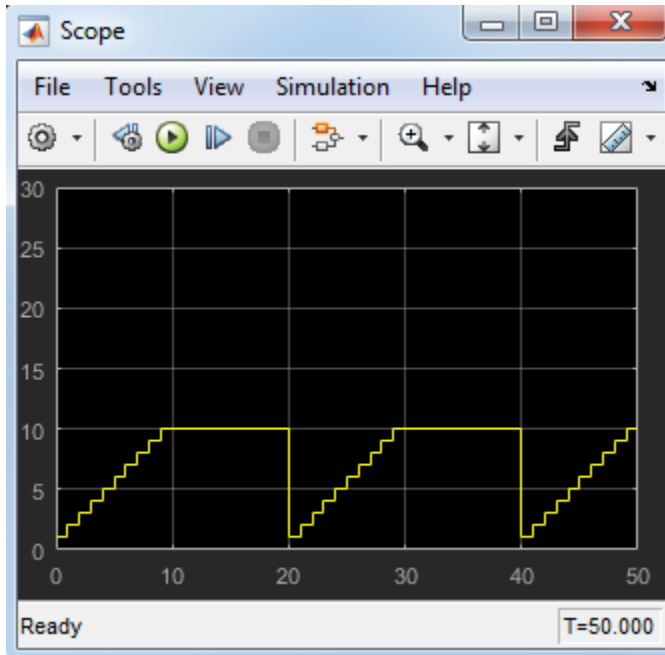
Set States When Enabling Property to Held

Suppose that you change the **States When Enabling** property for `Callee` to `Held`. During simulation, y maintains its most recent value when the function-call input event reenables the chart at $t = 20$ and $t = 40$.
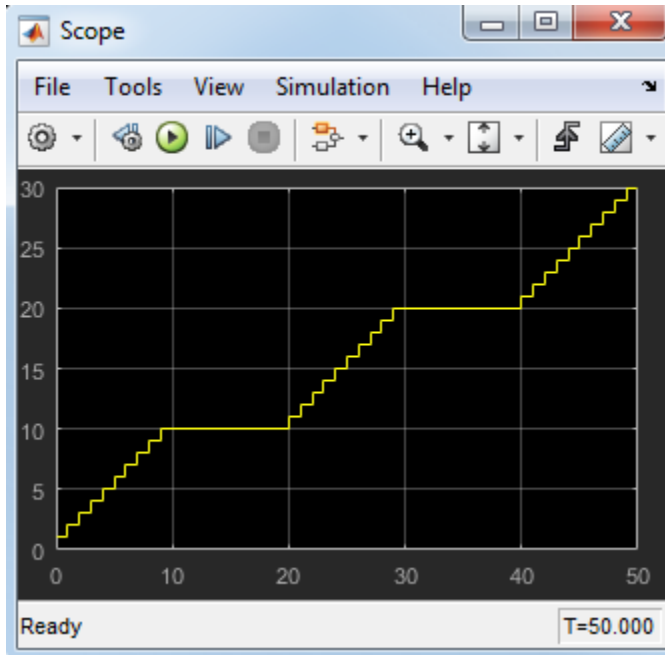
See Also

More About

- "Synchronize Model Components by Broadcasting Events" on page 10-2
- "Activate a Stateflow Chart by Sending Input Events" on page 10-9
- "Model References" (Simulink)

Activate a Simulink Block by Sending Output Events

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

1   Add an event to the Stateflow chart, as described in "Define Events in a Chart" on page 10-3.
2   Set the **Scope** property for the event to `Output to Simulink`. For each output event that you define, an output port appears on the Stateflow block.
3   An output event can activate other blocks in the model through a change in a control signal (an edge trigger) or a function call to a Simulink block.

   • To specify an edge-triggered output event, set the **Trigger** property to `Either Edge`.
   • To specify a function-call output event, set the **Trigger** property to `Function call`.

For more information, see "Synchronize Model Components by Broadcasting Events" on page 10-2.

Activate a Simulink Block by Using Edge Triggers

An edge-triggered output event activates a Simulink block to execute during the current time step of simulation. With this type of output event, a change in a control signal acts as a trigger. For more information, see "Using Triggered Subsystems" (Simulink).

When to Use Edge-Triggered Output Events

To activate a Simulink subsystem when your model requires regular or periodic subsystem execution, use an edge-triggered output event. For example, this model uses an edge-triggered output event to activate two triggered subsystems at regular intervals.
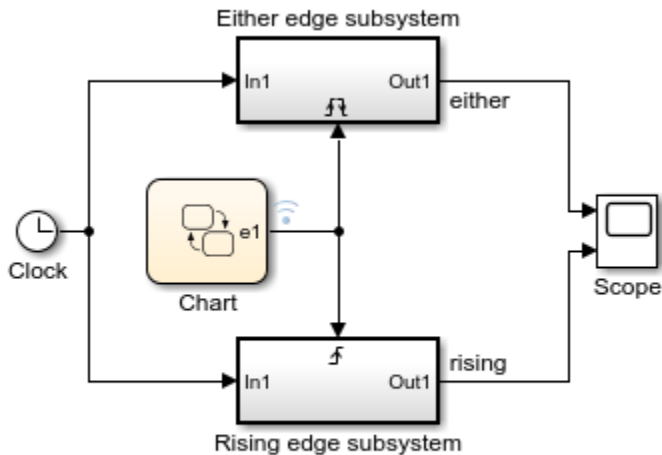
Copyright 2010-2018 The MathWorks, Inc.

The chart contains the edge-triggered output event `e1` which alternates between 0 and 1 during simulation.
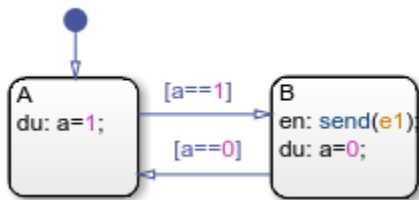
In a Stateflow chart, the **Trigger** property of an edge-triggered output event is always `Either Edge`. Simulink triggered subsystems can have a `Rising`, `Falling`, or `Either` edge trigger. The model shows the difference between triggering an `Either` edge subsystem from a `Rising` edge subsystem:

- The output event triggers the `Either` edge subsystem on every broadcast. The trigger occurs when the event signal switches from 0 to 1 or from 1 to 0.

- The output event triggers the `Rising` edge subsystem on every other broadcast. The trigger occurs only when the event signal switches from 0 to 1.
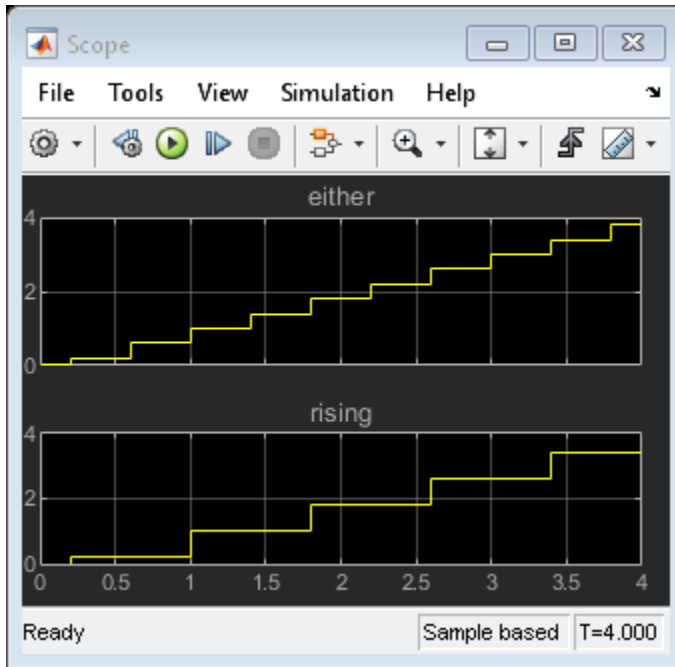
Queuing Behavior of Multiple Edge-Triggered Output Events

A chart dispatches only one broadcast of an edge-triggered output event for each time step. When there are multiple broadcasts in a single time step, the chart dispatches one broadcast and queues up the remaining broadcasts for dispatch in successive time steps. For example, in this model, the Caller chart uses the edge-triggered output event `output_cmd` to activate the Callee chart.
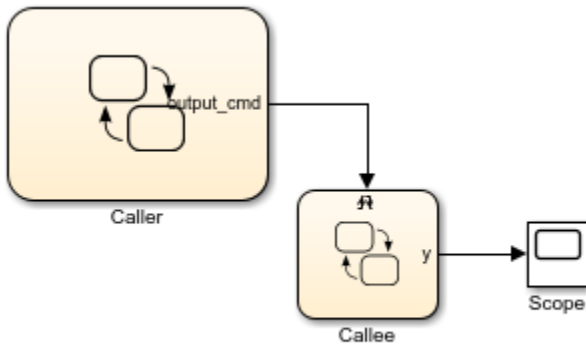
The Caller chart tries to broadcast the same edge-triggered output event four times in a single time step.
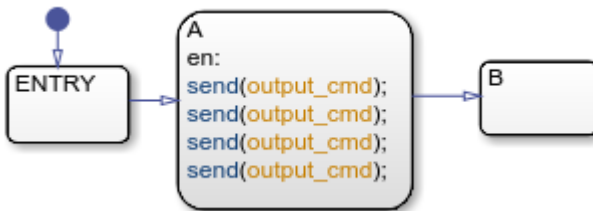
Each time the Callee chart is activated, the output data y increments by one.

When you simulate the model, at time $t = 1$, the Caller chart dispatches one of the four output events. The Callee chart executes once during that time step. The Caller chart queues up the other three event broadcasts for future dispatch at a time $t = 2$, $t = 3$, and $t = 4$. As a result, the value of y grows in increments of one at time $t = 1$, $t = 2$, $t = 3$, and $t = 4$.
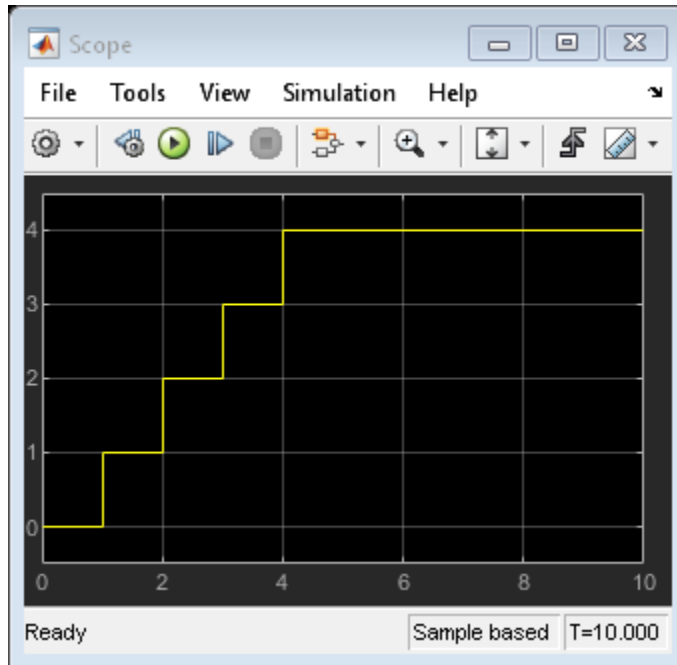
Activate a Simulink Block by Using Function Calls

A function-call output event activates a Simulink block to execute during the current time step of simulation. This type of output event works only on blocks that you can trigger with a function call. For more information, see "Using Function-Call Subsystems" (Simulink).

When to Use Function-Call Output Events

Use a function-call output event to activate a Simulink block when your model requires access to output data from the block in the same time step as the function call. For example, the model `sf_loop_scheduler` contains two function-call output events:

- In the Edge to Function chart, the output event `call` activates the Looping Scheduler chart.
- In the Looping Scheduler chart, the output event `A1` activates a Simulink subsystem.

For more information, see "Schedule A Subsystem Multiple Times in a Single Step" on page 26-18.
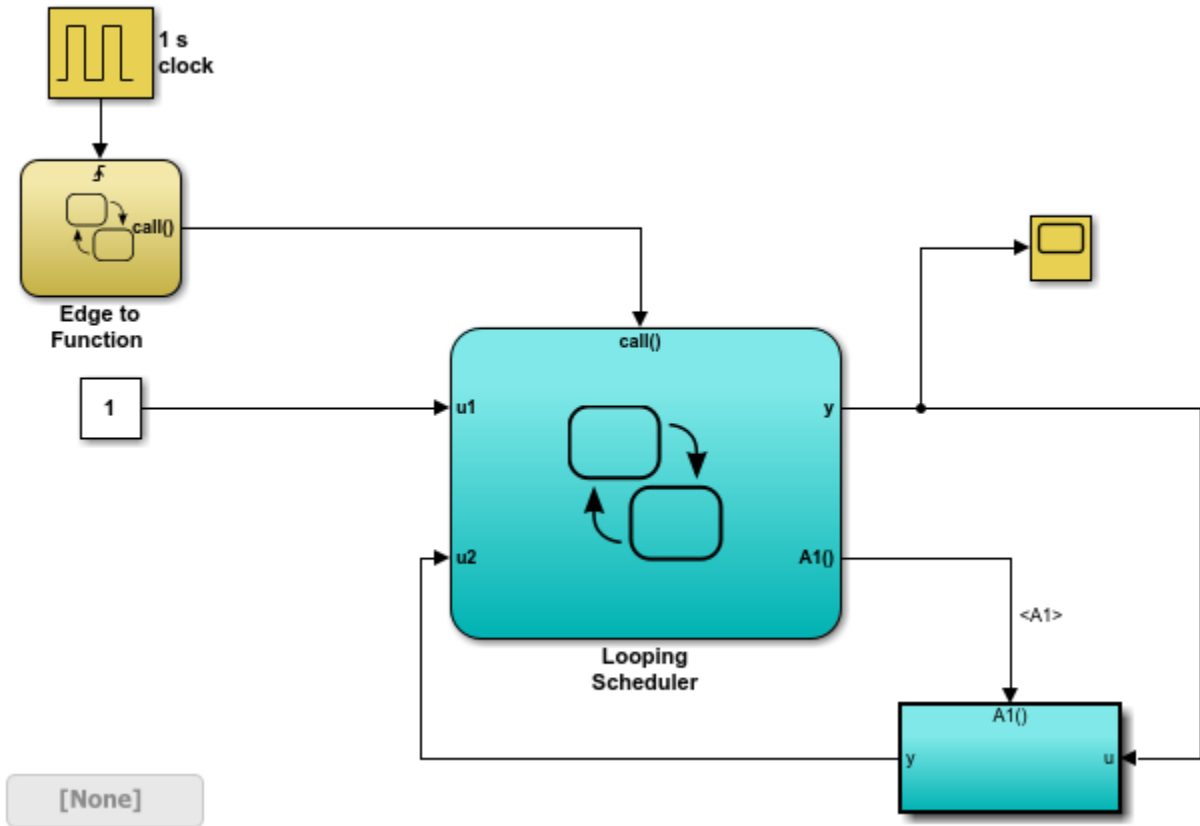
Interleaving Behavior of Multiple Function-Call Output Events

When there are multiple broadcasts of a function-call output event in a single time step, the chart dispatches all the broadcasts in that time step. Execution of function-call subsystems is interleaved with the execution of the chart, so that output from the function-call subsystem is available immediately in the chart. For example, in this model, the Caller chart uses the function-call output event `output_cmd` to activate the Callee chart.
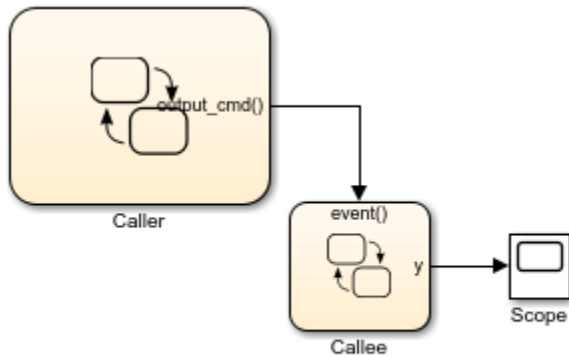
Copyright 2018 The MathWorks, Inc.

The Caller chart tries to broadcast the same function-call output event four times in a single time step.
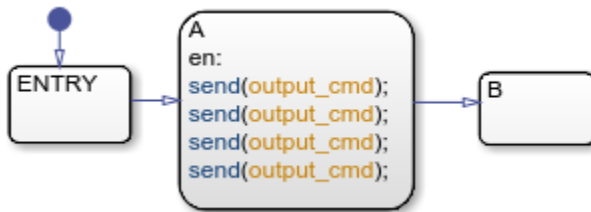
Each time the Callee chart is activated, the output data y increments by one.

When you simulate the model, the Caller chart dispatches all four output events at time t = 1. The Callee chart executes four times during that time step. Execution of the Callee chart is interleaved with execution of the Caller chart so that output from the Callee chart is immediately available. As a result, the value of | y | increases from 0 to 4 at time t = 1.
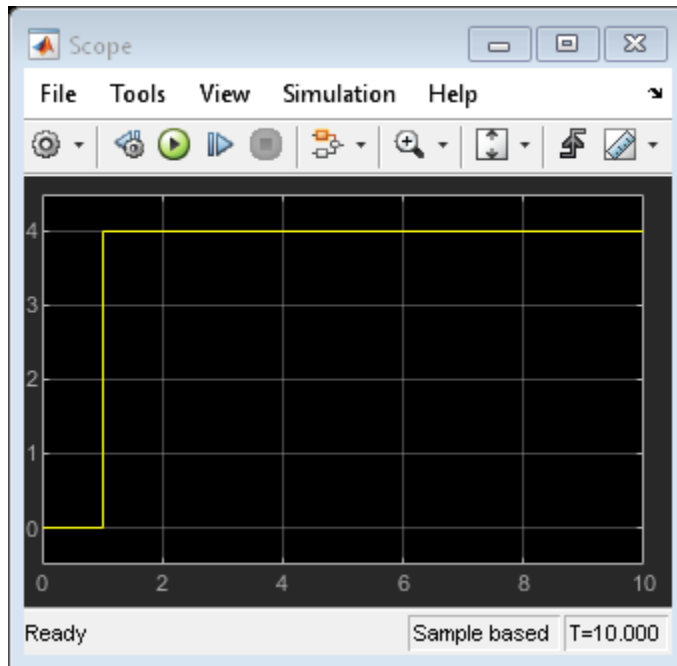
Approximate a Function Call by Using Edge-Triggered Events

If you cannot use a function-call output event, such as for HDL code generation, you can approximate a function call by using:

- An edge-triggered output event
- An enabled subsystem
- Two consecutive event broadcasts

The queuing behavior of consecutive edge-triggered output events enables you to approximate a function call with an enabled subsystem.

For example, in this model, the edge-triggered output event output_cmd activates the enabled subsystem.
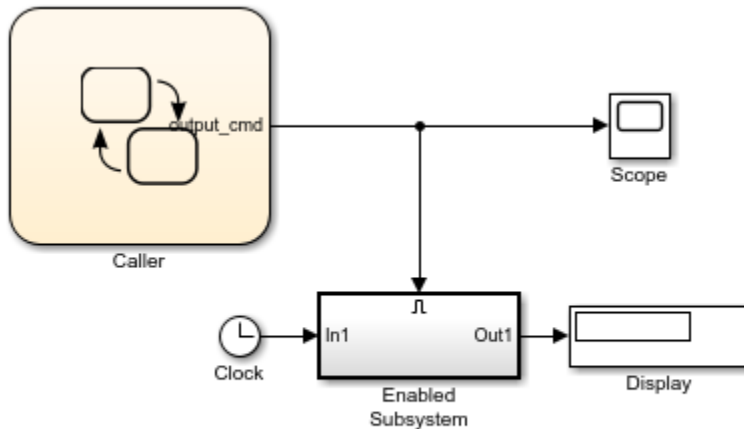
Copyright 2018 The MathWorks, Inc.

The Caller chart broadcasts the edge-triggered output event by using the `send` operator.
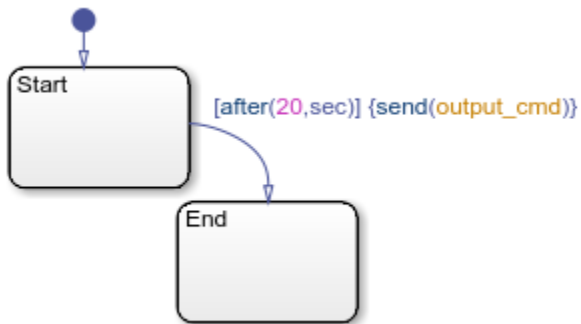
When simulation starts, the value of the trigger signal is 0. At time $t = 20$, the chart dispatches `output_cmd`, changing the value of the trigger signal to 1. The enabled subsystem becomes active and executes during that time step. Because no other event broadcasts occur, the enabled subsystem continues to execute at every time step until simulation ends at $t = 40$. The Display block shows a final value of 40.
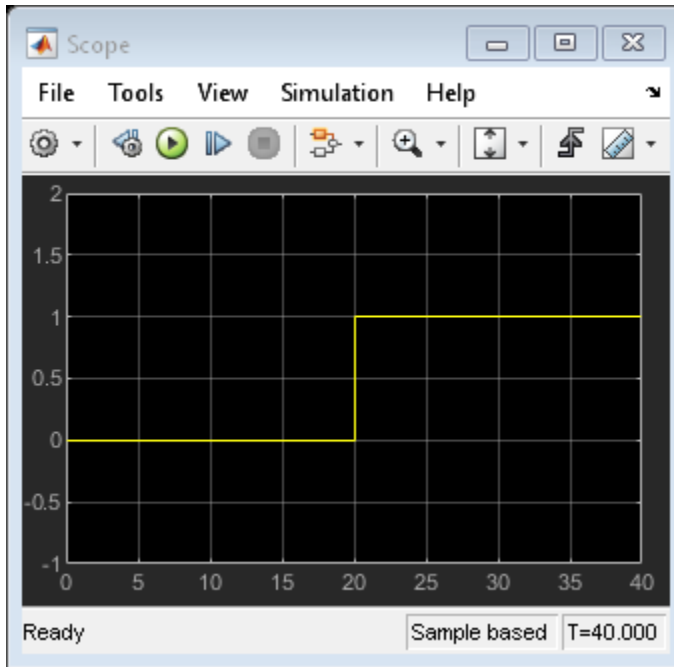
To approximate a function call, add a second event broadcast in the same action.
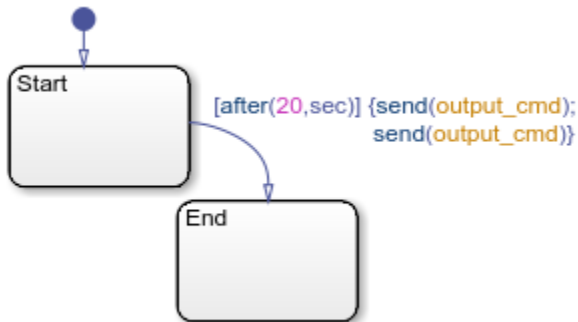
When simulation starts, the value of the trigger signal is 0. At time $t = 20$, the chart dispatches `output_cmd`, changing the value of the trigger signal to 1. The enabled subsystem becomes active and executes during that time step. The chart queues up the second event for dispatch at the next time step. At time $t = 21$, the chart dispatches the

second output event, which changes the value of the trigger signal back to 0. The enabled subsystem stops executing and the Display block shows a final value of 20.
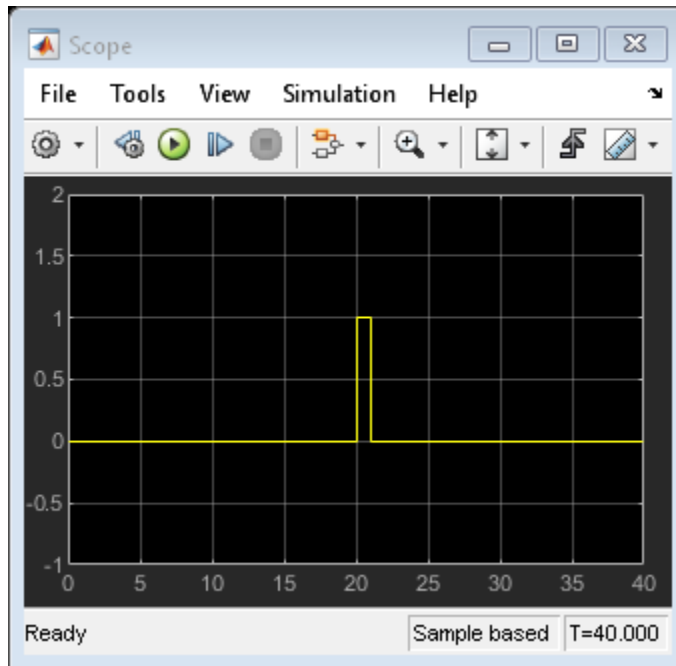
Although you can approximate a function call, there is a subtle difference in execution behavior. Execution of a function-call subsystem occurs *during* execution of the chart action that provides the trigger. Execution of an enabled subsystem occurs *after* execution of the chart action is complete.

Association of Output Events with Output Ports

When you define an output event in a chart, an output event port appears on the right side of a chart block. Output events must be scalar, but you can define multiple output events in a chart. The **Port** property of an output event specifies the position of the output port.

By default, **Port** values appear in the order in which you add output events. You can change these assignments by modifying the **Port** property of the events. When you change the **Port** property for an output event, the **Port** values for the remaining output events automatically renumber.

See Also

send

More About

- "Synchronize Model Components by Broadcasting Events" on page 10-2
- "Set Properties for an Event" on page 10-6
- "Schedule A Subsystem Multiple Times in a Single Step" on page 26-18
- "View Differences Between Stateflow Messages, Events, and Data" on page 11-2
- "Using Triggered Subsystems" (Simulink)
- "Using Function-Call Subsystems" (Simulink)

Define Chart Behavior by Using Implicit Events

Implicit events are built-in events that occur when a chart executes:

- Chart waking up
- Entry into a state
- Exit from a state
- Value assigned to an internal data object

These events are implicit because you do not define or trigger them explicitly. Implicit events are children of the chart in which they occur and are visible only in the parent chart.

Keywords for Implicit Events

To reference implicit events, action statements use this syntax:

```
event(object)
```

where `event` is the name of the implicit event and `object` is the state or data in which the event occurs.

Each keyword below generates implicit events in the action language notation for states and transitions.

Implicit Event	Meaning
change(*data_name*) or chg(*data_name*)	Specifies and implicitly generates a local event when Stateflow software writes a value to the variable *data_name*. Supported only in Stateflow charts in Simulink models. The variable *data_name* cannot be machine-parented data. This implicit event works only with data that is at the chart level or lower in the hierarchy. For machine-parented data, use change detection operators to determine when the data value changes. For more information, see "Detect Changes in Data Values" on page 12-69.

Implicit Event	Meaning
enter(*state_name*) or en(*state_name*)	Specifies and implicitly generates a local event when the specified *state_name* is entered. Supported only in Stateflow charts in Simulink models.
exit(*state_name*) or ex(*state_name*)	Specifies and implicitly generates a local event when the specified *state_name* is exited. Supported only in Stateflow charts in Simulink models.
tick	Specifies and implicitly generates a local event when the chart of the action being evaluated awakens.
wakeup	Equivalent to tick. Supported only in Stateflow charts in Simulink models.

If more than one object has the same name, use the dot operator to qualify the name of the object with the name of its parent. These examples are valid references to implicit events:

```
enter(switch_on)
en(switch_on)
change(engine.rpm)
```

Note The tick (or wakeup) event refers to the chart containing the action being evaluated. The event cannot refer to a different chart by argument.

Transition Between States Using Implicit Events

This example illustrates use of implicit tick events.
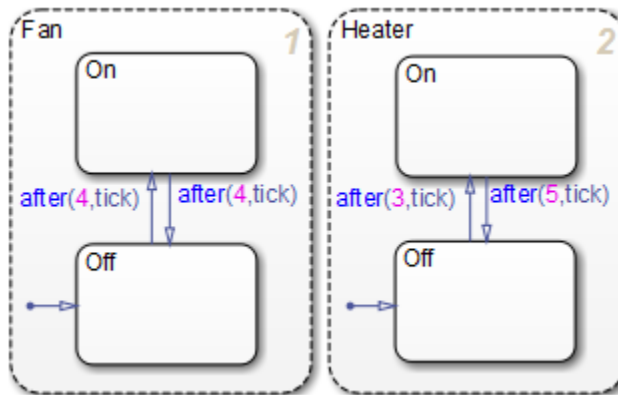
Fan and `Heater` are parallel (AND) superstates. The first time that an event awakens the Stateflow chart, the states `Fan.Off` and `Heater.Off` become active.

Assume that you are running a discrete-time simulation. Each time that the chart awakens, a `tick` event broadcast occurs. After four broadcasts, the transition from `Fan.Off` to `Fan.On` occurs. Similarly, after three broadcasts, the transition from `Heater.Off` to `Heater.On` occurs.

For information about the `after` operator, see "Control Chart Execution by Using Temporal Logic" on page 12-49.

Execution Order of Transitions with Implicit Events

Suppose that:

- Your chart contains parallel states.
- In multiple parallel states, the same implicit event is used to guard a transition from one substate to another.

When multiple transitions are valid in the same time step, the transitions execute based on the order in which they were created in the chart. This order does not necessarily match the activation order of the parallel states that contain the transitions. For example, consider the following chart:
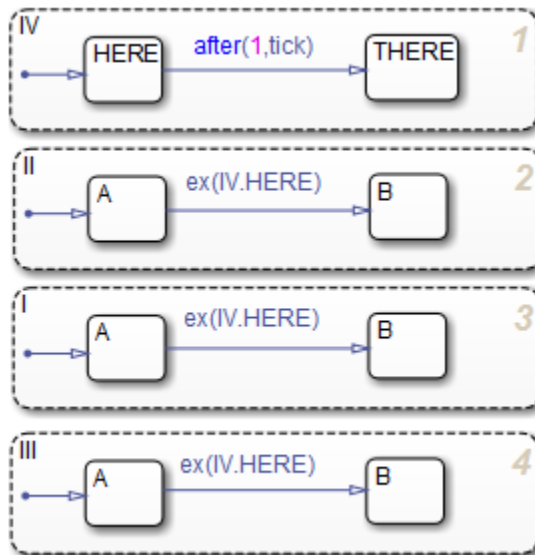
When the transition from IV.HERE to IV.THERE occurs, the condition ex(IV.HERE) is valid for the transitions from A to B for the parallel states I, II, and III. The three transitions from A to B execute in the order in which they were created: in state I, then II, and finally III. This order does not match the activation order of those states.

To ensure that valid transitions execute in the same order that the parallel states become active, use the in operator instead of implicit enter or exit events:
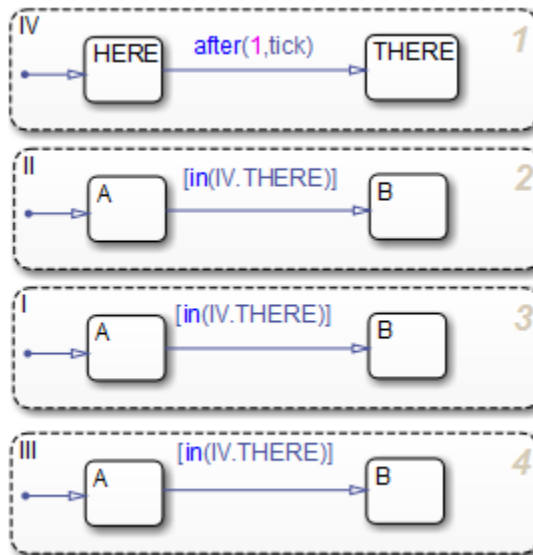
With this modification, the transitions from A to B occur in the same order as activation of the parallel states. For more information about the in operator, see "Check State Activity by Using the in Operator" on page 12-77.

Count Events by Using the temporalCount Operator

You can count occurrences of explicit and implicit events by using the `temporalCount` operator. For more information, see "Operators for Event-Based Temporal Logic" on page 12-50.

Collect and Store Input Data in a Vector

This model collects and stores input data in a vector during chart simulation:
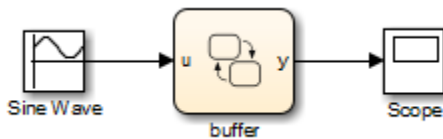
The chart contains two states and one MATLAB function:
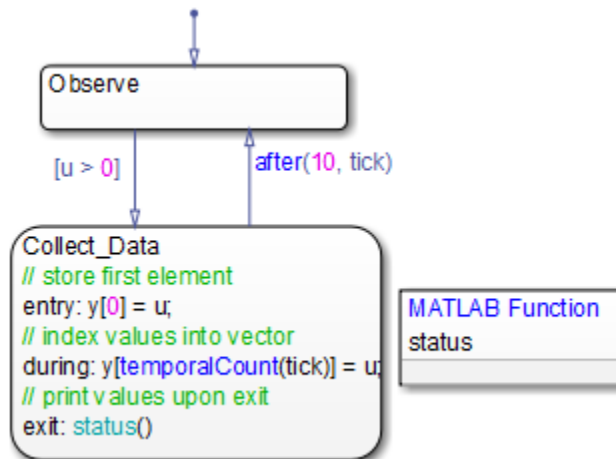
Stage 1: Observation of Input Data

The chart awakens and remains in the `Observe` state, until the input data `u` is positive. Then, the transition to the state `Collect_Data` occurs.

Stage 2: Storage of Input Data

After the state `Collect_Data` becomes active, the value of the input data u is assigned to the first element of the vector y. While this state is active, each subsequent value of u is assigned to successive elements of y using the `temporalCount` operator.

Stage 3: Display of Data Stored in the Vector

After 10 ticks, the data collection process ends, and the transition to the state `Observe` occurs. Just before the state `Collect_Data` becomes inactive, a function call to `status` displays the vector data at the MATLAB prompt.

For more information about ticks in a Stateflow chart, see "Define Chart Behavior by Using Implicit Events" on page 10-33.

11

Messages

View Differences Between Stateflow Messages, Events, and Data

This example compares the behavior of messages, events, and data in Stateflow®.
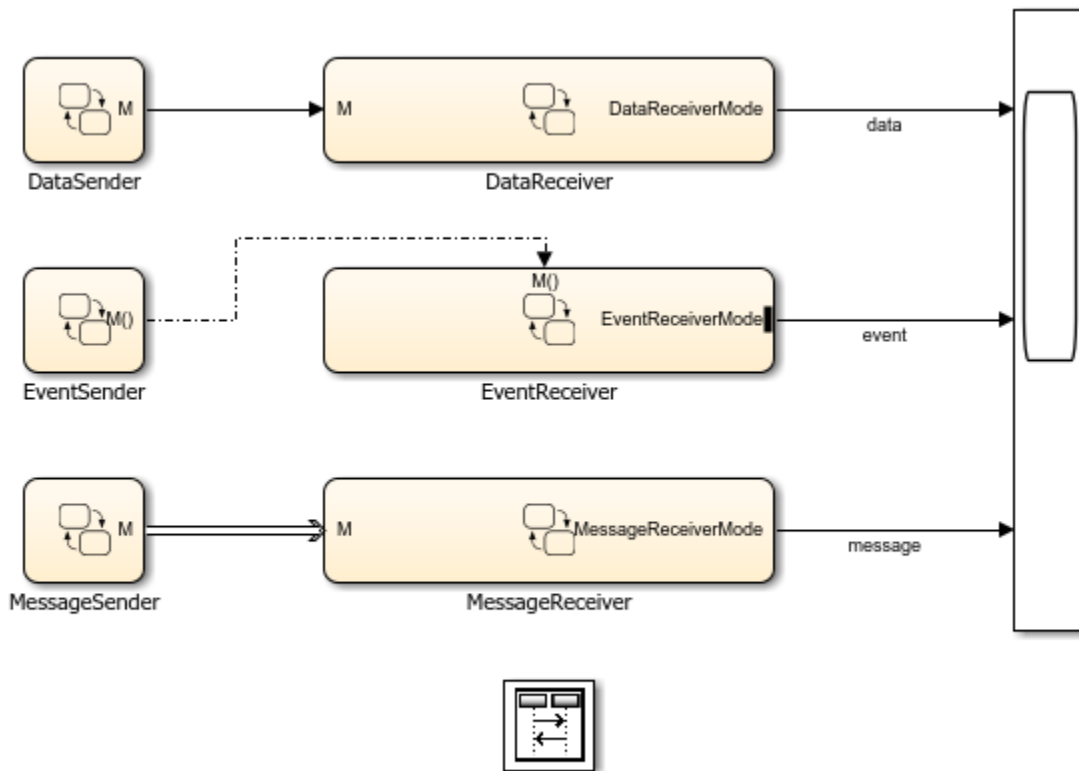
Copyright 2015-2018 The MathWorks, Inc.

Sender Charts

This model has three sender charts: `DataSender`, `EventSender`, and `MessageSender`. Each sender chart has one state. In the entry action of the state, the charts assign a value to data, send a function-call event, or send a message.
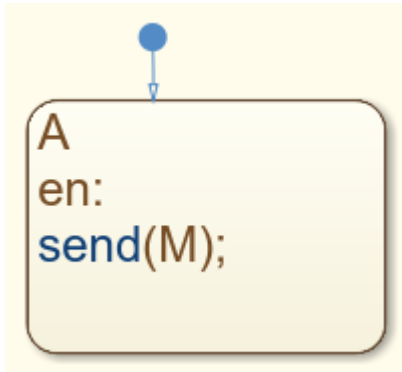
Receiver Charts

For each of the sender charts, there is a corresponding receiver chart. Each receiver chart has a state diagram with states A0, A1, A2, and A3. The implicit event after(3,sec) triggers the transition from A0 to A1. The data, event, or message from the corresponding sender chart guards the transitions between A1, A2, and A3.
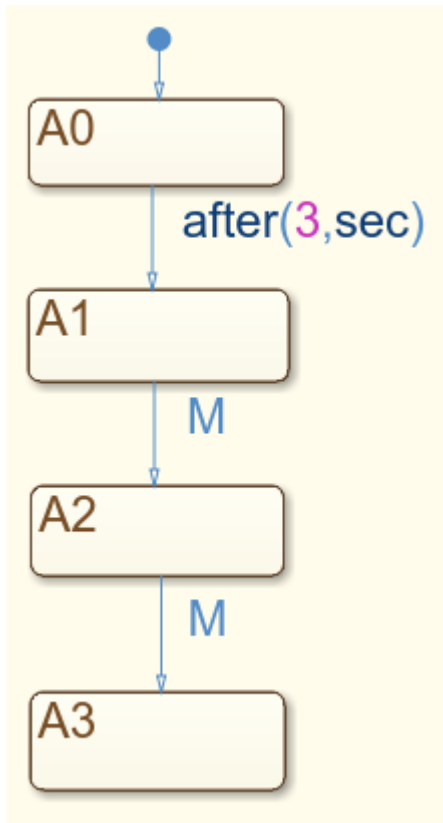
Scope Output

Each receiver chart has active state output enabled and connected to a scope. The scope shows which states are active in each time step. This output highlights the difference in behavior between output data, events, and messages.
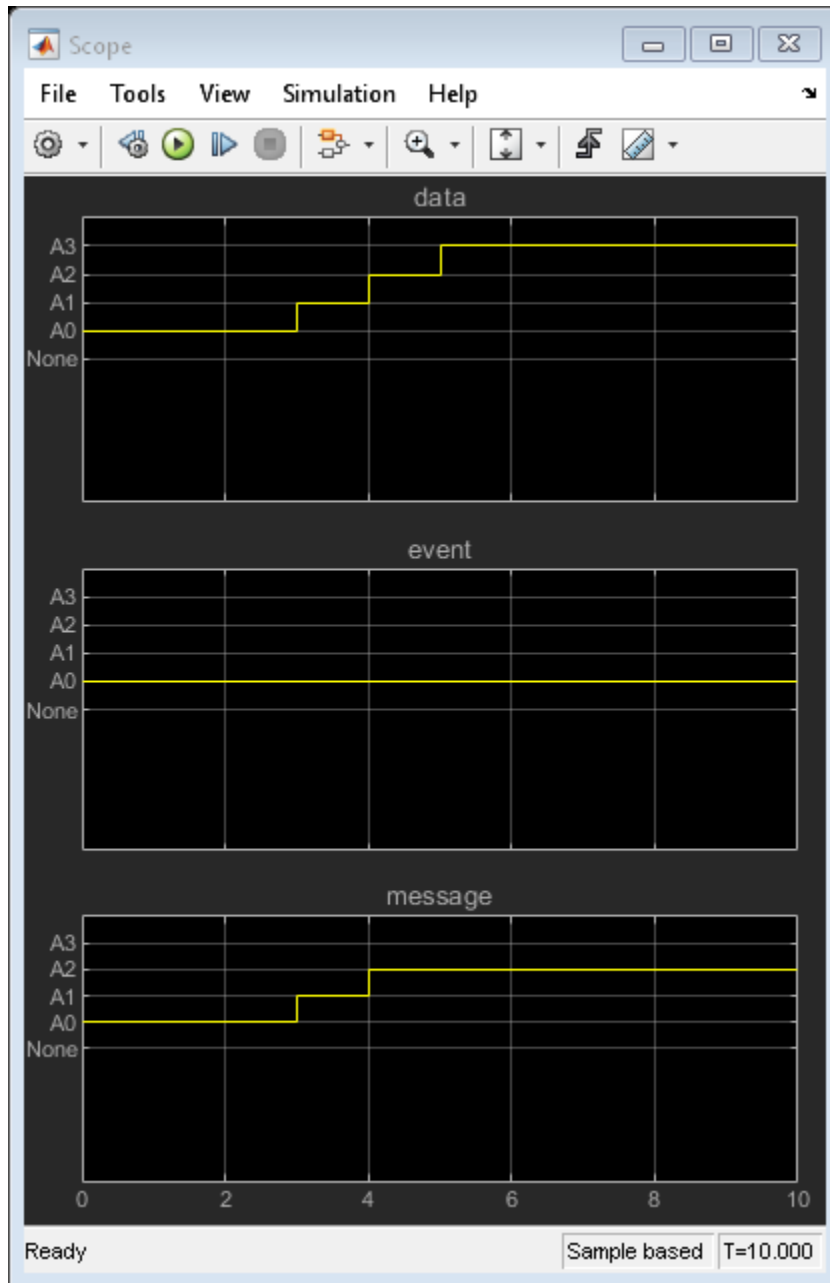
Behavior of Data

The DataSender chart assigns a value of 1 to the output data M, which connects as an input to the DataReceiver chart.

The DataReceiver chart executes once at every time step. At the start of simulation, state A0 is active. At time t=3, the transition from A0 to A1 occurs. At time t=4, the chart tests whether M equals 1. This condition is true, so the chart transitions from A1 to A2. At time t=5, M still equals 1, so the chart transitions from A2 to A3. On the scope, you see that DataReceiver changes states three times.

After data is assigned a value, it holds its value throughout the simulation. Therefore, each time that the DataReceiver evaluates the condition [M == 1], it transitions to a new state.
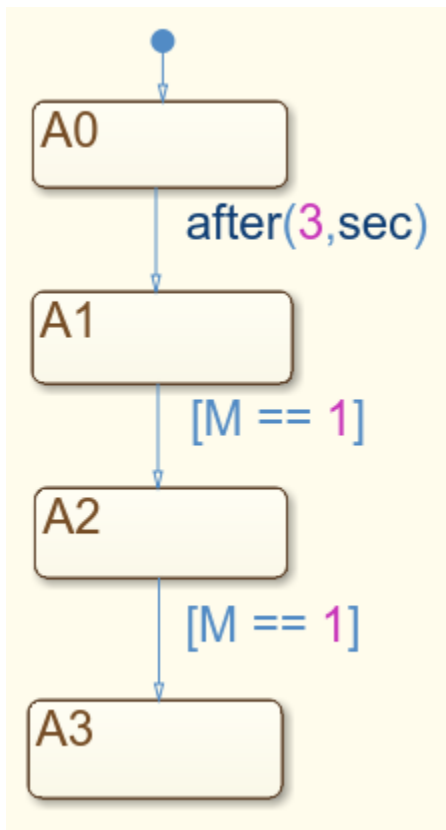
Behavior of Event

The `EventSender` chart uses the command `send(M)` to send a function-call output event to wake up the `EventReceiver` chart.

The `EventReceiver` chart executes only when the input event M wakes up the chart. At the start of simulation, state A0 is active. The transition from A0 to A1 is based on absolute-time temporal logic and is not valid at time `t=0`. A0 remains active and the chart goes back to sleep. Because `EventSender` sends the event M only once, `EventReceiver` does not wake up again. On the scope, you see that `EventReceiver` never transitions out of A0.

Events do not remain valid across time steps, so the receiving chart has only one chance to respond to the event. When `EventSender` sends the event, `EventReceiver` is not ready to respond to it. The opportunity for `EventReceiver` to transition in response to the event is lost.
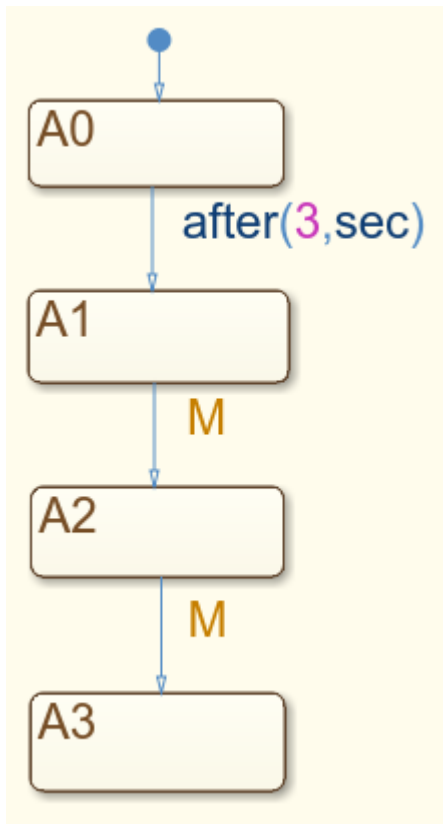
Behavior of Message

The `MessageSender` chart uses the syntax `send(M)` to send a message through the output message port. The message goes into the input message queue of the `MessageReceiver` chart. The message waits in the queue until `MessageReceiver` evaluates it.

The `MessageReceiver` chart executes once at every time step. At the start of simulation, state A0 is active. At time `t=3`, the transition from A0 to A1 occurs. At time `t=4`, the chart determines that M is present in the queue, so it takes the transition to A2. At the end of the time step, the chart removes M from the queue. At time `t=5`, there is no message present in the queue, so the chart does not transition to A3. A2 remains the active state. On the scope, you see that `MessageReceiver` changes state only two times.

Unlike events, messages are queued. The receiving chart can choose to respond to a message anytime after it was sent. Unlike data, the message does not remain valid indefinitely. The message is destroyed at the end of the time step.
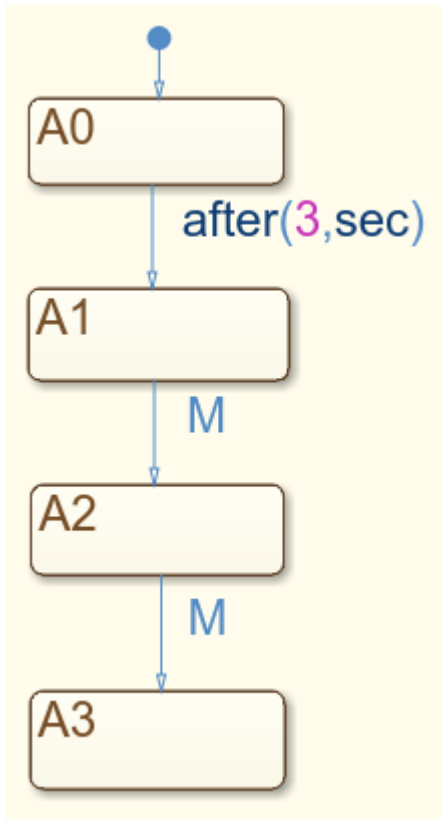
See Also

More About

- "Share Data with Simulink and the MATLAB Workspace" on page 9-23
- "Synchronize Model Components by Broadcasting Events" on page 10-2
- "Communicate with Stateflow Charts by Sending Messages" on page 11-10

Communicate with Stateflow Charts by Sending Messages

To communicate within and between Stateflow charts in a Simulink model, use messages. A message is a Stateflow object that communicates data locally or between charts. From a sender chart, you can send or forward a message. In the receiving chart, a queue receives the message and holds it until the chart can evaluate it.
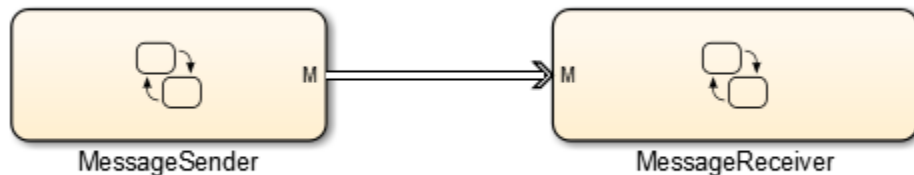
Messages are queued until the chart wakes up. When the chart wakes up, it can respond to the messages in the queue.

- Messages do not trigger charts to wake up.
- Messages are not lost when the receiver chart cannot immediately respond.

When a chart transition or state action evaluates the message, the chart determines if the queue contains any messages. If it does, the chart removes the message from the queue. The message remains valid until the end of the time step or until the chart forwards or discards it. While the message is valid, other transitions or actions can access the message data and the chart does not remove another message from the queue. The chart destroys all valid messages at the end of the current time step.

Define Messages in a Chart

You can add messages to a Stateflow chart by using the **Chart** menu in the Stateflow Editor, through the Symbols window, or through the Model Explorer.

Add Messages by Using the Stateflow Editor Menu

1 In the Stateflow Editor, select the menu option corresponding to the scope of the message that you want to add.

Scope	Menu Option
Input	**Chart > Add Inputs & Outputs > Message Input From Simulink**
Output	**Chart > Add Inputs & Outputs > Message Output To Simulink**
Local	**Chart > Add Other Elements > Local Message**

2 In the Message dialog box, specify data properties. For more information, see "Set Properties for a Message" on page 11-14.

Add Messages Through the Symbols Window

1 To open the Symbols window, select **View > Symbols**.

2 Click the **Create Message** icon .

3 In the row for the new message, under **TYPE**, click the icon and choose:

- `Input Message`
- `Local Message`
- `Output Message`

4 Edit the name of the message.

5 For input and output messages, click the **PORT** field and choose a port number.

6 To specify properties for the message, open the Property Inspector. In the Symbols window, right-click the row for the message and select **Explore**. For more information, see "Set Properties for a Message" on page 11-14.

Add Messages Through the Model Explorer

1 In the Stateflow Editor, select **View > Model Explorer**.

2 In the **Model Hierarchy** pane, select the object in the Stateflow hierarchy where you want to make the new message visible. The object that you select becomes the parent of the new message.

3   In the Model Explorer menu, select **Add > Message**. The new message with a default definition appears in the **Contents** pane of the Model Explorer.

4   In the **Message** pane, specify the properties of the message. For more information, see "Set Properties for a Message" on page 11-14.

Lifetime of a Stateflow Message

A Stateflow message has a finite lifetime. The lifetime begins when you send a message to an input or local queue with the `send` operator. The message remains in the queue until a transition or state `on` action evaluates it or the chart receives it by using the `receive` operator.

A message becomes valid when a chart evaluates or receives it. The message remains valid until:

• The end of the current time step, when the chart destroys any remaining valid messages.

• The chart forwards the message to another queue. The message continues its lifetime in the receiving queue.

• The chart discards the message.

While a message is valid, other transitions and actions can evaluate the message and access its data. To check if a message is valid, use the `isvalid` operator.

To view the interchange of messages during simulation, add a Sequence Viewer block to your Simulink model. The Sequence Viewer block displays:

• Sent messages

• Received messages

• Forwarded messages

• Dropped messages

• Destroyed messages

• Discarded messages

For more information, see "Use the Sequence Viewer Block to Visualize Messages, Events, and Entities" on page 11-28.

Limitations for Messages

You cannot use messages in:

- Moore charts
- Atomic subcharts
- Breakpoint condition expressions
- Model reference inputs and outputs

Messages do not support fixed-point data.

See Also

Related Examples
- "View Differences Between Stateflow Messages, Events, and Data" on page 11-2

More About
- "Set Properties for a Message" on page 11-14
- "Control Message Activity in Stateflow Charts" on page 11-18
- "Send Messages with String Data" on page 20-14
- "Use the Sequence Viewer Block to Visualize Messages, Events, and Entities" on page 11-28

Set Properties for a Message

A message is a Stateflow object that communicates data locally or between charts in a Simulink model. For more information, see "Communicate with Stateflow Charts by Sending Messages" on page 11-10.

You can specify message properties in either the Property Inspector or the Model Explorer.

- Property Inspector

    1   Open the Symbols window by selecting **View** > **Symbols**.

    2   Open the Property Inspector by selecting **View** > **Property Inspector**.

    3   In the Symbols window, select the message.

    4   In the Property Inspector window, edit the message properties.

- Model Explorer

    1   Open the Model Explorer by selecting **View** > **Model Explorer**.

    2   In the **Contents** pane, double-click the message.

    3   In the **Message** pane, edit the message properties.

For more information, see "Communicate with Stateflow Charts by Sending Messages" on page 11-10.

Stateflow Message Properties

Name

Name of the message. For more information, see "Rules for Naming Stateflow Objects" on page 2-4.

Scope

Scope of the message. The scope specifies where the message occurs relative to the parent object.

Scope	Description
`Input from Simulink`	Message that is received from another Stateflow chart. Each input message has a receiving queue.
`Output to Simulink`	Message that is sent through an output port to another Stateflow chart.
`Local`	Message that is local to the Stateflow chart. A local message has a receiving queue with the same properties as an input message queue. When you send a local message, a transition or action in the same chart can evaluate the local message. You cannot send a local message outside the chart.

Port

Index of the port associated with the message. This property applies only to input and output messages.

Size

Size of the message data field. For more information, see "Specify Size of Stateflow Data" on page 9-40.

Complexity

Specifies whether the message data field accepts complex values.

Complexity Setting	Description
`Off`	Data field does not accept complex values.
`On`	Data field accepts complex values.
`Inherited`	Data field inherits the complexity setting from a Simulink block.

The default value is `Off`. For more information, see "Complex Data in Stateflow Charts" on page 23-2.

Type

Type of the message data field. To specify the data type:

• From the **Type** drop-down list, select a built-in type.

- In the **Type** field, enter an expression that evaluates to a data type.
- In the Model Explorer, use the Data Type Assistant to specify a data **Mode**, and then specify the data type based on that mode. To display the Data Type Assistant, click the **Show data type assistant** button [»]. The Data Type Assistant is available only in the Model Explorer.

Messages do not support fixed-point data.

Add to Watch Window

Enables watching the message queue and data field in the Stateflow Breakpoints and Watch window. For more information, see "Watch Stateflow Data Values" on page 32-36.

Queue Capacity

For input and local messages, specifies the maximum number of messages held in the queue. If a chart sends a message when the receiving queue is full, a message overflow occurs. To avoid dropped messages, set the queue capacity high enough so incoming messages do not cause the queue to overflow. The maximum queue length is 2^{16}–1.

Queue Overflow Diagnostic

Specifies the diagnostic action when the number of incoming messages exceeds the queue capacity. The default option is Error.

Diagnostic Setting	Description
Error	When the queue overflows, simulation stops with an error.
Warning	When the queue overflows, it drops the last message and simulation continues with a warning.
None	When the queue overflows, it drops the last message and simulation continues without issuing a warning.

Queue Type

Specifies the order in which messages are removed from the queue. The default option is FIFO.

Queue Type Setting	Description
FIFO	First In, First Out
LIFO	Last In, First Out
Priority	Remove messages according to value in the data field. Choose **Priority order** from these options: • Ascending. The order of the message removed is based on an ascending order of the message data value. • Descending. The order of the message removed is based on a descending order of the message data value.

See Also

More About

- "Communicate with Stateflow Charts by Sending Messages" on page 11-10
- "Control Message Activity in Stateflow Charts" on page 11-18
- "Rules for Naming Stateflow Objects" on page 2-4
- "Specify Size of Stateflow Data" on page 9-40
- "Specify Type of Stateflow Data" on page 9-33
- "Complex Data in Stateflow Charts" on page 23-2
- "Watch Stateflow Data Values" on page 32-36

Control Message Activity in Stateflow Charts

A message is a Stateflow object that communicates data locally or between charts in a Simulink model. From a sender chart, you can send or forward a message. In the receiving chart, a queue receives the message and holds it until the chart can evaluate it.
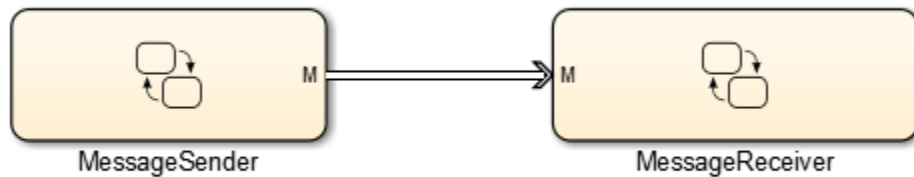
Using Stateflow operators, you can access message data, and send, receive, discard, or forward a message. You can also determine whether a message is valid and find the number of messages in a queue. For more information, see "Communicate with Stateflow Charts by Sending Messages" on page 11-10.

Access Message Data

Stateflow messages have a data field. To read or write to the message data field of a valid message, use dot notation syntax:

message_name`.data`

If you send a message without first assigning a value to the message data, the default value for numeric data is 0. For enumerated data, the default is the first value listed in the `enumeration` section of the definition, unless you specify otherwise in the `methods` section of the definition.

You cannot access message data for messages that are still in the queue or that have already been discarded.

Send a Message

To send a message, use the `send` operator:

```
send(message_name)
```

For example, in this chart, the entry action in state A sends a message M with a data value of 3. If the message is Local, then the message goes in the local message queue. If the message scope is Output, then the chart sends the message through the output port to the input message queue of the receiving chart.
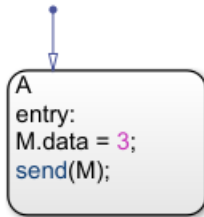
In a single time step, you can send multiple messages through an output port or to a local queue.

Guard Transitions and Actions

Messages can guard transitions or state actions of type on. During a time step, when the guarding message is evaluated for the first time, the chart removes the message from the queue and makes the message valid. While the message is valid, other transitions or actions can access the message data but they do not remove another message from the queue.

Guard a Transition with a Message

In this chart, a message M guards the transition from state A to state B. The transition occurs when both of these conditions are true:

- A message is present in the queue.
- The data value of the message is equal to 3.

If a message is not present or if the data value is not equal to 3, then the transition does not occur. If a message is present, it is removed from the queue regardless of whether the transition occurs.

Guard a State on Action with a Message

In this chart, a message M guards the on action in state A. When state A becomes active, it increments the value of x if both of these conditions are true:

- A message is present in the queue.
- The data value of the message is equal to 3.

If a message is not present or if the data value is not equal to 3, then the value of x does not change. If a message is present, it is removed from the queue regardless of whether x is modified.
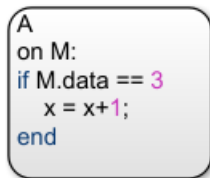
Receive a Message

To receive a message, use the receive operator:

receive(*message_name*)

If a valid message M exists, receive(M) returns true. If a valid message does not exist but there is a message in the queue, then the chart removes the message from the queue and receive(M) returns true. If a valid message does not exist and there are no messages in the queue, receive(M) returns false.

For example, in this chart, the during action in state A checks the queue for message M and increments the value of x if both of these conditions are true:

- A message is present in the queue.
- The data value of the message is equal to 3.

If a message is not present or if the data value is not equal to 3, then the value of x does not change. If a message is present, it is removed from the queue regardless of whether x is modified.

```
A
during:
if receive(M) && M.data == 3
   x = x+1;
end
```

Discard a Message

To discard a valid message, use the `discard` operator:

```
discard(message_name)
```

After a chart discards a message, it can remove another message from the queue in the same time step. A chart cannot access the data of a discarded message.

For example, in this chart, the `during` action in state A checks the queue for message M. If a message is present, the chart removes it from the queue. If the message has a data value equal to 3, the chart discards the message.

```
A
during:
if receive(M) == true
   if M.data == 3
      discard(M);
   end
end
```

11-21
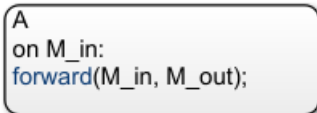
Forward a Message

To forward a message from an input queue to an output port, or to and from local message queues, use the `forward` operator:

`forward(`*input_message_name*`, `*output_message_name*`)`

After a chart forwards a message, it can remove another message from the queue in the same time step.

Forward an Input Message

In this chart, state A checks the input queue for message `M_in`. If a message is present, the chart removes the message from the queue and forwards it to the output port `M_out`. After the chart forwards the message, the message is no longer valid in state A.

Forward a Local Message

In this chart, state A checks the input queue for message `M_in`. If a message is present, the chart forwards the message to the local message queue `M_local`. After a delay of 0.3 seconds, the transition from state A to state B removes the message from the `M_local` message queue and forwards it to the output port `M_out`.
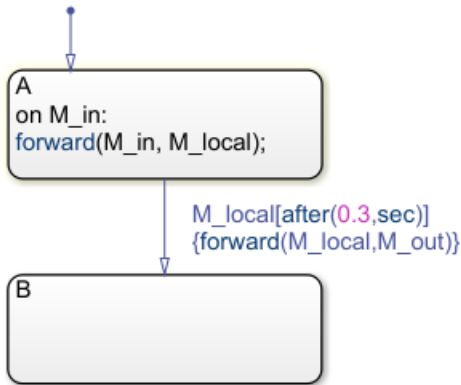
Determine if a Message Is Valid

To check if a message is valid, use the `isvalid` operator:

`isvalid(message_name)`

A message is valid if the chart has removed it from the queue and has not forwarded or discarded it. Use the `isvalid` operator to check if a message is valid in a Simulink model that contains more than one Stateflow chart.

For example, this chart first executes state A, as described in "Discard a Message" on page 11-21. When the chart executes state B, the `during` action checks whether the message M is valid. If the message is valid and has a data value equal to 6, the chart discards the message.
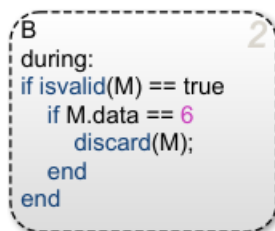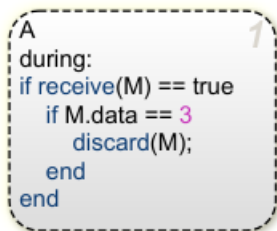
Determine the Length of the Queue

To check the number of messages in a message queue, use the `length` operator:

length(*message_name*)

For example, in this chart, the `during` action in state `A` checks the queue for message `M`. If a message is present, the chart removes it from the queue. If exactly seven messages remain in the queue, the chart increments the value of `x`.

```
A
during:
if receive(M) == true
    if length(M) == 7
        x = x+1;
    end
end
```
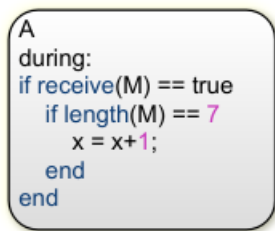
Determine When a Queue Overflows

To check whether a message is lost because it was sent to a queue that was already full, use the `overflowed` operator:

overflowed(*message_name*)

In each time step, the value of this operator is set when a chart adds a message to, or removes a message from, a queue. It is invalid to use the `overflowed` operator before sending or retrieving a message in the same time step.

By default, when a message queue overflows, simulation stops with an error. To prevent a run-time error and allow the `overflowed` operator to dynamically react to dropped messages, set the value of the **Queue Overflow Diagnostic** property to `Warning` or `None`. For more information, see "Queue Overflow Diagnostic" on page 11-16.

Check for Input Message Overflow

To check the overflow status of an input message queue, first remove a message from the queue. You can:

- Guard a transition with the message and the `overflowed` operator.
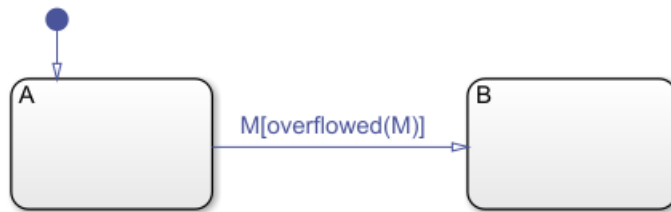
- Guard a transition with the message and call the `overflowed` operator in the entry action of the destination state.
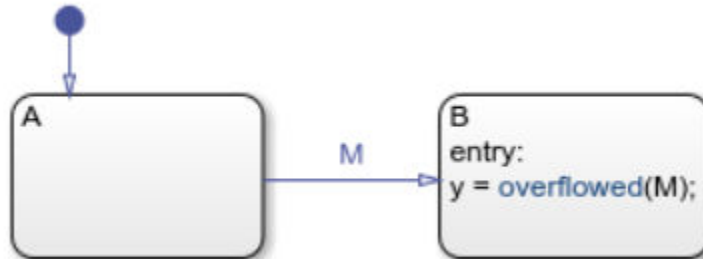
- Guard a state `on` action with the message and call the `overflowed` operator in the action.
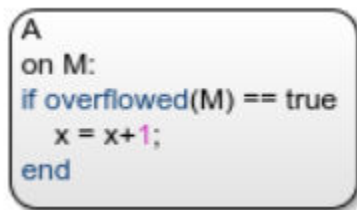
- In a state action, use the `receive` operator followed by the `overflowed` operator.

```
A
during:
if receive(M) == true
    if overflowed(M) == true
        y = y+1;
    end
end
```

Calling the `overflowed` operator before retrieving an input message in the same time step results in a run-time error.

Check for Output Message Overflow

To check the overflow status of an output message queue, first add a message to the queue. You can:

- Use the `send` operator followed by the `overflowed` operator.

```
A
entry:
M.data = 3;
send(M);
if overflowed(M) == true
    x = x+1;
end
```

- Use the `forward` operator followed by the `overflowed` operator.

```
A
on M1:
forward(M1, M);
if overflowed(M) == true
    x = x+1;
end
```

Calling the `overflowed` operator before sending or forwarding an output message in the same time step results in a run-time error.

Check for Local Message Overflow

To check the overflow status of a local message queue, either add a message to the queue or remove a message from the queue before calling the `overflowed` operator. Calling the `overflowed` operator before sending or retrieving a local message in the same time step results in a run-time error.

See Also

discard | forward | isvalid | length | overflowed | receive | send

More About

- "Communicate with Stateflow Charts by Sending Messages" on page 11-10
- "Set Properties for a Message" on page 11-14
- "Rules for Naming Stateflow Objects" on page 2-4
- "Specify Size of Stateflow Data" on page 9-40
- "Specify Type of Stateflow Data" on page 9-33
- "Complex Data in Stateflow Charts" on page 23-2
- "Watch Stateflow Data Values" on page 32-36

Use the Sequence Viewer Block to Visualize Messages, Events, and Entities

To see the interchange of messages and events between Stateflow charts in Simulink models and the movement of entities between SimEvents blocks, add a Sequence Viewer block to your Simulink model.

In the Sequence Viewer block, you can view event data related to Stateflow chart execution and the exchange of messages between Stateflow charts. The Sequence Viewer window shows messages as they are created, sent, forwarded, received, and destroyed at different times during model execution. The Sequence Viewer window also displays state activity, transitions, and function calls to Stateflow graphical functions, Simulink functions, and MATLAB functions.

With the Sequence Viewer block, you can visualize the movement of entities between blocks when simulating SimEvents models. All SimEvents blocks that can store entities appear as lifelines in the Sequence Viewer window. Entities moving between these blocks appear as lines with arrows. You can view calls to Simulink Function blocks and to MATLABFunction blocks.

You can add a Sequence Viewer block to the top level of a model or any subsystem. If you place a Sequence Viewer block in a subsystem that does not have messages, events, or state activity, the Sequence Viewer window informs you that there is nothing to display.

For instance, suppose that you simulate the Stateflow example `sf_msg_traffic_light`.
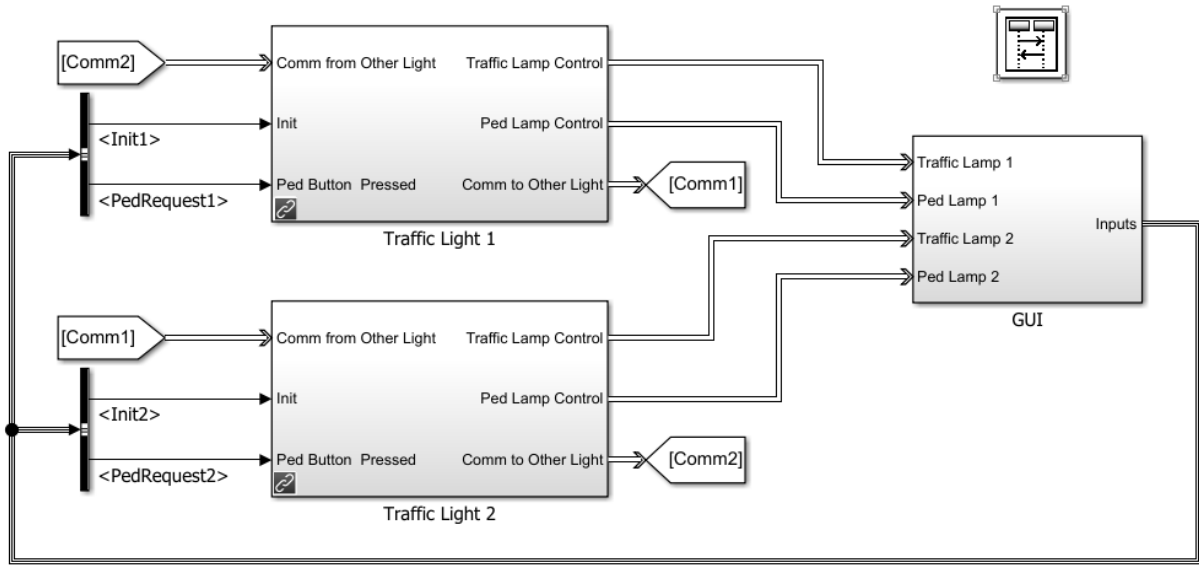
Copyright 2015, The MathWorks, Inc.

This model has three Simulink subsystems: Traffic Light 1, Traffic Light 2, and GUI. The Stateflow charts in these subsystems exchange data by sending messages. As messages pass through the system, you can view them in the Sequence Viewer window. The Sequence Viewer window represents each block in the model as a vertical lifeline with simulation time progressing downward.
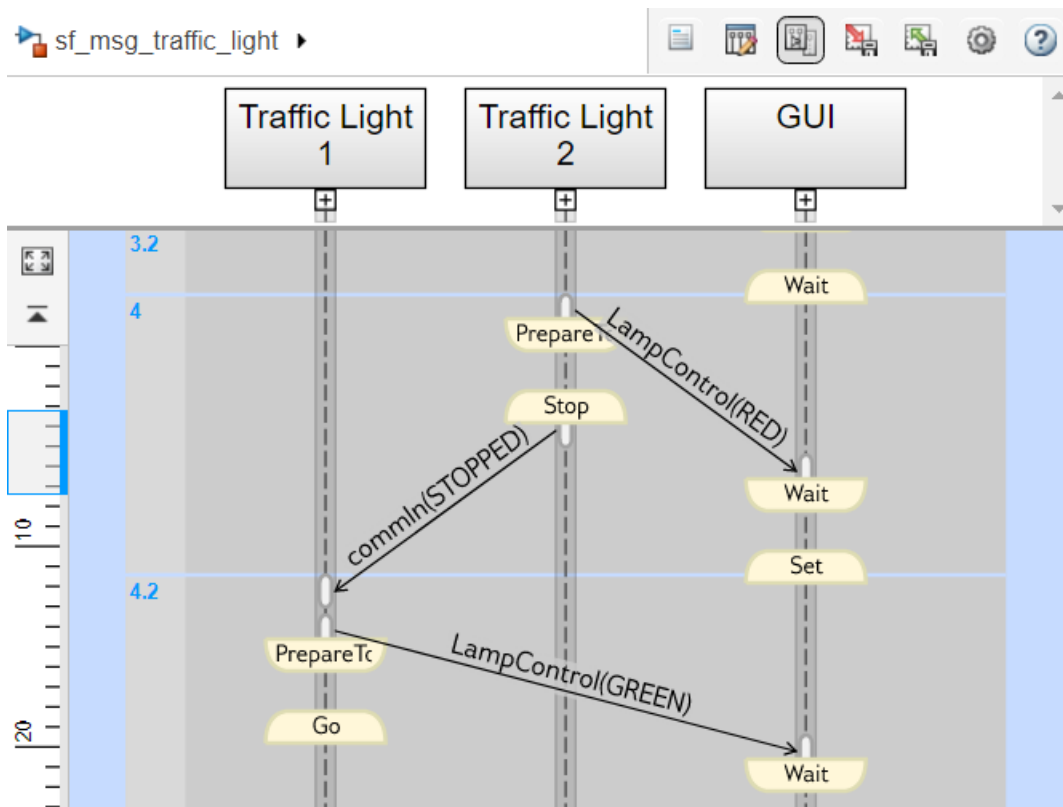
Components of the Sequence Viewer Window

Navigation Toolbar

At the top of the Sequence Viewer window, a navigation toolbar displays the model hierarchy path. Using the toolbar buttons, you can:

- ⊟ Show or hide the Property Inspector.

- ▦ Select an automatic or manual layout.

- ▤ Show or hide inactive lifelines.

- ⬒ Save Sequence Viewer block settings.

- Restore Sequence Viewer block settings.

- Configure Sequence Viewer block parameters.

- Access the Sequence Viewer block documentation.

Property Inspector

In the Property Inspector, you can choose filters to show or hide:

- Events
- Messages
- Function Calls
- State Changes and Transitions

Header Pane

The header pane below the Sequence Viewer toolbar shows lifeline headers containing the names of the corresponding blocks in a model.

- Gray rectangular headers correspond to subsystems.
- White rectangular headers correspond to masked subsystems.
- Yellow headers with rounded corners correspond to Stateflow charts.

To open a block in the model, click the name in the corresponding lifeline header. To show or hide a lifeline, double-click the corresponding header. To resize a lifeline header, click and drag its right-hand side. To fit all lifeline headers in the Sequence Viewer window, press the space bar.
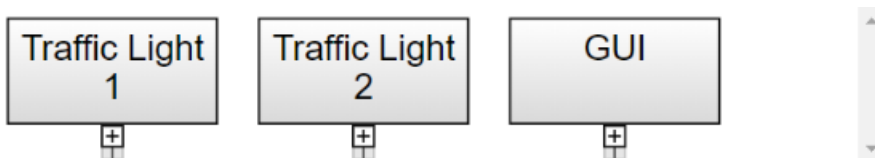
Message Pane

Below the header pane is the message pane. The message pane displays messages, events, and function calls between lifelines as arrows from the sender to the receiver. To display sender, receiver, and payload information in the Property Inspector, click the arrow corresponding to the message, event, or function call.
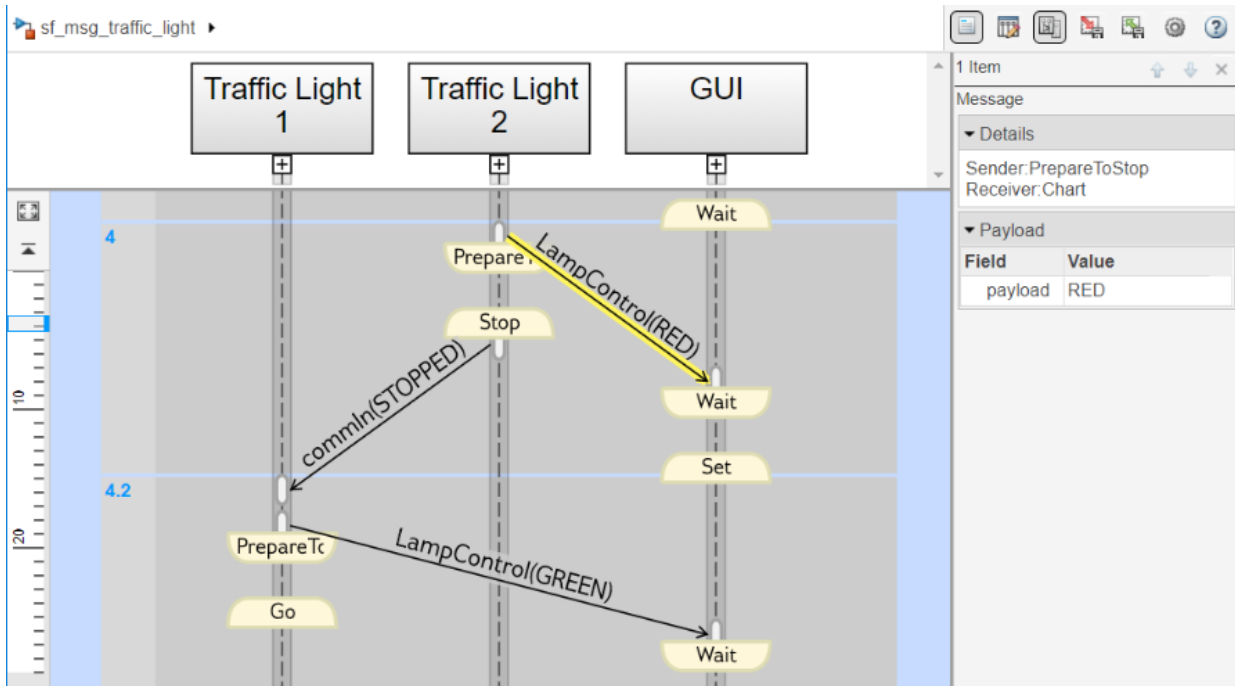
Navigate the Lifeline Hierarchy

In the Sequence Viewer window, the hierarchy of lifelines corresponds to the model hierarchy. When you pause or stop the model, you can expand or contract lifelines and change the root of focus for the viewer.

Expand a Parent Lifeline

In the message pane, a thick, gray lifeline indicates that you can expand the lifeline to see its children. To show the children of a lifeline, click the expander icon ⊞ below the header or double-click the parent lifeline.

For example, expanding the lifeline for the Traffic Light 1 block reveals two new lifelines corresponding to the Stateflow charts Ped Button Sensor and Controller.
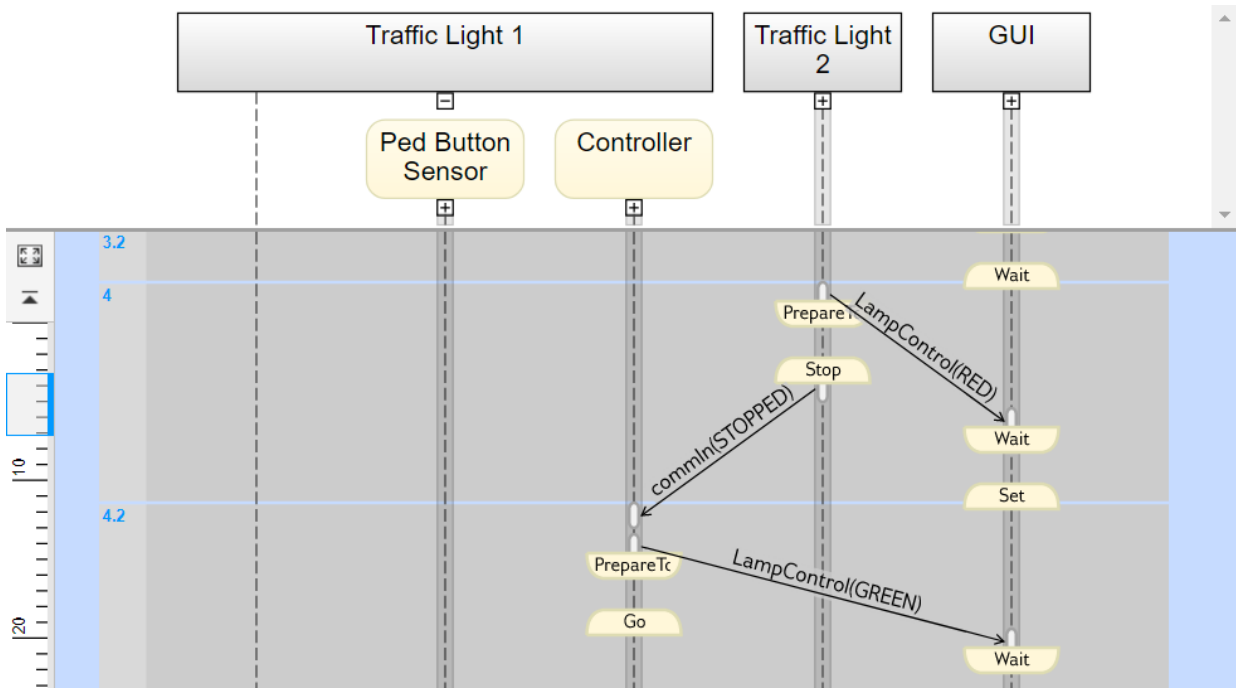
Expand a Masked Subsystem Lifeline

The Sequence Viewer window displays masked subsystems as white blocks. To show the children of a masked subsystem, point over the bottom left corner of the lifeline header and click the arrow.

For example, the GUI subsystem contains four masked subsystems: Traffic Lamp 1, Traffic Lamp 2, Ped Lamp 1, and Ped Lamp 2.
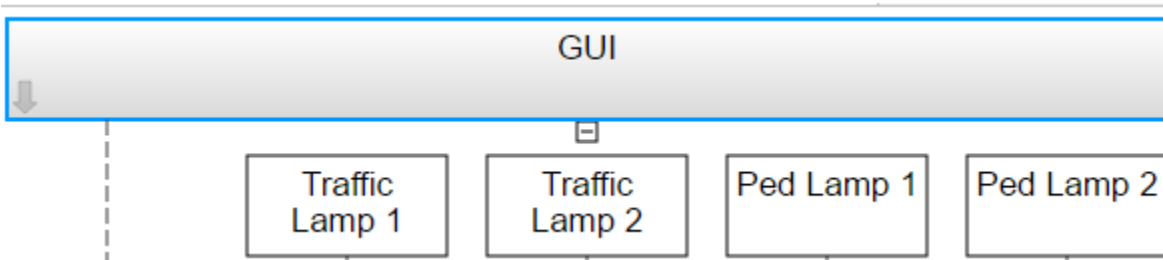
You can display the child lifelines in these masked subsystems by clicking the arrow in the parent lifeline header.
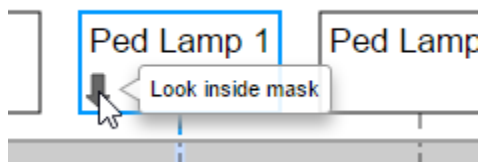
Change Root of Focus

To make a lifeline the root of focus for the viewer, point over the bottom left corner of the lifeline header and click the arrow. Alternatively, you can use the navigation toolbar at the top of the Sequence Viewer window to move the current root up and down the lifeline hierarchy. To move the current root up one level, press the **Esc** key.
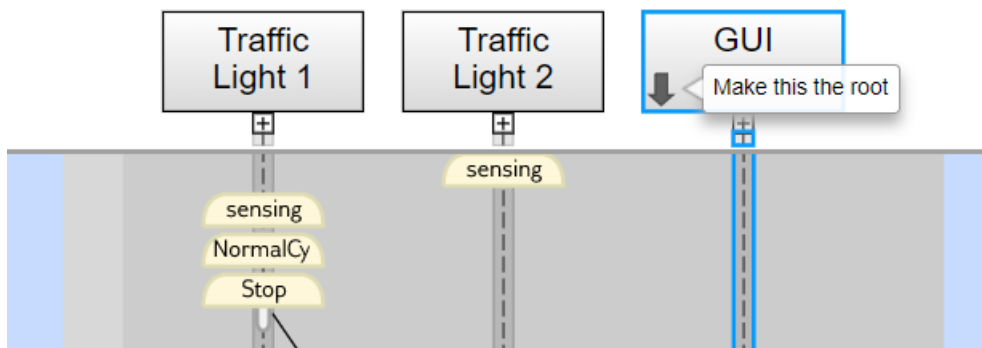
The Sequence Viewer window displays the current root lifeline path and shows its child lifelines. Any external events and messages are displayed as entering or exiting through vertical slots in the diagram gutter. When you point to a slot in the diagram gutter, a tooltip displays the name of the sending or receiving block.
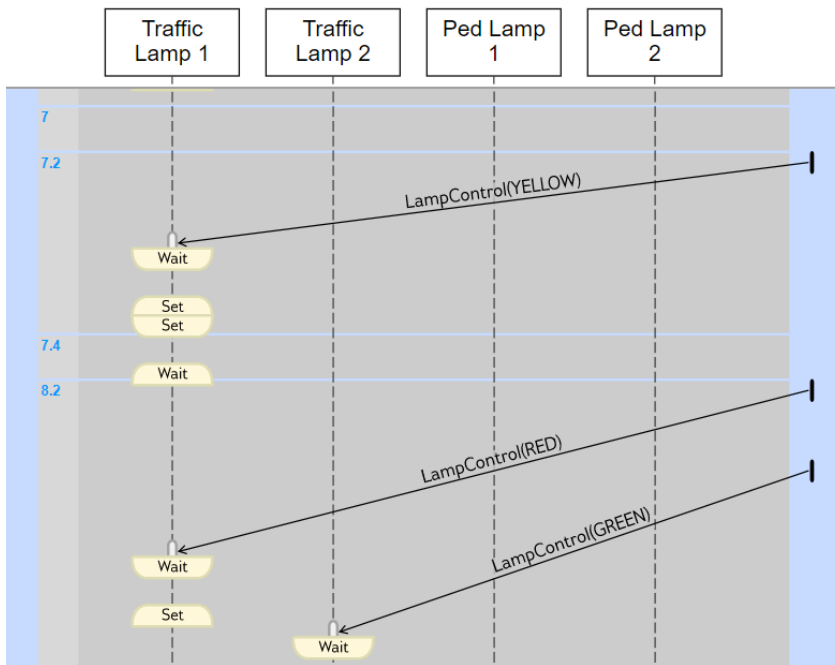
View State Activity and Transitions

To see state activity and transitions in the Sequence Viewer window, expand the state hierarchy until you have reached the lowest child state. Vertical yellow bars show which state is active. Blue horizontal arrows denote the transitions between states.

In this example, you can see a transition from Go to PrepareToStop followed, after 1 second, by a transition to Stop.
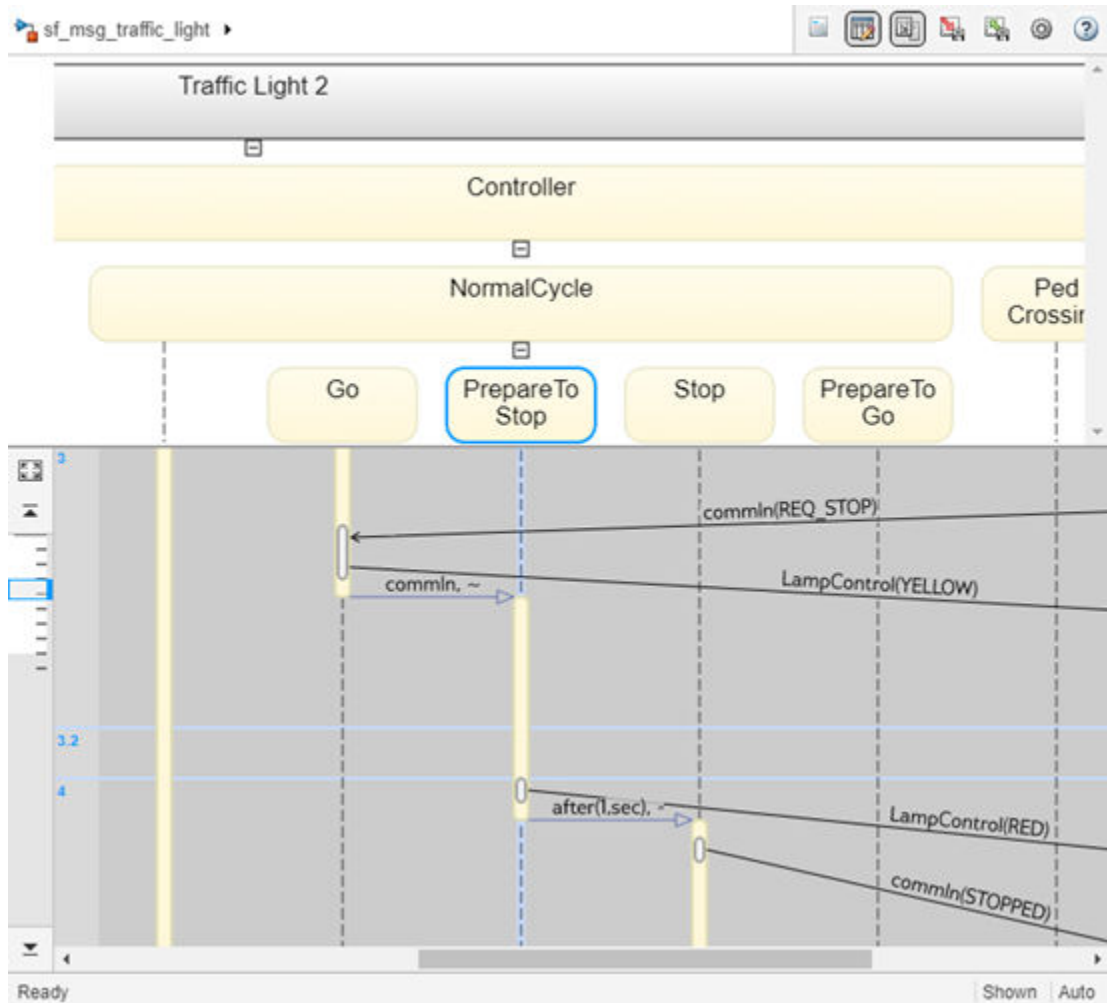
To display the start state, end state, and full transition label in the Property Inspector, click the arrow corresponding to the transition.

To display information about the interactions that occur while a state is active, click the yellow bar corresponding to the state. In the Property Inspector, use the **Search Up** and **Search Down** buttons to move through the transitions, messages, events, and function calls that take place while the state is active.

View Function Calls

The Sequence Viewer block displays function calls and replies. This table lists the type of support for each type of function call.

Function Call Type	Support
Calls to Simulink Function blocks	Fully supported
Calls to Stateflow graphical or Stateflow MATLAB functions	• Scoped — Select the **Export chart level functions** chart option. Use the *chartName.functionName* dot notation. • Global — Select the **Treat exported functions as globally visible** chart option. You do not need the dot notation.
Calls to function-call subsystems	Not displayed in the Sequence Viewer window

The Sequence Viewer window displays function calls as solid arrows labeled with the format *function_name*(*argument_list*). Replies to function calls are displayed as dashed arrows labeled with the format [*argument_list*]=*function_name*.

For example, in the model `slexPrinterExample`, a subsystem calls the Simulink Function block `addPrinterJob`. The function block replies with an output value of `false`.
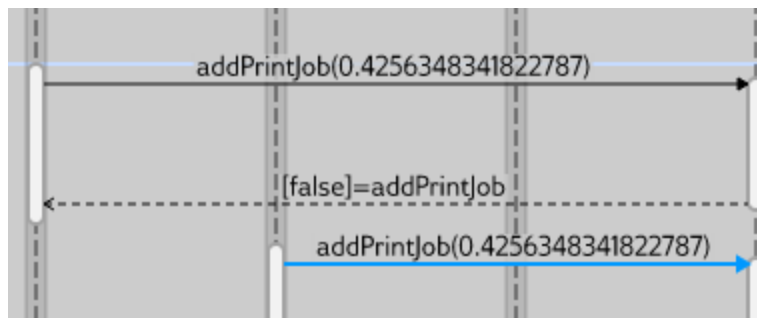
Simulation Time in the Sequence Viewer Window

The Sequence Viewer window shows events vertically, ordered in time. Multiple events in Simulink can happen at the same time. Conversely, there can be long periods of time during simulation with no events. As a consequence, the Simulink Viewer window shows

time by using a combination of linear and nonlinear displays. The time ruler shows linear simulation time. The time grid shows time in a nonlinear fashion. Each time grid row, bordered by two blue lines, contains events that occur at the same simulation time. The time strip provides the times of the events in that grid row.
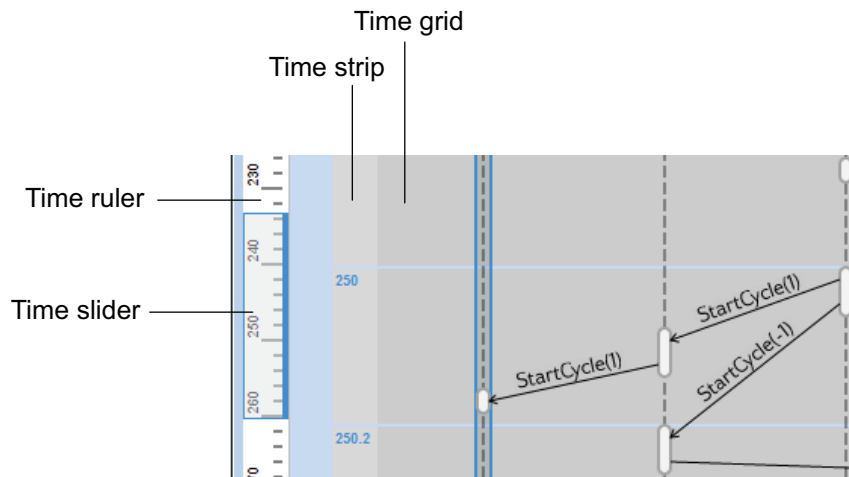
To show events in a specific simulation time range, use the scroll wheel or drag the time slider up and down the time ruler. To navigate to the beginning or end of the simulation, click the **Go to first event** or **Go to last event** buttons. To see the entire simulation duration on the time ruler, click the **Fit to view** button.

When using a variable step solver, you can adjust the precision of the time ruler. In the Model Explorer, on the **Main** tab of the Sequence Viewer Block Parameters pane, adjust the value of the **Time Precision for Variable Step** field.

Redisplay of Information in the Sequence Viewer Window

The Sequence Viewer block saves the order and states of lifelines between simulation runs. When you close and reopen the Sequence Viewer window, it preserves the last open lifeline state. To save a particular viewer state, click the **Save Settings** button in the toolbar. Saving the model then saves that state information across sessions. To load the saved settings, click the **Restore Settings** button.

You can modify the **Time Precision for Variable Step** and **History** parameters only between simulations. You can access the buttons in the toolbar before simulation or when the simulation is paused. During a simulation, the buttons in the toolbar are disabled.

See Also

Sequence Viewer

More About

- "Synchronize Model Components by Broadcasting Events" on page 10-2
- "Communicate with Stateflow Charts by Sending Messages" on page 11-10

Use Actions in Charts

State Action Types

States can have different action types, including `entry`, `during`, `exit`, `bind`, and, `on` actions. You specify the actions for a state by using a state action label with this overall format:

```
name/
entry:entry actions
during:during actions
exit:exit actions
bind:data_name, event_name
on event_name:on event_name actions
on message_name:on message_name actions
```

Enter actions of different types on separate lines after the name of the state. You can enter these actions in any order. If you do *not* specify the action type explicitly for a statement, the chart treats that statement as a combined `entry,during` action.

For example, this chart contains various state action types.
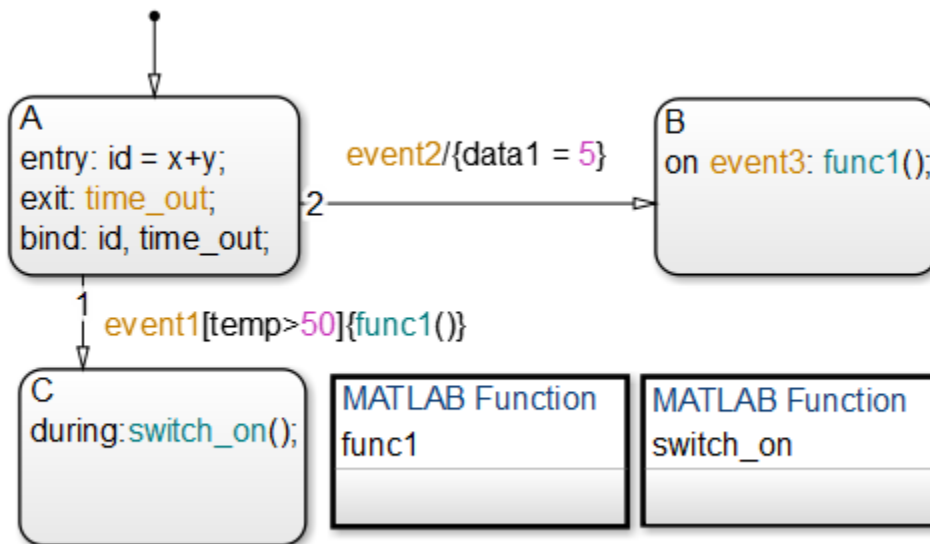
This table summarizes the different state action types.

State Action	Abbreviation	Description
`entry`	`en`	Executes when the state becomes active
`exit`	`ex`	Executes when the state is active and a transition out of the state occurs
`during`	`du`	Executes when the state is active and a specific event occurs
`bind`	none	Binds an event or data object so that only that state and its children can broadcast the event or change the data value
`on` *event_name*	none	Executes when the state is active and it receives a broadcast of *event_name*
`on` *message_name*	none	Executes when a message *message_name* is available
`on after(n, `*event_name*`)`	none	Executes when the state is active and after it receives n broadcasts of *event_name*
`on before(n, `*event_name*`)`	none	Executes when the state is active and before it receives n broadcasts of *event_name*
`on at(n, `*event_name*`)`	none	Executes when the state is active and it receives exactly n broadcasts of *event_name*
`on every(n, `*event_name*`)`	none	Executes when the state is active and upon receipt of every n broadcasts of *event_name*

For more information about the `after`, `before`, `at`, and `every` temporal logic operators, see "Control Chart Execution by Using Temporal Logic" on page 12-49.

Note You can call the temporal logic operators `after` and `before` by using the absolute-time keywords `sec`, `msec`, and `usec`. For details, see "Operators for Absolute-Time Temporal Logic" on page 12-56.

entry Actions

Entry actions are executed when a state becomes active. Entry actions consist of the prefix `entry` (or the abbreviation `en`) followed by a colon (`:`) and one or more actions. To separate multiple entry actions, use semicolons or commas. You can also enter the actions on separate lines.

In the preceding example, the entry action `id = x+y` executes when the chart takes the default transition and state A becomes active. See "Enter a Chart or State" on page 3-50.

exit Actions

Exit actions are executed when a state is active and a transition out of the state occurs. Exit actions consist of the prefix `exit` (or the abbreviation `ex`) followed by a colon (`:`) and one or more actions. To separate multiple exit actions, use semicolons or commas. You can also enter the actions on separate lines.

In the preceding example, the exit action `time_out` executes when the chart takes one of the transitions from state A to state B or C. See "Exit a State" on page 3-58.

during Actions

During actions are executed when a state is active, an event occurs, and no valid transition to another state is available. During actions consist of the prefix `during` (or the abbreviation `du`) followed by a colon (`:`) and one or more actions. To separate multiple during actions, use semicolons or commas. You can also enter the actions on separate lines.

In the preceding example, the during action `switch_on()` executes whenever the state C is active because there are no valid transitions to another state. See "Execution of a Stateflow Chart" on page 3-44.

bind Actions

You can bind the data and events to a state by using a `bind` action. A `bind` action consists of the prefix `bind` followed by a colon (`:`) and one or more events or data. To separate multiple events and data, use semicolons or commas. You can also enter the events and data on separate lines.

Only a state and its children can change data or broadcast events bound to that state. Other states can read the bound data or listen for the bound event, but they cannot change the bound data or send the bound events.

Bind actions apply to a chart whether the binding state is active or not. In the preceding example, the bind action `bind: id, time_out` for state A binds the data `id` and the event `time_out` to state A. This binding prevents any other state (or its children) in the chart from changing `id` or broadcasting event `time_out`.

If another state includes actions that change data or broadcast events that bind to another state, a parsing error occurs. This chart contains two state actions that produce parsing errors.
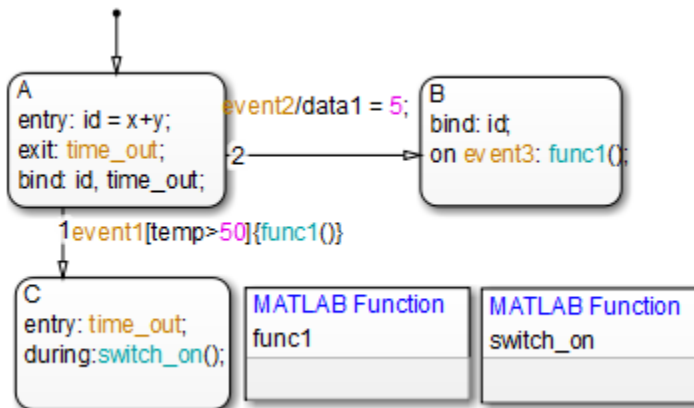
State Action	Reason for Parse Error
`bind: id` in state B	Only one state can change the data `id`, which is bound to state A
`entry: time_out` in state C	Only one state can broadcast the event `time_out`, which is bound to state A

Binding a function-call event to a state also binds the function-call subsystem that it calls. The function-call subsystem is enabled when the binding state is entered and disabled when the binding state is exited. For more information about this behavior, see "Control Function-Call Subsystems by Using bind Actions" on page 12-85.

Bind actions are supported only in Stateflow charts in Simulink models.

on Actions

On actions are executed when the state is active and it receives an event or message. On actions consist of the prefix on followed by a unique event *event_name* or message *message_name*, a colon (:), and one or more actions. To separate multiple on actions, use semicolons or commas. You can also enter the actions on separate lines.

You can specify actions for more than one event or message. For example, if you want different events to trigger different actions, enter multiple on action statements in the state action label:

```
on ev1: action1();
on ev2: action2();
```

If multiple events occur at the same time, the corresponding on actions execute in the order that they appear in the state action label. For instance, in the previous example, if events ev1 and ev2 occur at the same time, then action1() executes first and action2() executes second. See "Execution of a Stateflow Chart" on page 3-44.

See Also

Related Examples

- "Execution of a Stateflow Chart" on page 3-44
- "Enter a Chart or State" on page 3-50
- "Exit a State" on page 3-58
- "Eliminate Redundant Code by Combining State Actions" on page 12-11
- "Control Function-Call Subsystems by Using bind Actions" on page 12-85
- "Operators for Absolute-Time Temporal Logic" on page 12-56

Transition Action Types

Transitions can have different action types, which include event or message triggers, conditions, condition actions, and transition actions. The action types follow the label notation with this general format:

event_or_message trigger[*condition*]{*condition_action*}/{*transition_action*}

The following example shows typical transition label syntax:
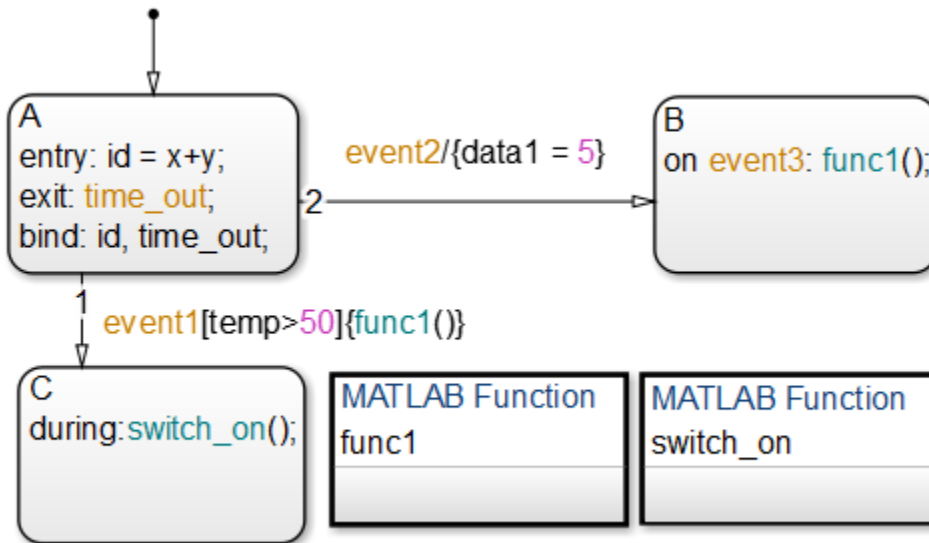
Transition	Event Trigger	Condition	Condition Action	Transition Action
State A to state C	event1	temp > 50	func1()	None
State A to state B	event2	None	None	data1 = 5

Event or Message Triggers

In transition label syntax, event or message triggers appear first as the name of an event or message. They have no distinguishing special character to separate them from other actions in a transition label. In the example in "Transition Action Types" on page 12-7, both transitions from state A have event triggers. The transition from state A to state B has the event trigger `event2` and the transition from state A to state C has the event trigger `event1`.

Event triggers specify an event that causes the transition to be taken, provided the condition, if specified, is true. Specifying an event is optional. Message triggers specify the transition to be taken if the message is present in the message queue. The absence of an event or message indicates that the transition is taken upon the occurrence of any event. Multiple events or messages are specified using the OR logical operator (`|`).

Conditions

In transition label syntax, conditions are Boolean expressions enclosed in square brackets (`[]`). In the example in "Transition Action Types" on page 12-7, the transition from state A to state C has the condition `temp > 50`.

A condition is a Boolean expression to specify that a transition occurs given that the specified expression is true. Follow these guidelines for defining and using conditions:

- The condition expression must be a Boolean expression that evaluates to true (1) or `false` (0).
- The condition expression can consist of any of the following:

  - Boolean operators that make comparisons between data and numeric values
  - A function that returns a Boolean value
  - An `in(state_name)` condition that evaluates to true when the state specified as the argument is active (see "Check State Activity by Using the in Operator" on page 12-77)

    Note A chart cannot use the `in` condition to trigger actions based on the activity of states in other charts.

  - Temporal logic conditions (see "Control Chart Execution by Using Temporal Logic" on page 12-49)

- The condition expression can call a graphical function, truth table function, or MATLAB function that returns a numeric value.

  For example, `[test_function(x, y) < 0]` is a valid condition expression.

  Note If the condition expression calls a function with multiple return values, only the first value applies. The other return values are not used.

- The condition expression should not call a function that causes the chart to change state or modify any variables.
- Boolean expressions can be grouped using & for expressions with AND relationships and | for expressions with OR relationships.
- Assignment statements are not valid condition expressions.
- Unary increment and decrement actions are not valid condition expressions.

Condition Actions

In transition label syntax, condition actions follow the transition condition and are enclosed in curly braces (`{}`). In the example in "Transition Action Types" on page 12-7, the transition from state A to state C has the condition action `func1()`, a function call.

Condition actions are executed as soon as the condition is evaluated as true, but before the transition destination has been determined to be valid. If no condition is specified, an implied condition evaluates to true and the condition action is executed.

Note If a condition is guarded by an event, it is checked only if the event trigger is active. If a condition is guarded by a message, it is checked only if the message is present.

Transition Actions

In transition label syntax, transition actions are preceded with a forward slash (`/`) and are enclosed in curly braces (`{}`). In the example in "Transition Action Types" on page 12-7, the transition from state A to state B has the transition action `data1 = 5`. In C charts, transition actions are not required to be enclosed in curly braces. In charts that use MATLAB as the action language, the syntax is auto corrected if the curly braces are missing from the transition action. See "Auto Correction When Using MATLAB as the Action Language" on page 13-2.

Transition actions execute only after the complete transition path is taken. They execute after the transition destination has been determined to be valid, and the condition, if specified, is true. If the transition consists of multiple segments, the transition action executes only after the entire transition path to the final destination is determined to be valid.

Transition actions are supported only in Stateflow charts in Simulink models.

Eliminate Redundant Code by Combining State Actions

You can combine `entry`, `during`, and `exit` actions that execute the same tasks in a state.

By combining state actions that execute the same tasks, you eliminate redundant code. For example:

Separate Actions	Equivalent Combined Actions

entry:
  y = 0;
  y=y+1;
during: y=y+1;
	entry: y = 0;
	entry, during: y=y+1;

en:
  fcn1();
  fcn2();
du: fcn1();
ex: fcn1();
	en, du, ex: fcn1();
	en: fcn2();

Combining state actions this way produces the same chart execution behavior (semantics) and generates the same code as the equivalent separate actions.

How to Combine State Actions

Combine a set of `entry`, `during`, and/or `exit` actions that perform the same task as a comma-separated list in a state. Here is the syntax:

`entry, during, exit:` *task1*; *task2*;...*taskN*;

You can also use the equivalent abbreviations:

`en, du, ex:` *task1*; *task2*;...*taskN*;

Valid Combinations

You can use any combination of the three actions. For example, the following combinations are valid:

- `en, du:`
- `en, ex:`

- du, ex:

- en, du, ex:

You can combine actions in any order in the comma-separated list. For example, `en, du:` gives the same result as `du, en:`.

Invalid Combinations

You cannot combine two or more actions of the same type. For example, the following combinations are invalid:

- en, en:

- ex, en, ex:

- du, du, ex:
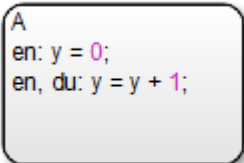
If you combine multiple actions of the same type, you receive a warning that the chart executes the action only once.

Order of Execution of Combined Actions

States execute combined actions in the same order as they execute separate actions:

1 `Entry` actions first, from top to bottom in the order they appear in the state

2 `During` actions second, from top to bottom

3 `Exit` actions last, from top to bottom

The order in which you combine actions does not affect state execution behavior. For example:

This state...	Executes actions in this order...

A
en: y = 0;
en, du: y = y + 1;
 	
	1 en: y = 0;
	2 en: y=y+1;
	3 du: y=y+1;

This state...	Executes actions in this order...

B
 en, du: y = y + 1;
 en: y = 0;
	1 en: y=y+1;
	 2 en: y = 0;
	 3 du: y=y+1;

C
 du, en: y = y + 1;
 en: y = 0;
	1 en: y=y+1;
	 2 en: y = 0;
	 3 du: y=y+1;

D
 du, en: y = y + 1;
 en, ex: y = 10;
	1 en: y=y+1;
	 2 en: y = 10;
	 3 du: y=y+1;
	 4 ex: y = 10;

Rules for Combining State Actions

- Do not combine multiple actions of the same type.
- Do not create data, events, or messages that have the same name as the action keywords: entry, en, during, du, exit, ex.

See Also

More About

- "State Action Types" on page 12-2

Supported Operations for Chart Data

Stateflow charts in Simulink models have an action language property that defines the operations that you can use in state and transition actions. The language properties are:

- MATLAB as the action language.

- C as the action language.

For more information, see "Differences Between MATLAB and C as Action Language Syntax" on page 13-6.

Binary Operations

This table summarizes the interpretation of all binary operations in Stateflow charts according to their order of precedence (0 = highest, 10 = lowest). Binary operations are left associative so that, in any expression, operators with the same precedence are evaluated from left to right. The order of evaluation for other operations is unspecified. For example, in this assignment

```
A = f() > g();
```

the order of evaluation of `f()` and `g()` is unspecified. For more predictable results, it is good coding practice to split expressions that depend on the order of evaluation into multiple statements.

Operation	Precedence	MATLAB as the Action Language	C as the Action Language
a ^ b	0	Power.	Power. Enable this operation by clearing the **Enable C-bit operations** chart property. See "Specify Properties for Stateflow Charts" on page 24-3.
a * b	1	Multiplication.	Multiplication.
a / b	1	Division.	Division.
a %% b	1	Not supported. Use the `rem` or `mod` function.	Remainder. Noninteger operands are first cast to integers.

Operation	Precedence	MATLAB as the Action Language	C as the Action Language
a + b	2	Addition.	Addition.
a - b	2	Subtraction.	Subtraction.
a >> b	3	Not supported. Use the `bitshift` function.	Shift a to the right by b bits. Noninteger operands are first cast to integers.
a << b	3	Not supported. Use the `bitshift` function.	Shift a to the left by b bits. Noninteger operands are first cast to integers.
a > b	4	Comparison, greater than.	Comparison, greater than.
a < b	4	Comparison, less than.	Comparison, less than.
a >= b	4	Comparison, greater than or equal to.	Comparison, greater than or equal to.
a <= b	4	Comparison, less than or equal to.	Comparison, less than or equal to.
a == b	5	Comparison, equal to.	Comparison, equal to.
a ~= b	5	Comparison, not equal to.	Comparison, not equal to.
a != b	5	Not supported. Use the operation a ~= b.	Comparison, not equal to.
a <> b	5	Not supported. Use the operation a ~= b.	Comparison, not equal to.

Operation	Precedence	MATLAB as the Action Language	C as the Action Language
a & b	6	Logical AND. For bitwise AND, use the `bitand` function.	• Bitwise AND (default). Noninteger operands are first cast to integers. Enable this operation by selecting the **Enable C-bit operations** chart property. • Logical AND. Enable this operation by clearing the **Enable C-bit operations** chart property. See "Specify Properties for Stateflow Charts" on page 24-3.
a ^ b	7	Not supported. For bitwise XOR, use the `bitxor` function.	Bitwise XOR (default). Noninteger operands are first cast to integers. Enable this operation by selecting the **Enable C-bit operations** chart property. See "Specify Properties for Stateflow Charts" on page 24-3.
a | b	8	Logical OR. For bitwise OR, use the `bitor` function.	• Bitwise OR (default). Noninteger operands are first cast to integers. Enable this operation by selecting the **Enable C-bit operations** chart property. • Logical OR. Enable this operation by clearing the **Enable C-bit operations** chart property. See "Specify Properties for Stateflow Charts" on page 24-3.

Operation	Precedence	MATLAB as the Action Language	C as the Action Language
a && b	9	Logical AND.	Logical AND.
a || b	10	Logical OR.	Logical OR.

Unary Operations and Actions

This table summarizes the interpretation of all unary operations and actions in Stateflow charts. Unary operations:

- Have higher precedence than the binary operators.

- Are right associative so that, in any expression, they are evaluated from right to left.

Operation	MATLAB as the Action Language	C as the Action Language
~a	Logical NOT. For bitwise NOT, use the bitcmp function.	• Bitwise NOT (default). Noninteger operands are first cast to integers. Enable this operation by selecting the **Enable C-bit operations** chart property. • Logical NOT. Enable this operation by clearing the **Enable C-bit operations** chart property. See "Specify Properties for Stateflow Charts" on page 24-3.
!a	Not supported. Use the operation ~a.	Logical NOT.
-a	Negative.	Negative.
a++	Not supported. Use the expression a = a+1.	Increment. Equivalent to a = a+1.
a--	Not supported. Use the expression a = a-1.	Decrement. Equivalent to a = a-1.

Assignment Operations

This table summarizes the interpretation of assignment operations in Stateflow charts.

Operation	MATLAB as the Action Language	C as the Action Language
`a = b`	Simple assignment.	Simple assignment.
`a := b`	Not supported. Use type cast operations to override fixed-point promotion rules. See "Type Cast Operations" on page 12-19.	Assignment of fixed-point numbers. See "Override Fixed-Point Promotion in C Charts" on page 22-25.
`a += b`	Not supported. Use the expression `a = a+b`.	Equivalent to `a = a+b`.
`a -= b`	Not supported. Use the expression `a = a-b`.	Equivalent to `a = a-b`.
`a *= b`	Not supported. Use the expression `a = a*b`.	Equivalent to `a = a*b`.
`a /= b`	Not supported. Use the expression `a = a/b`.	Equivalent to `a = a/b`.
`a &= b`	Not supported. Use the expression `a = bitand(a,b)`.	Equivalent to `a = a&b` (bitwise AND). Enable this operation by selecting the **Enable C-bit operations** chart property. See "Specify Properties for Stateflow Charts" on page 24-3.
`a ^= b`	Not supported. Use the expression `a = bitxor(a,b)`.	Equivalent to `a = a^b` (bitwise XOR). Enable this operation by selecting the **Enable C-bit operations** chart property. See "Specify Properties for Stateflow Charts" on page 24-3.
`a |= b`	Not supported. Use the expression `a = bitor(a,b)`.	Equivalent to `a = a|b` (bitwise OR). Enable this operation by selecting the **Enable C-bit operations** chart property. See "Specify Properties for Stateflow Charts" on page 24-3.

Pointer and Address Operations

This table summarizes the interpretation of pointer and address operations in Stateflow charts.

Operation	MATLAB as the Action Language	C as the Action Language
&a	Not supported.	Address operation. Use with custom code and Stateflow variables.
*a	Not supported.	Pointer operation. Use only with custom code variables.

For example, the model sf_bus_demo contains a custom C function that takes pointers as arguments. When the chart calls the custom code function, it uses the & operation to pass the Stateflow data by address. For more information, see "Integrate Custom Structures in Stateflow Charts" on page 25-11.

Type Cast Operations

To convert a value of one type to a value of another type, use type cast operations. You can cast data to an explicit type or to the type of another variable.

Cast to Explicit Data Type

To cast a numeric expression to an explicit data type, use a MATLAB type conversion function of the form:

`<type_fun>(expression)`

`<type_fun>` is a type conversion function that can be `double`, `single`, `int32`, `int16`, `int8`, `uint32`, `uint16`, or `uint8`. In charts that use C as the action language, `<type_fun>` can also be `boolean`. For example, this statement casts the expression x+3 to a 16-bit unsigned integer and assigns its value to the data `y`:

`y = uint16(x+3)`

Alternatively, in charts that use MATLAB as the action language, you can use the `cast` function with a type keyword `<type_key>`:

`cast(expression,<type_key>)`

Type keywords include `'double'`, `'single'`, `'int32'`, `'int16'`, `'int8'`, `'uint32'`, `'uint16'`, and `'uint8'`. For instance, as in the preceding example, this statement casts the expression x+3 to a 16-bit unsigned integer and assigns it to `y`:

`y = cast(x+3,'uint16')`

Cast Type Based on Other Data

To make type casting easier, you can convert the type of a numeric expression based on the types of other data.

In charts that use MATLAB as the action language, call the `cast` function with the keyword `'like'`. For example, this statement converts the value of `x+3` to the same type as that of data `z` and assigns it to `y`:

```
y = cast(x+3,'like',z)
```

In this case, the data `z` can have any acceptable Stateflow type.

In charts that use C as the action language, the `type` operator returns the type of an existing Stateflow data. Use this return value in place of an explicit type in a `cast` operation. For example, this statement converts the value of `x+3` to the same type as that of data `z` and assigns it to `y`:

```
cast(x+3,type(z))
```

Replace Operations with Application Implementations

If you have Embedded Coder or Simulink Coder, you can configure the code generator to apply a code replacement library (CRL) during code generation. The code generator changes the code that it generates for operations to meet application requirements. With Embedded Coder, you can develop and apply custom code replacement libraries.

Operation entries of the code replacement library can specify integral or fixed-point operand and result patterns. You can use operation entries for these operations:

- Addition +
- Subtraction -
- Multiplication *
- Division /

For example, in this expression, you can replace the addition operator + with a target-specific implementation if `u1`, `u2`, and `y` have types that permit a match with an addition entry in the code replacement library:

```
y = u1+u2
```

C chart semantics limit operator entry matching because the chart uses the target integer size as its intermediate type in arithmetic expressions. For example, this arithmetic expression computes the intermediate addition into a target integer:

```
y = (u1 + u2) % 3
```

If the target integer size is 32 bits, then you cannot replace this expression with an addition operator from the code replacement library and produce a signed 16-bit result without a loss of precision.

For more information about using code replacement libraries that MathWorks® provides, see "What Is Code Replacement?" (Simulink Coder) and "Code Replacement Libraries" (Simulink Coder). For information about developing custom code replacement libraries, see "What Is Code Replacement Customization?" (Embedded Coder) and "Code You Can Replace From Simulink Models" (Embedded Coder).

See Also

More About

- "Differences Between MATLAB and C as Action Language Syntax" on page 13-6
- "Specify Properties for Stateflow Charts" on page 24-3
- "Supported Operations for Fixed-Point Data" on page 22-21

Supported Symbols in Actions

Boolean Symbols, true and false

Use the symbols `true` and `false` to represent Boolean constants. You can use these symbols as scalars in expressions. Examples include:

```
cooling_fan = true;
heating_fan = false;
```

Tip These symbols are case-sensitive. Therefore, TRUE and FALSE are not Boolean symbols.

Do not use `true` and `false` in the following cases. Otherwise, error messages appear.

- Left-hand side of assignment statements

  - `true++;`
  - `false += 3;`
  - `[true, false] = my_function(x);`
- Argument of the `change` implicit event (see "Define Chart Behavior by Using Implicit Events" on page 10-33)

  - `change(true);`
  - `chg(false);`
- Indexing into a vector or matrix (see "Supported Operations for Vectors and Matrices" on page 17-11)

  - `x = true[1];`
  - `y = false[1][1];`

Note If you define `true` and `false` as Stateflow data objects, your custom definitions of `true` and `false` override the built-in Boolean constants.

Comment Symbols, %, //, /*

Use the symbols %, //, and /* to represent comments as shown in these examples:

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

You can also include comments in generated code for an embedded target (see "Model Configuration Parameters: Code Generation Comments" (Simulink Coder). C chart comments in generated code use multibyte character code. Therefore, you can have code comments with characters for non-English alphabets, such as Japanese Kanji characters.

Hexadecimal Notation Symbols, 0xFF

C charts support C style hexadecimal notation, for example, 0xFF. You can use hexadecimal values wherever you can use decimal values.

Infinity Symbol, inf

Use the MATLAB symbol inf to represent infinity in C charts. Calculations like n/0, where n is any nonzero real value, result in inf.

Note If you define inf as a Stateflow data object, your custom definition of inf overrides the built-in value.

Line Continuation Symbol, ...

Use the characters ... at the end of a line to indicate that the expression continues on the next line. For example, you can use the line continuation symbol in a state action:

```
entry: total1 = 0, total2 = 0, ...
  total3 = 0;
```

Literal Code Symbol, $

Use $ characters to mark actions that you want the parser to ignore but you want to appear in the generated code. For example, the parser does not process any text between the $ characters below.

```
$
ptr -> field = 1.0;
$
```

Note Avoid frequent use of literal symbols.

MATLAB Display Symbol, ;

Omitting the semicolon after an expression displays the results of the expression in the Diagnostic Viewer. If you use a semicolon, the results do not appear.

Single-Precision Floating-Point Number Symbol, F

Use a trailing F to specify single-precision floating-point numbers in C charts. For example, you can use the action statement x = 4.56F; to specify a single-precision constant with the value 4.56. If a trailing F does not appear with a number, double precision applies.

Time Symbol, t

For C charts, use the letter t to represent absolute time that the chart inherits from a Simulink signal in simulation targets. For example, the condition [t - On_time > Duration] specifies that the condition is true if the difference between the simulation time t and On_time is greater than the value of Duration.

The letter t has no meaning for nonsimulation targets, since t depends on the specific application and target hardware.

Note If you define t as a Stateflow data object, your custom definition of t overrides the built-in value.

For charts that use MATLAB as the action language the letter t is not a reserved symbol. To get the simulation time, use the function getSimulationTime().

Call C Library Functions in C Charts

Stateflow charts in Simulink models have an action language property that defines the syntax for state and transition actions. An icon in the lower-left corner of the chart canvas indicates the action language for the chart.

- MATLAB as the action language.

- C as the action language.

Call C Library Functions

You can call this subset of the C Math Library functions:

abs[*] [**]	acos[**]	asin[**]	atan[**]	atan2[**]	ceil[**]
cos[**]	cosh[**]	exp[**]	fabs	floor[**]	fmod[**]
labs	ldexp[**]	log[**]	log10[**]	pow[**]	rand
sin[**]	sinh[**]	sqrt[**]	tan[**]	tanh[**]	

[*] The Stateflow `abs` function goes beyond that of its standard C counterpart with its own built-in functionality. See "Call the abs Function" on page 12-26.

[**] You can also replace calls to the C Math Library with application-specific implementations for this subset of functions. For more information, see "Replacement of Math Library Functions with Application Implementations" on page 12-27.

When you call these math functions, double precision applies unless all the input arguments are explicitly single precision. When a type mismatch occurs, a cast of the input arguments to the expected type replace the original arguments. For example, if you call the `sin` function with an integer argument, a cast of the input argument to a floating-point number of type `double` replaces the original argument.

If you call other C library functions not listed above, include the appropriate `#include...` statement in the **Simulation Target** pane of the Configuration Parameters.

Call the abs Function

Interpretation of the Stateflow `abs` function goes beyond the standard C version to include integer and floating-point arguments of all types as follows:

- If `x` is an integer of type `int32`, the standard C function `abs` applies to `x`, or `abs(x)`.
- If `x` is an integer of type `int16` or `int8`, the standard C `abs` function applies to a cast of `x` as an integer of type `int32`, or `abs((int32)x)`.
- If `x` is a floating-point number of type `double`, the standard C function `fabs` applies to `x`, or `fabs(x)`.
- If `x` is a floating-point number of type `single`, the standard C function `fabs` applies to a cast of `x` as a `double`, or `fabs((double)x)`.
- If `x` is a fixed-point number, the standard C function `fabs` applies to a cast of the fixed-point number as a `double`, or `fabs((double) $V_x$)`, where V_x is the real-world value of `x`.

If you want to use the `abs` function in the strict sense of standard C, cast its argument or return values to integer types. See "Type Cast Operations" on page 12-19.

Note If you declare `x` in custom code, the standard C `abs` function applies in all cases. For instructions on inserting custom code into charts, see "Reuse Custom Code in Stateflow Charts" on page 30-26.

Call min and max Functions

You can call `min` and `max` by emitting the following macros automatically at the top of generated code.

```
#define min(x1,x2) ((x1) > (x2) ? (x2):(x1))
#define max(x1,x2) ((x1) > (x2) ? (x1):(x2))
```

To allow compatibility with user graphical functions named `min()` or `max()`, generated code uses a mangled name of the following form: `<prefix>_min`. However, if you export `min()` or `max()` graphical functions to other charts in your model, the name of these functions can no longer be emitted with mangled names in generated code and conflict occurs. To avoid this conflict, rename the `min()` and `max()` graphical functions.

Replacement of Math Library Functions with Application Implementations

You can configure the code generator to change the code that it generates for math library functions such that the code meets application requirements. To do this you configure the code generator to apply a code replacement library (CRL) during code generation. If you have an Embedded Coder license, you can develop and apply custom code replacement libraries.

For more information about replacing code, using code replacement libraries that MathWorks provides, see "What Is Code Replacement?" (Simulink Coder) and "Code Replacement Libraries" (Simulink Coder). For information about developing custom code replacement libraries, see "What Is Code Replacement Customization?" (Embedded Coder) and "Code You Can Replace From Simulink Models" (Embedded Coder).

Call Custom C Code Functions

You can specify custom code functions for use in C charts for simulation and C code generation.

- "Specify Custom C Functions for Simulation" on page 12-27
- "Specify Custom C Functions for Code Generation" on page 12-28
- "Guidelines for Calling Custom C Functions in Your Chart" on page 12-28
- "Guidelines for Writing Custom C Functions That Access Input Vectors" on page 12-28
- "Function Call in Transition Action" on page 12-29
- "Function Call in State Action" on page 12-29
- "Pass Arguments by Reference" on page 12-30

Specify Custom C Functions for Simulation

To specify custom C functions for simulation:

1   Open the Model Configuration Parameters dialog box.

2   Select **Simulation Target**.

3   Specify your custom C files, as described in "Access Custom C Code in Nonlibrary Charts" on page 30-5.

Specify Custom C Functions for Code Generation

To specify custom C functions for code generation:

1 Open the Model Configuration Parameters dialog box.

2 Select **Code Generation** > **Custom Code**.

3 Specify your custom C files, as described in "Integrate External Code for All Charts" (Simulink Coder).

Guidelines for Calling Custom C Functions in Your Chart

- Define a function by its name, any arguments in parentheses, and an optional semicolon.

- Pass parameters to user-written functions using single quotation marks. For example, `func('string')`.

- An action can nest function calls.

- An action can invoke functions that return a scalar value (of type `double` in the case of MATLAB functions and of any type in the case of C user-written functions).

Guidelines for Writing Custom C Functions That Access Input Vectors

- Use the `sizeof` function to determine the length of an input vector.

  For example, your custom function can include a for-loop that uses `sizeof` as follows:

  ```
  for(i=0; i < sizeof(input); i++) {
  ......
  }
  ```

- If your custom function uses the value of the input vector length multiple times, include an input to your function that specifies the input vector length.

  For example, you can use `input_length` as the second input to a `sum` function as follows:

  ```
  int sum(double *input, double input_length)
  ```

  Your `sum` function can include a for-loop that iterates over all elements of the input vector:

  ```
  for(i=0; i < input_length; i++) {
  ......
  }
  ```

Function Call in Transition Action

Example formats of function calls using transition action notation appear in the following chart.

A function call to `fcn1` occurs with `arg1`, `arg2`, and `arg3` if the following are true:

- `S1` is active.
- Event `e` occurs.
- Condition `c` is true.
- The transition destination `S2` is valid.

The transition action in the transition from `S2` to `S3` shows a function call nested within another function call.

Function Call in State Action

Example formats of function calls using state action notation appear in the following chart.
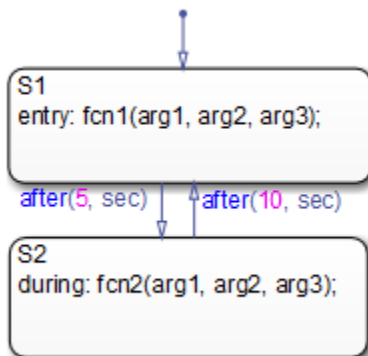
Chart execution occurs as follows:

1   When the default transition into S1 occurs, S1 becomes active.

2   The `entry` action, a function call to `fcn1` with the specified arguments, executes.

3   After 5 seconds of simulation time, S1 becomes inactive and S2 becomes active.

4   The `during` action, a function call to `fcn2` with the specified arguments, executes.

5   After 10 seconds of simulation time, S2 becomes inactive and S1 becomes active again.

6   Steps 2 through 5 repeat until the simulation ends.

Pass Arguments by Reference

A Stateflow action can pass arguments to a user-written function by reference rather than by value. In particular, an action can pass a pointer to a value rather than the value itself. For example, an action could contain the following call:

`f(&x);`

where `f` is a custom-code C function that expects a pointer to `x` as an argument.

If `x` is the name of a data item defined in the Stateflow hierarchy, the following rules apply:

- Do not use pointers to pass data items input from a Simulink model.

  If you need to pass an input item by reference, for example, an array, assign the item to a local data item and pass the local item by reference.

- If x is a Simulink output data item having a data type other than `double`, the chart property **Use Strong Data Typing with Simulink I/O** must be on (see "Specify Properties for Stateflow Charts" on page 24-3).

- If the data type of x is `boolean`, you must turn off the coder option **Use bitsets for storing state configuration**.

- If x is an array with its first index property set to 0 (see "Set Data Properties" on page 9-5), then you must call the function as follows.

  ```
  f(&(x[0]));
  ```

  This passes a pointer to the first element of x to the function.

- If x is an array with its first index property set to a nonzero number (for example, 1), the function must be called in the following way:

  ```
  f(&(x[1]));
  ```

  This passes a pointer to the first element of x to the function.

Access MATLAB Functions and Workspace Data in C Charts

Stateflow charts in Simulink models have an action language property that defines the syntax for state and transition actions. An icon in the lower-left corner of the chart canvas indicates the action language for the chart.

-

  MATLAB as the action language.

-

  C as the action language.

In charts that use C as the action language, you can call built-in MATLAB functions and access MATLAB workspace variables by using the `ml` namespace operator or the `ml` function.

Caution Because MATLAB functions are not available in a target environment, do not use the `ml` namespace operator and the `ml` function if you plan to build a code generation target.

---

ml Namespace Operator

For C charts, the `ml` namespace operator uses standard dot (`.`) notation to reference MATLAB variables and functions. For example, the statement `a = ml.x` returns the value of the MATLAB workspace variable `x` to the Stateflow data `a`.

For functions, the syntax is as follows:

`[`*return_val1, return_val2*`,...] = ml.`*matfunc*`(`*arg1, arg2*`,...)`

For example, the statement `[a, b, c] = ml.`*matfunc*`(x, y)` passes the return values from the MATLAB function *matfunc* to the Stateflow data `a`, `b`, and `c`.

If the MATLAB function you call does not require arguments, you must still include the parentheses. If you omit the parentheses, Stateflow software interprets the function name as a workspace variable, which, when not found, generates a run-time error during simulation.

Examples

In these examples, x, y, and z are workspace variables and d1 and d2 are Stateflow data:

- a = ml.sin(ml.x)

  In this example, the MATLAB function sin evaluates the sine of x, which is then assigned to Stateflow data variable a. However, because x is a workspace variable, you must use the namespace operator to access it. Hence, ml.x is used instead of just x.

- a = ml.sin(d1)

  In this example, the MATLAB function sin evaluates the sine of d1, which is assigned to Stateflow data variable a. Because d1 is Stateflow data, you can access it directly.

- ml.x = d1*d2/ml.y

  The result of the expression is assigned to x. If x does not exist prior to simulation, it is automatically created in the MATLAB workspace.

- ml.v[5][6][7] = ml.*matfunc*(ml.x[1][3], ml.y[3], d1, d2, '*string*')

  The workspace variables x and y are arrays. x[1][3] is the (1,3) element of the two-dimensional array variable x. y[3] is the third element of the one-dimensional array variable y. The last argument, '*string*', is a character vector.

  The return from the call to *matfunc* is assigned to element (5,6,7) of the workspace array, v. If v does not exist prior to simulation, it is automatically created in the MATLAB workspace.

ml Function

For C charts, you can use the ml function to specify calls to MATLAB functions. The format for the ml function call uses this notation:

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

evalString is an expression that is evaluated in the MATLAB workspace. It contains a MATLAB command (or a set of commands, each separated by a semicolon) to execute along with format specifiers (%g, %f, %d, etc.) that provide formatted substitution of the other arguments (*arg1*, *arg2*, etc.) into *evalString*.

The format specifiers used in ml functions are the same as those used in the C functions printf and sprintf. The ml function call is equivalent to calling the MATLAB eval

function with the `ml` namespace operator if the arguments *arg1*, *arg2*,... are restricted to scalars or literals in the following command:

`ml.eval(ml.sprintf(`*evalString*`, `*arg1*`, `*arg2*`,...))`

Stateflow software assumes scalar return values from `ml` namespace operator and `ml` function calls when they are used as arguments in this context. See "How Charts Infer the Return Size for ml Expressions" on page 12-39.

Examples

In these examples, `x` is a MATLAB workspace variable, and `d1` and `d2` are Stateflow data:

- `a = ml('sin(x)')`

  In this example, the `ml` function calls the MATLAB function `sin` to evaluate the sine of `x` in the MATLAB workspace. The result is then assigned to Stateflow data variable `a`. Because `x` is a workspace variable, and `sin(x)` is evaluated in the MATLAB workspace, you enter it directly in the *evalString* argument (`'sin(x)'`).

- `a = ml('sin(%f)', d1)`

  In this example, the MATLAB function `sin` evaluates the sine of `d1` in the MATLAB workspace and assigns the result to Stateflow data variable `a`. Because `d1` is Stateflow data, its value is inserted in the *evalString* argument (`'sin(%f)'`) using the format expression `%f`. This means that if `d1 = 1.5`, the expression evaluated in the MATLAB workspace is `sin(1.5)`.

- `a = ml('`*matfunc*`(%g, ''abcdefg'', x, %f)', d1, d2)`

  In this example, the expression `'`*matfunc*`(%g, ''abcdefg'', x, %f)'` is the *evalString* shown in the preceding format statement. Stateflow data `d1` and `d2` are inserted into that expression with the format specifiers `%g` and `%f`, respectively. `''abcdefg''` is a literal enclosed with two single pairs of quotation marks because it is part of the evaluation expression, which is already enclosed in single quotation marks.

- `sfmat_44 = ml('rand(4)')`

  In this example, a square 4-by-4 matrix of random numbers between 0 and 1 is returned and assigned to the Stateflow data `sf_mat44`. Stateflow data `sf_mat44` must be defined as a 4-by-4 array before simulation. If its size is different, a size mismatch error is generated during run-time.

ml Expressions

For C charts, you can mix `ml` namespace operator and `ml` function expressions along with Stateflow data in larger expressions. The following example squares the `sine` and `cosine` of an angle in workspace variable X and adds them:

```
ml.power(ml.sin(ml.X),2) + ml('power(cos(X),2)')
```

The first operand uses the `ml` namespace operator to call the `sin` function. Its argument is `ml.X`, since X is in the MATLAB workspace. The second operand uses the `ml` function. Because X is in the workspace, it appears in the *evalString* expression as X. The squaring of each operand is performed with the MATLAB `power` function, which takes two arguments: the value to square, and the power value, 2.

Expressions using the `ml` namespace operator and the `ml` function can be used as arguments for `ml` namespace operator and `ml` function expressions. The following example nests `ml` expressions at three different levels:

```
a = ml.power(ml.sin(ml.X + ml('cos(Y)')),2)
```

In composing your `ml` expressions, follow the levels of precedence set out in "Binary Operations" on page 12-14. Use parentheses around power expressions with the ^ operator when you use them in conjunction with other arithmetic operators.

Stateflow software checks expressions for data size mismatches in your actions during parsing of charts and during run time. Because the return values for `ml` expressions are not known until run time, Stateflow software must infer the size of their return values. See "How Charts Infer the Return Size for ml Expressions" on page 12-39.

Which ml Should I Use?

In most cases, the notation of the `ml` namespace operator is more straightforward. However, using the `ml` function call does offer a few advantages:

- Use the `ml` function to dynamically construct workspace variables.

  The following flow chart creates four new MATLAB matrices:
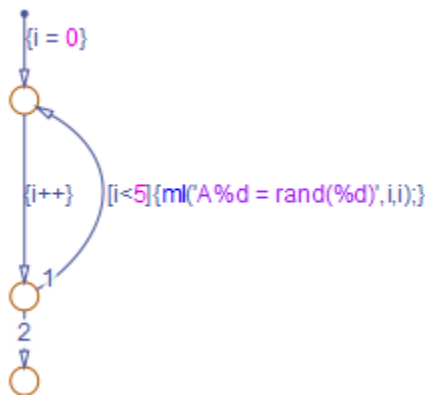
The `for` loop creates four new matrix variables in the MATLAB workspace. The default transition initializes the Stateflow counter `i` to 0, while the transition segment between the top two junctions increments it by 1. If `i` is less than 5, the transition segment back to the top junction evaluates the `ml` function call `ml('A%d = rand(%d)',i,i)` for the current value of `i`. When `i` is greater than or equal to 5, the transition segment between the bottom two junctions occurs and execution stops.

The transition executes the following MATLAB commands, which create a workspace scalar (`A1`) and three matrices (`A2`, `A3`, `A4`):

```
A1 = rand(1)
A2 = rand(2)
A3 = rand(3)
A4 = rand(4)
```

*   Use the `ml` function with full MATLAB notation.

    You cannot use full MATLAB notation with the `ml` namespace operator, as the following example shows:

    ```
    ml.A = ml.magic(4)
    B = ml('A + A''')
    ```

    This example sets the workspace variable `A` to a magic 4-by-4 matrix using the `ml` namespace operator. Stateflow data `B` is then set to the addition of `A` and its transpose matrix, `A'`, which produces a symmetric matrix. Because the `ml` namespace operator cannot evaluate the expression `A'`, the `ml` function is used instead. However, you can call the MATLAB function `transpose` with the `ml` namespace operator in the following equivalent expression:

```
B = ml.A + ml.transpose(ml.A)
```

As another example, you cannot use arguments with cell arrays or subscript expressions involving colons with the ml namespace operator. However, these can be included in an ml function call.

ml Data Type

Stateflow data of type ml is typed internally with the MATLAB type mxArray for C charts. You can assign (store) any type of data available in the Stateflow hierarchy to a data of type ml. These types include any data type defined in the Stateflow hierarchy or returned from the MATLAB workspace with the ml namespace operator or ml function.

Rules for Using ml Data Type

These rules apply to Stateflow data of type ml:

- You can initialize ml data from the MATLAB workspace just like other data in the Stateflow hierarchy (see "Initialize Data from the MATLAB Base Workspace" on page 9-24).

- Any numerical scalar or array of ml data in the Stateflow hierarchy can participate in any kind of unary operation and any kind of binary operation with any other data in the hierarchy.

  If ml data participates in any numerical operation with other data, the size of the ml data must be inferred from the context in which it is used, just as return data from the ml namespace operator and ml function are. See "How Charts Infer the Return Size for ml Expressions" on page 12-39.

  Note The preceding rule does not apply to ml data storing MATLAB 64-bit integers. You can use ml data to store 64-bit MATLAB integers but you cannot use 64-bit integers in C charts.

- You cannot define ml data with the scope **Constant**.

  This option is disabled in the Data properties dialog box and in the Model Explorer for Stateflow data of type ml.

- You can use ml data to build a simulation target but not to build an embeddable code generation target.

- If data of type `ml` contains an array, you can access the elements of the array via indexing with these rules:

  a You can index only arrays with numerical elements.

  b You can index numerical arrays only by their dimension.

  In other words, you can access only one-dimensional arrays by a single index value. You cannot access a multidimensional array with a single index value.

  c The first index value for each dimension of an array is 1, and not 0, as in C language arrays.

  In the examples that follow, `mldata` is a Stateflow data of type `ml`, `ws_num_array` is a 2-by-2 MATLAB workspace array with numerical values, and `ws_str_array` is a 2-by-2 MATLAB workspace array with character vector values.

  ```
  mldata = ml.ws_num_array; /* OK */
  n21 = mldata[2][1]; /* OK for numerical data of type ml */
  n21 = mldata[3]; /* NOT OK for 2-by-2 array data */
  mldata = ml.ws_str_array; /* OK */
  s21 = mldata[2][1]; /* NOT OK for character vector data of type ml*/
  ```

- `ml` data cannot have a scope outside a C chart; that is, you cannot define the scope of `ml` data as **Input from Simulink** or **Output to Simulink**.

Place Holder for Workspace Data

Both the `ml` namespace operator and the `ml` function can access data directly in the MATLAB workspace and return it to a C chart. However, maintaining data in the MATLAB workspace can present Stateflow users with conflicts with other data already resident in the workspace. Consequently, with the `ml` data type, you can maintain `ml` data in a chart and use it for MATLAB computations in C charts.

As an example, in the following statements, `mldata1` and `mldata2` are Stateflow data of type `ml`:

```
mldata1 = ml.rand(3);
mldata2 = ml.transpose(mldata1);
```

In the first line of this example, `mldata1` receives the return value of the MATLAB function `rand`, which, in this case, returns a 3-by-3 array of random numbers. Note that `mldata1` is not specified as an array or sized in any way. It can receive any MATLAB workspace data or the return of any MATLAB function because it is defined as a Stateflow data of type `ml`.

In the second line of the example, `mldata2`, also of Stateflow data type `ml`, receives the transpose matrix of the matrix in `mldata1`. It is assigned the return value of the MATLAB function `transpose` in which `mldata1` is the argument.

Note the differences in notation if the preceding example were to use MATLAB workspace data (`wsdata1` and `wsdata2`) instead of Stateflow `ml` data to hold the generated matrices:

```
ml.wsdata1 = ml.rand(3);
ml.wsdata2 = ml.transpose(ml.wsdata1);
```

In this case, each workspace data must be accessed through the `ml` namespace operator.

How Charts Infer the Return Size for ml Expressions

In C charts, Stateflow expressions using the `ml` namespace operator and the `ml` function evaluate in the MATLAB workspace at run time. The actual size of the data returned from the following expression types is known only at run time:

- MATLAB workspace data or functions using the `ml` namespace operator or the `ml` function call

  For example, the size of the return values from the expressions `ml.var`, `ml.func()`, or `ml(evalString, arg1, arg2,...)`, where *var* is a MATLAB workspace variable and *func* is a MATLAB function, cannot be known until run-time.

- Stateflow data of type `ml`
- Graphical functions that return Stateflow data of type `ml`

When these expressions appear in actions, Stateflow code generation creates temporary data to hold intermediate returns for evaluation of the full expression of which they are a part. Because the size of these return values is unknown until run time, Stateflow software must employ context rules to infer the sizes for creation of the temporary data.

During run time, if the actual returned value from one of these commands differs from the inferred size of the temporary variable that stores it, a size mismatch error appears. To prevent run-time errors, use the following guidelines to write actions with MATLAB commands or `ml` data:

Guideline		Example
Return sizes of MATLAB commands or data in an expression must match return sizes of peer expressions.		In the expression `ml.`*`func()`*` * (x + ml.y)`, if `x` is a 3-by-2 matrix, then `ml.`*`func()`* and `ml.y` are also assumed to evaluate to 3-by-2 matrices. If either returns a value of different size (other than a scalar), an error results during run-time.
Expressions that return a scalar never produce an error. You can combine matrices and scalars in larger expressions because MATLAB commands use scalar expansion.		In the expression `ml.x + y`, if `y` is a 3-by-2 matrix and `ml.x` returns a scalar, the resulting value is the result of adding the scalar value of `ml.x` to every member of `y` to produce a matrix with the size of `y`, that is, a 3-by-2 matrix. The same rule applies to subtraction (`-`), multiplication (`*`), division (`/`), and any other binary operations.
MATLAB commands or Stateflow data of type `ml` can be members of these independent levels of expression, for which resolution of return size is necessary:	Arguments The expression for each function argument is a larger expression for which the return size of MATLAB commands or Stateflow data of type `ml` must be determined.	In the expression `z + `*`func`*`(x + ml.y)`, the size of `ml.y` is independent of the size of `z`, because `ml.y` is used at the function argument level. However, the return size for *`func`*`(x + ml.y)` must match the size of `z`, because they are both at the same expression level.
	Array indices The expression for an array index is an independent level of expression that must be scalar in size.	In the expression `x + `*`array`*`[y]`, the size of `y` is independent of the size of `x` because `y` and `x` are at different levels of expression. Also, `y` must be a scalar.
The return size for an indexed array element access must be a scalar.		The expression `x[1][1]`, where `x` is a 3-by-2 array, must evaluate to a scalar.

Guideline	Example
MATLAB command or data elements used in an expression for the input argument of a MATLAB function called through the `ml` namespace operator are resolved for size. This resolution uses the rule for peer expressions (preceding rule 1) for the expression itself, because no size definition prototype is available.	In the function call `ml.func(x + ml.y)`, if `x` is a 3-by-2 array, `ml.y` must return a 3-by-2 array or a scalar.
MATLAB command or data elements used for the input argument for a graphical function in an expression are resolved for size by the function prototype.	If the graphical function `gfunc` has the prototype `gfunc(arg1)`, where `arg1` is a 2-by-3 Stateflow data array, the calling expression, `gfunc(ml.y + x)`, requires that both `ml.y` and `x` evaluate to 2-by-3 arrays (or scalars) during run-time.
`ml` function calls can take only scalar or character vector literal arguments. Any MATLAB command or data that specifies an argument for the `ml` function must return a scalar value.	In the expression `a = ml('sin(x)')`, the `ml` function calls the MATLAB function `sin` to evaluate the sine of `x` in the MATLAB workspace. Stateflow data variable `a` stores that result.
In an assignment, the size of the right-hand expression must match the size of the left-hand expression, with one exception. If the left-hand expression is a single MATLAB variable, such as `ml.x`, or Stateflow data of type `ml`, the right-hand expression determines the sizes of both expressions.	In the expression `s = ml.func(x)`, where `x` is a 3-by-2 matrix and `s` is scalar Stateflow data, `ml.func(x)` must return a scalar to match the left-hand expression, `s`. However, in the expression `ml.y = x + s`, where `x` is a 3-by-2 data array and `s` is scalar, the left-hand expression, workspace variable `y`, is assigned the size of a 3-by-2 array to match the size of the right-hand expression, `x+s`, a 3-by-2 array.
In an assignment, Stateflow column vectors on the left-hand side are compatible with MATLAB row or column vectors of the same size on the right-hand side. A matrix you define with a row dimension of 1 is considered a row vector. A matrix you define with one dimension or with a column dimension of 1 is considered a column vector.	In the expression `s = ml.func()`, where `ml.func()` returns a 1-by-3 matrix, if `s` is a vector of size 3, the assignment is valid.

Guideline	Example
If you cannot resolve the return size of MATLAB command or data elements in a larger expression by any of the preceding rules, they are assumed to return scalar values.	In the expression `ml.x = ml.y + ml.z`, none of the preceding rules can be used to infer a common size among `ml.x`, `ml.y`, and `ml.z`. In this case, both `ml.y` and `ml.z` are assumed to return scalar values. Even if `ml.y` and `ml.z` return matching sizes at run-time, if they return nonscalar values, a size mismatch error results.
The preceding rules for resolving the size of member MATLAB commands or Stateflow data of type `ml` in a larger expression apply only to cases in which numeric values are expected for that member. For nonnumeric returns, a run-time error results. **Note** Member MATLAB commands or data of type `ml` in a larger expression are limited to numeric values (scalar or array) only if they participate in numeric expressions.	The expression `x + ml.str`, where `ml.str` is a character vector workspace variable, produces a run-time error stating that `ml.str` is not a numeric type.

Special cases exist, in which no size checking occurs to resolve the size of MATLAB command or data expressions that are part of larger expressions. Use of the following expressions does not require enforcement of size checking at run-time:

- `ml.var`
- `ml.func()`
- `ml(evalString, arg1, arg2,...)`
- Stateflow data of type `ml`
- Graphical function returning a Stateflow data of type `ml`

In these cases, assignment of a return to the left-hand side of an assignment statement or a function argument occurs without checking for a size mismatch between the two:

- An assignment in which the left-hand side is a MATLAB workspace variable

  For example, in the expression `ml.x = ml.y`, `ml.y` is a MATLAB workspace variable of any size and type (structure, cell array, character vector, and so on).

- An assignment in which the left-hand side is a data of type `ml`

  For example, in the expression `m_x = ml.func()`, `m_x` is a Stateflow data of type `ml`.
- Input arguments of a MATLAB function

  For example, in the expression `ml.func(m_x, ml.x, gfunc())`, `m_x` is a Stateflow data of type `ml`, `ml.x` is a MATLAB workspace variable of any size and type, and `gfunc()` is a Stateflow graphical function that returns a Stateflow data of type `ml`. Although size checking does not occur for the input type, if the passed-in data is not of the expected type, an error results from the function call `ml.func()`.
- Arguments for a graphical function that are specified as Stateflow data of type `ml` in its prototype statement

---

Note If you replace the inputs in the preceding cases with non-MATLAB numeric Stateflow data, conversion to an `ml` type occurs.

Use Arrays in Actions

Array Notation

A Stateflow action in a C chart uses C style syntax and zero-based indexing by default to access array elements. This syntax differs from MATLAB notation, which uses one-based indexing. For example, suppose that you define a Stateflow input A of size [3 4]. To access the element in the first row, second column, use the expression A[0][1]. Other examples of zero-based indexing in C charts include:

```
local_array[1][8][0] = 10;

local_array[i][j][k] = 77;

var = local_array[i][j][k];
```

Note Use the same notation for accessing arrays in C charts, from Simulink models, and from custom code.

As an exception to zero-based indexing, **scalar expansion** is available. This statement assigns a value of 10 to all the elements of the array local_array.

```
local_array = 10;
```

Scalar expansion is available for performing general operations. This statement is valid if the arrays array_1, array_2, and array_3 have the same value for the **Sizes** property.

```
array_1 = (3*array_2) + array_3;
```

Arrays and Custom Code

C charts provide the same syntax for Stateflow arrays and custom code arrays.

Note Any array variable that is referred to in a C chart but is not defined in the Stateflow hierarchy is identified as a custom code variable.

Broadcast Local Events to Synchronize Parallel States

Directed Event Broadcasting

You can broadcast events directly from one state to another to synchronize parallel (AND) states in the same chart. Local events are supported only in Stateflow charts in Simulink models. The following rules apply:

- The receiving state must be active during the event broadcast.
- An action in one chart cannot broadcast events to states in another chart.

Using a directed local event broadcast provides the following benefits over an undirected broadcast:

- Prevents unwanted recursion during simulation.
- Improves the efficiency of generated code.

For information about avoiding unwanted recursion, see "Guidelines for Avoiding Unwanted Recursion in a Chart" on page 32-34.

Directed Local Event Broadcast Using send

The format of a directed local event broadcast with `send` is:

`send(event_name,state_name)`

where *event_name* is broadcast to *state_name* and any offspring of that state in the hierarchy. The event you send must be visible to both the sending state and the receiving state (*state_name*).

The *state_name* argument can include a full hierarchy path to the state. For example, if the state `A` contains the state `A1`, send an event `e` to state `A1` with the following broadcast:

`send(e, A.A1)`

Tip Do not include the chart name in the full hierarchy path to a state.

The following example of a directed local event broadcast uses the
send(*event_name*,*state_name*) syntax.
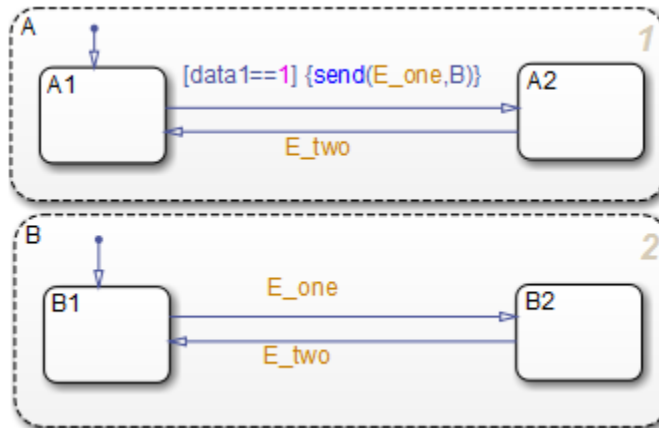
In this example, event E_one belongs to the chart and is visible to both A and B. See
"Directed Event Broadcast Using Send" on page B-50 for more information on the
semantics of this notation.

Directed Local Event Broadcast Using Qualified Event Names

The format of a directed local event broadcast using qualified event names is:

send(*state_name*.*event_name*)

where *event_name* is broadcast to its owning state (*state_name*) and any offspring of
that state in the hierarchy. The event you send is visible only to the receiving state
(*state_name*).

The *state_name* argument can also include a full hierarchy path to the receiving state.
Do not use the chart name in the full path name of the state.

The following example shows the use of a qualified event name in a directed local event
broadcast.
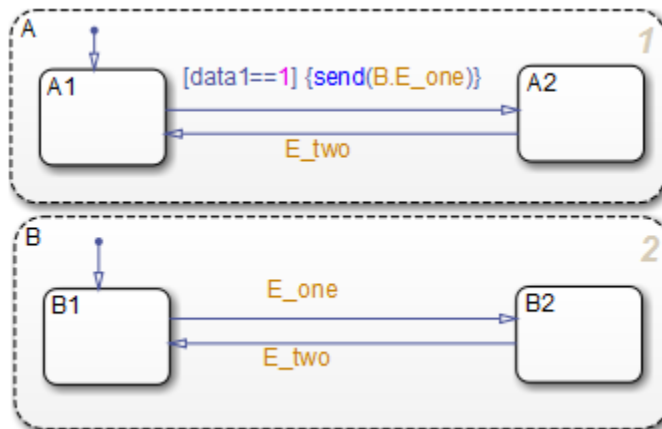
In this example, event E_one belongs to state B and is visible only to that state. See "Directed Event Broadcast Using Qualified Event Name" on page B-51 for more information on the semantics of this notation.

Diagnostic for Detecting Undirected Local Event Broadcasts

If you have *undirected* local event broadcasts in state actions or condition actions in your chart, a warning appears by default during simulation. Examples of state actions with undirected local event broadcasts include:

- `entry: send(E1)`, where E1 is a local event in the chart
- `exit: E2`, where E2 is a local event in the chart

You can control the level of diagnostic action for undirected local event broadcasts in the **Diagnostics** > **Stateflow** pane of the Model Configuration Parameters dialog box. Set the **Undirected event broadcasts** diagnostic to `none`, `warning`, or `error`. For more information, see the documentation for the "Undirected event broadcasts" (Simulink) diagnostic.

See Also
send

More About

- "Synchronize Model Components by Broadcasting Events" on page 10-2
- "Set Properties for an Event" on page 10-6
- "Broadcast Local Events in Parallel States" on page B-50

Control Chart Execution by Using Temporal Logic

Temporal logic controls execution of a chart in terms of time. In state actions and transitions, you can use two types of temporal logic: event-based and absolute-time. Event-based temporal logic tracks recurring event. Absolute-time temporal logic defines time periods based on the simulation time of your chart. To operate on these recurring events or simulation time, you use built-in functions called temporal logic operators.

Rules for Using Temporal Logic Operators

- Temporal logic operators can appear in only:

    - State actions
    - Transitions that originate from states
    - Transition segments that originate from junctions when the full transition path connects two states

    ---
    Note This restriction means that you cannot use temporal logic operators in default transitions or flow chart transitions.

    Every temporal logic operator has an associated state, which is the state in which the action appears or from which the transition originates.

- Use event notation to express event-based temporal logic in state actions. See "Notation for Event-Based Temporal Logic" on page 12-54.

- You can use any explicit or implicit event as a base event for a temporal operator. A base event is a recurring event on which a temporal operator operates.

- For a chart with no input events, you can use the `tick` or `wakeup` event to denote the implicit event of a chart waking up.

- Use one of the keywords `sec` , `msec`, or `usec` to define simulation time in seconds, milliseconds, or microseconds that have elapsed since activation of a state. These keywords are valid only in state actions and in transitions that originate from states.

    Use absolute-time temporal logic instead of the implicit `tick` event for these reasons:

    - Delay expressions that use absolute-time temporal logic are independent of the sample time of the model. However, the `tick` event is dependent on sample time.

- Absolute-time temporal logic works in charts that have function-call input events. The `tick` event does not work in charts with function-call inputs.

Operators for Event-Based Temporal Logic

For event-based temporal logic, use the operators described in this table.

Operator	Syntax	Description
`after`	`after(n,E)` E is the base event for the `after` operator and n is either: • A positive integer • An expression that evaluates to a positive integer value	Returns `true` if the base event E has occurred at least n times since activation of the associated state. Otherwise, the operator returns `false`. In a chart with no input events, `after(n,tick)` or `after(n,wakeup)` returns `true` if the chart has woken up n times or more since activation of the associated state. Resets the counter for E to 0 each time the associated state reactivates.
`at`	`at(n,E)` E is the base event for the `at` operator and n is either: • A positive integer. • An expression that evaluates to a positive integer value.	Returns `true` only at the nth occurrence of the base event E since activation of the associated state. Otherwise, the operator returns `false`. In a chart with no input events, `at(n,tick)` or `at(n,wakeup)` returns `true` if the chart has woken up for the nth time since activation of the associated state. Resets the counter for E to 0 each time the associated state reactivates.

Operator	Syntax	Description
before	before(n,E) E is the base event for the before operator and n is either: • A positive integer. • An expression that evaluates to a positive integer value.	Returns true if the base event E has occurred fewer than n times since activation of the associated state. Otherwise, the operator returns false. In a chart with no input events, before(n,tick) or before(n,wakeup) returns true if the chart has woken up fewer than n times since activation of the associated state. Resets the counter for E to 0 each time the associated state reactivates.
every	every(n,E) E is the base event for the every operator and n is either: • A positive integer. • An expression that evaluates to a positive integer value.	Returns true at every n^{th} occurrence of the base event E since activation of the associated state. Otherwise, the operator returns false. In a chart with no input events, every(n,tick) or every(n,wakeup) returns true if the chart has woken up an integer multiple of n times since activation of the associated state. Resets the counter for E to 0 each time the associated state reactivates.

Operator	Syntax	Description
temporalCount	temporalCount(E) E is the base event for the temporalCount operator.	Increments by 1 and returns a positive integer value for each occurrence of the base event E that takes place after activation of the associated state. Otherwise, the operator returns a value of 0. In a chart with no input events, temporalCount(tick) or temporalCount(wakeup) returns the number of times that the chart has woken up since activation of the associated state. Resets the counter for E to 0 each time the associated state reactivates.

You can use quotation marks to enclose the keywords `'tick'` and `'wakeup'`. For example, `after(5,'tick')` is equivalent to `after(5,tick)`.

Note Temporal logic operators compare the threshold n to an internal counter of integer type. If n is a fixed-point number defined by using either a slope that is not an integer power of two or a nonzero bias, then the comparison can yield unexpected results due to rounding. For more information, see "Relational Operations for Fixed-Point Data" on page 22-33.

Examples of Event-Based Temporal Logic

These examples illustrate usage of event-based temporal logic in state actions and transitions.

Operator	Usage	Example	Description
after	State action (on after)	on after(5,CLK): status('on');	A status message appears during each CLK cycle, starting 5 clock cycles after activation of the state.
after	Transition	ROTATE[after(10,CLK)]	A transition out of the associated state occurs only on broadcast of a ROTATE event, but no sooner than 10 CLK cycles after activation of the state.
before	State action (on before)	on before(MAX,CLK): temp++;	The temp variable increments once per CLK cycle until the state reaches the MAX limit.
before	Transition	ROTATE[before(10,CLK)]	A transition out of the associated state occurs only on broadcast of a ROTATE event, but no later than 10 CLK cycles after activation of the state.
at	State action (on at)	on at(10,CLK): status('on');	A status message appears at exactly 10 CLK cycles after activation of the state.

Operator	Usage	Example	Description
`at`	Transition	`ROTATE[at(10,CLK)]`	A transition out of the associated state occurs only on broadcast of a `ROTATE` event, at exactly 10 `CLK` cycles after activation of the state.
`every`	State action (`on every`)	`on every(5,CLK): status('on');`	A status message appears every 5 `CLK` cycles after activation of the state.
`temporalCount`	State action (`during`)	`du: y = mm[temporalCount(tick)];`	This action counts and returns the integer number of ticks that have elapsed since activation of the state. Then, the action assigns to the variable y the value of the `mm` array whose index is the value that the `temporalCount` operator returns.

Notation for Event-Based Temporal Logic

You can use one of two notations to express event-based temporal logic.

Event Notation

Use event notation to define a state action or a transition condition that depends only on a base event.

Event notation follows this syntax:

tlo(n,E)[C]

where

- *tlo* is a Boolean temporal logic operator (`after`, `before`, `at`, or `every`)
- n is the occurrence count of the operator
- E is the base event of the operator
- C is an optional condition expression

Conditional Notation

Use conditional notation to define a transition condition that depends on base and nonbase events.

Conditional notation follows this syntax:

E1[*tlo*(n,E2) && C]

where

- E1 is any nonbase event
- *tlo* is a Boolean temporal logic operator (`after`, `before`, `at`, or `every`)
- n is the occurrence count of the operator
- E2 is the base event of the operator
- C is an optional condition expression

Examples of Event and Conditional Notation

Notation	Usage	Example	Description
Event	State action (on after)	`on after(5,CLK): temp = WARM;`	The `temp` variable becomes WARM 5 CLK cycles after activation of the state.
Event	Transition	`after(10,CLK)[temp == COLD]`	A transition out of the associated state occurs if the `temp` variable is COLD, but no sooner than 10 CLK cycles after activation of the state.

Notation	Usage	Example	Description
Conditional	Transition	`ON[after(5,CLK) && temp == COLD]`	A transition out of the associated state occurs only on broadcast of an `ON` event, but no sooner than 5 `CLK` cycles after activation of the state and only if the `temp` variable is `COLD`.

Note You must use event notation in state actions because the syntax of state actions does not support conditional notation.

Operators for Absolute-Time Temporal Logic

For absolute-time temporal logic, use the operators described in this table.

Operator	Syntax	Description

`after`	`after(n,sec)`
	
	`after(n,msec)`
	
	`after(n,usec)`
	
	n is any positive number or expression. `sec`, `msec`, and `usec` are keywords that denote the simulation time elapsed since activation of the associated state.
		Returns `true` if n units of simulation time have elapsed since activation of the associated state. Otherwise, the operator returns `false`. Specify simulation time in seconds (`sec`), milliseconds (`msec`), or microseconds (`usec`).
		
		Resets the counter for `sec`, `msec`, and `usec` to 0 each time the associated state reactivates.

Operator	Syntax	Description
before	before(n,sec) before(n,msec) before(n,usec) n is any positive number or expression. sec, msec, and usec are keywords that denote the simulation time elapsed since activation of the associated state.	Returns true if fewer than n units of simulation time have elapsed since activation of the associated state. Otherwise, the operator returns false. Specify simulation time in seconds (sec), milliseconds (msec), or microseconds (usec). Resets the counter for sec, msec, and usec to 0 each time the associated state reactivates.
every	every(n,sec) every(n,msec) every(n,usec) n is any positive number or expression. sec, msec, and usec are keywords that denote the simulation time elapsed since activation of the associated state.	Returns true every n units of simulation time since activation of the associated state. Otherwise, the operator returns false. Specify simulation time in seconds (sec), milliseconds (msec), or microseconds (usec). Resets the counter for sec, msec, and usec to 0 each time the associated state reactivates. Use of every as an absolute-time temporal logic operator is supported only in standalone charts for execution as MATLAB objects.
temporalCount	temporalCount(sec) temporalCount(msec) temporalCount(usec) sec, msec, and usec are keywords that denote the simulation time elapsed since activation of the associated state.	Counts and returns the number of specified seconds (sec), milliseconds (msec), or microseconds (usec) of simulation time that have elapsed since activation of the associated state. Resets the counter for sec, msec and usec to 0 each time the associated state reactivates.

12-57

Operator	Syntax	Description
elapsed	elapsed(sec)	Equivalent to temporalCount(sec). Returns the simulation time in seconds (sec) that has elapsed since the activation of the associated state. Resets the counter for sec to 0 each time the associated state reactivates.
count	count(C)	Returns the number of ticks after the conditional expression, C, becomes true. The count operator is reset if the conditional expression becomes false. If the count operator is used within a state, it is reset when the state that contains it is entered. If the count operator is used on a transition, it is reset when the source state for that transition is entered. In a Simulink model, the value for count depends on the step size. Changing the solver or step size for your model affects the result for Stateflow charts that include the count operator.
duration	duration(C)	Returns the number of seconds after the conditional expression, C, becomes true. The duration operator is reset if the conditional expression becomes false. If the duration operator is used within a state, it is reset when the state that contains it is entered. If the duration operator is used on a transition, it is reset when the source state for that transition is entered.

You can use quotation marks to enclose the keywords `'sec'`, `'msec'`, and `'usec'`. For example, `after(5,'sec')` is equivalent to `after(5,sec)`.

Note Temporal logic operators compare the threshold n to an internal counter of integer type. If n is a fixed-point number defined by using either a slope that is not an integer power of two or a nonzero bias, then the comparison can yield unexpected results due to rounding. For more information, see "Relational Operations for Fixed-Point Data" on page 22-33.

Examples of Absolute-Time Temporal Logic

These examples illustrate absolute-time temporal logic in state actions and transitions.

Operator	Usage	Example	Description
after	State action (on after)	`on after(12.3,sec): temp = LOW;`	After 12.3 seconds of simulation time since activation of the state, `temp` variable becomes `LOW`.
after	Transition	`after(8,msec)`	After 8 milliseconds of simulation time have passed since activation of the state, a transition out of the associated state occurs.
after	Transition	`after(5,usec)`	After 5 microseconds of simulation time have passed since activation of the state, a transition out of the associated state occurs.

Operator	Usage	Example	Description
before	Transition	`[temp > 75 && before(12.34,sec)]`	If the variable `temp` exceeds 75 and fewer than 12.34 seconds have elapsed since activation of the state, a transition out of the associated state occurs.
temporalCount	State action (exit)	`exit: y = temporalCount(sec);`	This action counts and returns the number of seconds of simulation time that pass between activation and deactivation of the state.

Example of Defining Time Delays

This continuous-time chart defines two absolute time delays in transitions.
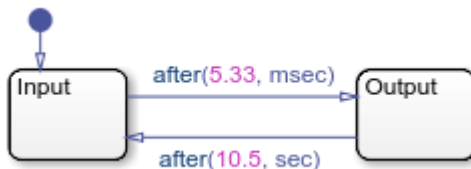
Chart execution consists of these steps:

- When the chart awakens, the state `Input` activates first.
- After 5.33 milliseconds of simulation time, the transition from `Input` to `Output` occurs.
- The state `Input` deactivates, and then the state `Output` activates.
- After 10.5 seconds of simulation time, the transition from `Output` to `Input` occurs.
- The state `Output` deactivates, and then the state `Input` activates.
- Steps 2 through 5 are repeated, until the simulation ends.

If a chart has a discrete sample time, any action in the chart occurs at integer multiples of this sample time. For example, suppose that you change the configuration parameters so that the Simulink® solver uses a fixed step of size 0.1 seconds. Then, the first transition from state Input to state Output occurs at t = 0.1 seconds. This behavior applies because the solver does not wake the chart at exactly t = 5.33 milliseconds. Instead, the solver wakes the chart at integer multiples of 0.1 seconds, such as t = 0.0 and 0.1 seconds.

Example of Detecting Elapsed Time

In this model, the Step block provides a unit step input to the chart.
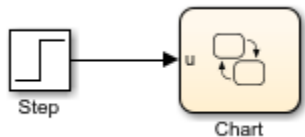
The chart determines when the input u equals 1:
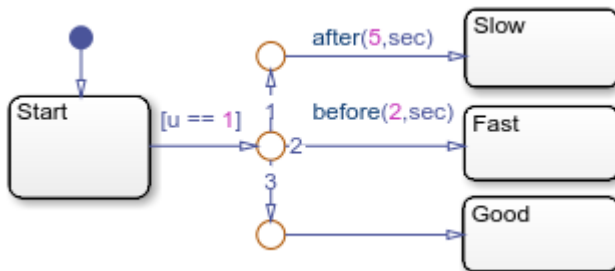
- If the input equals 1 before t = 2 seconds, a transition occurs from Start to Fast.
- If the input equals 1 between t = 2 and t = 5 seconds, a transition occurs from Start to Good.
- If the input equals 1 after t = 5 seconds, a transition occurs from Start to Slow.

Example of Absolute-Time Temporal Logic in an Enabled Subsystem

You can use absolute-time temporal logic in a chart that resides in a *conditionally executed* subsystem. When the subsystem is disabled, the chart becomes inactive and the

temporal logic operator pauses while the chart is asleep. The operator does not continue to count simulation time until the subsystem is reenabled and the chart is awake.

This model has an enabled subsystem with the **States when enabling** parameter set to held.
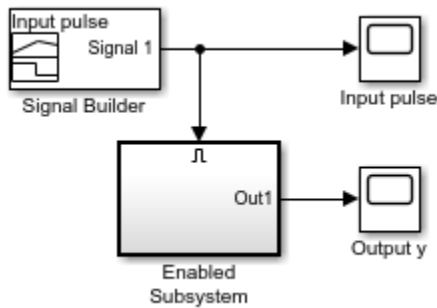
The subsystem contains a chart that uses the after operator.
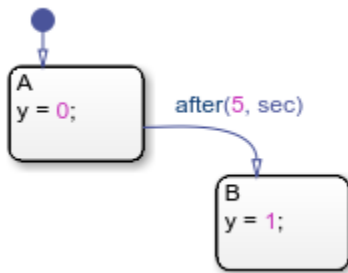
The Signal Builder block provides an input signal with these characteristics:

- Signal enables subsystem at t = 0.
- Signal disables subsystem at t = 2.
- Signal reenables subsystem at t = 6.
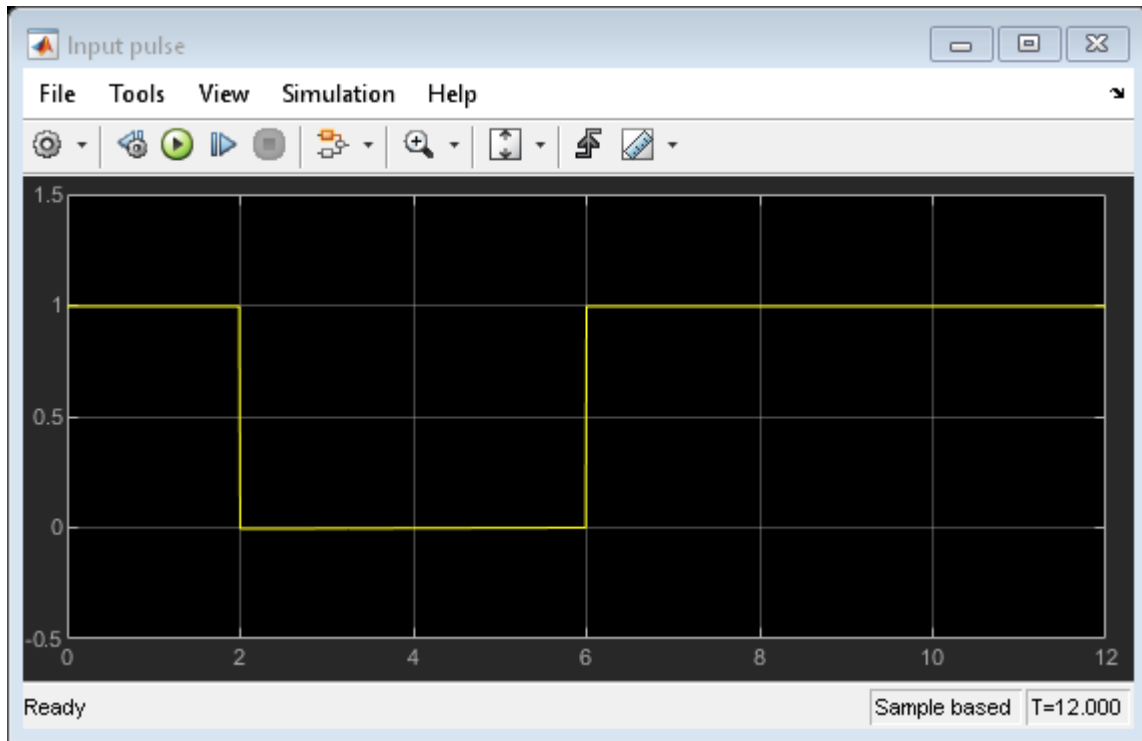
This graph shows the total time elapsed in an enabled state (either A or B).
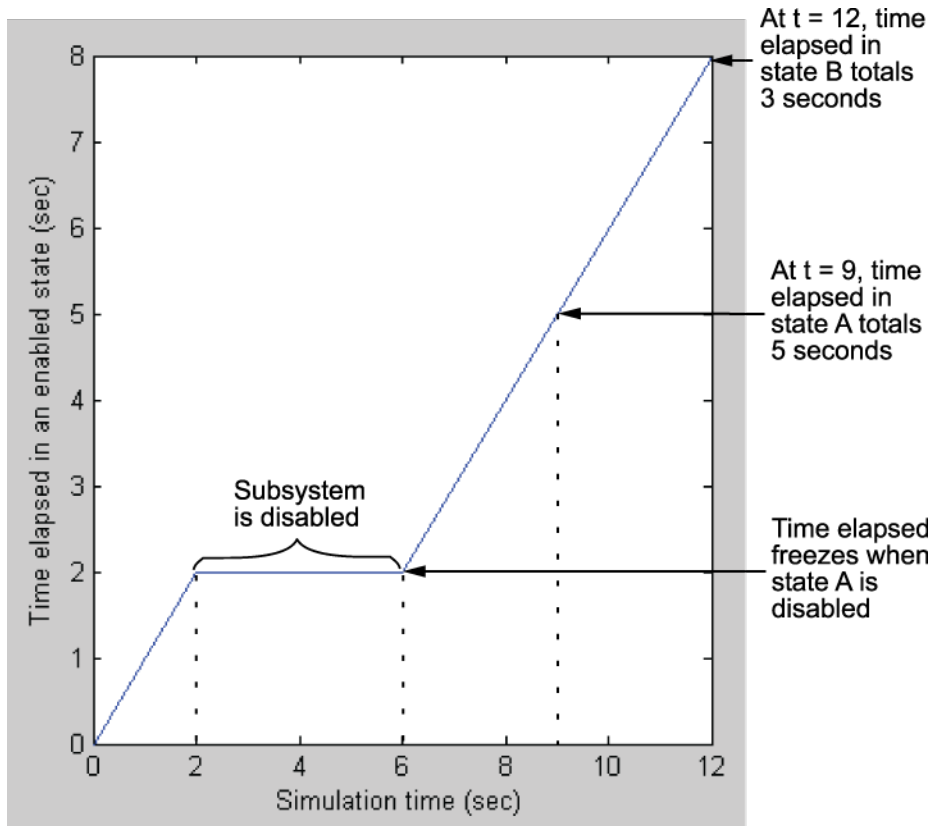
When the input signal enables the subsystem at time t = 0, the state A becomes active, or enabled. While the state is active, the time elapsed increases. However, when the subsystem is disabled at t = 2, the chart goes to sleep and state A becomes inactive.

