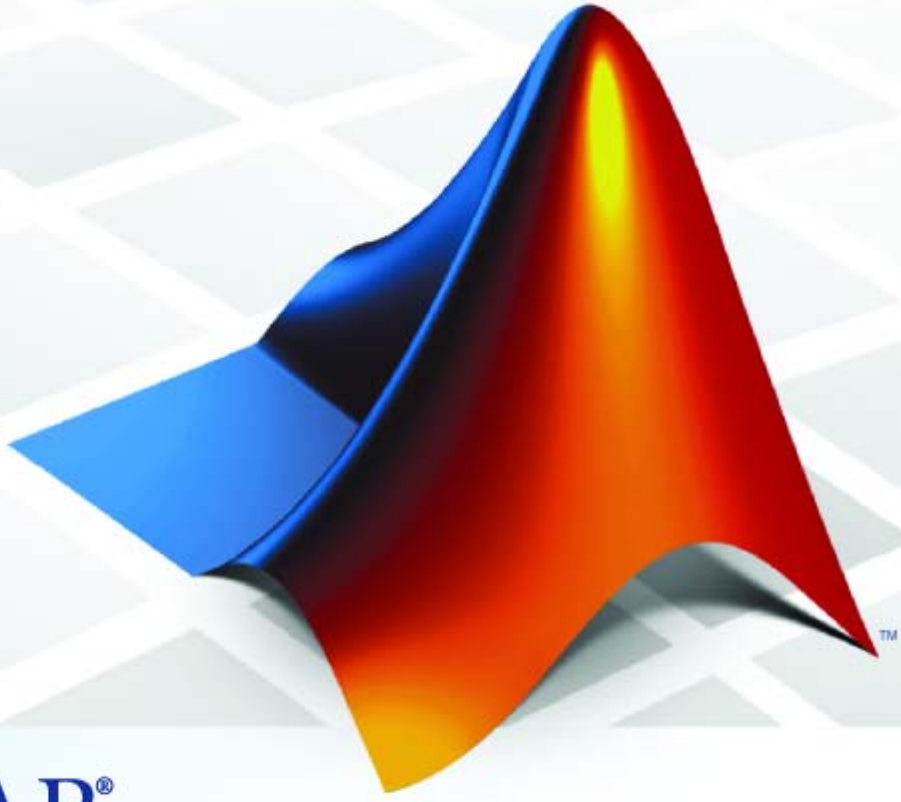
Stateflow® and Stateflow® Coder™ 7
User's Guide



MATLAB®
&SIMULINK®

The MathWorks™
Accelerating the pace of engineering and science

How to Contact The MathWorks

`www.mathworks.com`	Web
`comp.soft-sys.matlab`	Newsgroup
`www.mathworks.com/contact_TS.html`	Technical Support

`suggest@mathworks.com`	Product enhancement suggestions
`bugs@mathworks.com`	Bug reports
`doc@mathworks.com`	Documentation error reports
`service@mathworks.com`	Order status, license renewals, passcodes
`info@mathworks.com`	Sales, pricing, and general information

508-647-7000 (Phone)

508-647-7001 (Fax)

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

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

Stateflow® and Stateflow® Coder™ User's Guide

Trademarks

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

Patents

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

Revision History

May 1997	First printing	New
January 1999	Second printing	Revised for Version 2.0 (Release 11)
September 2000	Third printing	Revised for Version 4.0 (Release 12))
June 2001	Fourth printing	Revised for Version 4.1 (Release 12.1)
July 2002	Fifth printing	Revised for Version 5.0 (Release 13)
January 2003	Online only	Revised for Version 5.1 (Release 13SP1) (Renamed from Stateflow® User's Guide)
June 2004	Online only	Revised for Version 6.0 (Release 14)
October 2004	Online only	Revised for Version 6.1 (Release 14SP1)
March 2005	Online only	Revised for Version 6.21 (Release 14SP2)
September 2005	Online only	Revised for Version 6.3 (Release 14SP3)
March 2006	Online only	Revised for Version 6.4 (Release R2006a)
September 2006	Online only	Revised for Version 6.5 (Release R2006b)
March 2007	Online only	Revised for Version 6.6 (Release R2007a)
September 2007	Online only	Revised for Version 7.0 (Release R2007b)
March 2008	Online only	Revised for Version 7.1 (Release R2008a)
October 2008	Online only	Revised for Version 7.2 (Release R2008b)

Contents

1 Stateflow Chart Concepts

2 Stateflow Chart Notation

3 Stateflow Chart Semantics

4 Creating Stateflow Charts

5 Modeling Logic Patterns and Iterative Loops Using Flow Graphs

6 Building Mealy and Moore Charts

7 Extending Stateflow Charts

8 Defining Data

9 Defining Events

10 Using Actions in Stateflow Charts

11 Using Vectors and Matrices in Stateflow Charts

12 Using Enumerated Data in Stateflow Charts

13 Modeling Continuous-Time Systems in Stateflow Charts

14 Using Fixed-Point Data in Stateflow Charts

15 Using Complex Data in Stateflow Charts

Defining Interfaces to Simulink Models and the MATLAB Workspace

16

17 Working with Structures and Bus Signals in Stateflow Charts

18 Stateflow Design Patterns

19 Truth Table Functions

20 Using Embedded MATLAB Functions in Stateflow Charts

21 Using Simulink Functions in Stateflow Charts

22

23 Debugging and Testing Stateflow Charts

24 Exploring and Modifying Charts

A Semantic Rules Summary

1

Stateflow Chart Concepts

Finite State Machine Concepts

In this section...

What Is a Finite State Machine?

A Stateflow® chart is an example of a finite state machine. 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, provided that the condition defining the change is true.

For example, you can use a state machine to represent a car's automatic transmission. 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 state that is active is determined based on the occurrence of events under certain conditions. State-transition charts and bubble charts are graphical representations based on this approach.

Stateflow Chart Representations

A Stateflow chart uses a variant of the finite state machine notation established by Harel [1]. A chart is a graphical representation of a finite state machine, where *states* and *transitions* form the basic building blocks of the system. You can also represent stateless charts (flow graphs). You can include Stateflow charts as blocks in a Simulink® model. The collection of Stateflow 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/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 specify the destination state of a transition based on historical information. These characteristics go beyond what state-transition charts and bubble charts provide.

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 the following:

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

See Chapter 2, “Stateflow Chart Notation”, for detailed information on Stateflow chart notation.

Semantics

Semantics describe how the notation is interpreted and implemented. A completed Stateflow chart illustrates how the system will behave. A Stateflow chart contains actions associated with transitions and states. The semantics describe the sequence of these actions during chart execution.

If you know the semantics, you can design a sound Stateflow chart for code generation.

For the default semantics, see Chapter 3, “Stateflow Chart Semantics”.

References

For more information on finite state machine theory, consult these sources:

[1] Harel, David, “Statecharts: A Visual Formalism for Complex Systems,” *Science of Computer Programming 8*, 1987, pages 231-274.

[2] Hatley, Derek J., and Imtiaz A. Pirbhai, *Strategies for Real-Time System Specification*, Dorset House Publishing Co., Inc., NY, 1988.

Stateflow Charts and Simulink Models

In this section...

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 Stateflow chart consists of a set of graphical objects (states, boxes, functions, notes, transitions, connective junctions, and history junctions) and nongraphical objects (events, data, and targets).

There is a one-to-one correspondence between the Simulink model and the Stateflow machine. Each Stateflow block in the Simulink model is represented by 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 "Stateflow Hierarchy of Objects" on page 1-20.

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).

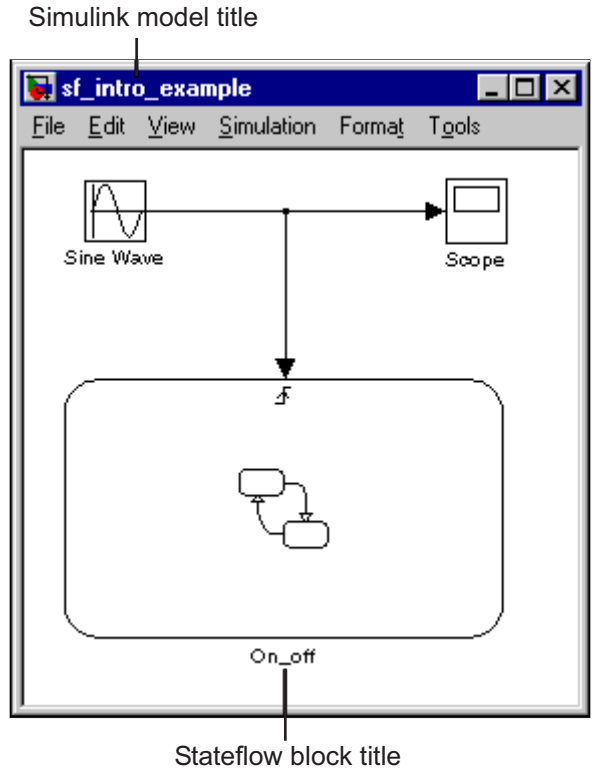
Stateflow charts are event-driven. Events can be local to the Stateflow block or can propagate to and from the Simulink model and external code sources.

Data can be local to the Stateflow block or can pass to and from the Simulink model and external code sources.

You must define the interface to each Stateflow block. Defining the interface for a Stateflow block can involve some or all of these tasks:

- Defining the Stateflow block update method
- Defining **Output to Simulink** 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 titled `sf_intro_example` consists of a Sine Wave source block, a Scope sink block, and a single Stateflow block, titled `On_off`.



For more information, see “Using Input Events to Activate a Stateflow Chart” on page 9-11 and Chapter 16, “Defining Interfaces to Simulink Models and the MATLAB Workspace”.

Stateflow Chart Objects

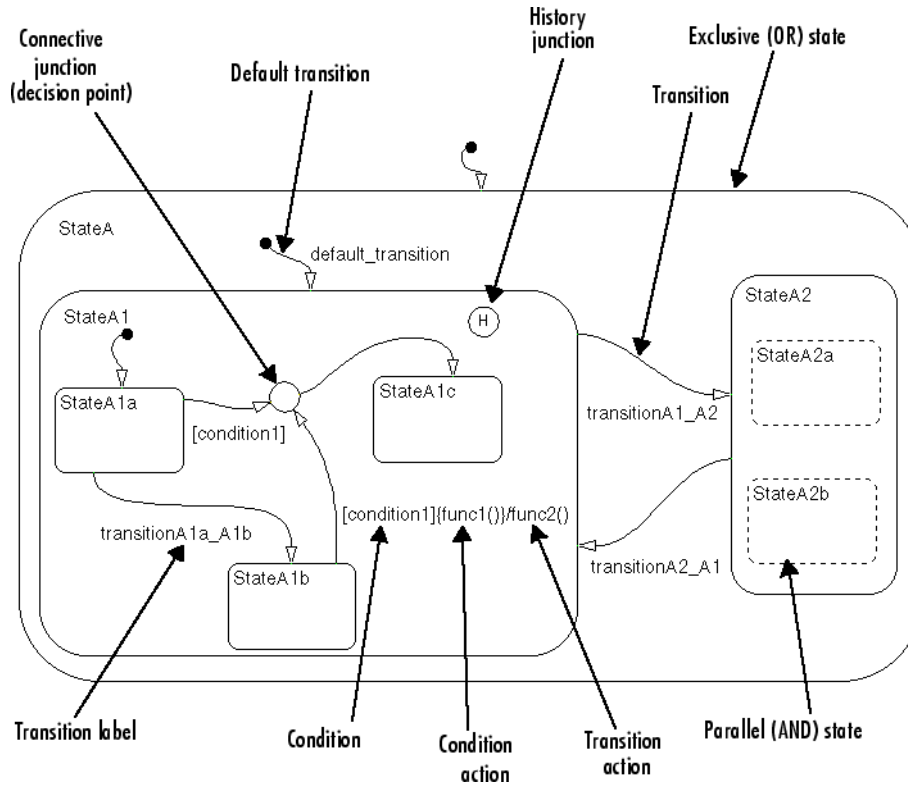
In this section...

All Stateflow objects are arranged in a hierarchy of objects. See "Stateflow Hierarchy of Objects" on page 1-20.

About Stateflow Objects

Stateflow charts consist of objects. Some of these objects are graphical, which you draw in a Stateflow chart. Others are nongraphical, which you reference textually in the chart.

The following sample chart displays some key graphical objects.



Stateflow® Graphical Objects

States

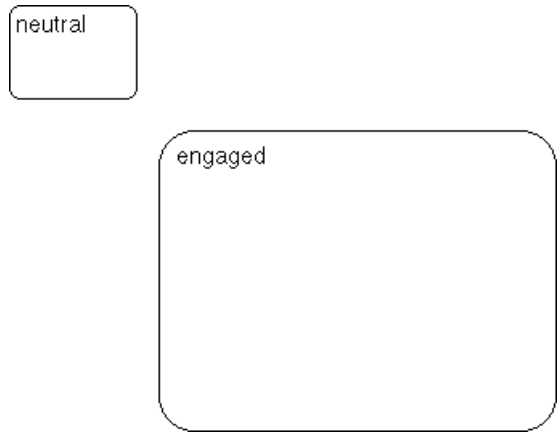
A *state* describes a mode of an event-driven system. The activity or inactivity of the states dynamically changes based on events and conditions.

Every state has a parent. In a Stateflow chart consisting of a single state, that state's parent is the chart itself (also called the chart root). You can place states within other higher-level states. In the preceding figure, `StateA1` is a child of `StateA`.

A state can have its activity history recorded in a *history junction*. History provides an efficient means of basing future activity on past activity. See "History Junction" on page 1-14.

States have labels that can specify actions executed in a sequence based upon action type. The action types are `entry`, `during`, `exit`, and `on`. See "Actions" on page 1-15.

Decomposition defines what a state can contain. You can use two types of states: exclusive (OR) and parallel (AND) states. Exclusive (OR) states are used to describe modes that are mutually exclusive. A chart or state that contains exclusive (OR) states has exclusive decomposition. The following transmission example has exclusive (OR) states.

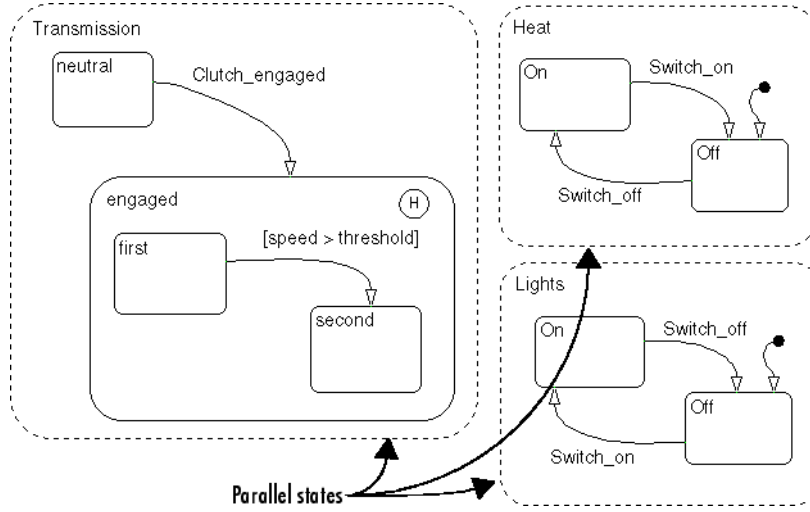


An automatic transmission can be set to either neutral or engaged. In this example, either the `neutral` state or the `engaged` state is active at any one time. Both cannot be active at the same time.

A chart or state with *parallel* states has two or more states that can be active at the same time. A chart or state that contains parallel (AND) states has parallel decomposition.

Parallel (AND) states are displayed as dashed rectangles. The activity of each parallel state is essentially independent of other states. In the chart in Stateflow® Graphical Objects on page 1-9, `StateA2` has parallel (AND) state decomposition. Its states, `StateA2a` and `StateA2b`, are parallel (AND) states.

This Stateflow chart has parallel superstate decomposition.

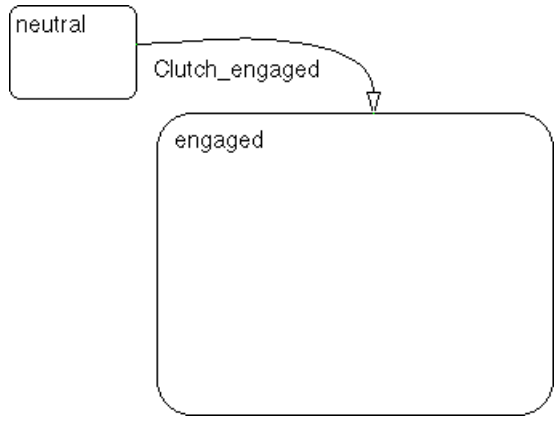


In this example, the transmission, heating, and light systems are parallel subsystems in a car. They are active at the same time and are physically independent of each other. There are many other parallel components in a car, such as the braking and windshield wiper subsystems.

Transitions

A *transition* is a graphical object that, in most cases, links one object to another. One end of a transition is attached to a source object and the other end to a destination object. The *source* is where the transition begins and the *destination* is where the transition ends. A *transition label* describes the circumstances under which the system moves from one state to another. The occurrence of an event causes a transition to take place. In the chart in Stateflow® Graphical Objects on page 1-9, the transition from `StateA1` to `StateA2` is labeled with the event `transitionA1_A2` that triggers the transition to occur.

Consider again the automatic transmission system. `clutch_engaged` is the event required to trigger the transition from `neutral` to `engaged`.

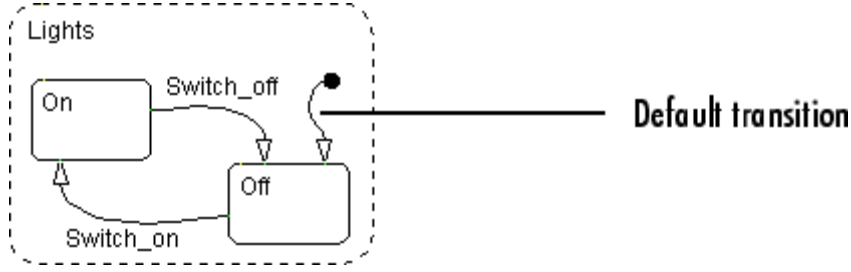


Default Transitions

Default transitions specify which exclusive (OR) state is to be active when there is ambiguity between two or more exclusive (OR) states at the same level in the hierarchy.

For example, in the chart in Stateflow® Graphical Objects on page 1-9, the default transition to `StateA1` defines whether `StateA1` or `StateA2` should be active when `State A` becomes active. In this case, when `StateA` is active, by default `StateA1` is also active.

In the following `Lights` subsystem, the default transition to the `Lights.Off` substate indicates that when the `Lights` superstate becomes active, the `Off` substate becomes active by default.



Note History junctions override default transition paths in superstates with exclusive (OR) decomposition. In parallel (AND) states, a default transition must always be present to indicate which of its exclusive (OR) states is active when the parallel state becomes active.

Events

Events drive the Stateflow chart execution but are nongraphical objects and are thus not represented directly in a Stateflow chart. You must define all events that affect the chart. The occurrence of an event causes the status of the states in the chart to be evaluated. The broadcast of an event can trigger a transition to occur or can trigger an action to be executed. Events are broadcast in a top-down manner starting from the event's parent in the hierarchy.

Events are created and modified using the Model Explorer, and they can exist at any level in the hierarchy. Events have properties such as a `scope`. The `scope` defines whether the event is

- Local to the Stateflow chart
- An input to the chart from its Simulink model
- An output from the chart to its Simulink model
- Exported to a (code) destination external to the chart and Simulink model
- Imported from a code source external to the chart and Simulink model

Data

Data objects are used to store numerical values for reference in the Stateflow chart. They are nongraphical objects and are thus not represented directly in a Stateflow chart.

You create and modify data objects for Stateflow charts in the Model Explorer. Data objects have a property called `scope` that defines whether the data object is

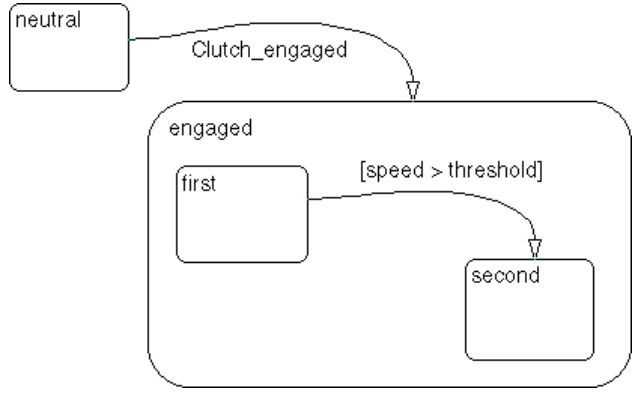
- Local to the Stateflow chart

- An input to the chart from its Simulink model
- An output from the chart to its Simulink model
- Nonpersistent temporary data
- Defined in the MATLAB® workspace
- A constant
- Exported to a (code) destination external to the chart and Simulink model
- Imported from a code source external to the chart and Simulink model

Conditions

A *condition* is a Boolean expression specifying that a transition occurs, given that the specified expression is true. In the component summary Stateflow chart, `[condition1]` represents a Boolean expression that must be true for the transition to occur.

In the automatic transmission system, the transition from `first` to `second` occurs if the transition condition `[speed > threshold]` is true.

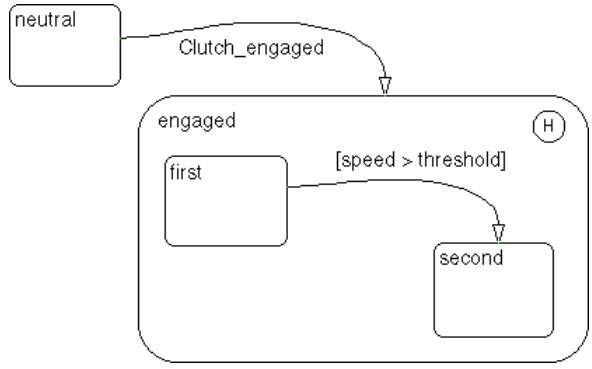


History Junction

A *history junction* records the most recently active state of a chart or superstate.

If a superstate with exclusive (OR) decomposition has a history junction, the destination substate is defined to be the substate that was most recently visited. A history junction applies to the level of the hierarchy in which it appears, and it overrides any default transitions. In the component summary Stateflow chart, the history junction in `StateA1` indicates that when a transition to `StateA1` occurs, the substate that becomes active (`StateA1a`, `StateA1b`, or `StateA1c`) is based on which of those substates was most recently active.

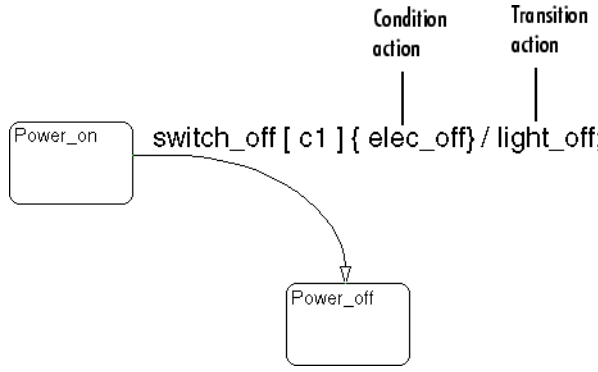
In the automatic transmission system, history indicates that when `clutch_engaged` causes a transition from `neutral` to the `engaged` superstate, the substate that becomes active, either `first` or `second`, is based on which of those substates was most recently active.



Actions

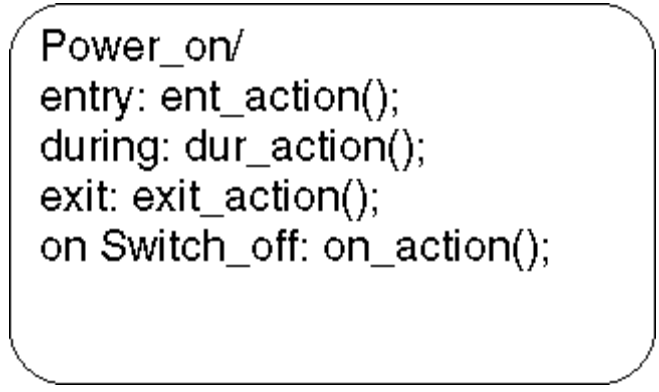
Actions take place as part of Stateflow chart execution. The action can be executed either as part of a transition from one state to another or based on the activity status of a state.

Transitions ending in a state can have *condition* actions and *transition* actions, as shown in the following example:



In the chart in Stateflow® Graphical Objects on page 1-9, the transition segment from `StateA1b` to the connective junction is labeled with the condition action `func1()` and the transition action `func2()`. The semantics of how and why actions take place are discussed throughout the examples listed in "Semantic Examples" on page 3-52.

States can have `entry`, `during`, `exit`, and `on` *`event_name`* actions. For example,



Action language defines the types of actions you can specify and their associated notations. An action can be a function call, the broadcast of an event, the assignment of a value to a variable, and so on.

A Stateflow chart supports both Mealy and Moore modeling paradigms for finite state machines. In the Mealy model, actions are associated with transitions, whereas in the Moore model, they are associated with states. For more information, see Chapter 6, "Building Mealy and Moore Charts".

A Stateflow chart supports state actions, transition actions, and condition actions. For more information, see the following:

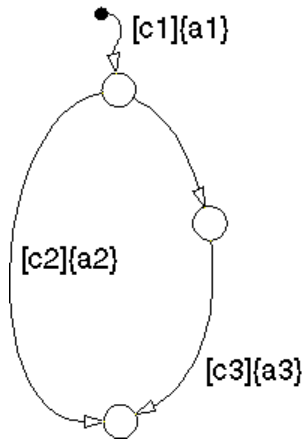
- "State Labels" on page 2-8 — Describes action language for states, which is included in the label for a state.
- "Transition Label Notation" on page 2-14 — Describes action language for transitions which is included in the label of a transition.
- "Labeling States" on page 4-13 — Shows you to label states with its name and actions in the Stateflow Editor.
- "Labeling Transitions" on page 4-20 — Shows you how to label transitions with actions in the Stateflow Editor.

Connective Junctions

Connective junctions are 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 alternative ways to represent desired system behavior. In the chart in "Stateflow Chart Objects" on page 1-8, the connective junction is used as a decision point for two transition segments that complete at `StateA1c`.

Transitions connected to junctions are called *transition segments*. Transitions, apart from default transitions, must go state to state. However, once the transition segments taken complete a state to state transition, the accumulation of the transition segments taken forms a complete transition.

The following example shows how connective junctions (displayed as small circles) are used to represent the flow of an `if-else` code structure shown in accompanying pseudocode.




```
if [c1]{
      a1
      if [c2]{
         a2
      }else if [c3]{
        a3
      }
}
```

This example executes as follows:

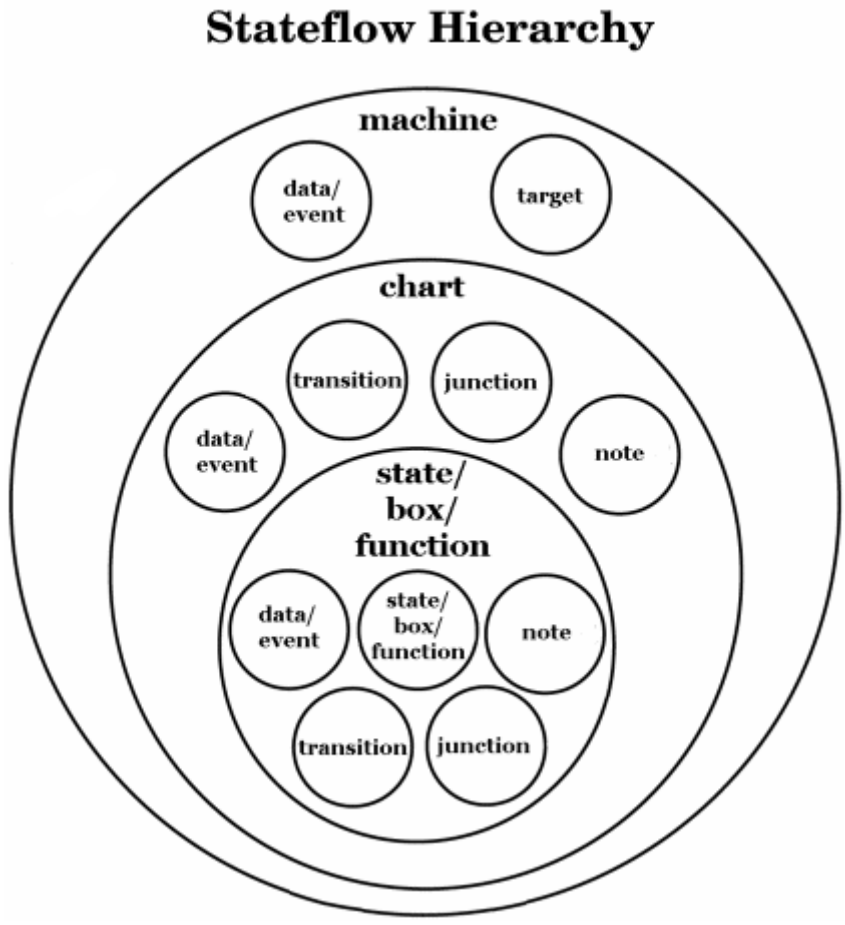
1. If condition `[c1]` is true, condition action `a1` is executed and the default transition to the top junction is taken.
2. The Stateflow chart now considers which transition segment to take out of the top junction (it can take only one). Junctions with conditions have priority over junctions without conditions, so the transition with the condition `[c2]` is considered first.
3. If condition `[c2]` is true, action `a2` is executed and the transition segment to the bottom junction is taken. Because there are no outgoing transition segments from the bottom junction, the chart is finished executing.
4. If condition `[c2]` is false, the empty transition segment on the right is taken (because it has no condition at all).
5. If condition `[c3]` is true, condition action `a3` is executed and the transition segment from the middle to the bottom junction is taken. Because there are no outgoing transition segments from the bottom junction, the chart is finished executing.
6. If condition `[c3]` is false, execution is finished at the middle junction.

The above steps describe the execution of the example chart for connective junctions with Stateflow chart semantics. These semantics describe how

objects in charts relate to each other during execution. See Chapter 3, "Stateflow Chart Semantics".

Stateflow Hierarchy of Objects

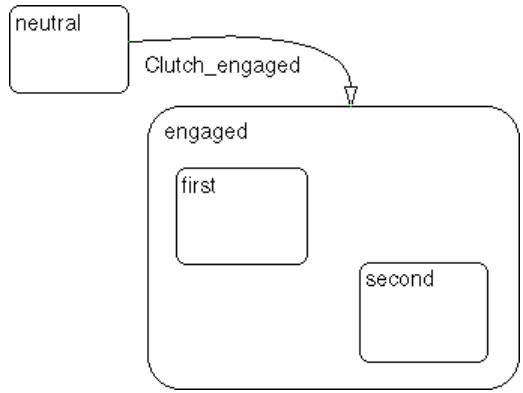
Stateflow machines arrange Stateflow objects in a hierarchy based on containment. That is, one Stateflow object can contain other Stateflow objects.



The highest object in Stateflow hierarchy is the Stateflow machine. It is defined as an object that contains all other Stateflow objects in a Simulink model. This means that the Stateflow machine contains all the Stateflow charts in a Simulink model. In addition, the Stateflow machine for a model

can also contain its own data, event, and target objects. Only a simulation target is added to the Stateflow machine by default when the model is created. All other data, event, and target objects must be added to the machine.

Similarly, charts can contain state, box, function, data, event, transition, junction, and note events. You use all of these objects to create a Stateflow chart. Continuing with the Stateflow hierarchy, states can contain all of these objects as well, including other states. You can represent state hierarchy with superstates and substates. For example, this chart has a superstate that contains two substates.



In the preceding chart, the `engaged` superstate contains the `first` and `second` substates. The `engaged` superstate is the parent in the hierarchy to the states `first` and `second`. When the event `clutch_engaged` occurs, the system transitions out of the `neutral` state to the `engaged` superstate. Transitions within the `engaged` superstate are intentionally omitted from this example for simplicity.

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 is made active, its parent superstate is also made 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 Model Explorer or the Stateflow Editor.

Exploring a Typical Stateflow Application

In this section...

About the Application

The modeling of a fault-tolerant fuel control system demonstrates how a Stateflow chart in a Simulink model can be used to model hybrid systems that contain both continuous dynamics and complex control logic.

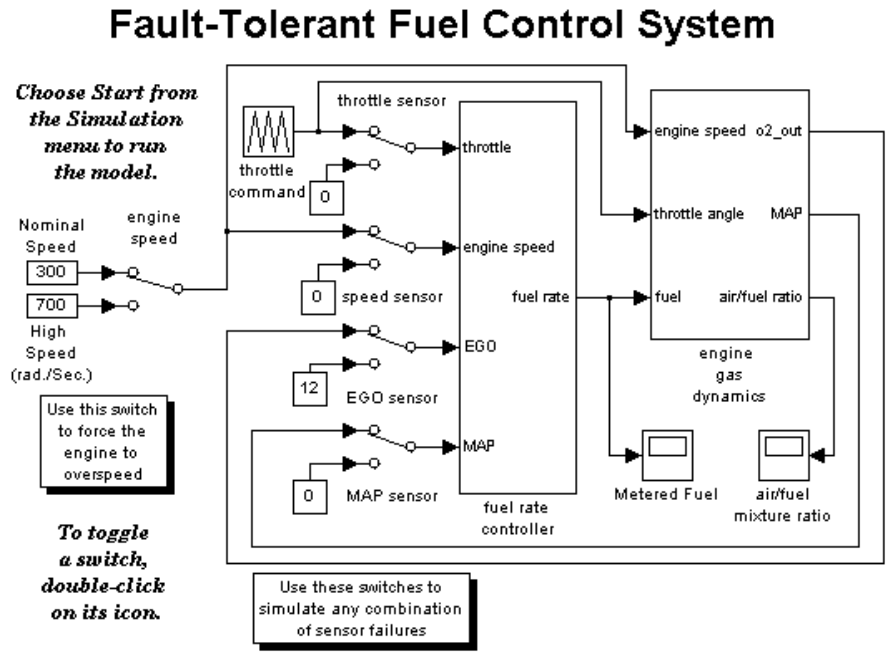
Simulink blocks model behavior based on a given sample time. Each loop of its block diagram is assigned an increment of sample time. Stateflow chart execution does not consider sample time. Internally, the chart can take many cycles of execution, which are assumed to take place during the sample time assigned in a Simulink model.

Overview of the "fuel rate controller" Model

The model described represents a fuel control system for a gasoline engine. This robust control system reacts to the detection of individual sensor failures and is dynamically reconfigured for uninterrupted operation. The mass flow rate of air pumped from the intake manifold, divided by the fuel rate, which is injected at the valves, gives the air/fuel ratio. The ideal mixture ratio provides a good compromise between power, fuel economy, and emissions. A target air/fuel ratio of 14.6 is assumed in this system.

A sensor (EGO) determines the amount of residual oxygen present in the exhaust gas. This sensor gives a good indication of the air/fuel ratio and provides a feedback measurement for closed-loop control. If the sensor indicates a high oxygen level, the controller increases the fuel rate. If the sensor detects a fuel-rich mixture (corresponding to a very low level of residual oxygen), the controller decreases the fuel rate.

The following figure shows the top level of the Simulink model (`fuelsys.mdl`). The model consists of a fuel rate controller and a subsystem to simulate engine gas dynamics.

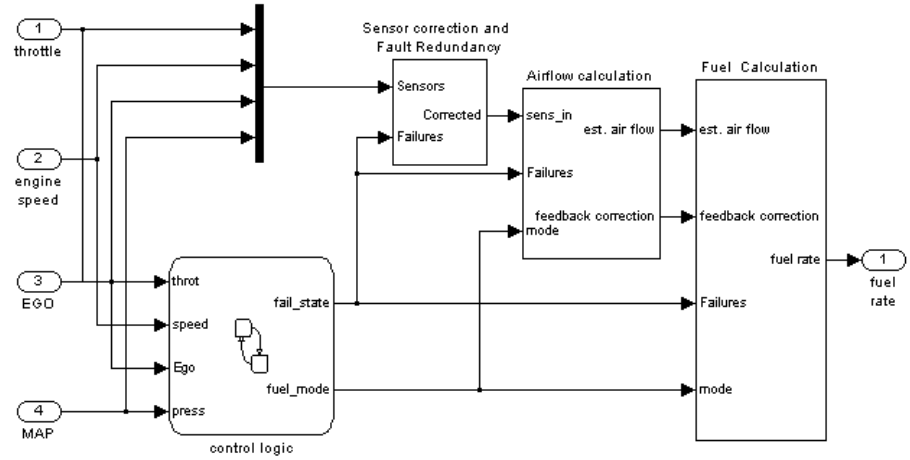


The fuel rate controller uses signals from the system sensors to determine the fuel rate that gives an ideal mixture. The fuel rate combines with the actual air flow in the engine gas dynamics model to determine the resulting mixture ratio as sensed at the exhaust.

To simulate failures in the system, you can selectively disable each of the four sensors: throttle angle, speed, exhaust gas (EGO), and manifold absolute pressure (MAP). These sensors appear in the Simulink model as Manual Switch blocks. You can toggle the position of a switch by double-clicking the icon prior to or during a simulation. Similarly, you can force the failure condition of a high engine speed by toggling the switch on the far left.

The controller uses the sensor input and feedback signals to adjust the fuel rate to provide an ideal ratio. The model uses four subsystems to implement

this strategy: control logic, sensor correction, airflow calculation, and fuel calculation. Under normal operation, the model estimates the airflow rate and multiplies the estimate by the reciprocal of the desired ratio to give the fuel rate. Feedback from the oxygen sensor provides a closed-loop adjustment of the rate estimation to maintain the ideal mixture ratio.



A detailed explanation of the Simulink blocks of the fault-tolerant control system is given in *Using Simulink and Stateflow in Automotive Applications*, a Simulink-Stateflow Technical Examples booklet published by The MathWorks, Inc. This section concentrates on the control logic implemented in a chart, but these key points are crucial to the interaction between Simulink models and Stateflow charts:

- The control logic monitors the input data readings from the sensors.
- The logic determines from these readings the sensors that have failed and outputs a failure state Boolean array as `fail_state`.
- Given the current failure state, the logic determines in which fueling mode the engine should run.

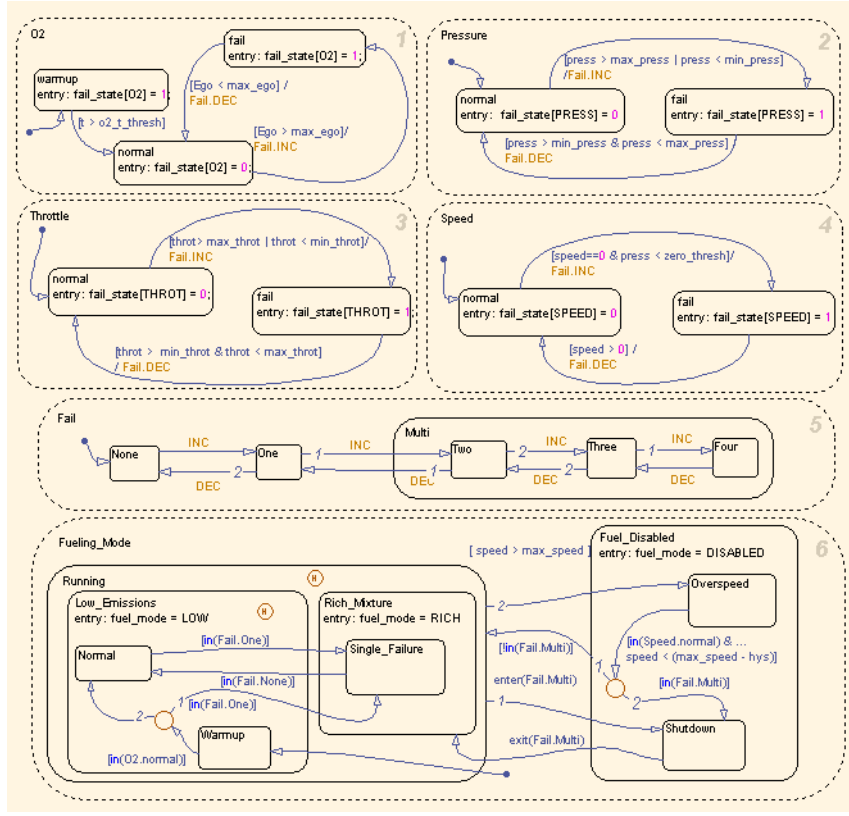
The fueling mode can be one of these options:

- **Low emissions mode** is the normal mode of operation where no sensors have failed.
- **Rich mixture mode** occurs when a sensor has failed, to ensure smooth running of the engine.
- **Shutdown mode** occurs when more than one sensor has failed, rendering the engine inoperable.

The Stateflow chart outputs fueling mode and failure state as `fuel_mode` and `fail_state`, respectively, to the rest of the model, which determines the fueling calculations.

Control Logic of the "fuel rate controller" Model

The Stateflow chart that implements control logic for the `fuelsys` model looks like this:



The chart contains six parallel states with dashed boundaries that represent concurrent modes of operation.

The four parallel states at the top of the chart correspond to four individual sensors. Each of these states has a substate that represents the normal or failing status of that sensor. These substates are mutually exclusive. For example, if the throttle sensor fails then the active substate of the `Throttle` state is `fail`.

Transitions determine how states can change and can be guarded by conditions. For example, the active state can change from the `normal` state to the `fail` state when the measurement from the throttle sensor exceeds `max_throt` or is below `min_throt`.

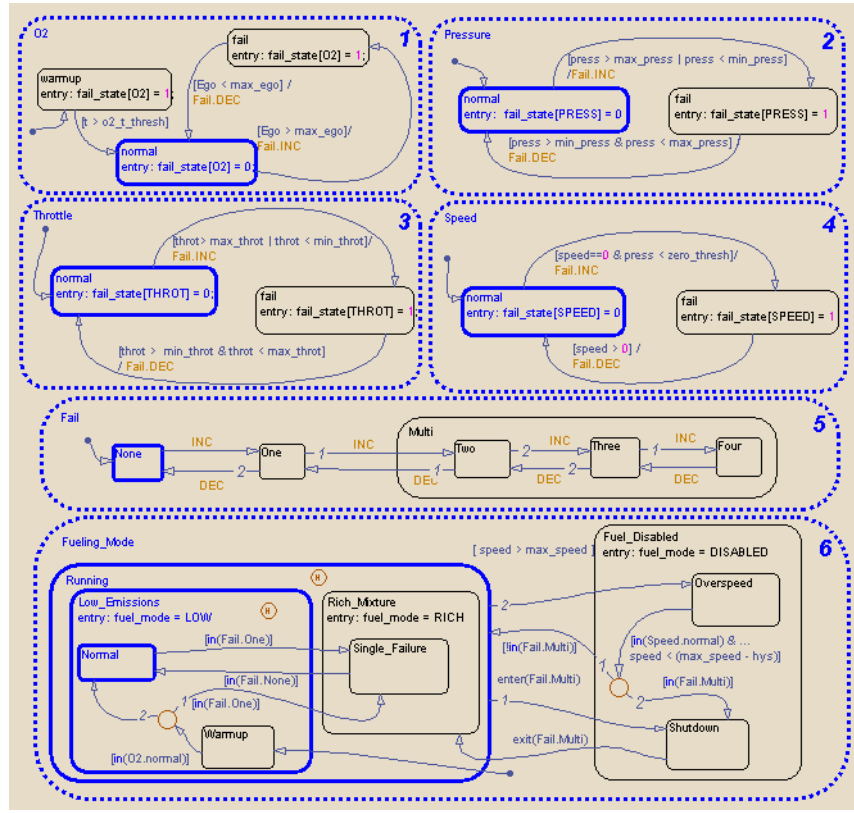
The two parallel states at the bottom consider the status of the four sensors simultaneously and determine the overall system operating mode. The `Fail` superstate stores the total number of sensor failures. This state is polled by the `Fueling_Mode` state that determines the fueling mode of the engine. If a single sensor fails, operation continues but the air/fuel mixture is richer to allow smoother running at the cost of higher emissions. If more than one sensor has failed, the engine shuts down as a safety measure, because the air/fuel ratio cannot be controlled reliably.

Although you can run Stateflow charts asynchronously by injecting events from a Simulink model, the fueling control logic is polled synchronously at a rate of 100 Hz. Therefore, the sensors are checked every 1/100 second to see if they have changed status, and the fueling mode is adjusted accordingly.

Simulating the "fuel rate controller" Model

On starting the simulation, and assuming no sensors have failed, the Stateflow chart initializes in the `Warmup` mode in which the oxygen sensor is in a warm-up phase.

If you animate the Stateflow chart, you can see the current state of the system highlighted on the chart.



After a given time period, defined by `o2_t_thresh`, the sensor reaches operating temperature and the system settles into the normal mode of operation, shown above, in which the fueling mode is set to `Normal`.

As the simulation progresses, the chart wakes up every 0.01 second. The events and conditions that guard the transitions are evaluated and if a transition is valid, it is taken and animated on the Stateflow chart.

For example, you can force a transition by switching a sensor to a failure value on the top-level Simulink model. The system detects throttle and pressure

sensor failures when their measured values fall outside nominal ranges. A manifold vacuum in the absence of a speed signal indicates a speed sensor failure. The oxygen sensor also has a nominal range for failure conditions but, because zero is both the minimum signal level and the bottom of the range, failure can be detected only when the oxygen level exceeds the upper limit.

Flip the Simulink switch for the manifold air pressure (MAP) sensor to the off position to see this sequence of transitions.

1. Switching the Simulink MAP sensor switch causes a value of zero to be read by the fuel rate controller.

2. When the chart next wakes up, the transition in the `Pressure` state from the `normal` substate to the `fail` substate occurs as the reading is now out of bounds.

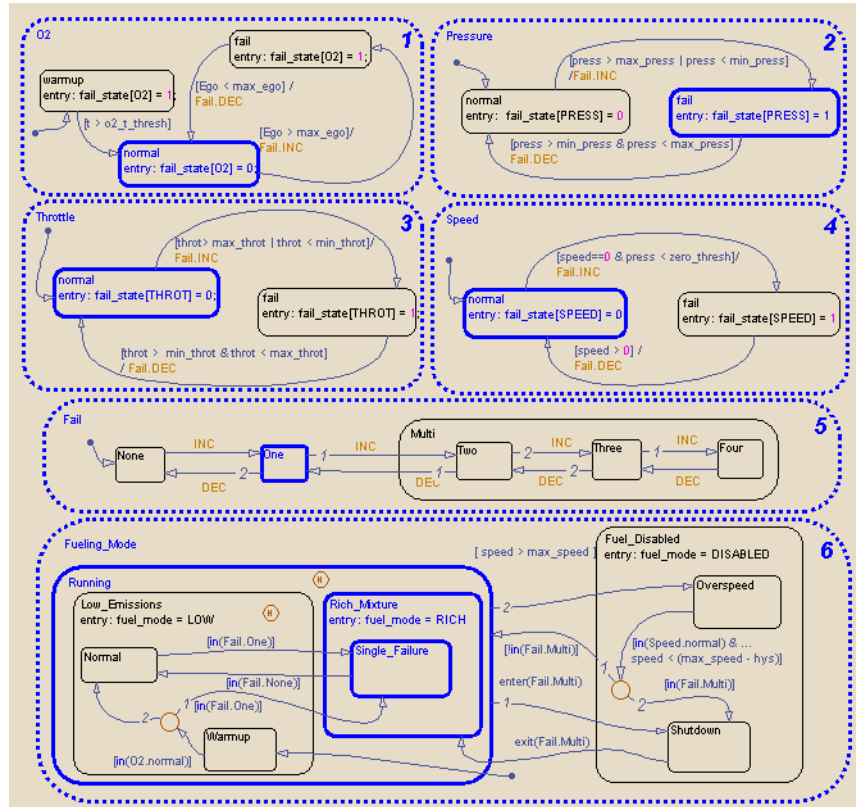
3. When a sensor fails, the chart always broadcasts the event `Fail.INC`, which makes triggering of global sensor failure logic independent of the sensor.

   This event causes a second transition from `None` to `One` in the `Fail` superstate.

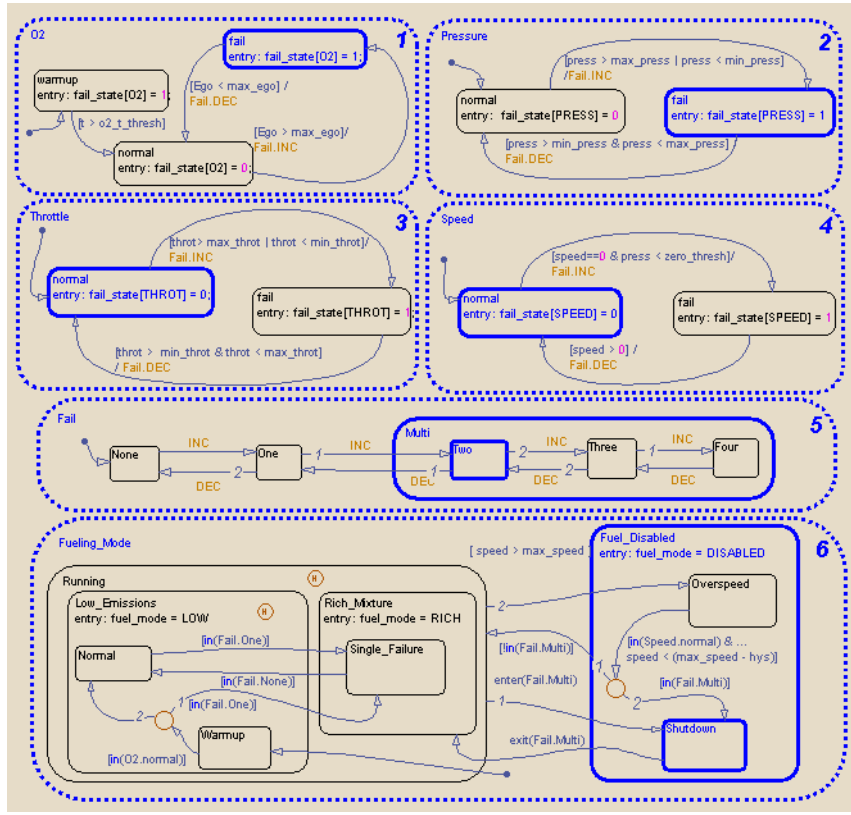
4. With the `Fail` superstate showing one failure, the condition that guards the transition from the `Low_Emissions.Normal` state to the `Rich_Mixture.Single_Failure` state is now valid. Therefore, the transition occurs.

5. The data `fuel_mode` is set to `RICH`.

After the transitions in the preceding steps occur, the chart appears as follows.



A second sensor failure causes the `Fail` superstate to enter the `Multi` state, broadcasting an implicit event that triggers the transition from the `Running` state to the `Shutdown` state. On entering the `Fuel_Disabled` superstate, the Stateflow data `fuel_mode` is set to `DISABLED`.

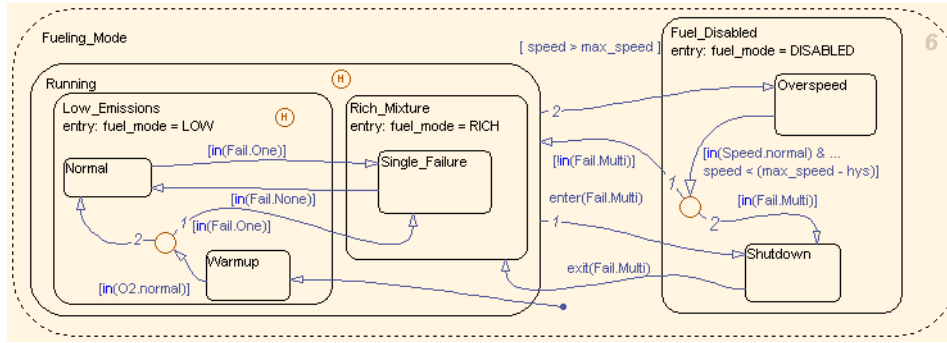


Using Clear Control Logic

The preceding example shows how you can represent control logic clearly. For example, conditions such as `in(Fail.None)` make the chart easy to read and the generated code more efficient.

Modifying the Model

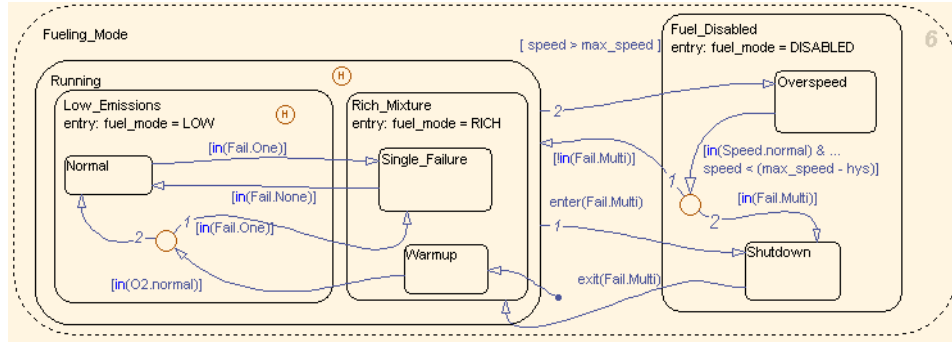
To illustrate how to modify the model, consider the `Warmup` state in the `Fueling_Mode` superstate. By default, fueling is set to the low emissions mode.



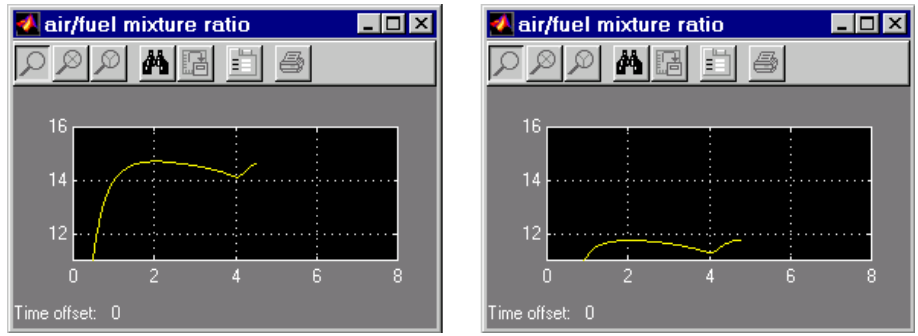
Suppose that changing the warm-up fueling mode to a rich mixture is beneficial. To modify the model, follow these steps:

1. In the Stateflow chart, enlarge the `Rich_Mixture` state.
2. Move the `Warmup` state from the `Low_Emissions` state to the `Rich_Mixture` state.

   The modified `Fueling_Mode` superstate should look something like this:



You can see the results of this change by observing the air/fuel mixture ratio for the first few seconds of engine operation. The left graph shows the air/fuel ratio for the original system. The right graph shows how the air/fuel ratio stays low during warm-up, indicating a rich mixture for the modified system.



2

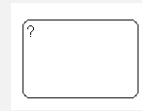
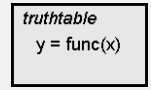
Stateflow Chart Notation

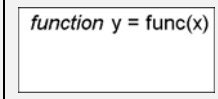
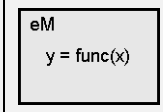
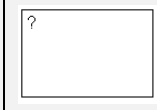
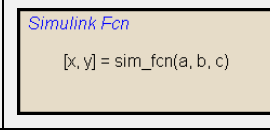
Overview of Stateflow Objects

In this section...

Graphical Objects

The following table gives the name of each graphical object in a Stateflow chart, its appearance when drawn in the Stateflow Editor (Notation), and the toolbar icon to use for drawing the object.

Name	Notation	Toolbar Icon
State		
Transition		Not applicable
History junction		
Default transition		
Connective junction		
Truth table function		

Name	Notation	Toolbar Icon
Graphical function		
Embedded MATLAB™ function		
Box		
Simulink function		

Nongraphical Objects

You can define data, event, and target objects that do not appear graphically in the Stateflow Editor. However, you can see them in the Model Explorer. See “Using the Model Explorer with Stateflow Objects” on page 24-2.

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 the action language of a Stateflow chart. For a full description of data objects, see Chapter 8, “Defining Data”.

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 a full description of event objects, see Chapter 9, “Defining Events”.

Target Objects

You build targets to execute the application you program in Stateflow charts and the Simulink model that contains them. A target is a program that executes a Stateflow chart or a Simulink model containing a Stateflow machine. You build a simulation target to execute a simulation of your model. You build an embeddable code generation target to execute the Simulink model on a supported processor environment. You build custom targets to pinpoint your application to a specific environment. For a full description of target objects, see Chapter 22, “Building Targets”.

Naming Stateflow Objects

You can name Stateflow objects with any combination of alphanumeric and special characters. The only restrictions are that names cannot begin with a numeric character or contain embedded spaces.

Name length should comply with the maximum identifier length enforced by Real-Time Workshop® code generation software. You can set this parameter in the **Real-Time Workshop > Symbols** pane of the Configuration Parameters dialog box. The default is 31 characters and the maximum length you can specify is 256 characters.

For more information, see “Maximum identifier length” in the Real-Time Workshop reference documentation.

For More Information on Stateflow Objects

Chapter 3, “Stateflow Chart Semantics” describes the various Stateflow objects in more detail.

States

In this section...

What Is a State?

A *state* describes a mode of a reactive Stateflow chart. States in a Stateflow chart represent these modes. The following table shows the button icon for a drawing a state in the Stateflow Editor and a short description.

Name	Button Icon	Description
State		Use a state to depict a mode of the system.

States can be active or inactive. When a state is active, the chart takes on that mode. When a state is inactive, the chart is not in that mode. The activity or inactivity of a chart's states dynamically changes based on events and conditions. The occurrence of events drives the execution of the Stateflow chart by making states become active or inactive. At any point in the execution of a Stateflow chart, there is a combination of active and inactive states.

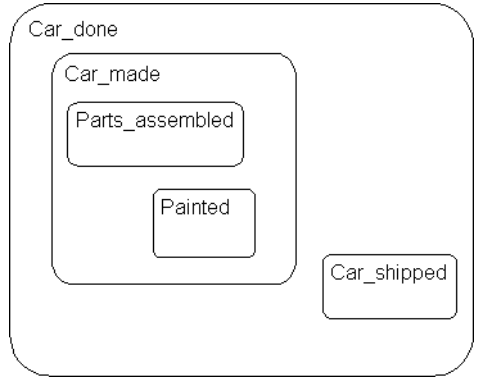
State Hierarchy

States can contain all other Stateflow objects except targets. 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 and event objects. The hierarchy of this containment appears in the Model Explorer. Data and event containment is defined by specifying the parent object when you create it. See Chapter 8, "Defining Data", Chapter 9, "Defining Events", and Chapter 16, "Defining Interfaces to Simulink Models and the MATLAB Workspace" for information and examples on representing data and event objects in the Model Explorer.

Representing State Hierarchy Example

In the following example, 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 Stateflow 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.

Stateflow hierarchy can also be represented textually, in which the Stateflow chart is represented by the slash (`/`) character and each level in the hierarchy of states is separated by the period (`.`) character. This 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`

State Decomposition

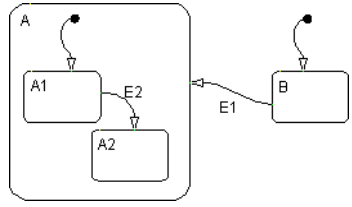
Every state (and chart) has a *decomposition* that dictates what kind of substates it can contain. All substates of a superstate must be of the same type as the superstate's decomposition. Decomposition for a state can be exclusive (OR) or parallel (AND). These types of decomposition are described in the following topics:

- "Exclusive (OR) State Decomposition" on page 2-7
- "Parallel (AND) State Decomposition" on page 2-7

Exclusive (OR) State Decomposition

Exclusive (OR) state decomposition for a superstate (or chart) is indicated when its substates have solid borders. Exclusive (OR) decomposition is used to describe system modes that are mutually exclusive. When a state has exclusive (OR) decomposition, only one substate can be active at a time. The children of exclusive (OR) decomposition parents are OR states.

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 any one time.

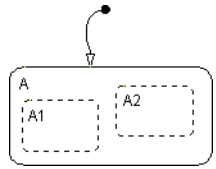


Parallel (AND) State Decomposition

The children of parallel (AND) decomposition parents are parallel (AND) states. Parallel (AND) state decomposition for a superstate (or chart) is indicated when its substates have dashed borders. This representation is

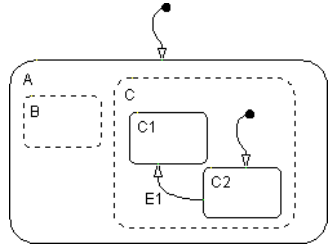
appropriate if all states at that same level in the hierarchy are always active at the same time.

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.