For 2 < t < 6, the time elapsed in an enabled state stays frozen at 2 seconds because neither state is active. When the chart wakes up at t = 6, state A becomes active again and time elapsed starts to increase. The transition from state A to state B depends on the time elapsed while state A is enabled, not on the simulation time. Therefore, state A stays active until t = 9, so that the time elapsed in that state totals 5 seconds.

When the transition from A to B occurs at t = 9, the output value y changes from 0 to 1.
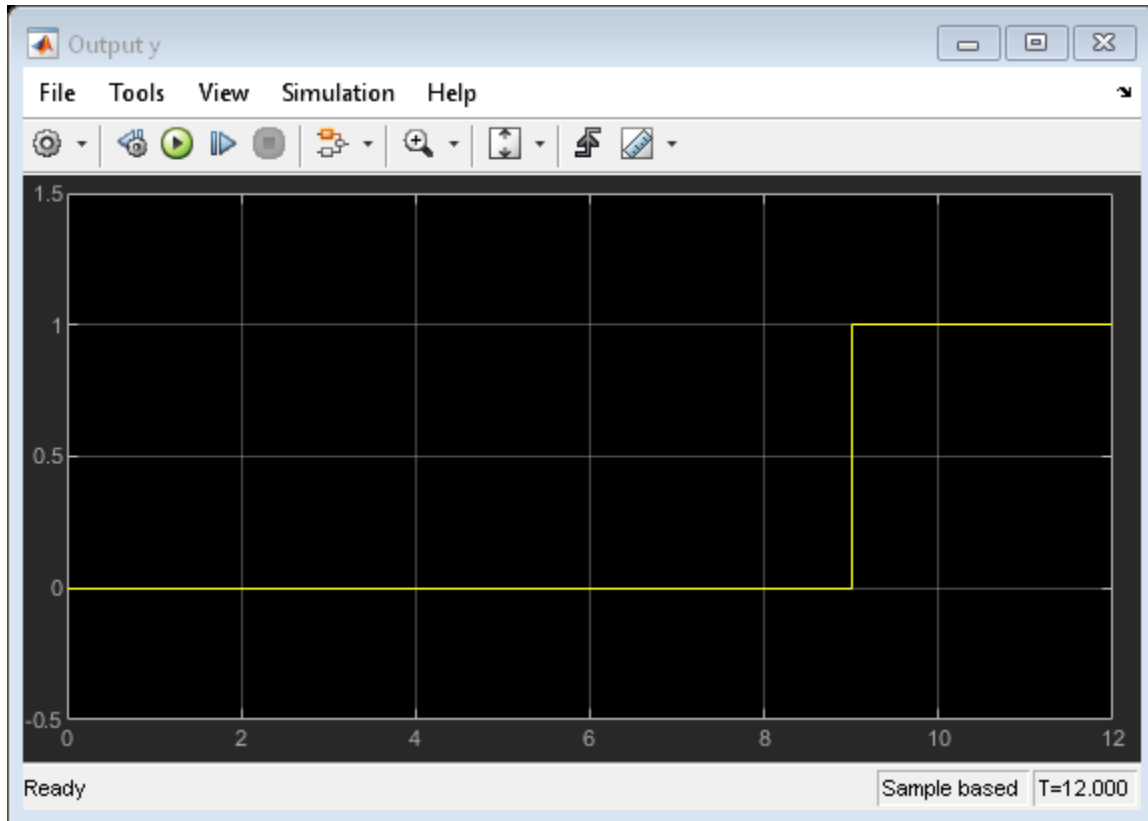
This model behavior applies only to subsystems where you set the Enable block parameter **States when enabling** to `held`. If you set the parameter to `reset`, the chart reinitializes completely when the subsystem is reenabled. In other words, default transitions execute and any temporal logic counters reset to 0.

These semantics also apply to the `before` operator.

Best Practices for Absolute-Time Temporal Logic

Use the after Operator to Replace the at Operator

If you use the `at` operator with absolute-time temporal logic, an error message appears when you try to simulate your model. Use the `after` operator instead.

Suppose that you want to define a time delay using the transition `at(5.33, sec)`.
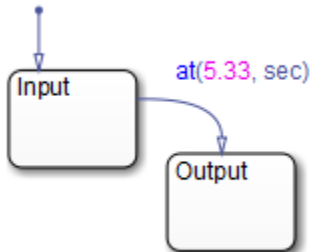
Change the transition to `after(5.33, sec)`, as shown in this chart.
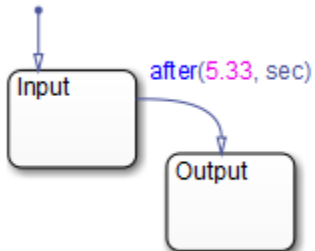
Use an Outer Self-Loop Transition with the after Operator to Replace the every Operator

In a Stateflow model, if you use the `every` operator with absolute-time temporal logic, an error message appears when you try to simulate your model. Use an outer self-loop transition with the `after` operator instead.

Suppose that you want to print a status message for an active state every 2.5 seconds during chart execution, as shown in the state action of `Check_status`.
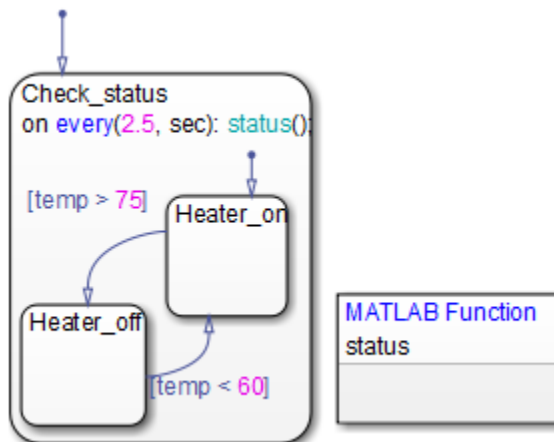
Replace the state action with an outer self-loop transition, as shown in this chart.
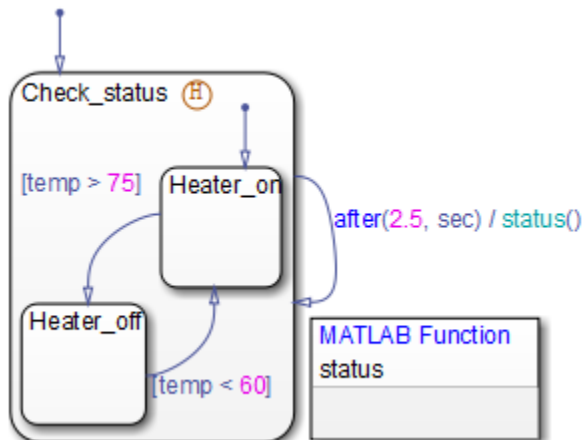
Also add a history junction in the state so that the chart remembers the state settings prior to each self-loop transition. See "Record State Activity by Using History Junctions" on page 8-2.

Use Charts with Discrete Sample Times for More Efficient Code Generation

The code generated for discrete charts that are not inside a triggered or enabled subsystem uses integer counters to track time instead of Simulink provided time. This

allows for more efficient code generation in terms of overhead and memory, as well as enabling this code for use in software-in-the-loop (SIL) and processor-in-the-loop (PIL) simulation modes. For more information, see "SIL and PIL Simulations" (Embedded Coder).

See Also

after | at | before | count | duration | elapsed | every | temporalCount

More About

- "Control Oscillations by Using the duration Operator" on page 12-95
- "Reduce Transient Signals by Using Debouncing Logic" on page 26-2

Detect Changes in Data Values

Stateflow charts can detect changes in data values from one time step to the next time step. All Stateflow charts can detect changes in chart input data. Charts that use C as the action language can also detect changes in chart output data, local chart variables, and data store memory data.

Change detection is supported only in Stateflow charts in Simulink models.

Change Detection Operators

To detect changes in Stateflow data, use the operators listed in this table.

Operator	Syntax	Description	Example
hasChanged	`tf = hasChanged(u)`	Detects changes in data value from the beginning of the last time step to the beginning of the current time step.	Returns `true` if u has changed value. `hasChanged(u)`
hasChangedFrom	`tf = hasChangedFrom(u,v)`	Detects changes in data value from a specified value at the beginning of the last time step to a different value at the beginning of the current time step.	Returns `true` if u has changed from the value v. `hasChangedFrom(u,v)`
hasChangedTo	`tf = hasChangedTo(u,v)`	Detects changes in data value to a specified value at the beginning of the current time step from a different value at the beginning of the last time step.	Returns `true` if u has changed to the value v. `hasChangedTo(u,v)`

Change detection operators return `true` if there is a change in data value and `false` if there is no change in data value.

The argument u can be:

- A scalar variable.
- A matrix or an element of a matrix. See "Supported Operations for Vectors and Matrices" on page 17-11.
- A structure or a field in a structure. See "Index and Assign Values to Stateflow Structures" on page 25-8.
- Any valid combination of structure fields or matrix indices.

Indices can be numbers or expressions that evaluate to a scalar value.

For the `hasChangedFrom` and `hasChangedTo` operators, the second argument `v` can be any expression that resolves to a value that is comparable with `u`.

- If `u` is a scalar, then `v` must resolve to a scalar value.
- If `u` is a matrix, then `v` must resolve to a matrix value with the same dimensions as `u`.

  Alternatively, in a chart that uses C as the action language, `v` can resolve to a scalar value. The chart uses scalar expansion to compare `u` to a matrix whose elements are all equal to the value specified by `v`. See "Convert Scalars to Nonscalars by Using Scalar Expansion" on page 17-9.
- If `u` is a structure, then `v` must resolve to a structure value whose field specification matches `u` exactly.

The scope of `u` depends on the chart action language:

- MATLAB as the action language: `Input` only.
- C as the action language: `Input`, `Output`, `Local`, or `Data Store Memory`.

The argument `u` cannot be a nontrivial expression or a custom code variable.

Example of Chart with Change Detection

This model shows how the operators `hasChanged`, `hasChangedFrom`, and `hasChangedTo` detect specific changes in an input signal. In this example, a Ramp block sends a discrete, increasing time signal to a chart.
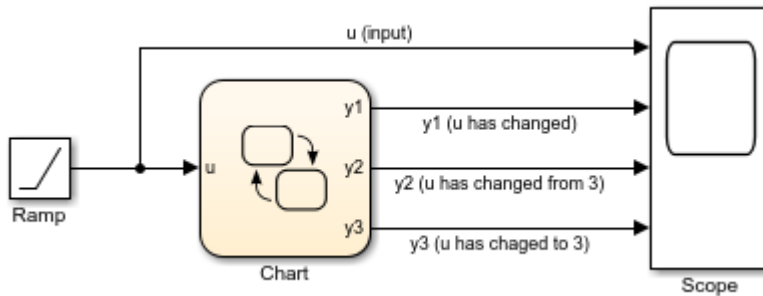
The model uses a fixed-step solver with a step size of 1. The signal increments by 1 at each time step. At each time step, the chart analyzes the input signal u for these changes:

- Any change from the previous time step
- A change to the value 3
- A change from the value 3

To check the signal, the chart calls three change detection operators in a transition action. The chart outputs the return values as y1, y2, and y3.
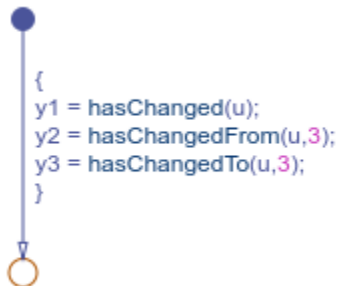
During simulation, the Scope block shows the input and output signals to the chart.

- u increases by 1 at every time step.
- y1 transitions to a value of 1 at time t = 1. y1 remains 1 because u continues to change at each subsequent time step.
- y2 transitions to 1 at time t = 4 when the value of u changes from 3 to 4. y2 transitions back to 0 at time t = 5 when u increases from 4 to 5.

12-71

- y3 transitions to 1 at time t = 3 when the value of u changes from 2 to 3. y3 transitions back to 0 at time t = 4 when u increases from 3 to 4.
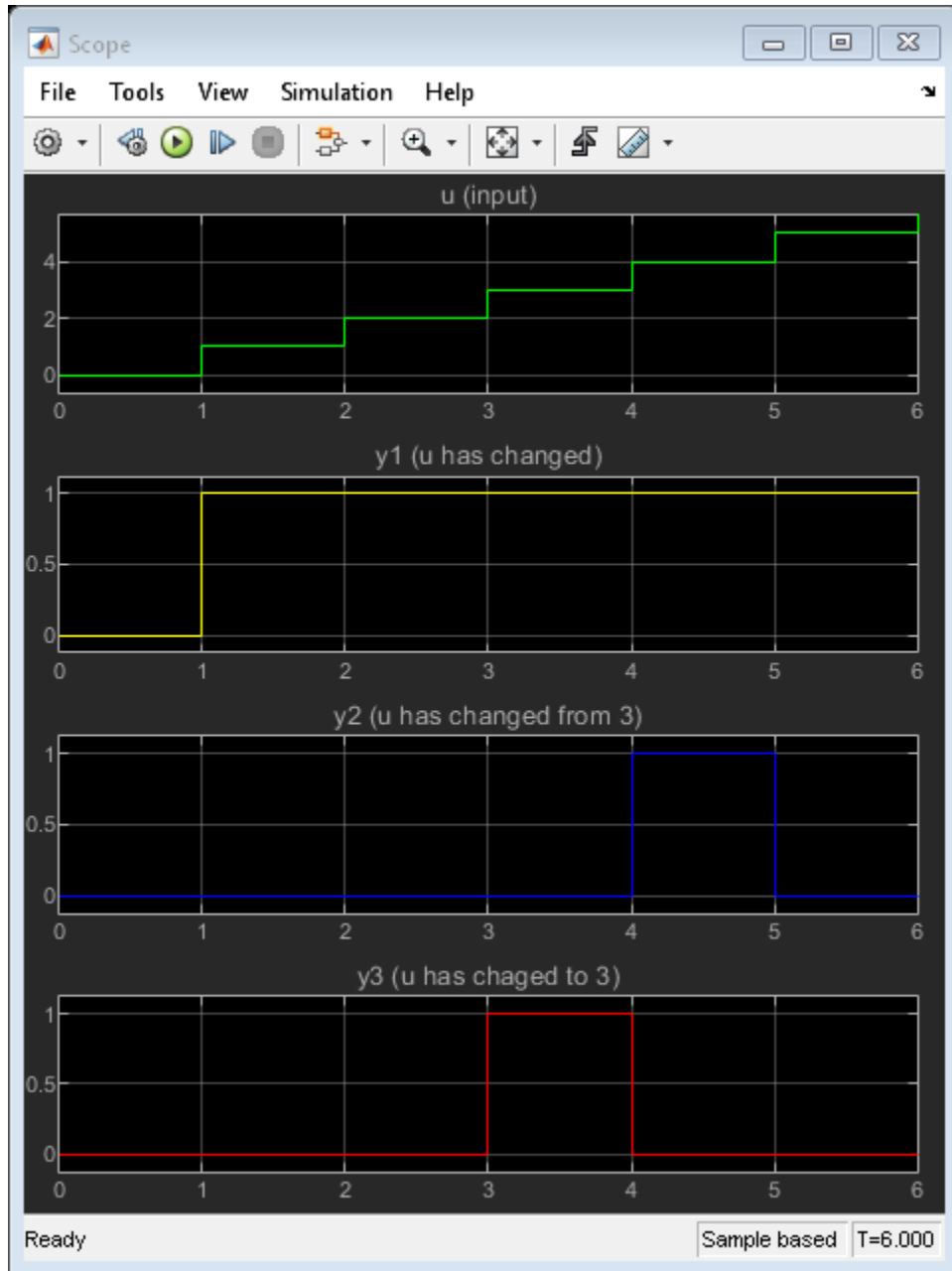
Implementation of Change Detection

A chart detects changes in chart data by evaluating values at time step boundaries. The

chart compares the value at the beginning of the previous execution step with the value at the beginning of the current execution step. To detect changes, the chart double-buffers these values in local variables.

Local Buffer	Description
*var_name*_prev	Value of data *var_name* at the beginning of the last time step
*var_name*_start	Value of data *var_name* at the beginning of the current time step

Double-buffering occurs once per time step except when multiple input events occur in the same time step. If multiple input events occur in the same time step, double-buffering occurs once per input event. See "Detect Value Changes Between Input Events" on page 12-76.

When you invoke a change detection operator in an action, the Stateflow chart:

- Double-buffers data values *after* a Simulink event triggers the chart but *before* the chart begins execution.
- Compares values in _prev and _start buffers. If the values match, the change detection operator returns false (no change); otherwise, it returns true (change).

This diagram places these tasks in the context of the chart life cycle.

event_n

event_2 ...

Simulink
event triggers
Stateflow
chart

KEY:

X_{prev} Value of data
at beginning of
last time step

X_{start} Value of data
at beginning of
current time step

X Value of data
at end of
current time step

Stateflow software
double-buffers data

$X_{prev} = X_{start}$
$X_{start} = X$

Is
chart
active?

Yes

No

Make
chart
active

Initialize chart
by executing
default
transition
paths

Chart
executes
active children
hierarchically
in top-down
order

hasChanged (x)
compares
X_{prev} and X_{start}

12-75

Chart goes
to sleep

Buffering occurs before chart execution and affects change detection when:

- Local variables have transient changes in value.
- Multiple input events occur in the same time step.

Filter Out Transient Value Changes

Stateflow charts attempt to filter out transient changes in local chart variables by evaluating their values only at time boundaries. The chart evaluates the specified local variable only once at the end of the execution step. The return value of the change detection operators remains constant even if the value of the local variable fluctuates within a given time step.

For example, suppose that in the current time step a local variable `temp` changes from its value at the previous time step but then reverts to the original value. The operator `hasChanged(temp)` returns `false` for the next time step, indicating that no change occurred.

Detect Value Changes Between Input Events

When multiple input events occur in the same time step, the Stateflow chart updates the `_prev` and `_start` buffers once per event. A chart detects changes between input events, even if the changes occur more than once in a given time step.

See Also

hasChanged | hasChangedFrom | hasChangedTo

More About

- "Tetris"

Check State Activity by Using the in Operator

In a Stateflow chart with parallel state decomposition, substates can be active at the same time. If you check state activity, you can synchronize substates in two parallel states.

For example, this chart has two parallel states: `Place` and `Tracker`. The transitions in `Tracker` check the state activity in `Place` and keep the substates synchronized. A change of active substate in `Place` causes a corresponding change of active substate in `Tracker`.

- If R becomes the active substate in `Place`, then `Moved_Right` becomes the active substate in `Tracker`.
- If L becomes the active substate in `Place`, then `Moved_Left` becomes the active substate in `Tracker`.
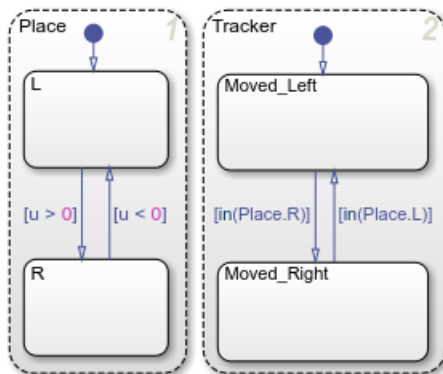
The in Operator

To check if a state is active in a given time step during chart execution, use the `in` operator:

```
in(S)
```

The `in` operator takes a qualified state name S and returns a Boolean output. If state S is active, `in` returns a value of 1. Otherwise, `in` returns a value of 0.

You can use the `in` operator in state actions and in transitions that originate from states.

Resolution of State Activity

Checking state activity is a two-part process. First, Stateflow resolves the qualified state name by performing a localized search of the chart hierarchy for a matching state. Then, Stateflow determines if the matching state is active.

The search begins at the hierarchy level where the qualified state name appears:

- For a state action, the starting point is the state containing the action.
- For a transition label, the starting point is the parent of the transition source.

The resolution process searches each level of the chart hierarchy for a path to the state. If a state matches the path, the process adds that state to the list of possible matches. Then, the process continues the search one level higher in the hierarchy. The resolution process stops after it searches the chart level of the hierarchy. If a unique match exists, the `in` operator checks if the matching state is active. Otherwise, the resolution process fails. Simulation stops and you see an error message.

This flow chart illustrates the different stages in the process for checking state activity.
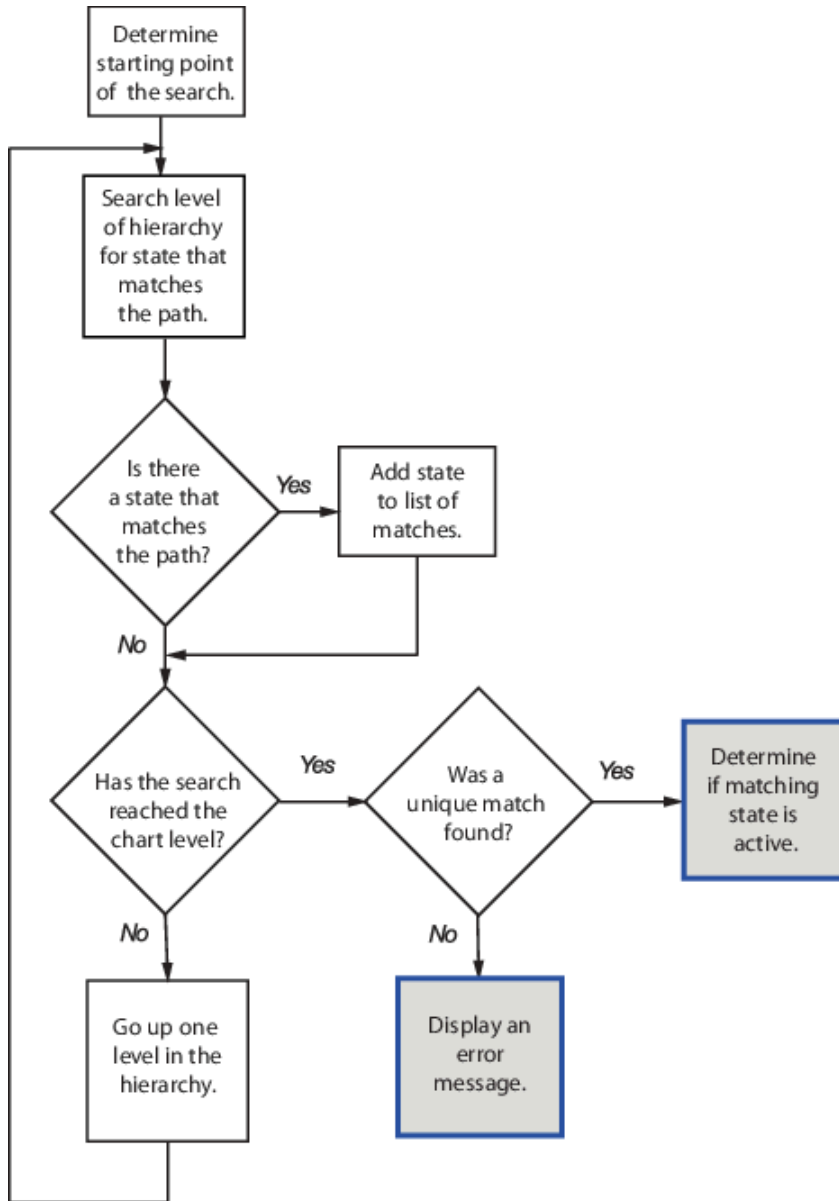
Best Practices for Checking State Activity

Resolving state activity:

- Does not perform an exhaustive search for all states in a chart.
- Does not stop after finding the first match.

To improve the chances of finding a unique search result when resolving qualified data names:

- Use specific paths in qualified data names.
- Give states unique names.
- Use states and boxes as enclosures to limit the scope of the path resolution search.

Examples of State Activity Resolution

Search Finds Local Copy of Substate

This chart contains parallel states A and B that have identical substates A1 and A2. The condition in(A1.Y) guards the transition from P to Q in A.A2 and in B.A2. Stateflow resolves each qualified state name as the local copy of the substate Y:

- In the state A, the condition in(A1.Y) checks the activity of state A.A1.Y.
- In the state B, the condition in(A1.Y) checks the activity of state B.A1.Y.
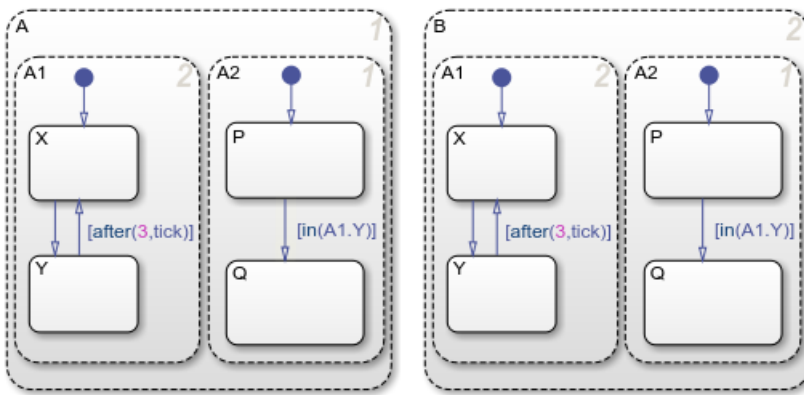
This table lists the different stages in the resolution process for the transition condition in state `A.A2`.

Stage	Description	Result
1	Starting in state `A.A2`, search for the state `A.A2.A1.Y`.	No match found.
2	Move up to the next level of the hierarchy (state A). Search for the state `A.A1.Y`	Match found.
3	Move up to the next level of the hierarchy (the chart level). Search for the state `A1.Y`	No match found.

The search ends with a single match found. Because the resolution algorithm localizes the scope of the search, the `in` operator guarding the transition in `A.A2` detects only the state `A.A1.Y`. The `in` operator guarding the transition in `B.A2` detects only the state `B.A1.Y`.

To check the state activity of the other copy of `Y`, use more specific qualified state names:

- In state `A`, use the expression `in(B.A1.Y)`.
- In state `B`, use the expression `in(A.A1.Y)`.

Search Produces No Matches

In this chart, the `during` action in state `A.B` contains the expression `in(Q.R)`. Stateflow cannot resolve the qualified state name `Q.R`.
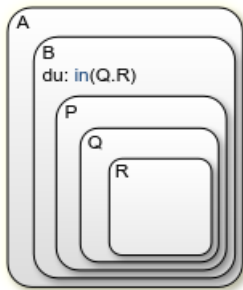
This table lists the different stages in the resolution process.

Stage	Description	Result
1	Starting in state `A.B`, search for the state `A.B.Q.R`.	No match found.
2	Move up to the next level of the hierarchy (state A). Search for the state `A.Q.R`.	No match found.
3	Move up to the next level of the hierarchy (the chart level). Search for the state `Q.R`.	No match found.

The search ends at the chart level with no match found for `Q.R`, resulting in an error.

To avoid this error, use a more specific qualified state name. For instance, check state activity by using the expression `in(P.Q.R)`.

Search Finds the Wrong State

In this chart, the `during` action in state `A.B` contains the expression `in(Q.R)`. When resolving the qualified state name `Q.R`, Stateflow cannot detect the substate `A.B.P.Q.R`.
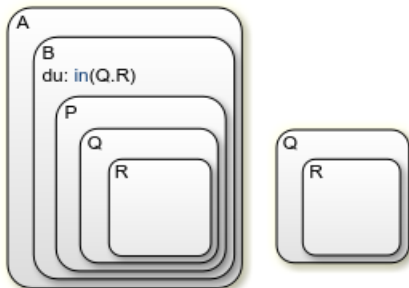
This table lists the different stages in the resolution process.

Stage	Description	Result
1	Starting in state `A.B`, search for the state `A.B.Q.R`.	No match found
2	Move up to the next level of the hierarchy (state A). Search for the state `A.Q.R`.	No match found.
3	Move up to the next level of the hierarchy (the chart level). Search for the state `Q.R`.	Match found.

The search ends with a single match found. The `in` operator detects only the substate `R` of the top-level state `Q`.

To check the state activity of `A.B.P.Q.R`, use a more specific qualified state name. For instance, use the expression `in(P.Q.R)`.

Search Produces Multiple Matches

In this chart, the `during` action in state `A.B` contains the expression `in(P.Q.R)`. Stateflow cannot resolve the qualified state name `P.Q.R`.
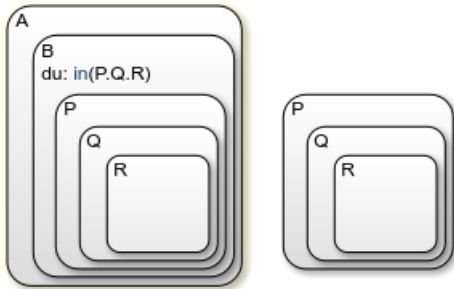
This table lists the different stages in the resolution process.

Stage	Description	Result
1	Starting in state `A.B`, search for the state `A.B.P.Q.R`.	Match found
2	Move up to the next level of the hierarchy (state A). Search for the state `A.P.Q.R`.	No match found.
3	Move up to the next level of the hierarchy (the chart level). Search for the state `P.Q.R`.	Match found.

The search ends at the chart level with two matches found for `P.Q.R`, resulting in an error.

To avoid this error:

- Use a more specific qualified state name. For instance:

  - To check the substate activity inside B, use the expression `in(B.P.Q.R)`.

- To check the substate activity in the top-level state P, use the expression `in(\P.Q.R)`.
- Rename one of the matching states.
- Enclose the top-level state P in a box or another state. Adding an enclosure prevents the search process from detecting substates in the top-level state.
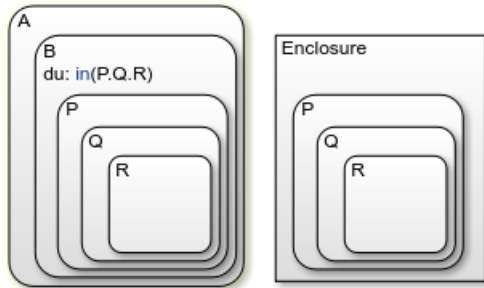
See Also

More About

Control Function-Call Subsystems by Using bind Actions

You can bind specified data and events to a state by using `bind` actions. Events bound to a state can be broadcast only by the actions in that state or its children. You can also bind a function-call event to a state to enable or disable the function-call subsystem that the event triggers. The function-call subsystem enables when the state with the bound event is entered and disables when that state is exited. Execution of the function-call subsystem is fully bound to the activity of the state that calls it.

Bind a Function-Call Subsystem to a State

By default, a function-call subsystem is controlled by the chart in which the associated function call output event is defined. This association means that the function-call subsystem is enabled when the chart wakes up and remains active until the chart goes to sleep. To achieve a finer level of control, you can bind a function-call subsystem to a state within the chart hierarchy by using a `bind` action (see "bind Actions" on page 12-4).

You can bind function-call output events to a state. When you create this type of binding, the function-call subsystem that is called by the event is also bound to the state. In this situation, the function-call subsystem is enabled when the state is entered and disabled when the state is exited.

When you bind a function-call subsystem to a state, you can fine-tune the behavior of the subsystem when it is enabled and disabled, as described in the following sections:

- "Handle Outputs When the Subsystem is Disabled" on page 12-85
- "Control Behavior of States When the Subsystem is Enabled" on page 12-87

Handle Outputs When the Subsystem is Disabled

Although function-call subsystems do not execute while disabled, their output signals are available to other blocks in the model. If a function-call subsystem is bound to a state, you can hold its outputs at their values from the previous time step or reset the outputs to their initial values when the subsystem is disabled. Follow these steps:

1   Double-click the outport block of the subsystem to open the Block Parameters dialog box.

Sink Block Parameters: Out1

Outport

Provide an output port for a subsystem or model. The 'Output when disabled' and 'Initial output' parameters only apply to conditionally executed subsystems. When a conditionally executed subsystem is disabled, the output is either held at its last value or set to the 'Initial output'.

| Main | Signal Attributes |

Port number:

```
1
```

Icon display: | Port number ▼ |

Source of initial output value: | Dialog ▼ |

Output when disabled: | reset ▼ |

Initial output:

```
0
```

| OK | Cancel | Help | Apply |

2 Select an option for **Output when disabled**:

Select:	To:
held	Maintain most recent output value

Select:	To:
reset	Reset output to its initial value

3 Click **OK** to record the settings.

> **Note** Setting **Output when disabled** is meaningful only when the function-call subsystem is bound to a state, as described in "Bind a Function-Call Subsystem to a State" on page 12-85.

Control Behavior of States When the Subsystem is Enabled

If a function-call subsystem is bound to a state, you can hold the subsystem state variables at their values from the previous time step or reset the state variables to their initial conditions when the subsystem executes. In this way, the binding state gains full control of state variables for the function-call subsystem. Follow these steps:

1 Double-click the trigger port of the subsystem to open the Block Parameters dialog box.
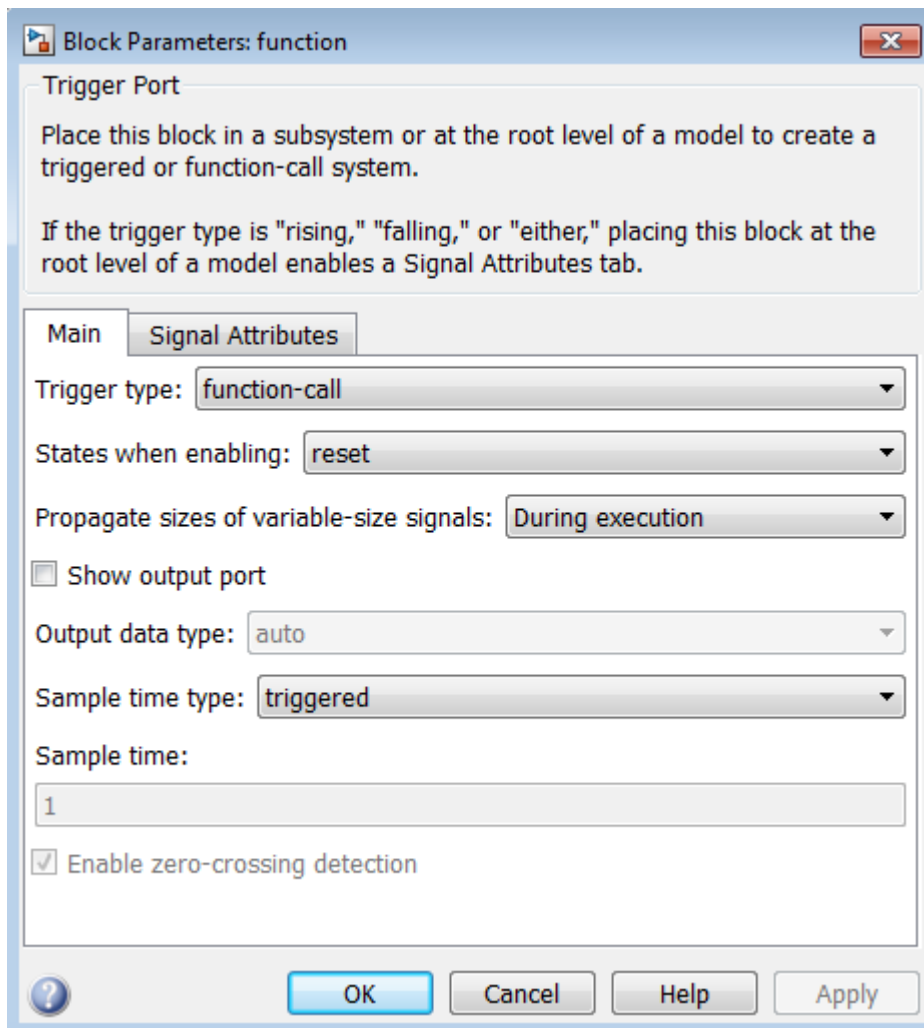
2   Select an option for **States when enabling**:

Select:	To:
held	Maintain most recent values of the states of the subsystem that contains the trigger port

Select:	To:
reset	Revert to the initial conditions of the states of the subsystem that contains this trigger port
inherit	Inherit this setting from the function-call initiator's parent subsystem. If the parent of the initiator is the model root, the inherited setting is held. If the trigger has multiple initiators, the parents of all initiators must have the same setting: either all `held` or all `reset`.

3 Click **OK** to record the settings.

Note Setting **States when enabling** is meaningful only when the function-call subsystem is bound to a state, as described in "Bind a Function-Call Subsystem to a State" on page 12-85.

Bind a Function-Call Subsystem to a State

This model triggers a function-call subsystem with a trigger event E that binds to a state of a chart. In the **Solver** pane of the Model Configuration Parameters dialog box, the model specifies a fixed-step solver with a fixed-step size of 1.
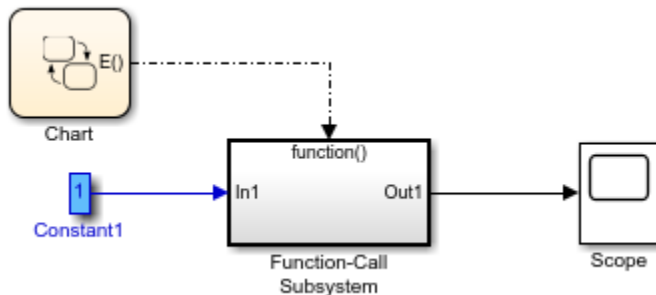
The chart contains two states. Event E binds to state A with the action

```
bind:E
```

Event E is defined for the chart with a scope of `Output to Simulink` and a trigger type of `function-call`.
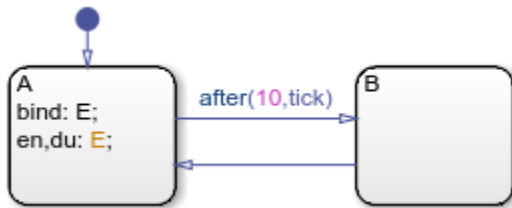
The function-call subsystem contains a trigger port block, an input port, an output port, and a simple block diagram. The block diagram increments a counter by 1 at each time step, using a Unit Delay block.
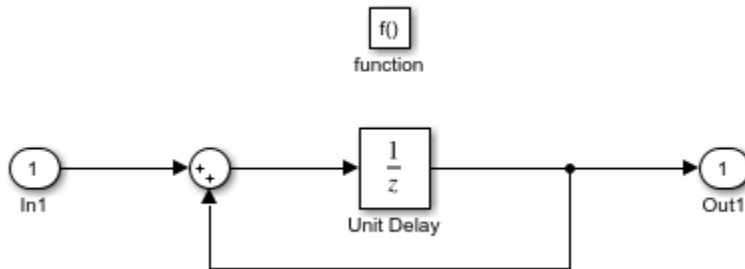
The Block Parameters dialog box for the trigger port contains these settings:

- **Trigger type**: `function-call`.
- **States when enabling**: `reset`. This setting resets the state values for the function-call subsystem to zero when it is enabled.
- **Sample time type**: `triggered`. This setting sets the function-call subsystem to execute only when it is triggered by a calling event while it is enabled.

Setting **Sample time type** to `periodic` enables the **Sample time** field below it, which defaults to 1. These settings force the function-call subsystem to execute for each time step specified in the **Sample time** field while it is enabled. To accomplish this, the state that binds the calling event for the function-call subsystem must send an event for the time step coinciding with the specified sampling rate in the **Sample time** field. States can send events with entry or during actions at the simulation sample rate.

- For fixed-step sampling, the **Sample time** value must be an integer multiple of the fixed-step size.

- For variable-step sampling, the Sample time value has no limitations.

To see how a state controls a bound function-call subsystem, begin simulating the model.

- At time t = 0, the default transition to state A occurs. State A executes its bind and entry actions. The binding action binds event E to state A, enabling the function-call subsystem and resetting its state variables to 0. The entry action triggers the function-call subsystem and executes its block diagram. The block diagram increments a counter by 1 using a Unit Delay block. The Unit Delay block outputs a value of 0 and holds the new value of 1 until the next call to the subsystem.

- At time t = 1, the next update event from the model tests state A for an outgoing transition. The transition to state B does not occur because the temporal operator `after(10,tick)` allows the transition to be taken only after ten update events are received. State A remains active and its during action triggers the function-call subsystem. The Unit Delay block outputs its held value of 1. The subsystem also increments its counter to produce the value of 2, which the Unit Delay block holds until the next triggered execution.

- The next eight update events increment the subsystem output by one at each time step.

- At time t = 10, the transition from state A to state B occurs. Because the binding to state A is no longer active, the function-call subsystem is disabled, and its output drops back to 0.

- At time t = 11, the transition from state B to state A occurs. Again, the binding action enables the function-call subsystem. Subsequent update events increment the subsystem output by one at each time step until the next transition to state B occurs at time t = 21.
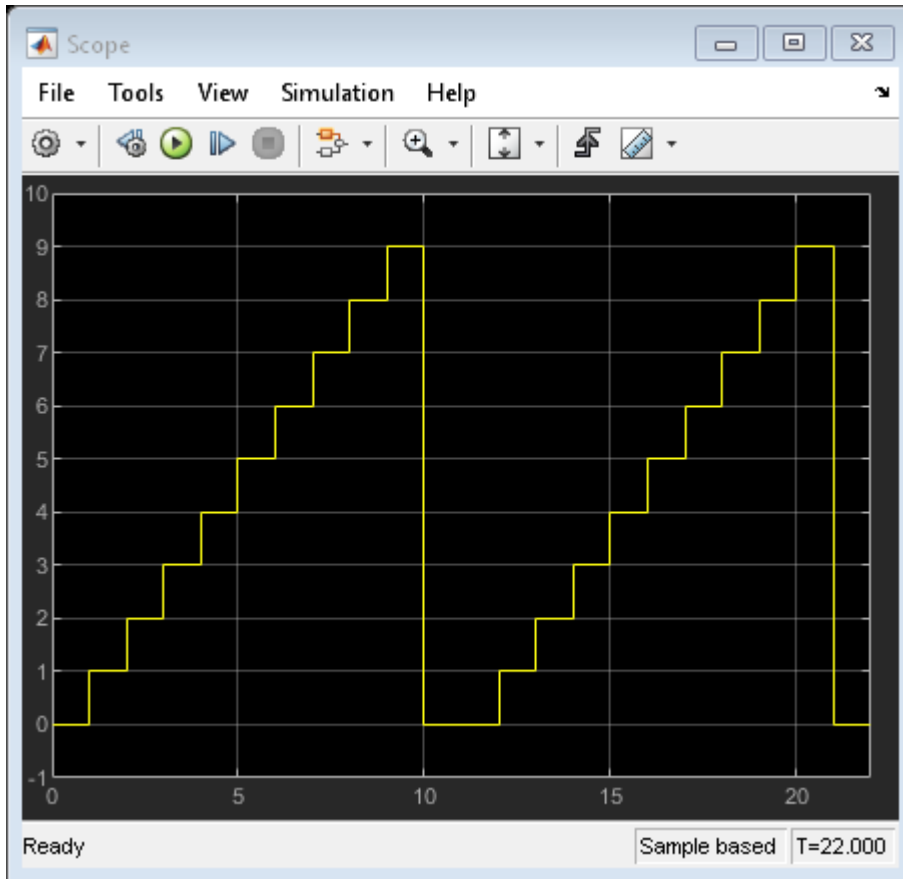
Avoid Muxed Trigger Events with Binding

Binding events gives control of a function-call subsystem to a single state in a chart. This control does not work when you allow other events to trigger the function-call subsystem through a mux. For example, this model defines two function-call events to trigger a function-call subsystem using a Mux block.
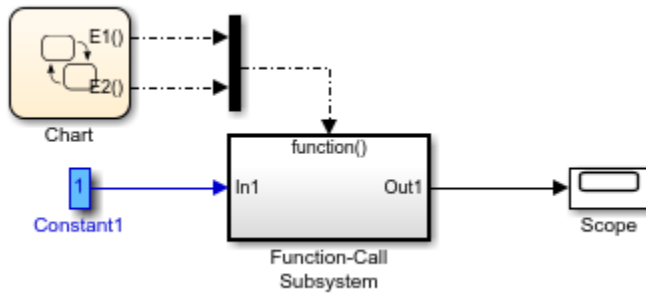
In the chart, E1 binds to state A, but E2 does not. State B sends the triggering event E2 in its entry action.
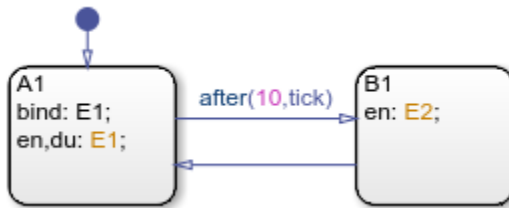
When you simulate this model, the output does not reset when the transition from state A to state B occurs.
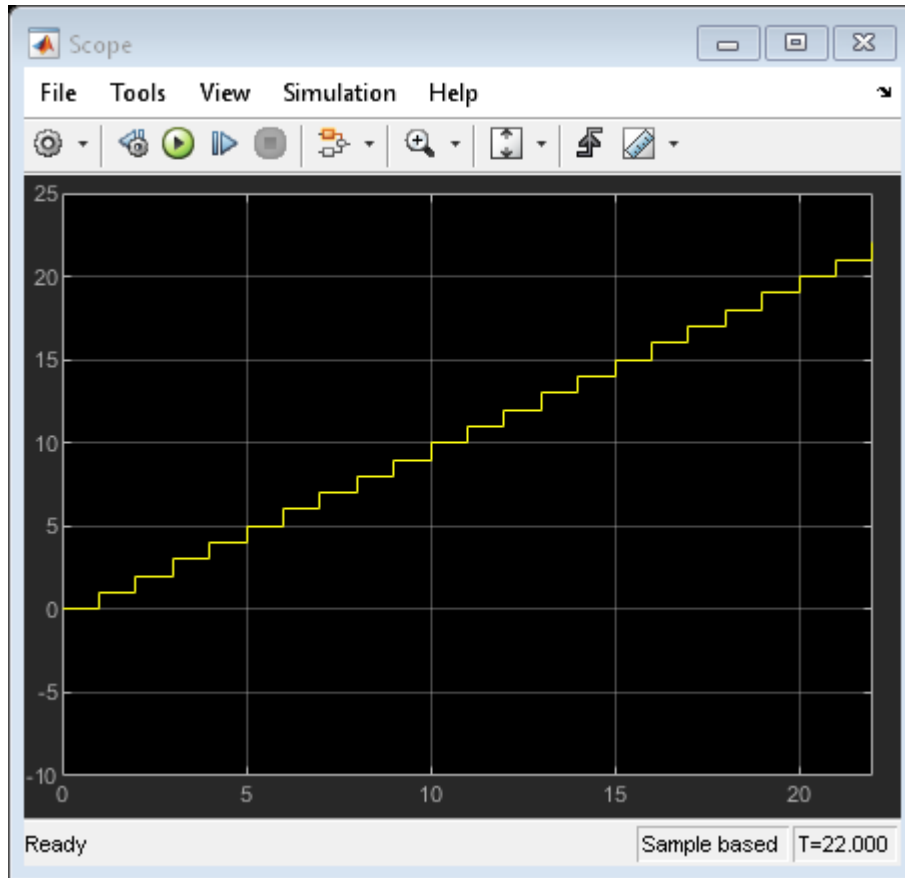
Binding is not recommended when you provide multiple trigger events to a function-call subsystem through a mux. Muxed trigger events can interfere with event binding and cause undefined behavior.

Control Oscillations by Using the duration Operator

The following example focuses on the gear logic of a car as it shifts from first gear to fourth gear.

When modeling the gear changes of this system, it is important to control the oscillation that occur. The model `sf_car` uses parallel state debouncer logic that controls which gear state is active. For more information about how debouncers work in Stateflow, see "Reduce Transient Signals by Using Debouncing Logic" on page 26-2.

You can simplify the debouncer logic by using the `duration` operator. You can see this simplification in the model `sf_car_using_duration`. The `duration` operator evaluates a condition expression and outputs the length of time that the expression has been `true`. When that length of time crosses a known time threshold, the state transitions to a higher or lower gear.

By removing the parallel state logic and using the `duration` operator, you can control oscillations with simpler Stateflow logic. The `duration` operator is supported only in Stateflow charts in a Simulink model.

Control Oscillation with Parallel State Logic

Open the model `sf_car`. While `shift_logic` is highlighted, select **Diagram > Mask > Look Under Mask**.

The Stateflow chart `shift_logic` controls which gear the car is in, given the speed of the car and how much throttle is being applied. Within `shift_logic` there are two parallel states: `gear_state` and `selection_state`. `gear_state` contains four exclusive states for each gear. `selection_state` determines whether the car is downshifting, upshifting, or remaining in its current gear.
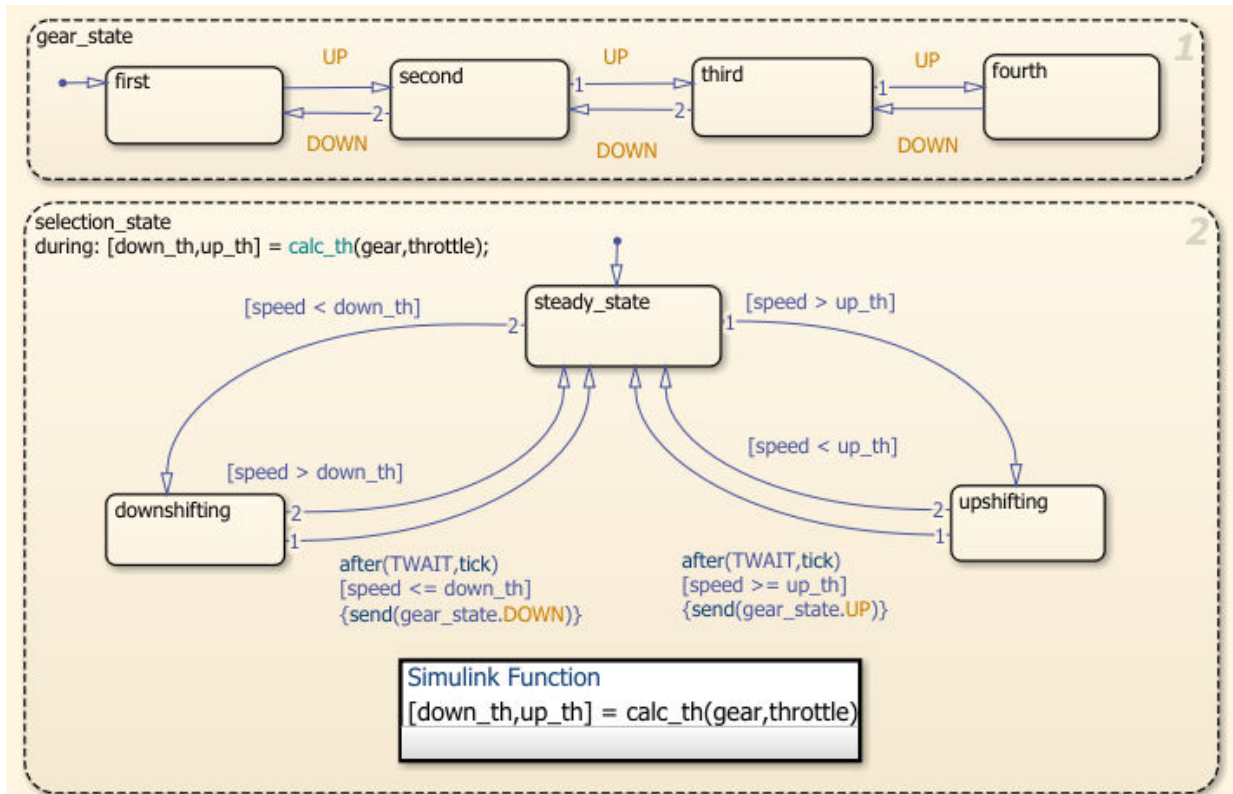
In this Stateflow chart, for the car to move from first gear to second gear, the event UP must be sent from `selection_state` to `gear_state`. The event is sent when the speed crosses the threshold and remains higher than the threshold for the length of time determined by TWAIT. When the event UP is sent, `gear_state` transitions from `first` to `second`.

Control Oscillation with the duration Operator

Open the model `sf_car_using_duration`. While `Gear_Logic` is highlighted, select **Diagram > Mask > Look Under Mask**.

Within `Gear_Logic` there are four exclusive states for each gear. The local variables `up` and `down` guard the transitions between each state.
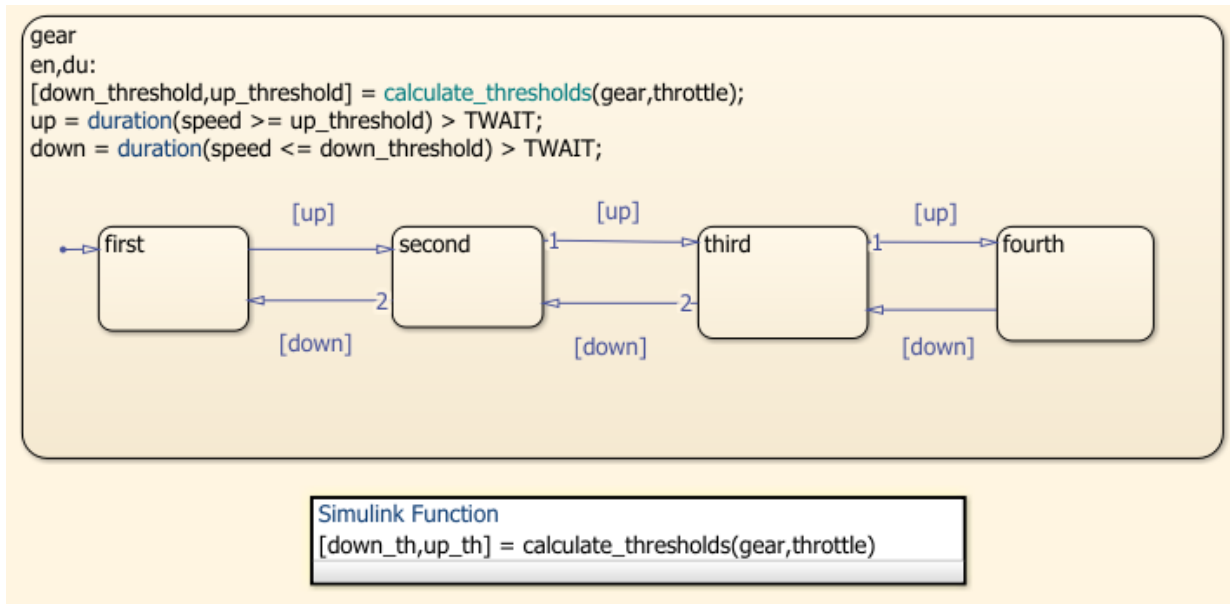
In this Stateflow chart, for the car to move from first gear to second gear, the condition up must be true. The condition up is defined as true if the length of time that the speed is greater than or equal to the threshold is greater than the length of time that is specified by TWAIT. The condition down is defined as true if the length of time that the speed is less than or equal to the threshold is greater than the length of time that is specified by TWAIT. The operator duration keeps track of the length of time that the speed has been above or below the threshold. When the up condition is met, the active state transitions from first to second.

By replacing the parallel state debouncer logic with the duration operator, you can create a simpler Stateflow chart to model the gear shifting.

See Also
duration

Related Examples
- "Reduce Transient Signals by Using Debouncing Logic" on page 26-2

•   "Control Chart Execution by Using Temporal Logic" on page 12-49

MATLAB Syntax Support for States and Transitions

Modify the Action Language for a Chart

Stateflow charts in Simulink models have an action language property that defines the syntax for state and transition actions. An icon in the lower-left corner of the chart canvas indicates the action language for the chart.

- MATLAB as the action language.

- C as the action language.

You can change the action language of a chart in the **Action Language** box of the Chart properties dialog box. For more information, see "Differences Between MATLAB and C as Action Language Syntax" on page 13-6.

Change the Default Action Language

MATLAB is the default action language syntax for new Stateflow charts. To create a chart that uses C as the action language, enter:

sfnew -c

To change the default action language of new charts, use these commands.

Command	Result
sfpref('ActionLanguage','MATLAB')	All new charts created have MATLAB as the action language, unless otherwise specified in sfnew.
sfpref('ActionLanguage','C')	All new charts created have C as the action language, unless otherwise specified in sfnew.

For more information, see sfnew.

Auto Correction When Using MATLAB as the Action Language

Stateflow charts that use MATLAB as the action language automatically correct common C constructs to MATLAB syntax:

- Increment and decrement operations such as `a++` and `a--`. For example, `a++` is changed to `a = a+1`.

- Assignment operations such as `a += b`, `a -= b`, `a *= b`, and `a /= b`. For example, `a += b` is changed to `a = a+b`.

- Evaluation operations such as `a != b` and `!a`. For example, `a != b` is changed to `a ~= b`.

- Comment markers `//` and `/* */` are changed to `%`.

C to MATLAB Syntax Conversion

For nonempty charts, after you change the action language property from C to MATLAB, a notification appears at the top of the chart. The notification provides the option to convert some of the C syntax to MATLAB syntax. In the notification, click the link to have Stateflow convert syntax in the chart. C syntax constructs that are converted include:

- Zero-based indexing.

- Increment and decrement operations such as `a++` and `a--`. For example, `a++` is changed to `a = a+1`.

- Assignment operations such as `a += b`, `a -= b`, `a *= b`, and `a /= b`. For example, `a += b` is changed to `a = a+b`.

- Binary operations such as `a %% b`, `a >> b`, and `a << b`. For example, `a %% b` is changed to `rem(a,b)`.

- Bitwise operations such as `a ^ b`, `a & b`, and `a | b`. For example, if the chart property **Enable C-bit operations** is selected, then `a ^ b` is changed to `bitxor(a,b)`.

- C style comment markers. For example, `//` and `/* */` are changed to `%`.

If the chart contains C constructs that cannot be converted to MATLAB, Stateflow shows a message in a dialog box. Click on the warnings link to display the warnings in the Diagnostic Viewer. Choose whether or not to continue with the conversion of supported syntax. C constructs not converted to MATLAB include:

- Explicit type casts with `cast` and `type`

- Operators such as &, * and `:=`

- Custom data

- Access to workspace variables using `ml` operator

- Functions not supported in code generation
- Hexadecimal and single precision notation
- Context-sensitive constants

Rules for Using MATLAB as the Action Language

Use unique names for data in a chart

Using the same name for data at different levels of the chart hierarchy causes a compile-time error.

Use unique names for functions in a chart

Using the same name for functions at different levels of the chart hierarchy causes a compile-time error.

Include a type prefix for identifiers of enumerated values

The identifier `TrafficColors.Red` is valid, but `Red` is not.

Use the MATLAB format for comments

Use % to specify comments in states and transitions for consistency with MATLAB. For example, the following comment is valid:

```
% This is a valid comment in the style of MATLAB
```

C style comments, such as `//` and `/* */`, are auto-corrected to use %.

Use one-based indexing for vectors and matrices

One-based indexing is consistent with MATLAB syntax.

Use parentheses instead of brackets to index into vectors and matrices

This statement is valid:

```
a(2,5) = 0;
```

This statement is not valid:

```
a[2][5] = 0;
```

Do not use control flow logic in condition actions and transition actions

If you try to use control flow logic in condition actions or transition actions, you get an error. Use of an `if`, `switch`, `for`, or `while` statement does not work.

Do not use transition actions in graphical functions

Transition labels in graphical functions do not support transition actions.

Enclose transition actions with braces

The following transition label contains a valid transition action:

```
E [x > 0] / {x = x+1;}
```

The following transition label:

```
E [x > 0] / x = x+1;
```

is incorrect, but is auto-corrected to the valid syntax.

Do not declare global or persistent variables in state actions

The keywords `global` and `persistent` are not supported in state actions.

To generate code from your model, use MATLAB language features supported for code generation

Otherwise, use `coder.extrinsic` to call unsupported functions, which gives the functionality that you want for simulation, but not in the generated code. For a list of supported features and functions, see "Language, Function, and Object Support" (Simulink).

Assign an initial value to local and output data

When using MATLAB as the action language, data read without an initial value causes an error.

Differences Between MATLAB and C as Action Language Syntax

Stateflow charts in Simulink models have an action language property that defines the syntax for state and transition actions. An icon in the lower-left corner of the chart canvas indicates the action language for the chart.

- MATLAB as the action language.

- C as the action language.

MATLAB is the default action language syntax for new Stateflow charts. To create a chart that uses C as the action language, enter:

```
sfnew -c
```

Compare Functionality of Action Languages

This table lists the most significant differences in functionality between the two action languages.

Functionality	MATLAB as the Action Language	C as the Action Language
Vector and matrix indexing	One-based indexing delimited by parentheses and commas. For example, A(4,5). See "Supported Operations for Vectors and Matrices" on page 17-11.	Zero-based indexing delimited by square brackets. For example, A[3][4]. See "Supported Operations for Vectors and Matrices" on page 17-11.

Functionality	MATLAB as the Action Language	C as the Action Language
C constructs: • Increment and decrement operations `a++` and `a--` • Assignment operations `a += b`, `a -= b`, `a *= b`, and `a /= b` • Evaluation operations `a != b` and `!a` • Binary operations `a %% b`, `a >> b`, `a << b`, `a & b`, and `a	b` • Comment markers `//` and `/* */`	Auto-correction to MATLAB syntax. For example, `a++` is corrected to `a = a+1`. See "Auto Correction When Using MATLAB as the Action Language" on page 13-2.	Supported. See "Supported Operations for Chart Data" on page 12-14.
Conditional and loop control statements in state actions	Supported. For example, you can use `if`, `for`, and `while` statements in state actions. See "Loops and Conditional Statements" (MATLAB).	Not supported. For conditional and loop patterns, use graphical functions instead. See "Reuse Logic Patterns by Defining Graphical Functions" on page 8-16.
Format of transition actions	Auto-correction encloses transition actions with braces {}. See "Transition Actions" on page 12-9.	Not required to enclose transition actions with braces {}. See "Transition Actions" on page 12-9.
Ordering of parallel states	Explicit ordering only. See "Execution Order for Parallel States" on page 3-86.	Explicit or implicit ordering. See "Execution Order for Parallel States" on page 3-86.

Functionality	MATLAB as the Action Language	C as the Action Language
Fixed-point constructs: • Special assignment operator `:=` • Context-sensitive constants such as `4.3C`	Not supported.	Supported. See "Override Fixed-Point Promotion in C Charts" on page 22-25 and "Fixed-Point Context-Sensitive Constants" on page 22-5.
Complex data	Use complex number notation `a + bi` or the `complex` operator. See "Supported Operations for Complex Data" on page 23-4.	Use the `complex` operator. Complex number notation is not supported. See "Supported Operations for Complex Data" on page 23-4.
Data type propagation	Follows MATLAB typing rules. For example, adding data of type `double` to data of type `int32` results in data of type `int32`.	Follows C typing rules. For example, adding data of type `double` to data of type `int32` results in data of type `double`.
Explicit type cast operations	Use one of these casting forms: • MATLAB type conversion function. For example, `single(x)`. • `cast` function with a type keyword. For example, `cast(x,'int8')`. • `cast` function with the `'like'` keyword. For example, `cast(x,'like',z)`. The `type` operator is not supported. See "Type Cast Operations" on page 12-19.	Use one of these casting forms: • MATLAB type conversion function. For example, `uint16(x)`. • `cast` function with the `type` operator. For example, `cast(x,type(z))`. Type keywords for the `cast` function are not supported. See "Type Cast Operations" on page 12-19.

Functionality	MATLAB as the Action Language	C as the Action Language
Scalar expansion	Not supported.	Supported. See "Convert Scalars to Nonscalars by Using Scalar Expansion" on page 17-9.
String data	Not supported.	Supported. See "Manage Textual Information by Using Strings" on page 20-2.
Specification of data properties: • First index • Save final value to base workspace • Units	Not supported.	Supported. For more information, see: • "First Index" on page 9-8 • "Save Final Value to Base Workspace" on page 9-19 • "Units" on page 9-19
Scope of data in graphical, truth table, and MATLAB functions	`Constant`, `Parameter`, `Input`, `Output`	`Local`, `Constant`, `Parameter`, `Input`, `Output`, `Temporary`
Dot notation for specifying states, local data, message, and local events inside MATLAB functions	Supported. See "Identify Data by Using Dot Notation" on page 9-50.	Not supported.

Functionality	MATLAB as the Action Language	C as the Action Language
Custom code functions and variables	Behavior depends on the **Import Custom Code** configuration parameter. • When **Import Custom Code** is on, both custom code functions and variables are supported in states and transitions (default). • When **Import Custom Code** is off, only custom code functions are supported. Use the `coder.ceval` function. See "Custom Code" and "Import custom code" (Simulink).	Custom code functions and variables are supported in states and transitions.
Row-major array layout in generated code	Not supported.	Supported. See "Row-Major Array Layout" on page 31-6.
Multidimensional array layout in generated code	Not supported.	Supported for local data. See "Multidimensional Array Layout" on page 31-7.
Structure parameters	Tunable and nontunable parameters are supported.	Only tunable parameters are supported.
Use of global `fimath` object	Supported.	Not supported.

See Also

`sfnew`

More About

- "Modify the Action Language for a Chart" on page 13-2
- "Supported Operations for Chart Data" on page 12-14
- "Supported Operations for Vectors and Matrices" on page 17-11
- "Supported Operations for Complex Data" on page 23-4
- "Supported Operations for Fixed-Point Data" on page 22-21
- "Execution Order for Parallel States" on page 3-86

Model an Assembly Line Feeder

Typical Approaches to Chart Programming

There are two general approaches to programming a Stateflow chart in a Simulink model:

- Identify the operating modes of your system.
- Identify the system interface, such as events to which your system reacts.

This tutorial uses the first approach— that is, start by identifying the operating modes of the system to program the chart.

Design Requirements

This example shows how to build a Stateflow chart using MATLAB as the action language. The model represents a machine on an assembly line that feeds raw material to other parts of the line. This feeder behaves as follows:

- At system initialization, check that the three sensor values are normal.

  A positive value means the sensor is working correctly. A zero means that the sensor is not working.
- If all sensor values are normal, transition from "system initialization" to "on".
- If the feeder does not leave initialization mode after 5 seconds, force the feeder into the failure state.
- After the system turns on, it starts counting the number of parts fed.
- At each time step, if any sensor reading is 2 or greater, the part has moved to the next station.
- If the alarm signal sounds, force the system into the failure state.

  An alarm signal can occur when an operator opens one of the safety doors on the feeder or a downstream problem occurs on the assembly line, which causes all upstream feeders to stop.
- If the all-clear signal sounds, resume normal operation and reset the number of parts fed to zero.
- The feeder LED changes color to match the system operating mode— orange for "system initialization", green for "on", and red for "failure state".

Identify System Attributes

Based on the description of feeder behavior, you can identify the key system attributes.

Attribute	Characteristic
Operating modes	• **System initialization**, to perform system checks before turning on the machine • **On**, for normal operation • **System failure**, for a recoverable machine failure flagged by an alarm
Transitions	• **System initialization** to **On** • **System initialization** to **Failure state** • **On** to **Failure state** • **Failure state** to **System initialization**
Parallel Modes	No operating modes run in parallel. Only one mode can be active at any time.
Default Mode	**System initialization**
Inputs	• Three sensor readings to detect if a part has moved to a downstream assembly station • An alarm signal that can take one of two values: 1 for on and 0 for off
Outputs	• Number of parts that have been detected as fed to a downstream assembly station • Color of the LED on the feeder

Build the Model Yourself or Use the Supplied Model

In this exercise, you add a Stateflow chart to a Simulink model that contains sensor and alarm input signals to the feeder.

To implement the model yourself, follow these exercises. Otherwise, you can open the completed model.

Add a Stateflow Chart to the Feeder Model

1 Open the partially built model.
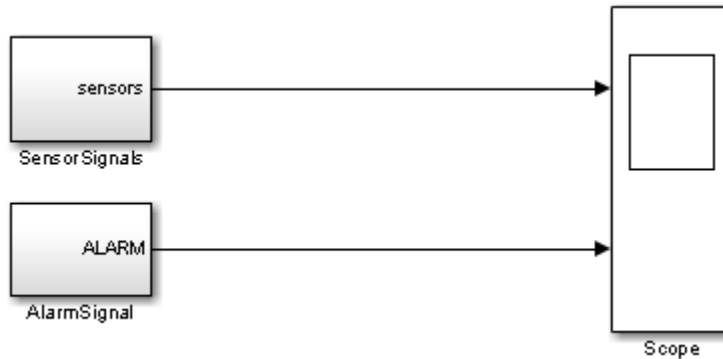
2 Double-click the `SensorSignals` block to see the three sensor signals represented by pulse generator blocks.

The `sensors` signal indicates when the assembly part is ready to move to the next station.

3 Double-click the `AlarmSignal` block to see the step blocks that represent the alarm signal.

When the ALARM signal is active, the machine turns off.

4 Run the model to see the output of the sensor and alarm signals in the Scope block.
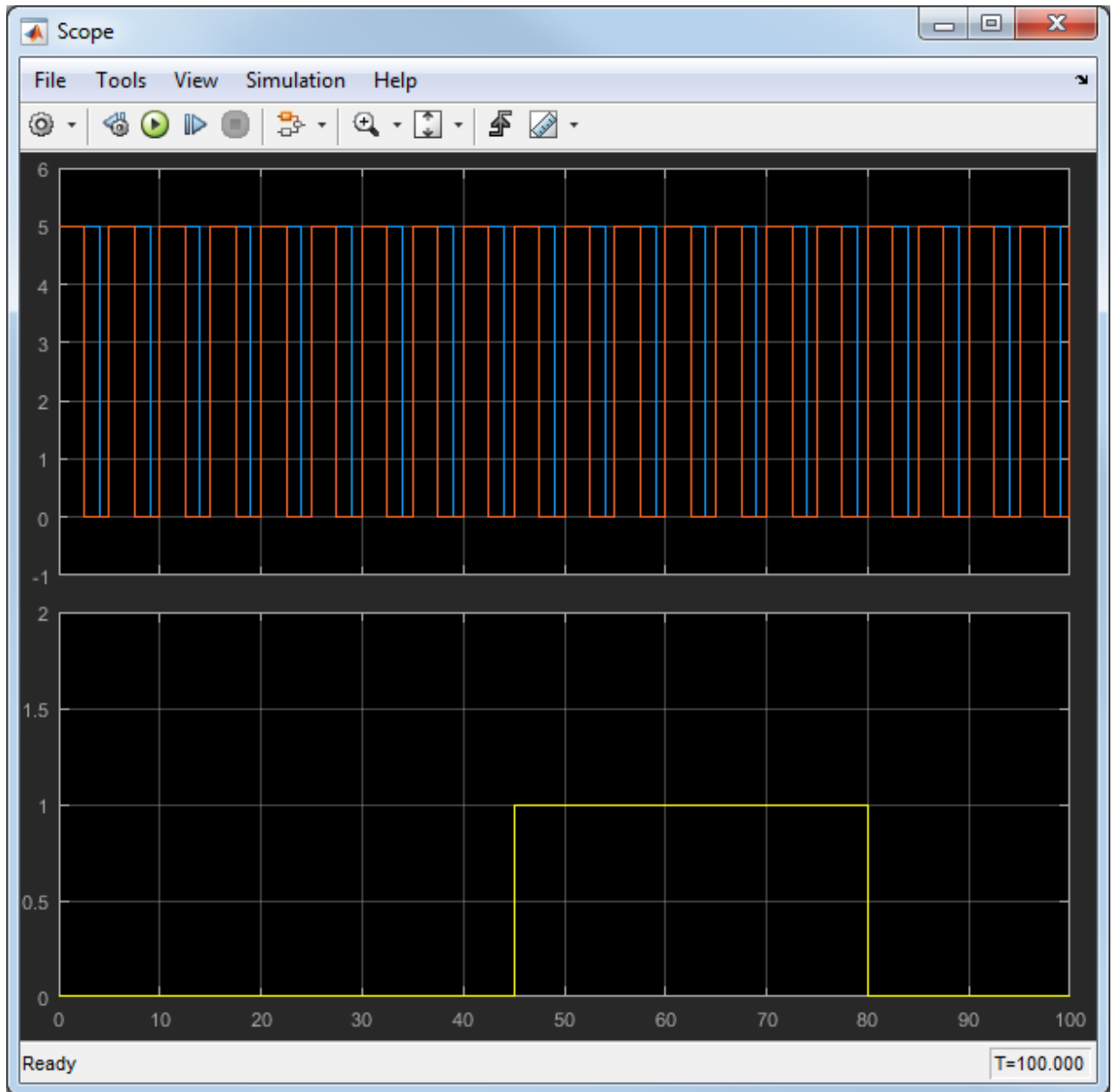
The upper axis shows the sensor signals. Only two sensor signals appear because two of the sensors have the same signal. The lower axis shows the alarm signal which turns the feeder off between the simulation time of 45 to 80 seconds.

5   Open the Stateflow Library by executing `sflib` at the MATLAB command prompt.

6   Select `Chart` and drag it into your model.

Tip To create a new model with an empty Stateflow chart which uses MATLAB as the action language, use the command,`sfnew`.

---

7   Delete the connections from the SensorSignals subsystem to the scope and from the AlarmSignal subsystem to the scope.

8   Rename the label `Chart` located under the Stateflow chart to `Feeder`. The model should now look like this:

Add States to Represent Operating Modes

Based on the system attributes previously described, there are three operating modes:

- System initialization
- On
- Failure state

To add states for modeling the behavior of these operating modes:

1   Double-click the Feeder Chart to begin adding states.

Note The MATLAB icon in the lower left corner of the chart indicates that you are using a Stateflow chart with MATLAB syntax.

---

2 Click the State Tool icon to bring a state into the chart.

3 Click the upper left corner of the state and type the name, `InitializeSystem`.

4 Repeat steps 2 and 3 to add two more states named `On` and `FailState`.

Implement State Actions

Decide the Type of State Action

States perform actions at different phases of their execution cycle from the time they become active to the time they become inactive. Three basic state actions are:

Type of Action	When Executed	How Often Executed While State Is Active
Entry	When the state is entered (becomes active)	Once
During	While the state is active and no valid transition to another state is available	At every time step
Exit	Before a transition is taken to another state	Once

For example, you can use `entry` actions to initialize data, `during` actions to update data, and `exit` actions to configure data for the next transition. For more information about other types of state actions, see "Syntax for States and Transitions".)

1 Press return after the `InitializeSystem` state name and add this text to define the state entry action:

```
entry:
Light = ORANGE;
```

An orange LED indicates entry into the `InitializeSystem` state.

Syntax for an entry action

The syntax for `entry` actions is:

`entry:` *one or more actions*;

2   Add the following code after the `FailState` state name to define the entry action:

```
entry:
Light = RED;
```

A red LED indicates entry in the `FailState`.

3   Add the following code after the `On` state name to define the entry action:

```
entry:
Light = GREEN;
partsFed = 0;
```

A green LED indicates entry in the `On` state. The number of parts fed is initialized to 0 each time we enter the `On` state

4   Add the following code to the `On` state after the entry action to check if there is a strong sensor signal and increment the parts fed to the next station:

```
during:
if(any(sensors >= 2))
    partsFed = partsFed + 1;
end
```

The `On` state checks the sensor signal to determine if a part is ready to be fed to the next assembly station. If the sensor signal is strong (the number of sensors that are on is greater than or equal to 2), then the chart counts the part as having moved on to the next station.

Syntax for during actions

The syntax for `during` actions is:

`during:` *one or more actions*;

The chart should now look like this figure.

```
InitializeSystem
entry:
Light = ORANGE;
```

```
FailState
entry:
Light = RED;
```

```
On
entry:
Light = GREEN;
partsFed = 0;

during:
if (any(sensors >= 2))
    partsFed = partsFed + 1;
end
```
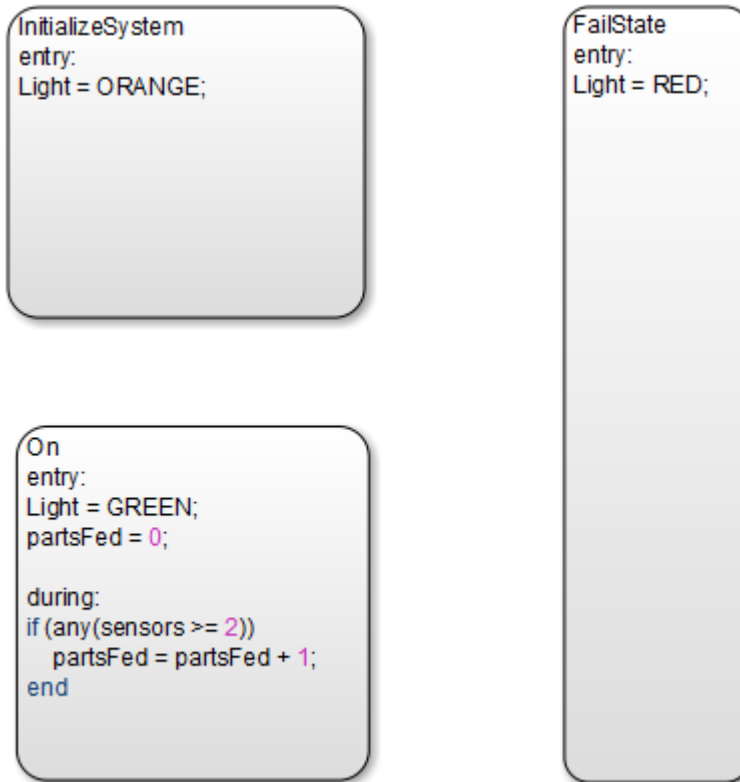
Specify Transition Conditions

Transition conditions specify when to move from one operating mode to another. When the condition is true, the chart takes the transition to the next state, otherwise, the current state remains active. For more information, see "Transitions" on page 2-18.

Based on the description of feeder behavior, specify the rules for transitions between states:

1 Connect a default transition to the InitializeSystem state to indicate the chart entry point.

"Default Transitions" on page 2-33 specify where to begin the simulation.

2 Draw a transition from the `InitializeSystem` state to the `On` state:

   a Move the mouse over the lower edge of the `InitializeSystem` state until the pointer shape changes to crosshairs.

   b Click and drag the mouse to the upper edge of the `On` state. You then see a transition from the `InitializeSystem` state to the `On` state.

   c Double-click the transition to add this condition:

```
[all(sensors>0)]
```

This transition condition verifies if all of the sensors have values greater than zero.

3 Repeat these steps to create these remaining transition conditions.

Transition	Condition
On to FailState	[Alarm == 1]
FailState to InitializeSystem	[Alarm == 0]

4 Draw another transition from `InitializeSystem` to `FailState`. On this transition, type the following to create the transition event:

```
after(5,sec)
```

If the sensors have not turned on after 5 seconds, this syntax specifies a transition from `InitializeSystem` to `FailState`.

Note The syntax on this transition is an event rather than a transition condition. For details, refer to"Control Chart Execution by Using Temporal Logic" on page 12-49.

The chart now looks like this figure.
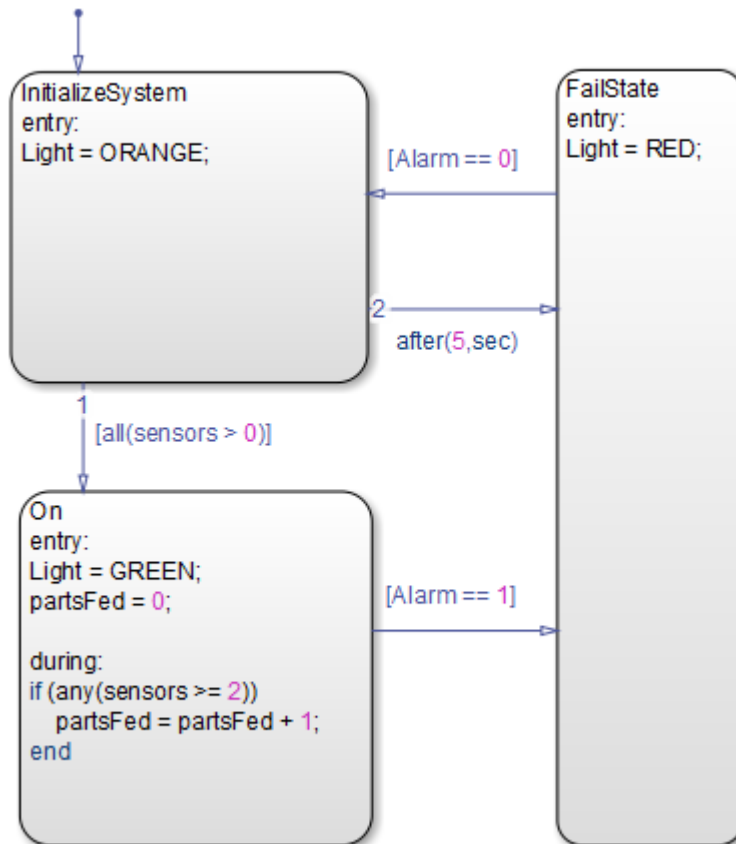
Note The outgoing transitions from `InitializeSystem` have a small label 1 and 2 to indicate the order in which transition segments are evaluated. If the numbers from the figure do not match your model, right click the transition and then change it by clicking on `Execution Order`. See "Transition Evaluation Order" on page 3-65 for details.

Define Data for Your System

Verify the Chart Data Properties

Start the simulation of your model. Errors about unresolved symbols appear, along with the Symbol Wizard.
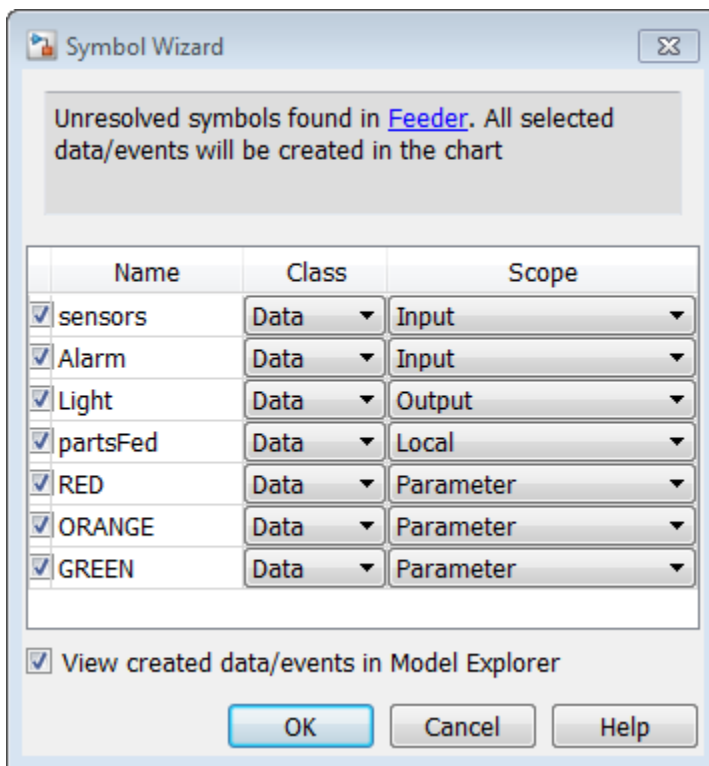
The Symbol Wizard does not automatically add any data to your chart. It identifies the unresolved data and infers the class and scope of that data using the inference rules of MATLAB expressions in Stateflow actions. In the chart:

- Data that is read from but not written to is inferred as *input* data. However, if the name of the data is in all uppercase letters, the Symbol Wizard infers the data as a *parameter*
- Data that is written to but not read from is inferred as *output* data.

- Data that is read from and written to is inferred as *local* data.

The Symbol Wizard infers the scope of the input data in your chart. However, you must fix the data scope for the `partsFed` Output. Follow these steps:

1 For the **partsFed** data: in the **Data Scope** column, select **Output** from the list

The Symbol Wizard now looks like this figure.
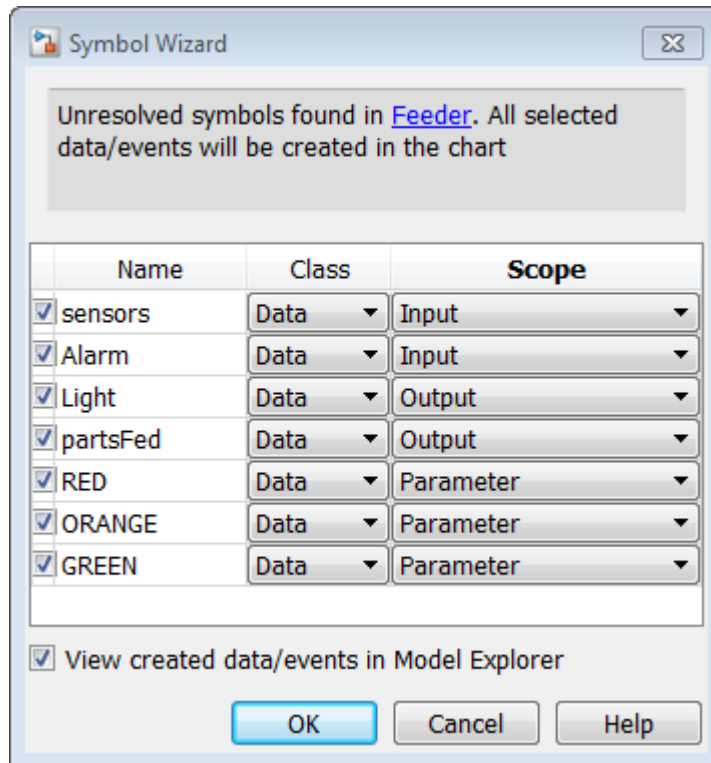
2 To add the data that the Symbol Wizard suggests, click **OK**.

3 Add initial values for the parameters. At the MATLAB command prompt, enter:

```
RED = 0;
```

4 Similarly, at the MATLAB command prompt, add the following initial values for the remaining parameters:

Parameter	Value
RED	**0**
ORANGE	**1**
GREEN	**2**

5 Return to the model and connect the inputs and outputs to their respective ports.
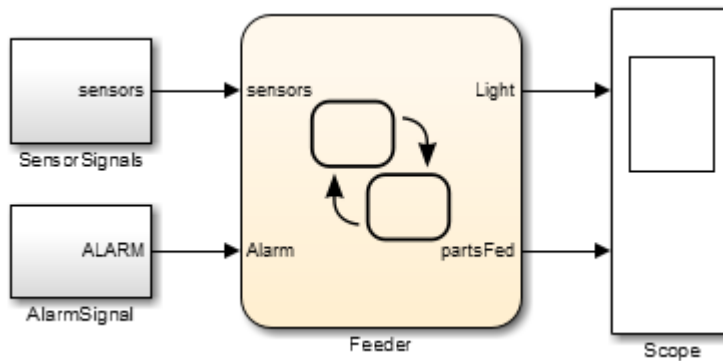
Verify the System Representation

1 Start the simulation.

Double-click the Scope block to verify that the model captures the expected feeder behavior.
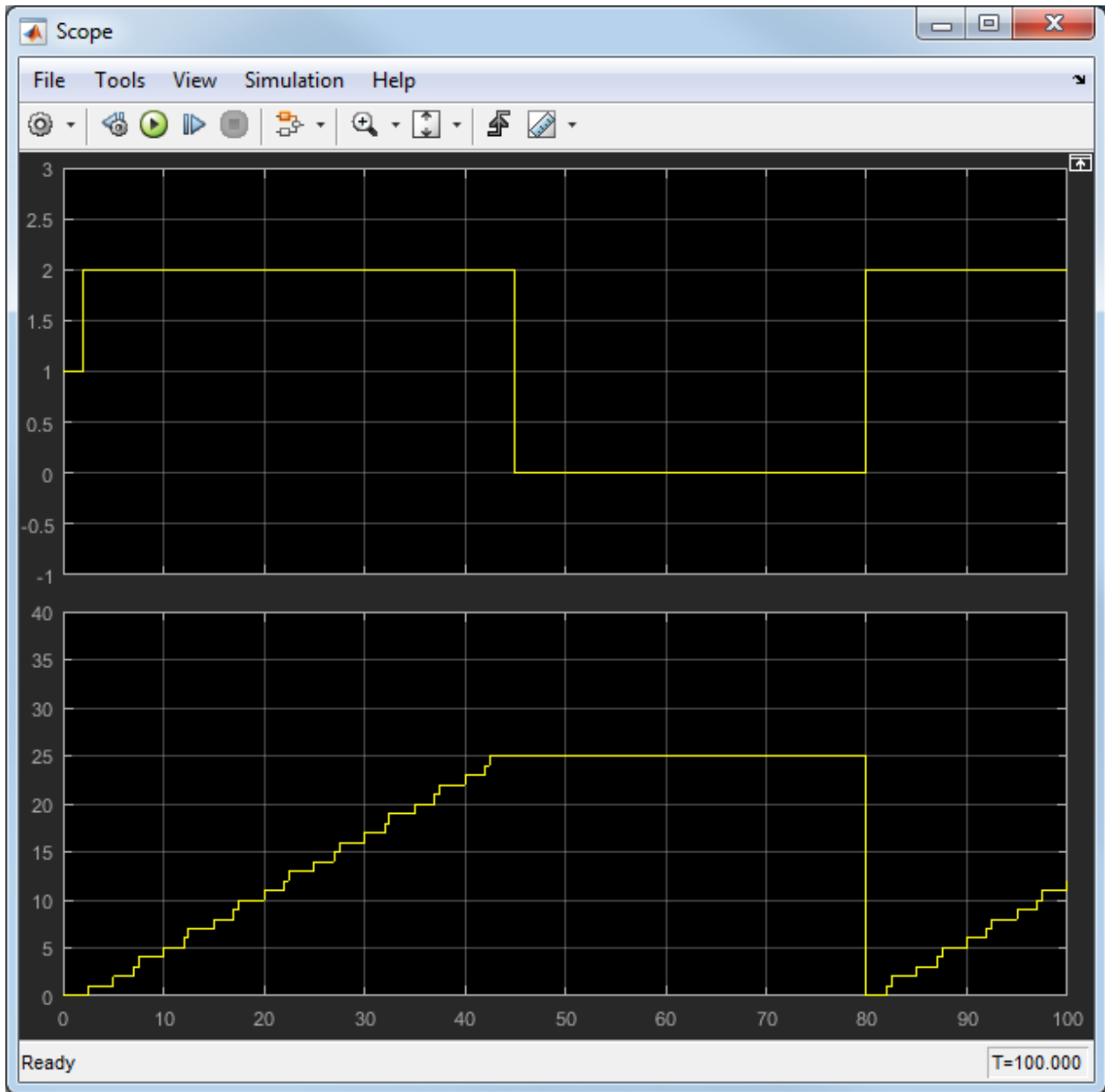
The upper axis shows the LED signal which varies between orange (1), green (2), and red (0) to indicate the current operating mode. The lower axis shows the number of parts fed to the next assembly station, which increases incrementally until the alarm signal turns the machine off and then resets.

Alternative Approach: Event-Based Chart

Another approach to programming the chart is to start by identifying parts of the system interface, such as events to which your system reacts.

In the previous example, when you use input data to represent an event, the chart wakes up periodically and verifies whether the conditions on transitions are valid. In this case, if ALARM == 1, then the transition to the failure state happens at the next time step. However, creating a Stateflow chart which reacts to input events allows you to react to the alarm signal when the event is triggered.

For details on when to use an event-based chart, see "Synchronize Model Components by Broadcasting Events" on page 10-2.

Identify System Attributes for Event-Driven Systems

In the event-based approach, the system attributes to consider first are the events, inputs, and outputs.

In the following table, consider the characteristics of the event-driven Feeder Model that are different from the system based on transition conditions.

Attributes	Characteristics
Events	Two asynchronous events: an alarm signal and an all-clear signal
Inputs	Three sensor readings to detect if a part has moved to a downstream assembly station

Feeder Chart Activated by Input Events

In this example, the feeder model reacts to input events using a triggered chart.
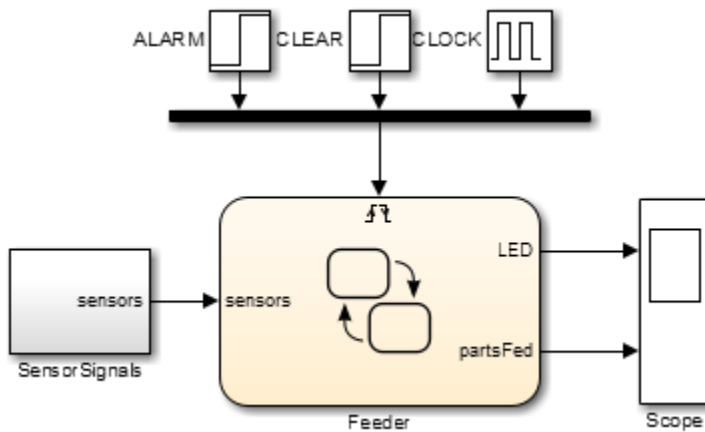
The chart now has only one input port on the left and an event triggered input on the top. For more information on how to create a Stateflow chart activated by events, see "Activate a Stateflow Chart by Sending Input Events" on page 10-9

When the ALARM signal triggers the chart, the chart responds to the trigger in that time step. If the current state is On when the alarm is triggered, then the current state transitions to FailState.
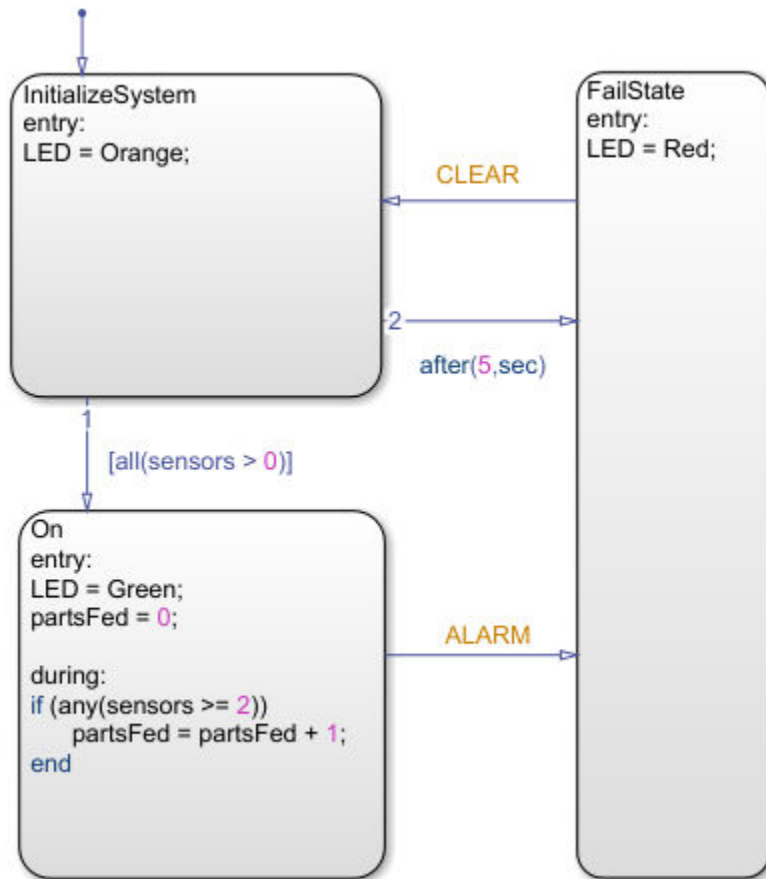
The scope output for the Event-triggered chart is in the following figure.
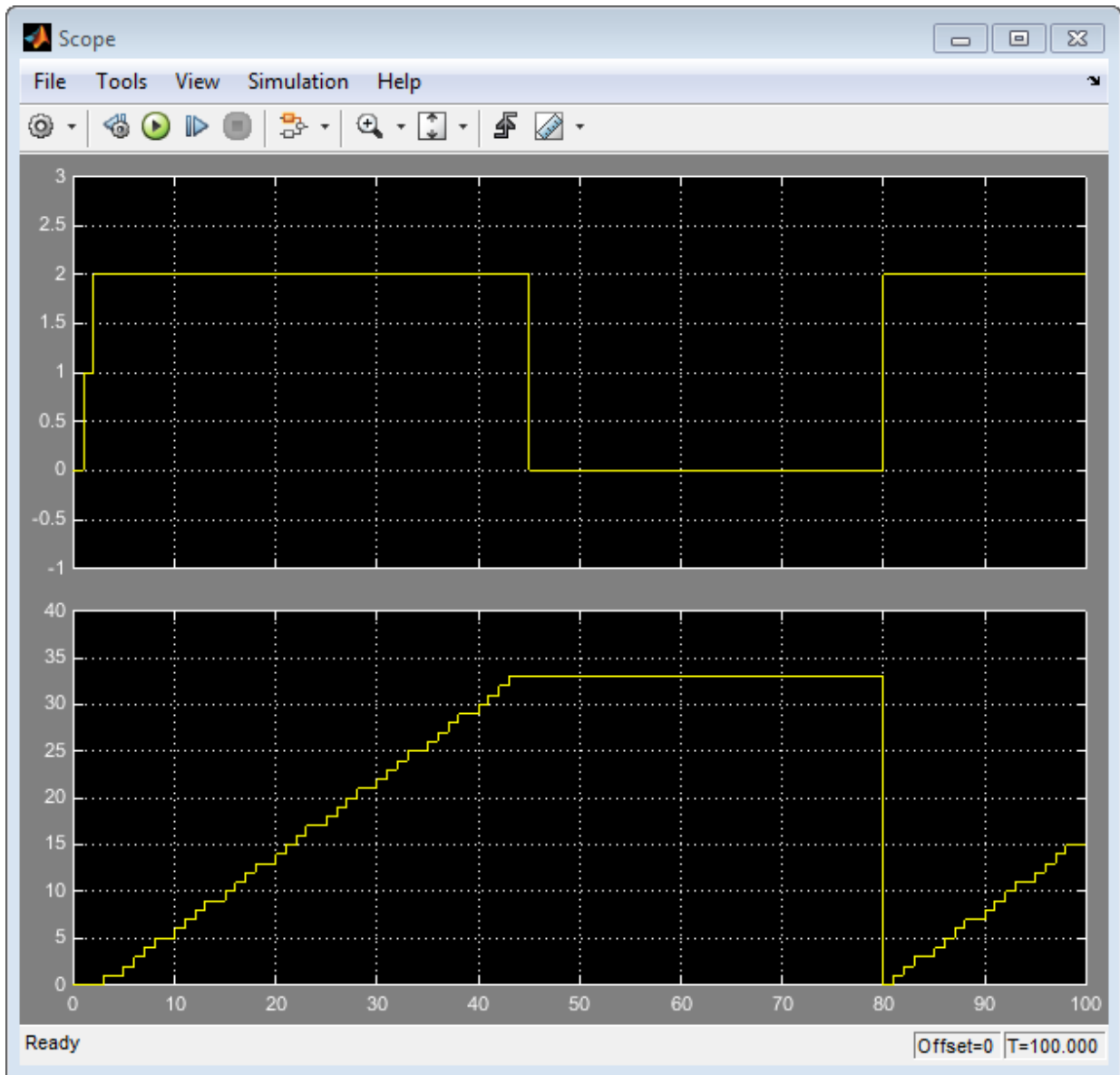
The upper axis shows the LED signal which varies between red (0), orange (1), and green (2) to indicate the current operating mode. The lower axis shows the number of parts fed

13-29

to the next assembly station, which increases incrementally until the alarm signal turns the machine off and then resets. However, the event-based simulation feeds more parts to the next assembly station due to clock and solver differences.

Use C Chart to Model Event-Driven System

Before you start building a chart, you identify system attributes by answering these questions:

1 What are your interfaces?

    a   What are the event triggers to which your system reacts?
    b   What are the inputs to your system?
    c   What are the outputs from your system?

2 Does your system have any operating modes?

    a   If the answer is yes, what are the operating modes?
    b   Between which modes can you transition? Are there any operating modes that can run in parallel?

    If your system has no operating modes, the system is *stateless*. If your system has operating modes, the system is *modal*.

After identifying your system attributes, you can follow a basic work flow for building Stateflow charts to model event-driven systems:

1 Define the interface to Simulink.
2 Define the states for modeling each mode of operation.
3 Define state actions and variables.
4 Define the transitions between states.
5 Decide how to trigger the chart.
6 Simulate the chart.
7 Debug the chart.

14

Tabular Expression of Modal Logic

State Transition Tables in Stateflow

A state transition table is an alternative way of expressing sequential modal logic. Instead of drawing states and transitions graphically in a Stateflow chart, use state transition tables to express the modal logic in tabular format. State transition tables are supported only as blocks in a Simulink model.

Benefits of using state transition tables include:

- Ease of modeling train-like state machines, where the modal logic involves transitions from one state to its neighbor
- Concise, compact format for a state machine
- Reduced maintenance of graphical objects

  When you add or remove states from a chart, you have to rearrange states, transitions, and junctions. When you add or remove states from a state transition table, you do not have to rearrange any graphical objects.

State transition tables support using MATLAB and C as the action language. For more information about the differences between these action languages, see "Differences Between MATLAB and C as Action Language Syntax" on page 13-6.

The following state transition table contains the modal logic for maintaining the temperature of a boiler between two set points:

| STATES | TRANSITIONS | |
	IF	ELSE-IF(2)
Normal	[ALARM]	
	Alarm ▼	▼

Off
entry:
boiler_cmd = 0;
doneWarmup = false;
	[temp <= reference_low]	

	Warmup ▼	▼
Warmup entry: boiler_cmd = 2;	[doneWarmup]	[after(10, sec)]
		{doneWarmup = true;}
	On ▼	On ▼
On entry: boiler_cmd = 1;	[temp >= reference_high]	
	Off ▼	▼
Alarm entry: boiler_cmd = 0;	[CLEAR]	
	Normal ▼	▼

This Stateflow chart represents the same modal logic:

Requirements:
1. Boiler maintains temperature between two setpoints.
2. The first time that the boiler turns on, it must warm up for 10 seconds.
3. The alarm needs to shut the boiler off.
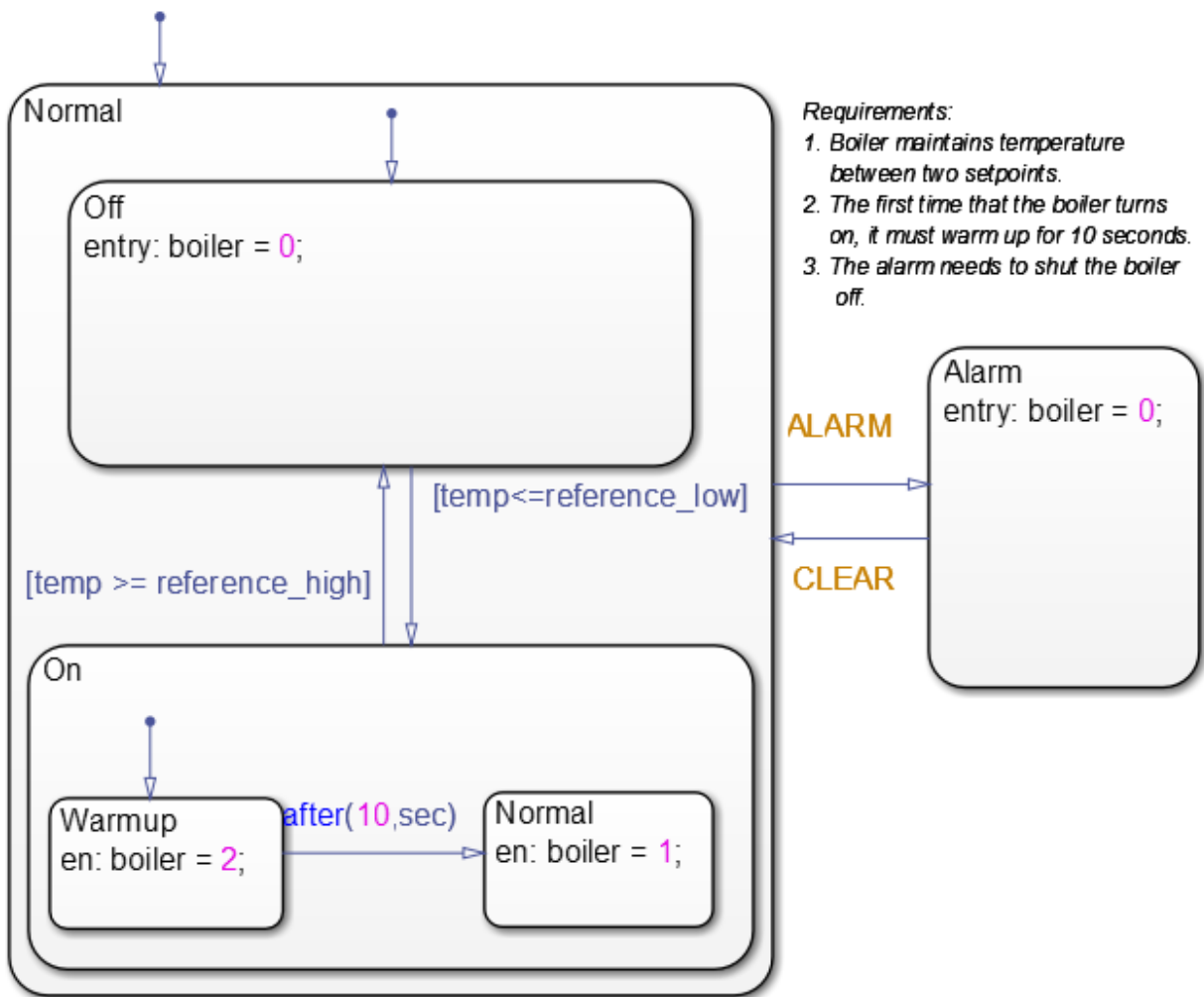
Rules for Using State Transition Tables

- If you specify an action in a transition cell, it must be a condition action.
- State transition tables must have at least one state row and one transition column.

Differences Between State Transition Tables and Charts

State transition tables support a subset of the most commonly used elements in Stateflow charts. Elements that state transition tables do not support include:

- Supertransitions
- Parallel (AND) decomposition
- Local events
- Flow charts
- Use of chart-level functions (graphical, truth table, MATLAB, and Simulink)

Anatomy of a State Transition Table

A state transition table contains the following components:

States column

Transitions column

Condition cell

Destination state cell

Condition action cell
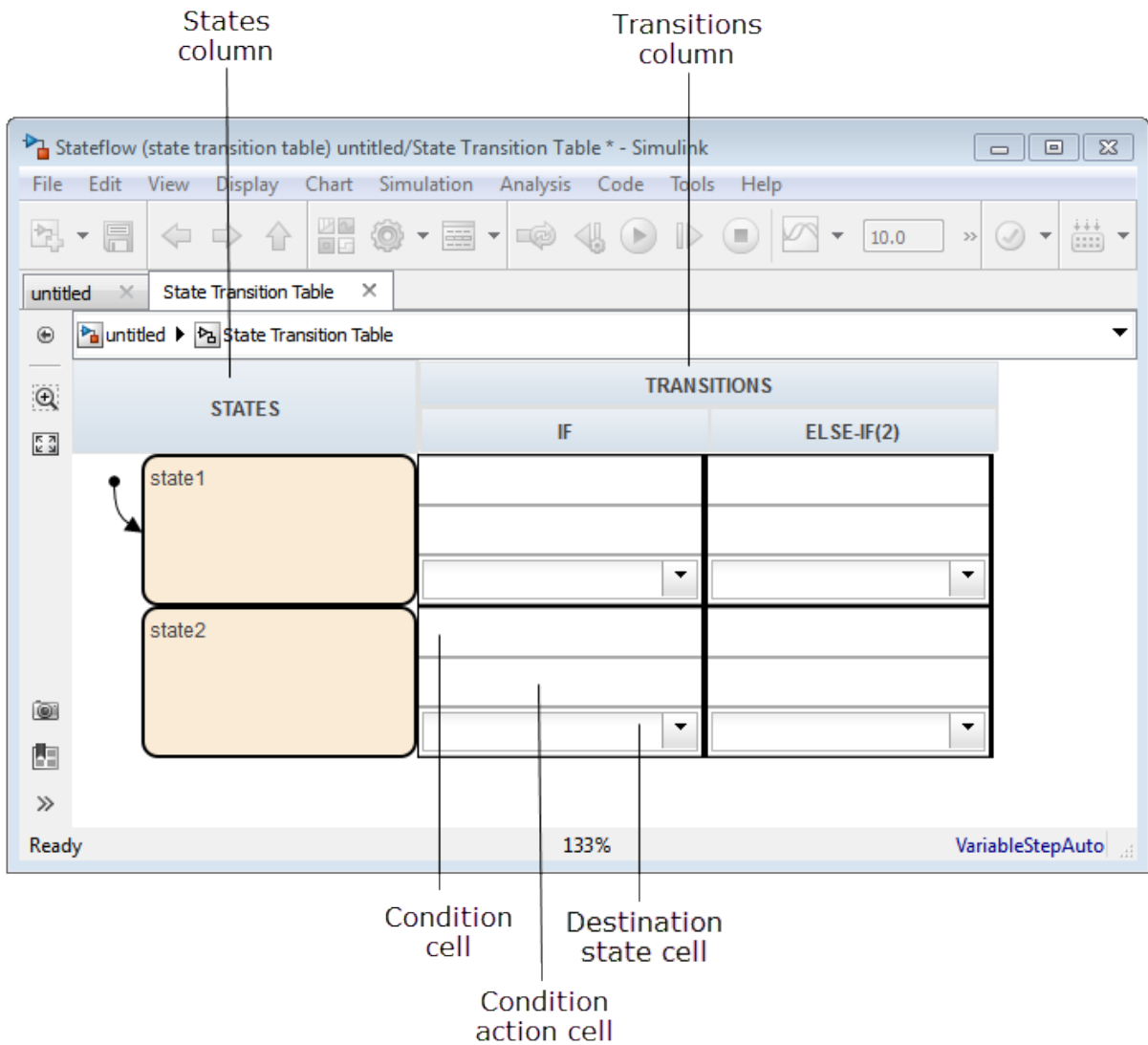
Each transition column contains the following state-to-state transition information:

- Condition
- Condition action

• Destination state

Create State Transition Table and Specify Properties

How to Create a New State Transition Table

At the MATLAB command prompt, enter:

```
sfnew('-STT')
```

From the Simulink Library Browser:

1. Select the Stateflow library.
2. Drag a state transition table into your model.

Properties for State Transition Tables

To access properties, in the Stateflow editor, select **Chart > Properties**.
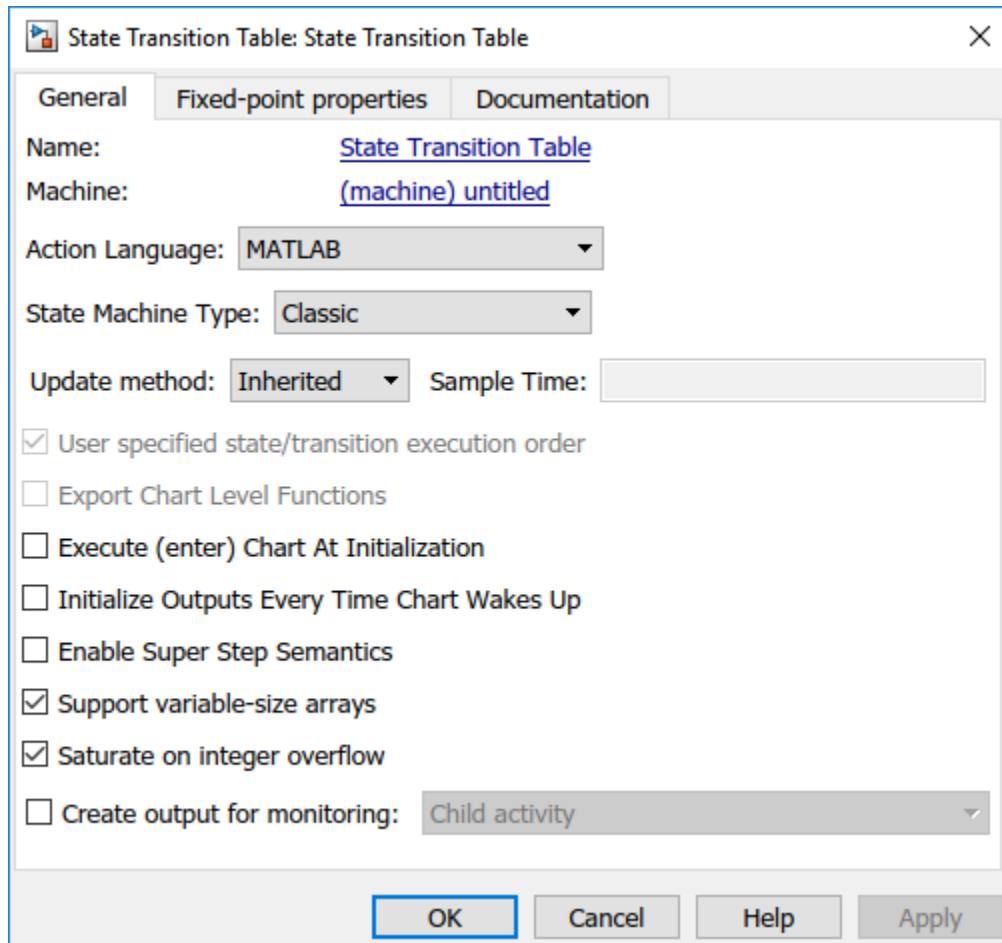
These properties are the same as those for charts that use MATLAB as the action language. For a description of each property, see "Specify Properties for Stateflow Charts" on page 24-3.

Generate Diagrams from State Transition Tables

Stateflow software automatically generates a read-only state transition diagram from the state transition table you create. As you enter changes to a state transition table,

Stateflow incrementally updates the diagram as well. To see the most up-to-date version of the underlying diagram, select **Chart > View auto-generated diagram**.

See Also

State Transition Table

More About

*   "State Transition Table Operations" on page 14-10
*   "Debug Run-Time Errors in a State Transition Table" on page 14-35
*   "Model Bang-Bang Controller by Using a State Transition Table" on page 14-20

State Transition Table Operations

A state transition table is an alternative way of expressing sequential modal logic. Instead of drawing states and transitions graphically in a Stateflow chart, use state transition tables to express the modal logic in tabular format. State transition tables are supported only as blocks in a Simulink model. For more information, see "State Transition Tables in Stateflow" on page 14-2.

To create state transition tables, use the Stateflow Editor. You can insert, edit, and move rows and columns. You can also add history junctions and set the default state for the state transition table.

Insert Rows and Columns

To insert a row:

1   Select an existing state.

2   Select **Chart** > **Insert Row** > and choose one of these options:

Option	Description
State Row	Inserts a state at the same level of hierarchy.
Child State Row	Inserts a state as a child of the selected state.
Default Transition Path Row	Inserts a row for specifying conditional default transition paths.
Inner Transition Path Row	Inserts a row for specifying inner transitions from the selected parent state to its child states.

To insert a column:

1   Select **Chart** > **Append Transition Column**. A new `else-if` column appears to the right of the last column.

Move Rows and Cells

To move a row, click the state cell and drag the row to a new location. As you drag the row, you see a visual indicator of where in the hierarchy the state will appear in its new position.

To move a transition cell, click anywhere in the cell and drag the condition, action, and destination cells as a unit to a new location. The transition cell you displace moves one cell to the right. If column does not exist, Stateflow creates one. The state transition table prevents you from moving cells to an invalid destination and alerts you to the problem.

Copy Rows and Transition Cells

To copy a row:

1 Right-click the state in the row you want to copy and select **Copy**.
2 Right-click the state in the destination row and select **Paste**.

   The row appears above the destination row.

To copy a transition cell:

1 Right-click a cell and select **Copy**.
2 Right-click a destination cell of the same type and select **Paste**.

   The new content overwrites the existing content at the destination. The state transition table prevents you from copying content to an invalid destination.

Set Default State

Right-click the state and select **Set to default**.

Add History Junction

You can add history junctions to states that have children. Right-click the state and select **Add history junction**.

Print State Transition Tables

To print a copy of the state transition table, select **File > Print > Print** or press **Ctrl+P (Command+P)**.

Select and Clear Table Elements

Task	Action
Select a cell for editing	Click the cell.
Exit edit mode in a cell	Press **Esc** or click another table, cell, row, or column.

Undo and Redo Edit Operations

To undo the effects of the previous operation, select **Edit + Undo** or press **Ctrl+Z (Command+Z)**.

To redo the effects of the previous operation, select **Edit + Redo** or press **Ctrl+Y (Command+Y)**.

You can undo and redo up to 10 operations.

Zoom

To zoom in and out of your State Transition Table, select **View > Zoom**. You can add new rows to your State Transition Table at the zoom level that is consistent with your chart. When you save your model, the zoom level is saved.

See Also

More About

View State Reactions by Using a State Transition Matrix

A state transition table is an alternative way of expressing sequential modal logic. Instead of drawing states and transitions graphically in a Stateflow chart, use state transition tables to express the modal logic in tabular format. State transition tables are supported only as blocks in a Simulink model. For more information, see "State Transition Tables in Stateflow" on page 14-2.

State Transition Matrix

The state transition matrix is an alternative view of a state transition table. In the state transition matrix, you can easily see how the model reacts to each condition and event from the state transition table.

Each row represents a state in the state transition table. Each column represents a condition or event. To see the reaction of a state to each event or condition, scan across the cells in a state row. To see how all states respond to an event or condition, scan down the cells of a column.

Create a State Transition Matrix

1 Open the model that you build in "Model Bang-Bang Controller by Using a State Transition Table" on page 14-20.

2 Select **Chart > View State Transition Matrix**.

States Filter states	ALARM	CLEAR	after(10, sec)	doneWarmup
Normal	1 Action: Destination: Alarm	X	X	X

Off
entry:
boiler_cmd = 0;
doneWarmup = false;
	X	X	X	X

Warmup entry: boiler_cmd = 2;	X	X	2 Action: doneWarmup = true; Destination: On	1 Action: Destination: On
On entry: boiler_cmd = 1;	X	X	X	X

The order of the state rows is the same as the state transition table. The order of the columns is based on the number of states that respond to the condition or event. The leftmost column is the condition or event that impacts the highest number of state cells. The rightmost column is the condition or event that impacts the fewest number of states.

The background color of the condition cell is light gray for states that do not react to the condition in the corresponding column. When a state has no further conditions to the right of the condition cell to follow, the condition cell is dark gray.

The execution order appears in the upper-right corner of each transition cell. The execution order is red if it is out of order relative to the event columns. For example, in the state Warmup, the doneWarmup condition is tested before after(10, sec). Because the after(10, sec) column is before the doneWarmup column, the execution order for each corresponding cell is shown in red.

If you change the state transition table, you must regenerate the state transition matrix.

Filter by State Name

To see a subset of state rows, you can filter rows based on state names. In the upper left corner of the state transition matrix, enter a state name in the `States` search box. Stateflow provides valid state name options as you type.

If you have too many events and conditions to view at once, you can scroll through the window horizontally. When you scroll to the right, you continue to see the state names as an overlay on each row.

Traceability to State Transition Table

In the state transition matrix cells, the state names, actions, conditions, and destinations are hyperlinks. To see the corresponding state, action, condition, or destination highlighted in the state transition table, click one of these hyperlinks. To remove the highlighting on the state transition table, select **Display** > **Remove Highlighting**.

For more information about traceability and generated C/C++ code, see "Trace Stateflow Elements in Generated Code" (Embedded Coder). For more information about traceability and generated HDL code, see "Navigate Between Simulink Model and HDL Code by Using Traceability" (HDL Coder).

See Also

More About

- "State Transition Tables in Stateflow" on page 14-2
- "Model Bang-Bang Controller by Using a State Transition Table" on page 14-20

Highlight Flow of Logic in a State Transition Table

A state transition table is an alternative way of expressing sequential modal logic. Instead of drawing states and transitions graphically in a Stateflow chart, use state transition tables to express the modal logic in tabular format. State transition tables are supported only as blocks in a Simulink model. For more information, see "State Transition Tables in Stateflow" on page 14-2.

To visualize a flow of logic, you can highlight one transition cell per row in your state transition table. Highlighting can be used to show the primary flow of logic from one state to another or the flow that represents an error condition.

The highlighting persists across MATLAB sessions and appears in the autogenerated state transition diagram and the state transition table.

To highlight transition cells:

1   In the transition table editor, right-click the transition cell and select **Mark as primary transition**.

    The transition cell appears with a red border.

2   To complete the flow, highlight additional cells, one per row.

    For example:
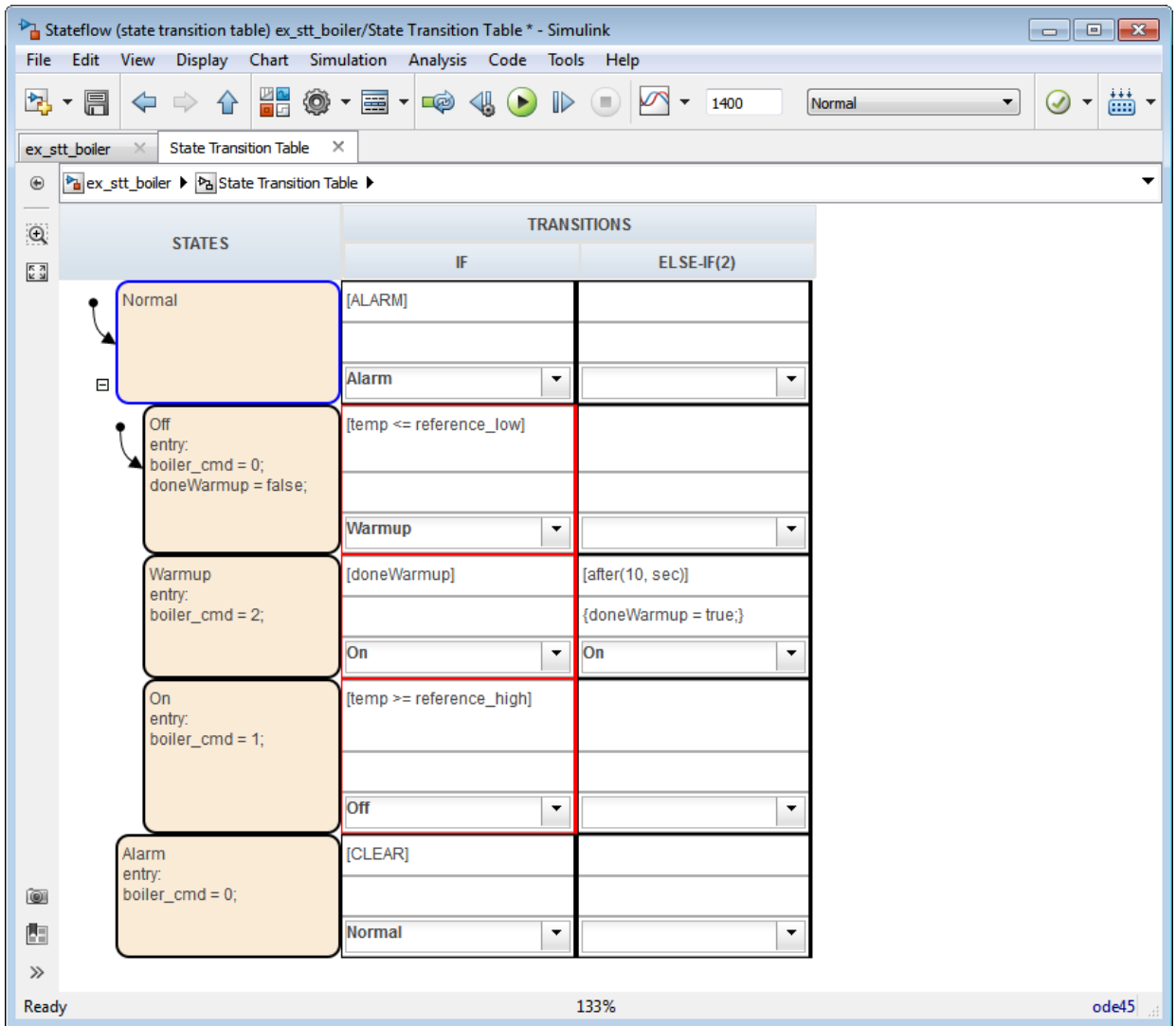
3 To view the flow in the autogenerated state diagram, select **Chart > View auto-generated diagram**.

The transitions that represent the flow appear highlighted in the diagram.

See Also

More About

State Transition Table Diagnostics

A state transition table is an alternative way of expressing sequential modal logic. Instead of drawing states and transitions graphically in a Stateflow chart, use state transition tables to express the modal logic in tabular format. State transition tables are supported only as blocks in a Simulink model. For more information, see "State Transition Tables in Stateflow" on page 14-2.

You can run diagnostic checks on a state transition table. From the Stateflow editor, select **Chart** > **Run Diagnostics**.

The diagnostics tool statically parses the table to find errors such as:

- States with no incoming transitions
- Transition cells with conditions or actions, but no destination
- Action text in a condition cell
- States that are unreachable from the default transition
- Default transition row without unconditional transition
- Inner transition row execution order mismatches. The inner transition row for a state must specify destination states from left to right in the same order as the corresponding states appear in the table, from top to bottom.

Each error is reported with a hyperlink to the corresponding object causing the error. These checks are also performed during simulation.

See Also

More About

- "State Transition Tables in Stateflow" on page 14-2
- "Debug Run-Time Errors in a State Transition Table" on page 14-35
- "Model Bang-Bang Controller by Using a State Transition Table" on page 14-20

Model Bang-Bang Controller by Using a State Transition Table

A state transition table is an alternative way of expressing sequential modal logic. Instead of drawing states and transitions graphically in a Stateflow chart, use state transition tables to express the modal logic in tabular format. State transition tables are supported only as blocks in a Simulink model. For more information, see "State Transition Tables in Stateflow" on page 14-2.

Design Requirements

This example shows how to model a bang-bang controller for temperature regulation of a boiler, using a state transition table. The controller must turn the boiler on and off to meet the following design requirements:

- High temperature cannot exceed 25 degrees Celsius.
- Low temperature cannot fall below 23 degrees Celsius.
- Steady-state operation requires a warm-up period of 10 seconds.
- When the alarm signal sounds, the boiler must shut down immediately.
- When the all-clear signal sounds, the boiler can turn on again.

Identify System Attributes

You can identify the operating modes and data requirements for the bang-bang controller based on its design requirements.

Operating Modes

The high-level operating modes for the boiler are:

- Normal operation, when no alarm signal sounds.
- Alarm state, during an alarm signal.

During normal operation, the boiler can be in one of three states:

- Off, when the temperature is above 25 degrees Celsius.
- Warm-up, during the first 10 seconds of being on.

- On, steady-state after 10 seconds of warm-up, when the temperature is below 23 degrees Celsius.

Data Requirements

The bang-bang controller requires the following data.

Scope	Description	Variable Name
Input	High temperature set point	`reference_high`
Input	Low temperature set point	`reference_low`
Input	Alarm indicator	`ALARM`
Input	All-clear indicator	`CLEAR`
Input	Current temperature of the boiler	`temp`
Local	Indicator that the boiler completed warm-up	`doneWarmup`
Output	Command to set the boiler mode: off, warm-up, or on	`boiler_cmd`

Build the Controller or Use the Supplied Model

To build the bang-bang controller model yourself using a state transition table, follow these exercises. Otherwise, you can open the completed model.

Create a New State Transition Table

To represent the bang-bang controller, use a state transition table. Compared to a graphical state transition diagram, the state transition table is a compact way to represent modal logic that involves transitions between neighboring states. For this example, use MATLAB as your action language.

1 Open the partially built boiler plant model.

This model contains all required Simulink blocks, except for the bang-bang controller.
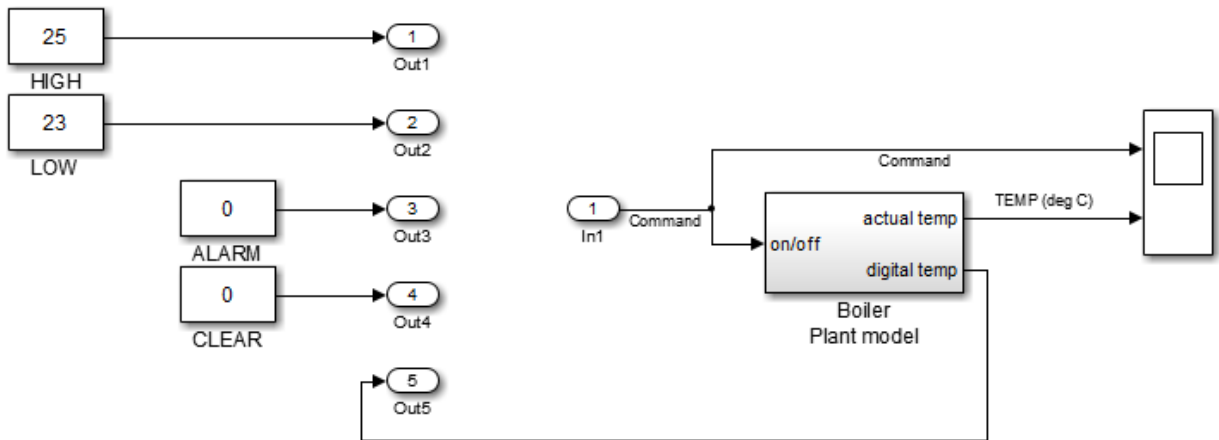
2   Delete the five output ports and the single input port.

3   Open the Library Browser by selecting **View > Library Browser**.

4   In the left pane of the Library Browser, select the Stateflow library, then drag a State Transition Table block from the right pane into your boiler model.

Your model looks like this model.
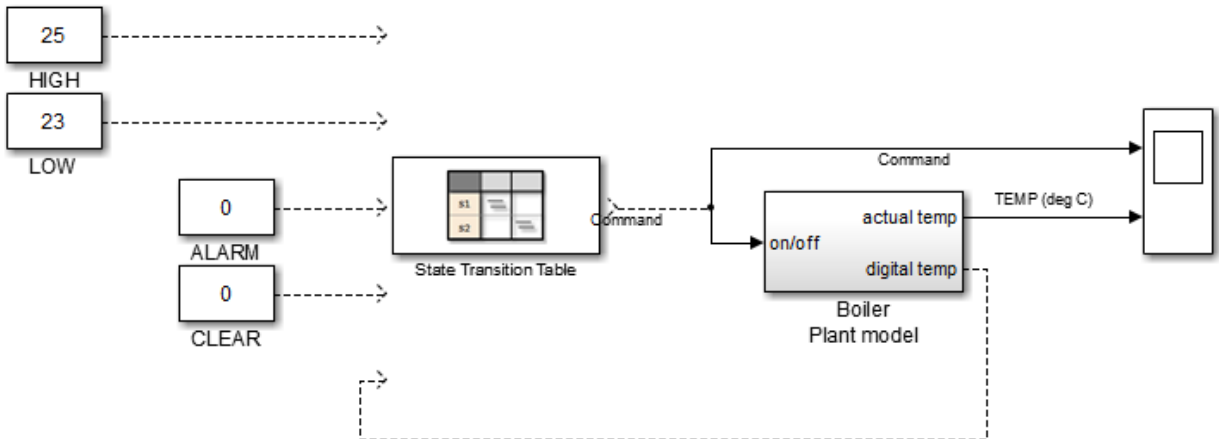
5   Close the Library Browser.

Now you are ready to add states and hierarchy to the state transition table.

Add States and Hierarchy

To represent the operating modes of the boiler, add states and hierarchy to the state transition table.

1   Open the state transition table.

2   Represent the high-level operating modes: normal and alarm.

    a   Double-click `state1` and rename it `Normal`.

    b   Double-click `state2` and rename it `Alarm`.

3   Represent the three states of normal operation as substates of `Normal`:

    a   Right-click the `Normal` state, select **Insert Row > Child State Row**, and name the new state `Off`.

    b   Repeat step **a** two more times to create the child states `Warmup` and `On`, in that order.

By default, when there is ambiguity, the top exclusive (OR) state at every level of hierarchy becomes active first. For this reason, the `Normal` and `Off` states appear with default transitions. This configuration meets the design requirements for this model. To set a default state, right-click the state and select **Set to default**.

Your state transition table looks like this table.
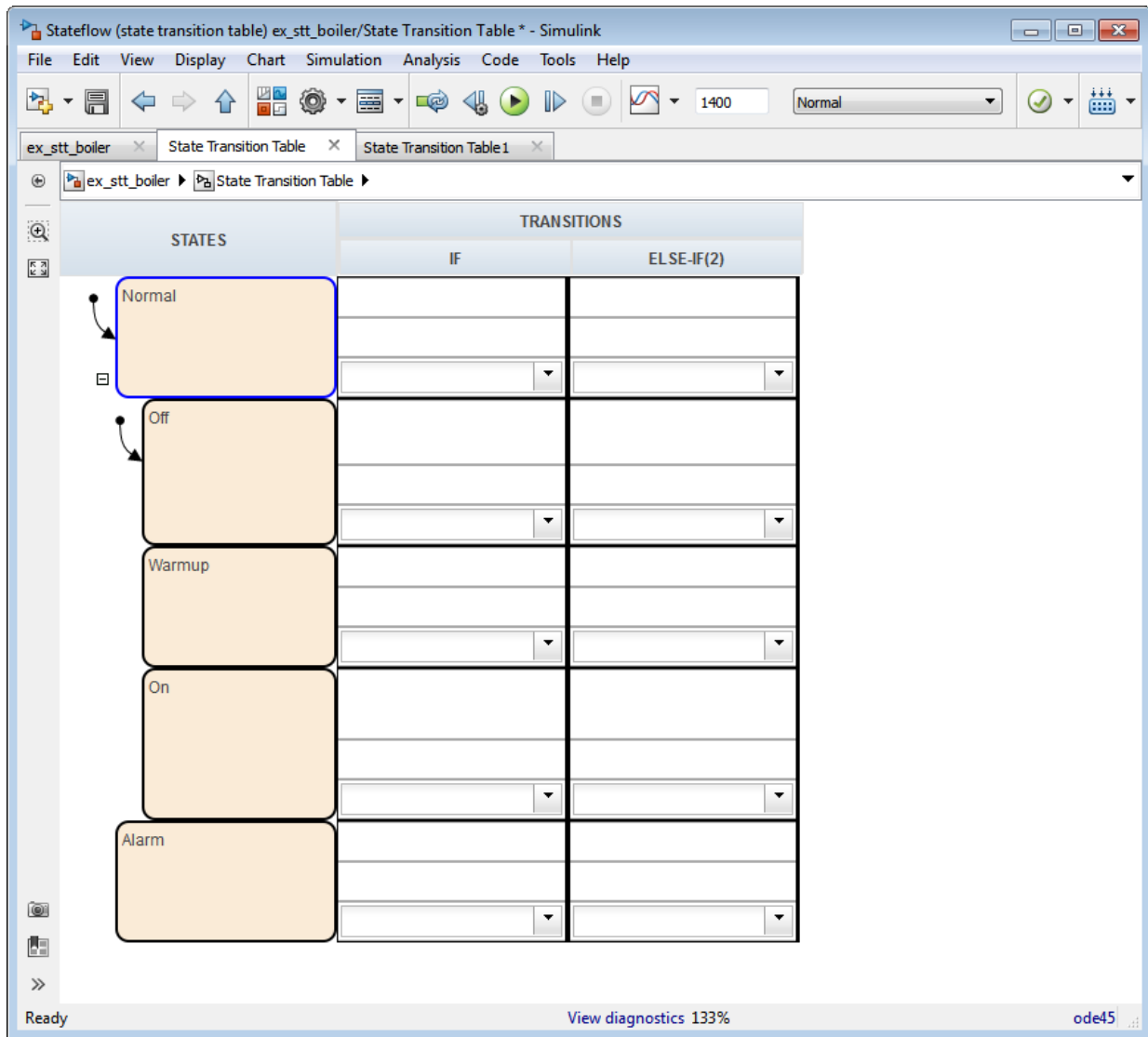
Now you are ready to specify actions for each state.

Specify State Actions

To describe the behavior that occurs in each state, specify state actions in the table. In this exercise, you initialize modes of operation as the boiler enters normal and alarm states, using the variables `boiler_cmd` and `doneWarmup` (described in "Data Requirements" on page 14-21).

1   In the following states, click after the state name, press **Enter**, and type the specified entry actions.

In State:	Type:	Resulting Behavior

`Off`	`entry:`
	`boiler_cmd = 0;`
	`doneWarmup = false;`
		Turns off the boiler and indicates that the boiler has not warmed up.

`Warmup`	`entry:` `boiler_cmd = 2;`	Starts warming up the boiler.
`On`	`entry:` `boiler_cmd = 1;`	Turns on the boiler.
`Alarm`	`entry:` `boiler_cmd = 0;`	Turns off the boiler.

2   Save the state transition table.

Your state transition table looks like this table.
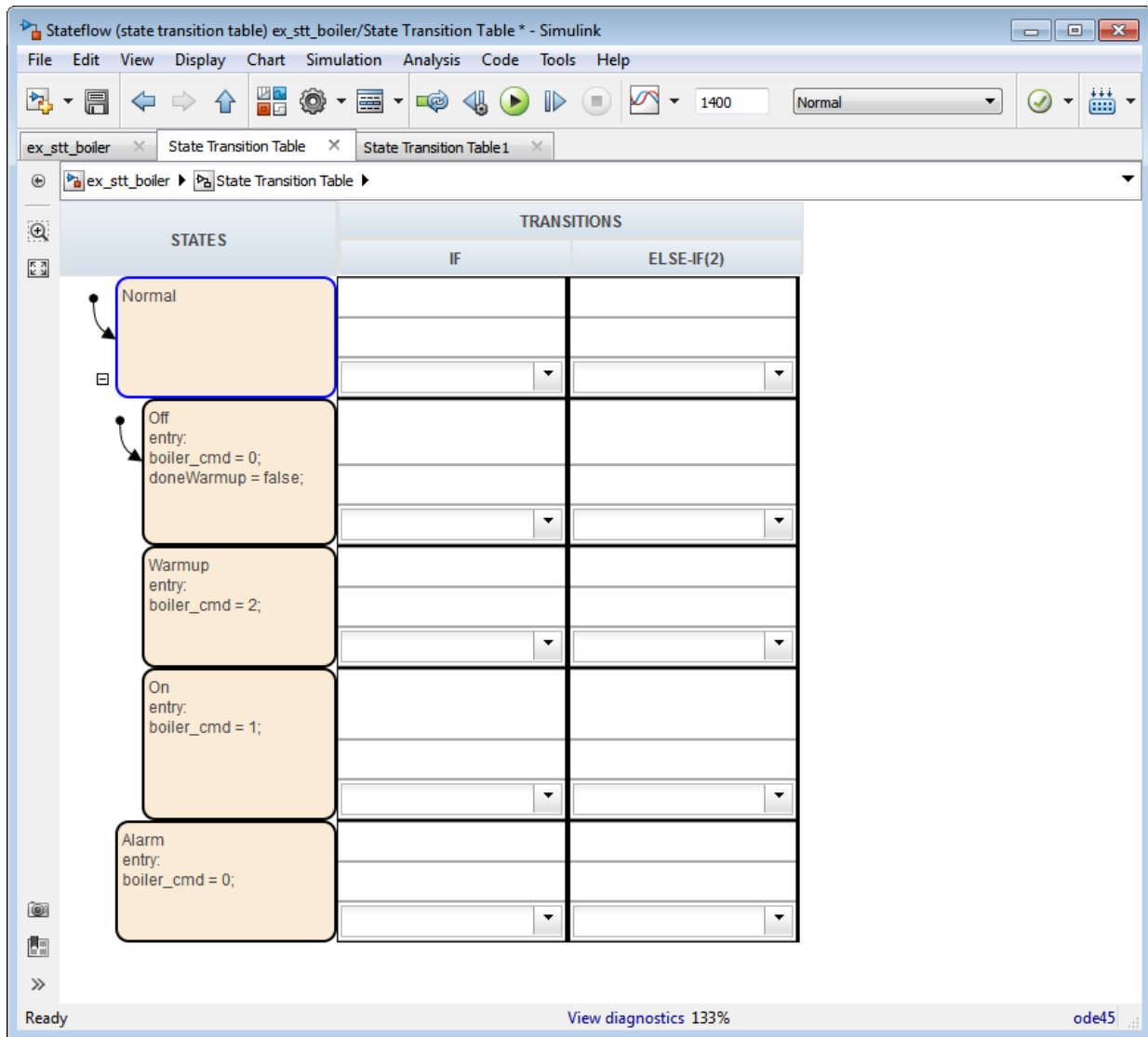
Now you are ready to specify the conditions and actions for transitioning from one state to another state.

Specify Transition Conditions and Actions

To indicate when to change from one operating mode to another, specify transition conditions and actions in the table. In this exercise, you construct statements using variables described in "Data Requirements" on page 14-21.

1   In the `Normal` state row, enter:

if
[ALARM]
Alarm

During simulation:

a   When first entered, the chart activates the `Normal` state.

b   At each time step, normal operation cycles through the `Off`, `Warmup`, and `On` states until the ALARM condition is true.

c   When the ALARM condition is true, the boiler transitions to the `Alarm` state and shuts down immediately.

2   In the `Off` state row, enter:

if
[temp <= reference_low]
Warmup

During simulation, when the current temperature of the boiler drops below 23 degrees Celsius, the boiler starts to warm up.

3   In the `Warmup` state row, enter:

if	else-if
[doneWarmup]	[after(10, sec)]
	{doneWarmup = true;}
On	**On**

During simulation, the boiler warms up for 10 seconds and then transitions to the `On` state.

4 In the `On` state row, enter:

if
[temp >= reference_high]
Off

During simulation, when the current temperature of the boiler rises above 25 degrees Celsius, the boiler shuts off.

5 In the `Alarm` state row, enter:

if
[CLEAR]
Normal

During simulation, when the all-clear condition is true, the boiler returns to normal mode.

6 Save the state transition table.

Your state transition table looks like this table.

Now you are ready to add data definitions using the Symbol Wizard.

Define Data

When you create a state transition table that uses MATLAB syntax, there are language requirements for C/C++ code generation. One of these requirements is that you define the size, type, and complexity of all MATLAB variables so that their properties can be determined at compile time. Even though you have not yet explicitly defined the data in your state transition table, you can use the Symbol Wizard. During simulation, the Symbol Wizard alerts you to unresolved symbols, infers their properties, and adds the missing data to your table.

1   In the Simulink model editor menu, select **Simulation > Run**.

Two dialog boxes appear:

- The Diagnostic Viewer indicates that you have unresolved symbols in the state transition table.
- The Symbol Wizard attempts to resolve the missing data. The wizard correctly infers the scope of all data except for the inputs `ALARM and CLEAR`.
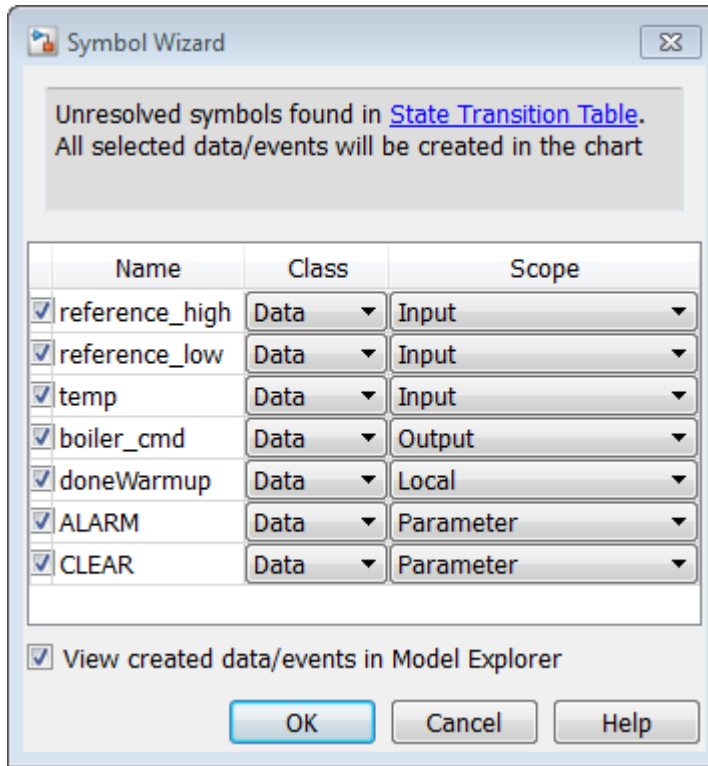
2 In the Symbol Wizard, correct the scopes of ALARM and CLEAR by selecting **Input** from their Scope drop-down lists.

3 When the Model Explorer opens, verify that the Symbol Wizard added all required data definitions correctly.

Some of the inputs are assigned to the wrong ports.

4 In the Contents pane of the Model Explorer, reassign input ports as follows:

Assign:	To Port:
reference_low	2
reference_high	1
temp	5
ALARM	3

Assign:	To Port:
CLEAR	4

5 Save the state transition table.

6 Close the Diagnostic Viewer and the Model Explorer.

In the Simulink model, the inputs and outputs that you defined appear in the State Transition Table block. Now you are ready to connect these inputs and outputs to the Simulink signals and run the model.

Connect the Transition Table and Run the Model

1 In the Simulink model, connect the state transition table to the Simulink inputs and outputs:
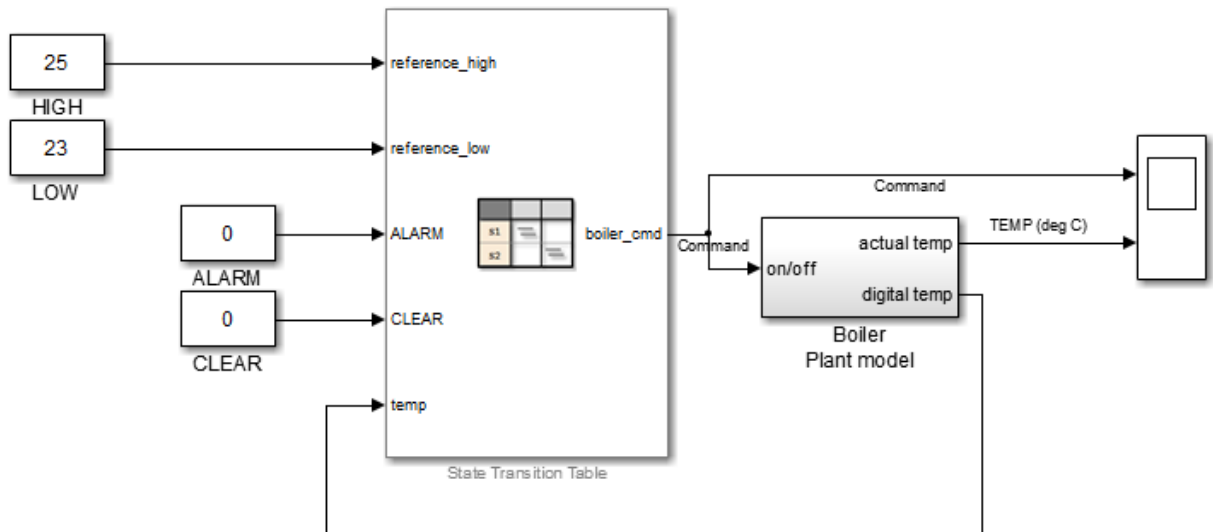
2 Save the model.

3 Reopen your state transition table.

4 Start the simulation by selecting **Simulation** > **Run**.

As the simulation runs, you can watch the animation in the state transition table activate different states.

The following output appears in the Scope block.
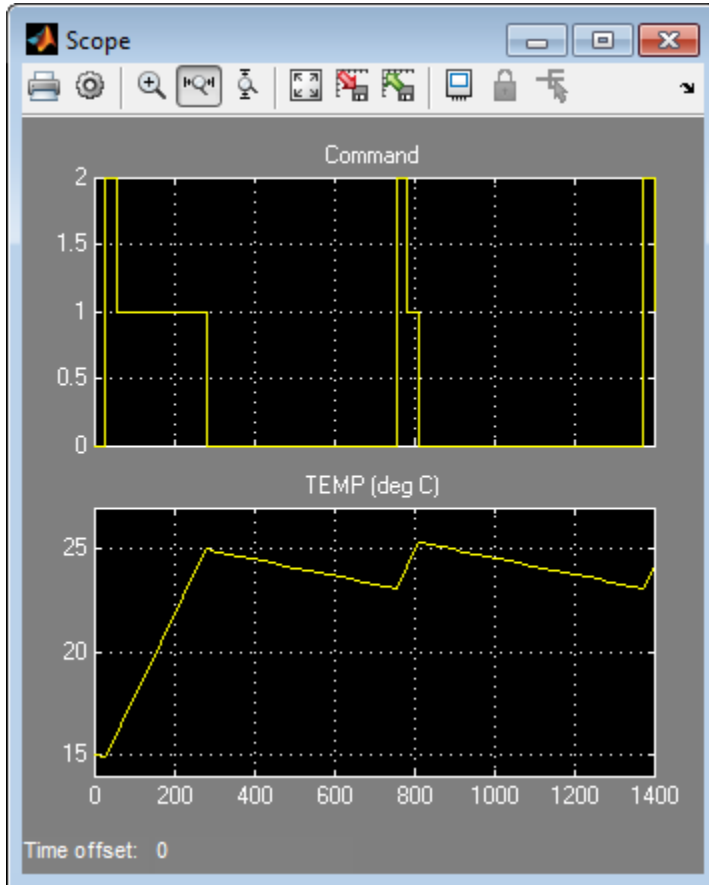
When performing interactive debugging, you can set breakpoints on different states and view the data values at different points in the simulation. For more information about debugging, see "Debugging Charts" on page 32-2.

View the Graphical Representation

Stateflow automatically generates a read-only graphical representation of the state transition table you created.

1 In the state transition table, select **Chart** > **View auto-generated diagram**.

The top-level state transition diagram:
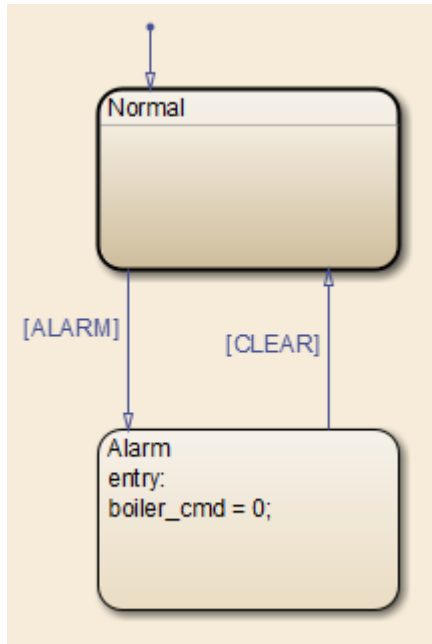
The Normal state appears as a subchart.

2 To view the states and transitions the chart contains, double-click the Normal state.

See Also

More About
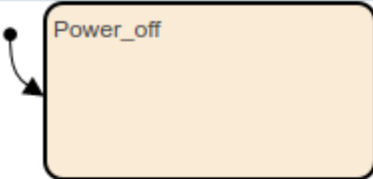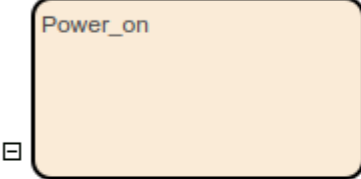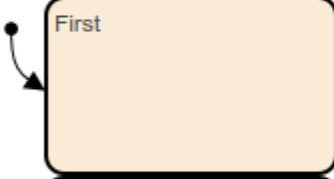
Debug Run-Time Errors in a State Transition Table

A state transition table is an alternative way of expressing sequential modal logic. Instead of drawing states and transitions graphically in a Stateflow chart, use state transition tables to express the modal logic in tabular format. State transition tables are supported only as blocks in a Simulink model. For more information, see "State Transition Tables in Stateflow" on page 14-2.

Create the Model and the State Transition Table

1 Create a Simulink model with a new State Transition Table (`sfnew -stt`).

2 Add the following states and transitions to your table:

STATES	TRANSITIONS	
	IF	ELSE-IF(2)
Power_off	[SWITCH]	
	Power_on ▼	▼
Power_on	[SWITCH]	
	Power_off ▼	▼
First	[Shift == 1]	
	Second ▼	▼
Second	[Shift == 1]	
	Third ▼	▼
Third	[Shift == 1]	
	First ▼	▼

The table has two states at the highest level in the hierarchy, Power_off and Power_on. By default, Power_off is active. The event SWITCH toggles the system between the Power_off and Power_on states. Power_on has three substates: First, Second, and Third. By default, when Power_on becomes active, First also becomes active. When Shift equals 1, the system transitions from First to

Second, Second to Third, and Third to First, for each occurrence of the event SWITCH. Then the pattern repeats.

3   Add two inputs on page 9-2 from Simulink:

- An event called SWITCH with a scope of **Input from Simulink** and a **Rising** edge trigger.

- A data called Shift with a scope of **Input from Simulink**.

4   In the model view, connect a Sine Wave block as the SWITCH event and a Step block as the Shift data for your State Transition Table.
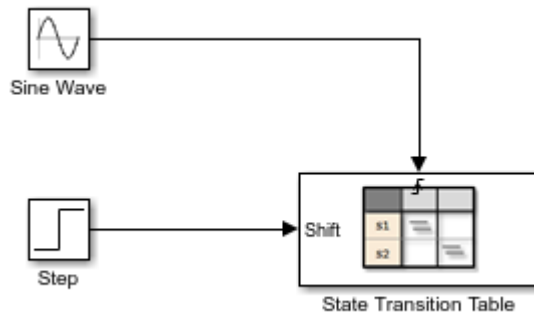
In the model, there is an event input and a data input. A Sine Wave block generates a repeating input event that corresponds with the Stateflow event SWITCH. The Step block generates a repeating pattern of 1 and 0 that corresponds with the Stateflow data object Shift. Ideally, the SWITCH event occurs at a frequency that allows at least one cycle through First, Second, and Third.

Debug the State Transition Table

To debug the table in "Create the Model and the State Transition Table" on page 14-35, follow these steps:

1   Right-click the Power_off state, and select **Set Breakpoint** > **On State Entry**.

2   Start the simulation.

Because you specified a breakpoint on Power_off, execution stops at that point.

3 Move to the next step by clicking the Step In button, .

4 To see the data used and the current values, hover your cursor over the different table cells.

Continue clicking the Step In button and watching the animating states. After each step, watch the chart animation to see the sequence of execution. Use the tooltips to see the data values.

Single-stepping shows that the loop from `First` to `Second` to `Third` inside the state `Power_on` does not occur. The transition from `Power_on` to `Power_off` takes priority.

Correct the Run-Time Error

In "Debug the State Transition Table" on page 14-37, you step through a simulation of a state transition table and find an error. The event `SWITCH` drives the simulation, but the simulation time passes too quickly for the input data object `Shift` to have an effect.

To correct this error:

1 Stop the simulation so that you can edit the table.

2 Add the condition `after(20.0, sec)` to the transition from `Power_on` to `Power_off`.

| STATES | TRANSITIONS | |
	IF	ELSE-IF(2)
Power_off ⊙	[SWITCH]	
	Power_on ▾	▾
Power_on	[SWITCH && after(20.0,sec)]	
	Power_off ▾	▾
First	[Shift == 1]	
	Second ▾	▾
Second	[Shift == 1]	
	Third ▾	▾
Third	[Shift == 1]	
	First ▾	▾

Now the transition from Power_on to Power_off does not occur until 20 seconds have passed.

3 Begin simulation.

4 Click the Step In button repeatedly to observe the fixed behavior.

See Also

Related Examples

- "State Transition Tables in Stateflow" on page 14-2
- "State Transition Table Diagnostics" on page 14-19
- "Model Bang-Bang Controller by Using a State Transition Table" on page 14-20

Make States Reusable with Atomic Subcharts

Create Reusable Subcomponents by Using Atomic Subcharts

An *atomic subchart* is a graphical object that helps you to create standalone subcomponents in a Stateflow chart. Atomic subcharts are supported only in Stateflow charts in Simulink models.

Atomic subcharts allow for:

- Faster simulation after making small changes to a chart with many states or levels of hierarchy.
- Reuse of the same state or subchart across multiple charts and models.
- Ease of team development for people working on different parts of the same chart.
- Manual inspection of generated code for a specific state or subchart in a chart.

An atomic subchart looks opaque and includes the label **Atomic** in the upper left corner. If you use a linked atomic subchart from a library, the label **Link** appears in the upper left corner.

Example of an Atomic Subchart

This example illustrates the difference between a normal subchart and an atomic subchart.
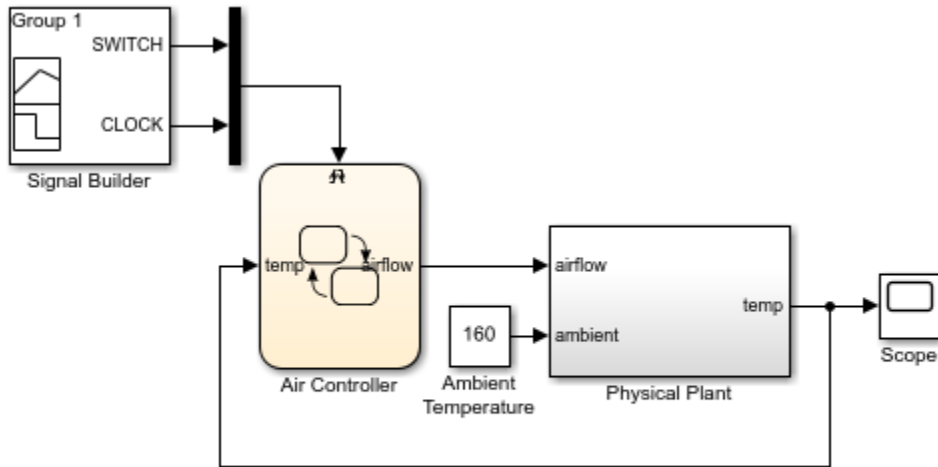
Copyright 2018 The MathWorks, Inc.

In the Air Controller chart, `PowerOff` is a normal subchart. `PowerOn` is an atomic subchart. Both subcharts look opaque, but the `PowerOn` includes the label **Atomic** on the upper left corner.
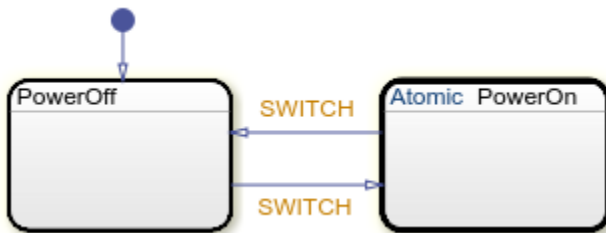
Benefits of Using Atomic Subcharts

Atomic subcharts combine the functionality of states, normal subcharts, and atomic subsystems. Atomic subcharts:

- Behave as standalone charts.

- Support usage as library links.
- Support the generation of reusable code.
- Allow mapping of inputs, outputs, parameters, data store memory, and input events.

Atomic subcharts do not support access to:

- Data at every level of the chart hierarchy.
- Event broadcasts outside the scope of the atomic subchart.

Atomic subcharts do not support explicit specification of sample time.

Create an Atomic Subchart

You can create an atomic subchart by converting an existing subchart or by linking a chart from a library model. After creating the atomic subchart, update the mapping of variables by right-clicking the atomic subchart and selecting **Subchart Mappings**. For more information, see "Map Variables for Atomic Subcharts and Boxes" on page 15-9.

Convert a Normal Subchart to an Atomic Subchart

To create a standalone component that allows for faster debugging and code generation workflows, convert an existing state or subchart into an atomic subchart. In your chart, right-click a state or a normal subchart and select **Group & Subchart > Atomic Subchart**. The label **Atomic** appears in the upper left corner of the subchart.

The conversion provides the atomic subchart with its own copy of every data object that the subchart accesses in the chart. Local data is copied as data store memory. The scope of other data, including input and output data, does not change.

Note If a state or subchart contains messages, you cannot convert it to an atomic subchart.

Link an Atomic Subchart from a Library

To create a subcomponent for reuse across multiple charts and models, create a link from a library model. Copy a chart in a library model and paste it to a chart in another model. If the library chart contains any states, it appears as a linked atomic subchart with the label **Link** in the upper left corner.

This modeling method minimizes maintenance of similar states. When you modify the atomic subchart in the library, your changes propagate to the links in all charts and models.

If the library chart contains only functions and no states, then it appears a linked atomic box in the chart. For more information, see "Reuse Functions by Using Atomic Boxes" on page 8-35.

Convert an Atomic Subchart to a Normal Subchart

Converting an atomic subchart back to a state or a normal subchart removes all of its variable mappings. The conversion merges subchart-parented data objects with the chart-parented data to which they map.

1   If the atomic subchart is a library link, right-click the atomic subchart and select **Library Link > Disable Link**.

2   To convert an atomic subchart back to a normal subchart, right-click the atomic subchart and clear the **Group & Subchart > Atomic Subchart** check box.

3   To convert the subchart back to a state, right-click the subchart and clear the **Group & Subchart > Subchart** check box.

4   If necessary, rearrange graphical objects in your chart.

You cannot convert an atomic subchart to a normal subchart if:

- The atomic subchart maps a parameter to an expression other than a single variable name. For example, mapping a parameter `data1` to one of these expressions prevents the conversion of an atomic subchart to a normal subchart:

  - `3`
  - `data2(3)`
  - `data2 + 3`
- Both of these conditions are true:

  - The atomic subchart contains MATLAB functions or truth table functions that use MATLAB as the action language.
  - The atomic subchart does not map each variable to a variable of the same name in the main chart.

When to Use Atomic Subcharts

Debug Charts Incrementally

Suppose that you want to test a sequence of changes in a chart that contains many states or several levels of hierarchy.

If you do not use atomic subcharts, when you make a small change to one part of a chart and start simulation, recompilation occurs for the entire chart. Because recompiling the entire chart can take a long time, you decide to make several changes before testing. However, if you find an error, you must step through all of your changes to identify the cause of the error.

In contrast, when you modify an atomic subchart, recompilation occurs for only the subchart and not for the entire chart. Incremental builds for simulation require less time to recompile. This reduction in compilation time enables you to test each individual change instead of waiting to test multiple changes at once. By testing each change individually, you can quickly identify a change that causes an error. For more information, see "Reduce the Compilation Time of a Chart" on page 15-43.

Reuse State Logic

Suppose that you want to reuse the same state or subchart many times to facilitate large-scale modeling.

If you do not use atomic subcharts, you have to maintain each copy of a subcomponent manually. For example, this chart contains two states with a similar structure. The only difference between the two states is the names of variables. If you make a change in state A, then you must make the same change in state B.
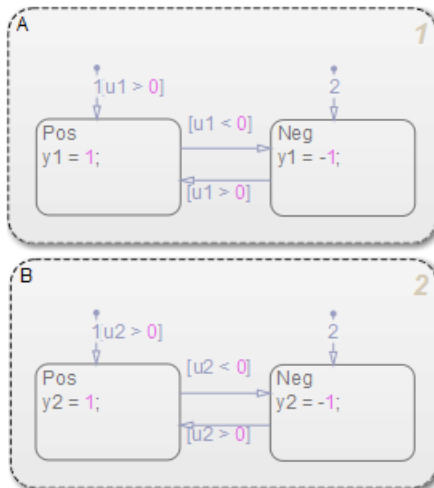
To enable reuse of subcomponents by using linked atomic subcharts, create a single copy of state A and store it as a chart in a library model. From that library, copy and paste the atomic subchart twice in your chart. Then update the mapping of subchart variables as needed.
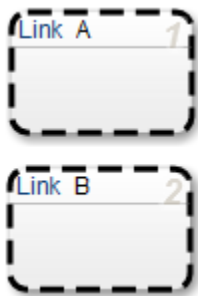
When you change an atomic subchart in a library, the change propagates to all library links. For more information, see "Reuse a State Multiple Times in a Chart" on page 15-34.

Develop Charts Used by Multiple People

Suppose that you want to break a chart into subcomponents because multiple people are working on different parts of the chart.

Without atomic subcharts, only one person at a time can edit the model. If someone edits one part of a chart while someone else edits another part of the same chart, you must merge those changes at submission time.

In contrast, you can store different parts of a chart as linked atomic subcharts. Because atomic subcharts behave as standalone objects, different people can work on different parts of a chart without affecting the other parts of the chart. At submission time, no merge is necessary because the changes exist in separate models. For more information, see "Divide a Chart into Separate Units" on page 15-45.

Inspect Generated Code

Suppose that you want to inspect code generated by Simulink Coder or Embedded Coder manually for a specific part of a chart.

If you do not use atomic subcharts, you generate code for an entire model in one file. To find code for a specific part of the chart, you have to look through the entire file.

In contrast, you can specify that the code for an atomic subchart appears in a separate file. This method of code generation enables unit testing for a specific part of a chart. You avoid searching through unrelated code and focus only on the code that interests you. For more information, see "Generate Reusable Code for Unit Testing" on page 15-48.

See Also

More About
- "Encapsulate Modal Logic by Using Subcharts" on page 8-5
- "Map Variables for Atomic Subcharts and Boxes" on page 15-9
- "Modeling an Elevator System Using Atomic Subcharts"
- "Modeling a Redundant Sensor Pair Using Atomic Subcharts"

Map Variables for Atomic Subcharts and Boxes

An atomic subchart is a graphical object that helps you to create standalone subcomponents in a Stateflow chart. Atomic subcharts are supported only in Stateflow charts in Simulink models. For more information, see "Create Reusable Subcomponents by Using Atomic Subcharts" on page 15-2.

To ensure that each variable in your atomic subchart or atomic box maps to the correct data in the main chart, edit the mapping on the **Mappings** tab in the **Properties** dialog box. For each atomic subchart variable, in the **Main chart symbol** field, you can select the name of the corresponding symbol from the drop-down list. Alternatively, you can type an expression specifying:

- A field of a Stateflow structure. See "Index and Assign Values to Stateflow Structures" on page 25-8.
- An element of a vector or matrix. See "Supported Operations for Vectors and Matrices" on page 17-11.
- Any valid combination of structure fields or matrix indices, such as `struct.field(1,2)` or `struct.field[0][1]`.

If you leave the **Main chart symbol** field empty, then Stateflow attempts to map the atomic subchart variable to a main chart variable with the same name.

You can map a variable in the atomic subchart to a symbol in the main chart that has a different scope. This table lists the possible mappings.

Atomic Subchart Symbol Scope	Main Chart Symbol Scope
Input	Input, Output, Local, Parameter
Output	Output, Local
Parameter	Parameter
Data Store Memory	Data Store Memory, Local
Input Event	Input Event

When you map data store memory in an atomic subchart to local data of enumerated type, you have two options for specifying the initial value of the data store memory:

- In the Data Properties dialog box, set the **Initial value** field for the chart-level local data.

- To apply the default value of the enumerated type, leave the **Initial value** field empty.

Map Input and Output Data for an Atomic Subchart

This model contains two Sine Wave blocks that supply input signals to a chart.
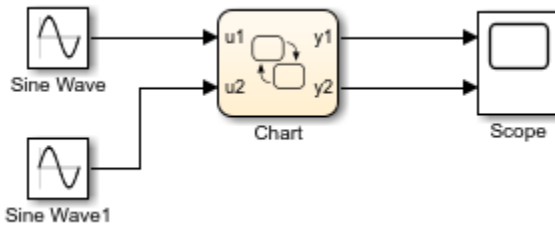
The chart consists of two linked atomic subcharts from the same library.
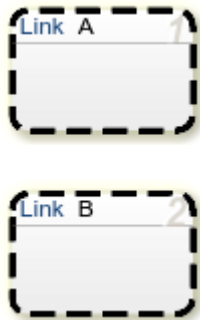
Both atomic subcharts contain these objects.
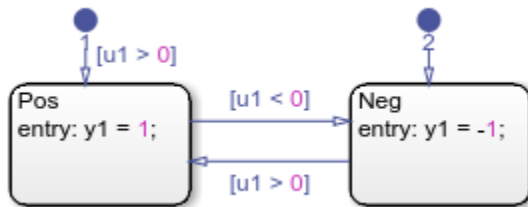
If you simulate the model, the output for y2 is zero.
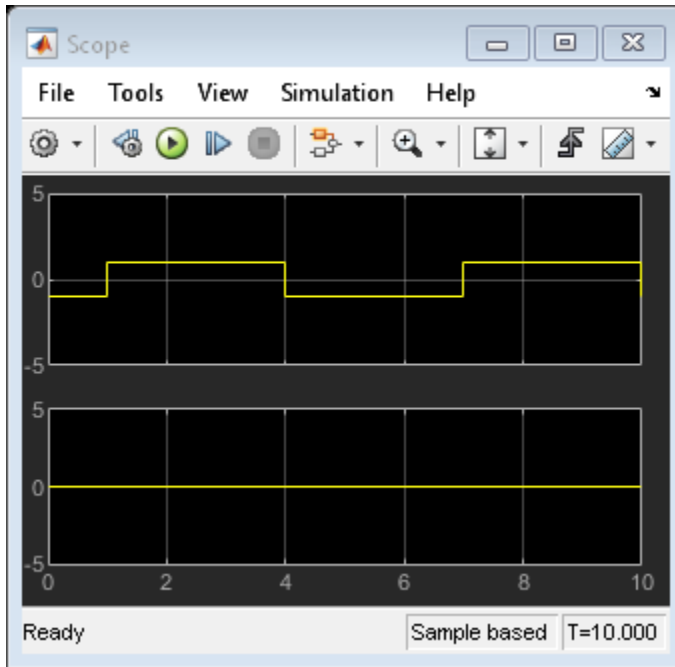
Because the symbols in atomic subchart A have the same name as the symbols u1 and y1 in the main chart, they map to the correct variables. The symbols in atomic subchart B do not map to the correct variables u2 and y2 in the main chart, so you must edit the mapping.