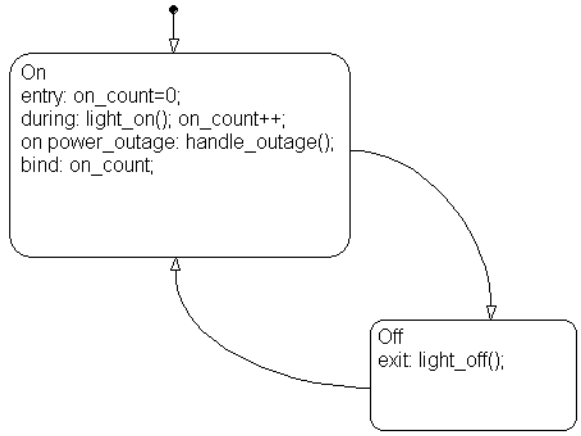


State Labels

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
bind:events, data
on event_name:on event_name actions
```

The following example demonstrates the components of a state label.



Each of the above actions is described in the subtopics that follow. For more information on state actions, see the following topics:

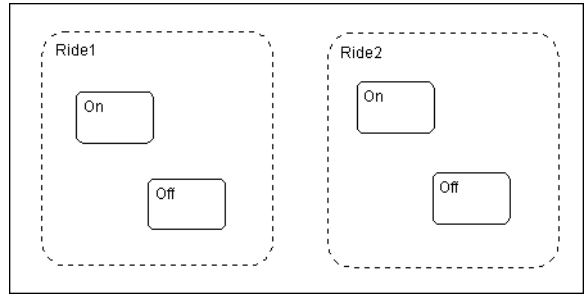
- "Entering, Executing, and Exiting a State" on page 3-33 — Describes how and when `entry`, `during`, `exit`, and `on` *`event_name`* actions are taken.
- "State Action Types" on page 10-2— Gives more detailed descriptions of each type of state action.

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 example, `Transmission` or `Green_on`.

The use of hierarchy provides some flexibility in the naming of states. The name that you enter as part of the label must be unique when preceded by its ancestor states. The name stored in the Stateflow hierarchy is the text you enter as the label on the state, preceded by the names of its parent states separated by periods. Each state can have the same name appear in the label of the state, as long as their full names within the Stateflow 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 Stateflow chart. Although the name portion of the label on these states is not unique, when the parent state is prefixed to the name in the hierarchy, the result is unique. The full names for these states are as follows:

- `Ride1.On`
- `Ride1.Off`
- `Ride2.On`
- `Ride2.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 by adding additional `on` *`event_name`* lines for different events.

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 Event_Name Action. Preceded by the prefix `on` *`event_name`*, where *`event_name`* is a unique event. 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.

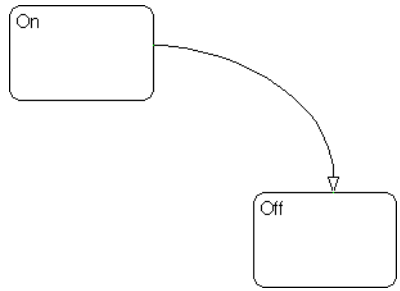
Bind Action. Preceded by the prefix `bind`. In the preceding label example, the data `on_count` is bound to the state `On`. This means that only the state `On` or a child of `On` can change the value of `on_count`. Other states, such as the state `Off`, can use `on_count` in its actions, but it cannot change its value in doing so.

Transitions

In this section...

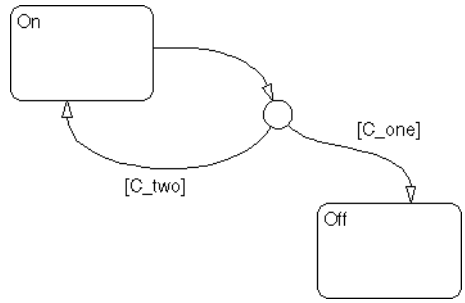
What Is a Transition?

A *transition* is a curved 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) object to another. A transition is attached to a source and a destination object. The *source* object is where the transition begins and the *destination* object is where the transition ends. This is an example of a transition from a source state, `On`, to a destination state, `Off`.



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.

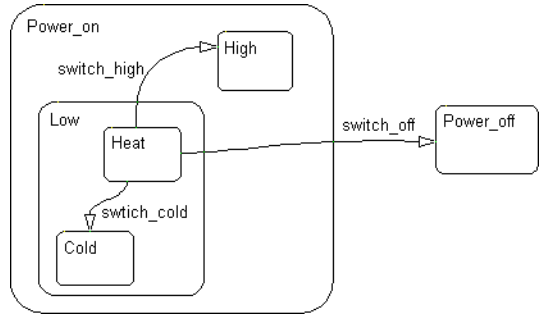
The following example has two segmented transitions: one from state `On` to state `Off`, and the other from state `On` to itself:



A default transition is a special type of transition that has no source object. See "Default Transitions" on page 2-25 for a description of a default transition.

Transition Hierarchy

Transitions cannot contain other objects like states can. However, transitions are contained by states. A transition's hierarchy is described in terms of the transition's parent, source, and destination. 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 Stateflow chart is represented by the `/` character. 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`

Transition Label	Transition Parent	Transition Source	Transition Destination
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

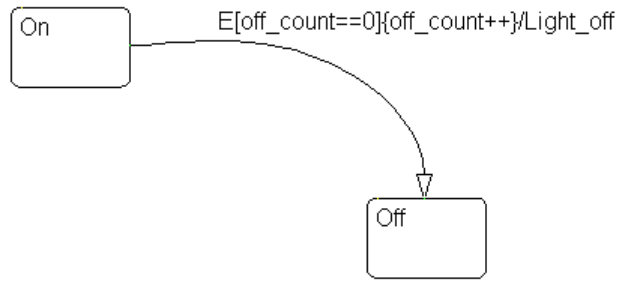
A transition is characterized by its *label*. The label can consist of an event, a condition, a condition action, and/or a transition action. The ? character is the default transition label. Transition labels have the following general format:

```
event[condition]{condition_action}/transition_action
```

You replace the names for *event*, *condition*, *condition_action*, and *transition_action* with appropriate contents as shown in the example "Transition Label Example" on page 2-14. Each part of the label is optional.

Transition Label Example

Use the following example to understand the parts of a transition label.



Event Trigger. Specifies an event that causes the transition to be taken, provided the condition, if specified, is true. Specifying an event is optional. The absence of an event indicates that the transition is taken upon the occurrence of any event. Multiple events are specified using the OR logical operator (|).

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

Condition. Specifies a boolean expression that, when true, validates a transition to be taken for the specified event trigger. Enclose the condition in square brackets (`[]`). See "Conditions" on page 10-9 for information on the condition notation.

In the preceding example, the condition `[off_count==0]` must evaluate as true for the condition action to be executed and for the transition from the source to the destination to be valid.

Condition Action. Follows the condition for a transition and is enclosed in curly braces (`{}`). It is executed as soon as the condition is evaluated as true and 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.

In the preceding example, if the condition `[off_count==0]` is true, the condition action `off_count++` is immediately executed.

Transition Action. Executes after the transition destination has been determined to be valid provided the condition, if specified, is true. If the transition consists of multiple segments, the transition action is only executed when the entire transition path to the final destination is determined to be valid. Precede the transition action with a `/`.

In the preceding example, if the condition `[off_count==0]` is true, and the destination state `Off` is valid, the transition action `Light_off` is executed.

Valid Transitions

In most cases, 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 are possible combinations of valid transition labels.

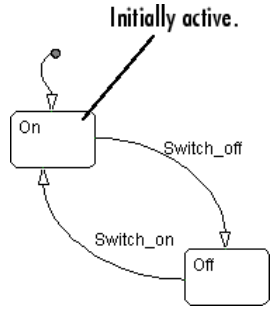
Transition Label	Is Valid If...
Event only	That event occurs
Event and condition	That event occurs and the condition is true
Condition only	Any event occurs and the condition is true
Action only	Any event occurs
Not specified	Any event occurs

Transition Connections

In this section...

Transitions to and from Exclusive (OR) States

This example shows simple transitions to and from exclusive (OR) states.

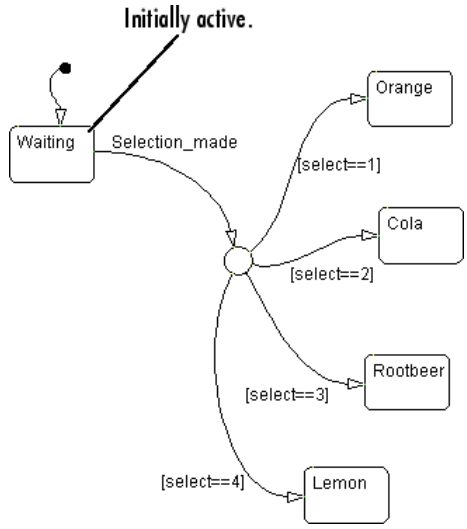


The transition `On→Off` is valid when state `On` is active and the event `Switch_off` occurs. The transition `Off→On` is valid when state `Off` is active and event `Switch_on` occurs.

See "Transitions to and from Exclusive (OR) States Examples" on page 3-54 for more information on the semantics of this notation.

Transitions to and from Junctions

This figure shows transitions to and from a connective junction.

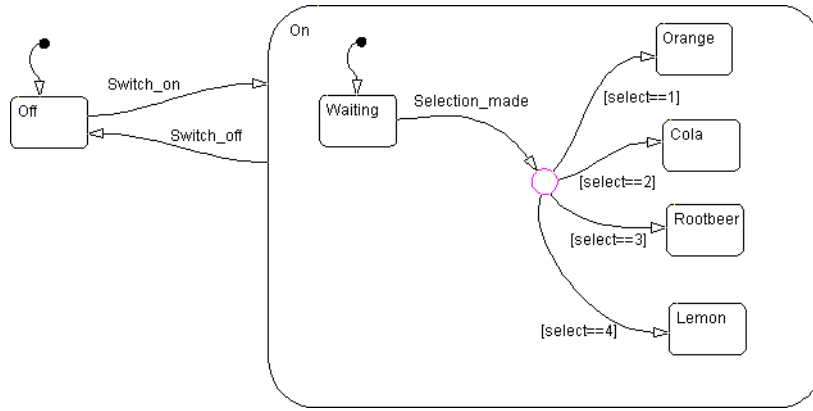


This example is a Stateflow chart of a soda machine. The Stateflow chart is called when the external event `Selection_made` occurs. The Stateflow chart awakens with the `Waiting` state active. The `Waiting` state is a common source state. When the event `Selection_made` occurs, the Stateflow chart transitions from the `Waiting` state to one of the other states based on the value of the variable `select`. One transition is drawn from the `Waiting` state to the connective junction. Four additional transitions are drawn from the connective junction to the four possible destination states.

See "Transitions from a Common Source to Multiple Destinations Example" on page 3-86 for more information on the semantics of this notation.

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.



This is an expansion of the soda machine Stateflow chart that includes the initial example of the `On` and `Off` exclusive (OR) states. `On` is now a superstate containing the `Waiting` and soda choices states. The transition `Off`→`On` is valid when state `Off` is active and event `Switch_on` occurs. Now that `On` is a superstate, this is an explicit transition to the `On` superstate.

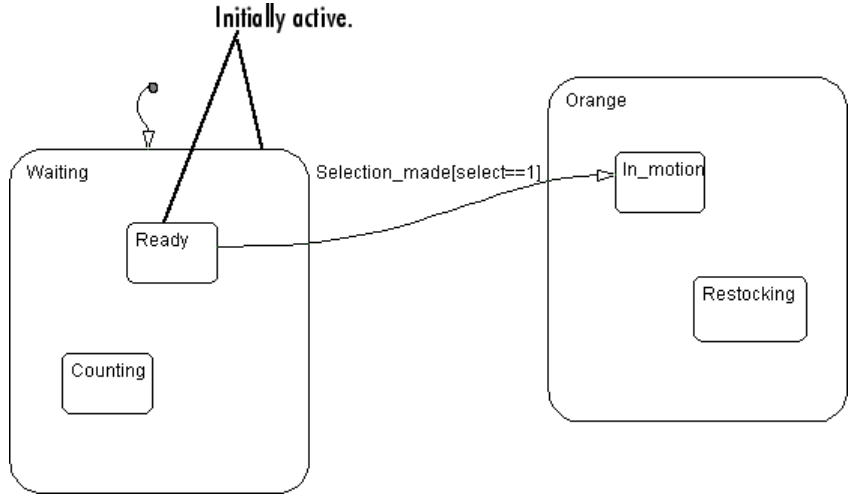
For a transition to a superstate to be valid, the destination substate must be implicitly defined. The destination substate for `On` is implicitly defined by making the `Waiting` substate the destination state of a default transition. This notation defines that the resultant transition is made from the `Off` state to the state `On.Waiting`.

The transition from `On` to `Off` is valid when state `On` is active and event `Switch_off` occurs. However, when the `Switch_off` event occurs, a transition to the `Off` state must take place no matter which of the substates of `On` is active. This top-down approach simplifies the Stateflow chart by looking at the transitions out of the superstate without considering all the details of states and transitions within the superstate.

See “Default Transition Examples” on page 3-66 for more information on the semantics of this notation.

Transitions to and from Substates

The following example shows transitions to and from exclusive (OR) substates.

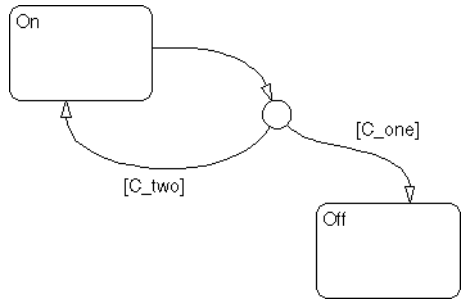


This Stateflow chart shows a transition from one OR substate to another OR substate: the transition from `Waiting.Ready` to `Orange.In_motion`. The transition to the state `In_motion` is valid when state `Waiting.Ready` is active and the event `Selection_made` occurs, providing that the variable `select` equals 1. This transition defines an explicit exit from the `Waiting.Ready` state and an implicit exit from the `Waiting` superstate. On the destination side, this transition defines an implicit entry into the `Orange` superstate and an explicit entry into the `Orange.In_motion` substate.

See "Transitioning from a Substate to a Substate with Events Example" on page 3-58 for more information on the semantics of this notation.

Self-Loop Transitions

A transition segment from a state to a connective junction that has an outgoing transition segment from the connective junction back to the state is a self-loop transition as shown in the following example:



See these sections for examples of self-loop transitions:

- "Connective Junction — Self-Loop Example" on page 2-32

  See "Self-Loop Transition Example" on page 3-82 for information on the semantics of this notation.

- "Connective Junction and For Loops Example" on page 2-33

  See "For-Loop Construct Example" on page 3-83 for information on the semantics of this notation.

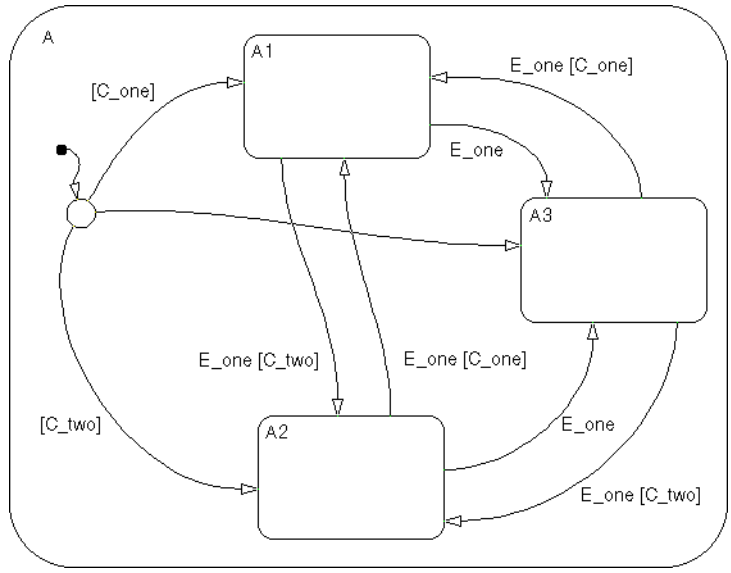
Inner Transitions

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" on page 2-21
- "After Using an Inner Transition to a Connective Junction" on page 2-22
- "Using an Inner Transition to a History Junction" on page 2-23

Before Using an Inner Transition

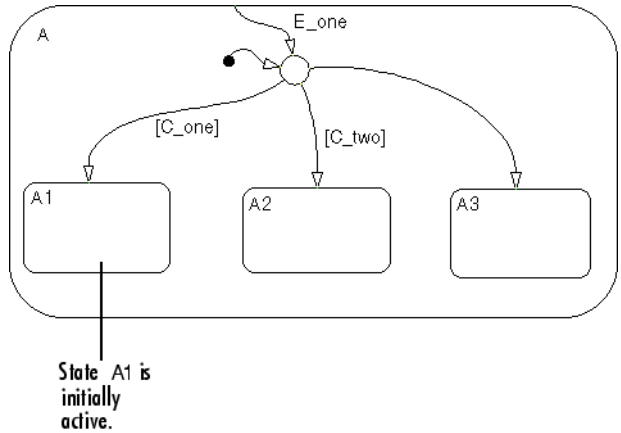
This is an example of a Stateflow chart that you can simplify by using an inner transition.



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 `[C_one]` and `[C_two]`. If `[C_one]` is true, the transition to `A1` is true. If `[C_two]` is true, the transition to `A2` is valid. If neither `[C_one]` nor `[C_two]` is true, the transition to `A3` is valid. The transitions among `A1`, `A2`, and `A3` are determined by `E_one`, `[C_one]`, and `[C_two]`.

After Using an Inner Transition to a Connective Junction

This example simplifies the preceding example using an inner transition to a connective junction.



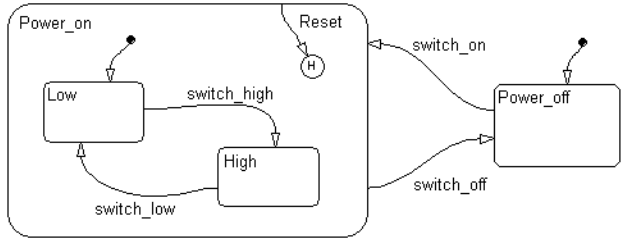
Any event occurs and awakens the Stateflow chart. The default transition to the connective junction is valid. The destination of the transitions is determined by `[C_one]` and `[C_two]`.

You can simplify the Stateflow chart by using an inner transition in place of the many 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_one` occurs, the inner transition is potentially valid. If `[C_one]` is true, the transition to `A1` is valid. If `[C_two]` is true, the transition to `A2` is valid. If neither `[C_one]` nor `[C_two]` is true, the transition to `A3` is valid. This solution is simpler than the previous one.

See "Processing the First Event with an Inner Transition to a Connective Junction" on page 3-74 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 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 "Use of History Junctions Example" on page 2-37 for another example using a history junction.

See "Inner Transition to a History Junction Example" on page 3-77 for more information on the semantics of this notation.

Default Transitions

In this section...

What Is a Default Transition?

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 using the Debugger with the **State Inconsistencies** option enabled.

This table shows the button icon and briefly describes a default transition.

Name	Button Icon	Description
Default transition		Use a default transition to indicate, when entering this level in the hierarchy, which object becomes active by default.

Labeling Default Transitions

In some circumstances, you might want to 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.

Note When labeling default transitions, take care to ensure that there is always at least one valid default transition. Otherwise, a Stateflow chart can transition into an inconsistent state.

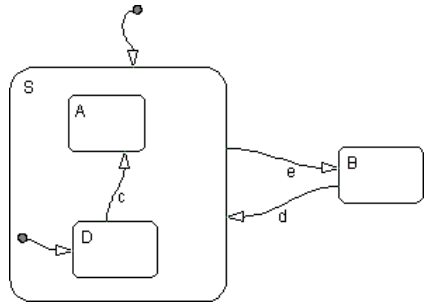
Default Transition Examples

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

- “Default Transition to a State Example” on page 2-26
- “Default Transition to a Junction Example” on page 2-27
- “Default Transition with a Label Example” on page 2-28

Default Transition to a State Example

This example shows a use of default transitions.



When the Stateflow chart first wakes up, it must decide whether to activate state `S` or state `B` since they are exclusive (OR) states. The answer is given by the default transition to superstate `S`, which is taken if valid. Because there are no conditions on this default transition, it is taken.

State `S`, which is now active, has two substates, `A` and `D`. Which substate becomes active? Only one of them can be active because they are exclusive (OR) states. The answer is given by the default transition to substate `D`, which is taken if valid. Because there are no conditions on this default transition, it is taken.

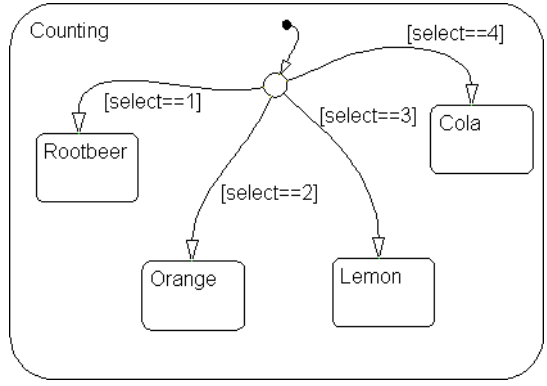
Suppose at a different execution point, the Stateflow chart is awakened by the occurrence of event `d` and state `B` is active. The transition from state `B` to state `S` is valid. When the system enters state `S`, it enters substate `D` because the default transition is defined.

See “Default Transition Examples” on page 3-66 for more information on the semantics of this notation.

The default transitions are required for the Stateflow chart to execute. Without the default transition to state `S`, when the Stateflow chart wakes up, none of the states becomes active. You can detect this situation at run-time by checking for state inconsistencies. See “Animating Stateflow Charts in Normal Mode” on page 23-3 for more information.

Default Transition to a Junction Example

This example shows a default transition to a connective junction.

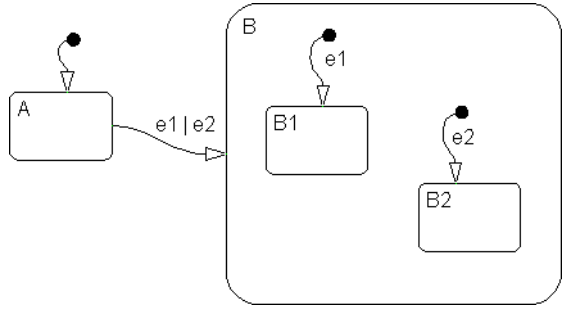


In this example, the default transition to the connective junction defines that upon entering the `Counting` state, the destination is determined by the condition on each transition segment.

See "Default Transition to a Junction Example" on page 3-67 for more information on the semantics of this notation.

Default Transition with a Label Example

The following example shows the labeling of default transitions.



If state `A` is initially active and either `e1` or `e2` occurs, the transition from state `A` to superstate `B` is valid. The substates `B1` and `B2` both have default transitions. The default transitions are labeled to specify the event that triggers the transition. If event `e1` occurs, the transition `A` to `B1` is valid. If event `e2` occurs, the transition `A` to `B2` is valid.

See “Labeled Default Transitions Example” on page 3-69 for more information on the semantics of this notation.

Connective Junctions

In this section...

What Is a Connective Junction?

The 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 "Connective Junction Examples" on page 3-79 for a summary of the semantics of connective junctions.

Flow Graph Notation with Connective Junctions

Flow graph 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,

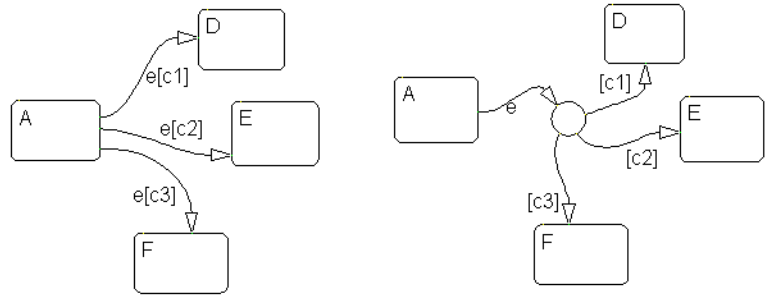
flow graph notation produces efficiently generated code that helps optimize memory use.

Flow graph notation employs combinations of the following:

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

Flow graph notation, states, and state-to-state transitions seamlessly coexist in the same Stateflow chart. The key to representing flow graph notation is in the labeling of the transitions (specifically the use of action language) as shown by the following examples.

Connective Junction with All Conditions Specified Example



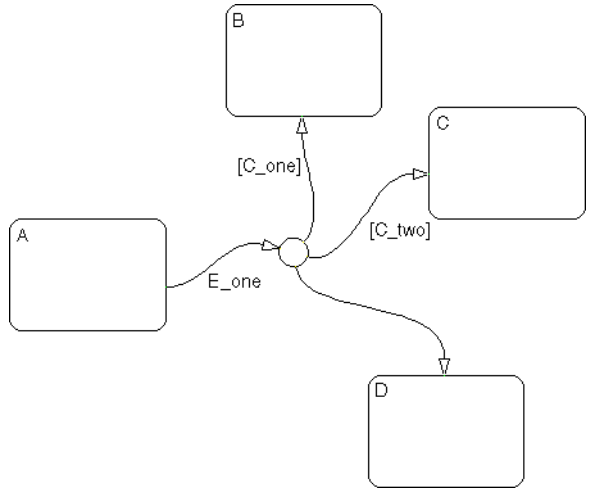
In the example on the left, if state `A` is active when event `e` occurs, the transition from state `A` to any of states `D`, `E`, or `F` takes place if one of the conditions `[c1]`, `[c2]`, or `[c3]` is met.

In the equivalent representation on the right, a transition from the source state to a connective junction is labeled by the event. Transitions from the connective junction to the destination states are labeled by the conditions. If state `A` is active when event `e` occurs, the transition from `A` to the connective junction occurs first. The transition from the connective junction to a destination state follows based on which of the conditions `[c1]`, `[c2]`, or `[c3]` is true. If none of the conditions is true, no transition occurs and state `A` remains active.

See "If-Then-Else Decision Construct Example" on page 3-80 for more information on the semantics of this notation.

Connective Junction with One Unconditional Transition Example

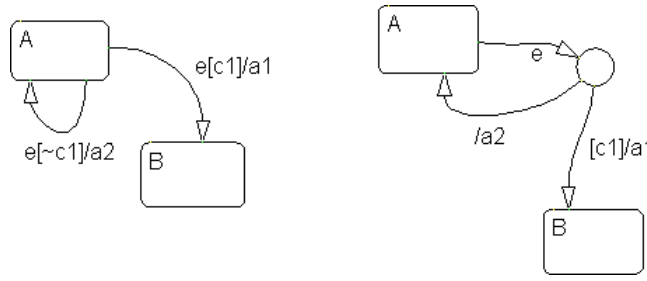
The transition A to B is valid when A is active, event E_one occurs, and [C_one] is true. The transition A to C is valid when A is active, event E_one occurs, and [C_two] is true. Otherwise, given A is active and event E_one occurs, the transition A to D is valid. If you do not explicitly specify condition [C_three], it is implicit that the transition condition is not [C_one] and not [C_two].



See "If-Then-Else Decision Construct Example" on page 3-80 for information on the semantics of this notation.

Connective Junction — Self-Loop Example

In some situations, the transition event occurs but a condition is not met. No transition is taken, but an action is generated. You can represent this situation by using a connective junction or a self-loop transition (transition from state to itself).



In the example on the left, if State `A` is active and event `e` occurs and the condition `[c1]` is met, the transition from `A` to `B` is taken, generating action `a1`. The transition from state `A` to state `A` is valid if event `e` occurs and `[c1]` is not true. In this self-loop transition, the system exits and reenters state `A`, and executes action `a2`.

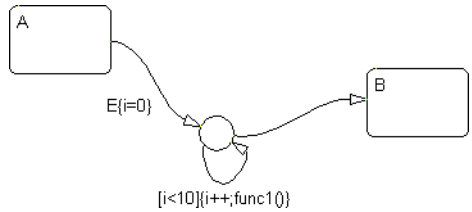
In the equivalent representation on the right, the use of a connective junction makes it unnecessary to specify the implied condition `[~c1]` explicitly.

See "Self-Loop Transition Example" on page 3-82 for more information on the semantics of this notation.

Connective Junction and For Loops Example

This example shows a combination of flow graph notation and state transition notation. Self-loop transitions to connective junctions can be used to represent `for` loop constructs.

In state `A`, event `E` occurs. The transition from state `A` to state `B` is valid if the conditions along the transition path are true. The first segment of the transition does not have a condition, but does have a condition action. The condition action, `{i=0}`, is executed. The condition on the self-loop transition is evaluated as true and the condition actions `{i++;func1()}` execute. The condition actions execute until the condition `[i<10]` is false. The condition actions on both the first segment and the self-loop transition to the connective junction effectively execute a `for` loop (for `i` values 0 to 9 execute `func1()`). The `for` loop is executed outside the context of a state. The remainder of the path is evaluated. Because there are no conditions, the transition completes at the destination, state `B`.



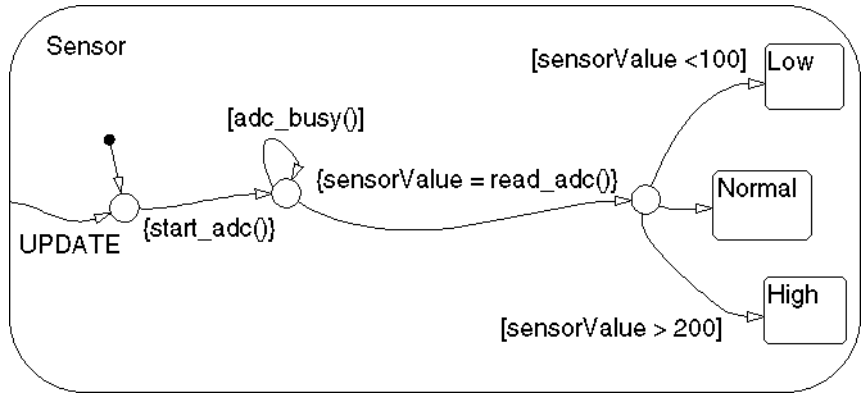
See "For-Loop Construct Example" on page 3-83 for information on the semantics of this notation.

Flow Graph Notation Example

This example shows a real-world use of flow graph notation and state transition notation. This Stateflow chart models an 8-bit analog-to-digital converter (ADC).

Consider the case when state `Sensor.Low` is active and event `UPDATE` occurs. The inner transition from `Sensor` to the connective junction is valid. The next transition segment has a condition action, `{start_adc()}`, which initiates a reading from the ADC. The self-loop on the second connective junction repeatedly tests the condition `[adc_busy()]`. This condition evaluates as true once the reading settles (stabilizes) and the loop completes. This self-loop transition is introduces the delay needed for the ADC reading to settle. The delay could have been represented by another state with a counter. Using flow graph notation in this example avoids an unnecessary use of a state and produces more efficient code.

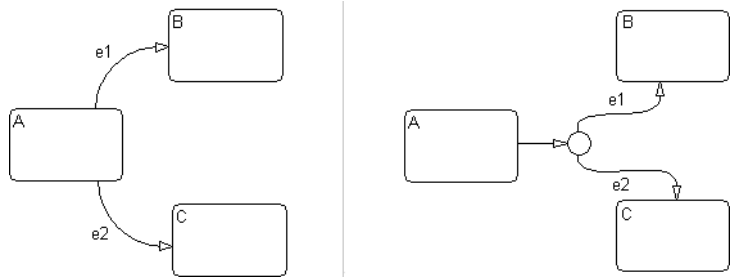
The next transition segment condition action, {`sensorValue=read_adc()`}, puts the new value read from the ADC in the data object `sensorValue`. The final transition segment is determined by the value of `sensorValue`. If `[sensorValue <100]` is true, the state `Sensor.Low` is the destination. If `[sensorValue >200]` is true, the state `Sensor.High` is the destination. Otherwise, state `Sensor.Normal` is the destination state.



See "Flow Graph Notation Example" on page 3-84 for information on the semantics of this notation.

Connective Junction from a Common Source to Multiple Destinations Example

Transitions `A` to `B` and `A` to `C` share a common source state `A`. An alternative representation uses one arrow from `A` to a connective junction, and multiple arrows labeled by events from the junction to the destination states `B` and `C`.



See "Transitions from a Common Source to Multiple Destinations Example" on page 3-86 for information on the semantics of this notation.

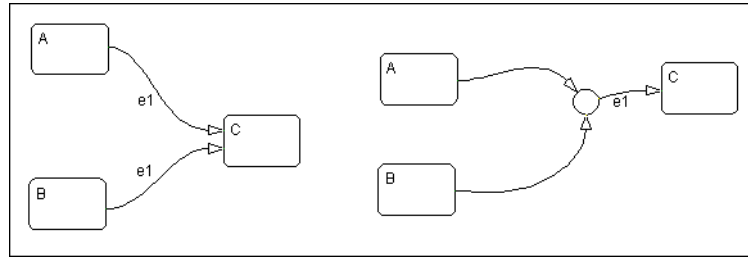
Connective Junction Common Events Example

Suppose, for example, that when event `e1` occurs, the system, whether it is in state `A` or `B`, transfers to state `C`. Suppose that transitions `A` to `C` and `B` to `C`

are triggered by the same event `e1`, so that both destination state and trigger event are common to the transitions. There are three ways to represent this:

- By drawing transitions from `A` and `B` to `C`, each labeled with `e1`
- By placing `A` and `B` in one superstate `S`, and drawing one transition from `S` to `C`, labeled with `e1`
- By drawing transitions from `A` and `B` to a connective junction, then drawing one transition from the junction to `C`, labeled with `e1`

This Stateflow chart shows the simplification using a connective junction.



See “Transitions from a Source to a Destination Based on a Common Event Example” on page 3-88 for information on the semantics of this notation.

History Junctions

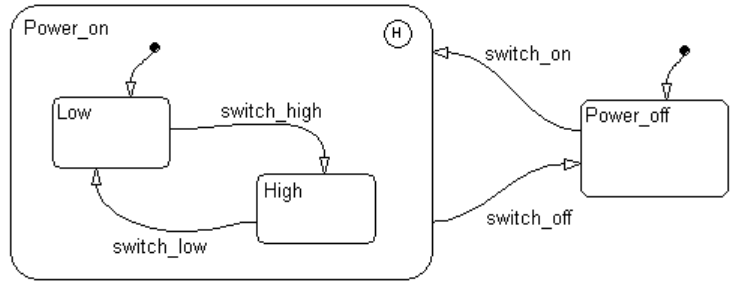
In this section...

What Is a History Junction?

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.

Use of History Junctions Example

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 could enter either `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. Since `Power_on.High` was the last active state, it becomes active again. After the first time `Power_on`

becomes active, the choice between entering `Power_on.Low` or `Power_on.High` is determined by the history junction.

See "Default Transition and a History Junction Example" on page 3-68 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 and/or condition, the active state is to be exited and then immediately reentered.

See "Using an Inner Transition to a History Junction" on page 2-23 for an example of this notation.

See "Inner Transition to a History Junction Example" on page 3-77 for more information on the semantics of this notation.

Graphical Functions

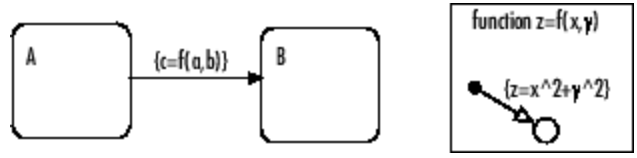
In this section...

What Is a Graphical Function?

A graphical function is a function defined graphically by a flow graph that includes Stateflow action language.

Example of Using a Graphical Function

This figure shows a graphical function side by side in a Stateflow chart with the transition that calls it:



In this example, the function `z = f(x,y)` is called in the condition action of the transition from state `A` to state `B`. The function is defined using symbols that are valid only within the function itself. The function is called using data objects available to states `A` and `B` and their parent states (if any).

Advantages of Using Graphical Functions

Graphical functions are similar to textual functions such as C and MATLAB functions in these ways:

- Graphical functions can accept arguments and return results.
- You can invoke graphical functions in transition and state actions.

Unlike C and MATLAB functions, however, graphical functions are native Stateflow graphical objects. You use the Stateflow Editor to create them,

and they reside in your Stateflow chart. This property makes graphical functions easier to create, access, and manage than textual custom code functions, whose creation requires external tools, and whose definition resides separately from the chart.

For more information, see “Using Graphical Functions to Extend Actions” on page 7-27.

Boxes

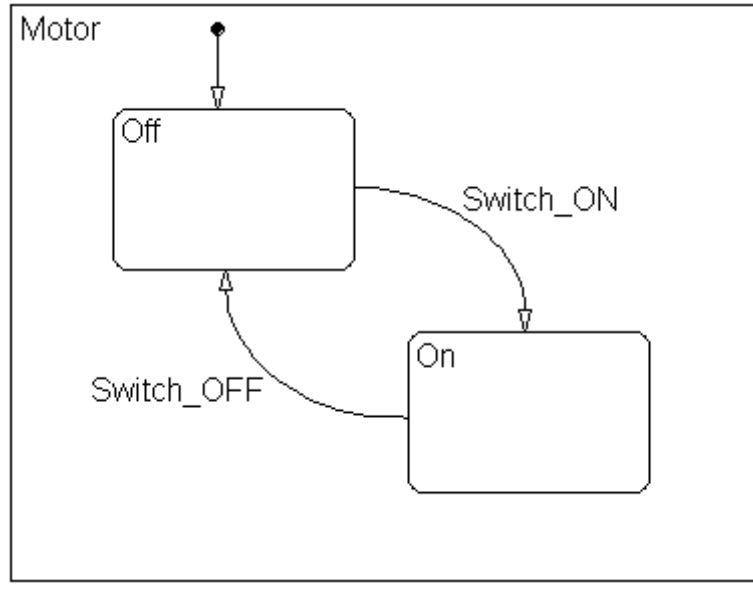
In this section...

What Is a Box?

A box is a graphical object that organizes other objects in your chart, such as functions and states.

Example of Using a Box

In this example, the box `Motor` groups together related states `Off` and `On`.



For rules of using boxes and other examples, see "Using Boxes to Extend Charts" on page 7-47.

3

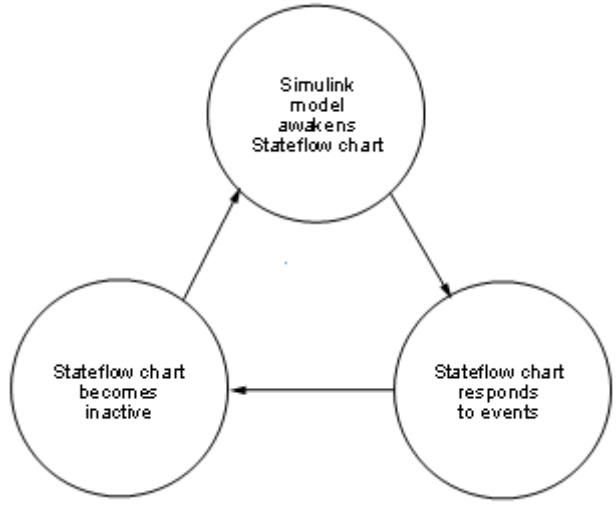
Stateflow Chart Semantics

Executing an Event

In this section...

How Stateflow Charts Respond to Events

Stateflow charts execute only 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. This means that all activity caused by the event in the chart is completed before returning to whatever activity 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 in the action language, as described in "Broadcasting Events in Actions" on page 10-50. For examples using event broadcasting and directed event broadcasting, see the following:

- "Condition Actions to Broadcast Events to Parallel (AND) States Example" on page 3-64
- "Cyclic Behavior to Avoid with Condition Actions Example" on page 3-64
- "Event Broadcast State Action Example" on page 3-93
- "Event Broadcast Transition Action with a Nested Event Broadcast Example" on page 3-96
- "Event Broadcast Condition Action Example" on page 3-100
- Directed Event Broadcasting

Events have hierarchy (a parent) and scope. The parent and scope together define a range of access to events. It is primarily the event's parent that determines who can trigger on the event (has receive rights). See the **Name** and **Parent** fields for an event in "Setting Properties for an Event" on page 9-7 for more information.

Processing Events

Stateflow charts process events from the top down through the hierarchy of the chart, as follows:

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 is executed (see "Executing an Active Chart" on page 3-6 and "Executing an Active State" on page 3-34). (The event *receiver* is the parent of the event unless the event was explicitly directed to a *receiver* 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.

For an understanding of early return logic, see "Early Return Logic for Event Broadcasts" on page 3-49.

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 "Defining Edge-Triggered Output Events" on page 16-24.

Executing a Chart

In this section...

Lifecycle 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 "Executing an Event" on page 3-2.

Executing 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 graphs (see "Executing a Set of Flow Graphs" on page 3-19). If this action does not cause an entry into a state and the chart has parallel decomposition, then each parallel state is entered (see "Entering a State" on page 3-33).

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

Executing an Active Chart

After a chart has been triggered the first time by the Simulink model, it is an active chart. When it 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 children are executed. Parallel states are executed in the same order that they are entered.

Executing a Chart with Super Step Semantics

About Super Step Semantics

By default, Stateflow charts execute once for each active input event. If there are no input events, then the charts execute once every time step. If you are modeling a system that must react quickly to inputs, you can enable super step semantics, a Stateflow chart property (see "Enabling Super Step Semantics" on page 3-7).

When you enable super step semantics, a Stateflow chart executes multiple times for every active input event or for every time step in which there are no input events. This means that the chart takes valid transitions until *either* of these conditions occurs:

- There are no more valid transitions, that is, the chart reached 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. 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.

Enabling Super Step Semantics

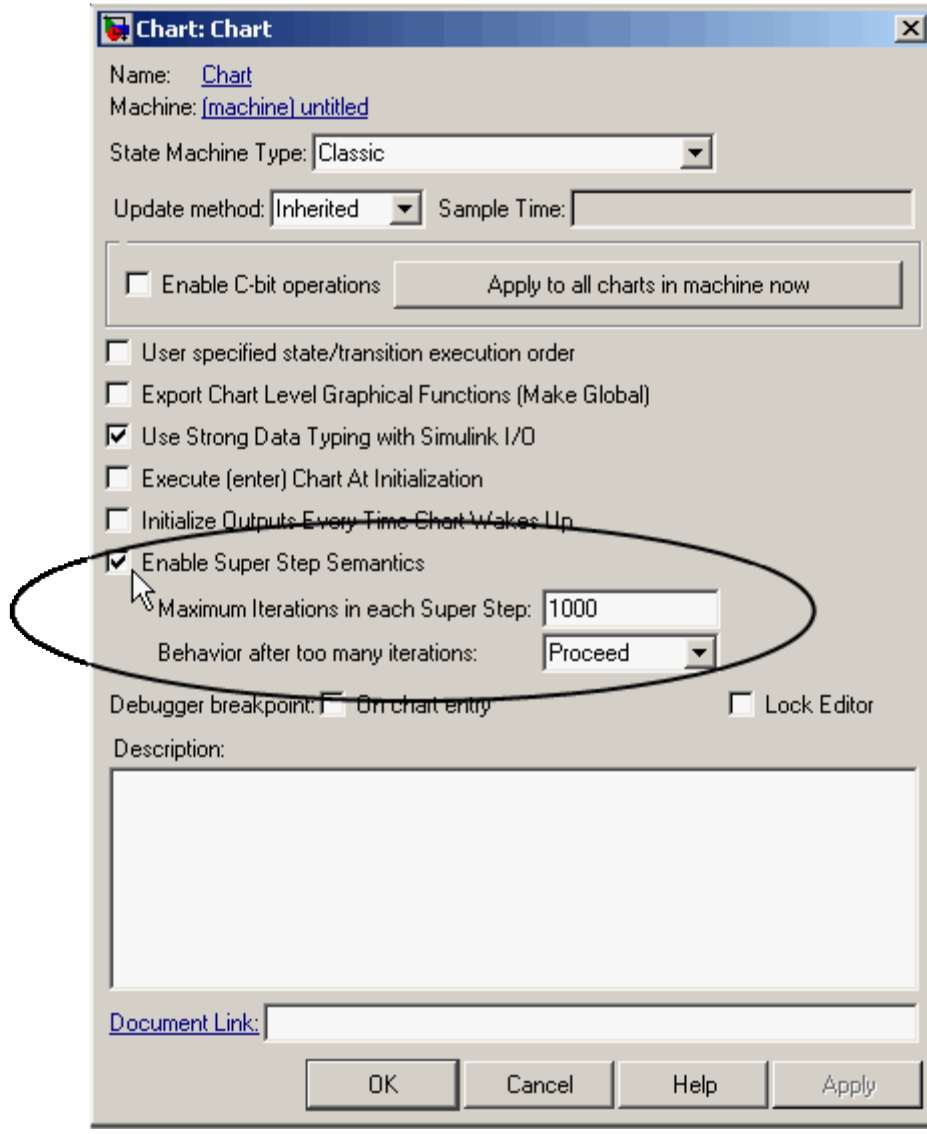
To enable super step semantics, follow these steps:

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

   The Chart properties dialog box opens on your desktop.

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

   Two additional fields appear, as shown:



3 Enter a value in the field **Maximum Iterations in Each Super Step**.

The value you enter is the maximum number of transitions a Stateflow chart can take in one super step. 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.

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

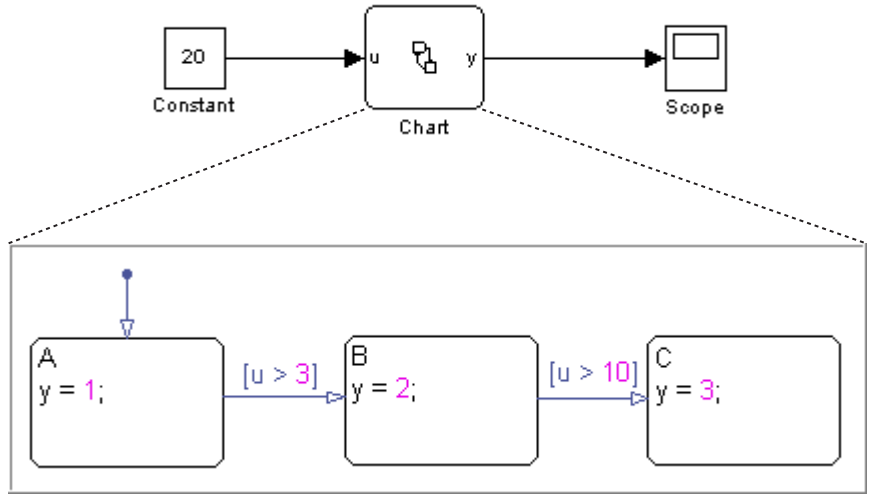
Your selection determines how the chart behaves during simulation if it exceeds the maximum number of iterations in the super step before reaching a stable state. Here are the options:

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** Choosing **Throw Error** can help detect infinite loops in transition cycles (see "Detecting Infinite Loops in Transition Cycles" on page 3-15.

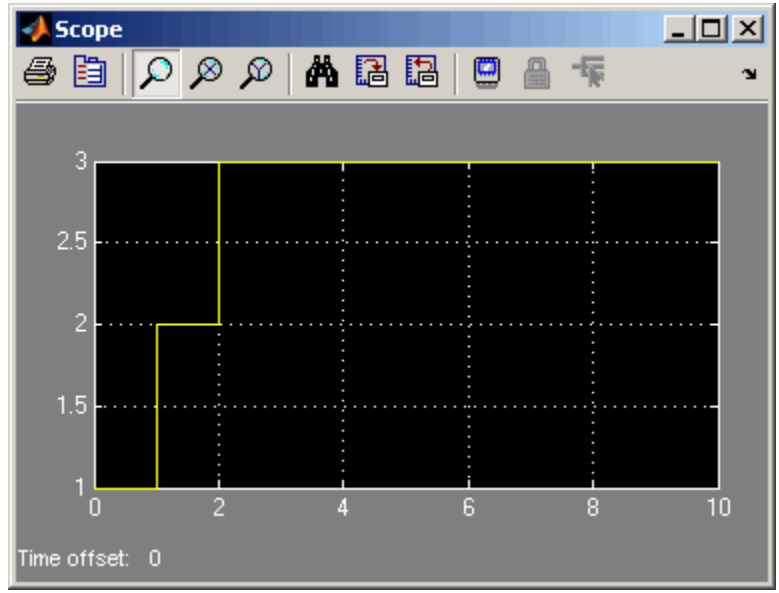
Note This option is relevant only for simulation targets. For embedded targets, code generation goes to the next time step rather than generating an error.

Super Step Example

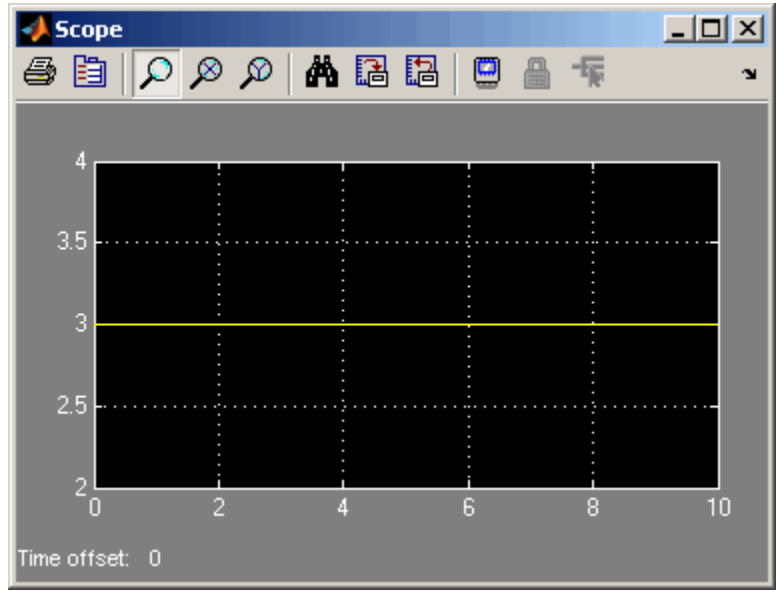
Here is a model that illustrates how super step semantics differs from Classic Stateflow chart semantics:



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 is valid. In Classic Stateflow chart semantics, the chart takes only one transition in each simulation step, incrementing y each time.

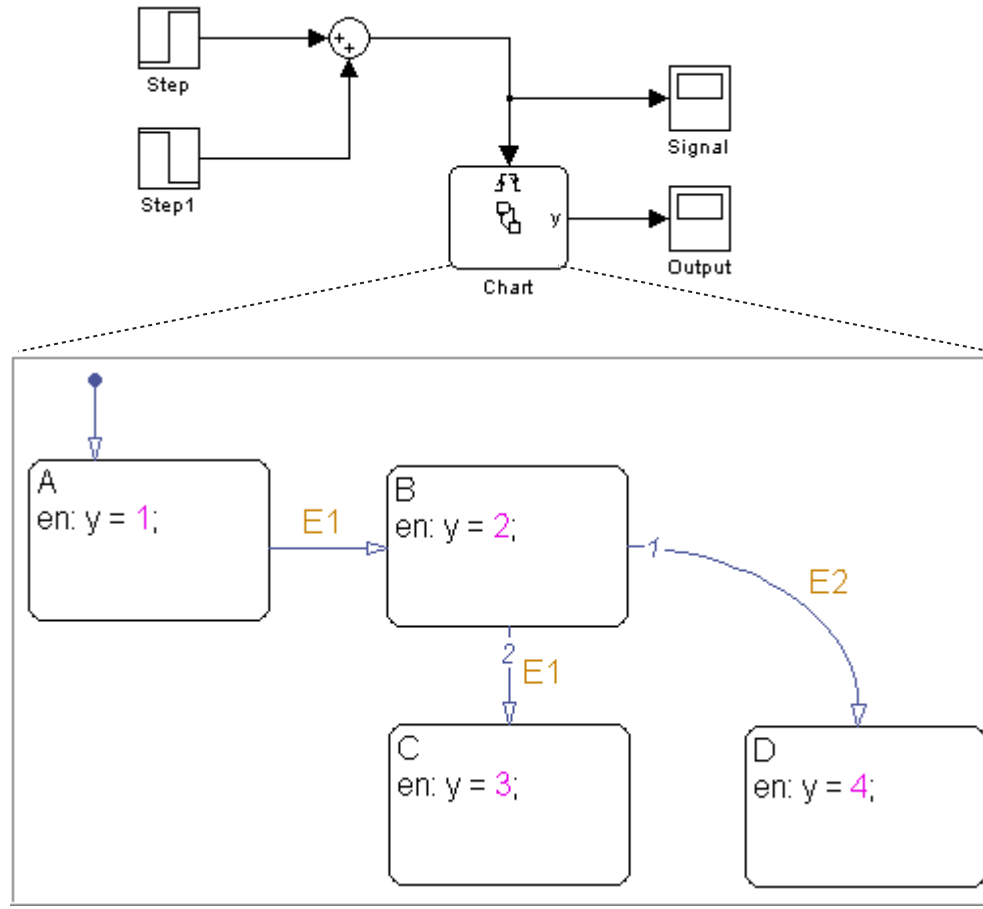


When you enable super step semantics, the chart takes all valid transitions in each time step, stopping at state `C` with `y` = 3.

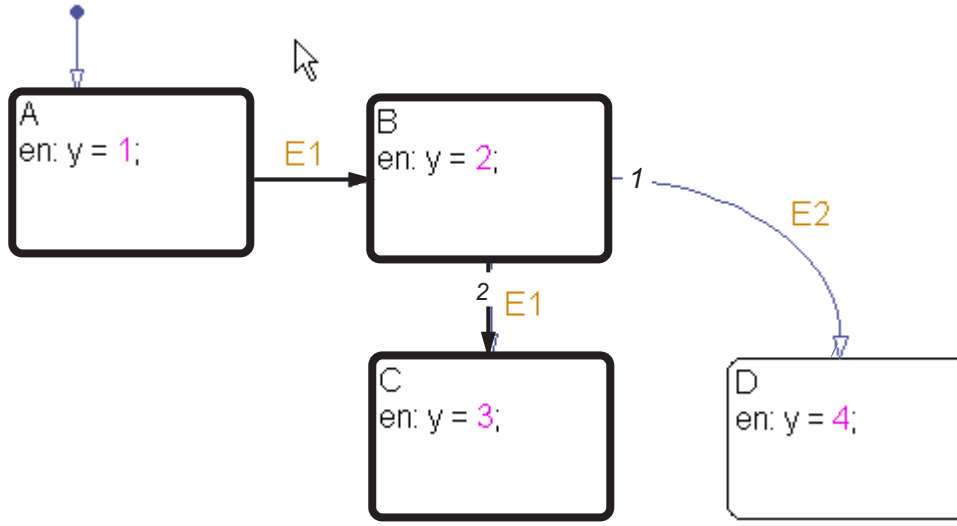


How Super Step Semantics Works with Multiple Input Events

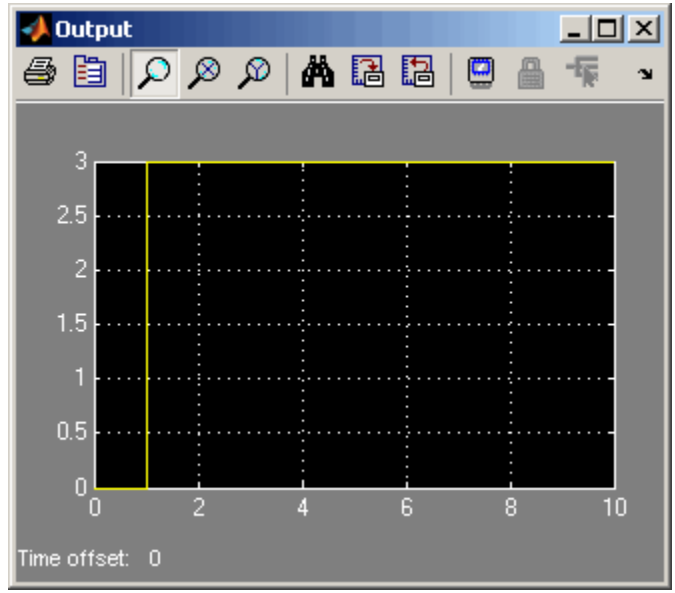
When you enable super step semantics for a Stateflow chart with multiple active input events, the chart takes all valid transitions for the first active event before it begins processing the next active event. For example, consider the following model:



In this model, the Step 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 Simulink model wakes up the Stateflow chart, events `E1` and `E2` are both active. If you enable super step semantics, the chart takes all valid transitions for event `E1`, as shown in the highlighted path:



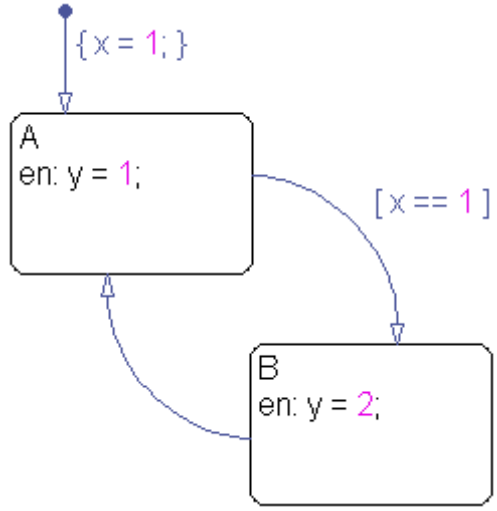
As you can see, the chart takes transitions from state `A` to `B` and then from state `B` to `C` in a single super step. The scope shows that `y` = 3 at the end of the super step:



In a super step, this chart never transitions to state `D` because there is no path from state `C` to state `D`.

Detecting 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 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 "Enabling Super Step Semantics" on page 3-7.

Executing a Chart at Initialization

By default, the first time a chart wakes up it executes the default transition paths. At this time it 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 option to *execute at initialization*. When you turn on this option, a chart's state configuration initializes at time 0 instead of at the first occurrence of an input event. The default transition paths of the chart are executed during the model initialization phase at time 0, corresponding to the mdlInitializeConditions() phase for S-functions.

You enable the option **Execute (enter) Chart At Initialization** in chart properties, as described in "Setting Properties for Individual Charts" on page 16-5.

Note If an output of this chart connects to a SimEvents® block, do not enable this option. To learn more about using Stateflow charts and SimEvents blocks together in a model, see the SimEvents software 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 the blocks connected to Stateflow chart input ports may not have initialized their outputs yet.
- Do not call exported graphical functions from other charts because those charts may not have been initialized yet.
- Do not broadcast function-call output events because the triggered subsystems may have not been initialized yet.

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

Executing a Transition

In this section...

About Transitions

Transitions play a large role in defining the animation or execution of a system. If your chart has exclusive (OR) states, its execution begins with the default transitions that point to the first active states in your chart.

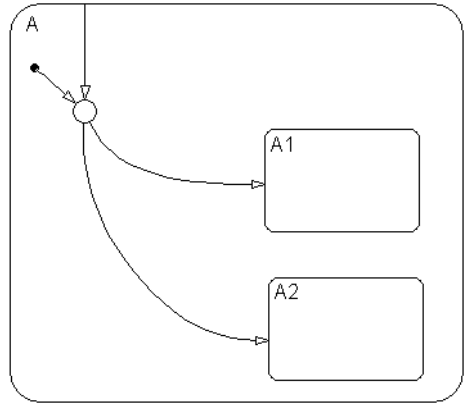
Transitions have sources and destinations; thus, any actions associated with the sources or destinations are related to the transition that joins them. The type of the source and destination is equally important to define the semantics.

Transition Flow Graph Types

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

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

Each set of flow graphs 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 graphs includes:

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

In addition, state A's set of default flow graphs 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 become members of more than one flow graph type.

Executing a Set of Flow Graphs

Each flow graph group is executed in the order of group priority until a valid transition is found. The default transitions group is executed first, followed by the outer transitions group and then the inner transitions group. Each flow graph group is executed with the following procedure.

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 "Evaluation Order for Outgoing Transitions" on page 3-21.

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. No more transition segments are tested and a transition path is formed by including the transition segment from each preceding junction back to the starting transition.
   b. The states that are the immediate children of the parent of the transition path are exited (see "Exiting an Active State" on page 3-35).
   c. The transition action for the final transition segment of the full transition path is executed.
   d. The destination state is entered (see "Entering a State" on page 3-33).
6. If the destination is a junction with no outgoing transition segments, do the following:
   a. Testing stops without any states being exited or entered.
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. Back up 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 graphs completes execution when all starting transitions have been tested.

Evaluation Order for Outgoing Transitions

In this section...

What Does Ordering Mean for Outgoing Transitions?

When multiple transitions originate from a single source (such as a state or junction), a Stateflow chart must determine in which order to evaluate those transitions. Order of evaluation depends on:

- Explicit ordering

  Specify explicitly the evaluation order of outgoing transitions on an individual basis (see "Explicit Ordering of Outgoing Transitions" on page 3-22).

- Implicit ordering

  Override explicit ordering by letting a Stateflow chart use internal rules to order transitions (see "Implicit Ordering of Outgoing Transitions" on page 3-27).

Note You can order transitions only within their type (inner, outer, or default). For more information, see "Transition Flow Graph Types" on page 3-18.

Outgoing transitions are assigned priority numbers based on order of evaluation. The lower the number, the higher the priority. The priority number appears on each outgoing transition.

Because evaluation order is a chart property, all outgoing transitions 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 Outgoing Transitions

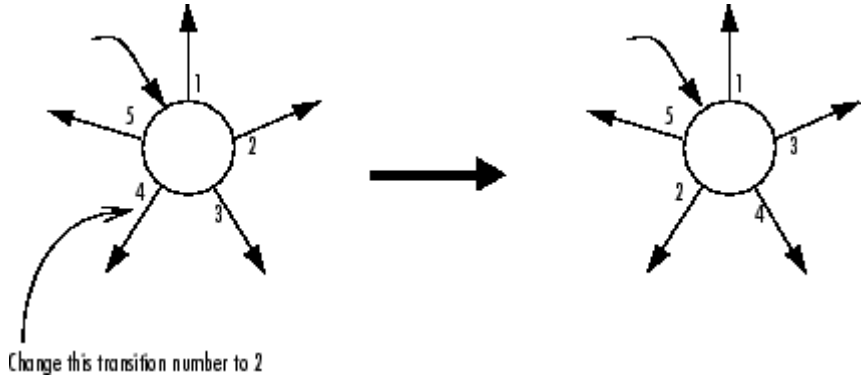
By default, a Stateflow chart orders outgoing transitions explicitly based on evaluation priorities you set.

How Explicit Ordering Works

When you open a new Stateflow chart, all outgoing transitions from a source are automatically numbered in the order you create them, starting with the next available number for the source.

You can change the order of outgoing transitions by explicitly renumbering them. When you change a transition number, the Stateflow chart automatically renumbers the other outgoing transitions for the source by preserving their relative order. This behavior is consistent with the renumbering rules for Simulink ports.

For example, if you have a source with five outgoing transitions, changing transition 4 to 2 results in the automatic renumbering shown.



Automatic Renumbering of Transitions During Explicit Reordering

Using Explicit Ordering for Transitions

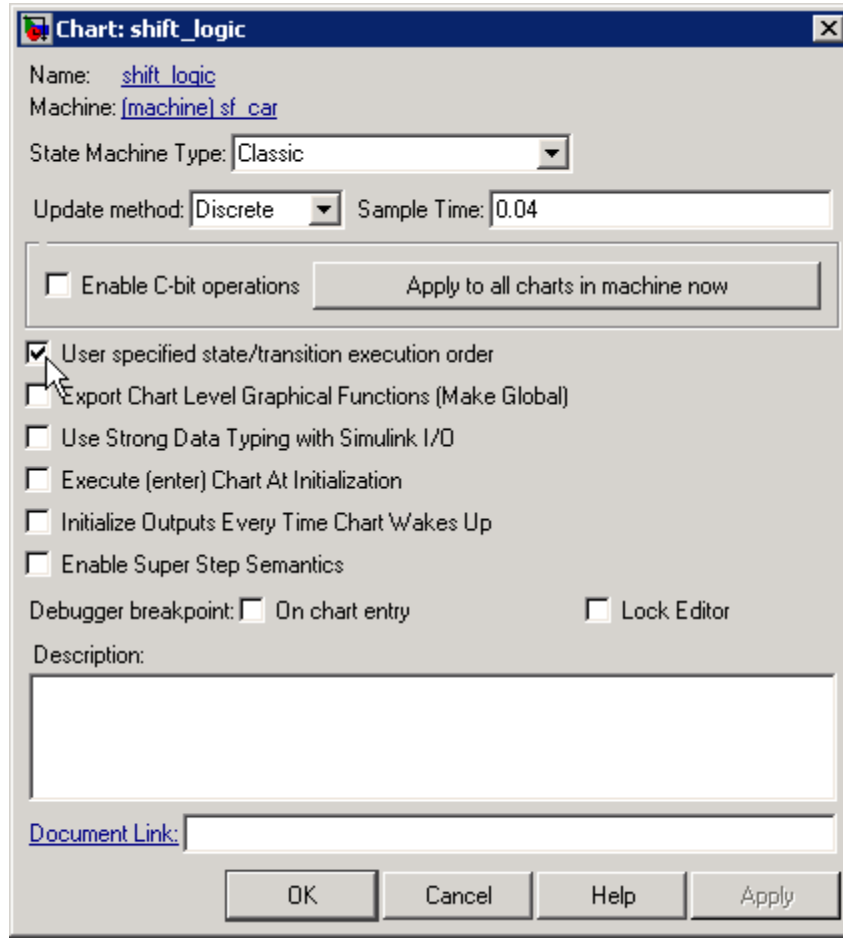
To use explicit ordering for transitions, perform these tasks:

Enabling Explicit Ordering at the Chart Level. To enable explicit ordering for transitions, follow these steps:

1 In the Stateflow Editor, from the **File** menu, select **Chart Properties**.

The Chart properties dialog box appears.

2 Select the **User specified state/transition execution order** check box.



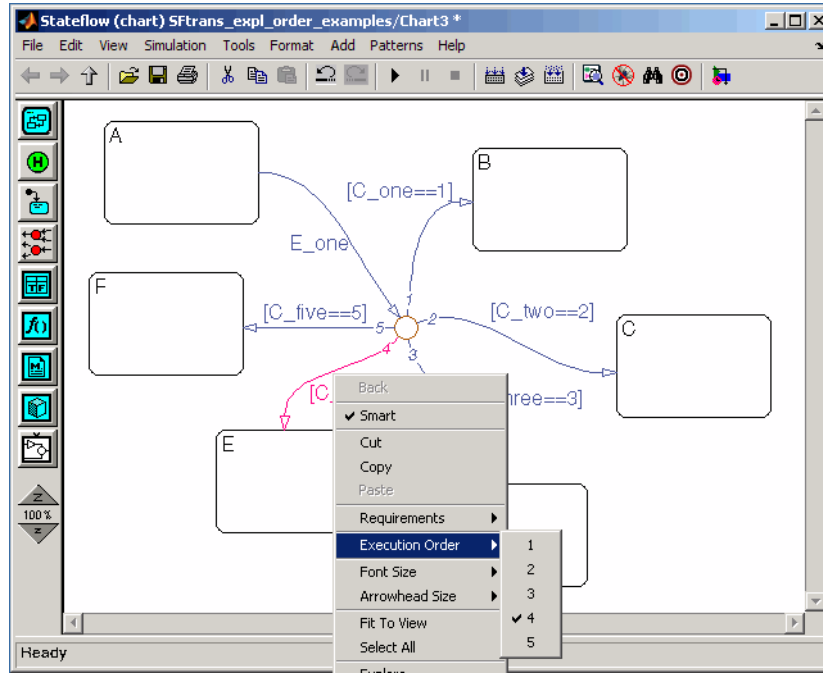
3 Click **OK**.

Setting Evaluation Order for Transitions Individually.

1 Right-click a transition and select **Execution Order**.

Note If you select **Execution Order** while in implicit ordering mode, the only option available is **Enable user-specified execution order for this chart**. This option opens the Chart properties dialog box where you can switch to explicit ordering mode, as described in "Using Explicit Ordering for Transitions" on page 3-22.

A context menu of available transition numbers appears, with a check mark next to the current number for this transition.



2 Select the new transition number.

The Stateflow chart automatically renumbers the other transitions for the source by preserving the relative transition order.

3 Repeat this procedure to renumber other transitions as needed.

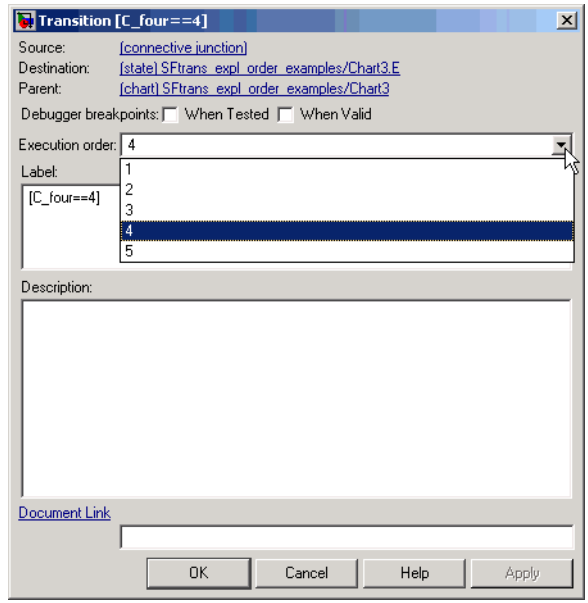
Another way to access the transition order number is through the properties dialog box.

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

The properties dialog box for the transition appears.

2 Click in the **Execution order** box.

A drop-down list of valid transition numbers appears.



3 Select the new transition number and click **Apply**.

Note If explicit ordering mode is enabled, the Stateflow chart assigns the new number to the current transition and automatically renumbers the other transitions. If the chart is in implicit ordering mode, an error dialog box appears and the old number is retained.

Implicit Ordering of Outgoing Transitions

How Implicit Ordering Works

In implicit ordering mode, a Stateflow chart evaluates a group of outgoing transitions from a single source based on these factors (in descending order of priority):

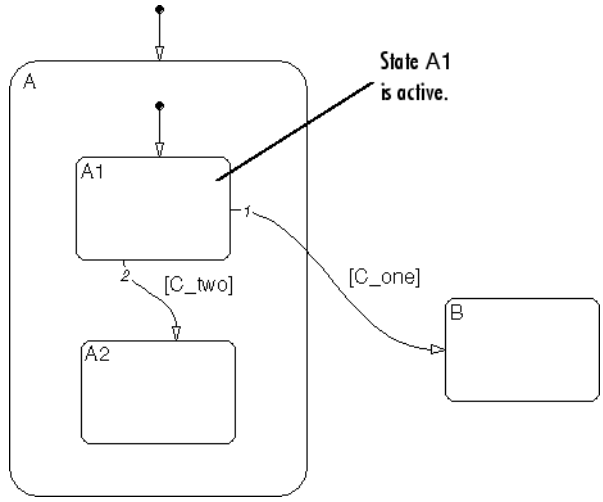
1. Hierarchy (see "Ordering by Hierarchy" on page 3-27)
2. Label (see "Ordering by Label" on page 3-28)
3. Angular surface position of transition source (see "Ordering by Angular Position of Source" on page 3-29)

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

Ordering by Hierarchy

A chart evaluates a group of outgoing transitions in an order based on the hierarchical level of the parent of each transition. The parent of a transition is the lowest level or innermost object in the Stateflow hierarchy that contains all parts of the transition, including any source state or junction and the endpoint object. For a group of outgoing transitions from a single source, the transition whose parent is at a higher hierarchical level than the parents of all other outgoing transitions is first in testing order, and so on.

Example of Ordering by Hierarchy.



- The parent of the transition from state A1 to state B is the chart.
- The parent of the transition from state A1 to state A2 is the state A.
- An event occurs while state A1 is active.

Since the chart is at a higher level in the Stateflow hierarchy than state A, the transition from state A1 to state B takes precedence over the transition from state A1 to state A2.

Ordering by 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

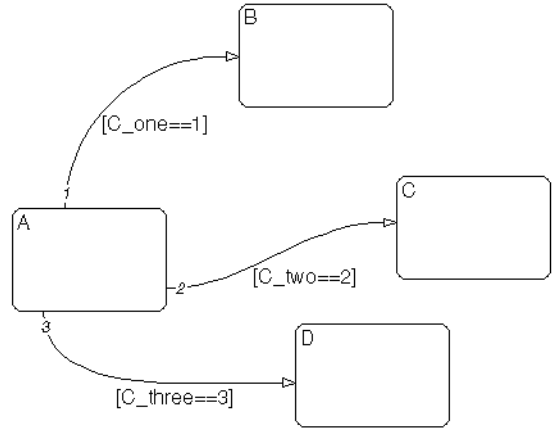
4 No label

Ordering by Angular Position of 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.

Note These evaluations proceed in a clockwise direction around the source object.

Example of Angular Ordering for a Source State.

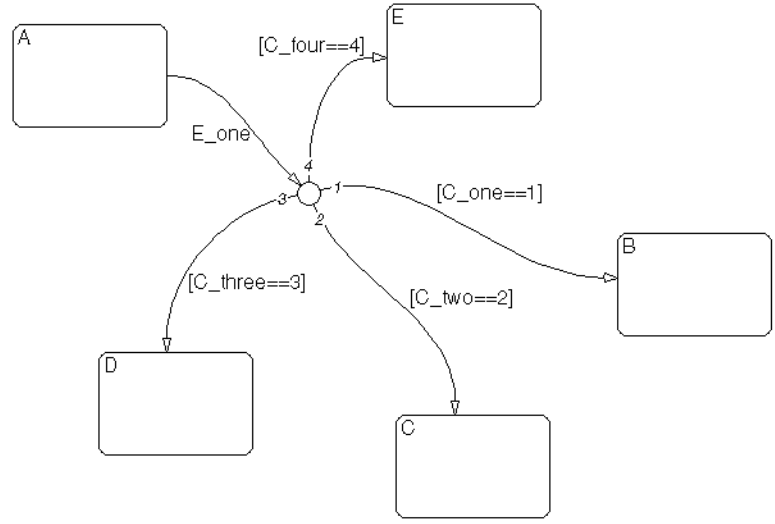


- For each outgoing transition from state `A`, the parent is the chart and the label contains a condition. Therefore, the outgoing transitions have equal hierarchical and label priority.
- The conditions [`C_one` == 1] and [`C_two` == 2] are false, and the condition [`C_three` == 3] is true.

The chart evaluates the outgoing transitions from state A in this order.

Phase	Chart evaluates transition to...	Condition is...	Transition occurs?
1	State B	False	No
2	State C	False	No
3	State D	True	Yes

Example of Angular Ordering for a Source Junction.



- For each outgoing transition from the junction, the parent is the chart and the label contains a condition. Therefore, the outgoing transitions have equal hierarchical and label priority.
- The conditions [C_one == 1] and [C_two == 2] are false, and the conditions [C_three == 3] and [C_four == 4] are true.
- The junction source point for the transition to state E is exactly 12 o'clock.

The chart evaluates the outgoing transitions from the junction in this order.

Phase	Chart evaluates transition to...	Condition is...	Transition occurs?
1	State `B`	False	No
2	State `C`	False	No
3	State `D`	True	Yes

Since the transition to state `D` occurs, the chart does not evaluate the transition to state `E`.

Using Implicit Ordering for Transitions

To use implicit ordering for transitions, follow these steps:

1. Open the properties dialog box for your chart by selecting **Chart Properties** from the **File** menu in the Stateflow Editor.
2. In the properties dialog box, clear the **User specified state/transition execution order** check box.
3. Click **OK**.

What Happens When You Switch Between Explicit and Implicit Ordering

If you switch to implicit ordering mode after explicitly ordering transitions, the transition order resets to follow the implicit rules. Similarly, if you switch back to explicit ordering mode, without changing the chart, you can restore the previous explicit transition order. All existing transitions in a chart retain their current order numbers until you explicitly change them.

Whenever you switch from one transition ordering mode to another, the diagnostic viewer displays warnings about the changes in transition evaluation order.

Note If you change back to explicit ordering mode after modifying the chart, you may not be able to restore the previous explicit transition order. Review the warnings in the diagnostic viewer and change the transition order, as necessary.

Entering, Executing, and Exiting a State

In this section...

Entering a State

A state is entered (becomes active) in one of the following ways:

- An incoming transition crosses state boundaries.
- An incoming transition ends at the state boundary.
- It is the parallel state child of an activated state.

A state performs its `entry` action (if specified) when it becomes active. The state is marked active before its `entry` action is executed and completed.

The execution steps for entering a state are as follows:

1 If the parent of the state is not active, perform steps 1 through 4 for the parent first.

2 If the state is a parallel state, check if a sibling parallel state previous in entry order is active. If so, start at step 1 for this parallel state.

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 3-40 and "Implicit Ordering of Parallel States" on page 3-42.

3 Mark the state active.

4 Perform any entry actions.

5 Enter children, if needed:

   a. Execute the default flow paths for the state unless it contains a history junction.

   b. If the state contains a history junction and there is an active child of this state at some point after the most recent chart initialization, perform the entry actions for that child.

   c. If this state has children that are parallel states (parallel decomposition), perform entry steps 1 - 5 for each state according to its entry order.

6. If the state is a parallel state, perform all entry steps for the sibling state next in entry order.

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.

8. The chart goes to sleep.

Executing an Active State

When states become active, they perform the following execution steps:

1. Execute the set of outer flow graphs (see "Executing a Set of Flow Graphs" on page 3-19).

   If this action causes a state transition, execution stops.

   Note This step is never required for parallel states.

2. Perform `during` actions and valid `on` *`event name`* actions.

   Note Stateflow charts process these actions based on their order of appearance in state labels.

3. Execute the set of inner flow graphs.

If this action does not cause a state transition, the active children are executed, starting at step 1. Parallel states are executed in the same order that they are entered.

Exiting an Active State

A state is exited (becomes inactive) in one of the following ways:

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

A state performs its `exit` actions before becoming inactive.

The execution steps for exiting a state are as follows:

1 Sibling parallel states exit starting with the last-entered and progress in reverse order to the first-entered. See step 2 of "Entering a State" on page 3-33.

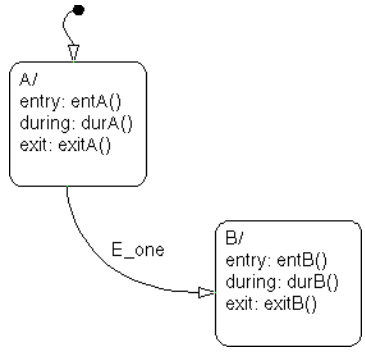
2 If a state has active children, performs the exit actions of the child states in the reverse order from when they were entered.

3 Perform any exit actions.

4 Mark the state as inactive.

State Execution Example

The following example shows how active and inactive states respond to events.



Inactive Chart Event Reaction

Inactive charts respond to events as follows:

1. An event occurs and the Stateflow chart wakes up.
2. The chart checks to see if there is a valid transition as a result of the event.

   A valid default transition to state `A` is detected.
3. State `A` is marked active.
4. State `A` entry actions (`entA()`) execute and complete.
5. The chart goes back to sleep.

Sleeping Chart Event Reaction

Sleeping charts respond to events as follows:

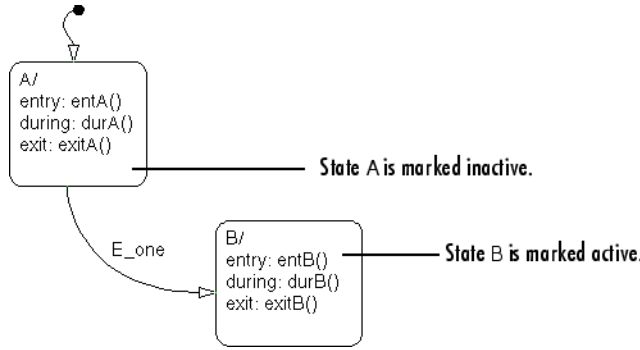
1. Event `E_one` occurs and the Stateflow chart wakes up.

   State `A` is active from the preceding steps 1 - 5.
2. The chart root checks to see if there is a valid transition as a result of `E_one`. A valid transition is detected from state `A` to state `B`.
3. State `A` exit actions (`exitA()`) execute and complete.
4. State `A` is marked inactive.

5 State `B` is marked active.

6 State `B` entry actions (`entB()`) execute and complete.

7 The chart goes back to sleep.



Execution Order for Parallel States

In this section...

What Does Ordering Mean 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, as described in "Entering, Executing, and Exiting a State" on page 3-33.

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 3-40).

- 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 3-42).

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 "Setting Execution Order for Parallel States Individually" on page 3-41. 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 "How a Chart Maintains Order of Parallel States" on page 3-43.

Using Explicit Ordering for Parallel States

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

1. "Enabling Explicit Ordering at the Chart Level" on page 3-40
2. "Setting Execution Order for Parallel States Individually" on page 3-41

Enabling Explicit Ordering at the Chart Level. To enable explicit ordering for parallel states, follow these steps:

1 Open the properties dialog box for your chart by selecting **Chart Properties** from the **File** menu in the Stateflow Editor.

Tip You can also use one of these methods:

- Right-click inside the top level of the chart and select **Properties** from the drop-down menu.
- Right-click inside one of the parallel states in the chart and select **Execution Order > Enable user-specified execution order for this chart** from the drop-down menu.

The properties dialog box appears.

2 In the properties dialog box, select the **User specified state/transition execution order** check box.

3 Click **OK**.

If your Stateflow 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 3-40). However you can now explicitly change execution order on a state-by-state basis, as described in "Setting Execution Order for Parallel States Individually" on page 3-41.

Setting 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.

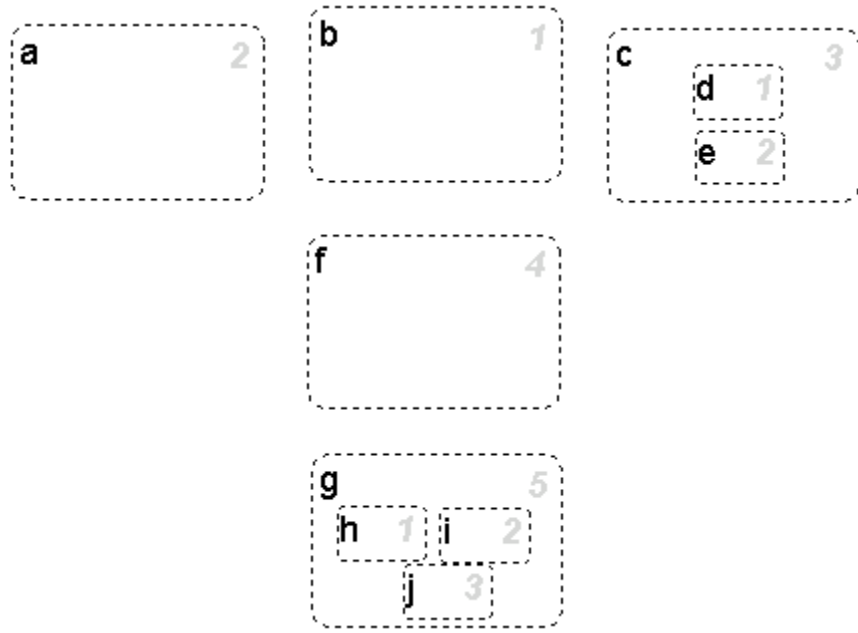
Implicit Ordering of Parallel States

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 illustrates 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.

Using Implicit Ordering for Parallel States

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

1. Open the properties dialog box for your chart by selecting **Chart Properties** from the **File** menu in the Stateflow Editor.

   Tip You can also right-click inside the top level of the chart and select **Properties** from the drop-down menu.

2. In the properties dialog box, clear the **User specified state/transition execution order** check box.

3. Click **OK**.

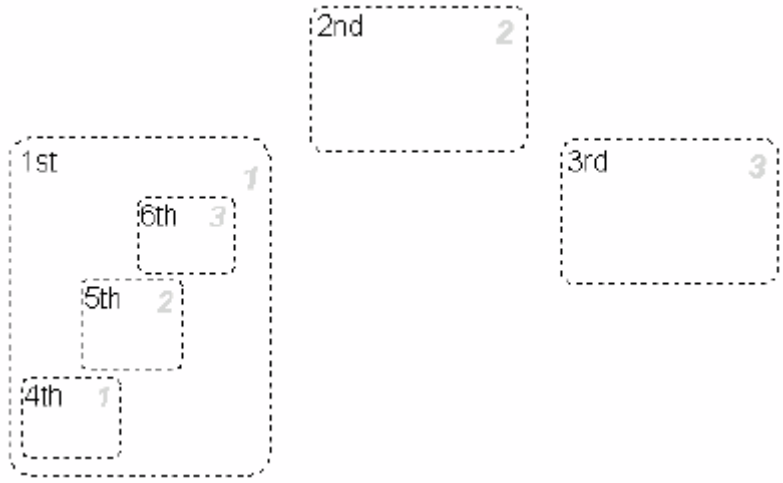
How a Chart Maintains Order of Parallel States

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:

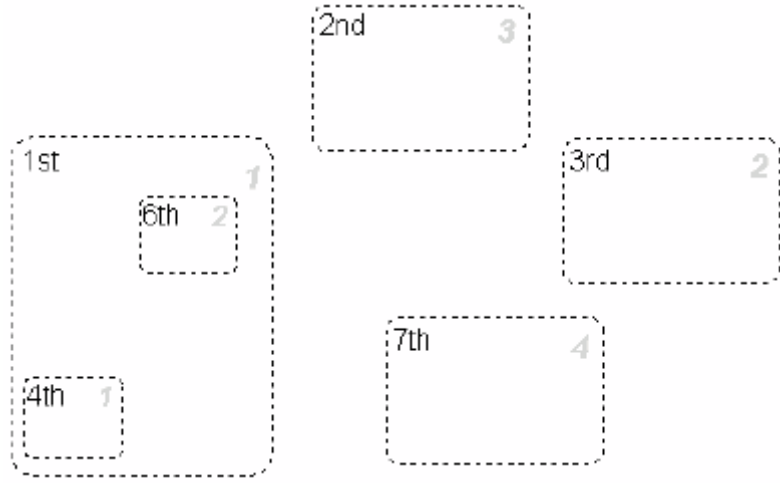


In this example, the ordinal names of the parallel states indicate the order in which they were created. The chart reprioritizes the parallel states and substates when you perform these actions:

1 Change the priority of top-level state `2nd` to `3`.

2 Add a top-level state `7th`.

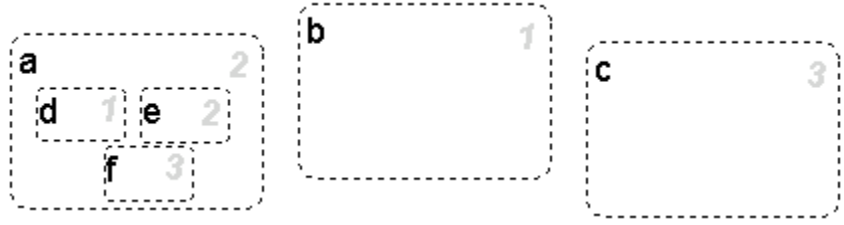
3 Remove substate `5th`.



The chart preserved the priority explicitly set for top-level state `2nd`, but renumbered 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:

How a Chart Assigns Execution Priorities to 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.

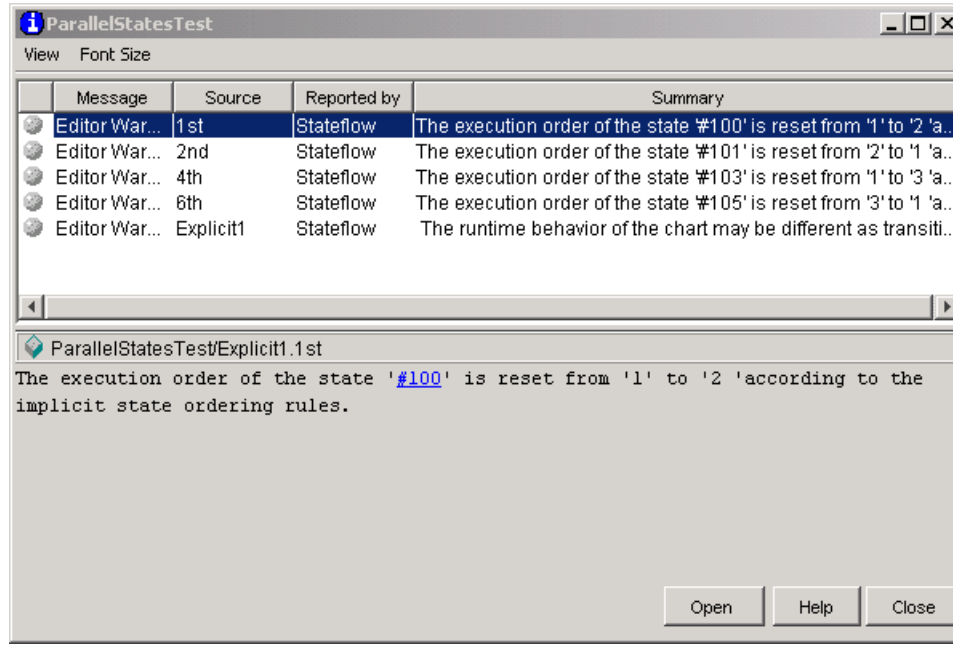
If you remove a state by...	And restore the state by...	What is the priority?
Deleting, cutting, dragging outside the boundaries of the parent state, or dragging so its boundaries overlap the parent state	Using the undo command	The original priority is restored.
Dragging outside the boundaries of the 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.

If you remove a state by...	And restore the state by...	What is the 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.
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.

What Happens When You Switch 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.

Whenever you switch from one ordering mode to another, the diagnostic viewer alerts you to changes in execution priorities. The following example shows the types of warnings issued after switching from explicit to implicit ordering for parallel states.



How a Chart Orders 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.

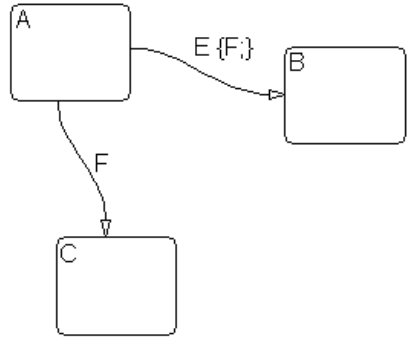
Early Return Logic for Event Broadcasts

A Stateflow chart uses early return logic to resolve conflicts with event broadcasts from state or transition actions.

The following are primary axioms of proper chart behavior:

1. Whenever a state is active, its parent should also be active.
2. A state (or chart) with exclusive (OR) decomposition must never have more than one active child.
3. If a parallel state is active, siblings with higher priority must also be active.

Stateflow charts run on a single thread. Therefore, charts must interrupt current activity to process events. In some cases, activity resulting from an event broadcast conflicts with the current activity. Charts resolve these conflicts by using early return logic, as in the following example:



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

1. The Stateflow 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 is found.
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 is found.

6 State A executes its exit action.

7 State A is marked inactive.

8 State C is marked active.

9 State C executes and completes its entry action.

State C is now the only active child 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 child of the chart. This behavior violates the second axiom that a state (or chart) with exclusive (OR) decomposition can never have more than one active child. Consequently, the chart uses early return logic and halts the transition from state A to state B.

To maintain primary axiomatic behavior, a Stateflow chart uses early return logic for event broadcasts in each of its action types 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.

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

Semantic Examples

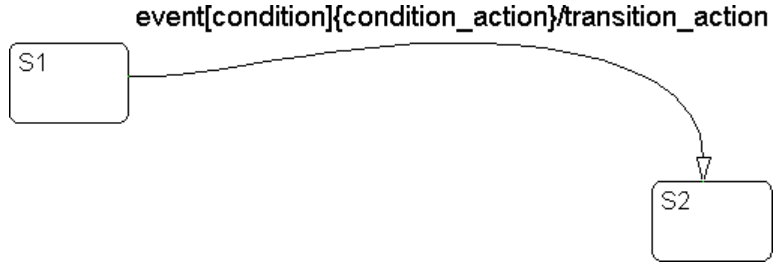
The following examples show the semantics (behavior) of Stateflow charts.

Transitions to and from Exclusive (OR) States Examples

In this section...

Label Format for a State-to-State Transition Example

The following example shows the general label format for a transition entering a state.



Stateflow charts execute this transition as follows:

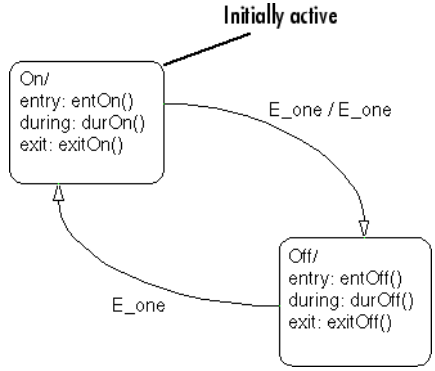
1. When an event occurs, state `S1` checks for an outgoing transition with a matching event specified.
2. If a transition with a matching event is found, the condition for that transition (`[condition]`) is evaluated.
3. If the condition `condition` is true, the condition action `condition_action` (`{condition_action}`) 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 `transition_action` is executed when the transition is taken.

7 State `S2` is entered.

Transitioning from State to State with Events Example

The following example shows the behavior of a simple transition focusing on the implications of whether states are active or inactive.



Processing of a First Event

Initially, the Stateflow chart is asleep. State `On` and state `Off` are OR states. State `On` is active. Event `E_one` occurs and awakens the chart. Event `E_one` is processed from the root of the chart down through the hierarchy of the chart:

1 The Stateflow 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 3-49.

5 State `Off` is marked active.

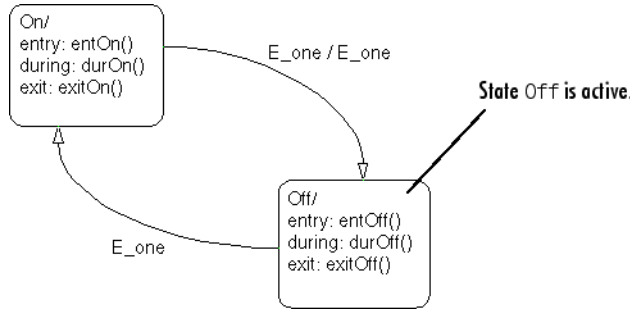
6 State `Off` entry actions (`entOff()`) execute and complete.

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.

Processing of a Second Event

Using the same example, what happens when the next event, `E_one`, occurs while state `Off` is active?



Again, initially the Stateflow chart is asleep. State `Off` is active. Event `E_one` occurs and awakens the chart. Event `E_one` is processed from the root of the chart down through the hierarchy of the chart with these steps:

1 The Stateflow 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.

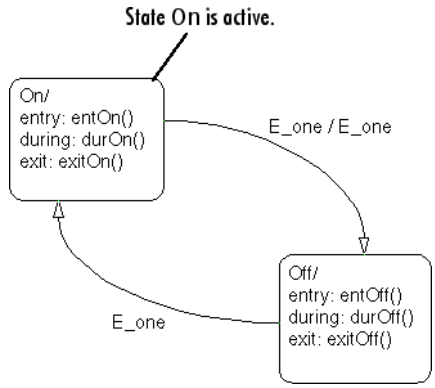
5 State `On` entry actions (`entOn()`) execute and complete.

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.

Processing of 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 Stateflow 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.

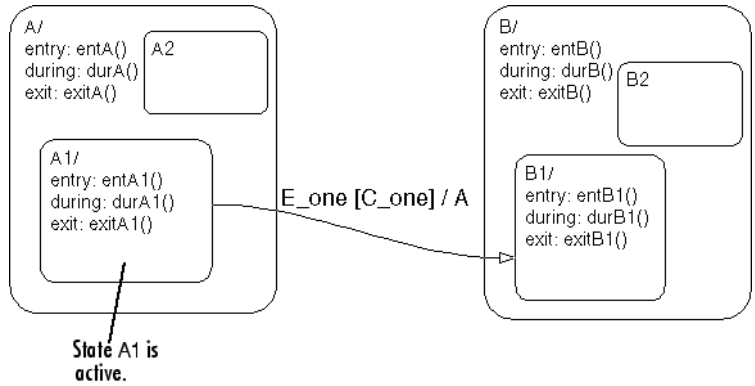
3 State `On` during actions (`durOn()`) execute and complete.

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.

Transitioning from a Substate to a Substate with Events Example

This example shows the behavior of a transition from an OR substate to an OR substate.



Initially, the Stateflow chart is asleep. State `A.A1` is active. Event `E_one` occurs and awakens the chart. Condition `C_one` is true. Event `E_one` is processed from the root of the chart down through the hierarchy of the chart:

1. The Stateflow 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.

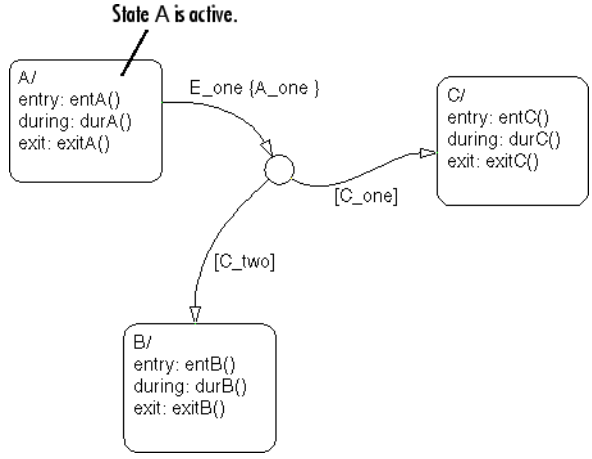
This sequence completes the execution of this Stateflow chart associated with event `E_one`.

Condition Action Examples

In this section...

Condition Action Example

This example shows the behavior of a simple condition action in a multiple segment transition.



Initially, the Stateflow chart is asleep. State `A` is active. Event `E_one` occurs and awakens the chart. Conditions `C_one` and `C_two` are false. Event `E_one` is processed from the root of the chart down through the hierarchy of the chart:

1. The Stateflow 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.

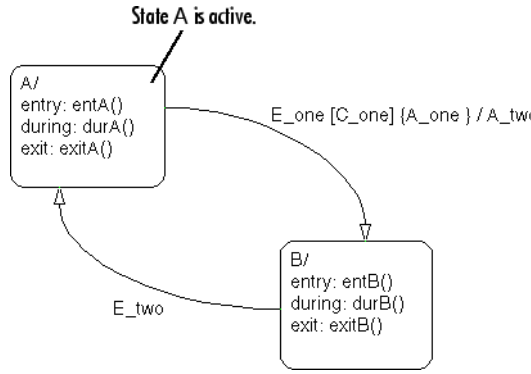
   State A remains active.

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 Actions Example

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 Stateflow chart is asleep. State A is active. Event E_one occurs and awakens the chart. Condition C_one is true. Event E_one is processed from the root of the chart down through the hierarchy of the chart:

1. The Stateflow 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

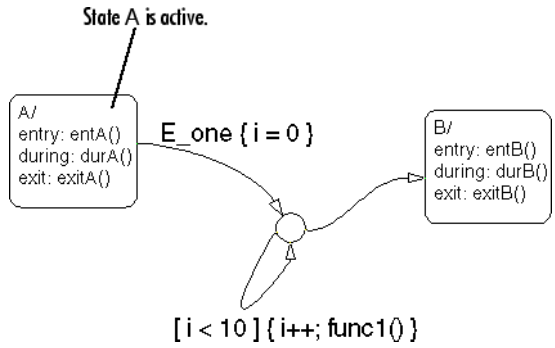
the valid transition and is immediately executed and completed. State `A` is still active.

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.

Condition Actions in For-Loop Construct Example

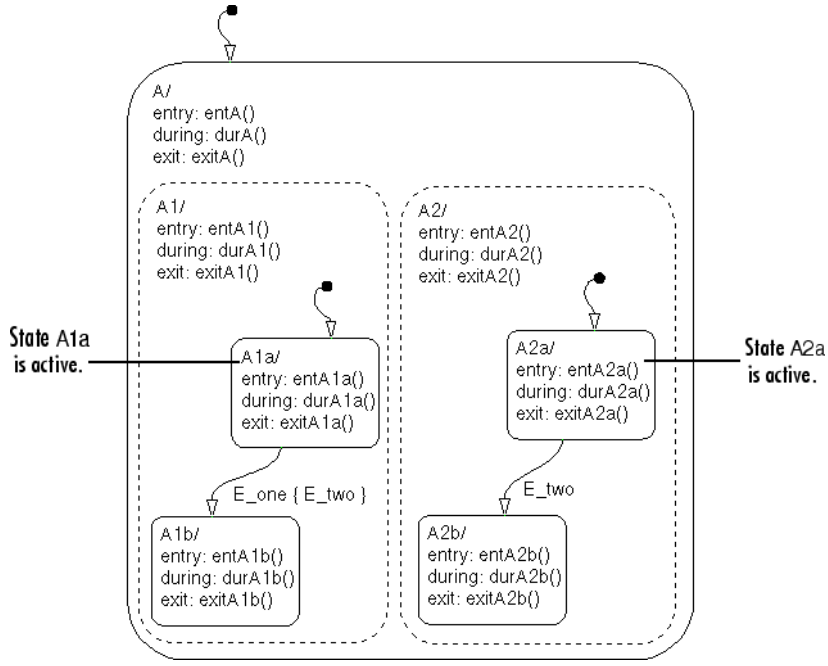
Condition actions and connective junctions are used to design a `for` loop construct. This example shows the use of a condition action and connective junction to create a `for` loop construct.



See “For-Loop Construct Example” on page 3-83 to see the behavior of this example.

Condition Actions to Broadcast Events to Parallel (AND) States Example

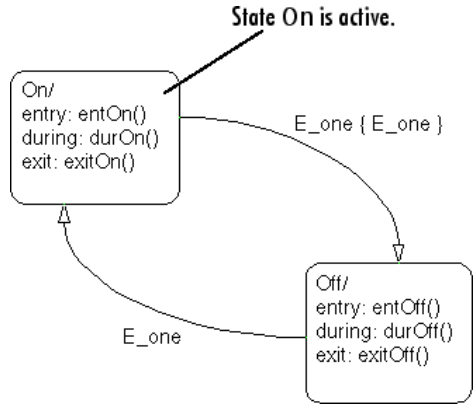
This example shows how to use condition actions to broadcast events immediately to parallel (AND) states.



See "Event Broadcast Condition Action Example" on page 3-100 to see the behavior of this example.

Cyclic Behavior to Avoid with Condition Actions Example

This example shows a notation to avoid when using event broadcasts as condition actions because the semantics results in cyclic behavior.



Initially, the Stateflow chart is asleep. State `On` is active. Event `E_one` occurs and awakens the chart. Event `E_one` is processed from the root of the chart down through the hierarchy of the chart:

1. The Stateflow 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.

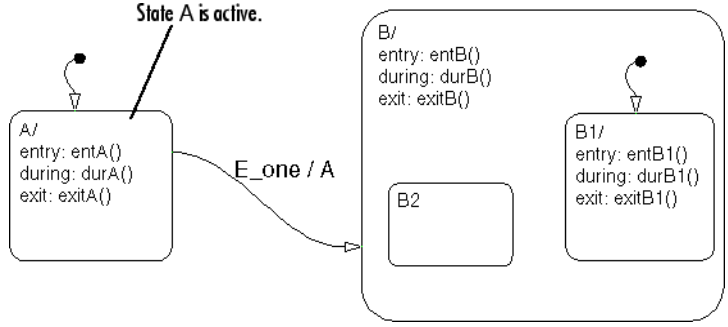
Steps 1 - 5 continue to execute in a cyclical manner. The transition label indicating a trigger on the same event as the condition action broadcast event results in unrecoverable cyclic behavior. This sequence never completes when event `E_one` is broadcast and state `On` is active.

Default Transition Examples

In this section...

Default Transition in Exclusive (OR) Decomposition Example

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 Stateflow chart is asleep. State `A` is active. Event `E_one` occurs and awakens the chart. Event `E_one` is processed from the root of the chart down through the hierarchy of the chart:

1. The Stateflow 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.

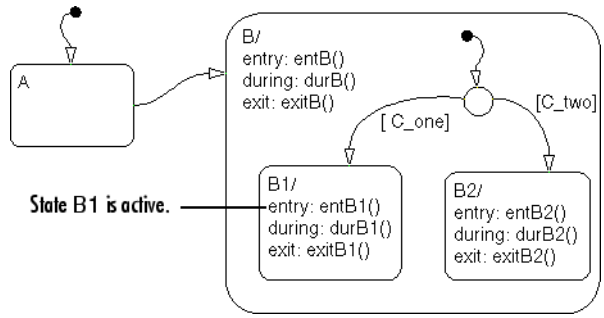
9 State B.B1 entry actions (entB1()) execute and complete.

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 Example

The following example shows the behavior of a default transition to a connective junction.



Initially, the Stateflow chart is asleep. State B.B1 is active. An event occurs and awakens the chart. Condition [C_two] is true. The event is processed from the root of the chart down through the hierarchy of the chart:

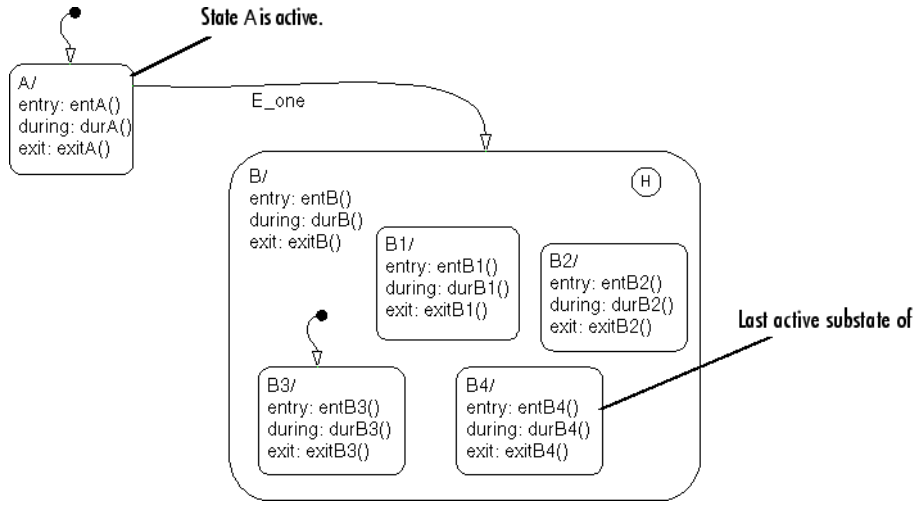
1 State B checks to see if there is a valid transition as a result of any event. There is none.

2 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 Example

This example shows the behavior of a superstate with a default transition and a history junction.



Initially, the Stateflow chart is asleep. State `A` is active. There is a history junction and state `B4` was the last active substate of superstate `B`. Event `E_one` occurs and awakens the chart. Event `E_one` is processed from the root of the chart down through the hierarchy of the chart:

1. The Stateflow 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.

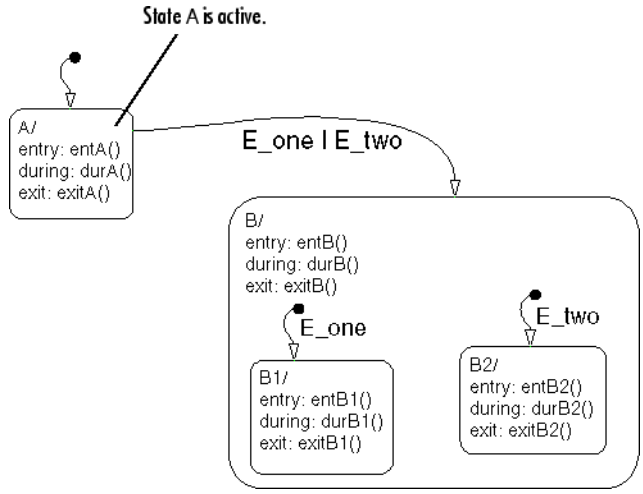
8 State B.B4 entry actions (entB4()) execute and complete.

9 The chart goes back to sleep.

This sequence completes the execution of this Stateflow chart associated with event E_one.

Labeled Default Transitions Example

This example shows the use of a default transition with a label.



Initially, the Stateflow chart is asleep. State A is active. Event E_one occurs, awakening the chart. Event E_one is processed from the root of the chart down through the hierarchy of the chart with the following steps:

1. The Stateflow 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.

Inner Transition Examples

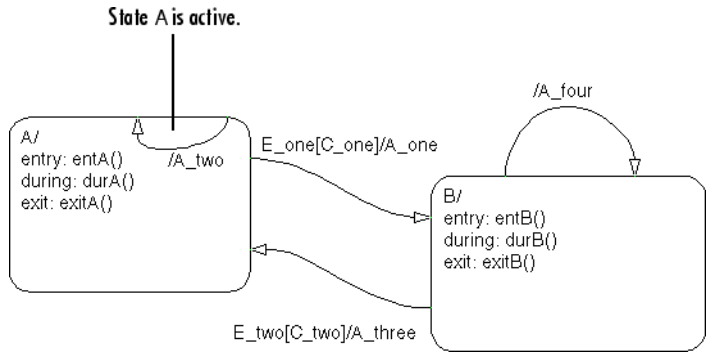
In this section...

Processing Events with an Inner Transition in an Exclusive (OR) State Example

This example shows what happens when processing three events using an inner transition in an exclusive (OR) state.

Processing One Event in an Exclusive (OR) State

This example shows the behavior of an inner transition.



Initially, the Stateflow chart is asleep. State `A` is active. Event `E_one` occurs and awakens the chart. Condition `[C_one]` is false. Event `E_one` is processed from the root of the chart down through the hierarchy of the chart:

1. The Stateflow 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.

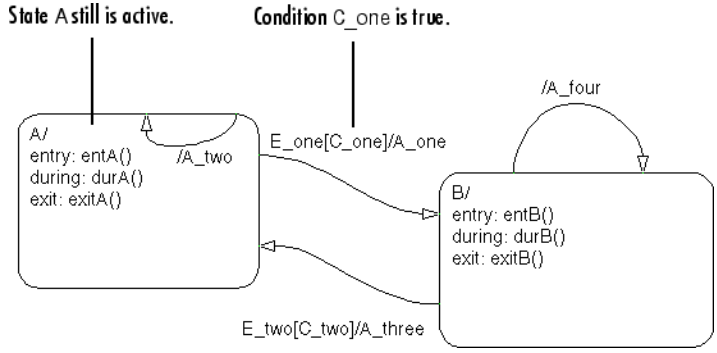
4 State A remains active. The inner transition action A_two is executed and completed. Because it is an inner transition, state A's exit and entry actions are not executed.

5 The chart goes back to sleep.

This sequence completes the execution of this Stateflow chart associated with event E_one.

Processing a Second Event in an Exclusive (OR) State

Using the previous example, this example shows what happens when a second event E_one occurs.



Initially, the Stateflow chart is asleep. State A is still active. Event E_one occurs and awakens the chart. Condition [C_one] is true. Event E_one is processed from the root of the chart down through the hierarchy of the chart with the following steps:

1 The Stateflow 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.

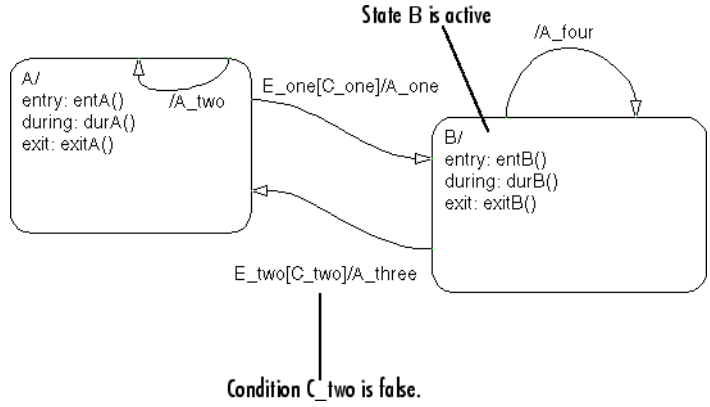
6 State `B` entry actions (`entB()`) execute and complete.

7 The chart goes back to sleep.

This sequence completes the execution of this Stateflow chart associated with event `E_one`.

Processing a Third Event in an Exclusive (OR) State

Using the previous example, this example shows what happens when a third event, `E_two`, occurs.



Initially, the Stateflow chart is asleep. State `B` is now active. Event `E_two` occurs and awakens the chart. Condition `[C_two]` is false. Event `E_two` is processed from the root of the chart down through the hierarchy of the chart:

1 The Stateflow 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.

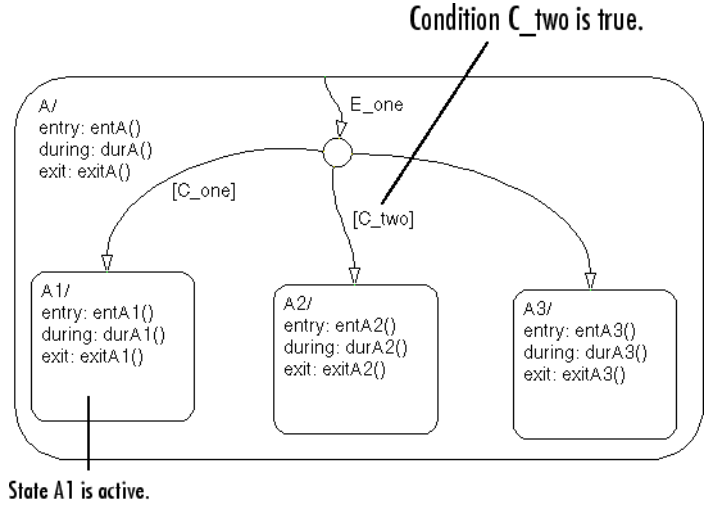
This sequence completes the execution of this Stateflow chart associated with event `E_two`. This example shows the difference in behavior between inner and self-loop transitions.

Processing Events with an Inner Transition to a Connective Junction Example

This example shows the behavior of handling repeated events using an inner transition to a connective junction.

Processing 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 of Outgoing Transitions" on page 3-27).



Initially, the Stateflow chart is asleep. State `A1` is active. Event `E_one` occurs and awakens the chart. Condition `[C_two]` is true. Event `E_one` is processed from the root of the chart down through the hierarchy of the chart:

1. The Stateflow 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.

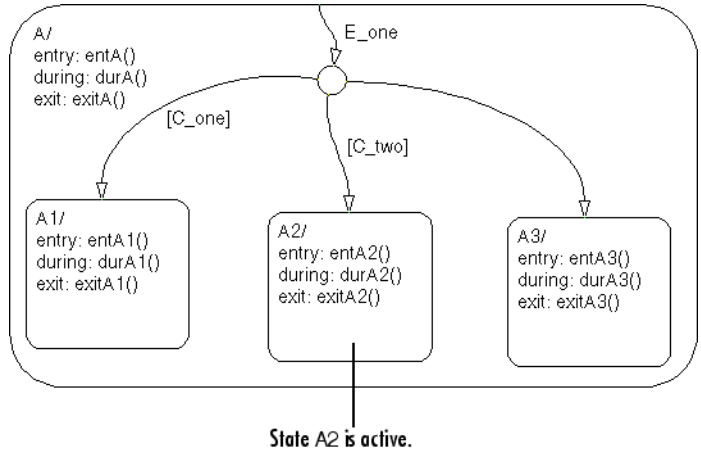
7 State A.A2 entry actions (entA2()) execute and complete.

8 The chart goes back to sleep.

This sequence completes the execution of this Stateflow chart associated with event E_one when condition C_two is true.

Processing 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 of Outgoing Transitions" on page 3-27).



Initially, the Stateflow chart is asleep. State A2 is active. Event E_one occurs and awakens the chart. Neither [C_one] nor [C_two] is true. Event E_one is processed from the root of the chart down through the hierarchy of the chart:

1 The Stateflow 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 evaluated before the unlabeled segment. The evaluation starts from a 12 o'clock position on the junction and progresses in a clockwise manner. Because neither `[C_one]` nor `[C_two]` is true, the unlabeled transition segment is evaluated and determined to be valid. The full transition from the inner transition to state `A.A3` is valid.

4 State `A.A2` exit actions (`exitA2()`) execute and complete.

5 State `A.A2` is marked inactive.

6 State `A.A3` is marked active.

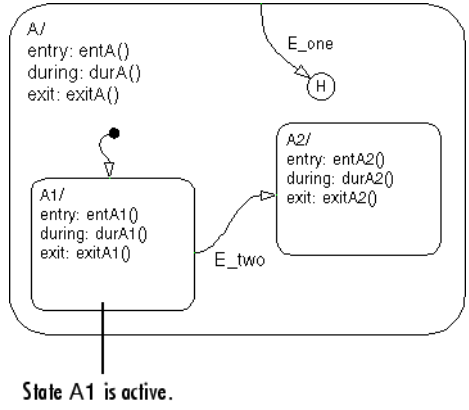
7 State `A.A3` entry actions (`entA3()`) execute and complete.

8 The chart goes back to sleep.

This sequence completes the execution of this Stateflow chart associated with event `E_one` when neither `[C_one]` nor `[C_two]` is true.

Inner Transition to a History Junction Example

This example shows the behavior of an inner transition to a history junction.



Initially, the Stateflow chart is asleep. State `A.A1` is active. There is history information because superstate `A` is active. Event `E_one` occurs and awakens the chart. Event `E_one` is processed from the root of the chart down through the hierarchy of the chart:

1. The Stateflow 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. According to the behavior of history junctions, 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.

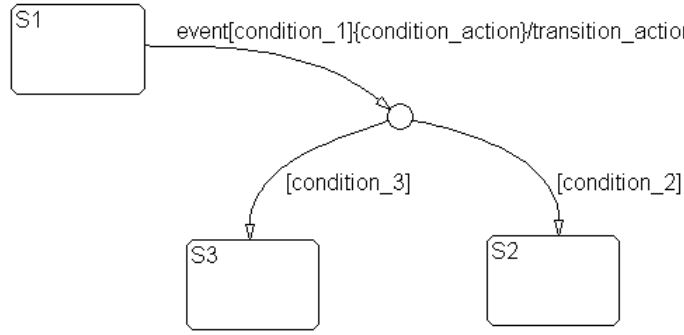
This sequence completes the execution of this Stateflow chart associated with event `E_one` when there is an inner transition to a history junction and state `A.A1` is active.

Connective Junction Examples

In this section...

Label Format for Transition Segments Example

The general label format for a transition segment entering a junction is the same as for transitions entering states, as shown in the following example.

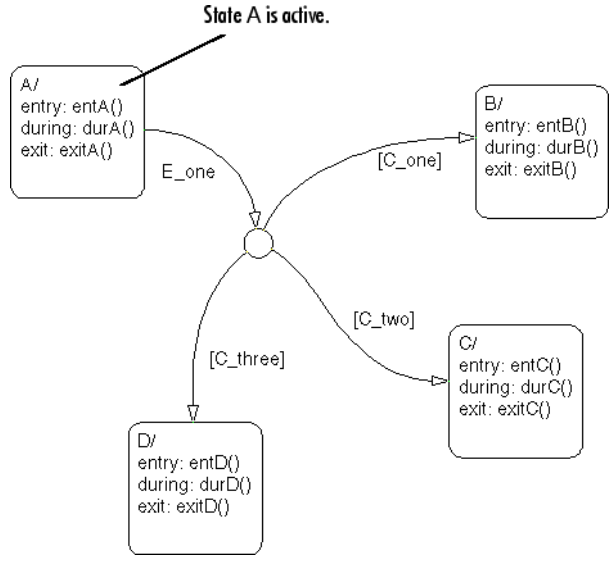


Execution of a transition in this example occurs as follows:

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 (`S1` to `S2`) is found.
5. State `S1` is exited (including execution of `S1`'s `exit` action).
6. The transition action `transition_action` is executed.
7. The completed state-to-state transition (`S1` to `S2`) is taken.
8. State `S2` is entered (including execution of `S2`'s `entry` action).

If-Then-Else Decision Construct Example

This example shows the behavior of an `if-then-else` decision construct. The chart uses implicit ordering of outgoing transitions (see "Implicit Ordering of Outgoing Transitions" on page 3-27).



Initially, the Stateflow chart is asleep. State `A` is active. Event `E_one` occurs and awakens the chart. Condition [`C_two`] is true. Event `E_one` is processed from the root of the chart down through the hierarchy of the chart:

1. The Stateflow 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. 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.

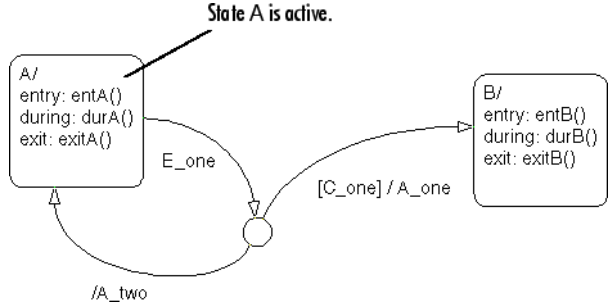
5. State `C` entry actions (`entC()`) execute and complete.

6 The chart goes back to sleep.

This sequence completes the execution of this Stateflow chart associated with event `E_one`.

Self-Loop Transition Example

This example shows the behavior of a self-loop transition using a connective junction.



Initially, the Stateflow chart is asleep. State `A` is active. Event `E_one` occurs and awakens the chart. Condition [`C_one`] is false. Event `E_one` is processed from the root of the chart down through the hierarchy of the chart:

1 The Stateflow 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 labeled with a condition and action 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.

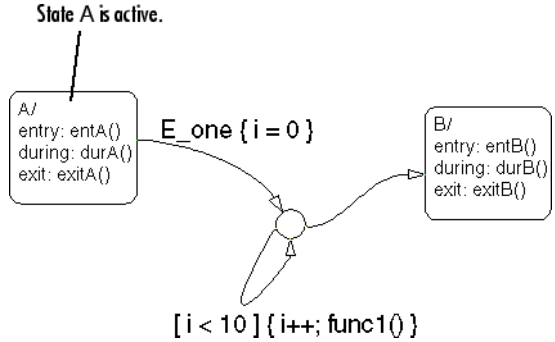
6 State `A` entry actions (`entA()`) execute and complete.

7 The chart goes back to sleep.

This sequence completes the execution of this Stateflow chart associated with event `E_one`.

For-Loop Construct Example

This example shows the behavior of a `for` loop using a connective junction. The chart uses implicit ordering of outgoing transitions (see "Implicit Ordering of Outgoing Transitions" on page 3-27).



Initially, the Stateflow chart is asleep. State `A` is active. Event `E_one` occurs and awakens the Stateflow chart. Event `E_one` is processed from the root of the Stateflow chart down through the hierarchy of the Stateflow chart:

1 The Stateflow 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`, is executed and completed. Of the two transition segments leaving the connective junction, the transition segment that is a self-loop back to the connective junction is evaluated next for validity. That segment takes priority in evaluation because it has a condition specified, whereas the other segment is unlabeled. This evaluation behavior reflects implicit ordering of outgoing transitions in the chart.

2 The condition [`i < 10`] is evaluated as true. The condition actions `i++` and a call to `func1` are executed and completed until the condition becomes

false. A connective junction is not a final destination; thus the transition destination remains to be determined.

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.

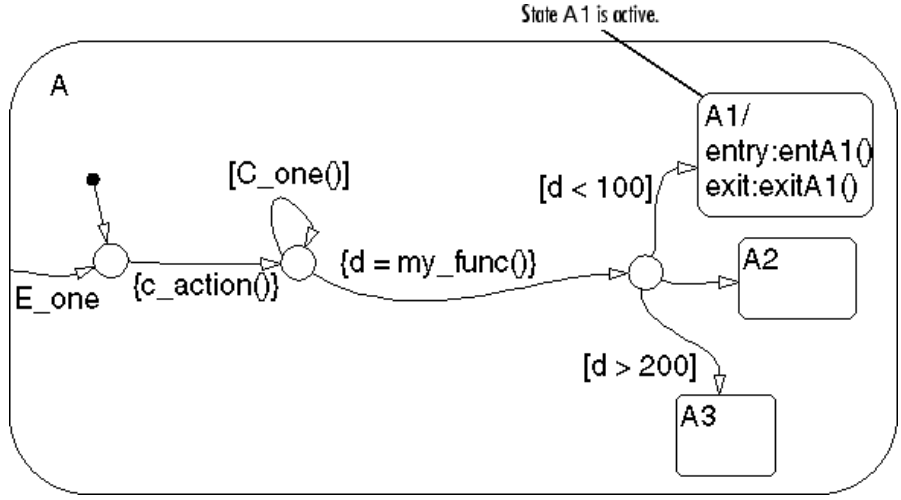
7 State `B` entry actions (`entB()`) execute and complete.

8 The chart goes back to sleep.

This sequence completes the execution of this chart associated with event `E_one`.

Flow Graph Notation Example

This example shows the behavior of a Stateflow chart that uses flow graph notation. The chart uses implicit ordering of outgoing transitions (see "Implicit Ordering of Outgoing Transitions" on page 3-27).



Initially, the Stateflow chart is asleep. State `A.A1` is active. The condition `[C_one()]` is initially true. Event `E_one` occurs and awakens the chart. Event `E_one` is processed from the root of the chart down through the hierarchy of the chart:

1. The Stateflow 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. There is only one outgoing transition and it has a condition action defined. The condition action is executed and completed.

4. The next possible segments are evaluated. There are two outgoing transitions; one is a conditional self-loop transition and the other is an unconditional transition segment. Because implicit ordering applies, the conditional transition segment takes precedence. Since the condition `[C_one()]` is tested to be 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()}`, is executed and completed. The returned value of `d` is 84.

6. The next possible transition segment is evaluated. There are three possible outgoing transition segments to consider. Two are conditional; one is unconditional. Because implicit ordering applies, the segment labeled with the condition `[d<100]` is evaluated first based on the geometry of the two outgoing conditional transition segments. Because the return 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.

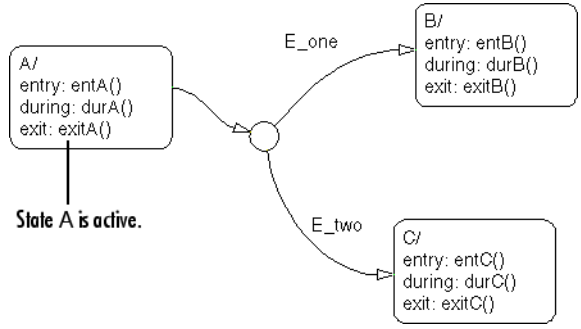
10 State A.A1 entry actions (entA1()) execute and complete.

11 The chart goes back to sleep.

This sequence completes the execution of this Stateflow chart associated with event E_one.

Transitions from a Common Source to Multiple Destinations Example

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 of Outgoing Transitions" on page 3-27).



Initially, the Stateflow chart is asleep. State A is active. 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:

1 The Stateflow chart root checks to see if there is a valid transition as a result of E_two. There is a valid transition segment 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.

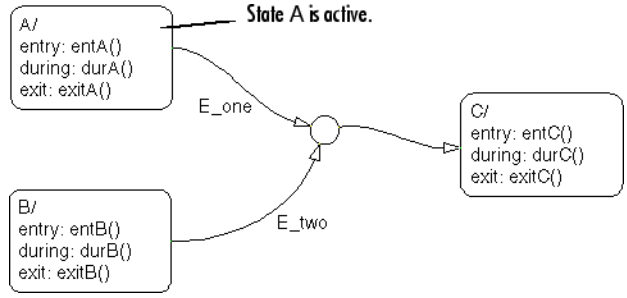
5 State C entry actions (entC()) execute and complete.

6 The chart goes back to sleep.

This sequence completes the execution of this Stateflow chart associated with event E_two.

Transitions from Multiple Sources to a Common Destination Example

This example shows the behavior of transitions from multiple sources to a single destination using a connective junction.



Initially, the Stateflow chart is asleep. State A is active. Event E_one occurs and awakens the chart. Event E_one is processed from the root of the chart down through the hierarchy of the chart:

1 The Stateflow 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 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.

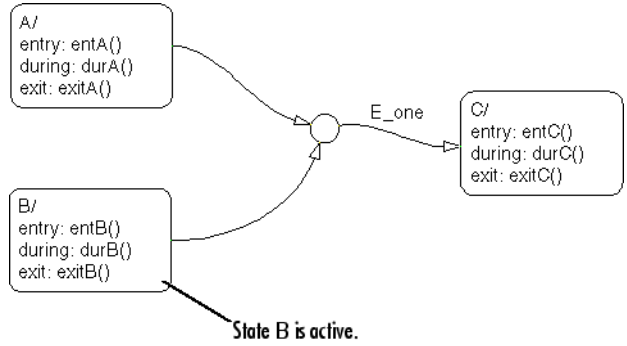
5 State C entry actions (entC()) execute and complete.

6 The chart goes back to sleep.

This sequence completes the execution of this Stateflow chart associated with event E_one.

Transitions from a Source to a Destination Based on a Common Event Example

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 Stateflow chart is asleep. State B is active. Event E_one occurs and awakens the chart. Event E_one is processed from the root of the chart down through the hierarchy of the chart:

1 The Stateflow 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 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.

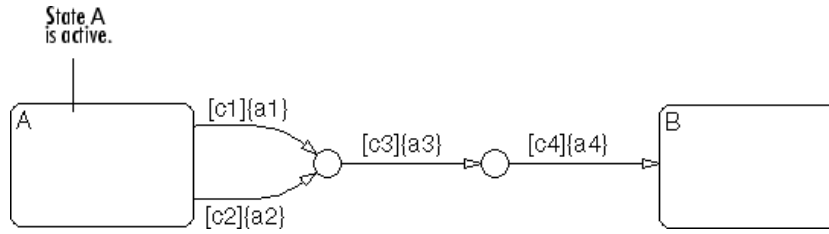
5 State C entry actions (entC()) execute and complete.

6 The chart goes back to sleep.

This sequence completes the execution of this Stateflow chart associated with event E_one.

Backtracking Behavior in Flow Graphs Example

This example shows the behavior of transitions with junctions that force backtracking behavior in flow graphs.



Initially, state A is active and conditions c1, c2, and c3 are true:

1 The Stateflow 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; therefore action a1 is executed.

3 Condition c3 is true; therefore action a3 is executed.

4 Condition c4 is not true; therefore control flow is backtracked 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.

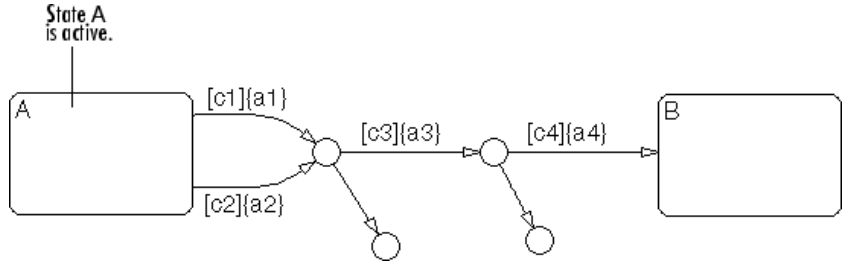
6 Condition c2 is true; therefore action a2 is executed.

7 Condition `c3` is true; therefore action `a3` is executed.

8 Condition `c4` is not true; therefore control flow is backtracked to state `A`.

9 The chart goes to sleep.

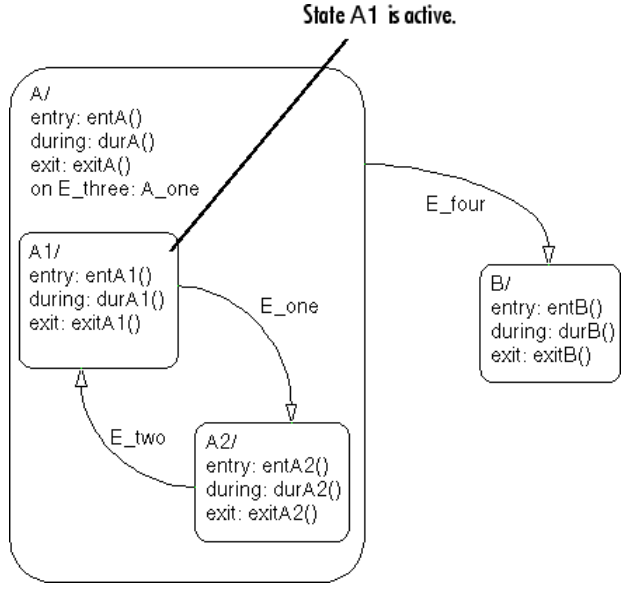
The preceding example shows the unanticipated behavior of executing both actions `a1` and `a2` and executing action `a3` twice. To resolve this problem, consider this design.



This example provides two junctions that allow flow to end if either `c3` or `c4` is not true. This leaves state A active without taking any unnecessary actions.

Event Actions in a Superstate Example

The following example shows the use of event actions in a superstate.



Initially, the Stateflow chart is asleep. State `A.A1` is active. Event `E_three` occurs and awakens the chart. Event `E_three` is processed from the root of the chart down through the hierarchy of the chart:

1. The Stateflow chart root checks to see if there is a valid transition as a result of `E_three`. There is no valid transition.
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. There are no valid transitions.
5. State `A1` during actions (`durA1()`) execute and complete.
6. The chart goes back to sleep.

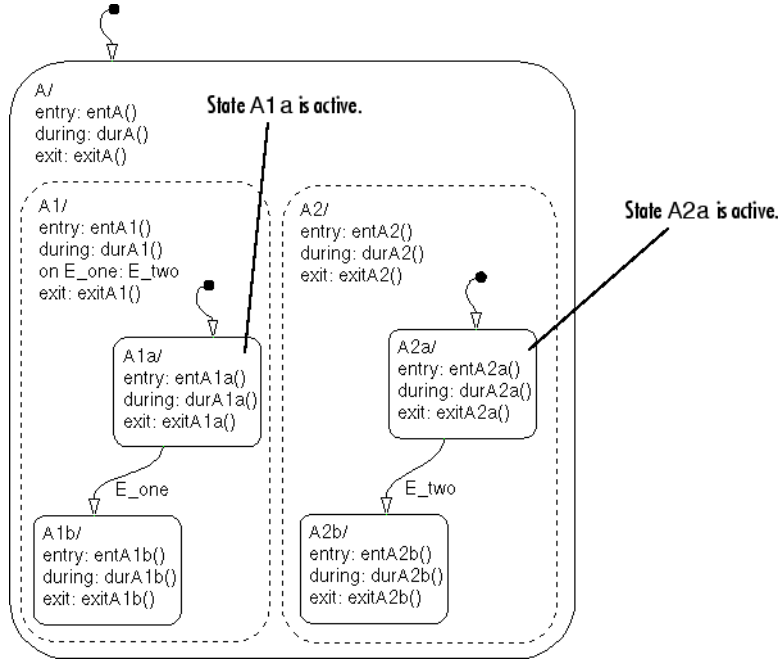
This sequence completes the execution of this Stateflow chart associated with event `E_three`.

Parallel (AND) State Examples

In this section...

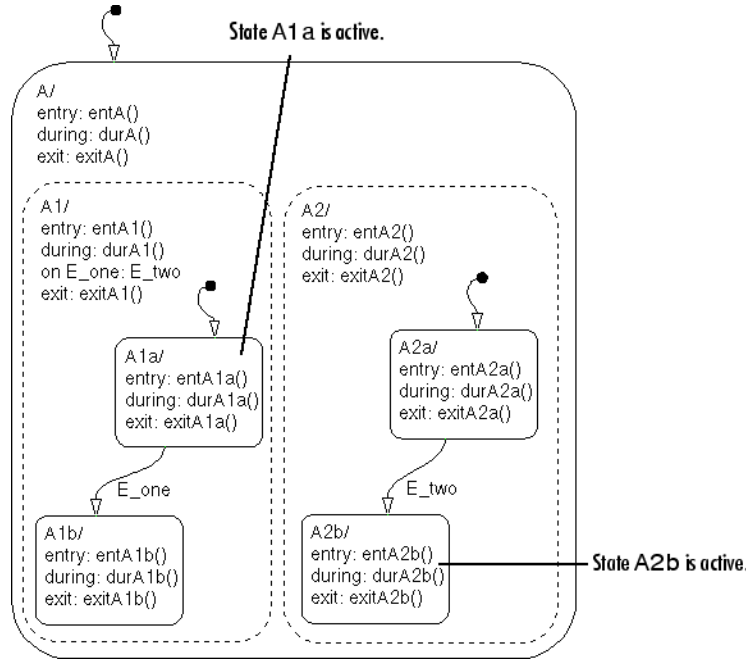
Event Broadcast State Action Example

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 3-42).



Initially, theStateflow chart is asleep. Parallel substates `A.A1.A1a` and `A.A2.A2a` are active. Event `E_one` occurs and awakens the chart. Event `E_one` is processed from the root of the chart down through the hierarchy of the chart:

1. The Stateflow 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. 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`. `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`. There is no valid transition.
   b. State A `during` actions (`durA()`) execute and complete.
   c. State A checks its children for valid transitions. There are no valid transitions.
   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. This diagram shows the Stateflow chart activity.



4 State `A.A1.A1a` executes and completes `exit` actions (`exitA1a`).

5 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`. There is a valid transition from state `A.A1.A1a` to state `A.A1.A1b`.

6 State `A.A1.A1a` is marked inactive.

7 State `A.A1.A1b` `entry` actions (`entA1b()`) execute and complete.

8 State `A.A1.A1b` is marked active.

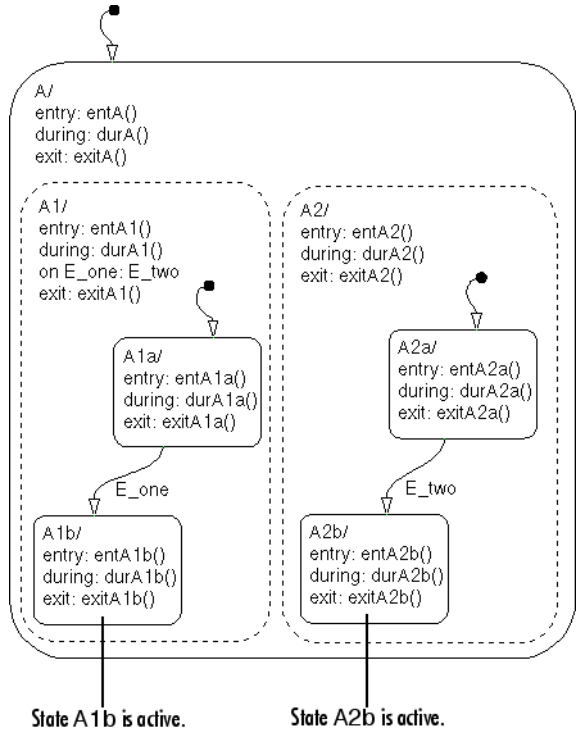
9 Parallel state `A.A2` is evaluated next. State `A.A2` `during` actions (`durA2()`) execute and complete. There are no valid transitions as a result of `E_one`.

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`.

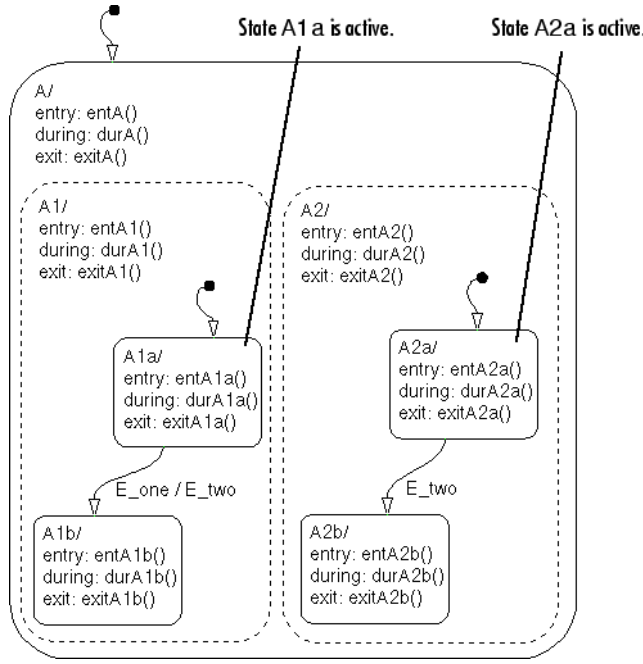
11 The chart goes back to sleep.

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`. This diagram shows the final chart activity.



Event Broadcast Transition Action with a Nested Event Broadcast Example

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 3-42).



Start of Event E_one Processing

Initially, the Stateflow chart is asleep. Parallel substates `A.A1.A1a` and `A.A2.A2a` are active. Event `E_one` occurs and awakens the chart. Event `E_one` is processed from the root of the chart down through the hierarchy of the chart:

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.

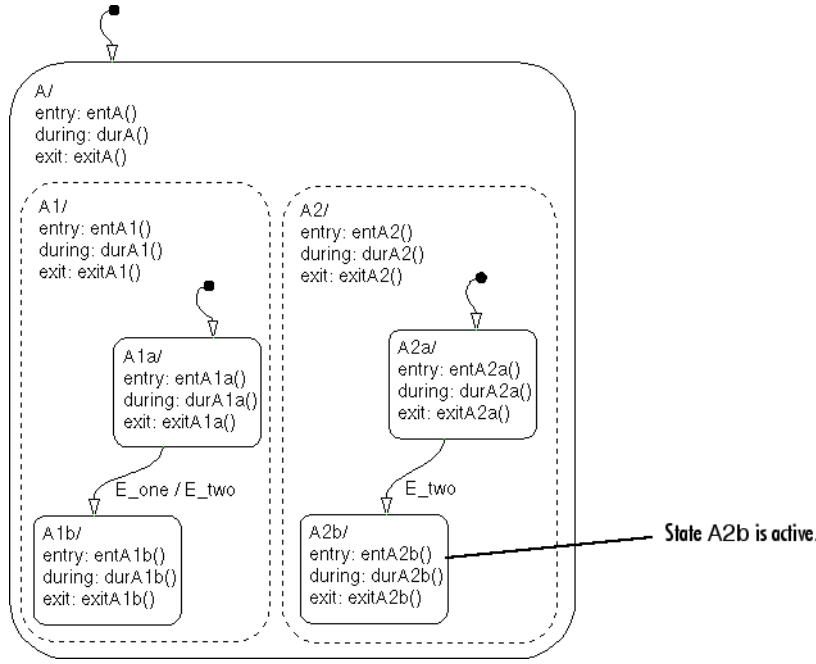
Event E_two Preempts E_one

7 Transition action generating event E_two is executed and completed:

- **a** The broadcast of event E_two now preempts the transition from state A1a to state A1b (triggered by event E_one) .
- **b** 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.
- **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_two Processing Ends

The following diagram shows Stateflow chart activity.



Event E_one Processing Resumes

8 State `A.A1.A1b` is marked active.

9 State `A.A1.A1b` entry actions (`entA1b()`) execute and complete.

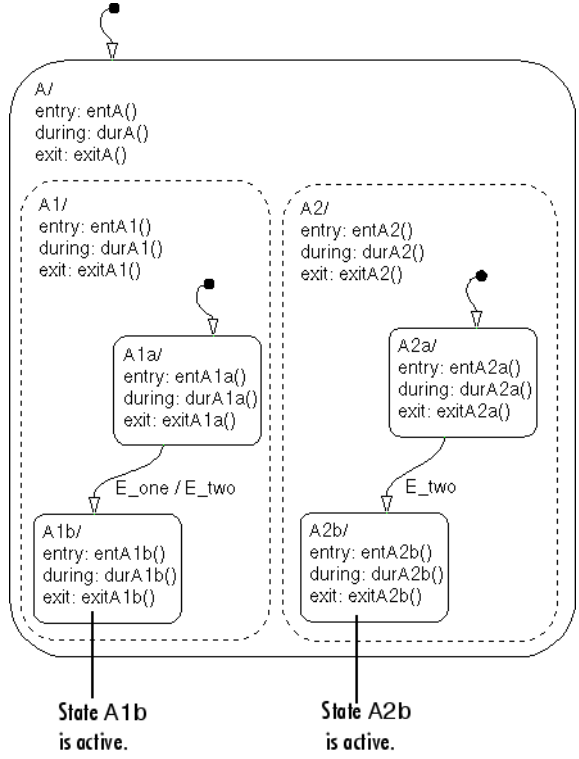
10 Parallel state `A.A2` is evaluated next. State `A.A2` during actions (`durA2()`) execute and complete. There are no valid transitions as a result of `E_one`.

11 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 transition action event broadcast of `E_two`.

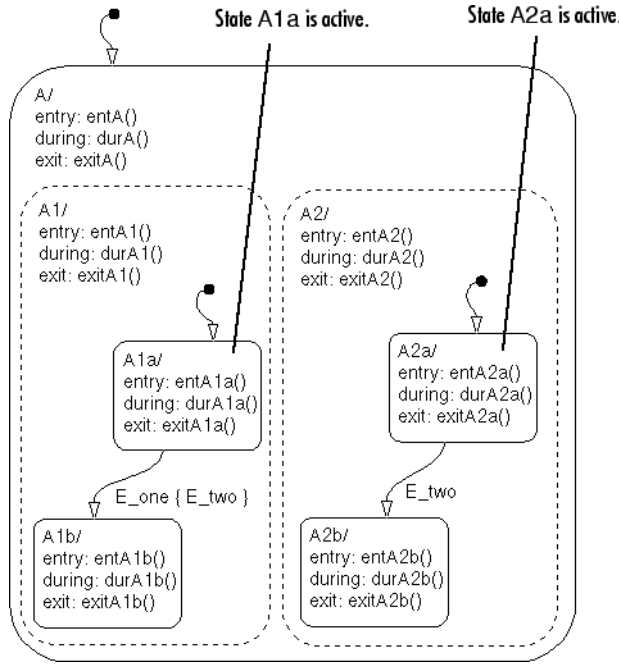
12 The chart goes back to sleep.

This sequence completes the execution of this Stateflow chart associated with event `E_one` and the transition action event broadcast to a parallel state of event `E_two`. This diagram shows the final chart activity.



Event Broadcast Condition Action Example

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 3-42).



Initially, the Stateflow chart is asleep. Parallel substates `A.A1.A1a` and `A.A2.A2a` are active. Event `E_one` occurs and awakens the chart. Event `E_one` is processed from the root of the chart down through the hierarchy of the chart:

1. The Stateflow 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 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`. There is a valid transition from state `A.A1.A1a` to state `A.A1.A1b`. There is also a valid condition action. The condition action event broadcast of `E_two` is executed and completed. 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.

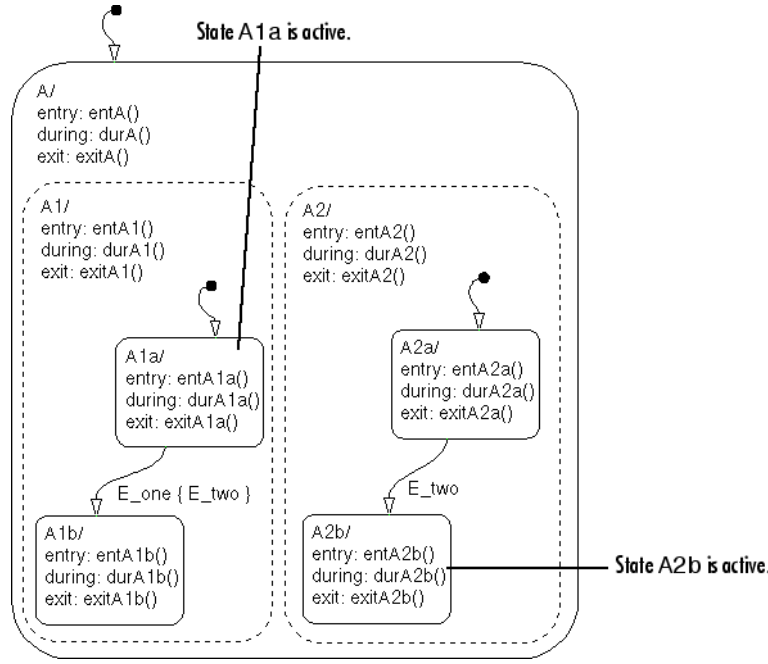
e State `A.A2` is evaluated. State `A.A2` `during` actions (`durA2()`) execute and complete. State `A.A2` checks for valid transitions. State `A.A2` has a valid transition as a result of `E_two` from state `A.A2.A2a` to state `A.A2.A2b`.

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` `entry` actions (`entA1b()`) execute and complete.

8 State `A.A1.A1b` is marked active.

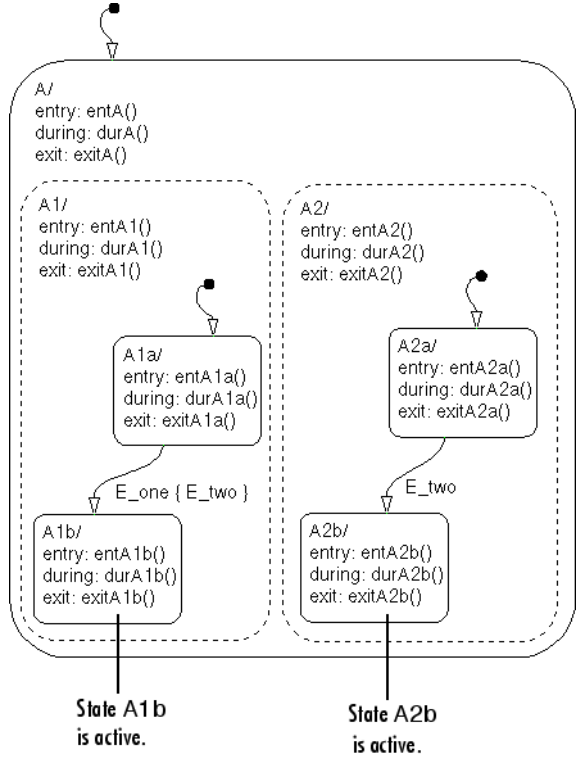
9 Parallel state `A.A2` is evaluated next. State `A.A2` `during` actions (`durA2()`) execute and complete. There are no valid transitions as a result of `E_one`.

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 condition action event broadcast to a parallel state of event `E_two`. This diagram shows the final chart activity.

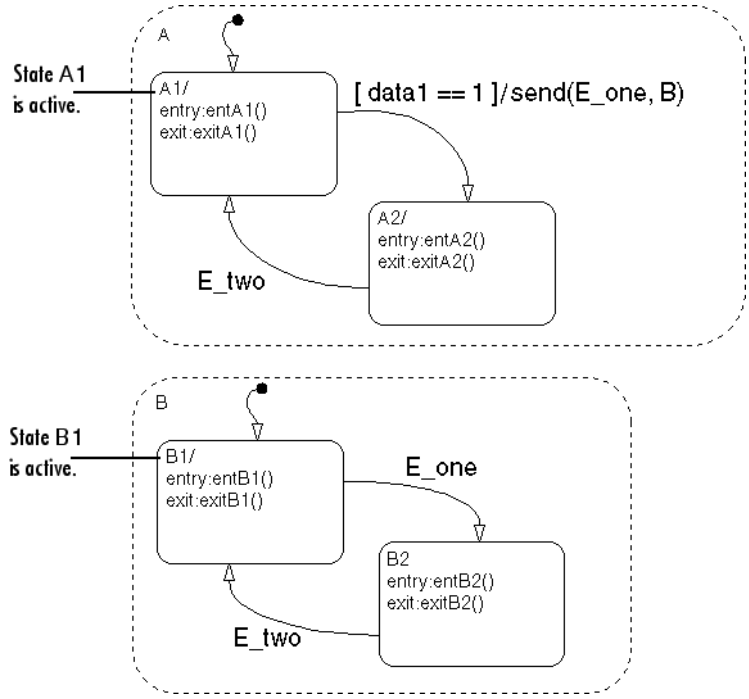


Directed Event Broadcasting Examples

In this section...

Directed Event Broadcast Using Send Example

This example shows the behavior of directed event broadcast using the `send(event_name,state_name)` function in a transition action. The chart uses implicit ordering of parallel states (see "Implicit Ordering of Parallel States" on page 3-42).



Initially, the Stateflow chart is asleep. Parallel substates `A.A1` and `B.B1` are active, which implies that parallel (AND) superstates `A` and `B` are also

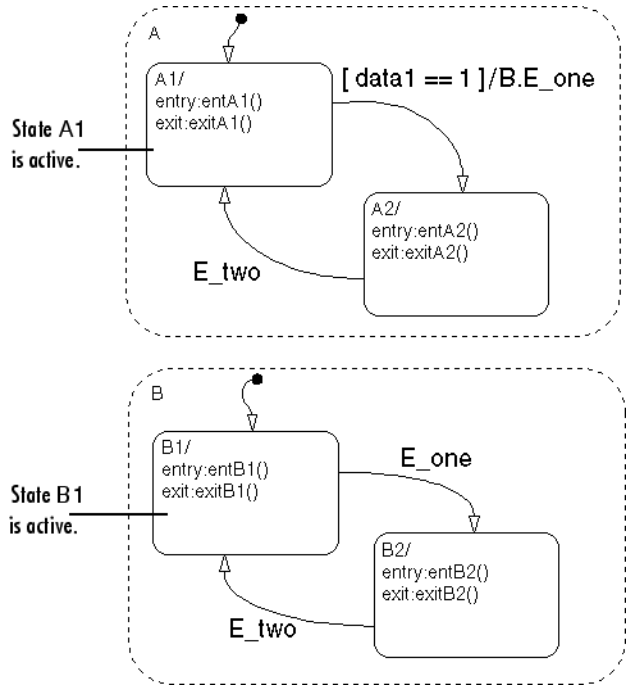
active. An event occurs and awakens the chart. The condition `[data1==1]` is true. The event is processed from the root of the chart down through the hierarchy of the chart:

1. The Stateflow 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. State `A.A1` exit actions (`exitA1()`) execute and complete.
4. State `A.A1` is marked inactive.
5. The transition action `send(E_one,B)` is executed and completed:
   a. The broadcast of event `E_one` is a nested event that awakens state `B`. Because state `B` is active, the directed broadcast is received and state `B` 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` is marked inactive.
   d. State `B.B2` is marked active.
   e. State `B.B2` entry actions (`entB2()`) execute and complete.
6. State `A.A2` is marked active.
7. State `A.A2` entry actions (`entA2()`) execute and complete.

This sequence completes the execution of this Stateflow chart associated with an event broadcast and the directed event broadcast to a parallel state of event `E_one`.

Directed Event Broadcasting Using Qualified Event Names Example

This example shows the behavior of directed event broadcast using a qualified event name in a transition action. The chart uses implicit ordering of parallel states (see "Implicit Ordering of Parallel States" on page 3-42).



Initially, the Stateflow chart is asleep. Parallel substates `A.A1` and `B.B1` are active, implying that parallel (AND) superstates `A` and `B` are also active. An event occurs and awakens the chart. The condition `[data1==1]` is true. The event is processed from the root of the chart down through the hierarchy of the chart:

1. The Stateflow 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. State `A.A1` exit actions (`exitA1()`) execute and complete.

4. State `A.A1` is marked inactive.

5. The transition action, a qualified broadcast of event `E_one` to state `B` (represented by the notation `B.E_one`), is executed and completed:

   a. The broadcast of event `E_one` is a nested event broadcast that awakens state `B`. Because state `B` is active, the directed broadcast is received and state `B` 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` is marked inactive.

   d. State `B.B2` is marked active.

   e. State `B.B2` `entry` actions (`entB2()`) execute and complete.

6. State `A.A2` is marked active.

7. State `A.A2` entry actions (`entA2()`) execute and complete.

This sequence completes the execution of this Stateflow chart associated with an event broadcast using a qualified event name to a parallel state.

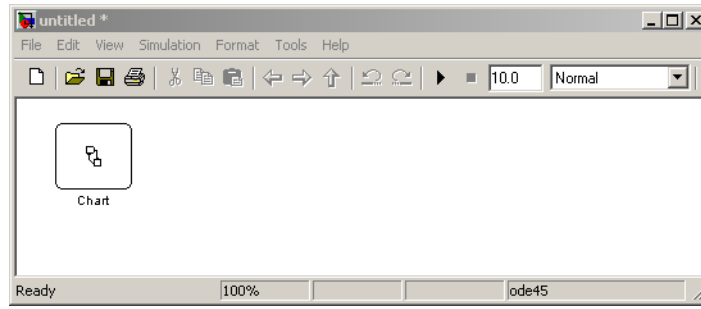
4

Creating Stateflow Charts

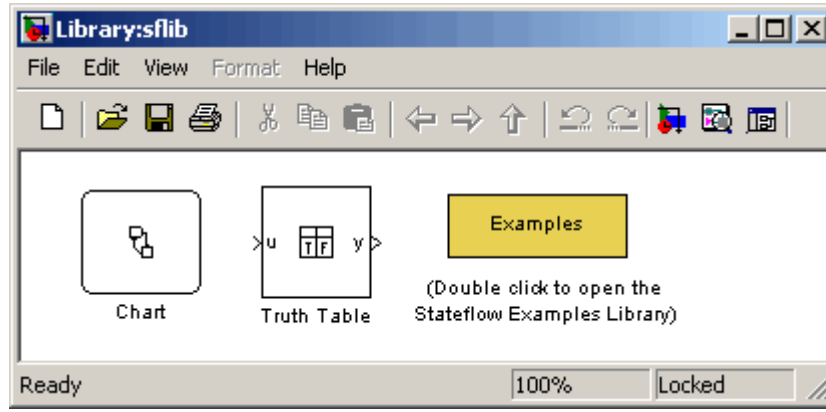
Creating a Stateflow Chart

You build a chart with Stateflow objects. You create charts by adding them to a Simulink model. Create a Stateflow chart in a Simulink model with the following steps:

1 Enter `sfnew` or `stateflow` at the MATLAB command prompt to create a new empty model with a Stateflow chart.



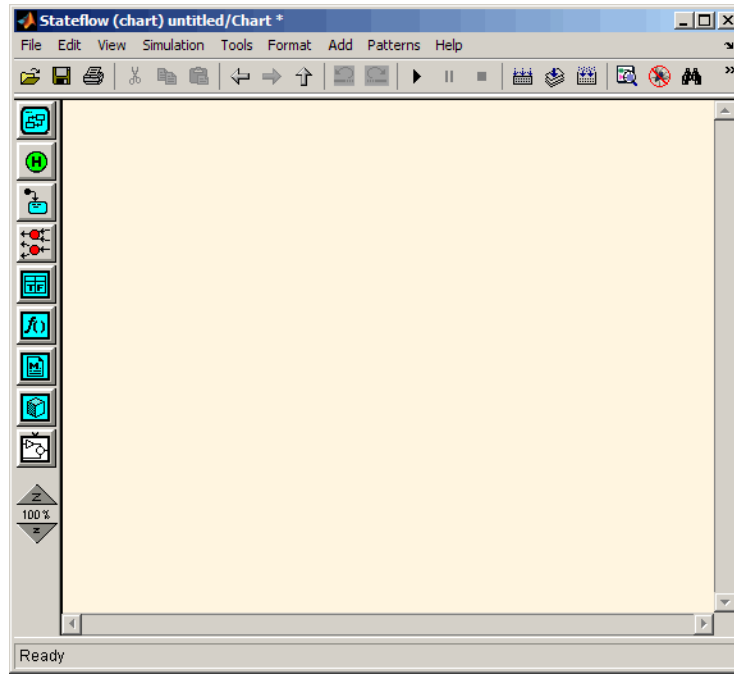
The `stateflow` command also displays the Stateflow block library.



You can drag and drop additional charts in your Simulink system from this library if you want to create multiple charts in your model. You can also drag and drop new charts into existing systems from the Stateflow library in the Simulink Library Browser. For information on creating your own chart libraries, see “Creating Chart Libraries” on page 16-27.

2 Open the chart by double-clicking the Chart block.

The empty chart appears in the Stateflow Editor.



3 Open the Chart properties dialog box.

See “Setting Properties for Individual Charts” on page 16-5.

4 In the Chart properties dialog box, select a chart type from the drop-down menu in the **State Machine Type** field:

Type	Description
Classic	The default machine type. Provides the full set of Stateflow chart semantics (see Chapter 3, “Stateflow Chart Semantics”).

Type	Description
Mealy	Machine type in which output is a function of inputs *and* state.
Moore	Machine type in which output is a function *only* of state.

Mealy and Moore charts use a subset of Stateflow chart semantics. For more information, see Chapter 6, "Building Mealy and Moore Charts".

5 In the Chart properties dialog box, specify an update method for the chart in the **Update method** field.

This value determines when and how often the chart is called during the execution of the Simulink model.

6 Use the Stateflow Editor to draw a Stateflow chart.

See "Using the Stateflow Editor" on page 4-27 and the rest of this chapter for more information on how to draw Stateflow charts.

7 Interface the chart to other blocks in your Simulink model, using events and data.

See Chapter 9, "Defining Events", Chapter 8, "Defining Data", and Chapter 16, "Defining Interfaces to Simulink Models and the MATLAB Workspace" for more information.

8 Rename and save the model by selecting **Save Model As** from the Stateflow Editor menu or **Save As** from the Simulink menu.

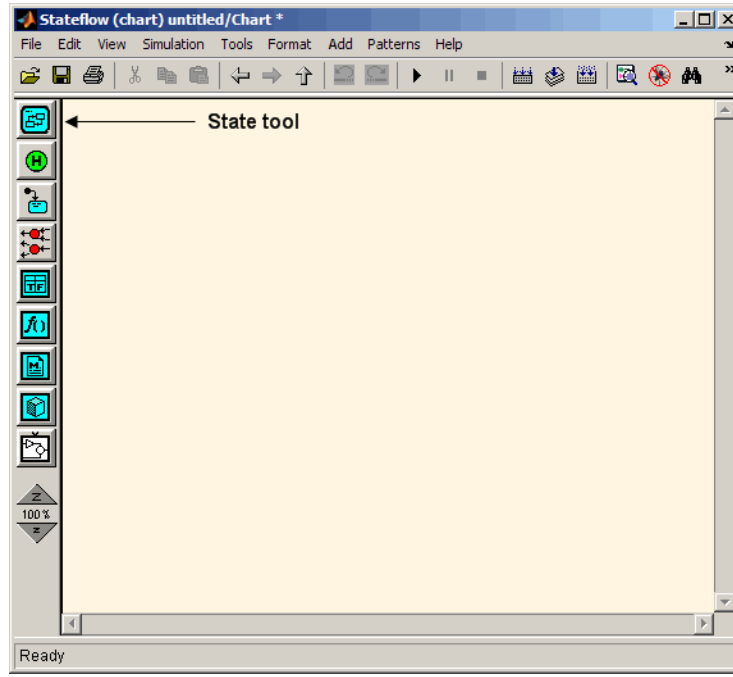
Note Trying to save a model with more than 25 characters produces an error. Loading a model with more than 25 characters produces a warning.

Working with States in Stateflow Charts

In this section...

Creating a State

You create states by drawing them in the Stateflow Editor for a particular Stateflow chart (block). The following is a depiction of the Stateflow Editor:



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 "Labeling States" on page 4-13 for more detail.

To delete a state, click it to select it and choose **Cut** from the **Edit** or any shortcut menu or press the **Delete** key.

Moving and Resizing 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.

Creating 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 a substate's parentage, 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 "Using Subcharts to Extend Charts" on page 7-5 for details.

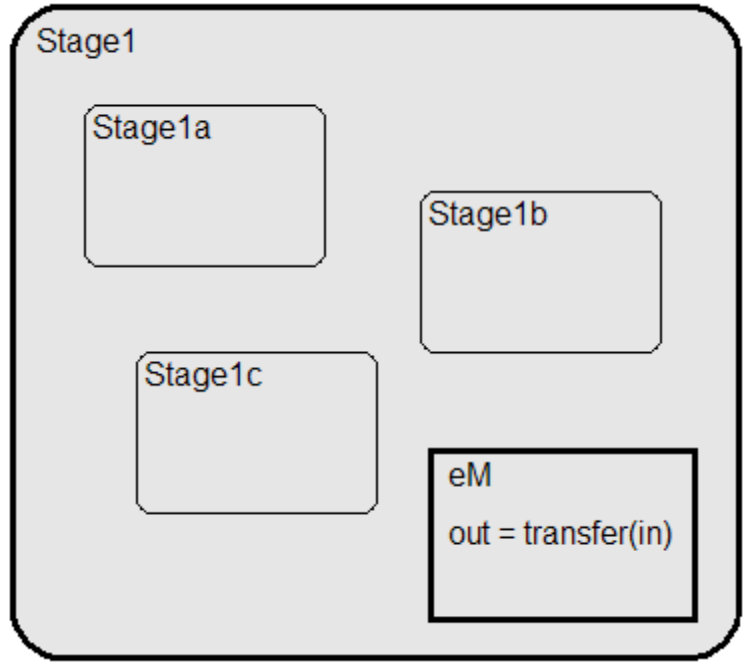
Grouping States

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 a chart. For example, moving a grouped state moves all substates and functions inside that state.

How to Group a State

To group a state, double-click the state or its border. The border of the state thickens and the background of the state appears gray to indicate that the state is grouped.



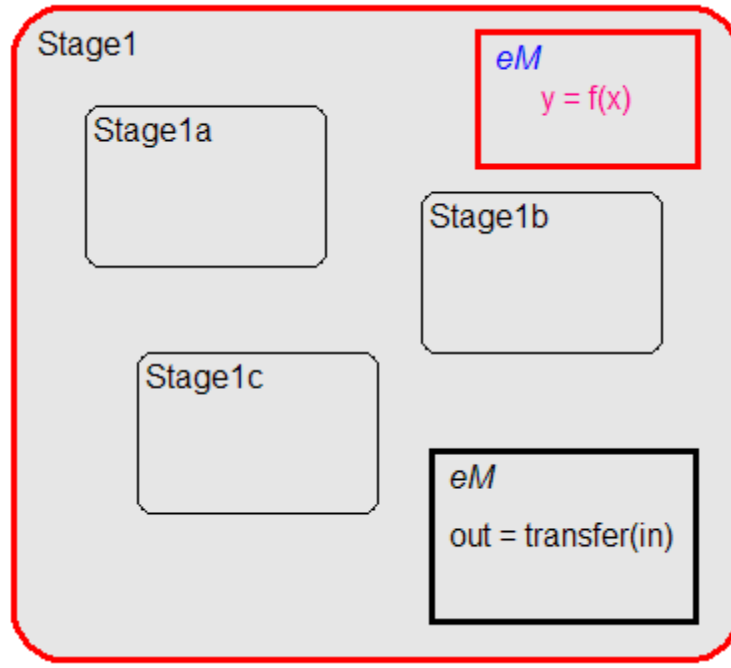
You can also group a state by right-clicking it and then selecting **Make Contents > Grouped** in the context menu.

When to Ungroup a State

You must ungroup a state before performing these actions:

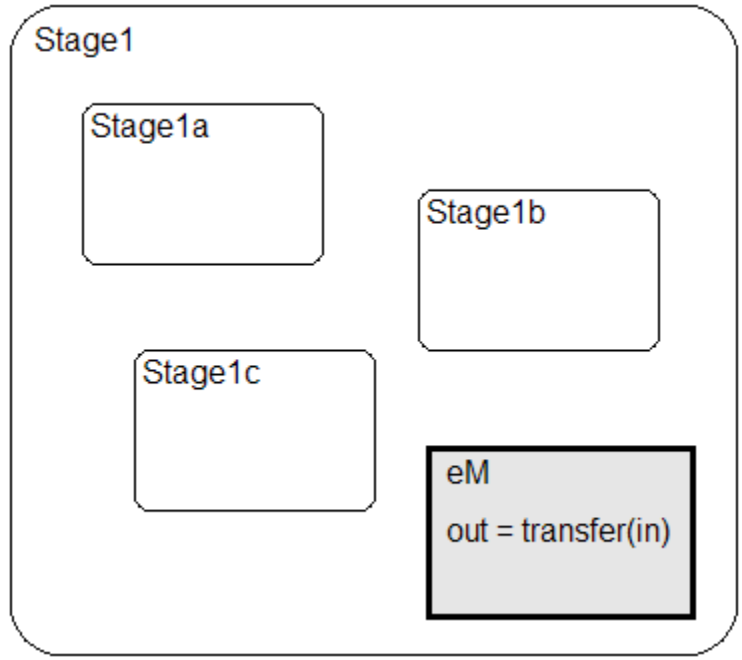
- Selecting objects inside the state
- Moving other graphical objects into the state

  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

To ungroup a state, double-click the state or its border. The border of the state thins and the background of the state no longer appears gray.



You can also ungroup a state by right-clicking it and then clearing **Make Contents > Grouped** in the context menu.

Specifying 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 shortcut menu, and choose either **Parallel (AND)** or **Exclusive (OR)** 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 shortcut menu, and choose either **Parallel (AND)** or **Exclusive (OR)** from the menu.

The appearance of a superstate's substates indicates the superstate's decomposition. 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.

Specifying 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 3-40 and "Implicit Ordering of Parallel States" on page 3-42.

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

Changing State Properties

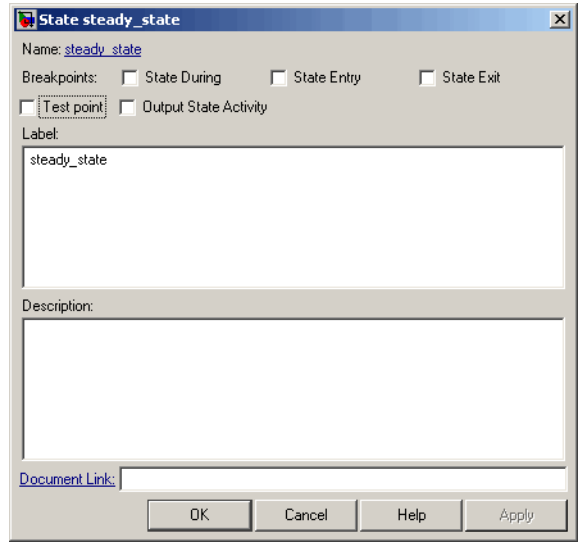
Use the State dialog box to view and change the properties for a state. To access the State dialog box for a particular state, do the following:

1. Right-click the state.

   A context menu appears.

2. Choose **Properties** from the context menu.

   The State dialog box for the state appears as shown.



The State dialog box contains the following properties for a state:

Field	Description
Name	Stateflow chart name; read-only; click this hypertext link to bring the state to the foreground.
Debugger breakpoints	Click the check boxes to set debugging breakpoints on the execution of state `entry`, `during`, or `exit` actions during simulation. See Chapter 23, "Debugging and Testing Stateflow Charts" for more information.
Test point	Select this check box to set 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 "Monitoring Test Points in Stateflow Charts" on page 23-37 in the online Stateflow software documentation.

Field	Description
Output State Activity	Select this check box to cause the Stateflow chart to output the activity status of this state to a Simulink model via a data output port on the Chart block containing the state. See "Outputting State Activity to a Simulink Model" on page 4-16 for more information.
Label	The label for the state. This includes the name of the state and its associated actions. See the section titled "Labeling States" on page 4-13 for detailed information.
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`.

3 After making changes, select one of these options:

- **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.

Labeling 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_name:on event_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.
`on`	*event_name* and *on event_name actions*	A specified event and Actions executed when a state is active and the specified event *event_name* occurs See "How Events Work in Stateflow Charts" on page 9-2 for information on defining and using events.

Entering 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.

   If you are finished labeling the state, click outside of the state. Otherwise, continue entering actions. To reedit the label, simply click the label text near the character position you want to edit.

Entering Actions

After entering the name of the state in the state's 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 `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.

- **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 <event_name> Actions** — begin with the keyword `on`, followed by a space and the name of an event, 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` *`event_name`* blocks in the state's label, each specifying the action for a particular event or set of events, for example:

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

Note The execution of the actions you enter for a state is dependent only on their action type, and not on the order in which you enter them in the label.

You can also edit the state's label through the properties dialog box for the state. See "Changing State Properties" on page 4-11.

Outputting State Activity to a Simulink Model

You can output the activity of a chart's states to a Simulink model via a data port on the state's Chart block. To enable output of a particular state's activity, first name the state (see "Entering the Name" on page 4-14), if unnamed, and then select the **Output State Activity** check box on the state's property dialog box (see "Changing State Properties" on page 4-11). A data output port appears on the Chart block containing the state. The port has the same name as the state. During simulation of a model, the port outputs 1 at each time step in which the state is active; 0, otherwise. Attaching a scope to the port allows you to monitor a state's activity visually during the simulation. See "Sharing Input and Output Data with Simulink Models" on page 8-27 for more information.

Note If a chart has multiple states with the same name, only one of those states can output activity data. If you check the `Output State Activity` property for more than one state with the same name, the chart outputs data only from the first state whose `Output State Activity` property you specified.

Working with Transitions in Stateflow Charts

In this section...

Creating a Transition

Use the following procedure for creating 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.

   Notice that the source of the transition sticks to the initial source point for the transition.

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.
- Newly created transitions have smart behavior. See "Setting Smart Behavior in Transitions" on page 7-17.

To delete a transition, select it and choose **Cut** from the **Edit** menu, or press the **Delete** key.

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

- "Creating Self-Loop Transitions" on page 4-23
- "Creating Default Transitions" on page 4-24

Creating Straight Transitions

While creating a transition, notice that the source of the transition sticks to the initial source point. This often results in a curved transition. To create a perfectly straight transition, while clicking and dragging from one state to another, do one of the following:

- Press the **S** key (works on all platforms).
- Right-click the mouse (works on *most* platforms).

Either of these actions straightens the transition perpendicular to the transition's source state or junction surface, if possible, and allows the transition source point to slide to maintain straightness. For states, if your pointer is out of range of perpendicularity with the source state, the transition is unaffected.

Labeling Transitions

Transition labels contain Stateflow action language that accompanies the execution of a transition. Creating and editing transition labels is described in the following topics:

- "Editing 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 2-14.

For more information on transition label contents, see Chapter 10, "Using Actions in Stateflow Charts".

Editing Transition Labels

Label unlabeled transitions as follows:

1. Select (left-click) the transition.

   The transition turns 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 [condition]{condition_action}/transition_action
```

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

Label Field	Description
`event`	Event 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.
`transition_action`	This action executes after the source state for the transition is exited but before the destination state is entered.

Transitions do not have to have labels. You can specify some, all, or none of the parts of the label. Valid transition labels are defined by the following:

- 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

Moving Transitions

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

- “Bowing the Transition Line” on page 4-22
- “Moving Transition Attach Points” on page 4-22
- “Moving Transition Labels” on page 4-22

In addition, transitions move along with the movements of states and junctions. See “Setting Smart Behavior in Transitions” on page 7-17 for a description of *smart* and *nonsmart* transition behavior.

Bowing 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.

Moving 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.

Moving 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 Stateflow Editor to deselect the label.

Changing 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 from the **Transition** shortcut menu:

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

2 Place your pointer over a selected transition and right-click to display the shortcut menu.

A menu of arrowhead sizes appears.

3 Select an arrowhead size from the menu.

To adjust arrowhead size from the **Junction** shortcut menu:

1 Select the junctions whose incoming arrowhead size you want to change.

2 Place your pointer over a selected junction and right-click.

3 In the resulting submenu, place your pointer over **Arrowhead Size**.

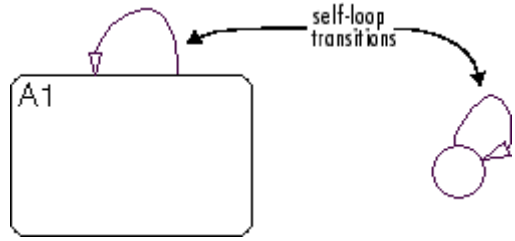
A menu of arrowhead sizes appears.

4 Select a size from the menu.

Creating Self-Loop Transitions

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

The following is an example of a self-loop transition:



To create a self-loop transition, follow these steps:

1. Create the transition as usual by clicking and dragging it out from the source state or junction.
2. 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 2-20.

Creating Default Transitions

A default transition is a transition with a destination (a state or a junction), but no apparent source object. See "Default Transitions" on page 1-12 for an explanation of 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 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 "Changing Transition Arrowhead Size" on page 4-23.

Default transitions can be labeled just like other transitions. See "Labeling Default Transitions" on page 2-26 for an example.

Changing Transition Properties

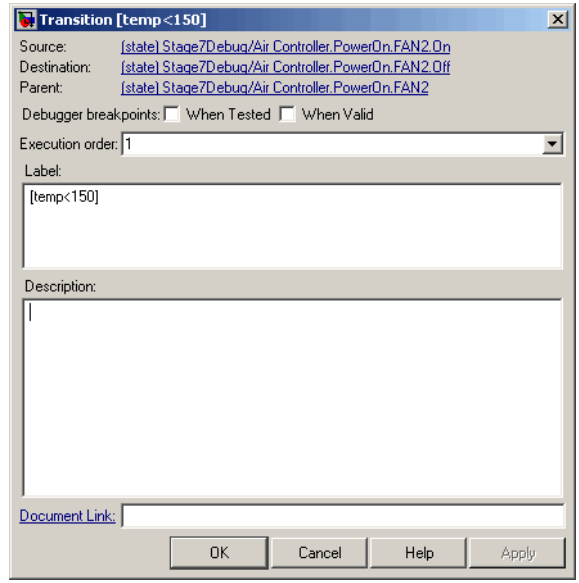
Use the Transition dialog box to view and change the properties for a transition. To access the Transitions dialog box for a particular transition, follow these steps:

1 Right-click on the transition.

A context menu appears.

2 Choose **Properties** from the context menu.

The Transition dialog box for the transitions appears.



The following table lists and describes the properties displayed for a transition in its Transition 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.
Debugger breakpoints	Select the check boxes to set debugging breakpoints either when the transition is tested for validity or when it is valid.
Execution order	The order in which the transition is executed.
Label	The transition's label. See "Transition Label Notation" on page 2-14 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`.

3 After making changes, select one of these options:

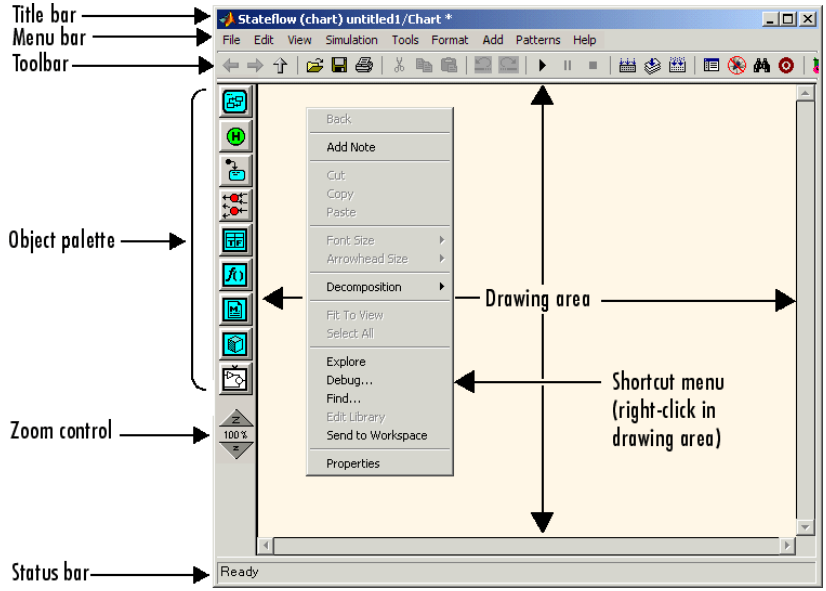
- **Apply** to save the changes and keep the Transition dialog box open.
- **Cancel** to return to the previous settings for the dialog box.
- **OK** to save the changes and close the dialog box.
- **Help** to display Stateflow online help in an HTML browser window.

Using the Stateflow Editor

In this section...

Stateflow Editor Window

You use the Stateflow Editor to draw, zoom, modify, print, and save a chart shown in the window. It has the following appearance:



The Stateflow Editor window includes the following elements:

- 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

  Most editor commands are available from the menu bar.

- 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 a chart's subchart hierarchy (see "Navigating Subcharts" on page 7-9).

- Object palette

Displays a set of tools for drawing states, transitions, and other state chart objects.

- Drawing area

  Displays an editable copy of a chart.

- Zoom control

  See "Viewing Stateflow Objects in the Model Explorer" on page 4-54 for information on using the zoom control.

- 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 Stateflow Editor, the shortcut menu applies to the chart object. See "Displaying the Context Menu for Objects" on page 4-29 for more information.

- Status bar

  Displays tool tips and status information.

Displaying the Context Menu for Objects

Every object that you create in a chart has a shortcut menu associated with it. To display the shortcut (context) menu, do the following:

1. Move your pointer over the object.
2. Right-click the object.

   A menu of operations that apply to the object appears.

To display the context menu for the chart object, do the following:

1. Move your pointer to an unoccupied location in the chart.

2 Right-click the location.

A menu of operations that apply to the chart appears.

Specifying Colors and Fonts in the Stateflow Editor

You can specify the color and font for items in the Stateflow Editor, as described in the following topics:

- "Changing Fonts for an Individual Text Item" on page 4-30 — Tells you how to set color and font for an individual item in the Stateflow Editor.
- "Using the Colors & Fonts Dialog Box" on page 4-30 — Shows you how to set default colors and fonts for all Stateflow Editor items in the Colors and Fonts dialog box.

Changing Fonts for an Individual Text Item

You can change the font for an individual text item as follows:

1 Right-click the text item.

2 In the context menu, select **Font Size** > *size of font*.

You can also specify the label font size of a particular object:

1 Left-click an individual text item in the Stateflow Editor.

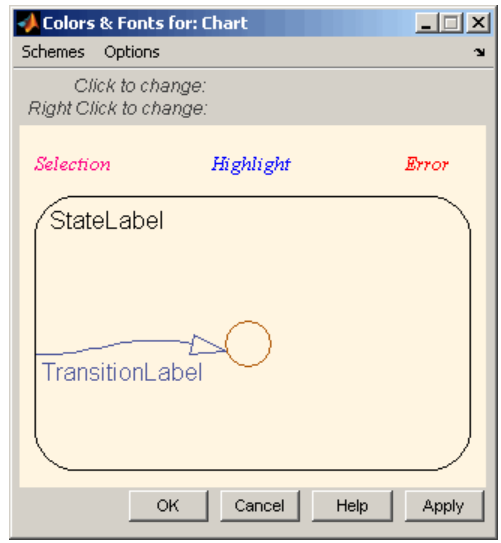
2 Select **Edit** > **Set Font Size**.

3 In the context menu, select the font size.

Using the Colors & Fonts Dialog Box

The Colors & Fonts dialog box allows you to specify a color scheme for a chart as a whole, or colors and label fonts for different types of objects in a chart.

To display the Colors & Fonts dialog box, select **Edit > Style** in the Stateflow Editor.



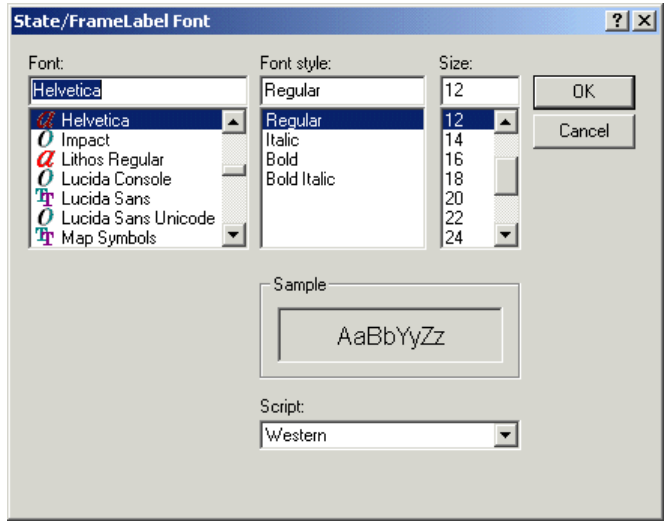
The drawing area of the dialog box displays examples of the types of objects whose colors and font labels you can specify. The examples use the colors and label fonts specified by the current color scheme for the chart. To choose another color scheme, select the scheme from the dialog box's **Schemes** menu. The dialog box displays the selected color scheme. Click **Apply** to apply the selected scheme to the chart or **OK** to apply the scheme and dismiss the dialog box.

To make the selected scheme the default scheme for all charts, select **Make this the "Default" scheme** from the dialog box's **Options** menu.

To modify the current scheme, position your pointer over the example of the type of object whose color or label font you want to change. Then left-click to change the object's color or right-click to change the object's font. If you left-click, a color chooser dialog box appears.

Use the dialog box to select a new color for the selected object type.

If the selected object is a label and you right-click, a font selection dialog box appears.



Use the font selector to choose a new font for the selected label.

To save changes to the default color scheme, select **Save defaults to disk** from the Colors & Fonts dialog box's **Options** menu.

Note Choosing **Save defaults to disk** has no effect if the modified scheme is not the default scheme.

Differentiating Syntax Elements in the Stateflow Action Language

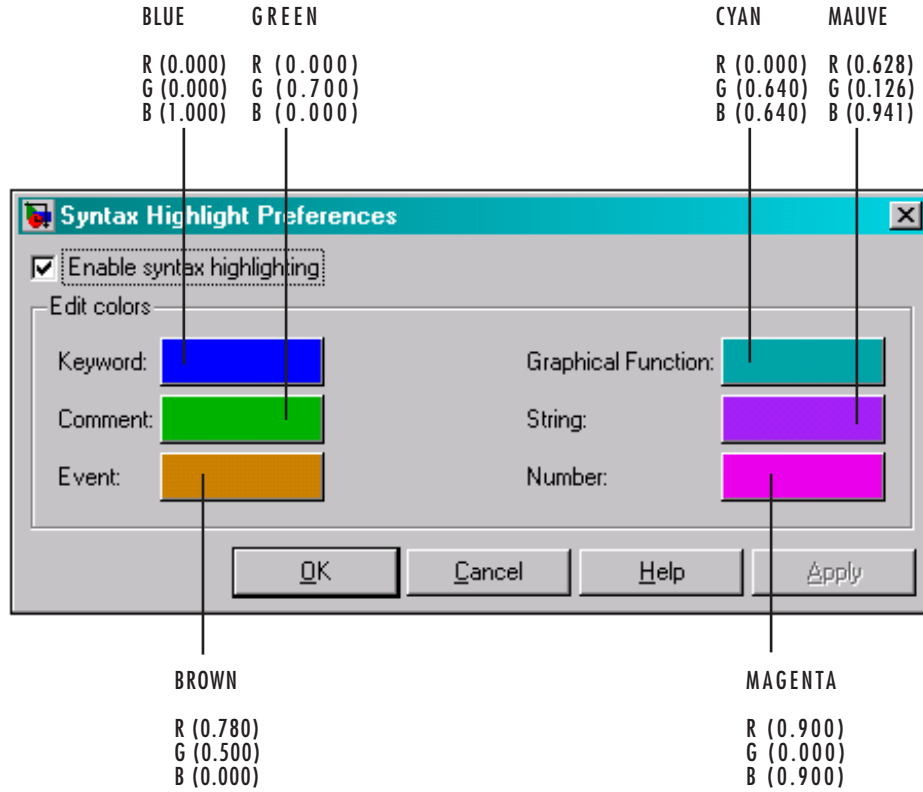
This release gives you the option of using color highlighting to differentiate the following syntax elements in the Stateflow action language:

- Keyword
- Comment
- Event
- Graphical function
- String
- Number

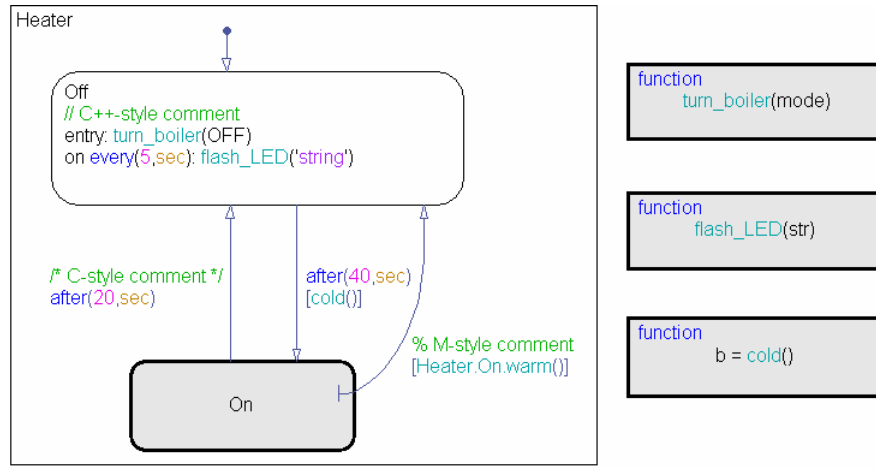
Note Syntax highlighting is a user preference, not a model preference.

Default Syntax Highlighting

Syntax highlighting is enabled by default, assigning the following colors to syntax elements:



This Stateflow chart illustrates the default highlighting for each language element:



If the parser cannot resolve a syntax element, the Stateflow Editor displays the element in the default text color.

To modify color assignments, see "Editing Syntax Highlighting" on page 4-35. To disable syntax highlighting, see "Enabling and Disabling Syntax Highlighting" on page 4-35.

Editing Syntax Highlighting

To edit syntax highlighting, follow these steps:

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

   The Syntax Highlight Preferences dialog box appears.

2. Click the color 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.

Enabling and Disabling Syntax Highlighting

You can toggle syntax highlighting from the **Tools** and **Edit** menus in the Stateflow Editor.

- From the **Tools** menu, select or clear **Syntax Coloring**.
- From the **Edit** menu, follow these steps:

  1 Select **Highlighting Preferences** to open the Syntax Highlight Preferences dialog box.

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

Selecting and Deselecting Graphical Objects

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

Select objects in the Stateflow Editor as follows:

- 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 you want to select, and then release the mouse button.

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

  This action 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 when you select a state and all of its objects and then Shift-click inside the containing state to deselect it.
- To select all objects in the Stateflow chart, from the **Edit** menu select **Select All**.

  You can also select all objects by selecting **Select All** from the right-click shortcut menu.
- To deselect all selected objects, press the **Esc** key.

  Pressing the **Esc** key again displays the parent of the current chart.

When an object is selected, it is highlighted in the color set as the selection color (blue by default; see “Specifying Colors and Fonts in the Stateflow Editor” on page 4-30 for more information).

Cutting and Pasting 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 Stateflow chart to another. The Stateflow Editor 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, select the object and choose **Cut** from one of the following:

- The **Edit** menu on the Stateflow Editor
- The right-click shortcut menu

To paste the most recently cut selection of objects, choose **Paste** from either of the following:

- The **Edit** menu on the Stateflow Editor
- The right-click shortcut menu

Copying 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 Stateflow Editor clipboard.

Note If you copy and paste a state in the Stateflow Editor, these rules apply.

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

Alternatively, to copy from one Stateflow chart to another, choose the **Copy** and then **Paste** menu items from one of these sources:

- The **Edit** menu in the Stateflow Editor window
- The right-click shortcut menu

Using Alignment, Distribution, and Resizing Commands for Chart Objects

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

- Alignment
- Distribution
- Resizing

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

- States
- Functions
- Boxes
- Junctions

Workflow for Aligning, Distributing, or Resizing Chart Objects

The basic workflow to align, distribute, or resize chart objects is:

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

   If necessary, select **User specified state/transition execution order** in the Chart properties dialog box.

   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 problem from occurring. For more information, see "Execution Order for Parallel States" on page 3-39 and "Evaluation Order for Outgoing Transitions" on page 3-21.

   ---

2. Decide which chart objects you want to align, distribute, or resize.

3 In the Stateflow Editor, select these objects in any order.

You can select objects one-by-one or by drawing a box around them.

4 Decide which object is the reference (or anchor) by which to align, distribute, or resize other chart objects.

5 Ensure that brackets appear around the reference object.

If necessary, right-click the object to mark it with brackets.

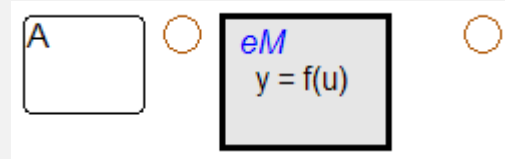
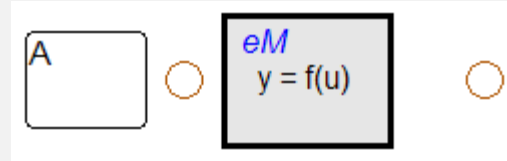
Note If you select objects one-by-one, the last object that you select acts as the reference.

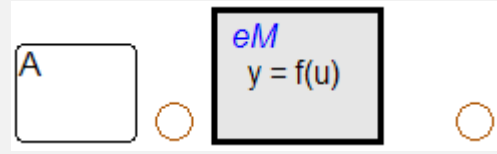
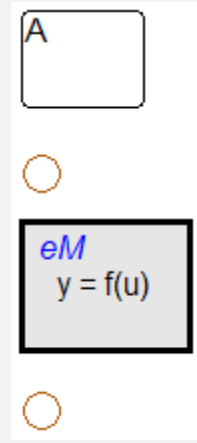
---

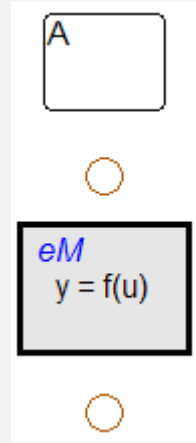
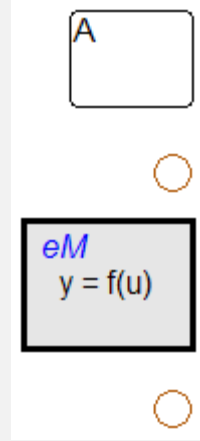
6 Select an option from the **Format** menu to align, distribute, or resize your chosen objects.

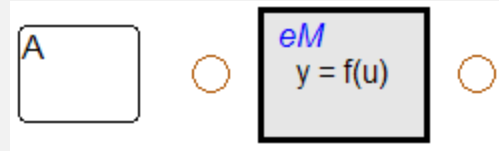
Options for Aligning, Distributing, and Resizing Chart Objects

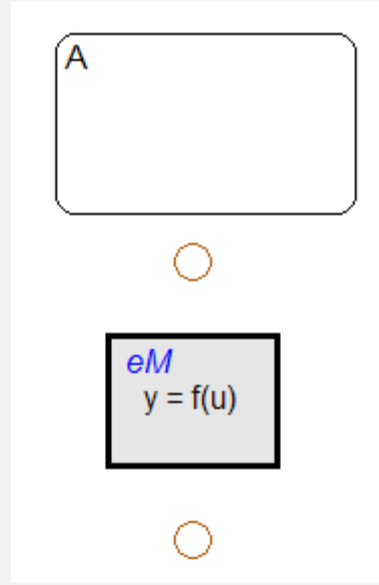
The following tables describe the options in the **Format** menu.

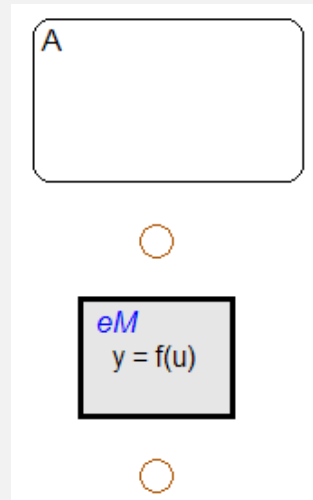
Option in the Align Items Submenu	Description	Example
Align Top Edges	Align selected objects along the top edges.	
Align Centers Horizontally	Align selected objects such that the centers fall on the same horizontal line.	