1 Right-click subchart B and select **Subchart Mappings**.

2 Under **Input Mapping**, specify the main chart symbol for u1 to be u2.

3 Under **Output Mapping**, specify the main chart symbol for y1 to be y2.

4 Click **OK**.
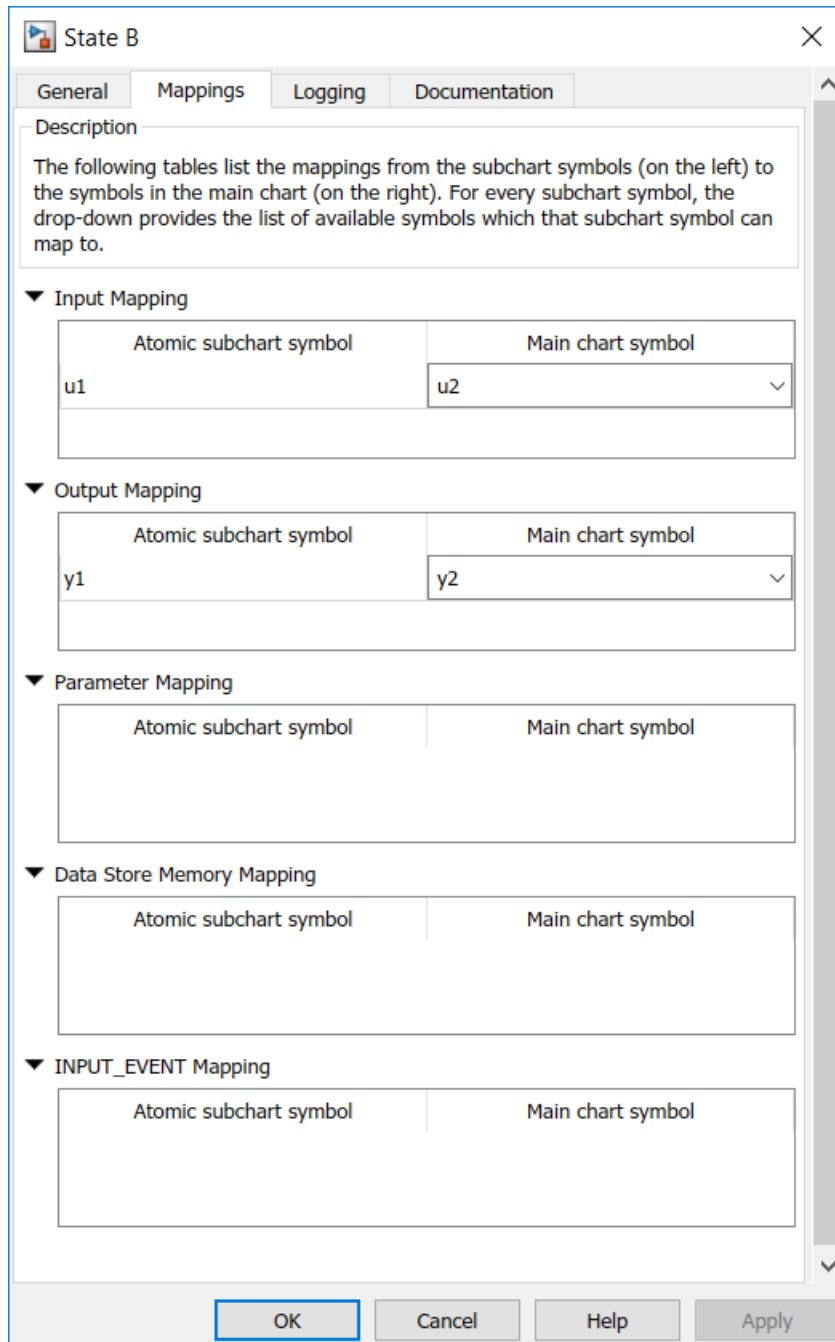
When you run the model again, you get these results.
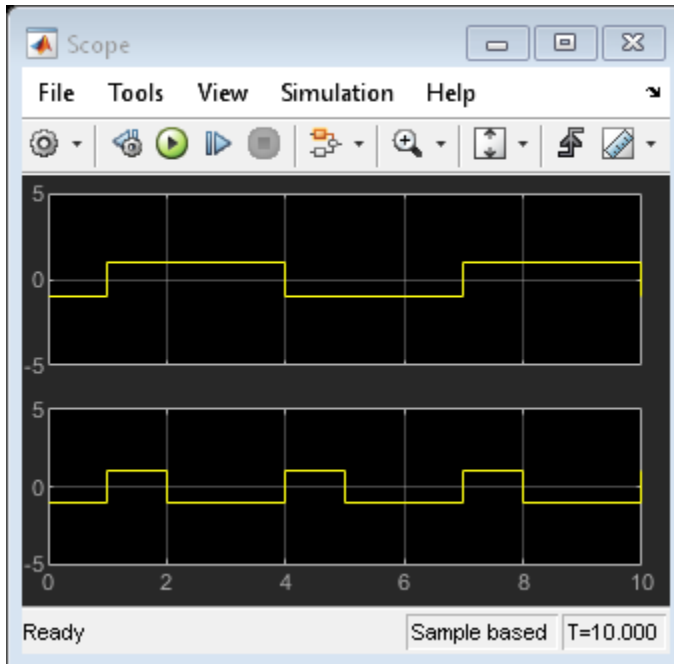
Map Atomic Subchart Variables to Bus Elements

This model contains two Sine Wave blocks that supply signals through a bus to a chart.
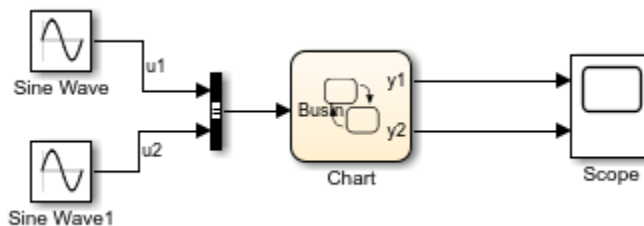
The chart consists of two linked atomic subcharts from the same library. Both atomic subcharts contain these objects.
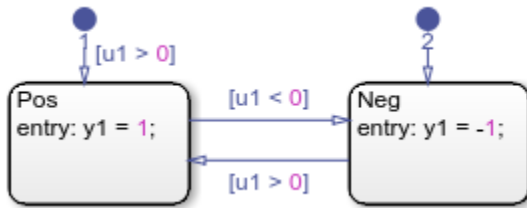
If you simulate the model, you get an error because, in each subchart, the input u1 does not map to any variable in the main chart. To edit the mapping for u1 in each subchart:

1 Right-click subchart A and select **Subchart Mappings**.
2 Under **Input Mapping**, specify the main chart symbol for u1 to be the first element in the bus: BusIn.u1.
3 Click **OK**.
4 Repeat for subchart B, specifying the main chart symbol for u1 to be the second element in the bus: BusIn.u2.

When you run the model again, you get these results.
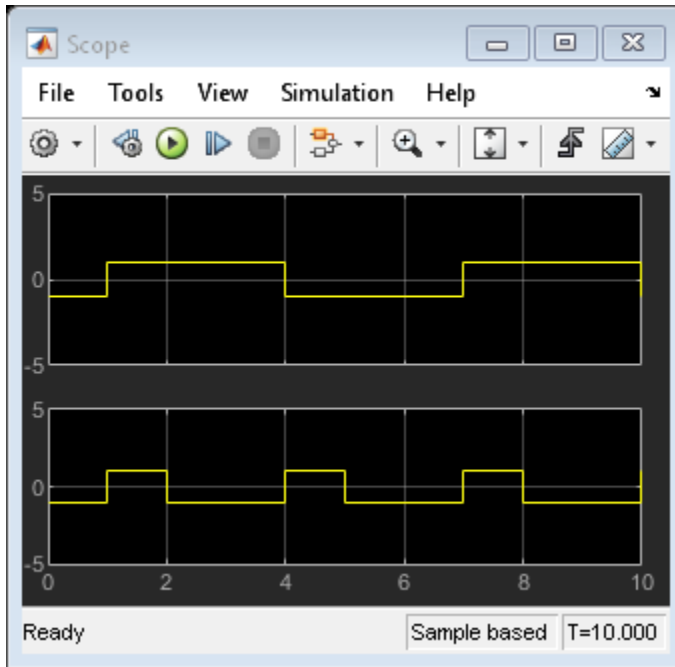
Map Atomic Subchart Variables to Elements of Matrix

When referring to elements of a vector or matrix, regardless of the action language of the chart, use:

- One-based indexing delimited by parentheses and commas. For example, `A(4,5)`.
- Zero-based indexing delimited by brackets. For example, `A[3][4]`.

Indices can be numbers or parameters in the chart. Use of other expressions as indices is not supported.

For example, this model contains two Sine Wave blocks that supply signals through a diagonal matrix to a chart.
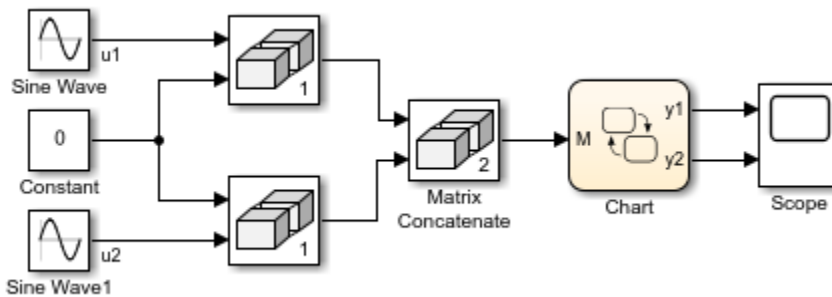
The chart consists of two linked atomic subcharts from the same library. Both atomic subcharts contain these objects.
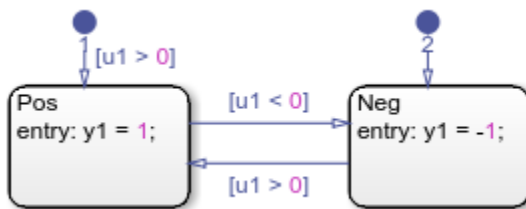
If you simulate the model, you get an error because, in each subchart, the input u1 does not map to any variable in the main chart. To edit the mapping for u1 in each subchart:

1   Right-click subchart A and select **Subchart Mappings**.
2   Under **Input Mapping**, specify the main chart symbol for u1 to be the top-left element in the matrix. For example, using zero-based indexing: M[0][0].
3   Click **OK**.
4   Repeat for subchart B, specifying the main chart symbol for u1 to be the bottom-right element in the matrix. For example, using one-based indexing: M(2,2).

When you run the model again, you get these results.
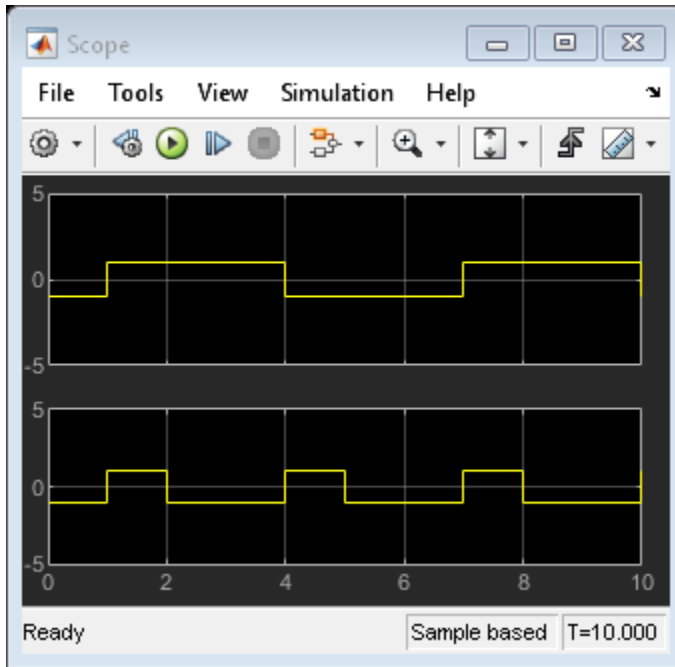
Map Atomic Subchart Parameters to Expressions

For atomic subcharts parameters, you can enter a constant expression or an expression that depends on one or more parameters in the main chart.

For example, this model contains two Sine Wave blocks that supply input signals to a chart.

15-17
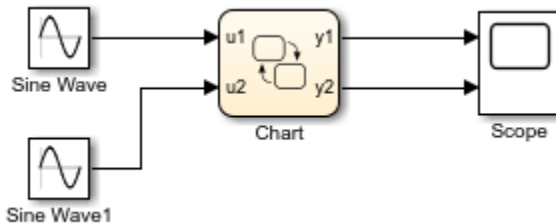
The chart consists of two linked atomic subchart from the same library. Both atomic subcharts contain these objects.
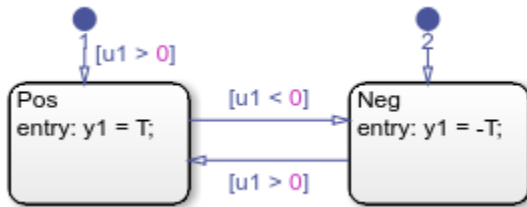
If you simulate the model, you get an error because the parameter T is undefined. To fix this error, specify an expression for T to evaluate in the mask workspace of the main chart:

1   Right-click subchart A and select **Subchart Mappings**.

2   Under **Parameter Mapping**, as the value for T, enter -1.

3   Click **OK**.

4   Repeat for subchart B, specifying the value of T as 2.

State A

General | **Mappings** | Logging | Documentation

Description

The following tables list the mappings from the subchart symbols (on the left) to the symbols in the main chart (on the right). For every subchart symbol, the drop-down provides the list of available symbols which that subchart symbol can map to.

▼ Input Mapping

Atomic subchart symbol	Main chart symbol
u1	u1

▼ Output Mapping

Atomic subchart symbol	Main chart symbol
y1	y1

▼ Parameter Mapping

Atomic subchart symbol	**Main chart symbol**
T	-1

▼ Data Store Memory Mapping

Atomic subchart symbol	Main chart symbol

▼ INPUT_EVENT Mapping

Atomic subchart symbol	Main chart symbol

OK | Cancel | Help | Apply

When you run the model again, you get these results.
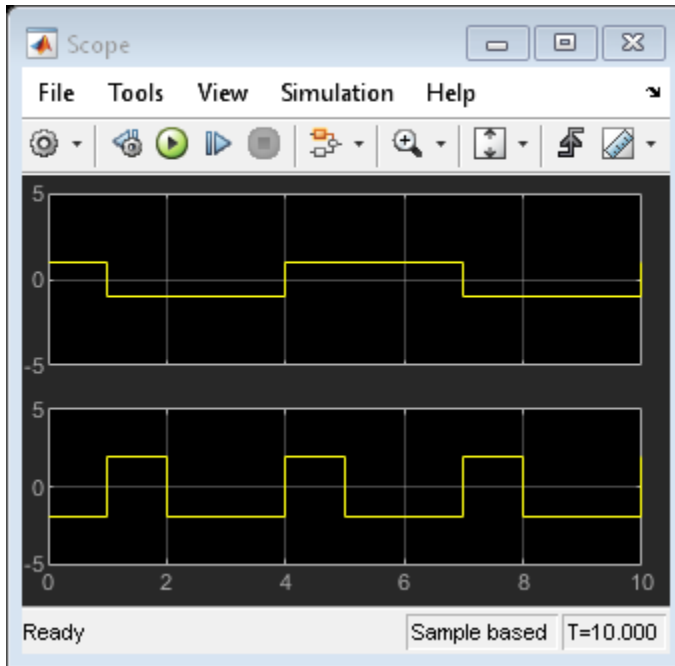
Map Input Events for an Atomic Subchart

The sf_yoyo model contains a Mux block that supplies input events to a chart.
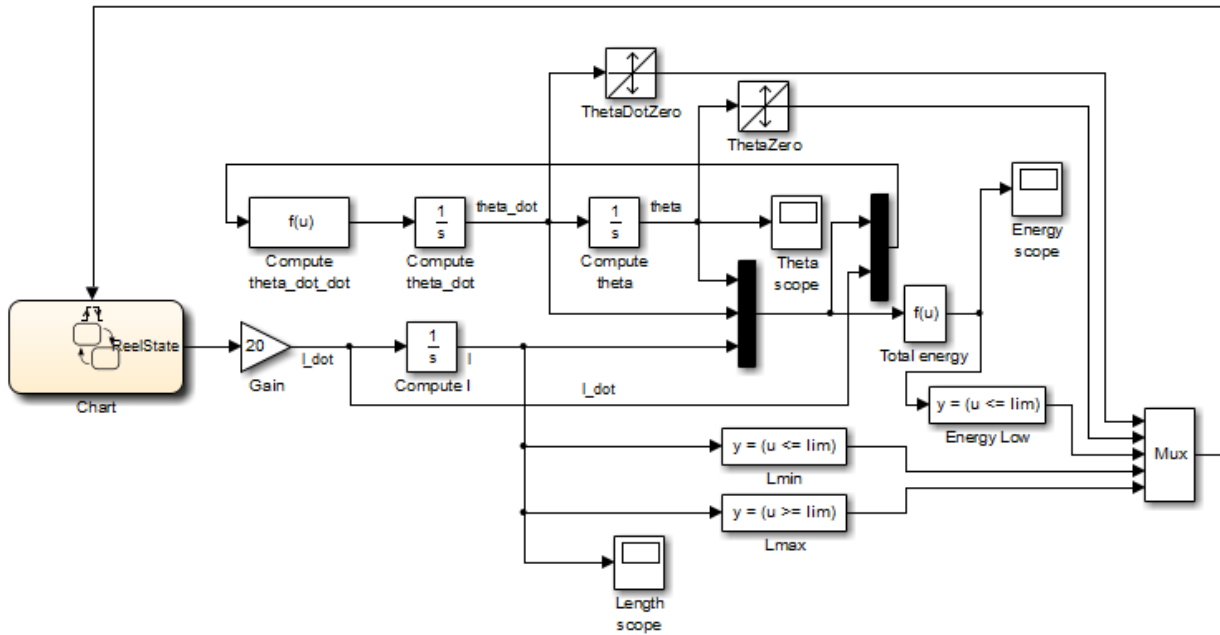
The chart contains two superstates: `Active` and `Inactive`. The `Active` state uses input events to guard transitions between different substates.
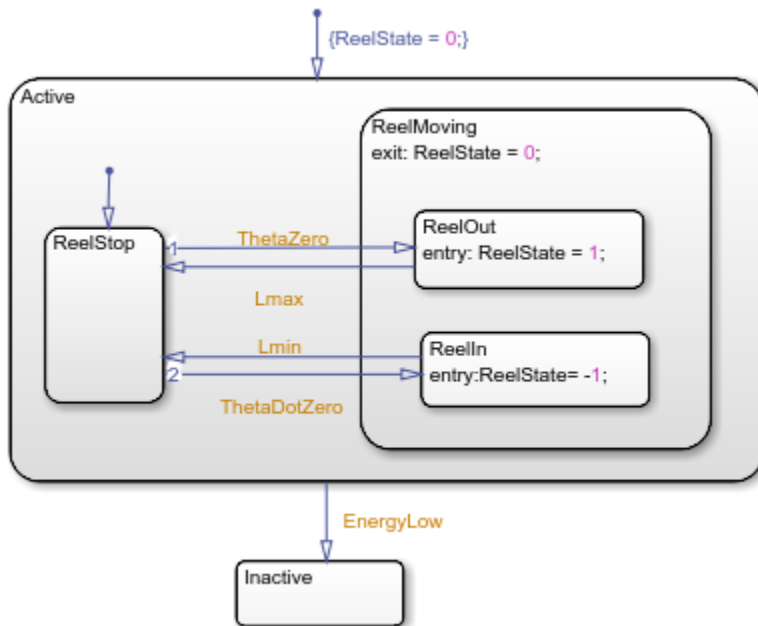
To convert the `Active` state to an atomic subchart:

1   Right-click the `Active` state and select **Group & Subchart** > **Atomic Subchart**.

2   Right-click the atomic subchart and select **Subchart Mappings**.

    Under **Input Event Mapping**, each atomic subchart symbol maps to the correct input event in the main chart.

    The default mappings also follow the rules of using input events in atomic subcharts. For more information, see "Rules for Using Atomic Subcharts" on page 15-29
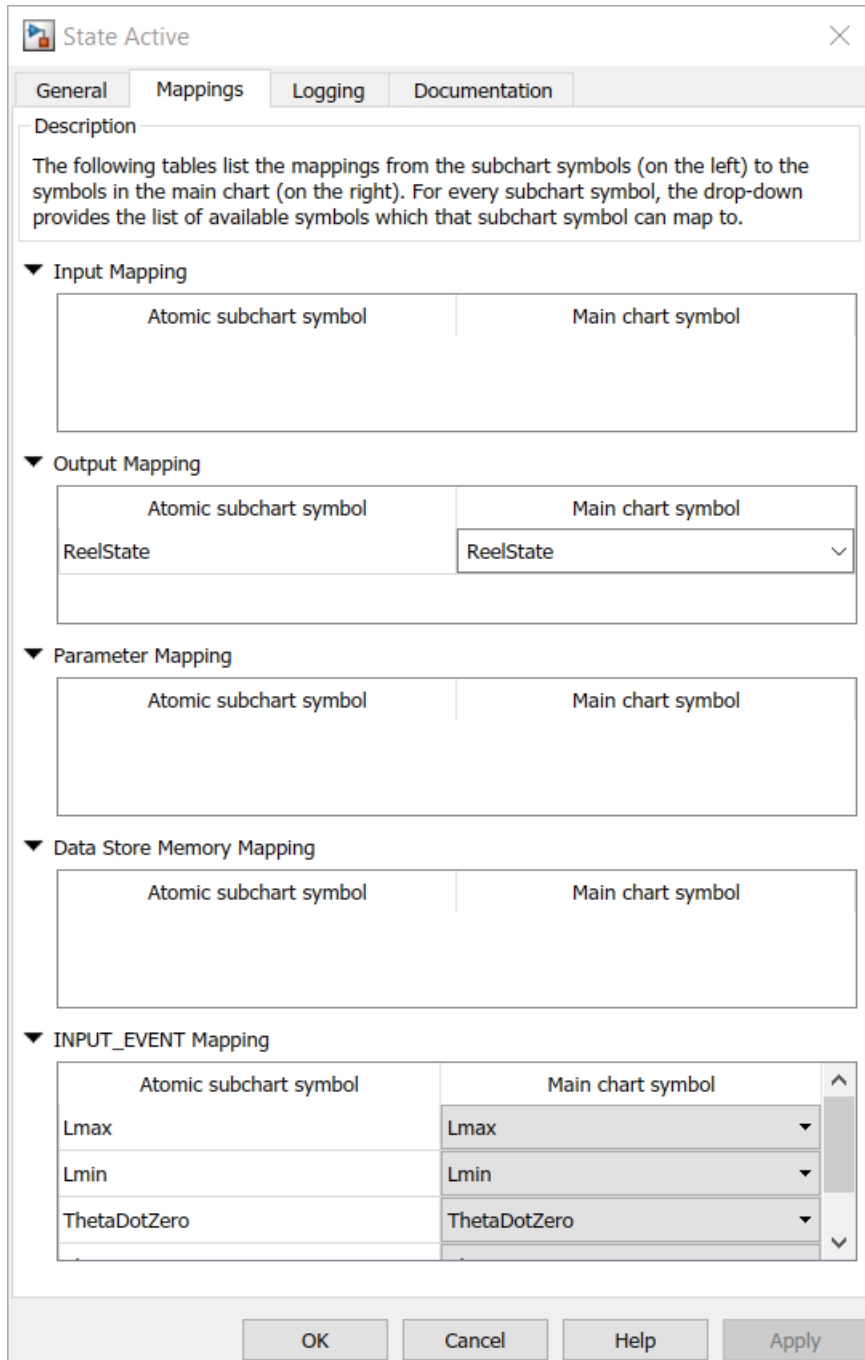
3 Click **OK**.

Disable Input Events for Atomic Subcharts

To use an instance of a library chart as an atomic subchart, but not the entire saved set of input events, you can disable input events.

1 Right-click the atomic subchart and select **Subchart Mappings**.

2 To disable an input event, under **Input Event Mapping**, select `<disabled>` in the **Main chart symbol** drop-down list.
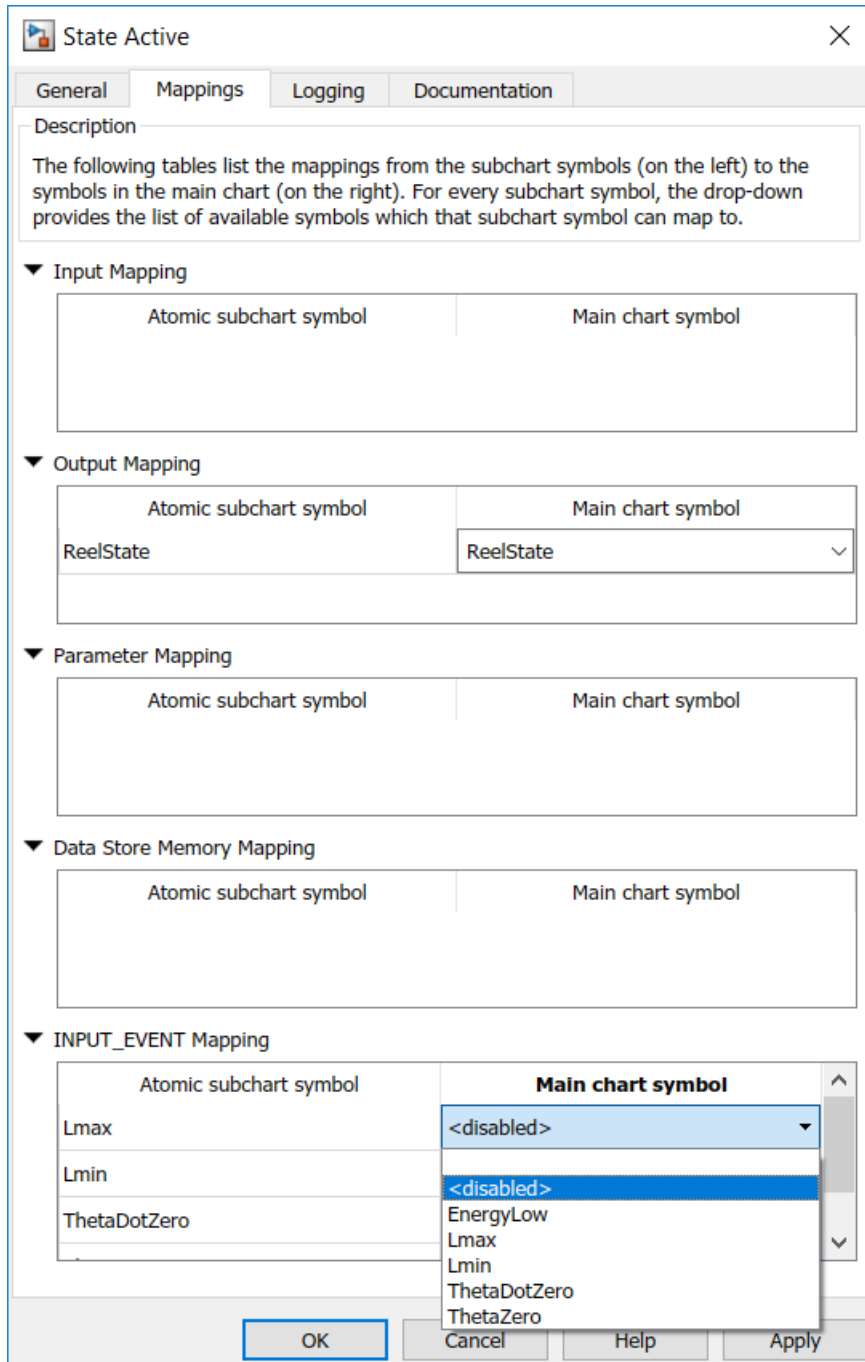
3 Click **OK**.

See Also

More About

- "Create Reusable Subcomponents by Using Atomic Subcharts" on page 15-2
- "Reuse Functions by Using Atomic Boxes" on page 8-35
- "Index and Assign Values to Stateflow Structures" on page 25-8
- "Supported Operations for Vectors and Matrices" on page 17-11

Generate Reusable Code for Atomic Subcharts

An atomic subchart is a graphical object that helps you to create standalone subcomponents in a Stateflow chart. Atomic subcharts are supported only in Stateflow charts in Simulink models. For more information, see "Create Reusable Subcomponents by Using Atomic Subcharts" on page 15-2.

How to Generate Reusable Code for Linked Atomic Subcharts

To specify code generation parameters for linked atomic subcharts from the same library:

1   Open the library model that contains your atomic subchart.

2   Unlock the library.

3   Right-click the library chart and select **Block Parameters**.

4   In the dialog box, specify the following parameters:

   a   On the **Main** tab, select **Treat as atomic unit**.

   b   On the **Code Generation** tab, set **Function packaging** to `Reusable function`.

   c   Set **File name options** to `User specified`.

   d   For **File name**, enter the name of the file with no extension.

   e   Click **OK** to apply the changes.

5   (OPTIONAL) Customize the generated function names for atomic subcharts:

   a   Open the Model Configuration Parameters dialog box.

   b   On the **Code Generation** pane, set **System target file** to `ert.tlc`.

   c   Navigate to the **Code Generation > Symbols** pane.

   d   For **Subsystem methods**, specify the format of the function names using a combination of the following tokens:

   - $R — root model name
   - $F — type of interface function for the atomic subchart
   - $N — block name
   - $H — subsystem index
   - $M — name-mangling text

15-27

    e   Click **OK** to apply the changes.

When you generate code for your model, a separate file stores the code for linked atomic subcharts from the same library.

How to Generate Reusable Code for Unlinked Atomic Subcharts

To specify code generation parameters for an unlinked atomic subchart:

1   In your chart, right-click the atomic subchart and select **Properties**.

2   In the dialog box, specify the following parameters:

    a   Set **Code generation function packaging** to `Reusable function`.

    b   Set **Code generation file name options** to `User specified`.

    c   For **Code generation file name**, enter the name of the file with no extension.

    d   Click **OK** to apply the changes.

3   (OPTIONAL) Customize the generated function names for atomic subcharts:

    a   Open the Model Configuration Parameters dialog box.

    b   On the **Code Generation** pane, set **System target file** to `ert.tlc`.

    c   Navigate to the **Code Generation > Symbols** pane.

    d   For **Subsystem methods**, specify the format of the function names using a combination of the following tokens:

- $R — root model name
- $F — type of interface function for the atomic subchart
- $N — block name
- $H — subsystem index
- $M — name-mangling text

    e   Click **OK** to apply the changes.

When you generate code for your model, a separate file stores the code for the atomic subchart. For more information, see "Generate Reusable Code for Unit Testing" on page 15-48.

Rules for Using Atomic Subcharts

An atomic subchart is a graphical object that helps you to create standalone subcomponents in a Stateflow chart. Atomic subcharts are supported only in Stateflow charts in Simulink models. For more information, see "Create Reusable Subcomponents by Using Atomic Subcharts" on page 15-2.

Define data in an atomic subchart explicitly

Be sure to define data that appears in an atomic subchart explicitly in the main chart. For instructions on how to define data in a chart, see "Add Data Through the Model Explorer" on page 9-3.

Map variables of linked atomic subcharts

When you use linked atomic subcharts, map the variables so that data in the subchart corresponds to the correct data in the main chart. Map subchart variables manually if, when you add the subchart, the variables do not have the same names as the corresponding symbols in the main chart. For more information, see "Map Variables for Atomic Subcharts and Boxes" on page 15-9.

Match size, type, and complexity of variables in linked atomic subcharts

Verify that the size, type, and complexity of variables in a subchart match the settings of the corresponding variables in the main chart. For more information, see "Map Variables for Atomic Subcharts and Boxes" on page 15-9.

Export chart-level functions if called from an atomic subchart

If your atomic subchart contains a function call to a chart-level function, export that function by selecting **Export Chart Level Functions**. Do not export graphical functions from an atomic subchart that maps variables to variables at the main chart level with a different scope. For more information, see "Export Stateflow Functions for Reuse" on page 8-22.

Do not mix edge-triggered and function-call input events in the same atomic subchart

Input events in an atomic subchart must all use edge-triggered type, or they must all use function-call type. This restriction is consistent with the behavior for the container chart. For more information, see "Best Practices for Using Events in Stateflow Charts" on page 10-4.

Do not map multiple input events in an atomic subchart to the same input event in the container chart

Each input event in an atomic subchart must map to a unique input event in the container chart. You can verify unique mappings of input events by opening the properties dialog box for the atomic subchart and checking the **Input Event Mapping** section of the **Mappings** tab.

Do not log signals from atomic subcharts that map variables with different scopes

If an atomic subchart maps variables to variables at the main chart level with a different scope, you cannot log signals for the chart.

Match the trigger type when mapping input events

Each input event in an atomic subchart must map to an input event of the same trigger type in the container chart.

Do not use atomic subcharts in continuous-time Stateflow charts

Continuous-time charts do not support atomic subcharts.

Do not use Moore charts as atomic subcharts

Moore charts do not have the same simulation behavior as Classic Stateflow charts with the same constructs.

Do not use outgoing transitions when an atomic subchart uses top-level local events

You cannot use outgoing transitions from an atomic subchart that uses local events at the top level of the subchart. Using this configuration causes a simulation error.

Avoid using execute-at-initialization with atomic subcharts

You get a warning when the following conditions are true:

- The chart property **Execute (enter) Chart At Initialization** is enabled.
- The default transition path of the chart reaches an atomic subchart.

If an `entry` action inside the atomic subchart requires access to a chart input or data store memory, you might get inaccurate results. To avoid this warning, you can disable

Execute (enter) Chart At Initialization or redirect the default transition path away from the atomic subchart.

For more information about execute-at-initialization behavior, see "Execution of a Chart at Initialization" on page 3-42.

Avoid using the names of subsystem parameters in atomic subcharts

If a parameter in an atomic subchart matches the name of a Simulink built-in subsystem parameter, the only mapping allowed for that parameter is `Inherited`. Specifying any other parameter mapping in the **Mappings** tab of the properties dialog box causes an error. You can, however, change the parameter value at the MATLAB prompt so that all instances of that parameter have the same value.

To get a list of Simulink subsystem parameters, enter:

```
param_list = sort(fieldnames(get_param('built-in/subsystem', 'ObjectParameters')));
```

Restrict use of machine-parented data

If your chart contains atomic subcharts, do not use machine-parented data with the following properties:

- Imported or exported
- Is 2-D or higher, or uses fixed-point type

Machine-parented data with these properties prevent reuse of generated code and other code optimizations.

Do not change the first index of local data to a nonzero value

When a data store memory in an atomic subchart maps to chart-level local data, the **First index** property of the local data must remain zero. If you change **First index** to a nonzero value, an error occurs when you try to update the diagram.

Use consistent settings for super-step semantics

When you use linked atomic subcharts, verify that your settings for super-step semantics match the settings in the main chart. For more information, see "Super Step Semantics" on page 3-73.

Restrictions for Converting to Atomic Subcharts

Rationale for Restrictions

Atomic subcharts facilitate the reuse of states and subcharts as standalone objects.

Data, Graphical Functions, and Events

To convert a state or subchart to an atomic subchart, access to objects not parented by the state or subchart must be one of the following:

- Chart-level data
- Chart-level graphical functions
- Input events

If the state or subchart accesses a chart-level graphical function, the chart must export that function. For more information, see "Export Stateflow Functions for Reuse" on page 8-22.

Do not export graphical functions from an atomic subchart that maps variables to variables at the main chart level with a different scope.

Event Broadcasts

The state or subchart that you want to convert to an atomic subchart cannot refer to:

- Local events that are outside the scope of that state or subchart
- Output events

The state or subchart you want to convert can refer to *input* events.

Local Data with a Nonzero First Index

The state or subchart that you want to convert to an atomic subchart cannot access local data where the **First index** property is nonzero. For the conversion process to work, the **First index** property of the local data must be zero, which is the default value.

Machine-Parented Data

The state or subchart that you want to convert to an atomic subchart cannot reside in a chart that uses machine-parented data with the following properties:

- Imported or exported
- Is 2-D or higher, or uses a fixed-point type

Machine-parented data with these properties prevent reuse of generated code and other code optimizations.

Strong Data Typing with Simulink Inputs and Outputs

To convert a state or subchart to an atomic subchart, your chart must use strong data typing with Simulink inputs and outputs.

To specify strong data typing:

1 Open the Chart properties dialog box.

2 Select **Use Strong Data Typing with Simulink I/O**. This option appears only for charts that use C as the action language.

3 Click **OK** to close the dialog box.

Supertransitions

The state or subchart that you want to convert to an atomic subchart cannot have any supertransitions crossing the boundary.

Masked Library Chart

You cannot use a masked library chart containing mask parameters as an atomic subchart.

Reuse a State Multiple Times in a Chart

An atomic subchart is a graphical object that helps you to create standalone subcomponents in a Stateflow chart. Atomic subcharts are supported only in Stateflow charts in Simulink models. For more information, see "Create Reusable Subcomponents by Using Atomic Subcharts" on page 15-2.

Goal of the Tutorial

Consider this model:
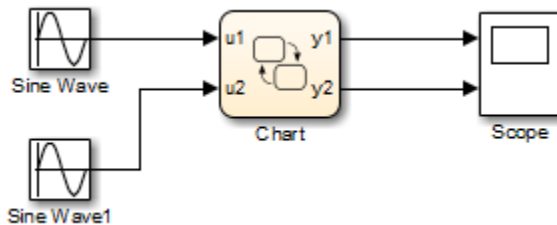
The top Sine Wave block uses a frequency of 1 radian per second, and the bottom Sine Wave block uses a frequency of 2 radians per second. The blocks use the same amplitude (1) and phase shift (0).

In the chart, each state uses saturator logic to convert the input sine wave to an output square wave of the same frequency. The states perform the same actions and differ only in the names of input and output data:
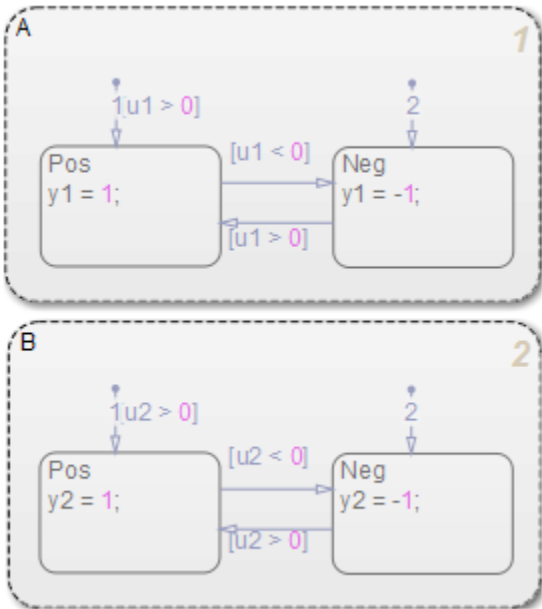
When you run the model, you get the following results:
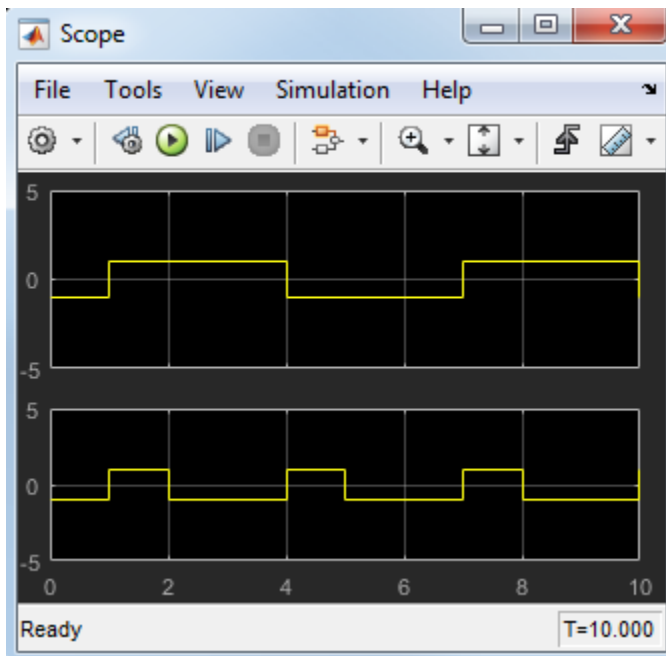
Suppose that you want to reuse the contents of state A in the chart. You can convert that state to an atomic subchart and then use multiple linked instances of that subchart in your chart.

Edit a Model to Use Atomic Subcharts

The sections that follow describe how to replace states in your chart with atomic subcharts. This procedure enables reuse of the same object in your model while retaining the same simulation results.

Step	Task	Reference
1	Change one of the states into an atomic subchart.	"Convert a State to an Atomic Subchart" on page 15-37
2	Create a library that contains this atomic subchart.	"Create a Library for the Atomic Subchart" on page 15-37

Step	Task	Reference
3	Replace the states in your chart with linked atomic subcharts.	"Replace States with Linked Atomic Subcharts" on page 15-38
4	Edit the mapping of input and output variables where necessary.	"Edit the Mapping of Input and Output Variables" on page 15-38

Convert a State to an Atomic Subchart

To convert state A to an atomic subchart, right-click the state and select **Group & Subchart** > **Atomic Subchart**. State A changes to an atomic subchart:
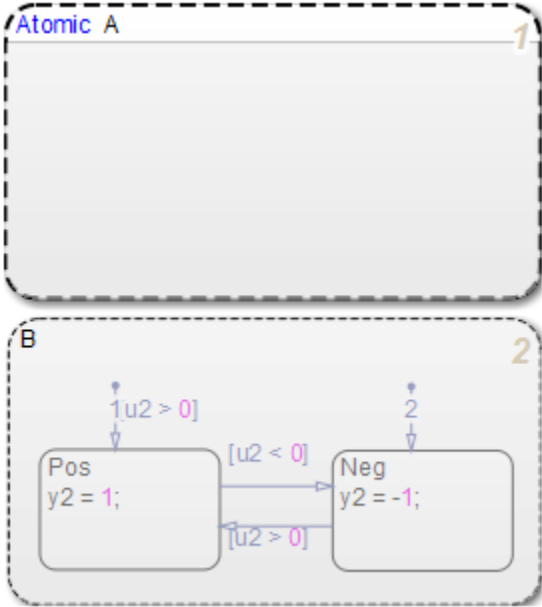
Create a Library for the Atomic Subchart

To enable reuse of the atomic subchart you created in "Convert a State to an Atomic Subchart" on page 15-37, store the atomic subchart in a library:

1. Create a new library model.
2. Copy the atomic subchart and paste in your library.

The atomic subchart appears as a standalone chart with an input and an output. This standalone property enables you to reuse the contents of the atomic subchart.
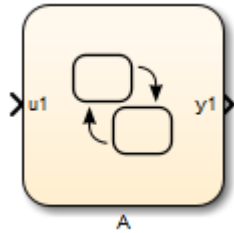
3 Save your library model.

Replace States with Linked Atomic Subcharts

To replace the states in your chart with linked atomic subcharts:

1 Delete both states from the chart.

2 Copy the atomic subchart in your library and paste in your chart twice.

3 Rename the second instance as B.

Each linked atomic subchart appears opaque and contains the label **Link** in the upper-left corner.
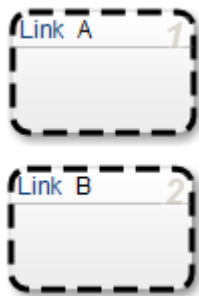
Edit the Mapping of Input and Output Variables

If you simulate the model now, the output for y2 is zero:
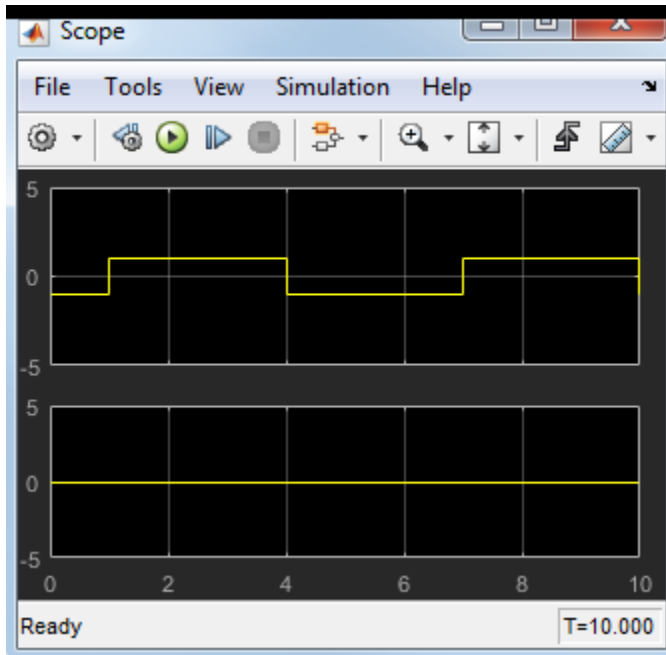
You also see warnings about unused data. These warnings appear because atomic subchart B uses u1 and y1 instead of u2 and y2. To fix these warnings, you must edit the mapping of input and output variables:

1 Open the properties dialog box for B.

2 Click the **Mappings** tab.
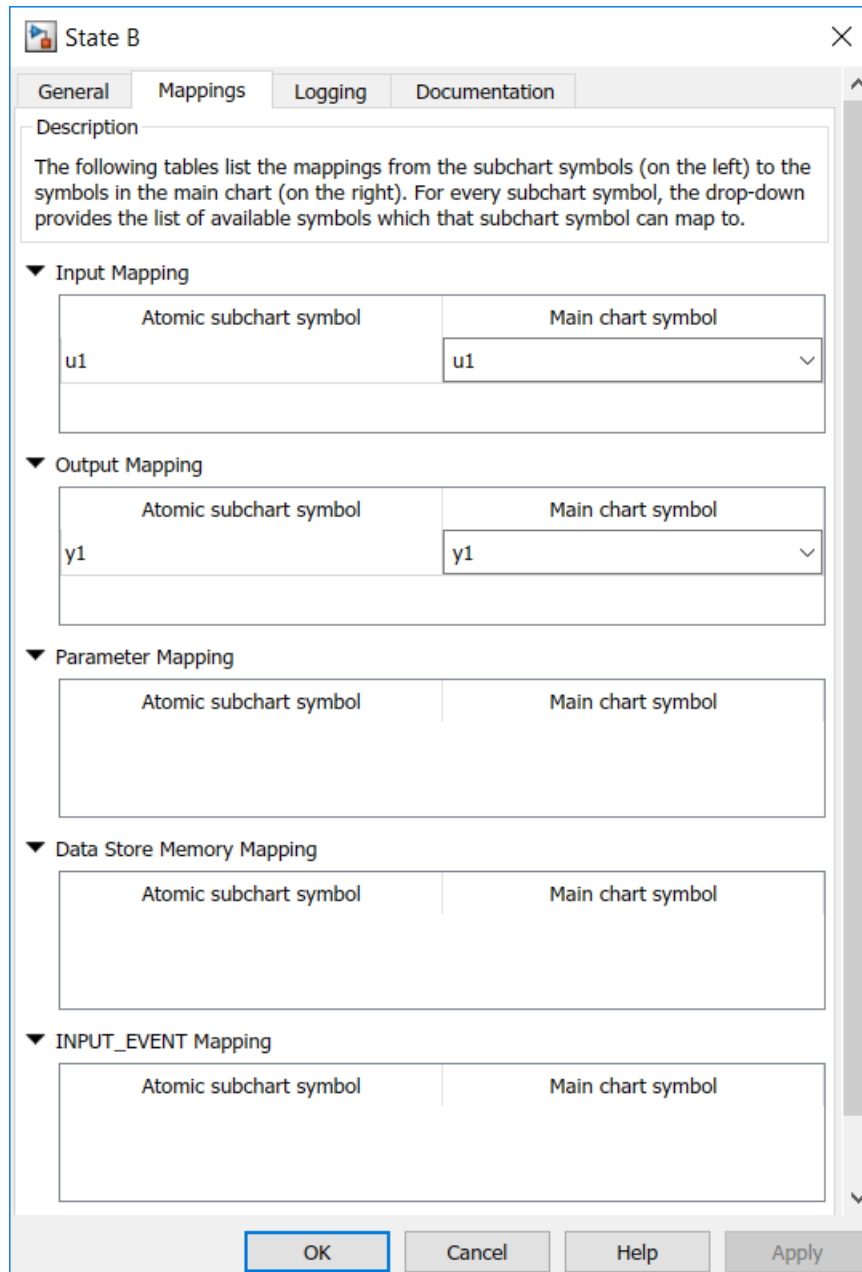
3   Under **Input Mapping**, select u2 from the drop-down list.

     The input variable in your atomic subchart now maps to the correct input variable in the main chart.

4   Under **Output Mapping**, select y2 from the drop-down list.

     The output variable in your atomic subchart now maps to the correct output variable in the main chart.

5   Click **OK**.

Run the New Model

When you simulate the new model, the results match those of the original design.
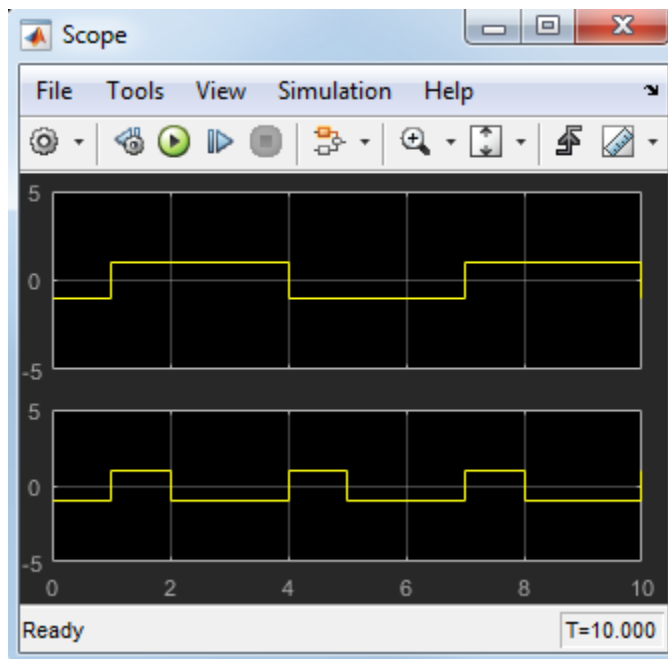
Propagate a Change in the Library Chart

Suppose that you edit the transition from Pos to Neg in the library chart:
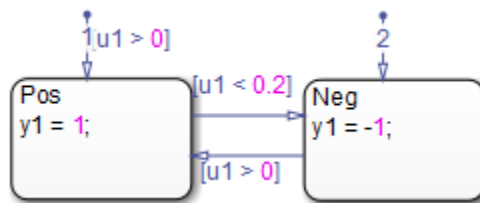
This change propagates to all linked atomic subcharts in your main chart. You do not have to update each state individually.

Reduce the Compilation Time of a Chart

An atomic subchart is a graphical object that helps you to create standalone subcomponents in a Stateflow chart. Atomic subcharts are supported only in Stateflow charts in Simulink models. For more information, see "Create Reusable Subcomponents by Using Atomic Subcharts" on page 15-2.

Goal of the Tutorial

Assume that you have the following model, and the chart has two states:
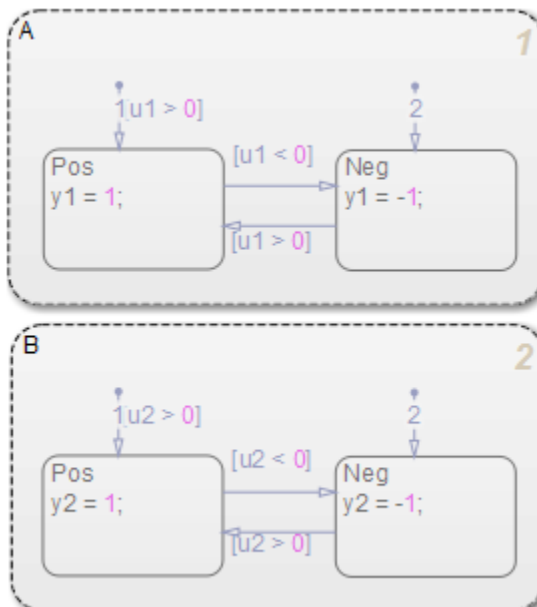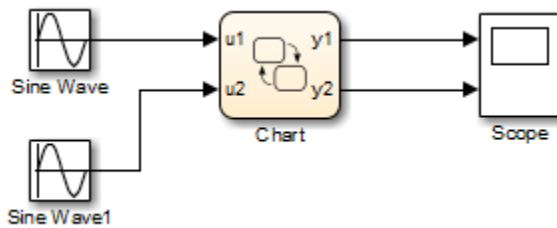
Suppose that you want to reduce the compilation time of the chart for simulation. You can convert state A to an atomic subchart. Then you can make changes, one by one, to state A and see how each change affects simulation results. Making one change requires recompilation of only the atomic subchart and not the entire chart.

Edit a Model to Use Atomic Subcharts

1   Right-click state A and select **Group & Subchart** > **Atomic Subchart**.

2   Double-click the atomic subchart.

    The contents of the subchart appear in a separate window.

3   Start simulation.

    Side-by-side animation for the main chart and the atomic subchart occurs.

4   In the atomic subchart, change the state action for Pos to y1 = 2.

5   Restart simulation.

    Recompilation occurs only for the atomic subchart and not the entire chart.

Divide a Chart into Separate Units

An atomic subchart is a graphical object that helps you to create standalone subcomponents in a Stateflow chart. Atomic subcharts are supported only in Stateflow charts in Simulink models. For more information, see "Create Reusable Subcomponents by Using Atomic Subcharts" on page 15-2.

Goal of the Tutorial

Assume that you have the following model, and the chart has two states:
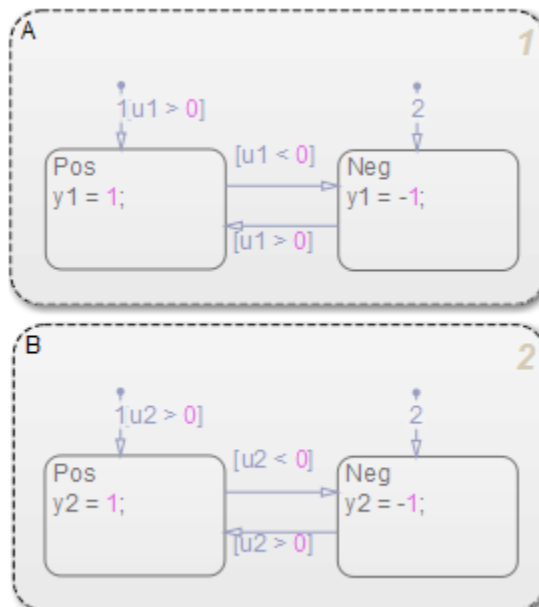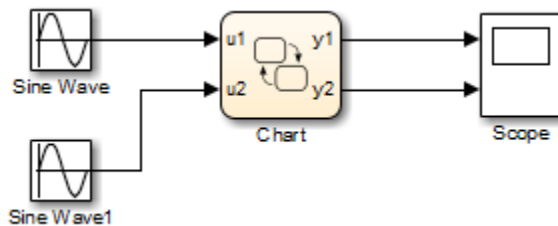
Suppose that you want to edit state A separately, while someone else is editing state B. You can convert state A to an atomic subchart for storage in a library model. After replacing state A with a linked atomic subchart, you can make changes separately in the library. These changes propagate automatically to the chart that contains the linked atomic subchart.

Edit a Model to Use Atomic Subcharts

1   Right-click state A and select **Group & Subchart** > **Atomic Subchart**.

2   Create a new library model.

3   Copy the atomic subchart and paste in your library.
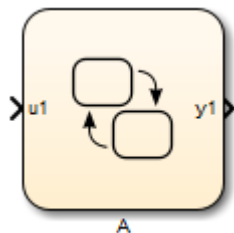
4   Save your library model.

5   In your main chart, delete state A.

6   Copy the atomic subchart in your library and paste in your main chart.
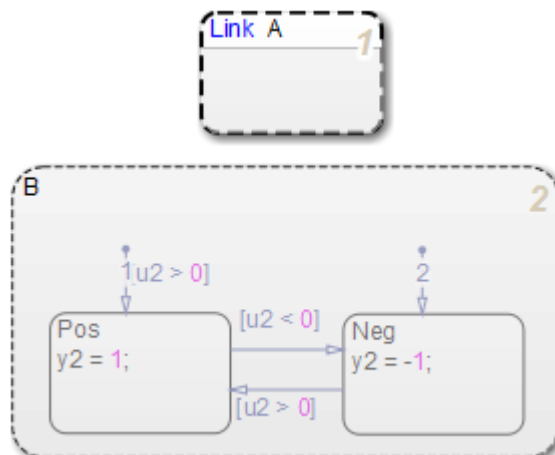
You can now edit state A separately from state B without any merge issues.

Generate Reusable Code for Unit Testing

An atomic subchart is a graphical object that helps you to create standalone subcomponents in a Stateflow chart. Atomic subcharts are supported only in Stateflow charts in Simulink models. For more information, see "Create Reusable Subcomponents by Using Atomic Subcharts" on page 15-2.

Goal of the Tutorial

Assume that you have the following model, and the chart has two states:
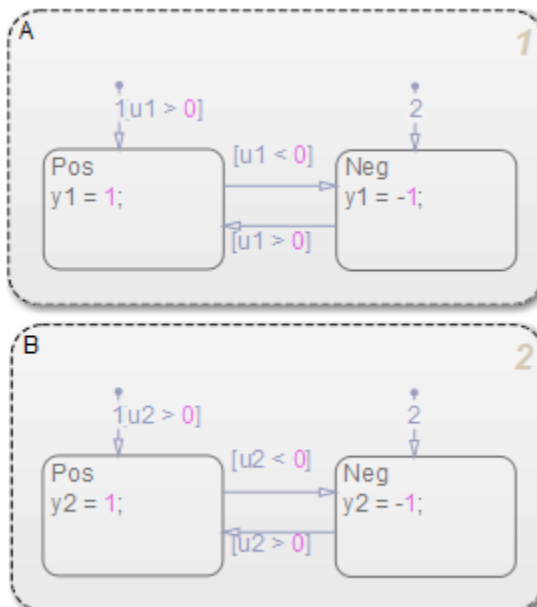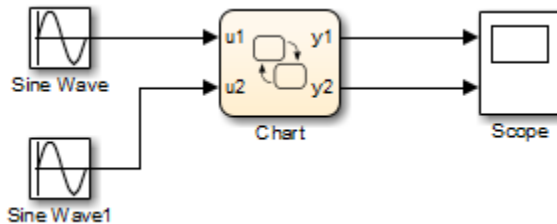
Suppose that you want to generate reusable code so that you can perform unit testing on state A. You can convert that part of the chart to an atomic subchart and then specify a separate file to store the generated code.

Convert a State to an Atomic Subchart

To convert state A to an atomic subchart, right-click the state and select **Group & Subchart** > **Atomic Subchart**. State A changes to an atomic subchart:
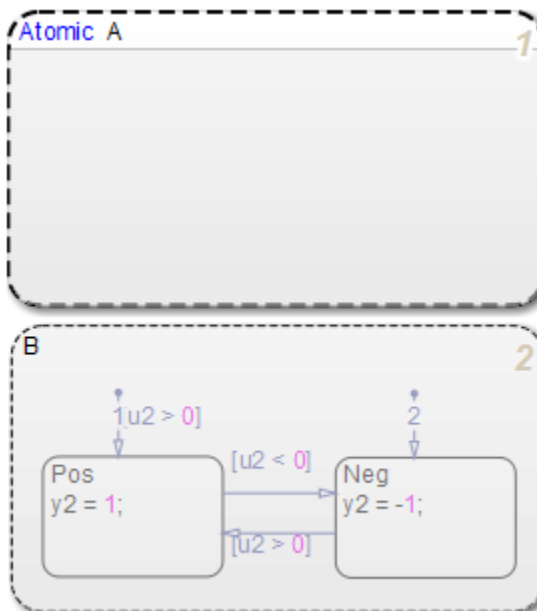
Specify Code Generation Parameters

Set Up a Standalone C File for the Atomic Subchart

1   Open the properties dialog box for A.

2   Set **Code generation function packaging** to `Reusable function`.

3   Set **Code generation file name options** to `User specified`.

4   For **Code generation file name**, enter `saturator` as the name of the file.

5 Click **OK**.

Set Up the Code Generation Report

1 Open the Model Configuration Parameters dialog box.

2 In the **Code Generation** pane, set **System target file** to `ert.tlc`.

3 In the **Code Generation > Report** pane, select **Create code generation report**.

This step automatically selects **Open report automatically** and **Code-to-model**.

4 Select **Model-to-code**.

5 Click **Apply**.

Customize the Generated Function Names

1 In the Model Configuration Parameters dialog box, go to the **Code Generation > Symbols** pane.

2 Set **Subsystem methods** to the format scheme RNMF, where:

- $R is the root model name.
- $N is the block name.
- $M is the mangle token.
- $F is the type of interface function for the atomic subchart.

For more information, see "Subsystem methods" (Simulink Coder).

3 Click **Apply**.

Generate Code for Only the Atomic Subchart

To generate code for your model, press **Ctrl+B**. In the code generation report that appears, you see a separate file that contains the generated code for the atomic subchart.

To inspect the code for `saturator.c`, click the hyperlink in the report to see the following code:

Line 28 shows that the `during` function generated for the atomic subchart has the name `ex_reuse_states_A_during`. This name follows the format scheme RNMF specified for **Subsystem methods**:

- $R is the root model name, `ex_reuse_states`.
- $N is the block name, A.
- $M is the mangle token, which is empty.
- $F is the type of interface function for the atomic subchart, `during`.

Note The line numbers shown can differ from the numbers that appear in your code generation report.

16

Save and Restore Simulations with Operating Point

Using Operating Points in Stateflow

An operating point is a snapshot of the complete state of a Simulink model at a specific time during simulation. For a Stateflow chart, an operating point includes:

- Activity of chart states
- Values of chart local data
- Values of chart output data
- Values of persistent data in MATLAB functions and Truth Table blocks

An operating point lists Stateflow objects in hierarchical order:

- Graphical objects grouped by type (box, function, or state) and in alphabetical order within each group
- Chart data grouped by scope (block output or local) and in alphabetical order within each group

For example, the following operating point illustrates the hierarchical structure of chartStateflow objects.

```
c =

  Block:    "shift_logic"    (handle)     (active)
  Path:     sf_car/shift_logic

  Contains:

    + gear_state         "State (AND)"          (active)
    + selection_state    "State (AND)"          (active)
      gear               "State output data"    gearType [1, 1]
      down_th            "Local scope data"     double [1, 1]
      up_th              "Local scope data"     double [1, 1]
```

The tree structure maps graphical and nongraphical objects to their respective locations in the chart hierarchy. If name conflicts exist, one or more underscores appear at the end of a name so that all objects have unique identifiers in the operating point hierarchy. Stateless flow charts have an empty operating point, because they do not contain states or persistent data.

For information about using an operating point for other blocks in a Simulink model, see "Save and Restore Simulation Operating Point" (Simulink).

Division of a Long Simulation into Segments

You can save the complete operating point of a model at any time during a long simulation. Then you can load that operating point and run specific segments of that simulation without starting from time t = 0, which saves time.

For directions, see "Divide a Long Simulation into Segments" on page 16-4.

Test of a Chart Response to Different Settings

You can modify and load the operating point of a chart to test the response of a chart to different settings. You can change the value of chart local or output data midway through a simulation or change state activity and then test how a chart responds.

Loading and modifying the operating point provides these benefits:

- Enables testing of a hard-to-reach chart configuration by loading a specific operating point
- Enables testing of the same chart configuration with different settings

For directions, see:

- "Test a Unique Chart Configuration" on page 16-9
- "Test a Chart with Fault Detection and Redundant Logic" on page 16-18

See Also

More About

Divide a Long Simulation into Segments

An operating point is a snapshot of the state of a Simulink model at a specific time during simulation. For a Stateflow chart, an operating point includes:

- Activity of chart states
- Values of chart local data
- Values of chart output data
- Values of persistent data in MATLAB functions and Truth Table blocks

For more information, see "Using Operating Points in Stateflow" on page 16-2.

Goal of the Tutorial

Suppose that you want to simulate the `sf_boiler` model without starting from t = 0.
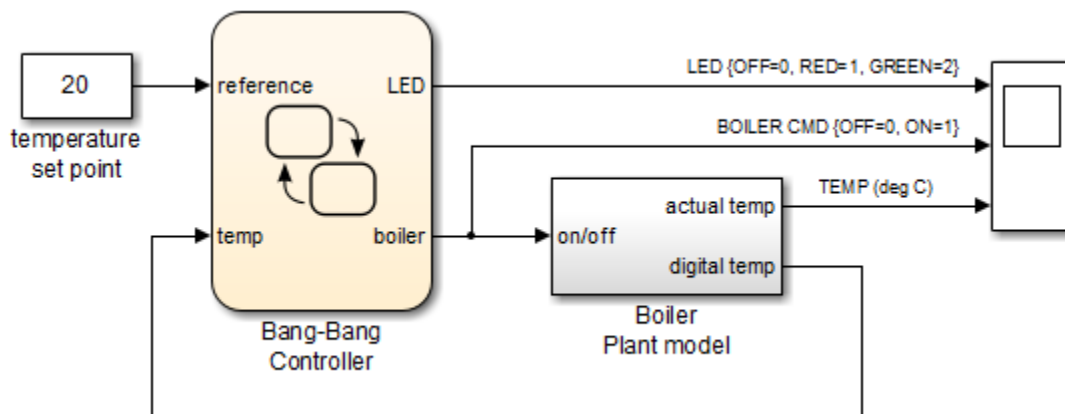
This model simulates for 1400 seconds, but the output that interests you occurs sometime between t = 400 and 600. You can simulate the model, save the operating point at time t = 400, and then load that operating point for simulation between t = 400 and 600.

Step	Task	Reference
1	Define the operating point for your chart.	"Define the Operating Point" on page 16-5

Step	Task	Reference
2	Load the operating point for your chart.	"Load the Operating Point" on page 16-6
3	Simulate the specific segment.	"Simulate the Specific Segment" on page 16-7

Define the Operating Point

1 Open the sf_boiler model.

2 Enable saving of an operating point.

  a Open the Model Configuration Parameters dialog box and go to the **Data Import/Export** pane.

  b Select the **Final states** check box.

  c Enter a name, such as sf_boiler_ctx01.

  d Select the **Save final operating point** check box.

  e Click **Apply**.

Programmatic equivalent

You can programmatically enable saving of an operating point:

```
set_param('sf_boiler','SaveFinalState','on', ...
'FinalStateName', ['sf_boiler_ctx01'], ...
'SaveOperatingPoint','on');
```

For details about setting model parameters, see set_param.

3 Define the start and stop times for this simulation segment.

  a In the Model Configuration Parameters dialog box, go to the **Solver** pane.

  b For **Start time**, enter 0.

  c For **Stop time**, enter 400.

  d Click **OK**.

Programmatic equivalent

You can programmatically set the start and stop times:

```
set_param('sf_boiler','StartTime','0', ...
'StopTime','400');
```

4   Start simulation.

When you simulate the model, you save the complete operating point at t = 400 in the variable `sf_boiler_ctx01` in the MATLAB base workspace.

5   Disable saving of an operating point.

This step prevents you from overwriting the operating point you saved in the previous step.

a   Open the Model Configuration Parameters dialog box and go to the **Data Import/Export** pane.

b   Clear the **Save final operating point** check box.

c   Clear the **Final states** check box.

d   Click **OK**.

Programmatic equivalent

You can programmatically disable saving of an operating point:

```
set_param('sf_boiler','SaveOperatingPoint','off', ...
'SaveFinalState','off');
```

Load the Operating Point

1   Enable loading of an operating point.

a   Open the Model Configuration Parameters dialog box and go to the **Data Import/Export** pane.

b   Select the **Initial state** check box.

c   Enter the variable that contains the operating point of your chart: `sf_boiler_ctx01`.

d   Click **Apply**.

Programmatic equivalent

You can programmatically enable loading of an operating point:

```
set_param('sf_boiler','LoadInitialState','on', ...
'InitialState', ['sf_boiler_ctx01']);
```

2   Define the new stop time for this simulation segment.

    a   In the Model Configuration Parameters dialog box, go to the **Solver** pane.

    b   For **Stop time**, enter 600.

    c   Click **OK**.

You do not need to enter a new start time because the simulation continues from where it left off.

Programmatic equivalent

You can programmatically set the new stop time:

```
set_param('sf_boiler','StopTime','600');
```

Simulate the Specific Segment

When you simulate the model, the following output appears in the Scope block.
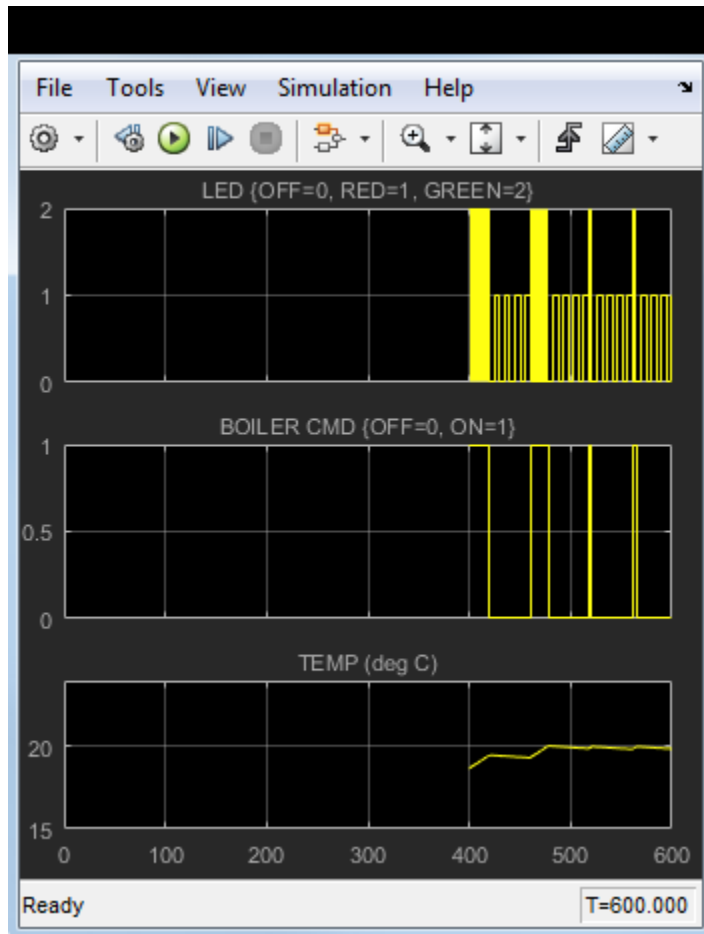
See Also

More About

- "Using Operating Points in Stateflow" on page 16-2
- "Methods for Interacting with the Operating Point of a Chart" on page 16-33
- "Best Practices for Saving the Operating Point of a Chart" on page 16-41

Test a Unique Chart Configuration

An operating point is a snapshot of the state of a Simulink model at a specific time during simulation. For a Stateflow chart, an operating point includes:

- Activity of chart states
- Values of chart local data
- Values of chart output data
- Values of persistent data in MATLAB functions and Truth Table blocks

For more information, see "Using Operating Points in Stateflow" on page 16-2.

Goal of the Tutorial

Suppose that you want to test the response of the `old_sf_car` model to a sudden change in value for `gear`.
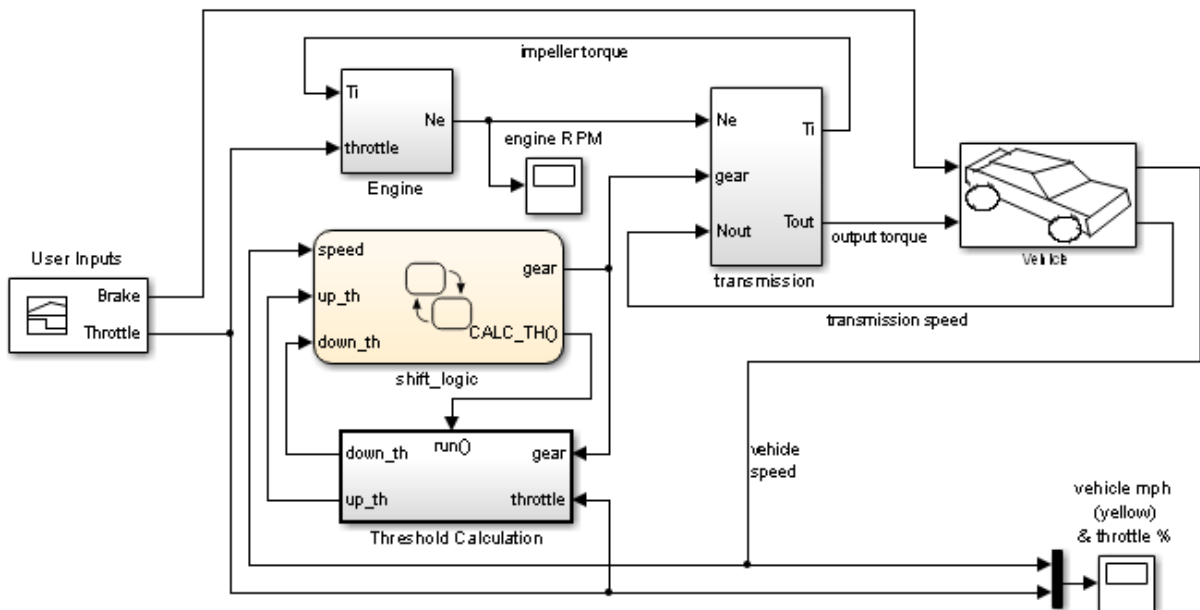
This model simulates for 30 seconds, but you want to see what happens when the value of gear changes at t = 10. You can simulate the model, save the operating point at t = 10, load and modify the operating point, and then simulate again between t = 10 and 20.

Step	Task	Reference
1	Define the operating point for your chart.	"Define the Operating Point" on page 16-10
2	Load the operating point and modify values.	"Load the Operating Point and Modify Values" on page 16-12
3	Test the modified operating point by running the model.	"Test the Modified Operating Point" on page 16-16

Define the Operating Point

1 Open the model old_sf_car.

2 Enable saving of an operating point.

    a Open the Model Configuration Parameters dialog box and go to the **Data Import/Export** pane.

    b Select the **Final states** check box.

    c Enter a name, such as old_sf_car_ctx01.

    d Select the **Save final operating point** check box.

    e Click **Apply**.

Programmatic equivalent

You can programmatically enable saving of an operating point:

```
set_param('old_sf_car','SaveFinalState','on', ...
'FinalStateName', 'old_sf_car_ctx01', ...
'SaveOperatingPoint','on');
```

For details about setting model parameters, see set_param.

3 Define the start and stop times for this simulation segment.

    a In the Model Configuration Parameters dialog box, go to the **Solver** pane.

    b For **Start time**, enter 0.

    c   For **Stop time**, enter 10.

    d   Click **OK**.

Programmatic equivalent

You can programmatically set the start and stop times:

```
set_param('old_sf_car','StartTime','0', ...
'StopTime','10');
```

4   Start simulation.

When you simulate the model, you save the complete operating point at t = 10 in the variable `old_sf_car_ctx01` in the MATLAB base workspace.

At t = 10, the engine is operating at a steady-state value of 2500 RPM.
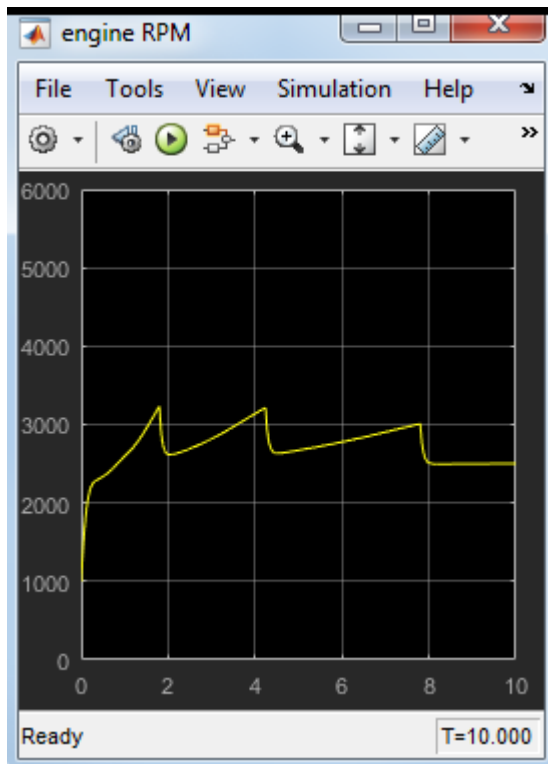
5 Disable saving of an operating point.

This step prevents you from overwriting the operating point you saved in the previous step.

a Open the Model Configuration Parameters dialog box and go to the **Data Import/Export** pane.

b Clear the **Save final operating point** check box.

c Clear the **Final states** check box.

d Click **OK**.

Programmatic equivalent

You can programmatically disable saving of an operating point:

```
set_param('old_sf_car','SaveOperatingPoint','off', ...
'SaveFinalState','off');
```

Load the Operating Point and Modify Values

1 Enable loading of an operating point.

a Open the Model Configuration Parameters dialog box and go to the **Data Import/Export** pane.

b Select the **Initial state** check box.

c Enter the variable that contains the operating point of your chart: old_sf_car_ctx01.

d Click **OK**.

Programmatic equivalent

You can programmatically enable loading of an operating point:

```
set_param('old_sf_car','LoadInitialState','on', ...
'InitialState', 'old_sf_car_ctx01');
```

2 Define an object handle for the operating point values of the shift_logic chart.

At the command prompt, type:

```
blockpath = 'old_sf_car/shift_logic';
c = old_sf_car_ctx01.get(modelOperatingPoint, 'blockpath');
```

Tip If the chart appears highlighted in the model window, you can specify the block path using `gcb`:

```
c = old_sf_car_ctx01.get(gcb);
```

Copy and reference operating points with the `get` method

The `get` method:

- Makes a copy of the operating point of your chart, which is stored in the final state data of the model.
- Provides a root-level handle or *reference* to the copy of the operating point, which is a hierarchical tree of graphical and nongraphical chart objects.

  Each node in this tree is also a handle to a state, data, or other chart object.

Note Because the entire tree consists of object handles, the following assignment statements do not work:

- `stateCopy = c.state`
- `dataCopy = c.data`
- `operatingPointCopy = c`

These assignments create copies of the object handles, not operating point values. The only way to copy operating point values is to use the `clone` method. For details, see "Methods for Interacting with the Operating Point of a Chart" on page 16-33 and "Rules for Using the Operating Point of a Chart" on page 16-38.

---

3 Look at the contents of the operating point.

```
c =

  Block:    "shift_logic"    (handle)    (active)
  Path:     old_sf_car/shift_logic

  Contains:

    + gear_state        "State (AND)"        (active)
    + selection_state   "State (AND)"        (active)
      gear              "Block output data"  double [1, 1]
```

The operating point of your chart contains a list of states and data in hierarchical order.

4 Highlight the states that are active in your chart at t = 10.

At the command prompt, type:

```
c.highlightActiveStates;
```

In the chart, all active states appear highlighted.
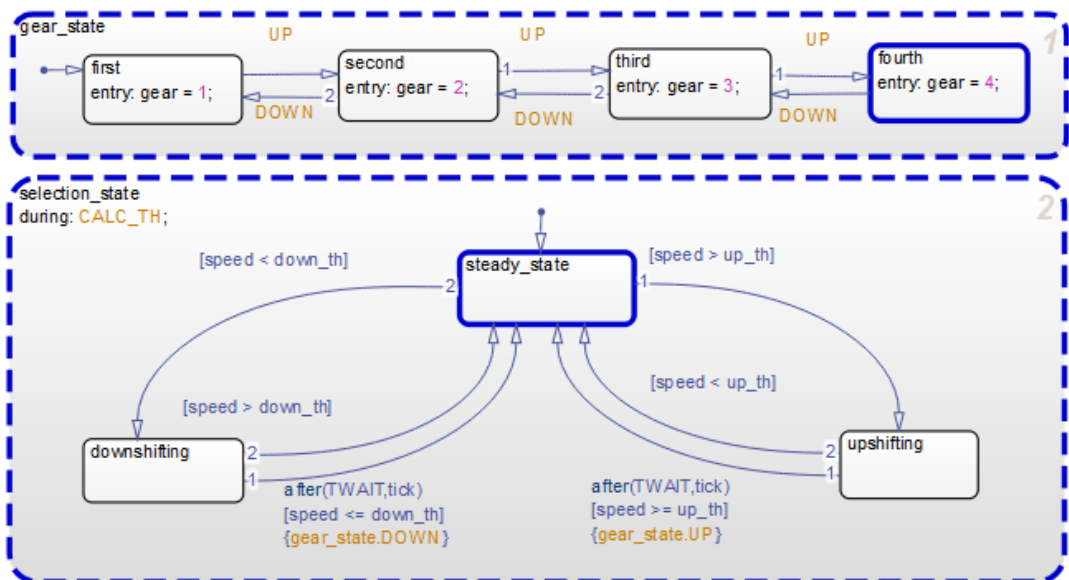
Tip To check if a single state is active, you can use the `isActive` method. For example, type:

```
c.gear_state.fourth.isActive
```

This command returns true (1) when a state is active and false (0) otherwise. For information on other methods, see "Methods for Interacting with the Operating Point of a Chart" on page 16-33.

5 Change the active substate of `selection_state` to `downshifting`.

Use this command:

```
c.selection_state.downshifting.setActive;
```

The newly active substate appears highlighted in the chart.
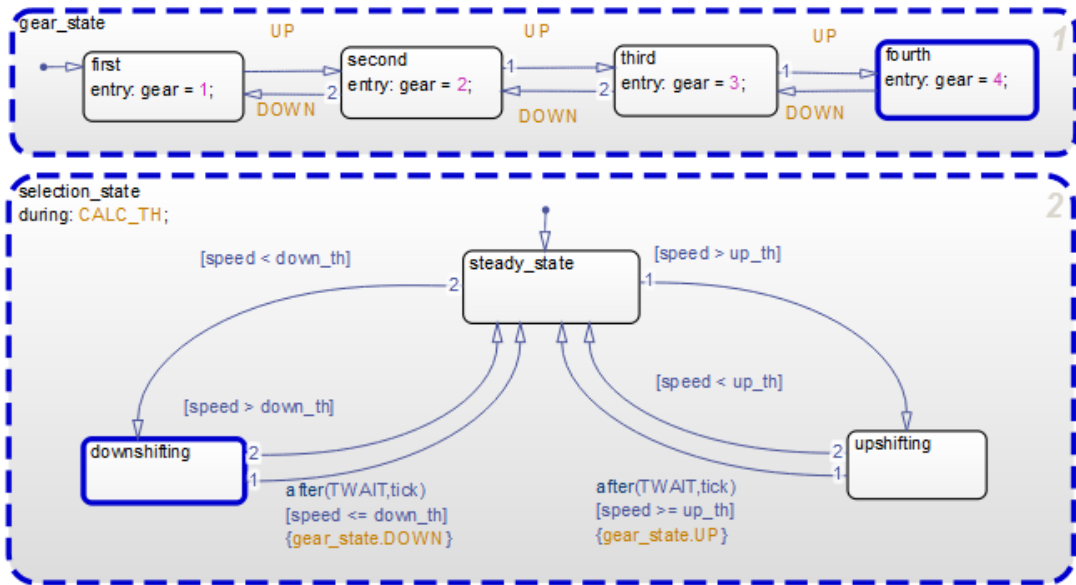
6   Change the value of output data `gear`.

When you type `c.gear` at the command prompt, you see a list of data properties similar to this:

```
>> c.gear

ans =

      Description: 'Block output data'
         DataType: 'double'
             Size: '[1, 1]'
            Range: [1x1 struct]
     InitialValue: [1x0 double]
            Value: 4
```

You can change the value of `gear` from 4 to 1 by typing

```
c.gear.Value = 1;
```

However, you cannot change the data type or size of `gear`. Also, you cannot specify a new value that falls outside the range set by the **Minimum** and **Maximum** parameters. For details, see "Rules for Modifying Data Values" on page 16-38 .

7   Save the modified operating point.

Use this command:

```
old_sf_car_ctx01 = old_sf_car_ctx01.set(blockpath, c);
```

Test the Modified Operating Point

1   Define the new stop time for the simulation segment to test.

   a   In the Model Configuration Parameters dialog box, go to the **Solver** pane.

   b   For **Stop time**, enter 20.

   c   Click **OK**.

You do not need to enter a new start time because the simulation continues from where it left off.

Programmatic equivalent

You can programmatically set the stop time:

```
set_param('old_sf_car','StopTime','20');
```

2   Start simulation.

The engine reacts as follows:
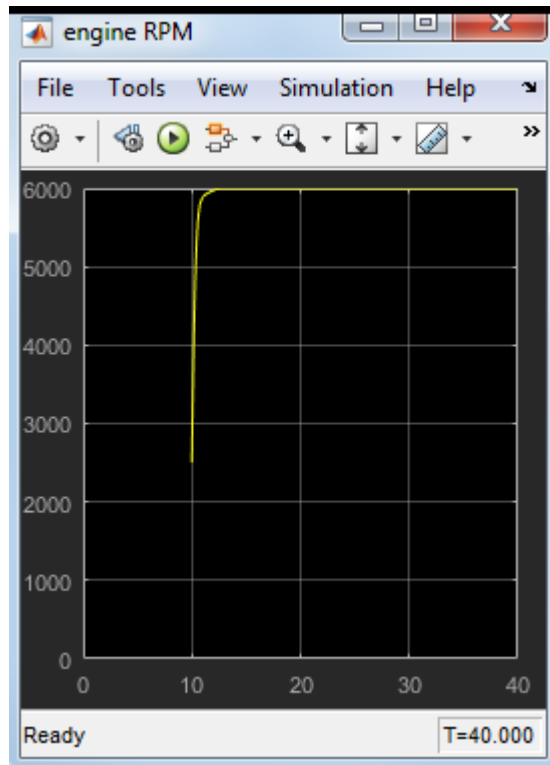
See Also

More About

- "Using Operating Points in Stateflow" on page 16-2
- "Methods for Interacting with the Operating Point of a Chart" on page 16-33
- "Best Practices for Saving the Operating Point of a Chart" on page 16-41

Test a Chart with Fault Detection and Redundant Logic

An operating point is a snapshot of the state of a Simulink model at a specific time during simulation. For a Stateflow chart, an operating point includes:

- Activity of chart states
- Values of chart local data
- Values of chart output data
- Values of persistent data in MATLAB functions and Truth Table blocks

For more information, see "Using Operating Points in Stateflow" on page 16-2.

Goal of the Tutorial

Suppose that you want to test the response of the `sf_aircraft` model to one or more actuator failures in an elevator system. For details of how this model works, see "Fault Detection Control Logic in an Aircraft Elevator Control System".
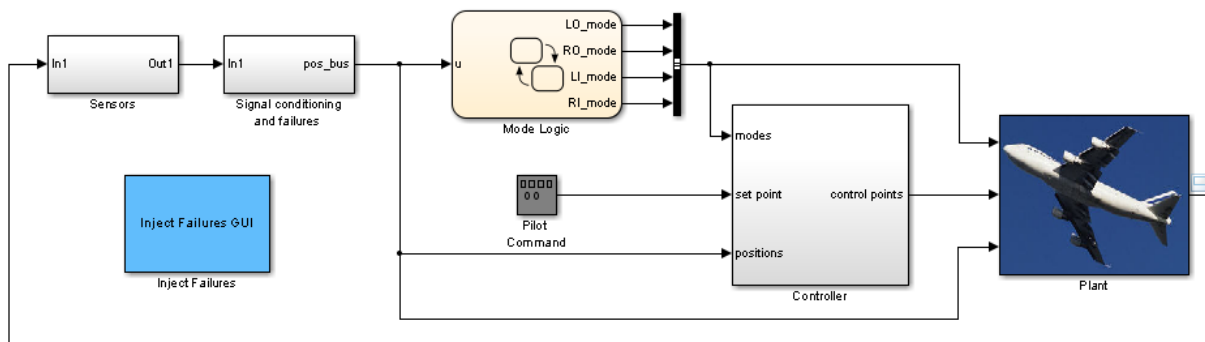
The Mode Logic chart monitors the status of actuators for two elevators. Each elevator has an outer (primary) actuator and an inner (secondary) actuator. In normal operation, the outer actuators are active and the inner actuators are on standby.
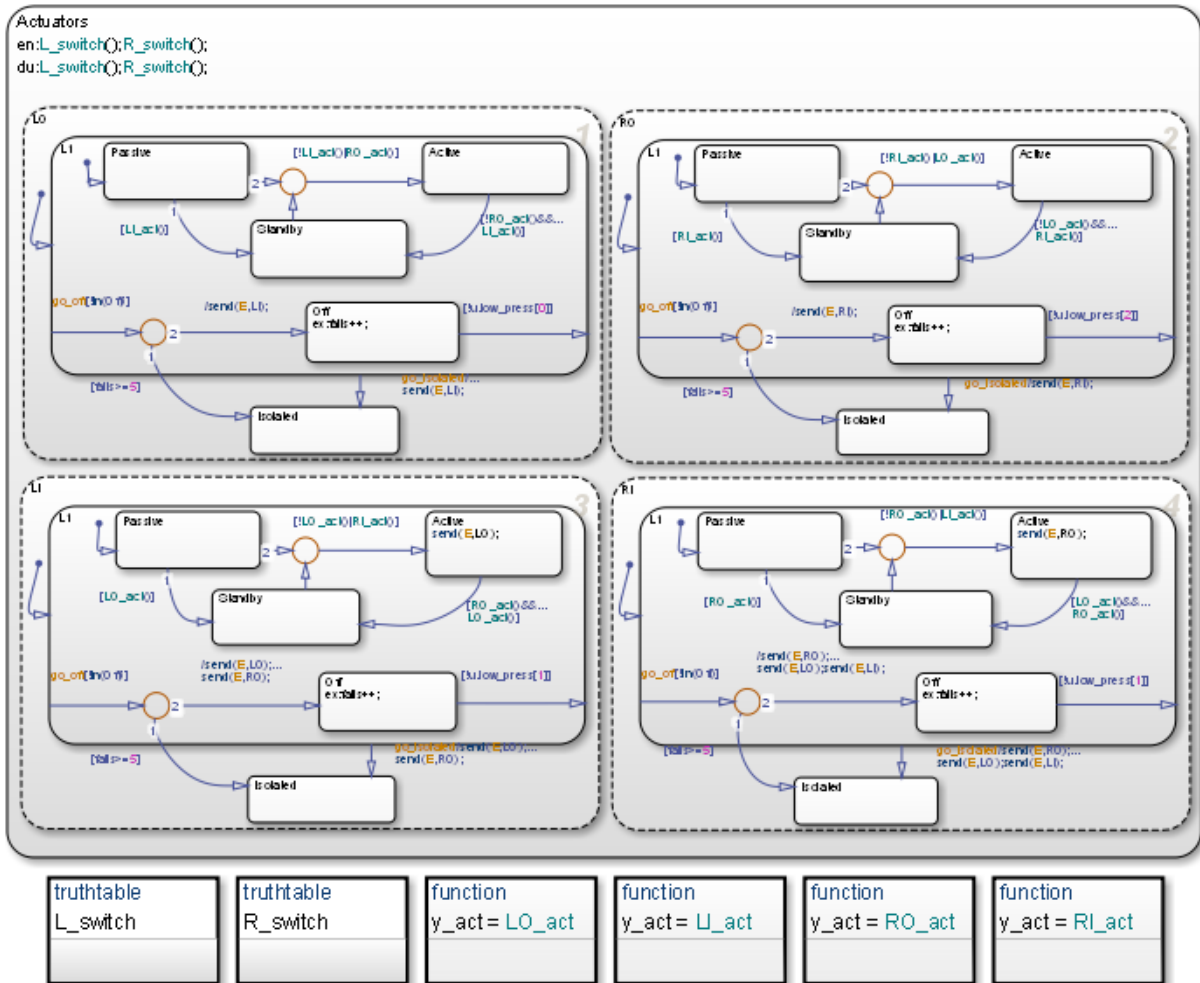
When the four actuators are working correctly, the left and right elevators reach steady-state positions in 3 seconds.
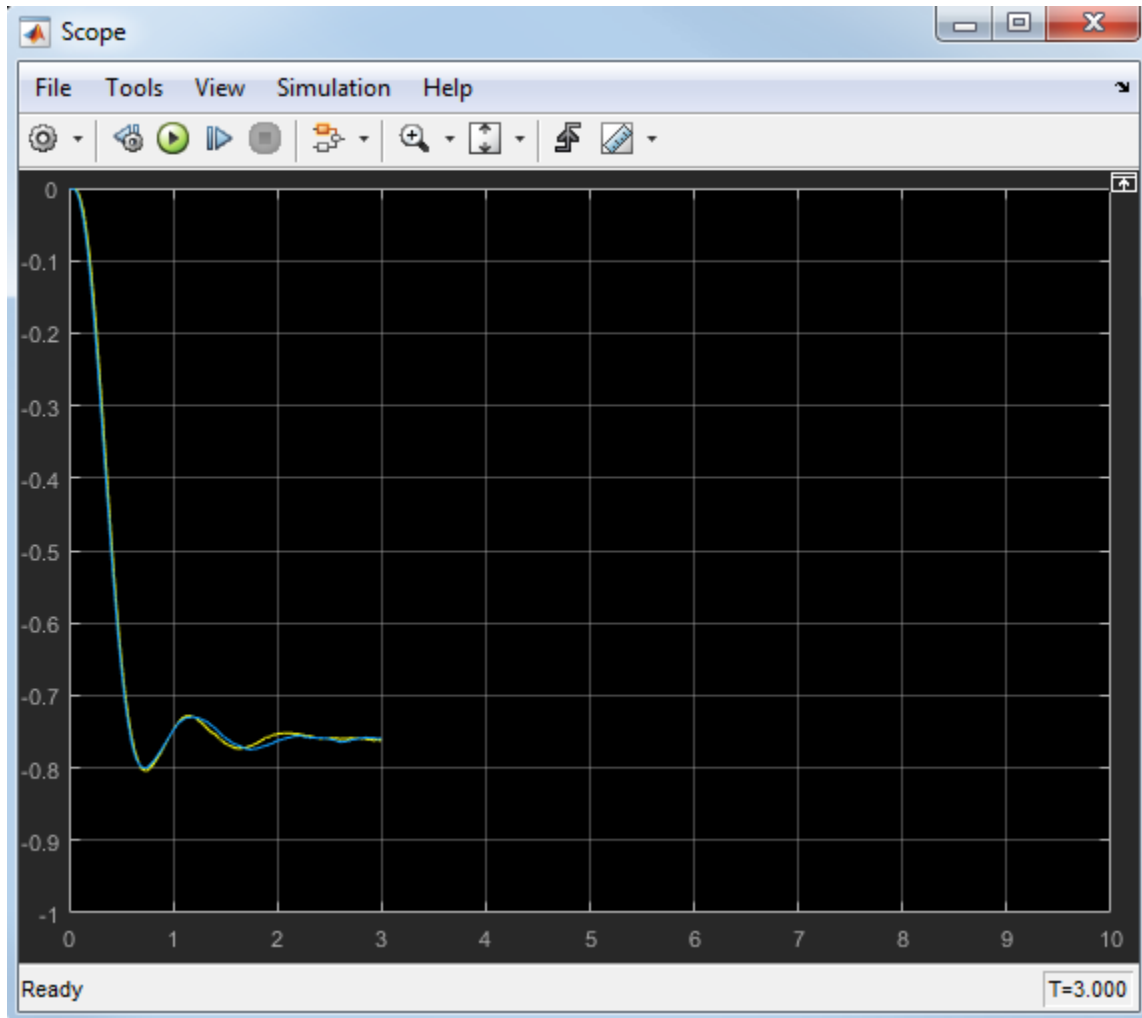
Suppose that you want to see what happens at t = 3 when at least one actuator fails. You can simulate the model, save the operating point at t = 3, load and modify the operating point, and then simulate again between t = 3 and 10.

Step	Task	Reference
1	Define the operating point for your chart.	"Define the Operating Point" on page 16-21
2	Load the operating point and modify values for one actuator failure.	"Modify Operating Point Values for One Actuator Failure" on page 16-22
3	Test the modified operating point by running the model.	"Test the Operating Point for One Failure" on page 16-26
4	Modify operating point values for two actuator failures.	"Modify Operating Point Values for Two Actuator Failures" on page 16-29
5	Test the modified operating point by running the model again.	"Test the Operating Point for Two Failures" on page 16-30

Define the Operating Point

1   Open the sf_aircraft model.

2   Enable saving of an operating point.

   a   Open the Model Configuration Parameters dialog box and go to the **Data Import/Export** pane.

   b   Select the **Final states** check box.

   c   Enter a name, such as xFinal.

   d   Select the **Save final operating point** check box.

   e   Click **Apply**.

   Programmatic equivalent

   You can programmatically enable saving of an operating point:

   ```
   set_param('sf_aircraft','SaveFinalState','on', ...
   'FinalStateName', ['xFinal'], ...
   'SaveOperatingPoint','on');
   ```

   For details about setting model parameters, see set_param.

3   Define the stop time for this simulation segment.

   a   In the Model Configuration Parameters dialog box, go to the **Solver** pane.

b For **Stop time**, enter 3.

c Click **OK**.

Programmatic equivalent

You can programmatically set the stop time:

```
set_param('sf_aircraft','StopTime','3');
```

4 Start simulation.

When you simulate the model, you save the complete operating point at t = 3 in the variable xFinal in the MATLAB base workspace.

5 Disable saving of an operating point.

This step prevents you from overwriting the operating point you saved in the previous step.

a Open the Model Configuration Parameters dialog box and go to the **Data Import/Export** pane.

b Clear the **Save final operating point** check box.

c Clear the **Final states** check box.

d Click **OK**.

Programmatic equivalent

You can programmatically disable saving of an operating point:

```
set_param('sf_aircraft','SaveOperatingPoint','off', ...
'SaveFinalState','off');
```

Modify Operating Point Values for One Actuator Failure

1 Enable loading of an operating point.

a Open the Model Configuration Parameters dialog box and go to the **Data Import/Export** pane.

b Select the **Initial state** check box.

c Enter the variable that contains the operating point of your chart: xFinal.

d Click **OK**.

Programmatic equivalent

You can programmatically enable loading of an operating point:

```
set_param('sf_aircraft','LoadInitialState','on', ...
'InitialState', ['xFinal']);
```

2 Define an object handle for the operating point values of the Mode Logic chart.

At the command prompt, type:

```
blockpath = 'sf_aircraft/Mode Logic';
c = xFinal.get(blockpath);
```

Tip If the chart appears highlighted in the model window, you can specify the block path using `gcb`:

```
c = xFinal.get(gcb);
```

Use the get method for Operating Points

The `get` method:

- Makes a copy of the operating point of your chart, which is stored in the final state data of the model.
- Provides a root-level handle or *reference* to the copy of the operating point, which is a hierarchical tree of graphical and nongraphical chart objects.

  Each node in this tree is also a handle to a state, data, or other chart object.

Note Because the entire tree consists of object handles, the following assignment statements do not work:

- `stateCopy = c.state`
- `dataCopy = c.data`
- `operatingPointCopy = c`

These assignments create copies of the object handles, not operating point values. The only way to copy operating point values is to use the `clone` method. For details,

16-23

3 Look at the contents of the operating point.

```
c =

  Block:    "Mode Logic"     (handle)     (active)
  Path:     sf_aircraft/Mode Logic

  Contains:

    + Actuators          "State (OR)"               (active)
    + LI_act             "Function"
    + LO_act             "Function"
    + L_switch           "Function"
    + RI_act             "Function"
    + RO_act             "Function"
    + R_switch           "Function"
    + LI_mode            "State output data"       sf_aircraft_ModeType [1,1]
    + LO_mode            "State output data"       sf_aircraft_ModeType [1,1]
    + RI_mode            "State output data"       sf_aircraft_ModeType [1,1]
    + RO_mode            "State output data"       sf_aircraft_ModeType [1,1]
```

The operating point of your chart contains a list of states, functions, and data in hierarchical order.

4 Highlight the states that are active in your chart at t = 3.

At the command prompt, type:

```
c.highlightActiveStates;
```

Active states appear highlighted. By default, the two outer actuators are active and the two inner actuators are on standby.
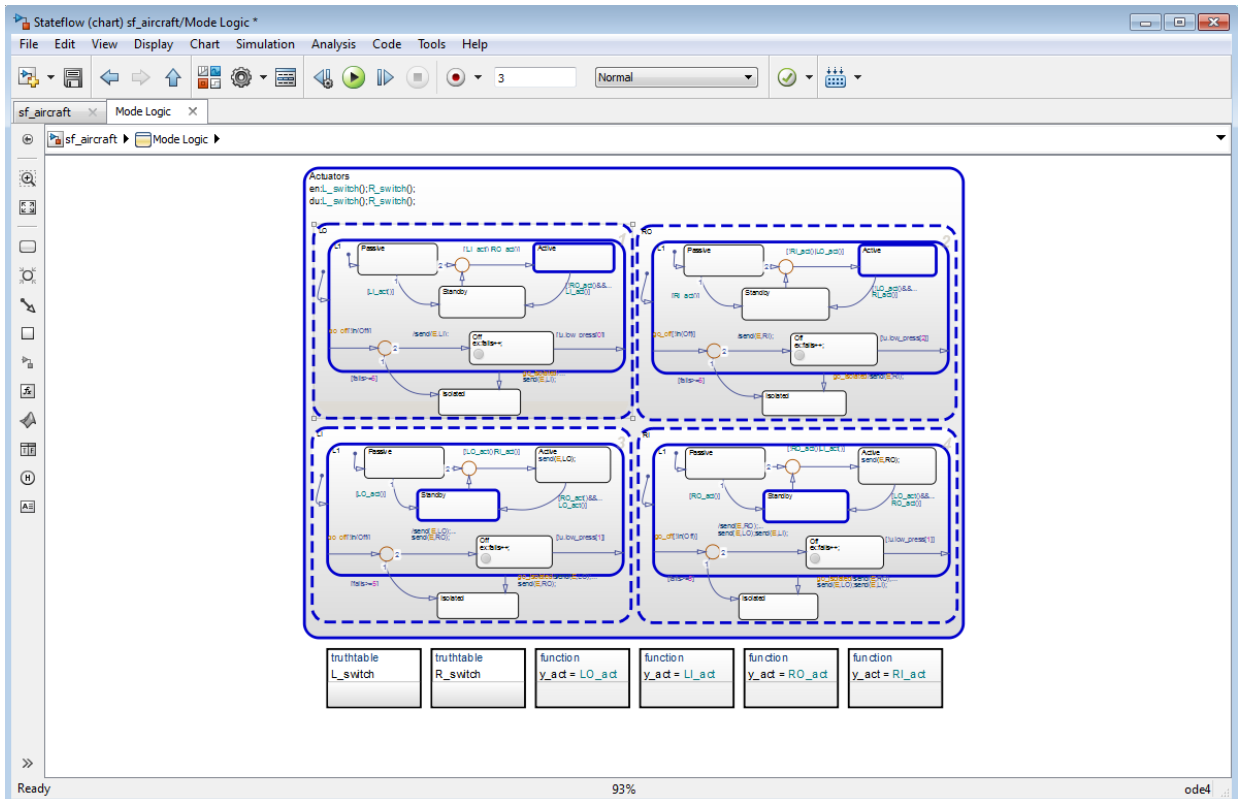
Tip To check if a single state is active, you can use the `isActive` method. For example, type:

```
c.Actuators.LI.L1.Standby.isActive
```

This command returns true (1) when a state is active and false (0) otherwise. For information on other methods, see "Methods for Interacting with the Operating Point of a Chart" on page 16-33.

5 Change the state activity in the chart to reflect one actuator failure.

Assume that the left outer (LO) actuator fails. To change the state, use this command:

```
c.Actuators.LO.Isolated.setActive;
```

The newly active substate appears highlighted in the chart.
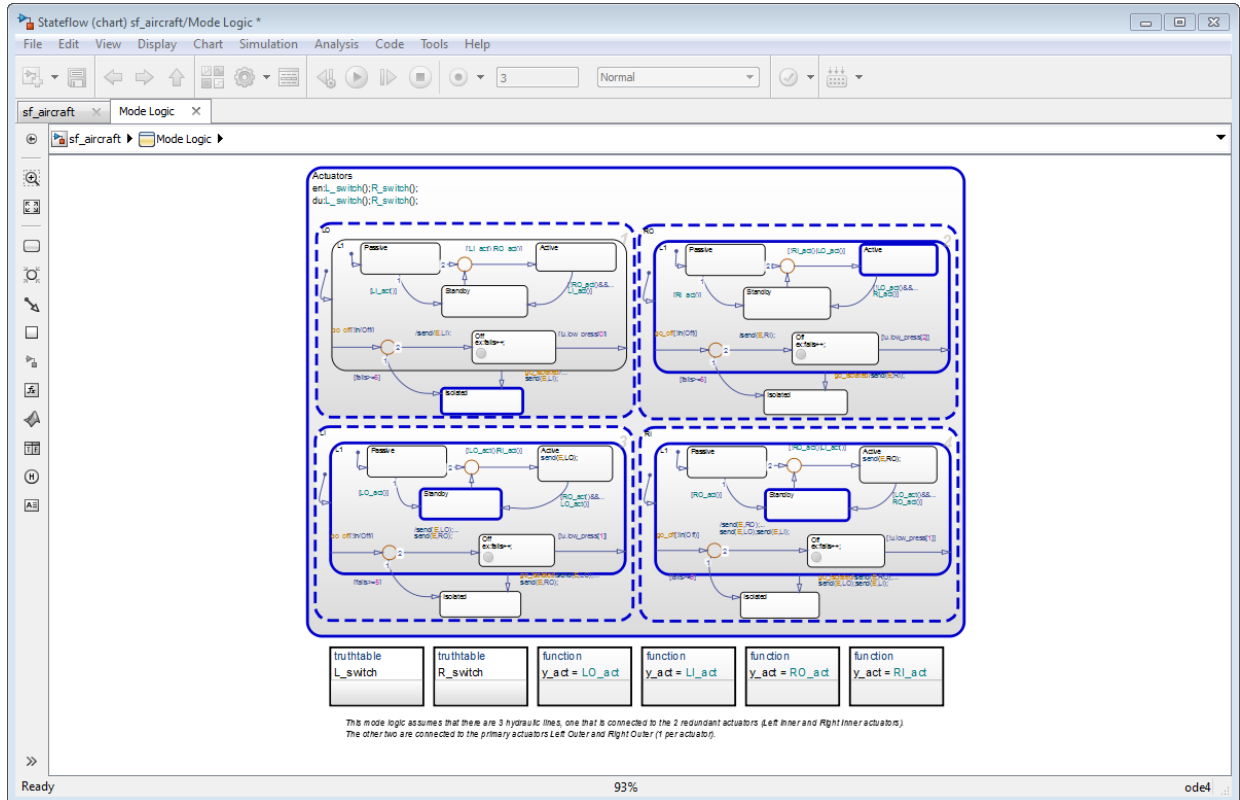
The `setActive` method ensures that the chart exits and enters the appropriate states to maintain state consistency. However, the method does not perform `entry` actions for the newly active substate. Similarly, the method does not perform `exit` actions for the previously active substate.

6 Save the modified operating point by using this command:

```
xFinal = xFinal.set(blockpath, c);
```

Test the Operating Point for One Failure

1 Define the new stop time for the simulation segment to test.

a   Go to the **Solver** pane of the Model Configuration Parameters dialog box.

b   For **Stop time**, enter 10.

c   Click **OK**.

You do not need to enter a new start time because the simulation continues from where it left off.

Programmatic equivalent

You can programmatically set the stop time:

```
set_param('sf_aircraft','StopTime','10');
```

2   Start simulation.

Chart animation shows that the other three actuators react appropriately to the failure of the left outer (LO) actuator.
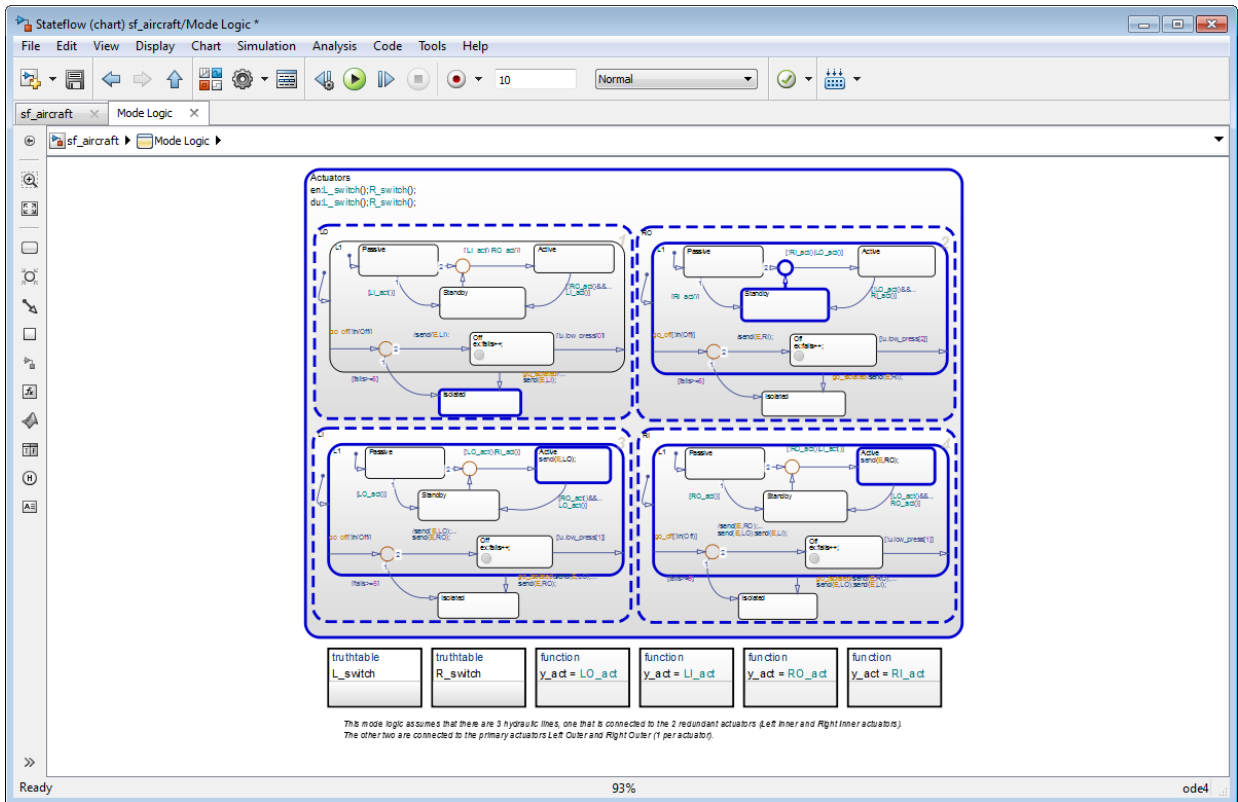
This actuator...	Switches from...	Because...
Left inner (LI)	Standby to active	The left elevator must compensate for the left outer (LO) actuator failure.
Right inner (RI)	Standby to active	The same hydraulic line connects to both inner actuators.
Right outer (RO)	Active to standby	Only one actuator per elevator can be active.

Both elevators continue to maintain steady-state positions.
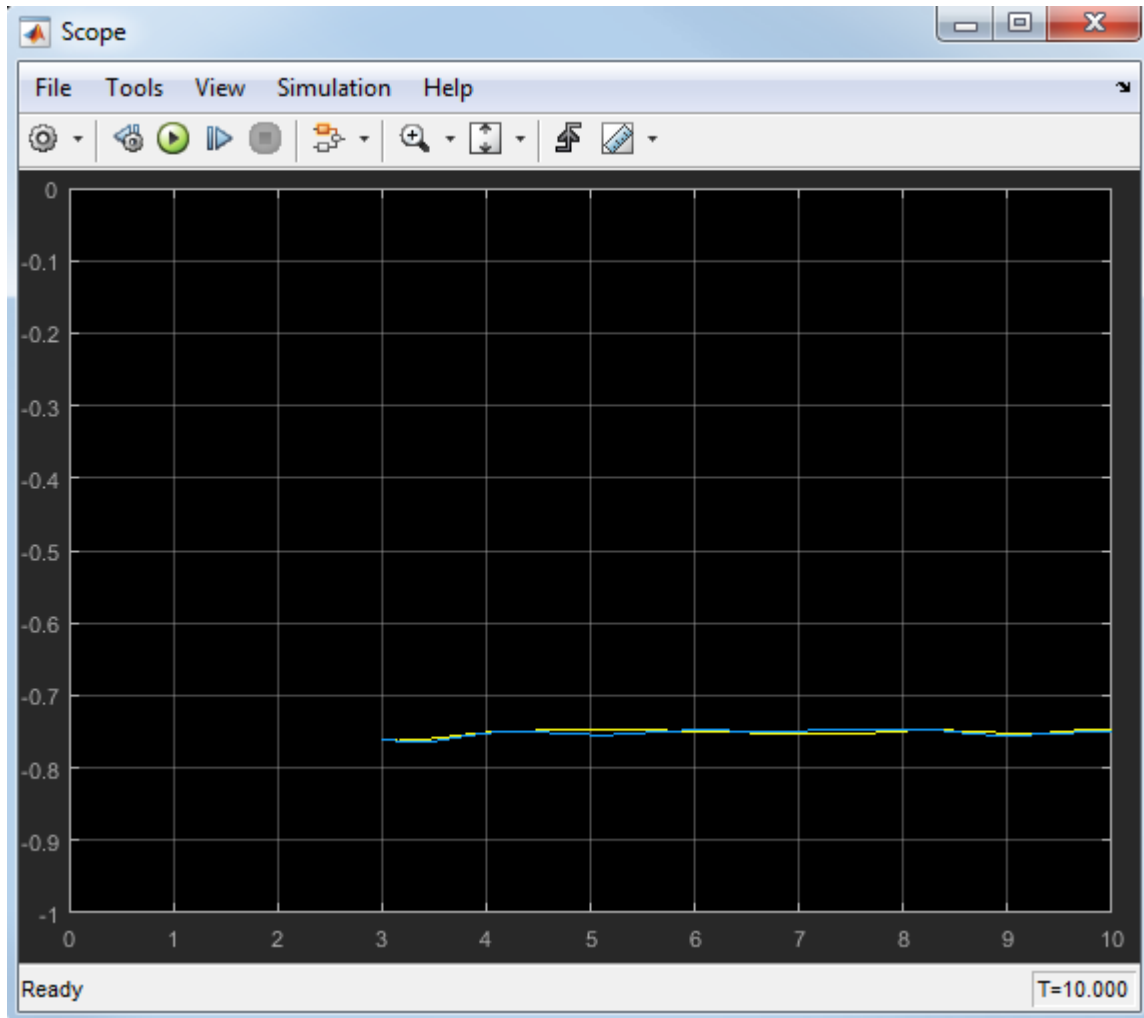
Modify Operating Point Values for Two Actuator Failures

1 Change the state activity in the chart to reflect two actuator failures.

Assume that the left inner (LI) actuator also fails. To change the state, use this command:

```
c.Actuators.LI.Isolated.setActive;
```

2 Save the modified operating point by using this command:

```
xFinal = xFinal.set(blockpath, c);
```

Test the Operating Point for Two Failures

1 In the Model Configuration Parameters dialog box, verify that the stop time is 10.

2 Restart simulation.

Because of failures in both actuators, the left elevator stops working. The right elevator maintains a steady-state position.
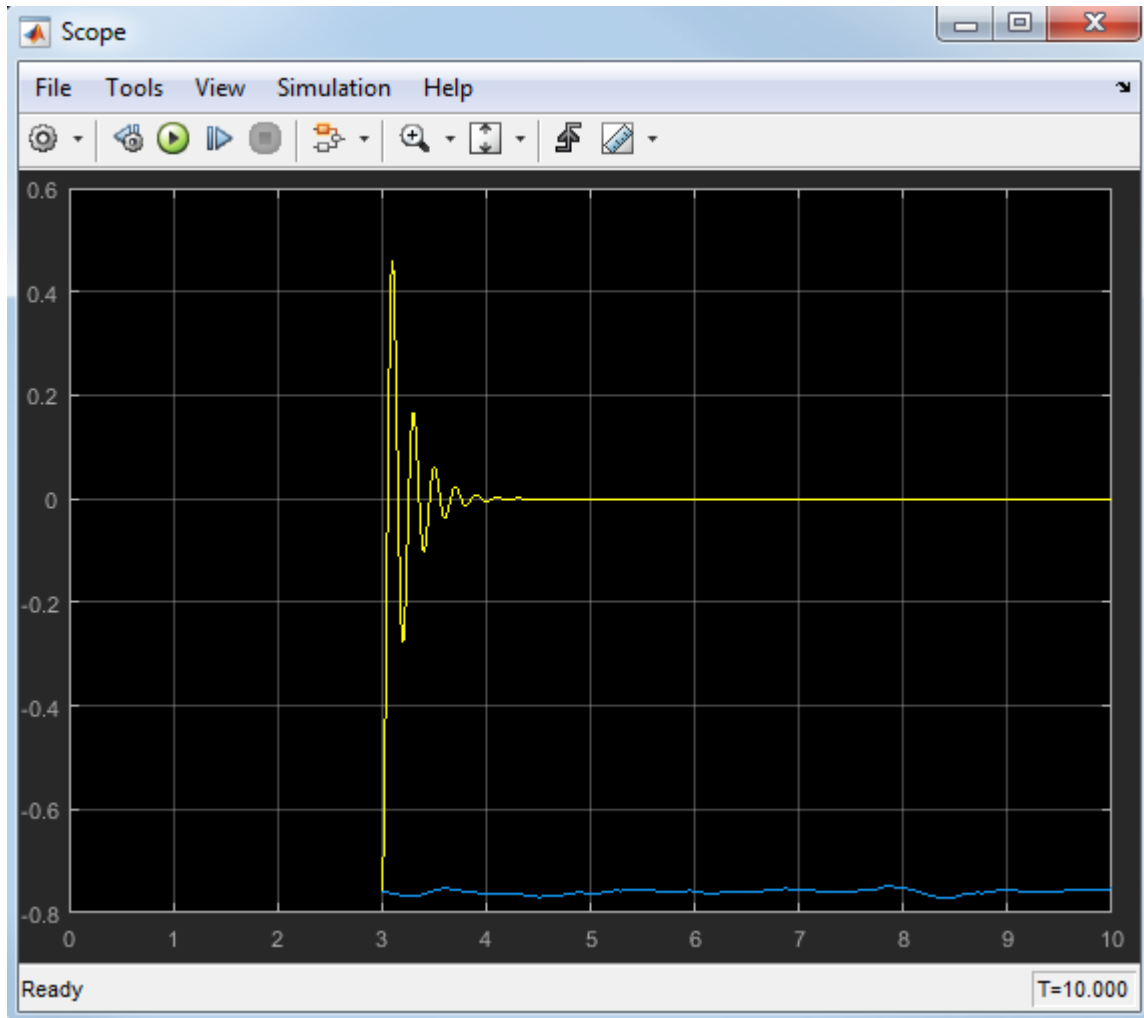
If you modify the operating point of your chart to test the response of the right elevator to actuator failures, you get similar results.

See Also

More About

- "Using Operating Points in Stateflow" on page 16-2
- "Methods for Interacting with the Operating Point of a Chart" on page 16-33
- "Best Practices for Saving the Operating Point of a Chart" on page 16-41

Methods for Interacting with the Operating Point of a Chart

An operating point is a snapshot of the state of a Simulink model at a specific time during simulation. For a Stateflow chart, an operating point includes:

- Activity of chart states
- Values of chart local data
- Values of chart output data
- Values of persistent data in MATLAB functions and Truth Table blocks

For more information, see "Using Operating Points in Stateflow" on page 16-2.

You can use the following methods to interact with the operating point of a chart. Assume that *ch* is a handle to the operating point of your chart, which you obtain using the `get` method.

Type of Object	Method	Description	Example
All chart objects	open	For graphical objects, highlights the object in the Stateflow Editor. For nongraphical objects, highlights the object in the Model Explorer. **Note** For persistent data in MATLAB functions, this method opens the function editor and highlights the persistent data at the exact line in the script.	*ch.data*.open

Type of Object	Method	Description	Example
Chart	checkStateConsistency	Verifies that all states in a chart are consistent. • If a state is inactive, no substates are active. • If a state with parallel decomposition is active, all substates are active. • If a state with exclusive decomposition is active, only one substate is active.	*ch*.checkStateConsistency
Chart	clone	Copies the entire chart operating point to a new variable.	*newOperatingPoint* = *ch*.clone
Chart	highlightActiveStates	Highlights all active states in the Stateflow Editor.	*ch*.highlightActiveStates
Chart	isStateConsistent	Returns true (1) if all states pass a consistency check and false (0) otherwise.	*ch*.isStateConsistent

Type of Object	Method	Description	Example
Chart	removeHighlighting	Removes all highlighting in the Stateflow Editor.	*ch*.removeHighlighting
State	isActive	Returns true (1) if a state is active and false (0) otherwise.	*ch.state*.isActive
State Must be an exclusive leaf state	setActive	Sets a state to be active. This method ensures that no other exclusive states at that level are active.	*ch.state.substate*.setActive
State Must have a history junction and exclusive substates	getPrevActiveChild	Returns the previously active substate.	*ch.state*.getPrevActiveChild
State Must be inactive; must have a history junction and exclusive substates	setPrevActiveChild	Sets the previously active substate.	*ch.state*.setPrevActiveChild('B') **Note** The argument must be the name of a substate (in quotes), or the full operating point path to a substate (without quotes).

See Also

More About

- "Using Operating Points in Stateflow" on page 16-2
- "Divide a Long Simulation into Segments" on page 16-4
- "Best Practices for Saving the Operating Point of a Chart" on page 16-41

Rules for Using the Operating Point of a Chart

An operating point is a snapshot of the state of a Simulink model at a specific time during simulation. For a Stateflow chart, an operating point includes:

- Activity of chart states
- Values of chart local data
- Values of chart output data
- Values of persistent data in MATLAB functions and Truth Table blocks

For more information, see "Using Operating Points in Stateflow" on page 16-2.

Limitations on Values You Can Modify

An operating point does not include information about these elements:

- Machine-parented data
- Persistent data in custom C code
- Persistent data in external MATLAB code

Therefore, you cannot modify the values of those elements.

Rules for Modifying Data Values

These rules apply when you modify data values:

- You cannot change the data type or size. Scalar data must remain scalar. Vector and matrix data must keep the same dimensions. The only exception to this rule is Stateflow data of `ml` type (see "ml Data Type" on page 12-37 for details).
- For enumerated data types, you can choose only enumerated values from the type definition. For other data types, new values must fall within the range that you specify in the **Minimum** and **Maximum** parameters.
- Use one-based indexing to define rows and columns of a matrix.

  Suppose that you want to change the value of an element in a 21-by-12 matrix. To modify the element in the first row and second column, type:

  ```
  c.state_name.data_name.Value(1,2) = newValue;
  ```

Rules for Modifying State Activity

These rules apply when you use the `setActive` method on an exclusive (OR) leaf state:

- State-parented local data does not reinitialize.
- The newly active state does not execute any entry actions. Similarly, the previously active state does not execute any exit actions.

  If you want these state actions to occur, you must execute them separately. For example, if your state actions assign values to data, you must assign the values explicitly.

- The `setActive` method tries to maintain state consistency by:

  - Updating state activity for parent, grandparent, and sibling states
  - Resetting temporal counters for newly active states
  - Updating values of state output data (read-only)
  - Enabling or disabling function-call subsystems and Simulink functions that bind to states

- The `highlightActiveStates` method also executes when these conditions are true:

  - The model is open.
  - The chart is visible.
  - The `highlightActiveStates` method has executed at least once, but not the `removeHighlighting` method.

Restriction on Continuous-Time Charts

After you load an operating point for a continuous-time chart, you can restart simulation from a nonzero time. However, you cannot modify the state activity or any data values, because the operating point for a continuous-time chart is read-only. For more information, see "Continuous-Time Modeling in Stateflow" on page 21-2.

No Partial Loading of a Operating Point

When you load an operating point, the complete operating point is available as a variable in the MATLAB base workspace. You cannot perform partial loading of an operating point for a subset of chart objects.

Restriction on Copying Operating Point Values

Use the `clone` method to copy an entire operating point to a new variable (see "Methods for Interacting with the Operating Point of a Chart" on page 16-33). You cannot copy a subset of operating point values, because the `clone` method works only at the chart level.

Suppose that you obtain a handle to the operating point of your chart using these commands:

```
blockpath = 'model/chart';
c = xFinal.get(modelOperatingPoint, blockpath);
```

Assignment statements such as `stateCopy = c.state`, `dataCopy = c.data`, and `operatingPointCopy = c` do not work. These assignments create copies of object handles, not operating point values.

See Also

More About

- "Using Operating Points in Stateflow" on page 16-2
- "Divide a Long Simulation into Segments" on page 16-4
- "Best Practices for Saving the Operating Point of a Chart" on page 16-41

Best Practices for Saving the Operating Point of a Chart

An operating point is a snapshot of the state of a Simulink model at a specific time during simulation. For a Stateflow chart, an operating point includes:

- Activity of chart states
- Values of chart local data
- Values of chart output data
- Values of persistent data in MATLAB functions and Truth Table blocks

For more information, see "Using Operating Points in Stateflow" on page 16-2.

Use MAT-Files to Save a Operating Point for Future Use

To save an operating point from the MATLAB base workspace, save the variable with final state data in a MAT-file.

For example, type at the command prompt:

```
save('sf_car_ctx01.mat', 'sf_car_ctx01')
```

For more information, see `save` in the MATLAB documentation.

Use Scripts to Save Operating Point Commands for Future Use

To save a list of operating point commands for future use, copy them from a procedure and paste them in a MATLAB script.

For example, to reuse the commands in "Divide a Long Simulation into Segments" on page 16-4, you can store them in a script named `sf_boiler_operatingpoint_commands.m`:

```
% Open the model.
sf_boiler;

% Set parameters to save the operating point at the desired time.
set_param('sf_boiler','SaveFinalState','on','FinalStateName',...
'sf_boiler_ctx01','SaveOperatingPoint','on');

% Specify the start and stop times for the simulation segment.
```

```matlab
set_param('sf_boiler','StartTime','0','StopTime','400');

% Simulate the model.
sim('sf_boiler');

% Disable saving of the operating point to avoid overwriting.
set_param('sf_boiler','SaveOperatingPoint','off', ...
'SaveFinalState','off');

% Load the operating point.
set_param('sf_boiler', 'LoadInitialState', 'on', ...
'InitialState', 'sf_boiler_ctx01');

% Specify the new stop time for the simulation segment.
set_param('sf_boiler','StopTime','600');

% Simulate the model.
sim('sf_boiler');
```

The start time does not change, but the operating point restore fast forwards the simulation to the time of the snapshot.

See Also

More About

- "Using Operating Points in Stateflow" on page 16-2
- "Divide a Long Simulation into Segments" on page 16-4
- "Rules for Using the Operating Point of a Chart" on page 16-38

17

Vectors and Matrices in Stateflow Charts

Vectors and Matrices in Stateflow Charts

When to Use Vectors and Matrices

Use vectors and matrices when you want to:

- Process multidimensional input and output signals
- Combine separate scalar data into one signal

For examples, see "Find Patterns in Data Transmission by Using Vectors" on page 17-16 and "Calculate Motion by Using Matrices" on page 17-18.

Where You Can Use Vectors and Matrices

You can define vectors and matrices at these levels of the Stateflow hierarchy:

- Charts
- Subcharts
- States
- Functions

You can use vectors and matrices to define:

- Input data
- Output data
- Local data
- Function inputs
- Function outputs

You can also use vectors and matrices as arguments for:

- State actions
- Transition actions
- MATLAB functions
- Truth table functions
- Graphical functions

- Simulink functions
- Change detection operators

For more information, see "Supported Operations for Vectors and Matrices" on page 17-11 and "Rules for Vectors and Matrices in Stateflow Charts" on page 17-3.

Rules for Vectors and Matrices in Stateflow Charts

These rules apply when you use vectors and matrices in Stateflow charts.

Use only operands of equal dimensions for element-wise operations

If you try to perform element-wise operations on vectors or matrices with unequal dimensions, a size mismatch error appears when you simulate your model. See "Supported Operations for Vectors and Matrices" on page 17-11.

Do not define vectors and matrices with ml base type

If you define a vector or matrix with `ml` base type, an error message appears when you try to simulate your model. This base type supports only scalar data.

For more information about this type, see "ml Data Type" on page 12-37.

Use only real numbers to set initial values of vectors and matrices

When you set the initial value for an element of a vector or matrix, use a real number. If you use a complex number, an error message appears when you try to simulate your model.

Note You can set values of vectors and matrices to complex numbers after initialization.

Do not use vectors and matrices with temporal logic operators

You cannot use a vector or matrix as an argument for temporal logic operators, because time is a scalar quantity.

Best Practices for Vectors and Matrices in Stateflow Charts

Perform Matrix Multiplication and Division Using MATLAB Functions

In a Stateflow chart, the multiplication and division operators perform element-wise multiplication and division. Use a MATLAB function to perform standard matrix multiplication and division.

For example, suppose that you want to perform standard matrix operations on two square matrices during simulation. Follow these steps:

1   In your chart, add a MATLAB function with the following signature:

```
[y1, y2, y3] = my_matrix_ops(u1, u2)
```

2   Double-click the function box to open the editor.

3   In the editor, enter the code below.

```
function [y1, y2, y3] = my_matrix_ops(u1, u2)
%#codegen

y1 = u1 * u2;  % matrix multiplication
y2 = u1 \ u2;  % matrix division from the right
y3 = u1 / u2;  % matrix division from the left
```

This function computes three values:

- y1 is the product of two input matrices u1 and u2.
- y2 is the matrix that solves the equation u1 * y2 = u2.
- y3 is the matrix that solves the equation y3 * u1 = u2.

4   Set properties for the input and output data.

   a   Open the Model Explorer.

   b   In the **Model Hierarchy** pane, navigate to the level of the MATLAB function.

   c   In the **Contents** pane, set properties for each data object.
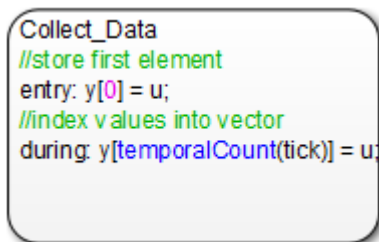
Note To initialize a matrix, see "Define a Matrix" on page 17-7.

Index a Vector Using the temporalCount Operator

When you index a vector, you can use the `temporalCount` operator to avoid using an extra variable for the index counter. This indexing method works for vectors that contain real or complex data.

For example, suppose that you want to collect input data in a buffer during simulation. Follow these steps:

1   Add this state to your Stateflow chart.

The state `Collect_Data` stores data in the vector y, which is of size `10`. The `entry` action assigns the value of input data u to the first element of y. The `during` action assigns the next nine values of input data to successive elements of the vector y until you store ten elements.

2   Add the input data u to the chart.

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

    b   In the Data properties dialog box, enter u in the **Name** field.

    c   Click **OK**.

3   Add the output data y to the chart.

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

    b   In the Data properties dialog box, enter y in the **Name** field.

    c   Enter `10` in the **Size** field.

    d   Click **OK**.

Note You do not need to set initial values for this output vector. By default, all elements initialize to 0.

For information about the `temporalCount` operator, see "Control Chart Execution by Using Temporal Logic" on page 12-49.

See Also

More About

Add Vector and Matrix Data

Define a Vector

Define a vector in a Stateflow chart as follows:

1 Add data to your chart as described in "Add Stateflow Data" on page 9-2.

2 In the **General** pane of the Data properties dialog box, enter the dimensions of the vector in the **Size** field.

   For example, enter [4 1] to specify a 4-by-1 vector.

3 Specify the name, base type, and other properties for the new data.

Note Vectors cannot have the base type `ml`. See "Rules for Vectors and Matrices in Stateflow Charts" on page 17-3.

---

4 Set initial values for the vector.

   - If initial values of all elements are the same, enter a real number in the **Initial value** field. This value applies to all elements of a vector of any size.

   - If initial values differ, enter real numbers in the **Initial value** field. For example, you can enter:

     [1; 3; 5; 7]

Tip If you want to initialize all elements of a vector to 0, do nothing. When no values are explicitly defined, all elements initialize to 0.

---

5 Click **Apply**.

Define a Matrix

Define a matrix in a Stateflow chart as follows:

1 Add data to your chart as described in "Add Stateflow Data" on page 9-2.

2 In the **General** pane of the Data properties dialog box, enter the dimensions of the matrix in the **Size** field.

   For example, enter [3 3] to specify a 3-by-3 matrix.

3 Specify the name, base type, and other properties for the new data.

Note Matrices cannot have the base type `ml`. See "Rules for Vectors and Matrices in Stateflow Charts" on page 17-3.

---

4 Set initial values for the matrix.

- If initial values of all elements are the same, enter a real number in the **Initial value** field. This value applies to all elements of a matrix of any size.

- If initial values differ, enter real numbers in the **Initial value** field. For example, you can enter:

  ```
  [1 2 3; 4 5 6; 7 8 9]
  ```

Tip If you want to initialize all elements of a matrix to 0, do nothing. When no values are explicitly defined, all elements initialize to 0.

---

5 Click **Apply**.

See Also

More About

- "Vectors and Matrices in Stateflow Charts" on page 17-2
- "Supported Operations for Vectors and Matrices" on page 17-11
- "Convert Scalars to Nonscalars by Using Scalar Expansion" on page 17-9

Convert Scalars to Nonscalars by Using Scalar Expansion

Scalar expansion is a method of converting scalar data to match the dimensions of vector or matrix data. For example, in this assignment, scalar expansion converts a value of 1 to a vector or matrix where all the elements are 1:

```
A = 1;
```

Scalar expansion is supported only in Stateflow charts that use C as the action language.

How Scalar Expansion Works for Functions

Suppose that you have a function signature $yy = f(uu)$, where the formal arguments yy and uu are scalars. Assume that you have a function call $y = f(u)$. The rules of scalar expansion for function calls with a single output follow.

Output y	Input u	Result
Scalar	Scalar	No scalar expansion occurs.
Vector or matrix	Scalar	Scalar expansion occurs for `f(u)` to match the dimensions of `y`.
Vector or matrix	Vector or matrix	Scalar expansion occurs so that `y[i] = f(u[i])`.
Scalar	Vector or matrix	An error message alerts you to a size mismatch.

For functions with multiple outputs, the same rules apply except for the case where the outputs and inputs of the function call are all vectors or matrices. In this case, scalar expansion does not occur, and an error message alerts you to a size mismatch.

The rules of scalar expansion apply to all functions that you use in Stateflow charts:

- MATLAB functions
- Graphical functions
- Simulink functions
- Truth table functions

See Also

More About

- "Vectors and Matrices in Stateflow Charts" on page 17-2
- "Add Vector and Matrix Data" on page 17-7
- "Supported Operations for Vectors and Matrices" on page 17-11

Supported Operations for Vectors and Matrices

Stateflow charts in Simulink models have an action language property that defines the syntax that you use to compute with vectors and matrices. The action language properties are:

- MATLAB as the action language.

- C as the action language.

For more information, see "Differences Between MATLAB and C as Action Language Syntax" on page 13-6.

Indexing Notation

In charts that use MATLAB as the action language, refer to elements of a vector or matrix by using one-based indexing delimited by parentheses. Separate indices for different dimensions with commas.

In charts that use C as the action language, refer to elements of a vector or matrix by using zero-based indexing delimited by brackets. Enclose indices for different dimensions in their own pair of brackets.

Example	MATLAB as the Action Language	C as the Action Language
The first element of a vector V	V(1)	V[0]
The ith element of a vector V	V(i)	V[i-1]
The element in row 4 and column 5 of a matrix M	M(4,5)	M[3][4]
The element in row i and column j of a matrix M	M(i,j)	M[i-1][j-1]

Binary Operations

This table summarizes the interpretation of all binary operations on vector and matrix operands according to their order of precedence (1 = highest, 3 = lowest). Binary operations are left associative so that, in any expression, operators with the same precedence are evaluated from left to right. Except for the matrix multiplication and division operators in charts that use MATLAB as the action language, all binary operators perform element-wise operations.

Operation	Precedence	MATLAB as the Action Language	C as the Action Language
a * b	1	Matrix multiplication.	Element-wise multiplication. For matrix multiplication, use the * operation in a MATLAB function.
a .* b	1	Element-wise multiplication.	Not supported. Use the operation a * b.
a / b	1	Matrix right division.	Element-wise right division. For matrix right division, use the / operation in a MATLAB function.
a ./ b	1	Element-wise right division.	Not supported. Use the operation a / b.
a \ b	1	Matrix left division.	Not supported. Use the \ operation in a MATLAB function.
a .\ b	1	Element-wise left division.	Not supported. Use the .\ operation in a MATLAB function.
a + b	2	Addition.	Addition.
a - b	2	Subtraction.	Subtraction.
a == b	3	Comparison, equal to.	Comparison, equal to.
a ~= b	3	Comparison, not equal to.	Comparison, not equal to.
a != b	3	Not supported. Use the operation a ~= b.	Comparison, not equal to.
a <> b	3	Not supported. Use the operation a ~= b.	Comparison, not equal to.

Unary Operations and Actions

This table summarizes the interpretation of all unary operations and actions on vector and matrix operands. Unary operations:

- Have higher precedence than the binary operators.
- Are right associative so that, in any expression, they are evaluated from right to left.
- Perform element-wise operations.

Example	MATLAB as the Action Language	C as the Action Language
~a	Logical NOT. For bitwise NOT, use the bitcmp function.	• Bitwise NOT (default). Enable this operation by selecting the **Enable C-bit operations** chart property. • Logical NOT. Enable this operation by clearing the **Enable C-bit operations** chart property. See "Enable C-Bit Operations" on page 24-6.
!a	Not supported. Use the operation ~a.	Logical NOT.
-a	Negative.	Negative.
a++	Not supported.	Increment all elements of the vector or matrix. Equivalent to a = a+1.
a--	Not supported.	Decrement all elements of the vector or matrix. Equivalent to a = a-1.

Assignment Operations

This table summarizes the interpretation of assignment operations on vector and matrix operands.

Operation	MATLAB as the Action Language	C as the Action Language
a = b	Simple assignment.	Simple assignment.
a += b	Not supported. Use the expression a = a+b.	Equivalent to a = a+b.

Operation	MATLAB as the Action Language	C as the Action Language
a -= b	Not supported. Use the expression a = a-b.	Equivalent to a = a-b.
a *= b	Not supported. Use the expression a = a*b.	Equivalent to a = a*b.
a /= b	Not supported. Use the expression a = a/b.	Equivalent to a = a/b.

Assign Values to Individual Elements of a Matrix

You can assign a value to an individual entry of a vector or matrix by using the syntax appropriate to the action language of the chart.

Example	MATLAB as the Action Language	C as the Action Language
Assign the value 10 to the first element of the vector V.	V(1) = 10;	V[0] = 10;
Assign the value 77 to the element in row 2 and column 9 of the matrix M.	M(2,9) = 77;	M[1][8] = 77;

Assign Values to All Elements of a Matrix

In charts that use MATLAB as the action language, you can specify all of the elements of a vector or matrix in a single statement. For example, this action assigns each element of the 2-by-3 matrix A to a different value:

```
A = [1 2 3; 4 5 6];
```

In charts that use C as the action language, you can use scalar expansion to set all of the elements of a vector or matrix to the same value. For example, this action sets all of the elements of the matrix A to 10:

```
A = 10;
```

Charts that use MATLAB as the action language do not support scalar expansion. For more information, see "Convert Scalars to Nonscalars by Using Scalar Expansion" on page 17-9.

See Also

More About

- "Vectors and Matrices in Stateflow Charts" on page 17-2
- "Add Vector and Matrix Data" on page 17-7
- "Convert Scalars to Nonscalars by Using Scalar Expansion" on page 17-9

Find Patterns in Data Transmission by Using Vectors

The model `sf_frame_sync_controller` is an example of using a vector in a Stateflow chart to find a fixed pattern in a data transmission.
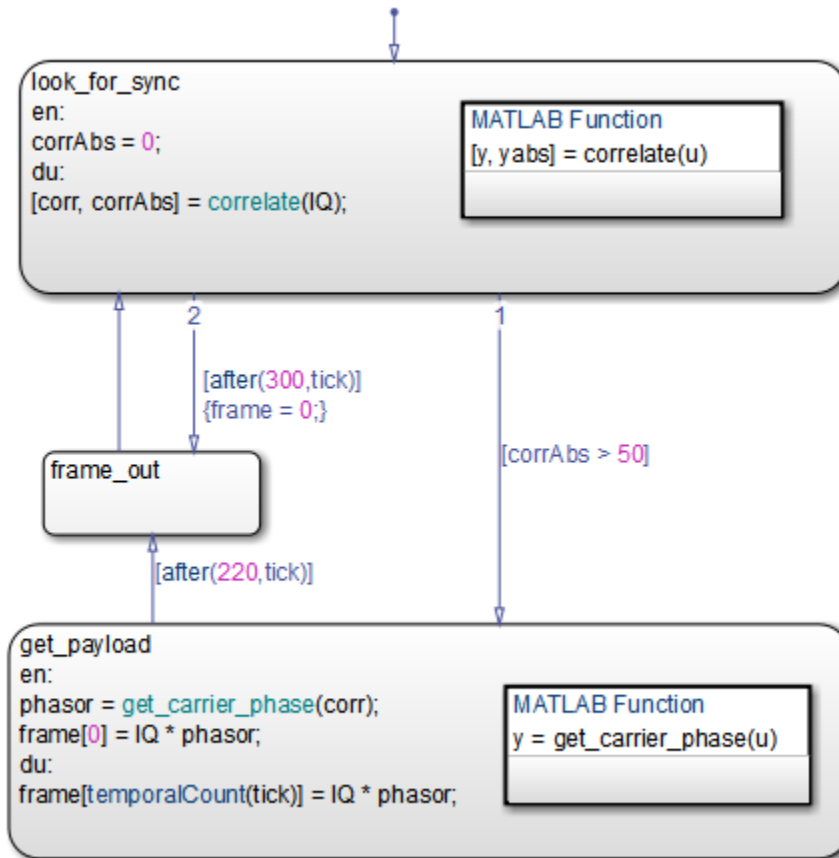
For details of how the chart works, see "Detect Valid Transmission Data Using Frame Synchronization" on page 23-15.

Storage of Complex Data in a Vector

The state `get_payload` stores complex data in the vector `frame`, which is of size 221. The `entry` action assigns the value of (`IQ * phasor`) to the first element of `frame`. The `during` action assigns the next 220 values of (`IQ * phasor`) to successive elements of `frame` until you store 221 elements. (For more information, see "Index a Vector Using the temporalCount Operator" on page 17-5.)

Scalar Expansion of a Vector

In the second outgoing transition of the state `look_for_sync`, the transition action `frame = 0` resets all elements of the vector `frame` to 0 via scalar expansion. (For more information, see "Convert Scalars to Nonscalars by Using Scalar Expansion" on page 17-9.)

See Also

More About

Calculate Motion by Using Matrices

The model `sf_pool` is an example of using matrices in a Stateflow chart to simulate the opening shot on a pool table.
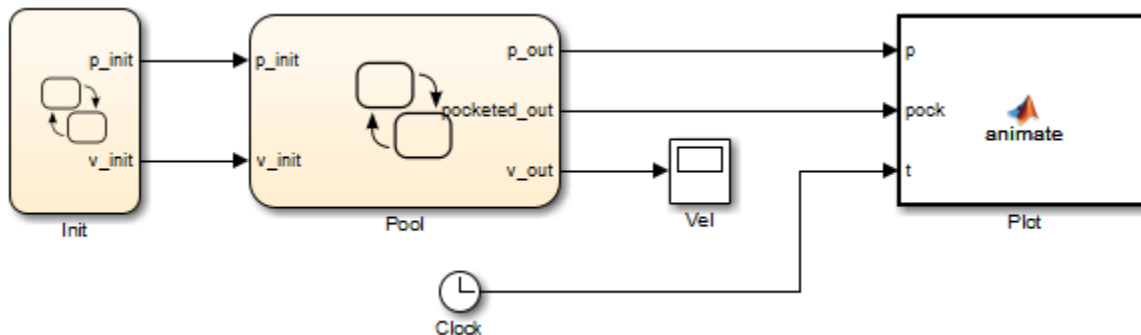
How the Model Works

The model consists of the following blocks.

Model Component	Description
Init chart	Initializes the position and velocity of the cue ball.
Pool chart	Calculates the two-dimensional dynamics of each ball on the pool table.
Plot block	Animates the motion of each ball during the opening shot.
Vel scope	Displays the velocity of each ball during the opening shot.
Clock	Provides the instantaneous simulation time to the Plot block.

Storage of Two-Dimensional Data in Matrices

To simulate the opening shot, the Pool chart stores two-dimensional data in matrices.

To store values for...	The Pool chart uses...
The instantaneous position of each ball	The 15-by-2 matrix `p`
The instantaneous velocity of each ball	The 15-by-2 matrix `v`
Friction and interaction forces acting on each ball	The 15-by-2 matrix `v_dot`
Boolean data on whether any two balls are in contact	The 15-by-15 matrix `ball_interaction`

Calculation of Two-Dimensional Dynamics of Each Ball

The Pool chart calculates the motion of each ball on the pool table using MATLAB functions that perform matrix calculations.

MATLAB Function	Description
`frictionForce`	Calculates the friction force acting on each ball.

MATLAB Function	Description
getBallInteraction	Returns a matrix of Boolean data on whether any two balls are in contact.
hasBallInteractionChanged	Returns 1 if ball interactions have changed and 0 otherwise.
initBalls	Initializes the position and velocity of every ball on the pool table.
interactionForce	Calculates the interaction force acting on each ball.
isAnyBallGoingToStop	Returns 1 if any ball has stopped moving and 0 otherwise.
isAnyBallNewlyPocketed	Returns 1 if any ball has been newly pocketed and 0 otherwise.
isAnyBallOutOfBounds	Returns true if any ball is out of bounds and false otherwise.
nearHole	Returns true if a ball is near a pocket on the pool table and false otherwise.
pocketNewBalls	Sets the velocity of a ball to 0 if it has been pocketed.
resetBallsPosAndVel	Resets the position and velocity of any ball that is out of bounds.
updateStopFlags	Keeps track of which balls have stopped moving.

Run the Model

To run the model, follow these steps:

1   Open the sf_pool model.
2   Start simulation.
3   Click anywhere in the animated pool table to specify the initial position of the cue ball.
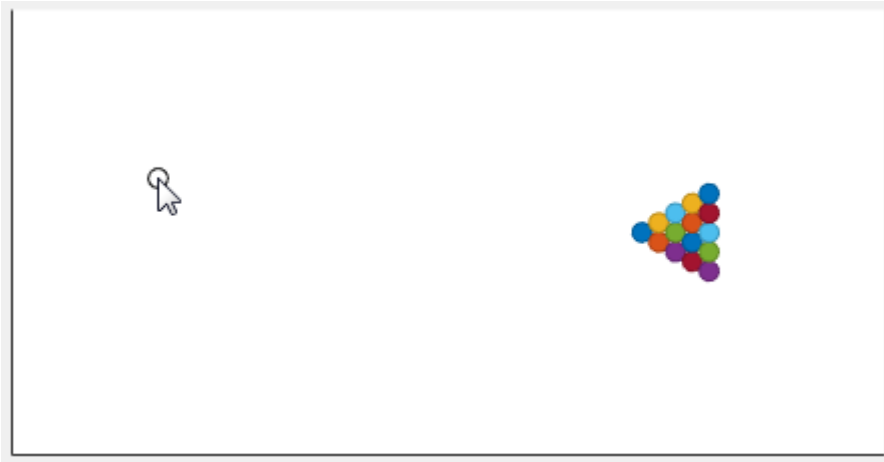
4 Click a different spot to specify the initial velocity of the cue ball.
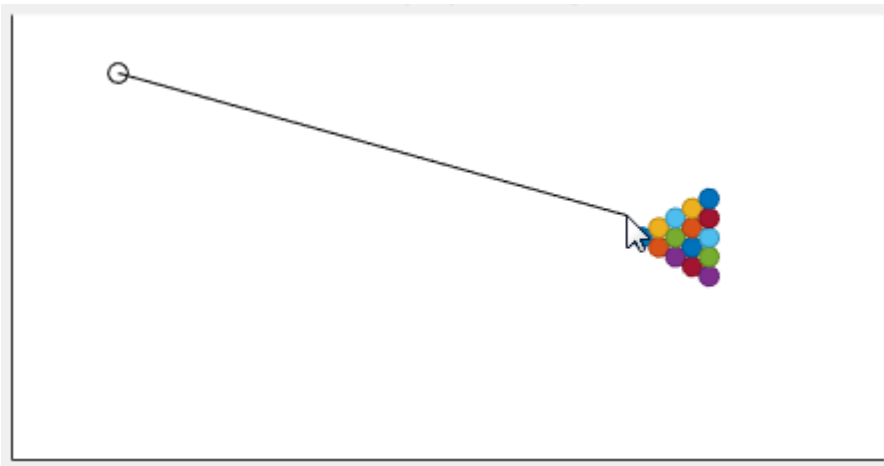
5 Watch the balls move across the pool table.

See Also

More About
- "Vectors and Matrices in Stateflow Charts" on page 17-2
- "Add Vector and Matrix Data" on page 17-7
- "Supported Operations for Vectors and Matrices" on page 17-11

18

Variable-Size Data in Stateflow Charts

Declare Variable-Size Data in Stateflow Charts

Variable-size data is data whose size can change at run time. In contrast, fixed-size data is data whose size is known and locked at compile time and does not change at run time. Use variable-size data if the output from a Stateflow chart in a Simulink model is an array whose size depends on the state of the chart.

Stateflow charts exchange variable-size data with other charts and blocks in their models through MATLAB functions, Simulink functions, and truth tables that use MATLAB as the action language. You pass variable-size data to these functions as chart-level inputs and outputs from state actions and transition logic. However, you must perform all computations with variable-size data inside the functions, not directly in states or transitions. For more information about the functions that interact with variable-size, chart-level inputs and outputs, see:

- "Reuse MATLAB Code by Defining MATLAB Functions" on page 28-2
- "Reuse Simulink Components in Stateflow Charts" on page 29-2
- "Reuse Combinatorial Logic by Defining Truth Tables" on page 27-2

Enable Support for Variable-Size Data

Support for variable-size data is enabled by default. To modify this option for individual charts:

1   Right-click an open area of the chart and select **Properties**.
2   In the Chart properties dialog box, select or clear the **Support variable-size arrays** check box.

After enabling support at the chart level, you can declare variable-size inputs and outputs.

Declare Variable-Size Inputs and Outputs

1   Add a data object to the chart, as described in "Add Stateflow Data" on page 9-2.
2   Set **Scope** property as `Input` or `Output`.
3   Select the **Variable size** check box.
4   Set the **Size** property for the data.

Scope	What to Specify
Input	Enter -1 to inherit size from Simulink or specify the explicit size and upper bound.
Output	Specify the explicit size and upper bound.

For example, this specification declares a variable-size input `data` that inherits its **Size** and **Type** from the Simulink model. To specify a 2-D matrix where the upper bounds are 2 for the first dimension and 4 for the second dimension, in the **Size** field, enter [2 4] instead.
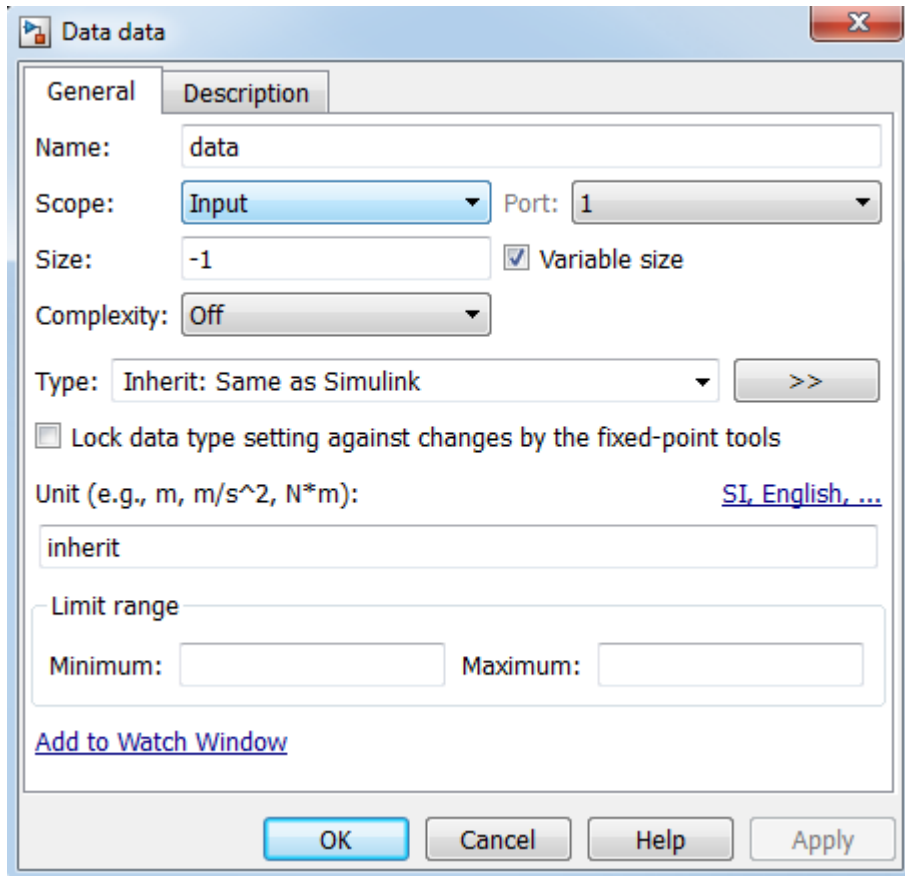
Rules for Using Variable-Size Data

- Declare variable-size data as chart inputs and outputs only, not as local data.

  See "Declare Variable-Size Inputs and Outputs" on page 18-2.
- Do not perform computations with variable-size data directly in states or transitions.
- Perform all computations with variable-size data in MATLAB functions, Simulink functions, and truth tables that use MATLAB as the action language.

  You can pass the data as inputs and outputs to MATLAB and Simulink functions in your chart from state actions and transition logic. MATLAB functions can also access the chart-level, variable-size data directly. For more information, see "Compute Output Based on Size of Input Signal" on page 18-5.

See Also

More About

- "Compute Output Based on Size of Input Signal" on page 18-5
- "Reuse MATLAB Code by Defining MATLAB Functions" on page 28-2
- "Reuse Simulink Components in Stateflow Charts" on page 29-2
- "Reuse Combinatorial Logic by Defining Truth Tables" on page 27-2

Compute Output Based on Size of Input Signal

This example shows how to modify the size of output data in a Stateflow® chart during run time. Stateflow charts exchange variable-size data with other charts and blocks in the model through MATLAB® functions, Simulink® functions, and MATLAB truth tables. This example illustrates this process by using MATLAB functions in two Stateflow charts.

In this model, one Stateflow chart, VarSizeSignalSource, uses temporal logic to generate a variable-size signal. A second chart, SizeBasedProcessing, computes the output based on the size of the signal generated by the first chart.

Copyright 2009-2018 The MathWorks, Inc.
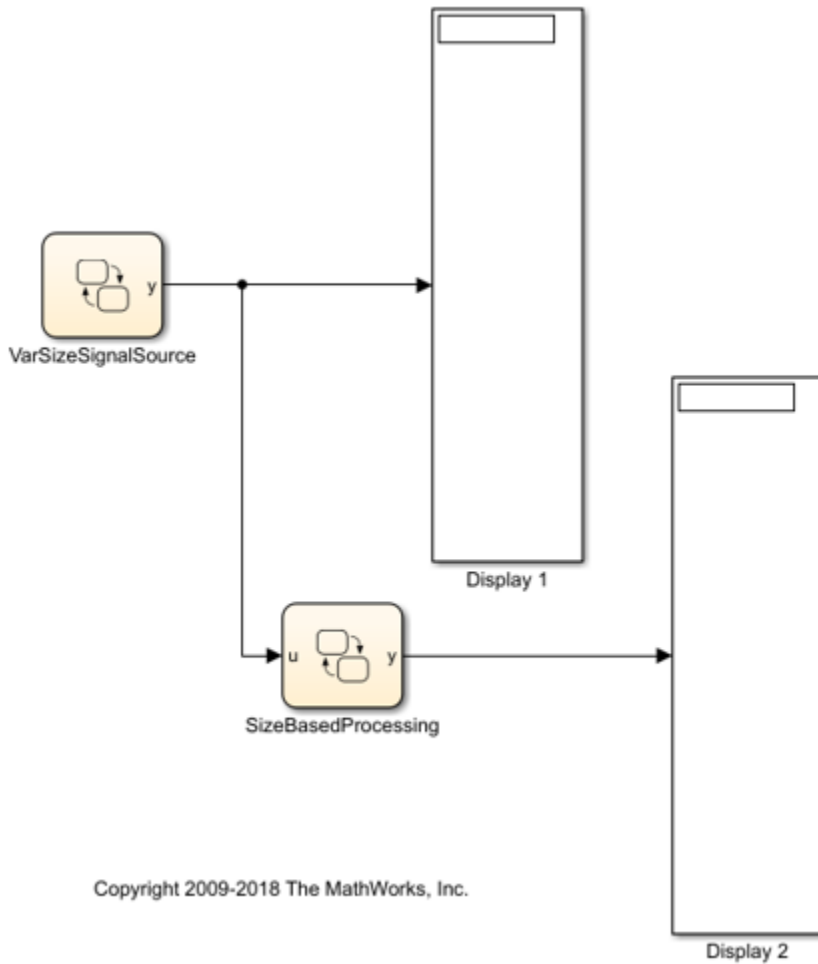
VarSizeSignalSource Chart

The VarSizeSignalSource chart uses temporal logic to transition between three states. Each state generates a different size output.
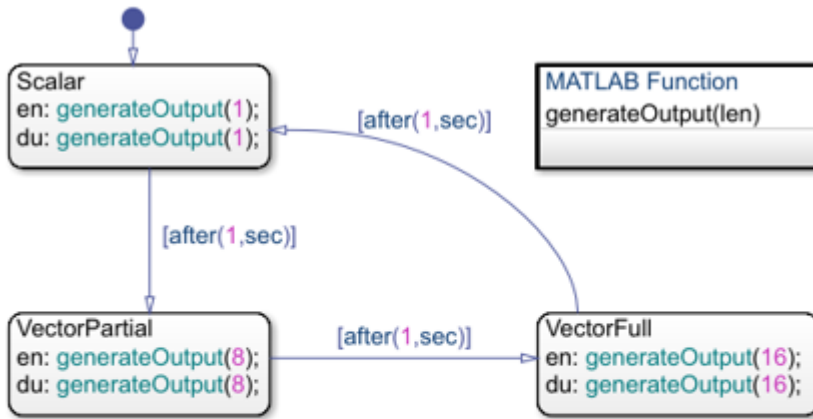
Use this Stateflow chart to generate an output signal with a variable size. In this chart, each state calls a MATLAB function that assigns to the chart output a different-size signal (within the bounds). During simulation, transitions between states change the size of the signal. You can use this variable-size signal to test other blocks and subsystems in your model.

The chart works like a source block. It has no input and one variable-size output y.
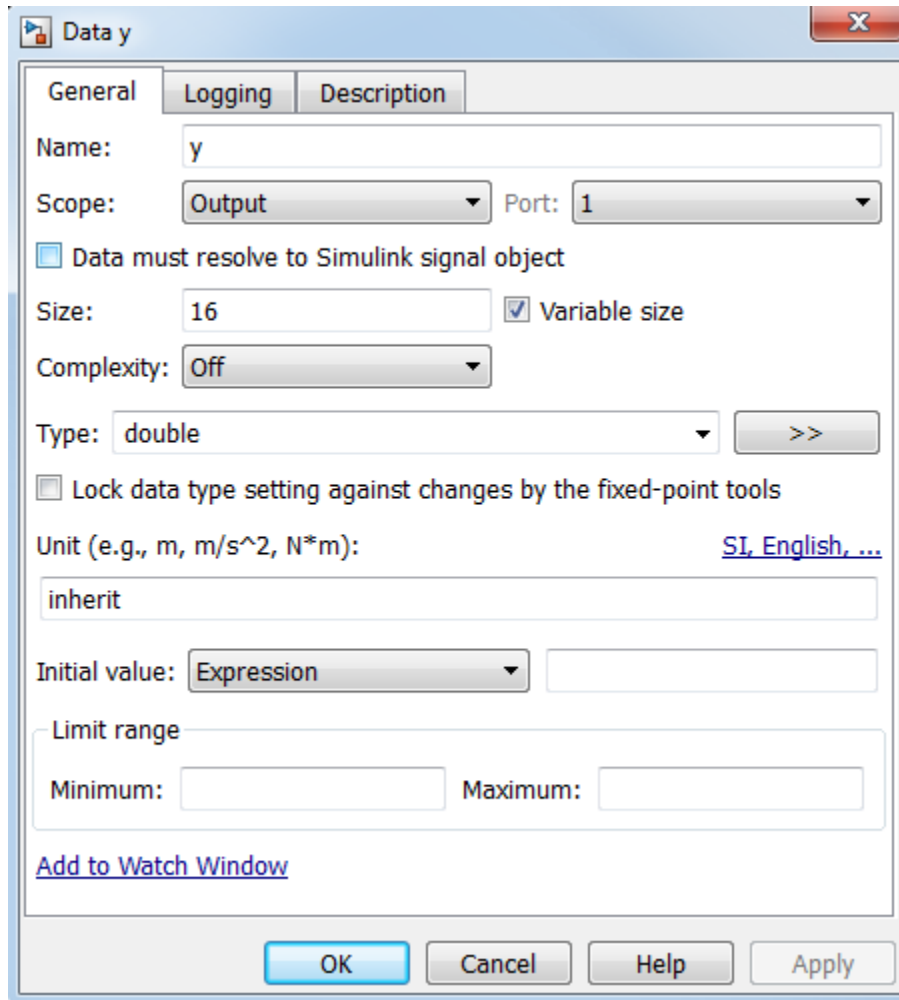
For variable-size outputs, you must explicitly specify the size and upper bound for each dimension. In this case, y is a vector whose length has an upper bound of 16.

Because states or transitions cannot read from or write to variable-size data, y does not appear in any state actions or transition logic. All computations involving variable-size data must occur in MATLAB functions in the chart.

MATLAB Function: generateOutput

MATLAB functions access variable-size, chart-level data directly. You do not pass the data as inputs or outputs to the functions. In this chart, the `generateOutput` function constructs the variable-size output vector `y` as a number sequence based on the input it receives.

```
function generateOutput(len)
%#codegen
assert(len<=16);
y = (1:len)';
```

MATLAB functions must be able to determine the upper bounds of variable-size data at compile time. In this case, however, the upper bound is `len`, an input for which the model computes the value at run time. To provide this information, the `assert` function specifies an explicit upper bound for `len` that matches the upper bound specified for the chart output `y`.

SizeBasedProcessing Chart

The SizeBasedProcessing chart uses three MATLAB functions to compute a variable-size output `y` based on the value of a variable-size input `u`.
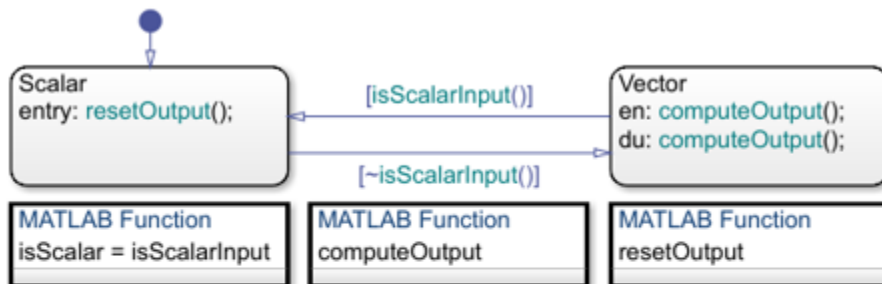
In this chart, MATLAB functions:
    (a) Determine the size of the input signal.
    (b) Use MATLAB syntax to compute variable-size outputs based on inputs.
    (c) Reset the output to zero when the input is scalar.

Key behavior:
    * Variable-size data is allowed only for chart input and output data.
    * Variable-size data "u" and "y" do not appear in state actions or on transitions.
    * MATLAB functions access variable-size data "u" and y" directly for computations.
    * MATLAB functions do not use variable dimensions for their own input and output data.

- Input `u` is the variable-size signal generated by the VarSizeSignalSource chart.
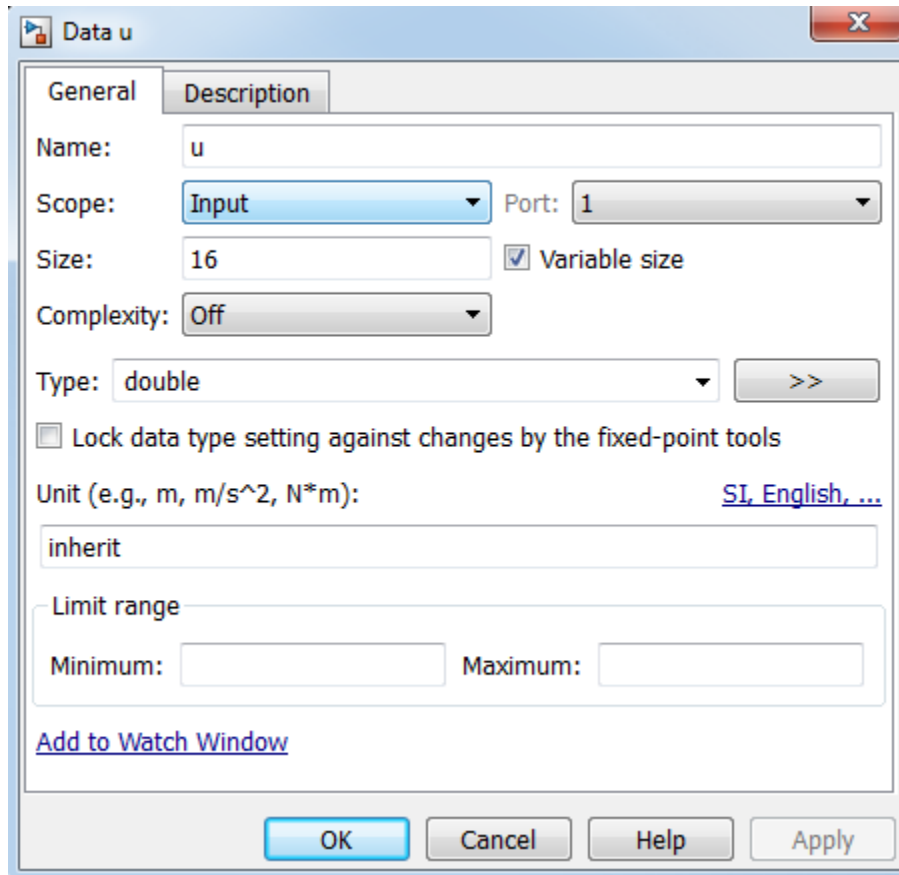
- Output y is a variable-size signal whose size depends on whether u is a scalar or vector.
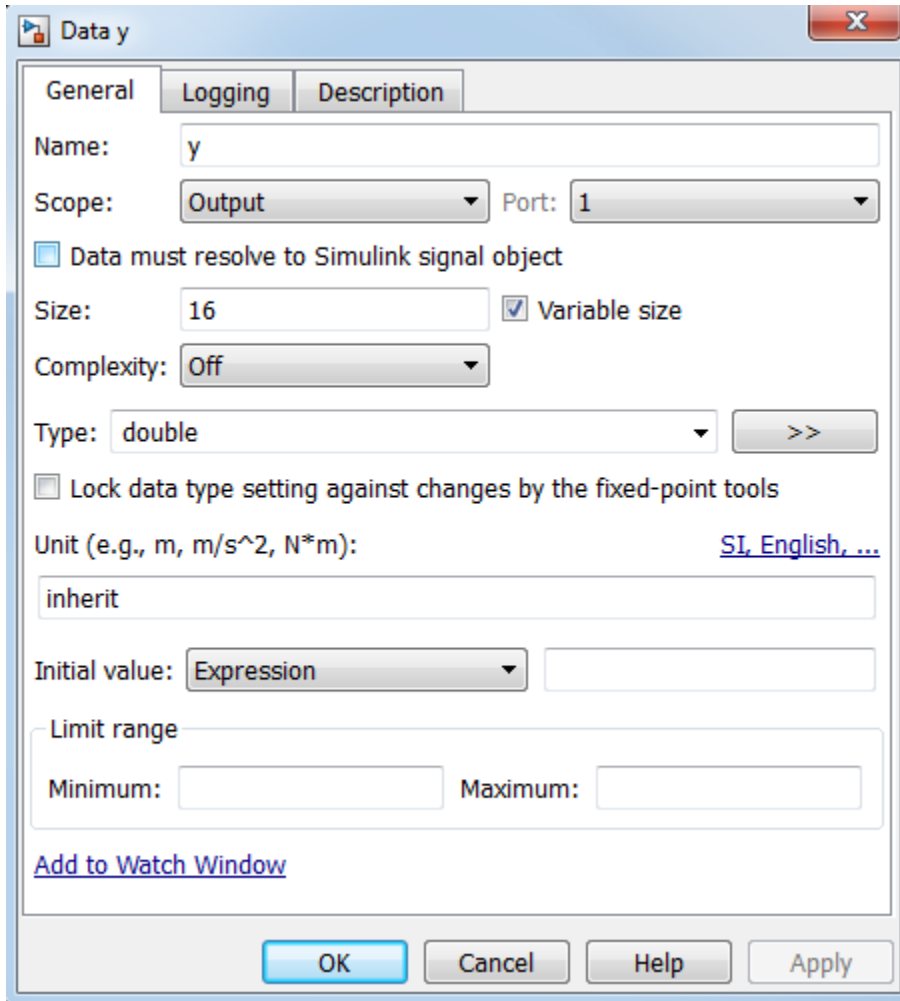
As in the chart VarSizeSignalSource, variable-size data does not appear in state actions or transition logic. Instead, states call MATLAB functions to compute the variable-size output. Transitions call a MATLAB function in a conditional statement to evaluate the variable-size input.

MATLAB Function: `isScalarInput`

This function tests whether chart input u, the signal generated by chart VarSizeSignalSource, is a scalar or vector value:

```
function isScalar = isScalarInput
%#codegen
isScalar = length(u)==1;
```

MATLAB Function: `computeOutput`

If input u is a vector, this function outputs the sine of each of its values:

```
function computeOutput
%#codegen
y = sin(u);
```

MATLAB Function: `resetOutput`

If input u is a scalar, this function outputs a value of zero:

```
function resetOutput
%#codegen
y = 0;
```

Simulate the Model

1 Open the model. The tabs along the top of the Editor canvas enable you to switch between the Simulink model and the two Stateflow charts.

2 Start simulation from the VarSizeSignalSource chart. The chart animation shows the active state cycling between the `Scalar`, `VectorPartial`, and `VectorFull` states.

3 Change tabs to view the SizeBasedProcessing chart. The chart animation shows the active state alternating between the `Scalar` and `Vector` states.

4 Change tabs to return to the Simulink model. The display blocks periodically show 1, 8, and 16 values from the variable-size vectors.

See Also

More About

*   "Declare Variable-Size Data in Stateflow Charts" on page 18-2

19

Enumerated Data in Charts

Reference Values by Name by Using Enumerated Data

To enhance the readability of a Stateflow chart, use enumerated data. With enumerated data, you can:

- Create a restricted set of values and refer to those values by name.
- Group related values into separate data types.
- Avoid defining a long list of constants.

Enumerated data is supported in Stateflow charts in Simulink models.

Example of Enumerated Data

An enumerated data type is a finite collection of *enumerated values* consisting of a name and an underlying integer value. For example, this chart uses enumerated data to refer to a set of colors.
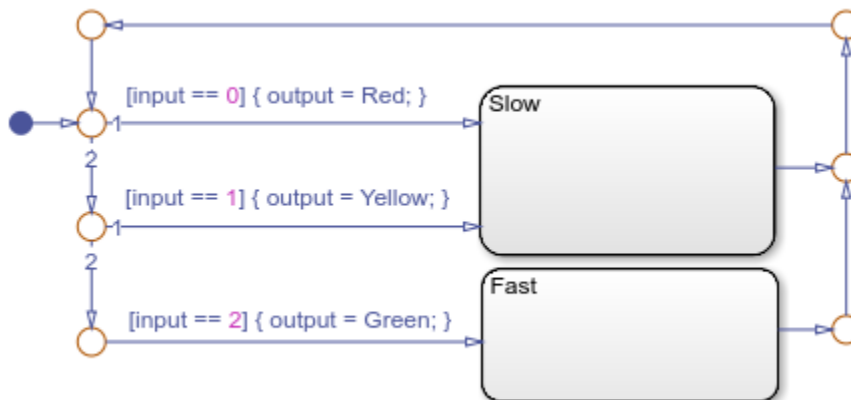
The enumerated data `output` is restricted to a finite set of values. You can refer to these values by their names: `Red`, `Yellow`, and `Green`.

Enumerated Value	Name	Integer Value
Red(0)	Red	0
Yellow(1)	Yellow	1

Enumerated Value	Name	Integer Value
Green(2)	Green	2

This MATLAB file defines the enumerated data type `BasicColors` referenced by the chart.

```
classdef BasicColors < Simulink.IntEnumType
  enumeration
    Red(0)
    Yellow(1)
    Green(2)
  end
end
```

Computation with Enumerated Data

An enumerated data type does not function as a numeric type despite the existence of the underlying integer values. You cannot use enumerated values directly in a mathematical computation. You can use enumerated data to control chart behavior based on assignments and comparisons. To assign or compare enumerated data, use the operations listed in this table.

Example	Description
a = exp	Assignment of a to exp, which must evaluate to an enumerated value
a == b	Comparison, equality
a != b	Comparison, inequality

Notation for Enumerated Values

To refer to an enumerated value, use prefixed or nonprefixed identifiers.

Prefixed Identifiers

To prevent name conflicts when referring to enumerated values in Stateflow charts, you can use prefixed identifiers of the form *Type*.*Name*. *Type* is an enumerated data type and *Name* is an enumerated value name. For example, suppose that you define three data types (`Colors`, `Temp`, and `Code`) that contain the enumerated name `Red`. By using prefixed notation, you can distinguish `Colors.Red` from `Temp.Red` and `Code.Red`.

Nonprefixed Identifiers

To minimize identifier length when referring to unique enumerated values, you can use nonprefixed enumerated value names. For example, suppose that the enumerated name `Red` belongs only to the data type `Colors`. You can then refer to this value with the nonprefixed identifier `Red`.

If your chart uses data types that contain identical enumerated names (such as `Colors.Red` and `Temp.Red`), use prefixed identifiers to prevent name conflicts.

Where to Use Enumerated Data

Use enumerated data at these levels of the Stateflow hierarchy:

- Chart
- Subchart
- State

Use enumerated data as arguments for:

- State actions
- Condition and transition actions
- Vector and matrix indexing
- MATLAB functions
- Graphical functions
- Simulink functions
- Truth Table blocks and truth table functions

If you have Simulink Coderinstalled, you can use enumerated data for simulation and code generation.

See Also

More About

- "Define Enumerated Data Types" on page 19-6

- "Assign Enumerated Values in a Chart" on page 19-15
- "Model Media Player by Using Enumerated Data" on page 19-20
- "Best Practices for Using Enumerated Data" on page 19-10
- "Simulink Enumerations" (Simulink)
- "Use Enumerated Data in Simulink Models" (Simulink)

Define Enumerated Data Types

To enhance the readability of a Stateflow chart, use enumerated data. With enumerated data, you can:

- Create a restricted set of values and refer to those values by name.
- Group related values into separate data types.
- Avoid defining a long list of constants.

Enumerated data is supported in Stateflow charts in Simulink models. For more information, see "Reference Values by Name by Using Enumerated Data" on page 19-2.

Before you can add enumerated data to a Stateflow chart, you must define an enumerated data type in a MATLAB class definition file. Create a different file for each enumerated type.

Elements of an Enumerated Data Type Definition

The enumerated data type definition consists of three sections of code.

Section of Code	Required?	Purpose
classdef	Yes	Provides the name of the enumerated data type
enumeration	Yes	Lists the enumerated values that the data type allows
methods	No	Provides methods that customize the data type

Define an Enumerated Data Type

1. Open a new file in which to store the data type definition. From the **Home** tab on the MATLAB toolstrip, select **New > Class**.
2. Complete the `classdef` section of the definition.

   ```
   classdef BasicColors < Simulink.IntEnumType
      ...
   end
   ```

   The `classdef` section defines an enumerated data type with the name `BasicColors`. Stateflow derives the data type from the built-in type `Simulink.IntEnumType`. The enumerated data type name must be unique among data type names and workspace variable names.

3 Define enumerated values in an `enumeration` section.

```
classdef BasicColors < Simulink.IntEnumType
  enumeration
    Red(0)
    Yellow(1)
    Green(2)
  end
end
```

An enumerated type can define any number of values. The `enumeration` section lists the set of enumerated values that this data type allows. Each enumerated value consists of a name and an underlying integer value. Each name must be unique within its type, but can also appear in other enumerated types. The default value is the first one in the list, unless you specify otherwise in the `methods` section of the definition.

4 (Optional) Customize the data type by using a `methods` section. The section can contain these methods:

- `getDefaultValue` specifies a default enumerated value other than the first one in the list of allowed values.

- `getDescription` specifies a description of the data type for code generated by Simulink Coder.

- `getHeaderFile` specifies custom header file that contains the enumerated type definition in code generated by Simulink Coder.

- `getDataScope` enables exporting or importing the enumerated type definition to or from a header file in code generated by Simulink Coder.

- `addClassNameToEnumNames` enhances readability and prevents name conflicts with identifiers in code generated by Simulink Coder.

For example, this MATLAB file presents a customized definition for the enumerated data type `BasicColors` that:

- Specifies that the default enumerated value is the last one in the list of allowed values.

- Includes a short description of the data type for code generated by Simulink Coder.

- Imports the definition of the data type from a custom header file to prevent Simulink Coder from generating the definition.

- Adds the name of the data type as a prefix to each enumeration member name in code generated by Simulink Coder.

```matlab
classdef BasicColors < Simulink.IntEnumType
  enumeration
    Red(0)
    Yellow(1)
    Green(2)
  end

  methods (Static = true)
    function retVal = getDefaultValue()
      % GETDEFAULTVALUE Specifies the default enumeration member.
      % Return a valid member of this enumeration class to specify the default.
      % If you do not define this method, Simulink uses the first member.
      retVal = BasicColors.Green;
    end

    function retVal = getDescription()
      % GETDESCRIPTION Specifies a string to describe this enumerated type.
      retVal = 'This defines an enumerated type for colors';
    end

    function retVal = getHeaderFile()
      % GETHEADERFILE Specifies the file that defines this type in generated code.
      % The method getDataScope determines the significance of the specified file.
      retVal = 'imported_enum_type.h';
    end

    function retVal = getDataScope()
      % GETDATASCOPE Specifies whether generated code imports or exports this type.
      % Return one of these strings:
      % 'Auto':     define type in model_types.h, or import if header file specifie
      % 'Exported': define type in a generated header file
      % 'Imported': import type definition from specified header file
      % If you do not define this method, DataScope is 'Auto' by default.
      retVal = 'Imported';
    end

    function retVal = addClassNameToEnumNames()
      % ADDCLASSNAMETOENUMNAMES Specifies whether to add the class name
      % as a prefix to enumeration member names in generated code.
      % Return true or false.
      % If you do not define this method, no prefix is added.
```

```
        retVal = true;
    end % function
  end % methods
end % classdef
```

5   Save the file on the MATLAB path. The name of the file must match exactly the name of the data type. For example, the definition for the data type `BasicColors` must reside in a file named `BasicColors.m`.

> **Tip** To add a folder to the MATLAB search path, type `addpath` *pathname* at the command prompt.

Specify Data Type in the Property Inspector

When you add enumerated data to your chart, specify its type in the Property Inspector.

1   In the **Type** field, select `Enum: <class name>`.

2   Replace `<class name>` with the name of the data type. For example, you can enter `Enum: BasicColors` in the **Type** field.

3   (Optional) Enter an initial value for the enumerated data by using a prefixed identifier. The initial value must evaluate to a valid MATLAB expression.

See Also

More About

- "Assign Enumerated Values in a Chart" on page 19-15
- "Model Media Player by Using Enumerated Data" on page 19-20
- "Best Practices for Using Enumerated Data" on page 19-10
- "Use Enumerated Data in Simulink Models" (Simulink)
- "Customize Simulink Enumeration" (Simulink)

Best Practices for Using Enumerated Data

To enhance the readability of a Stateflow chart, use enumerated data. With enumerated data, you can:

- Create a restricted set of values and refer to those values by name.
- Group related values into separate data types.
- Avoid defining a long list of constants.

Enumerated data is supported in Stateflow charts in Simulink models. For more information, see "Reference Values by Name by Using Enumerated Data" on page 19-2.

Guidelines for Defining Enumerated Data Types

Use Unique Name for Each Enumerated Type

To avoid name conflicts, the name of an enumerated data type cannot match the name of another data type or a variable in the MATLAB base workspace.

Use Same Name for Enumerated Type and Class Definition File

To enable resolution of enumerated data types for Simulink models, the name of the MATLAB file that contains the type definition must match the name of the data type.

Apply Changes in Enumerated Type Definition

When you update an enumerated data type definition for an open model, the changes do not take effect immediately. To see the effects of updating a data type definition:

1 Save and close the model.
2 Delete all instances of the data type from the MATLAB base workspace. To find these instances, type `whos` at the command prompt.
3 Open the model and start simulation or generate code by using Simulink Coder.

Guidelines for Referencing Enumerated Data

Ensure Unique Name Resolution for Nonprefixed Identifiers

If you use nonprefixed identifiers to refer to enumerated values in a chart, ensure that each enumerated name belongs to a unique enumerated data type.

Use Unique Identifiers for Enumerated Values

If an enumerated value uses the same identifier as a data object or a bus field, the chart does not resolve the identifier correctly. For example, this diagram shows the stages in which a chart tries to resolve the identifier `Colors.Red`.
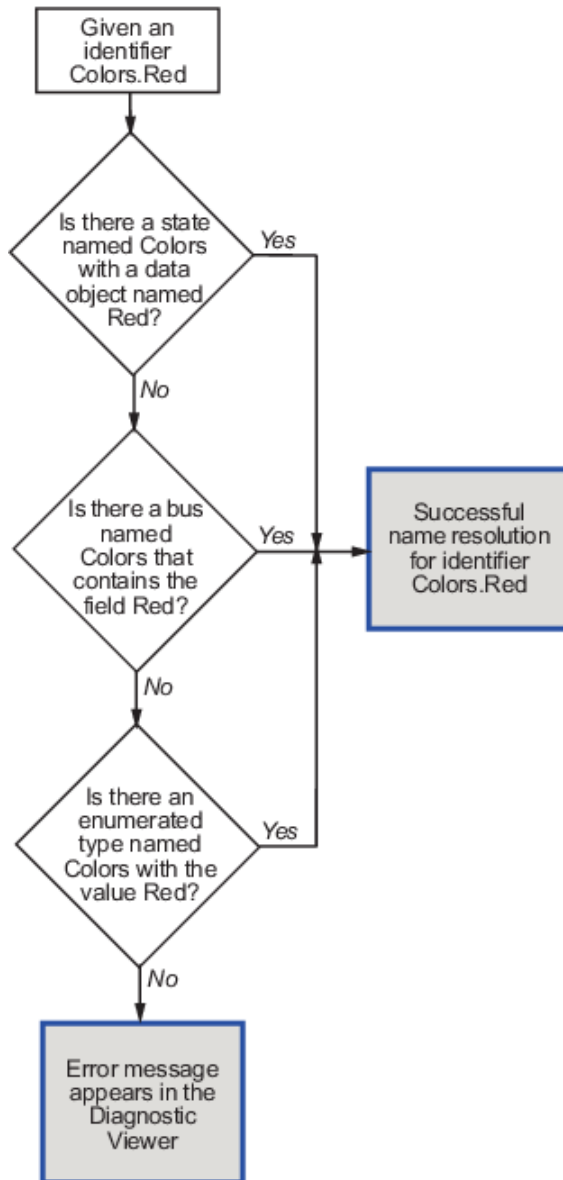
Set Initial Values of Enumerated Data by Using Prefixed Identifiers

If you choose to set an initial value for enumerated data, you must use a prefixed identifier in the **Initial value** field of the Property Inspector. For example, `BasicColors.Red` is a valid identifier, but `Red` is not. The initial value must evaluate to a valid MATLAB expression.

Enhance Readability of Generated Code by Using Prefixed Identifiers

If you add prefixes to enumerated names in the generated code, you enhance readability and avoid name conflicts with global symbols. For details, see "Use Enumerated Data in Generated Code" (Simulink Coder).

Guidelines and Limitations for Enumerated Data

Do Not Enter Minimum or Maximum Values for Enumerated Data

For enumerated data, leave the **Minimum** and **Maximum** fields of the Property Inspector empty. The chart ignores any values that you enter in these fields.

Whether these fields appear in the Property Inspector depends on which **Type** field option you use to define enumerated data.

Type Field Option	Appearance of the Minimum and Maximum Fields
`Enum: <class name>`	Not available
`<data type expression>` or `Inherit from Simulink`	Available

Do Not Assign Enumerated Values to Constant Data

Because enumerated values are constants, assigning these values to constant data is redundant and unnecessary. If you try to assign enumerated values to constant data, an error appears.

Do Not Use ml Namespace Operator to Access Enumerated Data

The `ml` operator does not support enumerated data.

Do Not Define Enumerated Data at Machine Level of Hierarchy

Machine-parented data is not supported for enumerated types.

See Also

More About
- "Reference Values by Name by Using Enumerated Data" on page 19-2
- "Define Enumerated Data Types" on page 19-6
- "Assign Enumerated Values in a Chart" on page 19-15
- "Model Media Player by Using Enumerated Data" on page 19-20

Assign Enumerated Values in a Chart

To enhance the readability of a Stateflow chart, use enumerated data. With enumerated data, you can:

- Create a restricted set of values and refer to those values by name.
- Group related values into separate data types.
- Avoid defining a long list of constants.

Enumerated data is supported in Stateflow charts in Simulink models. For more information, see "Reference Values by Name by Using Enumerated Data" on page 19-2.

Chart Behavior

This example shows how to build a chart that uses enumerated values to issue a status keyword.
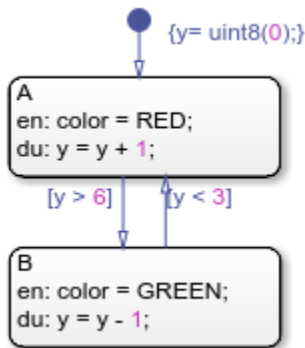
During simulation, the chart action alternates between states A and B.

Execution of State A

- At the start of the simulation, state A is entered.
- State A executes the `entry` action by assigning the value RED to the enumerated data `color`.
- The data y increments once per time step (every 0.2 seconds) until the condition `[y > 6]` is true.

- The chart takes the transition from state A to state B.

Execution of State B

- After the transition from state A occurs, state B is entered.
- State B executes the `entry` action by assigning the value `GREEN` to the enumerated data `color`.
- The data `y` decrements once per time step (every 0.2 seconds) until the condition `[y < 3]` is true.
- The chart takes the transition from state B back to state A.

Build the Chart

Add States and Transitions to the Chart

1 To create a Simulink model with an empty chart, at the MATLAB command prompt, enter `sfnew`.

2 In the empty chart, add states A and B. At the text prompt, enter the appropriate action statements.

3 Add a default transition to state A and transitions between states A and B.

4 Double-click each transition. At the text prompt, enter the appropriate condition.

Define an Enumerated Data Type for the Chart

1 To create a file in which to store the data type definition, from the **Home** tab on the MATLAB toolstrip, select **New > Class**.

2 In the MATLAB Editor, enter:

```
classdef TrafficColors < Simulink.IntEnumType
  enumeration
    RED(0)
    GREEN(10)
  end
end
```

The `classdef` section defines an integer-based enumerated data type named `TrafficColors`. The `enumeration` section contains the enumerated values that this data type allows followed by their underlying numeric value.

3 Save your file as `TrafficColors.m` in a folder on the MATLAB search path.

Define Chart Data

1   To resolve the undefined data, in the Symbols window, click the **Resolve undefined symbols** icon . The Stateflow Editor assigns an appropriate scope to each symbol in the chart.

Symbol	Scope
`color`	Output Data
`y`	Local Data
`GREEN`	Parameter Data
`RED`	Parameter Data

2   To specify `color` as enumerated data, in the Property Inspector:

- In the **Type** field, select `Enum: <class name>`. Replace `<class name>` with `TrafficColors`, the name of the data type that you defined previously.
- Under **Logging**, select the **Log signal data** check box.

3   To set the scope and type of `y`, in the Property Inspector:

- In the **Scope** field, select `Output`.
- In the **Type** field, select `uint8`.
- Under **Logging**, select the **Log signal data** check box.

4   In the Symbols window, delete the symbols `GREEN` and `RED`. The Stateflow Editor incorrectly identified these symbols as parameters before you specified `color` as enumerated data.

View Simulation Results

1
When you simulate the model, the Simulation Data Inspector icon is highlighted to indicate that it has new simulation data. To open the Simulation Data Inspector, click the icon.

2   In the Simulation Data Inspector, select the check boxes for the `color` and `y` signals so that they are displayed on separate axes.
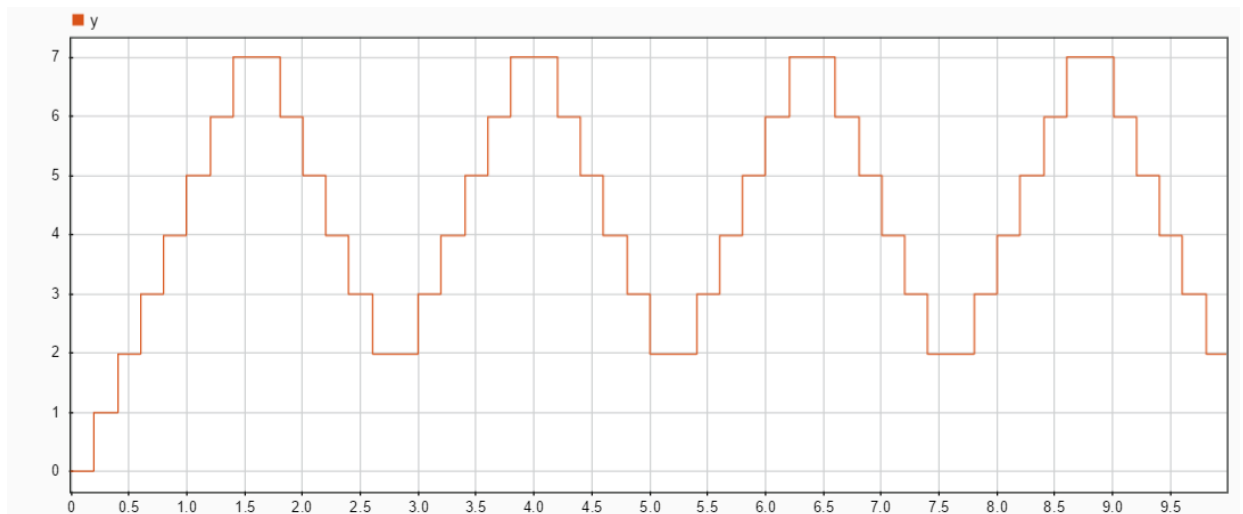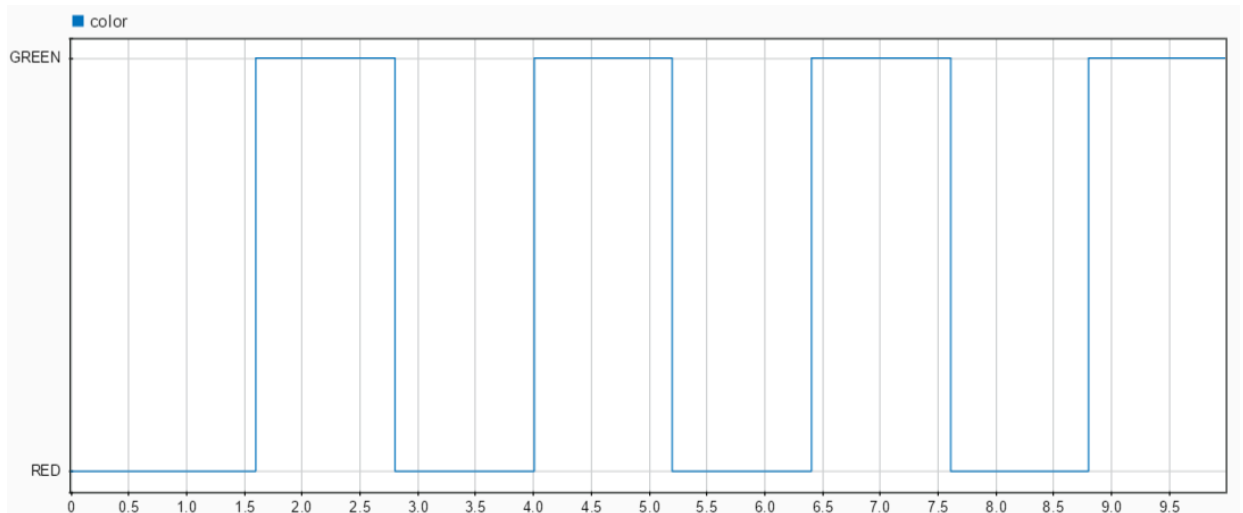
3 To access the logged data in the MATLAB workspace, call the signal logging object `logsout`. For example, at the command prompt, enter:

```
A = table(logsout.getElement('color').Values.Time, ...
    logsout.getElement('color').Values.Data);
A.Properties.VariableNames = {'SimulationTime','Color'};
A
```

```
A =

  9×2 table

    SimulationTime     Color
    _____     _____

               0       RED
             1.6       GREEN
             2.8       RED
               4       GREEN
             5.2       RED
             6.4       GREEN
             7.6       RED
             8.8       GREEN
              10       RED
```

See Also

More About

- "Define Enumerated Data Types" on page 19-6
- "Model Media Player by Using Enumerated Data" on page 19-20
- "Best Practices for Using Enumerated Data" on page 19-10
- "Access Signal Logging Data" on page 32-54
- "View Data with the Simulation Data Inspector" (Simulink)

Model Media Player by Using Enumerated Data

Model a media player by using enumerated data in three Stateflow charts.
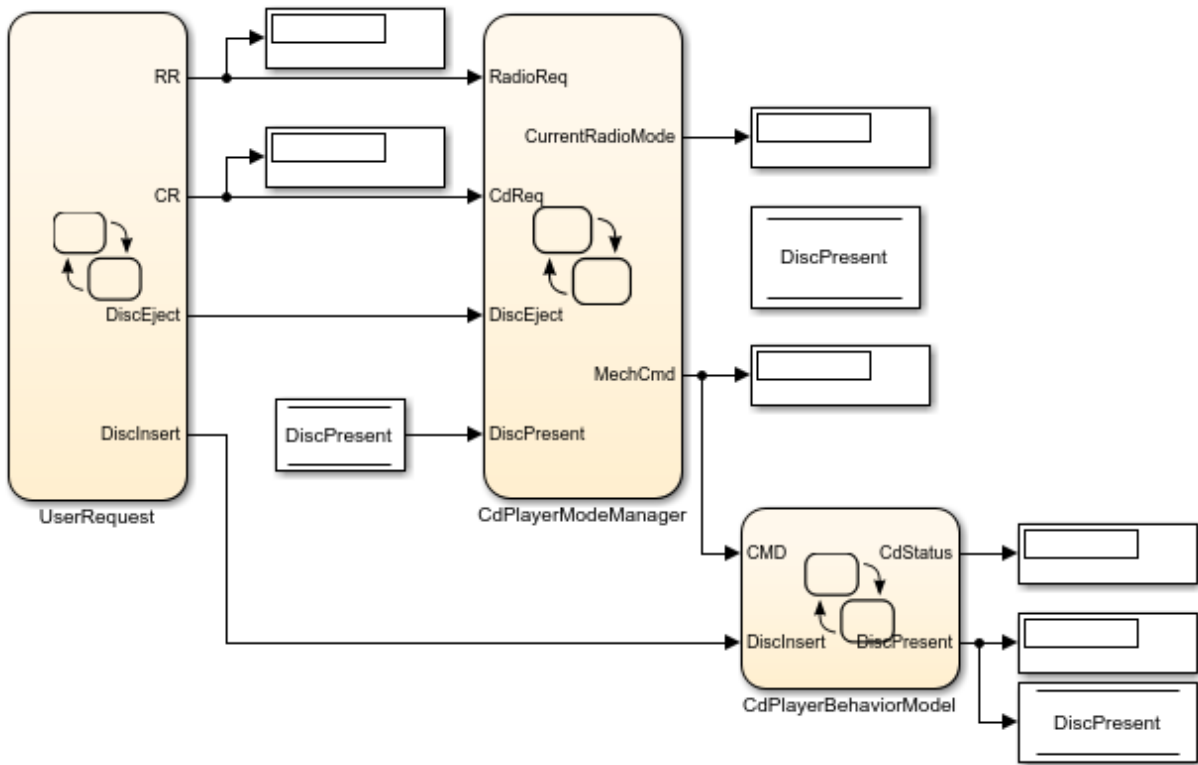
Model Component	Description
"UserRequest Chart" on page 19-23	Reads and stores user inputs from UI
"CdPlayerModeManager Chart" on page 19-24	Determines whether the media player operates in AM radio, FM radio, or CD player mode
"CdPlayerBehaviorModel Chart" on page 19-27	Describes behavior of the CD player component

This model uses two enumerated data types: `RadioRequestMode` and `CdRequestMode`. By grouping related values into separate data types:

- You enhance the readability of data values in each chart.
- You avoid defining a long list of constants and reduce the amount of data in your model.

Enumerated Data Type	Enumerated Values
`RadioRequestMode`	• `OFF(0)` • `CD(1)` • `FM(2)` • `AM(3)`
`CdRequestMode`	• `EMPTY(-2)` • `DISCINSERT(-1)` • `STOP(0)` • `PLAY(1)` • `REW(3)` • `FF(4)` • `EJECT(5)`

Run the Media Player Model

1 Start simulation of the model. The CD Player Helper appears.
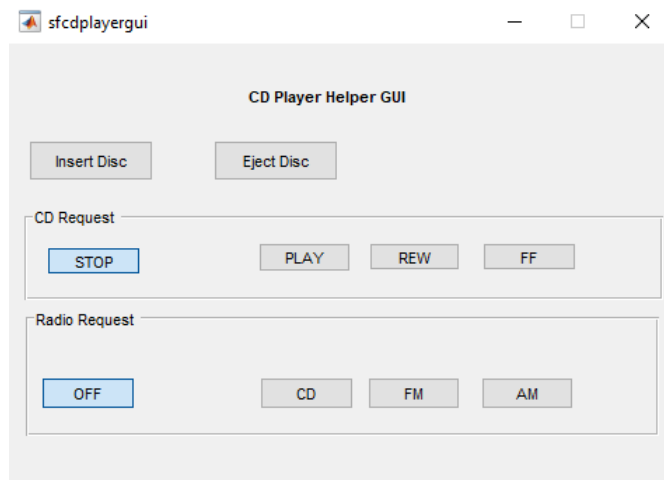
In the model, the Display blocks show the default settings of the media player.
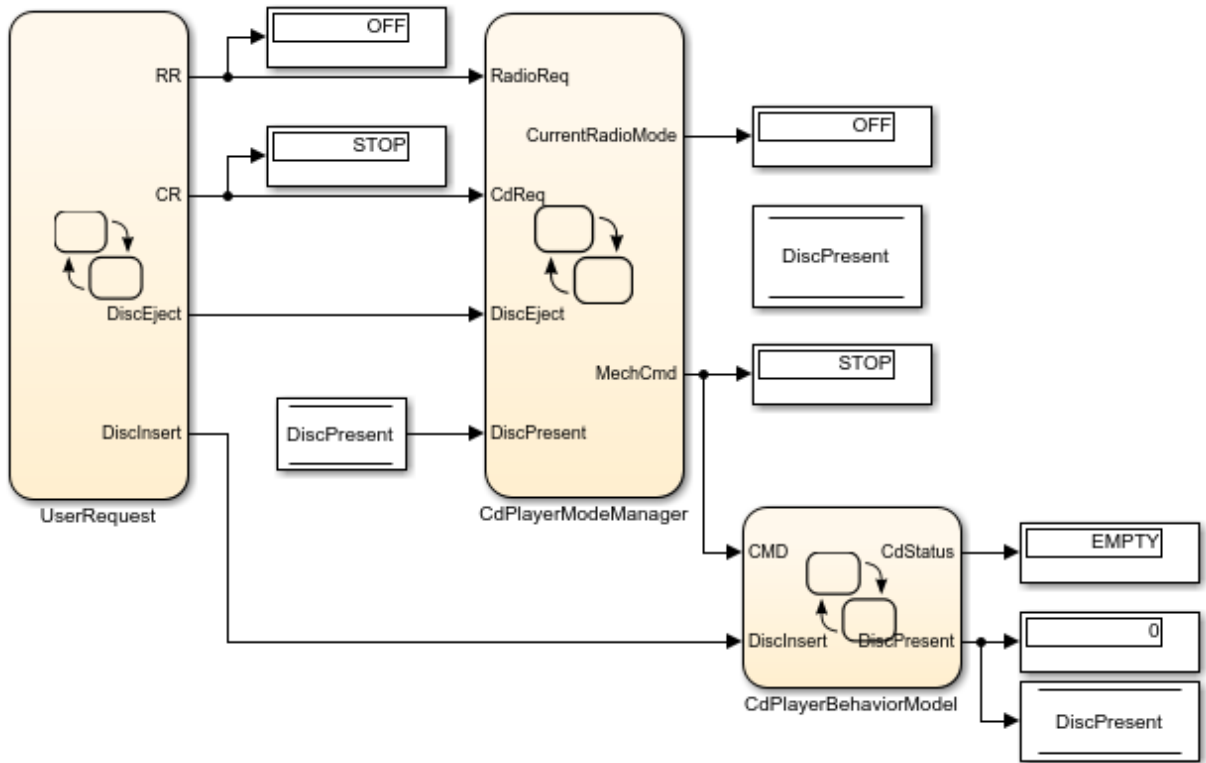
2. In the **Radio Request** section, click **CD**. The Display blocks for enumerated data RR and `CurrentRadioMode` change from `OFF` to `CD`.

3. Click **Insert Disc**. The Display block for enumerated data `CdStatus` changes from `EMPTY` to `DISCINSERT` to `STOP`.

4. In the **CD Request** section, click the **PLAY**. The Display blocks for enumerated data CR, `MechCmd`, and `CdStatus` change from `STOP` to `PLAY`.

5. To see other changes in the Display blocks, use the CD Player Helper to select other operating modes for the media player.

UserRequest Chart

This chart reads user inputs from the CD Player Helper and stores the information as output data.
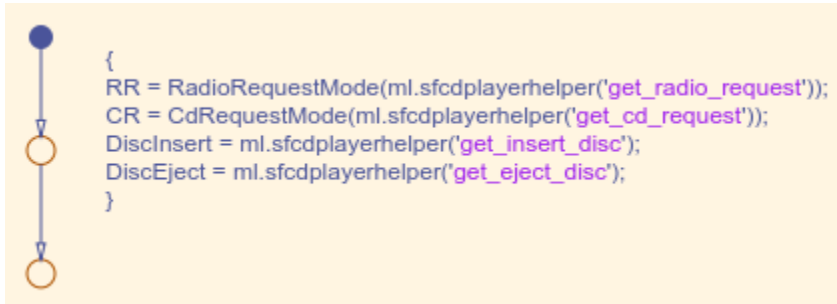
Key Features of the Chart

- Enumerated data RR and CR
- `ml` namespace operator to access MATLAB function `sfcdplayerhelper`

Chart Behavior

The chart calls the function `sfcdplayerhelper` on the MATLAB path, reads your interaction with the UI, and stores it as output data.

Output Data Name	Data Type	Button Selection	Description
RR	Enumerated	Buttons in the **Radio Request** section	Media player subcomponent to activate
CR	Enumerated	Buttons in the **CD Request** section	Operating mode of the CD player
DiscInsert	Boolean	**Insert Disc**	Setting for CD insertion
DiscEject	Boolean	**Eject Disc**	Setting for CD ejection

CdPlayerModeManager Chart

This chart activates the appropriate subcomponent of the media player depending on the inputs received from the UserRequest chart.
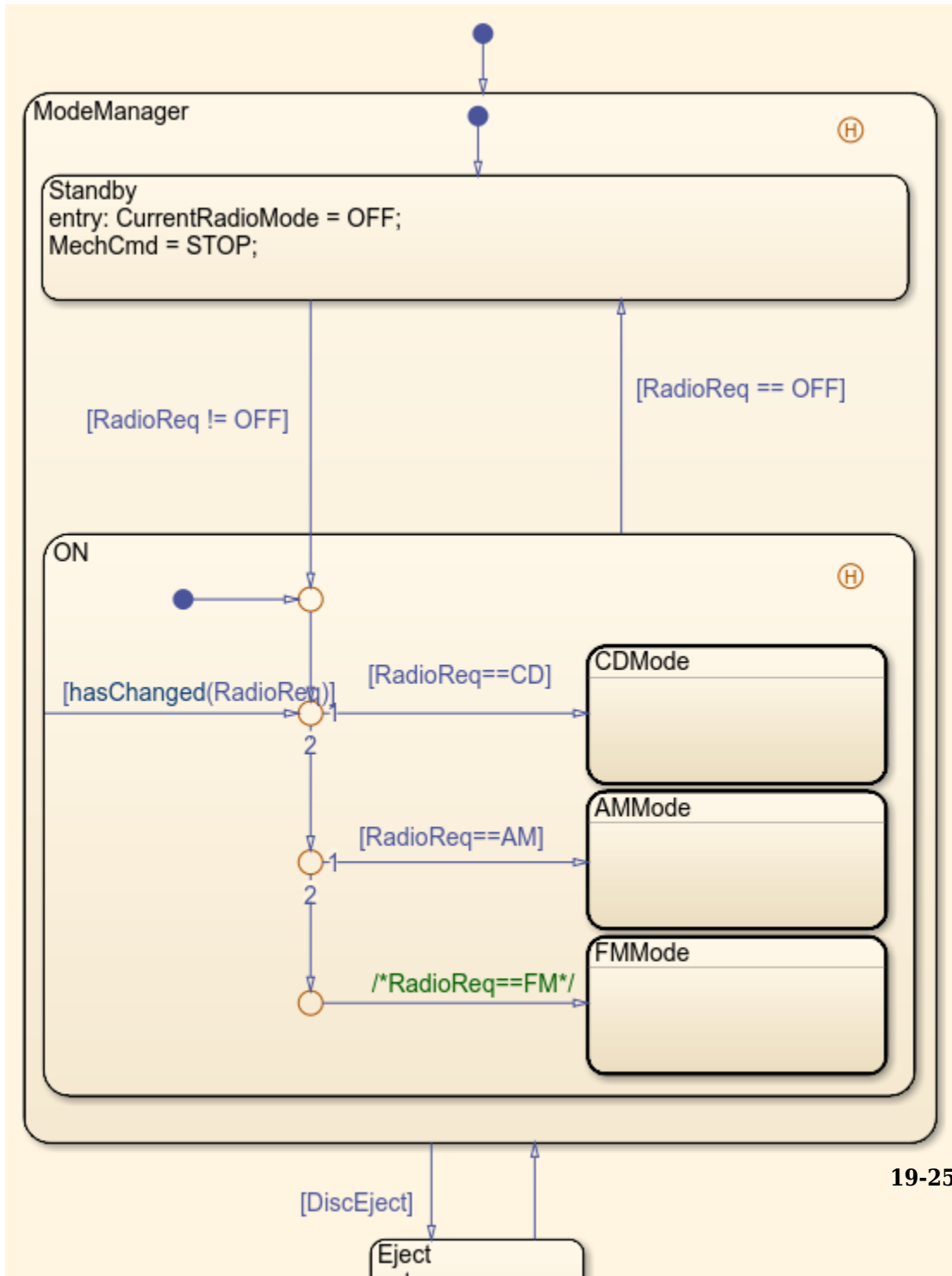
Key Features of the Chart

- Enumerated data `RadioReq`, `CdReq`, `CurrentRadioMode`, and `MechCmd`
- `hasChanged` operator to detect changes in the value of `RadioReq`
- Subcharts `CdMode`, `AMMode`, and `FMMode`

Chart Behavior

At the start of simulation, the `ModeManager` state becomes active. If the Boolean data `DiscEject` is 1 (or `true`), a transition to the `Eject` state occurs, followed by a transition back to the `ModeManager` state.

When `ModeManager` is active, the previously active substate (`Standby` or `ON`) as recorded by the history junction becomes active. Subsequent transitions between the `Standby` and `ON` substates depend on the enumerated data `RadioReq`:

- If `RadioReq` is `OFF`, the `Standby` substate is activated.
- If `RadioReq` is not `OFF`, the `ON` substate is activated.

In the `ON` substate, three subcharts represent the operating modes of the media player: CD player, AM radio, and FM radio. Each subchart corresponds to a different value of enumerated data `RadioReq`. The inner transition inside the `ON` state continually scans for changes in the value of `RadioReq`.

Value of Enumerated Data RadioReq	Active Subchart	Purpose of Subchart
CD	CDMode	Sets the media player to CD player mode. Outputs PLAY, REW, FF, and STOP commands to the CdPlayerBehaviorModel chart.
AM	AMMode	Sets the media player to AM radio mode. Outputs a STOP command to the CdPlayerBehaviorModel chart.

Value of Enumerated Data RadioReq	Active Subchart	Purpose of Subchart
FM	FMMode	Sets the media player to FM radio mode. Outputs a STOP command to the CdPlayerBehaviorModel chart.

CdPlayerBehaviorModel Chart

This chart activates the appropriate operating mode for the CD player depending on the input received from the CdPlayerBehaviorModel chart.
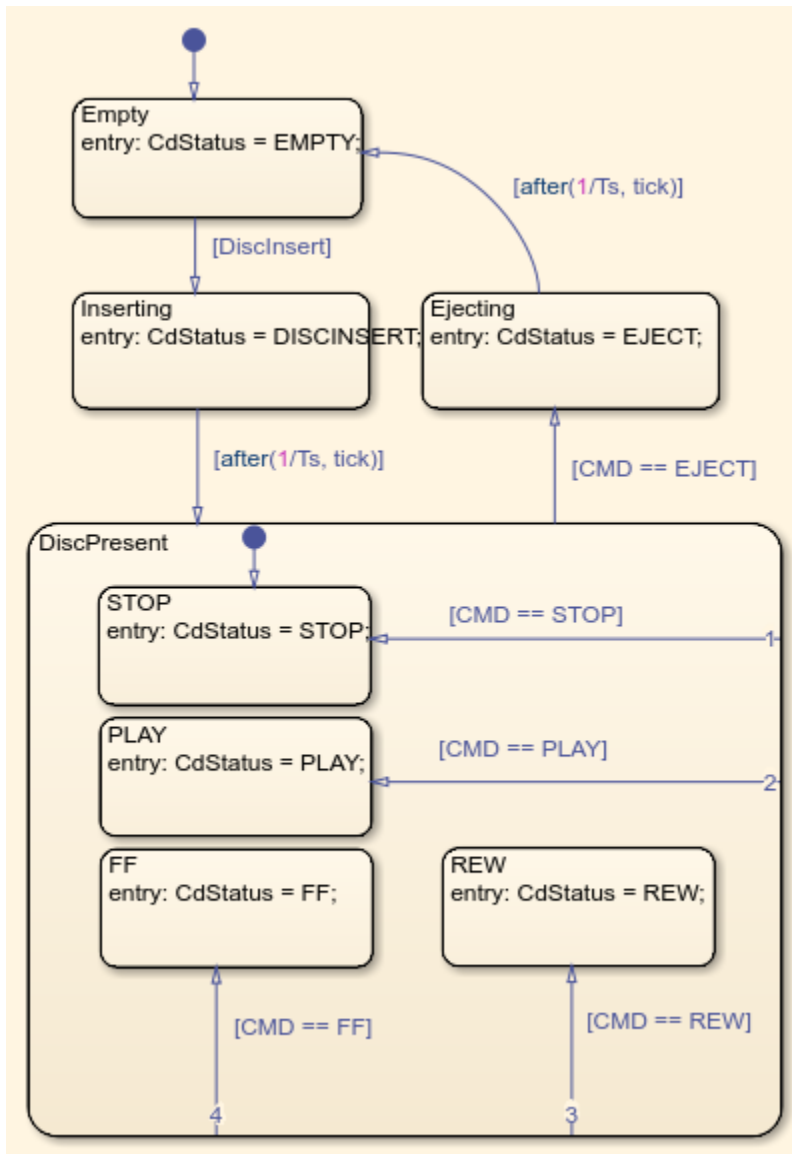
Key Features of the Chart

- Enumerated data CMD and CdStatus

- `after` temporal logic operator to control timing of transitions during disc insertion and ejection

Chart Behavior

At the start of simulation, the `Empty` state is activated.

If the Boolean data `DiscInsert` is 1 (or `true`), a transition to the `Inserting` state occurs. After a short time delay, a transition to the `DiscPresent` state occurs.

The `DiscPresent` state remains active until the data `CMD` becomes `EJECT`. At that point, a transition to the `Ejecting` state occurs. After a short time delay, a transition to the `Empty` state occurs.

Whenever a state transition occurs, the enumerated data `CdStatus` changes value to reflect the status of the CD player.

Active State	Value of Enumerated Data CdStatus	Behavior of CD Player
Empty	EMPTY	CD player is empty.
Inserting	DISCINSERT	CD is being inserted into the player.
Ejecting	EJECT	CD is being ejected from the player.
DiscPresent.STOP	STOP	CD is present and stopped.
DiscPresent.PLAY	PLAY	CD is present and playing.
DiscPresent.REW	REW	CD is present and rewinding.
DiscPresent.FF	FF	CD is present and fast forwarding.

See Also

More About

- "Reference Values by Name by Using Enumerated Data" on page 19-2

- "Define Enumerated Data Types" on page 19-6
- "Assign Enumerated Values in a Chart" on page 19-15
- "Best Practices for Using Enumerated Data" on page 19-10
- "Access MATLAB Functions and Workspace Data in C Charts" on page 12-32
- "Detect Changes in Data Values" on page 12-69
- "Control Chart Execution by Using Temporal Logic" on page 12-49

20

String Data in Charts

Manage Textual Information by Using Strings

To use textual data to control chart behavior and to manipulate text to create natural language output, use *strings*. String data is available only in Stateflow charts that use C as the action language.

Example of String Data

In Stateflow, a string is a piece of text surrounded by quotes ("…" or '…'). For example, this chart takes string data as input. Based on that input, the chart produces a corresponding string output.
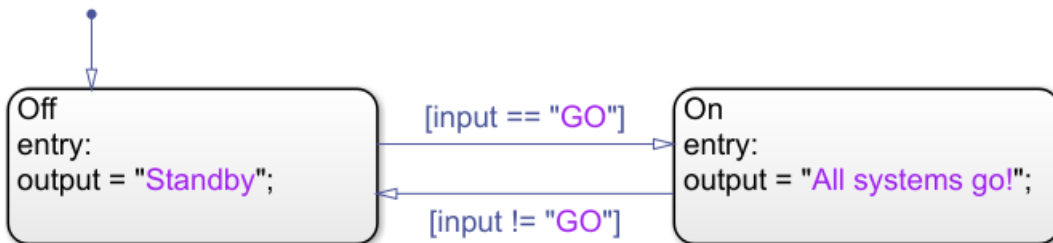
To specify a string symbol, set its **Type** field to `string`. Stateflow dynamically allocates memory space for this type of data.

Alternatively, you can create string data with a maximum number of characters. To specify a string symbol with a buffer size of *n* characters, set its **Type** field to `stringtype(n)`. The text of the string can be shorter than the buffer, but if it exceeds the buffer size, then the text in the string is truncated. For instance, if the symbol `output` in the previous chart is specified as `stringtype(10)`, then its value in the state `On` is truncated to `"All system"`. Depending on the configuration parameter **String truncation checking**, you can halt simulation and diagnose truncation of string data.

String Truncation Checking	Description
`error`	Simulation stops with an error.
`warning`	String is truncated. Simulation continues with a warning.

String Truncation Checking	Description
none	String is truncated. Simulation continues with no error or warning.

Note Unlike C or C++, Stateflow interprets escape sequences as literal characters. For example, the string "\n" contains two characters, backslash and n, and not a newline character.

Computation with Strings

To manipulate string data in a Stateflow chart, use the operators listed in this table.

Operator	Syntax	Description	Example
strcpy	dest = src	Assigns string src to dest.	Assigns string data to s1 and s2: s1 = 'hello'; s2 = "good bye";

	strcpy(dest,src)	An alternative way to execute dest = src.	Assigns string data to s3 and s4:
			
			strcpy(s3,'howdy');
			strcpy(s4,"so long");

strcat	dest = strcat(s1,...,sN)	Concatenates strings s1,...,sN.	Concatenates strings to form "Stateflow": s1 = "State"; s2 = "flow"; dest = strcat(s1,s2);
substr	dest = substr(str,i,n)	Returns the substring of length n starting at the i-th character of string str. Use zero-based indexing.	Extracts substring "Stateflow" from a longer string: str = "Stateflow rule the waves dest = substr(str,0,9);

Operator	Syntax	Description	Example

tostring	dest = tostring(X)	Converts numeric, Boolean, or enumerated data to string.	Converts numeric value to string "1.2345":
			
			`dest = tostring(1.2345);`
			
			Converts Boolean value to string "true":
			
			`dest = tostring(1==1);`
			
			Converts enumerated value to string "RED":
			
			`dest = tostring(RED);`

Operator	Syntax	Description	Example

strcmp	tf = strcmp(s1,s2)	Compares strings s1 and s2. Returns 0 if the two strings are identical. Otherwise returns a nonzero integer.
		
		• The sign of the output value depends on the lexicographic ordering of the input strings s1 and s2.
		• The magnitude of the output value depends on the compiler that you use. This value can differ in simulation and generated code.
		
		Strings are considered identical when they have the same size and content.
		
		This operator is consistent with the C library function strcmp or the C++ function string.compare, depending on the compiler that you select for code generation. The operator behaves differently than the function strcmp in MATLAB.
			Returns a value of 0 (strings are equal):
			
			`tf = strcmp("abc","abc");`
			
			Returns a nonzero value (strings are not equal):
			
			`tf = strcmp("abc","abcd");`

	s1 == s2	An alternative way to execute strcmp(s1,s2) == 0.	Returns a value of true: `"abc" == "abc";`
	s1 != s2	An alternative way to execute strcmp(s1,s2) != 0.	Returns a value of true: `"abc" != "abcd";`

Operator	Syntax	Description	Example
	`tf = strcmp(s1,s2,n)`	Returns 0 if the first n characters in s1 and s2 are identical.	Returns a value of 0 (substrings are equal): `tf = strcmp("abc","abcd",3);`
strlen	`L = strlen(str)`	Returns the number of characters in the string str.	Returns a value of 9: `L = strlen("Stateflow");`
str2double	`X = str2double(str)`	Converts the text in string str to a double-precision value. str contains text that represents a number. Text that represents a number can contain: • Digits • A decimal point • A leading + or - sign • An e preceding a power of 10 scale factor If str2double cannot convert text to a number, then it returns a NaN value.	Returns a value of -12.345: `X = str2double("-12.345");` Returns a value of 123400: `X = str2double("1.234e5");`
str2ascii	`A = str2ascii(str,n)`	Returns array of type uint8 containing ASCII values for the first n characters in str, where n is a positive integer. Use of variables or expressions for n is not supported.	Returns uint8 array {72,101,108,108,111}: `A = str2ascii("Hello",5);`

Operator	Syntax	Description	Example
ascii2str	dest = ascii2str(A)	Converts ASCII values in array A of type uint8 to string.	Returns string "Hi!": A[0] = 72; A[1] = 105; A[2] = 33; dest = ascii2str(A);

Where to Use Strings

Use string data at these levels of the Stateflow hierarchy:

- Chart
- Subchart
- State

Use string data as arguments for:

- State actions
- Condition and transition actions
- Graphical functions
- Simulink functions
- Truth table functions that use C as the action language

If you have Simulink Coder installed, you can use string data for simulation and code generation.

See Also
ascii2str | str2ascii | str2double | strcat | strcmp | strcpy | strlen | substr | tostring

More About

- "Log String Data to the Simulation Data Inspector" on page 20-9
- "Send Messages with String Data" on page 20-14
- "Share String Data with Custom C Code" on page 20-17
- "Simulate a Media Player by Using Strings" on page 20-22

- "Simulink Strings" (Simulink)

Log String Data to the Simulation Data Inspector

This example shows how to build a Stateflow chart that, based on numeric input data, concatenates string data into natural language output text. You can view the output text in the Simulation Data Inspector and in the MATLAB workspace. For more information on string data, see "Manage Textual Information by Using Strings" on page 20-2.
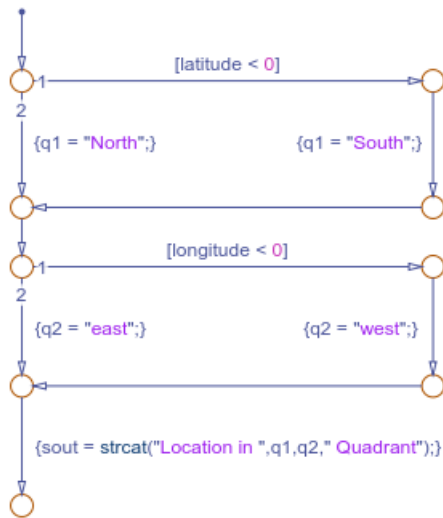
Chart Behavior

During simulation, Sine Wave blocks provide the position of a point moving along a closed path in terms of latitude and longitude coordinates. The chart examines these coordinates and assigns the strings q1 and q2 according to the information in this table.

Latitude	Longitude	q1	q2
positive	positive	"North"	"east"
positive	negative	"North"	"west"
negative	positive	"South"	"east"
negative	negative	"South"	"west"

Then, the statement

```
sout = strcat("Location in ",q1,q2," Quadrant");
```

concatenates these strings into an output string.

Build the Model

Add Junctions and Transitions

1 To create a Simulink model with an empty chart that uses the C action language, at the MATLAB command prompt, enter

   ```
   sfnew -c
   ```

2 In the empty chart, place a default transition. A junction appears. Point and drag from the edge of the junction to add other transitions and junctions.

3 Double-click each transition. At the text prompt, enter the appropriate condition or action statement.

Define Chart Data

1 To resolve the undefined data, in the Symbols window, click the **Resolve undefined symbols** icon ⊕. The Stateflow Editor assigns an appropriate scope to each symbol in the chart.

Symbol	Scope
latitude	Input Data
longitude	Input Data
q1	Local Data
q2	Local Data
sout	Output Data

2 To specify q1 as string data, in the **Type** field of the Property Inspector, select string. Repeat that specification for q2 and sout.

   Alternatively, to create string data with a maximum number of characters, specify each string as stringtype(n) using a suitable buffer size n to avoid truncation of its contents. For instance, this table lists suitable buffer sizes for the string data in the chart.

Symbol	Number of Characters	String Data Type
q1	5	stringtype(5)
q2	5	stringtype(5)
sout	30	stringtype(30)

Add Sources and Sinks to the Model

1   In the Simulink model, add two Sine Wave blocks and a Display block. Connect the blocks to the chart input and output ports.
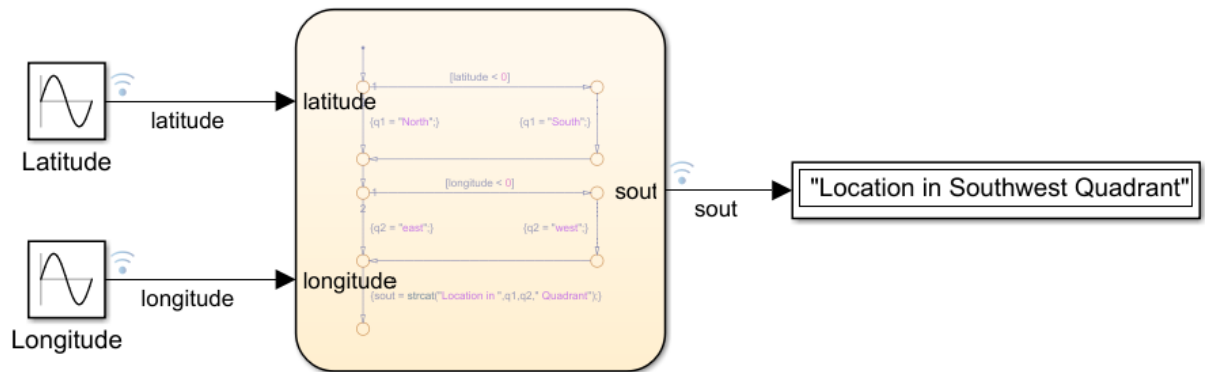
2   Set the Sine Wave block parameters as indicated in this table.

Block	Amplitude	Bias	Frequency	Phase
Latitude	50	0	1	0
Longitude	50	0	1	pi/2

3   Label the signals in the model as `latitude`, `longitude`, and `sout`. Right-click each signal and select **Log Selected Signals**.

View Simulation Results

1
When you simulate the model, the Simulation Data Inspector icon 📈 is highlighted to indicate that it has new simulation data. To open the Simulation Data Inspector, click the icon.

2   In the Simulation Data Inspector, select the check boxes for the `latitude`, `longitude`, and `sout` signals so that they are displayed on the same set of axes. The `latitude` and `longitude` signals appear as sinusoidal curves. The `sout` signal is shown as a transition plot. The value of the string is displayed inside a band and criss-crossed lines mark the changes in value.
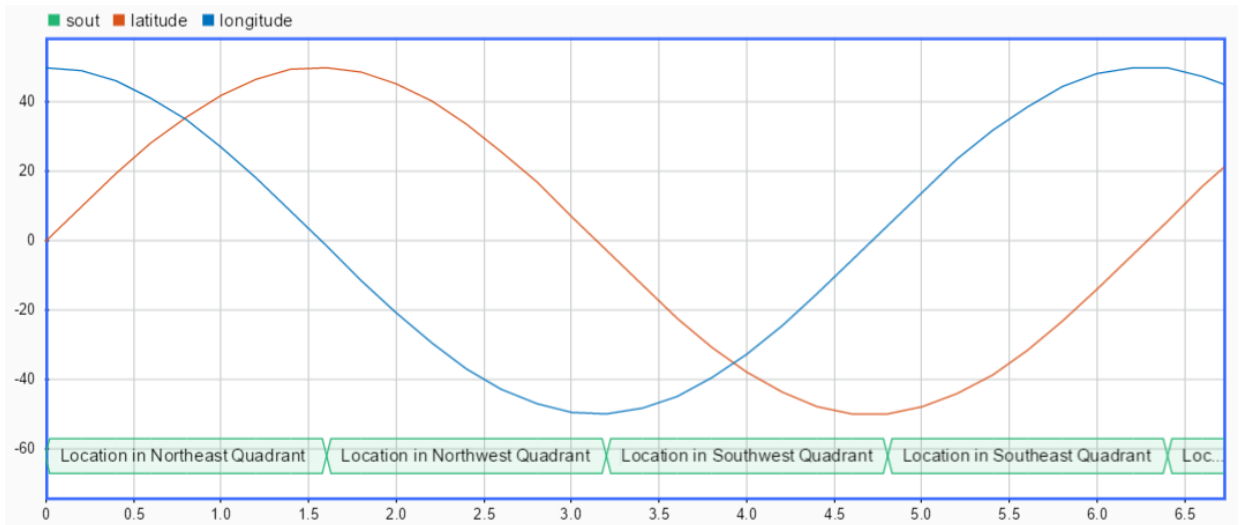
3   To access the logged data in the MATLAB workspace, call the signal logging object `logsout`. Stateflow exports the string data `sout` as a MATLAB string scalar. For example, at the command prompt, enter:

```
A = table(logsout.get('latitude').Values.Data, ...
   logsout.get('longitude').Values.Data, ...
   logsout.get('sout').Values.Data);
A.Properties.VariableNames = {'Latitude','Longitude','QuadrantInfo'};
A([4:8:30],:)
```

```
ans =

  4×3 table

    Latitude    Longitude              QuadrantInfo
    _____    _____    _____

     28.232       41.267     "Location in Northeast Quadrant"
     40.425      -29.425     "Location in Northwest Quadrant"
```

```
   -30.593      -39.548      "Location in Southwest Quadrant"
   -38.638       31.735      "Location in Southeast Quadrant"
```

See Also
`get` | `strcat` | `table`

More About

- "Manage Textual Information by Using Strings" on page 20-2
- "Send Messages with String Data" on page 20-14
- "Share String Data with Custom C Code" on page 20-17
- "Simulate a Media Player by Using Strings" on page 20-22
- "Access Signal Logging Data" on page 32-54
- "View State Activity by Using the Simulation Data Inspector" on page 24-33
- "View Data with the Simulation Data Inspector" (Simulink)

Send Messages with String Data

This example shows how to configure a pair of Stateflow charts that communicate by sending messages that carry string data. For more information, see "Communicate with Stateflow Charts by Sending Messages" on page 11-10.

This model contains two charts that use C as the action language. During simulation, the Emitter chart reads an input string key from the String Constant block and sends a message to the Receiver chart. The message data consists of the input string key. The Receiver chart compares the string with a constant keyword and returns an output string that grants or denies access.
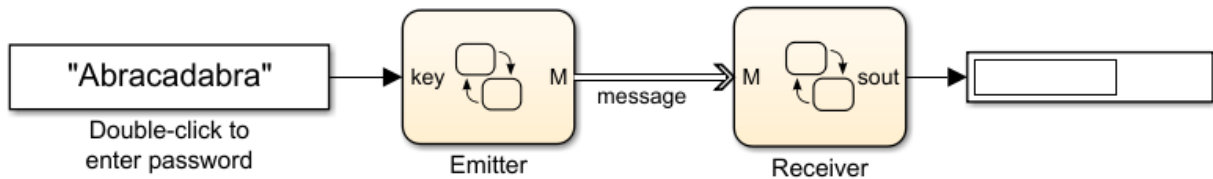
Emitter Chart

The Emitter chart consists of a single state A. When the state becomes active, it sets the data for the message M to the input value key and sends the message to the Receiver chart.
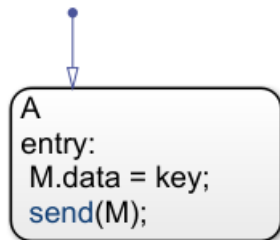
This table lists the scope and type for the symbols in the chart.

Symbol	Scope	Type
key	Input Data	Inherit: Same as Simulink

Symbol	Scope	Type
M	Output Message	string

Receiver Chart

The chart consists of two states joined by a transition. The input message M guards the transition. If there is a message present and its data value equals the constant string lock, then the state activity transitions from Off to On. The chart outputs the string value 'Access Granted'. If there is no message present, or if the data value does not equal lock, the chart does not take the transition and the output value is 'Access Denied'.
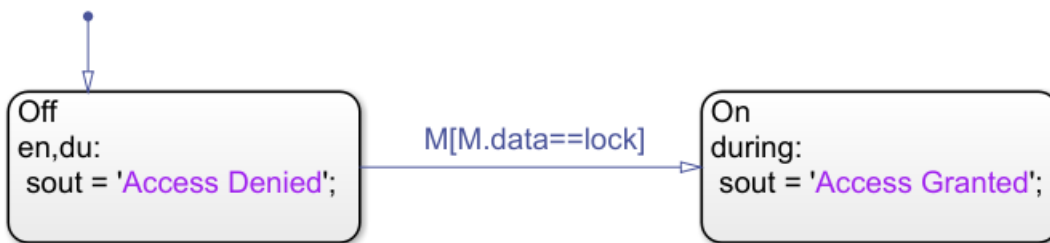
This table lists the scope and type for the symbols in the chart.

Symbol	Scope	Type
M	Input Message	Inherit: Same as Simulink
lock	Constant Data	string
sout	Output Data	string

The constant string lock contains a secret password, initially set to 'Open Sesame'. You can change the value of lock in the **Initial value** field of the Property Inspector.

View Simulation Results

During simulation, the model responds to the password that you enter in the String Constant block:

- If you enter an incorrect password, such as `"Abracadabra"`, then the model displays the output string `"Access Denied"`.
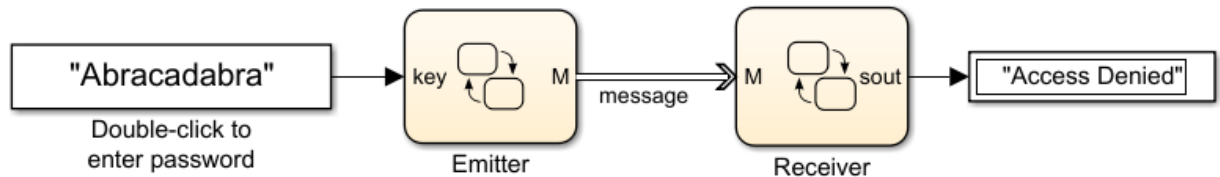
- If you enter the correct password, in this case, `"Open Sesame"`, then the model displays the output string `"Access Granted"`.
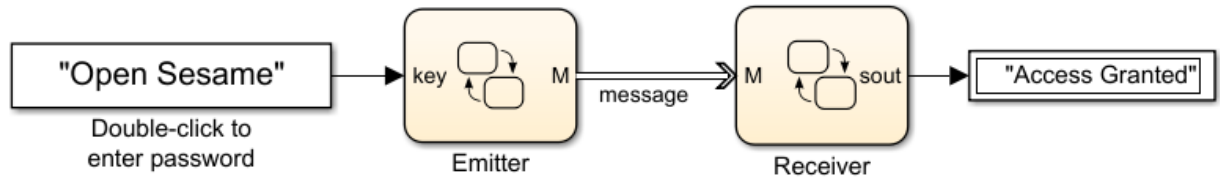
See Also

send

More About

- "Manage Textual Information by Using Strings" on page 20-2
- "Log String Data to the Simulation Data Inspector" on page 20-9
- "Share String Data with Custom C Code" on page 20-17
- "Simulate a Media Player by Using Strings" on page 20-22
- "Communicate with Stateflow Charts by Sending Messages" on page 11-10
- "View Differences Between Stateflow Messages, Events, and Data" on page 11-2

Share String Data with Custom C Code

This example shows how to share string data between a Stateflow® chart and custom C code. You can export string data from a Stateflow chart to a C function by using the `str2ascii` operator. You can import the output of your C code as string data in a Stateflow chart by using the `ascii2str` operator. By sharing data with custom code, you can augment the capabilities of Stateflow and leverage the software to take advantage of your preexisting code. For more information, see "Reuse Custom Code in Stateflow Charts" on page 30-26.
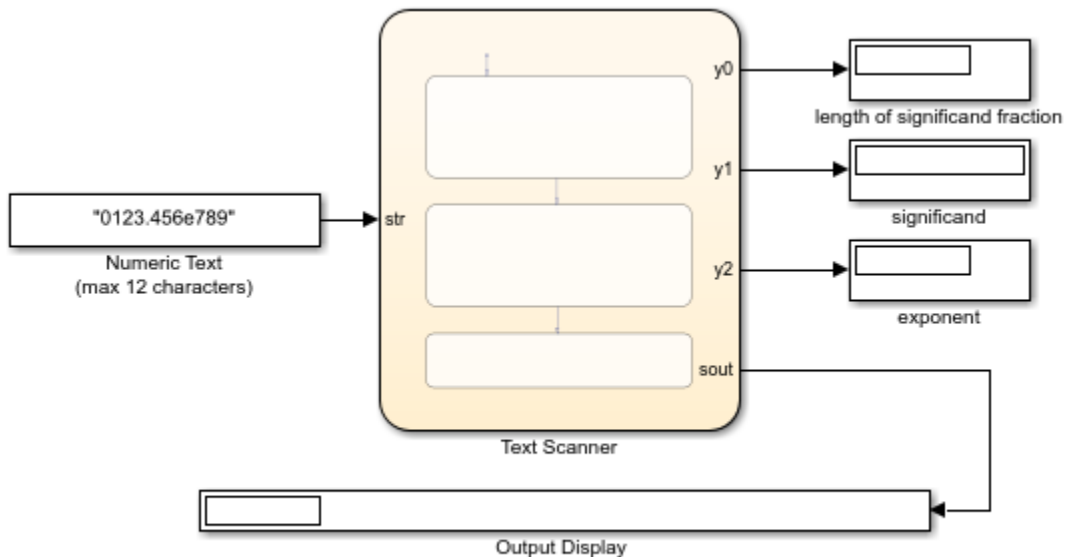
Copyright 2018-2019 The MathWorks, Inc.

This model contains a Stateflow chart that calls two functions from custom C code. During simulation, the chart takes as its input a string that contains text representing a floating-point number in exponential form. The chart consists of three states that:

- Search the input string for leading zeroes, a decimal point, and an `e`.

- Parse the string into double-precision numbers representing the significand and exponent parts of the input.

- Merge the numeric information into an output string expressing the input in scientific notation.

For example, if the input string is `"0123.456e789"`, then the chart outputs the string `"0123.456e789 means 1.23456 times ten to the 791th power"`.

```
Search
entry:
Astr = str2ascii(str,12);
searchfun(n,Astr);

% n[0] = number of leading zeroes
% n[1] = characters before decimal point
% n[2] = characters in significand
% n[3] = length of string
```

```
Parse
entry:
y0 = n[2]-n[0]+(n[1]==n[2])-2;
y1 = str2double(strcat("0",substr(str,0,n[2])))*pow(10,n[0]-n[1]+1);
y2 = str2double(strcat("0",substr(str,n[2]+1,n[3]-n[2]-1)))+(n[1]-n[0]-1);

% y0 = length of significand fraction
% y1 = significand part of input
% y2 = exponent part of input.
```

```
Merge
entry:
mergefun(Asout,Astr,y0,y1,y2);
sout = ascii2str(Asout);
```

Export String Data from Stateflow to C

You can use the `str2ascii` operator to convert string data into an array that you can export from a Stateflow chart to a custom C code function.

1   In the custom code function, declare the input variable as having type `char*`.
2   In the Stateflow chart, convert the string to an array of type `uint8` by calling the operator `str2ascii`.
3   Call the custom code function by passing the `uint8` array as an input.

For example, in the previous chart, the `Search` state converts the input string `str` to the `uint8` array `Asrt`. The `Search` state passes this array as an input to the custom code function `searchfun`:

```
extern void searchfun(int* n, char* strin)
{
   nout[0] = strspn(strin,"0");
   nout[1] = strcspn(strin,".e");
   nout[2] = strcspn(strin,"e");
   nout[3] = strlen(strin);
}
```

The `Search` state calls this function with the command `searchfun(n,Astr)`. The function populates the integer array `n` with these values:

*   `n[0]` contains the number of leading zeroes in the input string `str`.
*   `n[1]` contains the number of characters before the first instance of a decimal point or `e`. This result provides the number of characters before the decimal point in `str`.
*   `n[2]` contains the number of characters before the first instance of `e`. This result provides the number of characters in the significand in `str`.
*   `n[3]` contains the length of the input string `str`.

The `Parse` state uses these results to extract the values of the significand and exponent parts of the input.

Import String Data from C to Stateflow

You can import string data to a Stateflow chart by passing a pointer to an array of type `uint8` as an input to a custom C function.

1   In the custom code function, declare the input variable containing the pointer as having type `char*`.

2 Save the output string data from the custom code function at the location indicated by the pointer.

3 In the Stateflow chart, convert the `uint8` array to a string by calling the operator `ascii2str`.

For example, in the previous chart, the `Merge` state consolidates the numeric information obtained by the `Parse` state into an output string by calling the custom code function `mergefun`:

```
extern void mergefun(char* strout, char* strin, int in0, double in1, double in2)
{
    sprintf(strout, "%s means %1.*f times ten to the %dth power", strin, in0, in1, (int)
}
```

The `Merge` state calls the `mergefun` function with the command `mergefun(Aout,Astr,y0,y1,y2)`:

- `Aout` is an array of type `uint8` pointing to the output of the custom function.

- `Astr` is an array of type `uint8` corresponding to the input string to the chart.

- `y0` is an integer containing the number of digits to the right of the decimal point in the significand.

- `y1` and `y2` are double-precision numbers representing the significand and exponent parts of the input.

The function `mergefun` calls the C library function `sprintf`, merging the contents of `Astr`, `y1`, and `y2` and storing the result in the memory location indicated by `Aout`. The chart uses the operator `ascii2str` to convert this output to the string `sout`. In this way, the model imports the string constructed by the custom code function back into Stateflow.
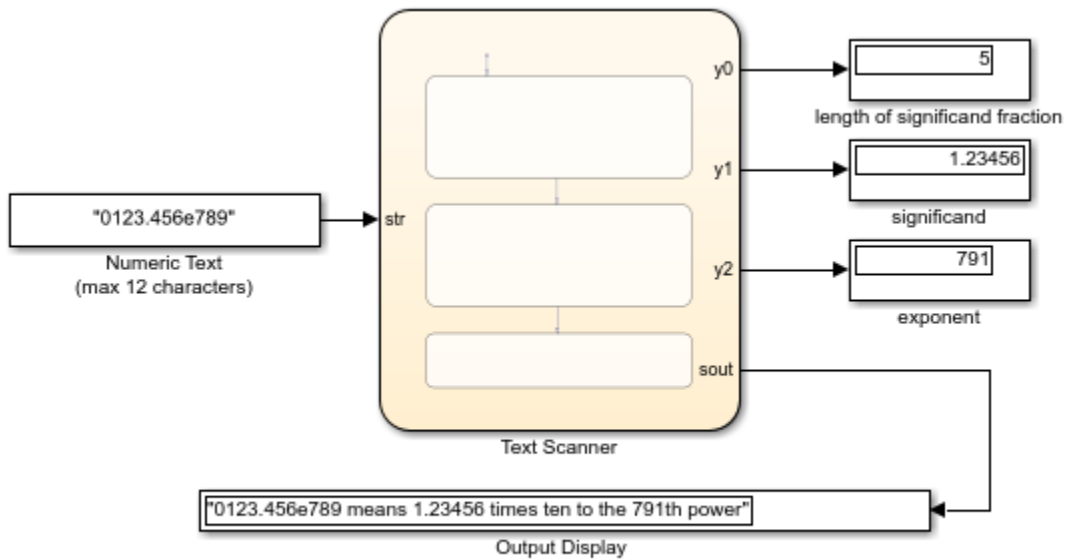
Copyright 2018-2019 The MathWorks, Inc.

See Also

ascii2str | str2ascii | str2double | strcat | substr

More About

- "Manage Textual Information by Using Strings" on page 20-2
- "Log String Data to the Simulation Data Inspector" on page 20-9
- "Send Messages with String Data" on page 20-14
- "Simulate a Media Player by Using Strings" on page 20-22
- "Reuse Custom Code in Stateflow Charts" on page 30-26

Simulate a Media Player by Using Strings

This example models a media player by using string data in two Stateflow® C action language charts. The model plays a clip of music and other pre-recorded sounds. During simulation, the charts use strings to control the behavior of the media player and provide natural language output messages.
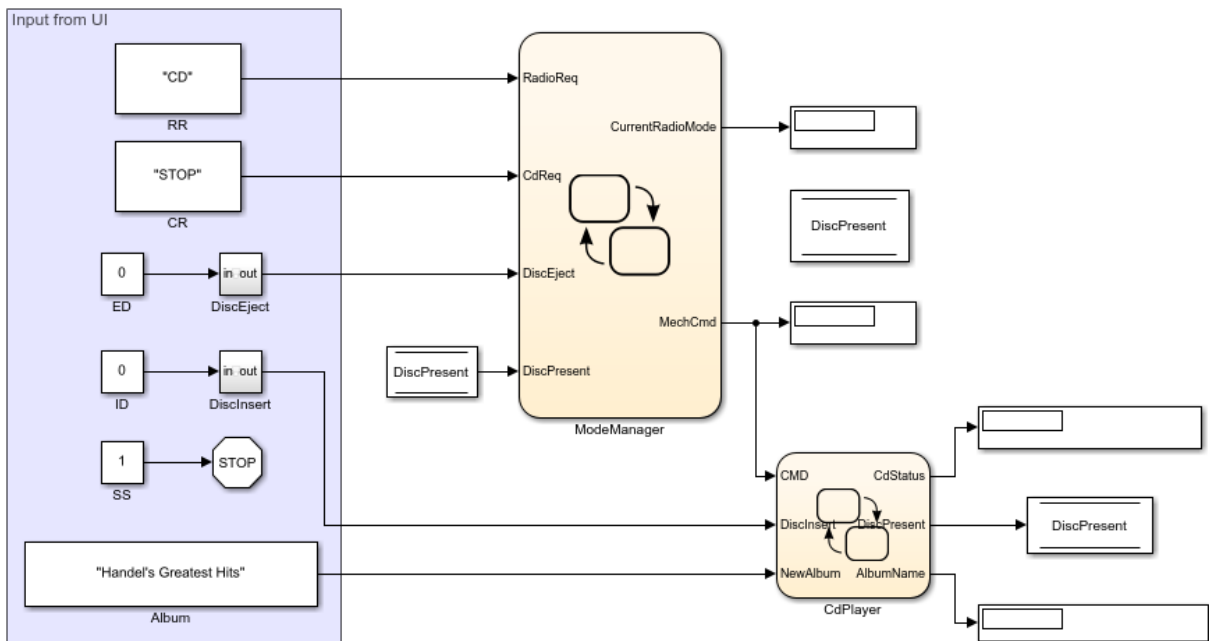
Copyright 1997-2018 The MathWorks, Inc.

Run the Media Player Model

1. Simulate the model. The Media Player Helper appears.
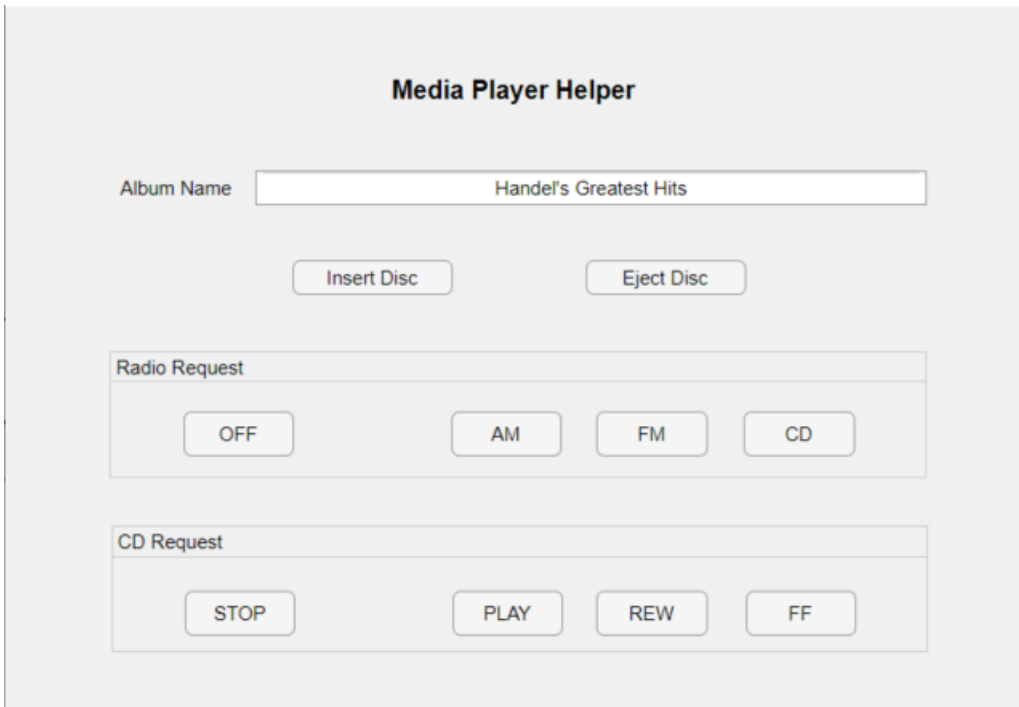
In the model, the Display blocks show the default settings of the media player:

- `CurrentRadioMode: "Standby (OFF)"`
- `MechCmd: "STOP"`
- `CdStatus: "EMPTY"`
- `AlbumName: "None"`

2. In the **Radio Request** section, click **CD**.

- The String Constant block RR changes from `"OFF"` to `"CD"`.
- The Display block for `CurrentRadioMode` changes from `"Standby (OFF)"` to `"CD Player"`.

3. Click **Insert Disc**.

- The Display block for `AlbumName` changes from `"None"` to `"Handel's Greatest Hits"`.

- The Display block for `CdStatus` changes from `"EMPTY"` to `"Reading: Handel's Greatest Hits"` to `"Stopped"`.

4. In the **CD Request** section, click **PLAY**.

- The String Constant block `CR` and the Display block for `MechCmd` change from `"STOP"` to `"PLAY"`.
- The Display block for `CdStatus` changes from `"Stopped"` to `"Playing: Handel's Greatest Hits"`.
- Music begins to play.

5. In the **CD Request** section, click **FF**.

- Music stops and chirping sounds begin.
- The String Constant block `CR` and the Display block for `MechCmd` change from `"PLAY"` to `"FF"`.
- The Display block for `CdStatus` changes from `"Playing: Handel's Greatest Hits"` to `"Forward >> Handel's Greatest Hits"`. The album name in this message scrolls forward across the display.

To see other changes in the Display blocks, use the Media Player Helper to select other operating modes or to enter a different album name.

ModeManager Chart

This chart activates the appropriate subcomponent of the media player (AM radio, FM radio, or CD player) depending on the inputs received from the Media Player Helper.
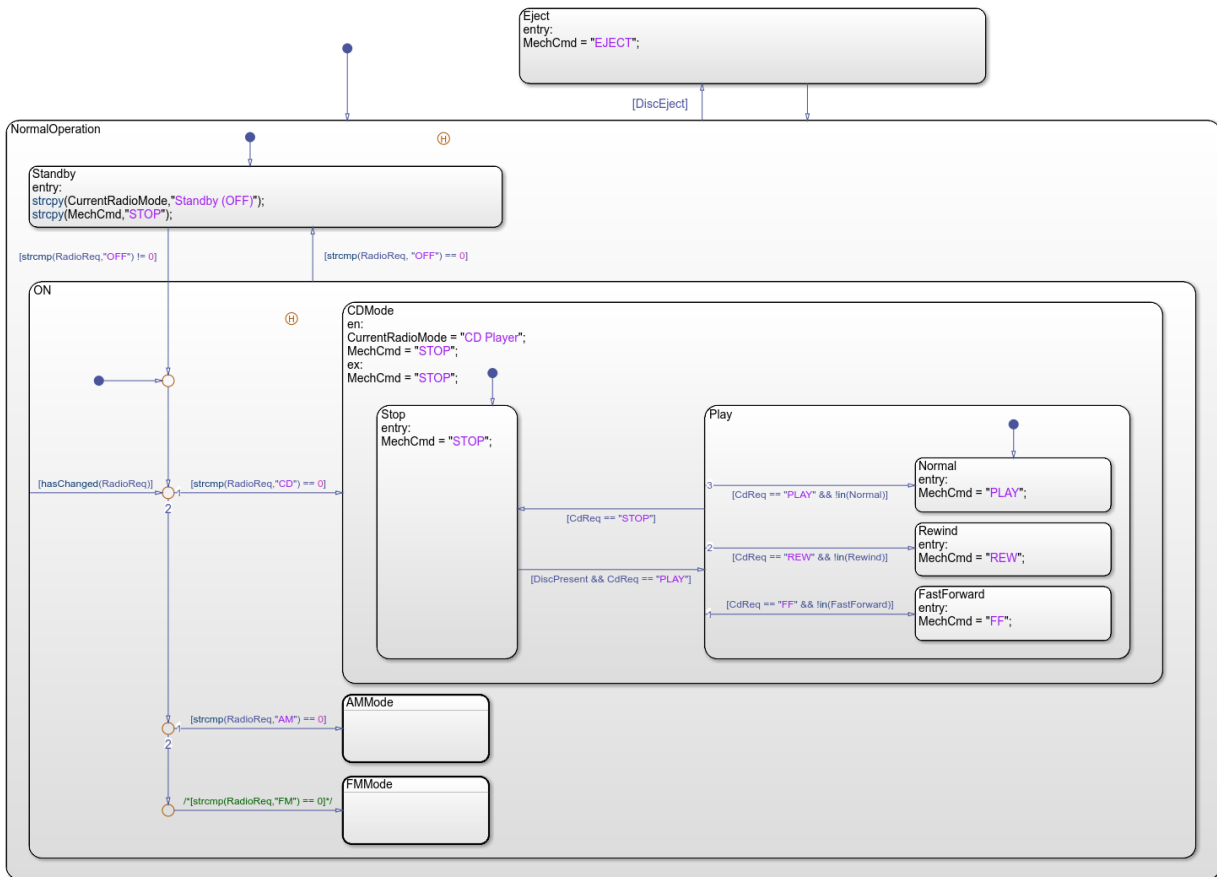
Key Features

- String data `RadioReq`, `CdReq`, and `MechCmd` control chart behavior.
- String data `CurrentRadioMode` provides natural language output.
- Operators `strcpy` and `=` assign values to string data.
- Operators `strcmp` and `==` compare values of string data.
- Operator `hasChanged` detects changes in the value of `RadioReq`.

Chart Behavior

At the start of simulation, the `NormalOperation` state becomes active. If the Boolean data `DiscEject` is `1` (or `true`), a transition to the `Eject` state occurs, followed by a transition back to the `NormalOperation` state.

When `NormalOperation` is active, the previously active substate (`Standby` or `ON`) as recorded by the history junction becomes active. Subsequent transitions between the `Standby` and `ON` substates depend on the value of the expression `strcmp(RadioReq,"OFF")`:

- If `strcmp` returns a value of zero, then `RadioReq` is `"OFF"` and the `Standby` substate is activated.

- If `strcmp` returns a nonzero value, then `RadioReq` is not `"OFF"` and the `ON` substate is activated.

In the `ON` substate, three substates represent the operating modes of the media player: CD player, AM radio, and FM radio. Each substate corresponds to a different value of string data `RadioReq`. The inner transition inside the `ON` state continually scans for changes in the value of `RadioReq`.

- If the value of `RadioReq` is `"CD"`, then the substate `CDMode` becomes active, setting the media player to CD player mode. The ModeManager chart outputs `"PLAY"`, `"REW"`, `"FF"`, and `"STOP"` commands to the CdPlayer chart through the string data `MechCmd`.

- If the value of `RadioReq` is `"AM"`, then the substate `AMMode` becomes active, setting the media player to AM radio mode. The ModeManager chart outputs a `"STOP"` command to the CdPlayer chart through the string data `MechCmd`.

- If the value of `RadioReq` is `"FM"`, then the substate `FMMode` becomes active, setting the media player to FM radio mode. The ModeManager chart outputs a `"STOP"` command to the CdPlayer chart through the string data `MechCmd`.

CdPlayer Chart

This chart activates the appropriate operating mode for the CD player depending on the input received from the ModeManager chart.

Empty
entry: AlbumName = "None";
CdStatus = "EMPTY";

[DiscInsert]

Inserting
entry: AlbumName = NewAlbum;
CdStatus = strcat("Reading: ",AlbumName);

Ejecting
entry: CdStatus = strcat("Ejecting: ",AlbumName);

[after(1/Ts, tick)]

[after(1/Ts, tick)]

[CMD == "EJECT"]

DiscPresent
en:
N = strlen(AlbumName);

[hasChanged(CMD)]

[CMD == "STOP"]

STOP
en:
CdStatus = "Stopped";
ml.stop(ml.music);
ml.stop(ml.chirp);

[CMD == "PLAY"]

PLAY
en:
CdStatus = strcat("Playing: ",AlbumName);
i = 0;
ml.stop(ml.chirp);
ml.play(ml.music);

[CMD == "REW"]

REW
en, du:
CdStatus = strcat("Reverse << ",substr(AlbumName,i,N));
ml.stop(ml.music);
ml.play(ml.chirp);

RewIndexUpdate

[CMD == "FF"]

FF
en, du:
CdStatus = strcat("Forward >> ",substr(AlbumName,i,N));
ml.stop(ml.music);
ml.play(ml.chirp);

FfIndexUpdate

Key Features

- String data `Cmd` controls chart behavior.

- String data `NewAlbum`, `AlbumName`, and `CdStatus` provide natural language output.

- Operators = and == assign and compare values of string data.

- Operators `strcat`, `strlen`, and `substr` produce text in output string `CdStatus`.

- Temporal logic operator `after` controls the timing of transitions during disc insertion and ejection.

- Namespace operator `ml` accesses MATLAB® `play` and `stop` functions to manage music and sounds.

Chart Behavior

At the start of simulation, the `Empty` state is activated.

If the Boolean data `DiscInsert` is 1 (or `true`), a transition to the `Inserting` state occurs. After a short time delay, a transition to the `DiscPresent` state occurs.

The `DiscPresent` state remains active until the data `CMD` becomes `"EJECT"`. At that point, a transition to the `Ejecting` state occurs. After a short time delay, a transition to the `Empty` state occurs.

Whenever a state transition occurs, the entry action in the new state changes the value of `CdStatus` to reflect the status of the CD player. In the `FF` or `REW` substates, the during actions continually change the value of `CdStatus` to produce a scrolling motion effect.

- When the active state is `Empty`, the value of `CdStatus` is `"EMPTY"`.

- When the active state is `Inserting`, the value of `CdStatus` is `"Reading: *AlbumName*"`.

- When the active state is `Ejecting`, the value of `CdStatus` is `"Ejecting: *AlbumName*"`.

- When the active state is `DiscPresent.STOP`, the value of `CdStatus` is `"Stopped"`.

- When the active state is `DiscPresent.PLAY`, the value of `CdStatus` is `"Playing: *AlbumName*"`.

- When the active state is `DiscPresent.REW`, the value of `CdStatus` is `"Reverse << *AlbumName*"`, where *AlbumName* scrolls backward across the display.

- When the active state is `DiscPresent.FF`, the value of `CdStatus` is `"Forward >> *AlbumName*"`, where *AlbumName* scrolls forward across the display.

See Also

after | strcat | strcmp | strcpy | strlen | substr

More About

- "Manage Textual Information by Using Strings" on page 20-2
- "Detect Changes in Data Values" on page 12-69
- "Control Chart Execution by Using Temporal Logic" on page 12-49
- "Access MATLAB Functions and Workspace Data in C Charts" on page 12-32
- "Record and Play Audio" (MATLAB)
- "Simulink Strings" (Simulink)

21

Continuous-Time Systems in Stateflow Charts

Continuous-Time Modeling in Stateflow

Hybrid systems use modal logic to transition from one mode to another in response to physical events and conditions. In these systems, continuous-time dynamics govern each mode. A simple example of this type of hybrid system is a bouncing ball. The ball moves continuously through the air until it hits the ground, at which point a mode change or discontinuity occurs. As a result, the ball suddenly changes direction and velocity. For more information, see "Model a Bouncing Ball in Continuous Time" on page 21-11.

Simulate hybrid systems that respond to continuous and discrete mode changes by configuring Stateflow charts for continuous-time modeling. In a Stateflow chart, you can represent modal logic succinctly and intuitively as a series of states, transitions, or flow charts. You can also represent state information as continuous local variables with automatic access to time derivatives.

Continuous-time simulation is supported only in Stateflow charts in Simulink models. If your continuous system does not contain modal logic, consider using a Simulink model. For more information, see "Model a Continuous System" (Simulink).

Configure a Stateflow Chart for Continuous-Time Simulation

Continuous updating is a Stateflow chart property.

1   Right-click inside a chart and select **Properties** from the context menu.
2   In the Chart Properties dialog box, set the **Update method** field to `Continuous`. When you select this option:

  • The **Enable zero-crossing detection** check box is selected.
  • The **Enable super step semantics** check box is unavailable.
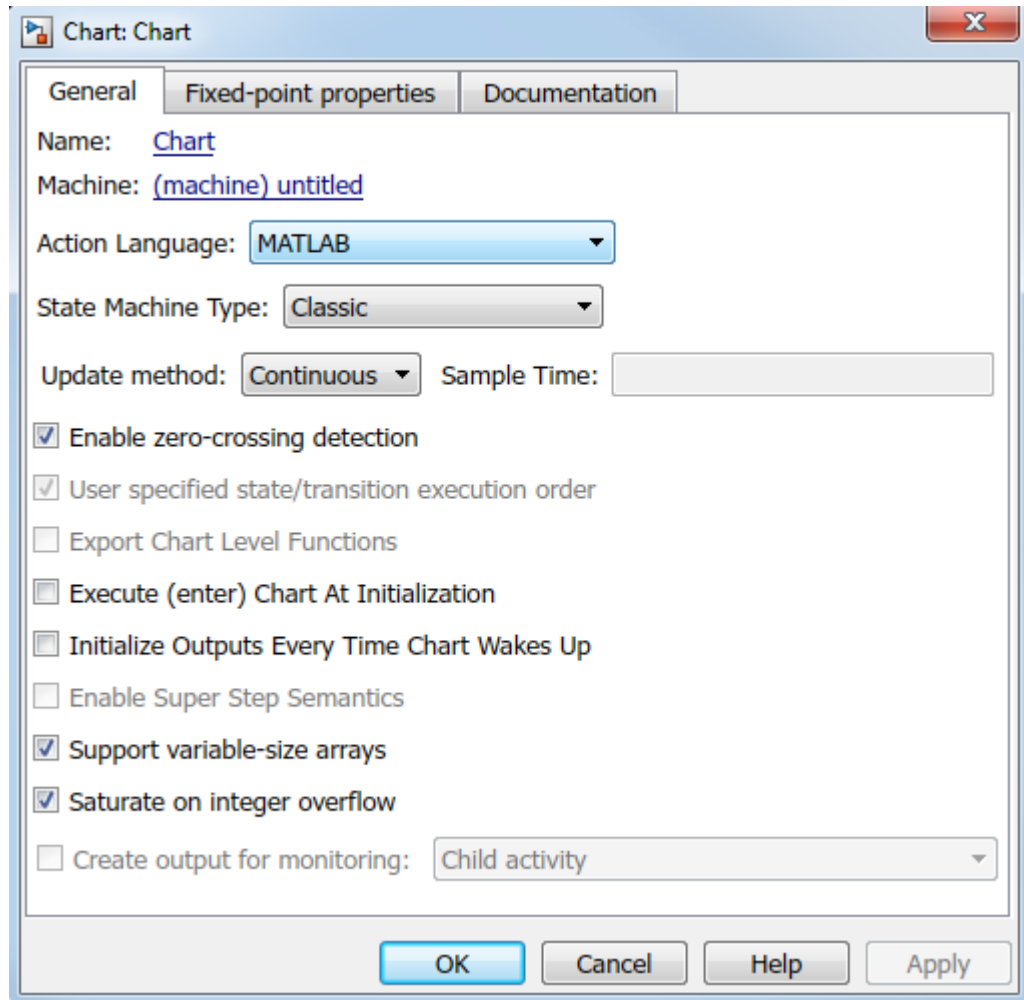
3 (Optional) By default, zero-crossing detection is enabled. To disable this option, clear the **Enable zero-crossing detection** check box. For more information, see "Disable Zero-Crossing Detection" on page 21-4.

4 Click **OK**.

Note You cannot use Moore charts for continuous-time modeling.

Interaction with Simulink Solver

Maintain Mode in Minor Time Steps

During continuous-time simulation, a Stateflow chart updates its mode only in major time steps. In a minor time step, the chart computes outputs based on the state of the chart during the last major time step. For more information, see "Minor Time Steps" (Simulink).

Compute Continuous State at Each Time Step

When you define local continuous variables, the Stateflow chart provides programmatic access to their derivatives. The Simulink solver computes the continuous state of the chart at the current time step based on the values of these variables and their derivatives at the previous time step. For more information, see "Continuous Versus Discrete Solvers" (Simulink).

Register Zero Crossings on State Transitions

To determine when a state transition occurs, a Stateflow chart registers a zero-crossing function with the Simulink solver. When Simulink detects a change of mode, the solver searches forward from the previous major time step to detect when the state transition occurred. For more information, see "Zero-Crossing Detection" (Simulink).

Disable Zero-Crossing Detection

Zero-crossing detection on state transitions can present a tradeoff between accuracy and performance. When detecting zero crossings, a Simulink model accurately simulates mode changes without unduly reducing step size. For systems that exhibit *chattering*, or frequent fluctuations between two modes of continuous operation, zero-crossing detection can potentially impact simulation time. Chattering requires a Simulink model to check for zero crossings in rapid succession, which can slow simulation. In these situations, you can:

- Disable zero-crossing detection.
- Choose a different zero-crossing detection algorithm for your chart.
- Modify parameters that control the frequency of zero crossings in your Simulink model.

You can choose from different zero-crossing detection algorithms on the **Solver** pane in the Model Configuration Parameters dialog box. For more information, see "Zero-Crossing Algorithms" (Simulink) and "Preventing Excessive Zero Crossings" (Simulink).

Guidelines for Continuous-Time Simulation

To maintain the integrity and smoothness of the results of a continuous-time simulation, constrain your charts to a restricted subset of Stateflow chart semantics. By restricting the semantics, the inputs do not depend on unpredictable factors such as:

- The number of minor intervals that the Simulink solver uses in each major time step.
- The number of iterations required to stabilize the integration and zero-crossings algorithms.

By minimizing these side effects, a Stateflow chart can maintain its state at minor time steps and update its state only during major time steps. Therefore, a Stateflow chart can compute outputs based on a constant state for continuous time.

Continuous-Time Charts Must Have at Least One State

During continuous-time simulation, a chart updates its outputs at minor time steps corresponding to the `during` actions of the active state. A chart with no states produces no output. To mimic the behavior of a stateless chart in continuous time, create a single state that calls a graphical function in its `during` action.

Update Local Data in entry, exit, and Transition Actions

To maintain precision in continuous-time simulation, update discrete and continuous local data only during major time steps corresponding to state transitions. During state transitions, only these types of actions occur:

- State `exit` actions, which occur before leaving the state at the beginning of the transition.
- State `entry` actions, which occur after entering the new state at the end of the transition.
- Transition actions, which occur during the transition.
- Condition actions on a transition, but only if the transition directly reaches a state. For example, this chart executes the action `n++` even when conditions `c2` and `c3` are false. Because there is no state transition, the condition action updates `n` in a minor time step and results in an error.

21-5
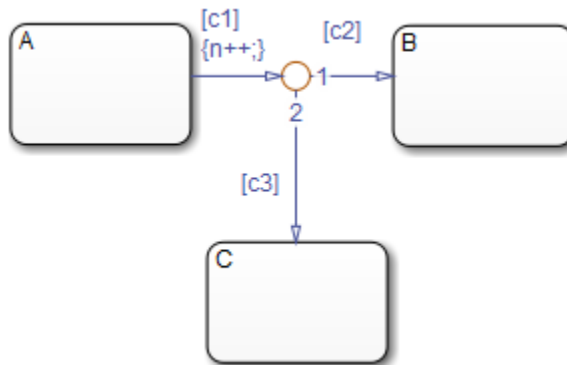
Do not write to local continuous data in state `during` actions because these actions occur in minor time steps.

Compute Derivatives in State during Actions

In minor time steps, a continuous-time chart executes only state `during` actions. Because Simulink models read continuous-time derivatives during minor time steps, compute derivatives in `during` actions to provide the most current calculation.

Do Not Read Outputs or Derivatives in State during Actions or in Transition Conditions

In minor time steps, it is possible that outputs and derivatives do not reflect their most current values. To provide smooth outputs, compute values from local discrete data, local continuous data, and chart inputs.

Do Not Call Simulink Functions in State during Actions or in Transition Conditions

You cannot call Simulink functions during minor time steps. Instead, call Simulink functions only in actions that occur during major time steps: state `entry` or `exit` actions and transition actions. Calling Simulink functions in state `during` actions or in transition conditions results in an error during simulation. For more information, see "Reuse Simulink Components in Stateflow Charts" on page 29-2.

Use Discrete Variables to Govern Conditions in during Actions

To prevent mode changes between major time steps, conditions that affect control flow in `during` actions depend on discrete variables. Discrete variables do not change value between major time steps.

Do Not Use Input Events

The presence of input events makes a chart behave like a triggered subsystem and unable to simulate in continuous time. For example, this model generates an error if the chart uses a continuous update method.
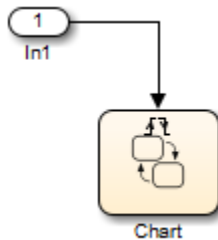
To mimic the behavior of an input event, pass the input signal through a Hit Crossing block as an input to the continuous-time chart.
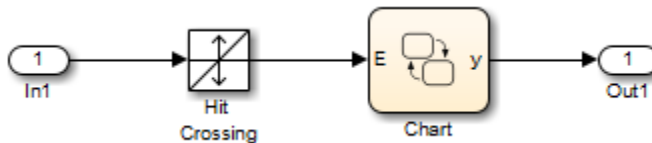
Do Not Use Inner Transitions

When a mode change occurs during continuous-time simulation, the `entry` action of the destination state indicates to the Simulink model that a state transition occurred. With an inner transition, the chart never executes the `entry` action. For more information, see "Inner Transitions" on page 3-46.

Limit Use of Temporal Logic

Do not use event-based temporal logic because in continuous-time simulation, there is no concept of a tick. Use only absolute-time temporal logic for continuous-time simulation. For more information, see "Operators for Absolute-Time Temporal Logic" on page 12-56.

Do Not Use Change Detection Operators

To implement change detection, Stateflow buffers variables in a way that affects the behavior of charts between a minor time step and the next major time step.

Do Not Modify Operating Point Values

Modifying the operating point of a continuous-time chart is not supported. If you load the operating point for a continuous-time chart, you cannot modify the activity of states or any values of local or output chart data. For more information, see "Rules for Using the Operating Point of a Chart" on page 16-38.

See Also

More About

- "Model a Bouncing Ball in Continuous Time" on page 21-11
- "Store Continuous State Information in Local Variables" on page 21-9
- "Solvers" (Simulink)
- "Zero-Crossing Detection" (Simulink)

Store Continuous State Information in Local Variables

To compute a continuous state, you must determine its time derivative. You can represent this information by using local variables that are updated in continuous time. Continuous-time simulation is supported only in Stateflow charts in Simulink models. For more information, see "Continuous-Time Modeling in Stateflow" on page 21-2.

Define Continuous-Time Variables

1 Configure the chart to update in continuous time, as described in "Configure a Stateflow Chart for Continuous-Time Simulation" on page 21-2.
2 Add a data object to your chart, as described in "Add Stateflow Data" on page 9-2.
3 Set the **Scope** property for the data object to `Local`.
4 Set the **Update Method** property for the data object to `Continuous`.

In a Stateflow chart, continuous-time variables always have type `double`.

Compute Implicit Time Derivatives

For each continuous-time variable, Stateflow implicitly creates a variable to represent its time derivative. A chart denotes time derivative variables as *variable_name*_dot. For example, `data_dot` represents the time derivative of a continuous variable `data`. You can write to the time derivative variable in the `during` action of a state. The time derivative variable does not appear in the Symbols Window or in the Model Explorer.

Note Do not explicitly define variables with the suffix `_dot` in a chart configured for continuous-time simulation.

Expose Continuous State to a Simulink Model

In a Stateflow chart, you represent the continuous state by using local variables rather than inputs or outputs. To expose the continuous state to a Simulink model, you must explicitly assign the local variables to Stateflow outputs in the `during` action of a state.

Guidelines for Continuous-Time Variables

- Scope for continuous-time variables can be `Local` or `Output`.
- Define continuous-time variables at the chart level or below in the Stateflow hierarchy.
- Expose the continuous state of a chart by assigning the local continuous-time variable to a Stateflow output.

See Also

More About

Model a Bouncing Ball in Continuous Time

This example shows how to configure a Stateflow® chart that simulates a bouncing ball in continuous time. The ball moves continuously through the air until it hits the ground, at which point a discontinuity occurs. As a result, the ball suddenly changes direction and velocity. For more information, see "Continuous-Time Modeling in Stateflow" on page 21-2.

The model `sf_bounce` contains a chart that updates in continuous-time. Local variables describe the dynamics of a free-falling ball in terms of position and velocity. During simulation, the model uses zero-crossing detection to determine when the ball hits the ground.

Dynamics of a Bouncing Ball

You can specify how a ball falls freely under the force of gravity in terms of position p and velocity v with this system of first-order differential equations.

$$\dot{p} = v$$

$$\dot{v} = -9.81$$

When $p <= 0$, the ball hits the ground and bounces. You can model the bounce by updating the position and velocity of the ball:

- Reset the position to $p = 0$.
- Reset the velocity to the negative of its value just before the ball hit the ground.
- To account for energy loss, multiply the new velocity by a coefficient of distribution (-0.8).

Configure Chart for Continuous-Time Simulation

In the model, the BouncingBall chart implements modal logic to simulate the continuous dynamics of free fall and the discrete changes associated with bouncing. In the Chart properties dialog box, these settings enable the BouncingBall chart to simulate in continuous time:

- **Update method** is `Continuous` so the chart employs continuous-time simulation to model the dynamics of the bouncing ball.

21-11

- **Enable zero-crossing detection** is selected so the Simulink® solver can determine exactly when the ball hits the ground. Otherwise, the Simulink model cannot simulate the physics accurately and the ball appears to descend below ground.

Modeling a Bouncing Ball
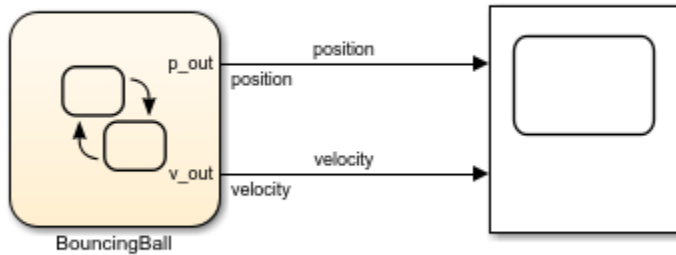
Copyright 2007-2019 The MathWorks, Inc.

Define Continuous-Time Variables

The BouncingBall chart has two continuous-time variables: p for position and v for velocity. For each one of these variables:

- **Scope** is Local.
- **Type** is double.
- **Update Method** is Continuous.

To expose the continuous state of the chart to the Simulink model, the BouncingBall chart has two output variables: p_out and v_out. For each one of these variables:

- **Scope** is Output.
- **Type** is double.
- **Update Method** is Discrete.

The chart defines the time derivative of continuous-time variables implicitly:

p_dot is the derivative of position p. v_dot as the derivative of velocity v.

In the Model Explorer, you can view the continuous-time local variables and the corresponding outputs in the chart. Implicit derivative variables do not appear in the Model Explorer or in the Symbols Window.
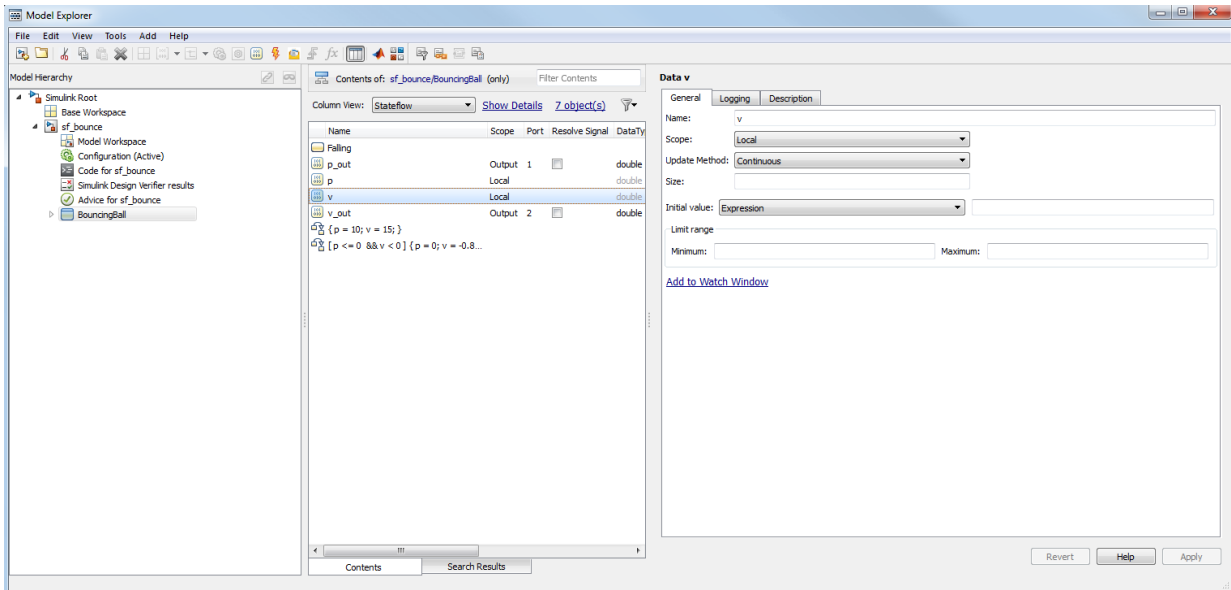
Model Continuous Dynamics of Free Fall

The BouncingBall chart consists of a single state `Falling` that numerically solves the differential equations for free fall. The default transition into the state sets the initial position to 10 m and the initial velocity to 15 m/s. The during actions in the state:

- Define the derivatives of position and velocity.
- Assign the values of the position and velocity of the ball to the output variables `p_out` and `v_out`.
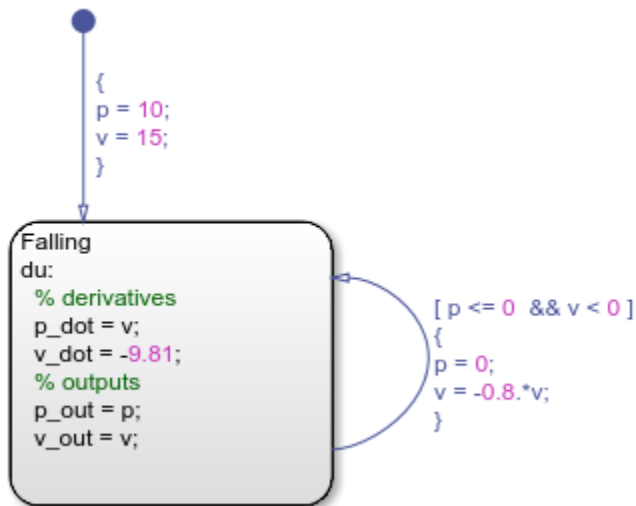
Model Discrete Effects of the Bounce

The Falling state has a self-loop transition that models the discontinuity of the bounce as an instantaneous mode change when the ball suddenly reverses direction. The condition on the transition determines when the ball hits the ground by checking its position p <= 0 and velocity v < 0. If the condition is valid, the condition action resets the position and velocity when the ball hits the ground.

Why Not Check for p == 0?

The ball hits the ground when position p is exactly zero. By relaxing the condition, you increase the tolerance within which the Simulink solver can detect when the position changes sign. For more information, see "How Blocks Work with Zero-Crossing Detection" (Simulink).

Why Check for v < 0?

The second part of the condition helps maintain the efficiency of the Simulink solver by minimizing the frequency of zero crossings. Without the second check, the condition remains true after the state transition, resulting in two successive zero crossings.

Validate Chart Semantics

The BouncingBall chart meets the design requirements defined in "Guidelines for Continuous-Time Simulation" on page 21-5. In particular, the chart:

- Initializes the local variables p and v on the default transition.
- Assigns values to the derivatives p_dot and v_dot in a during action.
- Writes to local variables p and v in a transition action.
- Does not contain events, inner transitions, event-based temporal logic, or change detection operators.

View Simulation Results

After you run the model, the scope shows the graphs of position and velocity. The position graph exhibits the expected bounce pattern.
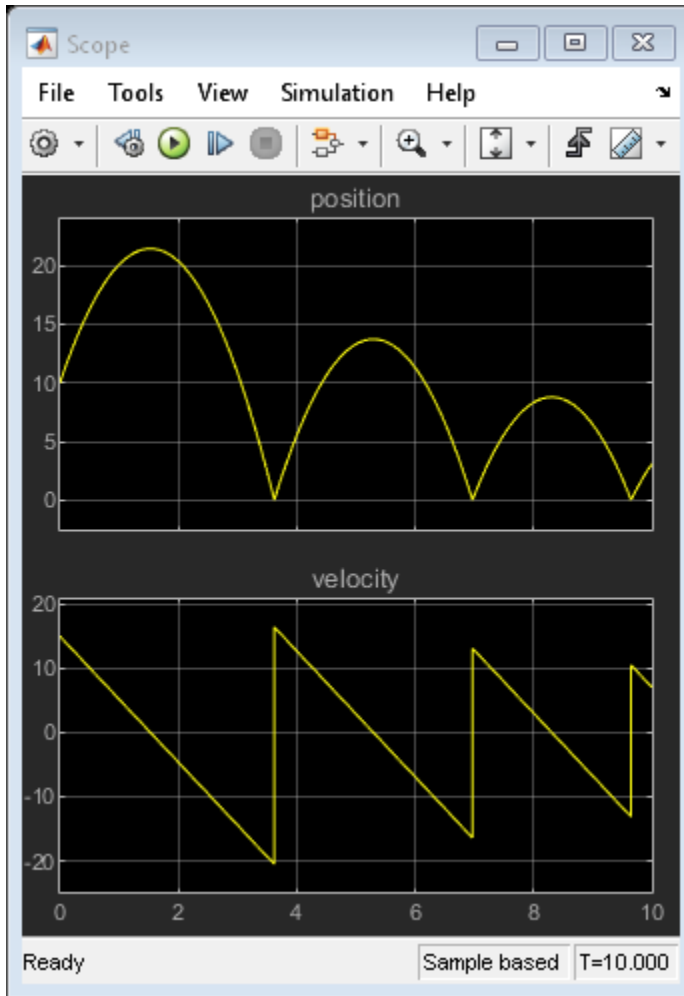
See Also

More About

- "Continuous-Time Modeling in Stateflow" on page 21-2
- "Store Continuous State Information in Local Variables" on page 21-9

22

Fixed-Point Data in Stateflow Charts

Fixed-Point Data in Stateflow Charts

Fixed-point numbers use integers and integer arithmetic to approximate real numbers. They are an efficient means for performing computations involving real numbers without requiring floating-point support in underlying system hardware.

Fixed-Point Numbers

Fixed-point numbers represent real numbers by using the encoding scheme:

$$V \approx \tilde{V} = SQ + B.$$

- V is a precise real-world value that you want to approximate with a fixed-point number.

- $\tilde{V}$ is the approximate real-world value that results from the fixed-point representation.

- Q is an integer that encodes the fixed-point number. This value is called the *quantized integer*.

- S is a coefficient that determines the precision of the fixed-point representation. This value is called the *slope*.

- B is an additive correction called the *bias*.

The quantized integer Q is the only part of the fixed-point representation that varies in value. In the generated code, the quantities S and B are constant and appear only as literal numbers or expressions. If a fixed-point number changes, its quantized integer Q changes but S and B remain unchanged.

To determine the quantized integer Q corresponding to a real-world value V, round the quantity $(V - B)/S$ to an integer. For example, to represent the number $V = 15.345$ in a fixed-point type with slope $S = 0.5$ and bias $B = 0.1$, you use the quantized integer

$$Q = \text{round}((V - B)/S) = \text{round}((15.345 - 0.1)/0.5) = \text{round}(30.49) = 30.$$

Because you round Q to an integer, you lose some precision in representing the number 15.345. The number that Q actually represents is

$$\tilde{V} = SQ + B = 0.5 \ \square \ 30 + 0.1 = 15.1.$$

Using fixed-point numbers to represent real numbers with integers involves the loss of some precision. However, with a suitable choice of S and B, you can minimize this loss to acceptable levels. For instance, by changing the coding scheme to use $S = 0.25$ and $B = 0.1$, you can represent the number $V = 15.345$ with greater precision as:

$$Q = \text{round}((V - B)/S) = \text{round}((15.345 - 0.1)/0.25) = \text{round}(60.98) = 61$$

$$\tilde{V} = SQ + B = 0.25 \ \square \ 61 + 0.1 = 15.35.$$

The difference between $\tilde{V}$ and V is always less than the slope S.

Specify Fixed-Point Data

In the Model Explorer, you can specify the fixed-point encoding for a data object by using the Data Type Assistant, as described in "Fixed-Point Data Properties" on page 9-10. Set the **Mode** field to `Fixed point` and specify these properties:

- **Signedness**: Choose `Signed` or `Unsigned`.
- **Word length**: Specify the bit size of the word that holds the quantized integer Q.
- **Scaling**: Choose `Binary point` or `Slope and bias`.

   - If you select `Binary point`, the Data Type Assistant displays the **Fraction length** field, which specifies the binary point location. Choosing a **Fraction length** of n defines a fixed-point encoding with a slope of $S = 2^{-n}$ and a bias of $B = 0$.

binary point (fraction length = 2)

binary point (fraction length = -2)
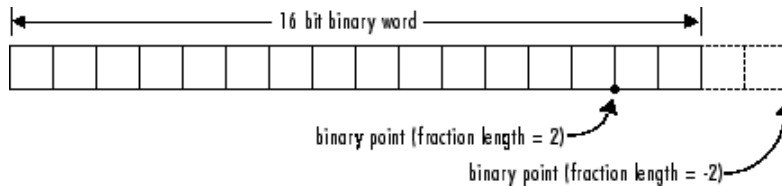
   - If you select `Slope and bias`, the Data Type Assistant displays fields for entering the **Slope** S and **Bias** B for the fixed-point encoding scheme.

Alternatively, you can specify the encoding for a fixed-point data object directly by using the `fixdt` function. In the Property Inspector or the Model Explorer, in the **Type** field, enter an expression in one of these formats:

```
fixdt(Signed, WordLength, FractionLength)
```

```
fixdt(Signed, WordLength, Slope, Bias)
```

Tip: Some encoding schemes are computationally expensive, particularly in multiplication and division operations. Selecting a slope that is an integer power of two and a zero bias avoids these computationally expensive instructions. Using binary point scaling is recommended.

Conversion Operations

Stateflow charts convert real numbers into fixed-point numbers during data initialization and as part of casting operations in the application. These conversions compute a quantized integer Q from a real number input. The type of conversion depends on the action language for the chart.

Conversion in Charts That Use MATLAB as the Action Language

In a chart that uses MATLAB as the action language, you define the method for all conversions through the fixed-point properties for the chart. See "Fixed-Point Properties" on page 24-9.

For example, if you set the **MATLAB Chart `fimath`** property to `Same as MATLAB`, then the chart rounds the resulting quantized integer to its nearest integer value.

Conversions in Charts That Use C as the Action Language

Charts that use C as the action language employ two methods for converting fixed-point data:

- Offline conversions initialize data during code generation. Offline conversions are designed to maximize accuracy. These conversions round the resulting quantized integer to its nearest integer value. Offline conversions are performed for initialization of data (variables and constants) in the Stateflow hierarchy and from the MATLAB workspace.

- Online conversions perform casting operations during run time. Online conversions are designed to maximize computational efficiency. They are faster and more efficient, but less precise than offline conversions. Instead of rounding Q to its nearest integer, online conversions round to the floor (except for division, which can round to 0, depending on the C compiler).

For example, this table illustrates the difference between offline and online conversions of real numbers to fixed-point numbers defined with a slope of $S = 2^{-4}$ and a bias of $B = 0$. For each real-world value V, the chart computes a quantized integer Q by rounding $(V-B)/S$ to the nearest integer (in offline conversion) or to the floor (in online conversion). For each conversion, $\tilde{V} = QS + B$ is the approximate real-world value resulting from Q.

Real-World Value		Offline Conversion		Online Conversion	
V	$(V-B)/S$	Q	$\tilde{V}$	Q	$\tilde{V}$
15.345	245.52	246	15.375	245	15.3125
3.45	55.2	55	3.4375	55	3.4375
1.0375	16.6	17	1.0625	16	1
2.06	32.96	33	2.0625	32	2

Fixed-Point Context-Sensitive Constants

In charts that use C as the action language, you can avoid explicit type casts by using fixed-point context-sensitive constants. These constants infer their type from the context in which they occur. They are written like ordinary numbers with the suffix C or c. For example, 4.3C and 123.4c are valid fixed-point context-sensitive constants that you can use in action statements.

Although fixed-point context-sensitive constants can appear in expressions with any data types (including integers and floating-point data), their main use is with fixed-point numbers. The algorithm that interprets the context-sensitive constant computes a type that provides maximum accuracy without overflow. The algorithm depends on:

- The operations in the expression
- The other data types in the context
- The value of the constant

Fixed-point context-sensitive constants infer their type according to these rules:

- In a casting operation, the constant has the type to which it is being cast.
- In a simple assignment operation of the form a = b:

    - If b is a context-sensitive constant, it has the same type as a.
    - If b is an addition or subtraction operation, then the constant has the same type as the other operand.
    - If b is a multiplication or division operation with a fixed-point operand, then the constant has the type that provides the best possible precision for a fixed-point result, as determined by the fixptbestexp function.

- If `b` is a multiplication or division operation with a floating-point operand of type `double` or `single`, then the constant has the same type as the floating-point operand.

- In a special assignment operation of the form `a  := b`:

  - If `b` is a context-sensitive constant, it has the same type as `a`.

  - If `b` is an arithmetic operation with a floating-point operand of type `double` or `single`, or if `a` is a floating-point data object, then the constant has the same type as the floating-point number.

  - If `b` is an addition or subtraction operation with a fixed-point operand and `a` is a fixed-point data object, then the constant has the same type as `a`.

  - If `b` is a multiplication or division operation with a fixed-point operand and `a` is a fixed-point data object, then the constant has the type that provides the best possible precision for a fixed-point result.

- As an argument in a function call, the constant has the same type as the formal argument.

You cannot use context-sensitive constants as both operands of a binary operation or as the leftmost operand of an assignment operation.

Tips for Using Fixed-Point Data

- Develop and test your application by using double- or single-precision floating-point numbers. Using double- or single-precision floating-point numbers does not limit the range or precision of your computations. Once your application works as designed, you can start substituting fixed-point data for double-precision data.

- In the Model Configuration Parameters dialog box, on the **Hardware Implementation** pane, set the integer word size for the simulation environment to the integer size of the intended target environment. Code generated by Stateflow uses this integer size to select result types for your fixed-point operations. See "Hardware Implementation Pane" (Simulink).

- When you simulate your model, use overflow detection to warn you when the result of a fixed-point operation exceeds the numeric capacity of its fixed-point type. In the Model Configuration Parameters dialog box, on the **Diagnostics > Data Validity** pane, set **Wrap on overflow** and **Saturate on overflow** to `error` or `warning`. If you encounter overflow errors in fixed-point data, increase the range of your data by:

- Increasing the **Word length** value for the overflowing fixed-point data. For example, change the number of bits used to encode the fixed-point data from 16 to 32. This action changes the base integer type for Q from `int16` to `int32`.

- Decreasing the **Fraction length** value (if using `Binary point` scaling) or increasing the **Slope** value (if using `Slope and bias` scaling). For example, decrease the **Fraction length** value from 4 to 1 (or, equivalently, increase the **Slope** value from $S = 2^{-4} = 0.0625$ to $S = 2^{-1} = 0.5$). This action increases the range of your fixed-point data but decreases the available precision.

For more information, see "Data Range Violations" on page 32-30.

- If you encounter issues with model behavior stemming from inadequate precision in your fixed-point data, increase the precision of your data by increasing the **Fraction length** value (if using `Binary point` scaling) or decreasing the **Slope** value (if using `Slope and bias` scaling). For example, increase the **Fraction length** value from 2 to 3 (or, equivalently, decrease the **Slope** value from $S = 2^{-2} = 0.25$ to $S = 2^{-3} = 0.125$). This action increases the precision of your fixed-point data but decreases the available range.

- In charts that use C as the action language, you can use a special assignment operation `:=` and context-sensitive constants to maintain as much precision as possible. See "Override Fixed-Point Promotion in C Charts" on page 22-25 and "Fixed-Point Context-Sensitive Constants" on page 22-5.

Note If you do not use context-sensitive constants with fixed-point types, noninteger numeric constants (constants that have a decimal point) can force fixed-point operations to produce floating-point results.

Automatic Scaling of Fixed-Point Data

Automatic scaling tools can change the settings of Stateflow fixed-point data. You can prevent automatic scaling by selecting the **Lock data type setting against changes by the fixed-point tools** option for the fixed-point data object. See "Lock Data Type Against Fixed-Point Tools" on page 9-9. For methods on autoscaling fixed-point data, see "Choosing a Range Collection Method" (Fixed-Point Designer).

Share Fixed-Point Data with Simulink Models

To share fixed-point data with Simulink models:

- Use the same property values to specify the data in the Stateflow chart and in the Simulink model. For an example of this method of sharing input data from a Simulink model using a Gateway In block, see "Model a Bang-Bang Controller by Using Fixed-Point Inputs" on page 22-10.

  For some Simulink blocks, you can specify the type of input or output data directly. For example, you can specify the fixed-point data type for a Constant block directly in the **Output data type** field by using the `fixdt` function.

- Define the data as **Input** or **Output** in the Stateflow chart and instruct the sending or receiving block in the Simulink model to inherit its type from the chart data. In many blocks, you can set data types through inheritance from the driving block, or through back propagation from the next block. You can set the data type of a Simulink block to match the data type of the Stateflow port to which it connects.

  For example, you can set the Constant block to inherit its type from the Stateflow **Input to Simulink** port that it supplies. Set the **Output data type** block parameter to `Inherit via back propagation`.

Implementation of Fixed-Point Data in Stateflow

Stateflow charts define fixed-point data types from the values that you specify for S, B, and the base integer type for Q.

- For each fixed-point data, the chart defines an integer variable for Q in the generated code. This integer is the only part of a fixed-point number that changes in value. The available base types for Q are the unsigned integer types `uint8`, `uint16`, and `uint32`, and the signed integer types `int8`, `int16`, and `int32`. If a fixed-point number has a slope $S = 1$ and a bias $B = 0$, it is equivalent to its quantized integer Q and behaves exactly as its base integer type.

- The slope S is factored into a coefficient F with $1 \leq F < 2$ and an integer power of two with exponent E:
  $$S = F \cdot 2^E.$$
  If the fractional slope F is greater than 1, it is converted into a fixed-point number. Encoding schemes with $F > 1$ can be computationally expensive, particularly in multiplication and division operations. Setting $F = 1$ avoids these computationally expensive instructions. In this setting, scaling by a power of 2 is implemented as bit shifts, which are more efficient than multiply instructions. Therefore, using binary-point-only scaling, in which $F = 1$ and $B = 0$, is recommended.

- Operations for fixed-point types are implemented with solutions for the quantized integer as described in "Arithmetic Operations for Fixed-Point Data" on page 22-32.

To generate efficient code, the fixed-point promotion rules choose values for slope and bias that cancel difficult terms in the solutions. See "Promotion Rules for Fixed-Point Operations" on page 22-34.

See Also

fixdt | fixptbestexp

More About

- "Fixed-Point Operations in Stateflow Charts" on page 22-32
- "Supported Operations for Fixed-Point Data" on page 22-21
- "Build a Low-Pass Filter by Using Fixed-Point Data" on page 22-15
- "Set Data Properties" on page 9-5

Model a Bang-Bang Controller by Using Fixed-Point Inputs

Run the Fixed-Point Bang-Bang Control Model

For this example, the model `sf_boiler` demonstrates an application of fixed-point data.
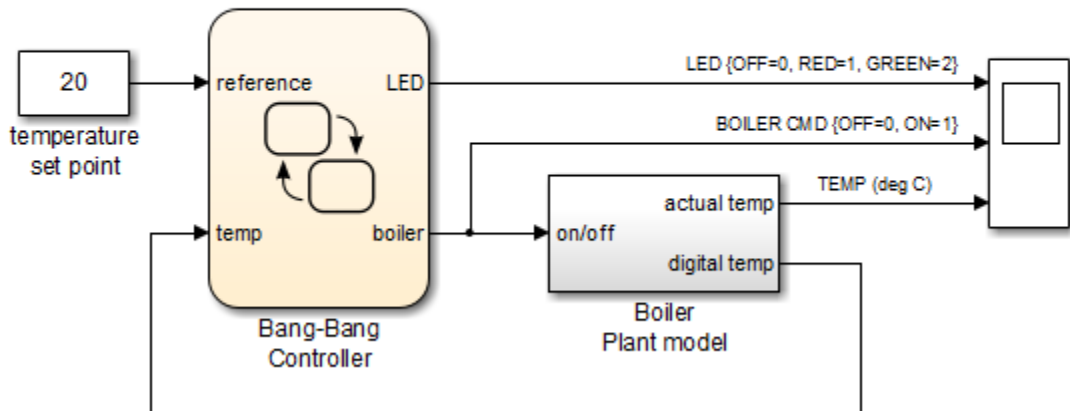
When you simulate the model, you get these results:
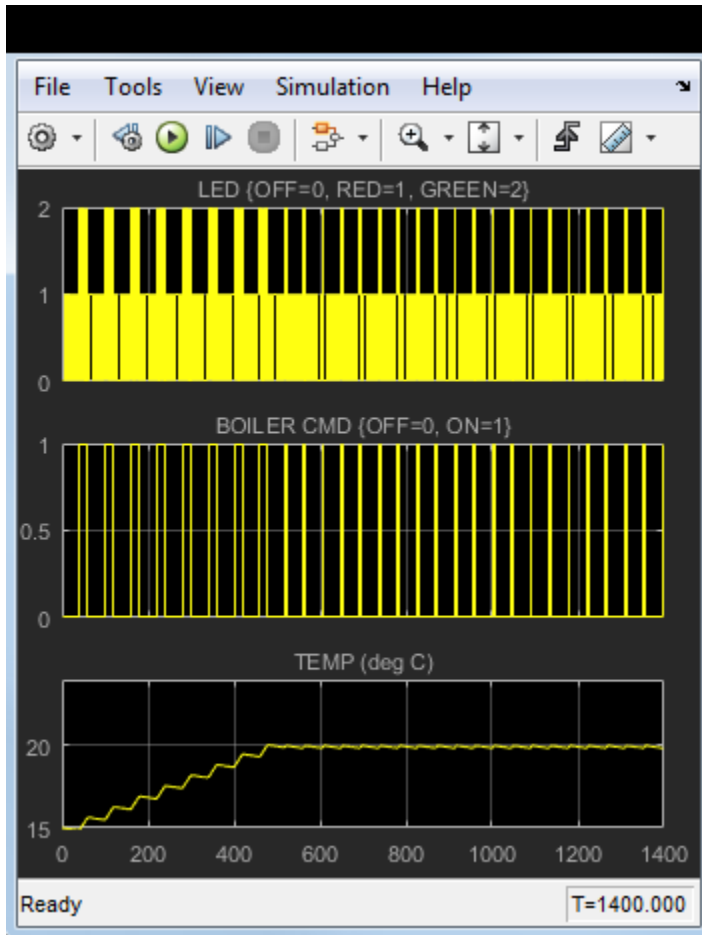
Explore the Fixed-Point "Bang-Bang Control" Model

To explore the model, follow these steps:

1 Double-click the Boiler Plant model subsystem block.

   The subsystem appears.
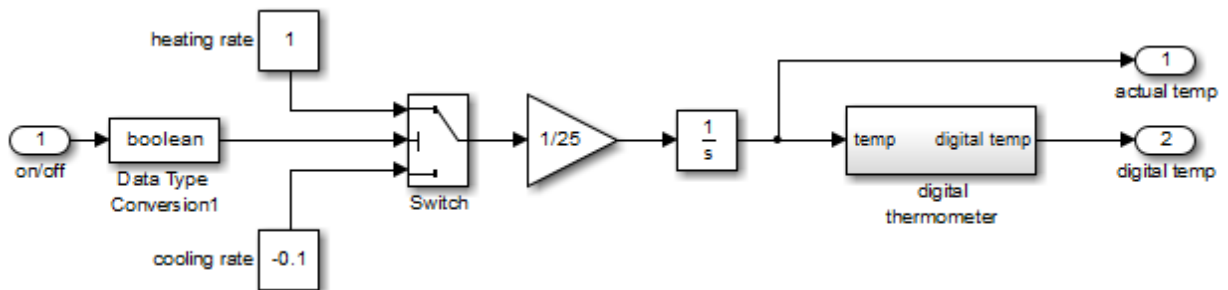
The Boiler Plant model subsystem simulates the temperature reaction of the boiler to periods of heating or cooling dictated by the Stateflow block. Depending on the Boolean value coming from the Controller, a temperature increment (+1 for heating, –0.1 for cooling) is added to the previous boiler temperature. The resulting boiler temperature is sent to the digital thermometer subsystem block.

2   In the Boiler Plant model subsystem, double-click the digital thermometer subsystem block.

The subsystem appears.
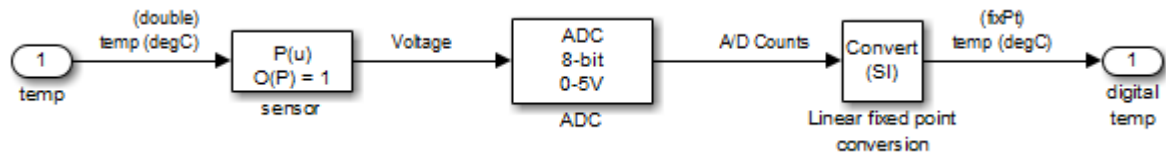
The digital thermometer subsystem produces an 8-bit fixed-point representation of the input temperature with the blocks described in the sections that follow.

Sensor Block

The sensor block converts input boiler temperature *(T)* to an intermediate analog voltage output *V* with a first-order polynomial that gives this output:

$$V = 0.05 \times T + 0.75$$

ADC Block

Double-click the ADC block to reveal these contents:
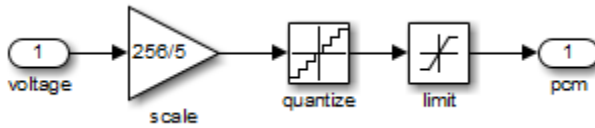
The ADC subsystem digitizes the analog voltage from the sensor block by multiplying the analog voltage by 256/5, rounding it to its integer floor, and limiting it to a maximum of 255 (the largest unsigned 8-bit integer value). Using the value for the output V from the sensor block, the new digital coded temperature output by the ADC block, $T_{digital}$, is given by this equation:

$$T_{digital} \quad = \quad (256/5) \quad \times \quad V \quad = \quad (256 \quad \times \quad 0.05/5) \quad \times \quad T \quad + \quad (256/5) \quad \times \quad 0.75$$

Linear fixed point conversion Block

The Linear fixed point conversion block informs the rest of the model that $T_{digital}$ is a fixed-point number with a slope value of 5/256/0.05 and an intercept value of –0.75/0.05. The Stateflow block Bang-Bang Controller receives this output and interprets it as a fixed-point number through the Stateflow data `temp`, which is scoped as **Input from Simulink** and set as an unsigned 8-bit fixed-point data with the same values for S and B set in the Linear fixed point conversion block.

The values for S and B are determined from the general expression for a fixed-point number:

$$V \qquad\qquad = \qquad\qquad SQ \qquad\qquad + \qquad\qquad B$$

Therefore,

$$Q \quad = \quad (V \quad – \quad B)/S \quad = \quad (1/S) \quad \times \quad V \quad + \quad (-1/S) \quad \times \quad B$$

Since $T_{digital}$ is now a fixed-point number, it is now the quantized integer Q of a fixed-point type. This means that $T_{digital} = Q$ of its fixed-point type, which gives this relation:

$$(1/S) \quad \times \quad V \quad + \quad (-1/S) \quad \times \quad B \quad = \quad (256 \quad \times \quad 0.05/5) \quad \times \quad T \quad + \quad (256/5) \quad \times \quad 0.75$$

Since T is the real-world value for the environment temperature, the above equation implies these relations:

$$V \qquad\qquad\qquad\qquad = \qquad\qquad\qquad\qquad T$$

and

$1/S$ = $(256 \times 0.05)/5$

S = $5/(256 \times 0.05)$ = 0.390625

and

$(-1/S) \times B$ = $(256/5) \times 0.75$

B = $-(256/5) \times 0.75 \times 5/(256 \times 0.05)$ = $-0.75/0.05$ = 15

By setting T_{digital} to be a fixed-point data as the output of the Linear fixed point conversion block and the input of the Stateflow block Bang-Bang Controller, the Stateflow chart interprets and processes this data automatically in an 8-bit environment with no need for any explicit conversions.

See Also

More About

- "Fixed-Point Data in Stateflow Charts" on page 22-2
- "Fixed-Point Operations in Stateflow Charts" on page 22-32
- "Supported Operations for Fixed-Point Data" on page 22-21

Build a Low-Pass Filter by Using Fixed-Point Data

This example shows how to build a Stateflow® chart that uses fixed-point data to implement a low-pass Butterworth filter. By designing the filter with fixed-point data instead of floating-point data, you can simulate your model using less memory. For more information, see "Fixed-Point Data in Stateflow Charts" on page 22-2.

```
{
y = x*b0 + x_n1*b1 - y_n1*a1;
x_n1 = x;
y_n1 = y;
}
```
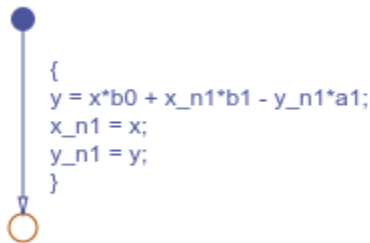
Build the Fixed-Point Butterworth Filter

The Low-Pass Filter chart is a stateless flow chart that accepts one input and provides one output. The chart contains these data symbols:

- x — **Scope**: Input, **Type**: Inherit:Same as Simulink
- y — **Scope**: Output, **Type**: fixdt(1,16,10)
- x_n1 — **Scope**: Local, **Type**: fixdt(1,16,12)
- y_n1 — **Scope**: Local, **Type**: fixdt(1,16,10)
- b0 — **Scope**: Parameter, **Type**: fixdt(1,16,15)
- b1 — **Scope**: Parameter, **Type**: fixdt(1,16,15)
- a1 — **Scope**: Parameter, **Type**: fixdt(1,16,15)

The values of b0, b1, and a1 are the coefficients of the low-pass Butterworth filter.

To build the Low-Pass Filter chart:

1. Create a Simulink® model with an empty Stateflow chart by entering sfnew at the MATLAB® command prompt.
2. In the Stateflow chart, add a flow chart with a single branch that assigns values to y, x_n1, and y_n1.

22-15

3 Add input, output, local, and parameter data to the chart, as described in "Add Stateflow Data" on page 9-2.

Define the Model Callback Function

Before loading the model, MATLAB calls the `butter` function to compute the values for the parameters `b0`, `b1`, and `a1`. The function constructs a first-order low-pass Butterworth filter with a normalized cutoff frequency of `(2*pi*Fc/(Fs/2))` radians per second, where:

- The sampling frequency is `Fs` = 1000 Hz.
- The cutoff frequency is `Fc` = 50 Hz.

The function output `B` contains the numerator coefficients of the filter in descending powers of `z`. The function output `A` contains the denominator coefficients of the filter in descending powers of `z`.

```
Fs = 1000;
Fc = 50;
[B,A] = butter(1,2*pi*Fc/(Fs/2));
b0 = B(1);
b1 = B(2);
a1 = A(2);
```

To define the preload callback for the model:

1 In the model window, select **File** > **Model Properties** > **Model Properties**.

2 In the Model Properties dialog box, on the **Callbacks** tab, select **PreLoadFcn**.

3 Enter the MATLAB code for the preload function call.

4 Click **OK**.

To load the parameter values to the MATLAB workspace, save, close, and reopen the model.

Add Other Blocks to the Model

To complete the model, add a Sine Wave block, a Data Type Conversion block, and a Scope block. Connect and label the blocks according to this diagram.
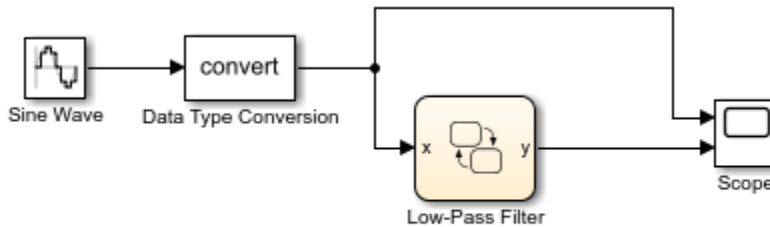
Copyright 2018 The MathWorks, Inc.

Sine Wave block

The Sine Wave block outputs a floating-point signal. The block has these settings:

- **Sine type**: Time based
- **Time**: Use simulation time
- **Amplitude**: 1
- **Bias**: 0
- **Frequency**: 2*pi*Fc
- **Phase**: 0
- **Sample time**: 1/Fs
- **Interpret vector parameters as 1-D**: On

Data Type Conversion block

The Data Type Conversion block converts the floating-point signal from the Sine Wave block to a fixed-point signal. By converting the signal to a fixed-point type, you can simulate your model using less memory. The block has these settings:

- **Output minimum**: []
- **Output maximum**: []
- **Output data type**: fixdt(1,16,14)
- **Lock output data type setting against changes by the fixed-point tools**: Off
- **Input and output to have equal**: Real World Value (RWV)
- **Integer rounding mode**: Floor

- **Saturate on integer overflow**: Off
- **Sample time**: -1

Scope block

The Scope block has two input ports that connect to the input and output signals for the Low-Pass Filter chart. To display the two signals separately, select a scope layout with two rows and one column.

Set Model Configuration Parameters

Because none of the blocks in the model have a continuous sample time, use a discrete solver with these configuration parameters:

- **Stop time**: 0.1
- **Type**: Fixed-step
- **Solver**: discrete (no continuous states)
- **Fixed-step size (fundamental sample time)**: 1/Fs

To configure the model:

1   In the Stateflow Editor, select **Simulation** > **Model Configuration Parameters**.
2   In the **Solver** pane, set the discrete solver parameters.
3   Click **OK**.

Run the Model

When you simulate the model, the Scope block displays two signals. The top signal shows the fixed-point version of the sine wave input to the chart. The bottom signal corresponds to the filtered output from the chart. The filter removes high-frequency values from the signal but allows low-frequency values to pass through the chart unchanged.
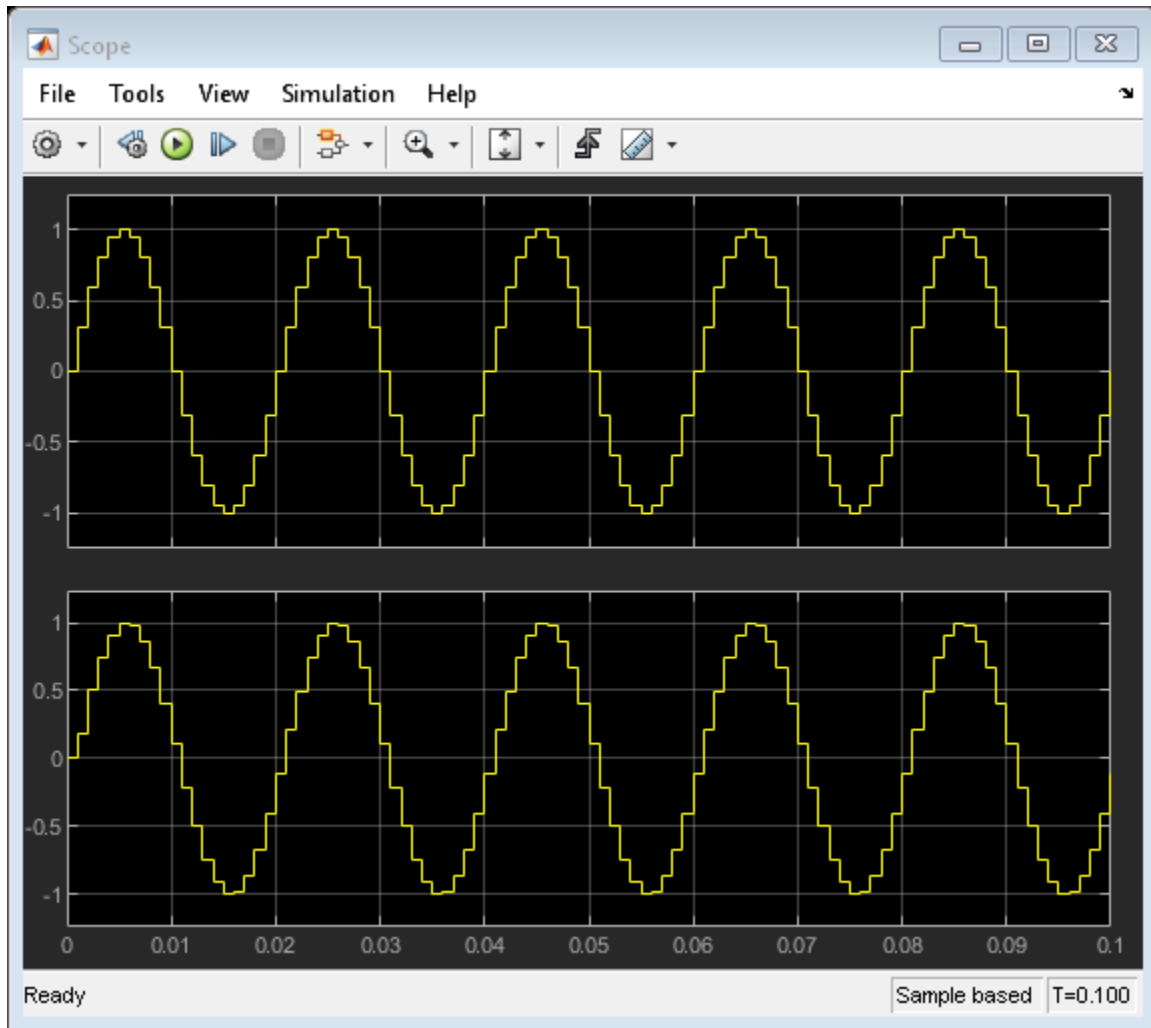
See Also

Data Type Conversion | Scope | Sine Wave | `butter` | `sfnew`

More About

Supported Operations for Fixed-Point Data

Stateflow charts in Simulink models have an action language property that defines the syntax that you use to compute with fixed-point data:

- MATLAB as the action language.

- C as the action language.

For more information, see "Differences Between MATLAB and C as Action Language Syntax" on page 13-6.

Binary Operations

This table summarizes the interpretation of all binary operations on fixed-point operands according to their order of precedence (0 = highest, 9 = lowest). Binary operations are left associative so that, in any expression, operators with the same precedence are evaluated from left to right.

Operation	Precedence	MATLAB as the Action Language	C as the Action Language
a ^ b	0	Power. Not supported for fixed-point operands defined by using either a slope that is not an integer power of two or a nonzero bias. Exponent operand must be a constant whose value is a non-negative integer.	Power. Enable this operation by clearing the **Enable C-bit operations** chart property. See "Enable C-Bit Operations" on page 24-6.
a * b	1	Multiplication. For fixed-point operands defined by using either a slope that is not an integer power of two or a nonzero bias, specify a chart `fimath` object with `ProductMode` set to `SpecifyPrecision`. See "Multiplication" on page 22-36.	Multiplication. Not supported for fixed-point operands defined by using a nonzero bias. See "Multiplication" on page 22-36.

Operation	Precedence	MATLAB as the Action Language	C as the Action Language
a / b	1	Division. Not supported for fixed-point operands defined by using either a slope that is not an integer power of two or a nonzero bias. See "Division" on page 22-37.	Division. Not supported for fixed-point operands defined by using a nonzero bias. See "Division" on page 22-37.
a + b	2	Addition. For fixed-point operands defined by using either a slope that is not an integer power of two or a nonzero bias, specify a chart `fimath` object with `SumMode` set to `SpecifyPrecision`. See "Addition and Subtraction" on page 22-35.	Addition. See "Addition and Subtraction" on page 22-35.
a - b	2	Subtraction. For fixed-point operands defined by using either a slope that is not an integer power of two or a nonzero bias, specify a chart `fimath` object with `SumMode` set to `SpecifyPrecision`. See "Addition and Subtraction" on page 22-35.	Subtraction. See "Addition and Subtraction" on page 22-35.
a > b	3	Comparison, greater than. See "Relational Operations for Fixed-Point Data" on page 22-33.	Comparison, greater than. Not supported for fixed-point operands with mismatched biases. See "Relational Operations for Fixed-Point Data" on page 22-33.

Operation	Precedence	MATLAB as the Action Language	C as the Action Language
a < b	3	Comparison, less than. See "Relational Operations for Fixed-Point Data" on page 22-33.	Comparison, less than. Not supported for fixed-point operands with mismatched biases. See "Relational Operations for Fixed-Point Data" on page 22-33.
a >= b	3	Comparison, greater than or equal to. See "Relational Operations for Fixed-Point Data" on page 22-33.	Comparison, greater than or equal to. Not supported for fixed-point operands with mismatched biases. See "Relational Operations for Fixed-Point Data" on page 22-33.
a <= b	3	Comparison, less than or equal to. See "Relational Operations for Fixed-Point Data" on page 22-33.	Comparison, less than or equal to. Not supported for fixed-point operands with mismatched biases. See "Relational Operations for Fixed-Point Data" on page 22-33.
a == b	4	Comparison, equal to. See "Relational Operations for Fixed-Point Data" on page 22-33.	Comparison, equal to. Not supported for fixed-point operands with mismatched biases. See "Relational Operations for Fixed-Point Data" on page 22-33.
a ~= b	4	Comparison, not equal to. See "Relational Operations for Fixed-Point Data" on page 22-33.	Comparison, not equal to. Not supported for fixed-point operands with mismatched biases. See "Relational Operations for Fixed-Point Data" on page 22-33.

Operation	Precedence	MATLAB as the Action Language	C as the Action Language
a != b	4	Not supported. Use the operation a ~= b. See "Relational Operations for Fixed-Point Data" on page 22-33.	Comparison, not equal to. Not supported for fixed-point operands with mismatched biases. See "Relational Operations for Fixed-Point Data" on page 22-33.
a <> b	4	Not supported. Use the operation a ~= b. See "Relational Operations for Fixed-Point Data" on page 22-33.	Comparison, not equal to. Not supported for fixed-point operands with mismatched biases. See "Relational Operations for Fixed-Point Data" on page 22-33.
a && b	8	Logical AND. See "Logical Operations for Fixed-Point Data" on page 22-33.	Logical AND. See "Logical Operations for Fixed-Point Data" on page 22-33.
a || b	9	Logical OR. See "Logical Operations for Fixed-Point Data" on page 22-33.	Logical OR. See "Logical Operations for Fixed-Point Data" on page 22-33.

Unary Operations and Actions

This table summarizes the interpretation of all unary operations and actions on fixed-point operands. Unary operations:

- Have higher precedence than binary operators.
- Are right associative so that, in any expression, they are evaluated from right to left.

Operation	MATLAB as the Action Language	C as the Action Language
~a	Not supported. Use the expression a == cast(0,'like',a). See "Logical Operations for Fixed-Point Data" on page 22-33.	Logical NOT. Enable this operation by clearing the **Enable C-bit operations** chart property. See "Logical Operations for Fixed-Point Data" on page 22-33 and "Enable C-Bit Operations" on page 24-6.

Operation	MATLAB as the Action Language	C as the Action Language
!a	Not supported. Use the expression a == cast(0,'like',a). See "Logical Operations for Fixed-Point Data" on page 22-33.	Logical NOT. See "Logical Operations for Fixed-Point Data" on page 22-33.
-a	Negative. See "Unary Minus" on page 22-38.	Negative. See "Unary Minus" on page 22-38.
a++	Not supported. Use the expression a = a+1.	Increment. Equivalent to a = a+1.
a--	Not supported. Use the expression a = a-1.	Decrement. Equivalent to a = a-1.

Assignment Operations

This table summarizes the interpretation of assignment operations on fixed-point operands.

Operation	MATLAB as the Action Language	C as the Action Language
a = b	Simple assignment.	Simple assignment.
a := b	Not supported. To override fixed-point promotion rules, use explicit type cast operations. See "Type Cast Operations" on page 12-19.	Special assignment that overrides fixed-point promotion rules. See "Override Fixed-Point Promotion in C Charts" on page 22-25.
a += b	Not supported. Use the expression a = a+b.	Equivalent to a = a+b.
a -= b	Not supported. Use the expression a = a-b.	Equivalent to a = a-b.
a *= b	Not supported. Use the expression a = a*b.	Equivalent to a = a*b.
a /= b	Not supported. Use the expression a = a/b.	Equivalent to a = a/b.

Override Fixed-Point Promotion in C Charts

In charts that use C as the action language, a simple assignment of the form a = b calculates an intermediate value for b according to the fixed-point promotion rules. Then

this intermediate value is cast to the type of a by using an online conversion. See "Promotion Rules for Fixed-Point Operations" on page 22-34 and "Conversion Operations" on page 22-4. Simple assignments are most efficient when both types have equal bias and slopes that either are equal or are both powers of two.

In contrast, a special assignment of the form a := b overrides this behavior by initially using the type of a as the result type for the value of b.

- Constants in b are converted to the type of a by using offline conversions.
- The expression b can contain at most one arithmetic operator (+, -, *, or /). The result is determined by using an online conversion.
- If b contains anything other than an arithmetic operation or a constant, then the special assignment operation behaves like the simple assignment operation (=).

Use the special assignment operation := when you want to:

- Avoid an overflow in an arithmetic operation. For example, see "Avoid Overflow in Fixed-Point Addition" on page 22-26.
- Retain precision in a multiplication or division operation. For example, see "Improve Precision in Fixed-Point Division" on page 22-27.

Note Using the special assignment operation := can result in generated code that is less efficient than the code you generate by using the normal fixed-point promotion rules.

Avoid Overflow in Fixed-Point Addition

You can use the special assignment operation := to avoid overflow when performing an arithmetic operation on two fixed-point numbers. For example, consider a chart that computes the sum a+b where $a = 2^{12}-1 = 4095$ and b = 1.

Suppose that:

- Both inputs are signed 16-bit fixed-point numbers with three fraction bits (type fixdt(1,16,3)).
- The output c is a signed 32-bit fixed-point number with three fraction bits (type fixdt(1,32,3)).
- The integer word size for production targets is 16 bits.

Because the target integer size is 16 bits, the simple assignment `c = a+b` adds the inputs in 16 bits before casting the sum to 32 bits. The intermediate result is 4096, which, as a type `fixdt(1,16,3)` value, results in an overflow.

In contrast, the special assignment `c := a+b` casts the inputs to 32 bits before computing the sum. The result of 4096 is safely computed as a type `fixdt(1,32,3)` value without an overflow.

Improve Precision in Fixed-Point Division

You can use the special assignment operation `:=` to obtain a more precise result when multiplying or dividing two fixed-point numbers. For example, consider a chart that computes the ratio `a/b` where `a` = 2 and `b` = 3.

Suppose that:

- The input `a` is a fixed-point number with four fraction bits (type `fixdt(1,16,4)`).
- The input `b` is a fixed-point number with three fraction bits (type `fixdt(1,16,3)`).
- The output `c` is a signed 16-bit fixed-point number with six fraction bits (type `fixdt(1,16,6)`).

The inputs correspond to these slopes and quantized integers:
$$S_a = 2^{-4}, Q_a = 32$$
$$S_b = 2^{-3}, Q_b = 24.$$

The simple assignment `c = a/b` first calculates an intermediate value for `a/b` according to the fixed-point promotion rules. The quantized integer is rounded to the floor:
$$S_{\text{int}} = S_a/S_b = 2^{-4}/2^{-3} = 2^{-1}$$
$$Q_{\text{int}} = Q_a/Q_b = 32/24 \approx 1.$$
The intermediate result is then cast as a signed 16-digit fixed-point number with six fraction bits:
$$S_c = 2^{-6} = 1/64$$
$$Q_c = S_{\text{int}}Q_{\text{int}}/S_c = 2^{-1}/2^{-6} = 2^5 = 32.$$
Therefore, the approximate real-world value for `c` is $\tilde{V}_c = S_cQ_c = 32/64 = 0.5$. This result is not a good approximation of the actual value of 2/3.

In contrast, the special assignment `c := a/b` calculates `a/b` directly as a signed 16-digit fixed-point number with six fraction bits. Again, the quantized integer is rounded to the floor:
$$S_c = 2^{-6} = 1/64$$
$$Q_c = (S_aQ_a)/(S_cS_bQ_b) = 128/3 \approx 42.$$

Therefore, the approximate real-world value for c is $\tilde{V}_c = S_c Q_c = 42/64 = 0.6563$. This result is a better approximation to the actual value of 2/3.

Compare Results of Fixed-Point Arithmetic

This example shows the difference between various implementations of fixed-point arithmetic in Stateflow charts. The model contains three charts that calculate the ratio a/b where a = 19 and b = 24. Both inputs are signed 16-digit fixed-point numbers with one fraction bit (type `fixdt(1,16,1)`). They correspond to these slopes and quantized integers:

$$S_a = 2^{-1}, Q_a = 38$$

$$S_b = 2^{-1}, Q_b = 48$$

The model calculates the value of a/b as a floating-point number of type `fixdt(1,16,1)` in three different ways:

- A type casting operation in a chart that uses MATLAB as the action language.
- A simple assignment operation in a chart that uses C as the action language.
- A special assignment operation in a chart that uses C as the action language.
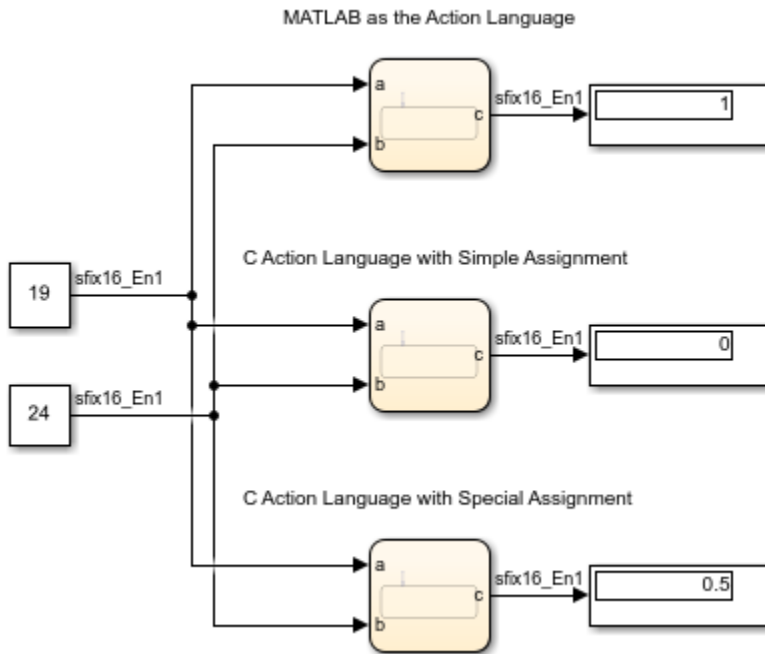
Type Casting in Chart That Uses MATLAB as the Action Language

The chart at the top of the model computes an intermediate value for `a/b`. The quantized integer for the intermediate value is rounded to the nearest integer:

$$S_{\text{int}} = S_{\text{a}}/S_{\text{b}} = 1$$

$$Q_{\text{int}} = Q_{\text{a}}/Q_{\text{b}} = 38/48 \approx 1.$$

The intermediate value is then cast as a signed 16-digit fixed-point number `c` with one fraction bit:

$$S_{\text{c}} = 2^{-1}$$

$$Q_{\text{c}} = S_{\text{int}} \cdot Q_{\text{int}}/S_{\text{c}} = 2.$$

The output value from this chart is

$$\tilde{V}_{\mathbf{c}} = S_{\mathbf{c}} \cdot Q_{\mathbf{c}} = 1.$$

Simple Assignment in Chart That Uses C as the Action Language

The middle chart also computes an intermediate value for `a/b`. In this case, the quantized integer for the intermediate value is rounded to the floor:

$$S_{\texttt{int}} = S_{\mathbf{a}}/S_{\mathbf{b}} = 1$$

$$Q_{\texttt{int}} = Q_{\mathbf{a}}/Q_{\mathbf{b}} = 38/48 \approx 0.$$

The intermediate value is then cast as a signed 16-digit fixed-point number `c` with one fraction bit:

$$S_{\mathbf{c}} = 2^{-1}$$

$$Q_{\mathbf{c}} = S_{\texttt{int}} \cdot Q_{\texttt{int}}/S_{\mathbf{c}} = 0.$$

The output value from this chart is

$$\tilde{V}_{\mathbf{c}} = S_{\mathbf{c}} \cdot Q_{\mathbf{c}} = 0.$$

Special Assignment in Chart That Uses C as the Action Language

The chart at the bottom of the model uses a special assignment of the form `c := a/b`. The value of the division is calculated directly as a signed 16-digit fixed-point number with one fraction bit. The quantized integer is rounded to the floor:

$$S_{\mathbf{c}} = 2^{-1}$$

$$Q_{\mathbf{c}} = (S_{\mathbf{a}} \cdot Q_{\mathbf{a}})/(S_{\mathbf{c}} \cdot S_{\mathbf{b}} \cdot Q_{\mathbf{b}}) = 19/12 \approx 1.$$

Therefore, the output value from this chart is

$$\tilde{V}_{\mathbf{c}} = S_{\mathbf{c}} \cdot Q_{\mathbf{c}} = 0.5.$$

The three results exhibit loss of precision compared to the floating-point answer of 19/24 = 0.7917. To minimize the loss of precision to an acceptable level in your application, adjust the encoding scheme in your fixed-point data.

See Also

More About

- "Fixed-Point Data in Stateflow Charts" on page 22-2
- "Fixed-Point Operations in Stateflow Charts" on page 22-32
- "Build a Low-Pass Filter by Using Fixed-Point Data" on page 22-15

Fixed-Point Operations in Stateflow Charts

Fixed-point numbers use integers and integer arithmetic to approximate real numbers. They enable you to perform computations involving real numbers without requiring floating-point support in underlying system hardware. For more information, see "Fixed-Point Data in Stateflow Charts" on page 22-2.

Arithmetic Operations for Fixed-Point Data

The general equation for a binary arithmetic operation between fixed-point operands is

c = a *<op>* b

where a and b are fixed-point numbers and <op> refers to addition, subtraction, multiplication, or division. The result of the operation is a fixed-point number c of the form

$$\tilde{V}_c = S_c Q_c + B_c.$$

The fixed-point type for c determines the slope S_c, the bias B_c, and the number of bits used to store the quantized integer Q_c. For each arithmetic operation, this table lists the value of the quantized integer Q_c in terms of the values of the operands (a and b) and the fixed-point type for c.

Operation		Slope and Bias Scaling	Binary-Point Scaling
Addition	c = a+b	$Q_c = \text{round}((S_a/S_c)Q_a + (S_b/S_c)Q_b + (B_a + B_b - B_c)/S_c)$	$Q_c = \text{round}((S_a/S_c)Q_a + (S_b/S_c)Q_b)$
Subtraction	c = a-b	$Q_c = \text{round}((S_a/S_c)Q_a - (S_b/S_c)Q_b - (B_a - B_b - B_c)/S_c)$	$Q_c = \text{round}((S_a/S_c)Q_a - (S_b/S_c)Q_b)$
Multiplication	c = a*b	$Q_c = \text{round}((S_aS_b/S_c)Q_aQ_b + (B_aS_b/S_c)Q_a + (B_bS_a/S_c)Q_b + (B_aB_b - B_c)/S_c)$	$Q_c = \text{round}((S_aS_b/S_c)Q_aQ_b)$
Division	c = a/b	$Q_c = \text{round}((S_aQ_a + B_a)/(S_c(S_bQ_b + B_b)) - (B_c/S_c))$	$Q_c = \text{round}((S_a/(S_bS_c))Q_a/Q_b)$

To simplify these expressions and avoid computationally expensive operations, use binary-point scaling to encode all your fixed-point data. With this setting, the slope is an integer power of two and the bias is zero. Then, all fixed-point operations consist of integer arithmetic and bit shifts on the quantized integers.

Note The result of an arithmetic operation depends on the rounding method that computes the value of the quantized integer Q_c. For more information, see "Conversion Operations" on page 22-4.

Relational Operations for Fixed-Point Data

You can use relational operations to compare fixed-point data. Comparing fixed-point values of different types can yield unexpected results because each operand is converted to a common type for comparison. Because of rounding or overflow errors during the conversion, some values that appear to be equal are not equal as fixed-point numbers.

For example, suppose that a and b represent the real-world value $V = 2.2$ in two different fixed-point encoding schemes:

- a is a fixed-point number with slope $S_a = 0.3$ and bias $B_a = 0.1$. The quantized integer for a is:
  $Q_a = (V - B_a)/S_a = (2.2 - 0.1)/0.3 = 7.$
- b is a fixed-point number with slope $S_b = 0.7$ and bias $B_b = 0.1$. The quantized integer for b is:
  $Q_b = (V - B_b)/S_b = (2.2 - 0.1)/0.7 = 3.$

To compare these values, the chart first converts them to a common fixed-point type with slope $S_{comp} = 1.06811 \cdot 10^{-5} \approx S_a/28087 \approx S_b \cdot 2^{-16}$ and bias $B_{comp} = 0.1$. (In this case, the slope S_{comp} arises from the approximate value of $S_a/S_b = 0.3/0.7 \approx 28087 \cdot 2^{-16}$.) In this common encoding scheme, a and b correspond to these quantized integers:
  $Q_{a'} = S_a Q_a/S_{comp} = Q_a(S_a/S_{comp}) \approx 7 \; \square \; 28087 = 196609$
  $Q_{b'} = S_b Q_b/S_{comp} = Q_b(S_b/S_{comp}) \approx 3 \; \square \; 2^{16} = 196608.$
After the conversion, the quantized integers are different. Although a and b represent the same real-world value, they are not equal as fixed-point numbers.

Note In charts that use C as the action language, comparisons of fixed-point operands with mismatched biases are not supported.

Logical Operations for Fixed-Point Data

In a logical operation, a fixed-point operand a is interpreted as `false` if it corresponds to the real-world value for zero in the fixed-point type of a. Otherwise, a is interpreted as `true`.

- In charts that use MATLAB as the action language, using `a` in a logical operation is equivalent to the expression `a ~= cast(0,'like',a)`.

- In charts that use C as the action language, using `a` in a logical operation is equivalent to the expression `a != 0c`, where `0c` is a fixed-point context-sensitive constant. See "Fixed-Point Context-Sensitive Constants" on page 22-5.

For example, suppose that `a` is a fixed-point number with a slope of $S_a = 0.25$ and a bias of $B_a = 5.1$. Using `a` in a logical operation is equivalent to testing whether the quantized integer Q_a satisfies the condition

$$Q_a = \text{round}((0 - B_a)/S_a) = \text{round}(-5.1 / 0.25) = \text{round}(-20.4) = -20.$$

Therefore, `a` is equivalent to `false` when its real-world approximation is

$$\tilde{V}_a = S_a Q_a + B_a = 0.25 \ (\,-20) + 5.1 = 0.1.$$

Promotion Rules for Fixed-Point Operations

The rules for selecting the numeric type used for the result of an operation on fixed-point numbers are called *fixed-point promotion rules*. These rules help to maintain computational efficiency and usability.

The fixed-point promotion rules determine a result type for an operation `c = a <op> b` by selecting the slope S_c, the bias B_c, and the number of bits w_c used to store the quantized integer Q_c. These parameters depend on the fixed-point types of the operands `a` and `b`, the operation `<op>` to be performed, and the action language property for the chart.

- In a chart that uses MATLAB as the action language, you control the fixed-point promotion rules through the fixed-point properties for the chart. See "Fixed-Point Properties" on page 24-9.

  - If you set the **MATLAB Chart fimath** property to `Same as MATLAB`, then arithmetic operations follow the default fixed-point promotion rules for MATLAB. See "Perform Fixed-Point Arithmetic" (Fixed-Point Designer).

  - If you specify a chart `fimath` object with `SumMode` and `ProductMode` set to `SpecifyPrecision`, then you can define the word length, slope, and bias for all sums and products explicitly. See "fimath Object Properties" (Fixed-Point Designer).

- In a chart that uses C as the action language, the fixed-point promotion rules determine the type for an intermediate value of the result. This intermediate value is then cast to the type that you specify for `c`.

For all arithmetic operations, the default number of bits w_c used to store the quantized integer is the larger value between:

- The maximum number of bits in the operand types (w_a and w_b).
- The number of bits in the integer word size for the target machine (w_{int}).

To set the value of w_{int}, open the **Model Configuration Parameters** dialog box. On the **Hardware Implementation** pane, select `Custom Processor` from the **Device vendor** drop-down list and enter the target integer word size in the **int** field. For more information, see "Hardware Implementation Pane" (Simulink).

You can avoid overflow and improve the precision in your floating-point operations by using the special assignment operation of the form `c := a <op> b`. The special assignment operation does not follow the fixed-point promotion rules. Instead, the chart determines the result of the operation by using the type that you specify for `c`. See "Override Fixed-Point Promotion in C Charts" on page 22-25.

Addition and Subtraction

By default, charts that use MATLAB as the action language support addition and subtraction only on fixed-point data defined through binary-point scaling. If either operand is a signed fixed-point number, then the result is also signed. The choice of word length accommodates the integer and fraction parts of each operand in addition to a possible carry bit. The fraction length of the result is equal to the fraction length of the most precise operand. To perform addition and subtraction on fixed-point data defined by using either a slope that is not an integer power of two or a nonzero bias, specify a chart `fimath` object with `SumMode` set to `SpecifyPrecision`.

Charts that use C as the action language support addition and subtraction for operands of all fixed-point data types. The result is a signed fixed-point number only if both operands are signed. Mixing signed and unsigned operands can yield unexpected results and is not recommended. The slope of the result is equal to the slope of the least precise operand. To simplify calculations and yield efficient code, the biases of the two inputs are added for an addition operation and subtracted for a subtraction operation.

	a	b	MATLAB as the Action Language	C as the Action Language
Sign	s_a	s_b	$s_c = s_a \mid\mid s_b$	$s_c = s_a$ && s_b
Word length	w_a	w_b	$w_c = \max(w_a - f_a, w_b - f_b) + \max(f_a, f_b) + 1$	$w_c = \max(w_a, w_b, w_{int})$

	a	b	MATLAB as the Action Language	C as the Action Language
Fraction length	f_a	f_b	$f_c = \max(f_a, f_b)$	$f_c = \min(f_a, f_b)$
Slope	S_a (2^{-f_a} if using binary-point scaling)	S_b (2^{-f_b} if using binary-point scaling)	$S_c = \min(S_a, S_b)$	$S_c = \max(S_a, S_b)$
Bias	B_a (0 if using binary-point scaling)	B_b (0 if using binary-point scaling)	$B_c = 0$	$B_c = B_a + B_b$ for addition or $B_c = B_a - B_b$ for subtraction

Multiplication

By default, charts that use MATLAB as the action language support multiplication only on fixed-point data defined through binary-point scaling. If either operand is a signed fixed-point number, then the result is also signed. A full precision product requires a word length equal to the sum of the word lengths of the operands. The fraction length of a product is the sum of the fraction lengths of the operands. To perform multiplication on fixed-point data defined by using either a slope that is not an integer power of two or a nonzero bias, specify a chart `fimath` object with `ProductMode` set to `SpecifyPrecision`.

Charts that use C as the action language support multiplication only on fixed-point data operands defined by nonzero biases. The result is a signed fixed-point number only if both operands are signed. Mixing signed and unsigned operands can yield unexpected results and is not recommended. The slope of a product is the product of the slopes of the operands.

	a	b	MATLAB as the Action Language	C as the Action Language
Sign	s_a	s_b	$s_c = s_a \mid\mid s_b$	$s_c = s_a$ && s_b
Word length	w_a	w_b	$w_c = w_a + w_b$	$w_c = \max(w_a, w_b, w_{int})$

	a	b	MATLAB as the Action Language	C as the Action Language
Fraction length	f_a	f_b	$f_c = f_a + f_b$	$f_c = f_a + f_b$
Slope	S_a (2^{-f_a} if using binary-point scaling)	S_b (2^{-f_b} if using binary-point scaling)	$S_c = S_a S_b$	$S_c = S_a S_b$
Bias	$B_a = 0$	$B_b = 0$	$B_c = 0$	$B_c = 0$

Division

Charts that use MATLAB as the action language support division only on fixed-point data defined through binary-point scaling. If either operand is a signed fixed-point number, then the result is also signed. A full precision quotient requires a word length equal to the maximum number of bits in the operands. The fraction length of a quotient is the difference of the fraction lengths of the operands.

Charts that use C as the action language support division for fixed-point data operands defined by nonzero biases. The result is a signed fixed-point number only if both operands are signed. Mixing signed and unsigned operands can yield unexpected results and is not recommended. The slope of a quotient is the quotient of the slopes of the operands.

	a	b	MATLAB as the Action Language	C as the Action Language
Sign	s_a	s_b	$s_c = s_a \,||\, s_b$	$s_c = s_a \,\&\&\, s_b$
Word length	w_a	w_b	$w_c = \max(w_a, w_b)$	$w_c = \max(w_a, w_b, w_{int})$
Fraction length	f_a	f_b	$f_c = f_a - f_b$	$f_c = f_a - f_b$
Slope	S_a (2^{-f_a} if using binary-point scaling)	S_b (2^{-f_b} if using binary-point scaling)	$S_c = S_a / S_b$	$S_c = S_a / S_b$
Bias	$B_a = 0$	$B_b = 0$	$B_c = 0$	$B_c = 0$

Unary Minus

The only unary operation that requires a promotion of its result type is the unary minus operation `c = -a`. Taking the negative of an unsigned fixed-point number can yield unexpected results and is not recommended. The word size of the result depends on the action language property of the chart. The slope of the result is equal to the slope of the operand. The bias of the result type is the negative of the bias of the operand.

	a	MATLAB as the Action Language	C as the Action Language
Sign	s_a	$s_c = s_a$	$s_c = s_a$
Word length	w_a	$w_c = w_a$	$w_c = \max(w_a, w_{\text{int}})$
Fraction length	f_a	$f_c = f_a$	$f_c = f_a$
Slope	S_a (2^{-f_a} if using binary-point scaling)	$S_c = S_a$	$S_c = S_a$
Bias	B_a (0 if using binary-point scaling)	$B_c = -B_a$	$B_c = -B_a$

Arithmetic with Mixed Numeric Types

This table summarizes the fixed-point promotion rules for a binary operation between a fixed-point number and an operand of a different numeric type.

Numeric Type of Second Operand	MATLAB as the Action Language	C as the Action Language
Floating-point numbers: • `single` • `double`	Before performing the operation, the chart casts the floating-point operand to a fixed-point number. The type used for the cast depends on the operation: • Addition and subtraction operations use the same type as the fixed-point operand. • Multiplication operations use the same word length and signedness as the fixed-point operand, and the best precision fraction length for a fixed-point result. The result of the operation is a fixed-point number.	Before performing the operation, the chart casts the fixed-point operand to a floating-point number . The casting operation uses the same type (`single` or `double`) as the floating-point operand. The result of the operation is a floating-point number.
Integers: • `int32` • `int16` • `int8` • `uint32` • `uint16` • `uint8`	The integer operand is treated as a fixed-point number of the same word length and signedness with slope $S = 1$ and bias $B = 0$. The result of the operation is a fixed-point number.	The integer operand is treated as a fixed-point number of the same word length and signedness with slope $S = 1$ and bias $B = 0$. The result of the operation is a fixed-point number.

See Also

More About

• "Fixed-Point Data in Stateflow Charts" on page 22-2
• "Supported Operations for Fixed-Point Data" on page 22-21

- "Build a Low-Pass Filter by Using Fixed-Point Data" on page 22-15

23

Complex Data

Complex Data in Stateflow Charts

Complex data is data whose value is a complex number. For example, in a Stateflow chart in Simulink model, an input signal with the value `3 + 5i` is complex. See "Complex Signals" (Simulink).

Define Complex Data

1   Add a data object to your chart, as described in "Add Stateflow Data" on page 9-2.
2   Set the **Complexity** property for the data object to `On`. For more information, see "Complexity" on page 9-8.
3   Specify the name, scope, size, base type, and other properties for the data object, as described in "Set Data Properties" on page 9-5.

  • Complex data does not support the scope `Constant`.
  • Complex data does not support the base types `ml`, `struct`, and `boolean`.

When to Use Complex Data

Use complex data when you model applications in communication systems and digital signal processing. For example, you can use this design pattern to model a frame synchronization algorithm in a communication system:

1   Use Simulink blocks (such as filters) to process complex signals.
2   Use charts to implement mode logic for frame synchronization.
3   Let the charts access complex input and output data so that nested MATLAB functions can drive the mode logic.

For an example of modeling a frame synchronization algorithm, see "Detect Valid Transmission Data Using Frame Synchronization" on page 23-15.

Note Continuous-time variables of complex type are *not* supported. For more information, see "Store Continuous State Information in Local Variables" on page 21-9.

Where You Can Use Complex Data

You can define complex data at these levels of the Stateflow hierarchy:

- Charts
- Subcharts
- States
- Functions

How You Can Use Complex Data

You can use complex data to define:

- Complex vectors
- Complex matrices

You can also use complex data as arguments for:

- State actions
- Transition actions
- MATLAB functions (see "Reuse MATLAB Code by Defining MATLAB Functions" on page 28-2)
- Truth table functions (see "Reuse Combinatorial Logic by Defining Truth Tables" on page 27-2)
- Graphical functions (see "Reuse Logic Patterns by Defining Graphical Functions" on page 8-16)
- Change detection operators (see "Detect Changes in Data Values" on page 12-69)

Note Exported functions do not support complex data as arguments.

See Also

More About

Supported Operations for Complex Data

Stateflow charts in Simulink models have an action language property that defines the syntax that you use to compute with complex data. The action language properties are:

- MATLAB as the action language.

- C as the action language.

For more information, see "Differences Between MATLAB and C as Action Language Syntax" on page 13-6.

Notation for Complex Data

In charts that use MATLAB as the action language, you can define complex data by using complex number notation `a + bi`, where `a` and `b` are real numbers. For example, this statement assigns a value of 3+4i to `x`:

```
x = 3 + 4i;
```

Alternatively, you can define complex data by using the `complex` operator:

```
complex(<real_part>,<imag_part>)
```

`<real_part>` and `<imag_part>` are arguments that define the real and imaginary parts of the complex number, respectively. The two arguments must be real values or expressions that evaluate to real values. As in the preceding example, this statement assigns a value of 3+4i to `x`:

```
x = complex(3,4);
```

Charts that use C as the action language do not support complex number notation `a + bi`. To define a complex number based on two real values, use the `complex` operator.

Binary Operations

This table summarizes the interpretation of all binary operations on complex operands according to their order of precedence (1 = highest, 3 = lowest). Binary operations are left associative so that, in any expression, operators with the same precedence are evaluated from left to right.

Operation	Precedence	MATLAB as the Action Language	C as the Action Language
a * b	1	Multiplication.	Multiplication.
a / b	1	Division.	Not supported. Use the \ operation in a MATLAB function. See "Perform Complex Division with a MATLAB Function" on page 23-12.
a + b	2	Addition.	Addition.
a - b	2	Subtraction.	Subtraction.
a == b	3	Comparison, equal to.	Comparison, equal to.
a ~= b	3	Comparison, not equal to.	Comparison, not equal to.
a != b	3	Not supported. Use the operation a ~= b.	Comparison, not equal to.
a <> b	3	Not supported. Use the operation a ~= b.	Comparison, not equal to.

Unary Operations and Actions

This table summarizes the interpretation of all unary operations and actions on complex data. Unary operations:

- Have higher precedence than the binary operators.
- Are right associative so that, in any expression, they are evaluated from right to left.

Operation	MATLAB as the Action Language	C as the Action Language
-a	Negative.	Negative.
a++	Not supported. Use the expression a = a+1.	Increment. Equivalent to a = a+1.
a--	Not supported. Use the expression a = a-1.	Decrement. Equivalent to a = a-1.

Assignment Operations

This table summarizes the interpretation of assignment operations in Stateflow charts.

Operation	MATLAB as the Action Language	C as the Action Language
a = b	Simple assignment.	Simple assignment.
a += b	Not supported. Use the expression a = a+b.	Equivalent to a = a+b.
a -= b	Not supported. Use the expression a = a-b.	Equivalent to a = a-b.
a *= b	Not supported. Use the expression a = a*b.	Equivalent to a = a*b.

Access Real and Imaginary Parts of a Complex Number

To access the real and imaginary parts of a complex number, use the `real` and `imag` operators.

real Operator

The `real` operator returns the value of the real part of a complex number:

```
real(<complex_expr>)
```

<complex_expr> is an expression that evaluates to a complex number. For example, if `frame(200)` evaluates to the complex number `8.23 + 4.56i`, this expression returns a value of `8.2300`:

```
real(frame(200))
```

imag Operator

The `imag` operator returns the value of the imaginary part of a complex number:

```
imag(<complex_expr>)
```

<complex_expr> is an expression that evaluates to a complex number. For example, if `frame(200)` evaluates to the complex number `8.23 + 4.56i`, this expression returns a value of `4.5600`:

```
imag(frame(200))
```

See Also

complex | i | imag | real

More About

- "Complex Data in Stateflow Charts" on page 23-2
- "Rules for Using Complex Data in C Charts" on page 23-8
- "Best Practices for Using Complex Data in C Charts" on page 23-11

Rules for Using Complex Data in C Charts

Complex data is data whose value is a complex number. For example, in a Stateflow chart in Simulink model, an input signal with the value `3 + 5i` is complex. See "Complex Data in Stateflow Charts" on page 23-2.

These rules apply when you use complex data in Stateflow charts that use C as the action language.

Do not use complex number notation in actions

C charts do not support complex number notation `(a + bi)`, where `a` and `b` are real numbers. Therefore, you cannot use complex number notation in state actions, transition conditions and actions, or any statements in C charts.

To define a complex number, use the `complex` operator as described in "Notation for Complex Data" on page 23-4.

Do not perform math function operations on complex data in C charts

Math operations such as `sin`, `cos`, `min`, `max`, and `abs` do not work with complex data in C charts. However, you can use MATLAB functions for these operations.

For more information, see "Perform Math Function Operations with a MATLAB Function" on page 23-11.

Mix complex and real operands only for addition, subtraction, and multiplication

If you mix operands for any other math operations in C charts, an error appears when you try to simulate your model.

To mix complex and real operands for division, you can use a MATLAB function as described in "Perform Complex Division with a MATLAB Function" on page 23-12.

Tip Another way to mix operands for division is to use the `complex`, `real`, and `imag` operators in C charts.

Suppose that you want to calculate `y = x1/x2`, where `x1` is complex and `x2` is real. You can rewrite this calculation as:

```
y = complex(real(x1)/x2, imag(x1)/x2)
```

For more information, see "Access Real and Imaginary Parts of a Complex Number" on page 23-6.

Do not define complex data with constant scope

If you define complex data with `Constant` scope, an error appears when you try to simulate your model.

Do not define complex data with `ml`, `struct`, or `boolean` base type

If you define complex data with `ml`, `struct`, or `boolean` base type, an error appears when you try to simulate your model.

Use only real values to set initial values of complex data

When you define the initial value for data that is complex, use only a real value. See "Additional Properties" on page 9-19 for instructions on setting an initial value in the Data properties dialog box.

Do not enter minimum or maximum values for complex data

In the Data properties dialog box, do not enter any values in the **Minimum** or **Maximum** field when you define complex data. If you enter a value in either field, an error message appears when you try to simulate your model.