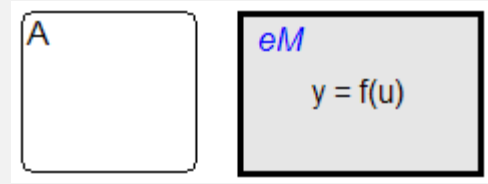
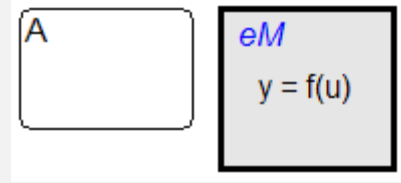
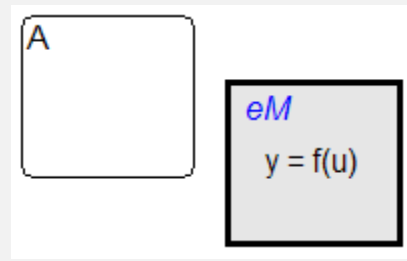
Option in the Align Items Submenu	Description	Example
Align Bottom Edges	Align selected objects along the bottom edges.	
Align Left Edges	Align selected objects along the left edges.	

Option in the Align Items Submenu	Description	Example
Align Centers Vertically	Align selected objects such that the centers fall on the same vertical line.	
Align Right Edges	Align selected objects along the right edges.	

Option in the Distribute Items Submenu	Description	Example
Distribute Items Horizontally	Distribute selected objects such that the center-to-center horizontal distance between any two objects is the same. **Note** The amount of 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.	

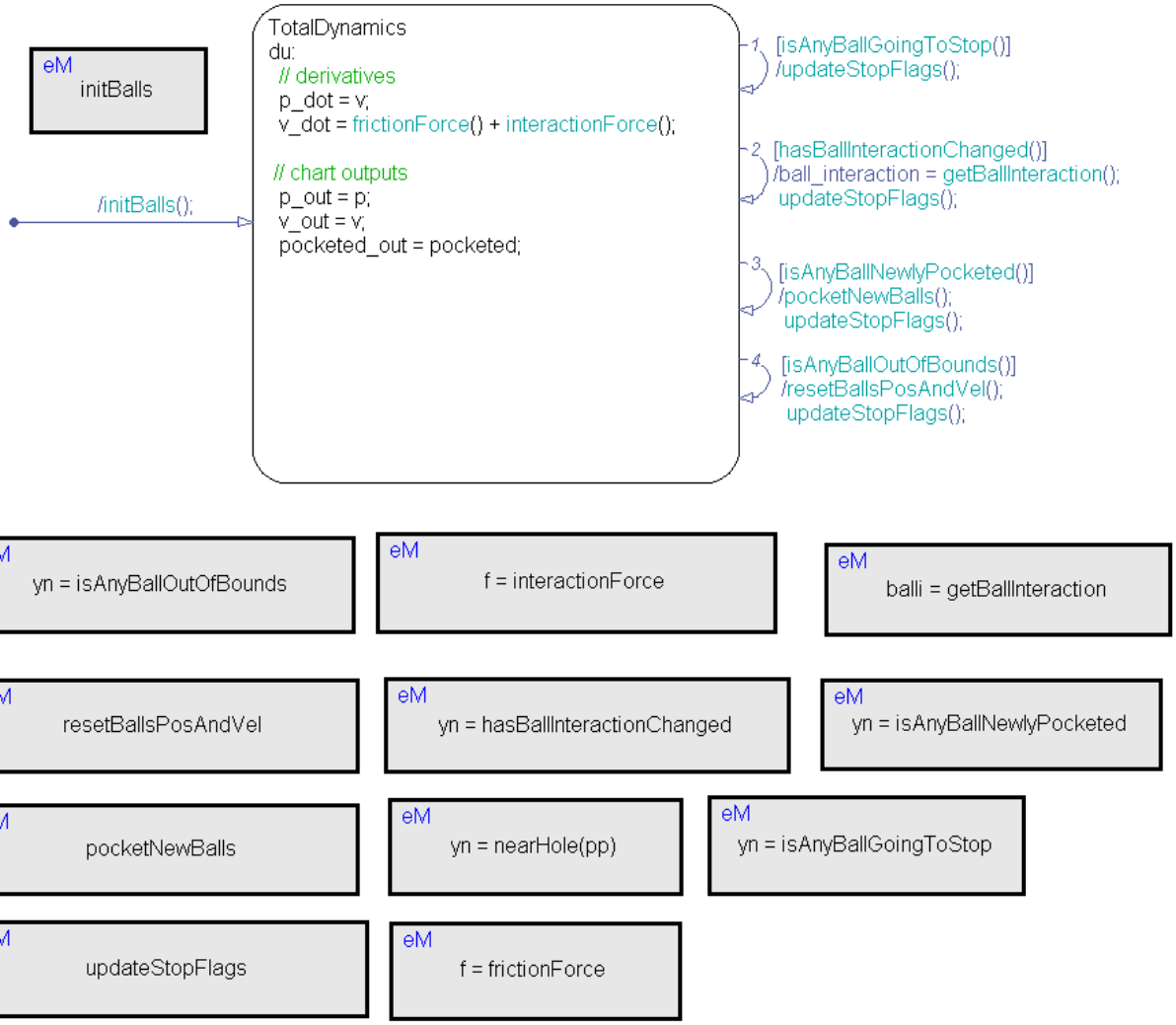
Option in the Distribute Items Submenu	Description	Example
Distribute Items Vertically	Distribute selected objects such that the center-to-center vertical distance between any two objects is the same. **Note** The amount of vertical space for distribution is the distance between the top edge of the highest 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.	

Option in the Distribute Items Submenu	Description	Example
Make Horizontal Gaps Even	Distribute selected objects such that the horizontal white space between any two objects is the same. **Note** The space restriction for **Distribute Items Horizontally** applies.	
Make Vertical Gaps Even	Distribute selected objects such that the vertical white space between any two objects is the same. **Note** The space restriction for **Distribute Items Vertically** applies.	

Option in the Resize Items Submenu	Description	Example
Make Items Same Height	Make selected objects have the same height.	
Make Items Same Width	Make selected objects have the same width.	
Make Items Same Size	Make selected objects have the same height and width.	

Example of Using Alignment Commands for Chart Objects

Suppose that you have a chart with multiple Embedded MATLAB functions.



To align the three Embedded MATLAB functions on the right, follow these steps:

1 Type `sf_pool` at the MATLAB command prompt.

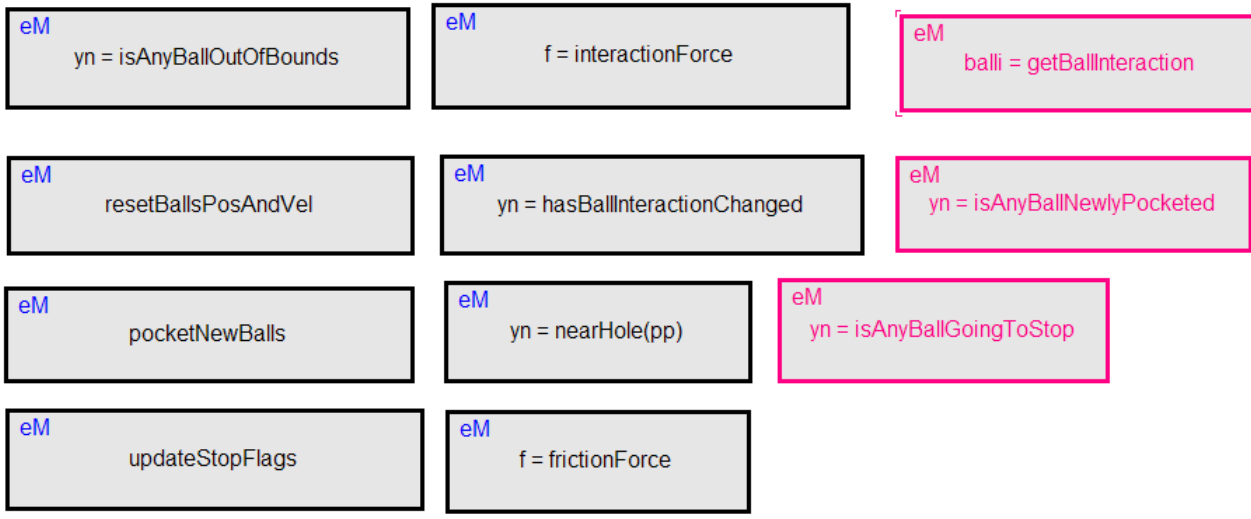
Note Expand the size of the Stateflow Editor to see the entire chart.

2 In the Stateflow Editor, click the Embedded MATLAB function `isAnyBallGoingToStop`.

3 Shift-click the Embedded MATLAB function `isAnyBallNewlyPocketed`.

4 Shift-click the Embedded MATLAB function `getBallInteraction`.

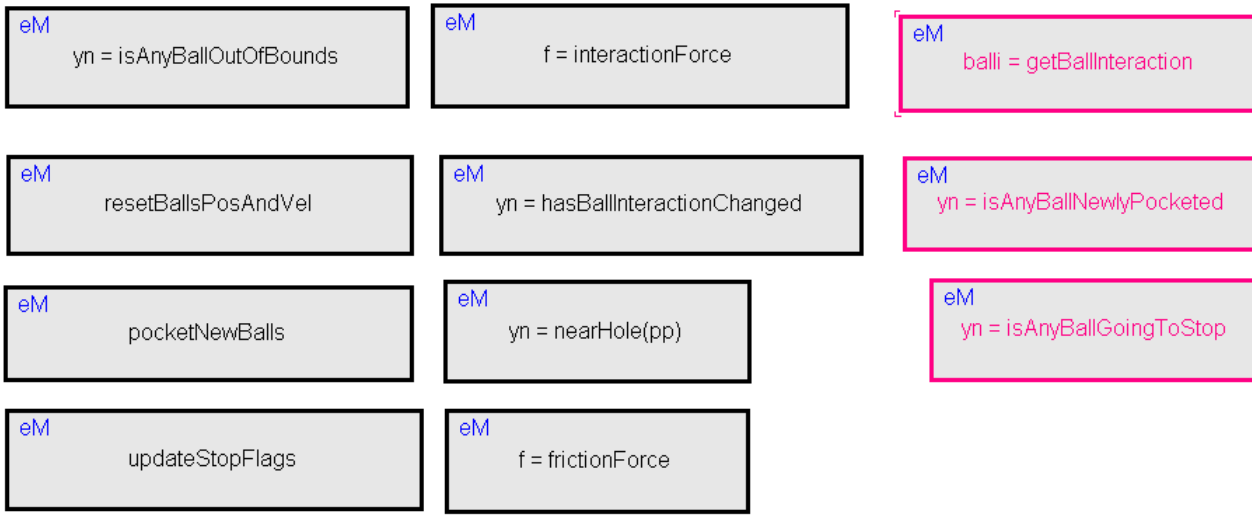
This object is the reference (or anchor) for aligning the three functions. Note that the function is marked by brackets.



5 Select **Format > Align Items > Align Right Edges**.

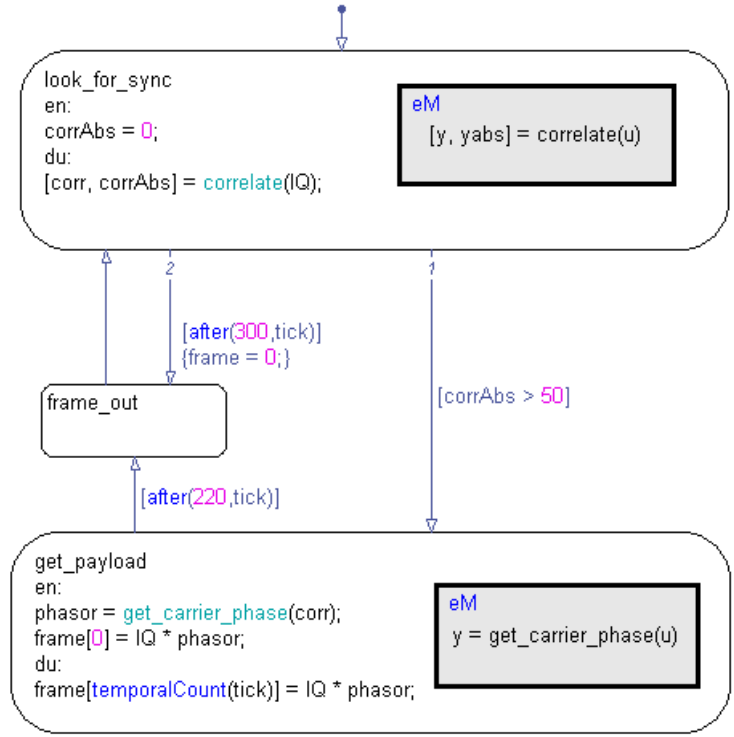
Note You can also right-click and select that option from the **Format** context menu.

This action aligns the right edges of the three functions based on the right edge of `getBallInteraction`.



Example of Using Distribution Commands for Chart Objects

Suppose that you have a chart with three states.



To distribute the three states vertically, follow these steps:

1. Type `sf_frame_sync_controller` at the MATLAB command prompt.

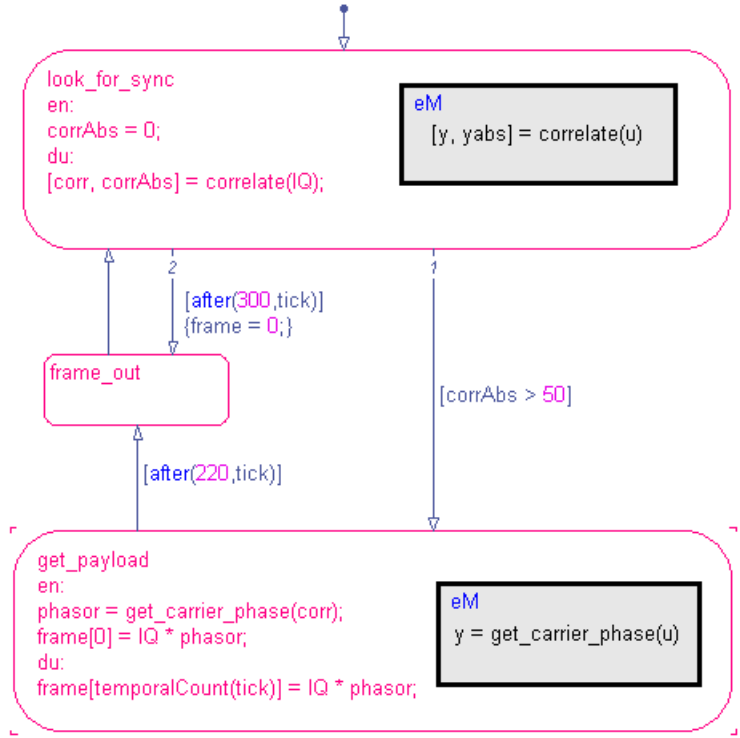
   Note Double-click the Frame Sync Controller block to open the chart.

2. In the Stateflow Editor, select the three states in any order.

3. Select **Format > Distribute Items > Make Vertical Gaps Even**.

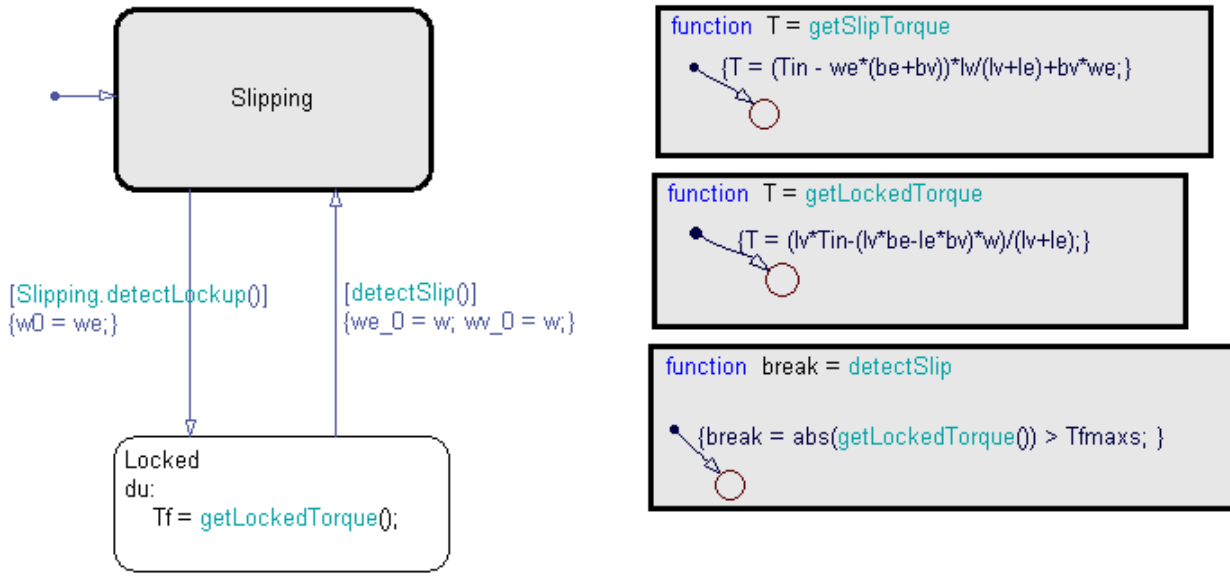
Note You can also right-click and select that option from the **Format** context menu.

This action ensures that the vertical spacing between any two states is the same.



Example of Using Resizing Commands for Chart Objects

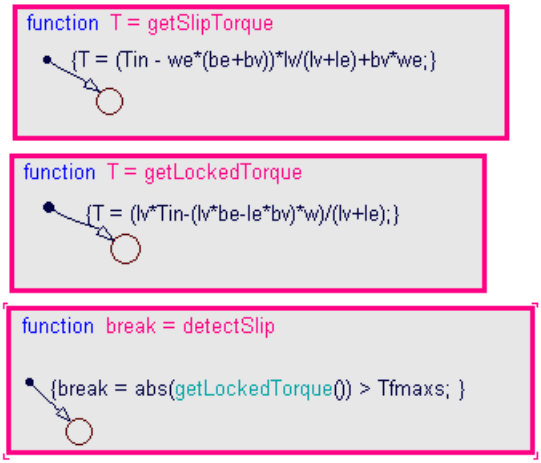
Suppose that you have a chart where the graphical functions have different sizes.



To resize the graphical functions to match the size of `detectSlip`, follow these steps:

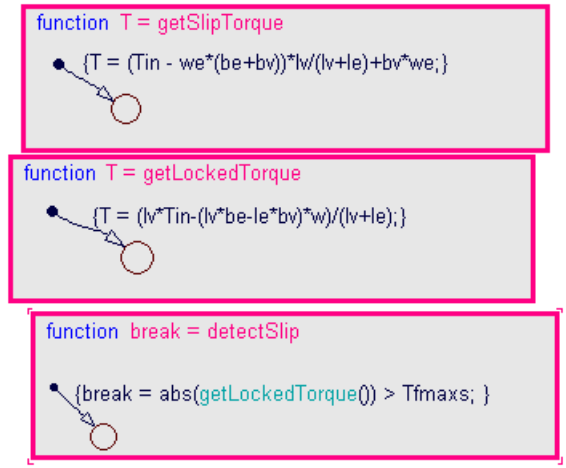
1. Type `sf_clutch` at the MATLAB command prompt.
2. In the Stateflow Editor, draw a box around the three graphical functions.
3. Ensure that brackets appear around the function box for `detectSlip`.

If necessary, right-click the function to mark it with brackets.



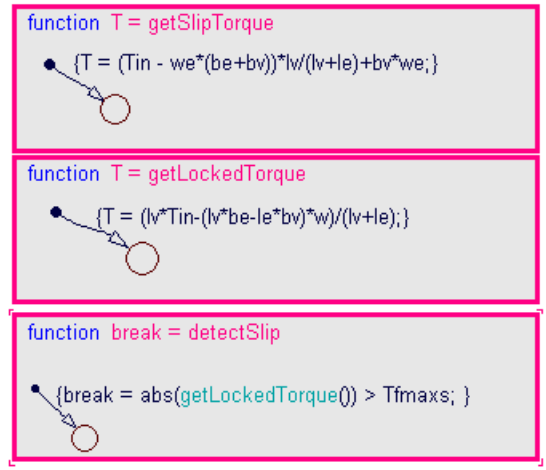
4 Select **Format > Resize Items > Make Items Same Size**.

This action ensures that the three functions are the same size.

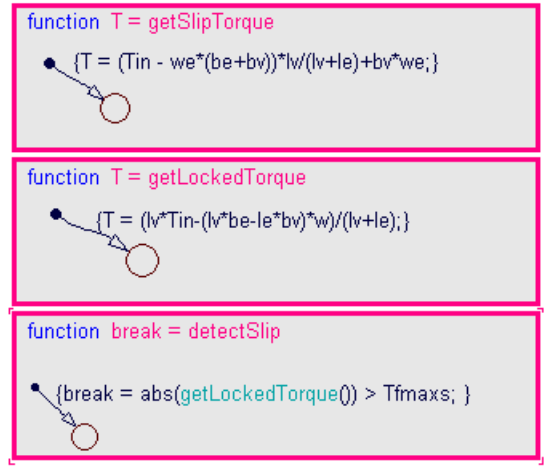


5 To make other corrections, you can follow these steps:

a To align the functions, select **Format > Align Items > Align Left Edges**.



b To distribute the functions, select **Format > Distribute Items > Make Vertical Gaps Even**.



Editing Object Labels

Some Stateflow objects (for example, states and transitions) have labels. To change these labels, place your pointer anywhere in the label and click. Your pointer changes to an I-beam. You can then edit the text.

You can use the shortcut (context) menu to change a label's font size:

1. Select the states whose label font size you want to change.
2. Right-click to display the shortcut menu.
3. Place your pointer over the **Font Size** menu item.

   A menu of font sizes appears.
4. Select the desired font size from the menu.

   The Stateflow Editor changes the font size of all labels on all selected states to the selected size.

Viewing Stateflow Objects in the Model Explorer

To view or modify Stateflow Editor objects in the Model Explorer, follow these steps:

1. Position your pointer over the state.
2. Right-click to display the context menu for the state.
3. Select **Explore** from the context menu.

   The Model Explorer opens (if not already open) and highlights the state in the left hierarchy pane to show any data or events defined by the state.

To view data and events defined by the parent state of a transition or junction, select **Explore** from the transition or junction's context menu.

The Model Explorer is the only place where you can view and modify data and events. See Chapter 8, "Defining Data" and Chapter 9, "Defining Events" for more details on using the Model Explorer to view, add, delete, and modify data and events for Stateflow objects. See also "Using the Model Explorer

with Stateflow Objects" on page 24-2 for more details on using the Model Explorer to view Stateflow objects.

Zooming a Chart

You can magnify or shrink a chart, using the following zoom controls:

- **Zoom Factor Selector**. Selects a zoom factor (see "Using the Zoom Factor Selector" on page 4-55).
- **Zoom In** button. Zooms in by the current zoom factor.

  You can also press the **R** key to increase the zoom factor.
- **Zoom Out** button. Zooms out by the current zoom factor.

  You can also press the **V** key to decrease the zoom factor.

Using the Zoom Factor Selector

The **Zoom Factor Selector** allows you to specify the zoom factor by

- Choosing a value from a menu.

  Click the selector to display the menu.
- Double-clicking the **Zoom Factor Selector** selects the zoom factor that will fit the view to all selected objects or all objects if none are selected.

  You can achieve the same effect by choosing **Fit to View** from the right-click context menu or by pressing the **F** key to apply the maximum zoom that includes all selected objects. Press the space bar to fit all objects to the view.
- Clicking the **Zoom Factor Selector** and dragging up or down.

  Dragging the mouse upward increases the zoom factor. Dragging the mouse downward decreases the zoom factor. Alternatively, right-clicking and dragging on the percentage value resizes while you are dragging.

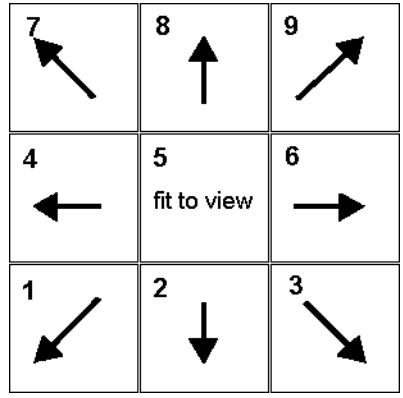
Zooming with Shortcut Keys

This table is a summary of the shortcut keys you can use to perform some of the zooming operations described above:

Key	Zoom Operation
F	Highlight (select) an object and press the **F** key to fit it to view.
space bar	Set to full view of chart.
R or **+**	Increase zoom factor.
V or **-**	Decrease zoom factor.

Moving in Zoomed Charts with Shortcut Keys

You can also use number keys to move in zoomed charts according to their layout in the number keypad:



You can enter numbers for moving from the number keys above the alphabetic keys at any time or from the number keypad if **NumLock** is engaged for the keyboard. The **5** key fits the currently selected object to full view. If no object is selected, the entire chart is fit to view.

Zooming a Chart Object Using the Stateflow API

How to Zoom a Chart Object

Use the Stateflow API method `fitToView` to zoom in on a graphical object in the Stateflow Editor. (See "Using the API" in the Stateflow API documentation for information about obtaining object handles.)

Objects You Can Zoom

You can zoom these objects in the Stateflow Editor:

- Charts
- Subcharts
- States
- Transitions
- Graphical functions
- Truth tables
- Embedded MATLAB functions
- Simulink functions
- Connective junctions
- History junctions
- Boxes
- Notes

Example of Zooming States in a Chart

Follow these steps to zoom in on different states:

1. At the MATLAB command prompt, type:

   ```
   old_sf_car;
   ```

   The chart `shift_logic` appears in the Stateflow Editor.

2. To define an object handle for the chart `shift_logic`, type:

   ```
   myChart = find(sfroot,'-isa','Stateflow.Chart','name', ...
   'shift_logic');
   ```

3. To define an object handle for the state `upshifting`, type:

   ```
   myState = find(sfroot,'-isa','Stateflow.State','name', ...
   'upshifting');
   ```

4 To zoom in on the state `upshifting`, type:

```
myState.fitToView;
```

The Stateflow Editor zooms in on the state and highlights it.

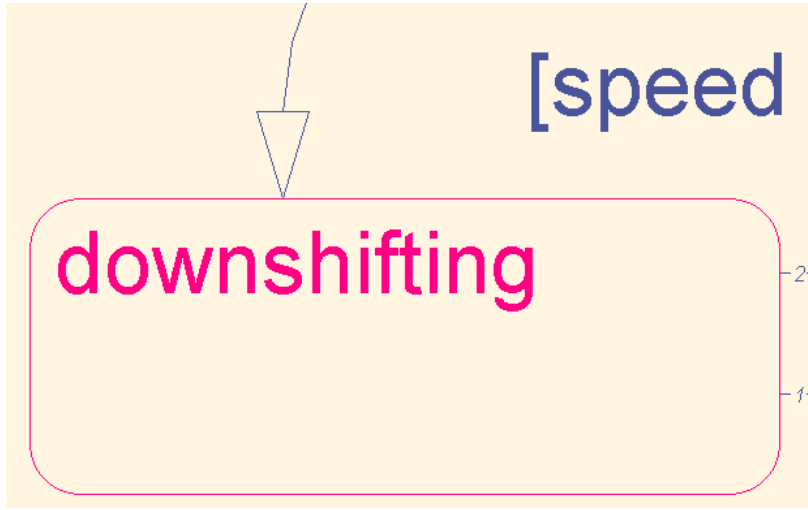
5 To define an object handle for the state `downshifting`, type:

```
myState = find(sfroot,'-isa','Stateflow.State','name', ...
'downshifting');
```

6 To zoom in on the state `downshifting`, type:

```
myState.fitToView;
```

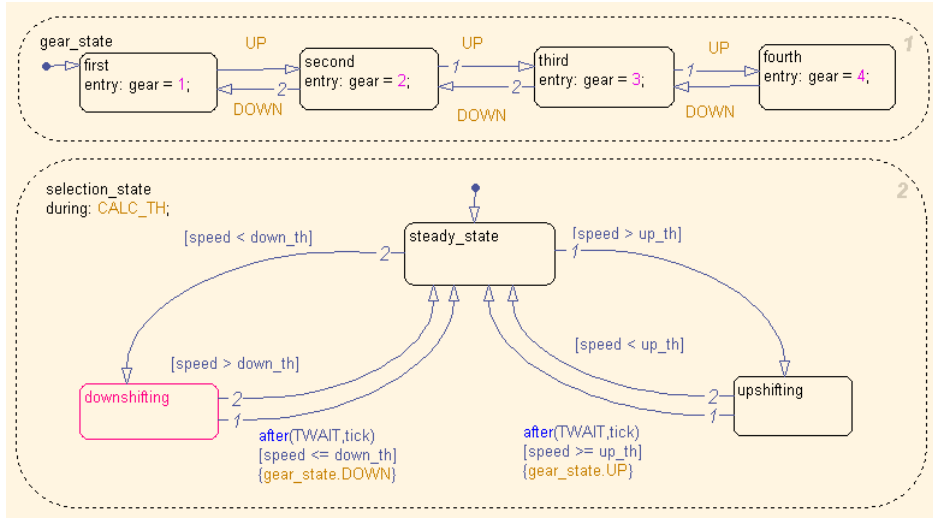
The Stateflow Editor zooms in on the state and highlights it.



7 To zoom back to the chart level in the Stateflow Editor, type:

```
myChart.fitToView;
```

The chart `shift_logic` reappears, as shown below.



8 You can also zoom in on a state using the `sfgco` function. Follow these steps:

   a Click any state in the chart.

   b At the MATLAB command prompt, type:

```
myState = sfgco;
```

      This command assigns the selected state to the object handle `myState`.

   c To zoom in on the selected state, type:

```
myState.fitToView;
```

      The Stateflow Editor zooms in on the state and highlights it.

Undoing and Redoing Editor Operations

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

To undo an operation in the Stateflow Editor, do one of the following:

- Select the **Undo** icon in the toolbar of the Stateflow Editor  .

  When you place your pointer over the **Undo** button, the tool tip that appears indicates the nature of the operation to undo.
- From the **Edit** menu, select **Undo.**

To redo an operation in the Stateflow Editor, do one of the following:

- Select the **Redo** icon in the toolbar of the Stateflow Editor  .

  When you place your pointer over the **Redo** button, the tool tip that appears indicates the nature of the operation to redo.
- From the **Edit** menu, select **Redo.**

Exceptions for Undo

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

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

  To understand subcharting, see "Using Subcharts to Extend Charts" on page 7-5.

- 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 "Drawing a Supertransition Into a Subchart" on page 7-11 and "Drawing a Supertransition Out of a Subchart" on page 7-14.

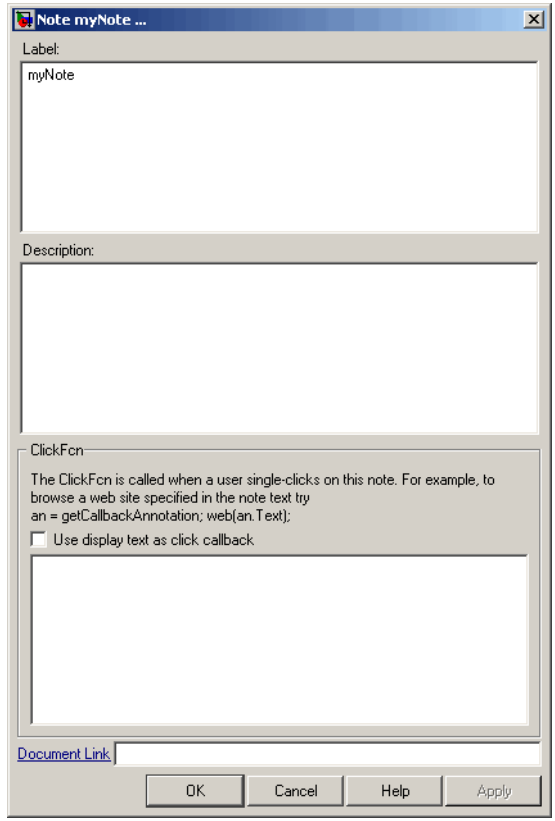
- You cannot undo any changes made to the Stateflow Editor through the Stateflow API.

  For a description of the Stateflow API (Application Programming Interface), see "Using the API" in the Stateflow API Guide.

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

Stateflow Chart Notes Dialog Box

You can use the chart notes dialog box to edit note properties.



The Note dialog box contains the following properties for a chart note:

Field	Description
Label	The label for the note. This includes the name of the note and its associated actions.
Description	Textual description/comment.

Field	Description
Use display text as click callback	Checking this option causes a Simulink model to treat the text in the Text field as the note's click function. The specified text must be a valid MATLAB expression comprising symbols that are defined in the MATLAB workspace when the user clicks this annotation. Note that selecting this option disables the **ClickFcn** edit field.
ClickFcn	Specifies MATLAB code to be executed when a user single-clicks this annotation. The Simulink model stores the code entered in this field.
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`.

See "Annotation Callback Functions" in the Simulink User's Guide for a description of the **ClickFcn** edit field.

Keyboard Shortcuts for Stateflow Charts

This table gives a comprehensive list of keyboard shortcuts for the Stateflow Editor.

Task	Windows® platform	UNIX® platform
Display the parent of the currently displayed chart or subchart. There is no limit on the time between the entry of each period.	.. (two periods)	.. (two periods)
Zoom in by an incremental amount.	**+** or **r** or **R**	**+** or **r** or **R**
Zoom out by an incremental amount.	**-** or **v** or **V**	**-** or **v** or **V**
Fit chart to screen.	**0** or **Space Bar**	**0** or **Space Bar**
Zoom to normal view.	**1**	**1**

Task	Windows® platform	UNIX® platform
Move the current Stateflow Editor view down within the full chart.	**2**	**2**
Move the current Stateflow Editor view down and right within the full chart.	**3**	**3**
Move the current Stateflow Editor view left within the full chart.	**4**	**4**
Fit the currently selected object to full view. If no object is selected, the chart is fit to full view.	**5**	**5**
Move the current Stateflow Editor view right within the full chart.	**6**	**6**
Move the current Stateflow Editor view up and left within the full chart.	**7**	**7**
Move the current Stateflow Editor view up within the full chart.	**8**	**8**
Move the current Stateflow Editor view up and right within the full chart.	**9**	**9**
Delete the selected objects.	**Delete**	**Delete**
Access the contents of the currently highlighted subchart or truth table.	**Enter**	**Enter**

Task	Windows® platform	UNIX® platform
Perform any of the following actions: • If you are editing the label of an object, the **Esc** key disables label editing but leaves the object selected. • If objects are selected, the **Esc** key deselects all objects in the current view. • If the current chart view is the contents of a subchart and no object is selected, the **Esc** key changes the view to the parent of the subchart. • If the current chart view is at the chart level and no object is selected, the **Esc** key displays the Simulink model window for that chart's block.	**Esc**	**Esc**
Fit the currently selected object to screen. If no object is selected, the chart is fit to screen.	**f** or **F**	**f** or **F**
Pan left	**d** or **D** or **Ctrl+Left Arrow**	**d** or **D** or **Ctrl+Left Arrow**
Pan right	**g** or **G** or **Ctrl+Right Arrow**	**g** or **G** or **Ctrl+Right Arrow**
Pan up	**e** or **E** or **Ctrl+Up Arrow**	**e** or **E** or **Ctrl+Up Arrow**
Pan down	**c** or **C** or **Ctrl+Down Arrow**	**c** or **C** or **Ctrl+Down Arrow**
Go back in pan/zoom history	**b** or **B**	**b** or **B**
Go forward in pan/zoom history	**t** or **T**	**t** or **T**

Task	Windows® platform	UNIX® platform
Select the first state, function, truth table, or box parented (contained) by the currently selected object in the same chart. Selection order of contained objects is top-down, left-right. See also **u** key.	**j** (jump) or **J**	**j** (jump) or **J**
Select the next state, function, truth table, or box at the same containment level. Selection order of objects is top-down, left-right.	**n** (next) or **N**	**n** (next) or **N**
Select the previous state, function, truth table, or box at the same containment level. Selection order of objects is top-down, left-right.	**p** (previous) or **P**	**p** (previous) or **P**
Select the parent object of the currently highlighted object in the same chart. See also **j** key.	**u** (up) or **U**	**u** (up) or **U**

Customizing the Stateflow Editor

You can write M-code to customize the Stateflow Editor by:

- Adding items and submenus to the end of Stateflow Editor menus (see "Adding Items to Stateflow Editor Menus" on page 4-66)
- Disabling and hiding items on menus in the Stateflow Editor (see "Disabling and Hiding Stateflow Editor Menu Items" on page 4-69)

Adding Items to Stateflow Editor Menus

You use the Simulink customization manager to add items, including submenus, to the end of menus in the Stateflow Editor. For example, you can add menu items that invoke your own M-code functions.

To add an item to the end of a Stateflow Editor menu, you must create the following functions in an `sl_customization.m` file on the MATLAB path:

- For each item, create a schema function, which defines a custom item on a menu owned by the Stateflow Editor.
- Create a custom menu function, which registers schema functions that define custom items that you want to add to a menu.
- Define the `sl_customization` function to register the custom menu function with the Simulink customization manager.
- Create callback functions for the items that you add to the Stateflow Editor menus.

For detailed descriptions of these procedures, see "Adding Items to Model Editor Menus" in the Simulink User's Guide.

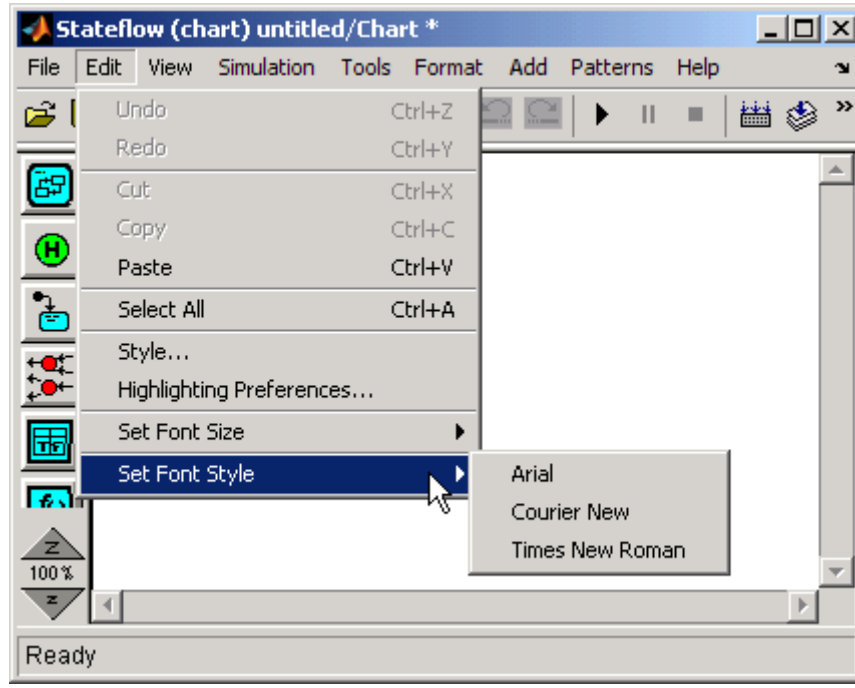
Code Example: Adding a Custom Submenu to the Stateflow Editor. The following `sl_customization.m` file adds a submenu called **Set Font Style** to the Stateflow Editor's Edit menu. The submenu contains three menu options for font style: **Arial**, **Courier New**, and **Times New Roman**. Your `sl_customization` function should accept one argument, a handle to an object called the `Simulink.CustomizationManager`. For example, you can set `cm = sl_customization_manager` at the MATLAB command line.

```
function sl_customization(cm)

  %% Register custom menu function.
  cm.addCustomMenuFcn('Stateflow:EditMenu', @getMyMenuItems);
end

%% Define the custom submenu function.

function schemaFcns = getMyMenuItems(callbackInfo)
  schemaFcns = {@getItem4};
end

%% Define the schema function for first submenu item
function schema = getItem1(callbackInfo)
  schema = sl_action_schema;
  schema.label = 'Arial';
  schema.userdata = 'font style Arial';
  schema.callback = @myCallback1;
end
```

```
%% Define the schema function for second submenu item.
function schema = getItem2(callbackInfo)
  schema = sl_action_schema;
  schema.label = 'Courier New';
  schema.userdata = 'font style Courier New';
  schema.callback = @myCallback1;

%% Define the schema function for third submenu item.
function schema = getItem3(callbackInfo)
  schema = sl_action_schema;
  schema.label = "Times New Roman';
  schema.userdata = 'font style Times New Roman';
  schema.callback = @myCallback1;
end

function myCallback1(callbackInfo)
  disp(['Callback for 'callbackInfo.userdata' was called']);
end

function schema = getItem4(callbackInfo)
  % Make a submenu label 'Set Font Style'
  % with the font styles defined in menu items above.
  schema = sl_container_schema;
  schema.label = 'Set Font Style';
  schema.childrenFcns = {@getItem1, @getItem2, @getItem3};
end
```

Note The `addCustomMenuFcn` function requires that you pass a string argument that identifies the menu or menu item you wish to customize. To determine the appropriate tag, see "Displaying Menu Tags" on page 4-71.

Custom Menu Example: Set Font Style. When you run `sl_customization(cm)` described in "Code Example: Adding a Custom Submenu to the Stateflow Editor" on page 4-67, the following new submenu appears in the Stateflow Editor.



Disabling and Hiding Stateflow Editor Menu Items

You can disable or hide items that appear on Stateflow Editor menus by

- Creating a filter function that disables or hides the menu item (see "Creating a Filter Function" in the Simulink User's Guide)
- Registering the filter function with the Simulink customization manager (see "Registering a Filter Function" in the Simulink User's Guide)

For detailed descriptions of these procedures, see "Disabling and Hiding Model Editor Menu Items" in the Simulink User's Guide.

Code Example: Disabling the Print Command in the Stateflow Editor.

The following `sl_customization.m` file disables the Print command in the File menu of the Stateflow Editor. The example assumes you set `cm = sl_customization_manager`.

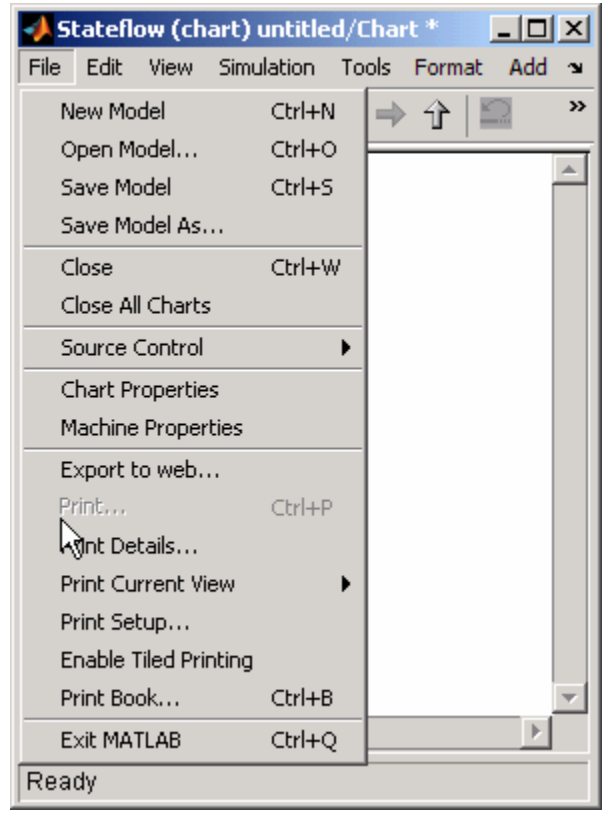
```
function sl_customization(cm)

  %%Register custom filter function.
  cm.addCustomFilterFcn('Stateflow:PrintMenuItem', @myFilter);

end

function state = myFilter(callbackInfo)
  state = 'Disabled';
end
```

Note The `addCustomFilterFcn` function requires that you pass a string argument that identifies the menu or menu item you wish to disable or hide. To determine the appropriate tag, see "Displaying Menu Tags" on page 4-71.

The `myFilter` function sets the state of the menu item. Valid states are:

- 'Hidden'
- 'Disabled'
- 'Enabled'

Custom Menu Example: Disable Print Menu Item. After you run `sl_customization(cm)` described in "Code Example: Disabling the Print Command in the Stateflow Editor" on page 4-69, the Stateflow Editor's File menu looks like this:

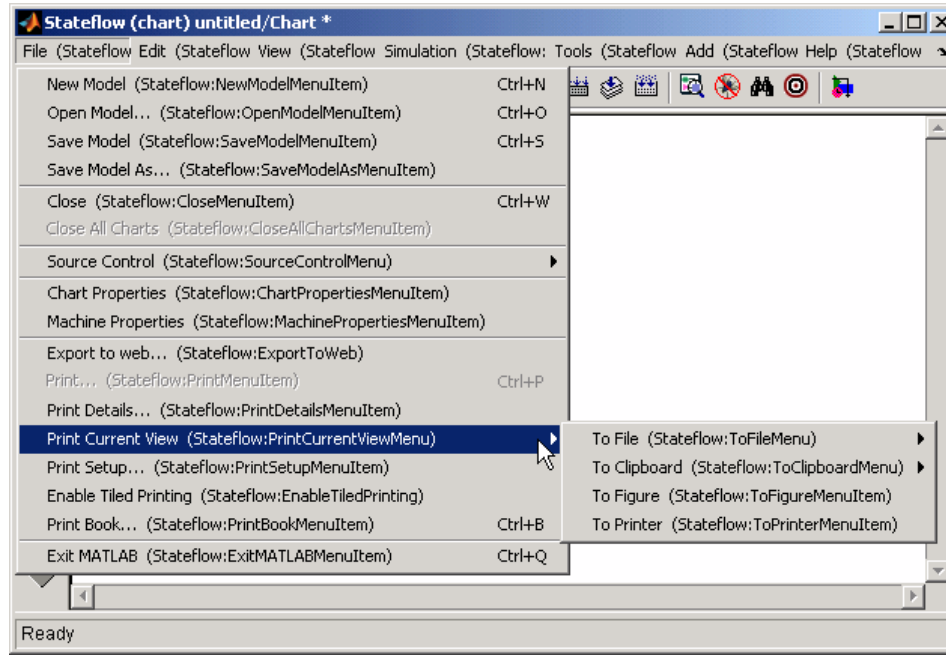


Displaying Menu Tags

To determine the tags that identify the menus or menu items you wish to customize on the Stateflow Editor, set the Simulink customization manager's `showWidgetIdAsToolTip` property to true by entering the following commands at the MATLAB command line:

```
cm = sl_customization_manager;
cm.showWidgetIdAsToolTip = true;
```

After enabling this property, the tag of each menu or menu item appears next to its label in the Stateflow Editor:



To turn off tag display, enter the following command at the MATLAB command line:

```
cm.showWidgetIdAsToolTip = false;
```

Note Some Stateflow Editor menu items may not work while menu tags are displayed. Thus, you should turn off menu tag display before attempting to use the menus.

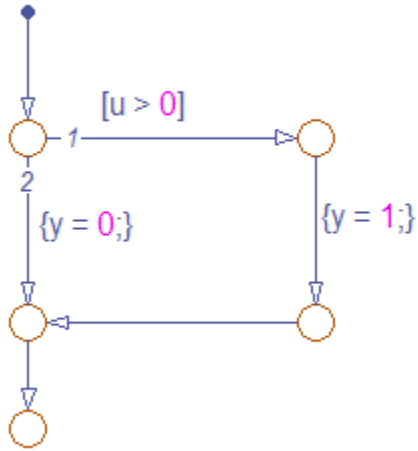
5

Modeling Logic Patterns and Iterative Loops Using Flow Graphs

What Is a Flow Graph?

A flow graph is a graphical construct that models logic patterns by using connective junctions and transitions. The junctions provide decision branches between alternate transition paths. You can use flow graphs to represent decision and iterative loop logic.

Here is an example of a flow graph that models simple `if-else` logic:



This flow graph models the following code:

```
if (u > 0)
{
   y = 1;
}
else
{
   y = 0;
}
```

Difference Between Flow Graphs and State Charts

A flow graph is known as a *stateless flow chart* because it cannot maintain its active state between updates. As a result, a flow graph always begins executing from a default transition and ends at a *terminating junction* (a junction that has no valid outgoing transitions).

By contrast, a state chart stores its current state in memory to preserve local data and activity between updates. As a result, state charts can begin executing where they left off in the previous time step, making them suitable for modeling reactive or supervisory systems that depend on history. In these kinds of systems, the current result depends on a previous result. For more information, see “What Is State?” on page 6-4 and Chapter 1, “Stateflow Chart Concepts”.

When to Use Flow Graphs

Use flow graphs to represent flow logic in graphical functions or between states in a chart. A best practice is to encapsulate flow graphs in graphical functions to create modular, reusable decision and loop logic that you can call anywhere in a Stateflow chart. For more information about graphical functions, see “Using Graphical Functions to Extend Actions” on page 7-27.

Creating Flow Graphs with the Pattern Wizard

In this section...

Why Use the Pattern Wizard?

The Pattern Wizard is a utility in the Stateflow Editor that generates common flow graph patterns for use in graphical functions and charts. Although you can also create flow graphs by hand, the Pattern Wizard offers several advantages:

- Generates common logic and iterative loop patterns automatically
- Generates patterns that comply with guidelines from the MathWorks Automotive Advisory Board (MAAB)
- Promotes consistency in geometry and layout across patterns
- Facilitates storing and reusing patterns from a central location

How to Create Reusable Flow Graphs

When you create flow graphs with the Pattern Wizard, you can save them to a central location where you can retrieve them for reuse. To create reusable flow graphs that comply with MAAB guidelines:

1. Open a Stateflow chart.

   ### How do I create and open a new Stateflow chart?

   a. Type `sfnew` or `stateflow` at the MATLAB command prompt.

      A Simulink model opens, containing an empty Stateflow chart.

b Double-click the chart to open it in the Stateflow Editor.

2 In the Stateflow Editor, select a flow graph pattern:

To Create:	Select:	Reference
`if` decision patterns	**Patterns > Add Decision**	"Decision Logic Patterns in Flow Graphs" on page 5-10
`for`, `while`, and `do while` loop patterns	**Patterns > Add Loop**	"Iterative Loop Patterns in Flow Graphs" on page 5-14

The Stateflow Patterns dialog box appears.

3 Enter a description of your pattern (optional).

If you leave this field blank, the Pattern Wizard adds a default description in the Stateflow Editor.

4 Specify conditions and actions (optional).

You can also add or change conditions and actions directly in the Stateflow Editor.

5 Click **OK**.

The pattern appears in your chart. The geometry and layout comply with MAAB guidelines.

6 Customize the pattern as desired.

For example, you may want to add or change flow graphs, conditions, or actions. See "Try It: Creating and Reusing a Custom Pattern with the Pattern Wizard" on page 5-16.

7 Save the pattern to a central location as described in "Saving and Reusing Flow Graph Patterns" on page 5-7.

You can now retrieve your pattern directly from the Stateflow Editor to reuse in graphical functions and charts. See "How to Add Flow Graph Patterns in

Graphical Functions" on page 5-8 and "How to Add Flow Graph Patterns in Charts" on page 5-9.

Saving and Reusing Flow Graph Patterns

Using the Pattern Wizard, you can save flow graph patterns in a central location, then easily retrieve and reuse them in Stateflow graphical functions and charts. The Pattern Wizard lets you access all saved patterns from the Stateflow Editor.

Guidelines for Creating a Pattern Directory

The Pattern Wizard uses a single, flat directory for saving and retrieving flow graph patterns. Follow these guidelines when creating your pattern directory:

- Store all flow graphs at the top level of the pattern directory; do not create subdirectories.
- Make sure all flow graph files have a `.mdl` extension.

How to Save Flow Graph Patterns for Easy Retrieval

1. Create a directory for storing your patterns according to "Guidelines for Creating a Pattern Directory" on page 5-7.
2. In your Stateflow chart, select flow graphs with the patterns you want to save.
3. Select **Patterns > Save Pattern**.

   The Pattern Wizard displays a message that prompts you to choose a directory for storing custom patterns.

   The Pattern Wizard stores your flow graphs in the pattern directory as an `.mdl` file. The patterns that you save in this directory appear in a drop-down list when you select **Patterns > Add Custom**, as described in "How to Add Flow Graph Patterns in Graphical Functions" on page 5-8 and "How to Add Flow Graph Patterns in Charts" on page 5-9.
4. Click **OK** to dismiss the message.

The Browse For Folder dialog box appears.

5 Select the folder you designated as your pattern directory (or create a new folder) and click **OK**.

The Save Pattern As dialog box appears.

6 Enter a name for your pattern and click **Save**.

The Pattern Wizard saves your pattern as an `.mdl` file in the designated pattern directory.

How to Change Your Pattern Directory

1 Rename your existing pattern directory.

2 Add a pattern as described in "How to Add Flow Graph Patterns in Graphical Functions" on page 5-8 or "How to Add Flow Graph Patterns in Charts" on page 5-9.

The Pattern Wizard prompts you to choose a directory.

3 Follow the instructions in "How to Save Flow Graph Patterns for Easy Retrieval" on page 5-7.

How to Add Flow Graph Patterns in Graphical Functions

1 Add a graphical function to your Stateflow chart.

See "Creating a Graphical Function" on page 7-28.

2 Make the graphical function into a subchart by right-clicking in the function box and selecting **Make Contents > Subcharted**.

The function box turns gray.

3 Double-click the subcharted graphical function to open it in the Stateflow Editor.

4 In the menu bar, select **Patterns > Add Custom**.

The Select a Custom Pattern dialog box appears, displaying all of your saved patterns.

Why doesn't my dialog box display any patterns?

You haven't saved any patterns for the Pattern Wizard to retrieve. See "Saving and Reusing Flow Graph Patterns" on page 5-7.

5 Select a pattern from the list in the dialog box and click **OK**.

The pattern appears in the graphical function, which expands to fit the flow graph.

6 Define all necessary inputs, outputs, and local data in the graphical function and the chart that calls it.

How to Add Flow Graph Patterns in Charts

1 In your Stateflow chart, select **Patterns > Add Custom**.

The Select a Custom Pattern dialog box appears, displaying all of your saved patterns.

2 Select a pattern from the list in the dialog box and click **OK**.

The pattern appears in the chart.

3 Adjust the chart by hand to:

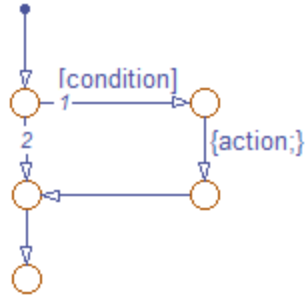
- Connect the flow graphs to the appropriate transitions.
- Ensure that there is only one default transition for exclusive (OR) states at each level of hierarchy.
- Define all necessary inputs, outputs, and local data.

MAAB-Compliant Patterns from the Pattern Wizard

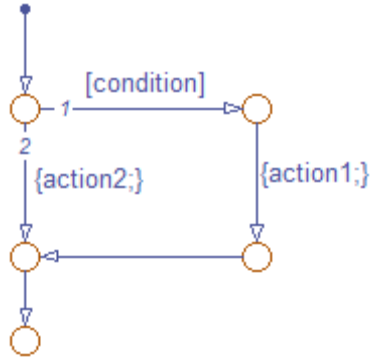
The Pattern Wizard generates MAAB-compliant decision and loop logic.

Decision Logic Patterns in Flow Graphs

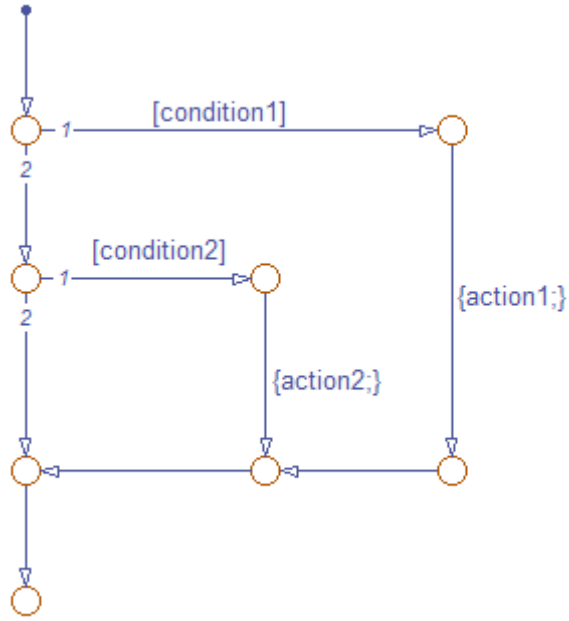
The Pattern Wizard generates the following MAAB-compliant decision logic patterns:



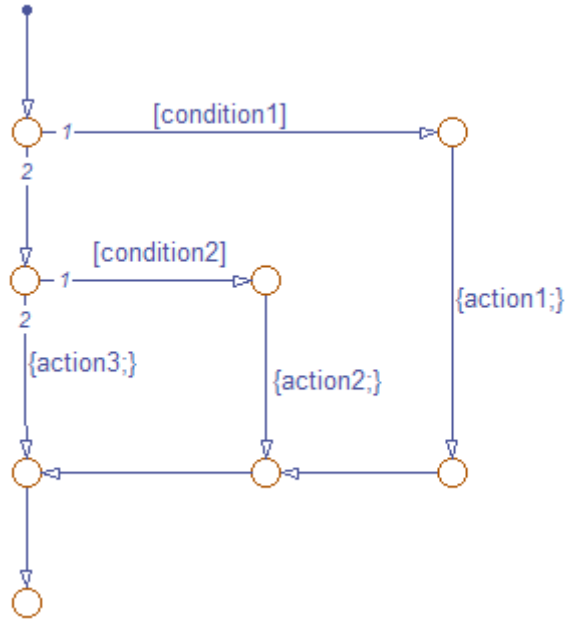
if



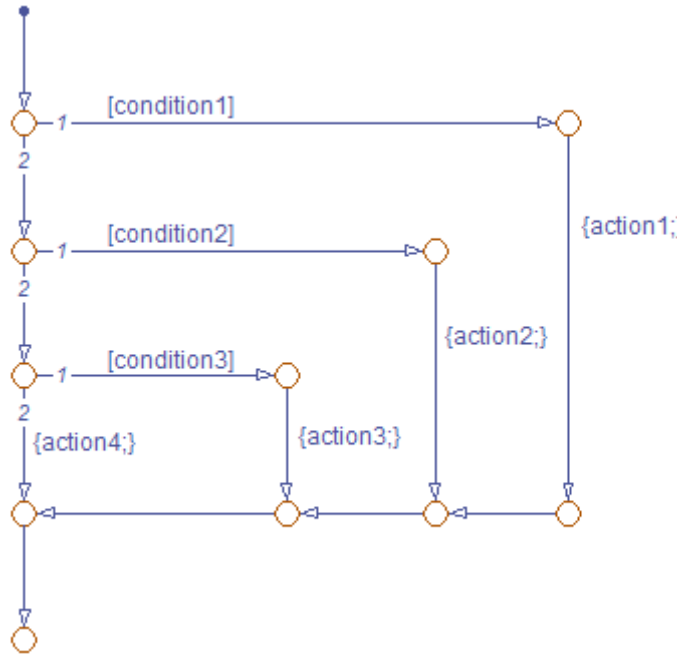
if-else



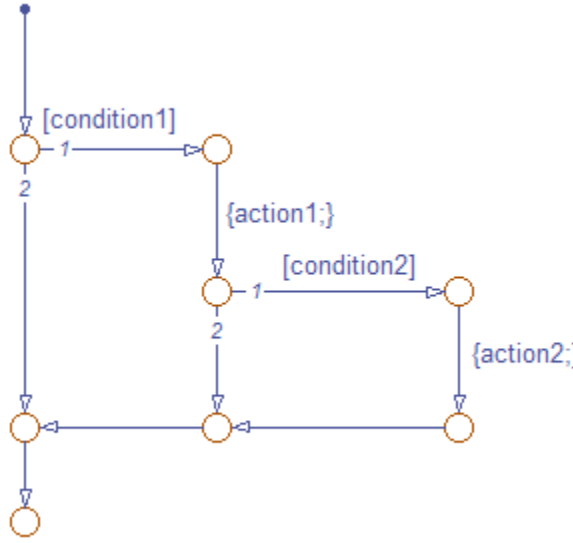
if-elseif



if-elseif-else



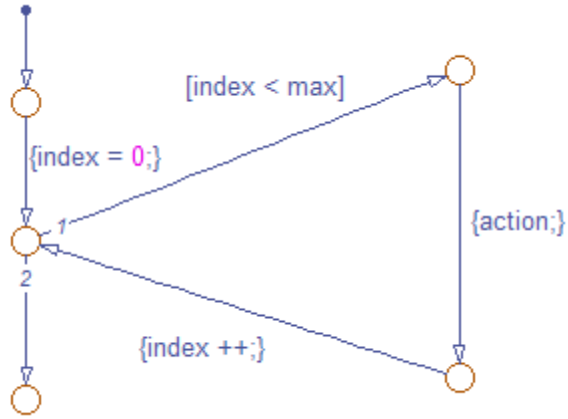
if-elseif-elseif-else



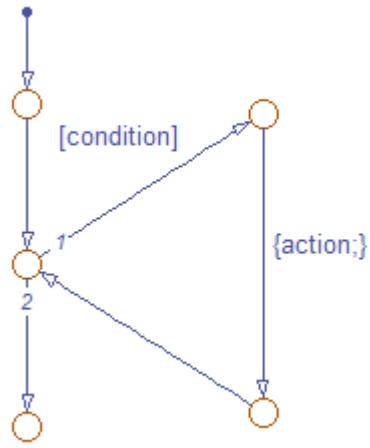
Nested if

Iterative Loop Patterns in Flow Graphs

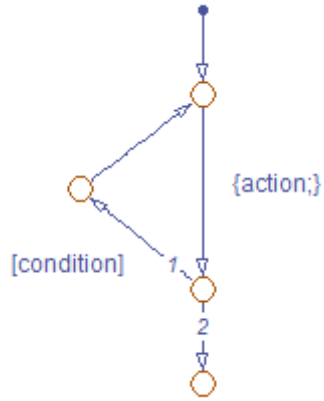
The Pattern Wizard generates the following MAAB-compliant iterative loop patterns:



for



while



do-while

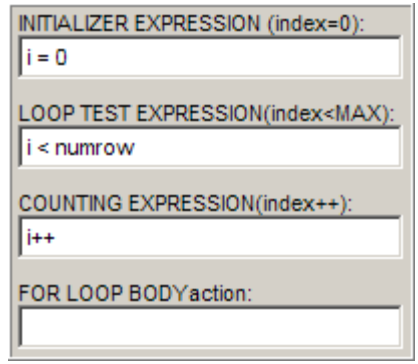
Try It: Creating and Reusing a Custom Pattern with the Pattern Wizard

This exercise shows how to create, modify, and save a custom flow graph pattern for iterating over the upper triangle of a two-dimensional matrix. In the upper triangle, the row index *i* is always less than or equal to column index *j*. This flow graph pattern uses nested `for`-loops to ensure that *i* never exceeds *j*.

Creating the Upper Triangle Iterator Pattern

1. Open a new (empty) chart in the Stateflow Editor.
2. Select **Patterns > Add Loop > For**

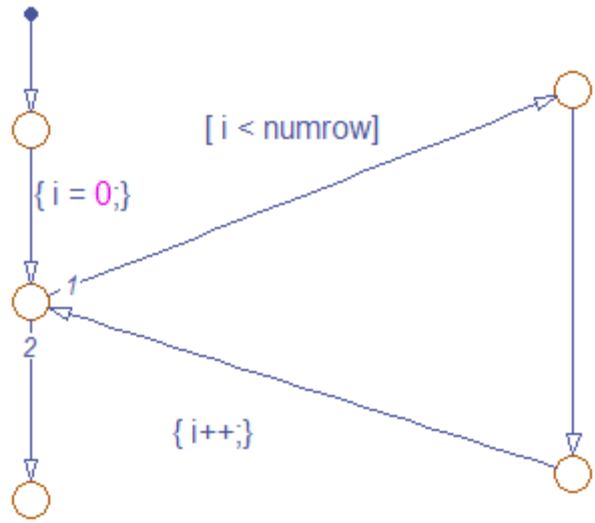
   The Stateflow Patterns dialog box opens on your desktop.
3. Enter the initializer, loop test, and counting expressions for iterating through the first dimension of the matrix, as follows:



Do not specify an action yet. You will add another loop for iterating the second dimension of the matrix.