Assign complex values only to data of complex type

If you assign complex values to real data types, an error appears when you try to simulate your model.

Note You can assign both real and complex values to complex data types.

Do not pass real values to function inputs of complex type

This restriction applies to the following types of chart functions:

- Graphical functions
- Truth table functions
- MATLAB functions

23-9

- Simulink functions

If your C chart passes real values to function inputs of complex type, an error appears when you try to simulate your model.

Do not use complex data with temporal logic operators

You cannot use complex data as an argument for temporal logic operators, because you cannot define time as a complex number.

See Also

More About

- "Complex Data in Stateflow Charts" on page 23-2
- "Supported Operations for Complex Data" on page 23-4
- "Best Practices for Using Complex Data in C Charts" on page 23-11

Best Practices for Using Complex Data in C Charts

Complex data is data whose value is a complex number. For example, in a Stateflow chart in Simulink model, an input signal with the value `3 + 5i` is complex. See "Complex Data in Stateflow Charts" on page 23-2.

When you use complex data in Stateflow charts that use C as the action language, follow these best practices.

Perform Math Function Operations with a MATLAB Function

Math functions such as `sin`, `cos`, `min`, `max`, and `abs` do not work with complex data in C charts. However, you can use a MATLAB function in your chart to perform math function operations on complex data.

A Simple Example

In the following chart, a MATLAB function calculates the absolute value of a complex number:
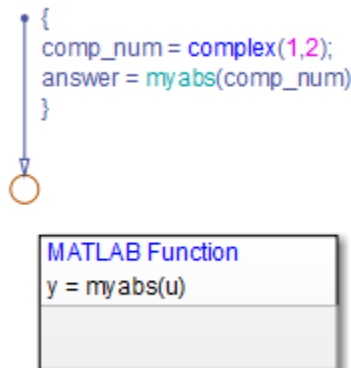
The value of `comp_num` is `1+2i`. Calculating the absolute value gives an answer of `2.2361`.

How to Calculate Absolute Value

Suppose that you want to find the absolute value of a complex number. Follow these steps:

1   Add a MATLAB function to your chart with this signature:

```
y = myabs(u)
```

2   Double-click the function box to open the editor.

3   In the editor, enter the code below:

```
function y = myabs(u)
%#codegen
y = abs(u);
```

The function `myabs` takes a complex input `u` and returns the absolute value as an output `y`.

4   Configure the input argument `u` to accept complex values.

    a   Open the Model Explorer.

    b   In the **Model Hierarchy** pane of the Model Explorer, navigate to the MATLAB function `myabs`.

    c   In the **Contents** pane of the Model Explorer, right-click the input argument `u` and select **Properties** from the context menu.

    d   In the Data properties dialog box, select `On` in the **Complexity** field and click **OK**.

You cannot pass real values to function inputs of complex type. For details, see "Rules for Using Complex Data in C Charts" on page 23-8.

Perform Complex Division with a MATLAB Function

Division with complex operands is not available as a binary or assignment operation in C charts. However, you can use a MATLAB function in your chart to perform division on complex data.

A Simple Example

In the following chart, a MATLAB function performs division on two complex operands:

The values of `comp_num` and `comp_den` are `1+2i` and `3+4i`, respectively. Dividing these values gives an answer of `0.44+0.08i`.

How to Perform Complex Division

To divide two complex numbers:

1   Add a MATLAB function to your chart with this function signature:

```
y = mydiv(u1, u2)
```

2   Double-click the function box to open the editor.

3   In the editor, enter the code below:

```
function y = mydiv(u1, u2)
%#codegen
y = u1 / u2;
```

The function `mydiv` takes two complex inputs, `u1` and `u2`, and returns the complex quotient of the two numbers as an output `y`.

4   Configure the input and output arguments to accept complex values.

    a   Open the Model Explorer.

    b   In the **Model Hierarchy** pane of the Model Explorer, navigate to the MATLAB function `mydiv`.

    c   For each input and output argument, follow these steps:

        i   In the **Contents** pane of the Model Explorer, right-click the argument and select **Properties** from the context menu.

>   **ii**   In the Data properties dialog box, select On in the **Complexity** field and click
>   **OK**.

You cannot pass real values to function inputs of complex type. For details, see "Rules for
Using Complex Data in C Charts" on page 23-8.

See Also

More About
*   "Complex Data in Stateflow Charts" on page 23-2
*   "Supported Operations for Complex Data" on page 23-4
*   "Rules for Using Complex Data in C Charts" on page 23-8

Detect Valid Transmission Data Using Frame Synchronization

Complex data is data whose value is a complex number. For example, in a Stateflow chart in Simulink model, an input signal with the value 3 + 5i is complex. See "Complex Data in Stateflow Charts" on page 23-2.

This model shows how to process complex data in transmission signals of a communication system.

What Is Frame Synchronization?

In communication systems, frame synchronization is a method of finding valid data in a transmission that consists of *data frames*. To aid frame synchronization, the transmitter inserts a fixed data pattern at the start of each data frame to mark the start of valid data. The receiver searches for the fixed pattern in each data frame and achieves frame synchronization when the correlation between the input data and the fixed pattern is high.

Model Structure

The model contains the following components.
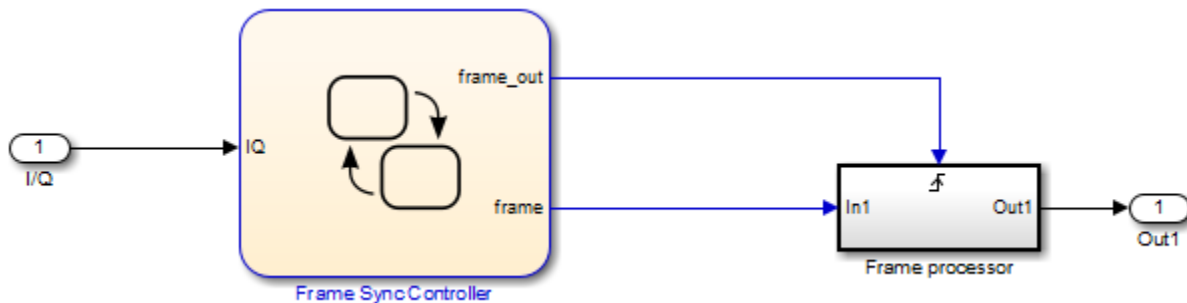
The C chart contains the following states, transitions, and MATLAB functions.
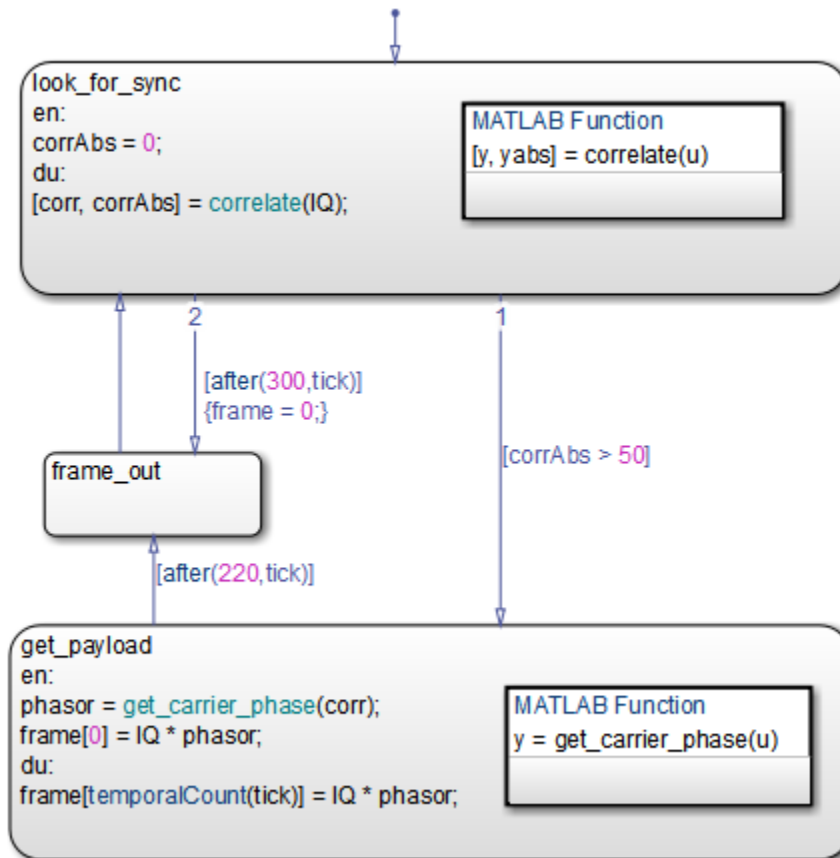
Key characteristics of the C chart include:

- Complex input and output signals

  The chart accepts a complex input signal I/Q. After synchronizing the data frame, the chart stores the valid data in a complex output signal frame.

- Complex multiplication

  The output signal frame is a vector of complex products between each valid data point and the phase angle of the carrier wave.

- Indexing into a complex vector

The chart uses the `temporalCount` operator to index into the complex vector `frame`.

- MATLAB functions with complex arguments

  The MATLAB functions `correlate` and `get_carrier_phase` have complex input and output arguments.

Simulation Results

The `sf_frame_sync_controller` model does not produce simulation results. The purpose of this example is to explain how to process complex data in a chart.

How the C Chart Works

The chart calculates the correlation between the input signal `I/Q` and the fixed data pattern `trainSig`. You define `trainSig` by writing and running a MATLAB script before you simulate the model.

- If the correlation exceeds 50 percent, frame synchronization occurs. The chart stores 220 valid data points in the complex vector `frame`.
- If the correlation stays below 50 percent after the chart has evaluated 300 data points, the frame synchronization algorithm resets.

Stage	Summary	Details
1	Activation of the frame synchronization algorithm	When the chart wakes up, the state `look_for_sync` activates to start the frame synchronization algorithm.
2	Calculation of correlation between the input signal and the fixed pattern	The MATLAB function `correlate` finds the correlation between the input signal `I/Q` and the fixed data pattern `trainSig`. Then, the function stores the complex correlation as `corr`.
3	Calculation of absolute value of the complex correlation	The MATLAB function `correlate` also finds the absolute value of `corr` and stores the output as `corrAbs`. The value of `corrAbs` is the correlation percentage, which can range from 0 to 100 percent. At 0 percent, there is no correlation; at 100 percent, there is perfect correlation.

Stage	Summary	Details
4	Identification of valid data in a frame	If `corrAbs` exceeds 50 percent, the correlation is high and the chart has identified the start of valid data in a data frame. The transition from the state `look_for_sync` to `get_payload` occurs. If `corrAbs` stays below 50 percent after the chart has evaluated 300 data points, the frame synchronization algorithm restarts.
5	Storage of valid data in a complex vector	When the correlation is high, the state `get_payload` activates. The MATLAB function `get_carrier_phase` finds the phase angle of the carrier wave and stores the value as `phasor`. Then, the state multiplies the input signal `I/Q` with the phase angle `phasor` and stores each complex product in successive elements of the vector `frame`.
6	Output of valid frame data	After collecting 220 data points, the chart outputs the vector `frame` to the next block in the model.
7	Restart of the frame synchronization algorithm	The state `look_for_sync` reactivates, and the frame synchronization algorithm restarts for the next data frame.

See Also

More About

- "Complex Data in Stateflow Charts" on page 23-2
- "Supported Operations for Complex Data" on page 23-4
- "Rules for Using Complex Data in C Charts" on page 23-8
- "Best Practices for Using Complex Data in C Charts" on page 23-11

Measure Frequency Response Using a Spectrum Analyzer

Complex data is data whose value is a complex number. For example, in a Stateflow chart in Simulink model, an input signal with the value 3 + 5i is complex. See "Complex Data in Stateflow Charts" on page 23-2.

This model shows measurement of the frequency response of a second-order system driven by a complex sinusoidal signal. A scope displays the measured frequency response as discrete Bode plots.

What Is a Spectrum Analyzer?

A spectrum analyzer is a tool that measures the frequency response (magnitude and phase angle) of a physical system over a range of frequencies.

Model Structure

The model sf_spectrum_analyzer contains the following components.
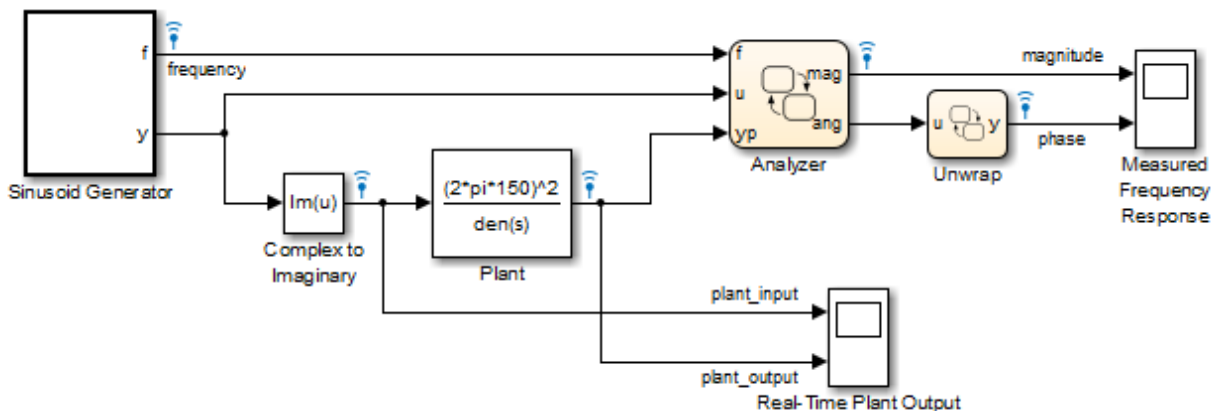
Model Component	Description
Sinusoid Generator block	Generates a complex sinusoidal signal of increasing frequency and supplies this signal to other blocks.

Model Component	Description
Complex to Imaginary block	Extracts the imaginary part of the complex signal from the Sinusoid Generator block so that a sine wave of increasing frequency can drive the Plant block.
Plant block	Uses a transfer function to describe a second-order system with a natural frequency of 150 Hz (300π radians per second) and a damping ratio of 0.3. Since the ratio is less than 1, this system is underdamped and contains two complex conjugate poles in the denominator of the transfer function. **Note** Typical applications implement the Plant block using a D/A (digital-to-analog) converter on the input signal and an A/D (analog-to-digital) converter on the output signal.
Analyzer chart	Calculates the frequency response of the second-order system defined by the Plant block.
Unwrap chart	Processes the phase angle output of the Analyzer chart.

Simulation Results

Simulation of the `sf_spectrum_analyzer` model produces discrete Bode plots in the Measured Frequency Response scope.
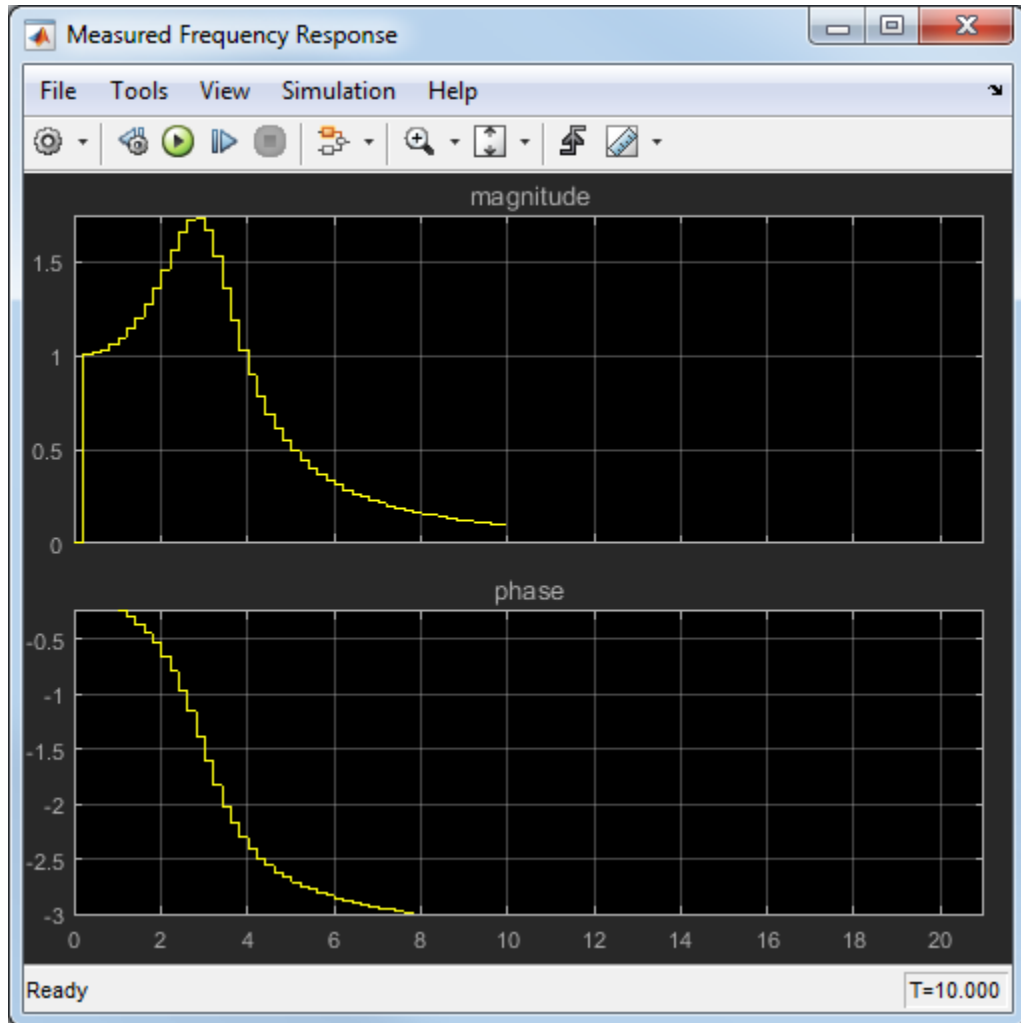
To adjust the scope display, right-click inside the grid and select **Autoscale** from the context menu.

- In the magnitude plot, the sharp peak is the response of the Plant block to a resonant frequency.
- In the phase plot, the angle changes from 0 to –π radians (–180 degrees). Each complex pole in the Plant block adds –π/2 radians to the phase angle.

23-21

How the Sinusoid Generator Block Works

This block is a masked chart that uses MATLAB as the action language. To access the chart, right-click the Sinusoid Generator block and select **Mask > Look Under Mask**.

Key characteristics of the signal generator chart include:

- Absolute-time temporal logic for controlling changes in frequency
- MATLAB code in the chart that generates a complex signal
- Transition condition that contains complex operands

Stage	Summary	Details
1	Signal frequency specification	When the chart awakens, the default transition sets the signal frequency f to `fstart` and activates state A. **Note** To set `fstart`, double-click the Sinusoid Generator block and enter a value (in Hz) in the **Initial frequency** field.
2	Complex signal generation	While state A is active, the chart generates the complex signal y based on frequency f and simulation time `t`.
3	Frequency and complex signal updates	If `delay` seconds have elapsed since activation of state A, the frequency f increases by an amount `fstep` and state A generates a new signal. Updates occur until the frequency f reaches the value `fstop`. **Note** To set `delay`, double-click the Sinusoid Generator block and enter a value (in seconds) in the **Delay at each frequency** field. To set `fstep`, enter a value (in Hz) in the **Step frequency** field.
4	Complex signal termination	When the frequency f reaches the value `fstop`, the state `Stopped` becomes active. The complex signal terminates and the simulation ends. **Note** To set `fstop`, double-click the Sinusoid Generator block and enter a value (in Hz) in the **Stop frequency** field.

How the Analyzer Chart Works

Key characteristics of the Analyzer chart include:

- Change detection of input frequency
- MATLAB code inside that chart that processes complex data

- State `during` action that contains complex operands

Stage	Summary	Details
1	State A activation	When the chart wakes up, the values of y and yn initialize to zero. • The data y stores the second-order system response to a signal from the Sinusoid Generator block. • The data yn stores an input signal of a given frequency.
2	Change detection of input frequency	The `hasChanged` operator detects if the input frequency f has changed since the previous time step. If so, MATLAB code calculates the magnitude and phase angle for the new frequency.

How the Unwrap Chart Works

This chart unwraps the phase angle output of the Analyzer chart. Unwrapping means preventing the phase angle from jumping more than π radians or dropping more than –π radians.
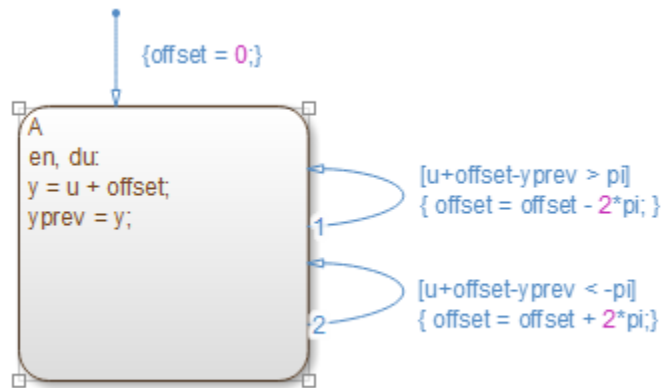
- If the phase angle jumps more than π radians, the chart subtracts 2π radians from the angle.
- If the phase angle drops more than –π radians, the chart adds 2π radians to the angle.

See Also

More About
- "Complex Data in Stateflow Charts" on page 23-2
- "Supported Operations for Complex Data" on page 23-4
- "Rules for Using Complex Data in C Charts" on page 23-8
- "Best Practices for Using Complex Data in C Charts" on page 23-11

24

Define Interfaces to Simulink Models and the MATLAB Workspace

Overview of Stateflow Block Interfaces

Stateflow Block Interfaces

Each Stateflow block interfaces to its Simulink model. Each Stateflow block can interface to sources external to the Simulink model (data, events, custom code). Events and data are the Stateflow objects that define the interface from the point of view of the Stateflow block.

Events can be local to the Stateflow block or can be propagated to and from the Simulink model and sources external to it. Data can be local to the Stateflow block or can be shared with and passed to the Simulink model and to sources external to the Simulink model. Messages can be local to the Stateflow block or can be passed through the Simulink to other Stateflow blocks.

The Stateflow interfaces include:

- Physical connections between Simulink blocks and the Stateflow block
- Event and data information exchanged between the Stateflow block and external sources
- Messages exchanged between Stateflow blocks.
- The properties of Stateflow charts
- Graphical functions exported from a chart

  See "Export Stateflow Functions for Reuse" on page 8-22 for more details.
- The MATLAB workspace

  See "Access MATLAB Functions and Workspace Data in C Charts" on page 12-32 for more details.
- Definitions in external code sources

Specify Properties for Stateflow Charts

Chart properties enable you to specify how your Stateflow chart interfaces with a Simulink model. You can specify chart properties in the Property Inspector, the Model Explorer, or the Chart properties dialog box.

- Property Inspector

  1 Open the Property Inspector by selecting **View > Property Inspector**.

  2 Click in the chart.

  3 In the Property Inspector, edit the chart properties.

- Model Explorer

  1 Open the Model Explorer by selecting **View > Model Explorer**.

  2 In the **Model Hierarchy** pane, select the chart.

  3 In the **Chart** pane, edit the chart properties.

- Chart properties dialog box

  1 Right-click in the chart.

  2 Select **Properties**.

  3 Edit the chart properties.

Stateflow Chart Properties

You can set the following chart properties in:

- The main and **Advanced** sections of the Property Inspector.
- The **General** tab of the Model Explorer or the Chart properties dialog box.

Name

Name of the chart (read-only). When you click the chart name hyperlink, the chart opens in the Stateflow editor.

Machine

Name of the Simulink subsystem (read-only). When you click the machine name hyperlink, the Machine properties dialog box opens. This property is not available in the Property Inspector.

Action Language

Action language that defines the syntax for state and transition actions in the chart. Options include:

- MATLAB
- C

The default value is MATLAB. For more information, see "Differences Between MATLAB and C as Action Language Syntax" on page 13-6.

State Machine Type

Type of state machine semantics to implement. Options include:

- Classic
- Mealy
- Moore

Classic charts provide the full set of Stateflow semantics. Mealy and Moore charts use a subset of these semantics. The default value is Classic. For more information, see "Overview of Mealy and Moore Machines" on page 7-2.

Update Method

Method by which a simulation updates or wakes up a chart in a Simulink model.

Setting	Description
Inherited	Input from the Simulink model determines when the chart wakes up during a simulation (default). If you define input events for the chart, the Stateflow chart is explicitly triggered by a signal on its trigger port originating from a connected Simulink block. You can set this trigger input event to occur in response to a Simulink signal. The Simulink signal can be `Rising`, `Falling`, or `Either` (rising and falling), or in response to a `Function Call`. For more information, see "Activate a Stateflow Chart by Sending Input Events" on page 10-9. If you do not define input events, the Stateflow chart implicitly inherits triggers from the Stateflow model. These implicit events are the discrete or continuous sample times of the Stateflow signals providing inputs to the chart. If you define data inputs, the chart awakens at the rate of the fastest data input. If you do not define any data input for the chart, the chart wakes up as defined by the execution behavior of its parent subsystem.
Discrete	The Simulink model generates an implicit event at regular time intervals to awaken the Stateflow chart at the rate that you specify in the **Sample Time** chart property. Other blocks in the Simulink model can have different sample times.
Continuous	The Stateflow chart updates its state during major time steps only, although it computes outputs and local continuous variables during major and minor time steps. The chart can register zero crossings, which allows Simulink models to sample Stateflow charts whenever state changes occur. The Stateflow chart computes derivatives for local continuous variables. For more information, see "Continuous-Time Modeling in Stateflow" on page 21-2.

Sample Time

The time interval at which the Stateflow chart wakes up during simulation. The sample time can be any nonzero number. The sample time is in the same units as the Simulink simulation time. Other blocks in the Simulink model can have different sample times. This option is available only when you set the chart property **Update method** to `Discrete`.

Enable Zero-Crossing Detection

Specifies that zero-crossing detection is enabled (default). This option is available only when you set the chart property **Update method** to `Continuous`. See "Disable Zero-Crossing Detection" on page 21-4.

Enable C-Bit Operations

Specifies that the operators &, ^, |, and ~ perform bitwise operations in action statements (default). If you clear this check box:

- & , |, and ~ perform logical operations.
- ^ performs the power operation.

This option is available only in charts that use C as the action language. For more information, see "Supported Operations for Chart Data" on page 12-14.

User Specified State/Transition Execution Order

Specifies that the chart uses explicit ordering of parallel states and transitions (default). You determine the order in which the chart executes parallel states and tests transitions originating from a source. This option is available only in charts that use C as the action language. For more information, see "Execution Order for Parallel States" on page 3-86 and "Evaluate Transitions" on page 3-63.

Export Chart Level Functions

Extends the scope of functions defined at the root level of the chart to other parts of the model. This option enables Simulink Caller blocks to call Stateflow functions in the local hierarchy by using qualified notation *chartName*.*functionName*. For more information, see "Export Stateflow Functions for Reuse" on page 8-22.

Treat Exported Functions as Globally Visible

Enables Stateflow and Simulink Caller blocks throughout the model to call functions exported from Stateflow without using qualified notation. This option is available only when you select the chart property **Export Chart Level Functions**. For more information, see "Export Stateflow Functions for Reuse" on page 8-22.

Use Strong Data Typing with Simulink I/O

Enables charts to interface directly with signals from Simulink models (default). The chart accepts only input signals whose data type matches the type of the corresponding

Stateflow data object. Otherwise, a type mismatch error occurs. This option is available only in charts that use C as the action language. For more information, see "Strong Data Typing with Simulink Inputs and Outputs" on page 9-39.

Note The **Use Strong Data Typing with Simulink I/O** chart property is provided for backward compatibility. Clearing this check box can produce unpredictable results and is not recommended.

Execute (Enter) Chart at Initialization

Specifies that the chart initializes its state configuration at time 0 instead of at the first occurrence of an input event. For more information, see "Execution of a Chart at Initialization" on page 3-42.

Initialize Outputs Every Time Chart Wakes Up

Specifies that the chart resets its output values every time that the chart wakes up, not only at time 0. Output values are reset whenever a chart is triggered by function call, edge trigger, or clock tick. If you set an initial value for an output data object, the output resets to that value. Otherwise, the output resets to zero. Select this option to:

- Ensure that all outputs are defined in every chart execution.
- Prevent latching of outputs (carrying over values of outputs computed in previous executions).
- Provide all chart outputs with a meaningful initial value.

For more information, see "Initial Value" on page 9-9.

Enable Super Step Semantics

Specifies that the chart can take multiple transitions in each time step until it reaches a stable state. This option is not available when you set the chart property **Update method** to `Continuous`. For more information, see "Super Step Semantics" on page 3-73.

Maximum Iterations in Each Super Step

Specifies the maximum number of transitions that the chart can take in each time step. The chart always takes one transition during a super step, so the value N that you specify represents the maximum number of *additional* transitions (for a total of $N+1$). This option

is available only when you select the chart property **Enable Super Step Semantics**. For more information, see "Maximum Number of Iterations" on page 3-73.

Behavior After Too Many Iterations

Specifies how the chart behaves after it reaches the maximum number of transitions in a time step.

Setting	Description
Proceed	Chart execution continues to the next time step.
Throw Error	Simulation stops and an error message appears. This setting is valid only for simulation. In generated code, chart execution always proceeds.

This option is available only when you select the chart property **Enable Super Step Semantics**.

Support Variable-Size Arrays

Specifies that chart supports input and output data that varies in dimension during simulation. See "Declare Variable-Size Inputs and Outputs" on page 18-2.

Saturate on Integer Overflow

Specifies that integer overflows saturate in the generated code. See "Handle Integer Overflow for Chart Data" on page 9-45.

States When Enabling

Specifies how states behave when function-call input events reenable the chart. Options include:

- Held
- Reset

See "Control States in Charts Enabled by Function-Call Input Events" on page 10-14.

Create Output for Monitoring

Specifies that the chart produces active state output. When you enable this option, you can select one of these activity types to output:

- `Child activity`
- `Leaf state activity`

See "Monitor State Activity Through Active State Data" on page 24-26.

Fixed-Point Properties

You can set fixed-point properties for the chart in:

- The **Fixed Point Properties** section of the Property Inspector.
- The **Fixed Point Properties** tab of the Model Explorer or the Chart properties dialog box.

Fixed-point properties are available only in charts that use MATLAB as the action language.

Treat These Inherited Simulink Signal Types as fi Objects

Specifies whether the chart treats inherited fixed-point and integer signals as Fixed-Point Designer™ `fi` objects.

Setting	Description
`Fixed-point`	The chart treats all fixed-point inputs as `fi` objects (default).
`Fixed-point & Integer`	The chart treats all fixed-point and integer inputs as `fi` objects.

MATLAB Chart fimath

Specifies default properties for the chart.

Setting	Description
`Same as MATLAB`	Use the same `fimath` properties as the current default `fimath` object in MATLAB.
`Specify Other`	Use your own default `fimath` object. You can: • Construct a `fimath` object inside the edit box. • Create a `fimath` object in the MATLAB or model workspace and enter its variable name in the edit box.

For more information, see "Fixed-Point Basics in MATLAB" (Fixed-Point Designer).

Additional Properties

You can set additional properties for the chart in:

- The **Info** tab of the Property Inspector.
- The **Documentation** tab of the Model Explorer or the Chart properties dialog box.

Description

Description of the chart. You can enter a brief description and comments.

Document Link

Link to online documentation for the chart. You can enter a web URL address or a MATLAB command that displays documentation in a suitable online format, such as an HTML file or text in the MATLAB Command Window. When you click the **Document link** hyperlink, Stateflow evaluates the link and displays the documentation.

Machine Properties

The Stateflow machine represents all of the Stateflow blocks in a model (including all charts, state transition tables, and truth tables). You can specify machine properties in the Machine properties dialog box.

1 Open the Model Explorer or the Chart properties dialog box for any chart in the model.
2 In the **Machine** chart property field, click the machine name link.
3 In the Machine properties dialog box, edit the properties for the Stateflow machine.

Simulink Model

Name of the Simulink model that defines this Stateflow machine (read-only). You change the model name when you save the model.

Creation Date

Date on which this Stateflow machine was created (read-only).

Creator

Name of the person who created this Stateflow machine.

Modified

Time of the most recent modification of this Stateflow machine.

Version

Version number of this Stateflow machine.

Description

Description of the Stateflow machine. You can enter a brief description and comments.

Document Link

Link to online documentation for the Stateflow machine. You can enter a web URL address or a MATLAB command that displays documentation in a suitable online format, such as an HTML file or text in the MATLAB Command Window. When you click the **Document link** hyperlink, Stateflow evaluates the link and displays the documentation.

See Also

More About
- "Differences Between MATLAB and C as Action Language Syntax" on page 13-6
- "Overview of Mealy and Moore Machines" on page 7-2
- "Continuous-Time Modeling in Stateflow" on page 21-2
- "Supported Operations for Chart Data" on page 12-14

Implement Interfaces to Simulink Models

Define a Triggered Stateflow Block

Essential conditions that define an edge-triggered Stateflow block are:

- The chart property **Update method** is `Discrete` or `Inherited`. See "Update Method" on page 24-4.
- The chart has an **Input from Simulink** event defined and an edge-trigger type specified. See "Activate a Stateflow Chart by Sending Input Events" on page 10-9.

Triggered Stateflow Block Example

The following model shows an edge-triggered Stateflow block named Callee:
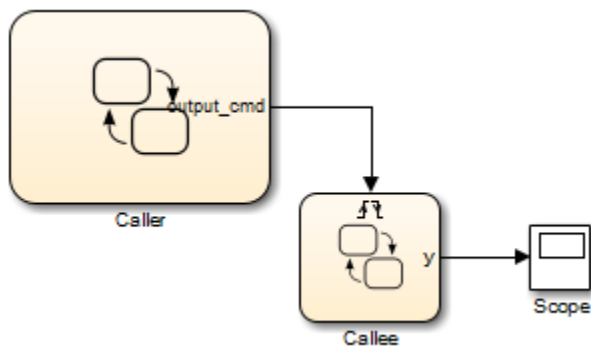
The **Input from Simulink** event has an **Either** edge-trigger type. If you define more than one **Input from Simulink** event, the Simulink model determines the sample times to be

consistent with various rates of all the incoming signals. The outputs of a triggered Stateflow block are held after the execution of the block.

Define a Sampled Stateflow Block

There are two ways you can define a sampled Stateflow block.

- Set the chart property **Update method** to `Discrete` and enter a **Sample Time** value. See "Update Method" on page 24-4.
- Add and define input data either in the Stateflow Editor by selecting **Chart > Add Inputs & Outputs > Data Input from Simulink** or in the Model Explorer. See "Share Input and Output Data with Simulink" on page 9-23.

Simulink determines the chart sample time to be consistent with the rate of the incoming data signal.

The **Sample Time** you set in the Chart properties dialog box takes precedence over the sample time of any input data.

Sampled Stateflow Block Example

You specify a discrete sample rate to have Simulink trigger a Stateflow block that does use an explicit trigger port. You can specify a sample time for the chart in the Chart properties dialog box. Simulink then calls the Stateflow block at a defined, regular sample time.
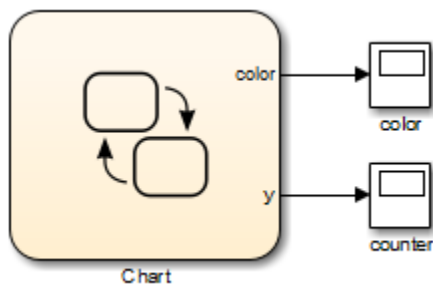
The outputs of a sampled Stateflow block are held after the execution of the block.

Define an Inherited Stateflow Block

Essential conditions that define an inherited trigger Stateflow block are:

- The chart property **Update method** is `Discrete` or `Inherited`. See "Update Method" on page 24-4.
- The chart has an **Input from Simulink** data object defined using the Stateflow Editor **Add** menu or the Model Explorer. See "Share Input and Output Data with Simulink" on page 9-23. Simulink determines the chart sample time to be consistent with the rate of the incoming data signal.

Inherited Stateflow Block Example

Simulink can trigger a Stateflow block that does not use an explicit trigger port or a specified discrete sample time. In this case, the Simulink calls the Stateflow block at a sample time determined by the model.

In this example, the chart contains two **Input from Simulink** data objects. Simulink determines the sample times to be consistent with the rates of both incoming signals.
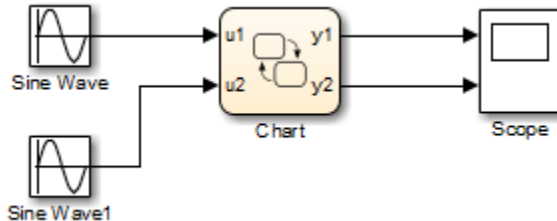
The outputs of an inherited trigger Stateflow block are held after the execution of the block.

Define a Continuous Stateflow Block

To define a continuous Stateflow block, set the chart **Update method** in the Chart properties dialog box to **Continuous**.

Define Function-Call Output Events

This topic shows you how to trigger a function-call subsystem in a Simulink model with a function-call output event in a Stateflow chart. The procedure assumes that you have a

programmed function-call subsystem and a Stateflow block in the model. Use the following steps to connect the Stateflow block to the function-call subsystem and trigger it during simulation.

1. In your chart, select **Chart > Add Inputs & Outputs > Event Output To Simulink**.

   The Event properties dialog box appears with a default name of `event` and a **Scope** of `Output to Simulink`.

2. Set **Trigger** to `Function Call`.

3. Name the event appropriately and click **OK** to close the dialog box.

   An output port with the name of the event you add appears on the right side of the Stateflow block.

4. Connect the output port on the Stateflow block for the function-call output event to the input trigger port of the subsystem.

   Avoid placing any other blocks in the connection lines between the Stateflow block and the function-call subsystem.

   Note You cannot connect a function-call output event from a chart to a Demux block to trigger multiple subsystems.

5. To execute the function-call subsystem, include an event broadcast of the function-call output event in the actions of the chart.

For examples of using function-call output events, see "Activate a Simulink Block by Using Function Calls" on page 10-25.

Define Edge-Triggered Output Events

Simulink controls the execution of edge-triggered subsystems with output events. Essential conditions that define this use of triggered output events are:

- The chart has an **Output to Simulink** event with the trigger type set to **Either**. See "Activate a Simulink Block by Sending Output Events" on page 10-21.

- The Simulink block connected to the edge-triggered **Output to Simulink** event has its own trigger type set to the equivalent edge trigger.

For examples of using edge-triggered output events, see "Activate a Simulink Block by Using Edge Triggers" on page 10-21.

Reuse Charts in Models with Chart Libraries

In Simulink, you can create your own block libraries as a way to reuse the functionality of blocks or subsystems in one or more models. Similarly, you can reuse a set of Stateflow algorithms by encapsulating the functionality in a chart library.

As with other Simulink block libraries, you can specialize each instance of chart library blocks in your model to use different data types, sample times, and other properties. Library instances that inherit the same properties can reuse generated code.

For more information about Simulink block libraries, see "Libraries" (Simulink).

Create Specialized Chart Libraries for Large-Scale Modeling

1. Add Stateflow charts with polymorphic logic to a Simulink model.

   Polymorphic logic is logic that can process data with different properties, such as type, size, and complexity.

2. Configure the charts to inherit the properties you want to specialize.

   For a list, see "Customize Properties of Library Blocks" on page 24-19.

3. Optionally, customize your charts using masking.

4. Simulate and debug your charts.

5. In Simulink, create a library model by selecting **File > New > Library**.

6. Copy or drag the charts into a library model.

For an example using MATLAB Function blocks, see "Create Custom Block Libraries" (Simulink).

Customize Properties of Library Blocks

You can customize instances of Stateflow library blocks by allowing them to inherit any of the following properties from Simulink.

Property	Inherits by Default?	How to Specify Inheritance
Type	Yes	Set the data type property to **Inherit: Same as Simulink**.
Size	Yes	Set the data size property to **-1**.
Complexity	Yes	Set the data complexity property to **Inherited**.
Limit range	No	Specify minimum and maximum values as Simulink parameters. For example, if minimum value = `aParam` and maximum value = `aParam` + 3, different instances of a Stateflow library block can resolve to different `aParam` parameters defined in their parent mask subsystems.
Initial value	Depends on scope	For local data, temporary data, and outputs, specify initial values as Simulink parameters. Other data always inherits the initial value: • Parameters inherit the initial value from the associated parameter in the parent mask subsystem. • Inputs inherit the initial value from the Simulink input signal. • Data store memory inherits the initial value from the Simulink data store to which it is bound.
Sampling mode (input)	Yes	Stateflow chart input ports always inherit sampling mode.
Data type override mode for fixed-point data	Yes	Different library instances inherit different data type override modes from their ancestors in the model hierarchy.
Sample time (block)	Yes	Set the block sample time property to **-1**.

Limitations of Library Charts

Events parented by a library Stateflow machine are invalid. The parser flags such events as errors.

MATLAB Workspace Interfaces

In this section...
"About the MATLAB Workspace" on page 24-21
"Examine the MATLAB Workspace" on page 24-21
"Interface the MATLAB Workspace with Charts" on page 24-21

About the MATLAB Workspace

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

Examine the MATLAB Workspace

Two commands, `who` and `whos`, show the current contents of the workspace. The `who` command gives a short list, while `whos` also gives size and storage information.

To delete all the existing variables from the workspace, enter `clear all` at the MATLAB command line. See the MATLAB documentation for more information.

Interface the MATLAB Workspace with Charts

A chart has the following access to the MATLAB workspace:

- You can access MATLAB data or MATLAB functions using the `ml` namespace operator or the `ml` function.

  See "Access MATLAB Functions and Workspace Data in C Charts" on page 12-32 for more information.

- You can use the MATLAB workspace to initialize chart data at the beginning of a simulation.

  See "Enter Expressions and Parameters for Data Properties" on page 9-20.

- You can save chart data to the workspace at the end of a simulation.

  See "Save Data to the MATLAB Base Workspace" on page 9-25 for more information.

Create a Mask to Share Parameters with Simulink

Creating masks for Stateflow charts, state transition tables, and truth tables simplifies how you use and share blocks in a Simulink model. The mask encapsulates the block by hiding the underlying logic and creates a user interface for the block. You can customize the block by:

- Changing the appearance with meaningful icons and ports.
- Creating a user interface for parameters.
- Adding customized documentation.

You decide which parameters to change through the mask user interface. You can provide meaningful descriptions of these parameters. For example, in the model `sf_car`, the `shift_logic` chart has a mask through which you can adjust the parameter `TWAIT`. To open the Mask Parameters dialog box, double-click the Stateflow chart. This dialog box contains a parameter description `"Delay before gear change (tick)"` and a box to edit the value. This value is tied to the parameter `TWAIT` inside the mask. When you edit the value in this box, Stateflow assigns the new value to `TWAIT` during simulation.
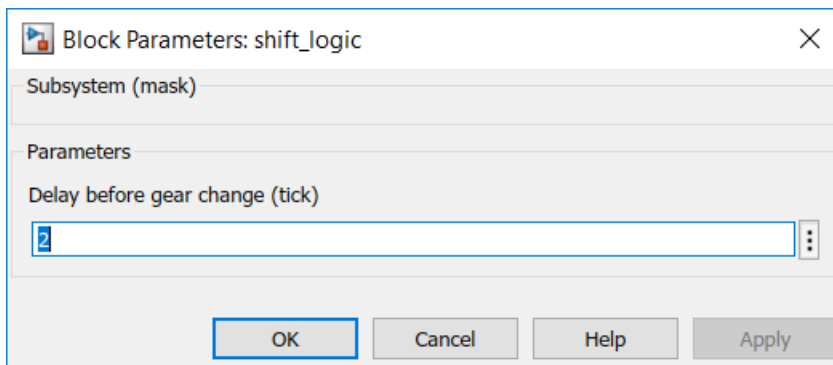
You can create other types of user interfaces for the mask parameters, such as check boxes, context menus, and option buttons.

You can create masks on Stateflow blocks accessible from the Simulink library: charts, state transition tables, and truth tables. You cannot mask atomic subcharts, states, or any other objects within a chart.

For more information, see "Create Block Masks" (Simulink).

Create a Mask for a Stateflow Chart

To create a mask for the Stateflow chart in the model `old_sf_car`:

1   Open the model `old_sf_car`.

2   In the Simulink Editor, select the chart `shift_logic`.

3   Open the Mask Editor by selecting **Diagram** > **Mask** > **Create Mask**.

Add an Icon to the Mask

To customize the appearance of the block icon, use drawing commands or load an image. For more information, see "Draw Mask Icon" (Simulink).

1   In the Mask Editor, select the **Icon & Ports** pane.

2   In the edit box under **Icon Drawing commands**, enter:

```
image('shift_logic.svg')
```

3   Click **Apply**.

Add Parameters to the Mask

When you create a mask for a Stateflow block, you can define a custom interface for the block parameters. You provide access to the block parameters by defining corresponding parameters with the same name in the Mask Editor. A user interface to these parameters is then provided through a Mask Parameters dialog box. The mask parameters appear as editable fields in the Mask Parameters dialog box. Stateflow applies these values to the corresponding block parameters during simulation.

For example, the chart `shift_logic` has a parameter `TWAIT`. To add `TWAIT` as a parameter to the mask:

1   In the Mask Editor, select the **Parameters & Dialog** pane.

2   Double-click the Edit parameter icon ☲.

3   Next to `edit`, under **Prompt**, enter the prompt for the new mask parameter in the Mask Parameters dialog box:

```
Delay before gear change(tick)
```

24-23

4 Under **Name**, enter the name of the parameter in the mask:

TWAIT

5 Click **Apply**.

6 Click **OK**.

View the New Mask

After creating a mask, the new icon for the `shift_logic` chart appears in the Simulink canvas. If you double-click the icon, the Mask Parameters dialog box opens. This dialog box has the prompt for the parameter TWAIT. The value in the edit box is assigned to the parameter TWAIT during simulation.
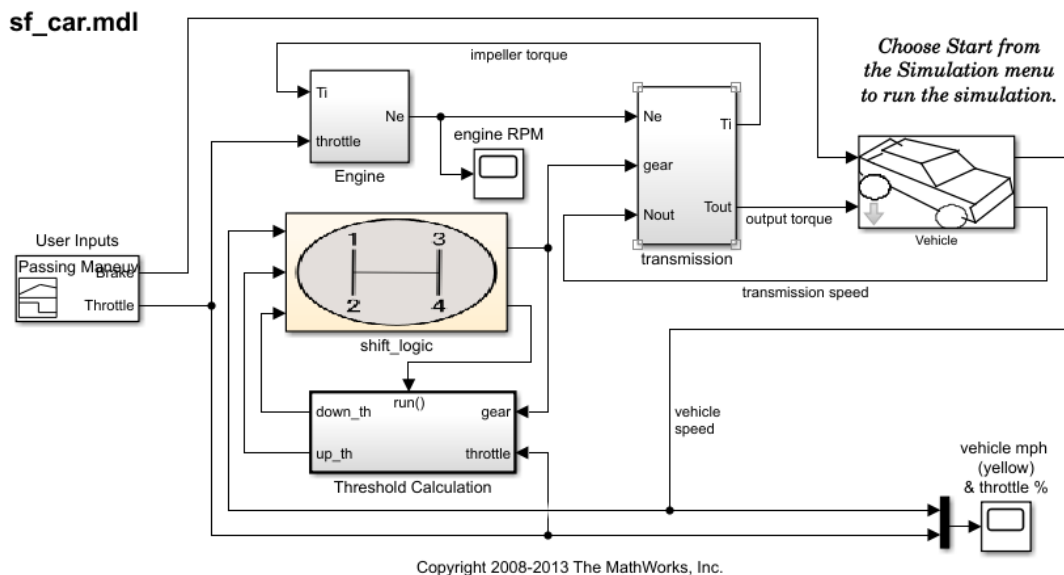
Look Under the Mask

You can view and edit the contents of a masked block by clicking the **Look inside mask** badge on the chart. The badge is a downward facing arrow in the lower-left corner of the chart. Alternatively, select **Diagram > Mask > Look Under Mask**. Looking under a mask does not unmask the block.

Edit the Mask

You can edit a mask by selecting **Diagram** > **Mask** > **Edit Mask**. In the Mask Editor, you can modify the mask icon, change the parameters, or add documentation. To remove the mask, click **Unmask** in the lower corner of the Mask Editor. After you change a mask, click **Apply** to save the changes.

See Also

More About

- "Share Parameters with Simulink and the MATLAB Workspace" on page 9-26
- "Masking Fundamentals" (Simulink)
- "Mask Editor Overview" (Simulink)
- "Draw Mask Icon" (Simulink)

Monitor State Activity Through Active State Data

Active state data can simplify the design of some Stateflow charts because you do not have to maintain data that is highly correlated to the chart hierarchy. When you enable active state data, Stateflow reports state activity through an output port to Simulink or as local data in your chart. Using active state data, you can:

- Avoid manual data updates reflecting chart activity.
- View chart activity by using a scope or the Simulation Data Inspector.
- Log chart activity for diagnostics.
- Drive other Simulink subsystems.

Types of Active State Data

When you enable active state data, Stateflow creates a Boolean or enumerated data object to match the activity type.

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

For self-activity of a chart or state, the data value is `true` when active and `false` when inactive. For child and leaf state activity, the data is an enumerated type. Stateflow can define the enumeration class or you can create the definition manually. For more information, see "Define State Activity Enumeration Type" on page 24-28.

You can enable active state data for a Stateflow chart, state, state transition table, or atomic subchart. This table lists the activity types supported by each kind of Stateflow object.

Stateflow Object	Self-Activity	Child Activity	Leaf State Activity
Charts	Not supported	Supported	Supported
States	Supported	Supported	Supported
Atomic subcharts	Supported at the container level	Supported inside the subchart	Supported inside the subchart

Stateflow Object	Self-Activity	Child Activity	Leaf State Activity
State transition tables	Not supported	Supported	Supported

Enable Active State Data

You can enable active state data in either the Property Inspector or the Model Explorer.

- Property Inspector

    1 Open the Property Inspector by selecting **View** > **Property Inspector**.

    2 In the Stateflow Editor canvas, select the Stateflow object to monitor.

    3 In the **Monitoring** section of the Property Inspector, select the **Create output for monitoring** check box and edit the active state data properties.

- Model Explorer

    1 Open the Model Explorer by selecting **View** > **Model Explorer**.

    2 In the **Model Hierarchy** pane, double-click the Stateflow object to monitor.

    3 In the Stateflow object pane, select the **Create output for monitoring** check box and edit the active state data properties.

Active State Data Properties

Activity Type

Type of state activity to monitor. Choose from these options:

- `Self activity`
- `Child activity`
- `Leaf state activity`

Data Name

Name of the active state data object. For more information, see "Rules for Naming Stateflow Objects" on page 2-4.

24-27

Enum Name

Name of the enumerated data type for the active state data object. This property applies only to child and leaf state activity.

Define Enumerated Type Manually

Specifies whether you define the enumerated data type manually. This property applies only to child and leaf state activity. For more information, see "Define State Activity Enumeration Type" on page 24-28.

Set Scope for Active State Data

By default, active state data has a scope of `Output`. Stateflow creates an output port on the chart block in the Simulink model.

To access active state data inside a Stateflow chart, change the scope to `Local` in the Symbols window or in the Model Explorer. For more information, see "Set Data Properties" on page 9-5.

You can specify information for code generation by binding the local active state data to a `Simulink.Signal` object. Modify the properties of the object through the `CoderInfo` property. For more information, see `Simulink.CoderInfo`.

Define State Activity Enumeration Type

By default, Stateflow defines the enumeration data type for child and leaf activity. If you select the **Define enumerated type manually** check box and no enumeration data type definition exists, then Stateflow provides a link to create a definition. Clicking the **Create enum definition from template** link generates a customizable definition.

The enumeration data type definition contains one literal for each state name plus an extra literal to indicate that no substate is active. For example, in the model `sf_car`, the state `gear_state` contains four child states that correspond to the gears in a car: `first`, `second`, `third`, `fourth`. The model specifies the child activity data type with this enumeration class definition:

```
classdef gearType < Simulink.IntEnumType
    enumeration
        None(0),
        first(1),
        second(2),
        third(3),
        fourth(4)
    end
...
end
```

For more information, see "Define Enumerated Data Types" on page 19-6.

The base storage type for automatically created enumerations defaults to `Native Integer`. For a smaller memory footprint, in the **Optimization** pane of the Configuration Parameters dialog box, change the value of the **Base storage type for automatically**

created enumerations field. For more information, see "Base storage type for automatically created enumerations" (Simulink Coder).

Leaf State Activity and Parallel States

When you enable leaf state activity, a state with parallel (AND) decomposition is treated as a leaf state. State activity of the parallel substates is not available because these substates are active simultaneously. For example, suppose that you enable leaf state activity for this chart. Because state B has parallel decomposition, its substates B1 and B2 are active simultaneously so B is treated as a leaf state of the chart.
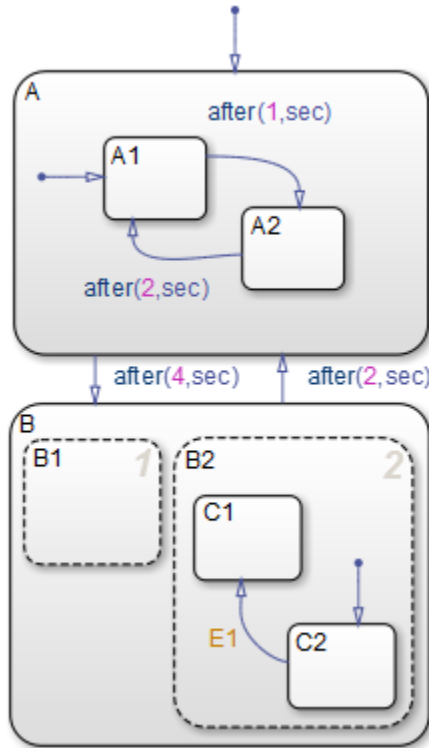
During simulation, a scope connected to the active state output data shows the enumerated values for the leaf states A1, A2, and B.
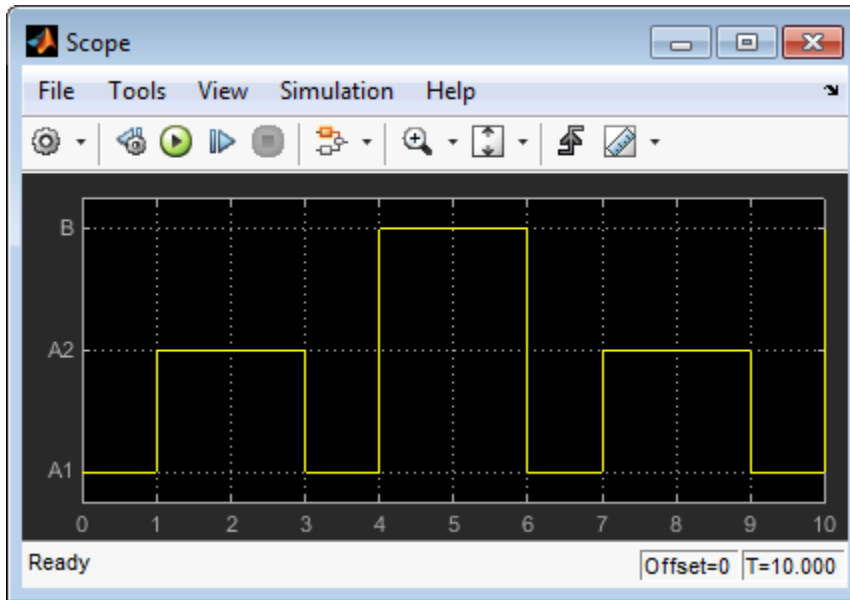
Limitations for Active State Data

- Enabling child activity output for states that have no children results in an error at compilation and run time.

- You cannot enable child or leaf state activity in charts or states with parallel decomposition. To check state activity in substates of parallel states, use the `in` operator. For more information, see "Check State Activity by Using the in Operator" on page 12-77.

- Do not select the chart property **Initialize Outputs Every Time Chart Wakes Up** on charts that use active state output data. With this setting, the behavior of the output data is unpredictable.

See Also

Simulink.Signal | Simulink.CoderInfo

More About

- "Simplify Stateflow Charts by Incorporating Active State Output" on page 24-37
- "View State Activity by Using the Simulation Data Inspector" on page 24-33
- "Check State Activity by Using the in Operator" on page 12-77
- "Rules for Naming Stateflow Objects" on page 2-4
- "Define State Activity Enumeration Type" on page 24-28
- "Base storage type for automatically created enumerations" (Simulink Coder)

View State Activity by Using the Simulation Data Inspector

You can use the Simulation Data Inspector to log state activity and data for your Stateflow chart in a Simulink model. With the Simulation Data Inspector, you can view and compare:

- Data from your chart
- Leaf chart activity
- Child chart activity
- Child state activity
- Self state activity
- Leaf state activity

Log to the Simulation Data Inspector from Stateflow

In this exercise, you use the Simulation Data Inspector to monitor the active state outputs for the Stateflow chart in the model `sf_car`.

To log a signal with the Simulation Data Inspector, highlight the signal line that you want to log. Right-click the `gear` signal and choose **Log Selected Signals** from the context menu. A logging badge appears next to the `gear` signal, indicating that the data from that signal is logged when the model is run.
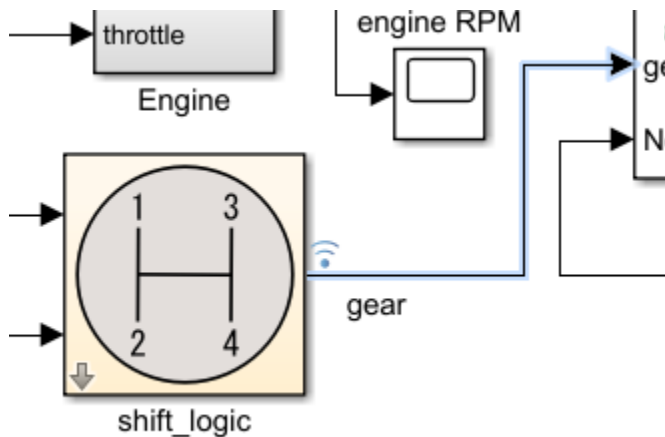
The logging badge ⋯ marks logged signals in the model.

To log the active state data from `gear_state`:

1 Open the Stateflow chart.

2 To select `gear_state`, click `gear_state`.

3 On the Simulink Editor toolbar, click the **Simulation Data Inspector** drop-down and select **Logging for state gear_state > Child state activity**
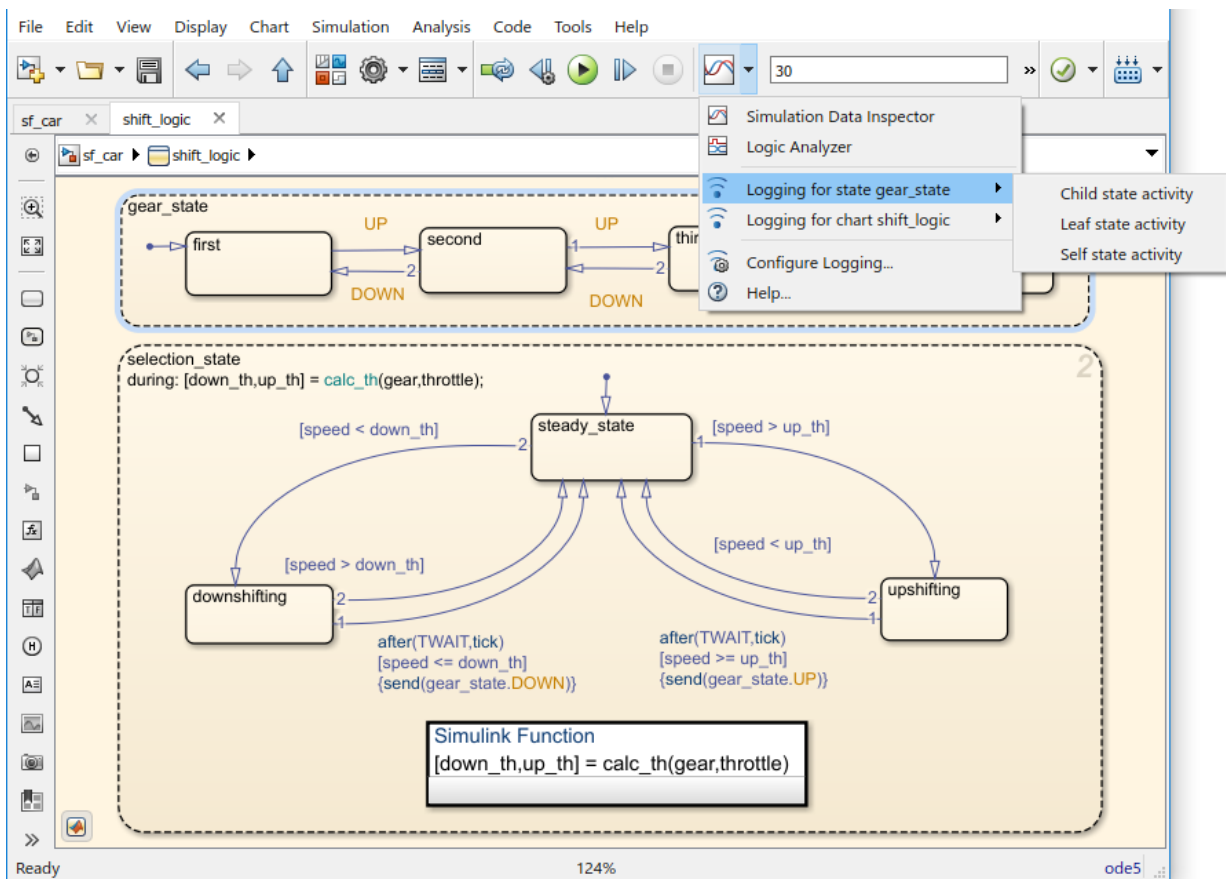
To select multiple signals to log at once:

1 In your Stateflow chart, select the states whose activity you want to log.

2 On the Simulink Editor toolbar, click the **Simulation Data Inspector** drop-down and select **Logging for selected states > Self state activity**.
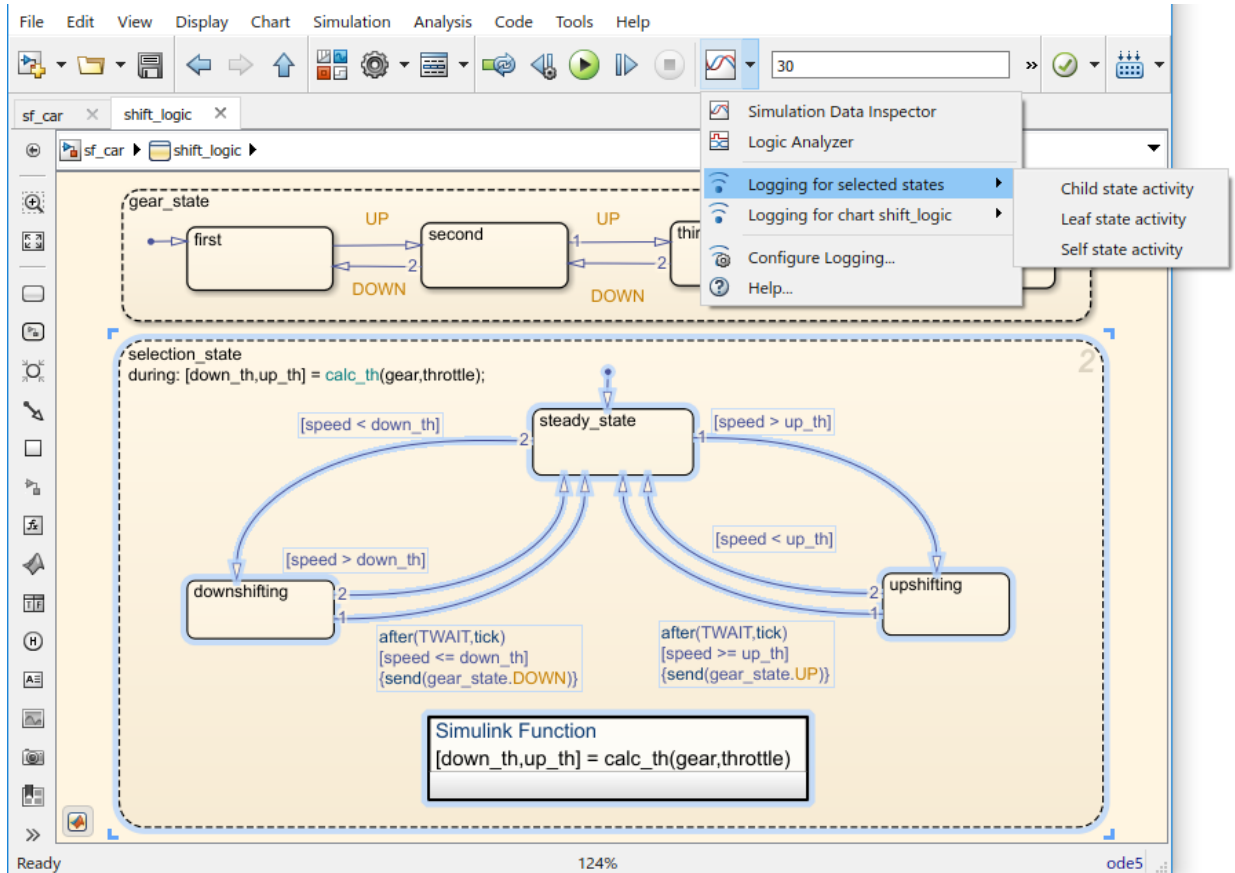
3 The logging badge appears on all highlighted states.

After running one or more simulations with signals marked for logging, click **Simulation Data Inspector** button [icon] on the Simulink editor toolbar and view your data. Multiple runs show up in the **Inspect** pane and can be viewed together. To choose which signals you want to plot, use the check boxes next to the signal names.
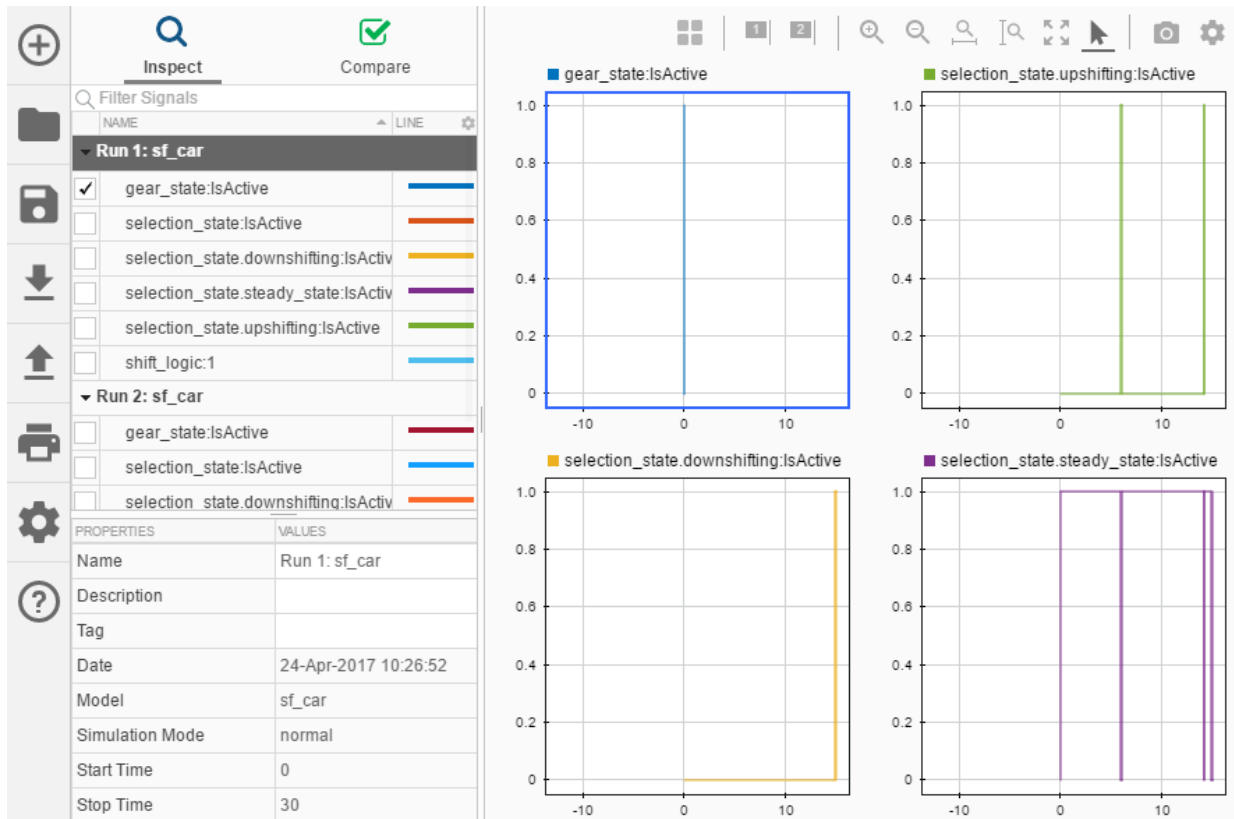
See Also

More About

- "Monitor State Activity Through Active State Data" on page 24-26
- "Simplify Stateflow Charts by Incorporating Active State Output" on page 24-37
- Simulation Data Inspector
- "Inspect Simulation Data" (Simulink)
- "Compare Simulation Data" (Simulink)

Simplify Stateflow Charts by Incorporating Active State Output

Active state data can simplify the design of some Stateflow charts because you do not have to maintain data that is highly correlated to the chart hierarchy. When you enable active state data, Stateflow reports state activity through an output port to Simulink or as local data in your chart. This example shows how to simplify the design of a Stateflow chart by adding active state output data. For more information, see "Monitor State Activity Through Active State Data" on page 24-26.

In the legacy model `old_sf_car`, the Stateflow chart shift_logic tracks child state activity in `gear_state` by updating the value of the output data `gear`.
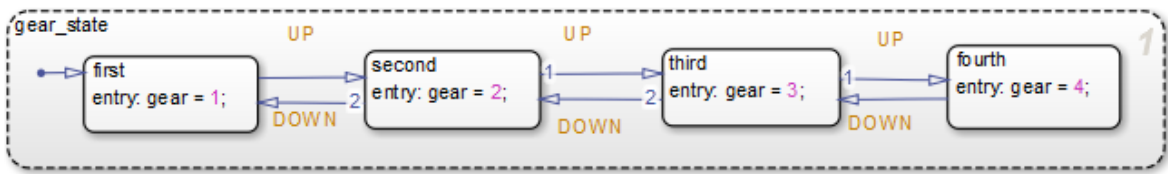
By incorporating active state data, the model `sf_car` avoids manual data updates reflecting chart activity. Instead, the chart outputs child state activity automatically through the active state output `gear`.
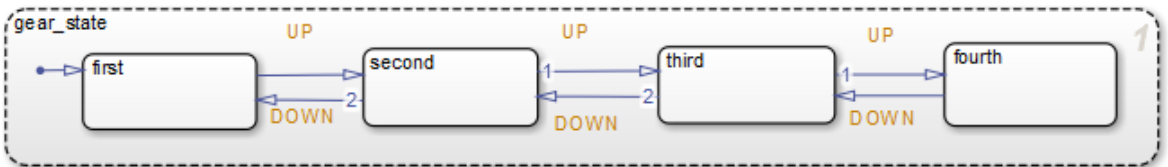
Modify the Model

To simplify the design of the `old_sf_car` model, eliminate data that is highly correlated to the chart hierarchy and enable automatic monitoring of child state activity in `gear_state`.

Eliminate Manual Tracking of State Activity

1   Open the model `old_sf_car`.

24-37

2   Open the Symbols window by selecting **View** > **Symbols**.

3   In each substate of `gear_state`, delete the `entry` action assigning a value to the output data variable `gear`.

4   In the Symbols window, right-click the output variable `gear` and select `Delete`.

Enable Active State Output

1   Open the Property Inspector by selecting **View** > **Property Inspector**.

2   In the Stateflow Editor canvas, select the state `gear_state`.

3   In the Property Inspector, select the **Create output for monitoring** check box and choose `Child activity`.

4   In the **Data name** field, enter the name `gear` of the active state data.

5   In the **Enum name** filed, enter the name `gearType` of the enumeration data type for the active state data.

6   In the Simulink model, reconnect the output signal `gear` from the `shift_logic` chart to the Transmission and Threshold Calculation subsystems.

View Simulation Results

The output of gear is an enumerated type managed by Stateflow. You can view the active
state output signal gear during simulation by connecting the chart to a Scope block. The
names of the enumerated values match the names of the states in gear_state. The
additional enumerated value of None indicates when no child is active.
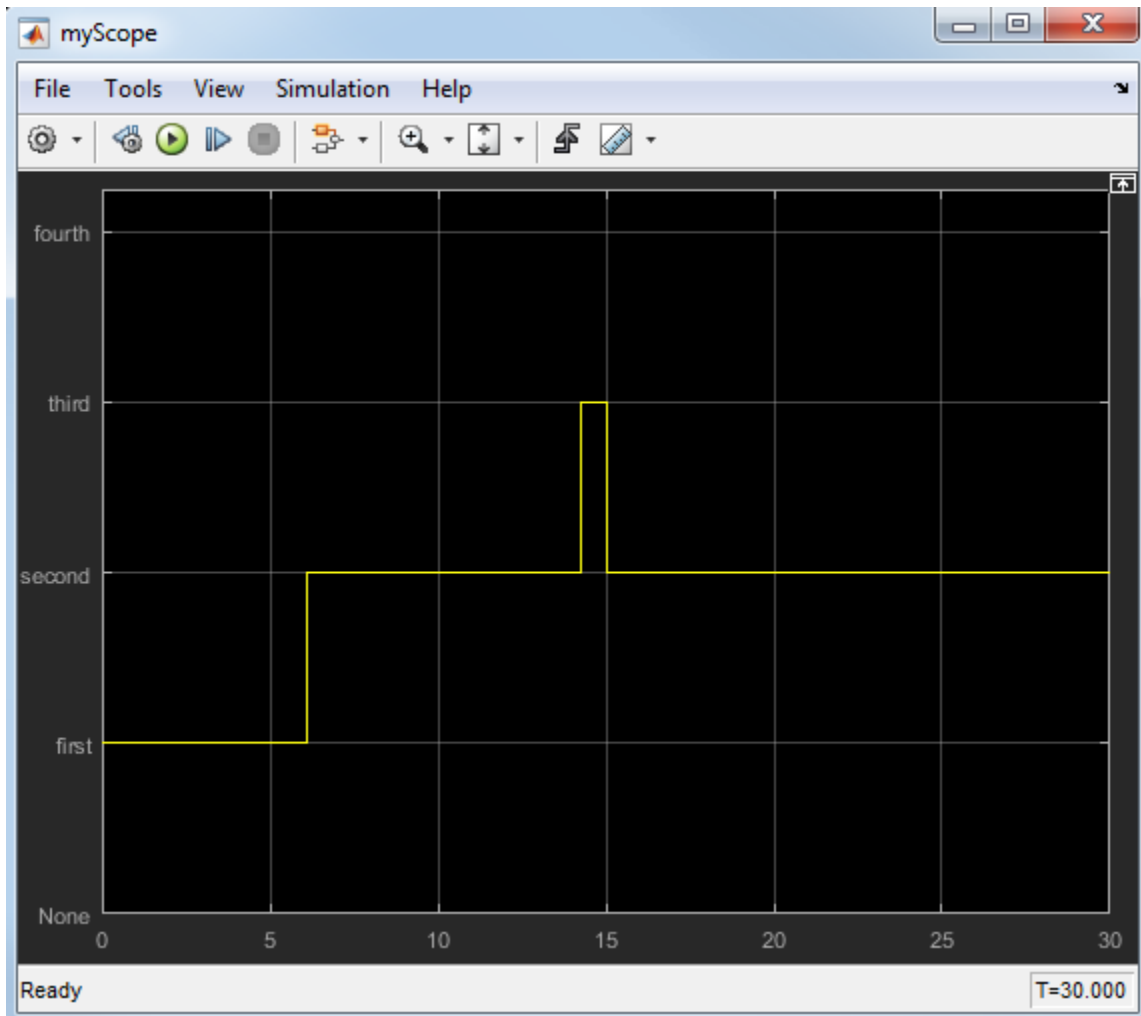
See Also

More About

- "Monitor State Activity Through Active State Data" on page 24-26
- "View State Activity by Using the Simulation Data Inspector" on page 24-33
- "Manage Data, Events, and Messages in the Symbols Window" on page 33-2

Specify Units for Stateflow Data

Units for Input and Output Data

Stateflow charts in Simulink models support the specification of physical units as properties for data inputs and outputs. Specify units by using the **Unit (e.g., m, m/s^2, N*m)** parameter for input or output data on charts, state transition tables, or truth tables. When you start typing in the field, this parameter provides matching suggestions for units that Simulink supports. By default, the property is set to inherit the unit from the Simulink signal on the corresponding input or output port. If you select the **Data must resolve to signal object** property for output data, you cannot specify units. In this case, output data is assigned the same unit type as the Simulink signal connected to the output port.

To display the units on the Simulink lines in the model, select **Display** > **Signals and Ports** > **Port Units**.

Consistency Checking

Stateflow checks the consistency of the signal line unit from Simulink with the unit setting for the corresponding input or output data in the Stateflow block. If the units do not match, Stateflow displays a warning during model update.

Units for Stateflow Limitations

The unit property settings do not affect the execution of the Stateflow block. Stateflow checks only consistency with the corresponding Simulink signal line connected to the input or output. It does not check consistency of assignments inside the Stateflow blocks. For example, Stateflow does not warn against an assignment of an input with unit set to `ft` to an output with unit set to `m`. Stateflow does not perform unit conversions.

See Also

More About

- "Unit Specification in Simulink Models" (Simulink)

25

Structures and Bus Signals in Stateflow Charts

Access Bus Signals Through Stateflow Structures

A Stateflow structure is a data type that you define from a `Simulink.Bus` object. Using Stateflow structures, you can bundle data of different size and type to create:

- Inputs and outputs that access Simulink bus signals from Stateflow charts, Truth Table blocks, and MATLAB Function blocks.
- Local data in Stateflow charts, truth tables, graphical functions, MATLAB functions, and boxes.
- Temporary data in Stateflow graphical functions, truth tables, and MATLAB functions.

For example, in the model `sf_bus_demo`, a Stateflow chart receives a bus input signal by using the structure `inbus` and outputs a bus signal from the structure `outbus`. The input signal comes from the Simulink Bus Creator block `COUNTERBUSCreator`, which bundles signals from two other Bus Creator blocks. The output structure `outbus` connects to a Simulink Bus Selector block. Both `inbus` and `outbus` derive their type from the `Simulink.Bus` object `COUNTERBUS`.
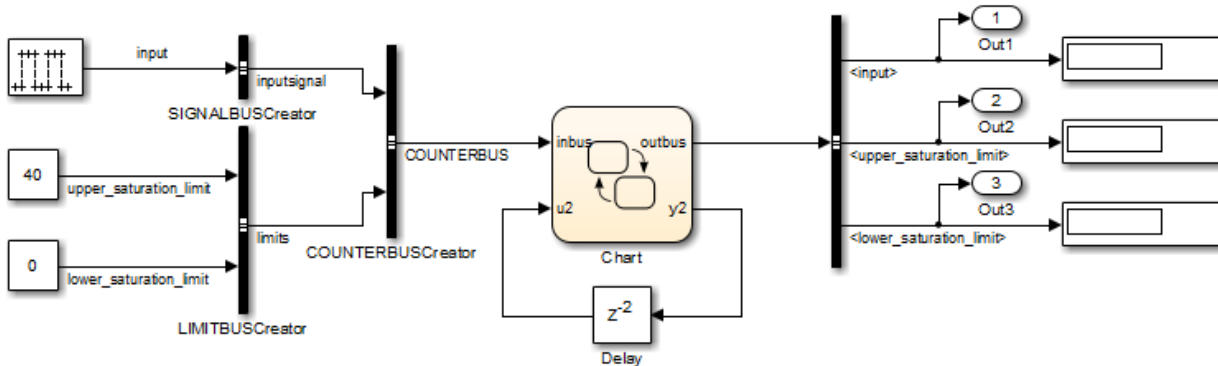
Copyright 2006-2011 The MathWorks, Inc.

The elements of a Stateflow structure data type are called *fields*. Fields can be any combination of individual signals, muxed signals, vectors, and other structures (also called substructures). Each field has its own data type. The data type does not have to match the type of any other field in the structure. For example, in the model `sfbus_demo`, each of the structures `inbus` and `outbus` has two fields:

- `inputsignal` is a substructure with one field, `input`.
- `limits` is a substructure with two fields, `upper_saturation_limit` and `lower_saturation_limit`.

Define Stateflow Structures

1 To define the structure data type, create a Simulink bus object in the base workspace, as described in "Create Bus Objects with the Bus Editor" (Simulink).

2 Add a data object to the chart, as described in "Add Stateflow Data" on page 9-2.

To define temporary structures in truth tables, graphical functions, and MATLAB functions, add a data object *to your function*. For more information, see "Add Data Through the Model Explorer" on page 9-3.

3 Set the **Scope** property for the structure. Your choices are:

- `Input`
- `Output`
- `Local`
- `Parameter`
- `Temporary`

4 Set the **Type** property for the structure. Depending on its scope, a Stateflow structure can have one of these data types.

Type	Description
`Inherit: Same as Simulink`	This option is available for input structures only. The input structure inherits its data type from the Simulink bus signal in your model that connects to it. The Simulink bus signal must be a nonvirtual bus. For more information, see "Virtual and Nonvirtual Buses" on page 25-6. In the base workspace, specify a `Simulink.Bus` object with the same properties as the bus signal that connects to the Stateflow input structure. These properties must match: • Number, name, and type of inputs • Dimension • Sample Time • Complexity • Sampling Mode If the input signal comes from a Bus Creator block, in the Bus Creator dialog box, specify an appropriate bus object for **Output data type** field. When you specify the bus object, Simulink verifies that the properties of the `Simulink.Bus` object in the base workspace match the properties of the Simulink bus signal.
`Bus: <object name>`	In the **Type** field, replace *<object name>* with the name of the `Simulink.Bus` object that defines the Stateflow structure. For input or output structures, you are not required to specify the bus signal in your Simulink model that connects to the Stateflow structure. If you do specify a bus signal, its properties must match the `Simulink.Bus` object that defines the Stateflow structure.
`<date type expression>`	In the **Type** field, replace *<data type expression>* with an expression that evaluates to a data type. For example: • Enter the name of the `Simulink.Bus` object that defines the Stateflow structure. • For structures with scopes other than `Output`, use the Stateflow `type on page 12-20` operator to copy the type of another structure. For more information, see "Specify Structure Types by Calling the type Operator" on page 25-5.

For example, in the `sfbus_demo` model, the input structure `inbus` and the output structure `outbus` derive their type through a type specification of the form `Bus: COUNTERBUS`.
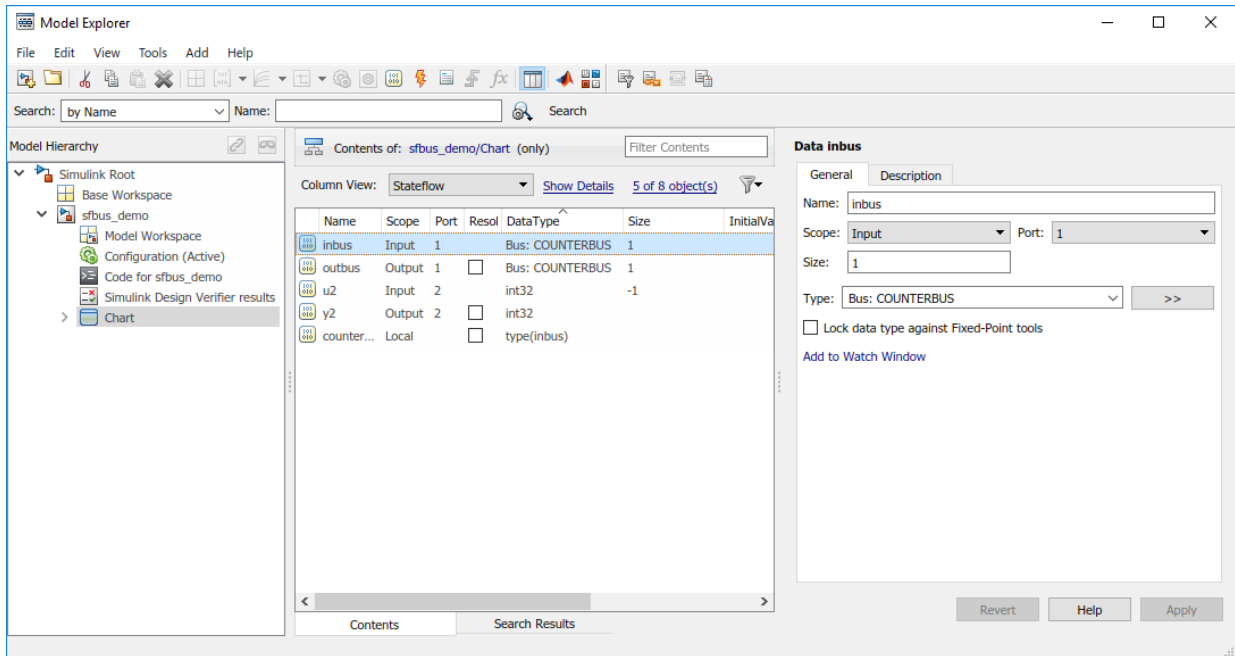
Specify Structure Types by Calling the type Operator

To specify structure types, you can use expressions that call the Stateflow `type on page 12-20` operator. This operator sets the type of one structure to the type of another structure in the Stateflow chart. For example, in the `sf_bus_demo` model, a `type` operator expression specifies the type of the local structure `counterbus_struct` in terms of the input structure `inbus`. Both structures are defined from the `Simulink.Bus` object `COUNTERBUS`. For more information, see "Derive Data Types from Other Data Objects" on page 9-37.
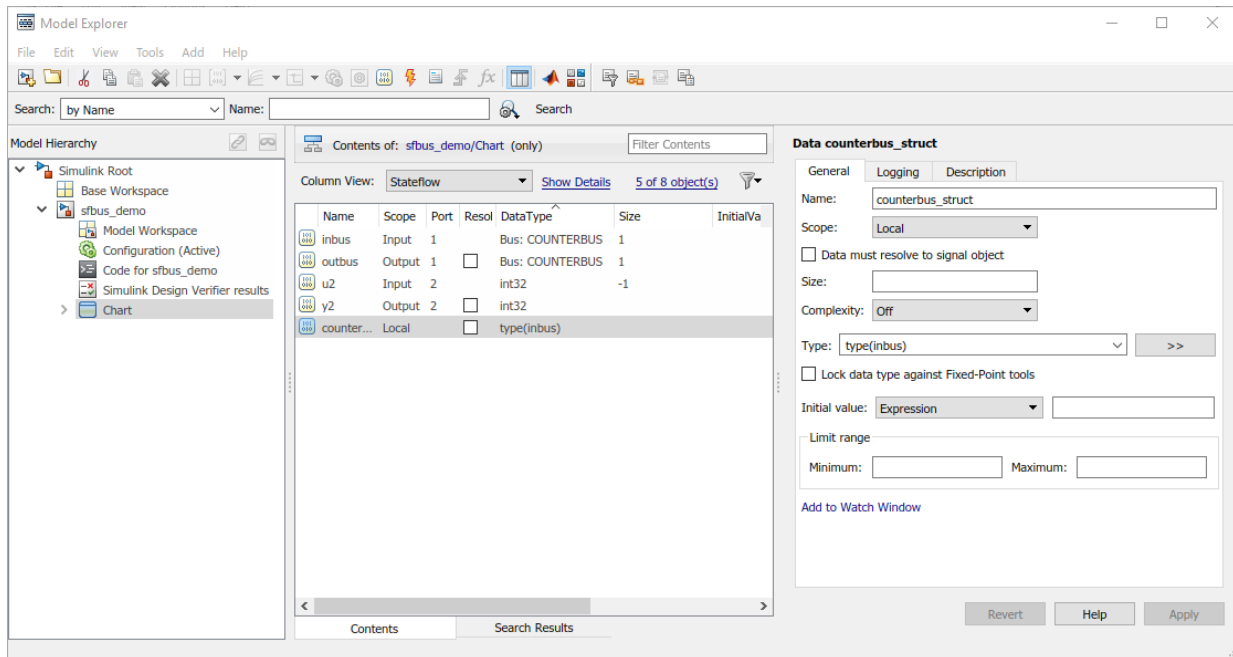
Virtual and Nonvirtual Buses

Simulink models support virtual and nonvirtual buses. Nonvirtual buses read their inputs from data structures stored in contiguous memory. Virtual buses read their inputs from noncontiguous memory. For more information, see "Virtual and Nonvirtual Buses" (Simulink).

Stateflow charts support only nonvirtual buses. Stateflow input structures can accept virtual bus signals and convert them to nonvirtual bus signals. Stateflow input structures cannot inherit properties from virtual bus signals. If the input to a chart is a virtual bus, set the **Type** property of the input structure through a type specification of the form `Bus: <object name>`.

Debug Structures

To debug a Stateflow structure, open the Stateflow Breakpoints and Watch window and examine the values of structure fields during simulation. To view the values of structure fields at the command line, use dot notation to index into the structure. For more

information, see "Watch Stateflow Data Values" on page 32-36 and "Index Substructures and Fields" on page 25-8.

Guidelines for Structure Data Types

- Define each structure from a `Simulink.Bus` object in the base workspace.
- Structures cannot have a constant scope.
- Structures of parameter scope must be tunable.
- Data array objects cannot contain structures.
- You cannot define structures for Stateflow machines. For more information, see "Hierarchy of Stateflow Objects" on page 1-7.

See Also

`Simulink.Bus`

Related Examples

- "Interface Simulink Bus Signals and Integrate Custom C Code"

More About

- "Index and Assign Values to Stateflow Structures" on page 25-8
- "Integrate Custom Structures in Stateflow Charts" on page 25-11
- "Add Stateflow Data" on page 9-2
- "Derive Data Types from Other Data Objects" on page 9-37
- "Watch Stateflow Data Values" on page 32-36
- "Create Bus Objects with the Bus Editor" (Simulink)
- "Virtual and Nonvirtual Buses" (Simulink)

Index and Assign Values to Stateflow Structures

Stateflow structures enable you to bundle data of different size and type together into a single `Simulink.Bus` object. Using dot notation and numeric indices, you can access and modify the contents of a Stateflow structure. For more information, see "Access Bus Signals Through Stateflow Structures" on page 25-2.

Index Substructures and Fields

To index substructures and fields of Stateflow structures, use dot notation. The first part of a name identifies the parent object. Subsequent parts identify the children along a hierarchical path. When the parent is a structure, its children are individual fields or fields that contain other structures (also called substructures). The names of the fields of a Stateflow structure match the names of the elements of the `Simulink.Bus` object that defines the structure.

For example, the C chart in this model contains an input structure `in`, an output structure `out`, and a local structure `subbus`.
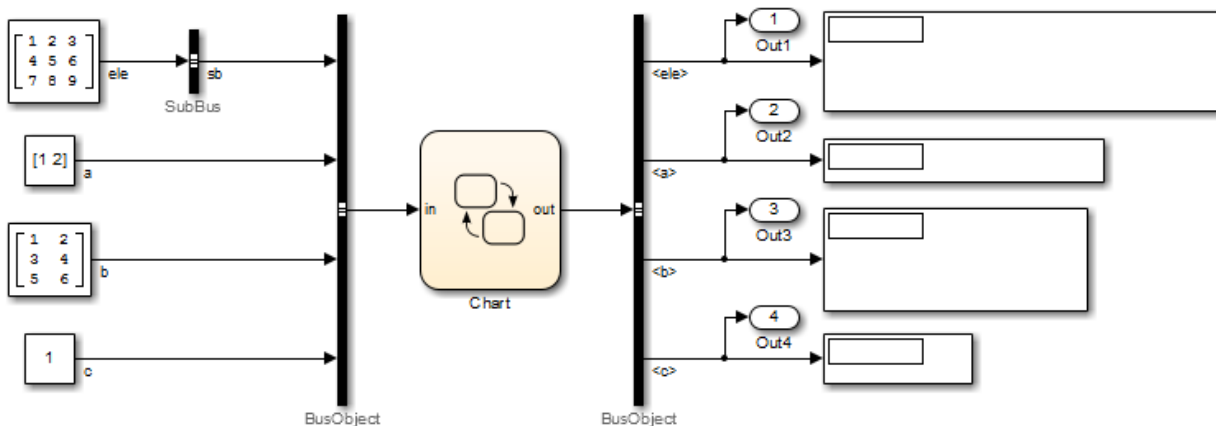
Structure	Scope	Simulink.Bus Object
in	Input	BusObject
out	Output	BusObject
subbus	Local	SubBus

The fields of the input structure `in` and the output structure `out` have the same name as the elements of `Simulink.Bus` object `BusObject` that defines them: `sb`, `a`, `b`, and `c`. The field of the local structure `subbus` has the same name `ele` as the element of `Simulink.Bus` object `SubBus`. This table lists how the Stateflow chart resolves symbols in dot notation for indexing the fields of these structures.

Dot Notation	Symbol Resolution
`in.c`	Field `c` of input structure `in`
`out.sb`	Substructure `sb` of output structure `out`
`in.a[0]`	First value of the vector field `a` of input structure `in`
`subbus.ele[1][1]`	Value in the second row, second column of field `ele` of local structure `subbus`
`in.sb.ele[2][3]`	Value in the third row, fourth column of field `ele` of substructure `in.sb`

Note In this example, the Stateflow chart uses brackets and zero-based indexing for vectors and arrays because C is the action language for the chart. For more information, see "Differences Between MATLAB and C as Action Language Syntax" on page 13-6.

Assign Values to Structures and Fields

You can assign values to any Stateflow structure, substructure, or a field of a structure with a scope different from `Input`.

- To assign one structure to another structure, define both structures from the same `Simulink.Bus` object in the base workspace.
- To assign one structure to a substructure of a different structure (or vice versa), define the structure and substructure from the same `Simulink.Bus` object.
- To assign a field of one structure to a field of another structure, the fields must have the same type and size. You can define the Stateflow structures from different `Simulink.Bus` objects.

This table presents valid and invalid structure assignments based on the structure specifications for the previous example.

Assignment	Valid or Invalid?	Explanation
`in = out;`	Invalid	You cannot write to input structures.
`out = in;`	Valid	Both `in` and `out` are defined from the same `Simulink.Bus` object `BusObject`.
`subbus = in;`	Invalid	The structures `subbus` and `in` are defined from different `Simulink.Bus` objects.
`in.sb = subbus;`	Invalid	You cannot write to substructures of input structures.
`out.sb = subbus;`	Valid	The substructure `out.sb` and the structure `subbus` are defined from the same `Simulink.Bus` object `SubBus`.
`in.c = out.c;`	Invalid	You cannot write to fields of input structures.
`out.sb.ele = subbus.ele;`	Valid	The field `out.sb.ele` has the same type and size as the field `subbus.ele` (3-by-3 matrices).
`subbus.ele[1][1] = in.c;`	Valid	The field `subbus.ele[1][1]` has the same type and size as the field `in.c` (scalars).

See Also

`Simulink.Bus`

More About

- "Access Bus Signals Through Stateflow Structures" on page 25-2
- "Identify Data by Using Dot Notation" on page 9-50

Integrate Custom Structures in Stateflow Charts

You can define custom structures in C code, which you can integrate with your Stateflow chart in a Simulink model for simulation and code generation. For example, the model sf_bus_demo uses a custom C function to write to the output structure outbus. By sharing data with custom code, you can augment the capabilities supported by Stateflow and take advantage of your preexisting code. For more information, see "Reuse Custom Code in Stateflow Charts" on page 30-26 and "Access Bus Signals Through Stateflow Structures" on page 25-2.
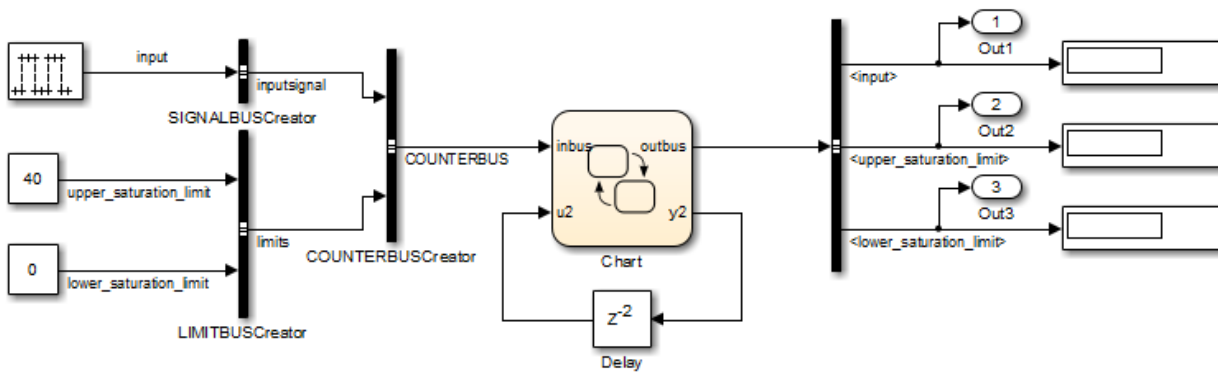
Copyright 2006-2011 The MathWorks, Inc.

Define Custom Structures in C Code

1   In your C code, define a structure by creating a custom header file. The header file contains typedef declarations matching the properties of the Simulink.Bus object that defines the Stateflow structure. For example, in the model sfbus_demo, the header file counterbus.h declares three custom structures:

```
...
typedef struct {
  int input;
} SIGNALBUS;

typedef struct {
  int upper_saturation_limit;
  int lower_saturation_limit;
```

```
} LIMITBUS;

typedef struct {
  SIGNALBUS inputsignal;
  LIMITBUS limits;
} COUNTERBUS;
...
```

2   In the Bus Editor, define a `Simulink.Bus` object that matches each custom structure `typedef` declaration. In the **Header file** field of each `Simulink.Bus` object, enter the name of the header file that contains the matching `typedef` declaration.
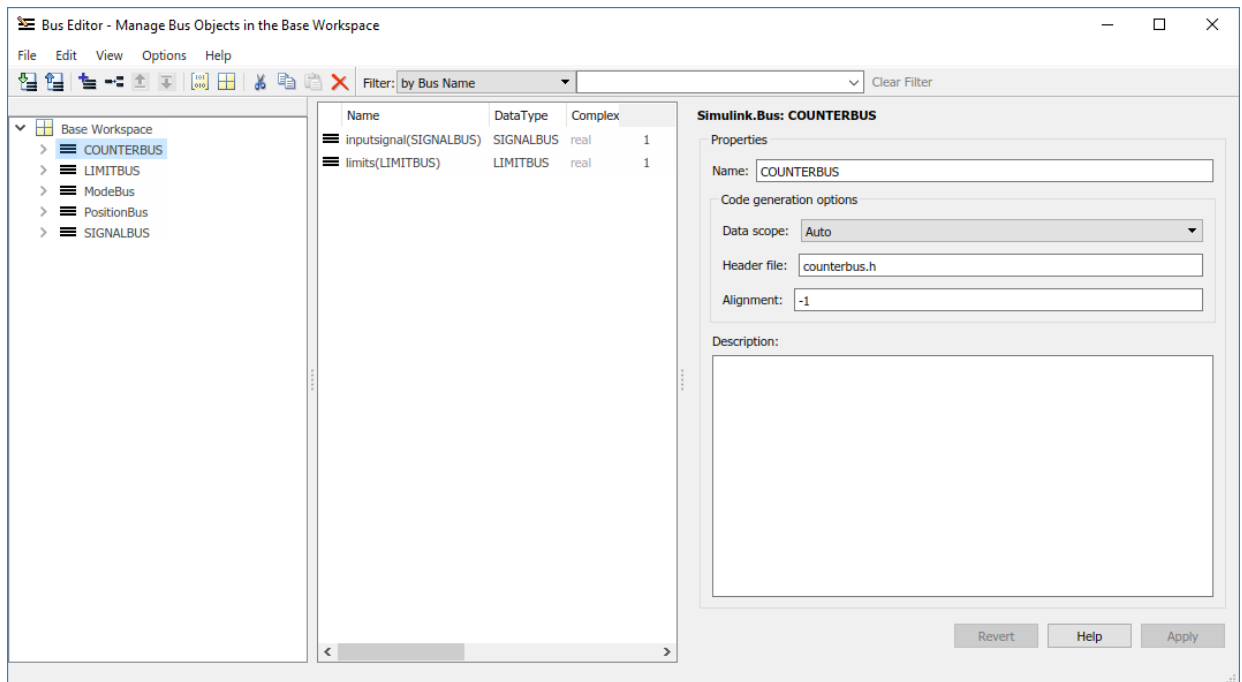
3   Configure your C action language chart to include custom C code.

- To include custom code for simulation, see "Access Custom C Code in Nonlibrary Charts" on page 30-5.

- To include custom code for code generation, see "Integrate External Code by Using Model Configuration Parameters" (Simulink Coder).

4   Build and run your model.

Pass Stateflow Structures to Custom Code

When you write custom code functions that take structure pointers as arguments, pass the Stateflow structures by address. To pass the address of a Stateflow structure or one of its fields to a custom function, use the & operator and dot notation:

- `&outbus` provides the address of the Stateflow structure `outbus`.
- `&outbus.inputsignal` provides the address of the substructure `inputsignal` of the structure `outbus`.
- `&outbus.inputsignal.input` provides the address of the field `input` of the substructure `outbus.inputsignal`.
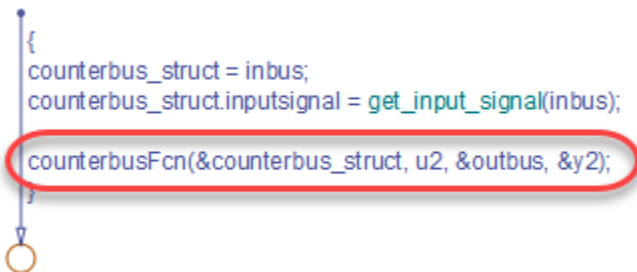
For more information, see "Index Substructures and Fields" on page 25-8.

For instance, the model `sfbus_demo` contains a custom C function `counterbusFcn` that takes structure pointers as arguments. The custom header file contains this function declaration:

```
extern void counterbusFcn
          (COUNTERBUS *u1, int u2, COUNTERBUS *y1, int *y2);
```

The chart passes the addresses to the Stateflow structures `counterbus_struct` and `outbus` by using this function call:

```
counterbusFcn(&counterbus_struct, u2, &outbus, &y2);
```


The function reads the value of the chart input `u2` and the local structure `counterbus_struct`. It writes to the chart output `y2` and the output structure `outbus`.

25-13

See Also

`Simulink.Bus`

Related Examples

- "Interface Simulink Bus Signals and Integrate Custom C Code"

More About

- "Access Bus Signals Through Stateflow Structures" on page 25-2
- "Index and Assign Values to Stateflow Structures" on page 25-8
- "Reuse Custom Code in Stateflow Charts" on page 30-26
- "Integrate External Code by Using Model Configuration Parameters" (Simulink Coder)
- "Identify Data by Using Dot Notation" on page 9-50

26

Stateflow Design Patterns

Reduce Transient Signals by Using Debouncing Logic

Why Debounce Signals

When a switch opens and closes, the switch contacts can bounce off each other before the switch completely transitions to an on or off state. The bouncing action can produce transient signals that do not represent a true change of state. Therefore, when modeling switch logic, it is important to filter out transient signals by using *debouncing* algorithms.

If you model a controller in a Stateflow chart, you do not want your switch logic to overwork the controller by turning it on and off in response to every transient signal it receives. To avoid this, design a Stateflow controller that uses temporal logic to debounce your input signals and determine whether a switch is actually on or off.

How to Debounce a Signal

There are two ways to debounce a signal by using Stateflow:

1 Filter out transient signals by using the `duration` temporal operator.
2 Filter out transient signals by using an intermediate graphical state. Use intermediate graphical state for advanced filtering techniques, such as fault detection.

The `duration` operator is supported only in Stateflow charts in a Simulink model.

Debounce Signals with the duration Operator

The model `sf_debouncer_using_duration` illustrates a design pattern that uses the `duration` operator to filter out transient signals.
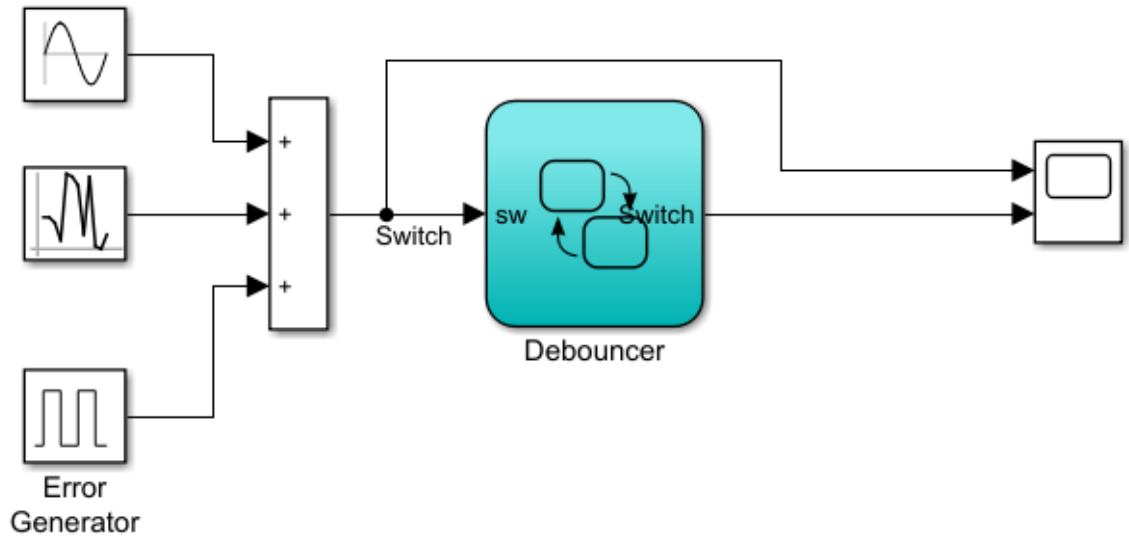
The Debouncer chart contains this logic.
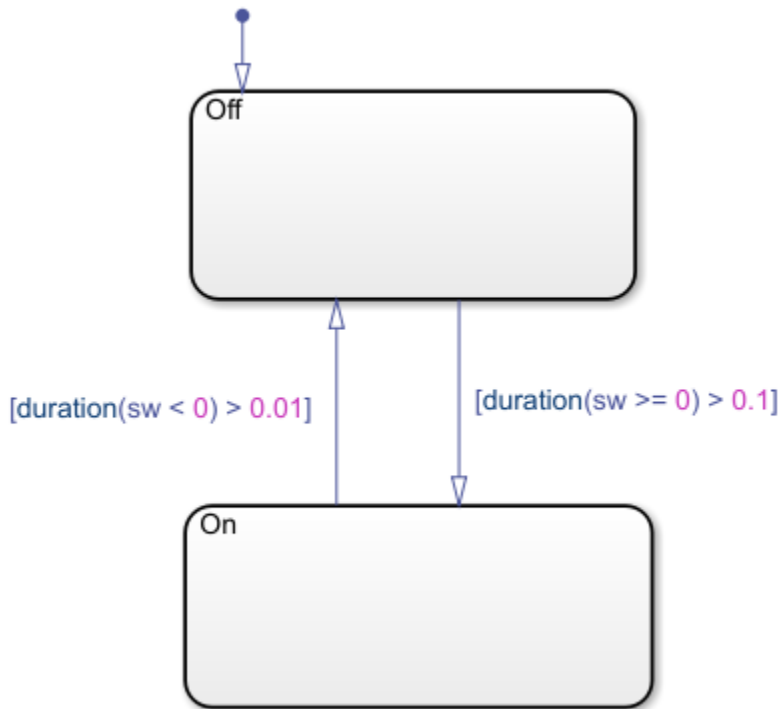
The debouncer design uses the duration(*n*) statement to implement absolute-time temporal logic.

The initial state for this model is Off. Using the duration operator, you can control what state the model is in based on how long the switch signal, sw, has been greater or less than zero. Once sw has been greater than or equal to zero for longer than 0.1 seconds, the switch moves from state Off to state On. Then, if sw has been less than zero for longer than 0.01 seconds, the switch moves from state On to state Off.

State Logic

The debouncer has two states, On and Off. The duration operator controls which state is active. The logic works as described in this table.

Input Signal	State	Result
Retains positive value for 0.1 second	On	Switch turns on
Retains negative value for 0.01 second	Off	Switch turns off

Run the Debouncer

1. Open the `sf_debouncer_using_duration` model.
2. Open the Stateflow chart Debouncer and the Scope block.
3. Simulate the chart.

   The scope shows how the debouncer isolates transient signals from the noisy input signal.
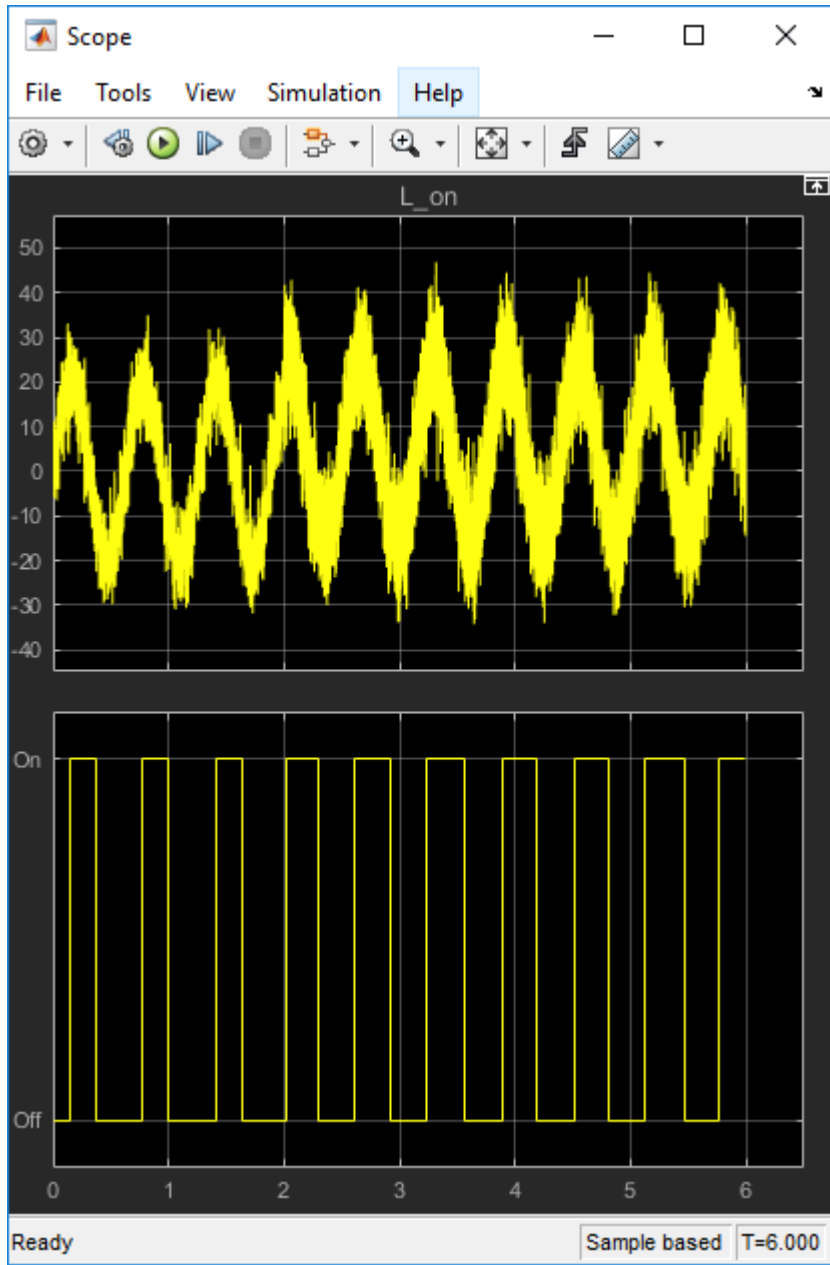
Debounce Signals with Fault Detection

The model sf_debouncer illustrates a design pattern that uses temporal logic and an intermediate state to isolate transient signals. With this design pattern, you can also include logic to detect faults and allow your system time to recover.
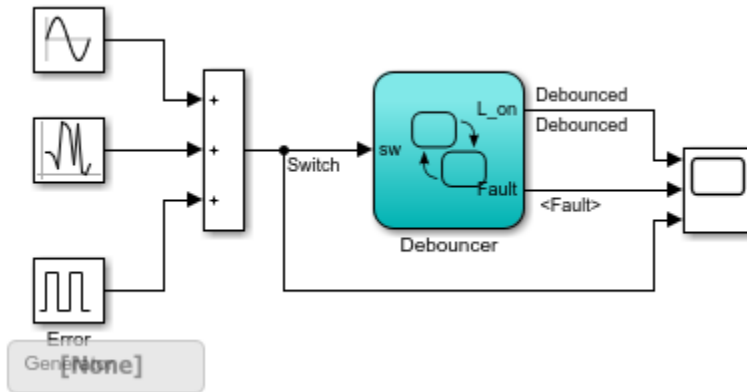
The Debouncer chart contains this logic.
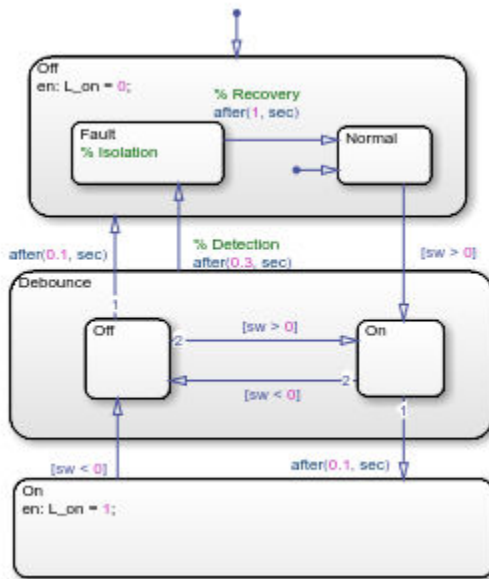
The debouncer design uses the `after(n, sec)` statement to implement absolute-time temporal logic. The keyword `sec` defines simulation time that has elapsed since activation of a state.

State Logic

The debouncer chart contains an intermediate state called Debounce. The Debounce state isolates transient inputs by checking whether the signals retain their positive or negative values, or fluctuate between zero crossings over a prescribed period. The logic works as shown in this table.

Input Signal	State	Transitions	Result
Retains positive value for 0.1 second	Debounce.On	On	Switch turns on
Retains negative value for 0.1 second	Debounce.Off	Off	Switch turns off

Input Signal	State	Transitions	Result
Fluctuates between zero crossings for 0.3 second	Debounce	Off.Fault **Note** The Debounce to Off.Fault transition comes from a higher level in the chart hierarchy and overrides the transitions from the Debounce.Off and Debounce.On substates.	Chart isolates the input as a transient signal and gives it time to recover.

Run the Debouncer

1 Open the `sf_debouncer`.
2 Open the Stateflow chart Debouncer and the Scope block.
3 Simulate the chart.

   The scope shows how the debouncer isolates transient signals from the noisy input signal.
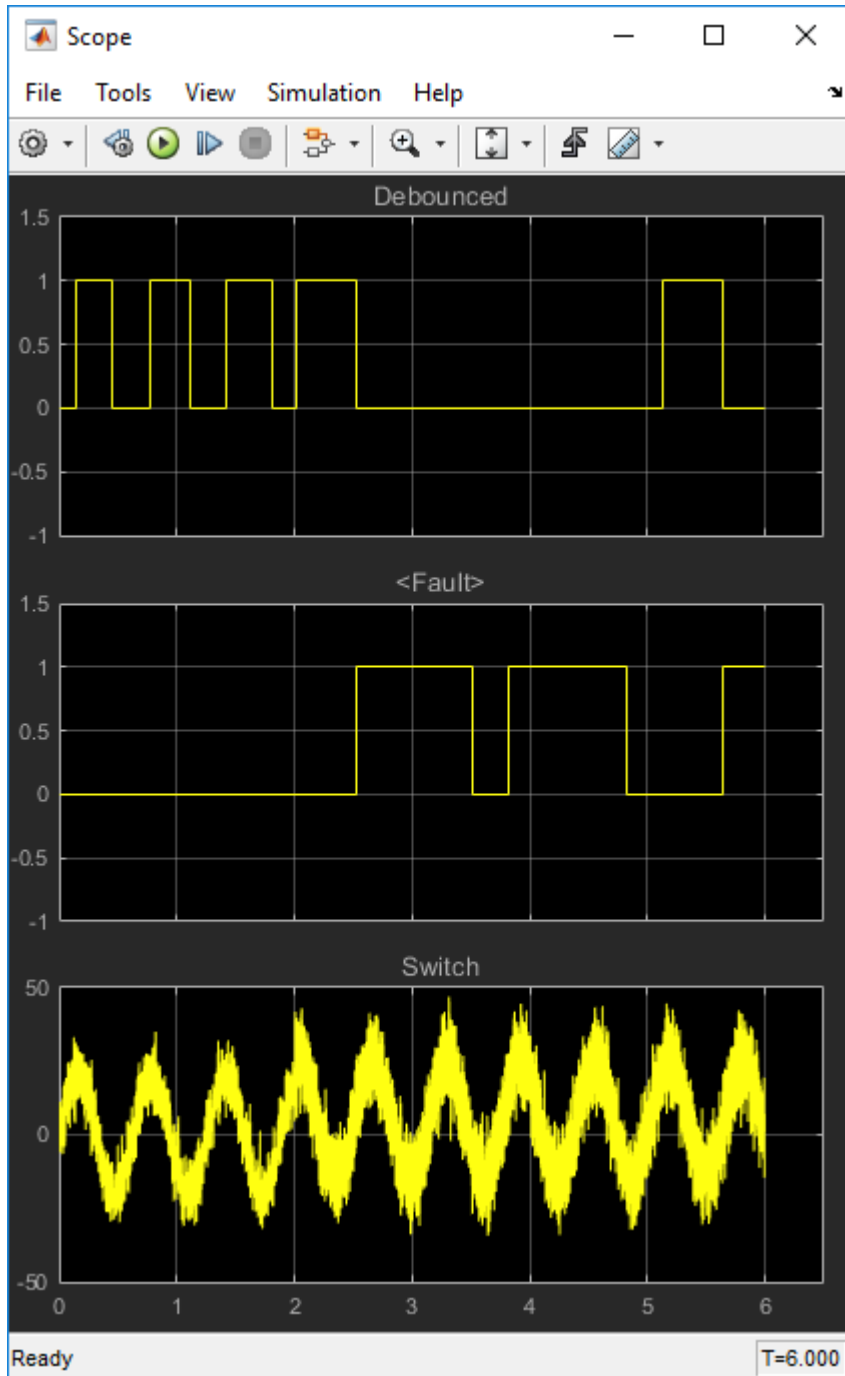
Use Event-Based Temporal Logic

As an alternative to absolute-time temporal logic, you can apply event-based temporal logic to determine true state in the Debouncer chart by using the `after(n, tick)` statement. The keyword `tick` specifies and implicitly generates a local event when the chart awakens.

The Error Generator block in the `sf_debouncer` model generates a pulse signal every 0.001 second. Therefore, to convert the absolute-time temporal logic specified in the Debouncer chart to event-based logic, multiply the *n* argument by 1000, as follows.

Absolute Time-Based Logic	Event-Based Logic
after (0.1, sec)	after (100, tick)
after (0.3, sec)	after (300, tick)
after (1, sec)	after (1000, tick)

See Also

duration

More About

- "Operators for Absolute-Time Temporal Logic" on page 12-56
- "Operators for Event-Based Temporal Logic" on page 12-50
- "Define Chart Behavior by Using Implicit Events" on page 10-33

Schedule Function Calls

You can schedule calls to Simulink and MATLAB functions by using conditional and time-based logic in Stateflow. You can design logic using temporal operators without requiring timers and counters. Temporal logic can be based on events or elapsed time.

See Also

Related Examples

- "Schedule Simulink Algorithms by Using Stateflow"

More About

- "Control Chart Execution by Using Temporal Logic" on page 12-49

Schedule Execution of Simulink Subsystems

When to Implement Schedulers

Use Stateflow charts to schedule the order of execution of Simulink subsystems *explicitly* in a model. Stateflow schedulers extend control of subsystem execution in a Simulink model, which determines order of execution *implicitly* based on block connectivity via sample time propagation.

Types of Schedulers

You can implement the following types of schedulers using Stateflow charts.

Scheduler Design Pattern	Description
Ladder logic scheduler on page 26-14	Schedules multiple Simulink subsystems to execute in a single time step
Loop scheduler on page 26-18	Schedules one Simulink subsystem to execute multiple times in a single time step
Temporal logic scheduler on page 26-22	Schedules Simulink subsystems to execute at specific times

Schedule Multiple Subsystems in a Single Step

The **ladder logic scheduler** design pattern allows you to specify the order in which multiple Simulink subsystems execute in a single time step. The model `sf_ladder_logic_scheduler` illustrates this design pattern.

The Ladder Logic Scheduler chart contains the following logic:

Sequencing
Ladder Logic
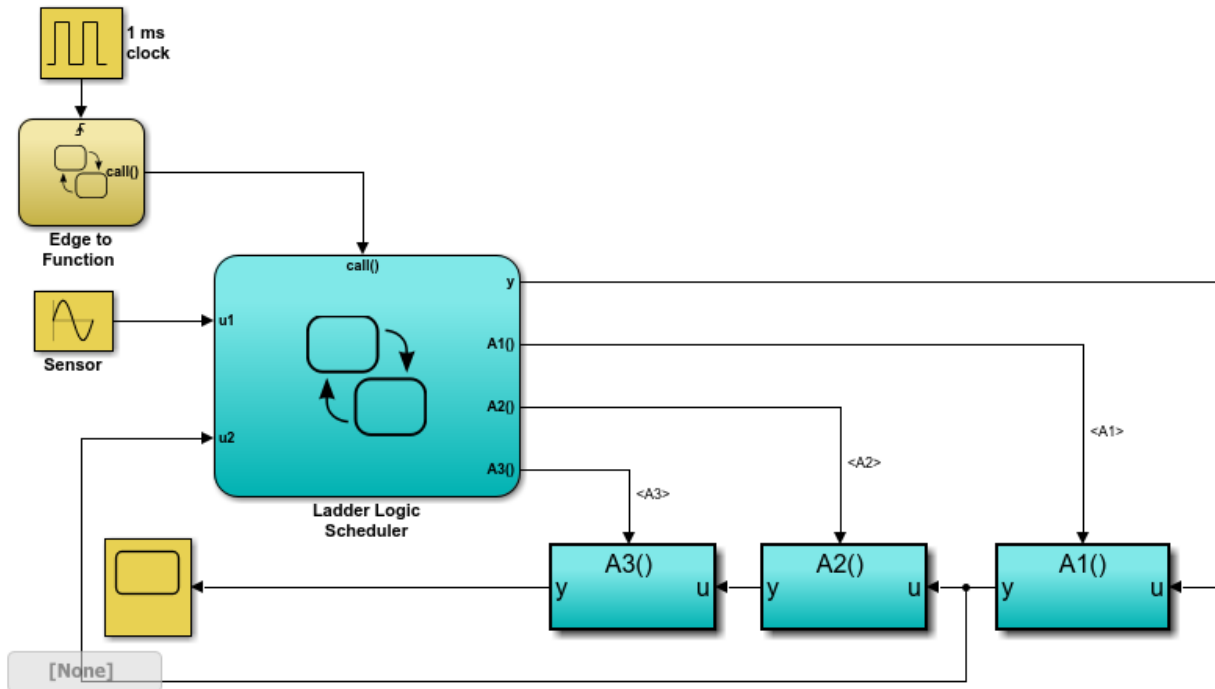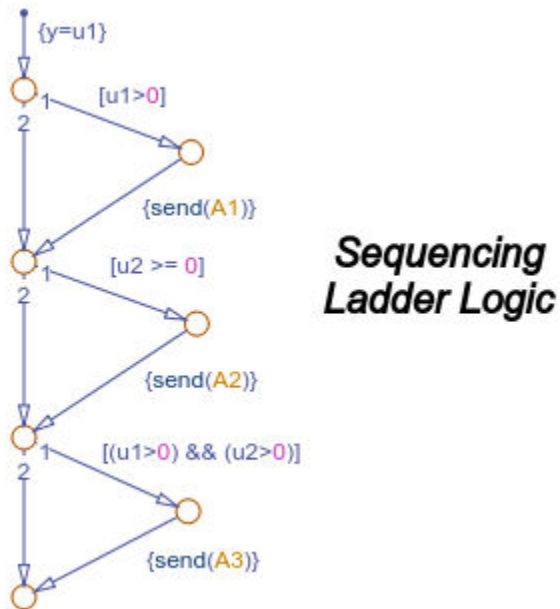
Key Behaviors of Ladder Logic Scheduler

The key behaviors of the ladder logic scheduler are:

- "Function-Call Output Events Trigger Multiple Subsystems" on page 26-15
- "Flow Chart Determines Order of Execution" on page 26-16

Function-Call Output Events Trigger Multiple Subsystems

In a given time step, the Stateflow chart broadcasts a series of function-call output events to trigger the execution of three function-call subsystems — A1, A2, and A3 — in the Simulink model in an order determined by the ladder logic scheduler. Here is the sequence of activities during each time step:

1   The Simulink model activates the Stateflow chart Edge to Function at a rising edge of the 1-millisecond pulse generator.

2   The Edge to Function chart broadcasts the function-call output event `call` to activate the Stateflow chart Ladder Logic Scheduler.

3 The Ladder Logic Scheduler chart broadcasts function-call output events to trigger the function-call subsystems A1, A2, and A3, based on the values of inputs u1 and u2 (see "Flow Chart Determines Order of Execution" on page 26-16).

Flow Chart Determines Order of Execution

The Ladder Logic Scheduler chart uses Stateflow flow charting capabilities to implement the logic that schedules the execution of the Simulink function-call subsystems. The chart contains a Stateflow flow chart that resembles a ladder diagram. Each rung in the ladder represents a rule or condition that determines whether to execute one of the Simulink function-call subsystems. The flow logic evaluates each condition sequentially, which has the effect of scheduling the execution of multiple subsystems within the same time step. The chart executes each subsystem by using the send action to broadcast a function-call output event (see "Directed Local Event Broadcast Using send" on page 12-45).

Here is the sequence of activities that occurs in the Ladder Logic Scheduler chart in each time step:

1 Assign output y to input u1.

2 If u1 is positive, send function-call output event A1 to the Simulink model.

The subsystem connected to A1 executes. This subsystem multiplies its input by a gain of 2 and passes this value back to the Stateflow Ladder Logic Scheduler chart as input u2. Control returns to the next condition in the Ladder Logic Scheduler.

3 If u2 is positive or zero, send function-call output event A2 to the Simulink model.

The subsystem connected to A2 executes. This subsystem outputs its input value unchanged. Control returns to the next condition in the Ladder Logic Scheduler.

4 If u1 and u2 are positive, send function-call output event A3 to the Simulink model.

The subsystem connected to A3 executes. This subsystem multiplies its input by a gain of 1.

5 The Ladder Logic Scheduler chart goes to sleep.

Run the Ladder Logic Scheduler

1 Open the sf_ladder_logic_scheduler model.

2 Open the Scope block.

3 Start simulation.

The scope shows how output y changes, depending on which subsystems the Ladder Logic Scheduler chart calls during each time step.
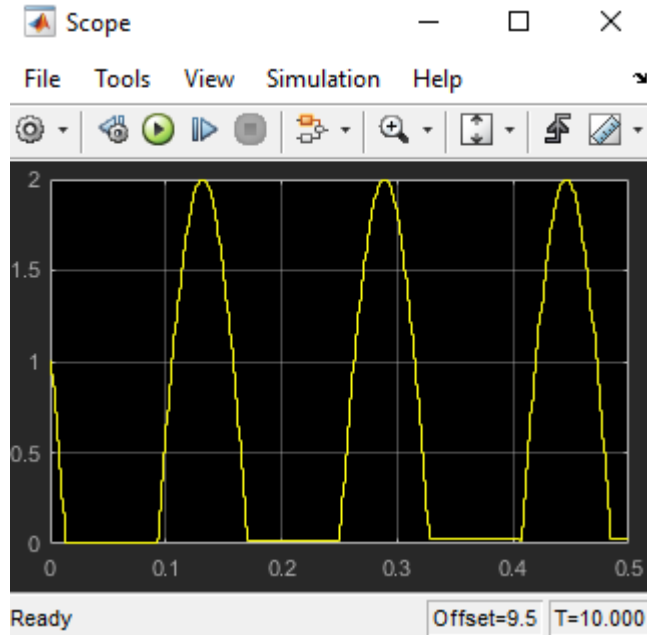
Tip If you keep the chart closed, the simulation runs much faster. For other tips, see "Speed Up Simulation" on page 30-17.

Schedule A Subsystem Multiple Times in a Single Step

With the **loop scheduler** design pattern, you can schedule one Simulink subsystem to execute multiple times in a single time step. The model `sf_loop_scheduler` illustrates this design pattern.
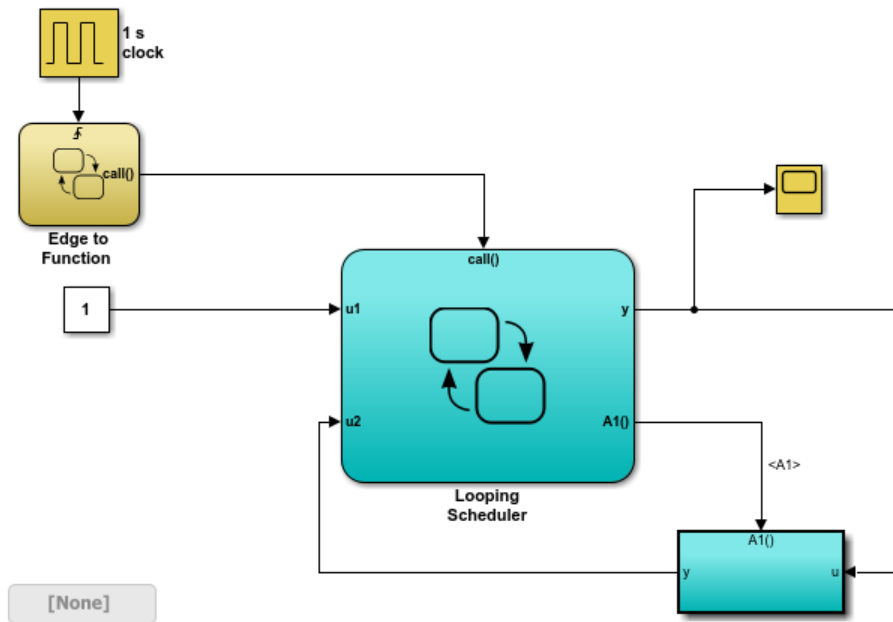
The Looping Scheduler chart contains the following logic:
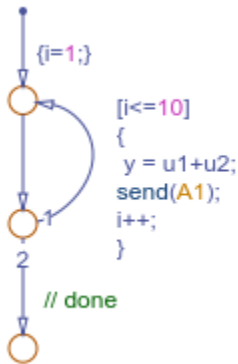
Key Behaviors of Loop Scheduler

The key behaviors of the loop scheduler are:

- "Function-Call Output Event Triggers Subsystem Multiple Times" on page 26-19
- "Flow Chart Implements For Loop" on page 26-19

Function-Call Output Event Triggers Subsystem Multiple Times

In a given time step, the Stateflow chart broadcasts a function-call output event to trigger the execution of the function-call subsystem A1 multiple times in the Simulink model. Here is the sequence of activities during each time step:

1. The Simulink model activates the Stateflow chart Edge to Function at a rising edge of the 1-millisecond pulse generator.

2. The Edge to Function chart broadcasts the function-call output event `call` to activate the Stateflow chart Looping Scheduler.

3. The Looping Scheduler chart broadcasts a function-call output event from a `for` loop to trigger the function-call subsystem A1 multiple times (see "Flow Chart Implements For Loop" on page 26-19).

Flow Chart Implements For Loop

The Looping Scheduler chart uses Stateflow flow charting capabilities to implement a `for` loop for broadcasting an event multiple times in a single time step. The chart contains a Stateflow flow chart that uses a local data variable `i` to control the loop. At each iteration,

the chart updates output y and issues the `send` action to broadcast a function-call output event that executes subsystem A1. Subsystem A1 uses the value of y to recompute its output and send the value back to the Looping Scheduler chart.

Run the Loop Scheduler

1 Open the `sf_loop_scheduler` model.

2 Open the Scope block.

3 Start simulation.

The scope displays the value of y at each time step.

In this example, the Looping Scheduler chart executes the `for` loop 10 times in each time step. During each iteration:

1 The chart increments y by 1 (the constant value of input u1).

2 The chart broadcasts a function-call output event that executes subsystem A1.

3 Subsystem A1 multiplies y by a gain of 1.

4   Control returns to the chart.

Schedule Subsystems to Execute at Specific Times

The **temporal logic scheduler** design pattern allows you to schedule Simulink subsystems to execute at specified times. The model `sf_temporal_logic_scheduler` illustrates this design pattern.
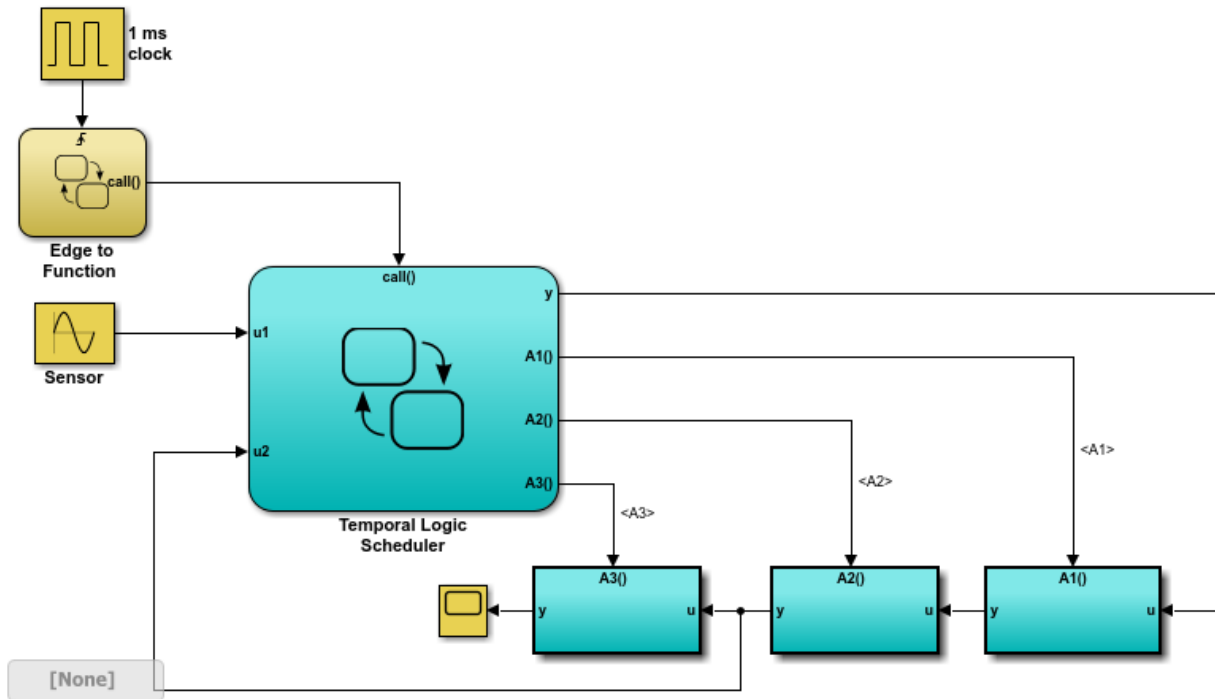
The Temporal Logic Scheduler chart contains the following logic:
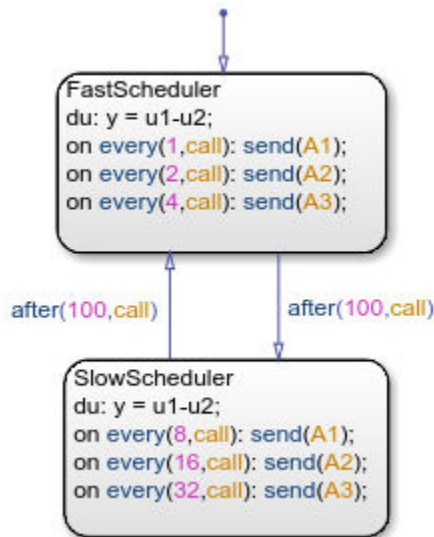
Key Behaviors of Temporal Logic Scheduler

The Temporal Logic Scheduler chart contains two states that schedule the execution of the function-call subsystems A1, A2, and A3 at different rates, as determined by the temporal logic operator every (see "Operators for Event-Based Temporal Logic" on page 12-50).

In the FastScheduler state, the every operator schedules function calls as follows:

- Sends A1 every time the function-call output event call wakes up the chart
- Sends A2 at half the base rate
- Sends A3 at one-quarter the base rate

The SlowScheduler state schedules function calls less frequently — at 8, 16, and 32 times slower than the base rate. The chart switches between fast and slow executions after every 100 invocations of the call event.

Run the Temporal Logic Scheduler

To run the sf_temporal_logic_scheduler model, follow these steps:

1 Open the model.

2 Open the Scope block.

3 Start simulation.

4 After the simulation ends, click the **Autoscale** button in the Scope block.

The scope illustrates the different rates of execution.
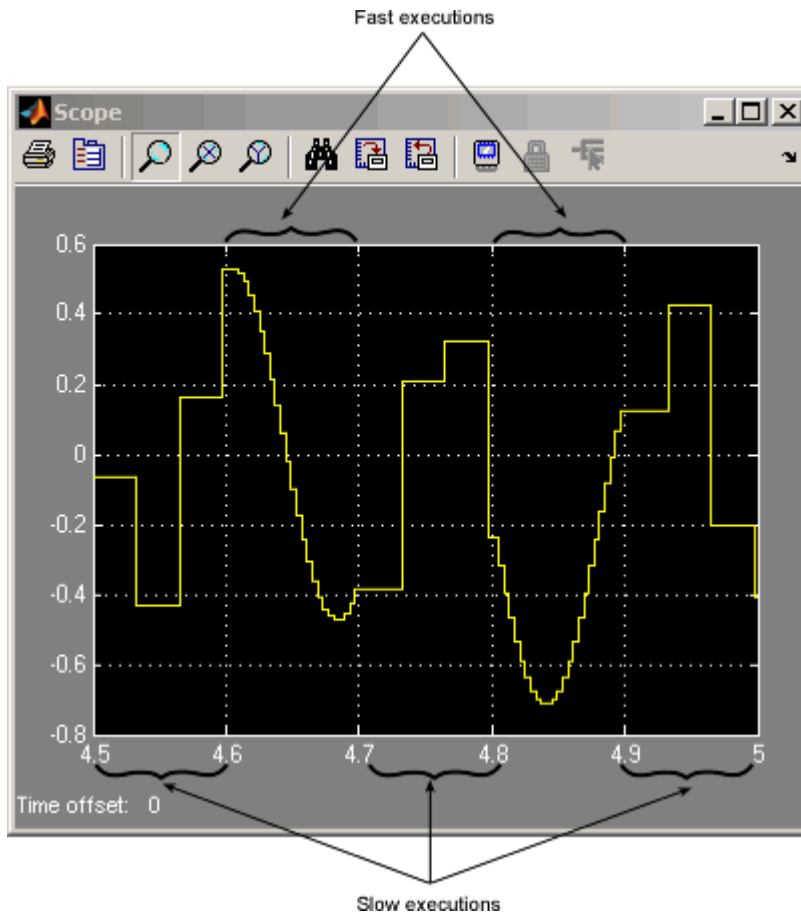
Implement Dynamic Test Vectors

When to Implement Test Vectors

Use Stateflow charts to create test vectors that change *dynamically* during simulation, based on the state of the system you are modeling.

For example, suppose you want to test an automatic car transmission controller in the situation where a car is coasting. To achieve a coasting state, a driver accelerates until the transmission shifts into the highest gear, then eases up on the gas pedal. To test this scenario, you could generate a signal that represents this behavior, as in the following Signal Builder block.
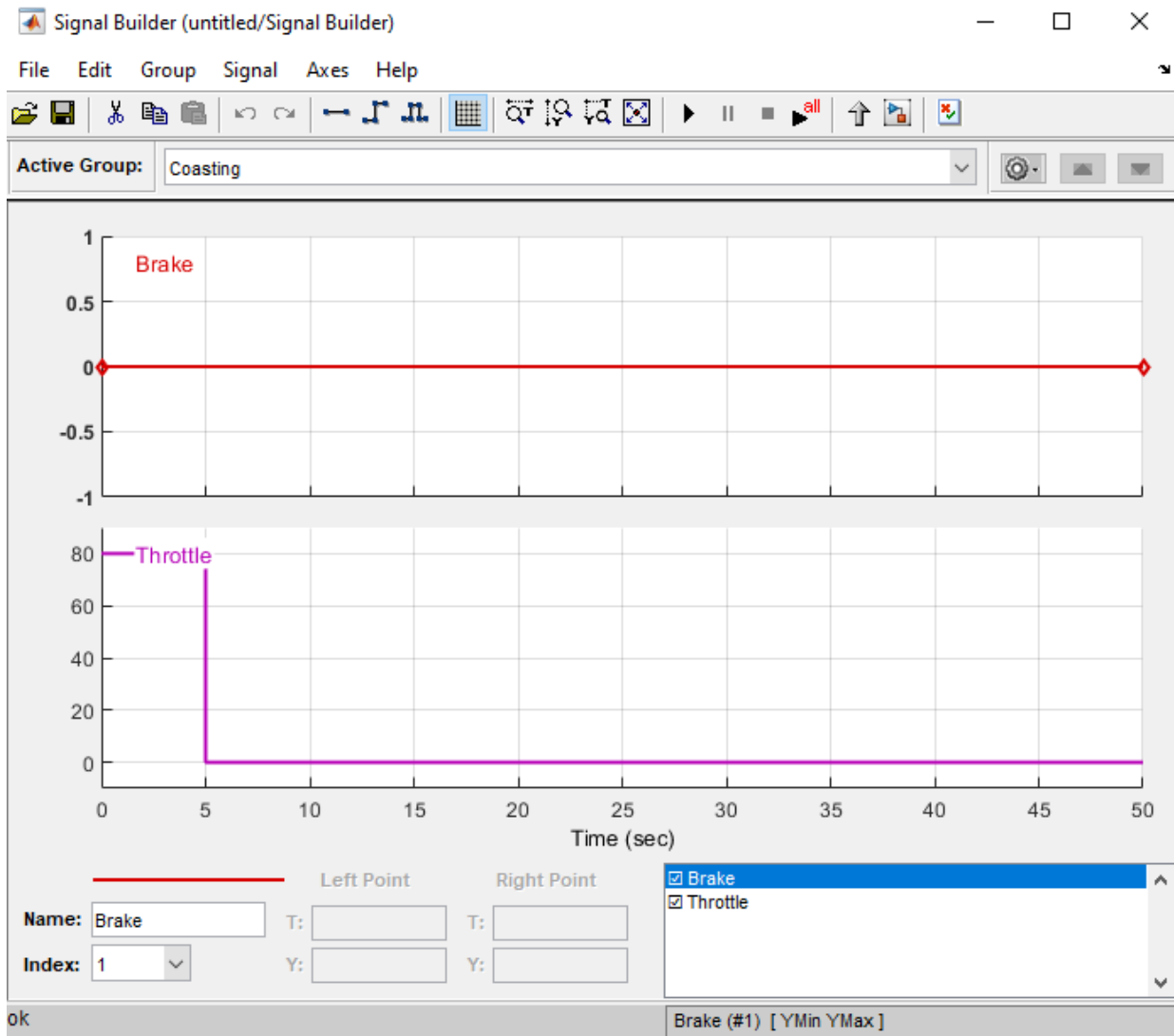
However, this approach has limitations. The signal changes value based on time, but cannot respond dynamically to changes in the system that are not governed by time alone. For example, how does the signal know when the transmission shifts into the highest gear? In this case, the signal assumes that the shift always occurs at time 5

because it cannot test for other deterministic conditions such as the speed of the vehicle. Moreover, you cannot change the signal based on outputs from the model.

By contrast, you can use Stateflow charts to develop test vectors that use conditional logic to evaluate and respond to changes in system state as they occur. For example, to test the coasting scenario, the chart can evaluate an output that represents the gear range and reduce speed only after the transmission shifts to the highest gear. That is, the car slows down as a direct result of the gear shift and not at a predetermined time.

A Dynamic Test Vector Chart

The following model of an automatic transmission controller uses a Stateflow chart to implement test vectors that represent brake, throttle, and gear shift dynamics. The chart, called Dynamic Test Vectors, interfaces with the rest of the model as shown.
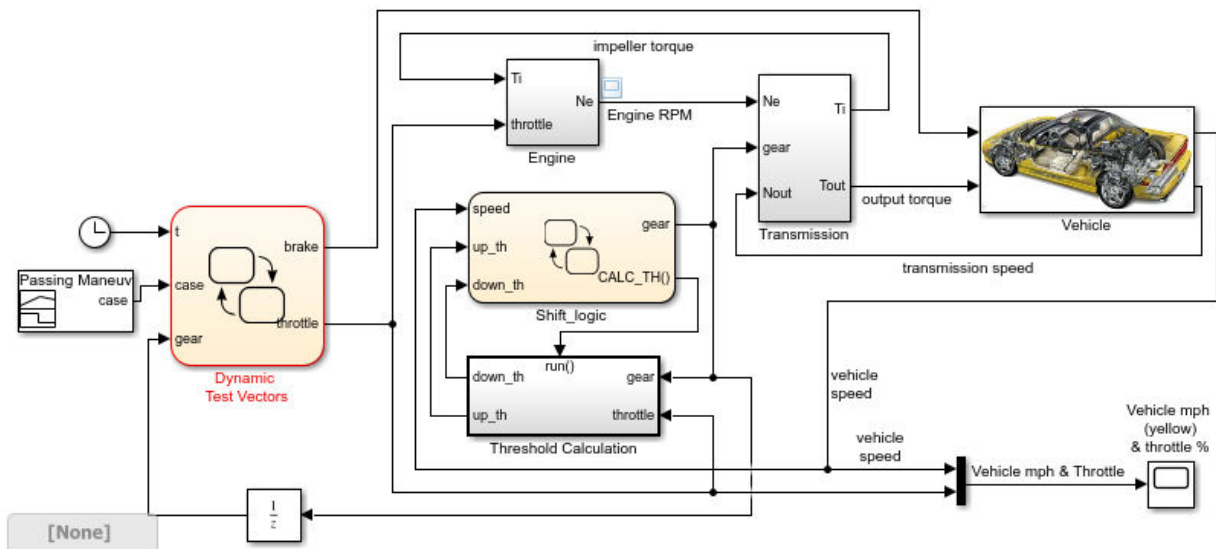
The chart models the dynamic relationship between the brake and throttle to test four driving scenarios. Each scenario is represented by a state.
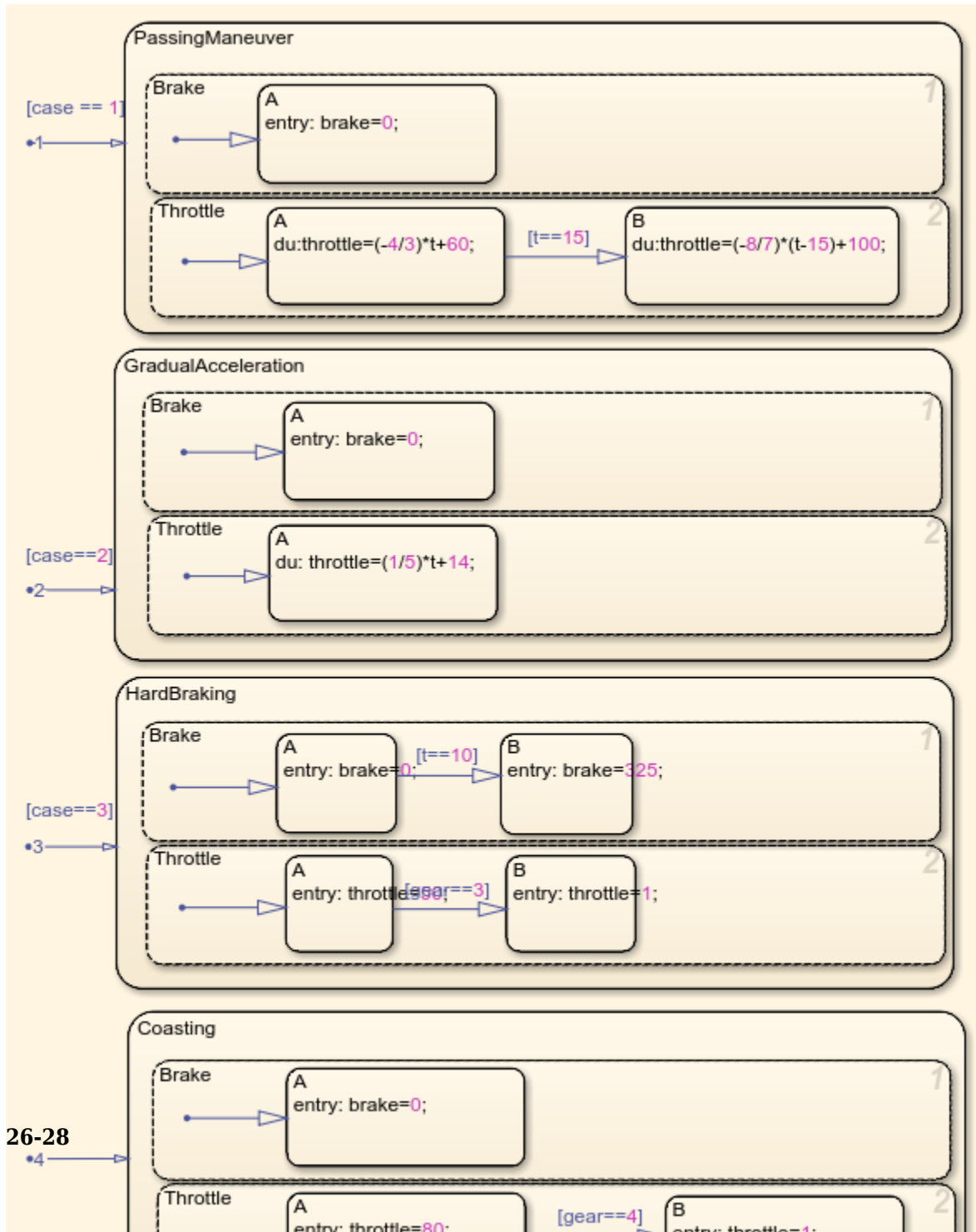
In some of these scenarios, the throttle changes in response to time; in other cases, it responds to gear selection, an output of the Stateflow chart Shift_logic. The Shift_logic chart determines the gear value based on the speed of the vehicle.

Key Behaviors of the Chart and Model

The key behaviors of the test vector chart and model are:

- "Chart Represents Test Cases as States" on page 26-29
- "Chart Uses Conditional Logic to Respond to Dynamic Changes" on page 26-29
- "Model Provides an Interface for Selecting Test Cases" on page 26-29

Chart Represents Test Cases as States

The Dynamic Test Vectors chart represents each test case as an exclusive (OR) state. Each state manipulates brake and throttle values in a unique way, based on the time and gear inputs to the chart.

The chart determines which test to execute from the value of a constant signal `case`, output from the Signal Builder block. Each test case corresponds to a unique signal value.

Chart Uses Conditional Logic to Respond to Dynamic Changes

The Dynamic Test Vectors chart uses conditions on transitions to test time and gear level, and then adjusts brake and throttle accordingly for each driving scenario. Stateflow charts provide many constructs for testing system state and responding to changes, including:

- Conditional logic (see "State Action Types" on page 12-2 and "Transition Action Types" on page 12-7)
- Temporal logic (see "Control Chart Execution by Using Temporal Logic" on page 12-49)
- Change detection operators (see "Detect Changes in Data Values" on page 12-69)
- MATLAB functions (see "Access MATLAB Functions and Workspace Data in C Charts" on page 12-32)

Model Provides an Interface for Selecting Test Cases

The model uses a Signal Builder block to provide an interface for selecting test scenarios to simulate.
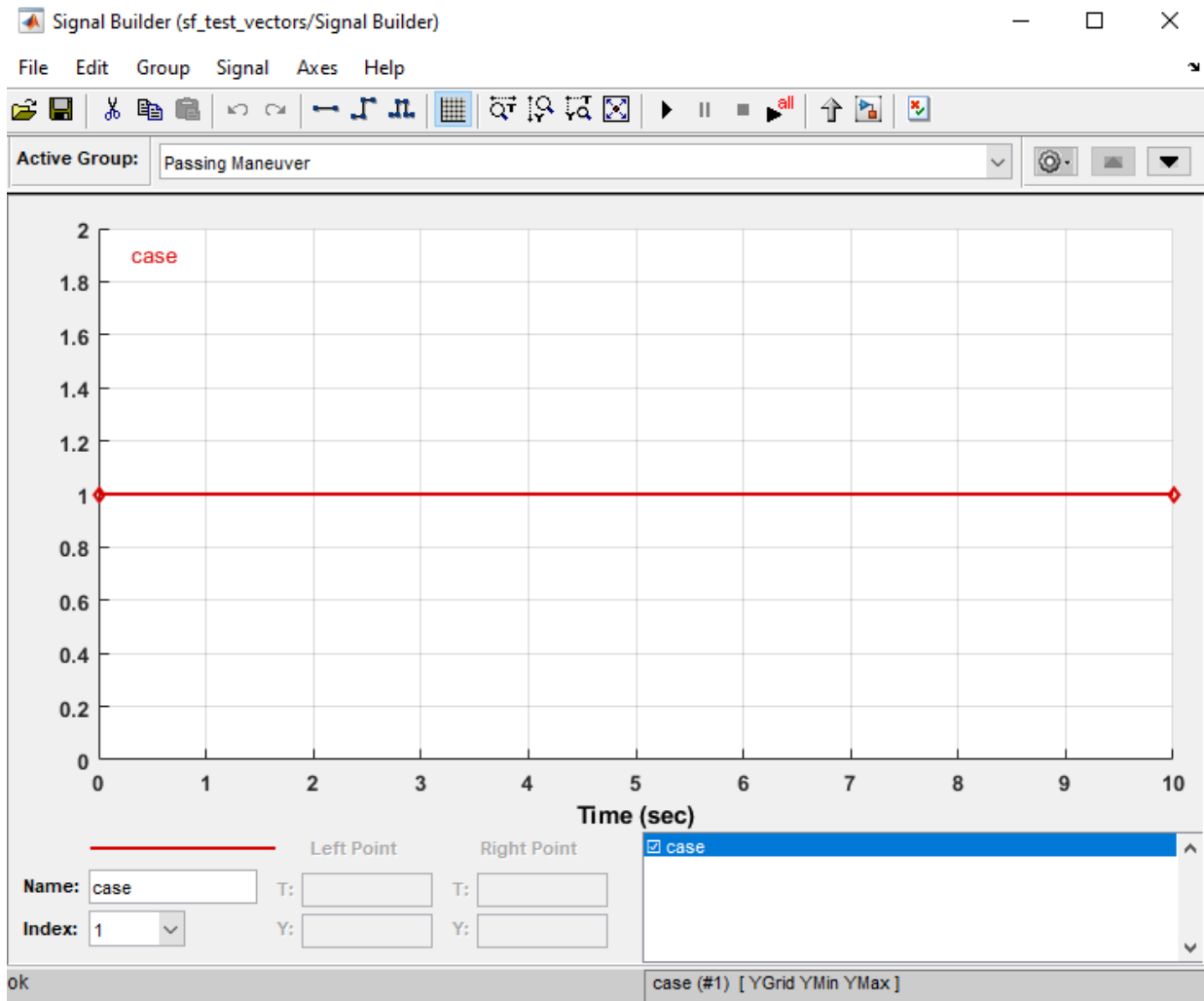
Select and Run Test Cases

In the Signal Builder, select and run test cases as follows:
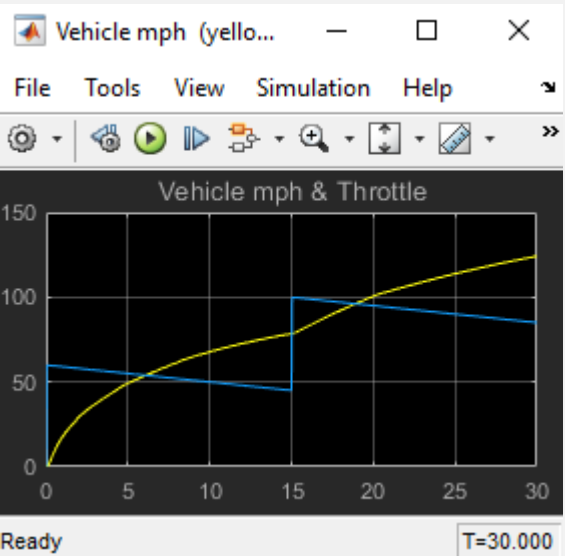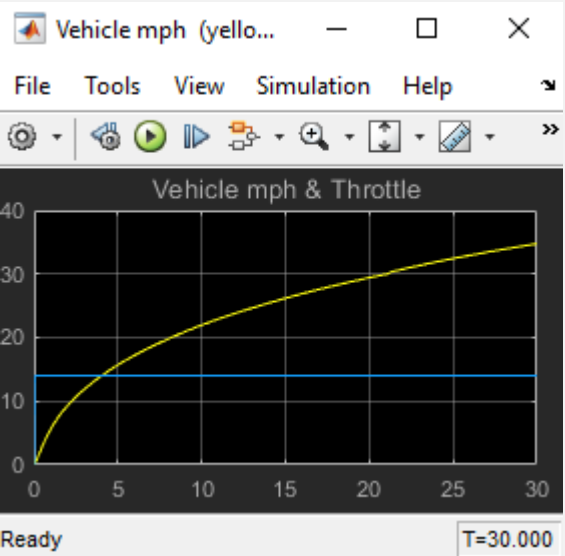
To Test:	Do This:
One case	Click the tab that corresponds to the driving scenario you want to test and click the **Start simulation** button:
All cases and produce a model coverage report (*requires a Simulink Coverage™ software license*)	Click the **Run all and produce coverage** button:
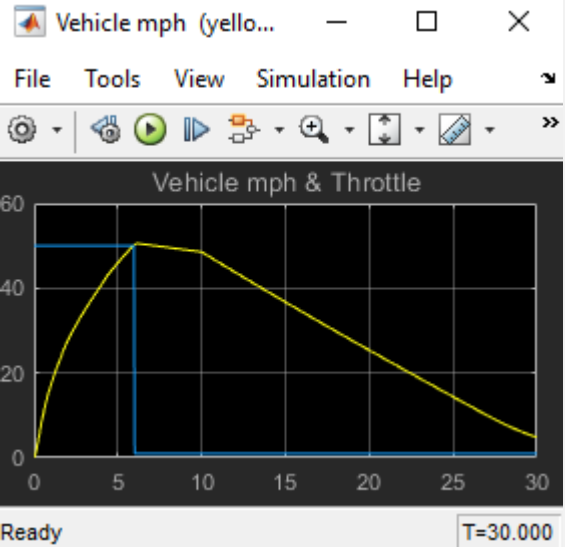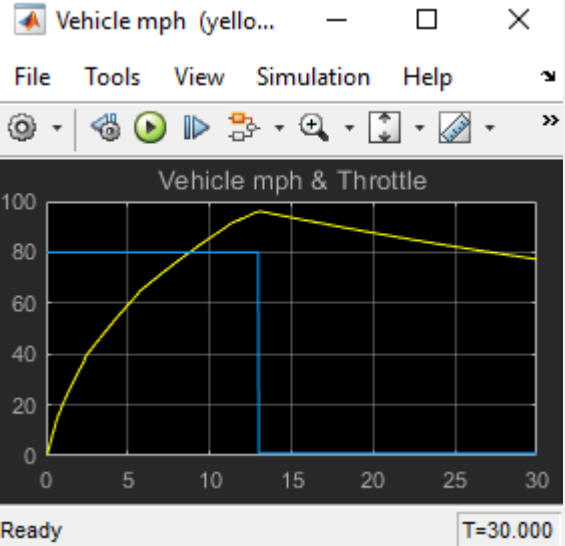
The Signal Builder block sends to the Dynamic Test Vectors chart one or more constant signal values that correspond to the driving scenarios you select. The chart uses these values to activate the appropriate test cases.

Run the Model with Stateflow Test Vectors

1   Open the sf_test_vectors model.
2   Open the Dynamic Test Vectors chart, the Signal Builder block, and the Scope block.
3   Select and simulate a driving scenario from the Signal Builder block, as described in "Select and Run Test Cases" on page 26-30.

   The scope shows the interaction between speed and throttle for the selected scenario.

Driving Scenario	Scope Display	Description
Passing Maneuver		Driver accelerates rapidly. At t = 15 seconds, steps the throttle to 100. With continued heavy throttle, the vehicle accelerates to about 100 MPH and then shifts into overdrive at about t = 21 seconds. The vehicle cruises along in fourth gear for the remainder of the simulation.
Gradual Acceleration		Driver maintains a slow but steady rate of acceleration.

Driving Scenario	Scope Display	Description
Hard Braking		Driver accelerates until the transmission shifts to third gear, then removes foot from the gas pedal. After a short delay, moves foot to the brake pedal and pushes hard.
Coasting		Driver accelerates until transmission shifts to highest gear, then eases up on the gas.

Map Fault Conditions to Actions by Using Truth Tables

Truth tables implement combinatorial logic design in a concise, tabular format. Truth tables are supported only in Stateflow blocks in a Simulink model. For more information, see "Reuse Combinatorial Logic by Defining Truth Tables" on page 27-2.

You can use truth tables to map fault conditions of a system directly to their consequent actions. For example, the model `sf_aircraft` maps the fault conditions and actions using a truth table. For details on this model, see "Fault Detection Control Logic in an Aircraft Elevator Control System".

The fault detection system for the aircraft elevator control system has these requirements.

Condition	Action
Hydraulic pressure 1 failure	While there are no other failures, turn off the left outer actuator.
Hydraulic pressure 2 failure	While there are no other failures, turn off the left inner actuator and the right inner actuator.
Hydraulic pressure 3 failure	While there are no other failures, turn off the right outer actuator.
Actuator position failure	While there are no other failures, isolate that specific actuator.
Hydraulic pressure 1 and left outer actuator failures	While there are no other failures, turn off the left outer actuator
Hydraulic pressure 2 and left inner actuator failures	While there are no other failures, turn off the left inner actuator.
Hydraulic pressure 3 and right outer actuator failures	While there are no other failures, turn off the right outer actuator
Multiple failures on left hydraulics and actuators	Isolate the left outer actuator and the left inner actuator.
Multiple failures on right hydraulics and actuators	Isolate the right outer actuator and the right inner actuator.

Condition	Action
Intermittent actuator failures	If an actuator has been switched on and off five times during operation, isolate that specific actuator.

Logic to satisfy these requirements is constructed using two truth tables in the chart Mode Logic; one for the right elevator (R_switch), and one for the left elevator (L_switch). This truth table is for the left elevator.

Stateflow (truth table) sf_aircraft/Mode Logic.L_switch

File Edit Settings Add Help

Condition Table

	Description	Condition	D1	D2	D3	D4	D5	D6	D7
1	Hydraulic system 1 Low pressure (Left Outer line)	u.low_press[0]	T	T	F	F	-	-	-
2	Left Outer actuator position failed	u.L_pos_fail[0]	-	-	T	T	-	-	-
3	Hydraulic system 2 Low pressure (Inner line)	u.low_press[1]	F	-	F	-	T	-	-
4	Left Inner actuator position failed	u.L_pos_fail[1]	F	-	F	-	-	T	-
		Actions: Specify a row from the Action Table	2	3,5	3	3,5	4	5	Default

Action Table

#	Description	Action
1	Default - All ok, do nothing.	Default:
2	Hydraulic System 1 Failure. Turn off Left Outer Actuator	send(go_off,Actuators.LO);
3	Left Outer Actuator Failure. Isolate Left Outer Actuator	send(go_isolated,Actuators.LO);
4	Hydraulic System 2 Failure. Turn off Left Inner Actuator	send(go_off,Actuators.LI);
5	Left Inner Actuator Failure. Isolate Left Inner Actuator	send(go_isolated,Actuators.LI);

The first requirement indicates that if a failure is only detected in the hydraulic pressure 1 system, turn off the left outer actuator. This requirement is represented in the decision `D1` in the truth table. If there is low pressure in the hydraulic system 1, then `D1` specifies that action 2 is performed. Action 2 sends an event `go_off` to the left actuator, `Actuators.LO`.

Similarly, the other requirements are mapped to the appropriate actions in the truth table. For example, if the left outer actuator fails, `D3` causes action 3. Action 3 sends the event `go_isolated` to `Actuators.LO` to isolate the left actuator.

The truth tables are called at entry(`en`) and during(`du`) actions for the chart so that fault checks execute at each time step.

Design for Isolation and Recovery in a Chart

Mode Logic for the Elevator Actuators

This example shows how the model `sf_aircraft` uses the chart `Mode Logic` to detect system faults and recover from failure modes for an aircraft elevator control system. For more information on this model, see "Fault Detection Control Logic in an Aircraft Elevator Control System".

There are two elevators in the system, each with an outer and inner actuator. The `Actuators` state has a corresponding substate for each of the four actuators. An actuator has five modes: `Passive`, `Active`, `Standby`, `Off`, and `Isolated`. By default, the outer actuators are on, and the inner actuators are on standby. If a fault is detected in the outer actuators, the system responds to maintain stability by turning the outer actuators off and activating the inner actuators.
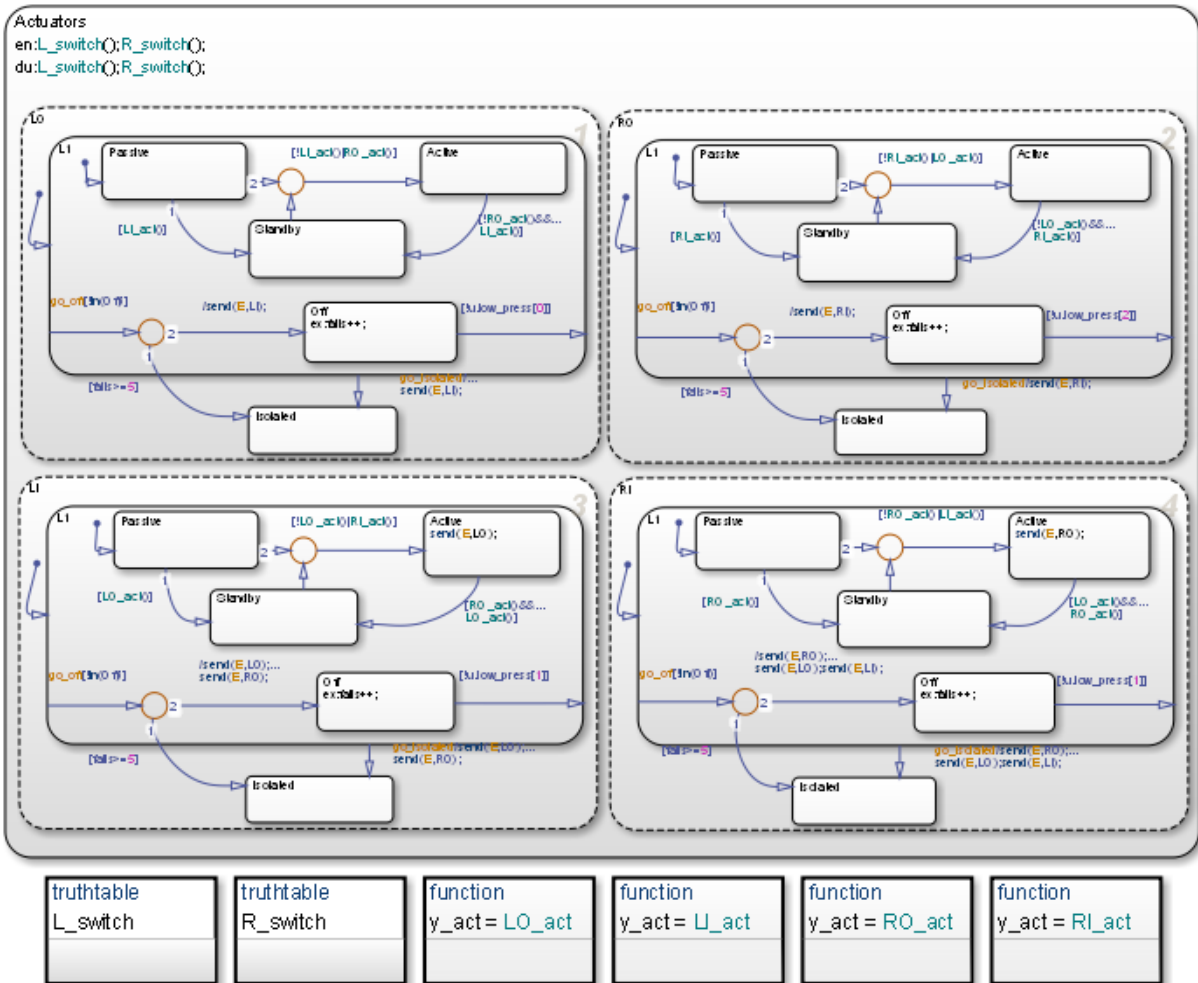
States for Failure and Isolation

Each actuator contains an `Off` state and an `Isolated` state. When the fault detection logic in one of the truth tables detects a failure, it broadcasts the event `go_off` or `go_isolated` to the failing actuator. For more information, see "Map Fault Conditions to Actions by Using Truth Tables" on page 26-34.

The `go_off` event instructs the failing actuator to transition to the `Off` state until the condition is resolved. The event `go_isolated` causes the failing actuator to transition to `Isolated`. Transitions to the `Isolated` state are from the superstate `L1`, which contains all the other operating modes. This state has no outgoing transitions, so that once an actuator has entered `Isolated` it remains there. Intermittent failures that cause an actuator to fail 5 or more times, also cause a transition to `Isolated`. The variable `fails` logs the number of failures for an actuator by incrementing each time a transition occurs out of `Off`.

Transitions for Recovery

Transitions in the substates for each actuator account for recovery requirements of the elevator system. These requirements derive from rules for symmetry and safety of the elevators such as:

- Only one actuator for an elevator must be active at one time.
- Outer actuators have priority over the inner actuators.
- Actuator activity should be symmetric if possible.
- Switching between actuators must be kept to a minimum.

For example, one requirement of the system is if one outer actuator fails, then the other outer actuator must move to standby and the inner actuators take over. Consequently, there is a transition from each `Active` state to `Standby`, and vice versa.
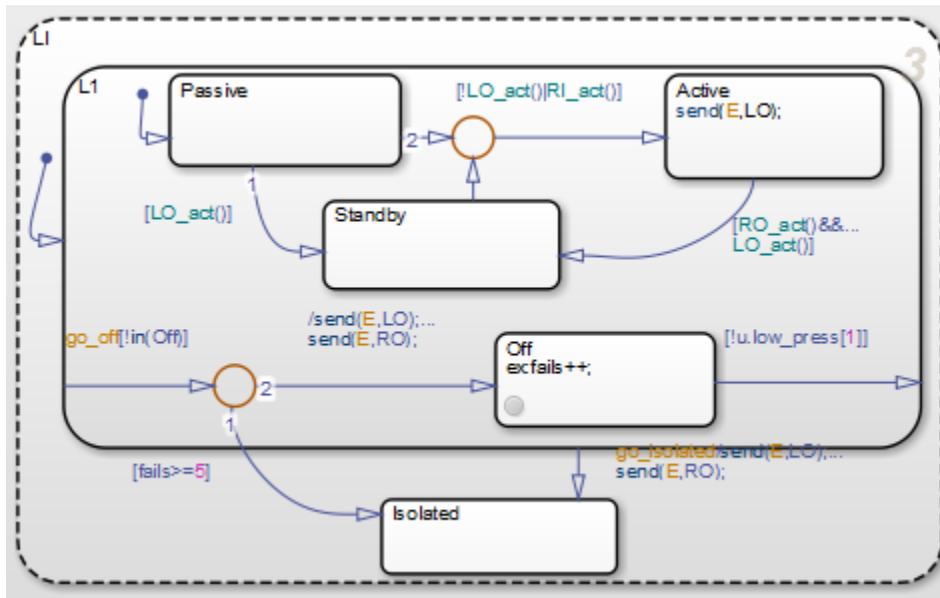
For the inner left actuator (`LI`), the transition to `Active` inside the `L1` superstate is conditionally based on `[!LO_act()|RI_act()]`. This causes the left inner actuator to turn on if the outer actuator (`LO`) has failed, or the right inner actuator (`RI`) has turned on.
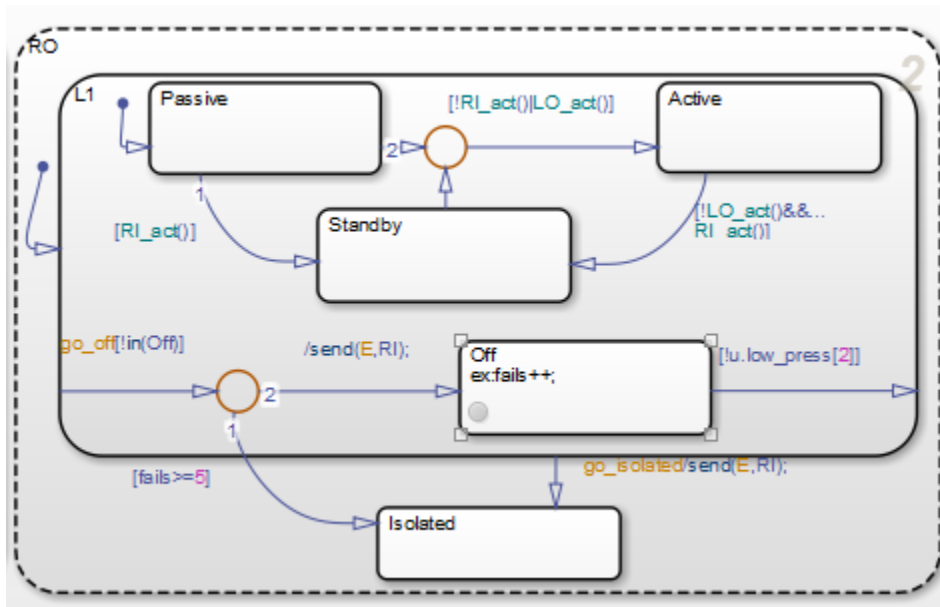
Another consequence if `L0` fails and moves out of `Active` is a transition that occurs in the right outer actuator (`R0`). The `R0` state transitions inside the `L1` superstate from `Active` to `Standby`. This satisfies the requirement of the outer actuators and inner actuators to work in symmetry.