4 Click **OK**.

The Pattern Wizard generates the first iterative loop in your chart:



This pattern from the Pattern Wizard:

- Conforms to all best practices for creating flow graphs, as described in "Best Practices for Creating Flow Graphs" on page 5-27.

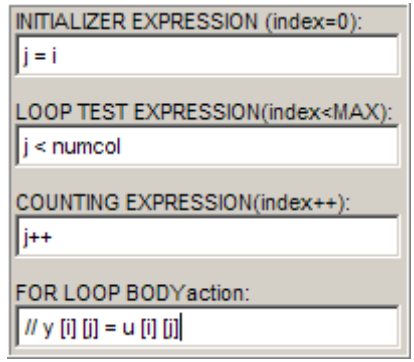
- Provides the correct syntax for conditions and condition actions.

5 Add the second loop by following these steps:

a Expand the editor window so the chart can accommodate a second pattern.

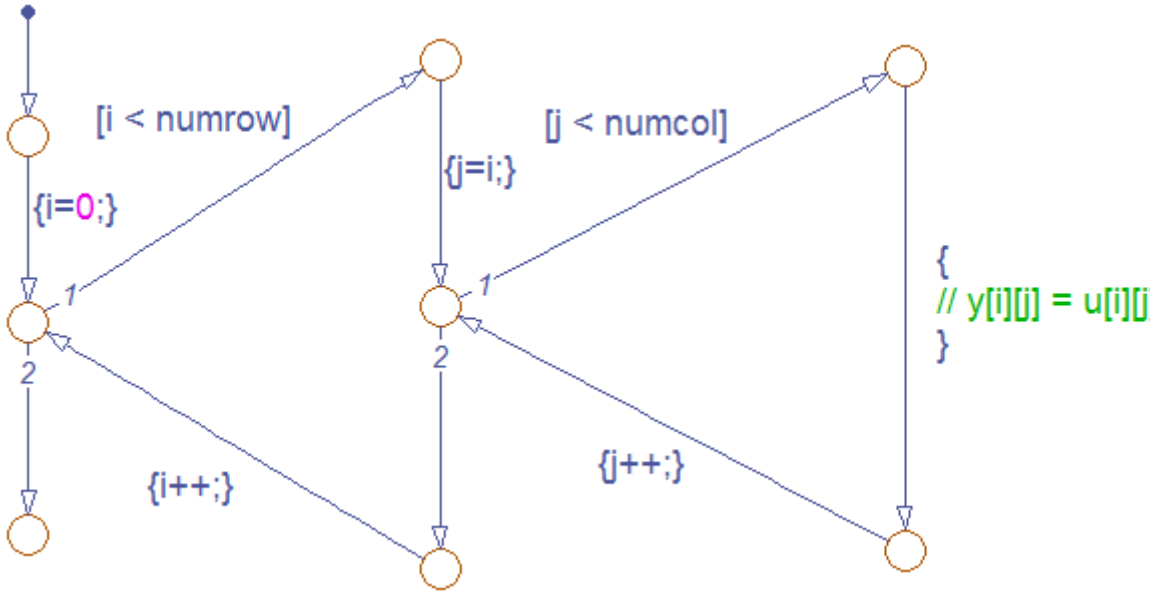
b Deselect all objects in the chart.

c Repeat steps 2, 3, and 4, this time specifying parameters for the second iterator *j*, and a placeholder for an action to retrieve each element in the upper triangle.



The Pattern Wizard generates the second loop pattern and leaves it selected so you can reposition it.

6 Nest the loop patterns as follows:



How can I nest the loop patterns?

Here is one way to nest the patterns:

a In the second pattern, delete the default transition.

b Delete the starting and terminating junctions.

c In the first pattern, delete the transition between `[i < numrow]` and `{i++;}`

d Move the second pattern into the first one by reconnecting transitions to junctions as necessary.

7 Inspect the flow graph to ensure that:

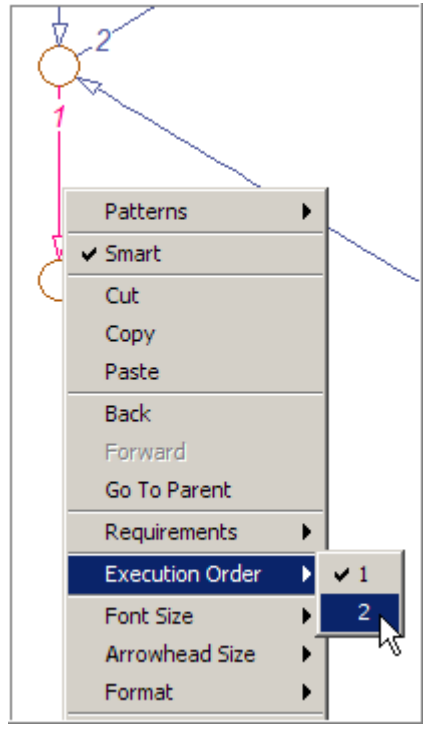
- There is only one default transition, attached to the first `for`-loop.
- You order the transitions as shown above.

What if the ordering is not correct?

To change the execution order of a transition:

1 Right-click to select the transition and select **Execution Order**.

2 Select the correct number:



The execution order of other transitions from the same junction adjust accordingly.

8 Save your chart.

Now you are ready to save your pattern to a central location for reuse (see "Saving the Upper Triangle Iterator Pattern for Reuse" on page 5-20.

Saving the Upper Triangle Iterator Pattern for Reuse

1 Create a directory for storing flow graph patterns, as described in "Guidelines for Creating a Pattern Directory" on page 5-7.

2 Open the Stateflow chart that contains the custom pattern.

3 In the chart, select the flow graph with the pattern that you want to save.

4 In the Stateflow Editor, select **Patterns > Save Pattern** and take one of these actions:

If you have...	Then Pattern Wizard...	Action
Not yet designated the pattern directory	Prompts you to create or select a pattern directory	Select the directory you just created. See "How to Save Flow Graph Patterns for Easy Retrieval" on page 5-7.
Already designated the pattern directory	Prompts you to save your pattern to the designated directory	Name your pattern and click **Save**.

The Pattern Wizard automatically saves your pattern as an `.mdl` file under the name you specify.

Adding the Upper Triangle Iterator Pattern to a Graphical Function

1 Open a new Stateflow chart.

2 Drag a graphical function into the chart from the object palette and enter the following function signature:

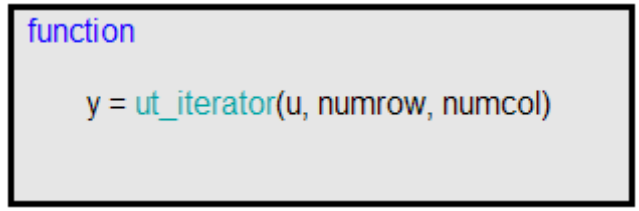
```
function y = ut_iterator(u, numrow, numcol)
```

The function takes three inputs:

Input	Description
u	2-D matrix
numrow	Number of rows in the matrix
numcol	Number of columns in the matrix

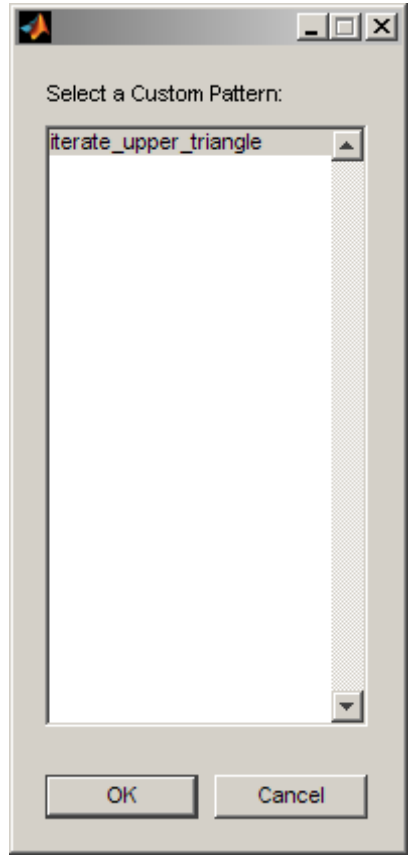
3 Right-click inside the function and select **Make Contents > Subcharted**.

The function should look like this:



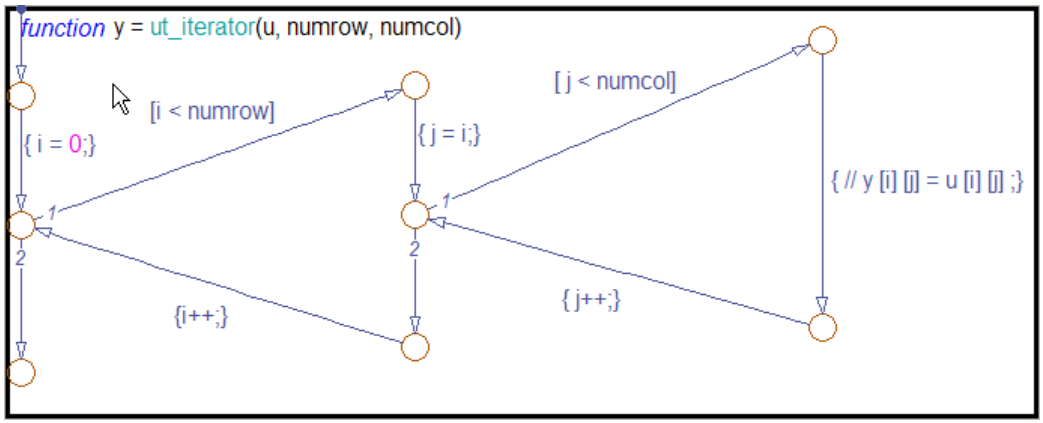
4 Double-click to open the subcharted function and select **Patterns > Add Custom**.

The Select a Custom Pattern dialog box appears, listing all the patterns you saved in your pattern directory.



5 Select your upper triangle iterator pattern and click **OK**.

The Pattern Wizard adds your custom pattern to the graphical function.



Before calling this function from a Stateflow chart, be sure to modify data names, types, and sizes as necessary and substitute an appropriate action.

Drawing and Customizing Flow Graphs By Hand

You can draw and customize flow graphs manually by using connective junctions as branch points between alternate transition paths.

How to Draw a Flow Graph

1. Open a Stateflow chart.
2. From the Stateflow Editor toolbar, drag one or more connective junctions into the chart using 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 graph should start.

How to Change Connective Junction Size

To change the size of connective junctions:

1. Select one or more connective junctions.
2. Right-click one of the selected junctions and select **Junction Size** from the drop-down menu.

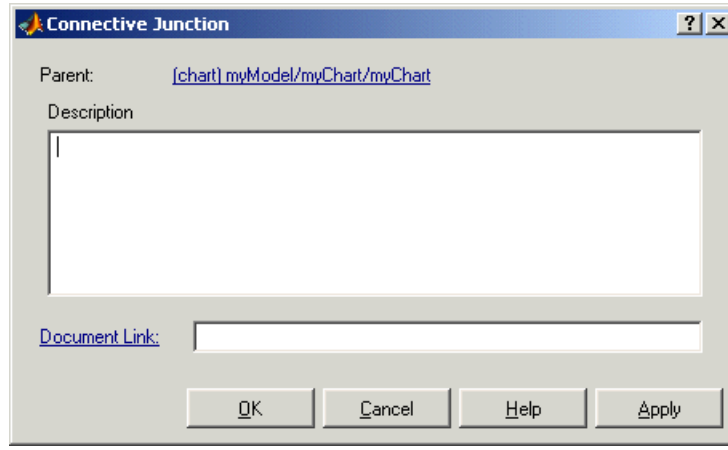
   A menu of junction sizes appears.
3. Select a junction size.

How to Modify Junction Properties

To modify the properties of a connective junction:

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

   The Connective Junction dialog box appears.



2 Edit the fields in the dialog as desired.

Field	Description
Parent	Parent of the connective junction (read-only). Click the hypertext link to bring the parent to the foreground.
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 Click **Apply** to save changes.

Best Practices for Creating Flow Graphs

Follow these best practices to create efficient, accurate flow graphs:

Use only one default transition

Flows graphs should have a single entry point.

Provide only one terminating junction

Multiple terminating junctions reduce readability of a flow graph.

Converge all transition paths to the terminating junction

This guideline ensures that execution of a flow graph always reach the termination point.

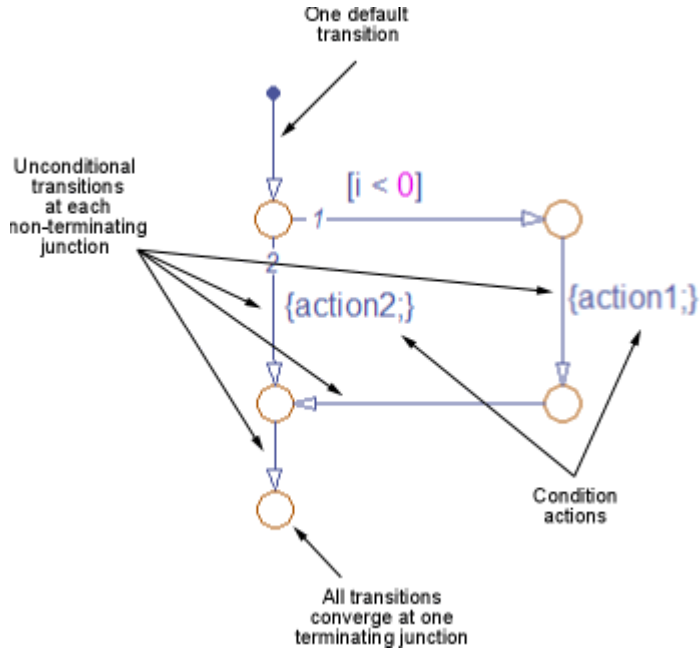
Provide an unconditional transition from every junction except the terminating junction

This guideline ensures that execution never backtracks in a flow graph.

Use condition actions to process updates, not transition actions

Flow graphs test transitions, but do not execute them (and, therefore, never execute transition actions).

The following example illustrates these best practices:


One default transition
Unconditional transitions at each non-terminating junction
[i < 0]
1
2
{action2;}
{action1;}
Condition actions
All transitions converge at one terminating junction

6

Building Mealy and Moore Charts

Overview of Mealy and Moore Machines

In this section...

Semantics of Mealy and Moore Machines

Mealy and Moore are often considered the basic, industry-standard paradigms for modeling finite-state machines. Generally in state machine models, the next state is a function of the current state and its inputs, as follows:

$$X(n+1) = f(X(n),u)$$

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

In this context, Mealy and Moore machines each have well-defined semantics.

Type of Machine	Semantics	Applications
Mealy	Output is a function of inputs *and* state: $y = g(X,u)$	Clocked synchronous machines where state transitions occur on clock edges
Moore	Output is a function *only* of state: $y = g(X)$	Clocked synchronous machines where outputs are modified at clock edges

You can create charts that implement pure Mealy or Moore semantics as a subset of Stateflow chart semantics (see "Creating Mealy and Moore Charts" on page 6-6). Mealy and Moore charts can be used in simulation and code generation of C and hardware description language (HDL).

Note To generate HDL code from Stateflow charts, you must use Simulink® HDL Coder™ software, which is available separately.

Running a Demo of Mealy and Moore Machines

Stateflow software ships with a demonstration that shows how to use Mealy and Moore machines for sequence recognition in signal processing. You can run the demo by following these steps:

1. At the MATLAB prompt, type this command:

   ```
   demos
   ```

   The Help browser appears, listing categories of demos in the left pane.

2. In the left pane, navigate to **Simulink > Stateflow > General Applications > Sequence Recognition Using Mealy and Moore Charts**.

3. Follow the instructions in the right pane of the Help browser.

The Default State Machine Type

When you create a Stateflow chart, the default type is a hybrid state machine model that combines the semantics of Mealy and Moore charts with the extended Stateflow chart semantics (see Chapter 3, "Stateflow Chart Semantics"). This default chart type is called *Classic*.

What Is State?

State is a combination of local data and chart activity. Therefore, computing state means updating local data and making transitions from a currently active state to a new state. State persists from one time step to another. In a Classic Stateflow chart, output behaves like state because output values persist between time steps. However, unlike state, output is available outside the chart through output ports. By contrast, output in Mealy and Moore charts does not persist and instead must be computed in each time step.

Availability of Output

Stateflow chart semantics guarantee that the output of Mealy and Moore machines is well defined at every time step by enforcing the option **Initialize Outputs Every Time Chart Wakes Up** for these chart types. This option initializes outputs to a default value whenever the chart is triggered (see "Setting Properties for Individual Charts" on page 16-5). Normally, charts compute output data in every execution. In this case, computed outputs override the default values. However, when output is not computed, the default value applies.

Mealy machines compute output on transitions, while Moore machines compute outputs in states. Therefore, Mealy charts can compute output earlier than Moore charts — that is, at the time the chart's default path executes. If you enable the chart property **Execute (enter) Chart At Initialization**, this computation occurs at t = 0 (first time step); otherwise, it occurs at t = 1 (next time step). By contrast, Moore machines can compute outputs only *after* the default path executes. Until then, outputs take the default values.

The following table summarizes the earliest time at which output can be computed in Mealy and Moore charts:

Execute (enter) Chart at Initialization	Mealy Computes Outputs at:	Moore computes Outputs at:
Enabled	t = 0	t = 1
Disabled	t = 1	t = 2

Advantages of Mealy and Moore Charts Over Classic Stateflow Charts

Mealy and Moore charts offer the following advantages over Classic Stateflow charts:

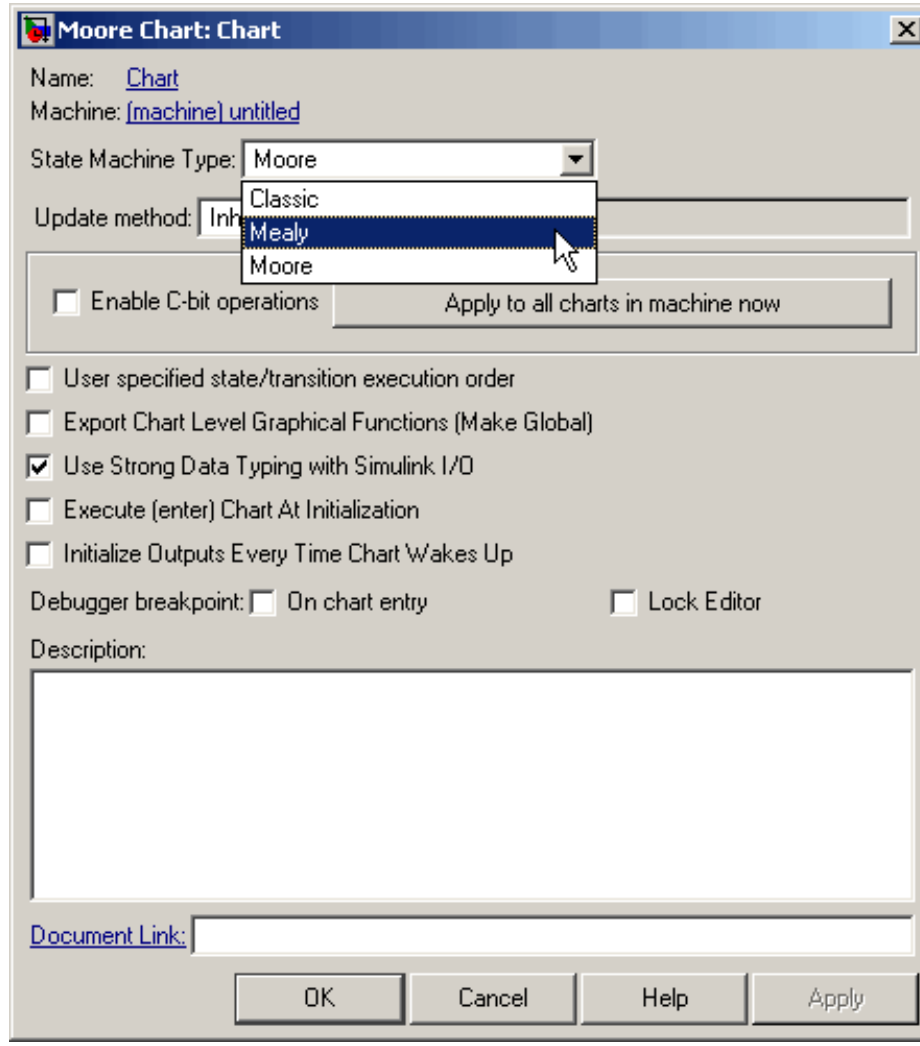
- You can verify the Mealy and Moore charts you create to ensure that they conform to their formal definitions and semantic rules. Error messages appear at compile time (not at design time).
- Moore charts provide a more efficient implementation than Classic charts, both for C and HDL targets.

Creating Mealy and Moore Charts

To create a new Mealy or Moore chart, follow these steps:

1. Add a new Stateflow block to a Simulink model; then double-click the block to open the Stateflow Editor.
2. Right-click in the Stateflow Editor and select **Properties**.

   The Chart Properties dialog box opens on your desktop.
3. From the State Machine Type drop-down menu, select **Mealy** or **Moore**.



4 Click **OK**.

The chart icon updates to display the selected chart type:

Mealy **Moore**

The title bar of the Stateflow Editor also displays the selected chart type.

5 Design your chart according to the guidelines for the chart type (see "Design Considerations for Mealy Charts" on page 6-9 and "Design Considerations for Moore Charts" on page 6-15.

Design Considerations for Mealy Charts

In this section...

Mealy Semantics

To ensure that output is a function of input *and* state, Mealy state machines enforce the following semantics:

- Outputs never depend on previous outputs.
- Outputs never depend on the next state.
- Chart wakes up periodically based on a system clock.

Note A Stateflow chart provides one time base for input and clock (see "Calculate Output and State Using One Time Base" on page 6-12).

- Chart must compute outputs whenever there is a change on the input port.
- Chart must compute outputs only in transitions, not in states.

Design Rules for Mealy Charts

To conform to the Mealy definition of a state machine, you must ensure that a Mealy chart computes outputs every time there is a change on the input port. As a result, you must follow a set of design rules for Mealy charts.

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 you from computing output without considering changes on the input port
- Ensuring that output depends on current state and not next state

Restrict Use of Data

You can define inputs, outputs, local data, parameters, and constants in Mealy charts, but other data restrictions apply:

Restrict Machine-Parented Data to Constants and Parameters.

Machine-parented data is data that you define for a Stateflow machine, which is the collection of all Stateflow blocks in a Simulink model. 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, you can define only constants and parameters at the machine level.

Note Chart parameters have constant value during simulation and code generation.

Do Not Define Data Store Memory. You cannot define data store memory (DSM) in Mealy charts because DSM objects can be modified by objects external to the chart. A Stateflow chart uses data store memory to share data with a Simulink model. Data store memory acts as global data that can be modified by other blocks and models in the Simulink hierarchy that contains the chart. Mealy charts should not access data that can change unpredictably.

Restrict Use of Events

You must limit the use of events in Mealy charts as follows:

Do:	Do Not:
Use input events to trigger the chart	Broadcast any type of event
Use event-based temporal logic to guard transitions You can use event-based temporal logic in Mealy charts because it behaves synchronously (see "Operators for Event-Based Temporal Logic" on page 10-57). Think of the change in value of a temporal logic condition as an event that the chart schedules internally. Therefore, at each time step, the chart retains its notion of state because it knows how many ticks remain before the temporal event executes. **Note** In Mealy charts, the base event for temporal logic operators must be a predefined event such as `tick` or `wakeup` (see "Referencing Implicit Events" on page 9-31).	Use local or machine-parented events to guard transitions You cannot use local or machine-parented events in Mealy charts because they are not deterministic. These events can occur while the chart computes outputs and, therefore, violate Mealy semantics that require charts to compute outputs whenever input changes.

Initialize Outputs Every Time Chart Wakes Up

To prevent latching of outputs, a Mealy chart automatically applies the initial value of its outputs every time it wakes up. This is a requirement for Mealy charts to ensure that outputs do not depend on previous values of outputs.

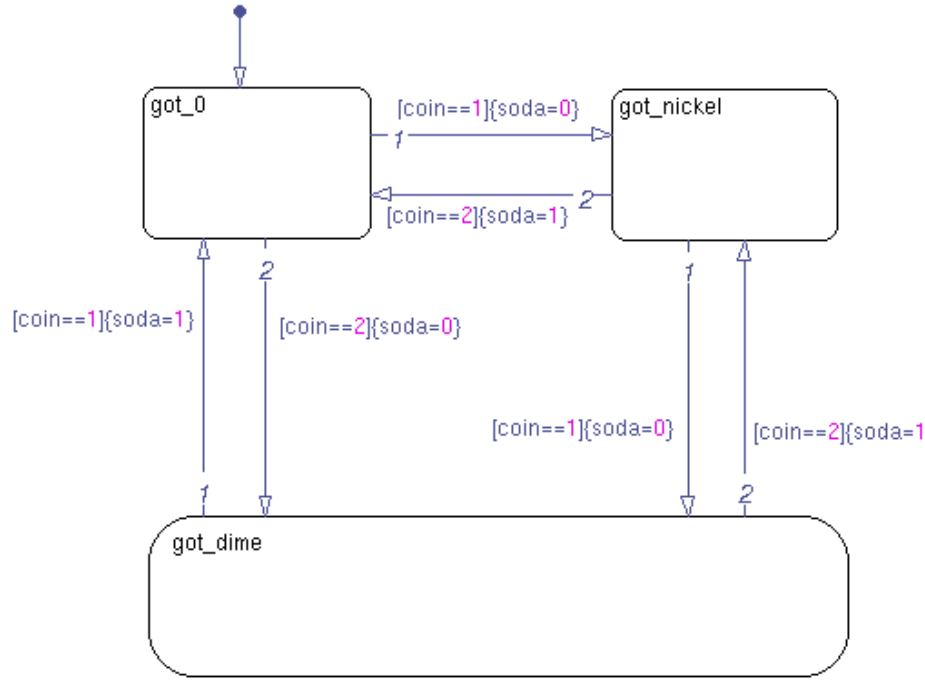
When you create a Mealy chart, it enforces the chart property **Initialize Outputs Every Time Chart Wakes Up**. For more information about this property, see "Setting Properties for Individual Charts" on page 16-5.

Calculate Output and State Using One Time Base

You can use one time base for clock and input, as determined by the Simulink solver (see "Solvers"). 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.

Example: Mealy Vending Machine

The following chart uses Mealy semantics to model a vending machine.



Opening the Model

To open the model of a Mealy vending machine, type `sf_mealy_vending_machine` at the MATLAB command prompt.

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, as follows:

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 is 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 the following 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.

Design Considerations for Moore Charts

In this section...

Moore Semantics

In Moore charts, output is a function of current state only. At every time step, a Moore chart wakes up, computes its outputs, and then evaluates its inputs to reconfigure itself for the next time step. For example, after evaluating its inputs, the Moore chart may take transitions to a new configuration of active states, also called *next state*. However, the Moore chart must always compute its outputs before changing state.

To ensure that output is a function *only* of state, Moore state machines enforce the following semantics:

- Outputs depend only on the current state, not the next state.
- Outputs never depend on previous outputs.
- Chart must compute outputs only in states, not in transitions.
- Chart must compute outputs before updating state.

Design Rules for Moore Charts

To conform to the Moore definition of a state machine, you must ensure that every time a Moore chart wakes up, it computes outputs from the current set of active states without regard to input. As a result, you must follow a set of design rules for Moore charts.

- "Compute Outputs in State Actions, Not on Transitions" on page 6-16
- "Restrict Data to Inputs, Outputs, and Constants" on page 6-18
- "Reference Input Only in Conditions" on page 6-19
- "Do Not Use Actions on Transitions" on page 6-20

- "Do Not Use Graphical Functions" on page 6-20
- "Do Not Use Truth Tables, Embedded MATLAB Functions, or Simulink Functions" on page 6-20
- "Restrict Use of Events" on page 6-21
- "Initialize Outputs Every Time Chart Wakes Up" on page 6-21

Compute Outputs in State Actions, Not on Transitions

To ensure that outputs depend solely on current state, you must compute outputs in state actions, subject to the following restrictions:

- "Combine During and Exit Actions" on page 6-16
- "Allow Actions in Leaf States Only" on page 6-17
- "Do Not Label State Actions" on page 6-18

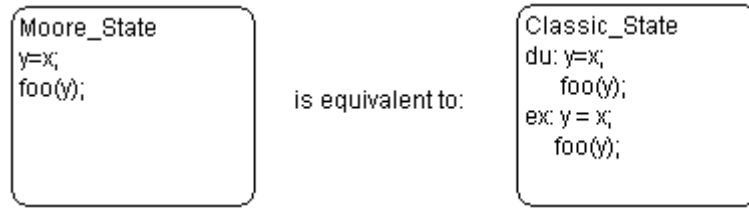
You cannot define actions on transitions because transitions almost always depend on inputs. For example, if you compute outputs in a condition action on a transition, the chart updates outputs whenever there is a change on the input — a violation of Moore semantics.

Combine During and Exit Actions. For Classic charts, you can define different types of actions in states (see "State Action Types" on page 10-2). Each action can consist of multiple command statements. In Moore charts, you can include *only one action per state*, but the chart executes the action as both a `during` *and* an `exit` action. This duality ensures that the chart never exits a state before computing its outputs because:

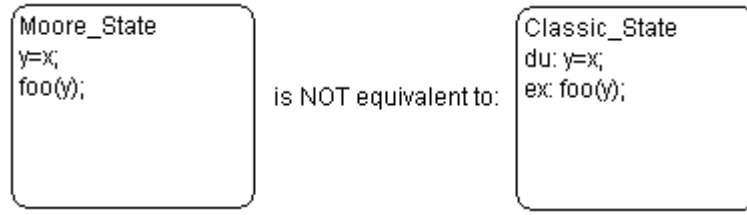
- The chart executes the action while the state is active and there are no valid transitions to take (like a `during` action)
- The chart also executes the action just before exiting the state to take a valid transition (like an `exit` action)

In other words, all active states in Moore charts compute their outputs in a consistent way whether an outer transition is valid or not.

To implement the duality of execution, the `during` and `exit` actions must be identical, as in this example.



Moore states do *not* differentiate between `during` and `exit` actions, as shown here.



Note There are no labels on state actions in Moore charts (see "Do Not Label State Actions" on page 6-18).

Allow Actions in Leaf States Only. In Moore charts, you can add actions only to leaf states. A leaf state is a state that resides at the lowest level of the Stateflow hierarchy and, therefore, does not parent any other states. This restriction ensures that when you compute outputs in state actions, the following is true:

- Outputs are not defined at multiple levels in the hierarchy with different values.
- The same top-down semantics apply for executing Moore charts as for Classic charts. In this way, charts compute outputs *as if* they evaluate actions before inner and outer flow graphs. This behavior guarantees that the outputs will be identical for both chart types.

You can compute outputs in leaf states that have exclusive (OR) or parallel (AND) decomposition. However, you should not compute the same outputs in

sibling parallel (AND) states because the values computed by the last state executed will prevail, overwriting the previously computed values.

For descriptions of chart execution semantics, see "Executing a Chart" on page 3-5 and Semantic Rules Summary.

Do Not Label State Actions. Do not label state actions in Moore charts with any keywords — such as `du`, `during`, `ex`, or `exit`. State actions behave in Moore charts as `during` and `exit` actions automatically, as explained in "Combine During and Exit Actions" on page 6-16. Moore charts never execute `entry` actions because these actions always execute as the result of a transition and, therefore, depend on inputs.

Restrict Data to Inputs, Outputs, and Constants

You can define inputs, outputs, parameters, and constants in Moore charts, but other data restrictions apply:

Do Not Define Local Data. You cannot define local data in Moore charts. In Classic charts, you can use local data to transfer inputs to outputs, as in this example:

```
local_D = input_U;
output_Y = local_D;
```

However, in Moore charts, you compute outputs from current state only, but never from local data. When a chart contains local data, it cannot easily verify that outputs do not depend on inputs.

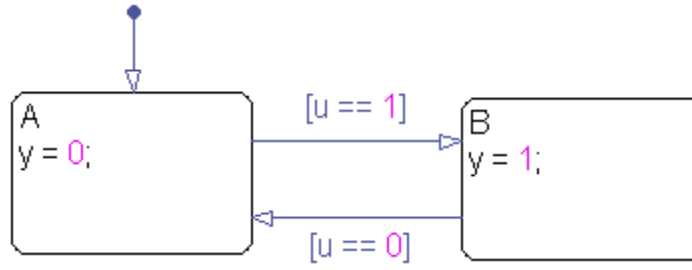
Restrict Machine-Parented Data to Constants and Parameters. Machine-parented data is data that you define for a Stateflow machine, which is the collection of Stateflow blocks in a Simulink model. The Stateflow machine is the highest level of the Stateflow hierarchy. When you define data at this level, 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, you can define only constants and parameters at the machine level.

Note Chart parameters have constant value during simulation and code generation.

Do Not Define Data Store Memory. You cannot define data store memory (DSM) in Moore charts because DSM objects can be modified by objects external to the chart. A Stateflow chart uses data store memory to share data with a Simulink model. Data store memory acts as global data that can be modified by other blocks and models in the Simulink hierarchy that contains the chart. Moore charts should not access data that can change unpredictably.

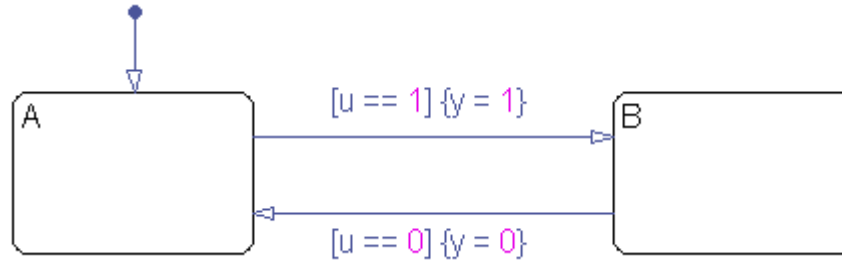
Reference Input Only in Conditions

In Classic Stateflow charts, you can test inputs in conditions on transitions, and then modify outputs in associated condition actions and transition actions. However, in Moore charts, outputs can never depend on inputs. Therefore, you can set up conditions on transitions that reference inputs, but you cannot add actions to transitions that modify outputs based on those conditions. For example, you can use these transitions in a Moore chart.



In this example, each transition tests input u in a condition, but modifies output y in a state action.

By contrast, these transitions are *illegal* in a Moore chart.



Here, each transition tests input u in a condition, but modifies output y in a condition action, based on the value of the input. This construct violates Moore semantics and generates a compiler error. Similarly, you cannot use transition actions in Moore charts.

Do Not Use Actions on Transitions

You cannot define condition actions or transition actions in Moore charts (see "Reference Input Only in Conditions" on page 6-19).

Do Not Use Graphical Functions

You cannot use graphical functions in Moore charts. This restriction prevents scenarios that violate Moore semantics, such as:

- Adding conditions that call functions which compute outputs as a side effect
- Adding state actions that call functions which reference inputs

Do Not Use Truth Tables, Embedded MATLAB Functions, or Simulink Functions

You cannot use truth tables, Embedded MATLAB functions, or Simulink functions in Moore charts. These restrictions prevent violations of Moore semantics during chart execution.

Restrict Use of Events

You must limit the use of events in Moore charts as follows:

Do:	Do Not:
Use input events to trigger the chart	Broadcast any type of event
Use event-based temporal logic to guard transitions You can use event-based temporal logic in Moore charts because it behaves synchronously (see "Operators for Event-Based Temporal Logic" on page 10-57). Think of the change in value of a temporal logic condition as an event that the chart schedules internally. Therefore, at each time step, the chart retains its notion of state because it knows how many ticks remain before the temporal event executes. **Note** In Moore charts, the base event for temporal logic operators must be a predefined event such as `tick` or `wakeup` (see "Referencing Implicit Events" on page 9-31).	Use local or machine-parented events to guard transitions You cannot use local or machine-parented events in Moore charts because they are not deterministic. These events can occur while the chart computes outputs and, therefore, violate Moore semantics that require charts to compute outputs whenever input changes.

Initialize Outputs Every Time Chart Wakes Up

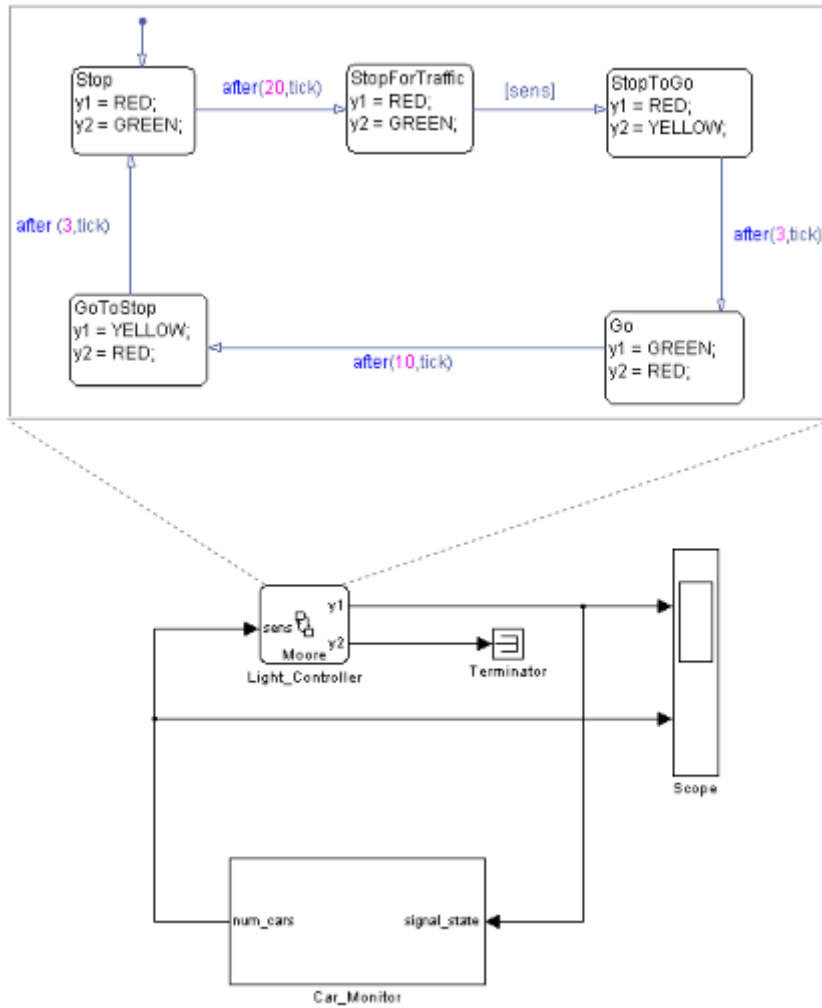
To prevent latching of outputs, a Moore chart automatically applies the initial value of its outputs every time it wakes up. This requirement for Moore charts ensures that outputs do not depend on previous values of outputs.

When you create a Moore chart, it automatically enables the chart property **Initialize Outputs Every Time Chart Wakes Up**, which you cannot

disable. For more information about this property, see “Setting Properties for Individual Charts” on page 16-5.

Example: Moore Traffic Light

The following chart uses Moore semantics to model a traffic light.



Opening the Model

To open the model of a Moore traffic light, type `sf_moore_traffic_light` at the MATLAB command prompt.

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, as follows:

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 the following Moore design rules:

- The chart computes outputs in state actions.
- Actions appear in leaf states only.
- Leaf states contain no more than one action.
- 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`).

Changing Chart Type

The 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 type usually requires you to redesign your chart to achieve *equivalent behavior* — that is, where both charts produce the same sequence of outputs given the identical sequence of inputs. To assist you, diagnostic messages appear at compile time (see "Debugging Mealy and Moore Charts" on page 6-27). In some cases, however, there may be no way to translate specific behaviors without violating chart definitions.

Here is a summary of what happens when you change chart types mid-design.

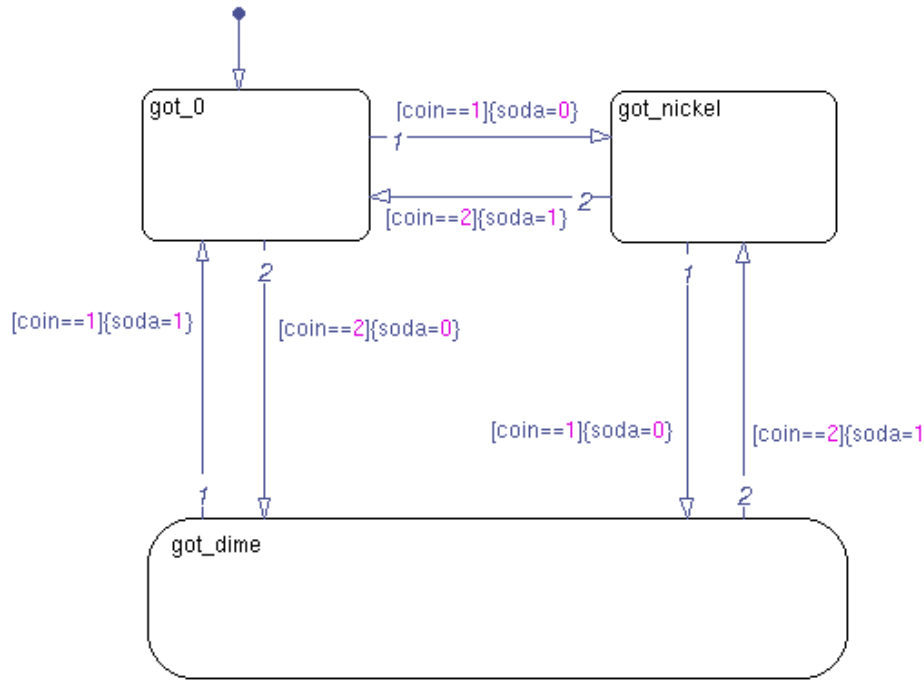
From	To	Result
Mealy	Classic	Mealy charts retain their semantics when changed to Classic type.
Classic	Mealy	If the Classic chart confirms to Mealy semantic rules, the Mealy chart exhibits equivalent behavior, provided that output is defined at every time step.
Moore	Classic	State actions in the Moore chart behave as `entry` actions because they are not labeled. Therefore, the Classic chart will not exhibit behavior that is equivalent to the original Moore chart. Requires redesign.
Classic	Moore	Actions that are unlabeled in the Classic chart (`entry` actions by default) behave as `during` and `exit` actions. Therefore, the Moore chart will not exhibit behavior that is equivalent to the original Classic chart. Requires redesign.
Mealy	Moore	Converting between these two types does not produce equivalent behavior because Mealy and Moore rules about placement of actions are mutually exclusive. Requires redesign.
Moore	Mealy	

Debugging Mealy and Moore Charts

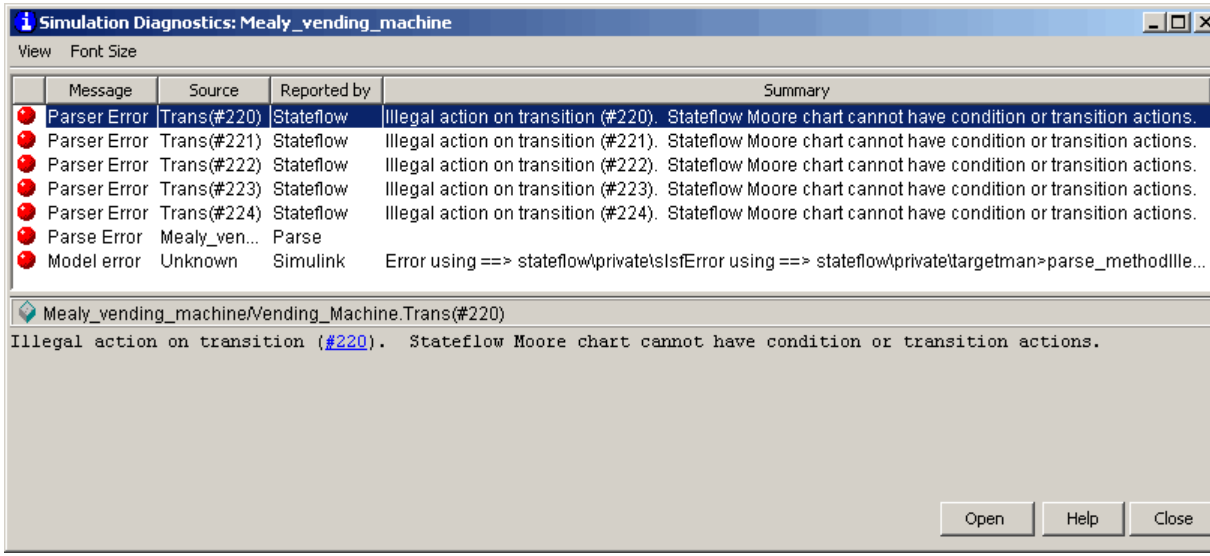
At compile time, informative diagnostic messages appear to help you:

- Design Mealy and Moore charts from scratch
- Redesign legacy Classic charts to conform to Mealy and Moore semantics
- Redesign charts to convert between Mealy and Moore types

For example, recall the Mealy vending machine chart described in "Example: Mealy Vending Machine" on page 6-12.



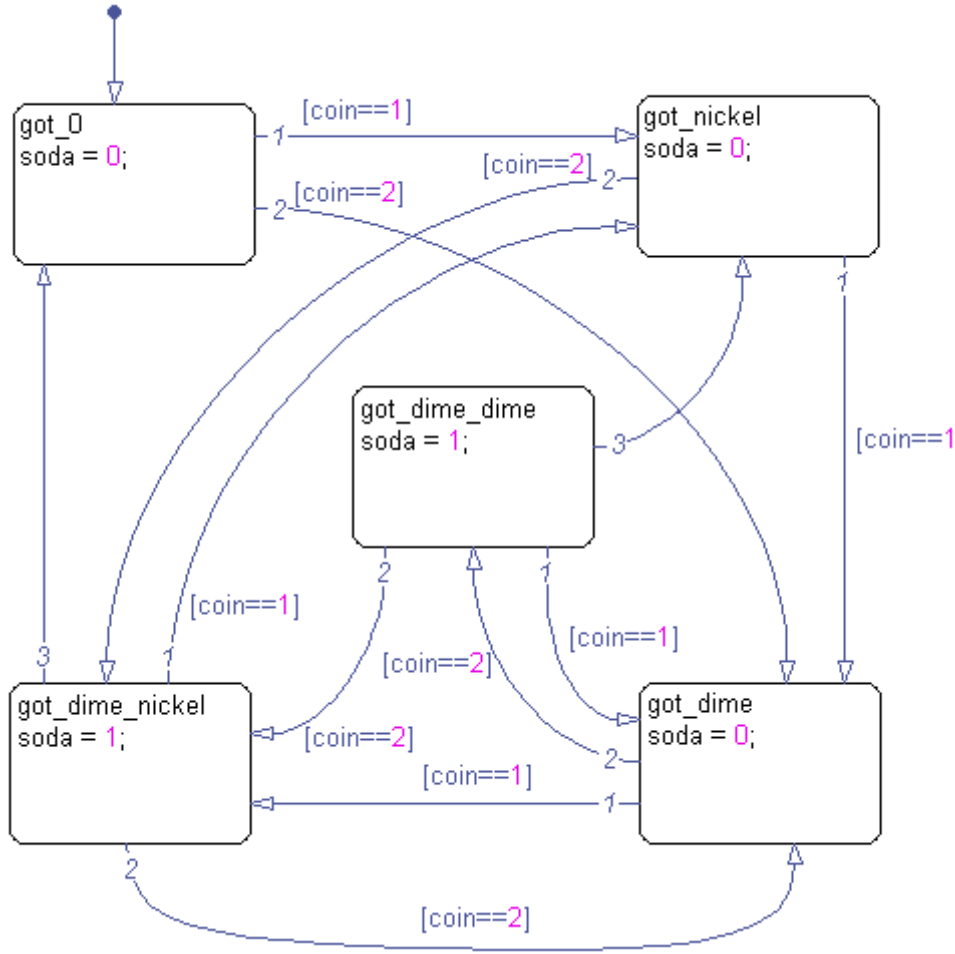
If you change the chart type to **Moore** and rebuild, you get the following diagnostic messages:



These diagnostics alert you to the fact that you cannot define actions on transitions. Without actions, you cannot compute outputs on transitions in Moore charts (see "Do Not Use Actions on Transitions" on page 6-20). According to Moore semantics, you must instead compute outputs in state actions (see "Design Rules for Moore Charts" on page 6-15).

In the Mealy chart, each condition action computes output (*whether or not soda is released*) based on input (*the coin received*). Each state represents one of the three possible coin inputs: nickel, dime, or no coin. The Mealy chart computes the output as it transitions to the next state. When you move this logic out of transitions and into state actions in the Moore chart, you need more states. The reason is that in the Moore chart, each state must represent not only coins received, but also the soda release condition. The Moore chart must compute output according to the active state *before* considering input. As a result, there will be a delay in releasing soda, even if the machine receives enough money to cover the cost.

The equivalent vending machine, designed as a Moore chart, is as follows.



This table compares the semantics of the two charts:

Mealy Vending Machine	Moore Vending Machine
Uses 3 states	Uses 5 states

Mealy Vending Machine	Moore Vending Machine
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

Note For this vending machine, Mealy is a better modeling paradigm because there is no delay in releasing soda once sufficient coins are received. By contrast, the Moore vending machine requires an extra time step to pass before producing soda. Since the Moore vending machine accepts a nickel, a dime, or no coin in a given time step, it is possible that the soda will be produced in a time step in which a coin is accepted toward the next purchase. In this situation, the delivery of a soda may appear to be in response to this coin, but actually occurs because the vending machine received the purchase price in previous time steps.

7

Extending Stateflow Charts

Using History Junctions to Extend Charts and States

In this section...

About History Junctions

History junctions extend the ability of charts and states by recording 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.

Creating a History Junction

To create a junction, do the following:

1. In the Stateflow Editor toolbar, click the **History Junction** icon  .
2. Move your pointer into the Stateflow Editor.

   The pointer takes on the shape of a junction.
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 Stateflow Editor toolbar, double-click the **History Junction** icon.
2. The button is now in multiple object mode.
3. Click anywhere in the drawing area to place a history junction.
4. Move to and click another location to create an additional history junction.

5 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.

Changing History Junction Size

To change the size of junctions, do the following:

1 Select the history junctions whose size you want to change.

2 Place your pointer over one of the junctions and right-click.

3 In the resulting submenu, place your pointer over **Junction Size**.

A menu of junction sizes appears.

4 Select a size from the menu of junction sizes.

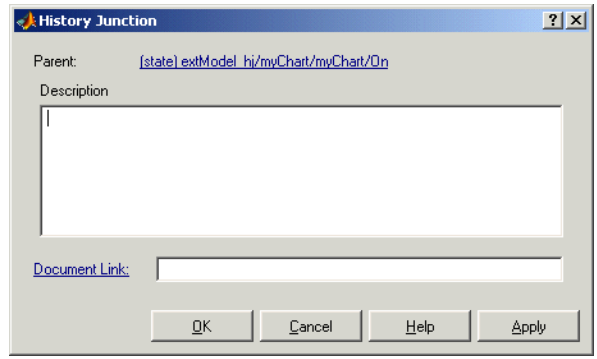
Changing History Junction Properties

To edit the properties for a junction, do the following:

1 Right-click a junction.

2 In the resulting submenu, select **Properties**.

The History Junction dialog box appears as shown.



3 Edit the fields in the properties dialog box, which are described in the following table:

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.
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`.

4 When finished editing, select one of the following:

- Select the **Apply** button to save the changes.
- Select the **Cancel** button to cancel any changes you've made.
- Select **OK** to save the changes and close the dialog box.
- Select the **Help** button to display the Stateflow online help in an HTML browser window.

Using Subcharts to Extend Charts

In this section...

What Is a Subchart?

You can create charts within charts. A chart that is embedded in another chart is called a *subchart*. The subchart can contain anything a top-level chart can, including other subcharts. In fact, you can nest subcharts to any level.

A subcharted state is a superstate of the states and charts that it contains. It 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. Further, 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 "Using Supertransitions to Extend Transitions" on page 7-10 for more information.

Subcharts enable you to reduce a complex chart to a set of simpler, hierarchically organized charts. This makes the chart easier to understand and maintain. Nor do you have to worry about changing the semantics of the chart in any way. Subchart boundaries are ignored during simulation and code generation.

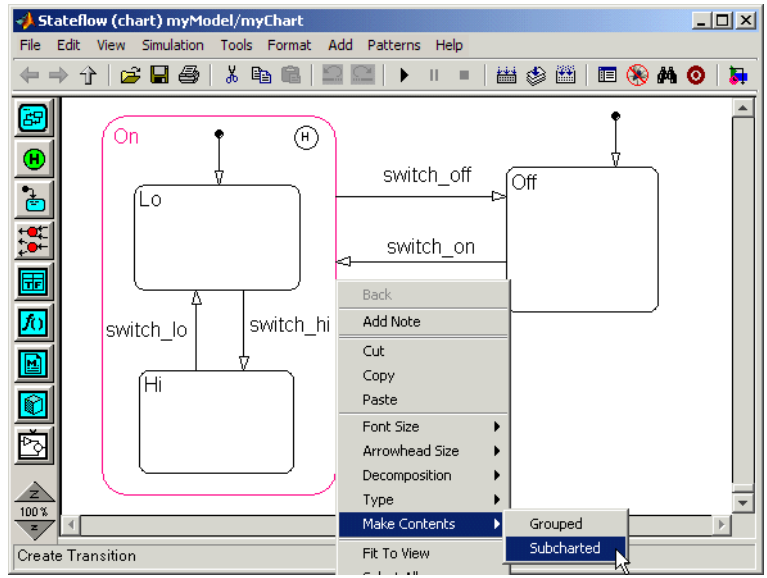
Subcharts define a containment hierarchy within a top-level chart. A subchart or top-level chart is the *parent* of the charts it contains at the first level and an *ancestor* of all the subcharts contained by its children and their descendants at lower levels.

Creating a Subchart

You create a subchart by converting an existing state, box, or graphical function into the subchart. The object to be converted can be one that you have created expressly for the purpose of making a subchart or it can be 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:

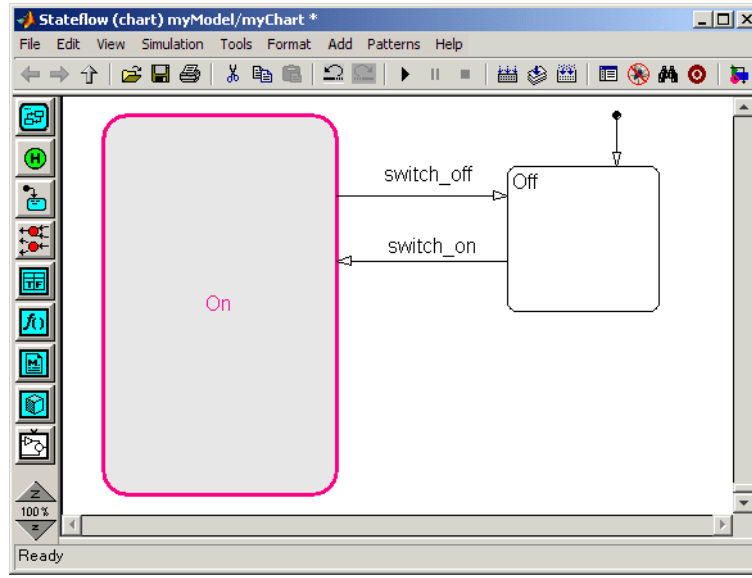
1 Select the object and right-click a state to display the shortcut menu for that state.



2 Select **Make Contents** from the resulting menu.

3 Select **Subcharted** from the resulting submenu.

This action converts the state (or a graphical function or box) to a subchart.



Note 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 "Using Boxes to Extend Charts" on page 7-47 for more information).

To convert the subchart back to its original form, right-click the subchart. In the context menu, select **Make Contents > Subcharted**.

Caution 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.

Manipulating 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

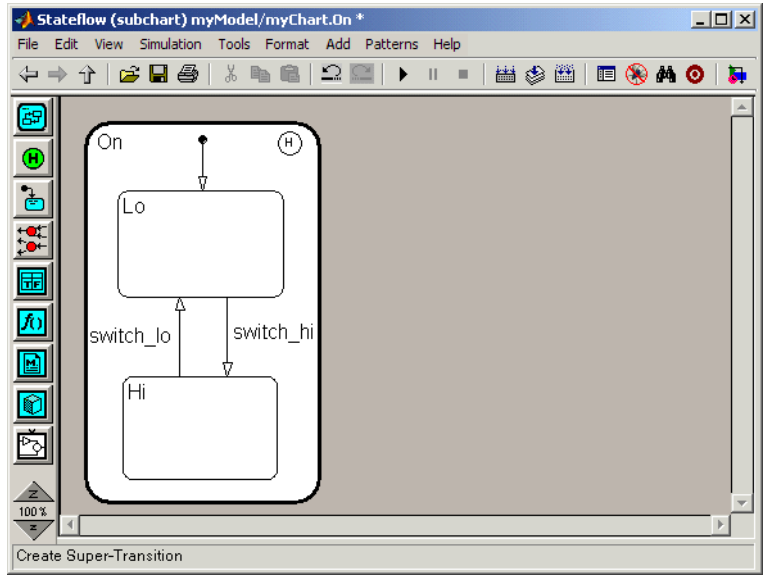
same or different levels in the chart hierarchy (see “Using Supertransitions to Extend Transitions” on page 7-10).

Opening 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.

The contents of the subchart appear, as shown.



A shaded border surrounds the contents of the subchart. The border displays supertransitions.

To return to the previous view, select **Back** from the shortcut menu, press the **Esc** key on your keyboard, or select the up or back arrow on the toolbar.

Editing a Subchart

After you open a subchart (see "Opening a Subchart" on page 7-8), 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 "Using Supertransitions to Extend Transitions" on page 7-10 for information on creating supertransitions.

Navigating Subcharts

The Stateflow Editor toolbar contains a set of buttons for navigating a chart's subchart hierarchy.

Tool	Description
	If the Stateflow Editor is displaying a subchart, replaces the subchart with the subchart's parent in the Stateflow Editor. If the Stateflow Editor is displaying a top-level chart, this button raises the Simulink model window containing that chart.
	Returns to the chart that you visited before the current chart. Lets you navigate up the hierarchy.
	Returns to the chart that you visited after visiting the current chart. Lets you navigate down the hierarchy.

Note You can also use the key sequence **..** (that is, press the period key twice) to navigate up to the parent object for a subcharted state, box, or function.

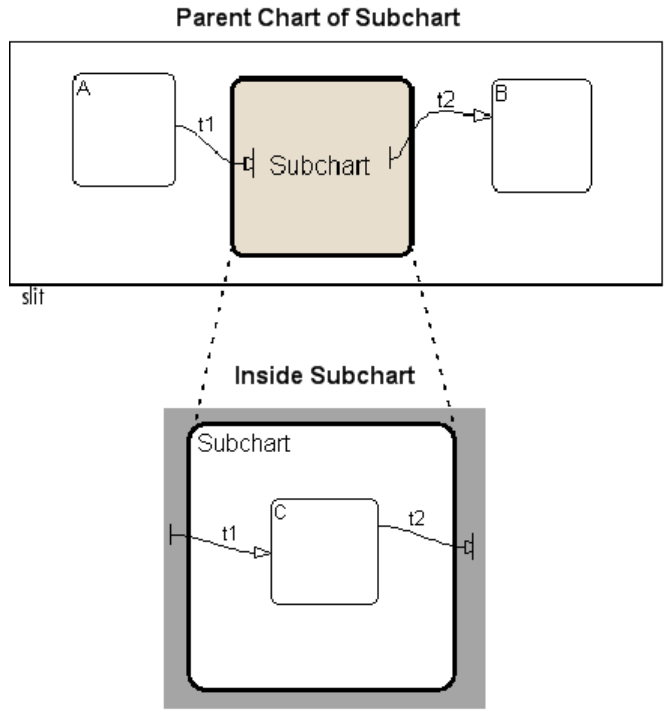
Using Supertransitions to Extend Transitions

In this section...

What Is a Supertransition?

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 following chart shows two supertransitions as seen from the perspective of a subchart and its parent chart, respectively.



In this example, supertransition `t1` goes from state A in the parent chart to state C in the subchart and supertransition `t2` goes from state C in the subchart to state B in the parent chart. Note that both segments of `t1` and `t2` have the same label.

Drawing a Supertransition Into a Subchart

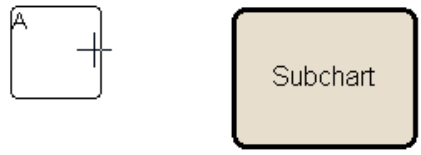
Use the following steps to draw a supertransition from an object outside a subchart to an object inside the subchart.

Caution 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.

---

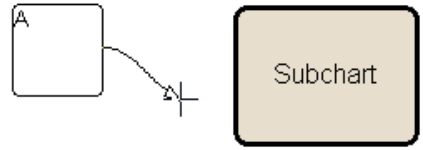
1 Position your pointer over the border of the state.

The pointer assumes the crosshairs shape.



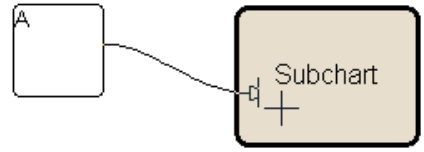
2 Drag the mouse.

Dragging the mouse causes a supertransition segment to appear. The segment looks like a regular transition. It is curved and is tipped by an arrowhead.



3 Drag the segment's tip anywhere just inside the border of the subchart.

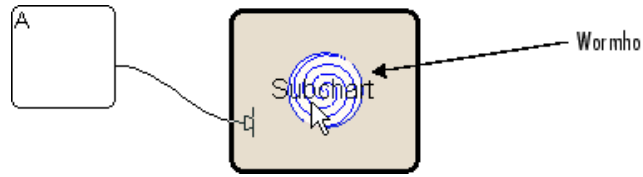
The arrowhead now penetrates the slit.



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.

4 Continue dragging your pointer toward the center of the subchart.

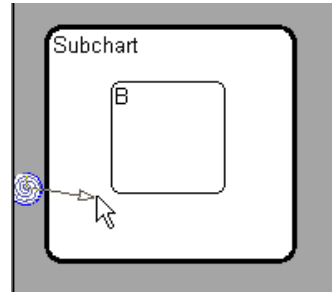
A wormhole appears in the center of the subchart.



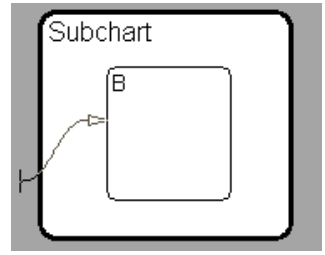
A *wormhole* allows you to open a subchart while drawing a supertransition.

5 Drag your pointer over the center of the wormhole.

The subchart opens. Now the wormhole and supertransition are visible inside the subchart.



6 Drag and drop the tip of the supertransition anywhere on the border of the object that you want to terminate the transition.



Note If the terminating object resides within a subchart in the current subchart, continue to drag the tip of the supertransition through the wormhole of the inner subchart and complete the connection inside the inner chart. In this way, you can draw a supertransition to an object at any subchart depth in the chart.

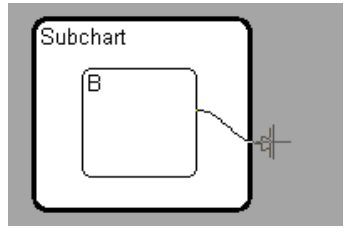
Drawing a Supertransition Out of a Subchart

Use the following steps to draw a supertransition out of a subchart.

Caution 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.

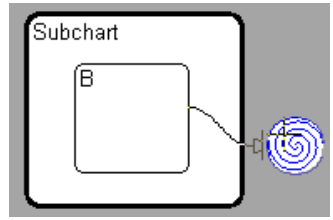
1. Draw an inner transition segment from the source object anywhere just outside the border of the subchart

   A slit appears as shown.



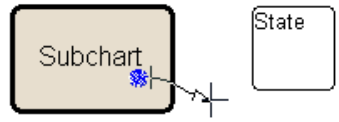
2. Keep dragging the transition away from the border of the subchart.

   A wormhole appears.

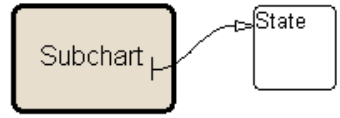


3 Drag the transition down the wormhole.

The parent of the subchart appears.



4 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, you can continue drawing by dragging the supertransition into the border of the parent subchart. This allows you to continue drawing the supertransition at the higher level. In this way, you can connect objects separated by any number of layers in the subchart hierarchy.

Labeling 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 “Labeling Transitions” on page 4-20). 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.

Extending Transitions with Smart Behavior

In this section...

About Smart Behavior Transitions

Transitions with smart behavior — known as *smart transitions* — attach their ends to the surfaces of Stateflow objects and, therefore, maintain their shapes and uniqueness when you rearrange chart objects.

Setting Smart Behavior in Transitions

Transitions are automatically created with smart behavior, on the assumption that this behavior is desirable in most circumstances. You can disable or enable smart behavior in existing transitions with the following procedure:

1. Right-click a transition.

   On the resulting menu, observe the selection titled **Smart**. If a check mark appears in front of **Smart**, the transition has smart behavior.

2. If **Smart** is not checked, select it to enable smart behavior.

   To disable smart transition behavior, select **Smart** if it is already checked.

See the following sections for a comparison of behavior between smart and nonsmart transitions:

- "What Smart Transitions Do" on page 7-18
- "What Nonsmart Transitions Do" on page 7-24

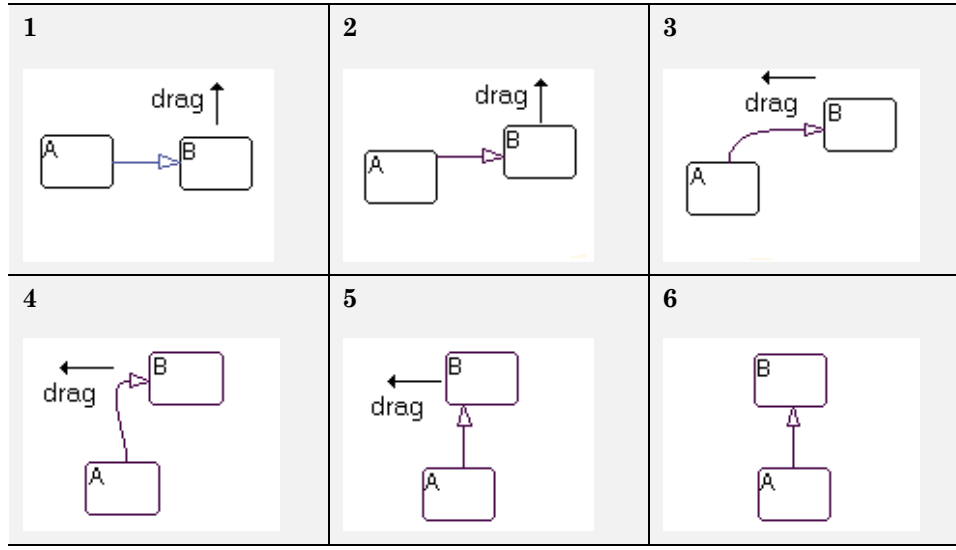
Note Transitions with smart behavior differ graphically only. Apart from graphical behavior, there is no difference in meaning between a transition with and without smart behavior.

What Smart Transitions Do

The following topics discuss some of the behaviors of smart transitions:

Smart Transitions Slide Around Surfaces

In the following example, state B is attached to state A by a smart transition. The example shows state B being dragged counterclockwise around the upper right corner of state A. When this occurs, state B turns to its selection color and the transition turns to a very light shade of gray, a sure sign of smart behavior. Dragging direction is shown by the arrows.



Note the following step-by-step behavior for the preceding example:

1 The first capture shows states A and B at the beginning of movement.

2 As B moves upward, the transition's back end slides upward on A, maintaining the transition straight.

3 As B moves around A's corner, the back end of the transition suddenly hops around A's upper right-hand corner. The transition is now curved from A's top surface to B's left side, maintaining perpendicularity with each attached state side.

Note A hop around a state's corner is a necessity because transitions are restricted from attaching at corners of states.

---

4 As B moves on top of A, the transition stays curved but its front end slides down to B's lower left-hand corner.

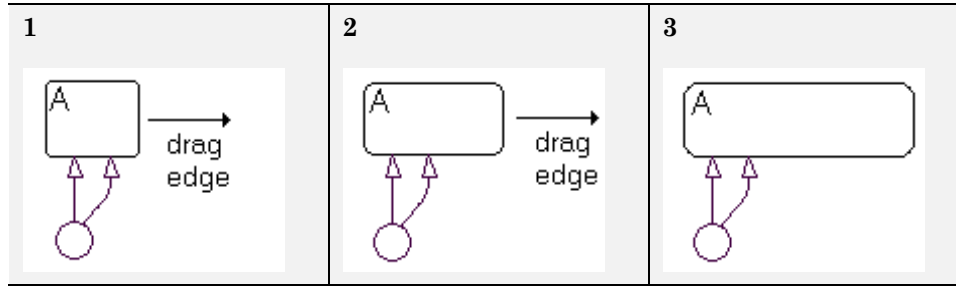
5 As B continues to move to the left over A, the transition's front end hops around B's lower left-hand corner.

6 Finally, as B moves directly over A, the transition's front end slides onto B's bottom edge.

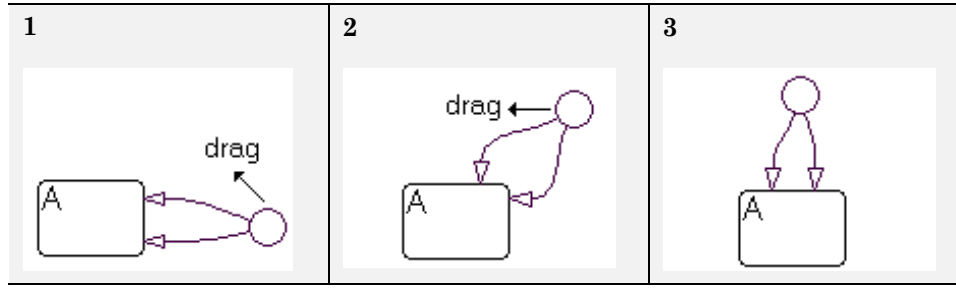
As B continues to circle A, steps 1 through 6 repeat for each of A's remaining sides.

Smart Transitions Slide and Maintain Shape

While transitions with smart behavior allow their ends to slide around the surfaces of their connected objects, they also attempt to maintain their original shape during moving. In the following example, a pair of transitions with smart behavior slide during a resizing to maintain their original shape.

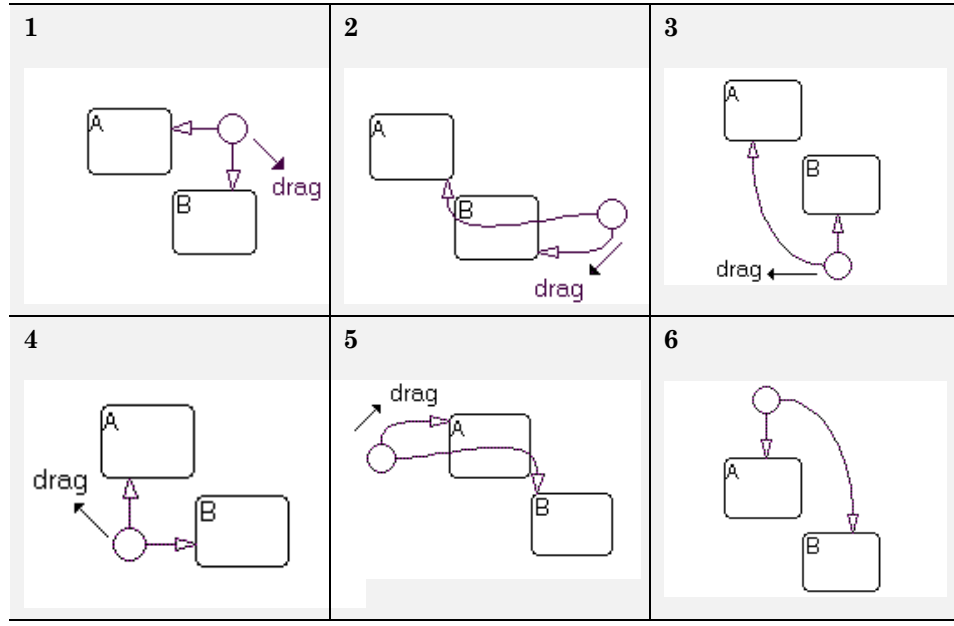


In the following example, the ends of a pair of transitions with smart behavior emanate from a junction and terminate in a state. As the junction is dragged around the state, the ends slide around the state and maintain the same relative spacing between each other. Direction is indicated by the arrows.



Smart Transitions Connect States to Junctions at 90 Degree Angles

Straight-line connections to states must be in one of four directions: left, right, up, or down. To maintain their straightness, smart transitions from junctions always seek to connect to a state through equivalent locations on the junction (left, right, top, bottom). In the following example, a junction is connected to two states, A and B. Watch the behavior of two straight smart transitions as the junction is moved to different locations.



1 The junction starts with two straight smart transition connections to states A and B.

2 The junction connects to state A through its left side. Since the junction is below A, only a curved connection is possible.

State B could be connected by a straight line through the junction's left side, but this is already occupied by the connection to A. Therefore, B is connected through the junction's bottom, and must be curved.

3 The junction connects to B by a straight transition through the junction's top connection. No straight-line connection to A is possible, therefore the junction is connected to state A with a curved transition through its left side.

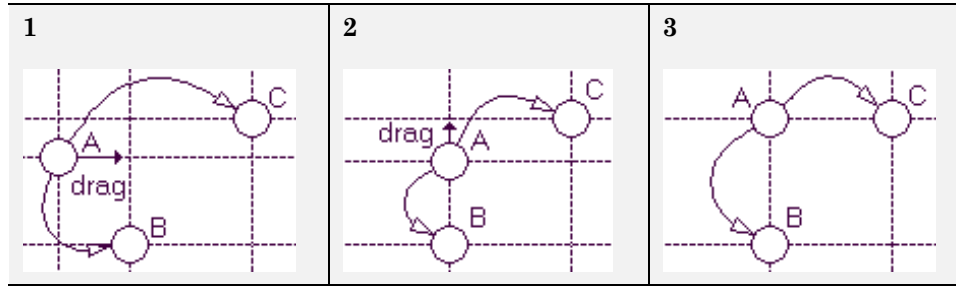
4 At this location (under A, to the left of B), straight-line transitions to A and B are possible from the junction's top and right connection points, respectively.

5 At the location left of state A, the junction connects to state B through its right connection point. Since the junction is above B, only a curved connection is possible.

6 Above A, a straight-line transition to state A is possible through the junction's bottom connector. A straight-line connection to state B is not possible, so the junction is connected to state B through a curved transition from its right connection.

Smart Transitions Snap to an Invisible Grid

Junctions that are connected to other junctions with smart transitions will snap to an invisible grid consisting of horizontal and vertical lines that pass through the center of each junction. The following example depicts this behavior.



Here, the invisible grid is depicted for each of the three junctions by dashed vertical and horizontal lines. Each junction is connected to each other through nonlinear smart transitions:

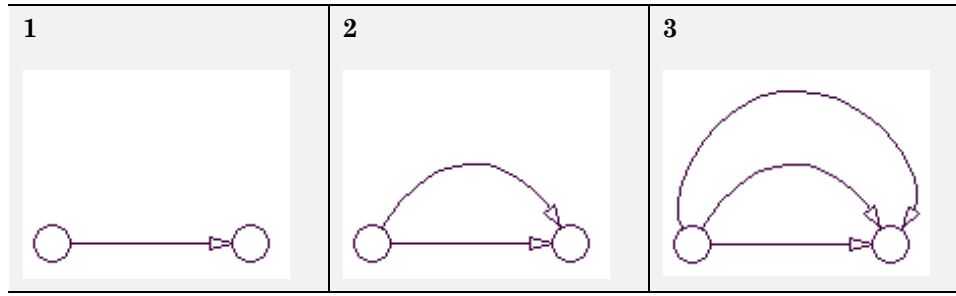
1. In the first scene, the snap grid for each junction does not overlap. The arrow indicates that junction A is being moved toward the vertical snap line for junction B.

2. When A is within a very small distance of B's snap line, A snaps into position directly above B and centered in its vertical snap line. The arrow indicates that A is now being moved toward the horizontal snap line for junction C.

3. When A is within a very small distance of C's horizontal snap line, A snaps into position directly to the side of C and centered in its horizontal snap line.

Smart Transitions Bow Symmetrically

Transitions with smart behavior bow symmetrically between junctions. In the following examples, transitions with smart behavior are drawn between two junctions:

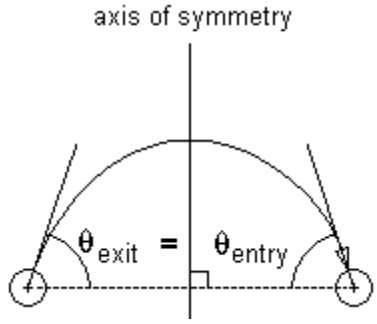


1. In the first case, a transition originates at the junction on the left and terminates on the left side of the right junction. This results in a straight line.

2. In the second case, a transition originates at the junction on the left and terminates on the top of the right junction. This results in a transition line bowed up.

3. In the third case, a transition originates at the junction on the left and terminates on the right side of the right junction. This results in a transition line bowed up even more.

Bowed smart transitions maintain symmetry by maintaining equality between transition entry and exit angles as shown below.



You can bow a smart transition between two junctions to any degree by placing your pointer on any point in the transition (except the attachment points) and clicking and dragging in a direction perpendicular to a straight line connecting the two junctions. You can move the mouse in any direction to bow the transition but only the component perpendicular to the straight line applies.

Disabling smart behavior for a transition allows you to distort the transition asymmetrically (see section "Nonsmart Transitions Distort Asymmetrically" on page 7-25). However, if you enable smart behavior again, the transition automatically returns to its prior symmetric bowed shape.

What Nonsmart Transitions Do

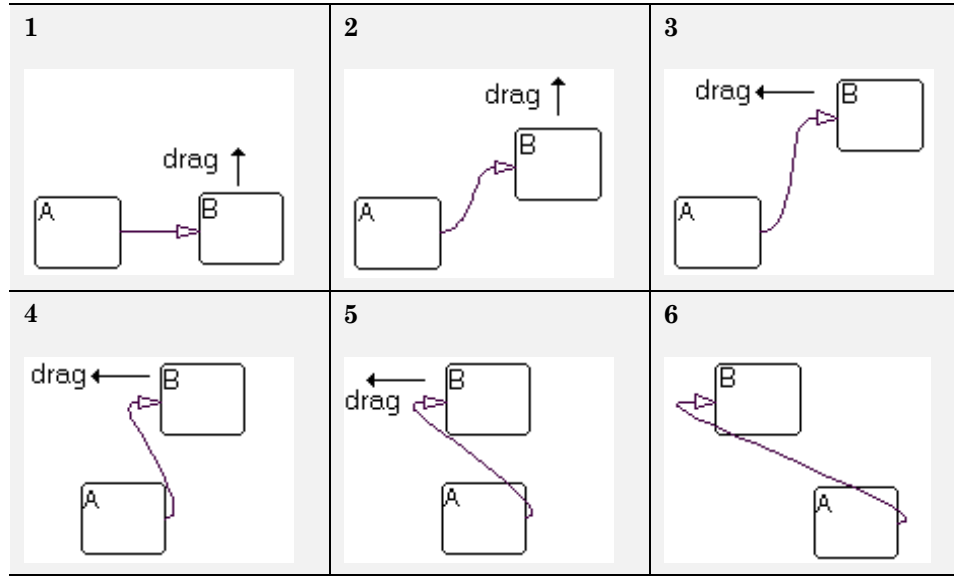
The following topics describe some of the behavior exhibited by transitions without smart behavior.

- "Nonsmart Transitions Anchor Connection Points" on page 7-25
- "Nonsmart Transitions Distort Asymmetrically" on page 7-25

You can disable and enable smart behavior in transitions. See the section "Setting Smart Behavior in Transitions" on page 7-17.

Nonsmart Transitions Anchor Connection Points

Contrast the example in the section "Smart Transitions Slide Around Surfaces" on page 7-18 with the example shown below.

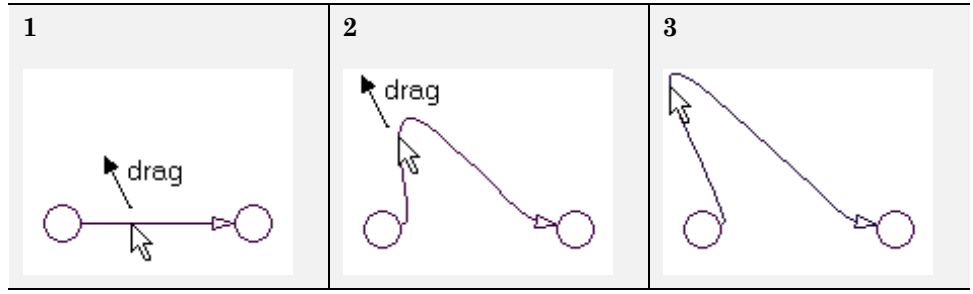


A nonsmart transition connects state A to state B. The pointer is then placed over state B and clicked and dragged to new locations counterclockwise around A. When this occurs, state B turns to its highlight color but the transition remains unchanged, a sure sign of a nonsmart transition.

As B is moved around A, the transition changes into a distorted curve that maintains the original attachment points. These remain unchanged in position, although the angle of attachment is always perpendicular to the side of the state.

Nonsmart Transitions Distort Asymmetrically

Simply by clicking and dragging on different locations along a transition without smart behavior, you can reshape it into an asymmetric curve suited to your individual preferences. This is illustrated in the following example:



For this example, use the following procedure:

1. Drag a horizontal transition between two junctions.
2. Right-click the transition and select **Smart** from the resulting shortcut menu to disable smart behavior.
3. Place your pointer anywhere on the transition.
4. Click and drag your pointer up and to the left.

Using Graphical Functions to Extend Actions

In this section...

What Is a Graphical Function?

A graphical function is a program that you write with flow graphs using connective junctions and transitions. You create a graphical function, fill it with a flow graph, and call it many times in the actions of states and transitions.

Why Use a Graphical Function?

A graphical function is easier to create, access, and manage than a textual function, such as a C or MATLAB function that you must define externally. Like a textual function, a graphical function can accept arguments and return values. Unlike a textual function, a graphical function is a native Stateflow object. You use the Stateflow Editor to create a graphical function that resides in your model along with the charts that invoke the function.

Where to Use a Graphical Function

A graphical 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. In particular, graphical functions are visible to the chart, to the parent state and its parents, and to sibling transitions and states. These exceptions apply:

- If the chart containing the function exports its graphical functions, the scope of the function is the entire Stateflow machine, which encompasses all the charts in the model. See "Exporting Chart-Level Graphical Functions" on page 7-35 for more information.
- A function that you define in a state or subchart overrides any functions of the same name in the parents and ancestors of that state or subchart.

Workflow for Defining a Graphical Function

Creating a Graphical Function

Use these steps to create a graphical function in your chart:

1 Click the graphical function icon in the Stateflow Editor toolbar:

2 Move your pointer to the location for the new graphical function in your chart and click to insert the function box.

3 Enter the function signature.

The function signature specifies a name for your function and the formal names for its arguments and return values. A signature has this syntax:

$$[r_1, r_2, \ldots, r_n] = \texttt{func}(a_1, a_2, \ldots, a_n)$$

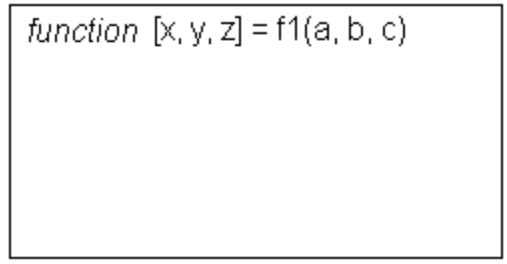
where `func` is the name of your function, a_1, a_2, ..., a_n are formal names for its arguments, and r_1, r_2, ..., r_n are formal names for its return values.

Note You can define arguments and return values as scalars, vectors, or 2-D matrices of any data type.

---

4 Click outside of the function box.

The following signature is for a graphical function that has the name `f1`, which takes three arguments (`a`, `b`, and `c`) and returns three values (`x`, `y`, and `z`).



Note You can use the Stateflow Editor to change the signature of your graphical function at any time. After you edit the signature, the Model Explorer updates to reflect the changes.

Programming a Graphical Function

To program a graphical function, follow these steps:

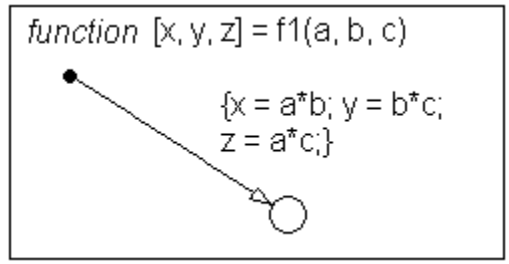
1. Click the default transition icon in the Stateflow Editor toolbar:

2. Move your pointer inside the function box in your chart and click to insert the default transition and its terminating junction.

3. Enter transition conditions and actions for your graphical function. If necessary, add connective junctions and transitions to your function.

Note Connective junctions and transitions are the only graphical elements you can use in a graphical function. Because a graphical function must execute completely when you call it, you cannot use states.

This function box shows a flow graph that returns different products of its arguments.

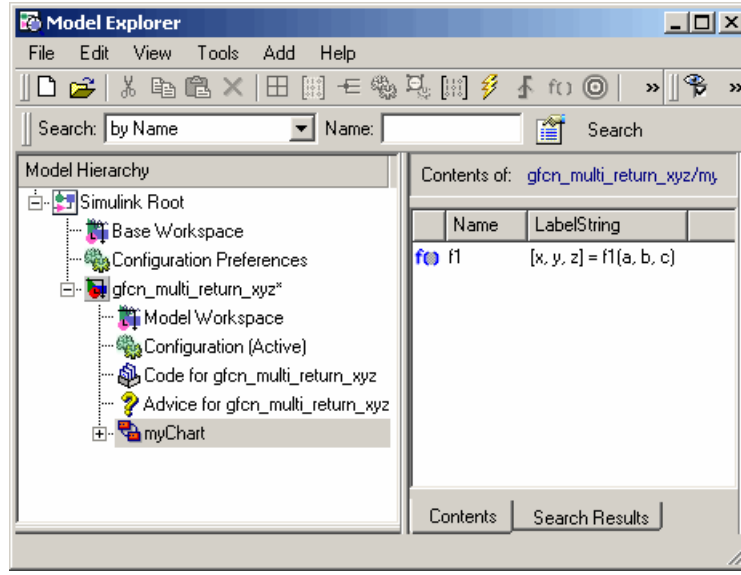


Defining Graphical Function Data

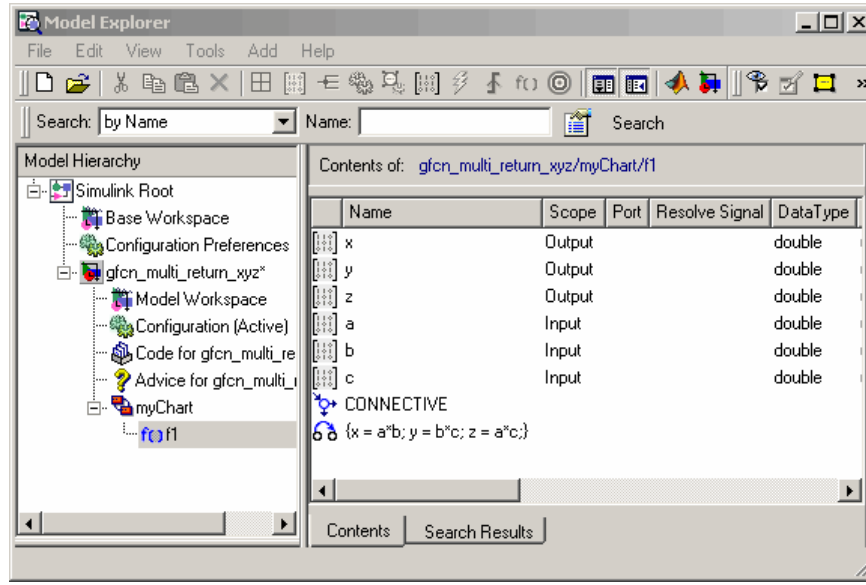
You must define the data in your graphical function:

1 In the Stateflow Editor, select **View** > **Model Explorer**.

The signature for your function appears as a property of its parent chart in the **Contents** pane of the Model Explorer.



2 Expand the parent object in the Model Explorer, so that you can see the return values and arguments of the function signature as data items that belong to your graphical function.



The **Scope** column in the Model Explorer indicates the role of each argument or return value. Arguments have the scope `Input`, and return values have the scope `Output`.

3 For each function argument and return value, right-click the data row in the Model Explorer and select **Properties** from the context menu.

4 In the Data properties dialog box for each argument and return value, specify the data properties.

These rules apply:

- Each argument and return value can be a scalar or matrix of values.
- Arguments cannot have initial values.

5 Create any additional data items that your function must have to process its programming.

Your function can access its own data or data belonging to parent states or the chart. The data items that you create for the function itself can have one of these scopes:

- `Local`

  Local data persists from one function call to the next.

- `Temporary`

  Temporary data initializes at the start of every function call.

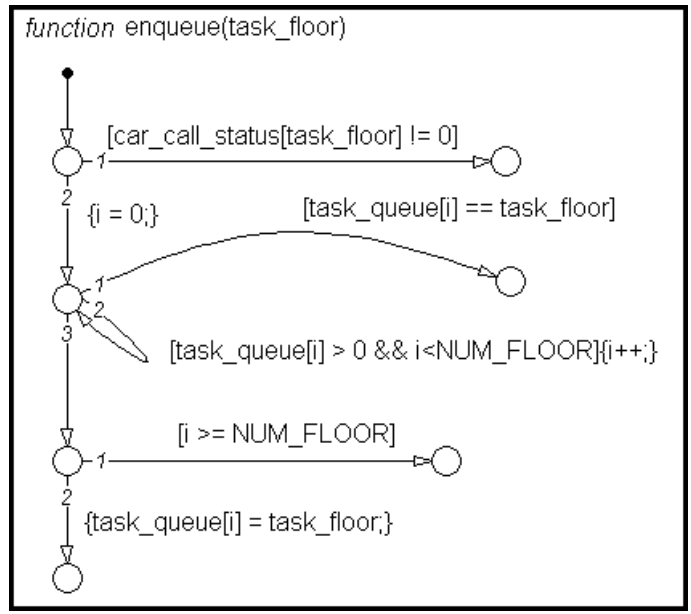
- `Constant`

  Constant data retains its initial value through all function calls.

Note You can initialize your function data (other than arguments and return values) from the MATLAB workspace. However, you can save only local items to this workspace.

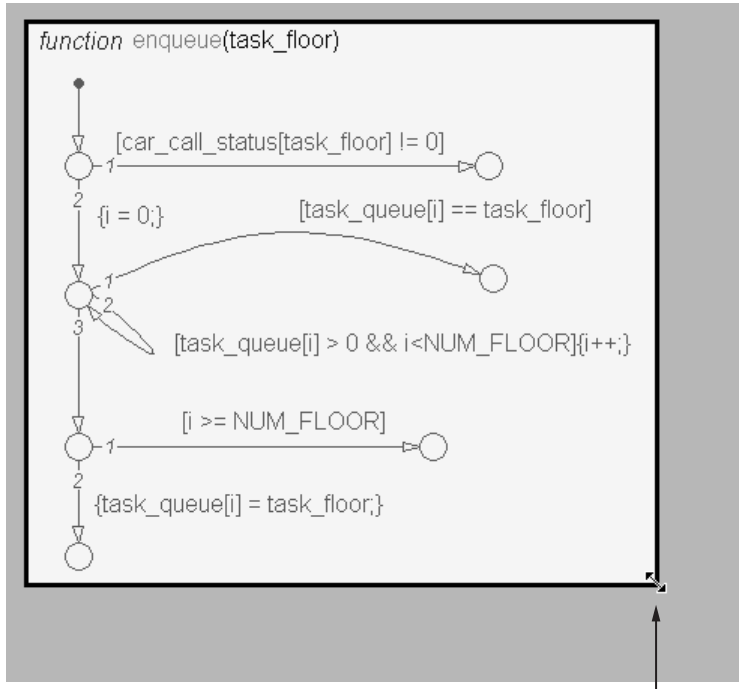
Managing Large Graphical Functions

You can make your graphical function as large as you want, as shown below.



However, if your function grows too large, you can hide its contents by right-clicking inside the function box and selecting **Make Contents > Subcharted** from the context menu. This option makes your graphical function opaque.

To access the programming of your subcharted graphical function, double-click it. This action dedicates the entire chart window to programming your function.



To access your original chart, click the Back button  .

Calling Graphical Functions in Stateflow Action Language

Description

To call your graphical function, use Stateflow action language. Any state or transition action in the scope of your function can perform a function call.

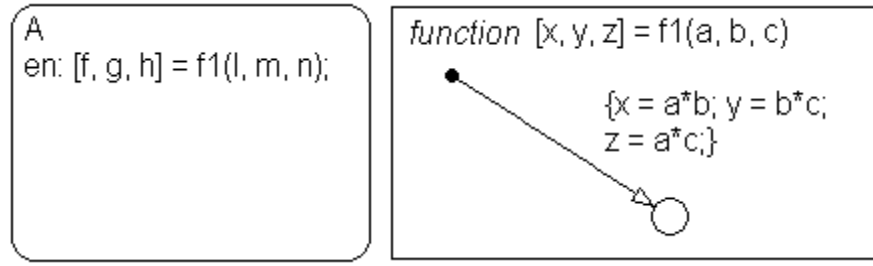
Syntax

Syntax for a function call is the same as that of a function signature, with actual arguments replacing the formal ones specified in a signature. If the data types of the actual and formal argument differ, a function casts the actual argument to the type of the formal argument.

See "Creating a Graphical Function" on page 7-28 for information about syntax for a function signature.

Example

In this example, a state `entry` action calls a graphical function that returns three products.



Exporting Chart-Level Graphical Functions

Why Export Graphical Functions?

When you export chart-level graphical functions, you extend the scope of your functions to all other charts in your model.

How to Export Chart-Level Graphical Functions

Perform these steps to export graphical functions to your main model:

1. Open the chart where your graphical function resides.
2. In the Stateflow Editor, select **File > Chart Properties**.

3 In the Chart properties dialog box, select **Export Chart Level Graphical Functions (Make Global)**.

4 If your graphical function resides in a library chart, link that chart to your main model.

Rules for Exporting Chart-Level Graphical Functions

Link library charts to your main model to export graphical functions from libraries

You must perform this step to export graphical functions from library charts. Otherwise, an error message appears when you simulate your model.

Do not export graphical functions that contain inputs or outputs of enumerated data type

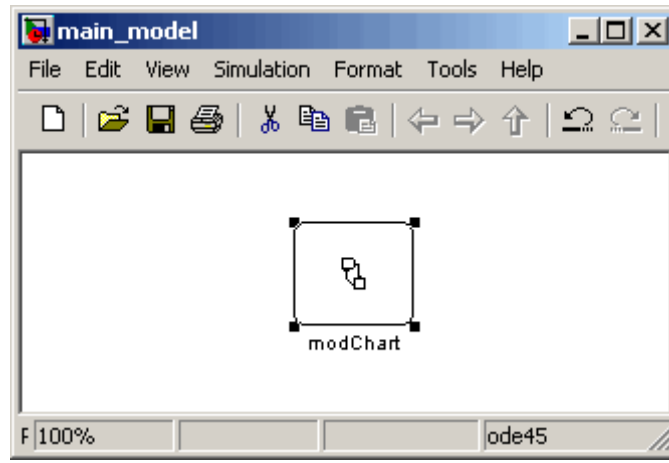
This restriction prevents error messages from appearing when you simulate your model.

Example of Exporting Chart-Level Graphical Functions

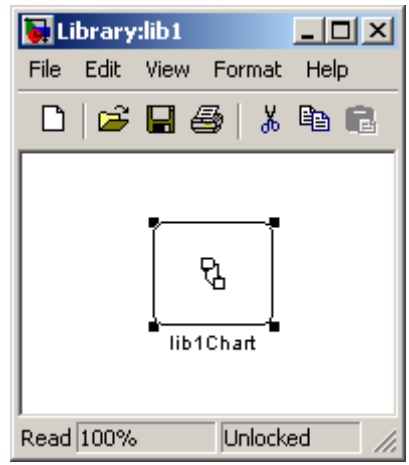
This example describes how to export graphical functions in library charts to your main model.

1 Create these objects:

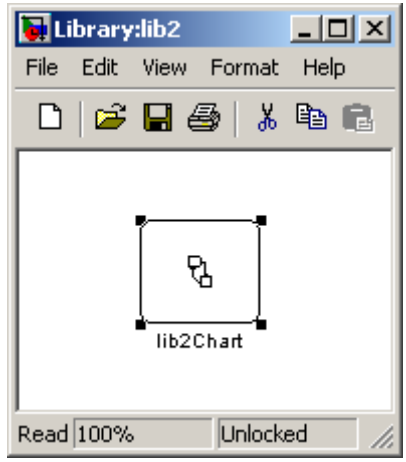
- Add a model named `main_model`, with a chart named `modChart`.



- Add a library model named `lib1`, with a chart named `lib1Chart`.

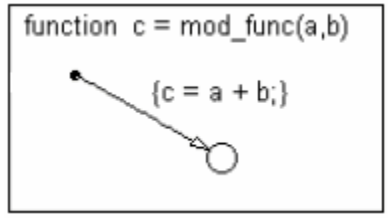


- Add a library model named `lib2`, with a chart named `lib2Chart`.

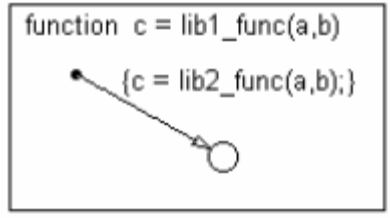


2 Create these graphical functions:

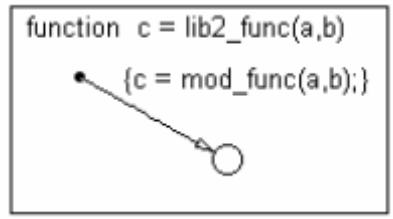
- For `modChart`, add this graphical function.



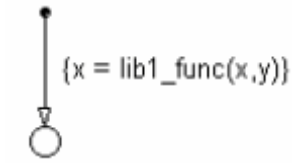
- For `lib1Chart`, add this graphical function.



- For `lib2Chart`, add this graphical function.



3 For `modChart`, add a default transition with this action.



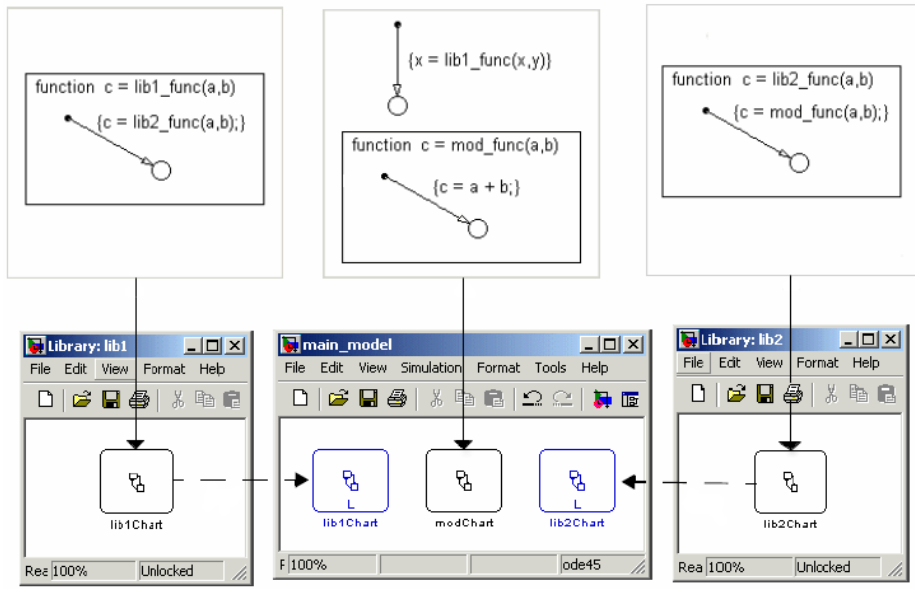
4 For each chart, follow these steps:

a Select **File > Chart Properties** in the Stateflow Editor.

b In the Chart properties dialog box, select the **Export Chart Level Graphical Functions (Make Global)** option.

c Click **OK**.

5 Drag `lib1Chart` and `lib2Chart` into `main_model` from `lib1` and `lib2`, respectively.



Each chart now defines a graphical function that any chart in `main_model` can call.

6 In the Stateflow Editor, select **View > Model Explorer**.

7 In the **Model Hierarchy** pane of the Model Explorer, navigate to `main_model`.

8 Add the data `x` to the Stateflow machine as follows:

- **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.

9 Add the data `y` to the Stateflow machine as follows:

- **a** Select **Add > Data**.

b In the **Name** column, enter `y`.

c In the **Initial Value** column, enter `1`.

d Use the default settings for other properties.

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

11 In the Configuration Parameters dialog box, go to the **Solver** pane.

12 In the **Solver options** section, make these changes:

a For **Type**, choose `Fixed-step`.

b For **Solver**, choose `Discrete (no continuous states)`.

c For **Fixed-step size**, enter `1`.

d Click **OK**.

These actions ensure that when you simulate your model, a discrete solver is used. For more information, see "Solvers" in the Simulink software documentation.

What Happens During Simulation. When you simulate the model, these actions take place during each time step.

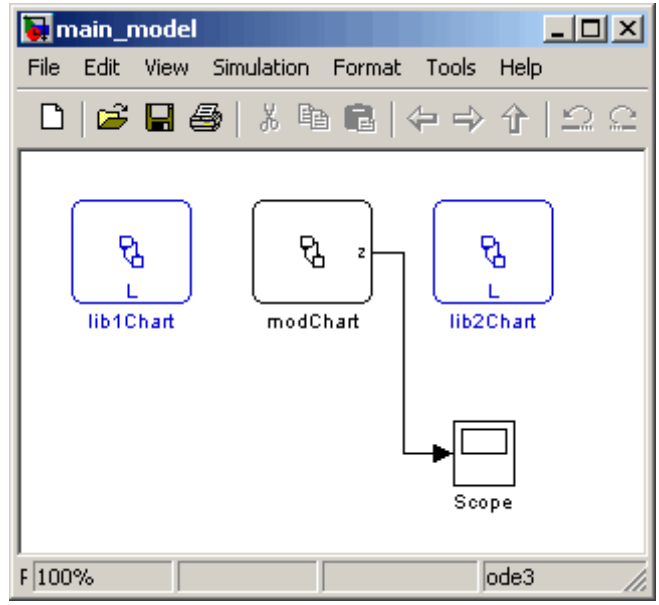
Phase	The object...	Calls the graphical function...	Which...
1	`modChart`	`lib1_func`	Reads two input arguments `x` and `y`
2	`lib1_func`	`lib2_func`	Passes the two input arguments
3	`lib2_func`	`mod_func`	Adds `x` and `y` and assigns the sum to `x`

How to View the Simulation Results. To view the simulation results, add a scope to your model. Follow these steps:

1. In the Simulink model window, select **View > Library Browser**.
2. From the Simulink/Sinks Library, select the Scope block and add it to the model.
3. Open the Model Explorer.
4. In the **Model Hierarchy** pane, navigate to `modChart`.
5. Add the output data `z` to the Stateflow chart as follows:
   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 Simulink model window, connect the outport from `modChart` to the inport of the Scope block.

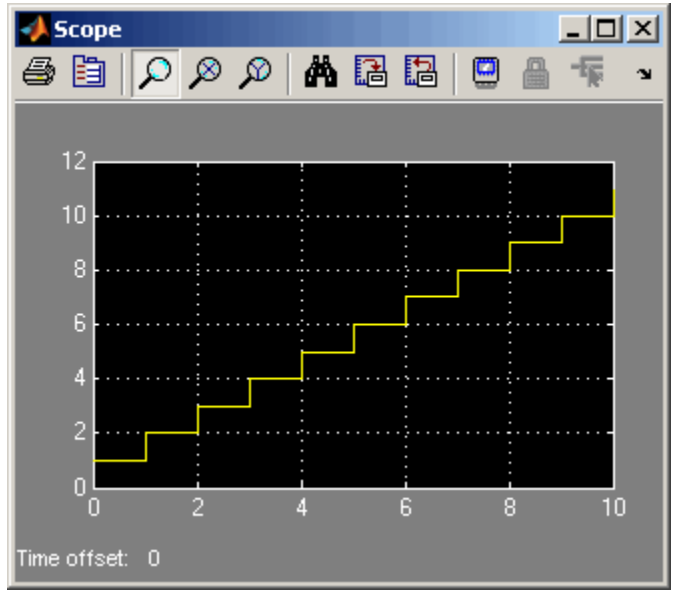


8 Double-click the Scope block to open the display.

9 Start simulation.

10 After the simulation ends, right-click in the scope display and select **Autoscale**.

The results look something like this.

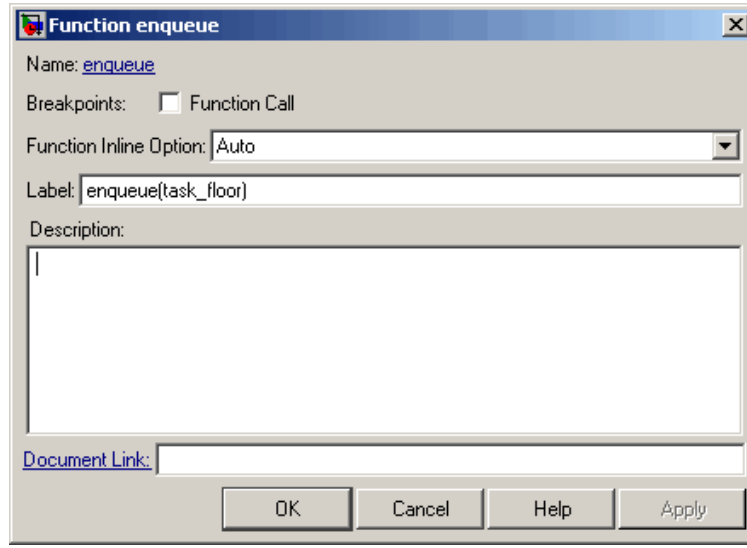


Specifying Graphical Function Properties

You can set general properties for your graphical function through its properties dialog box:

1. Right-click your graphical function box.
2. Select **Properties** from the context menu.

   The properties dialog box for your graphical function appears.



The fields in the properties dialog box are:

Field	Description
Name	Click this read-only function name to bring your function to the foreground in its native chart.
Breakpoints	Select **Function Call** to set a breakpoint that pauses simulation when your graphical function executes.

Field	Description
Function Inline Option	Select one of these options to control the inlining of your function in generated code: • **Auto** Decides whether or not to inline your function based on an internal calculation. • **Inline** Inlines your function as long as 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	Specify the signature label for your function in this field. See "Creating a Graphical Function" on page 7-28 for more information.
Description	Enter a 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`.

Using Boxes to Extend Charts

In this section...

When to Use Boxes

Use boxes to organize graphical objects in your chart.

Semantics of 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 "Using a Box to Group Functions" on page 7-50.

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 "Using a Box to Group States" on page 7-52.

Note To specify activation order explicitly on a state-by-state basis, you must select **User specified state/transition execution order** in the Chart properties dialog box. This option is selected by default when you create a new Stateflow chart. For details, see "Explicit Ordering of Parallel States" on page 3-40.

Rules for Using Boxes

When you use a box, these rules apply:

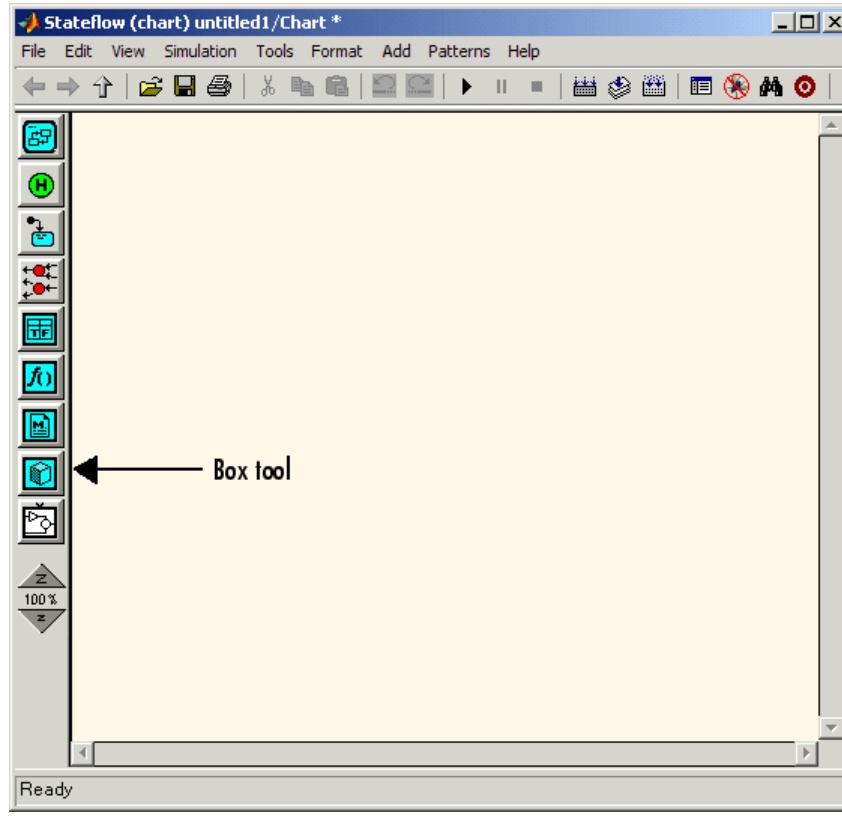
- You must include the box name in the path when you refer to a box-parented function or state from a location outside of the box.
- You can move or draw graphical objects inside a box, such as functions and states.

  You can draw a state around the objects you want inside it and then convert that state to a box. See "Changing a State to a Box" on page 7-50.
- 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 "Grouping States" on page 4-8.
- You can subchart a box to hide its elements. See "Using Subcharts to Extend Charts" on page 7-5.
- 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.

Drawing and Editing a Box

Creating a Box

You create boxes by drawing them in the Stateflow Editor with the box tool shown below.



1 Select the Box tool.

2 Move your pointer into the drawing area.

In the drawing area, your pointer becomes box-shaped.

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.

A text cursor appears in place of the question mark.

5 Enter a name for the box and then click outside of the box.

Deleting a Box

To delete a box, click it to select it and choose **Edit > Cut** from the context menu or press the **Delete** key.

Changing a State to a Box

You can change an existing state to a box and back to a state with this procedure:

1 Right-click the state.

2 From the context menu, select **Type**.

A submenu appears adjacent to the context menu.

3 From the submenu, select **Box**.

This action converts the state to a box, redrawing its border with sharp corners to indicate its changed status.

4 Repeat the preceding steps on the box and select **State** from the submenu instead of **Box** to change the box to a state.

Examples of Using Boxes

Using a Box to Group Functions

This chart shows a box named `Status` that groups together Embedded MATLAB functions.

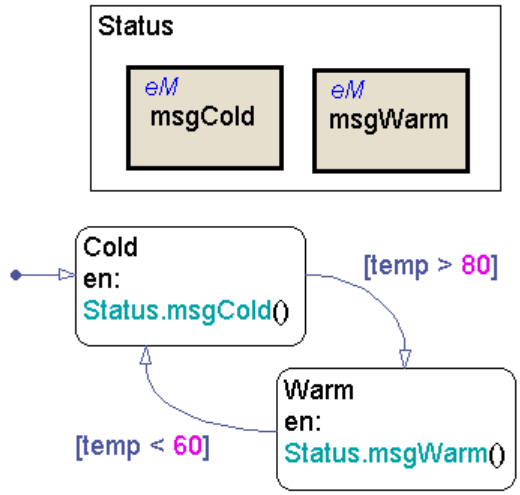


Chart execution takes place as follows:

1. The state `Cold` activates first.
2. Upon entry, the state `Cold` invokes the Embedded MATLAB function `Status.msgCold`.

   This function displays a status message that the temperature is cold.

   Note Because the Embedded 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 Embedded MATLAB function `Status.msgWarm`.

   This function displays a status message that the temperature is warm.

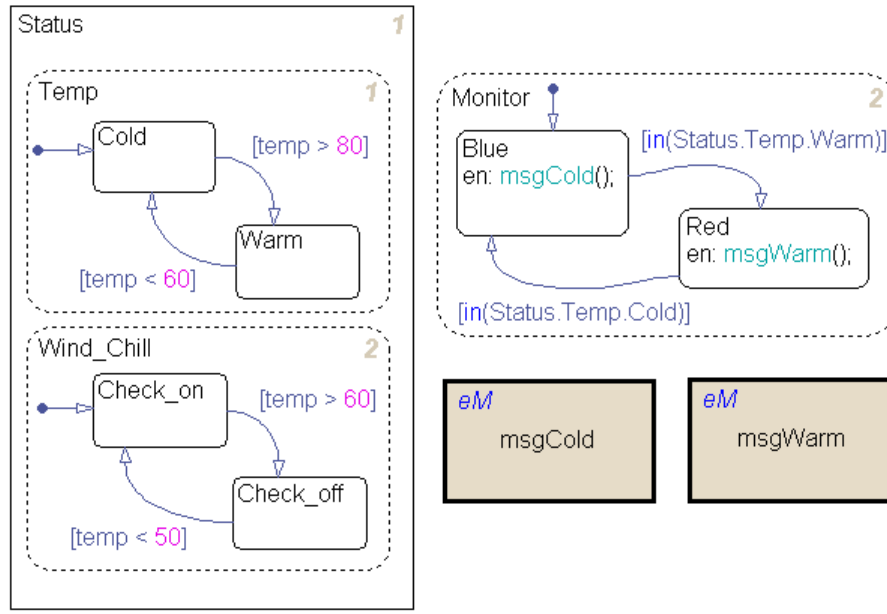
Note Because the Embedded MATLAB function resides inside a box, the path of the function call must include the box name `Status`. If you omit this prefix, an error message appears.

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.

Using a Box to Group States

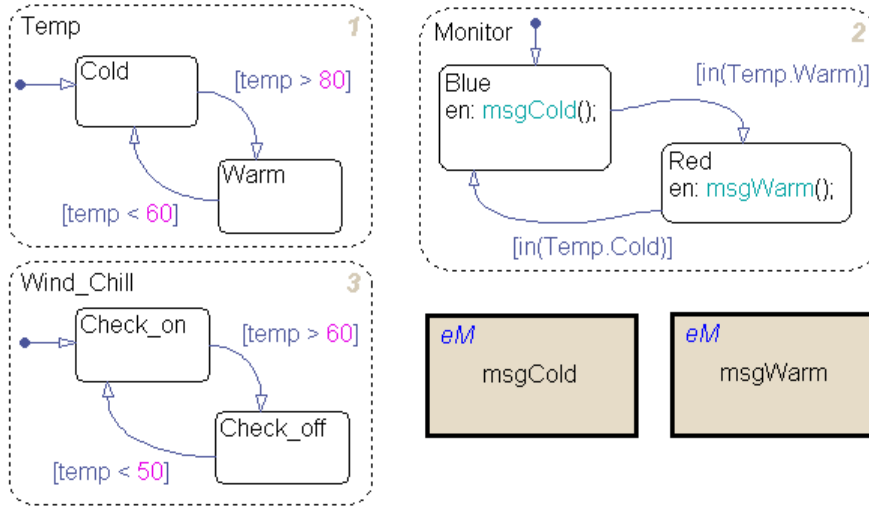
This chart shows a box named `Status` that groups together related states. The chart uses implicit ordering for parallel states, instead of the default explicit mode. (For details, see "Implicit Ordering of Parallel States" on page 3-42.)



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 "Checking State Activity" on page 10-89.

Using Notes to Extend Charts

In this section...

Creating Notes

You can enter comments/notes in any location on the chart with the following procedure:

1. Place your pointer at the desired location in the Stateflow chart.
2. Right-click the mouse.
3. From the resulting menu, select **Add Note**.

   A blinking cursor appears at the location you selected. Default text is italic, 9 point.
4. Begin typing your comments.

   As you type, the text moves left to right.
5. Press the **Return** key to start a new line.
6. When finished typing, click outside the typed note text.

Editing Existing Notes

To edit existing note text,

1. Left-click the mouse on the comment location you want to edit.
2. Once the blinking cursor appears, begin typing or use the arrow keys to move to a new text location.

Changing Note Font and Color

To change font and color for your chart notes, follow the procedures described in the section "Specifying Colors and Fonts in the Stateflow Editor" on page 4-30.

You can also change your note text to bold or italic text by doing the following:

1. Right-click the note text.
2. From the resulting shortcut menu, select **Text Format**.
3. From the resulting submenu, select **Bold** or **Italic** (default).

TeX Instructions

In the preceding procedure, note a third selection of the **Text Format** submenu called **TeX Instructions**. This selection sets the text `Interpreter` property to `Tex`, which allows you to use a subset of TeX commands embedded in the string to produce special characters such as Greek letters and mathematical symbols.

The **TeX Instructions** selection is used in the following example:

1. Right-click the text of an example note.
2. In the resulting shortcut menu, select **Text Format**.
3. In the submenu that results, make sure that **TeX Instructions** has a check mark positioned in front of it. Otherwise, select it.
4. Click the note text to place your pointer in it.
5. Replace the existing note text with the following expression.

   ```
   \it{\omega_N = e^{(-2\pii)/N}}
   ```

6. Click outside the note.

   The note now has the following appearance:

   $$\omega_N = e^{(-2\pi i)/N}$$

Moving Notes

To move your notes,

1. Place your pointer over the text of the note.
2. Click and drag the note to a new location.
3. Release the left mouse button.

Deleting Notes

To delete your notes, do the following:

1. Place your pointer over the text of the note.
2. Click and hold the left mouse button on the note.

   A dim rectangle appears surrounding the note.
3. Select the **Delete** key.

Alternatively, you can also do the following:

1. Place your pointer over the text of the note.
2. Right-click the note.
3. From the resulting shortcut menu, select **Cut**.

Printing Stateflow Charts

In this section...

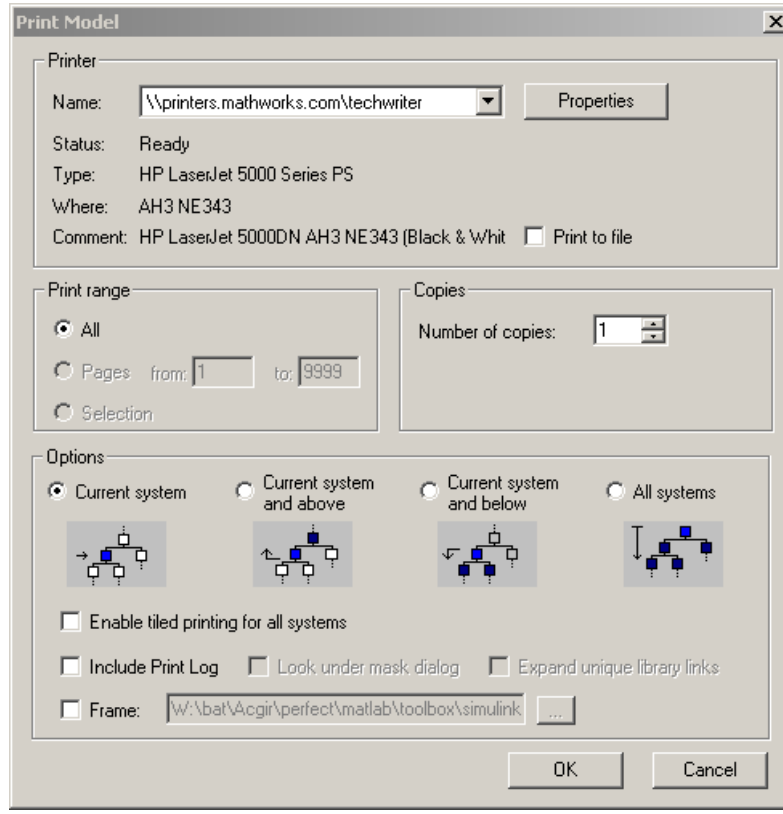
Printing Scaled Charts

By default, Stateflow software scales each chart that you print to fit on a single page. If you prefer to print charts without scaling to preserve clarity and detail, you can use tiled printing, as described in “Using Tiled Printing for Stateflow Charts” on page 7-61.

To print scaled charts, follow these steps:

1. Open the chart or subchart you want to print.
2. In the Stateflow Editor, select **File > Print**.

   The Print Model dialog box appears:



3 In the Print Model dialog box, select your printer and number of copies.

4 Select the charts you want to print by choosing one of these options:

Option	Description
Current system	Prints the current chart or subchart
Current system and above	Prints the current chart or subchart and all systems above it in the model hierarchy

Option	Description
Current system and below	Prints the current chart or subchart and all systems below it in the model hierarchy, with the option of looking into the contents of masked and library Simulink blocks
All systems	Prints all systems in the model hierarchy, with the option of looking into the contents of masked and library Simulink blocks

5 Customize your print job as needed using these options:

Option	Description
Enable tiled printing for all systems	Enables tiled printing for all charts and overrides any individual tiled-print settings. See "Using Tiled Printing for Stateflow Charts" on page 7-61.
Include Print Log	Includes a list of all printed charts.
Look under mask dialog	Prints the contents of Simulink masked subsystems when encountered at or below the level of the current chart or subchart (when printing **Current system and below**) or the top-level system (when printing **All systems**).

Option	Description
Expand unique library links	Prints the contents of library blocks that appear in Simulink subsystems that are printed with **Current system and below** or **All systems**.
Frame	Prints a title block frame with each chart. To learn how to create print frames, see "PrintFrame Editor Overview" in the Simulink User's Guide.

6 Click **OK**.

For more information about all print options, see "Printing a Block Diagram" in the Simulink software documentation.

Using Tiled Printing for Stateflow Charts

Stateflow charts support Simulink tiled printing options (see "Tiled Printing" in the Simulink documentation). Tiled printing enables you to print Stateflow charts without scaling to fit a page and, therefore, without sacrificing clarity and detail. With tiled printing, you can distribute a chart over a specified number of pages and, therefore, control the total size of the printed image. You can choose different tiled-print settings for each of your charts to customize the appearance of all printed images.

If you want to scale charts to fit on a single printed page, see "Printing Scaled Charts" on page 7-58.

Printing Charts on Tiled Pages

To print Stateflow charts on tiled pages, follow these steps:

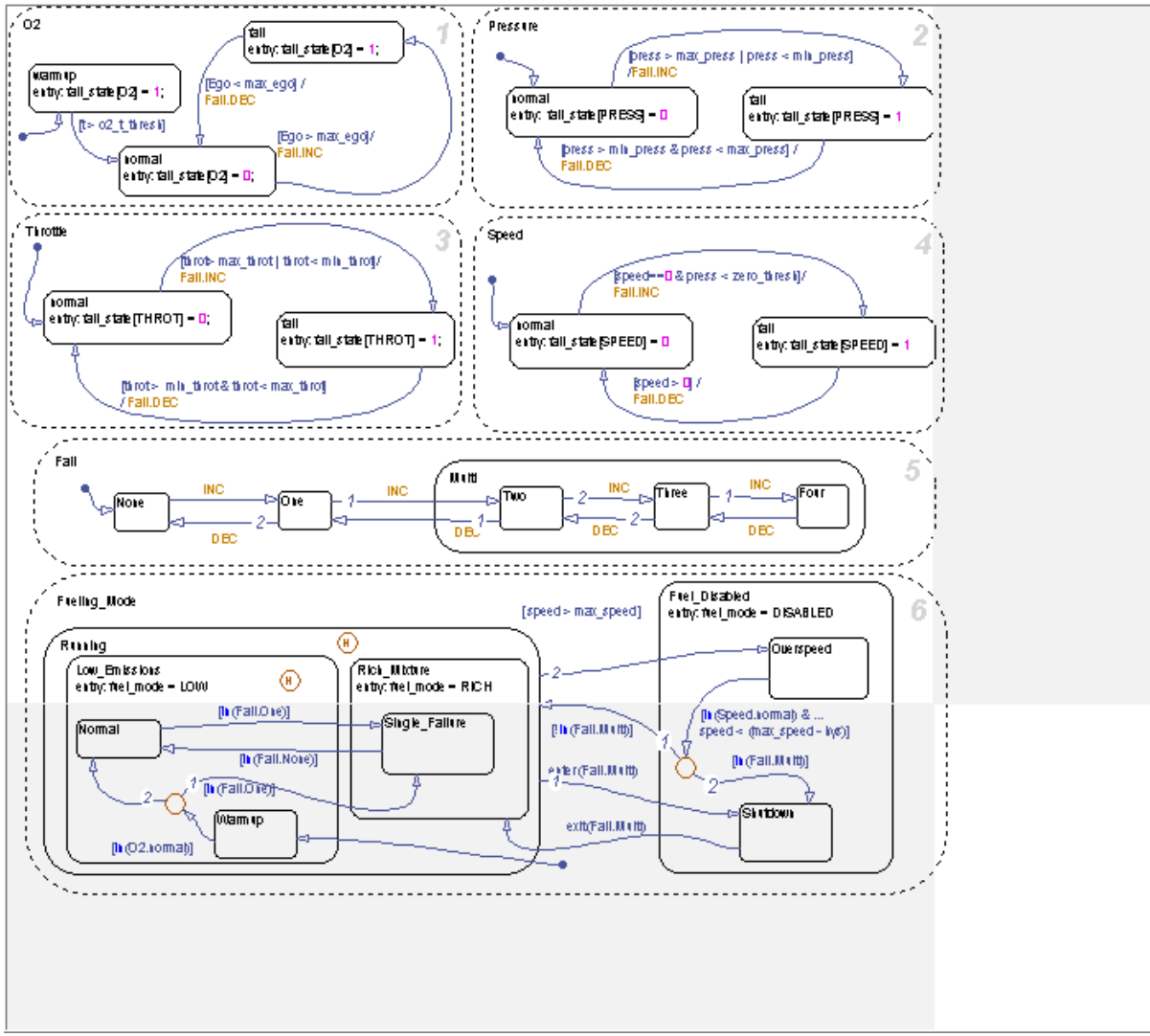
1 Open the chart or subchart you want to print.

2 In the Stateflow Editor, select **File > Enable Tiled Printing**.

To enable tiled printing for all systems in your model, select the **Enable tiled printing for all systems** check box on the Print dialog box

3 To visualize the chart's size and layout with respect to the page, select **View > Show Page Boundaries**.

If your chart is too large to fit on one page, the Stateflow Editor displays the page boundaries as tiles in a checkerboard pattern, as in this example:



In this chart, state #6 extends beyond the page boundary. To correct the problem, you can select and drag this state to a different tile so that it prints in its entirety on a separate page.

Note Stateflow software uses a row-major scheme to number tiled pages. For example, the first page of the first row is 1, the second page of the first row is 2, and so on.

4 Select **File > Print**.

By default, this command prints all of a system's tiled pages. Alternatively, you can specify a range of tiled page numbers to print. See "Printing Tiled Pages" in the Simulink documentation.

Generating a Model Report

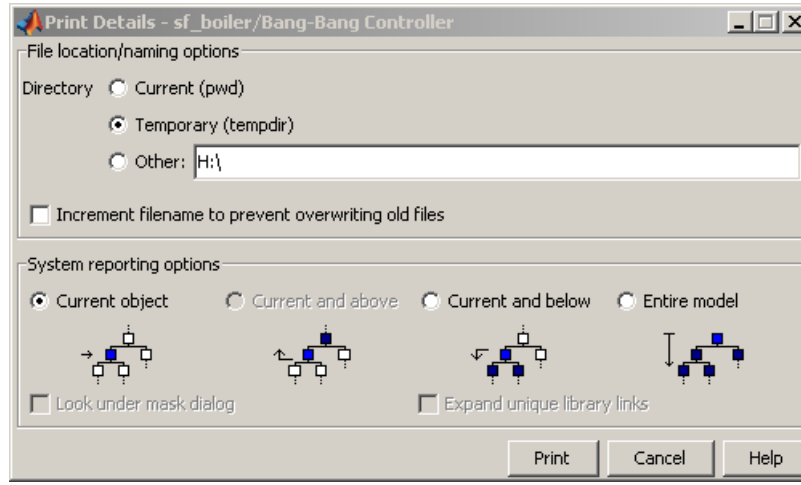
The **Print Details** report is an extension of the **Print Details** report in the Simulink model window. It provides a report of Stateflow and Simulink objects relative to the chart currently in view in the Stateflow Editor from which you select the report.