27

Truth Table Functions for Decision-Making Logic

Reuse Combinatorial Logic by Defining Truth Tables

Truth tables implement combinatorial logic design in a concise, tabular format. Truth tables are supported only in Stateflow blocks in a Simulink model. Typical applications for truth tables include decision making for:

- Fault detection and management
- Mode switching

You can add a Truth Table block directly to your Simulink model. You can also define a truth table function in a Stateflow chart, state, or subchart. The location of the function determines the set of states and transitions that can call the function.

- If you want to call the function only within one state or subchart and its substates, put your truth table function in that state or subchart. That function overrides any other functions of the same name in the parents and ancestors of that state or subchart.
- If you want to call the function anywhere in that chart, put your truth table function at the chart level.
- If you want to call the function from any chart in your model, put your truth table at the chart level and enable exporting of chart-level functions. For more information, see "Export Stateflow Functions for Reuse" on page 8-22.

For example, this truth table function has the name `ttable`. It takes three arguments (`x`, `y`, and `z`) and returns one output value (`r`).

```
truthtable
r = ttable(x,y,z)
```
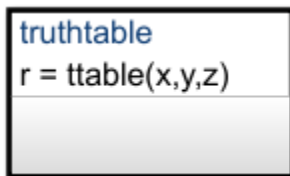
The function consists of this arrangement of conditions, decisions, and actions.

Condition	Decision 1	Decision 2	Decision 3	Default Decision
x == 1	T	F	F	-
y == 1	F	T	F	-

Condition	Decision 1	Decision 2	Decision 3	Default Decision
z == 1	F	F	T	-
Action	r = 1	r = 2	r = 3	r = 4

Each of the conditions entered in the **Condition** column must evaluate to true (nonzero value) or false (zero value). Outcomes for each condition are specified as T (true), F (false), or - (true or false). Each of the decision columns combines an outcome for each condition with a logical AND into a compound condition, which is referred to as a decision.

You evaluate a truth table one decision at a time, starting with **Decision 1**. The **Default Decision** covers all possible remaining decisions. If one of the decisions is true, you perform its action, and then the truth table execution is complete.

For example, if conditions $x == 1$ and $y == 1$ are false and condition $z == 1$ is true, then **Decision 3** is true and the variable r is set equal to 3. The remaining decisions are not tested and evaluation of the truth table is finished. If the first three decisions are false, then the **Default Decision** is automatically true and its action ($r=4$) is executed. This table lists pseudocode corresponding to the evaluation of this truth table example.

Pseudocode	Description
`if ((x == 1) & !(y == 1) & !(z == 1))` `    r = 1;`	If **Decision 1** is true, then set r=1.
`elseif (!(x == 1) & (y == 1) & !(z == 1))` `    r = 2;`	If **Decision 2** is true, then set r=2.
`elseif (!(x == 1) & !(y == 1) & (z == 1))` `    r = 3;`	If **Decision 3** is true, then set r=3.

`else`
`    r = 4;`
`endif`
	If all other decisions are false, then **Default Decision** is true. Set r=4.

Define a Truth Table Function

1
    In the object palette, click the truth table function icon ⊞ . Move your pointer to the location for the new truth table function in your chart.

2   Enter the signature label for the function, as described in "Declare Function Arguments and Return Values" on page 27-4.

3 Program the truth table function. For more information, see "Program a Truth Table" on page 27-11.

4 In the Model Explorer, expand the chart object and select the truth table function. The arguments and return values of the function signature appear as data items that belong to your function. Arguments have the scope `Input`. Return values have the scope `Output`.

5 In the Data properties dialog box for each argument and return value, specify the data properties, as described in "Set Data Properties" on page 9-5.

6 Create any additional data items required by your function. For more information, see "Add Data Through the Model Explorer" on page 9-3.

Your function can access its own data or data belonging to parent states or the chart. The data items in the function can have one of these scopes:

- `Local` — Local data persists from one function call to the next function call. Valid for C charts only.

- `Constant` — Constant data retains its initial value through all function calls.

- `Parameter` — Parameter data retains its initial value through all function calls.

- `Temporary` — Temporary data initializes at the start of every function call. Valid for C charts only.

In charts that use C as the action language, define temporary data when you want to use data that is only valid while a function executes. For example, you can designate a loop counter to have `Temporary` scope if the counter value does not need to persist after the function completes.

In charts that use MATLAB as the action language, you do not need to define temporary function data. If you use an undefined variable, Stateflow creates a temporary variable. The variable is available to the rest of the function.

You can initialize your function data (other than arguments and return values) from the MATLAB workspace. For more information, see "Initialize Data from the MATLAB Base Workspace" on page 9-24.

Declare Function Arguments and Return Values

The function signature label specifies a name for your function and the formal names for its arguments and return values. A signature label has this syntax:

```
[return_val1, return_val2,...] = function_name(arg1, arg2,...)
```

You can specify multiple return values and multiple input arguments. Each return value and input argument can be a scalar, vector, or matrix of values. For functions with only one return value, omit the brackets in the signature label.

You can use the same variable name for both arguments and return values. For example, a function with this signature label uses the variables y1 and y2 as both inputs and outputs:

```
[y1, y2, y3] = f(y1, u, y2)
```

If you export this function to C code, y1 and y2 are passed by reference (as pointers), and u is passed by value. Passing inputs by reference reduces the number of times that the generated code copies intermediate data, resulting in more optimal code.

Call Truth Table Functions in States and Transitions

You can call truth table functions from the actions of any state or transition. You can also call truth table functions from other functions. If you export a truth table function, you can call it from any chart in the model.

The syntax for a call to a truth table function is the same as the function signature, with actual arguments replacing the formal ones specified in a signature. If the data types of an actual and formal argument differ, a function casts the actual argument to the type of the formal argument.

Tip If the formal arguments of a function signature are scalars, verify that inputs and outputs of function calls follow the rules of scalar expansion. For more information, see "How Scalar Expansion Works for Functions" on page 17-9.

Specify Properties of Truth Table Functions

You can set general properties for your truth table function through its properties dialog box. To open the function properties dialog box, right-click the truth table function box and select **Properties** from the context menu.

Name

Function name. Click the function name link to bring your function to the foreground in its native chart.

Function Inline Option

Controls the inlining of your function in generated code:

- `Auto` — Determines whether to inline your function based on an internal calculation.
- `Inline` — Inlines your function if you do not export it to other charts and it is not part of a recursion. (A recursion exists if your function calls itself directly or indirectly through another function call.)
- `Function` — Does not inline your function.

Label

Signature label for your function. For more information, see "Declare Function Arguments and Return Values" on page 27-4.

Underspecification

Controls the level of diagnostics for underspecification in your truth table function. For more information, see "Correct Overspecified and Underspecified Truth Tables" on page 27-44.

Overspecification

Controls the level of diagnostics for overspecification in your truth table function. For more information, see "Correct Overspecified and Underspecified Truth Tables" on page 27-44.

Action Language

Controls the action language for your Stateflow truth table function. Choose between MATLAB or C. For more information, see "Language Options for Stateflow Truth Tables" on page 27-8.

Description

Function description. You can enter brief descriptions of functions in the hierarchy.

Document Link

Link to online documentation for the function. You can enter a web URL address or a MATLAB command that displays documentation in a suitable online format, such as an HTML file or text in the MATLAB Command Window. When you click the **Document link** hyperlink, Stateflow displays the documentation.

See Also

More About

- "Represent Combinatorial Logic Using Truth Tables" on page 27-10
- "Program a Truth Table" on page 27-11
- "Language Options for Stateflow Truth Tables" on page 27-8
- "Reusable Functions in Charts" on page 2-46
- "Export Stateflow Functions for Reuse" on page 8-22
- "Reuse Functions by Using Atomic Boxes" on page 8-35

Language Options for Stateflow Truth Tables

Truth tables implement combinatorial logic design in a concise, tabular format. Truth tables are supported only in Stateflow blocks in a Simulink model. For more information, see "Reuse Combinatorial Logic by Defining Truth Tables" on page 27-2.

C Truth Tables

Using C truth tables, you can specify conditions and actions with C as the action language. C truth tables support basic C constructs and provide access to MATLAB functions by using the `ml` namespace operator or `ml` function. To use C as the action language for your truth table, it must be inside a Stateflow C action language chart.

MATLAB Truth Tables

Truth Table blocks and truth tables inside charts that use MATLAB as the action language are MATLAB truth tables. In these MATLAB truth tables, you cannot specify C as the action language. You can specify conditions and actions in MATLAB truth tables, which provides optimizations for simulation and code generation.

MATLAB truth tables offer several advantages over C truth tables:

- MATLAB as the action language provides a richer syntax for specifying control flow logic in truth table actions. It provides `for` loops, `while` loops, nested `if` statements, and `switch` statements.
- You can call MATLAB functions directly in truth table actions. Also, you can call library functions (for example, MATLAB `sin` and `fft` functions) and generate code for these functions by using Simulink Codersoftware.
- You can create temporary or persistent variables during simulation or in code directly without having to define them in the Model Explorer.
- You have access to better debugging tools. You can set breakpoints on lines of code, step through code, and watch data values tool tips.
- You can use persistent variables in truth table actions. You can define data that persists across multiple calls to the truth table function during simulation.

Select a Language for Stateflow Truth Tables

If the truth table is inside a C action language Stateflow chart, you can specify an action language for your Stateflow truth table by using the **Property Inspector**.

1 Double-click the truth table.
2 Open the Property Inspector by selecting **View** > **Property Inspector**.
3 Under the **Properties** section, select **C** or **MATLAB** as the Action Language.

Note If you do not have the option to change the action language, your truth table is a MATLAB truth table.

Migration from C to MATLAB Truth Tables

When you migrate from a C truth table to a MATLAB truth table, you must verify that the code to program the actions conforms to MATLAB syntax. Between the two action languages, these differences exist.

Action language	Indices	Expression for *not equal to*
MATLAB	One-based	~=
C	Zero-based	!=

Represent Combinatorial Logic Using Truth Tables

Here is the recommended workflow for using truth tables in Simulink models. For more information, see "Reuse Combinatorial Logic by Defining Truth Tables" on page 27-2.

Step	Task	Reference
1	Add a truth table to your Simulink model.	"Define a Truth Table Function" on page 27-3
2	Specify properties of the truth table function.	"Specify Properties of Truth Table Functions" on page 27-5
3	Select an action language and program the conditions and actions in the truth table.	"Program a Truth Table" on page 27-11
4	Debug the truth table for syntax errors and for errors during simulation.	"Debug Run-Time Errors in a Truth Table" on page 27-33
5	Simulate the model and check the generated content for the truth tables.	"View Generated Content for Stateflow Truth Tables" on page 27-55

Program a Truth Table

Truth tables implement combinatorial logic design in a concise, tabular format. Truth tables are supported only in Stateflow blocks in a Simulink model. For more information, see "Reuse Combinatorial Logic by Defining Truth Tables" on page 27-2.

Open a Truth Table for Editing

After you create and label a truth table in a chart, you specify its logical behavior. To open the truth table, double-click the truth table function.
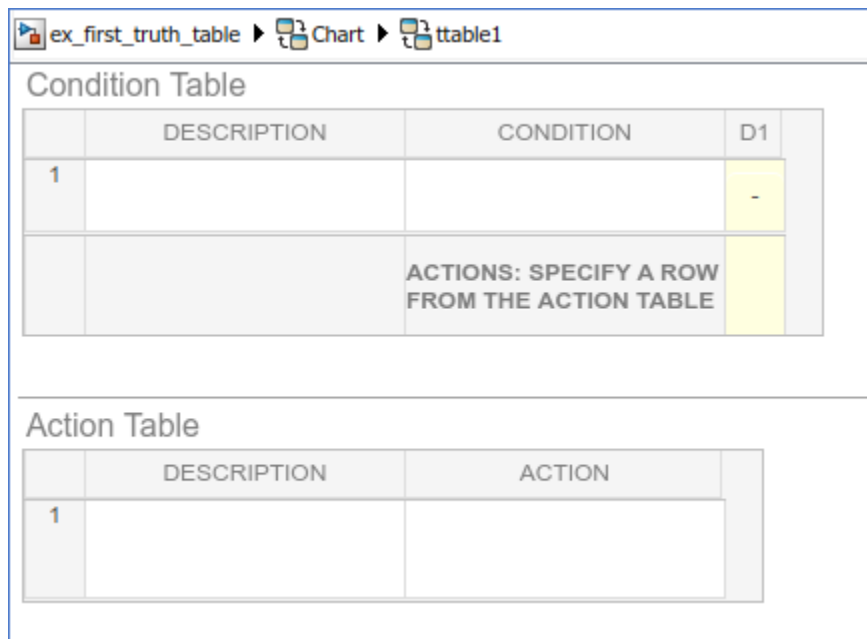
By default, a truth table contains a **Condition Table** and an **Action Table**, each with one row. The **Condition Table** contains a single decision column, **D1**, and a single action row.

Select an Action Language

If the truth table is inside a C action language Stateflow chart, you can specify the action language for your Stateflow truth table:

1   Open the Property Inspector by selecting **View** > **Property Inspector**.
2   Under the **Properties** section, select **C** or **MATLAB** as the Action Language.

Enter Truth Table Conditions

Conditions are the starting point for specifying logical behavior in a truth table. You open the truth table `ttable` for editing. You start programming the behavior of `ttable` by specifying conditions.

You enter conditions in the **Condition** column of the **Condition Table**. For each condition that you enter, you can enter an optional description in the **Description** column. To enter conditions for the truth table `ttable`:

1   Click the row on the **Condition Table** that you want to append.
2
    Click the **Append Row** button [icon] on the side panel twice.

    The truth table appends two rows to the bottom of the **Condition Table**.
3   Click and drag the bar that separates the **Condition Table** and the **Action Table** panes down to enlarge the **Condition Table** pane.
4   In the **Condition Table**, click the top cell of the **Description** column.

    A flashing text cursor appears in the cell, which appears highlighted.
5   Enter this text:

    `x is equal to 1`

    Condition descriptions are optional, but appear as comments in the generated code for the truth table.
6   To select the next cell on the right in the **Condition** column, press the right arrow.
7   In the first cell of the **Condition** column, enter:

    `XEQ1:`

    This text is an optional label that you can include with the condition. Each label must begin with an alphabetic character (`[a-z][A-Z]`) followed by any number of alphanumeric characters (`[a-z][A-Z][0-9]`) or an underscore (`_`).
8   Press **Enter** and this text:

    `x == 1`

This text is the actual condition. Each condition that you enter must evaluate to zero (false) or nonzero (true). You can use optional brackets in the condition (for example, `[x == 1]`).

In truth table conditions, you can use data that passes to the truth table function through its arguments. The preceding condition tests whether the argument x is equal to 1. You can also use data defined for parent objects of the truth table, including the chart.

9 Repeat the preceding steps to enter the other two conditions.

Enter Truth Table Decisions

Each decision column (**D1**, **D2**, and so on) binds a group of condition outcomes together with an AND relationship into a decision. The possible values for condition outcomes in a decision are T (true), F (false), and - (true or false). In "Enter Truth Table Conditions" on page 27-12, you entered conditions for the truth table `ttable`. Continue by entering values in the decision columns:

1   Click the column **Condition Table** that you want to append.

2   Click the **Append Column** button on the side panel twice.

3   Click the top cell in decision column **D1**.

    A flashing text cursor appears in the cell, which appears highlighted.

4   Press the space bar until a value of T appears.

    Pressing the space bar toggles through the possible values of F, T, and -. You can also enter these characters directly. Pressing 1 sets the value to T, while pressing 0 sets the value to F. Pressing x sets the value to -.

5   Press the down arrow key to advance to the next cell down in the **D1** column.

    In the decision columns, you can use the arrow keys to advance to another cell in any direction. You can also use the right and left arrow keys to advance left or right in these cells.

6   Enter the remaining values for the decision columns:

During execution of the truth table, decision testing occurs in left-to-right order. The order of testing for individual condition outcomes within a decision is undefined. Truth tables evaluate the conditions for each decision in top-down order (first condition 1, then condition 2, and so on). Because this implementation is subject to change in the future, do not rely on a specific evaluation order.

The Default Decision Column

The last decision column in `ttable`, **D4**, is the default decision for this truth table. The default decision covers any decisions not tested for in preceding decision columns to the

left. You enter a default decision as the last decision column on the right with an entry of - for all conditions in the decision. This entry represents any outcome for the condition, T or F.

In the preceding example, the default decision column, **D4**, specifies these decisions:

Condition	Decision 4	Decision 5	Decision 6	Decision 7	Decision 8
x == 1	F	T	F	T	T
y == 1	F	F	T	T	T
z == 1	F	T	T	F	T

Tip The default decision column must be the last column on the right in the **Condition Table**.

Enter Truth Table Actions

During execution of the truth table, decision testing occurs in left-to-right order. When a decision match occurs, the action in the **Action Table** specified in the **Actions** row for that decision column executes. Then the truth table exits.

In "Enter Truth Table Decisions" on page 27-14, you entered decisions in the truth table. The next step is to enter actions you want to occur for each decision in the **Action Table**. Later, you assign these actions to their decisions in the **Actions** row of the **Condition Table**.

This section describes how to program truth table actions with these topics:

- "Set Up the Action Table" on page 27-17 — Shows you how to set up the Action Table in truth table `ttable`.
- "Program Actions Using C Expressions" on page 27-18 — Provides sample code to program actions in `ttable`. Follow this section if you selected **C** as the language for this truth table.
- "Program Actions Using MATLAB Expressions" on page 27-21 — Provides sample MATLAB code to program actions in `ttable`. Follow this section if you selected **MATLAB** as the language for this truth table.

Set Up the Action Table

1   Click the row **Action Table** that you want to append.

2
Click the **Append Row** button on the side panel three times.

3 Program the actions using the language you selected for the truth table.

If you selected...	Use this procedure...
C	"Program Actions Using C Expressions" on page 27-18
MATLAB	"Program Actions Using MATLAB Expressions" on page 27-21

Program Actions Using C Expressions

Follow this procedure to program your actions using C as the action language:

1 Click the top cell in the **Description** column of the **Action Table**.

A flashing text cursor appears in the cell, which appears highlighted.

2 Enter the following description:

```
set r to 1
```

Action descriptions are optional, but appear as comments in the generated code for the truth table.

3 Press the right arrow key to select the next cell on the right, in the **Action** column.

4 Enter the following text:

```
A1:
```

You begin an action with an optional label followed by a colon (`:`). Later, you enter these labels in the **Actions** row of the **Condition Table** to specify an action for each decision column. Like condition labels, action labels must begin with an alphabetic character (`[a-z][A-Z]`) followed by any number of alphanumeric characters (`[a-z][A-Z][0-9]`) or an underscore (`_`).

5 Press **Enter** and enter the following text:

```
r=1;
```

In truth table actions, you can use data that passes to the truth table function through its arguments and return value. The preceding action, `r=1`, sets the value of the return value `r`. You can also specify actions with data defined for a parent object of the truth table, including the chart. Truth table actions can also broadcast or send events that are defined for the truth table, or for a parent, such as the chart itself.

> **Tip** If you omit the semicolon at the end of an action, the result of the action echoes to the MATLAB Command Window when the action executes during simulation. Use this echoing option as a debugging tool.

6 Enter the remaining actions in the **Action Table**, as shown:

ex_first_truth_table ▸ Chart ▸ ttable

Condition Table

	DESCRIPTION	CONDITION	D1	D2	D3	D4
1	x is equal to 1	XEQ1: x == 1	T	F	F	-
2	y is equal to 1	YEQ1: y == 1	F	T	F	-
3	z is equal to 1	ZEQ1: z == 1	F	F	T	-
		ACTIONS: SPECIFY A ROW FROM THE ACTION TABLE				

Action Table

	DESCRIPTION	ACTION
1	set r to 1	A1: r=1;
2	set r to 2	A2: r=2;
3	set r to 3	A3: r=3;
4	set r to 4	A4: r=4;

Now you are ready to assign actions to decisions, as described in "Assign Truth Table Actions to Decisions" on page 27-24.

Program Actions Using MATLAB Expressions

If you selected MATLAB as the action language, you can write MATLAB code to program your actions. Using this code, you can add control flow logic and call MATLAB functions directly. In the following procedure, you program an action in the truth table `ttable`, using the following features of MATLAB syntax:

- Persistent variables
- `if ... else ... end` control flows
- `for` loop

Follow these steps:

1 Click the top cell in the **Description** column of the **Action Table**.

A flashing text cursor appears in the cell, which appears highlighted.

2 Enter this description:

```
Maintain a counter and a circular
vector of values of length 6.
Every time this action is called,
output r takes the next value of
the vector.
```

Action descriptions are optional, but appear as comments in the generated code for the truth table.

3 Press the right arrow key to select the next cell on the right, in the **Action** column.

4 Enter the following text:

```
A1:
```

You begin an action with an optional label followed by a colon (`:`). Later, you enter these labels in the **Actions** row of the **Condition Table** to specify an action for each decision column. Like condition labels, action labels must begin with an alphabetic character (`[a-z][A-Z]`) followed by any number of alphanumeric characters (`[a-z][A-Z][0-9]`) or an underscore (`_`).

5 Press **Enter** and enter the following text:

```
persistent values counter;
cycle = 6;
```

27-21

```
coder.extrinsic('plot');

if isempty(counter)
   % Initialize counter to be zero
   counter = 0;
else
   % Otherwise, increment counter
   counter = counter + 1;
end

if isempty(values)
   % Values is a vector of 1 to cycle
   values = zeros(1, cycle);
   for i = 1:cycle
      values(i) = i;
   end

   % For debugging purposes, call the MATLAB
   % function "plot" to show values
   plot(values);
end

% Output r takes the next value in values vector
r = values( mod(counter, cycle) + 1);
```

In truth table actions, you can use data that passes to the truth table function through its arguments and return value. The preceding action sets the return value r equal to the next value of the vector `values`. You can also specify actions with data defined for a parent object of the truth table, including the chart. Truth table actions can also broadcast or send events that are defined for the truth table, or for a parent, such as the chart itself.

Note If you omit the semicolon at the end of an action, the result of the action echoes to the MATLAB Command Window when the action executes during simulation. Use this echoing option as a debugging tool.

6 Enter the remaining actions in the **Action Table**, as shown:

ex_first_truth_table ▶ Chart ▶ ttable ▼

Condition Table

	DESCRIPTION	CONDITION	D1	D2	D3	D4
1	x is equal to 1	XEQ1: x == 1	T	F	F	-
2	y is equal to 1	YEQ1: y == 1	F	T	F	-
3	z is equal to 1	ZEQ1: z == 1	F	F	T	-
		ACTIONS: SPECIFY A ROW FROM THE ACTION TABLE				

Action Table

	DESCRIPTION	ACTION
1	Maintain a counter and a circular vector of values of length 6. Every time this	A1: persistent values counter; cycle = 6;
2	set r to 1	A2: r=1;
3	set r to 2	A3: r=2;
4	set r to 3	A4: r=3;

If action `A1` executes during simulation, a plot of the `values` vector appears:
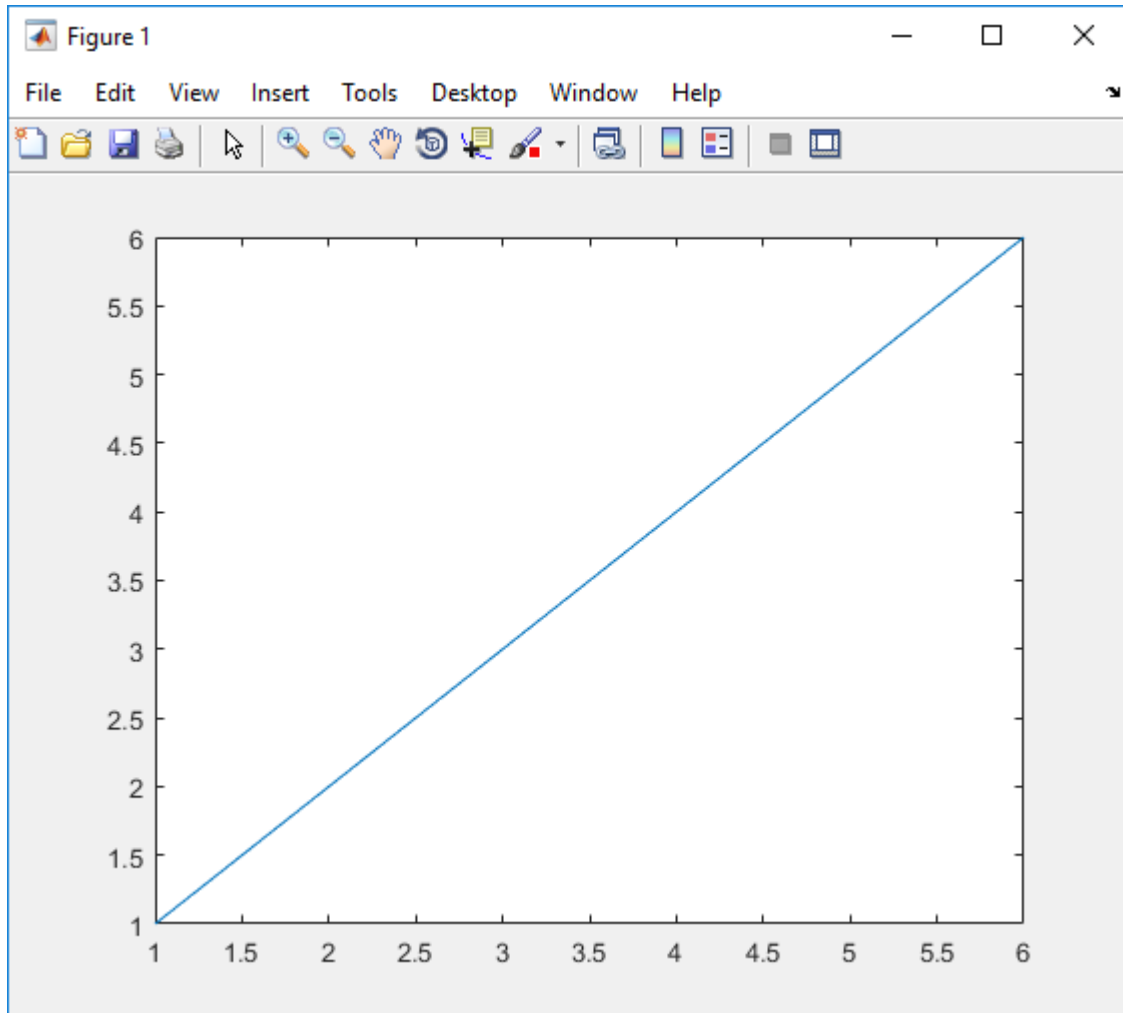
Now you are ready to assign actions to decision.

Assign Truth Table Actions to Decisions

You must assign at least one action from the **Action Table** to each decision in the **Condition Table**. The truth table uses this association to determine what action to execute when a decision tests as true.

Rules for Assigning Actions to Decisions

The following rules apply when you assign actions to decisions in a truth table:

- You specify actions for decisions by entering a row number or a label in the **Actions** row cell of a decision column.

  If you use a label specifier, the label must appear with the action in the **Action Table**.

- You must specify at least one action for each decision.

  Actions for decisions are not optional. Each decision must have at least one action specifier that points to an action in the **Action Table**. If you want to specify no action for a decision, specify a row that contains no action statements.

- You can specify multiple actions for a decision with multiple specifiers separated by a comma, semicolon, or space.

  For example, for the decision column **D1**, you can specify `A1,A2,A3` or `1;2;3` to execute the first three actions when decision **D1** is true.

- You can mix row number and label action specifiers interchangeably in any order.

  The following example uses both row and label action specifiers.

ex_first_truth_table ▶ ⊞ Chart ▶ ⊞ ttable   ▼

Condition Table

	DESCRIPTION	CONDITION	D1	D2	D3	D4
1	x is equal to 1	XEQ1: x == 1	T	F	F	-
2	y is equal to 1	YEQ1: y == 1	F	T	F	-
3	z is equal to 1	ZEQ1: z == 1	F	F	T	-
		ACTIONS: SPECIFY A ROW FROM THE ACTION TABLE	A1	2	A3	4

Action Table

	DESCRIPTION	ACTION
1	set r to 1	A1: r=1;
2	set r to 2	A2: r=2;
3	set r to 3	A3: r=3;
4	set r to 4	A4: r=4;

- You can specify the same action for more than one decision, as shown:

ex_first_truth_table ▸ Chart ▸ ttable

Condition Table

	DESCRIPTION	CONDITION	D1	D2	D3	D4
1	x is equal to 1	XEQ1: x == 1	T	F	F	-
2	y is equal to 1	YEQ1: y == 1	F	T	F	-
3	z is equal to 1	ZEQ1: z == 1	F	F	T	-
		ACTIONS: SPECIFY A ROW FROM THE ACTION TABLE	A1	1	A2	2

Action Table

	DESCRIPTION	ACTION
1	set r to 1	A1: r=1;
2	set r to 2	A2: r=2;

- Row number action specifiers in the **Actions** row of the **Condition Table** automatically adjust to changes in the row order of the **Action Table**.

How to Assign Actions to Decisions

This section describes how to assign actions to decisions in the truth table `ttable`. In this example, the **Actions** row cell for each decision column contains a label specified for each action in the **Action Table**. Follow these steps:

1   Click the bottom cell in decision column **D1**, the first cell of the **Actions** row of the **Condition Table**.

2   Enter the action specifier A1 for decision column **D1**.

When **D1** is true, action A1 in the **Action Table** executes.

3   Enter the action specifiers for the remaining decision columns:

ex_first_truth_table ▶ Chart ▶ ttable ▼

Condition Table

	DESCRIPTION	CONDITION	D1	D2	D3	D4
1	x is equal to 1	XEQ1: x == 1	T	F	F	-
2	y is equal to 1	YEQ1: y == 1	F	T	F	-
3	z is equal to 1	ZEQ1: z == 1	F	F	T	-
		ACTIONS: SPECIFY A ROW FROM THE ACTION TABLE	**A1**	**A2**	**A3**	**A4**

Action Table

	DESCRIPTION	ACTION
1	set r to 1	A1: r=1;
2	set r to 2	A2: r=2;
3	set r to 3	A3: r=3;
4	set r to 4	A4: r=4;

Now you are ready to perform the final step in programming a truth table.

Add Initial and Final Actions

In addition to actions for decisions, you can add initial and final actions to the truth table function. Initial actions specify an action that executes before any decision testing occurs. Final actions specify an action that executes as the last action before the truth table exits. To specify initial and final actions for a truth table, use the action labels `INIT` and `FINAL` in the **Action Table**.

Use this procedure to add initial and final actions that display diagnostic messages in the MATLAB Command Window before and after execution of the truth table `ttable`:

1 In the truth table, right-click row 1 of the **Action Table** and select **Insert Row**.

A blank row appears at the top of the **Action Table**.

2 Select **Edit > Append Row**.

A blank row appears at the bottom of the **Action Table**.

3 Click and drag the bottom border of the truth table to show all six rows of the **Action Table**:

Condition Table

	DESCRIPTION	CONDITION	D1	D2	D3	D4
1	x is equal to 1	XEQ1: x == 1	T	F	F	-
2	y is equal to 1	YEQ1: y == 1	F	T	F	-
3	z is equal to 1	ZEQ1: z == 1	F	F	T	-
		ACTIONS: SPECIFY A ROW FROM THE ACTION TABLE	A1	A2	A3	A4

Action Table

	DESCRIPTION	ACTION
1		
2	set r to 1	A1: r=1;
3	set r to 2	A2: r=2;
4	set r to 3	A3: r=3;
5	set r to 4	A4: r=4;
6		

4 Add the initial action in row 1 as follows:

Truth Table Type	Description	Action

C	Initial action:
	
	Display message
		`INIT:`
		
		`ml.disp('truth table ttable entered');`

MATLAB	Initial action:
	
	Display message
		`INIT:`
		
		`coder.extrinsic('disp');`
		
		`disp('truth table ttable entered');`

5 Add the final action in row 6 as follows:

Truth Table Type	Description	Action

C	Final action:
	
	Display message
		`FINAL:`
		
		`ml.disp('truth table ttable exited');`

MATLAB	Final action:
	
	Display message
		`FINAL:`
		
		`coder.extrinsic('disp');`
		
		`disp('truth table ttable exited');`

Although the initial and final actions for the preceding truth table example appear in the first and last rows of the **Action Table**, you can enter these actions in any row. You can also assign initial and final actions to decisions by using the action specifier `INIT` or `FINAL` in the **Actions** row of the **Condition Table**.

Debug Run-Time Errors in a Truth Table

Truth tables implement combinatorial logic design in a concise, tabular format. Truth tables are supported only in Stateflow blocks in a Simulink model. For more information, see "Reuse Combinatorial Logic by Defining Truth Tables" on page 27-2.

Check Truth Tables for Errors

Once you completely specify your truth tables, you begin the process of debugging them. The first step is to run diagnostics to check truth tables for syntax errors including overspecification and underspecification, as described in "Correct Overspecified and Underspecified Truth Tables" on page 27-44.

To check for syntax errors:

1   Double-click the truth table.

2
     In the truth table, select **Run Diagnostics** .

     For example, if you change the action for decision column **D4** to an action that does not exist, you get an error message in the Diagnostic Viewer.

Truth table diagnostics run automatically when you start simulation of the model with a new or modified truth table. If no errors exist, the diagnostic window does not appear and simulation starts immediately.

Debug a Truth Table During Simulation

You can use breakpoints in a Stateflow truth table to pause simulation and debug your logic. Once a breakpoint causes the simulation to pause, you can step through the actions and examine the data values at that specific point in the simulation.

With truth tables, you can set these different breakpoint types:

- Condition tested
- Decision tested
- Decision valid
- Action executed

After simulation stops at a breakpoint, you can continue chart execution on the Stateflow Editor toolbar, at the MATLAB command prompt, or by selecting a keyboard shortcut.

Toolbar Icon	Option	Command	Description	Keyboard Shortcut
	Continue	`dbcont`	Continue simulation to the next breakpoint.	**Ctrl+T**
	Step Over	`dbstep`	Advance to the next step in the truth table execution.	**F10**
	Step In	`dbstep in`	From a state or transition action that calls a function, advance to the first executable statement in the function. From a statement in a function containing another function call, advance to the first executable statement in the second function. Otherwise, advance to the next step in the truth table execution. (See Step Over.)	**F11**
	Step Out	`dbstep out`	From a function call, return to the statement calling the function. Otherwise, continue simulation to the next breakpoint. (See Continue.)	**Shift+F11**
	Step Forward		Exit debug mode and pause simulation before next time step.	
	Stop	`dbquit`	Exit debug mode and stop simulation.	**Ctrl+Shift+T**

Condition Breakpoints

To set a breakpoint when a condition is tested, right-click the condition cell and select **Set Breakpoint (Condition Tested)**. A red badge appears on the far left of the table next to the number of the condition. When you run the model, the simulation pauses when the condition is tested. Stateflow highlights the condition row being tested. Place your cursor over the data in the truth table to see its current value.

ex_first_truth_table ▶ 🔲 Chart ▶ 🔲 ttable   ▼

Condition Table

	DESCRIPTION	CONDITION	D1	D2	D3	D4
1 ⬤	x is equal to 1	XEQ1: x == 1	T	F	F	-
2	y is equal to 1	YEQ1: y == 1	F	T	F	-
3	z is equal to 1	ZEQ1: z == 1	F	F	T	-
		ACTIONS: SPECIFY A ROW FROM THE ACTION TABLE	A1	A2	A3	A4

Action Table

	DESCRIPTION	ACTION
1	Initial action: Display message	INIT: ml.disp('truth table ttable entered');
2	set t to 1	A1: t=1;
3	set t to 2	A2: t=2;
4	set t to 3	A3: t=3;
5	set t to 4	A4: t=4;
6	Final action: Display message	FINAL: ml.disp('truth table ttable exited');

Decision Breakpoints

To set a breakpoint when a decision is tested, right-click the top of the decision column and select **Set Breakpoint (Decision Tested)**. A red badge appears on the top of the decision column next to the number of the decision. When you run the model, the simulation pauses when the decision is tested. Stateflow highlights the decision column being tested. Place your cursor over the data in the truth table to see its current value.

ex_first_truth_table ▶ Chart ▶ ttable

Condition Table

	DESCRIPTION	CONDITION	D1 ●	D2	D3	D4
1	x is equal to 1	XEQ1: x == 1	T	F	F	-
2	y is equal to 1	YEQ1: y == 1	F	T	F	-
3	z is equal to 1	ZEQ1: z == 1	F	F	T	-
		ACTIONS: SPECIFY A ROW FROM THE ACTION TABLE	A1	A2	A3	A4

Action Table

	DESCRIPTION	ACTION
1	Initial action: Display message	INIT: ml.disp('truth table ttable entered');
2	set t to 1	A1: t=1;
3	set t to 2	A2: t=2;
4	set t to 3	A3: t=3;
5	set t to 4	A4: t=4;
6	Final action: Display message	FINAL: ml.disp('truth table ttable exited');

To set a breakpoint when a decision is valid, right-click the action cell at the bottom of the decision column and select **Set Breakpoint (Decision Valid)**. A red badge appears on the top of the cell next to the action number. When you run the model, the simulation pauses when the action is valid. Stateflow highlights the valid decision. Place your cursor over the data in the truth table to see its current value.

If there is more than one action to take when a decision is valid, the breakpoint is set for the first executable. In truth tables that use MATLAB as the action language, you can set a breakpoint on any line of executable code by setting breakpoints in the generated content. See "Debugging Generated Truth Table Content" on page 27-43.

ex_first_truth_table ▶ Chart ▶ ttable

Condition Table

	DESCRIPTION	CONDITION	D1	D2	D3	D4
1	x is equal to 1	XEQ1: x == 1	T	F	F	-
2	y is equal to 1	YEQ1: y == 1	F	T	F	-
3	z is equal to 1	ZEQ1: z == 1	F	F	T	-
		ACTIONS: SPECIFY A ROW FROM THE ACTION TABLE	A1	A2	A3	A4

Action Table

	DESCRIPTION	ACTION
1	Initial action: Display message	INIT: ml.disp('truth table ttable entered');
2	set t to 1	A1: t=1;
3	set t to 2	A2: t=2;
4	set t to 3	A3: t=3;
5	set t to 4	A4: t=4;
6	Final action: Display message	FINAL: ml.disp('truth table ttable exited');

Action Breakpoints

To set a breakpoint when a action is executed, right-click the action cell and select **Set Breakpoint (Action Executed)**. A red badge appears on the far left of the table next to the number of the action. When you run the model, the simulation pauses when the action is executed. Stateflow highlights the action row being tested. Place your cursor over the data in the truth table to see its current value.

If there is more than one action within the action cell, the breakpoint is set for the first action. In truth tables that use MATLAB as the action language, you can set a breakpoint on any line of executable code by setting breakpoints in the generated content. See "Debugging Generated Truth Table Content" on page 27-43.

ex_first_truth_table ▶ Chart ▶ ttable

Condition Table

	DESCRIPTION	CONDITION	D1	D2	D3	D4
1	x is equal to 1	XEQ1: x == 1	T	F	F	-
2	y is equal to 1	YEQ1: y == 1	F	T	F	-
3	z is equal to 1	ZEQ1: z == 1	F	F	T	-
		ACTIONS: SPECIFY A ROW FROM THE ACTION TABLE	A1	A2	A3	A4

Action Table

	DESCRIPTION	ACTION
1	Initial action: Display message	INIT: ml.disp('truth table ttable entered');
2 ⦿	set t to 1	A1: t=1;
3	set t to 2	A2: t=2;
4	set t to 3	A3: t=3;
5	set t to 4	A4: t=4;
6	Final action: Display message	FINAL: ml.disp('truth table ttable exited');

Edit Breakpoints

Click the breakpoint to open the **Edit Breakpoint** dialog box. From this window you can disable the breakpoint by unchecking **Enable Breakpoint**.

When you add a condition to a breakpoint, the breakpoint pauses the simulation only when its associated condition is `true`.

Debugging Generated Truth Table Content

You can use breakpoints on the generated content for truth tables that use MATLAB or C as the action language. For more information about how to generate content for truth tables, see "View Generated Content for Stateflow Truth Tables" on page 27-55.

For truth tables that use MATLAB as the action language, content is generated as MATLAB code. To set a breakpoint in the code, click the breakpoint alley next to the executable line where you want simulation to pause. See "Debug a MATLAB Program" (MATLAB). When you add an element to your truth table that changes the generated content, all breakpoints are regenerated.

For truth tables that use C as the action language, content is generated as a graphical function. To set a breakpoint in the graphical function, right-click the transition where you want to set a breakpoint and select **Set Breakpoint**. When you add an element to your truth table that changes the generated content, all breakpoints are regenerated if you recompile your truth table.

See Also

More About
- "View Generated Content for Stateflow Truth Tables" on page 27-55
- "Debugging a MATLAB Function Block" (Simulink)

Correct Overspecified and Underspecified Truth Tables

Truth tables implement combinatorial logic design in a concise, tabular format. Truth tables are supported only in Stateflow blocks in a Simulink model. For more information, see "Reuse Combinatorial Logic by Defining Truth Tables" on page 27-2.

Example of an Overspecified Truth Table

An overspecified truth table contains at least one decision that never executes because a previous decision specifies it in the **Condition Table**. The following example shows the **Condition Table** of an overspecified truth table.

ex_truthtable_overspecified ▸ Chart ▸ xyz ▾

Condition Table

	DESCRIPTION	CONDITION	D1	D2	D3
1	Condition C1	C1: x == 0	F	T	-
2	Condition C2	C2: y == 0	T	-	T
3	Condition C3	C3: z == 0	T	T	T
		ACTIONS: SPECIFY A ROW FROM THE ACTION TABLE	A1	A2	A3

Action Table

	DESCRIPTION	ACTION
1	Initial Action	INIT: ml.disp('beginning truth table');
2	Action 1	A1: x = 1;
3	Action 2	A2: y = 1;
4	Action 3	A3: z = 1;
5	Final Action	FINAL: ml.disp('ending truth table');

The decision in column **D3** (-TT) specifies the decisions FTT and TTT. These decisions are duplicates of decisions **D1** (FTT) and **D2** (TTT and TFT). Therefore, column **D3** is an overspecification.

The following example shows the **Condition Table** of a truth table that appears to be overspecified, but is not.

ex_truthtable_not_overspecified ▶ Chart ▶ xyz ▼

Condition Table

	DESCRIPTION	CONDITION	D1	D2	D3	D4
1	Condition C1	C1: x == 0	F	T	T	-
2	Condition C2	C2: y == 0	T	F	T	T
3	Condition C3	C3: z == 0	T	T	F	T
		ACTIONS: SPECIFY A ROW FROM THE ACTION TABLE	**A1**	**A2**	**A3**	**A4**

Action Table

	DESCRIPTION	ACTION
1	Initial Action	INIT: ml.disp('beginning truth table');
2	Action 1	A1: x = 1;
3	Action 2	A2: y = 1;
4	Action 3	A3: z = 1;
5	Action 4	A4: x = 0; y = 0; z = 0;
6	Final Action	FINAL: ml.disp('ending truth table');

27-47

In this case, the decision **D4** specifies two decisions (TTT and FTT). FTT also appears in decision **D1**, but TTT is not a duplicate. Therefore, this **Condition Table** is not overspecified.

Example of an Underspecified Truth Table

An underspecified truth table lacks one or more possible decisions that require an action to avoid undefined behavior. The following example shows the **Condition Table** of an underspecified truth table.

ex_truthtable_underspecified ▸ Chart ▸ xyz

Condition Table

	DESCRIPTION	CONDITION	D1	D2	D3
1	Condition C1	C1: x == 0	T	T	F
2	Condition C2	C2: y == 0	T	F	T
3	Condition C3	C3: z == 0	F	T	T
		ACTIONS: SPECIFY A ROW FROM THE ACTION TABLE	A1	A2	A3

Action Table

	DESCRIPTION	ACTION
1	Initial Action	INIT: ml.disp('beginning truth table');
2	Action 1	A1: x = 1;
3	Action 2	A2: y = 1;
4	Action 3	A3: z = 1;
5	Final Action	FINAL: ml.disp('ending truth table');

27-49

Complete coverage of the conditions in the preceding truth table requires a **Condition Table** with every possible decision:

ex_truthtable_completely_specified ▶ 🔲 Chart ▶ 🔲 xyz ▼

Condition Table

	DESCRIPTION	CONDITION	D1	D2	D3	D4	D5	D6	D7	D8
1	Condition C1	C1: x == 0	T	T	F	T	F	T	F	F
2	Condition C2	C2: y == 0	T	F	T	T	F	F	T	F
3	Condition C3	C3: z == 0	F	T	T	T	F	F	F	T
		ACTIONS: SPECIFY A ROW FROM THE ACTION TABLE	A1	A2	A3	A4	A5	A6	A7	A8

Action Table

	DESCRIPTION	ACTION
1	Initial Action	INIT: ml.disp('beginning truth table');
2	Action 1	A1: x = 1;
3	Action 2	A2: y = 1;
4	Action 3	A3: z = 1;
5	Action 4	A4: x = 2;
6	Action 5	A5: y = 2;
7	Action 6	A6: z = 2;
8	Action 7	A7:

27-51

A possible workaround is to specify an action for all other possible decisions through a default decision, named DA:

ex_truthtable_default_action ▶ Chart ▶ xyz ▼

Condition Table

	DESCRIPTION	CONDITION	D1	D2	D3	D4
1	Condition C1	C1: x == 0	T	T	F	-
2	Condition C2	C2: y == 0	T	F	T	-
3	Condition C3	C3: z == 0	F	T	T	-
		ACTIONS: SPECIFY A ROW FROM THE ACTION TABLE	A1	A2	A3	DA

Action Table

	DESCRIPTION	ACTION
1	Initial Action	INIT: ml.disp('beginning truth table');
2	Action 1	A1: x = 1;
3	Action 2	A2: y = 1;
4	Action 3	A3: z = 1;
5	Default Action	DA: x = 0; y = 0; z = 0;
6	Final Action	FINAL: ml.disp('ending truth table');

27-53

The last decision column is the default decision for the truth table. The default decision covers any remaining decisions not tested in the preceding decision columns. See "The Default Decision Column" on page 27-15 for an example and more complete description of the default decision column for a **Condition Table**.

View Generated Content for Stateflow Truth Tables

Truth tables implement combinatorial logic design in a concise, tabular format. Truth tables are supported only in Stateflow blocks in a Simulink model. For more information, see "Reuse Combinatorial Logic by Defining Truth Tables" on page 27-2.

Types of Generated Content

Stateflow software realizes the logical behavior specified in a truth table by generating content as follows:

Type of Truth Table	Generated Content
C	Graphical function
MATLAB	MATLAB code

View Generated Content

You generate content for a truth table when you simulate your model. Content regenerates whenever a truth table changes. To view the generated content of a truth table, follow these steps:

1   Simulate the model that contains the truth table.

2   Double-click the truth table.

3   Click the **View Generated Content** button:

How Stateflow Software Generates Graphical Functions for Truth Tables

This section describes how Stateflow software translates the logic of a C truth table into a graphical function.

In the following example, a C truth table has three conditions, four decisions and actions, and initial and final actions.

ex_truthtable_view_generated_content ▶ ⊞ Chart ▶ ⊞ xyz ▾

Condition Table

	DESCRIPTION	CONDITION	D1	D2	D3	D4
1	Condition C1	C1: x == 0	T	F	F	-
2	Condition C2	C2: y == 0	-	T	F	-
3	Condition C3	C3: z == 0	-	-	T	-
		ACTIONS: SPECIFY A ROW FROM THE ACTION TABLE	A1	A2	A3	DA

Action Table

	DESCRIPTION	ACTION
1	Initial Action	INIT: ml.disp('beginning truth table')
2	Action 1	A1: x = 1;
3	Action 2	A2: y = 1;
4	Action 3	A3: z = 1;
5	Default Action	DA: x = 0; y = 0; z = 0;
6	Final Action	FINAL: ml.disp('ending truth table');

Stateflow software generates a graphical function for the preceding truth table. The top half of the flow chart looks like this:
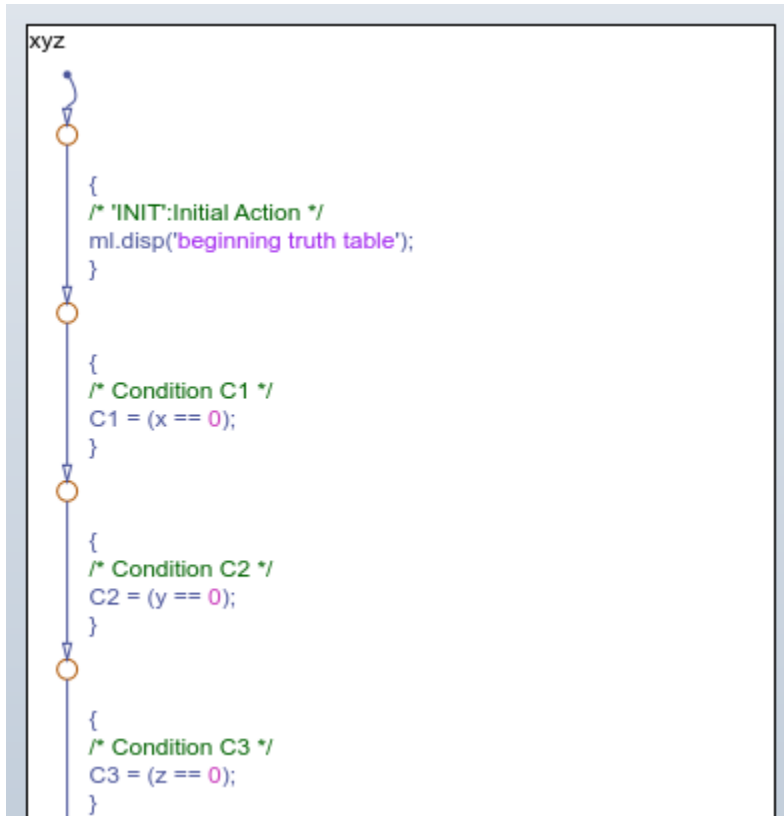
The top half of the flow chart executes as follows:

- Performs initial actions
- Evaluates the conditions and stores the results in temporary data variables

  The temporary data for storing conditions is based on the labels that you enter for the conditions. If you do not specify the labels, temporary data variables appear.

The bottom half of the flow chart looks like this:
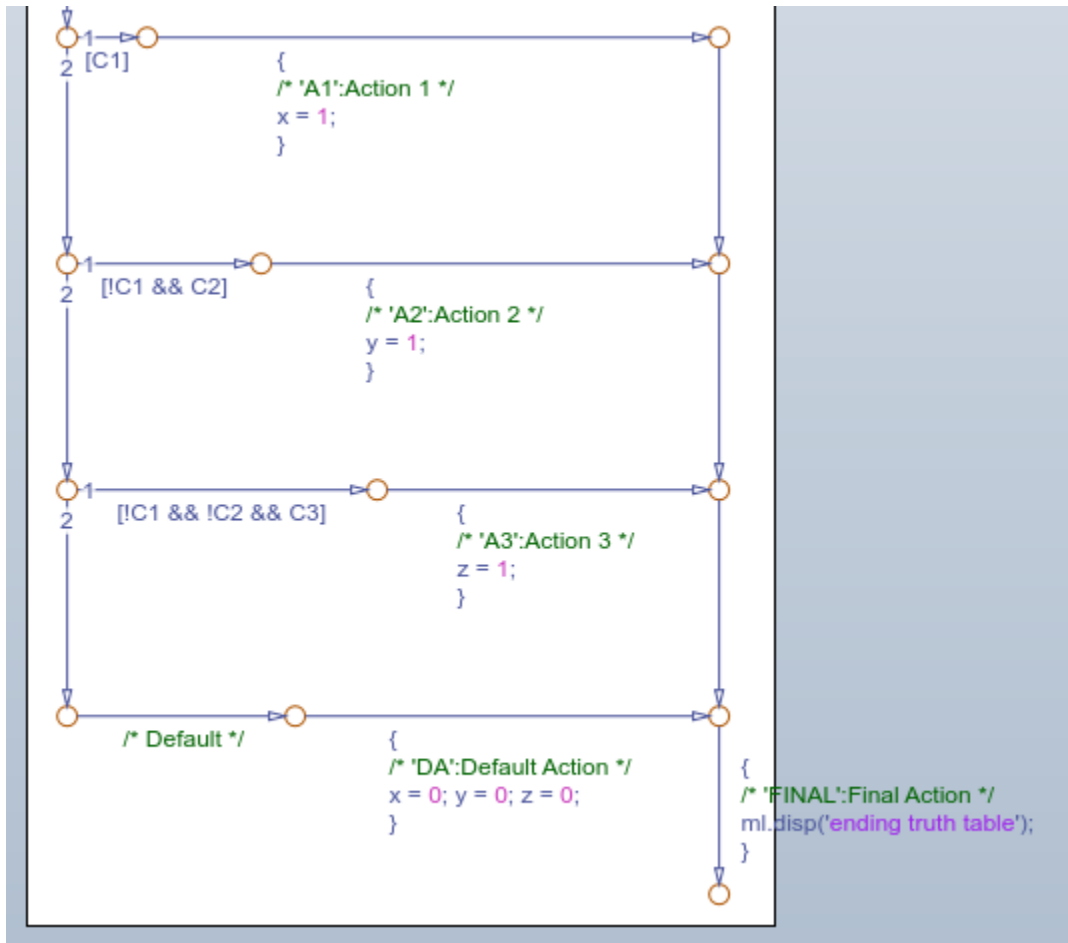
In the bottom half of the flow chart, the stored values for conditions determine which decision is true and what action to perform. Each decision appears as a fork from a connective junction with one of two possible paths:

- A transition segment with a decision followed by a segment with the consequent action

  The action appears as a condition action that leads to the FINAL action and termination of the flow chart.

- A transition segment that flows to the next fork for an evaluation of the next decision

  This transition segment has no condition or action.

This implementation continues from the first decision through the remaining decisions in left-to-right column order. When a decision match occurs, the action for that decision executes as a condition action of its transition segment. After the action executes, the flow chart performs the final action for the truth table and terminates. Therefore, only one action results from a call to a truth table graphical function. This behavior also means that no data dependencies are possible between different decisions.

How Stateflow Software Generates MATLAB Code for Truth Tables

Stateflow software generates the content of MATLAB truth tables as MATLAB code that represents each action as a *nested* function inside the main truth table function.

Nested functions offer these advantages:

- Nested functions are independent of one another. Variables are local to each function and not subject to naming conflicts.
- Nested functions can access all data from the main truth table function.

The generated content appears in the function editor, which provides tools for simulation and debugging.

Here is the generated content for the MATLAB truth table described in "Program Actions Using MATLAB Expressions" on page 27-21:

- Main truth table function

```
function r = ttable(x,y,z)

% Initialize condition vars to logical scalar
XEQ1 = false;
YEQ1 = false;
ZEQ1 = false;

% Condition #1, "XEQ1"
% x is equal to 1
XEQ1 = logical(x == 1);
```

```
% Condition #2, "YEQ1"
% y is equal to 1
YEQ1 = logical(y == 1);

% Condition #3, "ZEQ1"
% z is equal to 1
ZEQ1 = logical(z == 1);

if (XEQ1 && ~YEQ1 && ~ZEQ1) % D1
    A1();
elseif (~XEQ1 && YEQ1 && ~ZEQ1) % D2
    A2();
elseif (~XEQ1 && ~YEQ1 && ZEQ1) % D3
    A3();
else % Default
    A4();
end
```

- Action A1

```
function A1()
% Action #1, "A1"
% Maintain a counter and a circular vector of length 6.
% Every time this action is called,
% output t takes the next value of the vector.

persistent values counter;
cycle = 6;

if isempty(counter)
  % Initialize counter to be zero
  counter = 0;
else
  % Otherwise, increment counter
  counter = counter + 1;
end

if isempty(values)
  % Values is a vector of 1 to cycle
  values = zeros(1, cycle);
  for i = 1:cycle
    values(i) = i;
  end

  % For debugging purposes, call the MATLAB
```

```
  %  function "plot" to show values
  plot(values);
end

% Output r takes the next value in values vector
r = values( mod(counter, cycle) + 1);
```

- Actions A2, A3, and A4

```
function A2()
% Action #2, "A2"
% set r to 2

r=2;

%================================
function A3()
% Action #3, "A3"
% set r to 3

r=3;

%================================
function A4()
% Action #4, "A4"
% set r to 4

r=4;
```

Truth Table Operations

Truth tables implement combinatorial logic design in a concise, tabular format. Truth tables are supported only in Stateflow blocks in a Simulink model. For more information, see "Reuse Combinatorial Logic by Defining Truth Tables" on page 27-2.

To create truth tables, use the Stateflow Editor. You can insert, edit, and move rows and columns. You can also diagnose the truth table for syntax errors and view the auto-generated code for the truth table.

Append Rows and Columns

	Append Column adds a column on the right end of the selected table.
	Append Row adds a row to the bottom of the selected table.

Diagnose the Truth Table

	Run Diagnostics checks the truth table for syntax errors. See "Debug Run-Time Errors in a Truth Table" on page 27-33.

View Auto-generated Content

	View Auto-generated Content displays the code generated for the truth table. C truth tables generate graphical functions. MATLAB truth tables generate MATLAB code. For details, see "View Generated Content for Stateflow Truth Tables" on page 27-55.

Edit Tables

The default **Condition Table** and the default **Action Table** have one empty row. Click a cell to edit its text contents. To move horizontally between cells, use the up and down arrow keys.

To display only one of the two tables, double-click the header of the table that you want to display. To revert to the display of both tables, double-click the header of the displayed table.

Cells for the numbered rows in decision columns like **D1** can take values of T, F, or -. After you select one of these cells, you can use the space bar to step through the T, F, and - values. In these cells you can use the left, right, up, and down arrow keys to advance to another cell in any direction.

Move Rows and Columns

To move a condition or action row up or down:

1 To select the row, click the row header.

2 Drag the row to a new position.

The truth table renumbers the rows.

To move a decision column left or right:

1 To select the column, click the column header .

2 Drag the column to a new position.

The truth table renumbers the decision columns.

Note To select multiple rows or columns that you want to move, hold down the **Ctrl** key.

Select and Deselect Table Elements

Goal	Action
Select a cell for editing	Click the cell
Select text in a cell	Click and drag your pointer over the text
Select a row	Click the header for the row
Select a decision column in the **Condition Table**	Click the header for the column
Deselect a selected cell, row, or column	Click outside of the **Condition Table** or **Action Table**.

Undo and Redo Edit Operations

To undo the effects of the preceding operation, select **Edit > Undo** or press **Ctrl+Z (Command+Z)**.

To redo the effects of the preceding operation, select **Edit > Redo** or press **Ctrl+Y (Command+Y)**.

View the Stateflow Chart for the Truth Table

To navigate to the Stateflow chart for the truth table, press the **Esc** key.

28

MATLAB Functions in Stateflow Charts

Reuse MATLAB Code by Defining MATLAB Functions

A MATLAB function in a Stateflow chart is a graphical element that you use to write algorithms that are easier to implement by calling built-in MATLAB functions. Typical applications include:

- Matrix-oriented calculations
- Data analysis and visualization

This type of function is useful for coding algorithms that are more easily expressed by using MATLAB instead of the graphical Stateflow constructs. MATLAB functions also provide optimizations for generating efficient, production-quality C code for embedded applications.

A MATLAB function can reside anywhere in a chart, state, or subchart. The location of the function determines the set of states and transitions that can call the function.

- If you want to call the function only within one state or subchart and its substates, put your MATLAB function in that state or subchart. That function overrides any other functions of the same name in the parents and ancestors of that state or subchart.
- If you want to call the function anywhere in that chart, put your MATLAB function at the chart level.
- If you want to call the function from any chart in your model, put your MATLAB function at the chart level and enable exporting of chart-level functions. For more information, see "Export Stateflow Functions for Reuse" on page 8-22.

For example, this MATLAB function has the name `stdevstats`. It takes an argument `vals` and returns an output value `stdevout`.
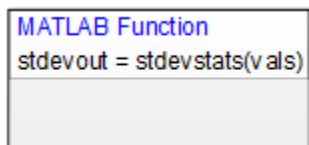
To compute the standard deviation of the values in `vals`, the function uses this code.

```
function stdevout = stdevstats(vals)
%#codegen
```

```
% Calculates the standard deviation for vals

len = length(vals);
stdevout = sqrt(sum(((vals-avg(vals,len)).^2))/len);

function mean = avg(array,size)
mean = sum(array)/size;
```

Define a MATLAB Function in a Chart

1
   In the object palette, click the MATLAB function icon   . Move your pointer to the location for the new MATLAB function in your chart.

2 Enter the signature label for the function, as described in "Declare Function Arguments and Return Values" on page 28-4.

3 To program the function, open the MATLAB editor by double-clicking the function box.

4 In the editor, enter the MATLAB code implementing your function.

5 In the Model Explorer, expand the chart object and select the MATLAB function. The arguments and return values of the function signature appear as data items that belong to your function. Arguments have the scope `Input`. Return values have the scope `Output`.

6 In the Data properties dialog box for each argument and return value, specify the data properties, as described in "Set Data Properties" on page 9-5.

7 Create any additional data items required by your function. For more information, see "Add Data Through the Model Explorer" on page 9-3.

   Your function can access its own data or data belonging to parent states or the chart. The data items in the function can have one of these scopes:

   - `Local` — Local data persists from one function call to the next function call. Valid for C charts only.
   - `Constant` — Constant data retains its initial value through all function calls.
   - `Parameter` — Parameter data retains its initial value through all function calls.
   - `Temporary` — Temporary data initializes at the start of every function call. Valid for C charts only.

In charts that use C as the action language, define temporary data when you want to use data that is only valid while a function executes. For example, you can designate a loop

counter to have `Temporary` scope if the counter value does not need to persist after the function completes.

In charts that use MATLAB as the action language, you do not need to define temporary function data. If you use an undefined variable, Stateflow creates a temporary variable. The variable is available to the rest of the function.

You can initialize your function data (other than arguments and return values) from the MATLAB workspace. For more information, see "Initialize Data from the MATLAB Base Workspace" on page 9-24.

Declare Function Arguments and Return Values

The function signature label specifies a name for your function and the formal names for its arguments and return values. A signature label has this syntax:

`[return_val1, return_val2,...] = function_name(arg1, arg2,...)`

You can specify multiple return values and multiple input arguments. Each return value and input argument can be a scalar, vector, or matrix of values. For functions with only one return value, omit the brackets in the signature label.

You can use the same variable name for both arguments and return values. For example, a function with this signature label uses the variables `y1` and `y2` as both inputs and outputs:

`[y1, y2, y3] = f(y1, u, y2)`

If you export this function to C code, `y1` and `y2` are passed by reference (as pointers), and `u` is passed by value. Passing inputs by reference reduces the number of times that the generated code copies intermediate data, resulting in more optimal code.

Call MATLAB Functions in States and Transitions

You can call MATLAB functions from the actions of any state or transition. You can also call MATLAB functions from other functions. If you export a MATLAB function, you can call it from any chart in the model.

The syntax for a call to a MATLAB function is the same as the function signature, with actual arguments replacing the formal ones specified in a signature. If the data types of an actual and formal argument differ, a function casts the actual argument to the type of the formal argument.

Tip If the formal arguments of a function signature are scalars, verify that inputs and outputs of function calls follow the rules of scalar expansion. For more information, see "How Scalar Expansion Works for Functions" on page 17-9.

Specify MATLAB Function Properties in a Chart

You can set general properties for your MATLAB function through its properties dialog box. To open the function properties dialog box, right-click the MATLAB function box and select **Properties** from the context menu.

Name

Function name. Click the function name link to open your function in the MATLAB editor.

Function Inline Option

Controls the inlining of your function in generated code:

- `Auto` — Determines whether to inline your function based on an internal calculation.
- `Inline` — Inlines your function if you do not export it to other charts and it is not part of a recursion. (A recursion exists if your function calls itself directly or indirectly through another function call.)
- `Function` — Does not inline your function.

Label

Signature label for your function. For more information, see "Declare Function Arguments and Return Values" on page 28-4.

Saturate on Integer Overflow

Specifies whether integer overflows saturate in the generated code. For more information, see "Handle Integer Overflow for Chart Data" on page 9-45.

MATLAB Function fimath

Defines the `fimath` properties for the MATLAB function. The `fimath` properties specified are associated with all `fi` and `fimath` objects constructed in the MATLAB function. Choose one of these options:

- **Same as MATLAB** — The function uses the same `fimath` properties as the current global `fimath`. The edit box appears dimmed and displays the current global `fimath` in read-only form. For more information on the global `fimath` and `fimath` objects, see the Fixed-Point Designer documentation.

- **Specify Other** — Specify your own `fimath` object by one of these methods:

  - Construct the `fimath` object inside the edit box.
  - Construct the `fimath` object in the MATLAB or model workspace and enter its variable name in the edit box.

Description

Function description. You can enter brief descriptions of functions in the hierarchy.

Document Link

Link to online documentation for the function. You can enter a web URL address or a MATLAB command that displays documentation in a suitable online format, such as an HTML file or text in the MATLAB Command Window. When you click the **Document link** hyperlink, Stateflow displays the documentation.

Enumerated and Variable-Size Data in MATLAB Functions

In Stateflow charts in Simulink models, you can use enumerated data and variable-size data in MATLAB functions. For more information, see "Code Generation for Enumerations" (Simulink) and "Declare Variable-Size Inputs and Outputs" (Simulink).

See Also

More About

- "Program a MATLAB Function in a Chart" on page 28-7
- "Reusable Functions in Charts" on page 2-46
- "Export Stateflow Functions for Reuse" on page 8-22
- "Reuse Functions by Using Atomic Boxes" on page 8-35

Program a MATLAB Function in a Chart

This example shows how to create a model with a Stateflow chart that calls two MATLAB functions, `meanstats` and `stdevstats`:

- `meanstats` calculates the mean of the values in `vals`.
- `stdevstats` calculates a standard deviation for the values in `vals`.

Build Model

Follow these steps:

1 Create a new model with the following blocks:
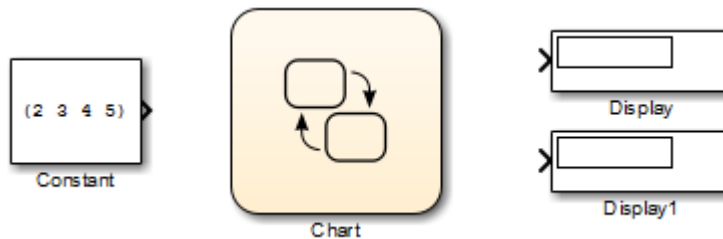
2 Save the model as `call_stats_function_stateflow`.
3 In the model, double-click the Chart block.
4 Drag two MATLAB functions into the empty chart using this icon from the toolbar:

A text field with a flashing cursor appears in the middle of each MATLAB function.
5 Label each function as shown:

You must label a MATLAB function with its signature. Use the following syntax:

```
[return_val1, return_val2,...] = function_name(arg1, arg2,...)
```

You can specify multiple return values and multiple input arguments, as shown in the syntax. Each return value and input argument can be a scalar, vector, or matrix of values.

Note For MATLAB functions with only one return value, you can omit the brackets in the signature label.

---

6 In the chart, draw a default transition into a terminating junction with this condition action:

```
{
mean = meanstats(invals);
stdev = stdevstats(invals);
}
```

The chart should look something like this:
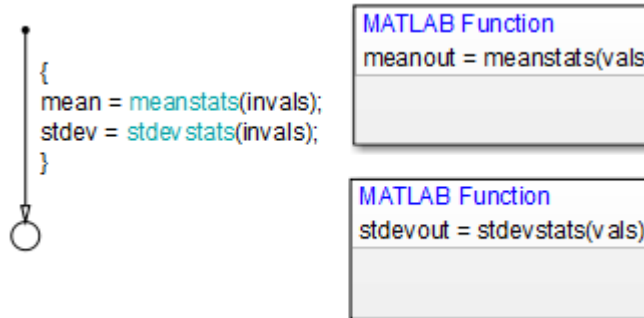
Tip If the formal arguments of a function signature are scalars, verify that inputs and outputs of function calls follow the rules of scalar expansion. For more information, see "How Scalar Expansion Works for Functions" on page 17-9.

7   In the chart, double-click the function `meanstats` to edit its function body in the editor.

8   In the function editor, select **Tools > Model Explorer** to open the Model Explorer.
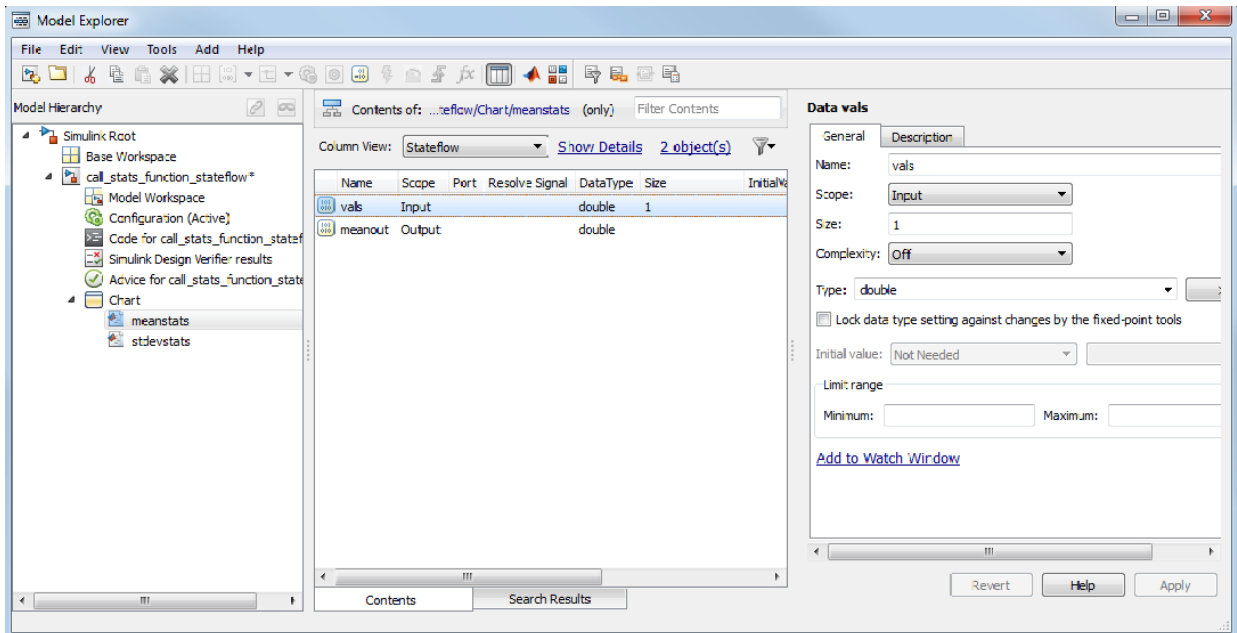
The function `meanstats` is highlighted in the **Model Hierarchy** pane. The **Contents** pane displays the input argument `vals` and output argument `meanout`. Both are scalars of type `double` by default.

9   Double-click the `vals` row under the **Size** column to set the size of `vals` to 4.

10   Back in the chart, double-click the function `stdevstats` and repeat the previous two steps.

11   Back in the **Model Hierarchy** pane of the Model Explorer, select **Chart** and add the following data:

Name	Scope	Size
`invals`	`Input`	4
`mean`	`Output`	Scalar (no change)
`stdev`	`Output`	Scalar (no change)

You should now see the following data in the Model Explorer.
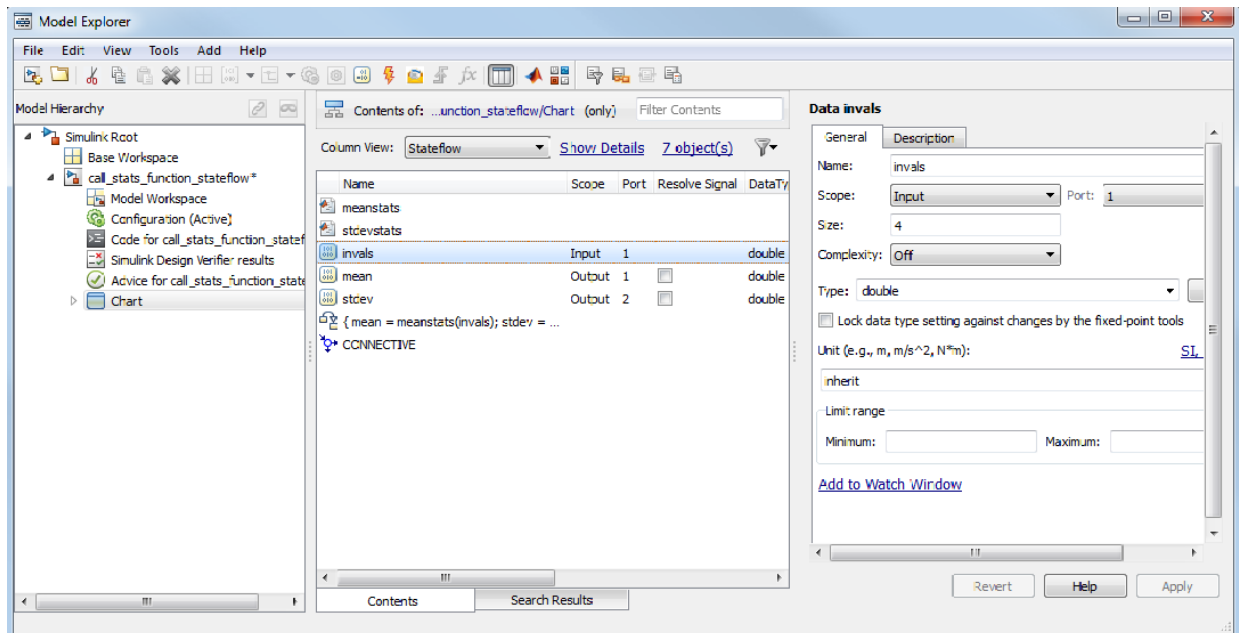
After you add the data `invals`, `mean`, and `stdev` to the chart, the corresponding input and output ports appear on the Stateflow block in the model.
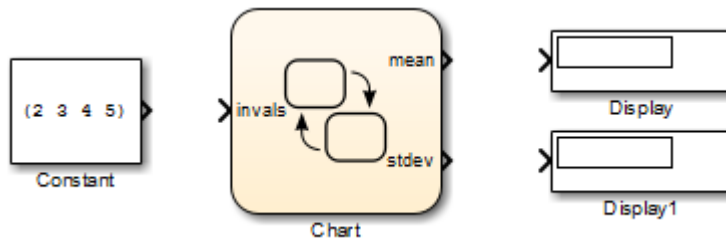
12 Connect the Constant and Display blocks to the ports of the Chart block and save the model.
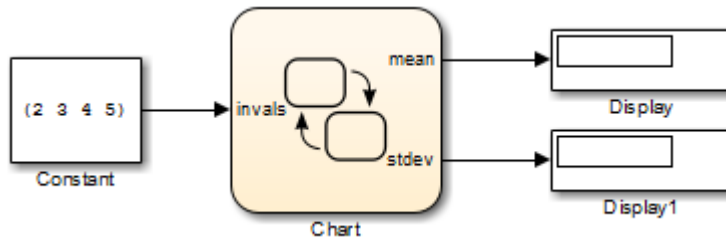
Program MATLAB Functions

To program the functions `meanstats` and `stdevstats`, follow these steps:

1 Open the chart in the model `call_stats_function_stateflow`.

2 In the chart, open the function `meanstats`.

The function editor appears with the header:

```
function meanout = meanstats(vals)
```

This header is taken from the function label in the chart. You can edit the header directly in the editor, and your changes appear in the chart after you close the editor.

3 On the line after the function header, enter:

```
%#codegen
```

The `%#codegen` compilation directive helps detect compile-time violations of syntax and semantics in MATLAB functions supported for code generation.

28-11

4 Enter a line space and this comment:

```
% Calculates the statistical mean for vals
```

5 Add the line:

```
len = length(vals);
```

The function `length` is an example of a built-in MATLAB function that is supported for code generation. You can call this function directly to return the vector length of its argument `vals`. When you build a simulation target, the function length is implemented with generated C code. Functions supported for code generation appear in "Functions and Objects Supported for C/C++ Code Generation — Alphabetical List" (MATLAB Coder).

The variable `len` is an example of implicitly declared local data. It has the same size and type as the value assigned to it — the value returned by the function `length`, a scalar `double`. To learn more about declaring variables, see "Data Definition Basics" (MATLAB Coder).

The MATLAB function treats implicitly declared local data as temporary data, which exists only when the function is called and disappears when the function exits. You can declare local data for a MATLAB function in a chart to be persistent by using the `persistent` construct.

6 Enter this line to calculate the value of `meanout`:

```
meanout = avg(vals,len);
```

The function `meanstats` stores the mean of `vals` in the Stateflow data `meanout`. Because these data are defined for the parent Stateflow chart, you can use them directly in the MATLAB function.

Two-dimensional arrays with a single row or column of elements are treated as vectors or matrices in MATLAB functions. For example, in `meanstats`, the argument `vals` is a four-element vector. You can access the fourth element of this vector with the matrix notation `vals(4,1)` or the vector notation `vals(4)`.

The MATLAB function uses the functions `avg` and `sum` to compute the value of `mean`. `sum` is a function supported for code generation. `avg` is a local function that you define later. When resolving function names, MATLAB functions in your chart look for local functions first, followed by functions supported for code generation.

> **Note** If you call a function that the MATLAB function cannot resolve as a local function or a function for code generation, you must declare the function to be extrinsic.

7   Now enter this statement:

```
coder.extrinsic('plot');
```

8   Enter this line to plot the input values in `vals` against their vector index.

```
plot(vals,'-+');
```

Recall that you declared `plot` to be an extrinsic function because it is not supported for code generation. When the MATLAB function encounters an extrinsic function, it sends the call to the MATLAB workspace for execution during simulation.

9   Now, define the local function `avg`, as follows:

```
function mean = avg(array,size)
mean = sum(array)/size;
```

The header for `avg` defines two arguments, `array` and `size`, and a single return value, `mean`. The local function `avg` calculates the average of the elements in `array` by dividing their sum by the value of argument `size`.

The complete code for the function `meanstats` looks like this:

```
function meanout = meanstats(vals)
%#codegen

% Calculates the statistical mean for vals

len = length(vals);
meanout = avg(vals,len);

coder.extrinsic('plot');
plot(vals,'-+');

function mean = avg(array,size)
mean = sum(array)/size;
```

10   Save the model.

11   Back in the chart, open the function `stdevstats` and add code to compute the standard deviation of the values in `vals`. The complete code should look like this:

```
function stdevout = stdevstats(vals)
%#codegen
```

28-13

```
% Calculates the standard deviation for vals

len = length(vals);
stdevout = sqrt(sum(((vals-avg(vals,len)).^2))/len);

function mean = avg(array,size)
mean = sum(array)/size;
```

12 Save the model again.

Connect Structures in MATLAB Functions to Simulink Bus Signals

About Structures in MATLAB Functions

MATLAB functions support MATLAB structures. You can create structures in top-level MATLAB functions to interface with Simulink bus signals at input and output ports. Simulink buses appear inside the MATLAB function as structures; structure outputs from the MATLAB function appear as buses.

You can also create structures as local and persistent variables in top-level functions and local functions of MATLAB functions.

Define Structures in MATLAB Functions

This section describes how to define structures in MATLAB functions.

- "Rules for Defining Structures in MATLAB Functions" on page 28-15
- "Define Structure Inputs and Outputs to Interface with Bus Signals" on page 28-15
- "Define Local and Persistent Structure Variables" on page 28-17

Rules for Defining Structures in MATLAB Functions

Follow these rules when defining structures for MATLAB functions in Stateflow charts:

- For each structure input or output in a MATLAB function, you must define a `Simulink.Bus` object in the base workspace to specify its type to the Simulink signal.
- MATLAB structures cannot inherit their type from Simulink signals.
- MATLAB functions support nonvirtual buses only (see "Virtual and Nonvirtual Buses" (Simulink)).
- Structures cannot have scopes defined as **Constant**.

Define Structure Inputs and Outputs to Interface with Bus Signals

When you create structure inputs in MATLAB functions, the function determines the type, size, and complexity of the structure from the Simulink input signal. When you create structure outputs, you must define their type, size, and complexity in the MATLAB function.

28-15

You can connect MATLAB structure inputs and outputs to any Simulink bus signal, including:

- Simulink blocks that output bus signals — such as Bus Creator blocks
- Simulink blocks that accept bus signals as input — such as Bus Selector and Gain blocks
- S-Function blocks
- Other MATLAB functions

To define structure inputs and outputs for MATLAB functions in Stateflow charts, follow these steps:

1   Create a Simulink bus object in the base workspace to specify the properties of the structure you will create in the MATLAB function.

For information about how to create Simulink bus objects, see "Create Bus Objects with the Bus Editor" (Simulink).

2   Open the Model Explorer and follow these steps:

a   In the **Model Hierarchy** pane, select the MATLAB function in your chart.

b   Add a data object, as described in "Add Data Through the Model Explorer" on page 9-3.

The Model Explorer adds a data object and opens a Properties dialog box in its right-hand **Dialog** pane.

c   In the Properties dialog box, enter the following information in the **General** tab fields:

Field	What to Specify
Name	Enter a name for referencing the structure in the MATLAB function. This name does not have to match the name of the bus object in the base workspace.
Scope	Select `Input` or `Output`.

Field	What to Specify
Type	Select `Bus: <bus object name>` from the drop-down list. Then, replace "*<bus object name>*" with the name of the `Simulink.Bus` object in the base workspace that defines the structure. For example: `Bus: inbus`.

d   To add or modify `Simulink.Bus` objects, open the Data Type Assistant. Then, click the **Edit** button to open the Simulink Bus Editor (see "Create Bus Objects with the Bus Editor" (Simulink)).

e   Click **Apply**.

3   If your structure is an output (has scope of `Output`), define the output implicitly in the MATLAB function to have the same type, size, and complexity as its `Simulink.Bus` object. For details, see "Structures" (MATLAB Coder).

Define Local and Persistent Structure Variables

You can define structures as local or persistent variables inside MATLAB functions. For details, see "Structures" (MATLAB Coder).

Debug a MATLAB Function in a Chart

Check MATLAB Functions for Syntax Errors

Before you can build a simulation application for a model, you must fix syntax errors. Follow these steps to check the MATLAB function `meanstats` for syntax violations:

1 Open the function `meanstats` inside the chart in the `call_stats_function_stateflow` model that you constructed in "Program a MATLAB Function in a Chart" on page 28-7.

The editor automatically checks your function code for errors and recommends corrections.

2 In the Stateflow Editor, select **Code** > **C/C++ Code** > **Build Model**.

If there are no errors or warnings, the Builder window appears and reports success. Otherwise, it lists errors. For example, if you change the name of local function `avg` to a nonexistent local function `aug` in `meanstats`, errors appear in the Diagnostic Viewer.

3 The diagnostic message provides details of the type of error and a link to the code where the error occurred. The diagnostic message also contains a link to a diagnostic report that provides links to your MATLAB functions and compile-time type information for the variables and expressions in these functions. If your model fails to build, this information simplifies finding sources of error messages and aids understanding of type propagation rules. For more information about this report, see "MATLAB Function Reports" (Simulink).

4 In the diagnostic message, click the link after the function name `meanstats` to display the offending line of code.

The offending line appears highlighted in the editor.

5 Correct the error by changing `aug` back to `avg` and recompile. No errors are found and the compile completes successfully.

Run-Time Debugging for MATLAB Functions in Charts

You use simulation to test your MATLAB functions for run-time errors that are not detectable by Stateflow. When you simulate your model, your MATLAB functions undergo diagnostic tests for missing or undefined information and possible logical conflicts as

described in "Check MATLAB Functions for Syntax Errors" on page 28-18. If no errors are found, the simulation of your model begins.

Follow these steps to simulate and debug the `meanstats` function during run-time conditions:

1 In the function editor, click the dash (`-`) character in the left margin of this line:

```
len = length(vals);
```

A small red ball appears in the margin of this line, indicating that you have set a breakpoint.

2 Start simulation for the model.

If you get any errors or warnings, make corrections before you try to simulate again. Otherwise, simulation pauses when execution reaches the breakpoint you set. A small green arrow in the left margin indicates this pause.

3 To advance execution to the next line, select **Step**.

Notice that this line calls the local function `avg`. If you select **Step** here, execution advances past the execution of the local function `avg`. To track execution of the lines in the local function `avg`, select **Step In** instead.

4 To advance execution to the first line of the called local function `avg`, select **Step In**.

Once you are in the local function, you can advance through one line at a time with the **Step** tool. If the local function calls another local function, use the **Step In** tool to step into it. To continue through the remaining lines of the local function and go back to the line after the local function call, select **Step Out**.

5 Select **Step** to execute the only line in `avg`.

When `avg` finishes its execution, you see a green arrow pointing down under its last line.

6 Select **Step** to return to the function `meanstats`.

Execution advances to the line after the call to `avg`.

7 To display the value of the variable `len`, place your pointer over the text `len` in the function editor for at least a second.

The value of `len` appears adjacent to your pointer.

You can display the value for any data in the MATLAB function in this way, no matter where it appears in the function. For example, you can display the values for the vector `vals` by placing your pointer over it as an argument to the function `length`, or as an argument in the function header.

You can also report the values for MATLAB function data in the MATLAB Command Window during simulation. When you reach a breakpoint, the `debug>>` command prompt appears in the MATLAB Command Window (you may have to press **Enter** to see it). At this prompt, you can inspect data defined for the function by entering the name of the data, as shown in this example:

```
debug>> len
len =
      4
debug>>
```

As another debugging alternative, you can display the execution result of a function line by omitting the terminating semicolon. If you do, execution results appear in the MATLAB Command Window during simulation.

8 To leave the function until it is called again and the breakpoint is reached, select **Continue**.

At any point in a function, you can advance through the execution of the remaining lines of the function with the **Continue** tool. If you are at the end of the function, selecting **Step** does the same thing.

9 Click the breakpoint to remove it and select **Quit Debugging** to complete the simulation.

In the model, the computed values of `mean` and `stdev` now appear in the Display blocks.
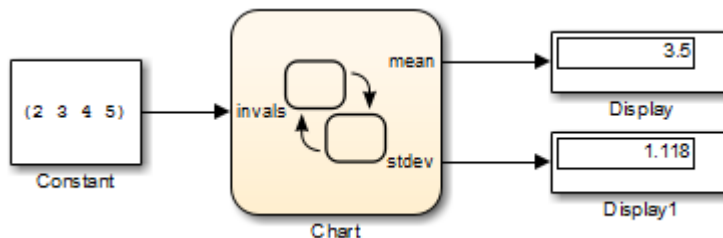
Check for Data Range Violations

To control data range checking, set **Simulation range checking** in the **Diagnostics: Data Validity** pane of the Model Configuration Parameters dialog box.

Specify a Range

To specify a range for input and output data, follow these steps:

1 In the Model Explorer, select the MATLAB function input or output of interest.

The Data properties dialog box opens in the **Dialog** pane of the Model Explorer.

2 In the Data properties dialog box, click the **General** tab and enter a limit range, as described in "Limit Range Properties" on page 9-10.

MATLAB Functions in a Stateflow Chart

The following model contains a Stateflow chart with a MATLAB function.
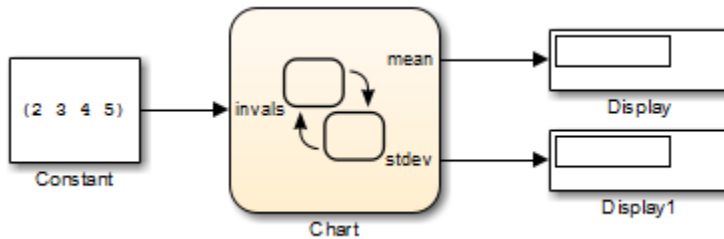
The chart contains the following logic:
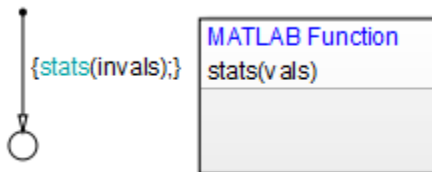
The function contains the following code:

```
function stats(vals)
%#codegen

% calculates a statistical mean and standard deviation
% for the values in vals.

len = length(vals);
mean = avg(vals, len);
stdev = sqrt(sum(((vals-avg(vals,len)).^2))/len);
coder.extrinsic('plot');
plot(vals,'-+');

function mean = avg(array,size)
mean = sum(array)/size;
```

Note in this example that the MATLAB function can call any of these types of functions:

- Local functions

  Local functions are defined in the body of the MATLAB function. In this example, `avg` is a local function.

- Stateflow functions

  Graphical, truth table, and other MATLAB functions can be called from a MATLAB function in a chart.

- MATLAB toolbox functions that support code generation

  Toolbox functions for code generation are a subset of the functions that you can call in the MATLAB workspace. These functions generate C code for building targets that conform to the memory and data type requirements of embedded environments. In this example, `length`, `sqrt`, and `sum` are examples of toolbox functions for code generation.

- MATLAB toolbox functions that do not support code generation

  You can also call *extrinsic* functions on the MATLAB path that do not generate code. These functions execute only in the MATLAB workspace during simulation of the model.

- Simulink Design Verifier functions

  Simulink Design Verifier software provides MATLAB functions for property proving and test generation.

  - `sldv.prove`
  - `sldv.assume`
  - `sldv.test`
  - `sldv.condition`

Define Data in MATLAB Functions

Define Enumerated Data in MATLAB Functions

Define and use enumerated data with MATLAB functions in Stateflow charts the same way as in MATLAB Function blocks in a model. For more information, see "Code Generation for Enumerations" (Simulink).

To learn how to define and use enumerated data in Stateflow charts, see "Define Enumerated Data Types" on page 19-6.

Declare Variable-Size Data in MATLAB Functions

Declare and use variable-size matrices and arrays with MATLAB functions in Stateflow charts the same way as in MATLAB Function blocks in a model. For more information, see:

- "Control Support for Variable-Size Arrays in a MATLAB Function Block" (Simulink)
- "Declare Variable-Size Inputs and Outputs" (Simulink)
- "Use a Variable-Size Signal in a Filtering Algorithm" (Simulink)

To learn how to declare variable-size data at the chart level, see "Declare Variable-Size Inputs and Outputs" on page 18-2.

Define Temporary Data

- "When to Define Temporary Data" on page 28-24
- "How to Define Temporary Data" on page 28-25

When to Define Temporary Data

In C charts, define temporary data when you want to use data that is only valid while a function executes. In charts that use MATLAB as the action language, you do not need to define temporary function data in the Model Explorer. If a variable is used and not previously defined, then it is automatically created. The variable is available to the rest of the function.

You can define temporary data in graphical, truth table, and MATLAB functions in your chart. For example, you can designate a loop counter to have **Temporary** scope if the counter value does not need to persist after the function completes.

How to Define Temporary Data

To define temporary data for a Stateflow function, follow these steps:

1 Open the Model Explorer.

2 In the Model Explorer, select the graphical, truth table, or MATLAB function that will use temporary data.

3 Select **Add > Data**.

The Model Explorer adds a default definition for the data in the Stateflow hierarchy, with a scope set to **Temporary** by default.

4 Change other properties of the data if necessary, as described in "Set Data Properties" on page 9-5.

Enhance Readability of Generated Code for MATLAB Functions

You can enhance the readability of generated code for MATLAB functions in Stateflow charts the same way as in MATLAB Function blocks in a model. For more information, see "Enhance Code Readability for MATLAB Function Blocks" (Embedded Coder).

29

Simulink Functions in Stateflow Charts

Reuse Simulink Components in Stateflow Charts

A Simulink function is a graphical object that you fill with Simulink blocks and call in the actions of states and transitions. Simulink functions are supported only in Stateflow charts in Simulink models.

Simulink functions provide an efficient model design and improve readability by minimizing graphical and nongraphical objects. Typical applications include:

- Defining a function that requires Simulink blocks, such as lookup tables (see "About Lookup Table Blocks" (Simulink))
- Scheduling execution of multiple controllers

You can call Simulink functions defined inside of a Stateflow chart from the same chart. You can also call functions defined by a Simulink Function block in the model.

Where to Define a Simulink Function in a Chart

A Simulink function can reside anywhere in a chart, state, or subchart. The location of a function determines its scope, that is, the set of states and transitions that can call the function. Follow these guidelines:

- If you want to call the function within only one state or subchart and its substates, put your Simulink function in that state or subchart. That function overrides any other functions of the same name in the parents and ancestors of that state or subchart.
- If you want to call the function anywhere in that chart, put your Simulink function at the chart level.

Call a Simulink Function from Multiple Sites in a Chart

If you call a Simulink function from multiple sites in a chart, all call sites share the state of the function variables. For example, suppose you have a chart with two calls to the same Simulink function at each time step.
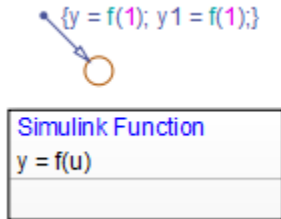
The function f contains a block diagram that increments a counter by 1 each time the function executes.
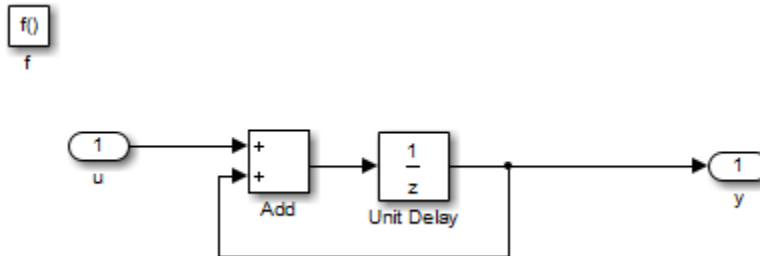
At each time step, the function f is called twice, which causes the counter to increment by 2. Because all call sites share the value of this counter, the data y and y1 increment by 2 at each time step.

Note This behavior also applies to external function-call subsystems in a Simulink model. For more information, see "Using Function-Call Subsystems" (Simulink).

Rules for Using Simulink Functions in Stateflow Charts

Do not call Simulink functions in state `during` actions or transition conditions of continuous-time charts

This rule applies to continuous-time charts because you cannot call functions during minor time steps. You can call Simulink functions in state `entry` or `exit` actions and transition actions. However, if you try to call Simulink functions in state `during` actions or transition conditions, an error message appears when you simulate your model.

For more information, see "Continuous-Time Modeling in Stateflow" on page 21-2.

Do not call Simulink functions in default transitions if you enable execute-at-initialization mode

If you select **Execute (enter) Chart At Initialization** in the Chart properties dialog box, you cannot call Simulink functions in default transitions that execute the first time that the chart awakens. Otherwise, an error message appears when you simulate your model.

Use only alphanumeric characters or underscores when naming input and output ports for a Simulink function

This rule ensures that the names of input and output ports are compatible with identifier naming rules of Stateflow charts.

Note Any space in a name automatically changes to an underscore.

Convert discontiguous signals to contiguous signals for Simulink functions

For Simulink functions inside a Stateflow chart, the output ports do not support discontiguous signals. If your function contains a block that outputs a discontiguous signal, insert a Signal Conversion block between the discontiguous output and the output port. This action ensures that the output signal is contiguous.

Blocks that can output a discontiguous signal include the Bus Creator block and the Mux block. For the Bus Creator block, the output is discontiguous only if you clear the **Output as nonvirtual bus** check box — that is, if the Bus Creator block outputs a virtual bus. If you select **Output as nonvirtual bus**, the output signal is contiguous and no conversion is necessary.

For more information, see Bus Creator, Mux, and Signal Conversion.

Do not export Simulink functions

If you try to export Simulink functions, an error appears when you simulate your model. To avoid this behavior, clear the **Export Chart Level Functions** check box in the Chart properties dialog box.

Use the Stateflow Editor to rename a Simulink function

If you try to use the Model Explorer to rename a Simulink function, the change does not appear in the chart. Edit the name in the Symbols window or click the function box in the Stateflow Editor to rename the function.

Do not use Simulink functions in Moore charts

This restriction prevents violations of Moore semantics during chart execution. For more information, see "Design Considerations for Moore Charts" on page 7-11.

Do not generate HDL code for Simulink functions

If you try to generate HDL code for charts that contain Simulink functions, an error message appears when you simulate your model. HDL code generation does not support Simulink functions.

Set properties of input ports explicitly for a Simulink function

The input ports of Simulink functions cannot inherit their sizes and data types. The output ports of a Simulink Function block in Simulink cannot inherit their sizes and data types. Therefore, you must set sizes and types explicitly when the inputs and outputs are not scalar data of type `double`.

The output ports of a Simulink function defined inside a chart can inherit sizes and data types based on connections inside the subsystem. Therefore, you can specify sizes and types of these outputs as inherited.

Tip To minimize updates required for changes in input port properties, you can specify sizes and data types as parameters.

Verify that function-call expressions have inputs and outputs of correct size

If the formal arguments of a function signature are scalars, verify that inputs and outputs of function calls follow the rules of scalar expansion. For more information, see "How Scalar Expansion Works for Functions" on page 17-9.

Pass arguments by value

Passing an argument to a Simulink function by reference generates a run-time error during simulation.

See Also

Simulink Function

More About

Share Functions Across Simulink and Stateflow

Communicate and Share Resources with Functions

This example shows a model calling functions across Simulink® and Stateflow®. The slexPrinterExample model has three computer clients sharing a printer. Each computer creates print jobs by calling the Simulink function addPrintJob.

In this example, the Stateflow chart communicates with the model by:

- Defining and exporting a graphical function that is called by Simulink.
- Calling a Simulink function that is defined in Simulink.

```
open_system('slexPrinterExample');
```
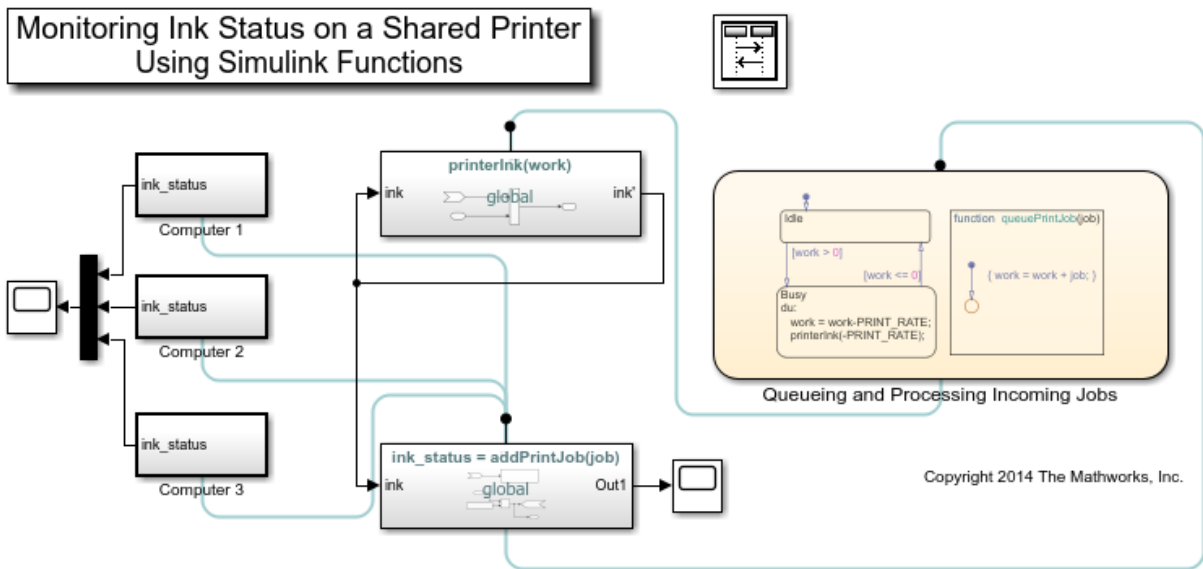

Each computer client invokes the printer server with a call to the Simulink function, addPrintJob. The addPrintJob function calls the Stateflow graphical function queuePrintJob to add the print job to the work load. The chart processes the work and calls the Simulink function printerInk to model usage of printer ink.

Call a Simulink Function from Stateflow

The function `printerInk` is defined in a Simulink Function block at the top level of the model. The function interface `printerInk(work)` defines one input argument. The Simulink Function, printerInk, also interacts with the model with signal lines through the inport `ink` and outport `ink'`. The state `Busy` matches the function signature for `printerInk(work)` by passing one input argument.

open_system('slexPrinterExample/Queueing and Processing Incoming Jobs')
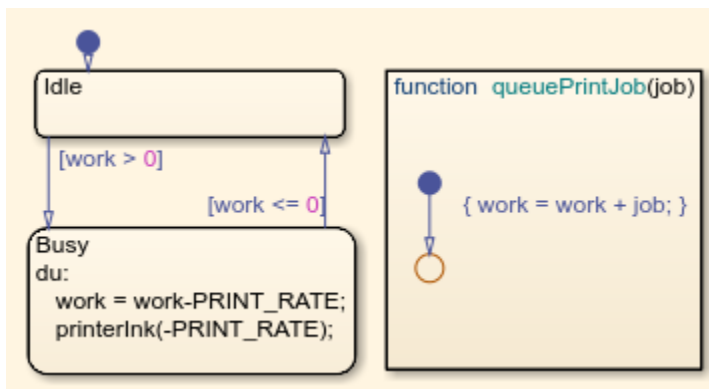
Export Stateflow Functions to Simulink

In the chart `Queuing` and `Processing Incoming Jobs`, the properties **Export Chart Level Functions** and **Treat Exported Functions as Globally Visible** are selected. These properties allow the Simulink function `addPrintJob` to call the chart graphical function, `queuePrintJob`.

See Also
Simulink Function

More About
- "Export Stateflow Functions for Reuse" on page 8-22
- "Simulink Functions Overview" (Simulink)
- "Model Reference Basics" (Simulink)

Why Use a Simulink Function in a Stateflow Chart?

In this section...
"Advantages of Using Simulink Functions in a Stateflow Chart" on page 29-9
"Benefits of Using a Simulink Function to Access Simulink Blocks" on page 29-9
"Benefits of Using a Simulink Function to Schedule Execution of Multiple Controllers" on page 29-12

Advantages of Using Simulink Functions in a Stateflow Chart

When you define a function that uses Simulink function-call subsystem blocks or schedule execution of multiple controllers without Simulink functions, the model requires these elements:

- Simulink function-call subsystem blocks
- Stateflow chart with function-call output events
- Signal lines between the chart and each function-call subsystem port

Simulink functions in a Stateflow chart provide these advantages:

- No function-call subsystem blocks
- No output events
- No signal lines

Benefits of Using a Simulink Function to Access Simulink Blocks

The sections that follow compare two ways of defining a function that uses Simulink blocks.

Model Method Without a Simulink Function

You define a function-call subsystem in the Simulink model (see "Using Function-Call Subsystems" (Simulink). Use an output event in a Stateflow chart to call the subsystem, as shown.
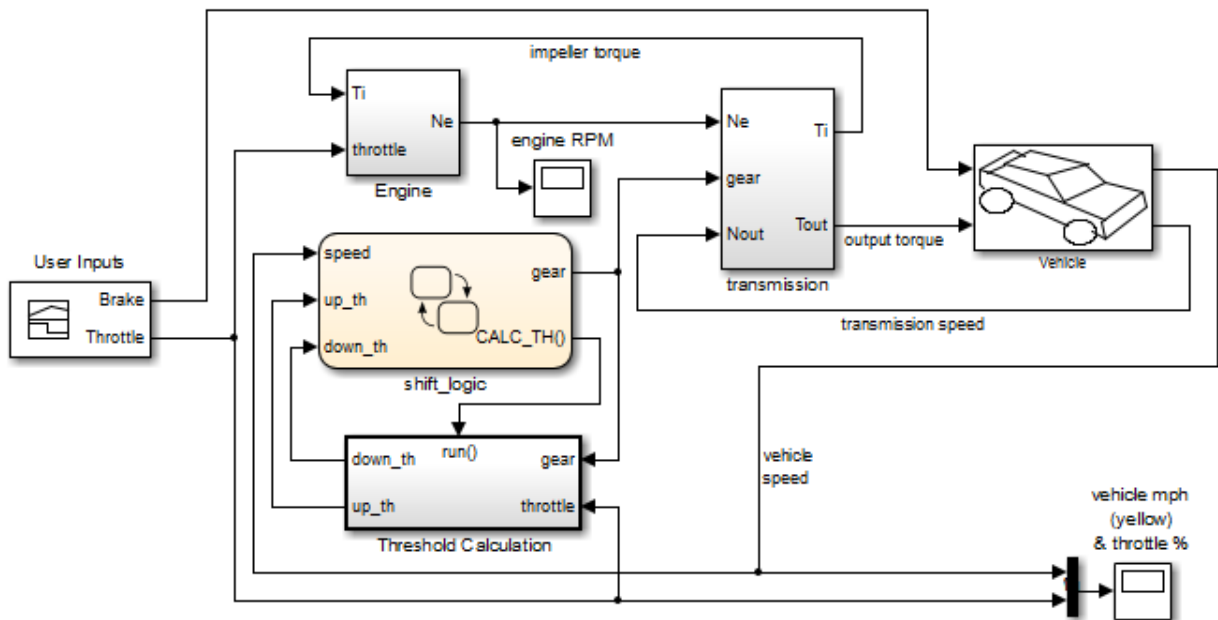
Model Method With a Simulink Function

You place one or more Simulink blocks in a Simulink function of a Stateflow chart. Use a function call to execute the blocks in that function, as shown.
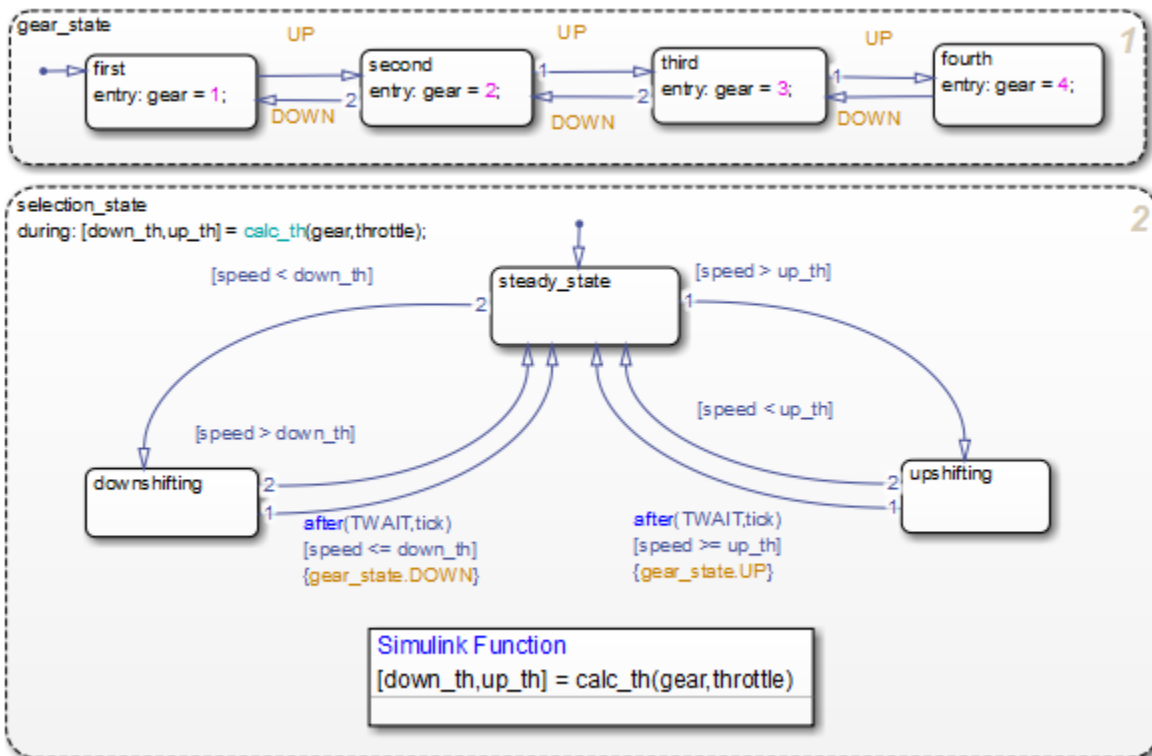
In the chart, the `during` action in `selection_state` contains a function call to `calc_th`, which is a function that contains Simulink blocks.

This modeling method minimizes the objects in your model.
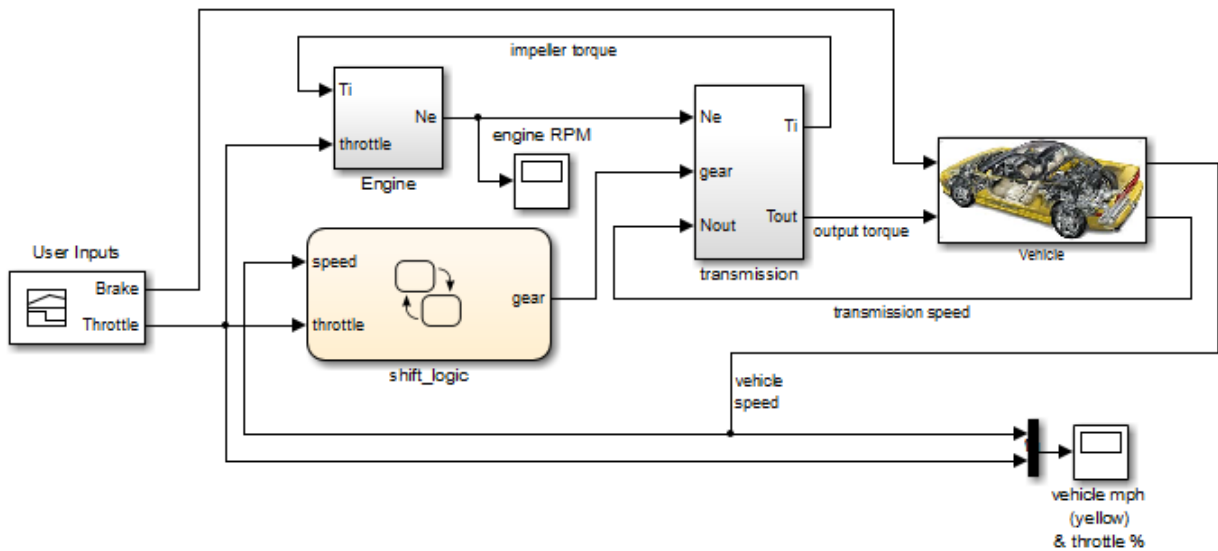
For more information, see "Improve Chart Design by Using Simulink Functions" on page 29-26.

Benefits of Using a Simulink Function to Schedule Execution of Multiple Controllers

The sections that follow compare two ways of scheduling execution of multiple controllers.

Model Method Without Simulink Functions

You define each controller as a function-call subsystem block and use output events in a Stateflow chart to schedule execution of the subsystems, as shown in the `sf_temporal_logic_scheduler` model.
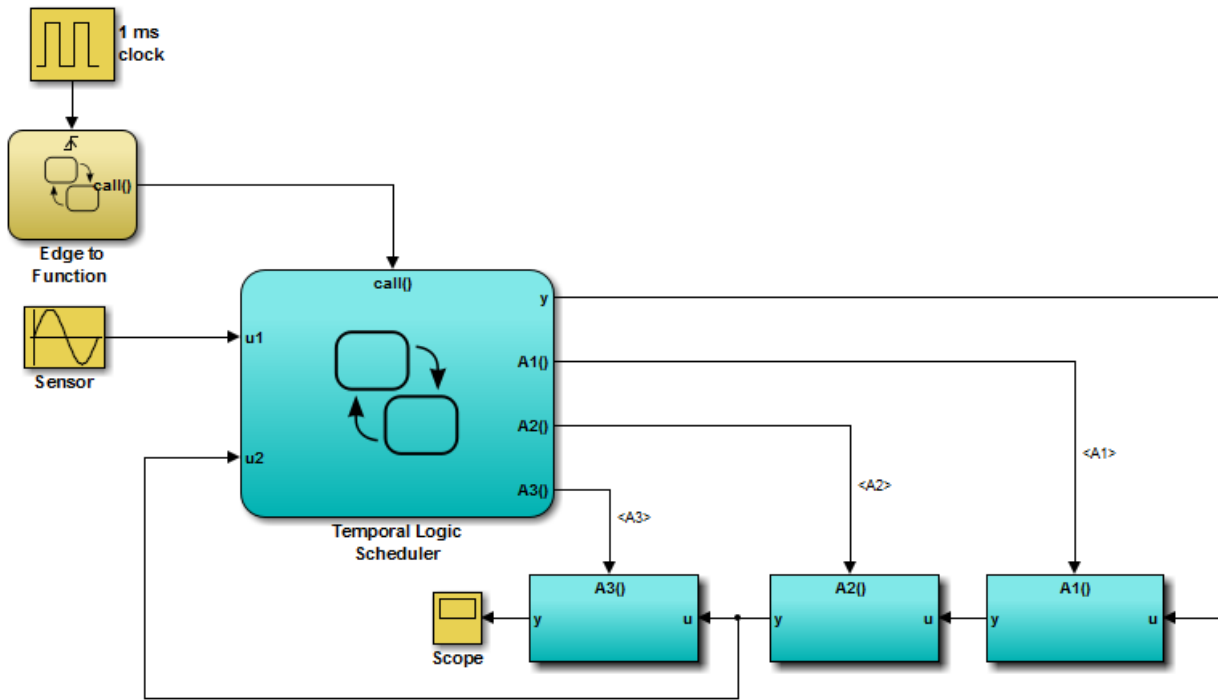
Model Method With Simulink Functions

You define each controller as a Simulink function in a Stateflow chart and use function calls to schedule execution of the subsystems, as shown in the `sf_temporal_logic_scheduler_with_sl_fcns` model.
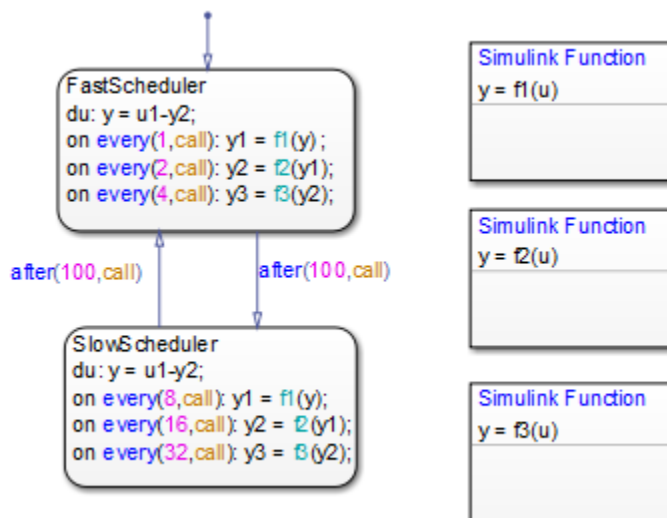
This modeling method minimizes the objects in your model.
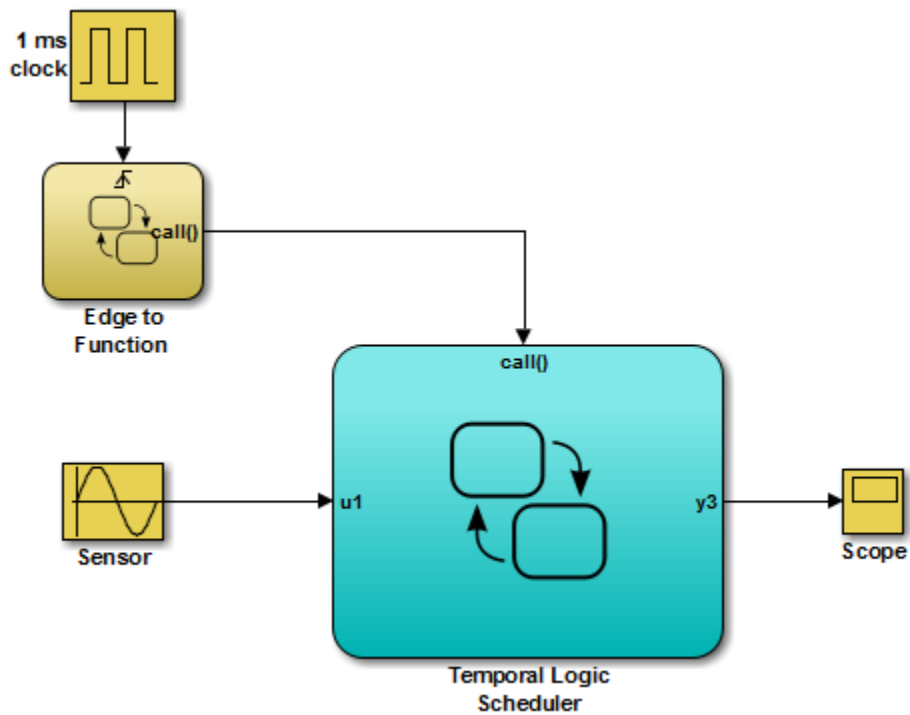
See Also

More About

- "Benefits of Using a Simulink Function to Access Simulink Blocks" on page 29-9
- "Benefits of Using a Simulink Function to Schedule Execution of Multiple Controllers" on page 29-12
- "Schedule Execution of Multiple Controllers" on page 29-36

Define a Simulink Function in a Stateflow Chart

A Simulink function is a graphical object that you fill with Simulink blocks and call in the actions of states and transitions. Simulink functions are supported only in Stateflow charts in Simulink models. For more information, see "Reuse Simulink Components in Stateflow Charts" on page 29-2.

Task 1: Add a Function to the Chart

Follow these steps to add a Simulink function to the chart:

1 Click the Simulink function icon in the Stateflow Editor toolbar:

2 Move your pointer to the location for the new Simulink function in your chart and click to insert the function box.

Tip You can also drag the function from the toolbar.

---

3 Enter the function signature.

The function signature specifies a name for your function and the formal names for the arguments and return values. A signature has this syntax:

[*r_1*, *r_2*,..., *r_n*] = simfcn(*a_1*, *a_2*,..., *a_n*)

where `simfcn` is the name of your function, *a_1*, *a_2*, ..., *a_n* are formal names for the arguments, and *r_1*, *r_2*, ..., *r_n* are formal names for the return values.

Note This syntax is the same as what you use for graphical functions, truth tables, and MATLAB functions. You can define arguments and return values as scalars, vectors, or matrices of any data type.

---

4 Click outside the function box.

The following example shows a Simulink function that has the name `sim_fcn`, which takes three arguments (`a`, `b`, and `c`) and returns two values (`x` and `y`).

Simulink Function
[x, y] = sim_fcn(a, b, c)
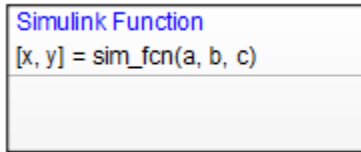
Note You can also create and edit a Simulink function by using API methods.

Task 2: Define the Subsystem Elements of the Simulink Function

Follow these steps to define the subsystem elements of the Simulink function:

1   Double-click the Simulink function box.

    The contents of the subsystem appear: input and output ports that match the function signature and a single function-call trigger port.

2   Add Simulink blocks to the subsystem.

3   Connect the input and output ports to each block.

Note You cannot delete the trigger port in the function.

The following example shows the subsystem elements for a Simulink function.
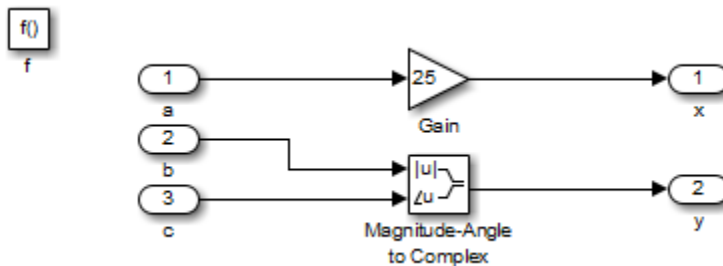
Task 3: Configure the Function Inputs

Follow these steps to configure inputs for the Simulink function:

1 Configure the input ports.

    a Double-click an input port to open the Block Parameters dialog box.

    b In the **Signal Attributes** pane, enter the size and data type.

       For example, you can specify a size of `[2 3]` for a 2-by-3 matrix and a data type of `uint8`.

2 Click **OK**.

Note An input port of a Simulink function cannot inherit size or data type. Therefore, you define the size and data type of an input that is not scalar data of type `double`. However, an output port can inherit size and data type.

See Also

More About

- "Reuse Simulink Components in Stateflow Charts" on page 29-2
- "Why Use a Simulink Function in a Stateflow Chart?" on page 29-9
- "Share Functions Across Simulink and Stateflow" on page 29-7

Bind a Simulink Function to a State

A Simulink function is a graphical object that you fill with Simulink blocks and call in the actions of states and transitions. Simulink functions are supported only in Stateflow charts in Simulink models. For more information, see "Reuse Simulink Components in Stateflow Charts" on page 29-2.

Bind Behavior of a Simulink Function

When a Simulink function resides inside a state, the function binds to that state. Binding results in the following behavior:

- Function calls can occur only in state actions and on transitions within the state and its substates.
- When the state is entered, the function is enabled.
- When the state is exited, the function is disabled.

For example, the following Stateflow chart shows a Simulink function that binds to a state.
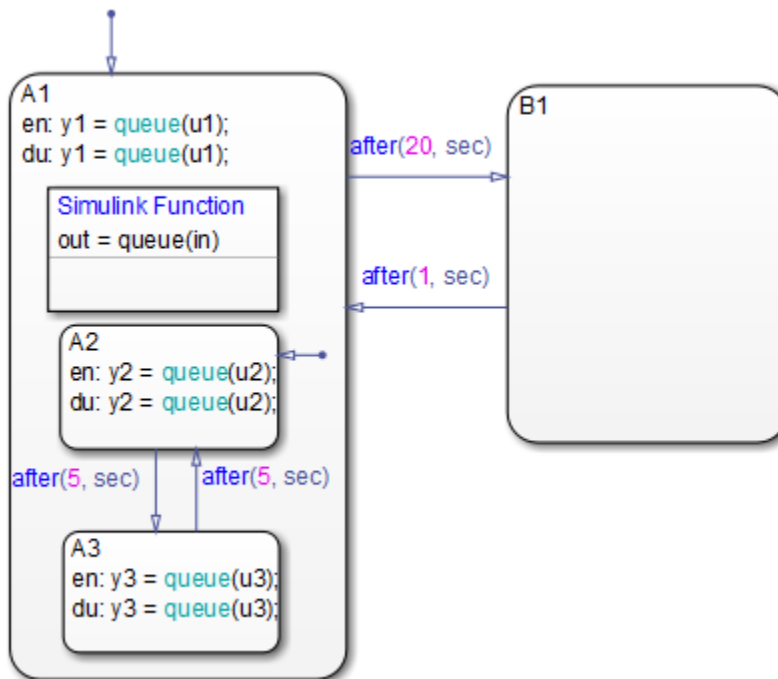
Because the function `queue` resides in state `A1`, the function binds to that state.

- State `A1` and its substates `A2` and `A3` can call `queue`, but state `B1` cannot.
- When state `A1` is entered, `queue` is enabled.
- When state `A1` is exited, `queue` is disabled.

Control Subsystem Variables When the Simulink Function Is Disabled

If a Simulink function binds to a state, you can hold the subsystem variables at their values from the previous execution or reset the variables to their initial values. Follow these steps:

1 In the Simulink function, double-click the trigger port to open the Block Parameters dialog box.

2 Select an option for **States when enabling**.

Option	Description	Reference Section
held	Holds the values of the subsystem variables from the previous execution	"How the Function Behaves When Variables Are Held" on page 29-23
reset	Resets the subsystem variables to their initial values	"How the Function Behaves When Variables Are Reset" on page 29-24

Example of Binding a Simulink Function to a State

This example shows how a Simulink function behaves when bound to a state.
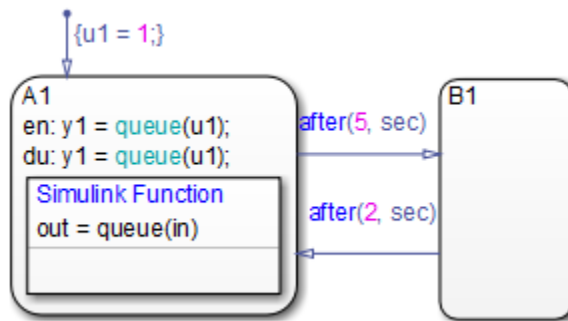
The function queue contains a block diagram that increments a counter by 1 each time the function executes.
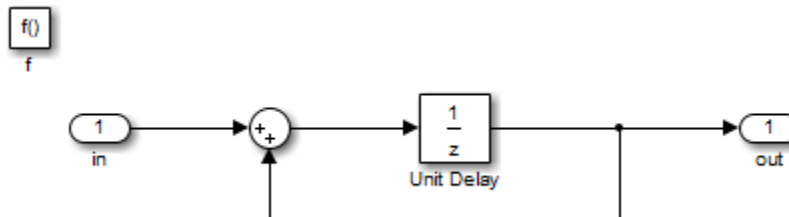
The Block Parameters dialog box for the trigger port appears as follows.
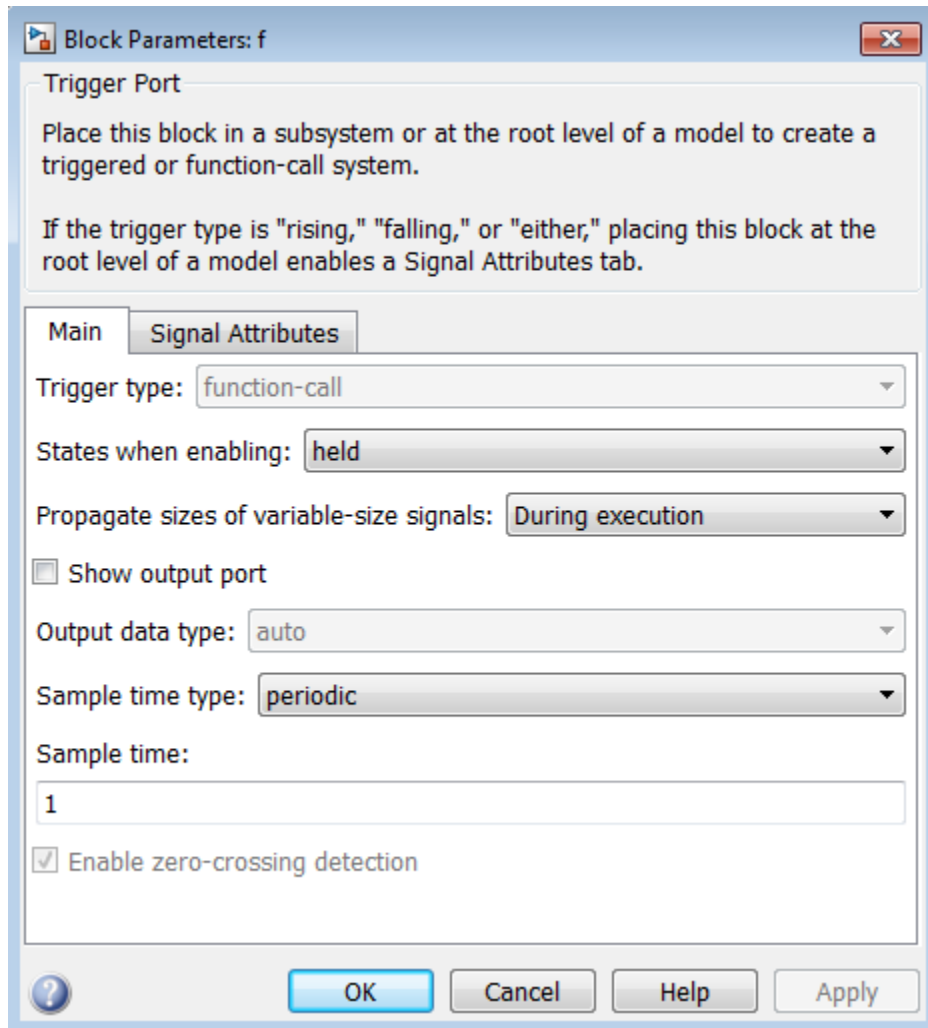
In the dialog box, setting **Sample time type** to `periodic` enables the **Sample time** field, which defaults to 1. These settings tell the function to execute for each time step specified in the **Sample time** field while the function is enabled.

Simulation Behavior of the Chart
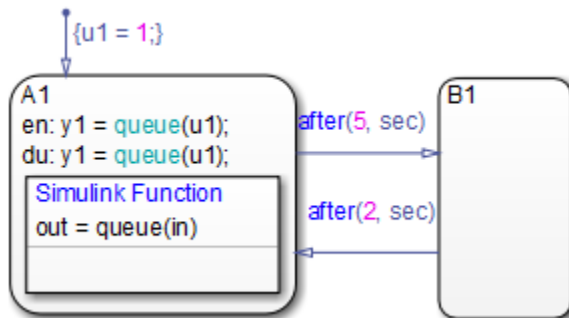
When you simulate the chart, the following actions occur.

1   The default transition to state A1 occurs, which includes setting local data u1 to 1.

2   When A1 is entered, the function queue is enabled.

3   Function calls to queue occur until the condition `after(5, sec)` is true.

4   The transition from state A1 to B1 occurs.

5   When A1 is exited, the function queue is disabled.

6   After two more seconds pass, the transition from B1 to A1 occurs.

7   Steps 2 through 6 repeat until the simulation ends.

How the Function Behaves When Variables Are Held

If you set **States when enabling** to `held`, the output y1 is as follows.
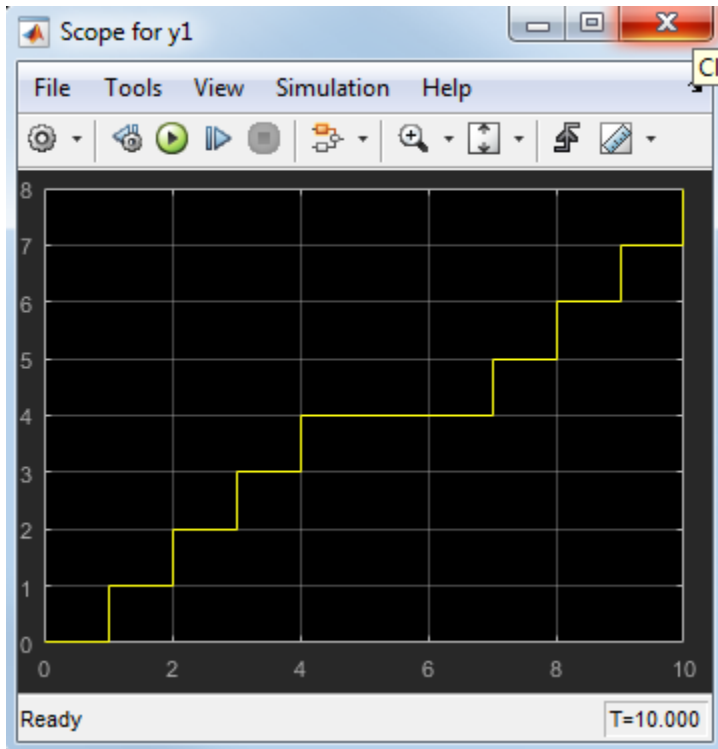
When state A1 becomes inactive at t = 5, the Simulink function holds the counter value. When A1 is active again at t = 7, the counter has the same value as it did at t = 5. Therefore, the output y1 continues to increment over time.

How the Function Behaves When Variables Are Reset

If you set **States when enabling** to reset, the output y1 is as follows.
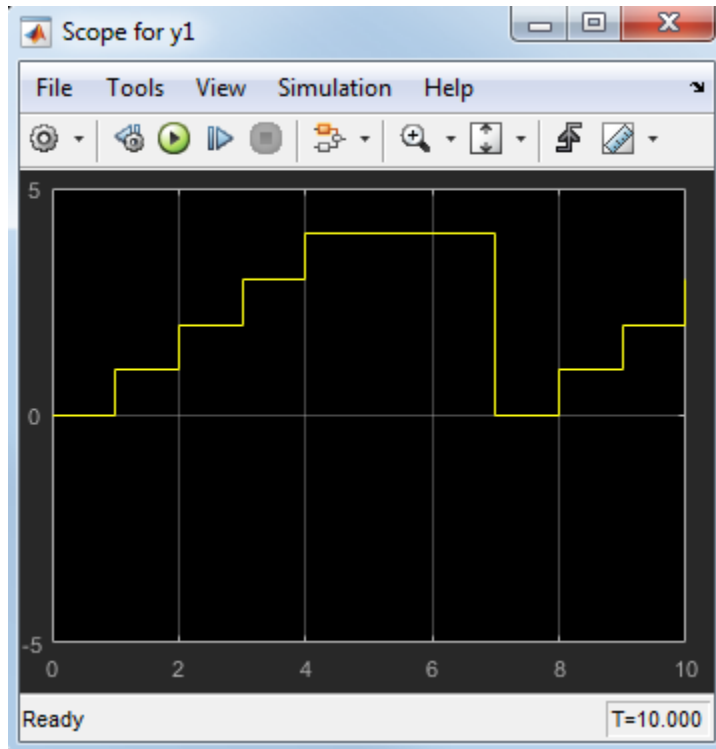
When state A1 becomes inactive at t = 5, the Simulink function does *not* hold the counter value. When A1 is active again at t = 7, the counter resets to zero. Therefore, the output y1 resets too.

See Also

More About

- "Reuse Simulink Components in Stateflow Charts" on page 29-2
- "Why Use a Simulink Function in a Stateflow Chart?" on page 29-9

Improve Chart Design by Using Simulink Functions

A Simulink function is a graphical object that you fill with Simulink blocks and call in the actions of states and transitions. Simulink functions are supported only in Stateflow charts in Simulink models. For more information, see "Reuse Simulink Components in Stateflow Charts" on page 29-2.

Goal of the Tutorial

The goal of this tutorial is to use a Simulink function in a Stateflow chart to improve the design of a model named `old_sf_car`.
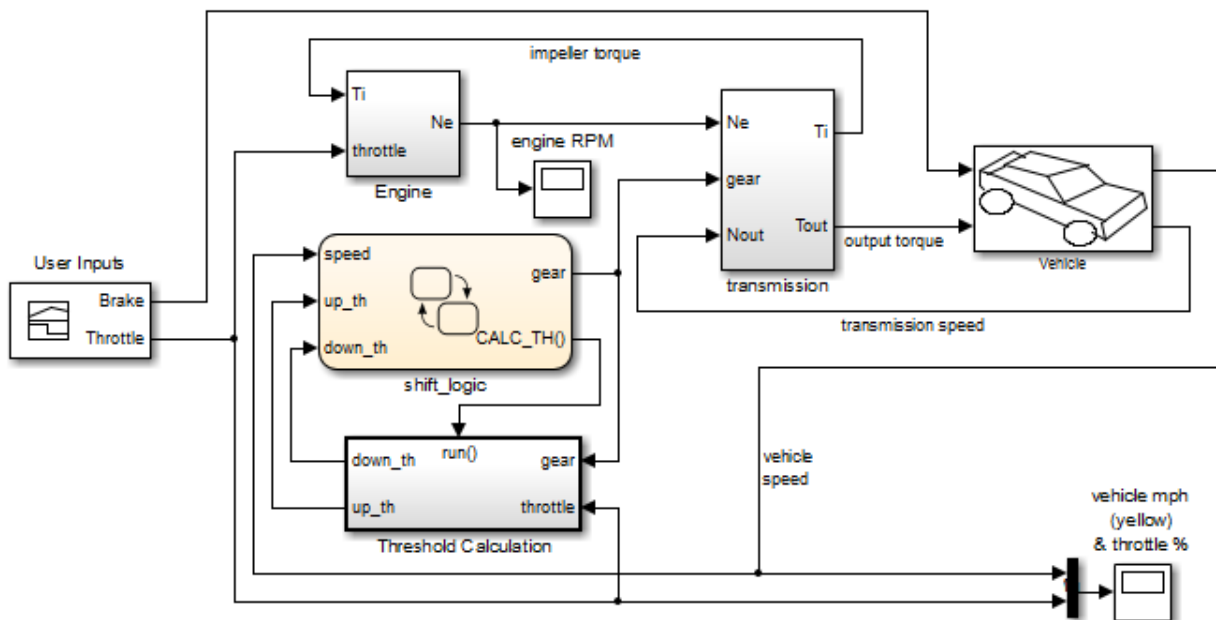
Rationale for Improving the Model Design

The `old_sf_car` model contains a function-call subsystem named Threshold Calculation and a Stateflow chart named shift_logic. The two blocks interact as follows:

- The chart broadcasts the output event `CALC_TH` to trigger the function-call subsystem.

- The subsystem uses lookup tables to interpolate two values for the shift_logic chart.
- The subsystem outputs (`up_th` and `down_th`) feed directly into the chart as inputs.

  No other blocks in the model access the subsystem outputs.

You can replace a function-call subsystem with a Simulink function in a chart when:

- The subsystem performs calculations required by the chart.
- Other blocks in the model do not need access to the subsystem outputs.

Edit a Model to Use a Simulink Function

The sections that follow describe how to replace a function-call subsystem in a Simulink model with a Simulink function in a Stateflow chart. This procedure reduces the number of objects in the model while retaining the same simulation results.

Step	Task	Reference
1	Open the model.	"Open the Model" on page 29-27
2	Move the contents of the function-call subsystem into a Simulink function in the chart.	"Add a Simulink Function to the Chart" on page 29-29
3	Change the scope of specific chart-level data to `Local`.	"Change the Scope of Chart Data" on page 29-32
4	Replace the event broadcast with a function call.	"Update State Action in the Chart" on page 29-33
5	Verify that function inputs and outputs are defined.	"Add Data to the Chart" on page 29-33
6	Remove unused items in the model.	"Remove Unused Items in the Model" on page 29-34

Note To skip the conversion steps, open the model `sf_car`.

Open the Model

Open the model `old_sf_car`. If you simulate the model, you see these results in the two scopes.
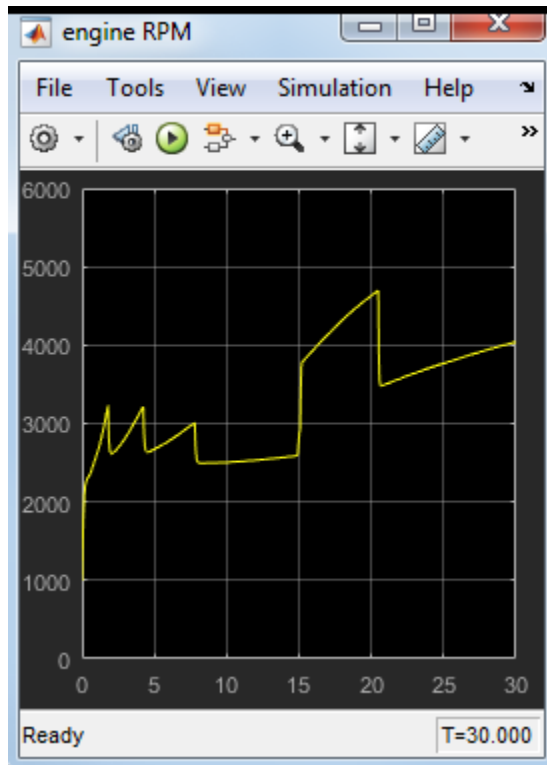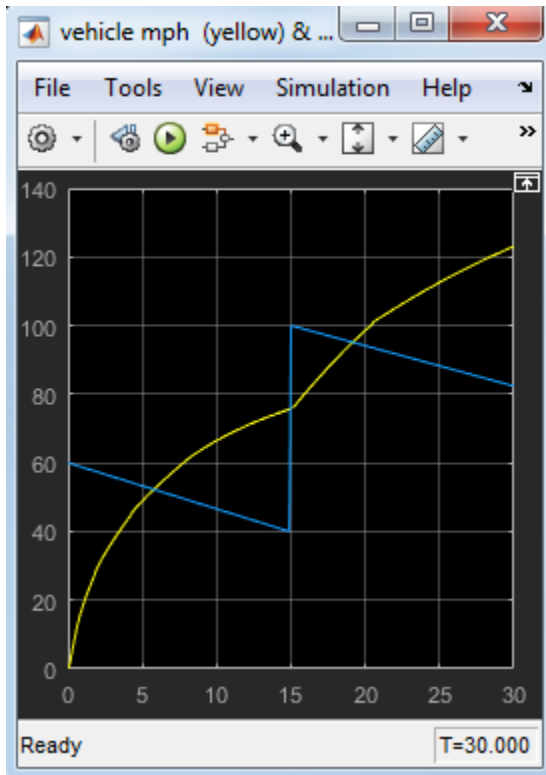
Add a Simulink Function to the Chart

Follow these steps to add a Simulink function to the shift_logic chart.

1   In the Simulink model, right-click the Threshold Calculation block in the lower left
    corner and select **Cut** from the context menu.
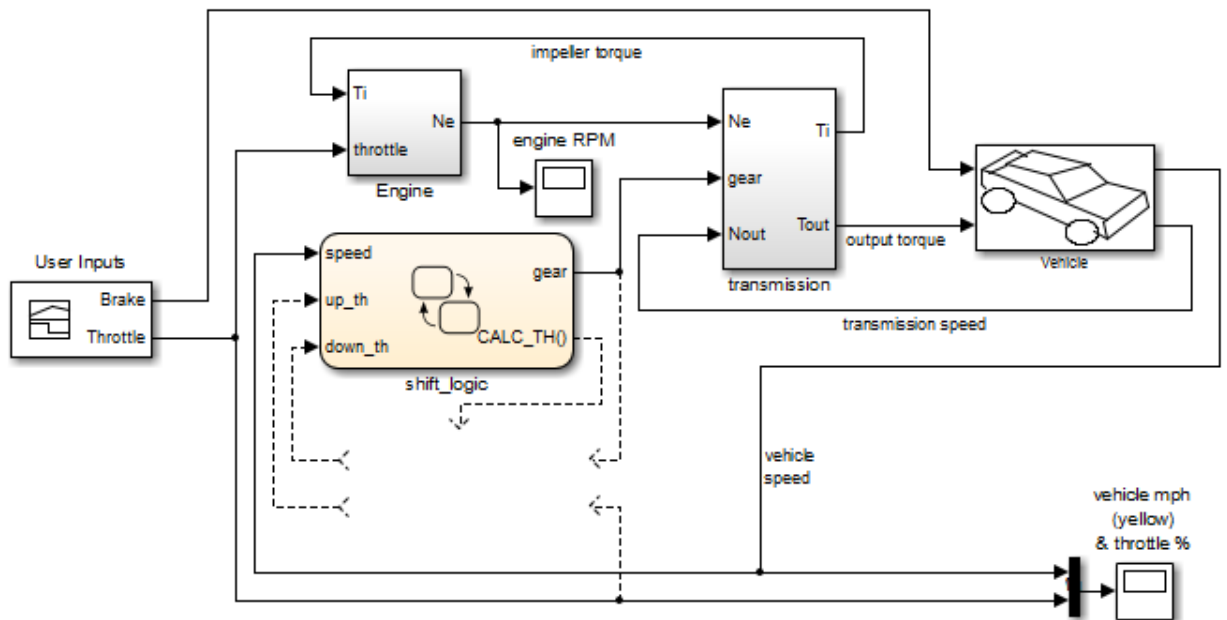
2   Open the shift_logic chart.

3   In the chart, right-click below `selection_state` and select **Paste** from the context menu.

4   Expand the new Simulink function so that the signature fits inside the function box.
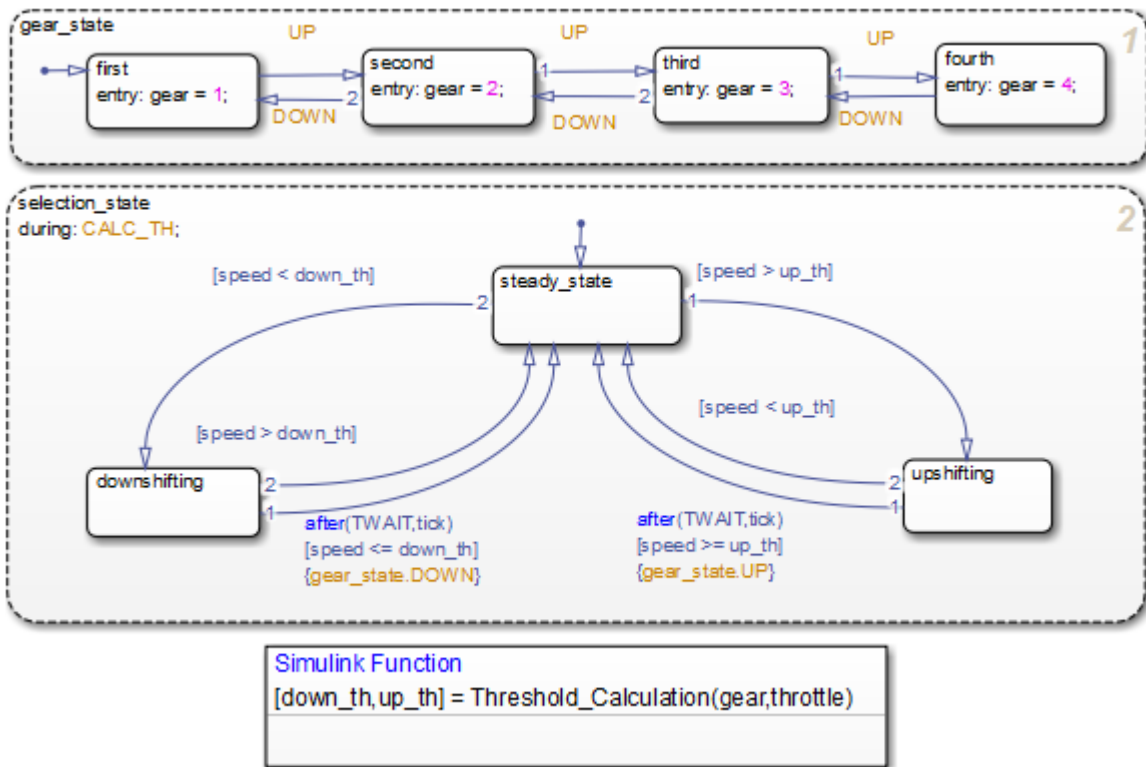
Tip To change the font size of a function, right-click the function box and select a new size from the **Font Size** menu.

5   Expand the border of `selection_state` to include the new function.
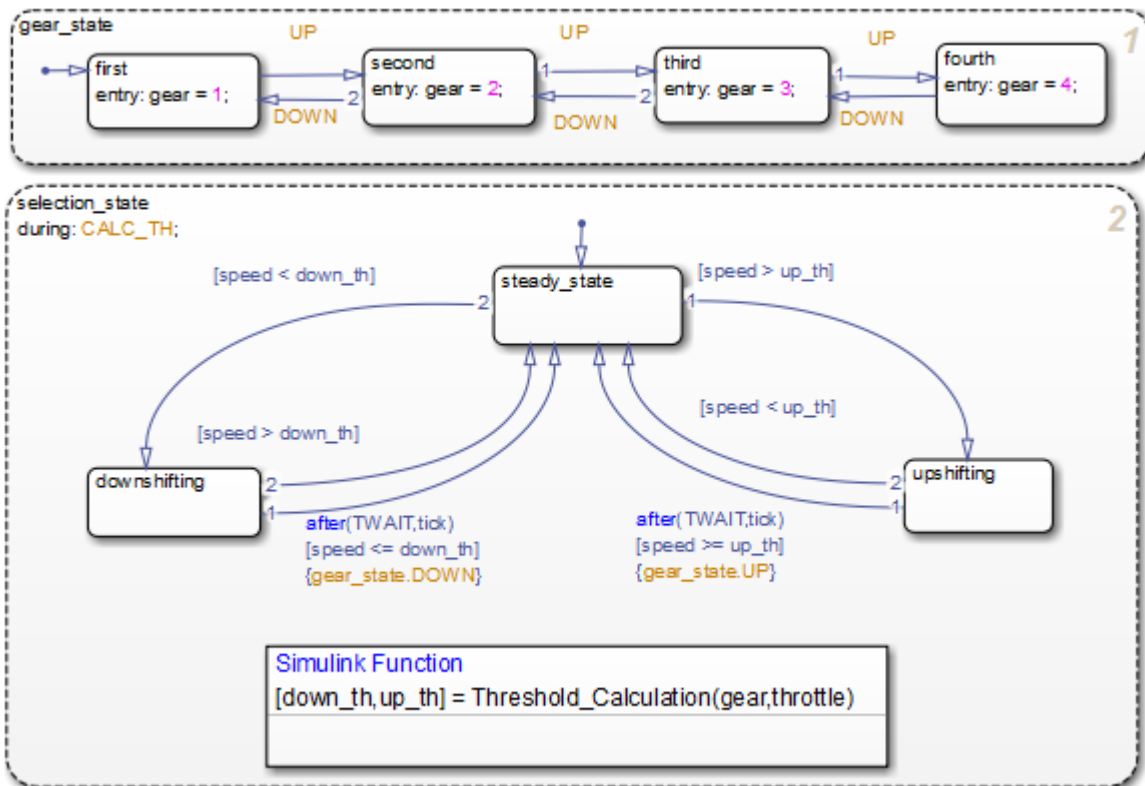
Note The function resides in this state instead of the chart level because no other state in the chart requires the function outputs `up_th` and `down_th`. See "Bind a Simulink Function to a State" on page 29-19.

6   Rename the Simulink function from `Threshold_Calculation` to `calc_threshold` by entering `[down_th, up_th] = calc_threshold(gear, throttle)` in the function box.

Change the Scope of Chart Data

In the Model Explorer, change the scope of chart-level data `up_th` and `down_th` to `Local` because calculations for those data now occur inside the chart.

Update State Action in the Chart

In the Stateflow Editor, change the `during` action in `selection_state` to call the Simulink function `calc_threshold`.

```
during: [down_th, up_th] = calc_threshold(gear, throttle);
```
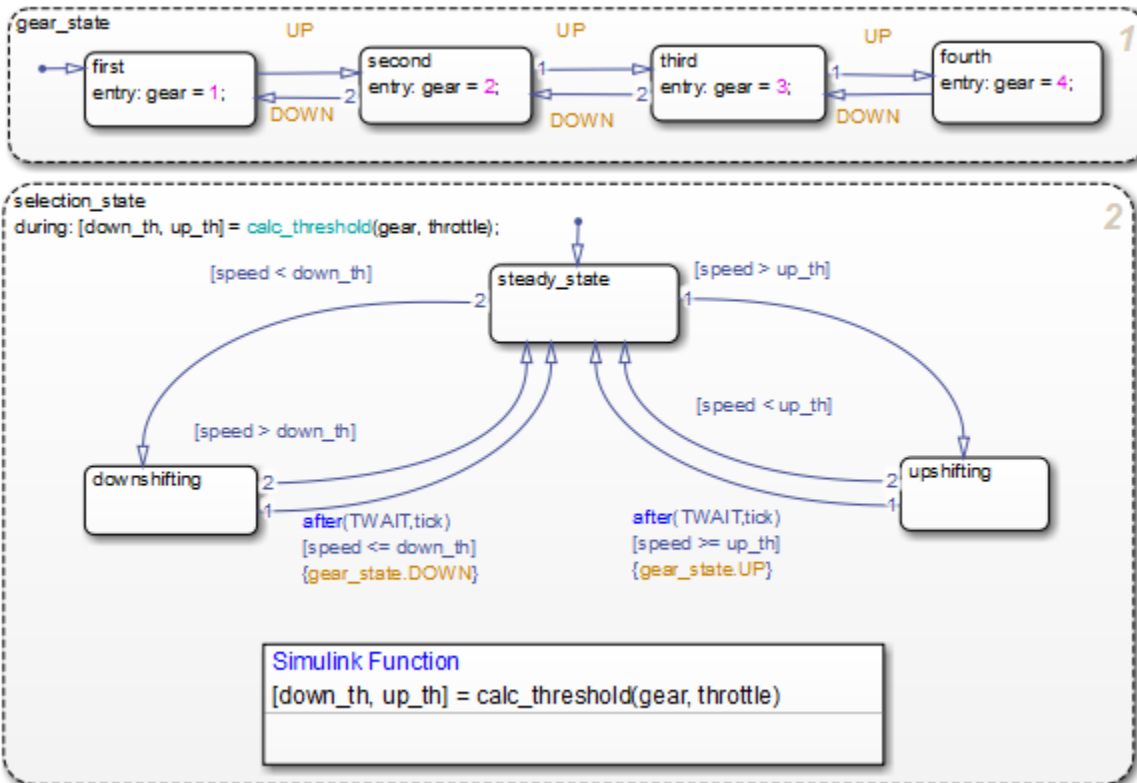

Add Data to the Chart

Because the function `calc_threshold` takes `throttle` as an input, you must define that data as a chart input. (For details, see "Add Stateflow Data" on page 9-2.)

1 Add input data `throttle` to the chart with a **Port** property of 1.

Using port 1 prevents signal lines from overlapping in the Simulink model.

2   In the Simulink model, add a signal line for `throttle` between the inport of the Engine block and the inport of the shift_logic chart.

Remove Unused Items in the Model

1   In the Model Explorer, delete the function-call output event `CALC_TH` because the Threshold Calculation block no longer exists.

2   Delete any dashed signal lines from your model.

Run the New Model

Your new model looks something like this:
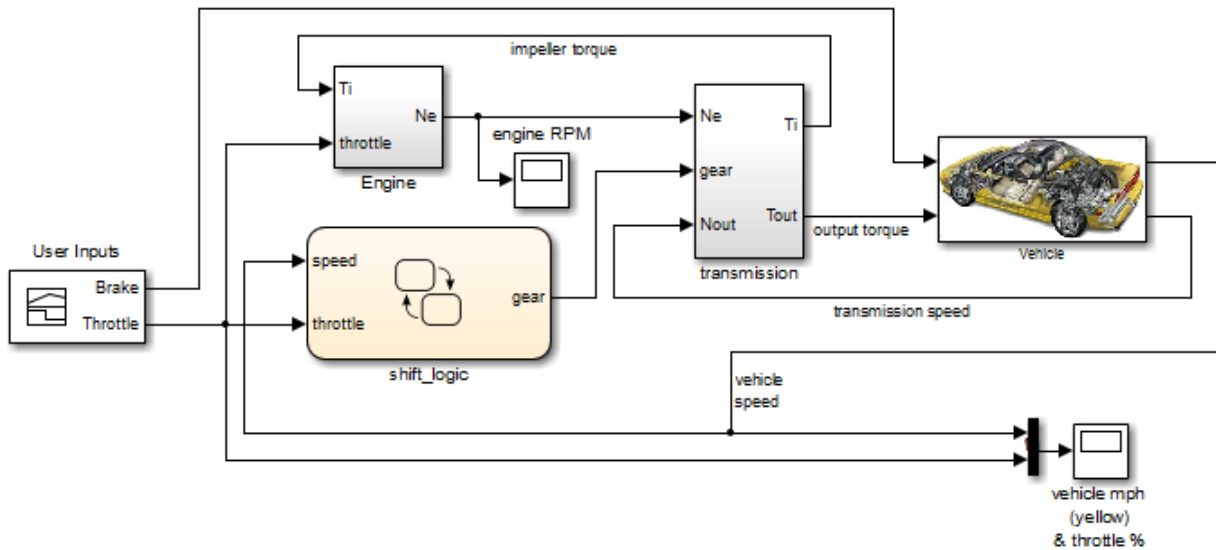
If you simulate the new model, the results match those of the original design.

See Also

More About

- "Reuse Simulink Components in Stateflow Charts" on page 29-2
- "Define a Simulink Function in a Stateflow Chart" on page 29-16

Schedule Execution of Multiple Controllers

A Simulink function is a graphical object that you fill with Simulink blocks and call in the actions of states and transitions. Simulink functions are supported only in Stateflow charts in Simulink models. For more information, see "Reuse Simulink Components in Stateflow Charts" on page 29-2.

Goal of the Tutorial

The goal of this tutorial is to use Simulink functions in a Stateflow chart to improve the design of a model named `sf_temporal_logic_scheduler`.
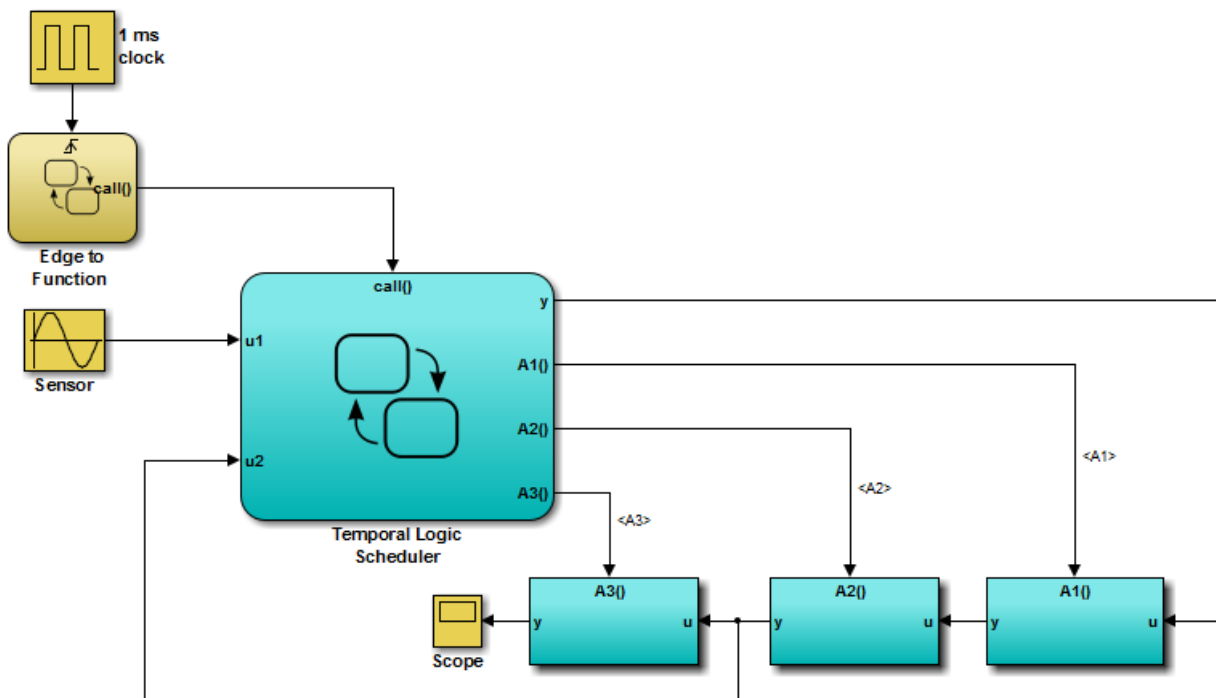
Rationale for Improving the Model Design

The `sf_temporal_logic_scheduler` model contains a Stateflow chart and three function-call subsystems. These blocks interact as follows:

- The chart broadcasts the output events A1, A2, and A3 to trigger the function-call subsystems.
- The subsystems A1, A2, and A3 execute at different rates defined by the chart.
- The subsystem outputs feed directly into the chart.

  No other blocks in the model access the subsystem outputs.

You can replace function-call subsystems with Simulink functions inside a chart when:

- The subsystems perform calculations required by the chart.
- Other blocks in the model do not need access to the subsystem outputs.

Edit a Model to Use Simulink Functions

The sections that follow describe how to replace function-call subsystem blocks in a Simulink model with Simulink functions in a Stateflow chart. This procedure reduces the number of objects in the model while retaining the same simulation results.

Step	Task	Reference
1	Open the model.	"Open the Model" on page 29-38
2	Move the contents of the function-call subsystems into Simulink functions in the chart.	"Add Simulink Functions to the Chart" on page 29-38
3	Change the scope of specific chart-level data to `Local`.	"Change the Scope of Chart Data" on page 29-41
4	Replace event broadcasts with function calls.	"Update State Actions in the Chart" on page 29-41
5	Verify that function inputs and outputs are defined.	"Add Data to the Chart" on page 29-42
6	Remove unused items in the model.	"Remove Unused Items in the Model" on page 29-42

Note To skip the conversion steps, you can access the new model directly.

Open the Model

Open the `sf_temporal_logic_scheduler` model. If you simulate the model, you see this result in the scope.
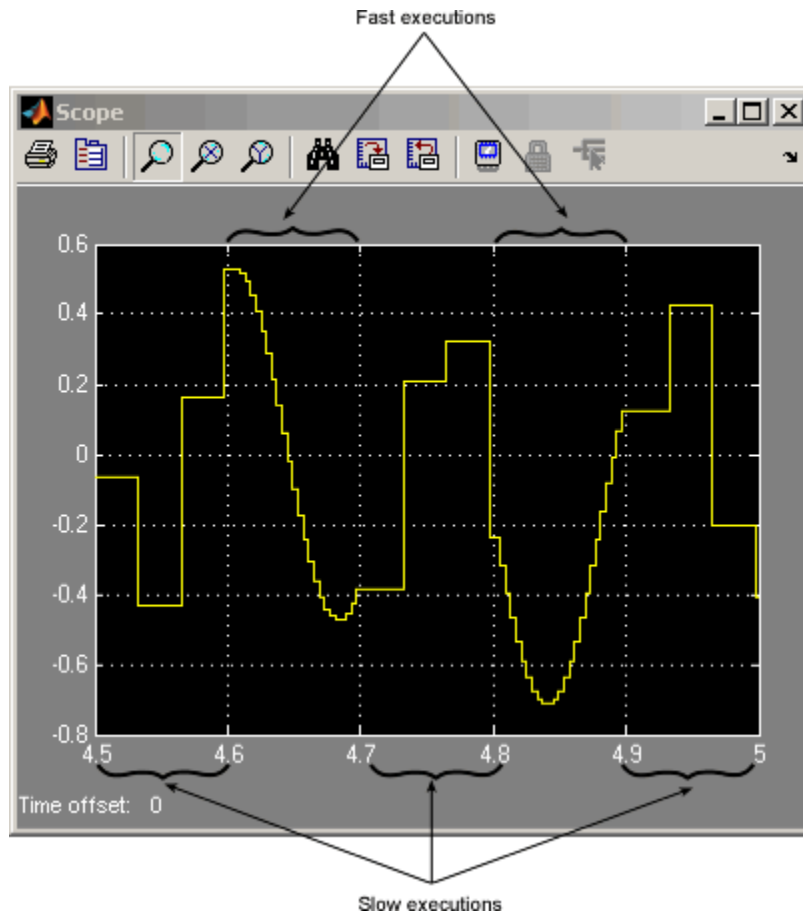
For more information, see "Schedule Subsystems to Execute at Specific Times" on page 26-22.

Add Simulink Functions to the Chart

Follow these steps to add Simulink functions to the Temporal Logic Scheduler chart.

1   In the Simulink model, right-click the A1 block in the lower right corner and select
Cut from the context menu.
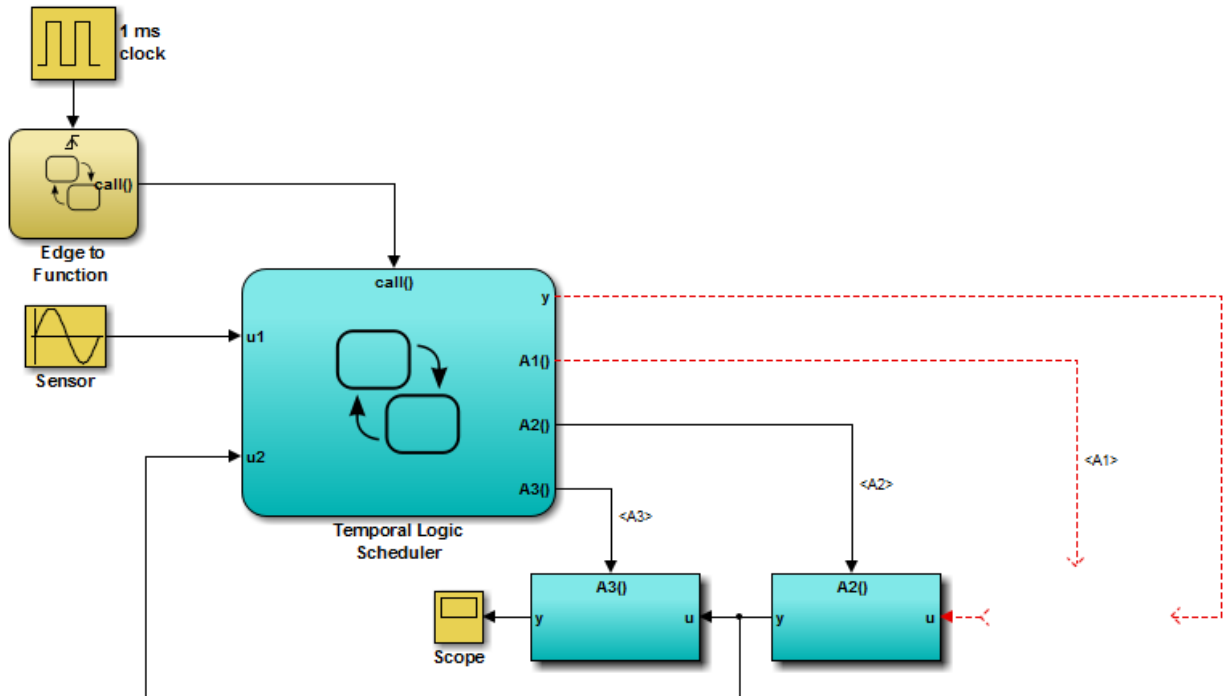
2   Open the Temporal Logic Scheduler chart.

3   In the chart, right-click outside any states and select **Paste** from the context menu.

4   Expand the new Simulink function so that the signature fits inside the function box.
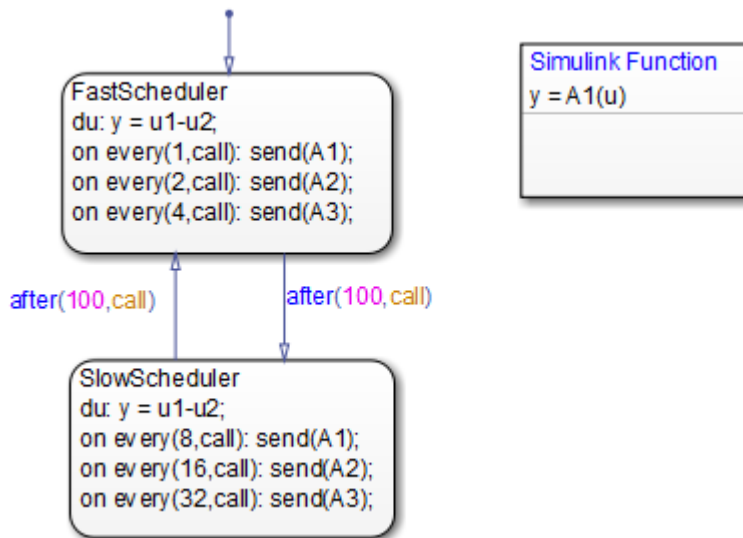
Tip To change the font size of a function, right-click the function box and select a new size from the **Font Size** menu.

---

5 Rename the Simulink function from A1 to f1 by entering y = f1(u) in the function box.

6 Repeat steps 1 through 5 for these cases:

- Copying the contents of A2 into a Simulink function named f2.
- Copying the contents of A3 into a Simulink function named f3.
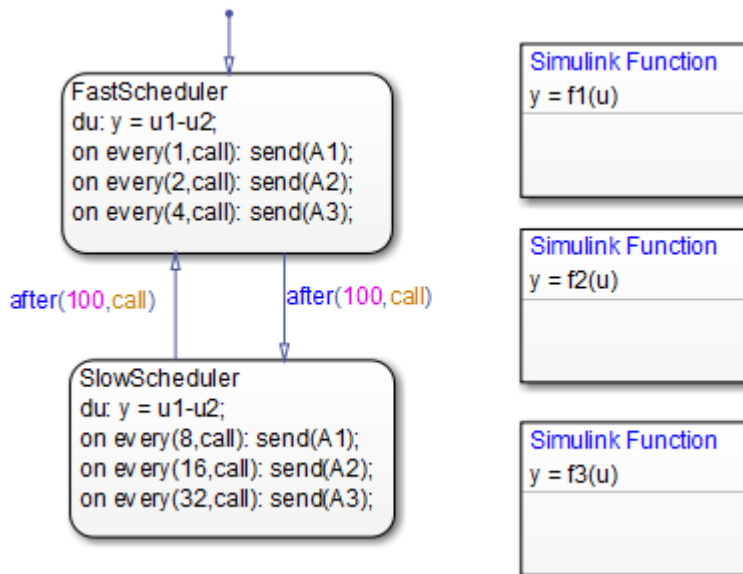
Note The new functions reside at the chart level because both states
`FastScheduler` and `SlowScheduler` require access to the function outputs.

Change the Scope of Chart Data

In the Model Explorer, change the scope of chart-level data `y` to `Local` because the
calculation for that data now occurs inside the chart.

Update State Actions in the Chart

In the Stateflow Editor, you can replace event broadcasts in state actions with function
calls.

1 Edit the state actions in `FastScheduler` and `SlowScheduler` to call the Simulink
   functions `f1`, `f2`, and `f3`.
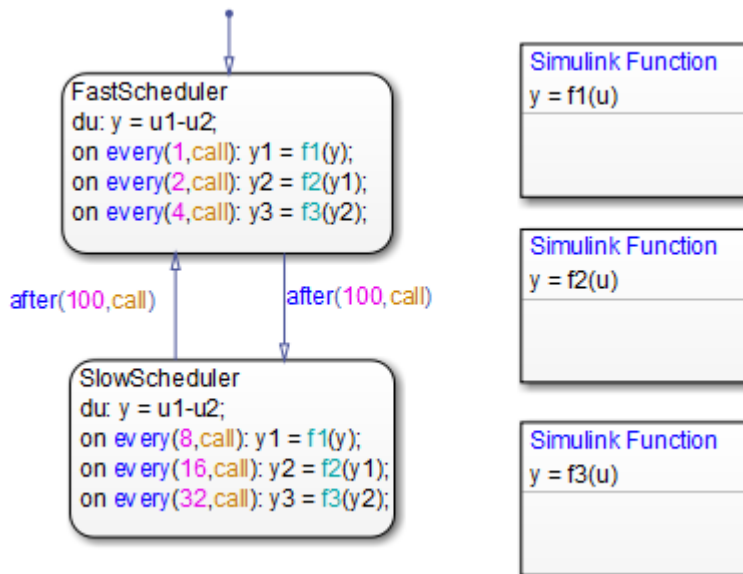
2 In both states, update each `during` action as follows.

```
du: y = u1-y2;
```

Add Data to the Chart

For the `on every` state actions of `FastScheduler` and `SlowScheduler`, define three data. (For details, see "Add Stateflow Data" on page 9-2.)

1 Add local data `y1` and `y2` to the chart.

2 Add output data `y3` to the chart.

3 In the model, connect the outport for `y3` to the inport of the scope.

Tip To flip the Scope block, right-click and select **Rotate & Flip > Flip Block** from the context menu.

Remove Unused Items in the Model

1 In the Model Explorer, delete output events A1, A2, and A3 and input data u2 because the function-call subsystems no longer exist.

2   Delete any dashed signal lines from your model.

Run the New Model

Your new model looks something like this:

If you simulate the new model, the results match those of the original design.

See Also

More About

- "Reuse Simulink Components in Stateflow Charts" on page 29-2
- "Define a Simulink Function in a Stateflow Chart" on page 29-16

30

Build Targets

Simulate a Chart That Calls Custom Code

Guidelines for Simulation

When you simulate a Stateflow block in a Simulink model, use these guidelines to choose the right procedure.

Do this step...	When...
Speed up simulation	You have a large model with many blocks. See "Speed Up Simulation" on page 30-17.
Include custom code	You want to take advantage of legacy code that augments model capabilities and also include custom variables and functions that you share between your custom code and Stateflow generated code. See "Access Custom Code Variables and Functions in Stateflow Charts" on page 30-15.
Include custom code only for library charts	You want to provide custom code in a portable, self-contained library for use in multiple models.

See Also

More About

- "Speed Up Simulation" on page 30-17
- "Set Up Your Own Target Compiler"

Access Custom C++ Code in Stateflow Charts

You can integrate custom code written in C or C++ with Stateflow charts in Simulink models. By sharing data and functions between your custom code and your Stateflow chart, you can augment the capabilities of Stateflow and leverage the software to take advantage of your preexisting code. For more information, see "Reuse Custom Code in Stateflow Charts" on page 30-26.

Task 1: Prepare Code Files

Prepare your custom C++ code for simulation as follows:

1   Add a C function wrapper to your custom code. This wrapper function executes the C++ code that you are including.

    The C function wrapper should have this form:

```
int my_c_function_wrapper()
{
    .
    .
    .
    //C++ code
    .
    .
    .
    return result;
}
```

2   Create a header file that prototypes the C function wrapper in the previous step.

    The header file should have this form:

```
int my_c_function_wrapper();
```

    The value `_cplusplus` exists if your compiler supports C++ code. The `extern "C"` wrapper specifies C linkage with no name mangling.

Task 2: Include Custom C++ Source and Header Files for Simulation

To include custom C++ code for simulation, you must configure your simulation target and select C++ as the custom code language:

1 Open the Model Configuration Parameters dialog box.
2 In the Model Configuration Parameters dialog box, select the **Simulation Target** pane.
3 Add your custom header file in the **Header file** subpane. Click **Apply**.
4 Add your custom C++ files in the **Source files** subpane. Click **Apply**.
5 In the Model Configuration Parameters dialog box, select the **Code Generation** pane.
6 Select C++ from the **Language** menu.
7 Click **OK**.

Task 3: Choose a C++ 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/`.

Task 4: Simulate the Model

Simulate your model by clicking the play button in the toolbar of the editor.

For information on setting simulation options using the command-line API, see "Command-Line API to Set Simulation and Code Generation Parameters" on page 30-19.

See Also

More About
*   "Reuse Custom Code in Stateflow Charts" on page 30-26
*   "C++ Code Generation and Integration in Stateflow"

Access Custom C Code in Nonlibrary Charts

You can integrate custom code written in C or C++ with Stateflow charts in Simulink models. By sharing data and functions between your custom code and your Stateflow chart, you can augment the capabilities of Stateflow and leverage the software to take advantage of your preexisting code. For more information, see "Reuse Custom Code in Stateflow Charts" on page 30-26.

Task 1: Include Custom C Code in the Simulation Target

Specify custom code options in the simulation target for your model:

1   Open the Model Configuration Parameters dialog box.

2   In the Model Configuration Parameters dialog box, select the **Simulation Target** pane.

The custom code options appear.
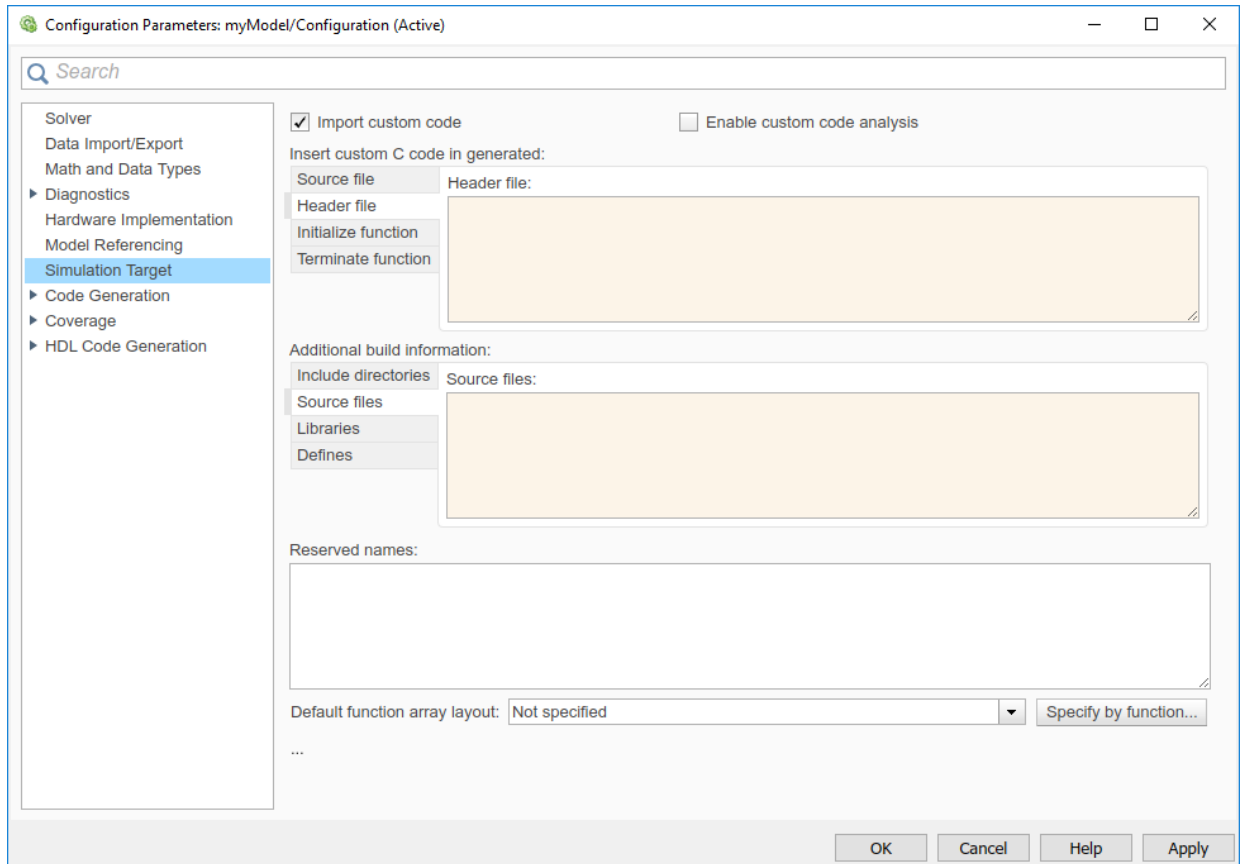
3 Specify your custom code in the subpanes.

Follow the guidelines in "Specify Relative Paths to Your Custom Code" on page 30-24.

- **Source file** — Enter code lines to include at the top of a generated source code file. These code lines appear at the top of the generated `model.c` source file, outside of any function.

  For example, you can include `extern int` declarations for global variables.

- **Header file** — Enter code lines to include at the top of the generated `model.h` header file that declares custom functions and data in the generated code. These

code lines appear at the top of all generated source code files and are visible to all generated code.

Note When you include a custom header file, you must enclose the file name in double quotes. For example, #include ''sample_header.h'' is a valid declaration for a custom header file.

Since the code you specify in this option appears in multiple source files that link into a single binary, limitations exist on what you can include. For example, do not include a global variable definition such as int x; or a function body such as

```
void myfun(void)
{
...
}
```

These code lines cause linking errors because their symbol definitions appear multiple times in the source files of the generated code. You can, however, include extern declarations of variables or functions such as extern int x; or extern void myfun(void);.

- **Initialize function** — Enter code statements that execute once at the start of simulation. Use this code to invoke functions that allocate memory or perform other initializations of your custom code.

- **Terminate function** — Enter code statements that execute at the end of simulation. Use this code to invoke functions that free memory allocated by the custom code or perform other cleanup tasks.

- **Include directories** — Enter a space-separated list of the folder paths that contain custom header files that you include either directly (see **Header file** option) or indirectly in the compiled target.

- **Source files** — Enter a list of source files to compile and link into the target. You can separate source files with a comma, a space, or a new line.

- **Libraries** — Enter a space-separated list of static libraries that contain custom object code to link into the target.

4 Click **OK**.

Tip If you want to rebuild the target to include custom code changes, select **Code > C/C ++ Code > Build Model** in the Stateflow Editor.

Task 2: Simulate the Model

Simulate your model by clicking the play button in the toolbar of the editor.

For information on setting simulation options using the command-line API, see "Command-Line API to Set Simulation and Code Generation Parameters" on page 30-19.

See Also

More About

- "Reuse Custom Code in Stateflow Charts" on page 30-26
- "Include Custom C Code Functions and Structures"
- "Share String Data with Custom C Code" on page 20-17
- "Integrate Custom Structures in Stateflow Charts" on page 25-11

Access Custom C Code in Library Charts

You can integrate custom code written in C or C++ with Stateflow charts in Simulink models. By sharing data and functions between your custom code and your Stateflow chart, you can augment the capabilities of Stateflow and leverage the software to take advantage of your preexisting code. For more information, see "Reuse Custom Code in Stateflow Charts" on page 30-26.

Task 1: Include Custom C Code in Simulation Targets for Library Models

Specify custom code options in the simulation target for each library model that contributes a chart to the main model:

1   In the Stateflow Editor, select **Simulation** > **Debug** > **Simulation Target For MATLAB & Stateflow**.

The Model Configuration Parameters dialog box appears.
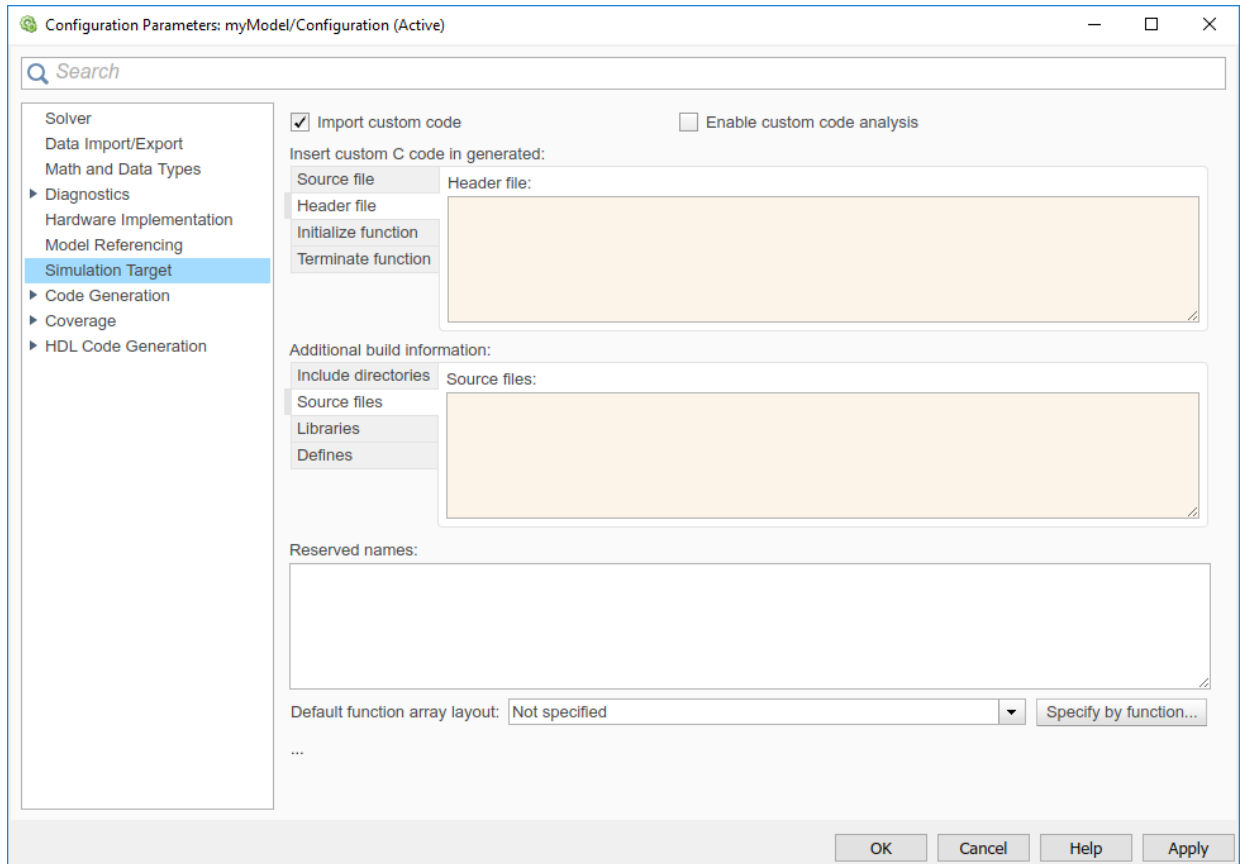
2   Select **Use local custom code settings (do not inherit from main model)**.

This step ensures that each library model retains its own custom code settings during simulation.

3   Specify your custom code in the subpanes.

Follow the guidelines in "Specify Relative Paths to Your Custom Code" on page 30-24.

Note See "Task 1: Include Custom C Code in the Simulation Target" on page 30-5 for descriptions of the custom code options.

---

4   Click **OK**.

Task 2: Simulate the Model

Simulate your model by clicking the play button in the toolbar of the editor.

For information on setting simulation options using the command-line API, see "Command-Line API to Set Simulation and Code Generation Parameters" on page 30-19.

Note You cannot simulate only the Stateflow blocks in a library model. You must first create a link to the library block in your main model and then simulate the main model.

See Also

More About

- "Reuse Custom Code in Stateflow Charts" on page 30-26
- "Include Custom C Code Functions and Structures"
- "Share String Data with Custom C Code" on page 20-17
- "Integrate Custom Structures in Stateflow Charts" on page 25-11

Access Custom C Code in All Charts in Model

You can integrate custom code written in C or C++ with Stateflow charts in Simulink models. By sharing data and functions between your custom code and your Stateflow chart, you can augment the capabilities of Stateflow and leverage the software to take advantage of your preexisting code. For more information, see "Reuse Custom Code in Stateflow Charts" on page 30-26.

To integrate custom C code that applies to all charts for simulation, perform the tasks that follow.

Task 1: Include Custom C Code in the Simulation Target for the Main Model

Specify custom code options in the simulation target for your main model:

1 Open the Model Configuration Parameters dialog box.
2 In the Model Configuration Parameters dialog box, select the **Simulation Target** pane.

The custom code options appear.
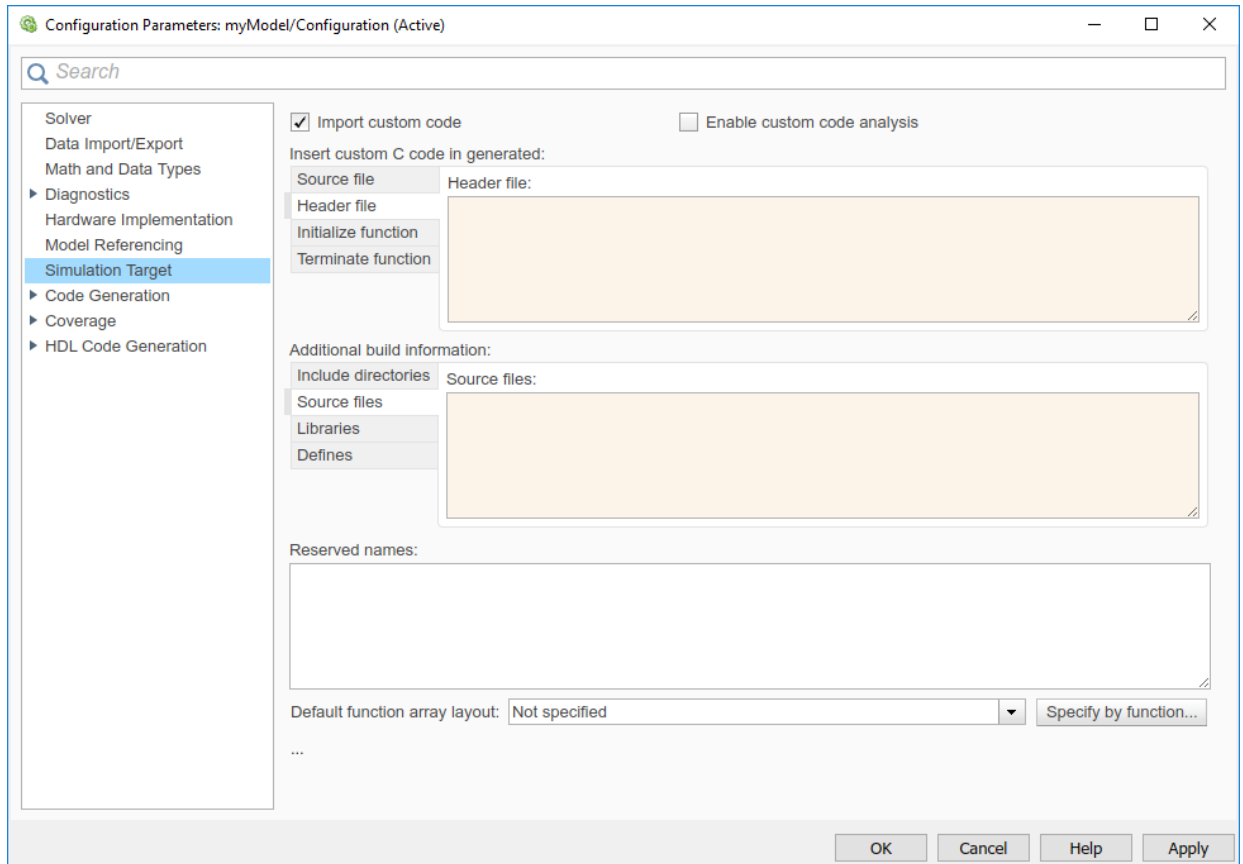
3 Specify your custom code in the subpanes.

Follow the guidelines in "Specify Relative Paths to Your Custom Code" on page 30-24.

Note See "Task 1: Include Custom C Code in the Simulation Target" on page 30-5 for descriptions of the custom code options.

---

4 Click **OK**.

By default, settings in the **Simulation Target** pane for the main model apply to all charts contributed by library models.

Tip If you want to rebuild the target to include custom code changes, select **Code > C/C ++ Code > Build Model** in the Stateflow Editor.

Task 2: Ensure That Custom C Code for the Main Model Applies to Library Charts

Configure the simulation target for each library model that contributes a chart to your main model:

1   In the Stateflow Editor, select **Simulation > Debug > Simulation Target For MATLAB & Stateflow**.

2   Clear the **Use local custom code settings (do not inherit from main model)** check box.

    This step ensures that library charts inherit the custom code settings of your main model.

3   Click **OK**.

Task 3: Simulate the Model

Simulate your model by clicking the play button in the toolbar of the editor.

For information on setting simulation options using the command-line API, see "Command-Line API to Set Simulation and Code Generation Parameters" on page 30-19.

See Also

More About

- "Reuse Custom Code in Stateflow Charts" on page 30-26
- "Include Custom C Code Functions and Structures"
- "Share String Data with Custom C Code" on page 20-17
- "Integrate Custom Structures in Stateflow Charts" on page 25-11

# Access Custom Code Variables and Functions in Stateflow Charts

You can integrate custom code written in C or C++ with Stateflow charts in Simulink models. By sharing data and functions between your custom code and your Stateflow chart, you can augment the capabilities of Stateflow and leverage the software to take advantage of your preexisting code. For more information, see "Reuse Custom Code in Stateflow Charts" on page 30-26.

Custom Code Variables in Charts That Use MATLAB as the Action Language

You can read and write the following C code variables directly in your charts that use MATLAB as the action language.

Custom C Code Type	Description
double	Double-precision floating point
single	Single-precision floating point
int8	Signed 8-bit integer
uint8	Unsigned 8-bit integer
int16	Signed 16-bit integer
uint16	Unsigned 16-bit integer
int32	Signed 32-bit integer
uint32	Unsigned 32-bit integer

By right clicking on the Stateflow object that uses your custom code, you can access your custom code variable. After right clicking on the object, hover over **Explore**. Your custom code variable appears, denoted by (C variable). Clicking the C variable allows you to access the custom code from MATLAB.

Custom Code Functions in Charts That Use MATLAB as the Action Language

You can use the following C function argument types directly in your charts that use MATLAB as the action language without using coder.ceval. For information on calling

external code from MATLAB code by using `coder.ceval`, see "Call C/C++ Code from MATLAB Code" (MATLAB Coder).

Custom C Function Argument Type	Description
`double`	Double-precision floating point
`single`	Single-precision floating point
`int8`	Signed 8-bit integer
`uint8`	Unsigned 8-bit integer
`int16`	Signed 16-bit integer
`uint16`	Unsigned 16-bit integer
`int32`	Signed 32-bit integer
`uint32`	Unsigned 32-bit integer

By right clicking on the Stateflow object that uses your custom code, you can access your custom code function. After right clicking on the object, hover over **Explore**. Your custom code function appears, denoted by `(C function)`. Clicking the C function allows you to access the custom code from MATLAB.

See Also

More About

- "Reuse Custom Code in Stateflow Charts" on page 30-26
- "Specify Relative Paths to Your Custom Code" on page 30-24

Speed Up Simulation

Improve Model Update Performance

Stateflow uses Just-In-Time (JIT) compilation mode to improve model update performance for most charts in Simulink models. Stateflow applies JIT mode to charts that qualify. For a chart in JIT mode, Stateflow generates an execution engine in memory for simulation. For these charts, Stateflow does not generate C code or a MEX file to simulate the chart. JIT mode provides the best performance during the compilation of a model.

Some charts do not qualify for JIT mode, such as charts with signal logging.

Stateflow models include debugging support for simulation. To gain optimal performance, turn off debugging by using this command:

```
sfc('coder_options', 'forceDebugOff', 1);
```

When you run this command, your Stateflow charts do not have debugging support or run-time error checking.

Note When you turn off debugging, animation is also turned off.

Disable Simulation Target Options That Impact Execution Speed

To simulate your model more quickly, in the Model Configuration Parameters dialog box, on the **Simulation Target** pane, clear the check boxes for these parameters:

- **Echo expressions without semicolons** — To disable run-time output in the MATLAB Command Window, such as actions that do not terminate with a semicolon, clear this check box.
- **Ensure responsiveness**— To disable ability to break out of long-running execution using Ctrl+C, clear this check box.

Click **OK**.

Keep Charts Closed to Speed Up Simulation

During model simulation, any open charts with animation enabled take longer to simulate. If you keep all charts closed, the simulation runs faster.

Keep Scope Blocks Closed to Speed Up Simulation

During model simulation, any open Scope blocks continuously update their display. If you keep all Scope blocks closed, you can speed up the simulation. After the simulation ends, you can open the Scope blocks to view the results.

Use Library Charts in Your Model

If your model contains multiple charts that do not use JIT mode and contain the same elements, you might generate multiple copies of identical simulation code. By using library charts, you can minimize the number of copies of identical simulation code. For example, using five library charts reduces the number of identical copies from five to one.

For more information, see "Create Specialized Chart Libraries for Large-Scale Modeling" on page 24-18.

Command-Line API to Set Simulation and Code Generation Parameters

In Stateflow charts in Simulink models, you can use the command-line API to set options in the Model Configuration Parameters dialog box for simulation and embeddable code generation, .

Set Parameters at the Command Line

1 At the MATLAB command prompt, type:

```
object_name = getActiveConfigSet(gcs)
```

This command returns an object handle to the model settings in the Model Configuration Parameters dialog box for the current model.

2 To set a parameter for that dialog box, type:

```
object_name.set_param('parameter_name', value)
```

This command sets a configuration parameter to the value that you specify.

For example, you can set the **Reserved names** parameter for simulation by typing:

```
cp = getActiveConfigSet(gcs)
cp.set_param('SimReservedNameArray', {'abc','xyz'})
```

Note You can also get the current value of a configuration parameter by typing:

```
object_name.get_param('parameter_name')
```

For more information about using `get_param` and `set_param`, see the Simulink documentation.

Simulation Parameters for Nonlibrary Models

The following table summarizes the parameters and values that you can set for simulation of nonlibrary models using the command-line API.

Parameter and Values	Dialog Box Equivalent	Description
SimIntegrity – 'off', **'on'**	**Ensure memory integrity**	Detect violations of memory integrity in code generated for MATLAB Function blocks and stop execution with a diagnostic.
SFSimEcho – 'off', **'on'**	**Echo expressions without semicolons**	Enable run-time output to appear in the MATLAB Command Window during simulation.
SimCtrlC – 'off', **'on'**	**Ensure responsiveness**	Enable responsiveness checks in code generated for MATLAB Function blocks.
SimBuildMode – **'sf_incremental_build'**, 'sf_nonincremental_build', 'sf_make', 'sf_make_clean', 'sf_make_clean_objects'	**Simulation target build mode**	Specify how you build the simulation target for a model.
SimReservedNameArray *string array* – **{}**	**Symbols > Reserved names**	Enter the names of variables or functions in the generated code that match the names of variables or functions specified in custom code to avoid naming conflicts.
SimParseCustomCode – 'off', **'on'**	**Import custom code**	Specify whether or not to parse the custom code and report unresolved symbols for the C charts in a model.
SimCustomSourceCode *string* – ''	**Source file**	Enter code lines to appear near the top of a generated source code file.
SimCustomHeaderCode *string* – ''	**Header file**	Enter code lines to appear near the top of a generated header file.

Parameter and Values	Dialog Box Equivalent	Description
SimCustomInitializer *string* – ''	**Initialize function**	Enter code statements that execute once at the start of simulation.
SimCustomTerminator *string* – ''	**Terminate function**	Enter code statements that execute at the end of simulation.
SimUserIncludeDirs *string* – ''	**Include directories**	Enter a space-separated list of folder paths that contain files you include in the compiled target. **Note** If your list includes any Windows® paths that contain spaces, each instance must be enclosed in double quotes within the argument, for example, `'C:\Project "C:\Custom Files"'`
SimUserSources *string* – ''	**Source files**	Enter a space-separated list of source files to compile and link into the target.
SimUserLibraries *string* – ''	**Libraries**	Enter a space-separated list of static libraries that contain custom object code to link into the target.

Simulation Parameters for Library Models

The following table summarizes the simulation parameters that apply to library models.

Parameter and Values	Dialog Box Equivalent	Description
SimUseLocalCustomCode – **'off'**, 'on'	**Use local custom code settings (do not inherit from main model)**	Specify whether a library model can use custom code settings that are unique from the main model to which the library is linked.
SimCustomSourceCode *string* – ''	**Source file**	Enter code lines to appear near the top of a generated source code file.
SimCustomHeaderCode *string* – ''	**Header file**	Enter code lines to appear near the top of a generated header file.
SimCustomInitializer *string* – ''	**Initialize function**	Enter code statements that execute once at the start of simulation.
SimCustomTerminator *string* – ''	**Terminate function**	Enter code statements that execute at the end of simulation.
SimUserIncludeDirs *string* – ''	**Include directories**	Enter a space-separated list of folder paths that contain files you include in the compiled target. **Note** If your list includes any Windows paths that contain spaces, each instance must be enclosed in double quotes within the argument, for example, 'C:\Project "C:\Custom Files"'
SimUserSources *string* – ''	**Source files**	Enter a space-separated list of source files to compile and link into the target.

Parameter and Values	Dialog Box Equivalent	Description
SimUserLibraries *string* – ''	**Libraries**	Enter a space-separated list of static libraries that contain custom object code to link into the target.

See Also

More About

- "Recommended Settings Summary for Model Configuration Parameters" (Simulink Coder)

Specify Relative Paths to Your Custom Code

You can integrate custom code written in C or C++ with Stateflow charts in Simulink models. By sharing data and functions between your custom code and your Stateflow chart, you can augment the capabilities of Stateflow and leverage the software to take advantage of your preexisting code. For more information, see "Reuse Custom Code in Stateflow Charts" on page 30-26.

Why Use Relative Paths?

If you specify paths and files with absolute paths and later move them, you must change these paths to point to new locations. To avoid this problem, use relative paths for custom code options that specify paths or files.

Search Relative Paths

Search paths exist relative to these folders:

- The current folder
- The model folder (if different from the current folder)
- The custom list of folders that you specify
- All the folders on the MATLAB search path, excluding the toolbox folders

Path Syntax Rules

When you construct relative paths for custom code, follow these syntax rules:

- You can use the forward slash (/) or backward slash (\) as a file separator, regardless of whether you are on a UNIX® or PC platform. The makefile generator returns the path names with the correct platform-specific file separators.

- You can use tokens that evaluate in the MATLAB workspace, if you enclose them with dollar signs ($...$). For example, consider this path:

  ```
  $mydir1$\dir1
  ```

  In this example, `mydir1` is a variable that you define in the MATLAB workspace as `'d:\work\source\module1'`. In the generated code, this custom include path appears as:

```
d:\work\source\module1\dir1
```

- You must enclose paths in double quotes if they contain spaces or other nonstandard path characters, such as hyphens (-).

See Also

More About

- "Reuse Custom Code in Stateflow Charts" on page 30-26

Reuse Custom Code in Stateflow Charts

You can integrate custom code written in C or C++ with Stateflow charts in Simulink models. By sharing data and functions between your custom code and your Stateflow chart, you can augment the capabilities of Stateflow and leverage the software to take advantage of your preexisting code.

Stateflow charts call custom code functions by using the same syntax as other reusable functions:

return_val = function_name(arg1,arg2,...)

Note Do not share fixed-point data between your custom code and your Stateflow chart. Data returned from custom code must be of type `double`.

Choose the Appropriate Procedure for Simulation

To choose the right procedure for simulation, click the highlighted block that best describes your goal.
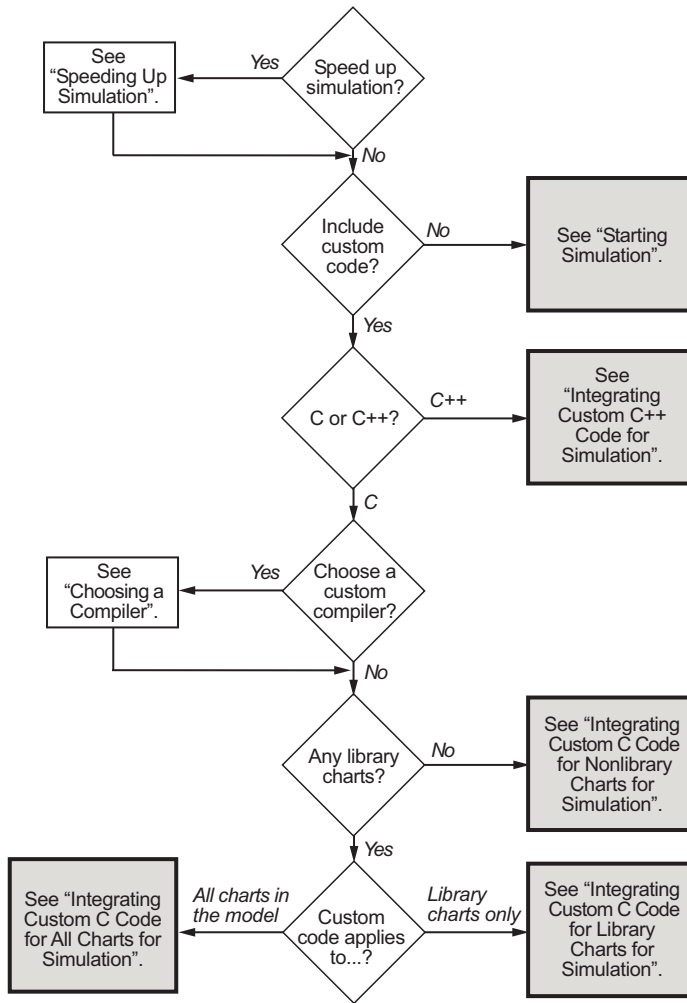
Use Custom Code to Define Global Constants

This example shows how to use custom C code to define constants that apply to all charts in your model.

The model contains a Stateflow® chart with an input, which you can set to 0 or 1 by toggling the manual switch in the model during simulation.
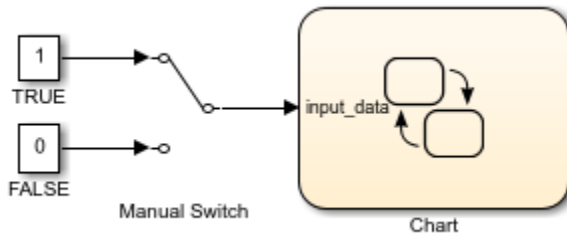
The chart contains two states A and B. In this example, you define two constants named TRUE and FALSE to guard the transitions between the states in the chart, instead of using the values 1 and 0. These custom definitions improve the readability of your chart actions. TRUE and FALSE are not Stateflow data objects.
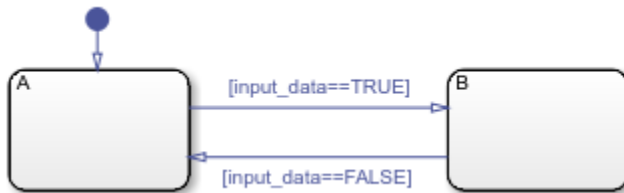
To enter the custom code that defines the two constants:

1   Open the Model Configuration Parameters dialog box.

2   Select the **Simulation Target** pane.

3   In the **Header file** subpane, enter #define and #include statements. For instance, in this example, you define the global constants with this code:

```
#define TRUE 1
#define FALSE 0
```

Because the two custom definitions appear at the top of the generated machine header file sf_custom_code_global_constants_sfun.h, you can use TRUE and FALSE in all charts that belong to this model.
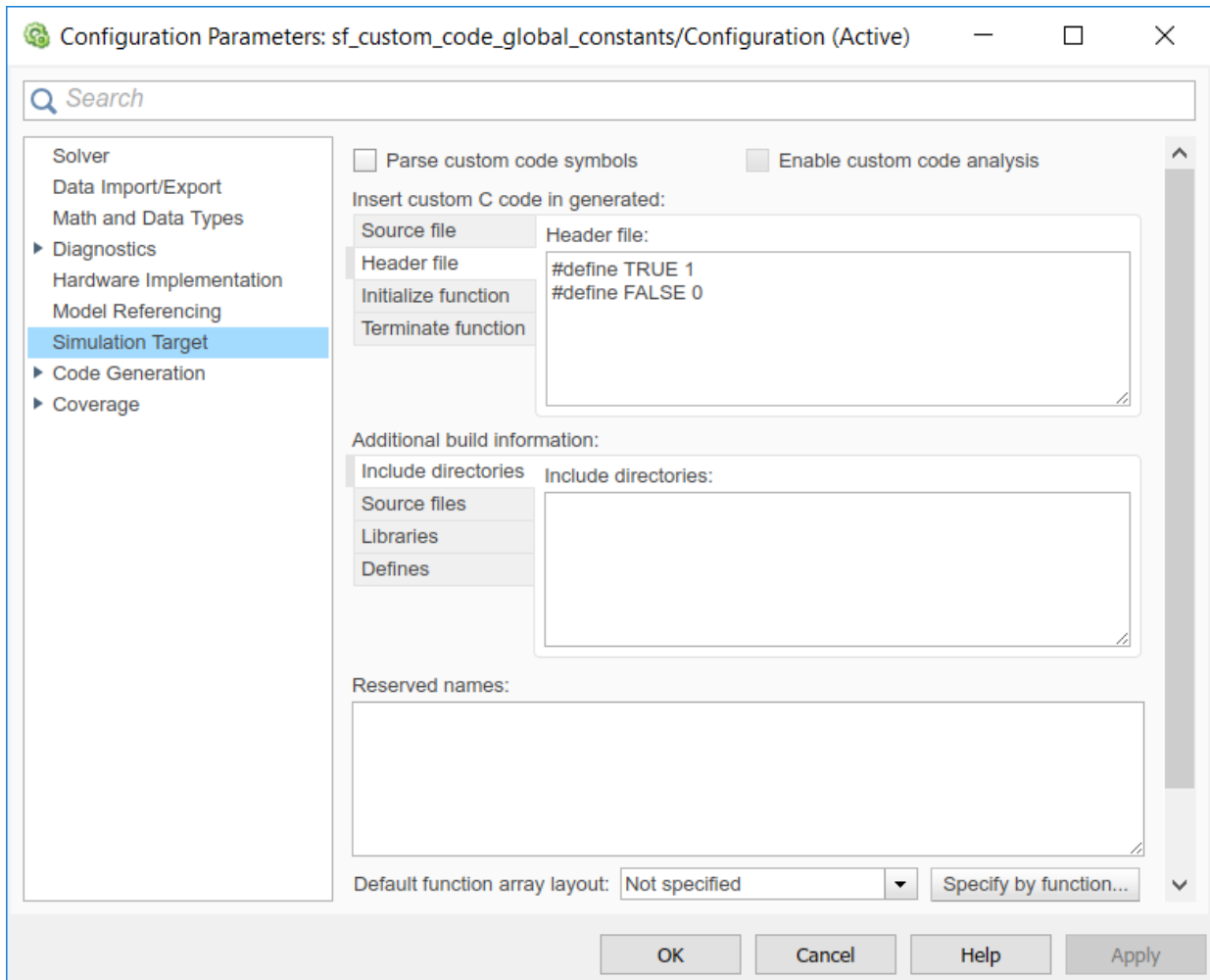
Use Custom Code to Define Constants, Variables, and Functions

This example show how to use custom C code to define constants, variables, and functions that apply to all charts in your model.

The model contains a Stateflow chart with an input signal from a Sine Wave block.
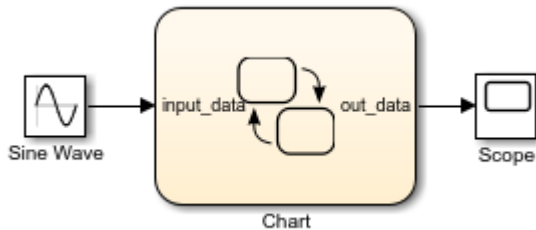
The chart contains two states A and B, and three data objects: `input_data`, `local_data`, and `out_data`. The chart accesses a custom variable named `my_global` and calls a custom function named `my_function`.
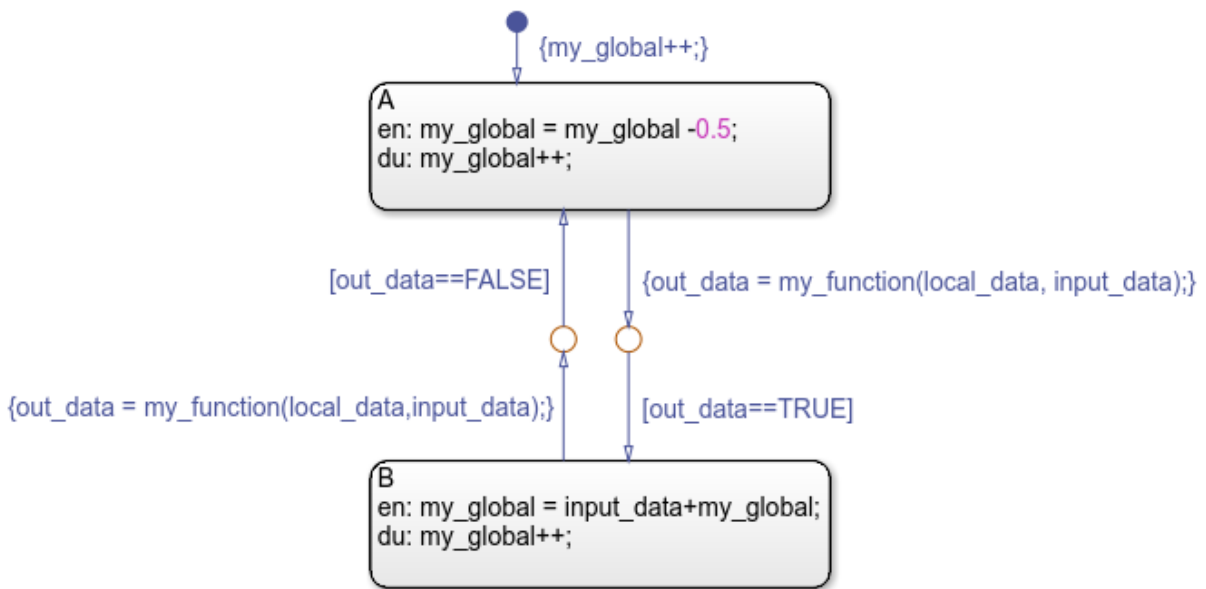
To configure the model to access the custom code:

1   Open the Model Configuration Parameters dialog box.

2   Select the **Simulation Target** pane.

3   In the **Header file** subpane, enter `#define` and `#include` statements. When you include a custom header file, you must enclose the file name in double quotes.

4 In the **Include directories** subpane, enter the name of the folder that contains your custom code files. To access custom code files in a subfolder of the model folder, use a relative path name of the form `.\subfolder_name`.

5 In the **Source files** subpane, enter the name of the source file that contains your custom code. To access a source file that resides in a subfolder of the model folder, use a relative path name of the form `.\subfolder_name\source_file.c`.

In this example, the custom code defines three constants, a variable, and a function by using these configurations:

- The **Header file** subpane contains this statement: `#include "sf_custom_code_constants_vars_fcns_hdr.h"`

- The **Include directories** subpane contains a single period (`.`) to indicate that all of the custom code files reside in the same folder as the model.

- The **Source file** subpane contains this file name: `sf_custom_code_constants_vars_fcns_src.c`
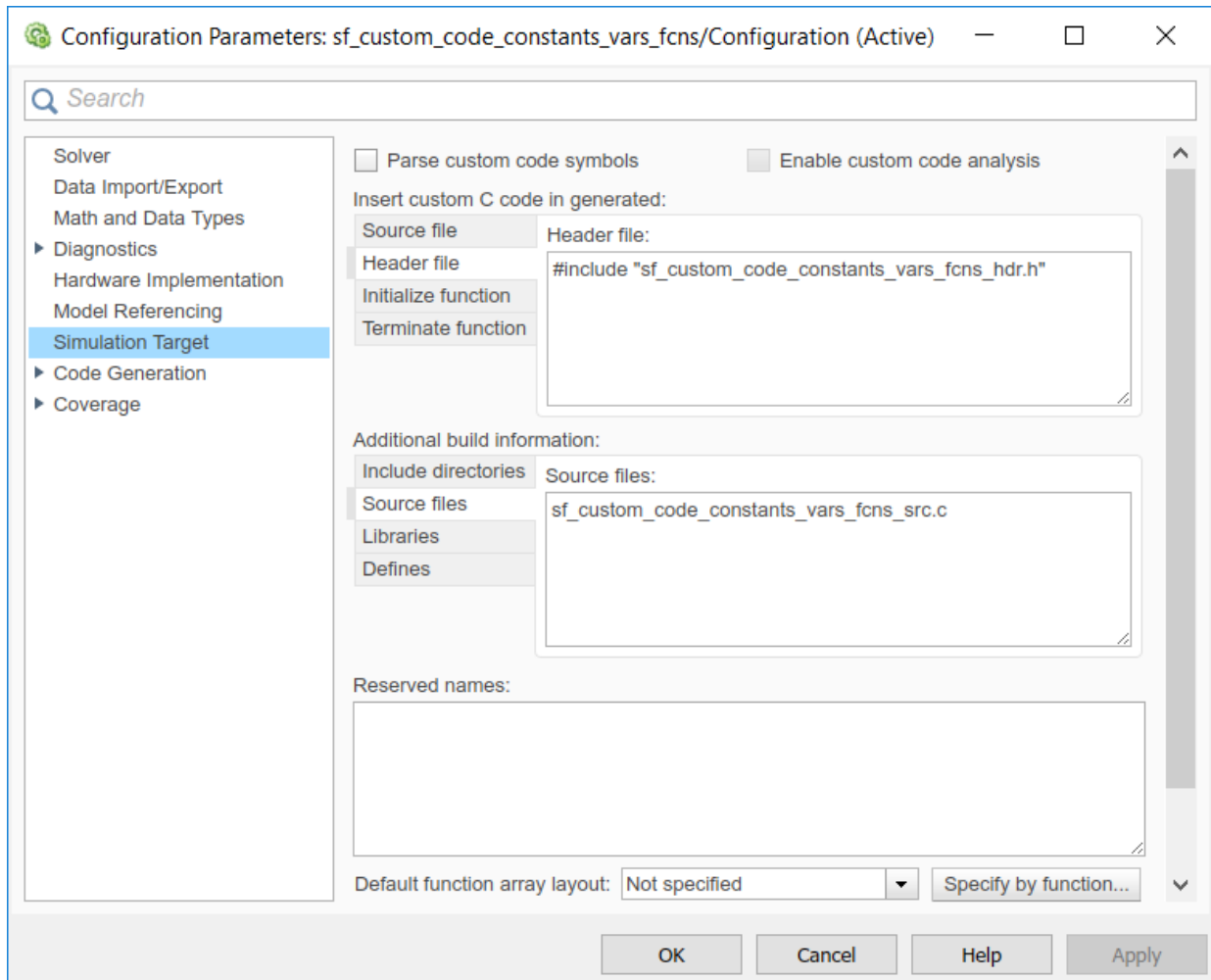
The custom header file sf_custom_code_constants_vars_fcns_hdr.h contains the definitions of three constants:

```
#define TRUE 1
#define FALSE 0
#define MAYBE 2
```

The header file also contains declarations for the variable my_global and the function my_function:

```
extern int myglobal;
extern int my_function(int var1, double var2);
```

The custom source file `sf_custom_code_constants_vars_fcns_src.c` is compiled in conjunction with the Stateflow generated code into a single S-function MEX file.

Because the custom definitions appear at the top of the generated machine header file `sf_custom_code_constants_vars_fcns_sfun.h`, you can access them in all charts that belong to this model.

See Also

More About

- "Access Custom C Code in Nonlibrary Charts" on page 30-5
- "Access Custom C Code in Library Charts" on page 30-9
- "Access Custom C Code in All Charts in Model" on page 30-12
- "Access Custom C++ Code in Stateflow Charts" on page 30-3

Parse Stateflow Charts

How the Stateflow Parser Works

The parser evaluates the graphical and nongraphical objects and data in each Stateflow machine against the supported chart notation and the action language syntax.

Calling the Stateflow Parser

When you simulate a model, build a target, or generate code for a target, Stateflow automatically parses the Stateflow machine. Also, you can invoke the Stateflow parser to check the syntax of your chart by selecting **Chart > Parse Chart** from the Stateflow Editor.

If parsing is unsuccessful, the chart automatically appears with the highlighted object that causes the first parse error. You can select the error in the diagnostic window to bring its source chart to the front with the source object highlighted. Any unresolved data or events in the chart are flagged in the Symbol Wizard.

Resolve Undefined Symbols in Your Chart

Symbols that appear in your chart but that you have not added as data, events, or messages are *undefined* or *unresolved*. You can resolve undefined symbols by using the Symbols window or the Symbol Wizard. For each undefined symbol, based on the symbol usage in the chart, Stateflow infers these properties:

- Class (data, event, or message)
- Scope
- Size
- Type
- Complexity

Resolve Symbols Through the Symbols Window

As you edit your chart, Stateflow detects undefined symbols and marks them in the

Symbols window with a red error icon . For each undefined symbol, the **TYPE** column displays the class and scope inferred from the usage in the chart. You can resolve undefined symbols individually or collectively.

- To define a symbol with the inferred class and scope, click the error icon and select **Fix**.
- To define a symbol with a different class or scope, select another combination of class and scope from the **TYPE** drop-down list.
- To resolve all of the undefined symbols with their inferred classes and scopes, click the

  Resolve undefined symbols button .
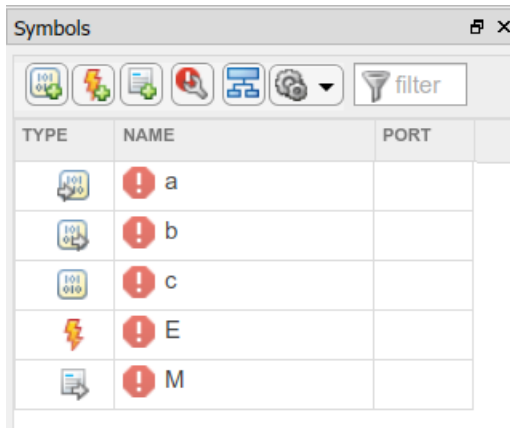
Resolve Symbols Through the Symbol Wizard

If Stateflow detects any undefined symbols when you parse a chart, update a diagram, or simulate a model, the Symbol Wizard opens and lists the undefined symbols. For each undefined symbol, the **Class** and **Scope** columns display the class and scope inferred from the usage in the chart. You can accept, modify, or reject each symbol definition that the Symbol Wizard suggests.

- To accept a definition with the inferred class and scope, select the check box in front of the symbol.
- To modify a definition, select a different class or scope from the **Class** or **Scope** drop-down lists.
- To reject a definition, clear the check box in front of the symbol.

After you edit the symbol definitions, add the symbols to the Stateflow hierarchy by clicking **OK**.
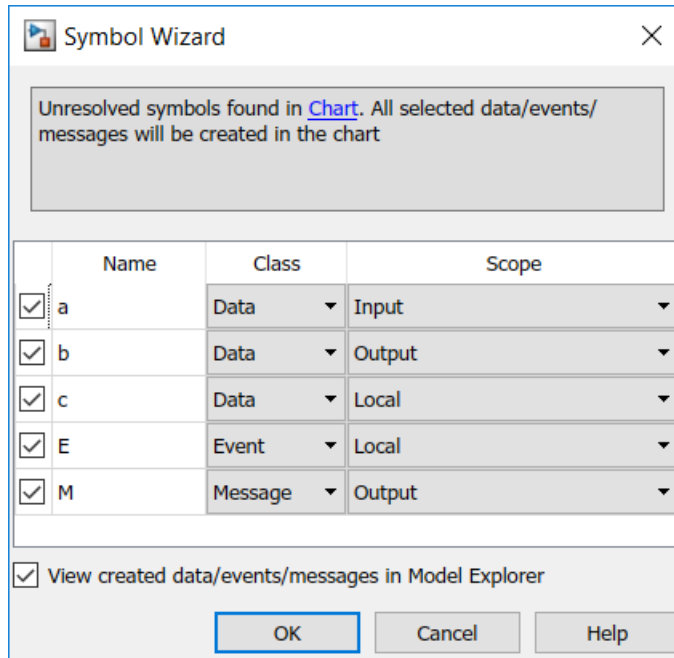
Detect Symbol Definitions in Custom Code

Detection of symbols defined in custom code depends on the model configuration parameter **Import custom code**.

- If you select **Import custom code**, the parser tries to find unresolved chart symbols in the custom code. If the custom code does not define these symbols, they appear in the Symbol Wizard.

- If you do not select **Import custom code**, the parser considers unresolved data symbols in the chart as defined in the custom code. If the custom code does not define these symbols, simulating and generating code from the model results in an error.

The **Import custom code** option is not available for charts that use MATLAB as the action language. For more information, see "Import custom code" (Simulink).

See Also

More About

- "Add Stateflow Data" on page 9-2
- "Set Data Properties" on page 9-5
- "Manage Data, Events, and Messages in the Symbols Window" on page 33-2
- "Detect Unused Data in the Symbols Window" on page 33-4
- "Parse Stateflow Charts" on page 30-34
- "Import custom code" (Simulink)

Call Extrinsic MATLAB Functions in Stateflow Charts

Stateflow charts in Simulink models have an action language property that defines the syntax for state and transition actions. An icon in the lower-left corner of the chart canvas indicates the action language for the chart.

- MATLAB as the action language.

- C as the action language.

In charts that use C as the action language, you can call built-in MATLAB functions and access MATLAB workspace variables by using the `ml` namespace operator or the `ml` function. For more information, see "Access MATLAB Functions and Workspace Data in C Charts" on page 12-32.

In charts that use MATLAB as the action language, you can call MATLAB functions supported for code generation directly. To call extrinsic functions that are not supported for code generation, you must use the `coder.extrinsic` function. When you declare a function with `coder.extrinisic(function_name)`, Stateflow creates a call to the function during simulation. In a Stateflow chart, you only declare `coder.extrinsic` once. You cannot declare reserved keywords with `coder.extrinsic`. For more information, see "Rules for Naming Stateflow Objects" on page 2-4.

For charts that include atomic subcharts, you must declare functions that are not supported for code generation with `coder.extrinsic` separately within the atomic subchart.

coder.extrinsic Function

To create a call for the extrinsic function `heaviside`, this model uses `coder.extrinsic`.
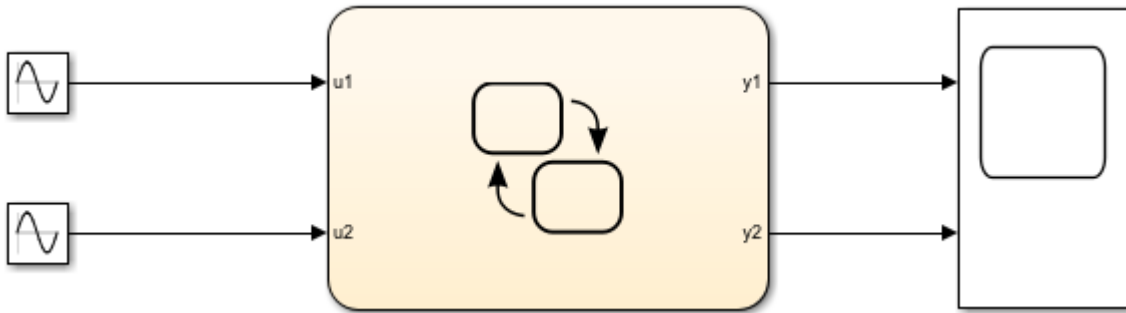
The chart contains two parallel states, A and B, and one graphical function block, `foo`. State A declares the function `heaviside`, which is not supported for code generation, by using `coder.extrinsic`. State B and the graphical function block also use `heaviside` without `coder.extrinsic`.

The input for state A is `u1`, a sine wave, and the input for state B is `u2`, a cosine wave. The graphical function `out` outputs the value of the `heaviside` function for the input `in`.
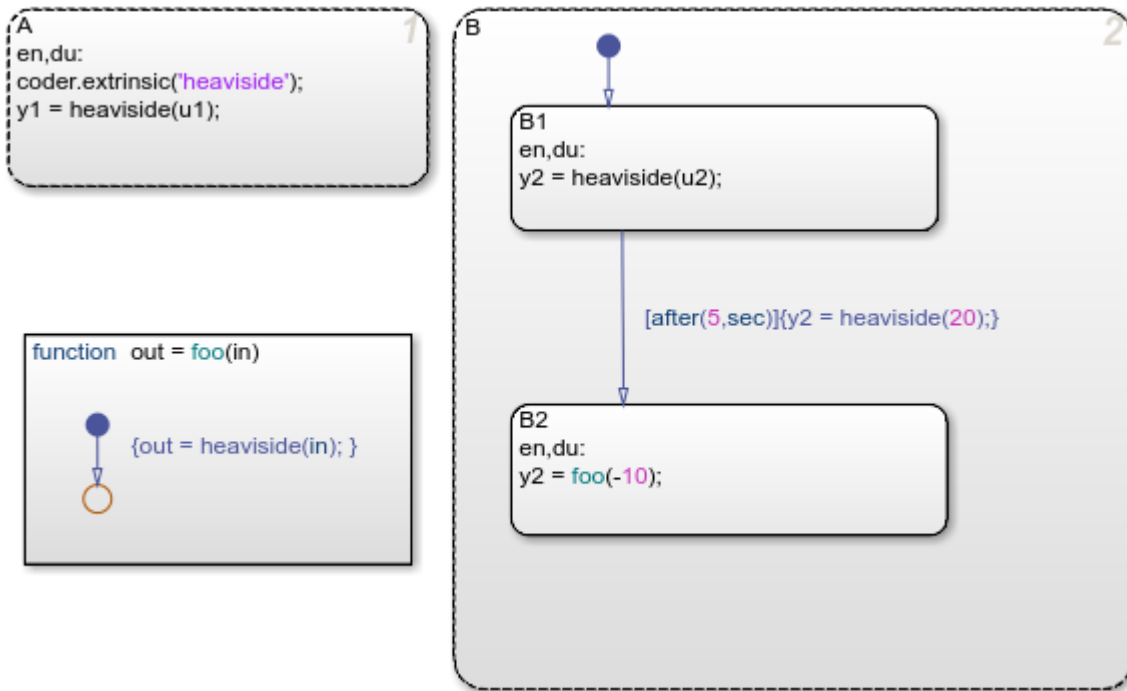
You only need to declare `heaviside` once in your chart using `coder.extrinsic`. After this you can use the `heaviside` function anywhere within your chart without `coder.extrinsic`. When generating code the functions that you declare using `coder.extrinsic` will have a call to the extrinsic function, and that function will not appear in the generated code.

To visualize the result of this chart, open the scope.
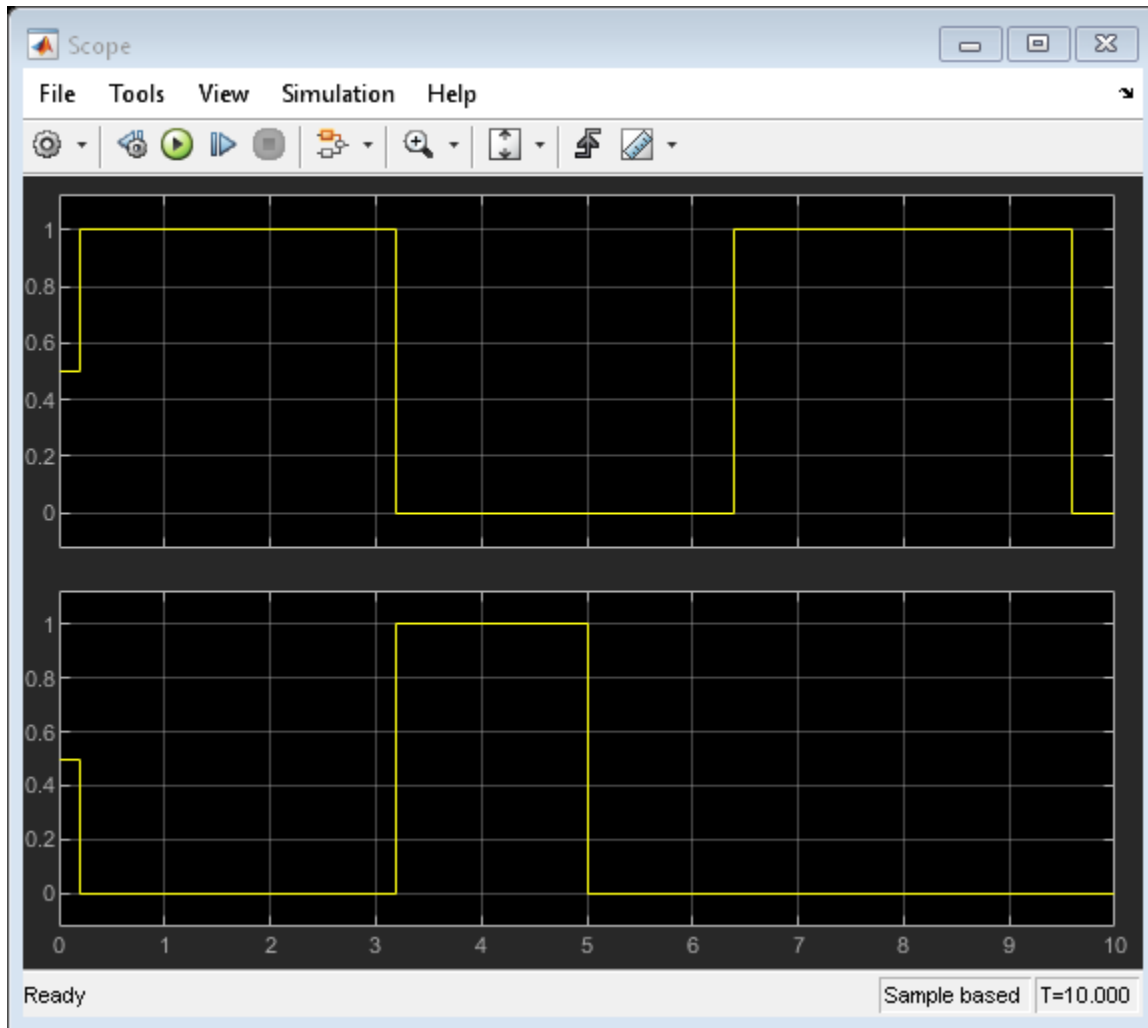
See Also
coder.extrinsic | heaviside

More About

- "Access MATLAB Functions and Workspace Data in C Charts" on page 12-32
- "Generate Code for Global Data" (MATLAB Coder)
- "Functions and Objects Supported for C/C++ Code Generation — Alphabetical List" (Simulink)

Code Generation

Generate C or C++ Code from Stateflow Blocks

If you have Simulink Coder or Embedded Coder, you can generate C or C++ code from Simulink models that include Stateflow blocks.

Use	Required Software License	Description
Rapid prototyping	Simulink Coder	Generate source code that you can use for real-time and nonreal-time applications, including simulation acceleration, rapid prototyping, and hardware-in-the-loop testing.
Production code deployment	Embedded Coder	Generate readable, compact, and fast code for use on embedded processors, on-target rapid prototyping boards, and microprocessors used in mass production.

The generated code does not contain code to interface with other blocks in a Simulink model or the MATLAB base workspace. You cannot generate code only for the Stateflow blocks in a library model. First create a link to the library block in your main model and then generate code for the main model.

Generate Code for Rapid Prototyping and Production Deployment

This table directs you to information about code generation based on your goals.

Goal	Simulink Coder Documentation	Embedded Coder Documentation
Generate C/C++ source code	"Source Code Generation in Simulink Coder" (Simulink Coder)	"Code Generation Basics" (Embedded Coder)
Generate C/C++ source code and build executable	"Program Building, Interaction, and Debugging in Simulink Coder" (Simulink Coder)	
Integrate external code	"External Code Integration" (Simulink Coder)	"External Code Integration" (Embedded Coder)

Goal	Simulink Coder Documentation	Embedded Coder Documentation
Include external code only for library charts in a portable, self-contained library for use in multiple models	"Integrate External Code for Library Charts" (Simulink Coder)	"Integrate External Code for Library Charts" (Embedded Coder)
Optimize generated code	• "Reduce Memory Usage for Boolean and State Configuration Variables" (Simulink Coder) • "Design Tips for Optimizing Generated Code for Stateflow Objects" (Simulink Coder)	• "Reduce Memory Usage for Boolean and State Configuration Variables" (Embedded Coder) • "Design Tips for Optimizing Generated Code for Stateflow Objects" (Embedded Coder)

Traceability of Stateflow Objects in Generated Code

Traceability comments provide a way to:

- Verify generated code. You can identify which Stateflow object corresponds to a line of code and track code from different objects that you have or have not reviewed.
- Include comments in code generated for large-scale models. You can identify objects in generated code and avoid manually entering comments or descriptions.

To enable traceability comments, you must have Embedded Coder or HDL Coder software. For C/C++ code generation, comments appear in the generated code for embedded real-time (`ert`) based targets only. For more information, see "Trace Stateflow Elements in Generated Code" (Embedded Coder) and "Navigate Between Simulink Model and HDL Code by Using Traceability" (HDL Coder).

See Also

More About

- "Generate Reusable Code for Unit Testing" on page 15-48
- "Generate Reusable Code for Atomic Subcharts" on page 15-27
- "Select Array Layout for Matrices in Generated Code" on page 31-5

Select Array Layout for Matrices in Generated Code

When generating code from a Stateflow chart that uses C as the action language, you can specify the array layout for matrices. For example, consider this matrix:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

By default, the code generator uses column-major layout to flatten the matrix as a one-dimensional array. The array is stored in memory with this arrangement:

```
{1, 4, 2, 5, 3, 6}
```

If you select row-major layout, the code generator flattens the matrix and stores it in memory with this arrangement:

```
{1, 2, 3, 4, 5, 6}
```

If you have Embedded Coder, you can preserve the multidimensionality of Stateflow local data, prevent its flattening, and implement the matrix as a two-dimensional array with this arrangement:

```
{{1, 2, 3}, {4, 5, 6}}
```

Row-major and multidimensional array layout can ease the integration with external code. For more information, see "Code Generation of Matrices and Arrays" (Simulink Coder) and "Dimension Preservation of Multidimensional Arrays" (Embedded Coder).

Column-Major Array Layout

By default, the **Array Layout** configuration parameter for a Simulink® model is `Column-Major`. When you click the **Build Model** icon, the code generator flattens all matrix data into one-dimensional arrays, arranging their contents in a column-major layout.

For example, this Stateflow chart contains local data x of size `[2 3]`. The state actions index the elements in x by row and column number.
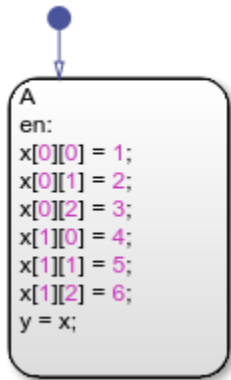
If you generate code, the file `sf_matrix_layout.c` implements the local data `x` in column-major layout with these lines of code:

```
...
sf_matrix_layout_DW.x[0] = 1.0;
sf_matrix_layout_DW.x[2] = 2.0;
sf_matrix_layout_DW.x[4] = 3.0;
sf_matrix_layout_DW.x[1] = 4.0;
sf_matrix_layout_DW.x[3] = 5.0;
sf_matrix_layout_DW.x[5] = 6.0;
...
```

The generated code refers to the elements of `x` by using only one index.

Row-Major Array Layout

Row-major layout can improve the performance of certain algorithms. For example, see "Interpolation Algorithm for Row-Major Array Layout" (Embedded Coder).

To select row-major array layout for the previous example:

1 Open the Model Configuration Parameters dialog box.

2 In the **Code Generation > Interface** pane, set the **Array Layout** parameter to `Row-Major`.

3 Click the **Build Model** icon.

The file `sf_matrix_layout.c` implements the local data `x` in row-major layout with these lines of code:

```
...
sf_matrix_layout_DW.x[0] = 1.0;
sf_matrix_layout_DW.x[1] = 2.0;
sf_matrix_layout_DW.x[2] = 3.0;
sf_matrix_layout_DW.x[3] = 4.0;
sf_matrix_layout_DW.x[4] = 5.0;
sf_matrix_layout_DW.x[5] = 6.0;
...
```

The generated code refers to the elements of `x` by using only one index.

Row-major layout is not supported in:

- Charts and state transition table blocks that use MATLAB as the action language.
- Charts that contain truth table functions that use MATLAB as the action language.
- Charts that contain MATLAB functions.
- Charts that use custom C code.
- Truth table blocks.

Multidimensional Array Layout

If you have Embedded Coder, you can generate code that preserves the multidimensionality of Stateflow local data without flattening the data as one-dimensional arrays. To select multidimensional layout for the previous example:

1   Open the Model Configuration Parameters dialog box.
2   In the **Code Generation > Interface** pane:

   - Set the **Array Layout** parameter to `Row-Major`.
   - Select the **Preserve Stateflow local data array dimensions** check box.
3   Click the **Build Model** icon.

The file `sf_matrix_layout.c` implements the local data `x` as a two-dimensional array with these lines of code:

```
...
sf_matrix_layout_DW.x[0][0] = 1.0;
```

```
sf_matrix_layout_DW.x[0][1] = 2.0;
sf_matrix_layout_DW.x[0][2] = 3.0;
sf_matrix_layout_DW.x[1][0] = 4.0;
sf_matrix_layout_DW.x[1][1] = 5.0;
sf_matrix_layout_DW.x[1][2] = 6.0;
...
```

The generated code refers to the elements of x by using two indices.

Multidimensional array layout is available for:

- Local data in Stateflow charts.
- Parameters and root-level inport and outport data in Simulink models.

Multidimensional layout is not available for bus signals containing multidimensional array data. Multidimensional layout is not supported in:

- Charts that contain messages.
- Reusable charts or charts in reusable parent subsystems.

For more information, see "Preserve Dimensions of Multidimensional Arrays in Generated Code" (Embedded Coder).

See Also

More About

- "Generate C or C++ Code from Stateflow Blocks" on page 31-2
- "Use Arrays in Actions" on page 12-44
- "Code Generation of Matrices and Arrays" (Simulink Coder)
- "Interpolation Algorithm for Row-Major Array Layout" (Embedded Coder)
- "Dimension Preservation of Multidimensional Arrays" (Embedded Coder)

Debug and Test Stateflow Charts

Debugging Charts

For Stateflow charts in Simulink models, you can perform most debugging tasks directly from the Stateflow Editor:

- Set breakpoints to stop execution in specific objects, such as charts, states, transitions, graphical functions, truth table functions, local events, and input events.
- Enable, disable, and set conditions on breakpoints in the Breakpoints and Watch window.
- Add data that you want to monitor during simulation in the Breakpoints and Watch window.
- After execution stops at a breakpoint, step through the simulation.

In addition, during simulation, you can display and change the values of Stateflow data in the MATLAB Command Window.

For information on debugging Stateflow chart objects that you execute in MATLAB, see "Debug a Standalone Stateflow Chart" on page 34-16.

See Also

More About

Animate Stateflow Charts

Set Animation Speeds

During simulation, animation provides visual verification that your chart behaves as you expect. Animation highlights active objects in a chart as execution progresses. For charts in a Simulink model, you can control the speed of chart animation during simulation, or turn animation off. In the Stateflow editor, select **Simulation > Stateflow Animation**, then select:

- **Lightning Fast**
- **Fast**
- **Medium**
- **Slow**
- **None**

Lightning Fast animation provides the fastest simulation speed by buffering the highlights. During **Lightning Fast** animation, the more recently highlighted objects are in a bolder, lighter blue. These highlights fade away as simulation time progresses.

The default animation speed, **Fast**, shows the active highlights at each time step. To add a delay with each time step, set the animation speed to **Medium** or **Slow**.

Maintain Highlighting

To maintain highlighting of active states in the chart after simulation ends, select **Simulation > Stateflow Animation > Maintain Highlighting**.

Disable Animation

Animation is enabled by default in Stateflow charts. To turn off animation for a chart, in the Stateflow editor, select **Simulation > Stateflow Animation > None**.

Animate Stateflow Charts as Generated Code Executes on a Target System

If you have a Simulink Coder license, you can use external mode to establish communication between a Simulink model and generated code downloaded to and

executing on a target system. Stateflow software can use the external mode communication channel to animate chart states. Also, you can designate chart data of local scope to be test points and view the test point data in floating scopes and signal viewers.

For more information, see:

- "External Mode Simulation with XCP Communication" (Simulink Coder)
- "External Mode Simulation with TCP/IP or Serial Communication" (Simulink Coder)

.

Set Breakpoints to Debug Charts

You enable debugging for a chart when you set a breakpoint. A breakpoint indicates a point at which Stateflow halts execution of a chart that you are simulating. While simulation is halted, you can view Stateflow data and interact with the MATLAB workspace to examine the status of the chart.

Breakpoints appear as circular red badges. For example, this chart contains breakpoints on the state On and on the transition from On to Off.
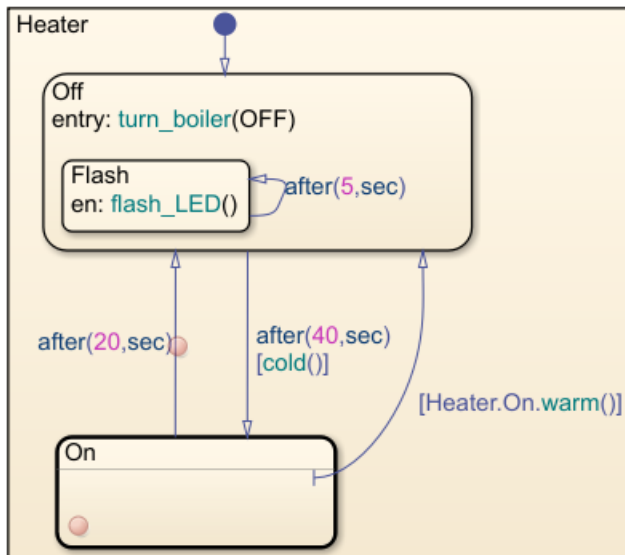
Set a Breakpoint for a Stateflow Object

You can set breakpoints on charts, states, transitions, graphical or truth table functions, and events.

Breakpoints on Charts

To set a breakpoint on a chart, right-click inside the chart and select **Set Breakpoint on Chart Entry**. This type of breakpoint stops simulation before entering the chart.

To remove the breakpoint, right-click inside the chart and clear the **Set Breakpoint on Chart Entry** option.

Breakpoints on States and Transitions

You can set different types of breakpoints on states and transitions.

Object	Breakpoint Type
State	`On State Entry` — Stop simulation before performing the state `entry` actions.
	`During State` — Stop simulation before performing the state `during` actions.
	`On State Exit` — Stop simulation after performing the state `exit` actions.
Transition	`When Transition is Valid` — Stop simulation after the transition tests valid, but before taking the transition.
	`When Transition is Tested` — Stop simulation before testing whether the transition is a valid path. If no condition exists on the transition, this breakpoint type is not available.

To set a breakpoint on a state or transition, right-click the state or transition and select **Set Breakpoint**. For states, the default breakpoint is `On State Entry` and `During State`. For transitions, the default breakpoint is `When Transition is Valid`. To change the type of breakpoint, click the breakpoint badge and select a different configuration of breakpoints. For more information, see "Change Breakpoint Types" on page 32-7.

To remove the breakpoint, right-click the state or transition and select **Clear Breakpoint**.

Breakpoints on Stateflow Functions

To set a breakpoint on a graphical or truth table function, right-click the function and select **Set Breakpoint During Function Call**. This type of breakpoint stops simulation before calling the function.

To remove the breakpoint, right-click the function and clear the **Set Breakpoint During Function Call** option.

Breakpoints on Events

You can select two types of breakpoints on events:

- `Start of Broadcast` — Stop simulation before broadcasting the event.
- `End of Broadcast` — Stop simulation after a Stateflow object reads the event.

Available breakpoints depend on the scope of the event.

Scope of Event	Start of Broadcast	End of Broadcast
Local	Available	Available
Input	Available	Not available
Output	Not available	Not available

To set or clear breakpoints for an event, modify the **Debugger Breakpoints** properties through the Property Inspector or the Model Explorer. For more information, see "Set Properties for an Event" on page 10-6.

Change Breakpoint Types

Breakpoints appear as circular red badges. A breakpoint badge can represent more than one type of breakpoint. To see a tooltip listing the breakpoint types set for a Stateflow object, point to its badge. For example, the badge on the state `On` represents two breakpoint types: `On State Entry` and `During State`.
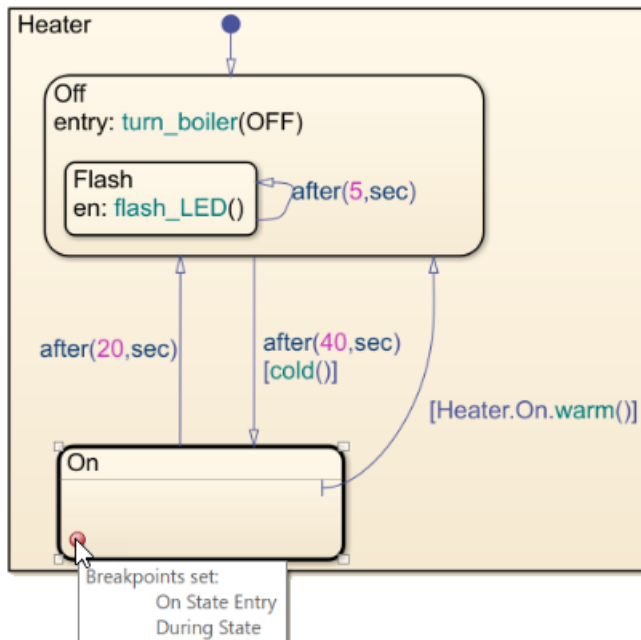
To change the type of breakpoint for an object, click the breakpoint badge. In the Breakpoints dialog box, you can select a different configuration of breakpoints, depending on the object type.
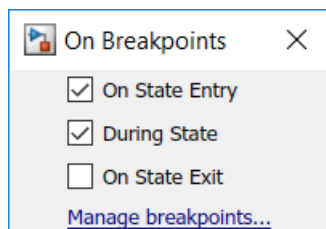
Clearing all of the check boxes in the Breakpoints dialog box removes the breakpoint.

Clicking the **Manage breakpoints** link opens the Stateflow Breakpoints and Watch window. In this window, you can manage conditions for all breakpoints. For each breakpoint, you can:

- Set conditions.
- Enable the breakpoint.
- Disable the breakpoint.
- Clear the breakpoint.
- View the number of times a breakpoint has been encountered during simulation.

For more information, see "Manage Stateflow Breakpoints and Watch Data" on page 32-17.

Control Chart Execution After a Breakpoint

When simulation stops at a breakpoint, Stateflow enters debugging mode.

- The MATLAB command prompt changes to debug>>.
- The Symbols window displays the value of each data object in the chart.
- The chart highlights active elements in blue and the currently executing object in green.

For example, this chart contains a breakpoint on entry in the HIGH state, a substate of On. When simulation stops at the breakpoint, the active state (On) appears highlighted in blue and the currently executing substate (HIGH) appears highlighted in green.
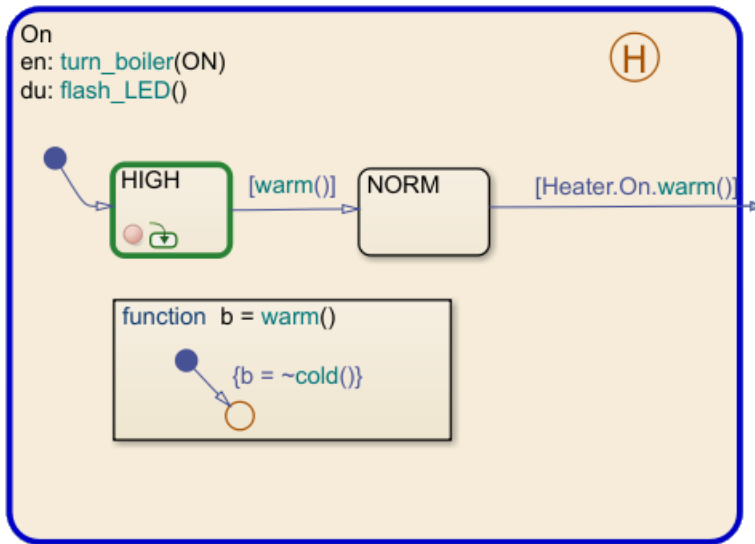
An execution status badge appears in the graphical object where execution pauses.

Badge	Description
	Execution stopped before entering a chart or state.
	Execution stopped in a state `during` action, graphical function, or truth table function.
	Execution stopped after exiting a state.
	Execution stopped before testing a transition.
	Execution stopped before taking a valid transition.

To see execution status, point to the badge. A tooltip indicates:

- Where the execution stopped
- Simulation time
- Current event (in the presence of a local or input event)
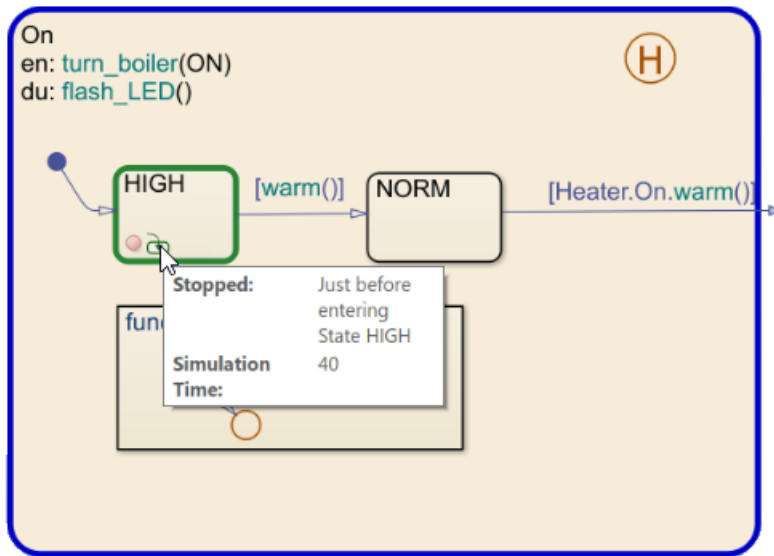
To view data values in the chart, point to a chart object. A tooltip displays:

- The value of each data and message that the selected object uses
- Temporal information (in the presence of a temporal logic operator)

For more information, see "Watch Data in the Stateflow Chart" on page 32-36.
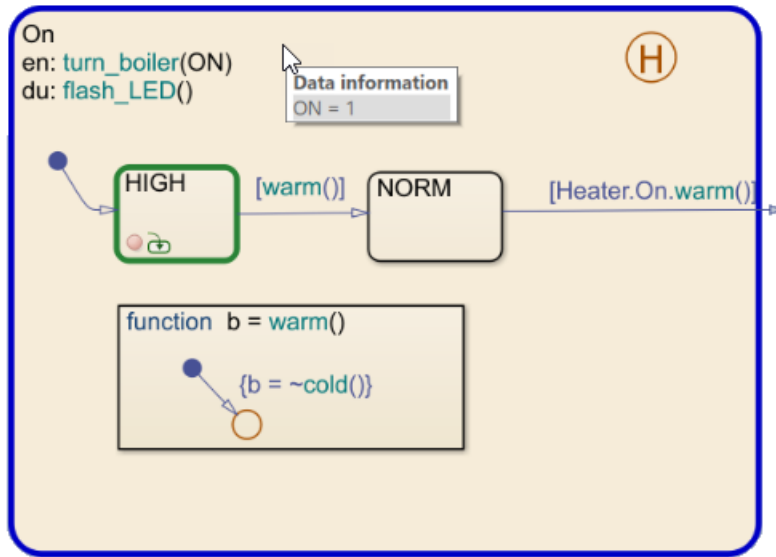
After simulation stops at a breakpoint, you can continue chart execution from the Stateflow Editor toolbar, at the MATLAB command prompt, or by selecting a keyboard shortcut.

Toolbar Icon	Option	Command	Description	Keyboard Shortcut
▶	Continue	dbcont	Continue simulation to the next breakpoint.	**Ctrl+T**

Toolbar Icon	Option	Command	Description	Keyboard Shortcut
	Step Over	dbstep	Advance to the next step in the chart execution. At the chart level, possible steps include: • Enter the chart • Test a transition • Execute a transition action • Activate a state • Execute a state action For more information, see "Execution of a Stateflow Chart" on page 3-44.	**F10**
	Step In	dbstep in	From a state or transition action that calls a function, advance to the first executable statement in the function. From a statement in a function containing another function call, advance to the first executable statement in the second function. Otherwise, advance to the next step in the chart execution. (See Step Over.)	**F11**

Toolbar Icon	Option	Command	Description	Keyboard Shortcut
	Step Out	dbstep out	From a function call, return to the statement calling the function. Otherwise, continue simulation to the next breakpoint. (See Continue.)	**Shift+F11**
	Step Forward		Exit debug mode and pause simulation before next time step.	
	Stop	dbquit	Exit debug mode and stop simulation.	**Ctrl+Shift+T**

In state or transition actions containing more than one statement, you can step through the individual statements one at a time by selecting **Step Over**. Stateflow highlights each statement before executing it. To execute a group of statements together, right-click the last statement in the group and select **Run To Cursor**.
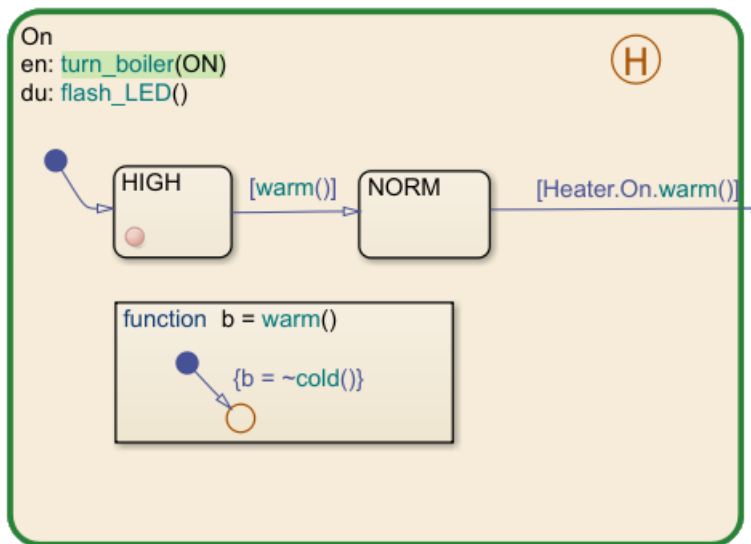
See Also

More About

- "Debugging Charts" on page 32-2
- "Manage Stateflow Breakpoints and Watch Data" on page 32-17
- "Change Data Values During Simulation" on page 32-41
- "Debug Run-Time Errors in a Chart" on page 32-24

Debug Truth Tables with Breakpoints

To enable debugging in a Stateflow truth table, set a breakpoint where you want the simulation to pause. When the simulation pauses you can view the data values and step through the simulation. While the simulation is paused you can interact with the MATLAB workspace to examine the status of the chart.

Breakpoints appear as circular red badges.

Set a Breakpoint for a Stateflow Object

Change Breakpoint Types

See Also

More About

- "Debugging Charts" on page 32-2
- "Manage Stateflow Breakpoints and Watch Data" on page 32-17
- "Change Data Values During Simulation" on page 32-41
- "Debug Run-Time Errors in a Chart" on page 32-24

Manage Stateflow Breakpoints and Watch Data

Set Conditions on Breakpoints

In the **Breakpoints** tab of the Stateflow Breakpoints and Watch window, you can enter a MATLAB expression as a condition on a Stateflow breakpoint. Simulation then pauses on that breakpoint only when the condition is true. You can use any valid MATLAB expression as a condition. This condition expression can include numerical values and any Stateflow data that is in scope at the breakpoint.

Note You cannot use message data in a breakpoint condition expression.

This example shows how to set a conditional breakpoint in the model `sf_car`. It shows how you can see `speed` increasing leading up to the car upshifting. First, set a breakpoint on the transition from `steady_state` to `upshifting` of type `When Transition is Tested`.
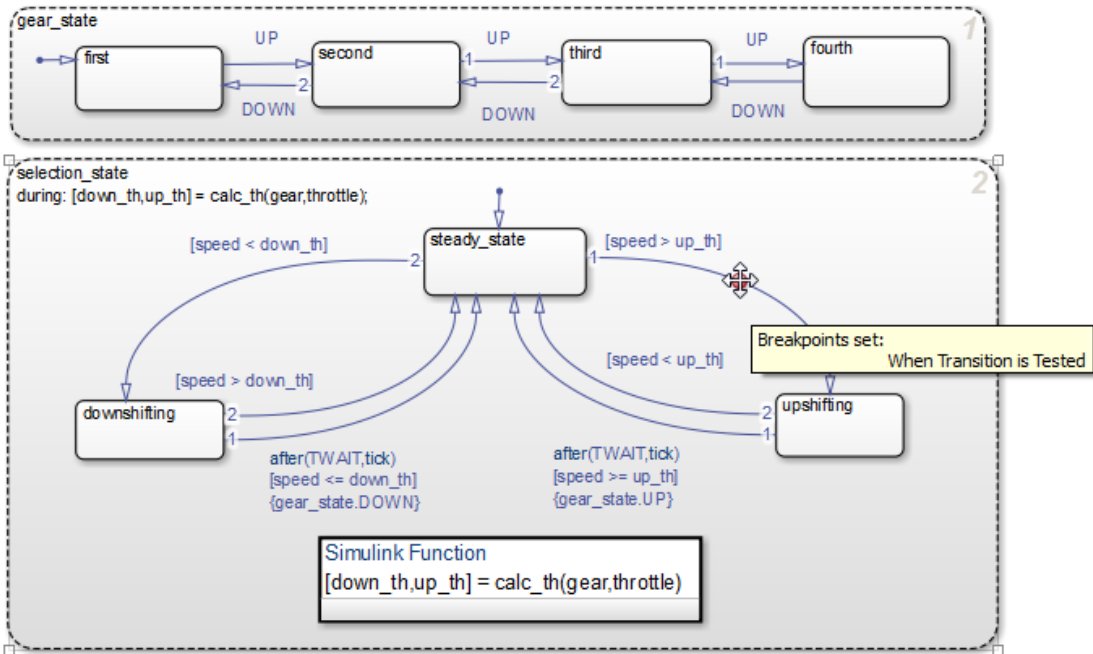
Without a condition on this breakpoint, simulation pauses every time the transition is tested, even when the value of `speed` is far below `up_th`. To inspect the chart just before the transition is taken, you want the software to pause simulation only when the value of `speed` is approaching the value of `up_th`. When you set the condition `speed > up_th-2` on the breakpoint, then simulation pauses only when the value of `speed` reaches within 2 of the value of `up_th`.
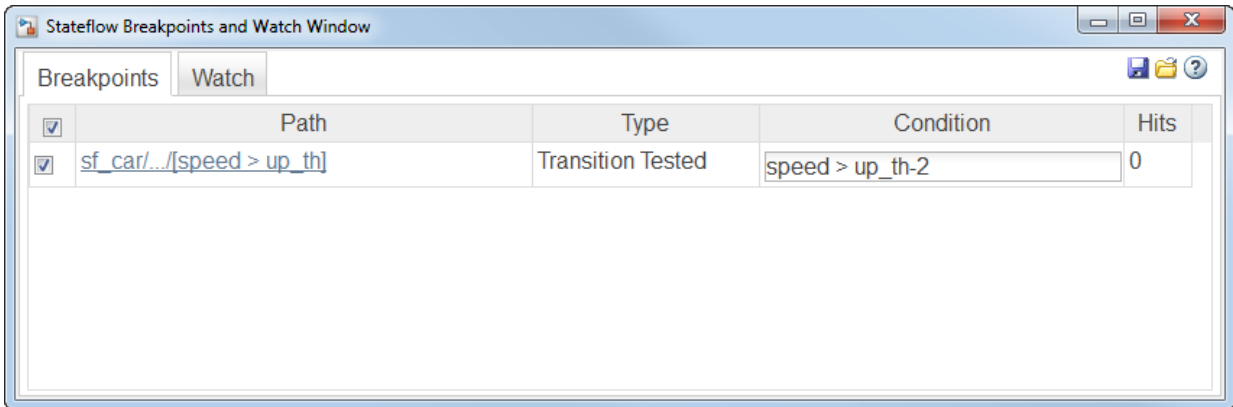
When simulation pauses, you can view the values for the variables speed and up_th in the **Watch** tab. To add each variable to the **Watch** tab:

1   In the chart, right-click the transition from steady_state to upshifting.

2   Select **Add to Watch Window**.

3   Choose the data variable name.

For more information, see "Watch Stateflow Data Values" on page 32-36.
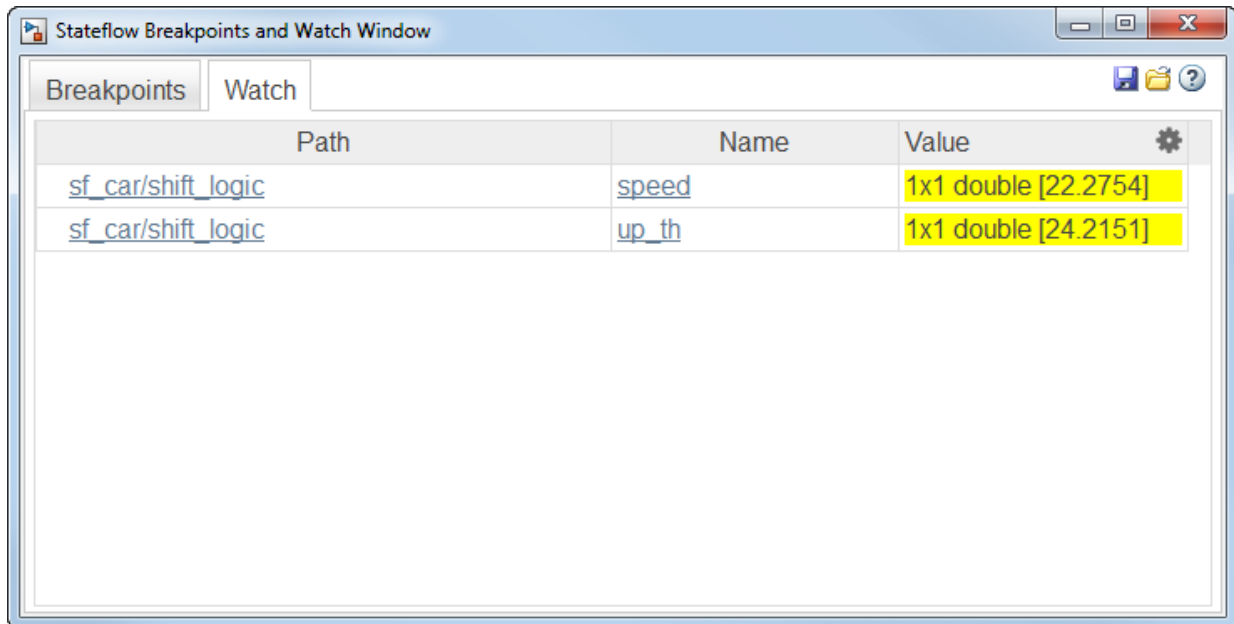
Disable and Enable Breakpoints

Disable breakpoints to prevent them from affecting chart execution. The breakpoint conditions are not deleted. The conditions are restored when you enable the breakpoint. To disable an individual breakpoint, in the **Breakpoints** tab of the Stateflow Breakpoints and Watch window, clear the check box next to the breakpoint name. For example, the following chart shows a disabled breakpoint on the state `steady_state`.
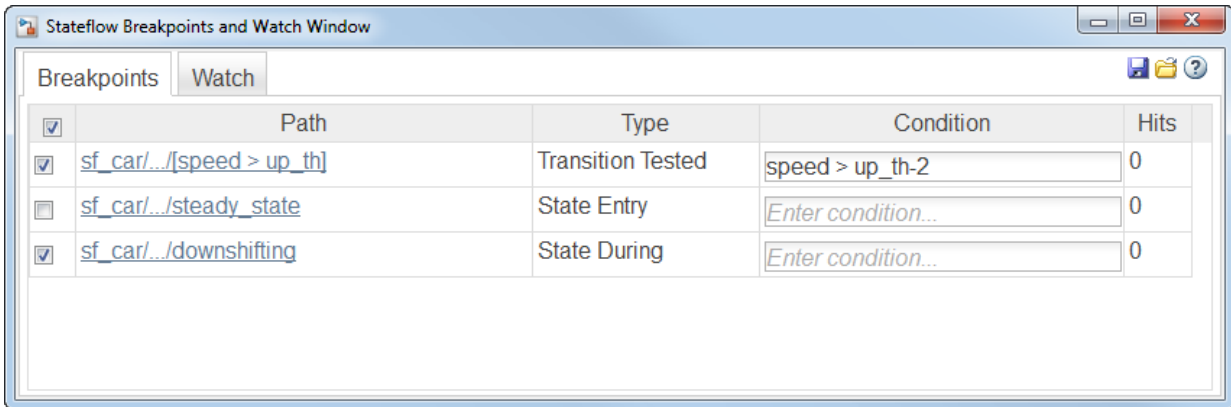
If all the breakpoints for a graphical object are disabled, its badge changes from red to gray.
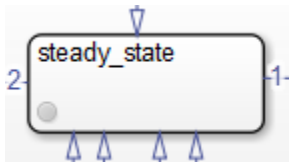
If there is at least one breakpoint enabled for an object, then the breakpoint badge remains red.

To enable the breakpoint, select the box next to the breakpoint name.

To disable and enable all Stateflow breakpoints, clear or check the box at the top of the check box column.

View Breakpoint Hits

In the **Breakpoints** tab, you can view the number of **Hits** for each breakpoint. This is the number of times simulation has paused on each breakpoint. The breakpoints **Hits** numbers resets to zero each time simulation for the model starts from the beginning, or when the condition is changed.

Clear Breakpoints and Watch Data

You can clear Stateflow breakpoints and watch data from the Stateflow Breakpoints and Watch window. Clearing a breakpoint removes it from the chart. Point to the name of the breakpoint or watch data. Click the ⊗ icon that appears to the right of the name.
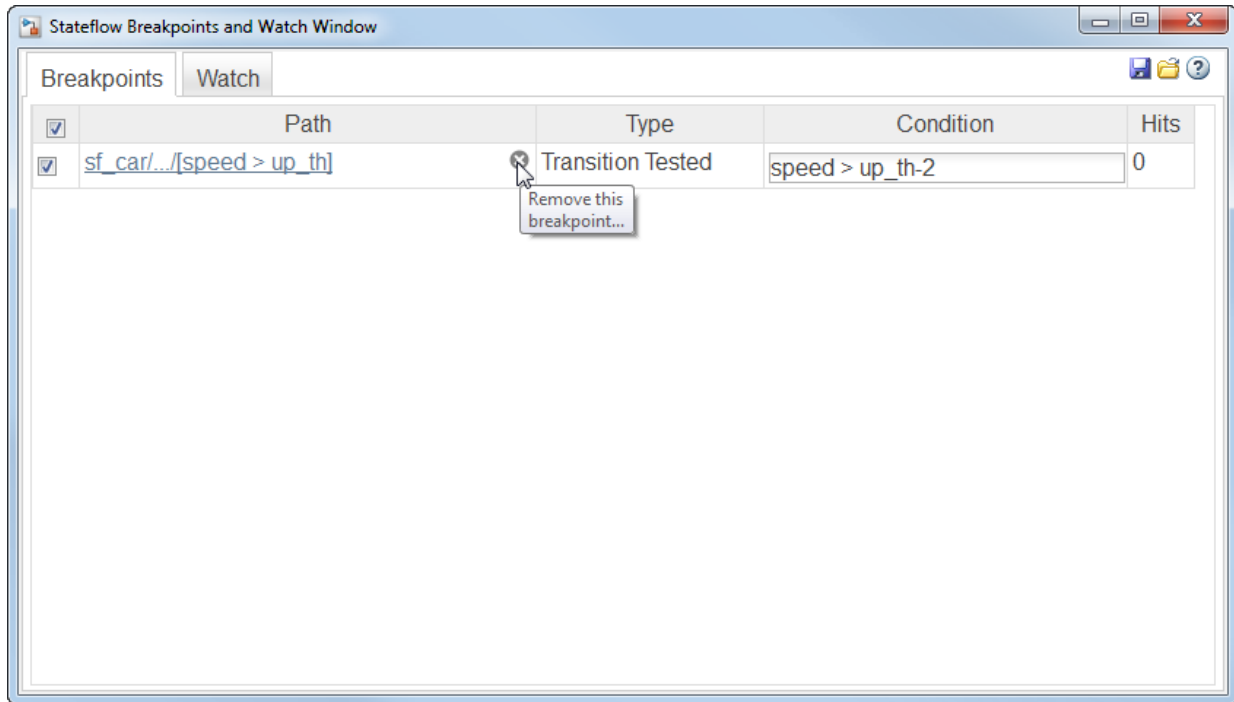
To add breakpoints and watch data, see "Set Breakpoints to Debug Charts" on page 32-5 and "Watch Stateflow Data Values" on page 32-36.

Format Watch Display

You can change the format of watch data. Select the gear icon, ⚙. From the drop-down list, choose the desired MATLAB format for each data type.
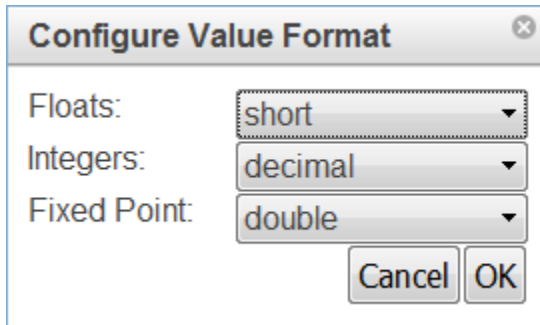
For more information on the floating point and integer formats, see `format`. For more information on fixed-point formats, see `storedInteger` and `double`.

Save and Restore Breakpoints and Watch Data

Breakpoints and watch data persist during a MATLAB session. When you close a model, breakpoints and watch data remain in the Stateflow Breakpoints and Watch window. During one MATLAB session, if you reopen a model, any breakpoints and watch data for that model are restored.

You can save a snapshot of the breakpoints and watch data list, and then reload it at any point in the current or subsequent MATLAB session. To save a snapshot, click the save icon, , and name the snapshot file. To restore a snapshot, click the restore icon, , and choose the snapshot file.

Debug Run-Time Errors in a Chart

In this section...
"Create the Model and the Stateflow Chart" on page 32-24
"Debug the Stateflow Chart" on page 32-25
"Correct the Run-Time Error" on page 32-26

Create the Model and the Stateflow Chart

In this topic, you create a Simulink model with a Stateflow chart to debug. Follow these steps:

1 Create the following Simulink model:
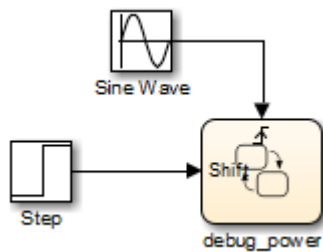
2 Add the following states and transitions to your chart:
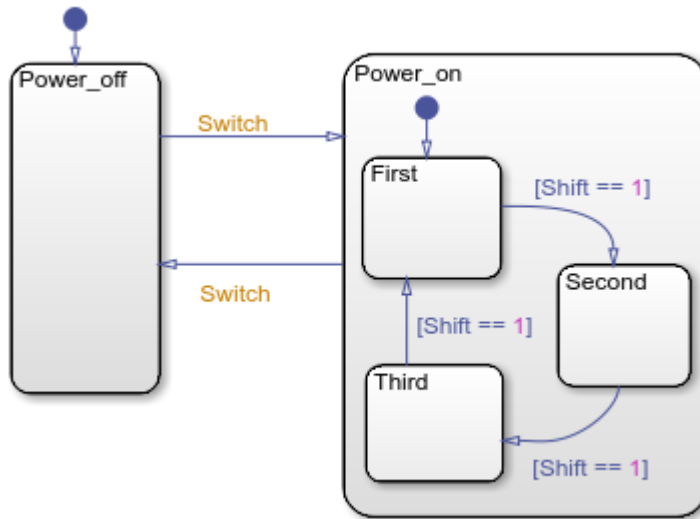
3. In your chart, add an event `Switch` with a scope of **Input from Simulink** and a **Rising** edge trigger.

4. Add a data `Shift` with a scope of **Input from Simulink**.

The chart has two states at the highest level in the hierarchy, `Power_off` and `Power_on`. By default, `Power_off` is active. The event `Switch` toggles the system between the `Power_off` and `Power_on` states. `Power_on` has three substates: `First`, `Second`, and `Third`. By default, when `Power_on` becomes active, `First` also becomes active. When `Shift` equals 1, the system transitions from `First` to `Second`, `Second` to `Third`, `Third` to `First`, for each occurrence of the event `Switch`, and then the pattern repeats.

In the model, there is an event input and a data input. A Sine Wave block generates a repeating input event that corresponds with the Stateflow event `Switch`. The Step block generates a repeating pattern of 1 and 0 that corresponds with the Stateflow data object `Shift`. Ideally, the `Switch` event occurs at a frequency that allows at least one cycle through `First`, `Second`, and `Third`.

Debug the Stateflow Chart

To debug the chart in "Create the Model and the Stateflow Chart" on page 32-24, follow these steps:

1   Right-click in the chart, and select **Set Breakpoint on Chart Entry**.

2   Start the simulation.

Because you specified a breakpoint on chart entry, execution stops at that point.

3   Click the Step In button, .

The Step In button executes the next step and stops.

4   Continue clicking the Step In button and watching the animating chart.

After each step, watch the chart animation to see the sequence of execution.

Single-stepping shows that the chart does not exhibit the desired behavior. The transitions from `First` to `Second` to `Third` inside the state `Power_on` are not occurring because the transition from `Power_on` to `Power_off` takes priority. The output display of code coverage also confirms this observation.

Correct the Run-Time Error

In "Debug the Stateflow Chart" on page 32-25, you step through a simulation of a chart and find an error: the event `Switch` drives the simulation but the simulation time passes too quickly for the input data object `Shift` to have an effect.

Correct this error as follows:

1   Stop the simulation so that you can edit the chart.

2   Add the condition `[t > 20.0]` to the transition from `Power_on` to `Power_off`.
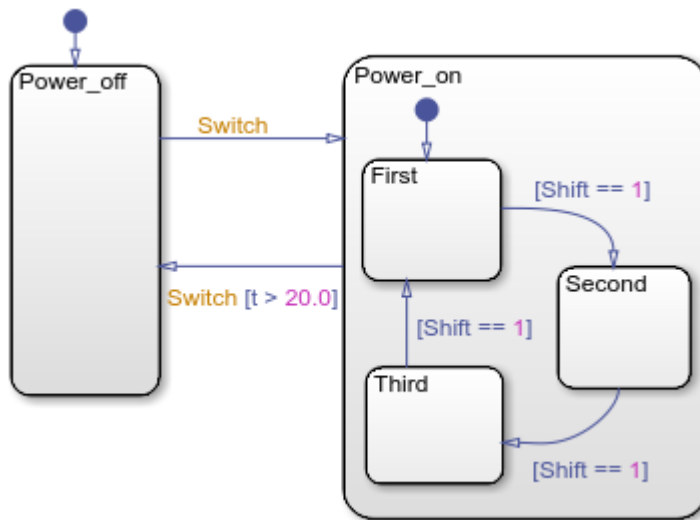
Now the transition from `Power_on` to `Power_off` does not occur until simulation time is greater than 20.0.

3 Begin simulation again.

4 Click the Step In button repeatedly to observe the new behavior.

Detect Common Modeling Errors During Chart Simulation

To avoid common design errors, you can run diagnostic checks that test the completeness of your Stateflow chart during compilation and simulation. Stateflow diagnostics detect state inconsistencies, violations in data ranges, and cyclic behavior in Stateflow charts in Simulink models.

State Inconsistencies

In a Stateflow chart, states are inconsistent if they violate one of these rules:

- An active state with exclusive (OR) decomposition and at least one substate has exactly one active substate.
- All substates of an active state with parallel (AND) decomposition are active.
- All substates of an inactive state are inactive regardless of the state decomposition.

While you edit your chart, the Stateflow editor displays potential causes for state inconsistencies by highlighting objects in red or orange. For more information, see "Detect Modeling Errors During Edit Time" on page 3-28.

Causes of State Inconsistency

One type of state inconsistency occurs when all of these conditions are true:

- A state with multiple substates has exclusive (OR) decomposition.
- The state does not contain a history junction.
- Every default path that leads to a substate is guarded by a condition or triggered by an event.
- Not all transitions into the state are supertransitions leading directly to a substate.

For example, this chart has a state inconsistency because there is no default transition to indicate which substate becomes active first.
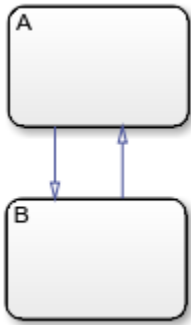
Adding an unconditional default transition to one of the states resolves the state inconsistency.
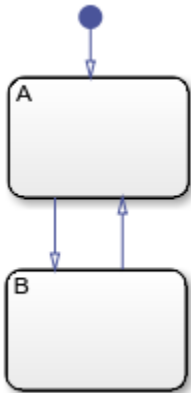
Detect State Inconsistencies

At compile time, Stateflow charts detect state inconsistencies caused by the omission of an unconditional default transition. To control the level of diagnostic action, open the Model Configuration Parameters dialog box. In the **Diagnostics > Stateflow** pane, for the diagnostic **No unconditional default transitions**, you can select `error`, `warning`, or `none`. The default setting is `error`. For more information, see "No unconditional default transitions" (Simulink).

Data Range Violations

During simulation, a data range violation occurs when:

- An integer or fixed-point operation overflows the numeric capacity of its result type. See "Handle Integer Overflow for Chart Data" on page 9-45 and "Fixed-Point Operations in Stateflow Charts" on page 22-32.

- The value of a data object is outside the range of the values specified by the **Initial value**, **Minimum**, and **Maximum** properties. See "Initial Value" on page 9-9 and "Limit Range Properties" on page 9-10.

For example, this chart contains local data `a` that has a **Minimum** value of 0 and a **Maximum** value of 2. The `entry` action in state A initializes `a` to 1. The `during` action increments the value of `a` by 1. After two time steps, the value of `a` exceeds its specified range, resulting in a data range violation.
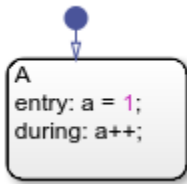
Detect Data Range Violations

At run time, Stateflow charts detect data range violations. To control the level of diagnostic action, open the Model Configuration Parameters dialog box. In the **Diagnostics > Data Validity** pane, you can select `error`, `warning`, or `none` for these diagnostics:

- **Simulation range checking** detects violations based on minimum-and-maximum range checks. The default setting is `none`.

- **Wrap on overflow** and **Saturate on overflow** detect violations that occur when integer or fixed-point operations exceed the numeric capacity of their result type. The default setting is `warning`.

For more information, see "Simulation range checking" (Simulink), "Wrap on overflow" (Simulink), and "Saturate on overflow" (Simulink).

Cyclic Behavior

Cyclic behavior occurs when a step or sequence of steps is repeated indefinitely during chart simulation.

For example, the actions in this chart produce an infinite cycle of recursive event broadcasts.

- The `during` action in state `C` broadcasts the event `E1`.
- The event `E1` triggers the transition from state `A.A1` to state `A.A2`. The condition action for this transition broadcasts the event `E2`.
- The event `E2` triggers the transition from state `B.B1` to state `B.B2`. The condition action for this transition broadcasts the event `E1`.

The event broadcasts in states `A` and `B` occur in condition actions, so the transitions do not take place until the chart processes the resulting events. The substates `A.A1` and `B.B1` remain active, so new event broadcasts continue to trigger the transitions and the process repeats indefinitely.
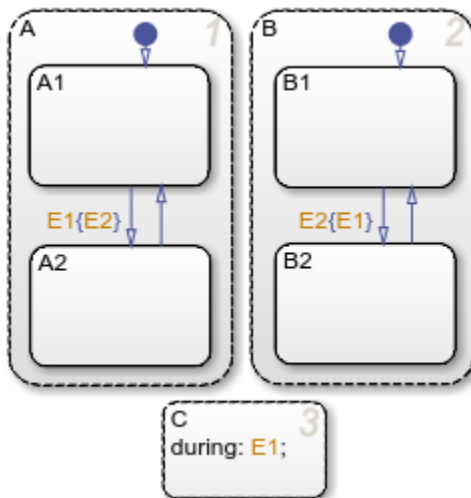
Because undirected local event broadcasts can cause unwanted recursive behavior, use of the `send` operator to broadcast directed local events is recommended. For more information, see "Broadcast Local Events to Synchronize Parallel States" on page 12-45.

Detect Cyclic Behavior During Simulation

During chart simulation, Stateflow charts use cycle detection algorithms to detect a class of infinite recursions caused by event broadcasts. Enable cycle detection through the **Simulation > Debug > MATLAB & Stateflow Error Checking Options > Detect Cycles** check box. Cyclic behavior checking is selected by default.

Stateflow charts also detect undirected local event broadcasts. To control the level of diagnostic action, open the Model Configuration Parameters dialog box. In the **Diagnostics > Stateflow** pane, for the **Undirected event broadcasts** diagnostic, you can select `error`, `warning`, or `none`. The default setting is `warning`. For more information, see "Undirected event broadcasts" (Simulink).

Limitations of Cyclic Behavior Detection

Stateflow cycle detection is limited to cases of recursion due to event broadcasts and does not extend to other types of cyclic behavior.

For instance, Stateflow cannot detect the infinite cycle in this flow chart. In this example, the default transition initializes the local data `i` to 0. The next transition segment increments `i`. The transition to the terminating junction is valid only when the condition `[i < 0]` is true. Because this condition is never true, an infinite cycle results.
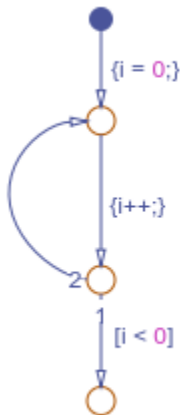
The model `sf_cycle_error_fix` provides suggestions for fixing cyclic behavior in flow charts. At the MATLAB command prompt, enter:

```
sfhelp('cycle_error');
```

See Also

More About

- "Detect Modeling Errors During Edit Time" on page 3-28
- "Model Configuration Parameters: Stateflow Diagnostics" (Simulink)

Guidelines for Avoiding Unwanted Recursion in a Chart

Recursion can be useful for controlling substate transitions among parallel states at the same level of the chart hierarchy. For example, you can send a directed event broadcast from one parallel state to a sibling parallel state to specify a substate transition. (For details, see "Directed Event Broadcasting" on page 12-45.) This type of recursive behavior is desirable and efficient.

However, unwanted recursion can also occur during chart execution. To avoid unwanted recursion, follow these guidelines:

Do not call functions recursively

Suppose that you have functions named f, g, and h in a chart. These functions can be any combination of graphical functions, truth table functions, MATLAB functions, or Simulink functions.

To avoid recursive behavior, do not:

- Have f calling g calling h calling f
- Have f, g, or h calling itself

Do not use undirected local event broadcasts

Follow these rules:

- Use directed local event broadcasts with the syntax send(*event*,*state*). The *event* is a local event in the chart and the *state* is the destination state that you want to wake up using the event broadcast.
- If the source of the local event broadcast is a state action, ensure that the destination state is *not* an ancestor of the source state in the chart hierarchy.
- If the source of the local event broadcast is a transition, ensure that the destination state is *not* an ancestor of the transition in the chart hierarchy.

  Also, ensure that the transition does not connect to the destination state.

If you have *undirected* local event broadcasts in state actions or condition actions in your chart, a warning appears by default during simulation. Examples of state actions with undirected local event broadcasts include:

- `entry: send(E1)`, where `E1` is a local event in the chart
- `exit: E2`, where `E2` is a local event in the chart

You can control the level of diagnostic action for undirected local event broadcasts in the **Diagnostics** > **Stateflow** pane of the Configuration Parameters dialog box. Set the **Undirected event broadcasts** diagnostic to `none`, `warning`, or `error`.

Watch Stateflow Data Values

Watch Data in the Stateflow Chart

During simulation, if you point at an object in the chart, a tooltip displays the value of the data and the messages that the selected object uses.

Object Type	Tooltip Information
States and transitions	Values of data, messages, and temporal logic expressions that the object uses
Graphical, truth table, and MATLAB functions	Values of local data, messages, inputs, and outputs in the scope of the function

Note If you select the chart properties **Export Chart Level Functions** and **Treat Exported Functions as Globally Visible**, then the tooltip does not display temporal logic data.

For example, this chart stops execution before entering the `second` state. Pointing to the superstate `gear` displays a tooltip showing the values of:

- Data such as `speed`, `up_threshold`, and `TWAIT`.
- Temporal logic expressions such as `duration(speed >= up_threshold)`.
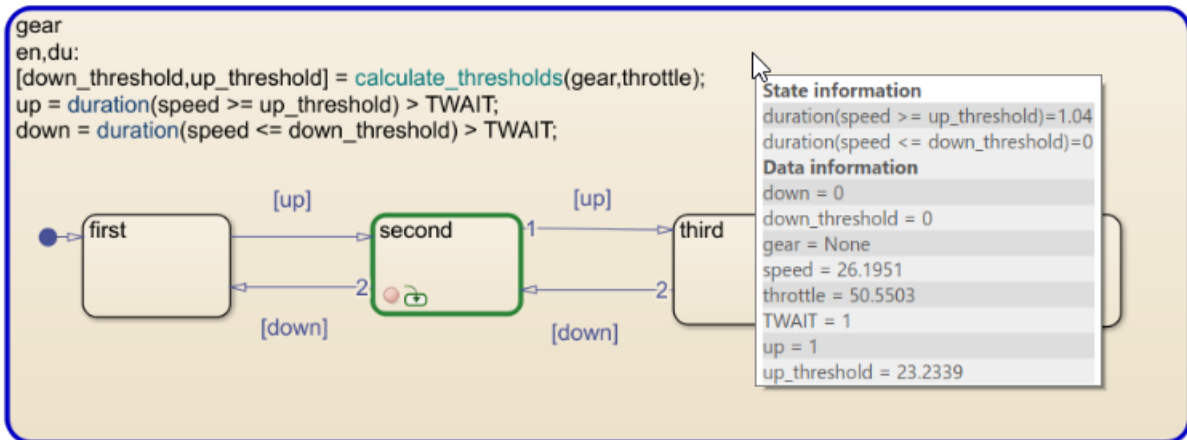
Because the value of `duration(speed >= up_threshold)` is greater than TWAIT, the chart takes the transition from `first` to `second`.

A chart determines the value of data and temporal logic expressions at different stages of a time step. For instance, in the previous example, the chart computes temporal information at the start of each time step and updates `speed` at the end of each time step.

If you advance through the simulation using the **Step Forward** option and observe data at the end of each time step, the temporal information in the tooltip can appear to lag behind the rest of the data. After observing the value of `speed` cross the `up_threshold`, you must step forward twice before `duration(speed >=`

`up_threshold)` becomes nonzero. To avoid this behavior, use the **Step Over** option and observe the data at shorter intervals.

Watch Stateflow Data in the MATLAB Command Window

When simulation reaches a breakpoint, you can view the values of Stateflow data in the MATLAB Command Window. In the following chart, a default transition calls a MATLAB function:
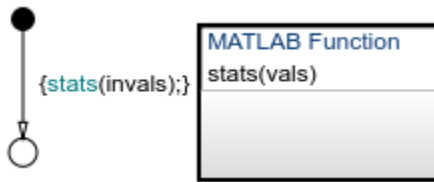
A breakpoint is set at the last executable line of the function:

```
function stats(vals)
%#codegen

% calculates a statistical mean and standard deviation
% for the values in vals.

len = length(vals);
mean = avg(vals, len);
stdev = sqrt(sum(((vals-avg(vals,len)).^2))/len);
coder.extrinsic('plot');
plot(vals,'-+');  % Breakpoint set at this line
```

When simulation reaches the breakpoint, you can display Stateflow data in the MATLAB Command Window.

1   At the MATLAB prompt, press **Enter**.

      A `debug>>` prompt appears.

2   Type `whos` to view the data that is visible at the current scope.

3   Enter the name of data array `vals` at the prompt to display its value.

4   Enter `vals(2:3)` to view specific values of that array.

The Command Line Debugger provides these commands during simulation:

Command	Description
dbstep	Advance to next executable line of code.

Command	Description
dbstep [in/ out]	When debugging MATLAB functions in a chart: • dbstep [in] advances to the next executable line of code. If that line contains a call to another function, execution continues to the first executable line of the function. • dbstep [out] executes the rest of the function and stops just after leaving the function.
dbcont	Continue execution to next breakpoint.
dbquit (ctrl-c)	Stop simulation of the model. Press **Enter** after this command to return to the command prompt.
help	Display help for command-line debugging.
print *var* ...or... *var*	Display the value of the variable *var*.
var(*i*)	Display the value of the *i*th element of the vector or matrix *var*.
var(*i:j*)	Display the value of a submatrix of the vector or matrix *var*.
save	Saves all variables to the specified file. Follows the syntax of the MATLAB save command. To retrieve variables in the MATLAB base workspace, use the load command after simulation has ended.
whos	Display the size and class (type) of all variables in the scope of the halted MATLAB function in your chart.

You can issue any other MATLAB command at the debug>> prompt but the results are executed in the Stateflow workspace. For example, you can issue the MATLAB command plot(*var*) to plot the values of the variable *var*.

To issue a command in the MATLAB base workspace at the debug>> prompt, use the evalin command with the first argument 'base' followed by the second argument command, for example, evalin('base','whos').

To return to the MATLAB base workspace, use the dbquit command.

Watch Data in the Stateflow Breakpoints and Watch Window

In the **Watch** tab of the Stateflow Breakpoints and Watch window, you can view current data and messages when simulation pauses. The **Watch** tab displays a list of watch data, and highlights the values that changed since the last time simulation paused. In the **Watch** tab, you can expand a message to view the message queue and message data values. To add Stateflow data or messages to the watch data list:

1 In the chart, right-click an object that uses the data or message.
2 Select **Add to Watch Window**.
3 Choose the data or message.

To add active state data and truth table data to the watch list, from the Model Explorer, open the Data Properties dialog box. Select **Add to Watch Window**.

You can choose the display format for each data type. For more information, see "Format Watch Display" on page 32-22.

Change Data Values During Simulation

How to Change Values of Stateflow Data

When your chart is in debug mode, you can test the simulation by changing the values of data in the chart. After the debug>> prompt appears, as described in "Watch Stateflow Data in the MATLAB Command Window" on page 32-37, you can assign a different value to your data. To change a data value, enter the new value at the prompt using the following format:

data_name = new_value

To change data values directly in the Symbols Window, see "Manage Data, Events, and Messages in the Symbols Window" on page 33-2.

For a list of data that you cannot change, see "Data That Is Read-Only During Simulation" on page 32-44. You cannot change message data values at the command line.

Examples of Changing Data Values

Scalar Example

Suppose that, after the debug>> prompt appears, you enter whos at the prompt and see the following data:

```
Name         Size       Bytes  Class

airflow      1x1            1  uint8 array
temp         1x1            8  double array
```

To change...	To this value...	Enter...
airflow	2	airflow = uint8(2)
temp	68.75	temp = 68.75

If you try to enter airflow = 2, you get an error message because MATLAB interprets that expression as the assignment of a double value to data of uint8 type. For reference, see "Cases When Casting Is Necessary" on page 32-45.

Multidimensional Example

Suppose that, after the `debug>>` prompt appears, you enter `whos` at the prompt and see the following data:

```
Name                    Size                Bytes  Class

ball_interaction        16x16                 256  int8 array
last_vel                16x2                  256  double array
stopped                 16x1                   16  int16 array
```

To change...	To this value...	Enter...
The element in row 8, column 8 of `ball_interaction`	1	`ball_interaction(8,8) = int8(1)`
The element in row 16, column 1 of `last_vel`	120.52	`last_vel(16,1) = 120.52`
The last element in `stopped`	0	`stopped(16) = int16(0)`

One-based indexing applies when you change values of Stateflow data while the chart is in debug mode.

Variable-Size Example

Suppose that, after the `debug>>` prompt appears, you enter `whos` at the prompt and see the following data:

```
Name   Size   Bytes  Class

y1     1x1       8  double array (variable sized: MAX 16x16)
y2     1x1       8  double array (variable sized: MAX 16x4)
```

To change...	To...	Enter...
y1	A 10-by-5 array of ones	`y1 = ones(10,5)`
y2	A 6-by-4 array of zeros	`y2 = zeros(6,4)`

Changing the dimensions of variable-size data works only when the new size does not exceed the dimension bounds.

Fixed-Point Example

Suppose that, after the `debug>>` prompt appears, you enter `whos` at the prompt and see the following data:

```
 Name        Size      Bytes  Class

 y_n1        1x1          2  fixpt (int16 array (2^-10)*SI)
 x_n1        1x1          2  fixpt (int16 array (2^-12)*SI)
```

Both `y_n1` and `x_n1` have signed fixed-point types, with a word length of 16. `y_n1` has a fraction length of 10 and `x_n1` has a fraction length of 12.

To change...	To this fixed-point value...	Enter...
y_n1	0.5410	y_n1 = fi(0.5412,1,16,10)
x_n1	0.4143	x_n1 = fi(0.4142,1,16,12)

For more information about using `fi` objects, see the Fixed-Point Designer documentation.

Enumerated Example

Suppose that, after the `debug>>` prompt appears, you enter `whos` at the prompt and see the following data:

```
Name                 Size      Bytes  Class

CurrentRadioMode     1x1          4  int32 array
MechCmd              1x1          4  int32 array
```

Assume that `CurrentRadioMode` and `MechCmd` use the enumerated types `RadioRequestMode` and `CdRequestMode`, respectively.

To change...	To this enumerated value...	Enter...
CurrentRadioMode	CD	CurrentRadioMode = RadioRequestMode.CD

To change...	To this enumerated value...	Enter...
MechCmd	PLAY	`MechCmd = CdRequestMode.PLAY`

You must include the enumerated type explicitly in the assignment. Otherwise, an error appears at the `debug>>` prompt.

Limitations on Changing Data Values

Data That Is Read-Only During Simulation

You cannot change data of the following scopes while the chart is in debug mode:

- Constant
- Input

Limitations on Changing Type and Size

The following data properties cannot change:

- Data type
- Size

However, for variable-size data, you can change the dimensions of the data as long as the size falls within the dimension bounds. For example, `varsizedData = ones(5,7);` is a valid assignment for a variable-size 10-by-10 array.

Limitations for Fixed-Point Data

- Do not assign a value that falls outside the range of values that the fixed-point type can represent. Avoid selecting a value that causes overflow.
- Sign, word length, fraction length, slope, and bias cannot change.

Limitations for Structures

- You cannot change the data type or size of any fields.
- Addition or deletion of fields does not work because the size of the structure cannot change.

Cases When Casting Is Necessary

When you change a data value, you must explicitly cast values for data of the following built-in types:

- `single`
- `int32`
- `int16`
- `int8`
- `uint32`
- `uint16`
- `uint8`

For example, the following assignments are valid:

- `my_data1 = uint8(2)`
- `my_data2 = single(5.3)`

Casting is not necessary when you change the value of data that is of type `double`.

Monitor Test Points in Stateflow Charts

This example shows you how to specify data or states as test points that you can plot with a floating scope or log to the MATLAB® base workspace during simulation.

About Test Points in Stateflow Charts

A Stateflow® test point is a signal that you can observe during simulation, for example, by using a Floating Scope block. You can designate states or local data with these properties as test points:

- **Size** - scalar, one-dimensional, or two-dimensional
- **Type** - any data type except `ml`
- **Location** - descendant of a Stateflow chart

You can specify individual data or states as test points by setting their **TestPoint** property via the Stateflow API, in the Property Inspector, or in the Model Explorer.

You can monitor individual Stateflow test points with a floating scope during model simulation. You can also log test point values into MATLAB workspace objects.

You can also use active state output to view or log state activity data in Simulink®. For more information, see "Monitor State Activity Through Active State Data" on page 24-26.

Set Test Points for Stateflow States and Data with the Property Inspector

You can explicitly set individual states, local data, and output data as test points in the Model Explorer. The following procedure shows how to set individual test points for Stateflow states and data.

1. Open the model.
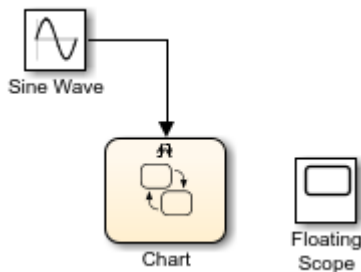
In the Stateflow chart, state A and its substate X are entered on the first `tic` event. State A and substate X stay active until 10 `tic` events have occurred, and then state B is entered. On the next event, state A and substate X are entered and the cycle continues. The data `data` belongs to substate X. The entry and during actions for substate X increment `data` while X is active for 10 `tic` events. When state B is entered, `data` reinitializes to zero, and then the cycle repeats.
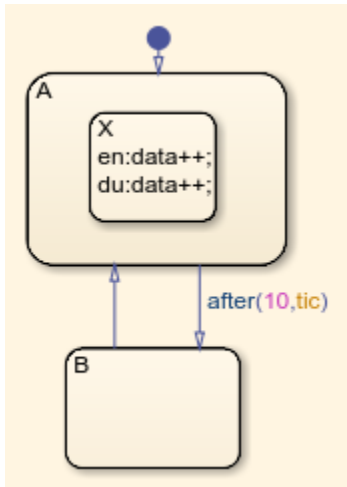
2. Open the Property Inspector, **View > Property Inspector**.

3. Select state A. Then, in the **Logging** section of the Property Inspector, select **Test Point**.

4. Repeat this for state X and B.

5. Open the Symbol viewer, **View > Symbols**.

6. Select the data `data`. Then, in the **Logging** section of the Property Inspector, select **Test Point**.

You can also log these test points. For instructions, see "Log Multiple Signals" on page 32-52.

Monitor Data Values and State Self Activity Using a Floating Scope

In this section, you configure a Floating Scope block to monitor a data value and the self activity of a state.

1. Open the model.
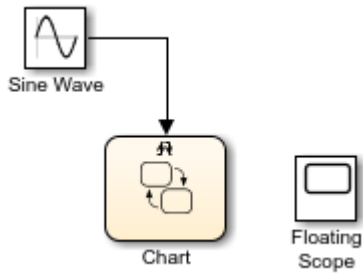
2. Double-click the Floating Scope block.

3. In the Floating Scope window, select **Simulation > Signal Selector**.

4. In the **Model hierarchy** pane, select the chart and in the **List contents**, select the signals.
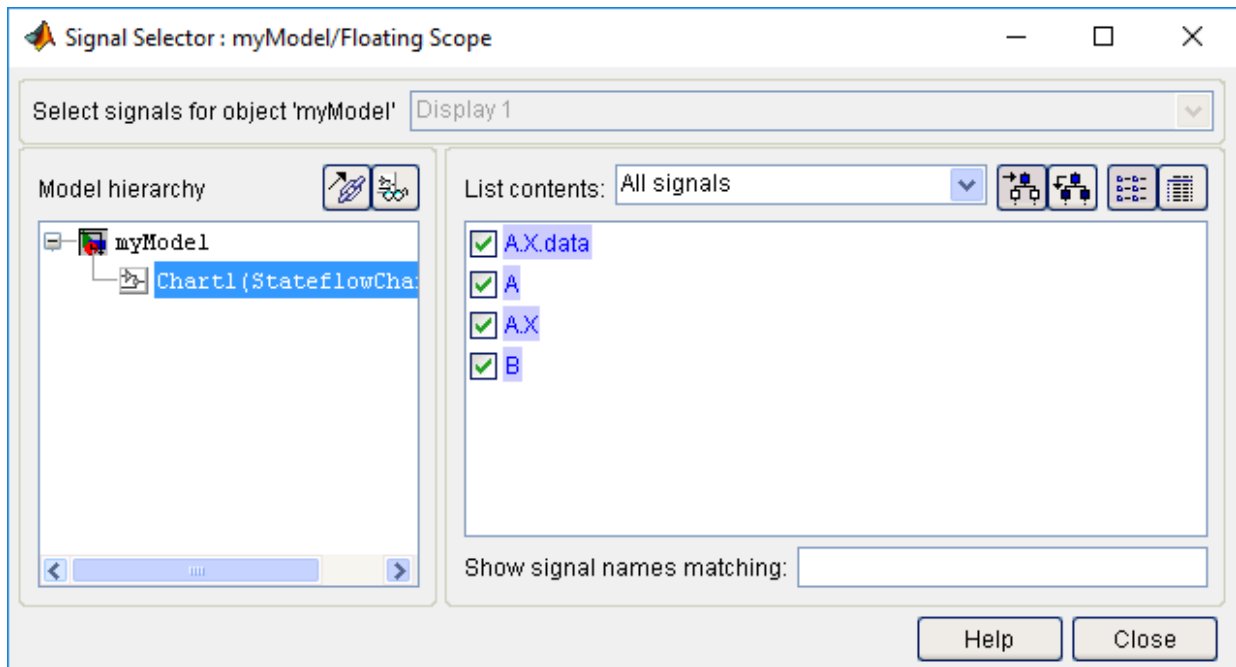
5. Simulate the model.
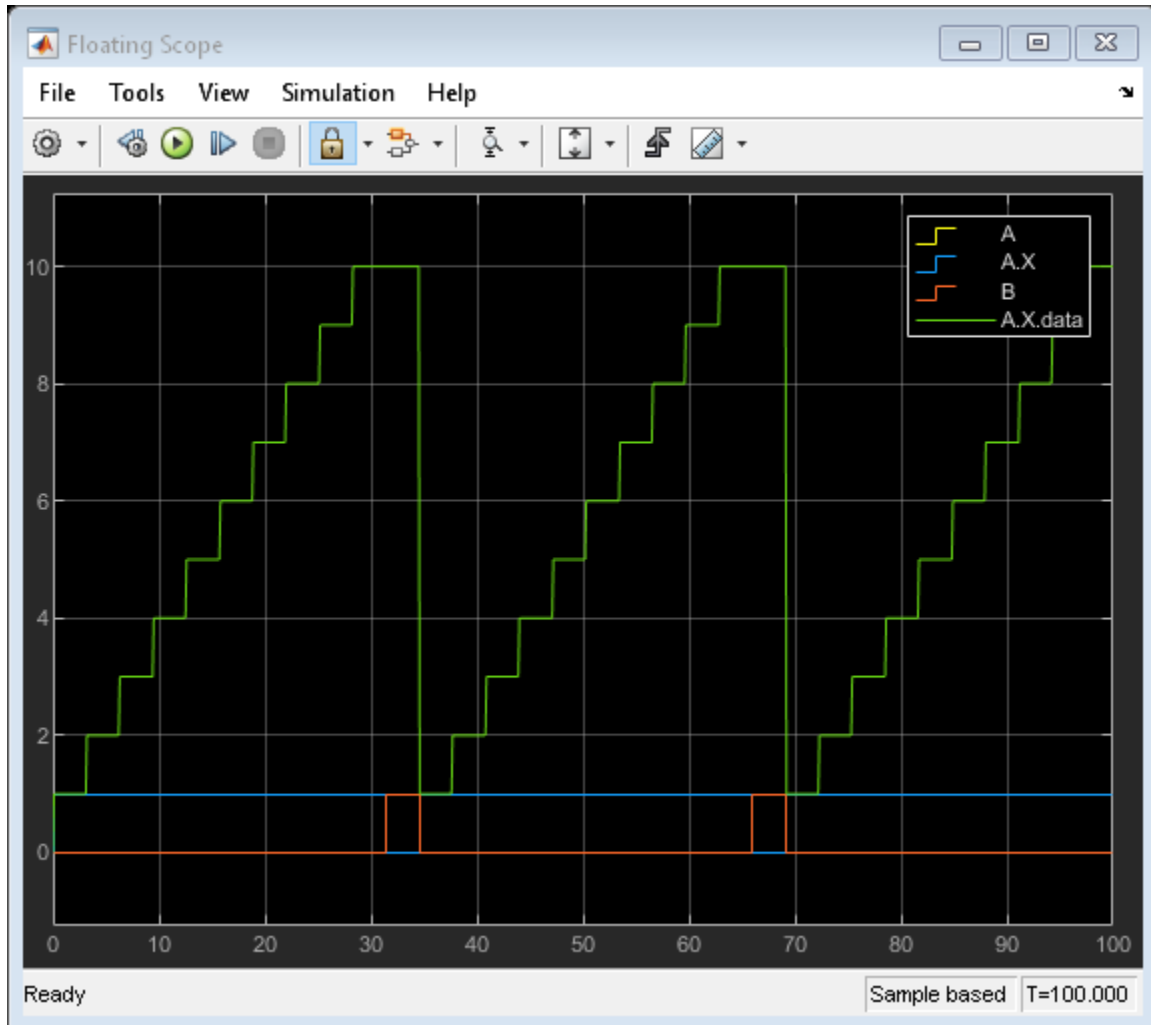
When state A.X is active, the signal value is 1. When that state is inactive, the signal value is 0. Because this value can be very low or high compared to other data, you might want to add a second Floating Scope block to compare the activity signal with other data.

See Also

More About

*   "Test Points" (Simulink)

Log Simulation Output for States and Data

When you simulate a Stateflow chart in a Simulink model, you can log values for local, output, and active state data into a `Simulink.SimulationData.Dataset` object. After simulation, you can access this object through the Simulation Data Inspector or in the MATLAB workspace. The workflow for logging data is:

1 Enable signal logging for the chart and choose a logging format. See "Enable Signal Logging" on page 32-51.

2 Configure states and data for signal logging. See "Configure States and Data for Logging" on page 32-52.

3 Simulate the chart.

4 Access the logged data. See "Access Signal Logging Data" on page 32-54.

Enable Signal Logging

Signal logging is enabled by default for models and charts. To disable or reenable signal logging:

1 Open the Model Configuration Parameters dialog box.

2 Select **Data Import/Export**.

3 In the **Signals** pane, select the **Signal logging** check box to enable logging for the chart. To disable logging, clear the check box.

4 (Optional) Specify a custom name for the signal logging object. The default name is `logsout`. Using this object, you can access the logging data in a MATLAB workspace variable.

5 (Optional) In the **Format** field, select a signal logging format. Options include:

- `Array`
- `Structure`
- `Structure with time`
- `Dataset`

The default setting is `Dataset`. For more information, see "Time, State, and Output Data Format" (Simulink).

6 Click **OK**.

Configure States and Data for Logging

You can set logging properties for states, local data, and output data from inside the chart, through the Stateflow Signal Logging dialog box, or programmatically from the command line.

Log Individual States and Data

Configure logging properties for one state or data object at a time through the Property Inspector, the Model Explorer, or the properties dialog box for the state or data object. Select the **Logging** tab and modify properties as needed. For more information, see "Logging Properties" on page 9-18.

For example, in the `sf_semantics_hotel_checkin` model:

1 Open the `Hotel` chart.
2 Open the Symbols window by selecting **View** > **Symbols**.
3 Open the Property Inspector by selecting **View** > **Property Inspector**.
4 Configure the `service` local data for logging.

   a In the Symbols window, select `service`.
   b In the Property Inspector, on the **Logging** tab, select the **Log signal data** check box.

5 Configure the `Dining_area` state for logging.

   a On the Stateflow editor, select the `Dining_area` state.
   b In the Property Inspector, on the **Logging** tab, select the **Log self activity** check box.
   c By default, the logging name for this state is the hierarchical signal name `Check_in.Checked_in.Executive_suite.Dining_area`. To assign a shorter name to the state, in the **Logging Name** field, select `Custom` and enter `Dining Room`.

Log Multiple Signals

Configure logging properties for multiple states and data objects through the Stateflow Signal Logging dialog box. Select which chart objects to log from a list of all states, local, and output data. For more information, see "Logging Properties" on page 9-18.

For example, in the `sf_semantics_hotel_checkin` model:

1 Open the `Hotel` chart.

2 Open the Stateflow Signal Logging dialog box by selecting **Simulation** > **Output** > **Log Chart Signals**.

3 Select the check boxes next to the `Check_in.Checked_in.Executive_suite.Dining_area` state and the local variable `service`.

4 Change the logging name of `Check_in.Checked_in.Executive_suite.Dining_area` to the custom name `Dining Room`.

5 Click **OK**.
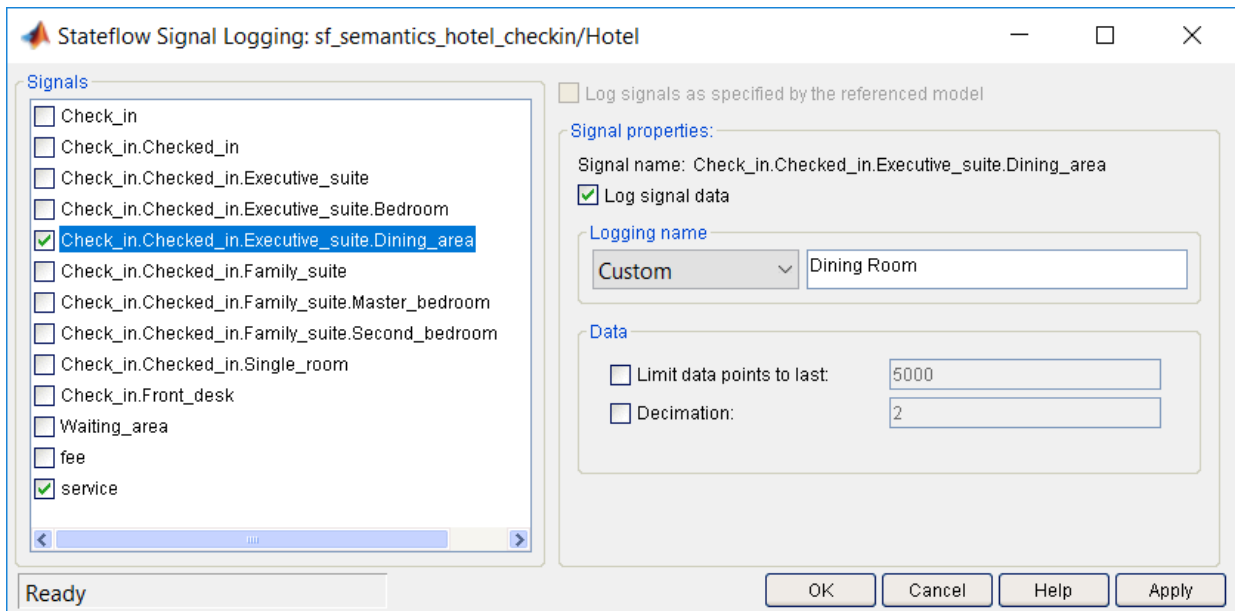
Log Chart Signals by Using the Command-Line API

Configure logging properties for states and data objects programmatically from the command line. To enable logging for a states or data object, get a handle for the object and set its `LoggingInfo.DataLogging` property to `1`. For more information on the Stateflow Programmatic Interface, see "Logging Properties".

For example, in the `sf_semantics_hotel_checkin` model:

1. Open the `Hotel` chart.

2. Get a handle for the `Dining_area` state:

```
rt = sfroot;
diningState = rt.find('-isa','Stateflow.State','Name','Dining_area');
```

3. Get a handle for the local data `service`:

```
serviceData = rt.find('-isa','Stateflow.Data','Name','service');
```

4. Enable logging for the `Dining_area` state and the `service` data:

```
diningState.LoggingInfo.DataLogging = 1;
serviceData.LoggingInfo.DataLogging = 1;
```

5. Change the logging name of the `Dining_area` state to the custom name `Dining Room`:

```
% Enable custom naming
diningState.LoggingInfo.NameMode = 'Custom';

% Enter the custom name
diningState.LoggingInfo.LoggingName = 'Dining Room';
```

Access Signal Logging Data

During simulation, Stateflow saves logged data in a
`Simulink.SimulationData.Dataset` signal logging object. The default name of the
signal logging object is `logsout`.

For example, suppose that you configure the `sf_semantics_hotel_checkin` model to
log the `service` local data and the activity of the `Dining_area` state. After starting the
simulation, you check into the hotel by toggling the first switch and order room service
multiple times by toggling the second switch. After stopping the simulation, you can view
the logged data through the Simulation Data Inspector or in the MATLAB workspace.

View Logged Data Through the Simulation Data Inspector

When you simulate the model, the Simulation Data Inspector icon is highlighted to
indicate that it has new simulation data.

1.
   To open the Simulation Data Inspector, click the icon .

2. Inspect and compare the signals logged during simulation. See Simulation Data
   Inspector.

View Logged Data in the MATLAB Workspace

1 To access the signal logging object, at the MATLAB command prompt, enter:

```
logsout

logsout =

Simulink.SimulationData.Dataset 'logsout' with 2 elements

                         Name          BlockPath
                         _____  _____
    1  [1x1 State]       Dining Room   sf_semantics_hotel_checkin/Hotel
    2  [1x1 Data ]       service       sf_semantics_hotel_checkin/Hotel
```

2 To access logged elements, use the `get` method. You can access logged elements by name, index, or block path.

```
diningLog = logsout.get('Dining Room')

diningLog =

  Stateflow.SimulationData.State
  Package: Stateflow.SimulationData

  Properties:
        Name: 'Dining Room'
    BlockPath: [1×1 Simulink.SimulationData.BlockPath]
       Values: [1×1 timeseries]

serviceLog = logsout.get('service')

serviceLog =

  Stateflow.SimulationData.Data
  Package: Stateflow.SimulationData

  Properties:
        Name: 'service'
    BlockPath: [1×1 Simulink.SimulationData.BlockPath]
       Values: [1×1 timeseries]
```

3 To access the logged data and time of each logged element, use the `Values.Data` and `Values.Time` properties. For example:

- Arrange logged data in tabular form by using the `table` function.

32-55

```
T1 = table(diningLog.Values.Time,diningLog.Values.Data);
T1.Properties.VariableNames = {'Time','Data'}

T1 =

  6×2 table

      Time        Data
    _____    ____

            0      0
    1.8607e+06     1
    1.9653e+06     0
    1.9653e+06     1
    1.9653e+06     0
    2.2912e+06     1

T2 = table(serviceLog.Values.Time,serviceLog.Values.Data);
T2.Properties.VariableNames = {'Time','Data'}

T2 =

  6×2 table

      Time        Data
    _____    ____
    1.7076e+06     0
    1.8607e+06     1
    1.9653e+06     2
    1.9653e+06     3
    1.9653e+06     4
    2.2912e+06     5
```

- View logged data in a figure window by using the `plot` function.

```
X = serviceLog.Values.Time;
Y = serviceLog.Values.Data;
plot(X,Y,'-o')
xlabel('Time')
ylabel('Data')
```
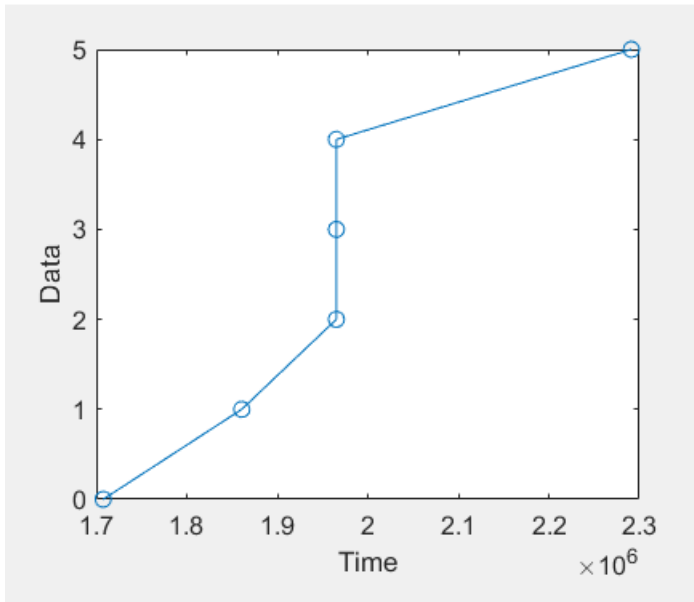
- Export logged data to an Excel® spreadsheet by passing an array of logged values to the `xlswrite` function:

```
A = [double(diningLog.Values.Time) double(diningLog.Values.Data)];
xlswrite('dining_log.xls',A);
```

Log Multidimensional Data

Stateflow logs each update to a multidimensional signal as a single change. For example, updating two elements of a matrix A separately

```
A[1][1] = 1;
A[1][2] = 1;
```

produces two different changes in the logged data. In contrast, updating a matrix A in a single command

```
A = 1;
```

produces a single change in the logged data, even though the command implies `A[i][j] = 1` for all values of `i` and `j`.

Limitations on Logging Data

When simulating models in external mode, logging of Stateflow data is not supported.

If you log state activity or data from a chart with **Fast Restart** enabled, any run after the first run duplicates the first logged data points. When you run algorithms that process these data points, you must account for this duplication.

See Also

Simulink.SimulationData.Dataset | get | plot | table | xlswrite

More About

- "Configure a Signal for Logging" (Simulink)
- "Logging and Accessibility Options" (Simulink)
- "Export Signal Data Using Signal Logging" (Simulink)

Log Data in Library Charts

In Simulink, you can create your own block libraries as a way to reuse the functionality of blocks or subsystems in one or more models. Similarly, you can reuse a set of Stateflow algorithms by encapsulating the functionality in a library chart.

As with other Simulink block libraries, you can specialize each instance of chart library blocks in your model to use different data types, sample times, and other properties. Library instances that inherit the same properties can reuse generated code.

For more information about Simulink block libraries, see "Libraries" (Simulink).

How Library Log Settings Influence Linked Instances

Chart instances inherit logging properties from the library chart to which they are linked. You can override logging properties in the instance, but only for signals you select in the library. You cannot select additional signals to log from the instance.

Override Logging Properties in Chart Instances

To override properties of logged signals in chart instances, use one of the following approaches.

Approach	How To Use
Simulink Signal Logging Selector dialog box	See "Override Logging Properties with the Logging Selector" on page 32-60
Command-line interface	See "Override Logging Properties with the Command-Line API" on page 32-61

Override Logging Properties in Atomic Subcharts

The model `sf_atomic_sensor_pair` simulates a redundant sensor pair as atomic subcharts `Sensor1` and `Sensor2` in the chart `RedundantSensors`. Each atomic subchart contains instances of the states `Fail`, `FailOnce`, and `OK` from the library chart `sf_atomic_sensor_lib`.
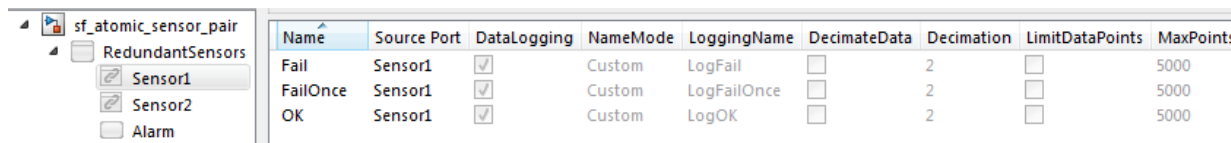
Override Logging Properties with the Logging Selector

1   Open the example library `sf_atomic_sensor_lib`.

2   Unlock the library by selecting **Diagram** > **Unlock Library**.

3   In the SingleSensor chart, select **Simulation** > **Output** > **Log Chart Signals**.

4   In Stateflow Signal Logging dialog box, set the following logging properties, then click **OK**.

For Signal:	What to Specify:
`Fail`	• Select the **Log signal data** check box. • Change **Logging name** to the custom name `LogFail`. • Click **Apply**.
`FailOnce`	• Select the **Log signal data** check box. • Change **Logging name** to the custom name `LogFailOnce`. • Click **Apply**.
`OK`	• Select the **Log signal data** check box. • Change **Logging name** to the custom name `LogOK`. • Click **Apply**.

5   Open the model `sf_atomic_sensor_pair`. This model contains two instances of the library chart.

6   Open the Model Configuration Parameters dialog box.

7   In the **Data Import/Export** pane, click **Configure Signals to Log** to open the Simulink Signal Logging Selector.

8   In the **Model Hierarchy** pane, expand `RedundantSensors`, and click `Sensor1` and `Sensor2`.

Each instance inherits logging properties from the library chart. For example:

9   Now, override some logging properties for `Sensor1`:

a In the **Model Hierarchy** pane, select `Sensor1`.

b Change **Logging Mode** to `Override signals`.

The selector clears all **DataLogging** check boxes for the model.

c Enable logging only for the `Fail` and `FailOnce` states in `Sensor1`:

Select **DataLogging** for these two signals. Leave **DataLogging** cleared for the `OK` signal.

d Append the text `Sensor1` to the logging names for `Fail` and `FailOnce`:

Double-click the logging names for signals `Fail` and `FailOnce`, and rename them `LogFailSensor1` and `LogFailOnceSensor1`, respectively.

The settings should look like this:

Logging Mode: Override signals									

Model Hierarchy		Contents of: Sensor1							Filter Contents
▲ sf_atomic_sensor_pair*		**Name**	Source Port	DataLogging	NameMode	LoggingName	DecimateData	Decimation	LimitDataPoints	MaxPoints
▲ RedundantSensors		Fail	Sensor1	✓	Custom	LogFailSensor1	☐	2	☐	5000
Sensor1		FailOnce	Sensor1	✓	Custom	LogFailOnceSensor1	☐	2	☐	5000
Sensor2		OK	Sensor1	☐	Custom	LogOK	☐	2	☐	5000
Alarm										
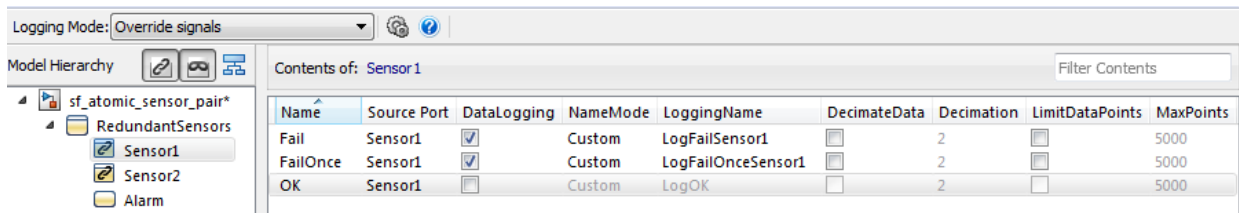
Override Logging Properties with the Command-Line API

1 Open the example library `sf_atomic_sensor_lib`.

2 Log the signals `Fail`, `FailOnce`, and `OK` in the `SingleSensor` chart using these commands:

```
% Get states in the SingleSensor chart
rt=sfroot;
states = rt.find('-isa', 'Stateflow.State');

% Enable logging for each state
for i = 1: length(states)
    states(i).LoggingInfo.DataLogging = 1;
end
```

3 Open the model `sf_atomic_sensor_pair`. This model contains two instances of the library chart.

4 Create a `ModelLoggingInfo` object for the model.

This object contains a vector `Signals` that stores all logged signals.

```
mi = Simulink.SimulationData.ModelLoggingInfo. ...
createFromModel('sf_atomic_sensor_pair')
```

The result is:

```
mi =

  Simulink.SimulationData.ModelLoggingInfo
  Package: Simulink.SimulationData

  Properties:
                    Model: 'sf_atomic_sensor_pair'
              LoggingMode: 'OverrideSignals'
    LogAsSpecifiedByModels: {}
                  Signals: [1x6 Simulink.SimulationData.SignalLoggingInfo]
```

The `Signals` vector contains the signals marked for logging in the library chart:

- Library instances of `Fail`, `FailOnce`, and `OK` states in atomic subchart `Sensor1`
- Library instances of `Fail`, `FailOnce`, and `OK` states in atomic subchart `Sensor2`

5 Make sure that `LoggingMode` equals `'OverrideSignals'`.

6 Create a block path to each logged signal whose properties you want to override.

To access signals inside Stateflow charts, use `Simulink.SimulationData.BlockPath(paths, subpath)`, where `subpath` represents a signal inside the chart.

To create block paths for the signals `Fail`, `FailOnce`, and `OK` in the atomic subchart `Sensor1` in the `RedundantSensors` chart:

```
failPath = Simulink.SimulationData. ...
BlockPath('sf_atomic_sensor_pair/RedundantSensors/Sensor1','Fail')

failOncePath = Simulink.SimulationData. ...
BlockPath('sf_atomic_sensor_pair/RedundantSensors/Sensor1','FailOnce')

OKPath = Simulink.SimulationData. ...
BlockPath('sf_atomic_sensor_pair/RedundantSensors/Sensor1','OK')
```

7 Get the index of each logged signal in the `Simulink.SimulationData.BlockPath` object.

To get the index for the signals `Fail`, `FailOnce`, and `OK`:

```
failidx = mi.findSignal(failPath);
failOnceidx = mi.findSignal(failOncePath);
OKidx = mi.findSignal(OKPath);
```

8 Override some logging properties for the signals in `Sensor1`:

    a   Disable logging for signal `OK`:

```
mi.Signals(OKidx).LoggingInfo.DataLogging = 0;
```

    b   Append the text `Sensor1` to the logging names for `Fail` and `FailOnce`:

```
% Enable custom naming
mi.Signals(failidx).LoggingInfo.NameMode = 1;
mi.Signals(failOnceidx).LoggingInfo.NameMode = 1;

% Enter the custom name
mi.Signals(failidx).LoggingInfo.LoggingName = 'LogFailSensor1';
mi.Signals(failOnceidx).LoggingInfo.LoggingName = 'LogFailOnceSensor1';
```

  9   Apply the changes:

```
set_param(bdroot, 'DataLoggingOverride', mi);
```

See Also

`Simulink.SimulationData.BlockPath` |
`Simulink.SimulationData.ModelLoggingInfo`

Commenting Stateflow Objects in a Chart

In this section...
"Comment Out a Stateflow Object" on page 32-64
"How Commenting Affects the Chart and Model" on page 32-64
"Add Text to a Commented Object" on page 32-66
"Limitations on Commenting Objects" on page 32-66

Comment Out a Stateflow Object

Commenting out a Stateflow object excludes it from simulation. To comment out a Stateflow object, right-click the object and select **Comment Out**. Use commenting to:

- Debug a chart by making minor changes between simulation runs.
- Test and verify the effects of objects on simulation results.
- Create incremental changes for rapid, iterative design.

How Commenting Affects the Chart and Model

A commented object is not visible to the rest of the chart and the model. Commented objects in a chart are excluded from:

- Simulation
- Logging
- Code generation
- Animation
- Debugging
- Active state output

When a chart is parsed, references to commented functions or states result in errors.

When you explicitly comment out a Stateflow object with **Comment Out**, the object appears gray with a badge . The software implicitly comments out some associated objects. Implicitly commented objects also appear gray, but do not have a badge. For example, if you explicitly comment out a state or junction, all incoming and outgoing

transitions are implicitly commented out. In this image of `sf_car`, the state `steady_state` is explicitly commented. The transitions in and out of `steady_state` are implicitly commented.
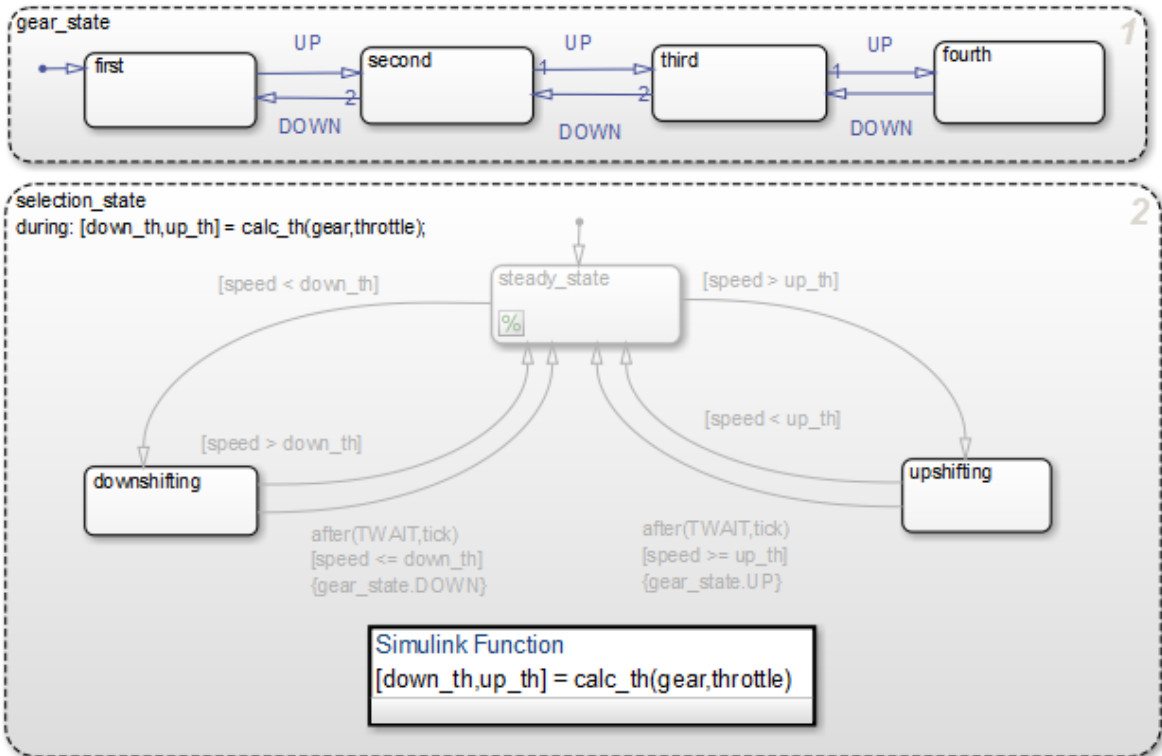
Explicitly Commented Stateflow Object	Implicit Results
States	All incoming and outgoing transitions, and child objects are implicitly commented out.
Transitions	None
Junctions	All connected transitions are implicitly commented out.
Functions	The software cannot invoke a commented function from any chart or model.

Explicitly Commented Stateflow Object	Implicit Results
Data	You cannot explicitly comment out data. If you comment out the parent object, then the software cannot reference the data.
Events	You cannot explicitly comment out events. If you comment out the parent object, then the software cannot reference the event.

To uncomment an object, right-click the commented object and select **Uncomment**. All implicitly commented objects are restored as well. Implicitly commented objects cannot be uncommented directly.

Add Text to a Commented Object

Add a note to the commented object by clicking the badge . Point to a badge to see the associated comments. You cannot add notes to implicitly commented objects.

Limitations on Commenting Objects

When you comment out an atomic subchart, the objects inside the chart do not appear implicitly commented. However, a commenting badge is displayed in the lower-left corner of the chart.

33

Explore and Modify Charts

Manage Data, Events, and Messages in the Symbols Window

In the Symbols window, you can view and manage data, events, and messages while working in the Stateflow editor. To open the Symbols window, select **View > Symbols**. From the Symbols window you can:

- Add and delete data, events, and messages.
- Set the object type and scope.
- Change the port number.
- Edit the name of an object and update all instances of the object name in the chart.
- Undo and redo changes in type, name, and port number.
- Detect unused objects.
- Detect and fix unresolved objects.
- Trace between objects in the window and where the objects are used in the chart.
- View and edit object properties in the Property Inspector.

The rows in the Symbols window display object hierarchy. Expand an object in the window to see data, events, and messages parented by that object. By default, all the nongraphical objects in a chart are listed in the window. To view only the objects that are used at the current level of hierarchy and below, select the ⊞ icon. To search for specific symbols, type in the Filter search box ▼.

Add and Modify Data, Events, and Messages

To add a nongraphical object to a Stateflow block, in the Symbols window:

1   Select one of these icons.

Object	Icon
Data	
Event	
Message	

2   In the row for the new object, under **TYPE**, choose the object type.

3   Edit the name of the object.

4   For input and output objects, under **PORT**, choose a port number.

5   To view the object in the Property Inspector, right-click the object and select `Inspect`.

6   In the Property Inspector, modify the object properties.

After you add objects through the Symbols window, the objects appear as unused until you reference them in your Stateflow design.

In the Symbols window, you can modify the name, type, and port number of Stateflow objects. Edit the name of objects in the **NAME** field. When you rename an object, select **Shift+Enter** to rename all instances of the object throughout the state machine. To change the type or port number of an object, click the corresponding field and select from the available options. To delete an object from the state machine, right-click the object and select `Delete`. To either undo or redo these changes, use the **Edit** menu.

Detect Unused Data in the Symbols Window

The Symbols window indicates unused data, messages, functions, and events with a yellow warning icon. To delete unused objects, right-click the object in the Symbols window and select **Delete**. By removing objects that have no effect on simulation, you can reduce the size of your model. In this chart, after you add `data`, it first appears as unused. After you reference `data` in the chart, the warning sign disappears.
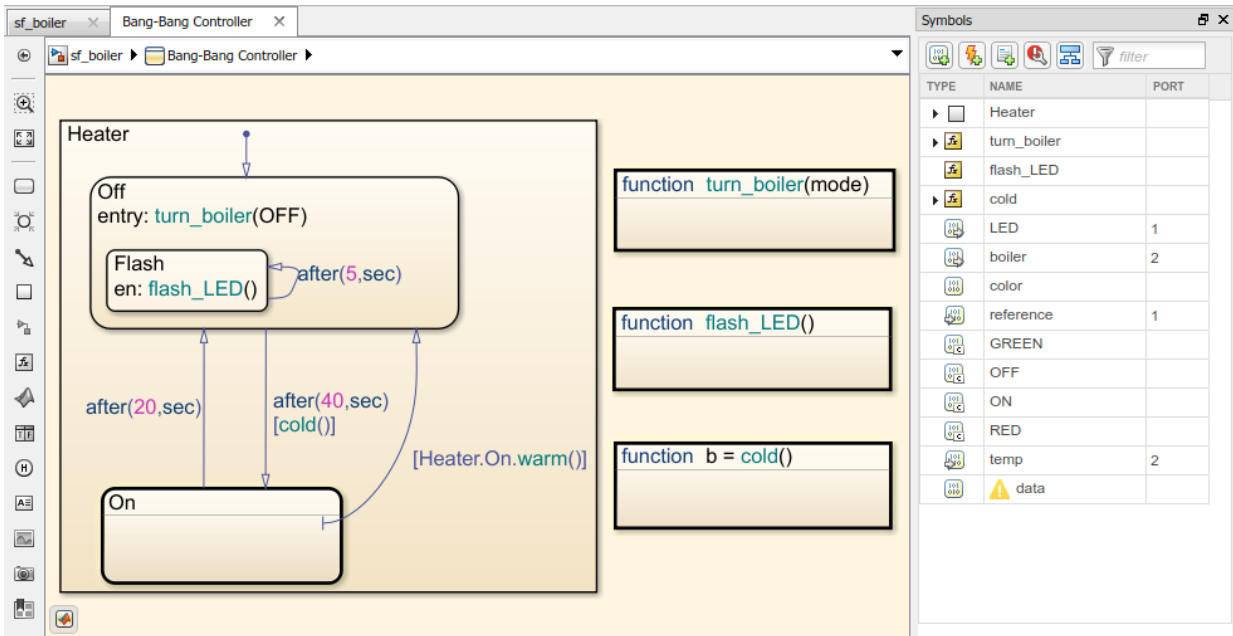
The following types of unused data are not detected:

- Machine-parented data
- Inputs and outputs of MATLAB functions
- Data of parameter scope in a chart that contains atomic subcharts

Trace Data, Events, and Messages

Stateflow provides traceability between the state machine and nongraphical symbols. When you select a symbol in the Symbols window, Stateflow highlights sections of the chart that access that symbol. When you select an object in your chart, Stateflow highlights the symbols that the object uses.

To control when the objects and symbols are highlighted, select the preference button

. A drop-down menu appears.

For Stateflow to highlight symbols in the Symbols window that an object uses, select **Highlight used symbols**. For Stateflow to highlight objects in the chart that use an symbol, select **Highlight uses on diagram**. With **Highlight uses on diagram** you can choose to highlight:

- All uses of the symbol in your chart.
- Objects from where the symbol is read.
- Objects to where the symbol is written.

For example, open the model `sf_tetris2` and double-click the chart `TetrisLogic`. In the Symbols window, when you select constant `ARENA_HEIGHT`, the states and functions that use `ARENA_HEIGHT` are highlighted. If the chart does not use an object, the symbol appears in the window with a yellow warning icon ⚠.

To see the uses of the constant `ARENA_HEIGHT`, open the function `freeze`.
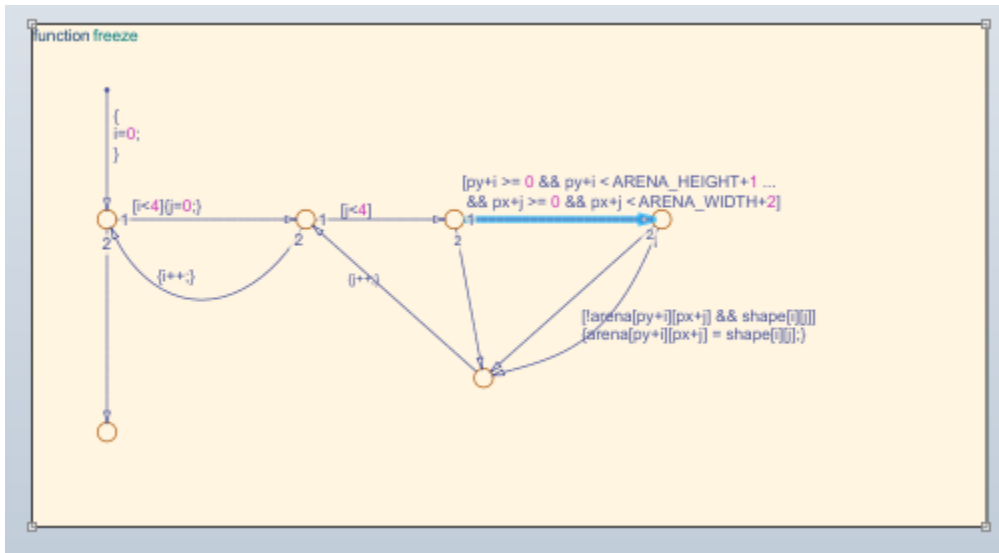
You can also select a graphical object such as a state, transition, or function in the chart and view the symbols that the object uses. For example, in the chart `TetrisLogic`, expand the symbol `MainArea` in the Symbols window. If you select the state `FreezeShape` in the chart, then the local data `shape` and the function `freeze()` are highlighted in the Symbols window. This highlighting indicates that those objects are used inside the state `FreezeShape`.
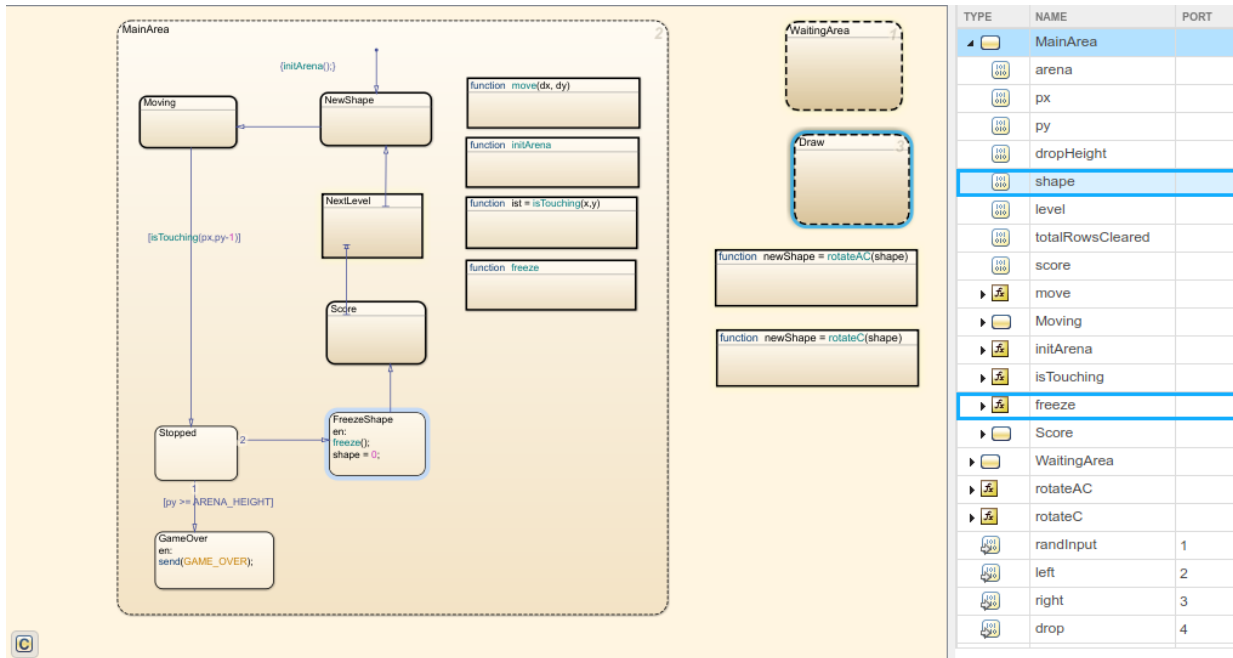
When in debugging mode, the values of each data are displayed in the **VALUE** column of the Symbols window. Stateflow updates the values periodically when the simulation is running. The value column highlights changes to data values as the changes occur. When the debugger is stopped at a breakpoint, you can update the initial value or change the value of a symbols in either the command prompt or the Symbols window.

Data or Message	Update Initial Value	Update During Debugging
Input	No	No
Output	Yes	Yes
Parameter	No	No
Constant	Yes	No
Data Store Memory	No	Yes
Local	Yes	Yes

For bus elements, you can change the value of a symbols in either the command prompt or the Symbols window.

Bus Element	Update Initial Value	Update During Debugging
Input	No	No
Output	No	Yes
Parameter	No	No
Constant	No	No
Data Store Memory	No	Yes
Local	No	Yes

In the Symbols window multidimensional arrays appear as the data type and size of the array. If the array does not exceed more than 100 elements, hover over the symbol to view the elements. For arrays that contain more than 100 elements, view the elements by using the command prompt.

When simulation is paused, hover over messages in the canvas to view payloads in the queue. This is similar to the hover functionality on the canvas. For other non-scalar objects, the size and data type appear. To see these values, use the Watch window. See "Watch Stateflow Data Values" on page 32-36 and "Manage Stateflow Breakpoints and Watch Data" on page 32-17.
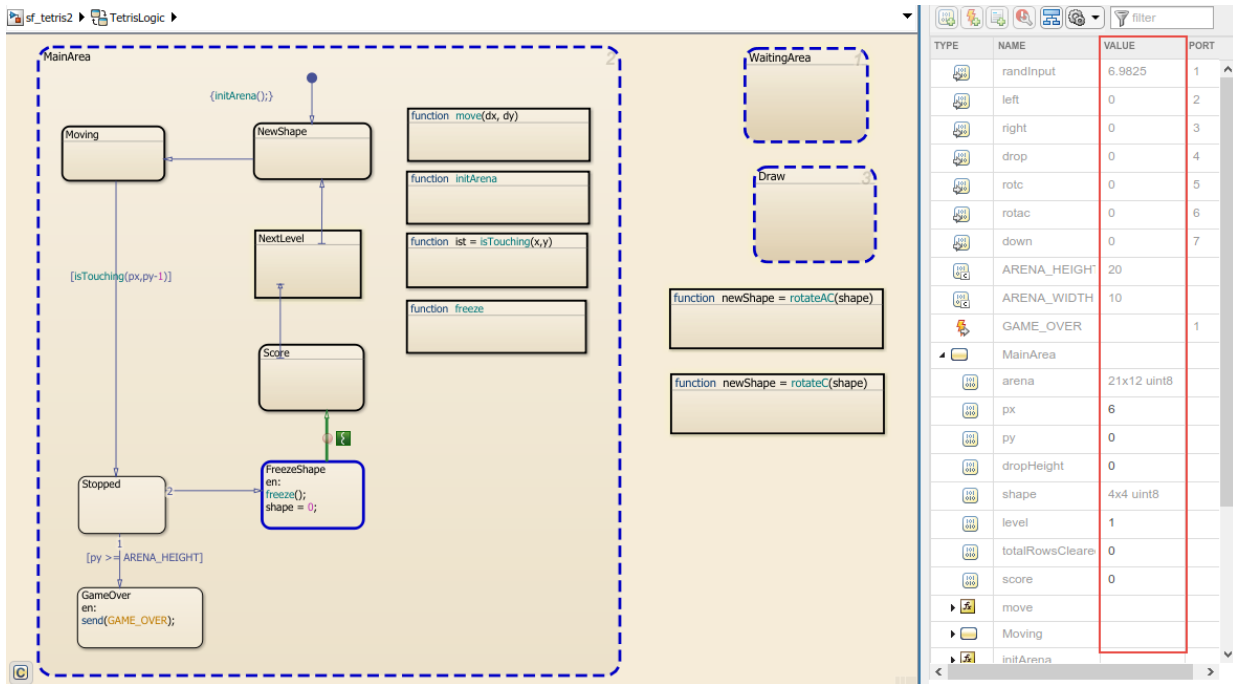
Symbols Window Limitations

You cannot add the types of objects listed in the table via the Symbols window. To add these types of objects, use the Model Explorer. As a best practice, avoid using machine-parented data.

Object	Via Symbols Window	Visible in Symbols Window
Data, events, and messages parented by a state	No	Yes
Data, events, and messages inside a function	No	Yes
Data and events parented by the state machine	No	No

Additional limitations:

- When you modify the code in a MATLAB function, the changes are not updated in the Symbols window until after you save the MATLAB function.
- You cannot undo or redo changes to input and output for MATLAB functions.
- You cannot recover deleted data, events, or messages from a state transition table.
- You cannot undo scope changes to data parented by graphical functions, MATLAB functions, and truth tables.
- You cannot undo renaming an object for truth tables.
- When you delete data for objects contained in a Simulink based state, the object remains in your Simulink based state and the data symbol is shown as undefined in the Symbols window.

See Also

More About

- "Resolve Undefined Symbols in Your Chart" on page 30-35
- "Add Stateflow Data" on page 9-2
- "Set Data Properties" on page 9-5

Use the Model Explorer with Stateflow Objects

In this section...
"View Stateflow Objects in the Model Explorer" on page 33-13
"Edit Chart Objects in the Model Explorer" on page 33-15
"Add Data and Events in the Model Explorer" on page 33-15
"Rename Objects in the Model Explorer" on page 33-15
"Set Properties for Chart Objects in the Model Explorer" on page 33-15
"Move and Copy Data and Events in the Model Explorer" on page 33-16
"Change the Port Order of Input and Output Data and Events" on page 33-17
"Delete Data and Events in the Model Explorer" on page 33-18

View Stateflow Objects in the Model Explorer

You can use one of these methods for opening the Model Explorer:

- In the Stateflow Editor, select **View > Model Explorer**.
- Right-click an empty area in the chart and select **Explore**.

The Model Explorer appears something like this:
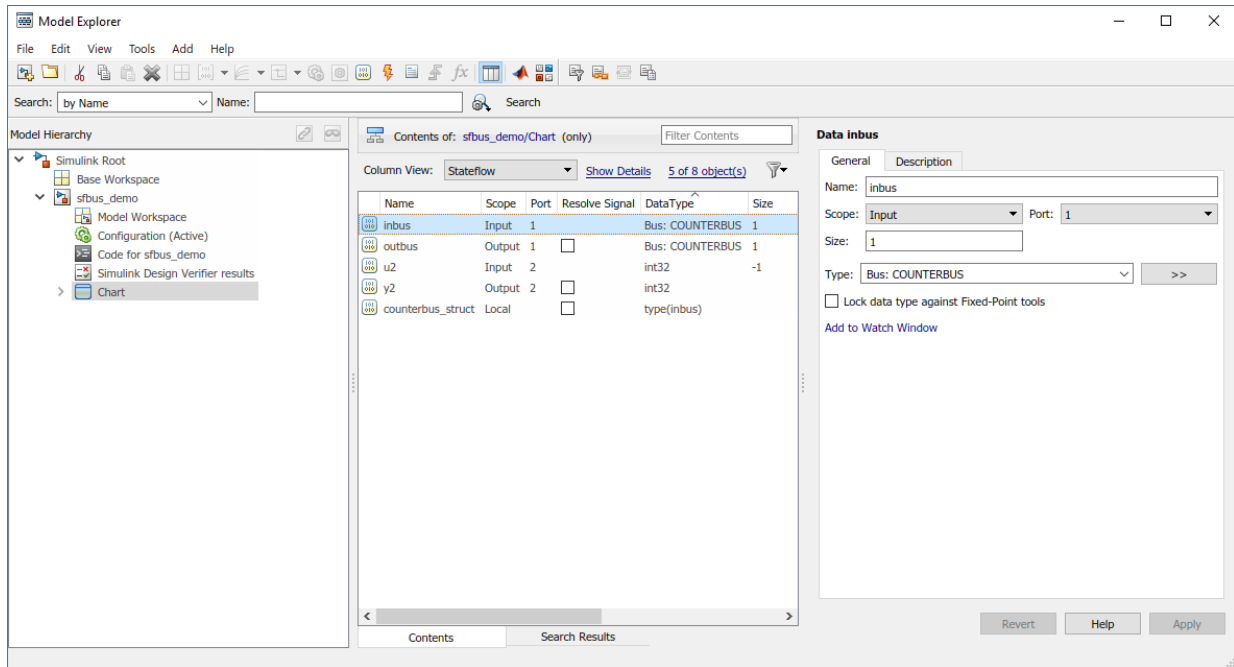
The main window has two panes: a **Model Hierarchy** pane on the left and a **Contents** pane on the right. When you open the Model Explorer, the Stateflow object you are editing appears highlighted in the **Model Hierarchy** pane and its objects appear in the **Contents** pane. This example shows how the Model Explorer appears when opened from the chart.

The **Model Hierarchy** pane displays the elements of all loaded Simulink models, which includes Stateflow charts. A preceding plus (+) character for an object indicates that you can expand the display of its child objects by double-clicking the entry or by clicking the plus (+). A preceding minus (-) character for an object indicates that it has no child objects.

Clicking an entry in the **Model Hierarchy** pane selects that entry and displays its child objects in the **Contents** pane. A hypertext link to the currently selected object in the **Model Hierarchy** pane appears after the **Contents of:** label at the top of the **Contents** pane. Click this link to display that object in its native editor. In the preceding example, clicking the link sfbus_demo/Chart displays the contents of the chart in its editor.

Each type of object, whether in the **Model Hierarchy** or **Contents** pane, appears with an adjacent icon. Subcharted objects (states, boxes, or graphical functions) appear altered with shading.

The display of child objects in the **Contents** pane includes properties for each object, most of which are directly editable. You can also access the properties dialog box for an object from the Model Explorer. See "Set Properties for Chart Objects in the Model Explorer" on page 33-15 for more details.

Edit Chart Objects in the Model Explorer

To edit a chart object that appears in the **Model Hierarchy** pane of the Model Explorer:

1 Right-click the object.

2 Select **Open** from the context menu.

   The selected object appears highlighted in the chart.

Add Data and Events in the Model Explorer

To add data or events using the Model Explorer, see the following links:

- "Add Data Through the Model Explorer" on page 9-3
- "Add Events Through the Model Explorer" on page 10-4

Rename Objects in the Model Explorer

To rename a chart object in the Model Explorer:

1 Right-click the object row in the **Contents** pane of the Model Explorer and select **Rename**.

   The name of the selected object appears in a text edit box.

2 Change the name of the object and click outside the edit box.

Set Properties for Chart Objects in the Model Explorer

To change the property of an object in the **Contents** pane of the Model Explorer:

1   In the **Contents** pane, click in the row of the displayed object.

2   Click an individual entry for a property column in the highlighted row.

- For text properties, such as the Name property, a text editing field with the current text value overlays the displayed value. Edit the field and press the **Enter** key or click anywhere outside the edit field to apply the changes.

- For properties with enumerated entries, such as the Scope, Trigger, or Type properties, select from a drop-down combo box that overlays the displayed value.

- For Boolean properties (properties that are set on or off), select or clear the box that appears in place of the displayed value.

To set all the properties for an object displayed in the **Model Hierarchy** or **Contents** pane of the Model Explorer:

1   Right-click the object and select **Properties**.

      The properties dialog box for the object appears.

2   Edit the appropriate properties and click **Apply** or **OK**.

To display the properties dialog box dynamically for the selected object in the **Model Hierarchy** or **Contents** pane of the Model Explorer:

1   Select **View > Show Dialog Pane**.

      The properties dialog box for the selected object appears in the far right pane of the Model Explorer.

Move and Copy Data and Events in the Model Explorer

Note If you move an object to a level in the hierarchy that does not support the **Scope** property for that object, the **Scope** automatically changes to **Local**.

To move data and event objects to another parent:

1   Select the data or event to move in the **Contents** pane of the Model Explorer.

      You can select a contiguous block of items by highlighting the first (or last) item in the block and then using **Shift** + click for highlighting the last (or first) item.

2 Click and drag the highlighted objects from the **Contents** pane to a new location in the **Model Hierarchy** pane to change its parent.

A shadow copy of the selected objects accompanies the mouse cursor during dragging. If no parent is chosen or the parent chosen is the current parent, the mouse cursor changes to an X enclosed in a circle, indicating an invalid choice.

To cut or copy the selected data or event:

1 Select the event or data to cut or copy in the **Contents** pane of the Model Explorer.

2 In the Model Explorer, select **Edit > Cut** or **Edit > Copy**.

If you select **Cut**, the selected items are deleted and then copied to the clipboard for copying elsewhere. If you select **Copy**, the selected items are left unchanged.

You can also right-click a single selection and select **Cut** or **Copy** from the context menu. The Model Explorer also uses the keyboard equivalents of **Ctrl+X** (Cut) and **Ctrl+C** (Copy) on a computer running the UNIX or Windows operating system.

3 Select a new parent object in the **Model Hierarchy** pane of the Model Explorer.

4 Select **Edit > Paste**. The cut items appear in the **Contents** pane of the Model Explorer.

You can also paste the cut items by right-clicking an empty part of the **Contents** pane and selecting **Paste** from the context menu. The Model Explorer also uses the keyboard equivalent of **Ctrl+V** (Paste) on a computer running the UNIX or Windows operating system.

Change the Port Order of Input and Output Data and Events

Input data, output data, input events, and output events each have numerical sequences of port index numbers. You can change the order of indexing for event or data objects with a scope of **Input to Simulink** or **Output to Simulink** in the **Contents** pane of the Model Explorer as follows:

1 Select one of the input or output data or event objects.

2 Click the **Port** property for the object.

3 Enter a new value for the Port property for the object.

The remaining objects in the affected sequence are automatically assigned a new value for their **Port** property.

Delete Data and Events in the Model Explorer

Delete data and event objects in the **Contents** pane of the Model Explorer as follows:

1 Select the object.

2 Press the **Delete** key.

   You can also select **Edit > Cut** or **Ctrl+X** from the keyboard to delete an object.

Use the Search and Replace Tool

Open the Search & Replace Tool

To open the Search & Replace dialog box:

1 Open a chart.

2 Select **Edit** > **Find & Replace in Chart**.
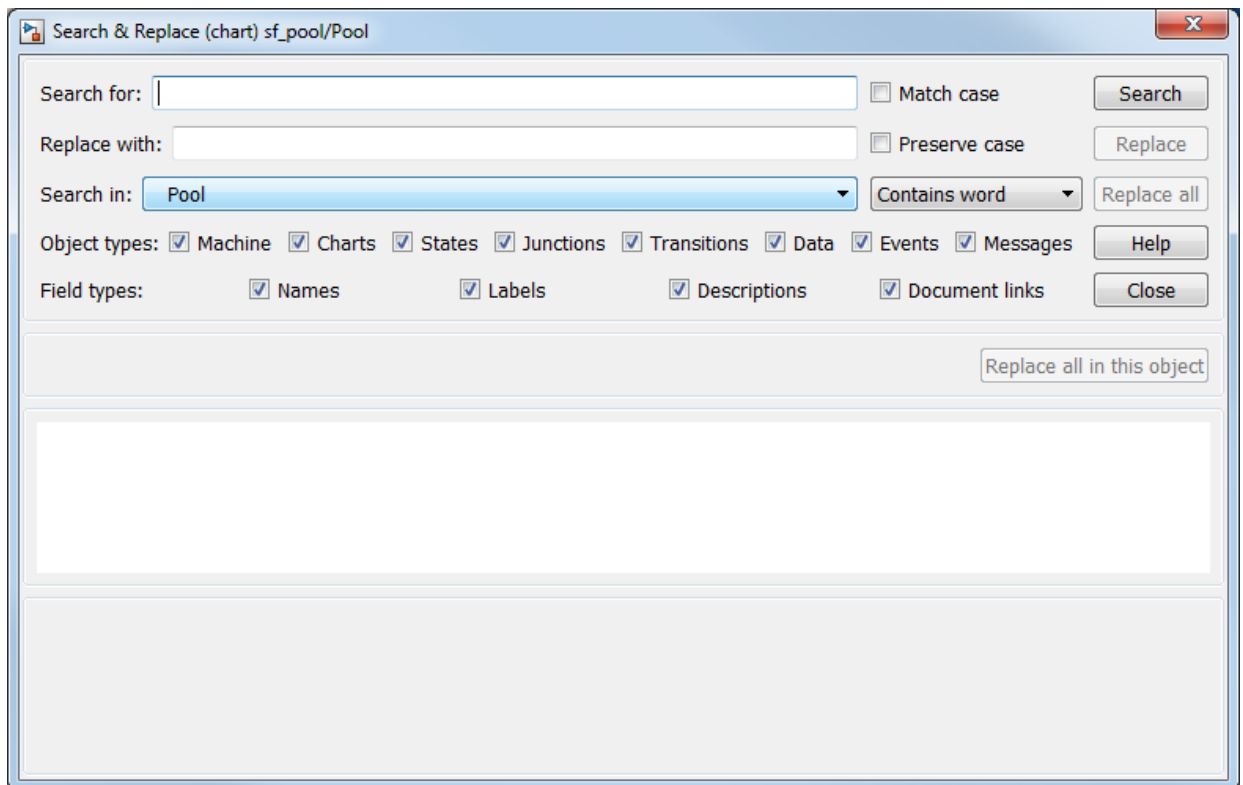
The Search & Replace dialog box contains the following fields:

- **Search for**

Enter search pattern text in the **Search for** text box. You can select the interpretation of the search pattern with the **Match case** check box and the **Match options** field (unlabeled and just to the right of the **Search in** field).

- **Match case**

  If you select this check box, the search is case sensitive and the Search & Replace tool finds only text matching the search pattern exactly.

- **Replace with**

  Specify the text to replace the text found when you select any of the **Replace** buttons (**Replace**, **Replace all**, **Replace all in this object**). See "Use Replace Buttons" on page 33-28.

- **Preserve case**

  This option modifies replacement text. For an understanding of this option, see "Replacing with Case Preservation" on page 33-27.

- **Search in**

  By default, the Search & Replace tool searches for and replaces text only within the current Stateflow chart that you are editing in the Stateflow Editor. You can select to search the machine owning the current Stateflow chart or any other loaded machine or chart by accessing this selection box.

- **Match options**

  This field is unlabeled and just to the right of the **Search in** field. You can modify the meaning of your search text by entering one of the selectable search options. See "Refine Searches" on page 33-21.

- **Object types** and **Field types**

  Under the **Search in** field are the selection boxes for **Object types** and **Field types**. These selections further refine your search and are described below.

- **Search** and **Replace** buttons

  These are described in "Use the Search Button and View Area" on page 33-24 and "Use Replace Buttons" on page 33-28.

- **View Area**

  The bottom half of the Search & Replace dialog box displays the result of a search. This area is described in "A Breakdown of the View Area" on page 33-26.

Refine Searches

Enter search pattern text in the **Search for** text box. You can use one of the following settings to further refine the meaning of the text entered.

Match case

By selecting the **Match case** option, you enable case-sensitive searching. In this case, the Search & Replace tool finds only text matching the search pattern exactly.

By clearing the **Match case** option, you enable case-insensitive searching. In this case, search pattern characters entered in lower- or uppercase find matching text with the same sequence of base characters in lower- or uppercase. For example, the search entry`"AnDrEw"` finds the matching text `"andrew"` or `"Andrew"` or `"ANDREW"`.

Preserve case

This option modifies replacement text and not search text. For details, see "Replacing with Case Preservation" on page 33-27.

Contains word

Select this option to specify that the search pattern text is a whole word expression used in a Stateflow chart with no specific beginning and end delimiters. In other words, find the specified text in any setting.

Suppose that you have a state with this label and entry action:

```
throt_fail
entry: fail_state[THROT] = 1;
```

Searching for the text `fail` with the **Contains word** option finds two occurrences of `fail`.

Match whole word

Select this option to specify that the search pattern in the **Search for** field is a whole word expression used in a Stateflow chart with beginning and end delimiters consisting of a blank space or a character that is not alphanumeric and not an underscore character (_).

In the previous example of a state named `throt_fail`, if **Match whole word** is selected, searching for `fail` finds no text within that state. However, searching for `"fail_state"`

does find the text `"fail_state"` as part of the second line since it is delimited by a space at the beginning and a left square bracket (`[`) at the end.

Regular expression

Set the **Match options** field to **Regular expression** to search for text that varies from character to character within defined limits.

A regular expression is text composed of letters, numbers, and special symbols that defines one or more candidates. Some characters have special meaning when used in a regular expression, while other characters are interpreted as themselves. Any other character appearing in a regular expression is ordinary, unless a back slash (`\`) character precedes it.

If the **Match options** field is set to **Regular expression** in the previous example of a state named `throt_fail`, searching for `"fail_"` matches the `"fail_"` text that is part of the second line, character for character. Searching with the regular expression `"\w*_"` also finds the text `"fail_"`. This search uses the regular expression shorthand `"\w"` that represents any part-of-word character, an asterisk (*) that represents any number of any characters, and an underscore (_) that represents itself.

For a list of regular expression meta characters, see "Regular Expressions" (MATLAB).

Specify the Search Scope

You specify the scope of your search by selecting from the field regions discussed in the topics that follow.

Search in

You can select a whole machine or individual chart for searching in the **Search in** field. By default, the current chart in which you opened the Search & Replace tool is selected.

To select a machine, follow these steps:

1   Select the down arrow of the **Search in** field.

    A list of the currently loaded machines appears with the current machine expanded to reveal its Stateflow charts.

2   Select a machine.

To select a Stateflow chart for searching, follow these steps:

1 Select the down arrow of the **Search in** field again.

This list contains the previously selected machine expanded to reveal its Stateflow charts.

2 Select a chart from the expanded machine.

Object Types

Limit your search by deselecting one or more object types.

Note You cannot search in state transition tables with this tool.

Field Types

Limit your search by deselecting one or more field types.

Available field types are as follows.

Names

Machines, charts, data, and events have valid **Name** fields. States have a **Name** defined as the top line of their labels. You can search and replace text belonging to the **Name** field of a state in this sense. However, if the Search & Replace tool finds matching text in a state's **Name** field, the rest of the label is subject to later searches for the specified text whether or not the label is chosen as a search target.

Note The **Name** field of machines and charts is an invalid target for the Search & Replace tool. Use the Simulink model window to change the names of machines and charts.

Labels

Only states and transitions have labels.

Descriptions

All objects have searchable **Description** fields.

Document links

All objects have searchable **Link** fields.

33-23

Use the Search Button and View Area

This topic contains the following subtopics:

Click **Search** to initiate a single-search operation. If an object match is made, its text fields appear in the **Viewer** pane in the middle of the Search & Replace dialog box. If the object is graphical (state, transition, junction, chart), the matching object appears highlighted in a **Portal** pane below the **Viewer** pane.

A Breakdown of the View Area

The view area of the Search & Replace dialog box displays matching text and its containing object, if viewable. In the previous example, taken from the `sf_pool` model, a search for the word `"friction"` finds the **Description** field for the state `TotalDynamics`. The resulting view area consists of these parts:

Icon

Displays an icon appropriate to the object containing the matching text. These icons are identical to the icons in the Model Explorer that represent Stateflow objects displayed in "View Stateflow Objects in the Model Explorer" on page 33-13.

Full Path Name of Containing Object

This area displays the full path name for the object that contains the matching text:

```
(<type>) <machine name>/<subsystem>/<chart
name>.[p₁]...[pₙ].<object name> (<id>)
```

where p_1 through p_n denote the object's parent states.

Viewer

This area displays the matching text as a highlighted part of all search-qualified text fields for the owner object. If other occurrences exist in these fields, they too are highlighted, but in lighter shades.

To invoke the properties dialog box for the owner object, double-click anywhere in the Viewer pane.

Portal

This area contains a graphic display of the object that contains the matching text. That object appears highlighted.

To display the highlighted object in the Stateflow Editor, double-click anywhere in the Portal pane.

The Search Order

If you specify an entire machine as your search scope in the **Search in** field, the Search & Replace tool starts searching at the beginning of the first chart of the model, regardless of the Stateflow chart that appears in the Stateflow Editor when you begin your search.

After searching the first chart, the Search & Replace tool continues searching each chart in model order until all charts for the model have been searched.

If you specify a Stateflow chart as your search scope, the Search & Replace tool begins searching at the beginning of the chart. The Search & Replace tool continues searching the chart until all the chart objects have been searched.

The search order when searching an individual chart for matching text is equivalent to a depth-first search of the Model Explorer. Starting at the highest level of the chart, the Model Explorer hierarchy is traversed downward from parent to child until an object with no child is encountered. At this point, the hierarchy is traversed upward through objects already searched until an unsearched sibling is found and the process repeats.

Specify the Replacement Text

The Search & Replace tool replaces matching text with the exact (case-sensitive) text you entered in the **Replace With** field unless you selected the **Preserve case** option.

Replacing with Case Preservation

If you choose the **Preserve case** option, matching text is replaced based on one of these conditions:

*   Whisper

    Matching text has only lowercase characters. Matching text is replaced entirely with the lowercase equivalent of all replacement characters. For example, if the replacement text is `"ANDREW"`, the matching text `"bill"` is replaced by `"andrew"`.

*   Shout

    Matching text has only uppercase characters. Matching text is replaced entirely with the uppercase equivalent of all replacement characters. For example, if the replacement text is `"Andrew"`, the matching text `"BILL"` is replaced by `"ANDREW"`.

*   Proper

    Matching text has uppercase characters in the first character position of each word. Matching text is replaced entirely with the case equivalent of all replacement characters. For example, if the replacement text is `"andrew johnson"`, the matching text `"Bill Monroe"` is replaced by `"Andrew Johnson"`.

*   Sentence

33-27

Matching text has an uppercase character in the first character position of a sentence with all other sentence characters in lowercase. Matching text is replaced in like manner, with the first character of the sentence given an uppercase equivalent and all other sentence characters set to lowercase. For example, if the replacement text is `"andrew is tall."`, the matching text `"Bill is tall."` is replaced by `"Andrew is tall."`.

If the matching text does not follow any of these patterns, then the text and case replacement match the user input.

Use Replace Buttons

You can activate the replace buttons (**Replace**, **Replace all**, **Replace all in this object**) only after a search that finds text.

Replace

When you select the **Replace** button, the current instance of text matching the text in the **Search for** field is replaced by the text you entered in the **Replace with** field. The Search & Replace tool then searches for the next occurrence of the **Search for** text.

Replace all

When you select the **Replace all** button, all instances of text matching the **Search for** field are replaced by the text entered in the **Replace with** field. Replacement starts at the point of invocation to the end of the current Stateflow chart. If you initially skip through some search matches with the **Search** button, these matches are also skipped when you select the **Replace all** button.

Replace all in this object

When you select the **Replace all in this object** button, all instances of text matching the **Search for** field are replaced by text you entered in the **Replace with** field everywhere in the current Stateflow object regardless of previous searches.

Search and Replace Messages

Informational and warning messages appear in the **Full Path Name Containing Object** field along with a defining icon.

 – Informational Messages

 – Warnings

The following messages are informational:

Please specify a search string

A search was attempted without search text specified.

No Matches Found

No matches exist in the selected search scope.

Search Completed

No more matches exist in the selected search scope.

The following warnings refer to invalid conditions for searching or replacing:

Invalid option set

The object types and field types that you selected are incompatible.

Match object not currently editable

The matching object is not editable by replacement due to one of these problems.

Problem	Solution
A simulation is running.	Stop the simulation.
You are editing a locked library block.	Unlock the library.
The current object or its parent has been manually locked.	Unlock the object or its parent.

The following warnings appear if the Search & Replace tool must find the object again and its matching text field. If the original matching object is deleted or changed before an ensuing search or replacement, the Search & Replace tool cannot continue.

Search object not found

If you search for text, find it, and then delete the containing object, this warning appears if you continue to search.

Match object not found

If you search for text, find it, and then delete the containing object, this warning appears if you perform a replacement.

Match not found

If you search for text, find it, and then change the object containing the text, this warning appears if you perform a replacement.

Search string changed

If you search for text, find it, and then change the **Search for** field, this warning appears if you perform a replacement.

Standalone Stateflow Charts for Execution in MATLAB

Create Stateflow Charts for Execution as MATLAB Objects

To combine the advantages of state machine programming with the full functionality of MATLAB, create a standalone Stateflow chart outside of a Simulink model. Save the standalone chart with the extension `.sfx` and execute it as a MATLAB object. Refine your design by using chart animation and graphical debugging tools.

With standalone charts, you can create MATLAB applications such as:

- MATLAB App Designer user interfaces that use mode logic to manage the behavior of widgets. See "Design Human-Machine Interface Logic by Using Stateflow Charts" on page 34-28.
- Communication protocols and data stream processing applications that use sequential logic. See "Model a Communications Protocol by Using Chart Objects" on page 34-34.
- Data Acquisition Toolbox™ or Instrument Control Toolbox™ applications that use timer-based logic to monitor and control external tasks. See "Implement a Financial Strategy by Using Stateflow" on page 34-38.

These applications can be shared and executed without requiring a Stateflow license. For more information, see "Share Standalone Charts" on page 34-4.

Construct a Standalone Chart

To construct a standalone Stateflow chart, open the Stateflow editor by using the `edit` function. For example, at the MATLAB Command Window, enter:

```
edit chart.sfx
```

If the file `chart.sfx` does not exist, the Stateflow editor opens an empty chart with the name `chart`. Otherwise, the editor opens the chart defined by the `sfx` file.

After you save a standalone chart, the `help` function displays information about executing it in MATLAB:

```
help chart.sfx
```

Create a Stateflow Chart Object

To execute a standalone chart in MATLAB, first create a Stateflow chart object. Use the name of the `sfx` file for the standalone chart as a function. Specify the initial values of data as name-value pairs. For example, this command creates the chart object `chartObject`, initializes `data1` and `data2`, and executes its default transition:

```
chartObject = chart('data1',value1,'data2',value2)
```

To display chart information, such as the syntax for execution, the values of the chart data, and the list of active states, use the `disp` function:

```
disp(chartObject)
```

Execute a Standalone Chart

After you define a Stateflow chart object, you can execute the standalone chart by calling the `step` function (with data values, if necessary):

```
step(chartObject,'data1',value1,'data2',value2)
```

Alternatively, you can call one of the input event functions:

```
event_name(chartObject,'data1',value1,'data2',value2)
```

If your chart has graphical or MATLAB functions, you can call them directly in the MATLAB Command Window. Calling a chart function does not execute the standalone chart.

```
function_name(chartObject,u1,u2)
```

You can execute a standalone chart without opening the Stateflow Editor. If the chart is open, then the Stateflow Editor highlights active states and transitions through chart animation.

For the purposes of debugging and unit testing, you can execute a standalone chart directly from the Stateflow editor. During execution, you enter data values and broadcast events from the user interface. For more information, see "Execute and Unit Test Stateflow Chart Objects" on page 34-9.

You can execute a standalone chart from a MATLAB script, a Simulink model, or an App Designer user interface. For more information, see:

Share Standalone Charts

You can share standalone charts with collaborators who do not have a Stateflow license.

If your collaborators have Stateflow, they can open, edit, and execute your standalone charts. During execution, chart animation and debugging is supported. Run-time error messages contain hyperlinks to the state or transition in the chart where the error occurs.

If your collaborators do not have Stateflow, they can execute your standalone charts as MATLAB objects without opening the Stateflow editor. Chart animation and debugging is not supported. Run-time error messages do not contain hyperlinks to the chart.

Properties and Functions of Stateflow Chart Objects

A Stateflow chart object encapsulates data and operations in a single structure by providing:

• Private properties that contain the internal state variables for the standalone chart.

• A `step` function that calls the various operations implementing the chart semantics.

A chart object can have other properties and functions that correspond to the various elements present in the chart.

Standalone Chart Elements	Chart Object Elements
Local and constant data	Public properties
Input events	Functions that execute the chart
Graphical and MATLAB functions	Functions that you can call from the MATLAB Command Window

Chart Object Configuration Options

When you create a chart object, you can specify chart behavior by including these configuration options as name-value pairs.

Configuration Option	Description	Example
`-animationDelay`	Specify the delay that the chart animation uses to highlight each transition segment. The default value is `0.01` seconds. To produce chart animation with no delays, set to zero.	Create a chart object that has very slow animation by specifying one second delays: `chartObject = chart('-animationDelay',1)`
`-enableAnimation`	Enable chart animation. The default value is `true`.	Create a chart object that has animation disabled: `chartObject = chart('-enableAnimation',f`
`-executeInitStep`	Enable the initial execution of default transitions. The default value is `true`.	Create a chart object but do not execute the default transition: `chartObject = chart('-executeInitStep',f`
`-warningOnUninitializedData`	Enable the warning about empty chart data after initializing the chart object. The default value is `true`.	Eliminate the warning when creating a chart object: `chartObject = chart('-warningOnUninitial`

Initialization of Chart Data

In the Stateflow Editor, you can use the Symbols window to specify initial values for chart data. When you create a chart object, chart data is initialized in alphabetical order according to its scope. Constant data is initialized first. Local data is initialized last.

If you use an expression to specify an initial value, then the chart attempts to resolve the expression by:

- Using the values of other data in the chart.
- Calling functions on the search path.

For example, suppose that you specify an initial value for the local data x by using the expression y. Then:

- If the chart has a constant called y, y is initialized before x. The local data x is assigned the same initial value as y.

- If the chart has a local data called y, x is initialized before y. The local data x is assigned to an empty array. If the configuration option -warningOnUninitializedData is set to true, a warning occurs.

- If the chart has no data named y, x is initialized by calling the function y. If the file y.m is not on the search path, this error occurs:

```
Undefined function or variable 'y'.
```

Stateflow does not search the MATLAB workspace to resolve initial values, so this error occurs even if there is a variable called y in the MATLAB workspace.

Capabilities and Limitations

Supported features:

- Classic chart semantics with MATLAB as the action language. You can use the full functionality of MATLAB, including those functions that are restricted for code generation in Simulink. See "Execute Stateflow Chart Objects Through Scripts and Models" on page 34-21.

> **Note** In standalone Stateflow charts, the operating system command symbol ! is not supported. To execute operating system commands, use the function system.

- Exclusive (OR) and Parallel (AND) state decomposition with hierarchy. See "State Decomposition" on page 2-16 and "State Hierarchy" on page 2-14.

- Flow charts, graphical functions, and MATLAB functions. See "Reusable Components in Charts".

- Chart local and constant data without restriction to type. See "Execute and Unit Test Stateflow Chart Objects" on page 34-9.

- Input events. See "Design Human-Machine Interface Logic by Using Stateflow Charts" on page 34-28.

- Temporal logic operators:

  - after, every, and temporalCount operate on the number of invocations (tick), absolute time (sec, msec, and usec), and input events.

  - count operates on the number of chart invocations (tick).

  - elapsed operates on absolute time (sec).

For more information, see "Design Human-Machine Interface Logic by Using Stateflow Charts" on page 34-28.

Note In standalone Stateflow charts, absolute time temporal logic is not supported in transitions with conditions.

- Function `getActiveStates` to access the states that are active during execution of the chart. To store the active states as a cell array, enter:

```
states = getActiveStates(chartObject)
```

- Function `matlab.internal.getcode.sfxfile` that generates the code implementing the standalone chart as a MATLAB object. To store the code as a character string `code`, enter:

```
code = matlab.internal.getcode.sfxfile('chart.sfx')
```

To save the code in the file `chart.m`, enter:

```
code = matlab.internal.getcode.sfxfile('chart.sfx',true)
```

Not supported (content specific to Simulink):

- Sample time and continuous-time semantics.
- C action language.
- Simulink functions and Simulink subsystems as states.
- Input, output, and parameter data.
- Data store memory data.
- Output and local events.
- Input, output, and local messages.

Other limitations:

- No Mealy or Moore semantics.
- No State Transition Tables.
- No Truth Table functions.
- No state-parented local data or functions.
- No transition actions (actions that execute after the source state for the transition is exited but before the destination state is entered).

- No change detection operators such as `hasChanged`, `hasChangedFrom`, or `hasChangedTo`.

See Also

`disp` | `edit` | `help`

More About

- "Execute and Unit Test Stateflow Chart Objects" on page 34-9
- "Execute Stateflow Chart Objects Through Scripts and Models" on page 34-21
- "Design Human-Machine Interface Logic by Using Stateflow Charts" on page 34-28
- "Implement a Financial Strategy by Using Stateflow" on page 34-38
- "Model a Communications Protocol by Using Chart Objects" on page 34-34

Execute and Unit Test Stateflow Chart Objects

A standalone Stateflow chart is a MATLAB class that defines the behavior of a finite state machine. Standalone charts implement classic chart semantics with MATLAB as the action language. You can program the chart by using the full functionality of MATLAB, including those functions that are restricted for code generation in Simulink. For more information, see "Create Stateflow Charts for Execution as MATLAB Objects" on page 34-2.

Example of a Standalone Stateflow Chart

The file `sf_chart.sfx` contains the standalone Stateflow chart `sf_chart`. The chart has an local data u, x, and y.
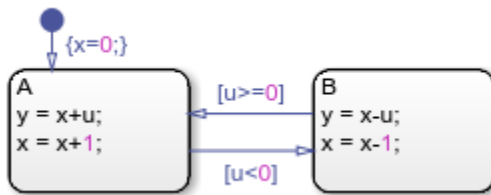
This example shows how to execute this chart from the Stateflow editor and in the MATLAB Command Window.

Execute a Standalone Chart from the Stateflow Editor

To unit test and debug a standalone chart, you can execute the chart directly from the Stateflow editor. During execution, you enter data values and broadcast events from the user interface.

1   Open the chart in the Stateflow editor:

    edit sf_chart.sfx

2   In the Symbols window, enter a value of u = 1 and click the **Run** icon ⏵. The editor creates a Stateflow chart object my_sf_chart and executes the default transition. The chart:

- Initializes x to the value of 0.
- Makes state A the active state.
- Assigns y to the value of 1.
- Increases the value of x to 1.

The chart animation highlights the active state A. The Symbols window displays the values u = 1, x = 1, and y = 1. The chart maintains its current state and local data until the next execution command.
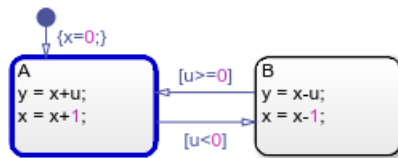
3   Click the **Run** icon ▶ again. Because the value of u does not satisfy the condition [u<0] to transition out of state A, this state remains active and the values of x and y increase to 2. The Symbols window now displays the values u = 1, x = 2, and y = 2.
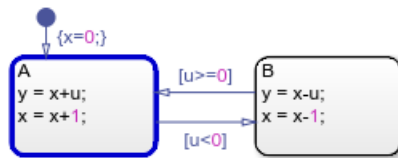
4   In the Symbols window, enter a value of u = −1 and click the **Run** icon ▶. The negative data value triggers the transition to state B. The Symbols window displays the values u = −1, x = 1, and y = 3.
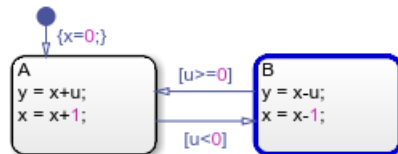
5   You can modify the value of any chart data in the Symbols window. For example, enter a value of x = 3. The chart will use this data value in the next time execution step.

6   Enter a value of u = 2 and click the **Run** icon ▶. The chart transitions back to state A. The Symbols window displays the values u = 2, x = 4, and y = 5.
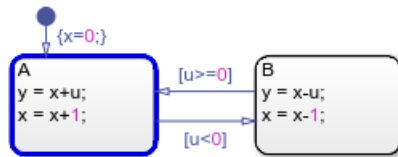
7   To stop the chart animation and remove the Stateflow chart object `my_sf_chart` from the MATLAB workspace, click the **Stop** icon ■.

To interrupt the execution and step through each action in the chart, add breakpoints before you execute the chart. For more information, see "Debug a Standalone Stateflow Chart" on page 34-16.

Execute a Standalone Chart in MATLAB

You can execute a standalone chart in MATLAB without opening the Stateflow editor. If the chart is open, then the Stateflow editor highlights active states and transitions through chart animation.

1   Open the chart in the Stateflow editor. In the MATLAB Command Window, enter:

```
edit sf_chart.sfx
```

2   Create the Stateflow chart object by using the name of the `sfx` file for the standalone chart as a function. Specify the initial value for the data u as a name-value pair.

```
s = sf_chart('u',1)

Stateflow Chart

   Execution Function
    y = step(s)

   Local Data
      u          : 1
      x          : 1
      y          : 1
   Active States: {'A'}
```

This command creates the chart object s, executes the default transition, and initializes the values of x and y. The Stateflow editor animates the chart and highlights the active state A.
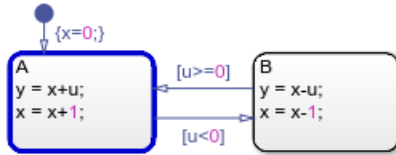
3 To execute the chart, call the `step` function. For example, suppose that you call the `step` function with a value of u = 1:

```
step(s,'u',1)
```

```
disp(s)
```

<u>Stateflow Chart</u>

```
    Execution Function
     y = step(s)

    Local Data
        u            : 1
        x            : 2
        y            : 2
    Active States: {'A'}
```

Because the value of u does not satisfy the condition [u<0] to transition out of state A, this state remains active and the values of x and y increase to 2.
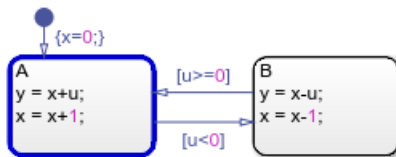
4 Execute the chart again, this time with a value of u = −1:

```
step(s,'u',-1)
```

```
disp(s)
```

```
Stateflow Chart

    Execution Function
     y = step(s)

    Local Data
        u           : -1
        x           : 1
        y           : 3
    Active States: {'B'}
```

The negative data value triggers the transition to state B. The value of x decreases to 1 and the value of y increases to 3.
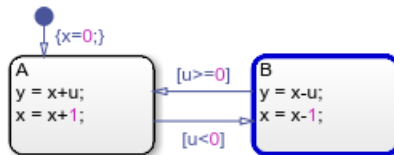
5   To access the value of any chart data, use dot notation. For example, you can assign a value of 3 to the local data x by entering:

```
s.x = 3
```

```
Stateflow Chart

    Execution Function
     y = step(s)

    Local Data
        u           : -1
        x           : 3
        y           : 3
    Active States: {'B'}
```

The standalone chart will use this data value in the next time execution step.

6   Execute the chart with a value of u = 2:

```
step(s,'u',2)
```

```
disp(s)
```

```
Stateflow Chart
```

```
Execution Function

 y = step(s)

Local Data
    u          : 2
    x          : 4
    y          : 5
Active States: {'A'}
```

The chart transitions back to state A and modifies the values of x and y.
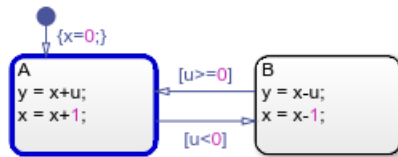
7   To stop the chart animation, delete the Stateflow chart object s from the MATLAB workspace:

```
delete(s)
```

Execute Multiple Chart Objects

You can execute multiple chart objects defined by the same standalone chart. Concurrent chart objects maintain their internal state independently, but remain associated to the same chart in the editor. The chart animation reflects the state of the chart object most recently executed. Executing multiple chart objects while the Stateflow editor is open can produce confusing results and is not recommended.

See Also
delete | disp | edit

More About

• "Design Human-Machine Interface Logic by Using Stateflow Charts" on page 34-28

Debug a Standalone Stateflow Chart

A standalone Stateflow chart is a MATLAB class that defines the behavior of a finite state machine. Standalone charts implement classic chart semantics with MATLAB as the action language. You can program the chart by using the full functionality of MATLAB, including those functions that are restricted for code generation in Simulink. For more information, see "Create Stateflow Charts for Execution as MATLAB Objects" on page 34-2.

You can execute a standalone chart in the MATLAB Command Window or through the Stateflow editor. To enable debugging, set a breakpoint in the standalone chart or in a MATLAB script that executes the chart. Breakpoints interrupt the execution of a chart. While execution is stopped, you can step through each action in the chart, view data values, and interact with the MATLAB workspace to examine the state of the chart.

Note When debugging a standalone chart that you execute from a MATLAB script, first open the Stateflow editor. Attempting to debug a standalone chart before opening the Stateflow editor at least once can produce unexpected results.

For information on debugging Stateflow charts in Simulink models, see "Debugging Charts" on page 32-2.

Set and Clear Breakpoints

You can set breakpoints on charts, states, and transitions.

- To add a breakpoint on a chart, right-click inside the chart and select **Set Breakpoint on Chart Entry**. This type of breakpoint interrupts the execution before the chart is entered. To remove the breakpoint, right-click inside the chart and clear the **Set Breakpoint on Chart Entry** option.
- To add a breakpoint on a state, right-click the state and select **Set Breakpoint**. This type of breakpoint interrupts the execution before the chart performs the `entry` and `during` actions for the state. To remove the breakpoint, right-click the state and select **Clear Breakpoint**.
- To add a breakpoint on a transition, right-click the transition and select **Set Breakpoint**. This type of breakpoint interrupts the execution when the transition becomes valid. To remove the breakpoint, right-click the transition and select **Clear Breakpoint**.

Breakpoints appear as circular red badges. For example, this chart contains breakpoints on the state A and on the transition from A to B.
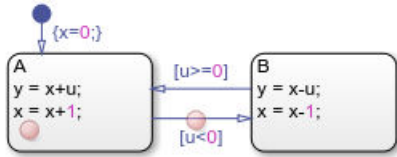
To remove all of the breakpoints in a chart, right-click inside the chart and select **Clear All Breakpoints In Chart**.

Control Chart Execution After a Breakpoint

When execution stops at a breakpoint, Stateflow enters debugging mode.

- The MATLAB command prompt changes to K>>.
- The Symbols window displays the value of each data object in the chart.
- The chart highlights active elements in blue and the currently executing object in green.

For example, when the execution stops at the breakpoint in state A, the border of the state and the first statement in the state entry action appear highlighted in green.
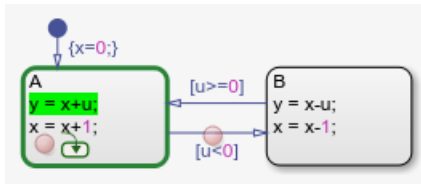
An execution status badge appears in the graphical object where execution pauses.

Badge	Description
	Execution stopped before entering a chart or state.
	Execution stopped in a state during action.
	Execution stopped after exiting a state.

34-17

Badge	Description
	Execution stopped before taking a valid transition.

You can continue the execution from the Stateflow editor, at the debugging prompt, or by selecting a keyboard shortcut.

Toolbar Icon	Option	Command	Description	Keyboard Shortcut
	Continue	dbcont	Continue execution to the next breakpoint.	**Ctrl+T**
	Step Over	dbstep	Advance to the next step in the chart execution. At the chart level, possible steps include: • Enter the chart • Execute a transition action • Activate a state • Execute a state action For more information, see "Execution of a Stateflow Chart" on page 3-44.	**F10**

Toolbar Icon	Option	Command	Description	Keyboard Shortcut
	Step In	`dbstep in`	From a state or transition action that calls a function, advance to the first executable statement in the function. From a statement in a function containing another function call, advance to the first executable statement in the second function. Otherwise, advance to the next step in the chart execution. (See the Step Over option.)	**F11**
	Step Out	`dbstep out`	From a function call, return to the statement calling the function. Otherwise, continue execution to the next breakpoint. (See the Continue option.)	**Shift+F11**
	Stop	`dbquit`	Exit debug mode and interrupt the execution. When you execute the standalone chart from the Stateflow editor, this action removes the chart object from the MATLAB workspace.	**Ctrl+Shift+T**

In state or transition actions containing more than one statement, you can step through the individual statements one at a time by selecting **Step Over**. The Stateflow Editor highlights each statement before executing it.

Examine and Change Values of Chart Data

When Stateflow is in debug mode, the Symbols window displays the value of each data object in the chart. You can also examine data values by pointing to a state or a transition in the chart. A tooltip displays the value of each data object used in the selected state or transition.

To test the behavior of your chart, in the Symbols window, you can change the value of a data object during execution. Alternatively, at the debugging prompt, enter the new value by using the name `this` in place of the chart object name. For instance, to change the value of the local data `x`, enter:

```
this.x = 7
```

The new value appears in the Symbols window.

See Also

More About

- "Create Stateflow Charts for Execution as MATLAB Objects" on page 34-2
- "Execute and Unit Test Stateflow Chart Objects" on page 34-9
- "Debugging Charts" on page 32-2

Execute Stateflow Chart Objects Through Scripts and Models

A standalone Stateflow chart is a MATLAB class that defines the behavior of a finite state machine. Standalone charts implement classic chart semantics with MATLAB as the action language. You can program the chart by using the full functionality of MATLAB, including those functions that are restricted for code generation in Simulink. For more information, see "Create Stateflow Charts for Execution as MATLAB Objects" on page 34-2.

This example shows how to execute a standalone Stateflow chart by using a MATLAB script or a Simulink model.

Count Ways to Make Change for Currency

The file `sf_change.sfx` defines a standalone Stateflow chart that counts the number of ways to make change for a given amount of money. The chart contains these data objects:

- `x` is the amount of money to change. The default value is 100.
- `coinValues` is a vector of coin denominations arranged in increasing order. `coinNames` is an array of corresponding coin names. The default values represent standard American coins (pennies, nickels, dimes, and quarters).
- `tally` is the number of valid change configurations.
- `tabula` is an array containing the different valid change configurations.
- `chg`, `done`, `i`, and `n` are local data used by the change-counting algorithm.
- `textWidth` and `quietMode` are local data that control how the chart displays its results.
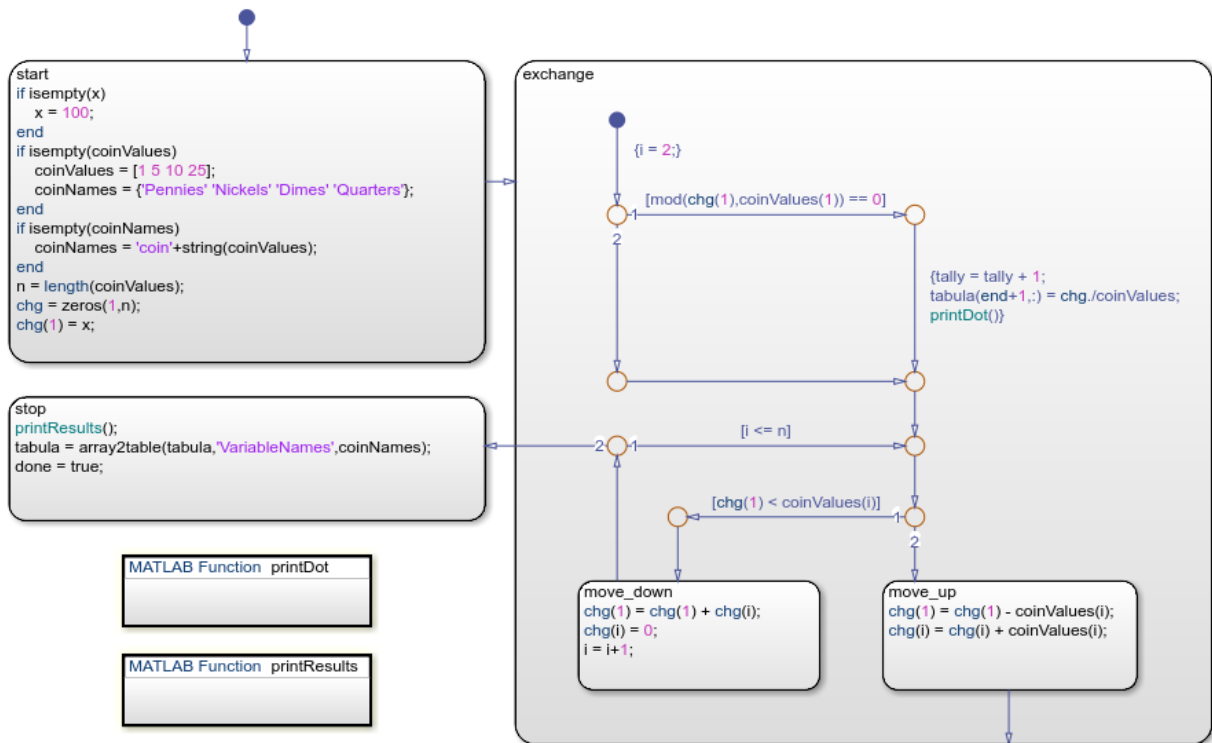
Copyright 2018-2019 The MathWorks, Inc.

The chart begins with a potential change configuration consisting entirely of the lowest-value coins, specified by an index of 1. At each execution step, the state `exchange` modifies this configuration in one of two ways:

- The substate `move_up` exchanges some lowest-value coins for a coin with a higher value, specified by the index `i`.

- The substate `move_down` exchanges all of the coins with the value specified by the index `i` for lowest-value coins. Then `move_up` exchanges some lowest-value coins for a coin with a value specified by the index `i+1` or higher.

A potential change configuration is valid when the number of cents represented by the lowest-value coins is divisible by the value of that type of coin. When the chart encounters a new valid configuration, it increments `tally` and appends the new configuration to `tabula`.

When no more coin exchanges are possible, the state `stop` becomes active. This state prints the results of the computation, transforms the contents of `tabula` to a table, and sets the value of `done` to `true`.

Execute Standalone Chart in a MATLAB Script

To run the change-counting algorithm to completion, you must execute the standalone chart multiple times. For example, the MATLAB script `sf_change_script.m` creates a chart object `chartObj` and initializes the value of the local data `x` to 27. The configuration option `'-warningOnUninitializedData'`, which the script sets to `false`, eliminates the warning that `tabula` is an empty array in the new chart object. The `while` loop executes the chart until the local data `done` becomes `true`. Finally, the script displays the value of `tabula`.

```
chartObj = sf_change('x',27,'--warningOnUninitializedData',false);

while ~chartObj.done
    step(chartObj);
end

disp(chartObj.tabula)
```

When you run this script, the standalone chart calculates the number of ways to make change for 27 cents by using standard American coins:

```
sf_change_script
```

```
.............
There are 13 ways to make change for 27 cents.
    Pennies    Nickels    Dimes    Quarters
    _____    _____    _____    _____

      27          0          0         0
      22          1          0         0
      17          2          0         0
      12          3          0         0
       7          4          0         0
       2          5          0         0
      17          0          1         0
      12          1          1         0
       7          2          1         0
       2          3          1         0
       7          0          2         0
```

```
2        1        2        0
2        0        0        1
```

To determine the number of ways to make change for a different amount, or to use a different system of currency, change the values of x and `coinValues`. For example, to use British currency, initialize `coinValues` to [1 2 5 10 20 25 50].

Execute Standalone Chart in a Simulink Model

You can execute a standalone Stateflow chart from within a Simulink model. For example, the model `sf_change_model` contains two Stateflow charts that use the standalone chart `sf_change` to count the number of ways to make change for 27 cents in two different currency systems. You can simulate the model, but the functions that execute the standalone chart do not support code generation.
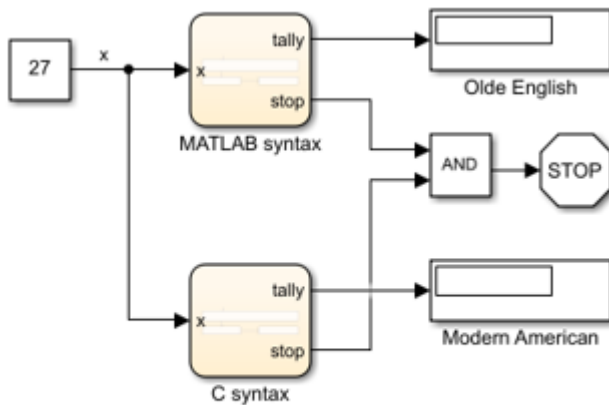
Copyright 2018-2019 The MathWorks, Inc.

Each chart contains these states:

- `Initialize` creates a local chart object `chartObj` that implements the change-counting algorithm for the input value x.

- `Execute` calls the `step` function to execute the standalone chart and stores the result as the output data `tally`.

- `Finish` displays the results of the algorithm in the Diagnostic Viewer window and sets the value of the output data `done` to `true`.

When both charts reach their respective `Finish` state, the simulation of the model stops and the Display blocks show the final values of the two tallies.

Execution Using MATLAB as the Action Language

The chart `MATLAB syntax` uses MATLAB as the action language. To execute the standalone Stateflow chart, this chart must follow these guidelines:

- The local variable `chartObj` that contains the handle to the chart object has type `Inherit: From definition in chart`.

- Before creating the chart object, the `Initialize` state calls the `coder.extrinsic` function to declare `sf_change` as an extrinsic function that is restricted for code generation in Simulink. See "Call Extrinsic MATLAB Functions in Stateflow Charts" on page 30-39.

- The `Execute` and `Finish` states access the local data for the standalone chart by calling the `get` function.
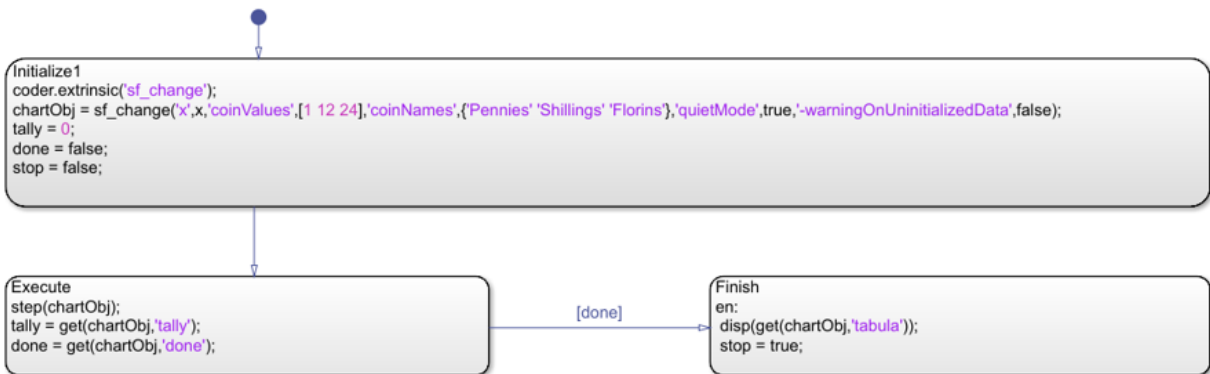
When you simulate this chart with an input of `x = 27`, the Display block `Olde English` shows a tally of 4. The Diagnostic Viewer window shows these results:

```
Pennies     Shillings     Florins
_____     _____     _____
   27           0            0
   15           1            0
```

```
3              2              0
3              0              1
```

Execution Using C as the Action Language

The chart `C syntax` uses C as the action language. To execute the standalone Stateflow chart, this chart must follow these guidelines:

- The local variable `chartObj` that contains the handle to the chart object has type `ml`.

- The `Initialize` state calls the `ml` function to create the chart object.

- The `Execute` and `Finish` states use the `ml` namespace operator to access the `step`, `get`, and `displ` functions to execute the standalone chart, access its local data, and display the results of the algorithm.

For more information, see "Access MATLAB Functions and Workspace Data in C Charts" on page 12-32.

Initialize
chartObj = ml('sf_change("x",%d,"coinValues",[2 3 7],"coinNames",{"Safety" "FieldGoal" "TouchDown"},"quietMode",true,"-warningOnUninitializedData",false)',x);
tally = 0;
done = false;
stop = false;

Execute1
ml.step(chartObj);
tally = ml.get(chartObj,'tally');
done = ml.get(chartObj,'done');

[done]

Finish1
en:
ml.disp(ml.get(chartObj,'tabula'));
stop = true;
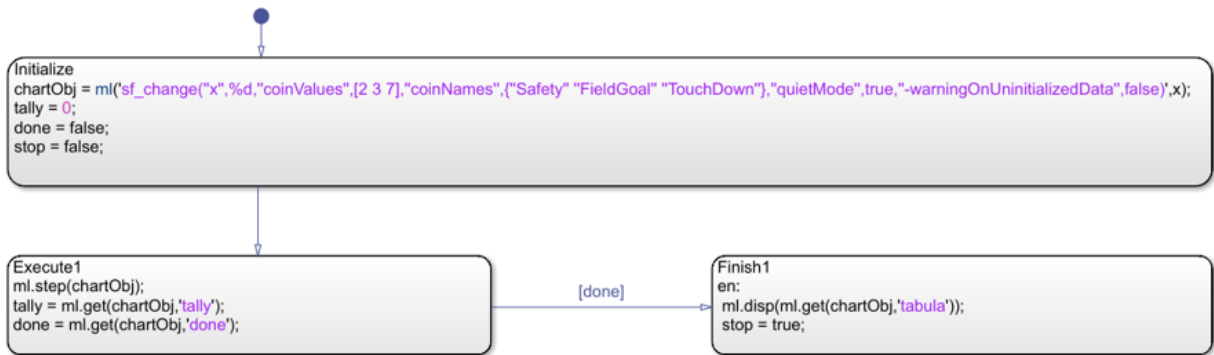
When you simulate this chart with an input of x = 27, the Display block `Modern American` shows a tally of 13. The Diagnostic Viewer window shows these results:

Safety	FieldGoal	TouchDown
12	1	0
9	3	0
6	5	0
3	7	0
0	9	0
10	0	1

```
7          2          1
4          4          1
1          6          1
5          1          2
2          3          2
3          0          3
0          2          3
```

See Also

More About

- "Create Stateflow Charts for Execution as MATLAB Objects" on page 34-2
- "Execute and Unit Test Stateflow Chart Objects" on page 34-9
- "Call Extrinsic MATLAB Functions in Stateflow Charts" on page 30-39
- "Access MATLAB Functions and Workspace Data in C Charts" on page 12-32

Design Human-Machine Interface Logic by Using Stateflow Charts

A standalone Stateflow chart is a MATLAB class that defines the behavior of a finite state machine. Standalone charts implement classic chart semantics with MATLAB as the action language. You can program the chart by using the full functionality of MATLAB, including those functions that are restricted for code generation in Simulink. For more information, see "Create Stateflow Charts for Execution as MATLAB Objects" on page 34-2.

You can execute a standalone Stateflow chart by invoking its input events and using temporal operators. The event- and timer-driven execution workflow is suitable for designing the logic underlying human-machine interfaces (HMIs) and graphical user interfaces (UIs).

- When you use the MATLAB App Designer, callback functions from the interface widgets invoke events in the chart.
- In the Stateflow chart, temporal operators and local data control the properties of the user interface.

For more information on how to use MATLAB to create graphical user interfaces, see "App Designer" (MATLAB).

Control an App Designer User Interface

This user interface contains an On-Off switch that controls a lamp. When the switch is in the On position, the lamp lights up in one of two modes, solid or blinking, depending on the position of the Mode option button. You control the rate of blinking by moving the Blink Rate slider.
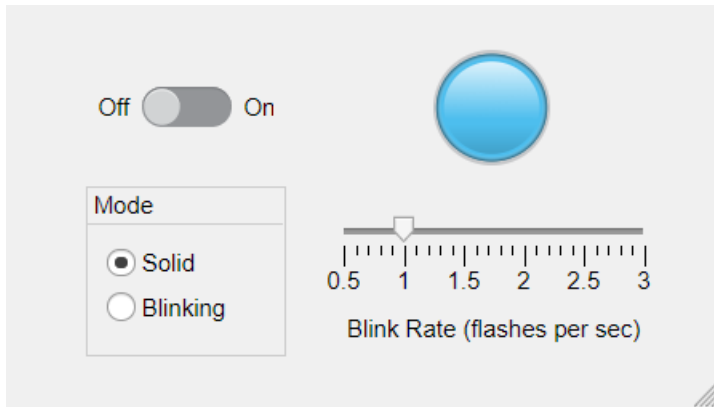
The file `sf_lamp_logic.sfx` defines a standalone Stateflow chart that implements the logic for the user interface. The chart has input events (`ON`, `OFF`, `BLINKING`, and `SOLID`) and local data (`delay` and `app`). The actions in the chart control which widgets are accessible from each state. For instance, the actions in the `Off` state cause the Lamp widget, the Mode option buttons, and the Blink Rate slider in the user interface to appear dimmed.
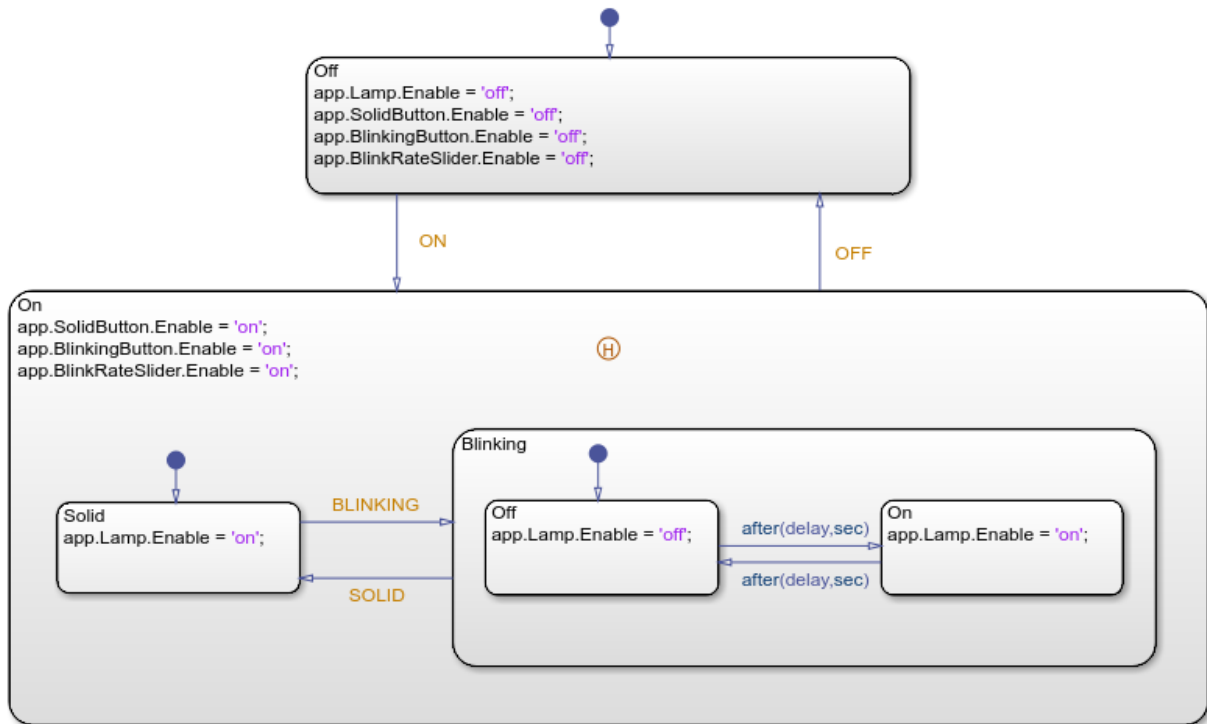
Copyright 2018-2019 The MathWorks, Inc.

In the `On` state, the substates `Solid` and `Blinking` denote the two modes of operation. To implement a blinking lamp, the chart relies on the temporal logic operator `after`. When the chart enters the state `Blinking.Off`, the expression `after(delay,sec)` on the outgoing transition creates a MATLAB timer object that executes the chart after a number of seconds. The chart then transitions to the state `Blinking.On` and creates another timer object to trigger the transition back to `Blinking.Off`. While the chart continually transitions between the two states, you can adjust the rate of blinking by changing the value of the local data delay or transition out of blinking mode by invoking the input events `SOLID` or `OFF`.

The history junction in the On state preserves information on the most recently active substate so that the user interface returns to the previous mode of operation when you turn on the lamp.

Execute Standalone Chart by Using Events

You can execute the standalone chart by calling its input event functions in the MATLAB Command Window. The Stateflow editor shows the effects of each of these commands by highlighting active states and transitions through chart animation.

1 Open the standalone chart in the Stateflow editor.

   ```
   edit sf_lamp_logic.sfx
   ```

2 Create the chart object L and initialize the value of delay to 0.5. This value corresponds to a blinking rate of one flash per second.

   ```
   L = sf_lamp_logic('delay',0.5);
   ```

3 Turn on the lamp.

   ```
   ON(L)
   ```

4 Switch to blinking mode.

   ```
   BLINKING(L)
   ```

5 Set the value of delay to 0.25. This value corresponds to a blinking rate of two flashes per second.

   ```
   L.delay = 0.25;
   ```

6 Switch to solid mode.

   ```
   SOLID(L)
   ```

7 Turn off the lamp.

   ```
   OFF(L)
   ```

8 Delete the chart object L from the MATLAB workspace.

   ```
   delete(L)
   ```

Connect Standalone Chart to User Interface

To establish a bidirectional connection between the user interface and the standalone Stateflow chart, open the App Designer window and select **Code View**.

1   In the App Designer window, create a private property `lampLogic` to store the handle to the Stateflow chart object.

```
properties (Access = private)
    lampLogic
end
```

2   Create a `StartupFcn` callback function that creates the chart object and sets its local data `app` to the user interface handle. Assign the chart object handle to the `lampLogic` private property.

```
% Code that executes after component creation
function StartupFcn(app)
    app.lampLogic = sf_lamp_logic('delay',0.5,'app',app);
end
```

3   Create a `CloseRequestFcn` callback function that deletes the chart object when you close the user interface.

```
% Close request function: UIFigure
function UIFigureCloseRequest(app, event)
    delete(app.lampLogic);
    delete(app);
end
```

4   For each one of the user interface widgets, add a callback function that invokes the appropriate event on the standalone chart.

- `ValueChangedFcn` callback function for Switch widget:

```
function SwitchValueChanged(app,event)
    value = app.Switch.Value;
    switch lower(value)
        case 'off'
            OFF(app.lampLogic);
        case 'on'
            ON(app.lampLogic);
    end
end
```

- `SelectionChangedFcn` callback function for Mode Button widget:

```
function ModeButtonGroupSelectionChanged(app,event)
    selectedButton = app.ModeButtonGroup.SelectedObject;
    if app.SolidButton == selectedButton
        SOLID(app.lampLogic);
    else
```

```
        BLINKING(app.lampLogic);
    end
end
```

- `ValueChangedFcn` callback function for Blink Rate Slider widget:

```
function BlinkRateSliderValueChanged(app,event)
    app.lampLogic.delay = 0.5/app.BlinkRateSlider.Value;
end
```

When you run the user interface, you can observe the effects of adjusting the control widgets on the chart canvas and on the lamp widget.

See Also

More About

- "Create Stateflow Charts for Execution as MATLAB Objects" on page 34-2
- "History Junctions" on page 2-44
- "Activate a Stateflow Chart by Sending Input Events" on page 10-9
- "Control Chart Execution by Using Temporal Logic" on page 12-49
- "App Designer" (MATLAB)

Model a Communications Protocol by Using Chart Objects

A standalone Stateflow chart is a MATLAB class that defines the behavior of a finite state machine. Standalone charts implement classic chart semantics with MATLAB as the action language. You can program the chart by using the full functionality of MATLAB, including those functions that are restricted for code generation in Simulink. For more information, see "Create Stateflow Charts for Execution as MATLAB Objects" on page 34-2.

Implement a Symbol-Detection Algorithm

This example shows how to use a standalone Stateflow chart to model a frame-synchronization and symbol-detection component in a communications system. The input to the system consists of a binary signal of zeros and ones received every 10 milliseconds. The input signal can contain any combination of:

*   A 770-ms pulse (77 consecutive ones) to mark the beginning and end of a frame of data and to ensure system synchronization.

*   A 170-ms pulse (17 consecutive ones) to indicate symbol A.

*   A 470-ms pulse (47 consecutive ones) to indicate symbol B.

The file `sf_frame_search.sfx` defines a standalone Stateflow chart that implements this communication protocol. The chart consists of two outer states in parallel decomposition. The `Initialize` state resets the value of the local data `symbol` at the start of each execution step. The `Search` state contains the logic that defines the symbol-detection algorithm. When this state detects one of the pulses allowed by the communication protocol, the name of the corresponding symbol is stored as `symbol` and displayed in the MATLAB Command Window. Parallel decomposition enables the chart to preprocess the input data. For more information, see "State Decomposition" on page 2-16.
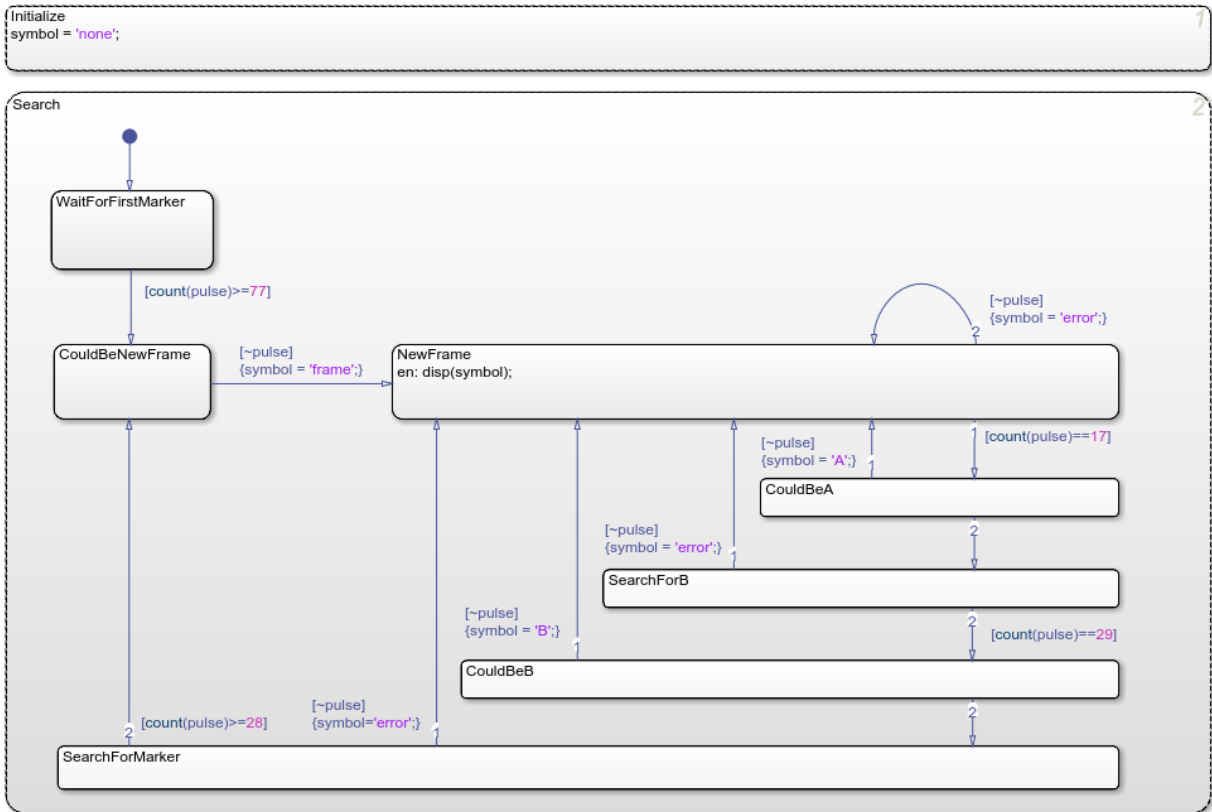
To track the length of a pulse through several execution steps, the chart uses the `count` operator. This operator simplifies the design of the chart by eliminating the need for a manual counter. For example, the condition `[count(pulse)==17]` guards the outgoing transition from the substate `NewFrame`. This condition becomes true when the data `pulse` is one for 17 consecutive execution steps. In this case, the chart transitions to the `CouldBeA` substate. If this transition is followed by an input of zero, then the chart registers the reception of symbol A and transitions back to the `NewFrame` substate.

Otherwise, the chart transitions to the `SearchForB` state from which the condition `[count(pulse)==29]` searches for an additional 29 ones to mark symbol B.

Execute Standalone Chart

In the MATLAB script `sf_frame_tester.m`, the sample code generates a short signal consisting of several valid pulses and one transmission error. The error consists of a 470-ms pulse that is too long to represent symbol A and too short to represent symbol B.

```matlab
f = sf_frame_search('pulse','0');    % create chart object
sendPulse(f,77);                      % frame marker
sendPulse(f,17);                      % A
sendPulse(f,47);                      % B
sendPulse(f,37);                      % transmission error
sendPulse(f,47);                      % B
sendPulse(f,17);                      % A
sendPulse(f,77);                      % frame marker
delete(f);                            % delete chart object

function sendPulse(f,n)
% Send a pulse of n ones and one zero to chart object f.

    for i = 1:n
        step(f,'pulse',1)
        printDot(1)
    end

    printDot(0)
    step(f,'pulse',0)

function printDot(x)
    persistent k
    if isempty(k)
        k = 1;
    end

    if x == 0
        fprintf('\n');
        k = 1;
    elseif k == 50
        fprintf('.\n');
        k = 1;
    else
```

```
        fprintf('.');
        k = k+1;
    end
end
end
```

Running the script produces these results in the MATLAB Command Window:

```
sf_frame_Tester

..............................................
...........................
frame
................
A
.............................................
B
.....................................
error
.............................................
B
................
A...................................................
...........................
frame
```

See Also

count

More About

- "Create Stateflow Charts for Execution as MATLAB Objects" on page 34-2
- "Execute and Unit Test Stateflow Chart Objects" on page 34-9
- "Execute Stateflow Chart Objects Through Scripts and Models" on page 34-21
- "State Decomposition" on page 2-16
- "Control Chart Execution by Using Temporal Logic" on page 12-49

# Implement a Financial Strategy by Using Stateflow

A standalone Stateflow chart is a MATLAB class that defines the behavior of a finite state machine. Standalone charts implement classic chart semantics with MATLAB as the action language. You can program the chart by using the full functionality of MATLAB, including those functions that are restricted for code generation in Simulink. For more information, see "Create Stateflow Charts for Execution as MATLAB Objects" on page 34-2.

Compute Bollinger Bands

This example shows how to use a standalone Stateflow chart to model a financial trading strategy known as Bollinger Bands. The strategy is to maintain a moving average of N stock prices for some commodity and issuing trading instructions depending on the value of the stock:

- "Buy" when the value of the stock drops K standard deviations below the moving average.
- "Sell" when the value of the stock rises K standard deviations above the moving average.
- "Hold" when the value of the stock is within K standard deviations of the moving average.

Typical implementations for this strategy use values of N = 20 and K = 2.

The file sf_stock_watch.sfx defines a standalone Stateflow chart that implements this financial strategy. The chart consists of two outer states in parallel decomposition.

- The StockTicker subchart records the current price of a stock. The subchart hides the details for calculating stock prices. To access real-time market data from financial data providers, one possible implementation involves the use of the Datafeed Toolbox™. For details, see "Datafeed Toolbox".

- The FinancialAdvisor state uses the last N stock prices to compute high and low bands. Depending on the current price relative to these bands, the state generates "buy," "sell," or "hold" instructions. The action on every(1,sec) creates a MATLAB® timer to execute the chart every second. See "Control Chart Execution by Using Temporal Logic" on page 12-49.

Parallel decomposition is a common design pattern that enables your algorithm to preprocess input data. For more information, see "State Decomposition" on page 2-16.
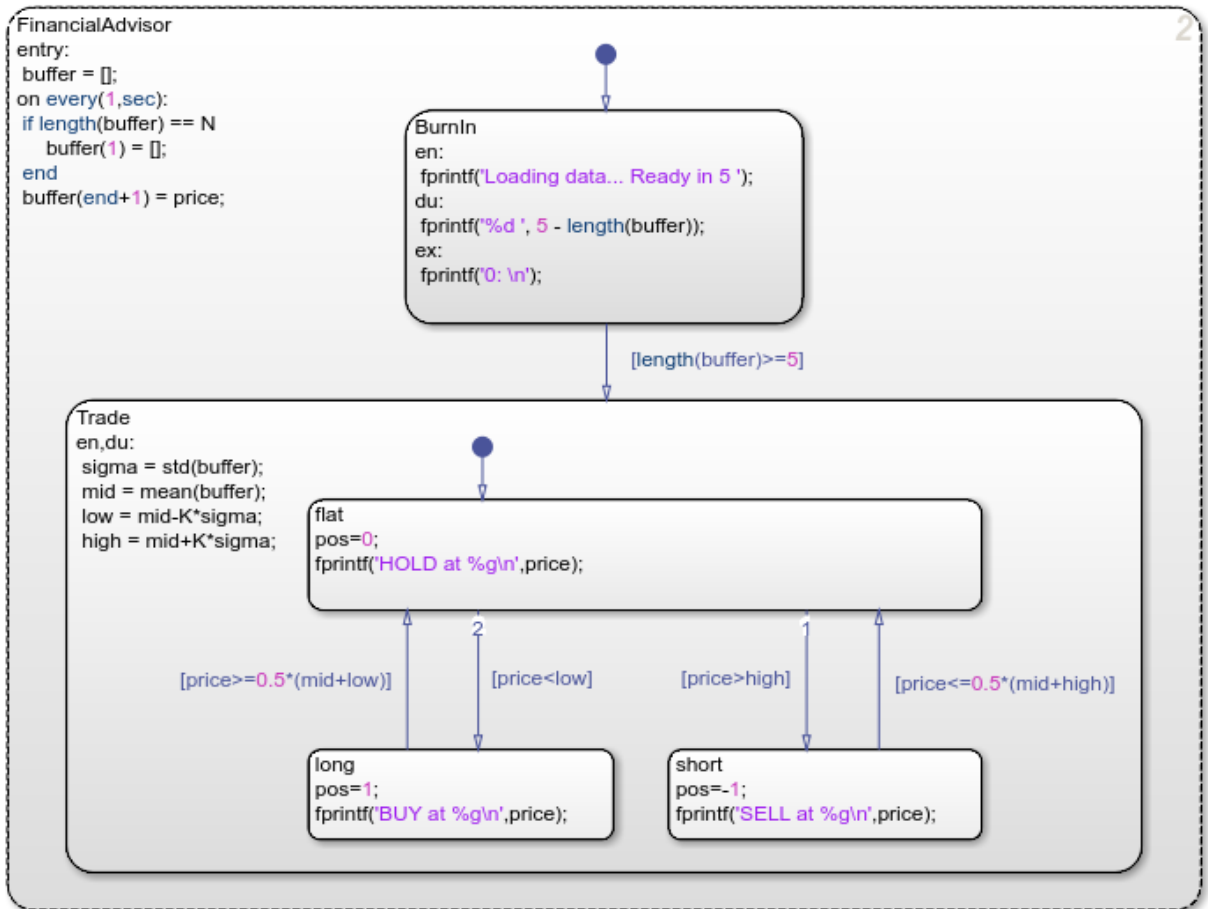
Execute Standalone Chart

To execute the standalone chart, create a Stateflow chart object w with the configuration option '-warningOnUninitializedData' set to `false`:

```
w = sf_stock_watch('-warningOnUninitializedData',false);

Loading data... Ready in 5 4 3 2 1 0:
HOLD at 14.1942
SELL at 14.2802
SELL at 14.247
HOLD at 14.2025
BUY at 14.1444
```

The chart continues to run until you delete the chart object w from the MATLAB workspace:

```
delete(w);
```

See Also

More About

- "Create Stateflow Charts for Execution as MATLAB Objects" on page 34-2
- "Execute and Unit Test Stateflow Chart Objects" on page 34-9
- "State Decomposition" on page 2-16
- "Control Chart Execution by Using Temporal Logic" on page 12-49
- "Datafeed Toolbox"

Analog Trigger App by Using Stateflow Charts

This example shows how to create an analog-triggered data acquisition app by using Stateflow®, Data Acquisition Toolbox™, and App Designer.

Data Acquisition Toolbox provides functionality for acquiring measurement data from a DAQ device or audio soundcard. For certain applications, an analog-triggered acquisition that starts capturing or logging data based on a condition in the analog signal being measured is recommended. Software-analog triggered acquisition enables you to capture only a segment of interest out of a continuous stream of measurement data. For example, you can capture an audio recording when the signal level passes a certain threshold.

This example app, created by using App Designer and Stateflow, shows how to implement these operations:

- Control the app state logic by using a Stateflow chart.
- Discover available DAQ devices and select which device to use.
- Configure device acquisition parameters.
- Display a live plot in the app UI during acquisition.
- Perform a triggered data capture based on a programmable trigger condition.
- Save captured data to a MATLAB® base workspace variable.
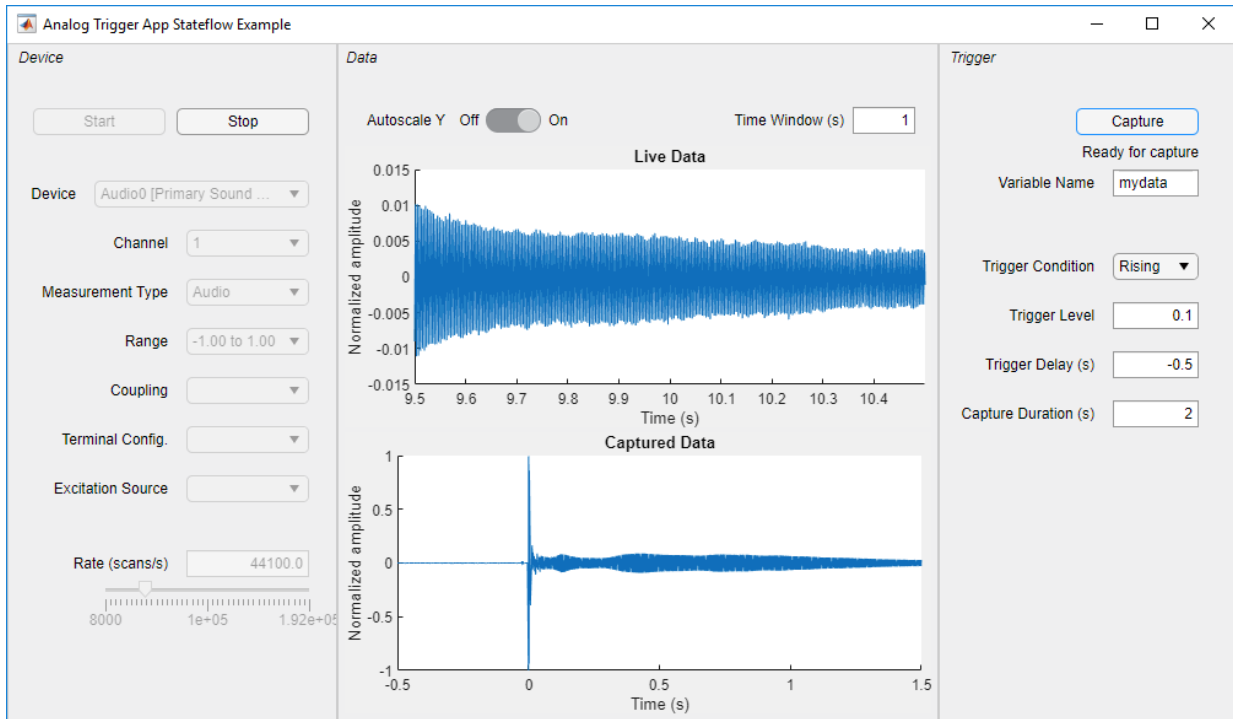
By default, the app opens in design mode in App Designer. To run the app click the **Run** button or execute the app from the command line:

```
AnalogTriggerAppStateflow
```

Requirements

This example app requires:

- MATLAB R2019a or later
- Data Acquisition Toolbox
- Stateflow for creating and editing charts
- A supported DAQ device or sound card. For example, any National Instruments or Measurement Computing device that supports analog input `Voltage` or `IEPE` measurements and background acquisition.
- Corresponding hardware support package and device drivers

34-43

App States and the Stateflow Chart

When creating an app that has complex logic, consider the various states that correspond to the operating modes of the app. You can use a Stateflow chart to visualize and organize these app states. Use transitions between states to implement the control logic of your app. For example, the file `AnalogTriggerAppChart.sfx` defines the Stateflow chart that controls the logic for this app. The chart can transition between states based on an action in the app UI or on a data-driven condition. For example, if you click the **Start** button, the chart transitions from the `Configuration` state to the `Acquisition` state. If the value of the signal crosses the specified trigger level, the chart transitions from the `LookingForTrigger` state to the `CapturingData` state.
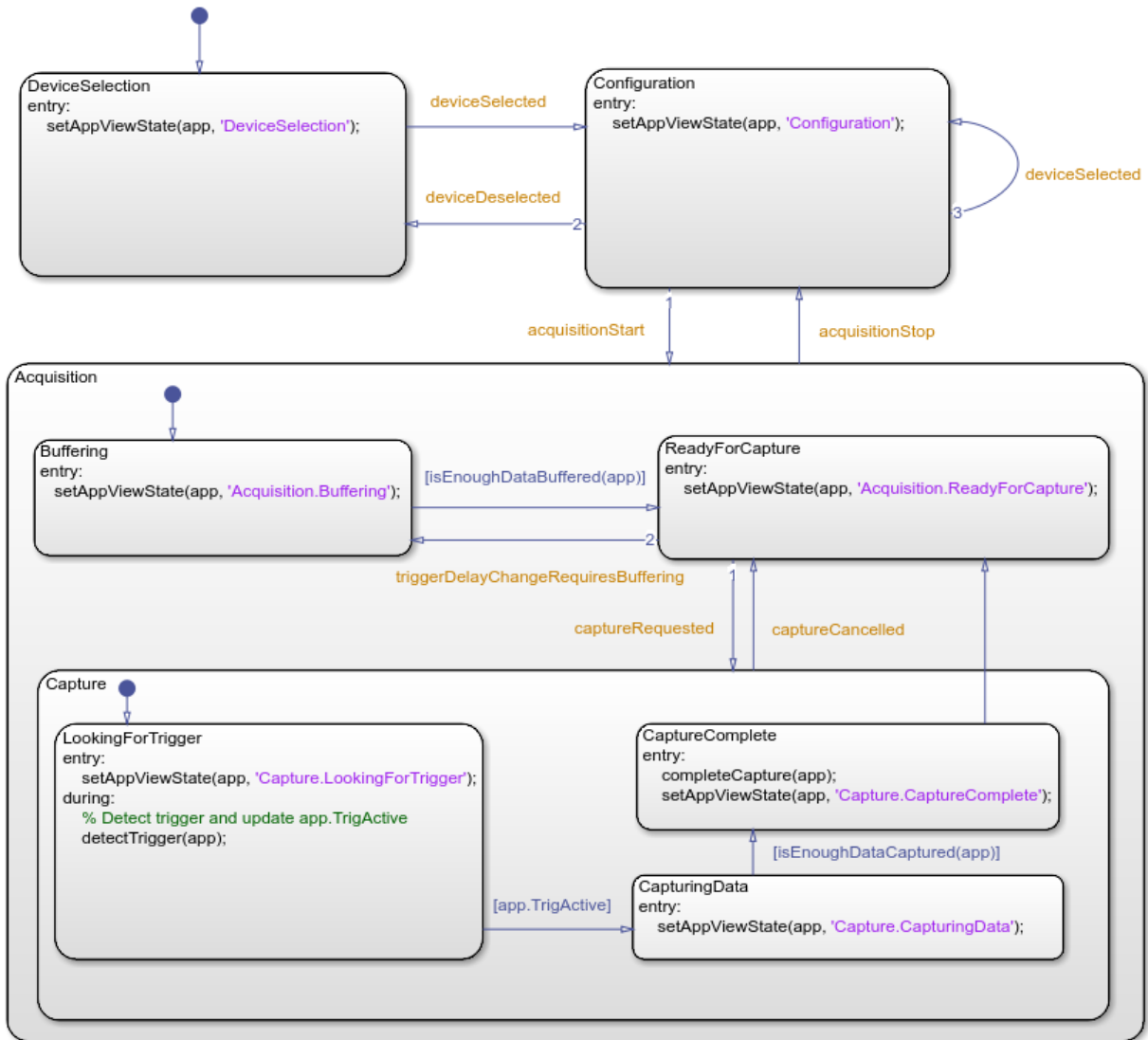
Integrating the App with the Stateflow Chart

To establish a bidirectional connection between the MATLAB app and the Stateflow chart, in the `startupFcn` function of your app, create a chart object and store its handle in an app property.

```
app.Chart = AnalogTriggerAppChart('app',app);
```

The app uses this handle to trigger state transitions in the chart. For example, when you click **Start**, the `StartButtonPushed` app callback function calls the `acquisitionStart` input event for the chart. This event triggers the transition from the `Configuration` state to the `Acquisition` state.

To evaluate transition conditions that are not events in the chart, the app calls the `step` function for the chart object. For example, while acquiring data from the device, the `dataAvailable_Callback` app function periodically calls the `step` function. When the trigger condition is detected, the chart transitions from the `LookingForTrigger` State to the `CapturingData` state.

In the Stateflow chart, store the app object handle as chart local data. To share public properties and call public functions of the app, the Stateflow chart can use this handle in state actions, transition conditions, or transition actions.

See Also
Chart

More About
- "Design Human-Machine Interface Logic by Using Stateflow Charts" on page 34-28

Semantic Rules Summary

Summary of Chart Semantic Rules

In this section...
"Enter a Chart" on page A-2
"Execute an Active Chart" on page A-2
"Enter a State" on page A-2
"Execute an Active State" on page A-3
"Exit an Active State" on page A-3
"Execute a Set of Flow Charts" on page A-4
"Execute an Event Broadcast" on page A-5

Enter a Chart

The set of default flow paths execute (see "Execute a Set of Flow Charts" on page A-4). If this action does not cause a state entry and the chart has parallel decomposition, then each parallel state becomes active (see "Enter a State" on page A-2).

If executing the default flow paths does not cause state entry, a state inconsistency error occurs.

Execute an Active Chart

If the chart has no states, each execution is equivalent to initializing a chart. Otherwise, the active children execute. Parallel states execute in the same order that they become active.

Enter a State

1 If the parent of the state is not active, perform steps 1 through 4 for the parent.

2 If this state is a parallel state, check that all siblings with a higher (that is, earlier) entry order are active. If not, perform steps 1 through 5 for these states first.

Parallel (AND) states are ordered for entry based on whether you use explicit ordering (default) or implicit ordering. For details, see "Explicit Ordering of Parallel States" on page 3-86 and "Implicit Ordering of Parallel States" on page 3-88.

3 Mark the state active.

4 Perform any entry actions.

5 Enter children, if needed:

    a If the state contains a history junction and there was an active child of this state at some point after the most recent chart initialization, perform the entry actions for that child. Otherwise, execute the default flow paths for the state.

    b If this state has children that are parallel states (parallel decomposition), perform entry steps 1 through 5 for each state according to its entry order.

    c If this state has only one child substate, the substate becomes active when the parent becomes active, regardless of whether a default transition is present. Entering the parent state automatically makes the substate active. The presence of any inner transition has no effect on determining the active substate.

6 If this state is a parallel state, perform all entry steps for the sibling state next in entry order if one exists.

7 If the transition path parent is not the same as the parent of the current state, perform entry steps 6 and 7 for the immediate parent of this state.

Execute an Active State

1 The set of outer flow charts execute (see "Execute a Set of Flow Charts" on page A-4). If this action causes a state transition, execution stops. (Note that this step never occurs for parallel states.)

2 During actions and valid on-event actions are performed.

3 The set of inner flow charts execute. If this action does not cause a state transition, the active children execute, starting at step 1. Parallel states execute in the same order that they become active.

Exit an Active State

1 If this is a parallel state, make sure that all sibling states that became active after this state have already become inactive. Otherwise, perform all exiting steps on those sibling states.

2 If there are any active children, perform the exit steps on these states in the reverse order that they became active.

3 Perform any exit actions.

4 Mark the state as inactive.

Execute a Set of Flow Charts

Flow charts execute by starting at step 1 below with a set of starting transitions. The starting transitions for inner flow charts are all transition segments that originate on the respective state and reside entirely within that state. The starting transitions for outer flow charts are all transition segments that originate on the respective state but reside at least partially outside that state. The starting transitions for default flow charts are all default transition segments that have starting points with the same parent:

1 Ordering of a set of transition segments occurs.

2 While there are remaining segments to test, testing a segment for validity occurs. If the segment is invalid, testing of the next segment occurs. If the segment is valid, execution depends on the destination:

States

    a Testing of transition segments stops and a transition path forms by backing up and including the transition segment from each preceding junction until the respective starting transition.

    b The states that are the immediate children of the parent of the transition path exit (see "Exit an Active State" on page A-3).

    c The transition action from the final transition path executes.

    d The destination state becomes active (see "Enter a State" on page A-2).

Junctions with no outgoing transition segments

Testing stops without any state exits or entries.

Junctions with outgoing transition segments

Step 1 is repeated with the set of outgoing segments from the junction.

3 After testing all outgoing transition segments at a junction, backtrack the incoming transition segment that brought you to the junction and continue at step 2, starting with the next transition segment after the backtrack segment. The set of flow charts finishes execution when testing of all starting transitions is complete.

Execute an Event Broadcast

Output edge-trigger event execution is equivalent to changing the value of an output data value. All other events have the following execution:

1 If the *receiver* of the event is active, then it executes (see "Execute an Active Chart" on page A-2 and "Execute an Active State" on page A-3). (The event *receiver* is the parent of the event unless a direct event broadcast occurs using the send() function.)

If the receiver of the event is not active, nothing happens.

2 After broadcasting the event, the broadcaster performs early return logic based on the type of action statement that caused the event.

Action Type	Early Return Logic
State Entry	If the state is no longer active at the end of the event broadcast, any remaining steps in entering a state do not occur.
State Exit	If the state is no longer active at the end of the event broadcast, any remaining exit actions and steps in state transitioning do not occur.
State During	If the state is no longer active at the end of the event broadcast, any remaining steps in executing an active state do not occur.
Condition	If the origin state of the inner or outer flow chart or parent state of the default flow chart is no longer active at the end of the event broadcast, the remaining steps in the execution of the set of flow charts do not occur.
Transition	If the parent of the transition path is not active or if that parent has an active child, the remaining transition actions and state entry do not occur.

Semantic Examples

Categories of Semantic Examples

The following examples show the detailed semantics (behavior) of Stateflow charts.

Transition Between Exclusive States

Label Format for a State-to-State Transition

The following example shows the general label format for a transition entering a state.
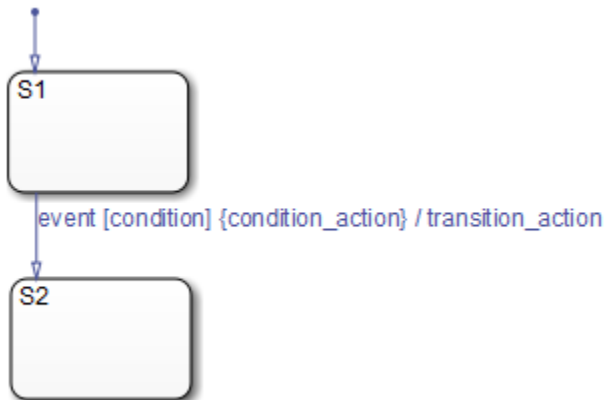
A chart executes this transition as follows:

1   When an event occurs, state S1 checks for an outgoing transition with a matching event specified.

2   If a transition with a matching event is found, the condition for that transition ([condition]) is evaluated.

3   If the condition is true, condition_action is executed.

4   If there is a valid transition to the destination state, the transition is taken.

5   State S1 is exited.

6   The transition_action is executed when the transition is taken.

7   State S2 is entered.

Transition from State to State with Events

The following example shows the behavior of a simple transition focusing on the implications of whether states are active or inactive.
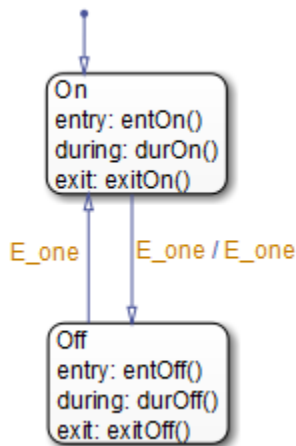
Process a First Event

Initially, the chart is asleep. State `On` and state `Off` are OR states. State `On` is active. Event `E_one` occurs and awakens the chart, which processes the event from the root down through the hierarchy:

1. The chart root checks to see if there is a valid transition as a result of `E_one`. A valid transition from state `On` to state `Off` is detected.
2. State `On` exit actions (`exitOn()`) execute and complete.
3. State `On` is marked inactive.
4. The event `E_one` is broadcast as the transition action.

    This second event `E_one` is processed, but because neither state is active, it has no effect. If the second broadcast of `E_one` resulted in a valid transition, it would preempt the processing of the first broadcast of `E_one`. See "Early Return Logic for Event Broadcasts" on page 3-93.

5. State `Off` is marked active.
6. State `Off` entry actions (`entOff()`) execute and complete.
7. The chart goes back to sleep.

This sequence completes the execution of the Stateflow chart associated with event `E_one` when state `On` is initially active.

Process a Second Event

Using the same example, what happens when the next event, E_one, occurs while state Off is active?
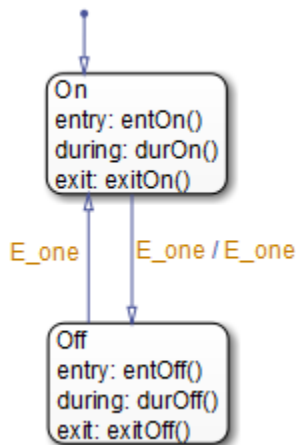
Initially, the chart is asleep. State Off is active. Event E_one occurs and awakens the chart, which processes the event from the root down through the hierarchy:

1   The chart root checks to see if there is a valid transition as a result of E_one.

     A valid transition from state Off to state On is detected.
2   State Off exit actions (exitOff()) execute and complete.
3   State Off is marked inactive.
4   State On is marked active.
5   State On entry actions (entOn()) execute and complete.
6   The chart goes back to sleep.

This sequence completes the execution of the Stateflow chart associated with the second event E_one when state Off is initially active.

Process a Third Event

Using the same example, what happens when a third event, E_two, occurs?
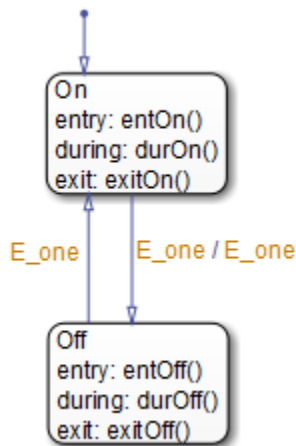
Notice that the event E_two is not used explicitly in this example. However, its occurrence (or the occurrence of any event) does result in behavior. Initially, the chart is asleep and state On is active.

1   Event E_two occurs and awakens the chart.

    Event E_two is processed from the root of the chart down through the hierarchy of the chart.

2   The chart root checks to see if there is a valid transition as a result of E_two. There is none.

3   State On during actions (durOn()) execute and complete.

4   The chart goes back to sleep.

This sequence completes the execution of the Stateflow chart associated with event E_two when state On is initially active.

Tip Avoid using undirected local event broadcasts, which can cause unwanted recursive behavior in your chart. Use the send operator for directed local event broadcasts. For more information, see "Broadcast Local Events to Synchronize Parallel States" on page 12-45.

You can set the diagnostic level for detecting undirected local event broadcasts. In the Model Configuration Parameters dialog box, go to the **Diagnostics > Stateflow** pane and

set the **Undirected event broadcasts** diagnostic to `none`, `warning`, or `error`. The default setting is `warning`.

Transition from a Substate to a Substate with Events

This example shows the behavior of a transition from an OR substate to an OR substate.
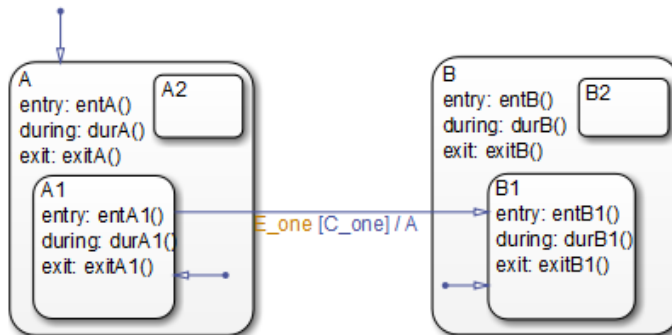
Initially, the chart is asleep. State `A.A1` is active. Condition `C_one` is true. Event `E_one` occurs and awakens the chart, which processes the event from the root down through the hierarchy:

1 The chart root checks to see if there is a valid transition as a result of `E_one`. There is a valid transition from state `A.A1` to state `B.B1`. (Condition `C_one` is true.)

2 State A during actions (`durA()`) execute and complete.

3 State A.A1 exit actions (`exitA1()`) execute and complete.

4 State A.A1 is marked inactive.

5 State A exit actions (`exitA()`) execute and complete.

6 State A is marked inactive.

7 The transition action, A, is executed and completed.

8 State B is marked active.

9 State B entry actions (`entB()`) execute and complete.

10 State B.B1 is marked active.

11 State B.B1 entry actions (`entB1()`) execute and complete.

12 The chart goes back to sleep.

This sequence completes the execution of this Stateflow chart associated with event
E_one.

Control Chart Execution by Using Condition Actions

Condition Action Behavior

This example shows the behavior of a simple condition action in a transition path with multiple segments. The chart uses implicit ordering of outgoing transitions (see "Implicit Ordering" on page 3-65).

Initially, the chart is asleep. State A is active. Conditions C_one and C_two are false. Event E_one occurs and awakens the chart, which processes the event from the root down through the hierarchy:

1   The chart root checks to see if there is a valid transition as a result of E_one. A valid transition segment from state A to a connective junction is detected. The condition action A_one is detected on the valid transition segment and is immediately executed and completed. State A is still active.

2   Because the conditions on the transition segments to possible destinations are false, none of the complete transitions is valid.

3   State A during actions (durA()) execute and complete.

    State A remains active.

4   The chart goes back to sleep.

This sequence completes the execution of this Stateflow chart associated with event E_one when state A is initially active.

Condition and Transition Action Behavior

This example shows the behavior of a simple condition and transition action specified on a transition from one exclusive (OR) state to another.
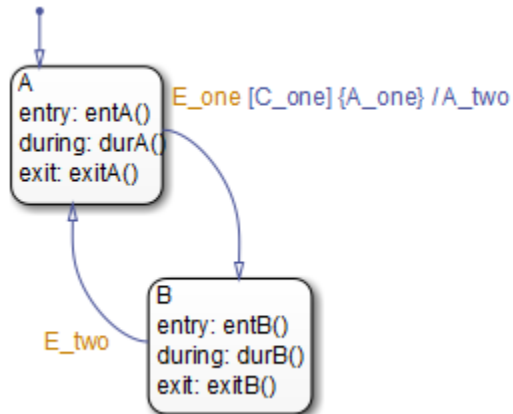
Initially, the chart is asleep. State A is active. Condition C_one is true. Event E_one occurs and awakens the chart, which processes the event from the root down through the hierarchy:

1 The chart root checks to see if there is a valid transition as a result of E_one. A valid transition from state A to state B is detected. The condition C_one is true. The condition action A_one is detected on the valid transition and is immediately executed and completed. State A is still active.

2 State A exit actions (ExitA()) execute and complete.

3 State A is marked inactive.

4 The transition action A_two is executed and completed.

5 State B is marked active.

6 State B entry actions (entB()) execute and complete.

7 The chart goes back to sleep.

This sequence completes the execution of this Stateflow chart associated with event E_one when state A is initially active.

B-11

Create Condition Actions Using a For-Loop

Condition actions and connective junctions are used to design a `for` loop construct. This example shows the use of a condition action and connective junction to create a `for` loop construct. The chart uses implicit ordering of outgoing transitions (see "Implicit Ordering" on page 3-65).
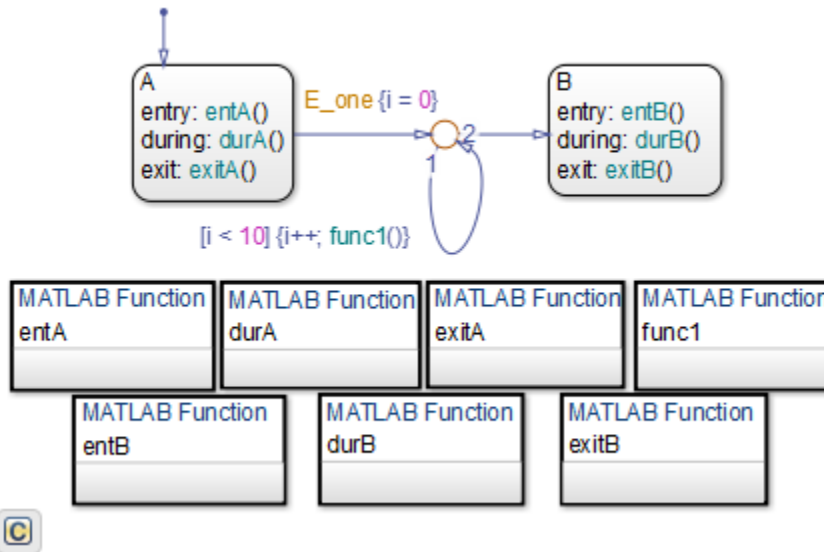
See "For-Loop Construct" on page B-31 to see the behavior of this example.

Broadcast Events to Parallel (AND) States Using Condition Actions

This example shows how to use condition actions to broadcast events immediately to parallel (AND) states. The chart uses implicit ordering of parallel states (see "Implicit Ordering of Parallel States" on page 3-88).
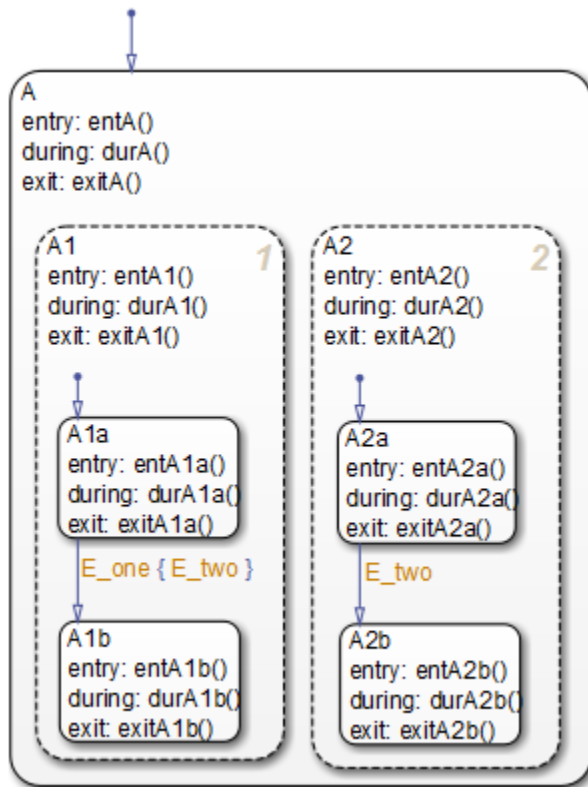
See "Broadcast Events in Condition Actions" on page B-46 to see the behavior of this example.

Tip Avoid using undirected local event broadcasts, which can cause unwanted recursive behavior in your chart. Use the `send` operator for directed local event broadcasts. For more information, see "Broadcast Local Events to Synchronize Parallel States" on page 12-45.

You can set the diagnostic level for detecting undirected local event broadcasts. In the Model Configuration Parameters dialog box, go to the **Diagnostics > Stateflow** pane and set the **Undirected event broadcasts** diagnostic to `none`, `warning`, or `error`. The default setting is `warning`.

Avoid Cyclic Behavior

This example shows a notation to avoid when using event broadcasts as condition actions because the semantics results in cyclic behavior.
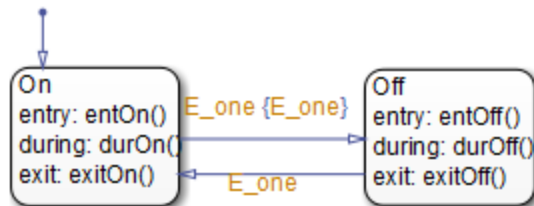
Initially, the chart is asleep. State On is active. Event E_one occurs and awakens the chart, which processes the event from the root down through the hierarchy:
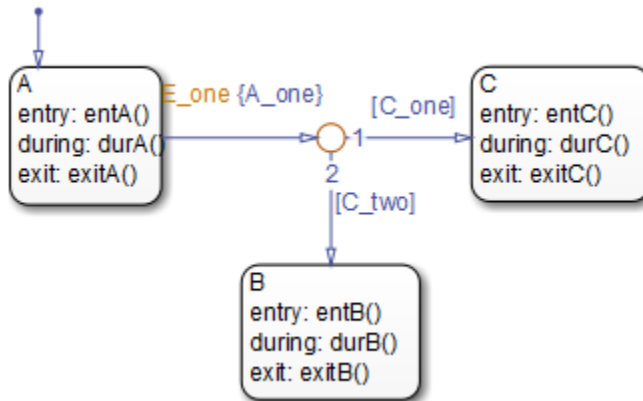
1   The chart root checks to see if there is a valid transition as a result of E_one.

    A valid transition from state On to state Off is detected.

2   The condition action on the transition broadcasts event E_one.

3   Event E_one is detected on the valid transition, which is immediately executed. State On is still active.

4   The broadcast of event E_one awakens the chart a second time.

5   Go to step 1.

Steps 1 through 5 continue to execute in a cyclical manner. The transition label indicating a trigger on the same event as the condition action broadcast event results in unrecoverable cyclic behavior. This sequence never completes when event E_one is broadcast and state On is active.

Tip Avoid using undirected local event broadcasts, which can cause unwanted recursive behavior in your chart. Use the send operator for directed local event broadcasts. For more information, see "Broadcast Local Events to Synchronize Parallel States" on page 12-45.

You can set the diagnostic level for detecting undirected local event broadcasts. In the Model Configuration Parameters dialog box, go to the **Diagnostics > Stateflow** pane and set the **Undirected event broadcasts** diagnostic to none, warning, or error. The default setting is warning.

See Also

More About

- "Transition Action Types" on page 12-7
- "Broadcast Local Events to Synchronize Parallel States" on page 12-45
- "Supported Operations for Chart Data" on page 12-14

Control Chart Execution by Using Default Transitions

Default Transition in Exclusive (OR) Decomposition

This example shows a transition from an OR state to a superstate with exclusive (OR) decomposition, where a default transition to a substate is defined.
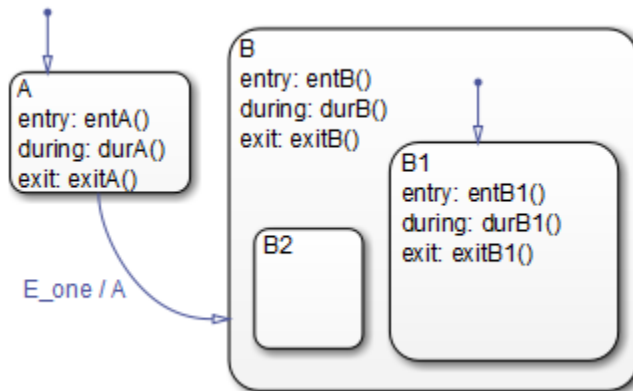

Initially, the chart is asleep. State A is active. Event E_one occurs and awakens the chart, which processes the event from the root down through the hierarchy:

1. The chart root checks to see if there is a valid transition as a result of E_one. There is a valid transition from state A to superstate B.
2. State A exit actions (exitA()) execute and complete.
3. State A is marked inactive.
4. The transition action, A, is executed and completed.
5. State B is marked active.
6. State B entry actions (entB()) execute and complete.
7. State B detects a valid default transition to state B.B1.
8. State B.B1 is marked active.
9. State B.B1 entry actions (entB1()) execute and complete.
10. The chart goes back to sleep.

This sequence completes the execution of this Stateflow chart associated with event E_one when state A is initially active.

Default Transition to a Junction

The following example shows the behavior of a default transition to a connective junction. The default transition to the junction is valid only when state B is first entered, not every time the chart wakes up.
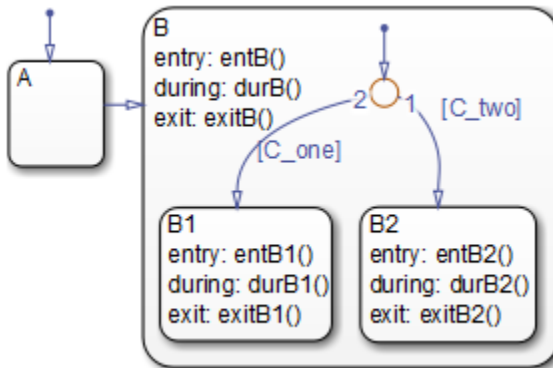
For this example, initially, the chart is asleep. State `B.B1` is active. Condition `[C_two]` is true. An event occurs and awakens the chart, which processes the event from the root down through the hierarchy:

1 State `B` checks to see if there is a valid transition as a result of any event. There is none.

2 State `B during` actions (`durB()`) execute and complete.

3 State `B1` checks to see if there is a valid transition as a result of any event. There is none.

4 State `B1 during` actions (`durB1()`) execute and complete.

This sequence completes the execution of this Stateflow chart associated with the occurrence of any event.

Default Transition and a History Junction

This example shows the behavior of a superstate with a default transition and a history junction. The chart uses implicit ordering of outgoing transitions (see "Implicit Ordering" on page 3-65).

Initially, the chart is asleep. State A is active. A history junction records the fact that state B4 is the previously active substate of superstate B. Event E_one occurs and awakens the chart, which processes the event from the root down through the hierarchy:

1 The chart root checks to see if there is a valid transition as a result of E_one.

There is a valid transition from state A to superstate B.

2 State A exit actions (exitA()) execute and complete.

3 State A is marked inactive.

4 State B is marked active.

5 State B entry actions (entB()) execute and complete.

6 State B uses the history junction to determine the substate destination of the transition into the superstate.

The history junction indicates that substate B.B4 was the last active substate, which becomes the destination of the transition.

7   State B.B4 is marked active.

8   State B.B4 entry actions (`entB4()`) execute and complete.

9   The chart goes back to sleep.

This sequence completes the execution of this Stateflow chart associated with event E_one.

Labeled Default Transitions

This example shows the use of a default transition with a label. The chart uses implicit ordering of outgoing transitions (see "Implicit Ordering" on page 3-65).
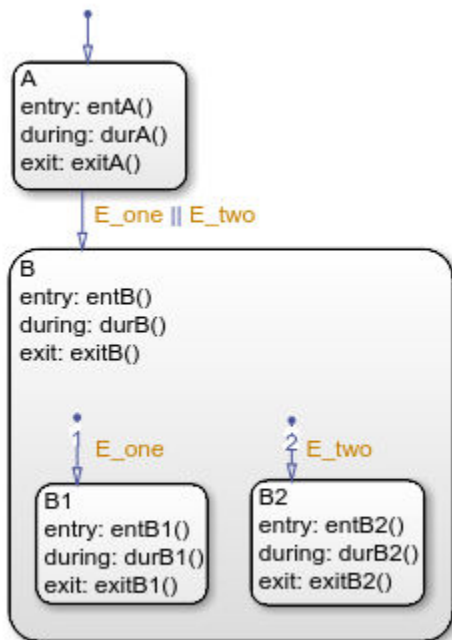
Initially, the chart is asleep. State A is active. Event E_one occurs and awakens the chart, which processes the event from the root down through the hierarchy:

1   The chart root checks to see if there is a valid transition as a result of E_one.

There is a valid transition from state A to superstate B. The transition is valid if event E_one or E_two occurs.

2 State A exit actions execute and complete (`exitA()`).

3 State A is marked inactive.

4 State B is marked active.

5 State B entry actions execute and complete (`entB()`).

6 State B detects a valid default transition to state B.B1. The default transition is valid as a result of E_one.

7 State B.B1 is marked active.

8 State B.B1 entry actions execute and complete (`entB1()`).

9 The chart goes back to sleep.

This sequence completes the execution of this Stateflow chart associated with event E_one when state A is initially active.

Process Events in States Containing Inner Transitions

Process Events with an Inner Transition in an Exclusive (OR) State

This example shows what happens when processing three events using an inner transition in an exclusive (OR) state.

Process One Event in an Exclusive (OR) State

This example shows the behavior of an inner transition. The chart uses implicit ordering of outgoing transitions (see "Implicit Ordering" on page 3-65).
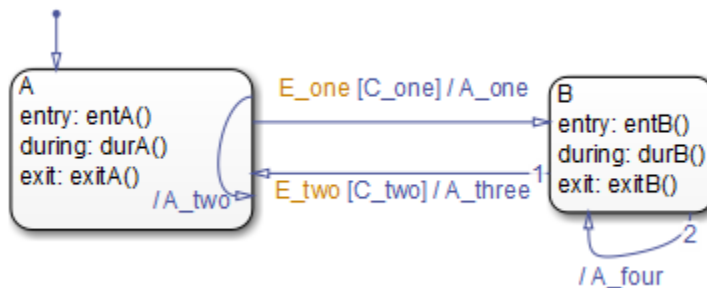
Initially, the chart is asleep. State A is active. Condition [C_one] is false. Event E_one occurs and awakens the chart, which processes the event from the root down through the hierarchy:

1. The chart root checks to see if there is a valid transition as a result of E_one. A potentially valid transition from state A to state B is detected. However, the transition is not valid, because [C_one] is false.

2. State A during actions (durA()) execute and complete.

3. State A checks its children for a valid transition and detects a valid inner transition.

4. State A remains active. The inner transition action A_two is executed and completed. Because it is an inner transition, state A's exit and entry actions are not executed.

5. The chart goes back to sleep.

This sequence completes the execution of this Stateflow chart associated with event E_one.

B-21

Process a Second Event in an Exclusive (OR) State

Using the previous example, this example shows what happens when a second event
E_one occurs. The chart uses implicit ordering of outgoing transitions (see "Implicit
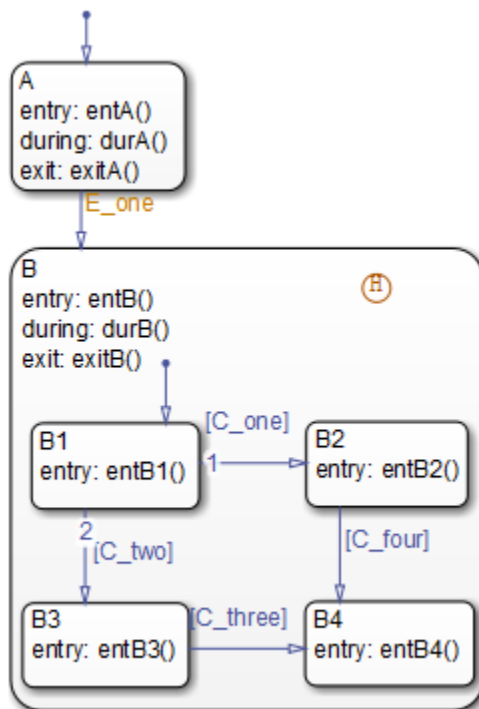Ordering" on page 3-65).
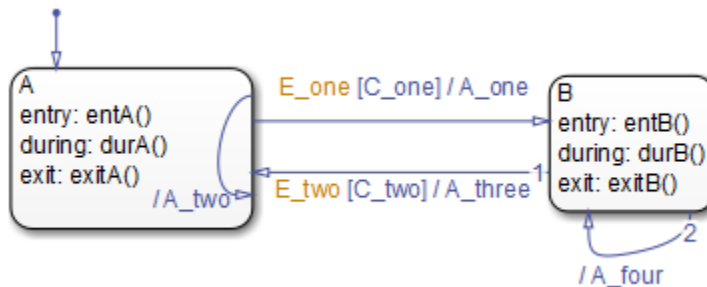
Initially, the chart is asleep. State A is still active. Condition [C_one] is true. Event
E_one occurs and awakens the chart, which processes the event from the root down
through the hierarchy:

1   The chart root checks to see if there is a valid transition as a result of E_one.

     The transition from state A to state B is now valid because [C_one] is true.

2   State A exit actions (exitA()) execute and complete.

3   State A is marked inactive.

4   The transition action A_one is executed and completed.

5   State B is marked active.

6   State B entry actions (entB()) execute and complete.

7   The chart goes back to sleep.

This sequence completes the execution of this Stateflow chart associated with event
E_one.

Process a Third Event in an Exclusive (OR) State

Using the previous example, this example shows what happens when a third event,
E_two, occurs. The chart uses implicit ordering of outgoing transitions (see "Implicit
Ordering" on page 3-65).

Initially, the chart is asleep. State B is now active. Condition [C_two] is false. Event E_two occurs and awakens the chart, which processes the event from the root down through the hierarchy:

1   The chart root checks to see if there is a valid transition as a result of E_two.

A potentially valid transition from state B to state A is detected. The transition is not valid because [C_two] is false. However, active state B has a valid self-loop transition.

2   State B exit actions (exitB()) execute and complete.

3   State B is marked inactive.

4   The self-loop transition action, A_four, executes and completes.

5   State B is marked active.

6   State B entry actions (entB()) execute and complete.

7   The chart goes back to sleep.

This sequence completes the execution of this Stateflow chart associated with event E_two. This example shows the difference in behavior between inner and self-loop transitions.

Process Events with an Inner Transition to a Connective Junction

This example shows the behavior of handling repeated events using an inner transition to a connective junction.

Process the First Event with an Inner Transition to a Connective Junction

This example shows the behavior of an inner transition to a connective junction for the first event. The chart uses implicit ordering of outgoing transitions (see "Implicit Ordering" on page 3-65).
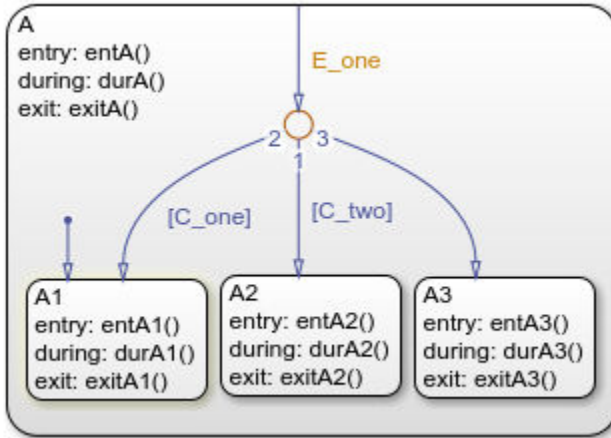
Initially, the chart is asleep. State A1 is active. Condition [C_two] is true. Event E_one occurs and awakens the chart, which processes the event from the root down through the hierarchy:

1   The chart root checks to see if there is a valid transition at the root level as a result of E_one. There is no valid transition.

2   State A during actions (durA()) execute and complete.

3   State A checks itself for valid transitions and detects that there is a valid inner transition to a connective junction.

   The conditions are evaluated to determine whether one of the transitions is valid. Because implicit ordering applies, the segments labeled with a condition are evaluated before the unlabeled segment. The evaluation starts from a 12 o'clock position on the junction and progresses in a clockwise manner. Because [C_two] is true, the inner transition to the junction and then to state A.A2 is valid.

4   State A.A1 exit actions (exitA1()) execute and complete.

5   State A.A1 is marked inactive.

6   State A.A2 is marked active.

7   State A.A2 entry actions (`entA2()`) execute and complete.

8   The chart goes back to sleep.

This sequence completes the execution of this Stateflow chart associated with event `E_one` when state `A1` is active and condition `[C_two]` is true.

Process a Second Event with an Inner Transition to a Connective Junction

Continuing the previous example, this example shows the behavior of an inner transition to a junction when a second event `E_one` occurs. The chart uses implicit ordering of outgoing transitions (see "Implicit Ordering" on page 3-65).
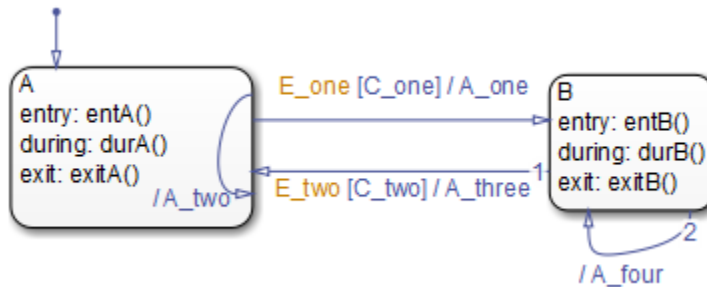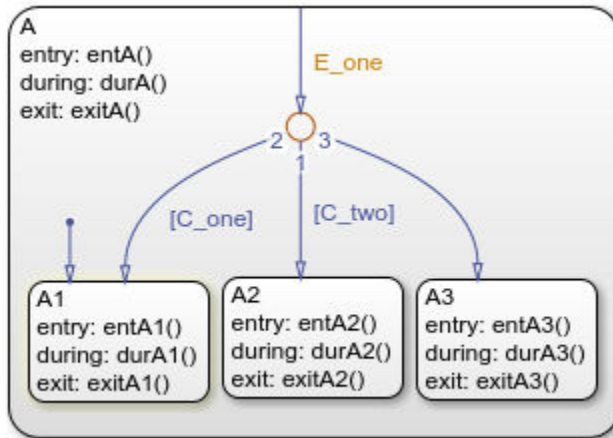

Initially, the chart is asleep. State `A2` is active. Condition `[C_two]` is true. Event `E_one` occurs and awakens the chart, which processes the event from the root down through the hierarchy:

1   The chart root checks to see if there is a valid transition at the root level as a result of `E_one`. There is no valid transition.

2   State A during actions (`durA()`) execute and complete.

3   State A checks itself for valid transitions and detects a valid inner transition to a connective junction.

    The conditions are evaluated to determine whether one of the transitions is valid. Because implicit ordering applies, the segments labeled with a condition are

B-25

evaluated before the unlabeled segment. The evaluation starts from a 12 o'clock position on the junction and progresses in a clockwise manner. Because [C_two] is true, the inner transition to the junction and then to state A.A2 is valid.

4 State A.A2 exit actions (exitA2()) execute and complete.

5 State A.A2 is marked inactive.

6 State A.A2 is marked active.

7 State A.A2 entry actions (entA2()) execute and complete.

8 The chart goes back to sleep.

This sequence completes the execution of this Stateflow chart associated with event E_one when state A2 is active and condition [C_two] is true. For a state with a valid inner transition, an active substate can be exited and reentered immediately.

Inner Transition to a History Junction

This example shows the behavior of an inner transition to a history junction.
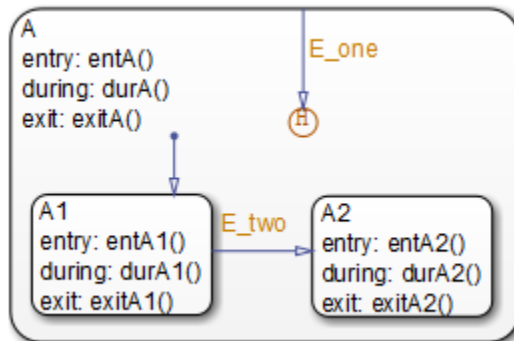
Initially, the chart is asleep. State A.A1 is active. History information exists because superstate A is active. Event E_one occurs and awakens the chart, which processes the event from the root down through the hierarchy:

1 The chart root checks to see if there is a valid transition as a result of E_one. There is no valid transition.

2 State A during actions execute and complete.

3 State A checks itself for valid transitions and detects that there is a valid inner transition to a history junction. Based on the history information, the last active state, `A.A1`, is the destination state.

4 State `A.A1` exit actions execute and complete.

5 State `A.A1` is marked inactive.

6 State `A.A1` is marked active.

7 State `A.A1` entry actions execute and complete.

8 The chart goes back to sleep.

This sequence completes the execution of this Stateflow chart associated with event `E_one` when there is an inner transition to a history junction and state `A.A1` is active. For a state with a valid inner transition, an active substate can be exited and reentered immediately.

B-27

Represent Multiple Paths by Using Connective Junctions

Label Format for Transition Segments

The label format for a transition segment entering a junction is the same as for transitions entering states, as shown in the following example. The chart uses implicit ordering of outgoing transitions (see "Implicit Ordering" on page 3-65).
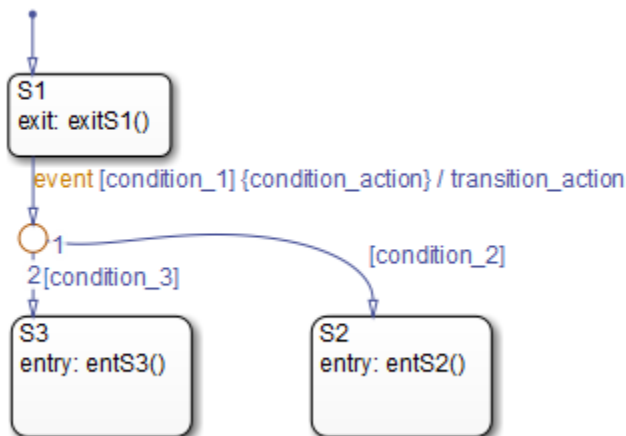
Execution of a transition in this example occurs as follows:

1 When an event occurs, state S1 is checked for an outgoing transition with a matching event specified.

2 If a transition with a matching event is found, the transition condition for that transition (in brackets) is evaluated.

3 If condition_1 evaluates to true, the condition action condition_action (in braces) is executed.

4 The outgoing transitions from the junction are checked for a valid transition. Since condition_2 is true, a valid state-to-state transition from S1 to S2 exists.

5 State S1 exit actions execute and complete.

6 State S1 is marked inactive.

7 The transition action transition_action executes and completes.

8 The completed state-to-state transition from S1 to S2 occurs.

9   State S2 is marked active.

10   State S2 entry actions execute and complete.

If-Then-Else Decision Construct

This example shows the behavior of an `if-then-else` decision construct. The chart uses implicit ordering of outgoing transitions (see "Implicit Ordering" on page 3-65).

Initially, the chart is asleep. State A is active. Condition `[C_two]` is true. Event `E_one` occurs and awakens the chart, which processes the event from the root down through the hierarchy:

1   The chart root checks to see if there is a valid transition as a result of `E_one`.

    A valid transition segment from state A to the connective junction exists. Because implicit ordering applies, the transition segments beginning from a 12 o'clock position on the connective junction are evaluated for validity. The first transition segment, labeled with condition `[C_one]`, is not valid. The next transition segment, labeled with the condition `[C_two]`, is valid. The complete transition from state A to state C is valid.

2   State A exit actions (`exitA()`) execute and complete.

3   State A is marked inactive.

4   State C is marked active.

5   State C entry actions (`entC()`) execute and complete.

6   The chart goes back to sleep.

This sequence completes the execution of this Stateflow chart associated with event E_one.

Self-Loop Transition

This example shows the behavior of a self-loop transition using a connective junction. The chart uses implicit ordering of outgoing transitions (see "Implicit Ordering" on page 3-65).

Initially, the chart is asleep. State A is active. Condition [C_one] is false. Event E_one occurs and awakens the chart, which processes the event from the root down through the hierarchy:

1   The chart root checks to see if there is a valid transition as a result of E_one. A valid transition segment from state A to the connective junction exists. Because implicit ordering applies, the transition segment labeled with a condition is evaluated for validity. Because the condition [C_one] is not valid, the complete transition from state A to state B is not valid. The transition segment from the connective junction back to state A is valid.

2   State A exit actions (exitA()) execute and complete.

3   State A is marked inactive.

4 The transition action `A_two` is executed and completed.

5 State A is marked active.

6 State A entry actions (`entA()`) execute and complete.

7 The chart goes back to sleep.

This sequence completes the execution of this Stateflow chart associated with event `E_one`.

For-Loop Construct

This example shows the behavior of a `for` loop using a connective junction. The chart uses implicit ordering of outgoing transitions (see "Implicit Ordering" on page 3-65).
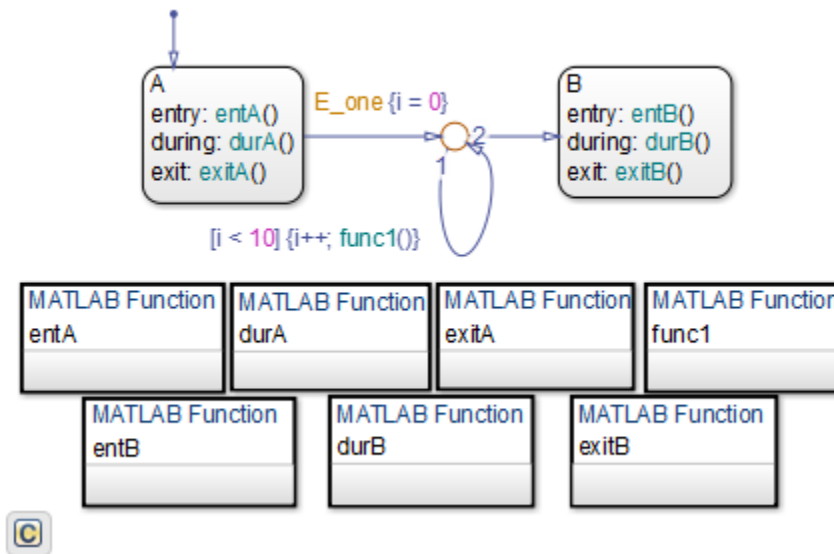
Initially, the chart is asleep. State A is active. Event `E_one` occurs and awakens the chart, which processes the event from the root down through the hierarchy:

1 The chart root checks to see if there is a valid transition as a result of `E_one`. There is a valid transition segment from state A to the connective junction. The transition segment condition action, `i = 0`, executes and completes. Of the two transition

segments leaving the connective junction, the transition segment that is a self-loop back to the connective junction evaluates next for validity. That segment takes priority in evaluation because it has a condition, whereas the other segment is unlabeled. This evaluation behavior reflects implicit ordering of outgoing transitions in the chart.

2 The condition `[i < 10]` evaluates as true. The condition actions `i++` and a call to `func1` execute and complete until the condition becomes false. Because a connective junction is not a final destination, the transition destination is still unknown.

3 The unconditional segment to state B is now valid. The complete transition from state A to state B is valid.

4 State A exit actions (`exitA()`) execute and complete.

5 State A is marked inactive.

6 State B is marked active.

7 State B entry actions (`entB()`) execute and complete.

8 The chart goes back to sleep.

This sequence completes the execution of this chart associated with event `E_one`.

Flow Chart Notation

This example shows the behavior of a Stateflow chart that uses flow chart notation. The chart uses implicit ordering of outgoing transitions (see "Implicit Ordering" on page 3-65).
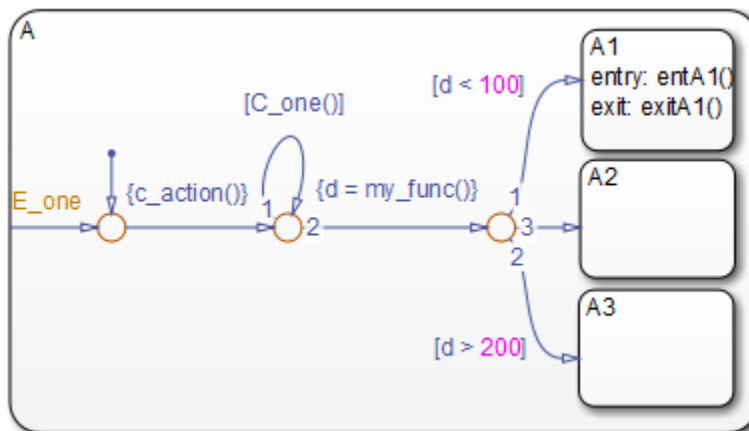
Initially, the chart is asleep. State `A.A1` is active. The condition `[C_one()]` is initially true. Event `E_one` occurs and awakens the chart, which processes the event from the root down through the hierarchy:

1 The chart root checks to see if there is a valid transition as a result of `E_one`. There is no valid transition.

2 State `A` checks itself for valid transitions and detects a valid inner transition to a connective junction.

3 The next possible segments of the transition are evaluated. Only one outgoing transition exists, and it has a condition action defined. The condition action executes and completes.

4 The next possible segments are evaluated. Two outgoing transitions exist: a conditional self-loop transition and an unconditional transition segment. Because implicit ordering applies, the conditional transition segment takes precedence. Since the condition `[C_one()]` is true, the self-loop transition is taken. Since a final transition destination has not been reached, this self-loop continues until `[C_one()]` is false.

   Assume that after five iterations, `[C_one()]` is false.

5 The next possible transition segment (to the next connective junction) is evaluated. It is an unconditional transition segment with a condition action. The transition segment is taken and the condition action, `{d=my_func()}`, executes and completes. The returned value of `d` is 84.

6 The next possible transition segment is evaluated. Three outgoing transition segments exist: two conditional and one unconditional. Because implicit ordering applies, the segment labeled with the condition `[d < 100]` evaluates first based on the geometry of the two outgoing conditional transition segments. Because the returned value of `d` is 84, the condition `[d < 100]` is true and this transition to the destination state `A.A1` is valid.

7 State `A.A1` exit actions (`exitA1()`) execute and complete.

8 State `A.A1` is marked inactive.

9 State `A.A1` is marked active.

10 State `A.A1` entry actions (`entA1()`) execute and complete.

11 The chart goes back to sleep.

This sequence completes the execution of this Stateflow chart associated with event `E_one`.

B-33

Transition from a Common Source to Multiple Destinations

This example shows the behavior of transitions from a common source to multiple conditional destinations using a connective junction. The chart uses implicit ordering of outgoing transitions (see "Implicit Ordering" on page 3-65).
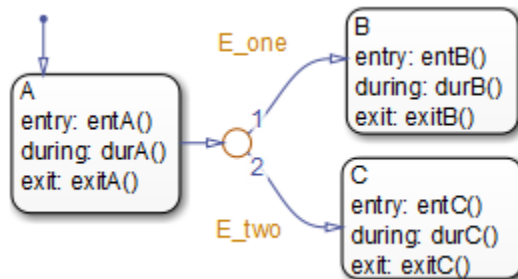
Initially, the chart is asleep. State A is active. Event E_two occurs and awakens the chart, which processes the event from the root down through the hierarchy:

1   The chart root checks to see if there is a valid transition as a result of E_two. A valid transition segment exists from state A to the connective junction. Because implicit ordering applies, evaluation of segments with equivalent label priority begins from a 12 o'clock position on the connective junction and progresses clockwise. The first transition segment, labeled with event E_one, is not valid. The next transition segment, labeled with event E_two, is valid. The complete transition from state A to state C is valid.

2   State A exit actions (exitA()) execute and complete.

3   State A is marked inactive.

4   State C is marked active.

5   State C entry actions (entC()) execute and complete.

6   The chart goes back to sleep.

This sequence completes the execution of this Stateflow chart associated with event E_two.

Resolve Equally Valid Transition Paths

What Are Conflicting Transitions?

Conflicting transitions are two equally valid paths from the same source in a Stateflow chart during simulation. In the case of a conflict, the chart evaluates equally valid transitions based on ordering mode in the chart: explicit or implicit.

- For explicit ordering (the default mode), evaluation of conflicting transitions occurs based on the order you specify for each transition. For details, see "Explicit Ordering" on page 3-65.
- For implicit ordering in C charts, evaluation of conflicting transitions occurs based on internal rules described in "Implicit Ordering" on page 3-65.

Example of Conflicting Transitions

The following chart has two equally valid transition paths:
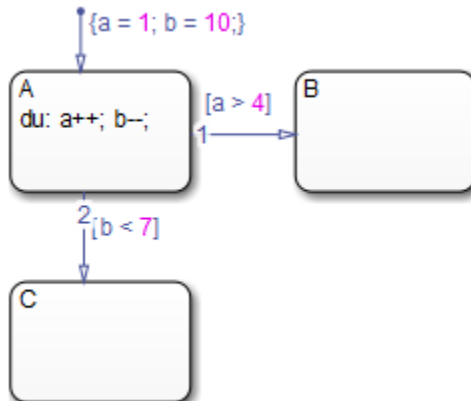
Conflict Resolution for Implicit Ordering

For implicit ordering, the chart evaluates multiple outgoing transitions with equal label priority in a clockwise progression starting from the twelve o'clock position on the state. In this case, the transition from state A to state B occurs.

Conflict Resolution for Explicit Ordering

For explicit ordering, the chart resolves the conflict by evaluating outgoing transitions in the order that you specify explicitly. For example, if you right-click the transition from

B-35

state A to state C and select **Execution Order > 1** from the context menu, the chart evaluates that transition first. In this case, the transition from state A to state C occurs.

How the Transition Conflict Occurs

The default transition to state A assigns data a equal to 1 and data b equal to 10. The during action of state A increments a and decrements b during each time step. The transition from state A to state B is valid if the condition [a > 4] is true. The transition from state A to state C is valid if the condition [b < 7] is true. During simulation, there is a time step where state A is active and both conditions are true. This issue is a transition conflict.

Transition from Multiple Sources to a Common Destination

This example shows the behavior of transitions from multiple sources to a single destination using a connective junction.
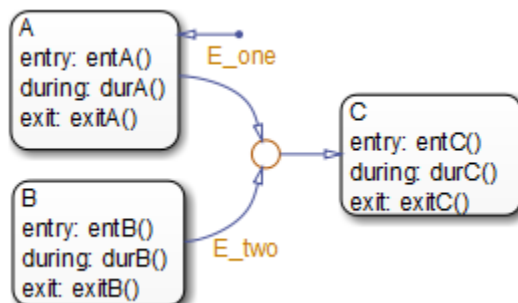
Initially, the chart is asleep. State A is active. Event E_one occurs and awakens the chart, which processes the event from the root down through the hierarchy:

1   The chart root checks to see if there is a valid transition as a result of E_one. A valid transition segment exists from state A to the connective junction and from the junction to state C.

2   State A exit actions (exitA()) execute and complete.

3   State A is marked inactive.

4   State C is marked active.

5   State C entry actions (entC()) execute and complete.

6   The chart goes back to sleep.

This sequence completes the execution of this Stateflow chart associated with event `E_one`.

Transition from a Source to a Destination Based on a Common Event

This example shows the behavior of transitions from multiple sources to a single destination based on the same event using a connective junction.

Initially, the chart is asleep. State `B` is active. Event `E_one` occurs and awakens the chart, which processes the event from the root down through the hierarchy:

1   The chart root checks to see if there is a valid transition as a result of `E_one`. A valid transition segment exists from state `B` to the connective junction and from the junction to state C.

2   State `B` exit actions (`exitB()`) execute and complete.

3   State `B` is marked inactive.

4   State `C` is marked active.

5   State `C` entry actions (`entC()`) execute and complete.

6   The chart goes back to sleep.

This sequence completes the execution of this Stateflow chart associated with event `E_one`.

Backtrack in Flow Charts

This example shows the behavior of transitions with junctions that force backtracking behavior in flow charts. The chart uses implicit ordering of outgoing transitions (see "Implicit Ordering" on page 3-65).
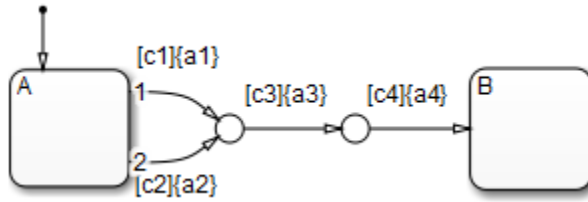
Initially, state A is active and conditions c1, c2, and c3 are true:

1   The chart root checks to see if there is a valid transition from state A.

There is a valid transition segment marked with the condition c1 from state A to a connective junction.

2   Condition c1 is true and action a1 executes.

3   Condition c3 is true and action a3 executes.

4   Condition c4 is not true and control flow backtracks to state A.

5   The chart root checks to see if there is another valid transition from state A.

There is a valid transition segment marked with the condition c2 from state A to a connective junction.

6   Condition c2 is true and action a2 executes.

7   Condition c3 is true and action a3 executes.

8   Condition c4 is not true and control flow backtracks to state A.

9   The chart goes to sleep.

The preceding example shows the unexpected behavior of executing both actions a1 and a2. Another unexpected behavior is the execution of action a3 twice. To resolve this problem, consider adding unconditional transitions to terminating junctions.
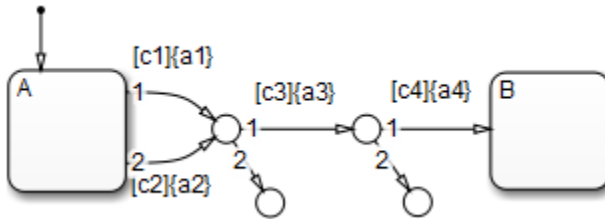
The terminating junctions allow flow to end if either c3 or c4 is not true. This design leaves state A active without executing unnecessary actions.

Control Chart Execution by Using Event Actions in a Superstate

The following example shows the use of event actions in a superstate.
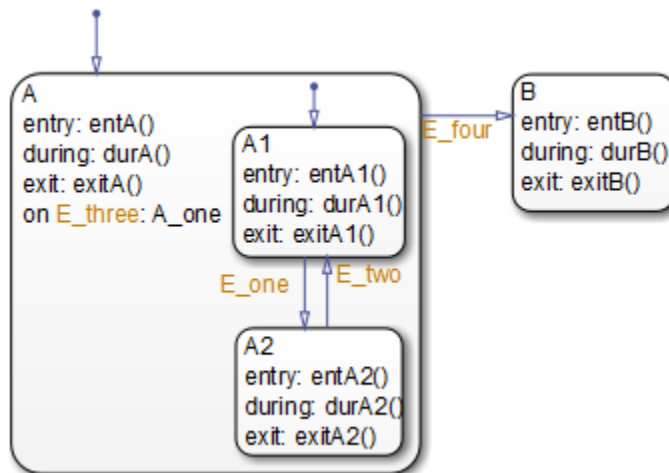
Initially, the chart is asleep. State A.A1 is active. Event E_three occurs and awakens the chart, which processes the event from the root down through the hierarchy:
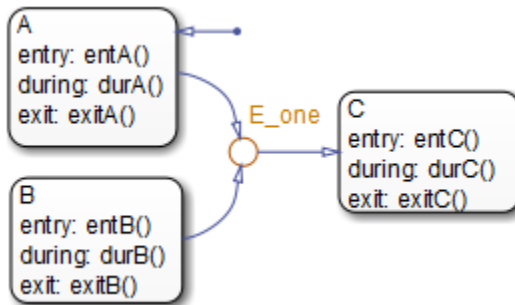
1   The chart root checks to see if there is a valid transition as a result of E_three. No valid transition exists.

2   State A during actions (durA()) execute and complete.

3   State A executes and completes the on event E_three action (A_one).

4   State A checks its children for valid transitions. No valid transitions exist.

5   State A1 during actions (durA1()) execute and complete.

6   The chart goes back to sleep.

This sequence completes the execution of this Stateflow chart associated with event E_three.

Undirected Broadcast Events in Parallel States

Broadcast Events in State Actions

This example shows the behavior of event broadcast actions in parallel states. The chart uses implicit ordering of parallel states (see "Implicit Ordering of Parallel States" on page 3-88).

Initially, the chart is asleep. Parallel substates A.A1.A1a and A.A2.A2a are active. Event E_one occurs and awakens the chart, which processes the event from the root down through the hierarchy:

1. The chart root checks to see if there is a valid transition at the root level as a result of E_one. No valid transition exists.

2. State A `during` actions (`durA()`) execute and complete.

3. The children of state A are parallel (AND) states. Because implicit ordering applies, the states are evaluated and executed from left to right and top to bottom. State A.A1 is evaluated first. State A.A1 `during` actions (`durA1()`) execute and complete. State A.A1 executes and completes the `on E_one` action and broadcasts event E_two. The `during` and on *event_name* actions are processed based on their order of appearance in the state label:

   a. The broadcast of event E_two awakens the chart a second time. The chart root checks to see if there is a valid transition as a result of E_two. No valid transition exists.

   b. State A `during` actions (`durA()`) execute and complete.

   c. State A checks its children for valid transitions. No valid transitions exist.

   d. State A's children are evaluated starting with state A.A1. State A.A1 `during` actions (`durA1()`) execute and complete. State A.A1 is evaluated for valid transitions. There are no valid transitions as a result of E_two within state A1.

   e. State A1a's `during` actions (`durA1a()`) execute.

   f. State A.A2 is evaluated. State A.A2 `during` actions (`durA2()`) execute and complete. State A.A2 checks for valid transitions. State A.A2 has a valid transition as a result of E_two from state A.A2.A2a to state A.A2.A2b.

   g. State A.A2.A2a `exit` actions (`exitA2a()`) execute and complete.

   h. State A.A2.A2a is marked inactive.

   i. State A.A2.A2b is marked active.

   j. State A.A2.A2b `entry` actions (`entA2b()`) execute and complete.

4. The processing of E_one continues once the on event broadcast of E_two has been processed. State A.A1 checks for any valid transitions as a result of event E_one. A valid transition exists from state A.A1.A1a to state A.A1.A1b.

5. State A.A1.A1a executes and completes `exit` actions (`exitA1a`).

6. State A.A1.A1a is marked inactive.

7. State A.A1.A1b is marked active.

8. State A.A1.A1b `entry` actions (`entA1b()`) execute and complete.

9. Parallel state A.A2 is evaluated next. State A.A2 `during` actions (`durA2()`) execute and complete. There are no valid transitions as a result of E_one.

10 State `A.A2.A2b during` actions (`durA2b()`) execute and complete.

State `A.A2.A2b` is now active as a result of the processing of the `on` event broadcast of `E_two`.

11 The chart goes back to sleep.

This sequence completes the execution of this Stateflow chart associated with event `E_one` and the `on` event broadcast to a parallel state of event `E_two`. The final chart activity is that parallel substates `A.A1.A1b` and `A.A2.A2b` are active.

Tip Avoid using undirected local event broadcasts, which can cause unwanted recursive behavior in your chart. Use the `send` operator for directed local event broadcasts. For more information, see "Broadcast Local Events to Synchronize Parallel States" on page 12-45.

You can set the diagnostic level for detecting undirected local event broadcasts. In the Model Configuration Parameters dialog box, go to the **Diagnostics > Stateflow** pane and set the **Undirected event broadcasts** diagnostic to `none`, `warning`, or `error`. The default setting is `warning`.

Broadcast Events in Transition Actions

This example shows the behavior of an event broadcast transition action that includes a nested event broadcast in a parallel state. The chart uses implicit ordering of parallel states (see "Implicit Ordering of Parallel States" on page 3-88).
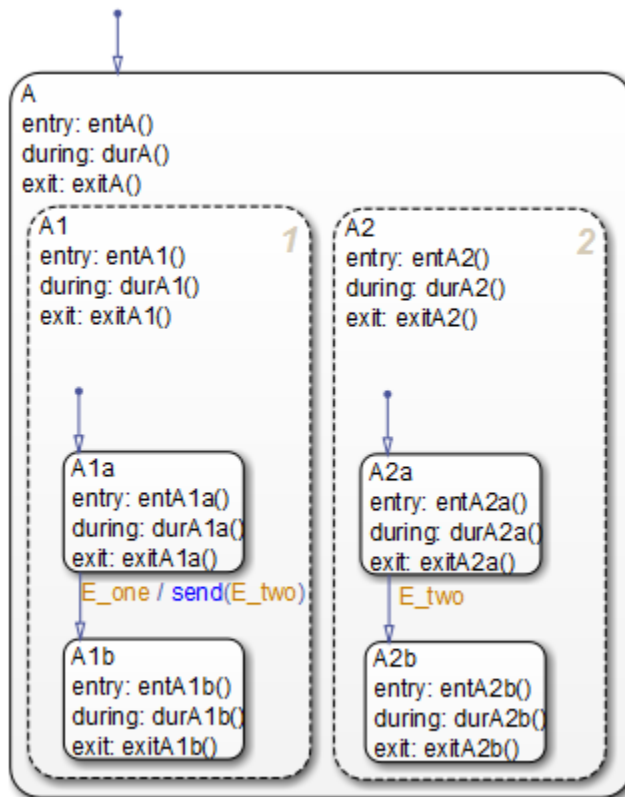
Start of Event E_one Processing

Initially, the chart is asleep. Parallel substates `A.A1.A1a` and `A.A2.A2a` are active. Event `E_one` occurs and awakens the chart, which processes the event from the root down through the hierarchy:
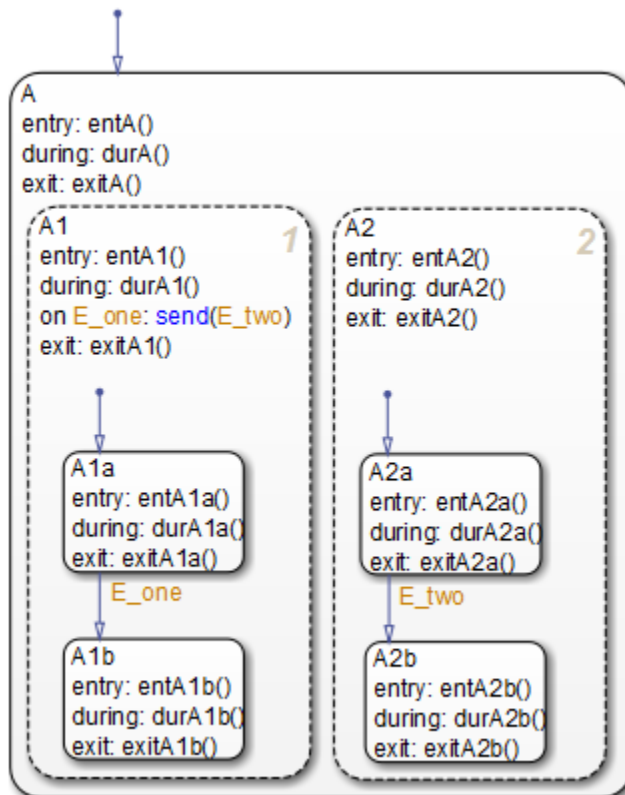
1 The chart root checks to see if there is a valid transition as a result of `E_one`. There is no valid transition.

2 State `A during` actions (`durA()`) execute and complete.

3 State A's children are parallel (AND) states. Because implicit ordering applies, the states are evaluated and executed from left to right and top to bottom. State `A.A1` is evaluated first. State `A.A1during` actions (`durA1()`) execute and complete.

4 State `A.A1` checks for any valid transitions as a result of event `E_one`. There is a valid transition from state `A.A1.A1a` to state `A.A1.A1b`.

5 State `A.A1.A1a` executes and completes `exit` actions (`exitA1a`).

6 State `A.A1.A1a` is marked inactive.

Event E_two Preempts E_one

1 The transition action that broadcasts event `E_two` executes and completes:

    a The broadcast of event `E_two` now preempts the transition from state `A1a` to state `A1b` that event `E_one` triggers.

    b The broadcast of event `E_two` awakens the chart a second time. The chart root checks to see if there is a valid transition as a result of `E_two`. No valid transition exists.

    c State A `during` actions (`durA()`) execute and complete.

    d State A's children are evaluated starting with state `A.A1`. State `A.A1during` actions (`durA1()`) execute and complete. State `A.A1` is evaluated for valid transitions. There are no valid transitions as a result of `E_two` within state `A1`.

    e State `A.A2` is evaluated. State `A.A2` `during` actions (`durA2()`) execute and complete. State `A.A2` checks for valid transitions. State `A.A2` has a valid transition as a result of `E_two` from state `A.A2.A2a` to state `A.A2.A2b`.

    f State `A.A2.A2a` `exit` actions (`exitA2a()`) execute and complete.

    g State `A.A2.A2a` is marked inactive.

    h State `A.A2.A2b` is marked active.

    i State `A.A2.A2b` `entry` actions (`entA2b()`) execute and complete.

Event E_one Processing Resumes

1 State `A.A1.A1b` is marked active.

2 State `A.A1.A1b` entry actions (`entA1b()`) execute and complete.

3 Parallel state `A.A2` is evaluated next. State `A.A2` during actions (`durA2()`) execute and complete. There are no valid transitions as a result of `E_one`.

4 State `A.A2.A2b` during actions (`durA2b()`) execute and complete.

    State `A.A2.A2b` is now active as a result of the processing of event broadcast `E_two`.

5 The chart goes back to sleep.

This sequence completes the execution of this Stateflow chart associated with event `E_one` and the event broadcast on a transition action to a parallel state of event `E_two`. The final chart activity is that parallel substates `A.A1.A1b` and `A.A2.A2b` are active.

Tip Avoid using undirected local event broadcasts, which can cause unwanted recursive behavior in your chart. Use the `send` operator for directed local event broadcasts. For more information, see "Broadcast Local Events to Synchronize Parallel States" on page 12-45.

You can set the diagnostic level for detecting undirected local event broadcasts. In the Model Configuration Parameters dialog box, go to the **Diagnostics > Stateflow** pane and set the **Undirected event broadcasts** diagnostic to `none`, `warning`, or `error`. The default setting is `warning`.

Broadcast Events in Condition Actions

This example shows the behavior of a condition action event broadcast in a parallel (AND) state. The chart uses implicit ordering of parallel states (see "Implicit Ordering of Parallel States" on page 3-88).

Initially, the chart is asleep. Parallel substates `A.A1.A1a` and `A.A2.A2a` are active. Event `E_one` occurs and awakens the chart, which processes the event from the root down through the hierarchy:

1. The chart root checks to see if there is a valid transition as a result of `E_one`. No valid transition exists.

2. State `A` `during` actions (`durA()`) execute and complete.

3. State A's children are parallel (AND) states. Because implicit ordering applies, the states are evaluated and executed from top to bottom, and from left to right. State `A.A1` is evaluated first. State `A.A1` `during` actions (`durA1()`) execute and complete.

4. State `A.A1` checks for any valid transitions as a result of event `E_one`. A valid transition from state `A.A1.A1a` to state `A.A1.A1b` exists. A valid condition action

also exists. The condition action event broadcast of `E_two` executes and completes. State `A.A1.A1a` is still active:

a   The broadcast of event `E_two` awakens the Stateflow chart a second time. The chart root checks to see if there is a valid transition as a result of `E_two`. There is no valid transition.

b   State `A` `during` actions (`durA()`) execute and complete.

c   State A's children are evaluated starting with state `A.A1`. State `A.A1` `during` actions (`durA1()`) execute and complete. State `A.A1` is evaluated for valid transitions. There are no valid transitions as a result of `E_two` within state `A1`.

d   State `A1a` `during` actions (`durA1a()`) execute.

e   State `A.A2` is evaluated. State `A.A2` `during` actions (`durA2()`) execute and complete. State `A.A2` checks for valid transitions. State `A.A2` has a valid transition as a result of `E_two` from state `A.A2.A2a` to state `A.A2.A2b`.

f   State `A.A2.A2a` `exit` actions (`exitA2a()`) execute and complete.

g   State `A.A2.A2a` is marked inactive.

h   State `A.A2.A2b` is marked active.

i   State `A.A2.A2b` `entry` actions (`entA2b()`) execute and complete.

5   State `A.A1.A1a` executes and completes `exit` actions (`exitA1a`).

6   State `A.A1.A1a` is marked inactive.

7   State `A.A1.A1b` is marked active.

8   State `A.A1.A1b` `entry` actions (`entA1b()`) execute and complete.

9   Parallel state `A.A2` is evaluated next. State `A.A2` `during` actions (`durA2()`) execute and complete. There are no valid transitions as a result of `E_one`.

10   State `A.A2.A2b` `during` actions (`durA2b()`) execute and complete.

State `A.A2.A2b` is now active as a result of the processing of the condition action event broadcast of `E_two`.

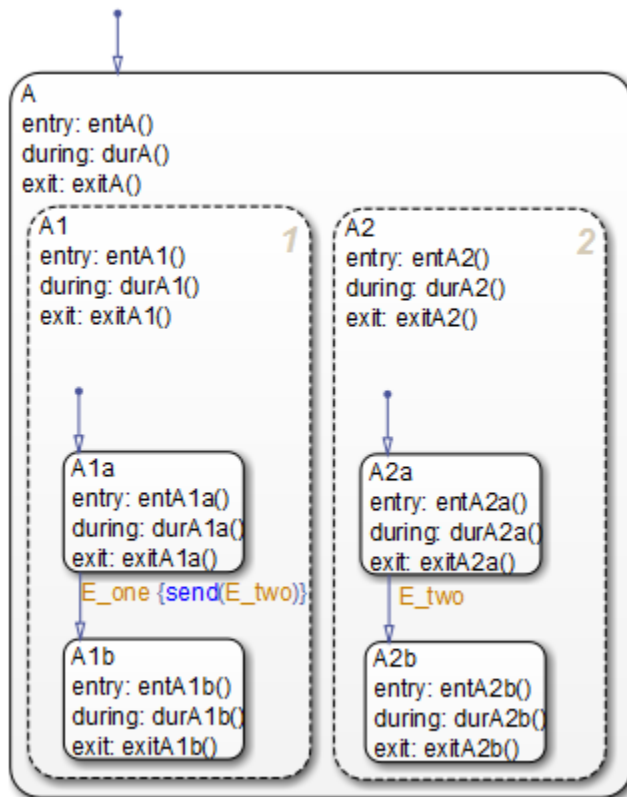**11**   The chart goes back to sleep.

This sequence completes the execution of this Stateflow chart associated with event `E_one` and the event broadcast on a condition action to a parallel state of event `E_two`. The final chart activity is that parallel substates `A.A1.A1b` and `A.A2.A2b` are active.

Tip Avoid using undirected local event broadcasts, which can cause unwanted recursive behavior in your chart. Use the `send` operator for directed local event broadcasts. For

more information, see "Broadcast Local Events to Synchronize Parallel States" on page 12-45.

You can set the diagnostic level for detecting undirected local event broadcasts. In the Model Configuration Parameters dialog box, go to the **Diagnostics > Stateflow** pane and set the **Undirected event broadcasts** diagnostic to `none`, `warning`, or `error`. The default setting is `warning`.

Broadcast Local Events in Parallel States

Directed Event Broadcast Using Send

This example shows the behavior of directed event broadcast using the
send(*event_name*,*state_name*) syntax on a transition. The chart uses implicit
ordering of parallel states (see "Implicit Ordering of Parallel States" on page 3-88).
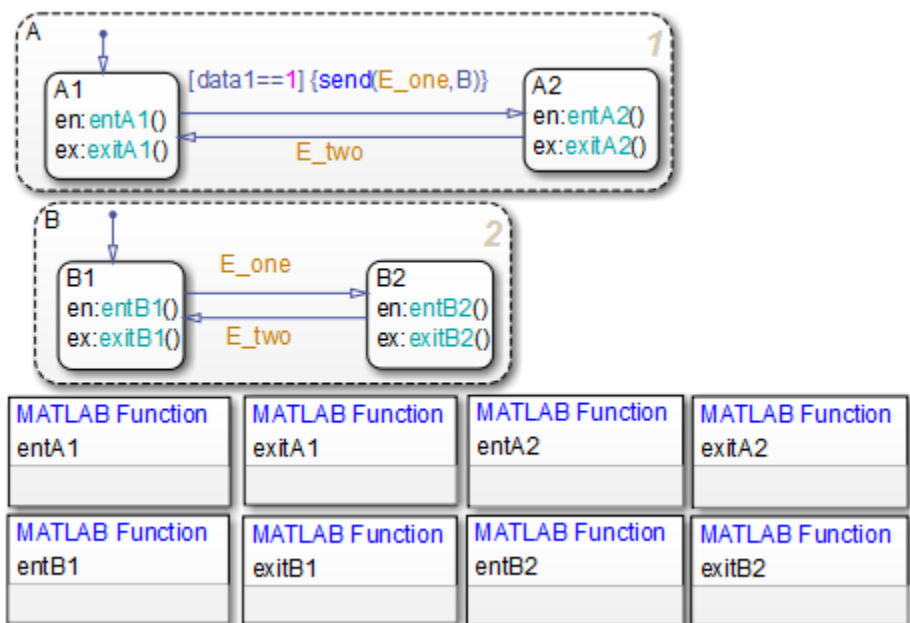
Initially, the chart is asleep. Parallel substates A.A1 and B.B1 are active, which implies
that parallel (AND) superstates A and B are also active. The condition [data1==1] is
true. The event E_one belongs to the chart and is visible to both A and B.

After waking up, the chart checks for valid transitions at every level of the hierarchy:

1   The chart root checks to see if there is a valid transition as a result of the event.
     There is no valid transition.

2   State A checks for any valid transitions as a result of the event. Because the condition
     [data1==1] is true, there is a valid transition from state A.A1 to state A.A2.

3 The action `send(E_one,B)` executes:

   a The broadcast of event `E_one` reaches state B. Because state B is active, that state receives the event broadcast and checks to see if there is a valid transition. There is a valid transition from `B.B1` to `B.B2`.

   b State `B.B1` `exit` actions (`exitB1()`) execute and complete.

   c State `B.B1` becomes inactive.

   d State `B.B2` becomes active.

   e State `B.B2` `entry` actions (`entB2()`) execute and complete.

4 State `A.A1` `exit` actions (`exitA1()`) execute and complete.

5 State `A.A1` becomes inactive.

6 State `A.A2` becomes active.

7 State `A.A2` `entry` actions (`entA2()`) execute and complete.

This sequence completes execution of a chart with a directed event broadcast to a parallel state.

Directed Event Broadcast Using Qualified Event Name

This example shows the behavior of directed event broadcast using a qualified event name on a transition. The chart uses implicit ordering of parallel states (see "Implicit Ordering of Parallel States" on page 3-88).

B-51

The only differences from the chart in "Directed Event Broadcast Using Send" on page B-50 are:

- The event E_one belongs to state B and is visible only to that state.
- The action send(E_one,B) is now send(B.E_one).

  Using a qualified event name is necessary because E_one is not visible to state A.

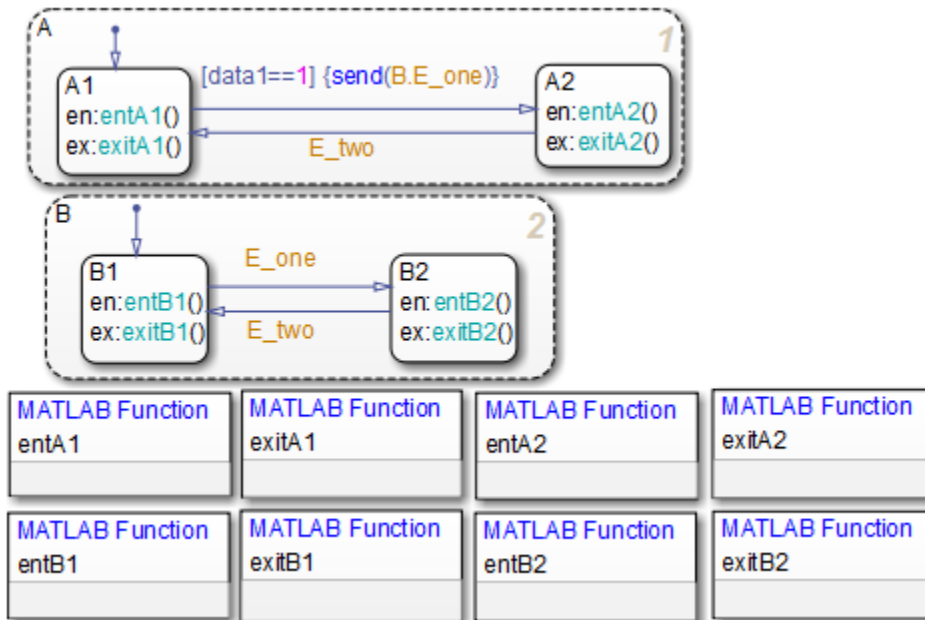After waking up, the chart checks for valid transitions at every level of the hierarchy:

1 The chart root checks to see if there is a valid transition as a result of the event. There is no valid transition.

2 State A checks for any valid transitions as a result of the event. Because the condition [data1==1] is true, there is a valid transition from state A.A1 to state A.A2.

3 The action send(B.E_one) executes and completes:

   a The broadcast of event E_one reaches state B. Because state B is active, that state receives the event broadcast and checks to see if there is a valid transition. There is a valid transition from B.B1 to B.B2.

    b   State `B.B1` `exit` actions (`exitB1()`) execute and complete.

    c   State `B.B1` becomes inactive.

    d   State `B.B2` becomes active.

    e   State `B.B2` `entry` actions (`entB2()`) execute and complete.

4   State `A.A1` `exit` actions (`exitA1()`) execute and complete.

5   State `A.A1` becomes inactive.

6   State `A.A2` becomes active.

7   State `A.A2` `entry` actions (`entA2()`) execute and complete.

This sequence completes execution of a chart with a directed event broadcast using a qualified event name to a parallel state.

See Also

send

More About

- "Broadcast Local Events to Synchronize Parallel States" on page 12-45

Glossary

actions	*Actions* take place as part of Stateflow chart execution. The action can execute as part of a transition from one state to another, or depending on the activity status of a state. Transitions can contain condition actions and transition actions.
	Action language defines the categories of actions you can specify and their associated notations. For example, states can have `entry`, `during`, `exit`, and on *event_name* actions.
API (application programming interface)	Format you can use to access and communicate with an application program from a programming or script environment.
atomic box	Graphical object that enables you to reuse functions across multiple charts. For more information, see "When to Use Atomic Boxes" on page 8-39.
atomic subchart	Graphical object that enables you to reuse states and subcharts across multiple charts. For more information, see "When to Use Atomic Subcharts" on page 15-6.
box	Graphical object that groups together other graphical objects in your chart. For details about how a box affects chart execution, see "Group Chart Objects by Using Boxes" on page 8-29.
chart instance	Link from a model to a chart stored in a Simulink library. A chart in a library can have many chart instances. Updating the chart in the library automatically updates all instances of that chart.
condition	Boolean expression to specify that a transition occurs when the specified expression is true.
connective junction	Illustrates decision points in the system. A connective junction is a graphical object that simplifies Stateflow chart representations and facilitates generation of

efficient code. Connective junctions provide different ways to represent desired system behavior.

See "Connective Junctions" on page 2-37 for more information.

data   *Data* objects store numerical values for reference in the Stateflow chart.

decomposition   A state has a *decomposition* when it consists of one or more substates. A chart that contains at least one state also has decomposition. Rules govern how you can group states in the hierarchy. A superstate has either parallel (AND) or exclusive (OR) decomposition. All substates at a particular level in the hierarchy must have the same decomposition.

- Parallel (AND) State Decomposition

  Parallel (AND) state decomposition applies when states have dashed borders. This decomposition describes states at that same level in the hierarchy that can be active at the same time. The activity within parallel states is essentially independent.

- Exclusive (OR) State Decomposition

  Exclusive (OR) state decomposition applies when states have solid borders. This decomposition describes states that are mutually exclusive. Only one state at the same level in the hierarchy can be active at a time.

default transition   Primarily used to specify which exclusive (OR) state is to be entered when there is ambiguity among two or more neighboring exclusive (OR) states. For example, default transitions specify which substate of a superstate with exclusive (OR) decomposition the system enters by default in the absence of any other information. Default transitions can also specify that a junction should be entered by default. The default transition object is a transition with a destination but no source object.

See "Default Transitions" on page 2-33 for more information.

events

Events drive chart execution. All events that affect the chart must be defined. The occurrence of an event causes the status of states in a chart to be evaluated. The broadcast of an event can trigger a transition to occur or an action to execute. Events are broadcast in a top-down manner starting from the event's parent in the hierarchy.

Finder

Tool to search for objects in Stateflow charts on platforms that do not support the Simulink Find tool.

finite state machine (FSM)

Representation of an event-driven system. FSMs are also used to describe reactive systems. In an event-driven or reactive system, the system transitions from one mode or state to another prescribed mode or state, provided that the condition defining the change is true.

flow chart

Set of decision flow paths that start from a transition segment that, in turn, starts from a state or a default transition segment.

flow path

Ordered sequence of transition segments and junctions where each succeeding segment starts on the junction that terminated the previous segment.

flow subgraph

Set of decision flow paths that start on the same transition segment.

graphical function

A chart function whose logic is defined by a flow chart. See "Reuse Logic Patterns by Defining Graphical Functions" on page 8-16.

hierarchy

Hierarchy enables you to organize complex systems by placing states within other higher-level states. A hierarchical design usually reduces the number of transitions and produces neat, more manageable charts. See "Hierarchy of Stateflow Objects" on page 1-7 for more information.

history junction

Specifies the destination substate of a transition based on historical information. If a superstate has a history junction, the transition to the destination substate is the substate that was most recently active. The history junction applies only to the level of the hierarchy in which it appears.

See the following sections for more information:

- "History Junctions" on page 2-44
- "Default Transition and a History Junction" on page B-17
- "Labeled Default Transitions" on page B-19
- "Inner Transition to a History Junction" on page B-26

inner transitions

Transition that does not exit the source state. Inner transitions are useful when defined for superstates with exclusive (OR) decomposition. Use of inner transitions can greatly simplify chart layout.

See "Inner Transitions" on page 2-29 and "Inner Transition to a History Junction" on page B-26 for more information.

library link

Link to a chart that is stored in a library model.

library model

Stateflow model that is stored in a Simulink library. You can include charts from a library in your model by copying them. When you copy a chart from a library into your model, you create only a link to the library chart. You can create multiple links to a single chart. Each link is called a *chart instance*. When you include a chart from a library in your model, you also include its Stateflow machine. Therefore, a Stateflow model that includes links to library charts has multiple Stateflow machines.

When you simulate a model that includes charts from a library model, you include all charts from the library model even if links exist only for some of its models. You can simulate a model that includes links to a library model

only when all charts in the library model are free of parse and compile errors.

machine Collection of all Stateflow blocks defined by a Simulink model. This collection excludes chart instances from library links. If a model includes any library links, it also includes the Stateflow machines defined by the models from which the links originate.

MATLAB function A chart function that works with a subset of the MATLAB programming language.

Mealy machine An industry-standard paradigm for modeling finite-state machines, where output is a function of both inputs *and* state.

Model Explorer Use to add, remove, and modify data, event, and target objects in the Stateflow hierarchy. See "Use the Model Explorer with Stateflow Objects" on page 33-13 for more information.

Moore machine An industry-standard paradigm for modeling finite-state machines, where output is a function *only* of state.

notation Defines a set of objects and the rules that govern the relationships between those objects. Stateflow chart notation provides a way to communicate the design information in a Stateflow chart.

Stateflow chart notation includes:

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

parallelism A system with *parallelism* can have two or more states that can be active at the same time. The activity of parallel states is essentially independent. Parallelism is represented with a parallel (AND) state decomposition.

See "State Decomposition" on page 2-8 for more information.

S-function
When you simulate a Simulink model containing Stateflow charts, you generate an *S-function* (MEX-file) for each Stateflow machine. This generated code is a simulation target.

semantics
Semantics describe how the notation is interpreted and implemented behind the scenes. A completed Stateflow chart communicates how the system will behave. A chart contains actions associated with transitions and states. The semantics describe in what sequence these actions take place during chart execution.

Simulink function
A chart function that you fill with Simulink blocks and call in the actions of states and transitions. This function provides an efficient model design and improves readability by minimizing the graphical and nongraphical objects required in a model. In a Stateflow chart, this function acts like a function-call subsystem block of a Simulink model.

state
A *state* describes a mode of a reactive system. A reactive system has many possible states. States in a chart represent these modes. The activity or inactivity of the states dynamically changes based on transitions among events and conditions.

Every state has hierarchy. In a chart consisting of a single state, the parent of that state is the Stateflow chart itself. A state also has history that applies to its level of hierarchy in the chart. States can have actions that execute in a sequence based upon action type. The action types are `entry`, `during`, `exit`, or `on event_name` actions.

Stateflow block
Masked Simulink model that is equivalent to an empty, untitled Stateflow chart. Use the Stateflow block to include a chart in a Simulink model.

The control behavior modeled by a Stateflow block complements the algorithmic behavior modeled in Simulink block diagrams. By incorporating Stateflow

blocks into Simulink models, you can add complex event-driven behavior to Simulink simulations. You create models that represent both data and decision flow by combining Stateflow blocks with the standard Simulink and toolbox block libraries.

Stateflow chart   A Stateflow chart is a graphical representation of a finite state machine where *states* and *transitions* form the basic building blocks of the system. See "Stateflow Charts and Simulink Models" on page 1-4 for more information.

Stateflow Finder   Use to display a list of objects based on search criteria you specify. You can directly access the properties dialog box of any object in the search output display by clicking that object.

subchart   Chart contained by another chart. See "Encapsulate Modal Logic by Using Subcharts" on page 8-5.

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

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

supertransition   Transition between objects residing in different subcharts. See "Move Between Levels of Hierarchy by Using Supertransitions" on page 8-9 for more information.

target   A container object for the generated code from the Stateflow charts in a model. The collection of all charts for a model appears as a Stateflow machine. Therefore, target objects belong to the Stateflow machine.

The code generation process can produce these target types: simulation, embeddable, and custom.

transition   The circumstances under which the system moves from one state to another. Either end of a transition can be attached to a source and a destination object. The *source* is where the transition begins and the *destination* is where the transition ends. Usually, the occurrence of an event causes a transition to take place.

transition path   Flow path that starts and ends on a state.

transition segment   A state-to-junction, junction-to-junction, or junction-to-state part of a complete state-to-state transition.

truth table function   A chart function that specifies logical behavior with conditions, decisions, and actions. Truth tables are easier to program and maintain than graphical functions.