To generate a model report on chart objects, follow these steps:

1 Open the chart or subchart for which you want a report.

2 In the Stateflow Editor, select **File > Print Details**.

The Print Details dialog box appears as follows:



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 of this dialog box, see "Generating a Model Report" in the Simulink software documentation. For details on the report you receive from the option you choose in the **System reporting options** section, see "System Report Options" on page 7-66 and "Report Format" on page 7-66.

4 Click **Print**.

The Print Details dialog box appears and tracks the activity of the report generator during report generation. See "Generating a Model Report" in the Simulink software documentation for more details on this window.

If no serious errors occur, the HTML report appears in your default browser.

Note You can also use MATLAB® Report Generator™ software to generate a report that documents an entire model, including both Simulink and Stateflow objects. See the MATLAB Report Generator User's Guide.

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 Stateflow Editor and its immediate parent Simulink system.
- **Current and above** — This option is grayed out and unavailable for printing chart details in the Stateflow Editor.
- **Current and below** — Reports on the chart or subchart in the current Stateflow Editor 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 in the Stateflow Editor.
- 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.

Printing the Current Chart

The **Print Current View** option prints an individual chart or subchart as follows:

1. Open the chart or subchart that you want to print.
2. In the Stateflow Editor, select **File > Print Current View**.
3. In the submenu, choose one of these options:
   - **To File** — Converts the current view to a graphics file.

     Select the format for the graphics file from a submenu of graphical file types.
   - **To Clipboard** — Copies the current view to the system clipboard.

     Select the format for the clipboard copy from a submenu of graphical formats.
   - **To Figure** — Converts the current view to a MATLAB figure window.
   - **To Printer** — Prints the current view on the current printer.

You can also print the current view from the MATLAB command line using the `sfprint` function.

8

Defining Data

Adding Data

In this section...

When to Add Data

Add data when you want to define data that is visible to a specific level of the Stateflow hierarchy.

Where You Can Use Data

You can store and retrieve data that resides internally in the Stateflow workspace, and externally in the Simulink model or application that embeds the Stateflow chart. Stateflow actions can reference internal and external data.

Adding Data Using the Stateflow Editor

How to Add Data

To add data using the Stateflow Editor, follow these steps:

1. In the Stateflow Editor, select **Add** > **Data**.
2. In the context menu, select a scope for the new data object.

   See "Scope" on page 8-9 for a description of each type of scope.

   Selecting scope adds a default definition of the new data object to the Stateflow hierarchy and displays the Data properties dialog box.
3. Specify properties for the new data object in the Data properties dialog box, as described in "Setting Data Properties in the Data Dialog Box" on page 8-6.

Visibility of Data You Add in the Stateflow Editor

If you add data in the Stateflow Editor, that data is visible to all objects in the chart.

Adding Data Using the Model Explorer

How to Add Data

To add data using the Model Explorer, follow these steps:

1. In the Stateflow Editor, select **Tools** > **Explore**.

   The Model Explorer opens. If no object is selected, the current chart or subchart appears highlighted in the **Model Hierarchy** pane. Otherwise, the selected object appears highlighted.

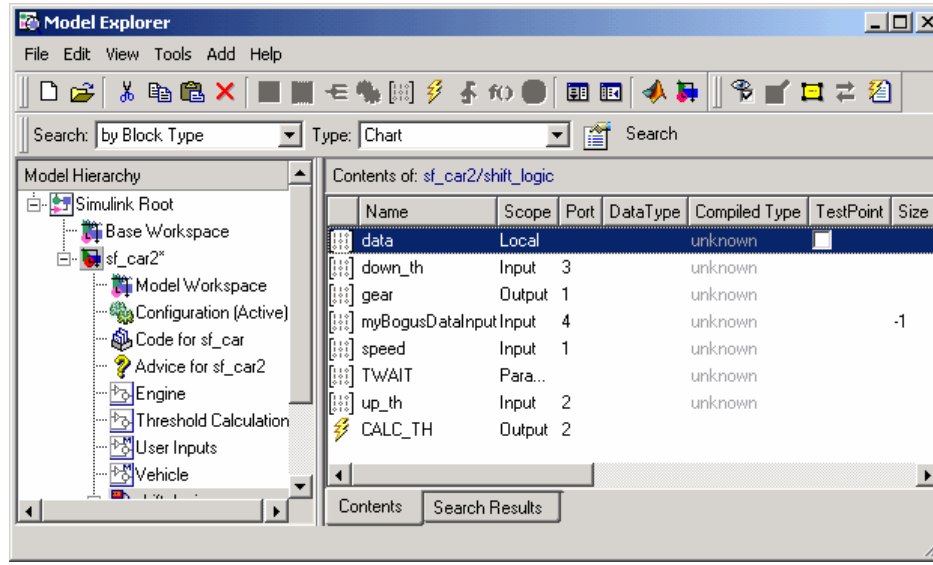
2. In the **Model Hierarchy** pane, select the object in the Stateflow hierarchy where you want the new data to be visible.

   The object you select becomes the parent of the data object.

3. In the Model Explorer, select **Add** > **Data**, or click the **Add Data** button:

   This action adds a default definition for the data in the hierarchy, and the data definition appears in a new row in the Model Explorer **Contents** pane.



4 Change the properties of the data, as described in "Setting Data Properties in the Data Dialog Box" on page 8-6.

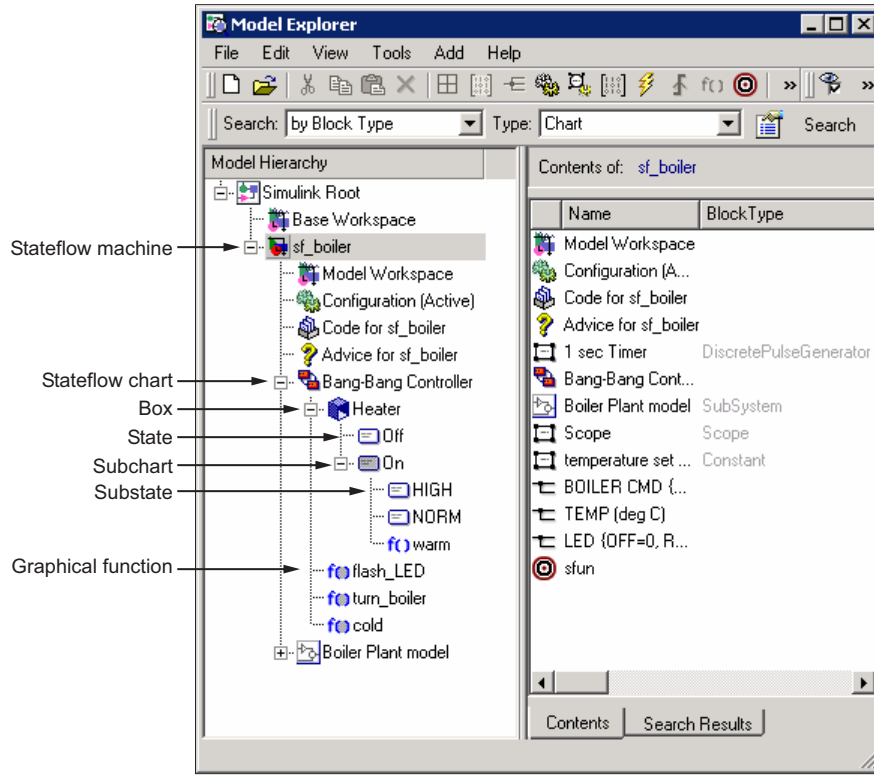
Visibility of Data You Add in the Model Explorer

In the Model Explorer, you can add data that is visible at these levels in the Stateflow hierarchy:

- Stateflow machine
- Stateflow chart
- Box
- State
- Subchart
- Substate
- Function

  Stateflow charts can contain graphical, truth table, and Embedded MATLAB functions.

Stateflow objects that can parent data in the model hierarchy appear in this diagram.

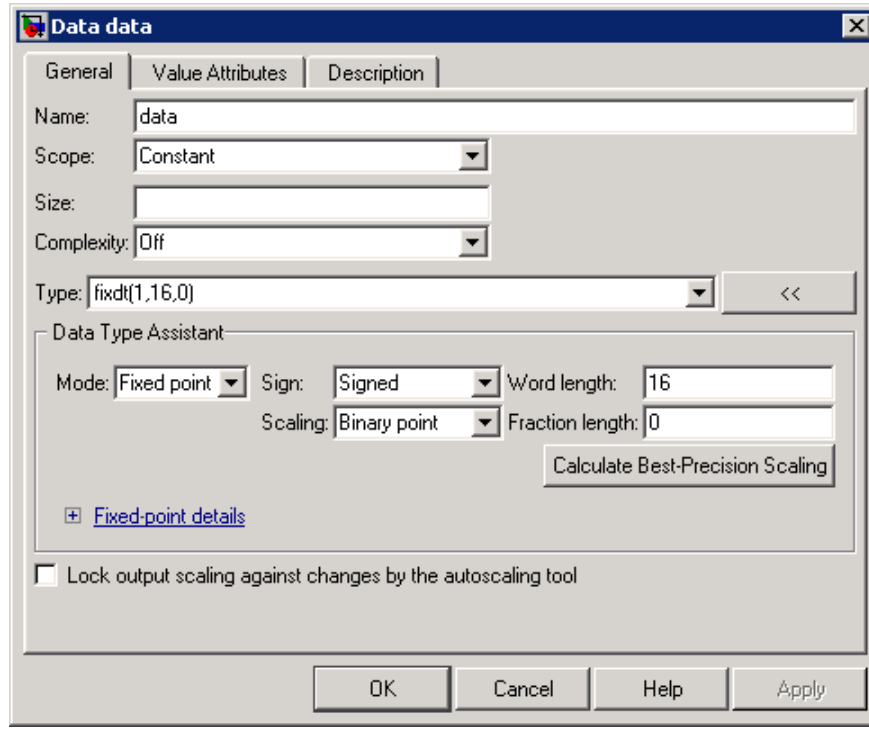


Setting Data Properties in the Data Dialog Box

In this section...

What Is the Data Properties Dialog Box?

You use the Data properties dialog box to set and modify the properties of data objects. Properties vary according to the scope and type of the data object. The Data properties dialog box displays only the property fields relevant to the data object you are defining. For example, the dialog box displays these properties and default values for a data object whose scope is `Constant` and type is `Fixed point`.



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.

When to Use the Data Properties Dialog Box

- Use the **General** pane to define the name, scope, size, complexity, or type of a data object. See “Properties You Can Set in the General Pane” on page 8-8.
- Use the **Value Attributes** pane to set an initial value, limit range, and index into a data object array. See “Properties You Can Set in the Value Attributes Pane” on page 8-20.

- Use the **Description** pane to enter a data description and link to documentation about the data object. See "Properties You Can Set in the Description Pane" on page 8-23.

Opening the Data Properties Dialog Box

To open the Data properties dialog box, use one of these methods:

- Add a new data object in the Stateflow Editor, as described in "Adding Data Using the Stateflow Editor" on page 8-2.

  After you add the data object, the Data properties dialog box appears.

- Open the Data properties dialog box from the Model Explorer for a data object that already exists in the Stateflow hierarchy. Use one of these techniques:

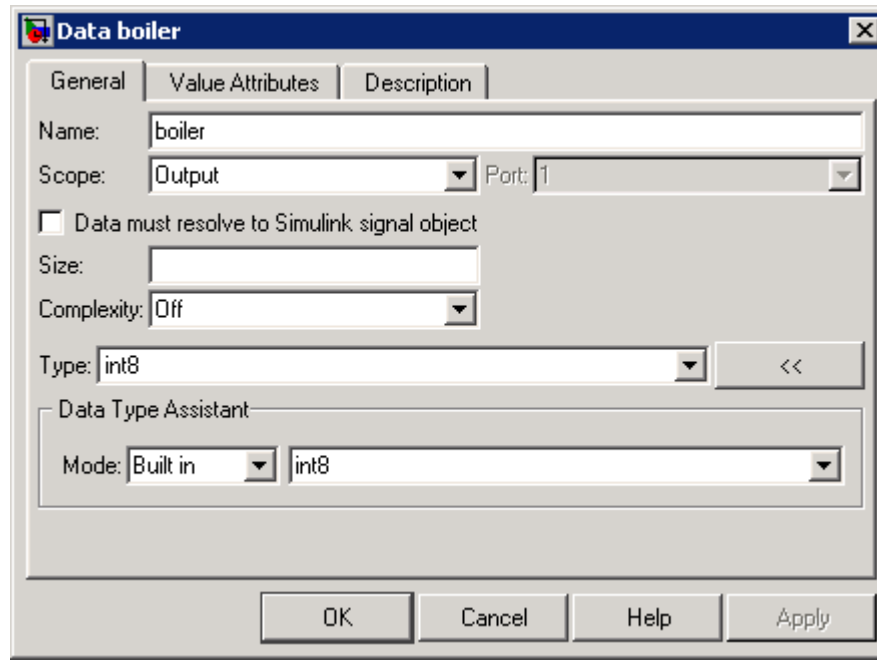
  - Double-click the data object in the **Contents** pane.
  - Right-click the data object in the **Contents** pane and select **Properties**.
  - Select the data object in the **Contents** pane and then select **View** > **Dialog View**.

    The Data properties dialog box opens inside the Model Explorer.

  For more information about adding data objects in the Model Explorer, see "Adding Data Using the Model Explorer" on page 8-3.

Properties You Can Set in the General Pane

The **General** pane of the Data properties dialog box appears as shown.



You can set these properties in the **General** pane.

Name

Name of the data object. Name length should comply with the maximum identifier length enforced by Real-Time Workshop code generation software. You can set this parameter in the **Symbols** pane of the Configuration Parameters dialog box (see "Maximum identifier length" in the Real-Time Workshop Reference documentation). The default length is 31 characters and the maximum length you can specify is 256 characters. The name can consist of any combination of alphanumeric and special characters; however, it cannot begin with a numeric character or contain embedded spaces.

Scope

Location where data resides in memory, relative to its parent. You can set scope to one of these values:

Scope Value	Description
Local	Data defined in the current Stateflow 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 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 parameter. See "Mask Editor" in Simulink software documentation for information on how to assign a parameter to a masked subsystem. See "Sharing Simulink Parameters with Stateflow Charts" on page 8-29 to learn how to use Simulink block parameters with Stateflow charts.
Input	Input argument to a function if the parent is a graphical, truth table, or Embedded MATLAB function. Otherwise, the Simulink model provides the data to the Stateflow chart via an input port on the Stateflow block. See "Sharing Input and Output Data with Simulink Models" on page 8-27.
Output	Return value of a function if the parent is a graphical, truth table, or Embedded MATLAB function. Otherwise, the Stateflow chart provides the data to the Simulink model via an output port on the Stateflow block. See "Sharing Input and Output Data with Simulink Models" on page 8-27.
Data Store Memory	Data object that binds to a Simulink data store, which is a signal that functions like a global variable because all blocks in a model can access that signal. This binding allows the Stateflow chart to read and write the Simulink data store, thereby sharing global data with the model. The Stateflow object must have the same name as the Simulink data store. See "Sharing Global Data with Simulink Models" on page 8-32.

Scope Value	Description
Temporary	Data that persists only during the execution of a function. You can define temporary data only for a graphical, truth table, or Embedded MATLAB function, as described in "Defining Temporary Data" on page 8-53.
Exported	Data from the Simulink model that is made available to external code defined in the Stateflow hierarchy, as described in "Sharing Stateflow Data with External Modules" on page 8-39. You can define exported data only for a Stateflow machine.
Imported	Data parented by the Simulink model that is defined by external code embedded in the Stateflow machine, as described in "Sharing Stateflow Data with External Modules" on page 8-39. 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 "Sharing Input and Output Data with Simulink Models" on page 8-27.

Data must resolve to Simulink signal object

Option that 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
- Minimum value
- Maximum value
- Initial value

- Storage class (in Real-Time Workshop generated code)
- Sampling mode (for Truth Table block output data)

For more information, see “Resolving Data Properties from Simulink Signal Objects” on page 8-55.

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, where the number of dimensions equals the length of the vector and the size of each dimension 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 of its properties — including 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 “Sizing Stateflow Data” on page 8-50.

Complexity

Option that specifies whether or not the data object accepts complex values. You can choose one of these settings:

Complexity 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.

For more information, see “How Complex Data Works in Stateflow Charts” on page 15-2.

Type

Type of data object. You can specify the data type by:

- Selecting a built-in type from the **Type** drop-down list.
- Using the Data Type Assistant to specify a data **Mode** and then specifying the data type based on that mode.

Note Click the Show data type assistant button  to display the Data Type Assistant.

- Entering an expression in the **Type** field that evaluates to a data type.

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 message appears when you try to simulate your model.

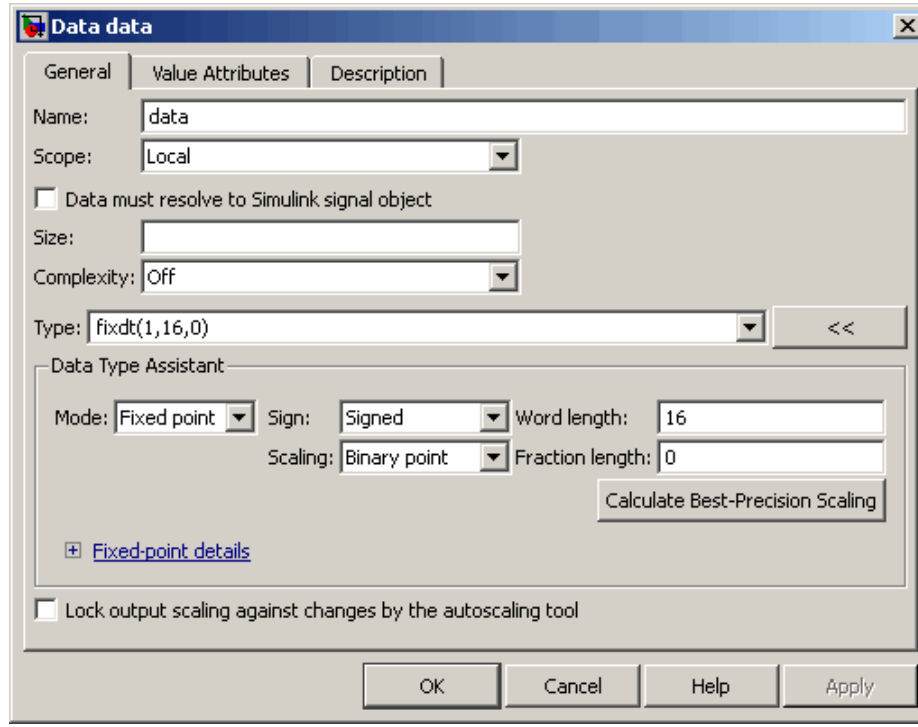
To ensure that a data type definition is valid for fixed-point data, use one of the two options above.

For more information, see "Typing Stateflow Data" on page 8-42.

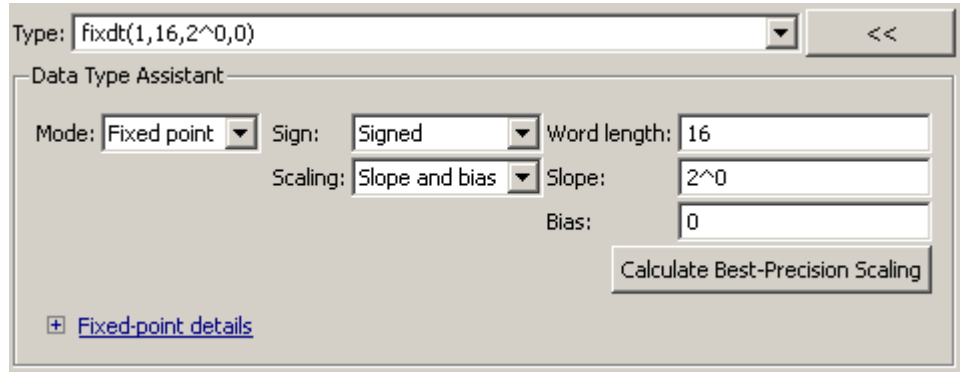
Fixed-Point Data Properties

Properties that apply only to fixed-point data. For a detailed discussion about fixed-point data, see "Fixed-Point Concepts" in the *Simulink® Fixed Point™ User's Guide*.

When the Data Type Assistant **Mode** is `Fixed point`, the Data Type Assistant displays fields for specifying additional information about your fixed-point data.



If the **Scaling** is `Slope and bias` rather than `Binary point`, the Data Type Assistant displays a **Slope** field and a **Bias** field rather than a **Fraction length** field.

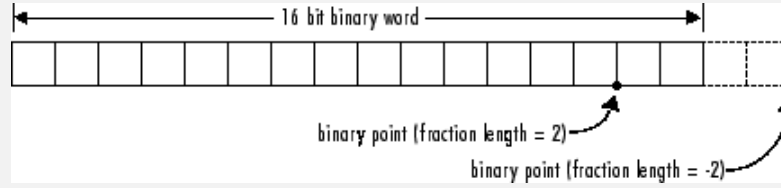


You can use the Data Type Assistant to set these fixed-point properties:

Sign. Specify whether you want the fixed-point data to be `Signed` or `Unsigned`. Signed data can represent positive and negative values, but unsigned data represents positive values only. The default setting is `Signed`.

Word length. Specify the bit size of the word that will hold the quantized integer. Large word sizes represent large values with greater precision than small word sizes. Word length can be any integer between 0 and 32. The default bit size is 16.

Scaling. Specify the method for scaling your fixed-point data to avoid overflow conditions and minimize quantization errors. The default method is `Binary point` scaling. You can select one of two scaling modes:

Scaling Mode	Description
`Binary point`	If you select this mode, the Data Type Assistant displays the **Fraction Length** field, which specifies the binary point location. Binary points can be positive or negative integers. A positive integer moves the binary point left of the rightmost bit by that amount. For example, an entry of 2 sets the binary point in front of the second bit from the right. A negative integer moves the binary point further right of the rightmost bit by that amount, as in this example:  The default binary point is 0.
`Slope and bias`	If you select this mode, the Data Type Assistant displays fields for entering the **Slope** and **Bias**. Slope can be any positive real number, and the default slope is 1.0. Bias can be any real number, and the default bias is 0.0. You can enter slope and bias as expressions that contain parameters you define in the MATLAB workspace.

Note Use binary-point scaling whenever possible to simplify the implementation of fixed-point data in generated code. Operations with fixed-point data using 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" in the *Simulink Fixed Point User's Guide*.

Calculate Best-Precision Scaling. Click this button to calculate "best-precision" values for both `Binary point` and `Slope and bias` scaling, based on the Limit range properties you specify in the **Value Attributes** pane of the Data properties dialog box.

To automatically calculate best precision scaling values:

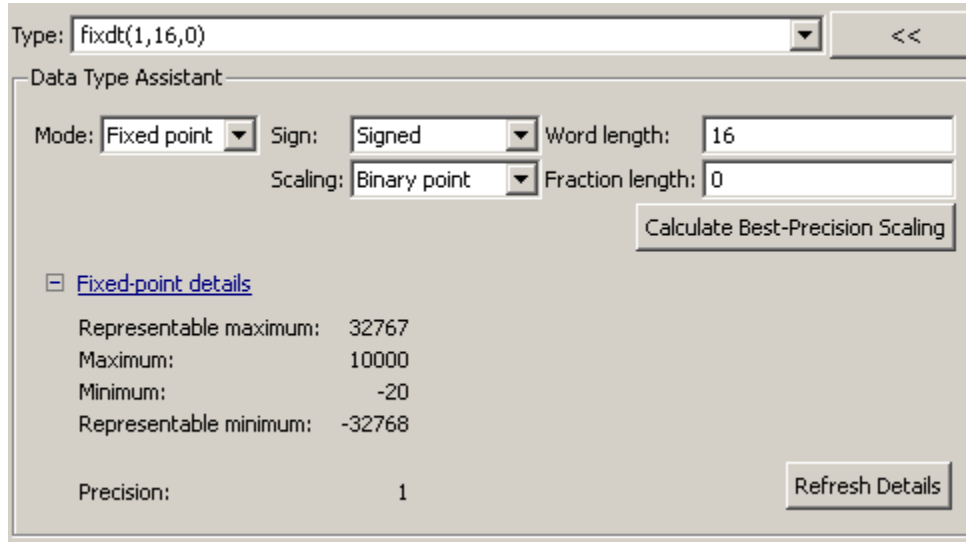
1. In the Data properties dialog box, select the **Value Attributes** tab.
2. Specify **Limit range** properties.
3. Select the **General** tab.
4. Select the option **Calculate Best-Precision Scaling**.

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" in the *Simulink Fixed Point User's Guide*.

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 the `Parameter` scope.

Lock output scaling against changes by the autoscaling tool. Check this box to prevent a Simulink model from replacing the current fixed-point type with a type that the autoscaling tool chooses. See "Automatic Scaling" in the *Simulink Fixed Point User's Guide* for instructions on autoscaling fixed-point data.

Showing Fixed-Point Details. When you specify a fixed-point data type, you can use the **Fixed-point details** subpane to see information about the fixed-point data type that is currently defined in the Data Type Assistant. To see the subpane, click the expander next to **Fixed-point details** in the Data Type Assistant. The **Fixed-point details** subpane appears at the bottom of the Data Type Assistant.



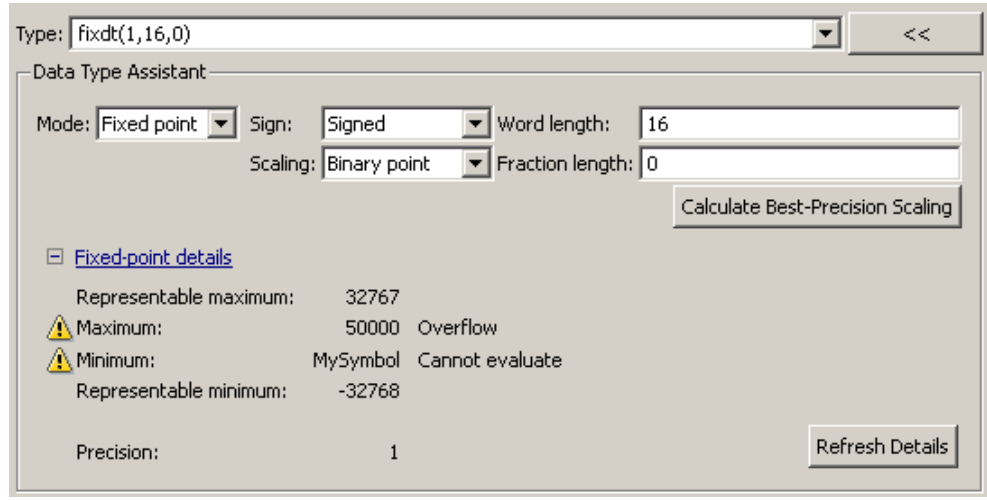
The rows labeled `Minimum` and `Maximum` show the same values that appear in the corresponding **Minimum** and **Maximum** fields in the **Value Attributes** pane of the Data properties dialog box. See "Checking Signal Ranges" and "Checking Parameter Values" for more information.

The rows labeled `Representable minimum`, `Representable maximum`, and `Precision` show the minimum value, maximum value, and precision that can be represented by the fixed-point data type currently displayed in the Data Type Assistant. See "Fixed-Point Concepts" in the *Simulink Fixed Point User's Guide* for information about these three quantities.

The values displayed by the **Fixed-point details** subpane *do not* automatically update if you click **Calculate Best-Precision Scaling**, or change the range limits, the values that define the fixed-point data type, or anything elsewhere in the model. To update the values shown in the **Fixed-point details** subpane, click **Refresh Details**. The Data Type Assistant then updates or recalculates all values and displays the results.

Clicking **Refresh Details** does not change anything in the model; it changes only the display. Click **OK** or **Apply** to put the displayed values into effect. If the value of a field cannot be known without first compiling the model, the **Fixed-point details** subpane shows the value as `Unknown`. If any errors

occur when you click **Refresh Details**, the **Fixed-point details** subpane shows an error flag on the left of the applicable row, and a description of the error on the right. . For example, the next figure shows two errors.



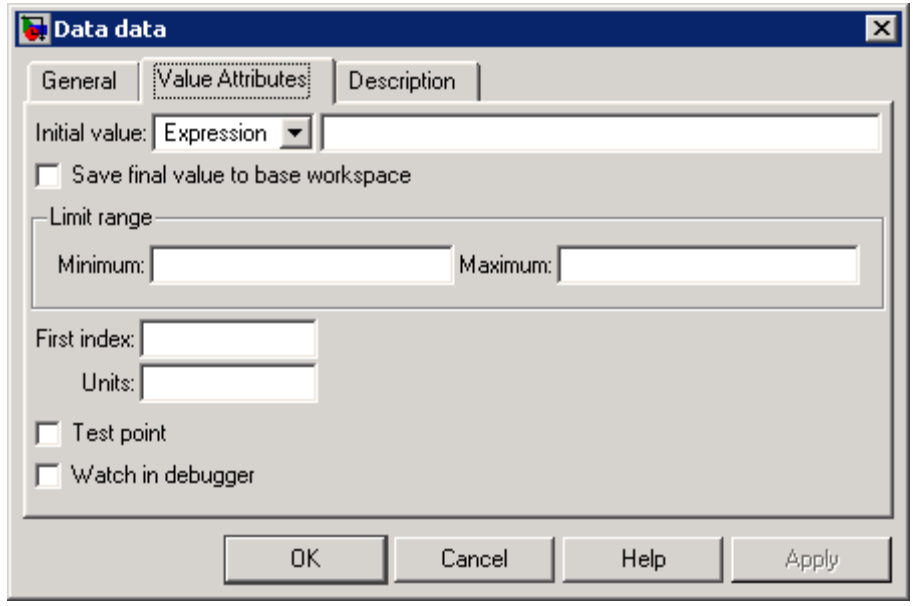
The row labeled `Minimum` shows the error `Cannot evaluate` because evaluating the expression `MySymbol`, specified in the **Minimum** field of the **Value Attributes** pane, did not return an appropriate numeric value. When an expression does not evaluate successfully, the **Fixed-point details** subpane displays the unevaluated expression (truncating to 10 characters if necessary to save space) in place of the unavailable value.

To correct this error, you would need to define `MySymbol` in an accessible workspace to provide an appropriate numeric value. After you clicked **Refresh Details**, the value of `MySymbol` would appear in place of its unevaluated text, and the error indicator and error description would disappear.

To correct the overflow error shown for `Maximum`, you would need to decrease the value in the **Maximum** field of the **Value Attributes** pane, increase **Word length**, or decrease **Fraction length** (or perform a combination of these changes) sufficiently to allow the fixed-point data type to represent the maximum value that it could have.

Properties You Can Set in the Value Attributes Pane

The **Value Attributes** pane of the Data properties dialog box appears as shown.



You can set these properties in the **Value Attributes** pane.

Initial value

Initial value of the data object. If you do not specify a value, the default is 0.0. The options for initializing values depend on the scope of the data object, as follows:

Scope	What to Specify for Initial Value
Local	Expression or parameter defined in the Stateflow hierarchy, MATLAB workspace, or Simulink masked subsystem

Scope	What to Specify for Initial Value
Constant	Constant value or expression. The expression is evaluated when you update the chart, and the resulting value is used as a constant for running the Stateflow 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 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 is bound.

For more information, see "Initializing Data from the MATLAB Base Workspace" on page 8-29 and "Sharing Simulink Parameters with Stateflow Charts" on page 8-29.

Save final value to base workspace

Option that assigns the value of the data item to a variable of the same name in the model workspace at the end of simulation (see "Using Model Workspaces" in the Simulink software documentation).

Limit range properties

Range of acceptable values for this data object. Stateflow software uses this range to validate the data object during simulation. To establish the range, specify these properties:

- **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.

- **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.

The largest value you can set for **Maximum** is `inf`, and the smallest value you can set for **Minimum** is `-inf`.

Note A Simulink model uses the **Limit range** properties to calculate best-precision scaling for fixed-point data types. You must specify a maximum or minimum value before you can select the **Calculate Best-Precision Scaling** option in the **General** pane. For more information, see "Scaling" on page 8-15.

For more information on entering values for **Limit range** properties, see "Entering Expressions and Parameters for Data Properties" on page 8-24.

First index

Index of the first element of the data array. The default value is `0`.

Units

Units of measurement that you want to associate with the data object. The string in this field resides with the data object in the Stateflow hierarchy.

Test point

Option that designates the data object as a test point. Enabling this option guarantees that you can observe the data object during simulation (see "Working with Test Points" in Simulink software documentation). Data objects can be test points if:

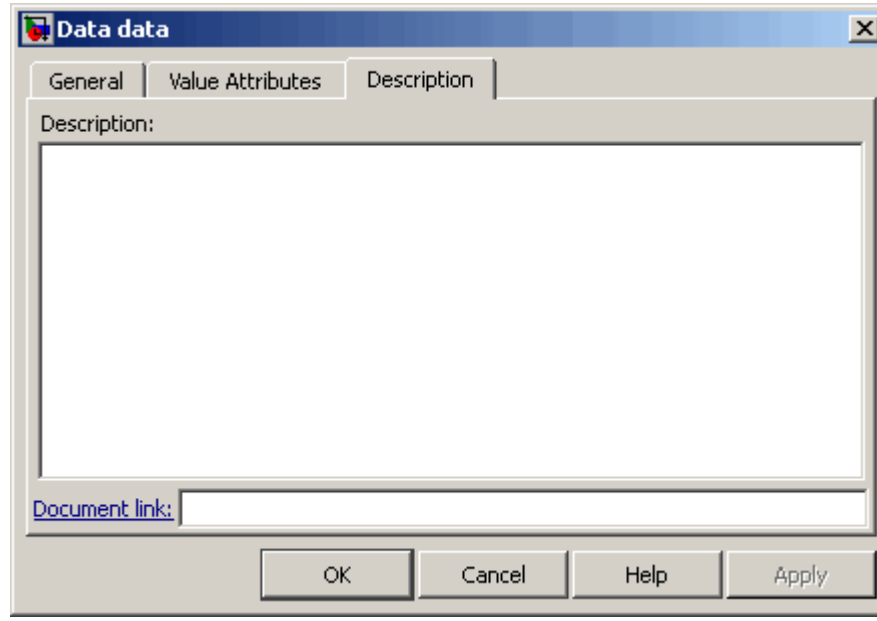
- Scope is **Local**
- Parent is not a Stateflow machine
- Data type is not **ml**

Watch in Stateflow Debugger

Option that enables you to watch the data values in the Stateflow Debugger (see “Watching Data in the Stateflow Debugger” on page 23-30).

Properties You Can Set in the Description Pane

The **Description** pane of the Data properties dialog box appears as shown.



You can set these properties in the **Description** pane.

Description

Description of the data object.

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 at the bottom of the

properties dialog box, Stateflow software evaluates the link and displays the documentation.

Entering Expressions and Parameters for Data Properties

You can enter expressions as values for these properties in the Data properties dialog box:

- "Size" on page 8-12
- "Type" on page 8-13
- "Initial value" on page 8-20
- Minimum and Maximum (see "Limit range properties" on page 8-21)
- "Fixed-Point Data Properties" on page 8-13

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 software assumes a default value, as follows:

Field	Default
Initial value	0.0
Maximum	`inf`
Minimum	`-inf`
Word length	16
Slope	1.0
Bias	0.0
Binary point	0

Field	Default
First index	0
Size	• −1 (inherited), for inputs, parameters, and Embedded MATLAB function outputs • `scalar`, for all other data objects

Using Parameters in Expressions

You can include parameters in expressions. A parameter is a constant that you can:

- Define in the MATLAB workspace (see "Initializing Data from the MATLAB Base Workspace" on page 8-29)
- Derive from a Simulink block parameter that you define and initialize in the parent masked subsystem (see "Sharing Simulink Parameters with Stateflow Charts" on page 8-29)

You can mix both types of parameters in an expression.

Using Constants in Expressions

You can use two types of constants in expressions in the Data properties dialog box:

- Numeric constants of the appropriate type and size
- Stateflow constants

  Stateflow constants are read-only data objects that you add to your chart with the scope **Constant** (see "Adding Data" on page 8-2). Stateflow constants retain their initial values, which you set in the Data properties dialog box (see "Initial value" on page 8-20).

Using Arithmetic Operators in Expressions

You can use these arithmetic operators in expressions in the Data properties dialog box:

- +

- –
- *
- /

Calling Functions in Expressions

In fields that accept expressions, you can call functions that return property values of other variables defined in the Stateflow hierarchy, MATLAB workspace, or Simulink masked subsystem. For example, these functions can return appropriate values for specified fields in the Data properties dialog box:

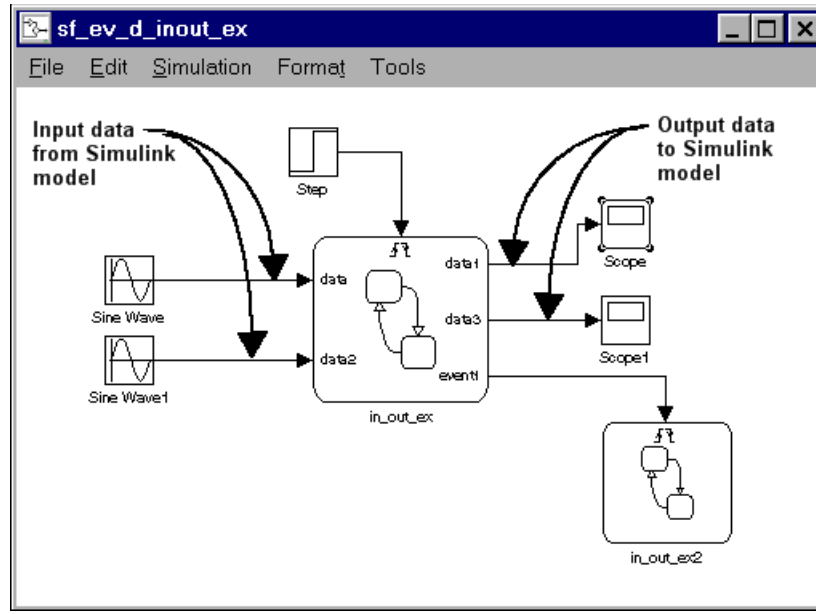
Function	Returns	For Field
MATLAB function `size`	Size of input array	Size
Stateflow function `type`	Type of input data	Data type
MATLAB function `min`	Smallest element or elements of input array	Minimum
MATLAB function `max`	Largest element or elements of input array	Maximum
Simulink function `fixdt`	`Simulink.NumericType` object that describes a fixed-point or floating-point data type	Data type

Sharing Data with Simulink Models and the MATLAB Workspace

In this section...

Sharing Input and Output Data with Simulink Models

Data flows between Simulink models and Stateflow charts via input ports and output ports on the Stateflow chart block. The following example shows a Stateflow chart block with input and output data ports connected to the Simulink model.



To add input or output data to a Stateflow chart, follow these steps:

1 Add a data object to the Stateflow chart, as described in "Adding Data Using the Stateflow Editor" on page 8-2.

Note You must add the data to the Stateflow chart, not to any other object in the chart.

---

2 Open the Data properties dialog box, as described in "Opening the Data Properties Dialog Box" on page 8-8.

3 Set the **Scope** property to one of these values:

- **Input**

  This setting is the same as **Input from Simulink** in the **Add > Data** menu in the Stateflow Editor. A Simulink input port appears on the Stateflow chart block in the model.

- **Output**

  This setting is the same as **Output to Simulink** in the **Add > Data** menu in the Stateflow Editor. A Simulink output port appears on the Stateflow chart block in the model.

You assign inputs and outputs to ports in the order in which you add the data. For example, you assign the first input to input port 1 and the third output to output port 3. You can change port assignments by editing the value in the **Port** field of the Data properties dialog box.

4 Set the type of the input or output data, as described in "Typing Stateflow Data" on page 8-42.

5 Decide if you want to use strong data typing with the Simulink model, as described in "Strong Data Typing with Simulink I/O" on page 8-49.

6 Set the size of the input or output data, as described in "Sizing Stateflow Data" on page 8-50.

Note You cannot type or size Stateflow input data to accept frame-based data from a Simulink model.

Sharing Simulink Parameters with Stateflow Charts

When to Share Simulink Parameters

Share Simulink parameters with Stateflow charts to maintain consistency with your Simulink model.

How to Share Simulink Parameters

To share Simulink parameters for a masked subsystem with a Stateflow chart, follow these steps:

1. In the Simulink mask editor for the parent subsystem, define and initialize a Simulink parameter (see "Mask Editor" in the Simulink software documentation).
2. In the Stateflow hierarchy, define a data object with the same name as the parameter (see "Adding Data" on page 8-2).
3. Set the scope of the data object to **Parameter**.

   A Stateflow chart defines data of scope **Parameter** as a constant. You cannot change a parameter value during model execution.

When simulation starts, Simulink software attempts to resolve the Stateflow data object to a parameter at the lowest level masked subsystem. If unsuccessful, Simulink software moves up the model hierarchy to resolve the data object to a parameter at higher level masked subsystems.

Initializing Data from the MATLAB Base Workspace

You can initialize data from the MATLAB base workspace. Initialization requires that you define data in both the MATLAB base workspace and the Stateflow hierarchy as follows:

1. Define and initialize a variable in the MATLAB workspace.
2. In the Stateflow hierarchy, define a data object with the same name as the MATLAB variable (see "Adding Data" on page 8-2).
3. Set the scope of the Stateflow data object to **Parameter**.

When simulation starts, data resolution occurs. During this process, the Stateflow data object gets its initial value from the associated MATLAB variable. For example, if the variable is an array, each element of the Stateflow array initializes to the same value as the corresponding element of the MATLAB array.

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]`.

Time of Initialization

Data parent and scope control initialization time for Stateflow data objects.

Data Parent	Scope	When Initialized
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 Embedded MATLAB functions)	Input, Output	Function-call invocation
	Local	Start of simulation or when chart reinitializes as part of an enabled Simulink subsystem

Saving Data to the MATLAB Workspace

For all scopes except **Constant** and **Parameter**, you can instruct the Stateflow chart to save the final value of a data object at the end of simulation in the MATLAB base workspace (not as a masked subsystem parameter).

Use one of these techniques:

- In the **Value Attributes** pane of the Data properties dialog box, select the check box **Save final value to base workspace**.
- In the **Contents** pane of the Model Explorer, follow these steps:
  1. Select the row of the data object.
  2. Select the check box in the **SaveToWorkspace** column.

Sharing Global Data with Simulink Models

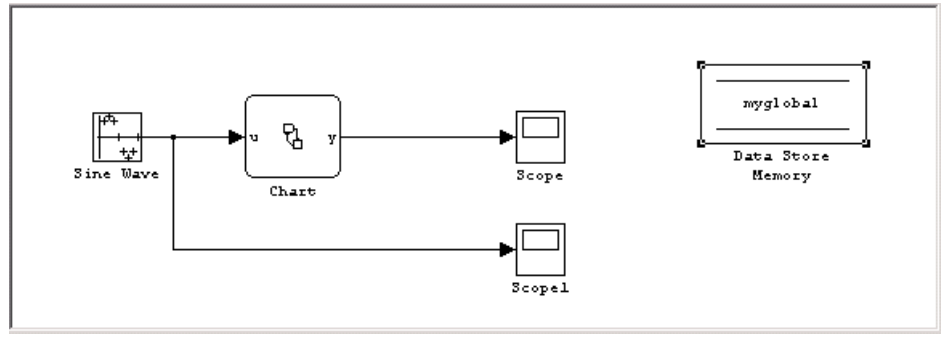
In this section...

About Data Stores

You can use an interface to direct Stateflow charts to access global variables in Simulink models. A Simulink model implements global variables as *data stores*, created either as data store memory blocks or as instances of `Simulink.Signal` objects. Data stores enable multiple Simulink blocks to share data without the need for explicit I/O 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 using Stateflow action language.

How Stateflow Charts Work with Local and Global Data Stores

Stateflow charts can interface with local and global data stores. Local data stores, often implemented as data store memory blocks, are visible to all blocks in one model. To interact with local data stores, a Stateflow chart must reside in the model where you define the local data store, as shown below.



Global data stores have a broader scope, which crosses model reference boundaries. To interact with global data stores, a Stateflow chart must reside either in the top model — where the global data store is defined — or in any model that the top model references. You implement global data stores as Simulink signal objects.

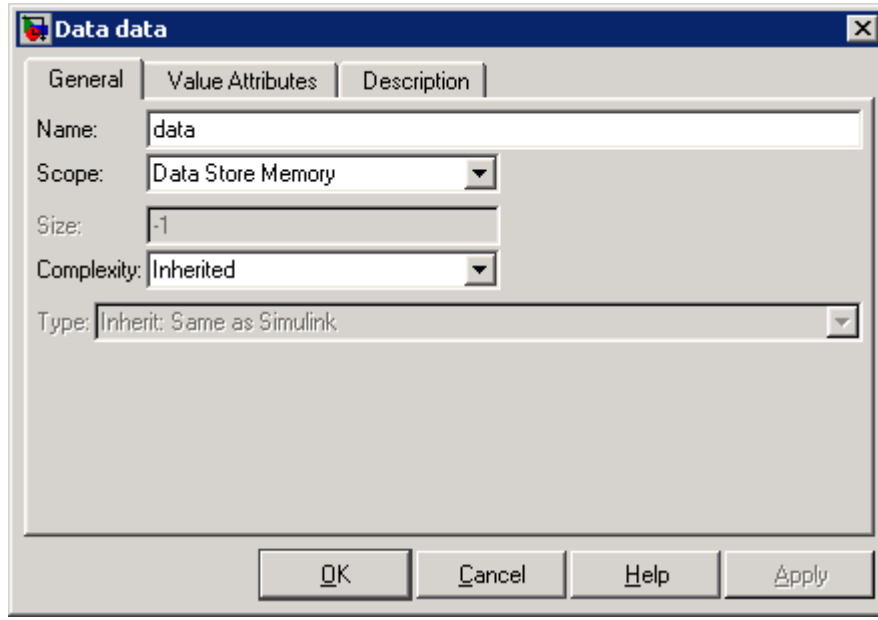
Accessing Data Store Memory from a Stateflow Chart

To access global data in a Simulink model from a Stateflow chart, you must bind a Stateflow data object to a Simulink data store — either a data store memory block or a signal object (see "Binding a Stateflow Data Object to Data Store Memory" on page 8-33). After you create the binding, the Stateflow data object becomes a symbolic representation of Simulink data store memory. You can then use this symbolic object to store and retrieve global data using Stateflow action language (see "Reading and Writing Global Data Programmatically" on page 8-35).

Binding a Stateflow Data Object to Data Store Memory

To bind a Stateflow data object to Simulink data store memory, you must create a data object in the Stateflow hierarchy with the same name as the data store and with scope set to `Data Store Memory`. The Stateflow data object inherits all properties from the data store to which you bind the object. Follow guidelines for specifying data store properties in "Tips for Using Data Stores in Stateflow Charts" on page 8-60.

This properties dialog box shows a Stateflow data object that you bind to a data store.



Note You cannot edit properties that the data object inherits from the data store.

Using the Stateflow Editor to Bind a Data Object

In the Stateflow Editor, follow these steps:

1. Select **Add** > **Data** > **Data Store Memory**.

   The properties dialog box for the new data object appears with scope property set to **Data Store Memory**.

2. In the **Name** field of the Data properties dialog box, enter the name of the Simulink data store to which you want to bind.

3. Click **OK**.

Using the Model Explorer to Bind a Data Object

To use the Model Explorer, follow these steps:

1. In the Stateflow Editor, select **Tools** > **Explore**.

   The Model Explorer appears.

2. In the Model Explorer, select **Add** > **Data**.

   The Model Explorer adds a data object to the Stateflow chart.

3. Double-click the new data object to open its properties dialog box, and enter the following information in the **General** pane:

Field	What to Specify
Name	Enter the name of the Simulink data store memory block to which you want to bind.
Scope	Select `Data Store Memory` from the drop-down menu.

4. Click **OK**.

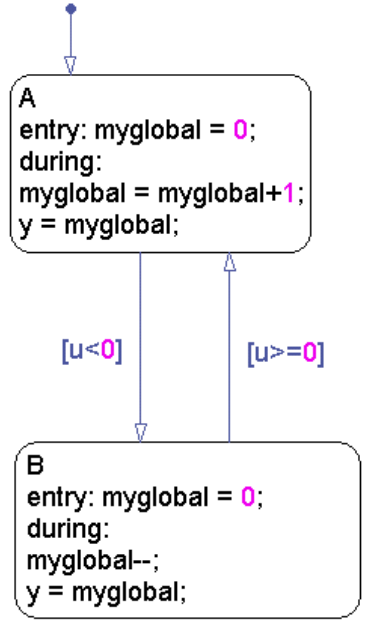
Resolving Data Store Bindings

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.

Reading and Writing Global Data Programmatically

You can use the Stateflow data object that you bind to Simulink data store memory to store and retrieve global data in states and transitions using Stateflow action language. Think of this object as a global variable that you reference by its symbolic name — the same name as the data store to which you bind the object. When you store numeric values in this variable, you are writing to Simulink data store memory. Similarly, when you retrieve numeric values from this variable, you are reading from the data store memory.

This example of Stateflow action language reads from and writes to a data store memory block called `myglobal`.



Diagnostics for Sharing Data Between Stateflow Charts and Simulink Blocks

Errors to Check For

Multiple reads and writes can occur unintentionally in the same time step. To detect these situations, you can configure data store memory blocks to generate errors or warnings for these conditions:

- Read before write
- Write after write
- Write after read

Note These diagnostics are available only for data store memory blocks used within a single Simulink model, not for data stores created from Simulink signal objects. In other words, these diagnostics do not work for global data stores that cross model reference boundaries.

When to Enable Diagnostics

Enable diagnostics on data store memory blocks to ensure the validity of data that multiple unconnected blocks share while running at different rates. In this scenario, you can detect conditions when writes do not occur before reads in the same time step. To prevent these violations, see "Tips for Using Data Stores in Stateflow Charts" on page 8-60.

When to Disable Diagnostics

If you use a data store memory block as a persistent global storage area for accumulating values across time steps, disable diagnostics to avoid generating unnecessary warnings.

How to Set Diagnostics for Shared Data

To set diagnostics on data store memory blocks, follow these steps:

1. Double-click the data store memory block in your Simulink model to open its Block Parameters dialog box.
2. Click the **Diagnostics** tab.
3. Enable diagnostics by selecting `warning` or `error` from the drop-down menu for each condition you want to detect.
4. Click **OK**.

Sharing Data Between Charts and with External Modules

In this section...

Sharing Data Between Charts in a Stateflow Machine

You can share data between Stateflow charts in a single Stateflow machine by:

- Defining data store memory objects that are parented by each Stateflow *chart* that wants to share the data.
- Defining local data that are parented by the Stateflow *machine*

Sharing Data Store Memory Between Charts in a Stateflow Machine

You can use data store memory objects to share data between selected charts in a single Stateflow machine. Follow these steps:

1. Use the Model Explorer to add a data object to each Stateflow chart that wants to share the data, as described in "Adding Data Using the Model Explorer" on page 8-3.
2. Give each data object the same name.
3. Set the scope of each data object to **Data Store Memory**.

Each data store memory object you add represents a common area of memory storage and functions as a global variable. You can extend the use of data store memory objects to share data with Simulink models, as described in "Sharing Global Data with Simulink Models" on page 8-32.

Sharing Local Data Among All Charts in a Stateflow Machine

To share local data among all charts in a single Stateflow machine, follow these steps:

1. Use the Model Explorer to add a data object to the Stateflow machine in which the charts reside, as described in "Adding Data Using the Model Explorer" on page 8-3.

2. Set the scope of the data object to **Local**, as described in "Setting Data Properties in the Data Dialog Box" on page 8-6.

The new data object is visible to all charts in the parent Stateflow machine.

Sharing Stateflow Data with External Modules

A Stateflow machine can share data with external modules, such as Stateflow charts in other machines or external code assigned to the machine. Sharing data requires that a Stateflow machine *export* the data definition to the external module and that the external module *import* the data definition from the Stateflow machine. Similarly, a Stateflow machine can import data that an external module exports.

Exporting Data to External Modules

To export data from the Stateflow machine to external modules, follow these steps:

1. In the Model Explorer, add a data object to the Stateflow *machine*, as described in "Adding Data Using the Model Explorer" on page 8-3.

2. Set the scope of the data to **Exported**.

When You Export Data to External Code Assigned to the Stateflow Machine. For each exported data object, the Stateflow code generator creates a C declaration of the form

```
type data;
```

where `type` is the C type of the exported data object — such as `int16` or `double` — and `data` is the name of the Stateflow object. For example, suppose that your Stateflow machine defines an exported `int16` item named `counter`. The Stateflow code generator exports the item as the C declaration

```
int16_T counter;
```

where `int16_T` is a defined type for `int16` integers in Stateflow charts.

The code generator includes declarations for exported data in the generated target's global header file. This inclusion makes the declarations visible to external code compiled into or linked to the target.

See "Exported Data" on page 16-30 for an example of Stateflow data exported to Stateflow external code.

When You Export Data to an External Stateflow Machine. For each Stateflow machine that wants to share the data exported from the external machine, you must define a data object of the same name as the exported data and set the object scope to **Imported**.

Importing Data from External Modules

To import externally defined data into a Stateflow machine, follow these steps:

1. In the Model Explorer, add a data object to the Stateflow *machine*, as described in "Adding Data Using the Model Explorer" on page 8-3.
2. Give the data object the same name as the external data.
3. Set the scope of the data to **Imported**.

When You Import Data from External Code Assigned to the Stateflow Machine. For each imported data object, the Stateflow code generator assumes that external code provides a prototype of the form

```
type data;
```

where `type` is the C data type corresponding to the Stateflow data type of the imported item — such as `int32` or `double` — and `data` is the name of the Stateflow object. For example, suppose that your Stateflow machine defines an imported `int32` integer named `counter`. The Stateflow code generator expects the item to be defined in the external C code as

```
int32_T counter;
```

See “Imported Data” on page 16-32 for an example of Stateflow external code data imported into the Stateflow machine.

When You Import Data from an External Stateflow Machine. Make sure that the external Stateflow machine contains a data definition of scope **Exported** with the same name as the imported data objects.

Typing Stateflow Data

In this section...

What Is Data Type?

The term *data type* refers to the way computers represent numbers in memory. The 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.

Specifying Data Type and Mode

To specify the *type* of a Stateflow data object:

1. Open the Data properties dialog box, as described in "Opening the Data Properties Dialog Box" on page 8-8.
2. Select the **Scope** of the data object for which you want to set the data type.

   For more information, see "Properties You Can Set in the General Pane" on page 8-8.
3. Click the Show data type assistant button 

.

Note If you know the specific data type you want to use, you can enter the data type directly in the **Type** field, or select it from the **Type** drop-down list, instead of using the Data Type Assistant. For more information, see "Working with Data Types" in the Simulink software documentation.

4 Choose a **Mode** in the Data Type Assistant section of the dialog box.

You can choose from these modes for each scope:

Scope	Data Type Modes				
	Inherit	Built in	Fixed point	Expression	Bus Object
Local		yes	yes	yes	yes
Constant		yes	yes	yes	
Parameter	yes	yes	yes	yes	
Input	yes	yes	yes	yes	yes
Output	yes	yes	yes	yes	yes
Data Store Memory	yes				

5 Based on the mode you select, specify a data type as follows:

Mode	What To Specify
Inherit	You cannot specify a value. You inherit the data type from previously defined data, based on the scope you select for the data object: • If scope is **Input**, you inherit the data type from the Simulink input signal on the designated input port (see "Sharing Input and Output Data with Simulink Models" on page 8-27). • If scope is **Output**, you inherit the data type from the Simulink output signal on the designated output port (see "Sharing Input and Output Data with Simulink Models" on page 8-27). **Note** Avoid inheriting data types from output signals. See "Avoid Inheriting Output Data Properties from Simulink Blocks" on page 8-60. • If scope is **Parameter**, you inherit the data type from the associated parameter, which you can define in a Simulink model or the MATLAB workspace (see "Sharing Data with Simulink Models and the MATLAB Workspace" on page 8-27). • If scope is **Data Store Memory**, you inherit the data type from the Simulink data store to which you bind the data object (see "Sharing Global Data with Simulink Models" on page 8-32).
Built in	Select a data type from the drop-down list of supported data types, as described in "Built-In Data Types" on page 8-46.
Fixed point	Specify the following information about the fixed-point data: • Whether the data is signed or unsigned • Word length • Scaling mode For information on how to specify these fixed-point data properties, see "Fixed-Point Data Properties" on page 8-13.

Mode	What To Specify
Expression	Enter an expression that evaluates to a data type in the **Type** field. You can use these expressions: • Alias type from the MATLAB workspace, as described in "Typing Data by Using an Alias" on page 8-48 • `type` operator to specify the type of previously defined data, as described in "Deriving Data Types from Previously Defined Data" on page 8-47 • `fixdt` function to create a `Simulink.NumericType` object that describes a fixed-point or floating-point data type For more information on how to build expressions in the Data properties dialog box, see "Entering Expressions and Parameters for Data Properties" on page 8-24.
Bus object	In the **Bus object** field, enter the name of a `Simulink.Bus` object to associate with the Stateflow bus object structure. You must define the bus object in the base workspace. If you have not yet defined a bus object, click **Edit** to create or edit a bus object in the Bus Types Editor. **Note** You can also inherit bus object properties from Simulink signals. For more information about Stateflow bus object structures, see Chapter 17, "Working with Structures and Bus Signals in Stateflow Charts".

6 Click **Apply** to save the data type settings.

Built-In Data Types

You can choose from these built-in data types:

Data Type	Description
`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. See "ml Data Type" on page 10-38. **Note** `ml` data cannot have a scope outside the Stateflow hierarchy; that is, it cannot have a scope of **Input to Simulink** or **Output to Simulink**.

Inheriting Data Types from Simulink Objects

Stateflow data objects of scope **Input**, **Output**, **Parameter**, and **Data Store Memory** can inherit their data types from Simulink objects, as follows:

Scope:	Can inherit type from:
Input	Simulink input signal connected to corresponding input port in Stateflow chart
Output	Simulink output signal connected to corresponding output port in Stateflow chart **Note** Avoid inheriting data types from output signals. See "Avoid Inheriting Output Data Properties from Simulink Blocks" on page 8-60.
Parameter	Corresponding MATLAB workspace variable or Simulink parameter in a masked subsystem
Data Store Memory	Corresponding Simulink data store

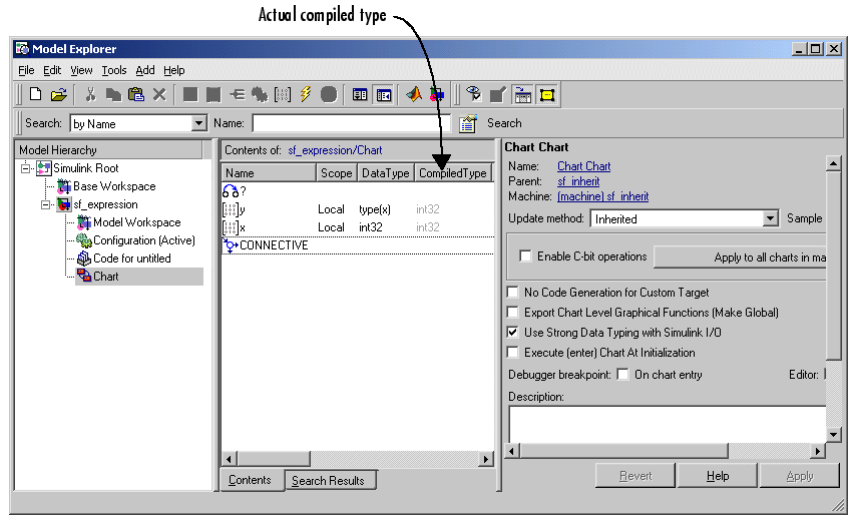
To configure these objects to inherit data types, create the corresponding objects in the Simulink model, and then select `Inherit: Same as Simulink` from the **Type** drop-down list in the Data properties dialog box. For more information, see "Specifying Data Type and Mode" on page 8-42.

To determine the data types that the objects inherit, build the Simulink model and look at the **Compiled Type** column for each Stateflow data object in the Model Explorer.

Note Stateflow blocks in libraries can inherit data types. However, multiple instances of the same library in a model must inherit the same data type.

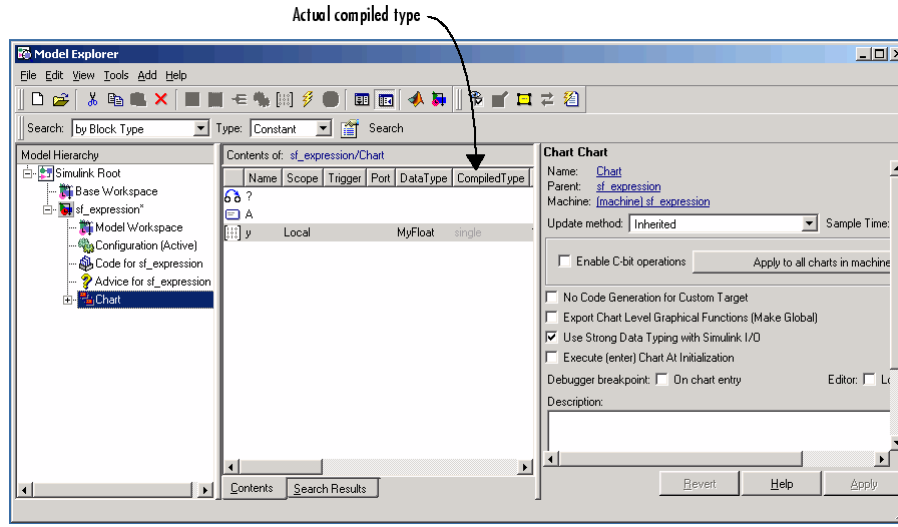
Deriving Data Types from Previously Defined Data

You can use the `type` operator to derive data types from previously defined Stateflow data. In the following example, the Stateflow operator `type(x)` specifies the data type of the Stateflow local data object `y`, where `x` is a local data object of type `int32`. After you build your model, the **Compiled Type** column of the Model Explorer displays the type of each data object in the compiled simulation application.



Typing Data by Using an Alias

You can specify the type of Stateflow data by using a Simulink data type alias (see `Simulink.AliasType` in the Simulink Reference documentation). After you build your model, the **Compiled Type** column of the Model Explorer displays the type used in the compiled simulation application. In the following example, you use the alias `MyFloat`, with `BaseType` set to `single`, to type the Stateflow local data `y`.



Strong Data Typing with Simulink I/O

By default, inputs to and outputs from Stateflow charts are of type `double`. Input signals from Simulink models convert to the type of the corresponding input data objects in Stateflow charts. Likewise, the data output objects convert to `double` before they are exported as output signals to Simulink models.

To interface directly with signals of data types other than `double` without the need for conversion, enable the option **Use Strong Data Typing with Simulink I/O** for the Stateflow chart (see "Specifying Chart Properties" on page 16-5). When you enable this option, the Stateflow chart accepts input signals of any data type that Simulink models support, provided that the type of the input signal matches the type of the corresponding Stateflow input data object. Otherwise, you receive a type mismatch error.

Note For fixed-point data, enable the **Use Strong Data Typing with Simulink I/O** option to flag mismatches between input or output fixed-point data in Stateflow charts and their counterparts in Simulink models.

Sizing Stateflow Data

In this section...

About Stateflow Data Sizes

Stateflow data can be a scalar, vector, or N-dimensional matrix, depending on its scope (see "Size" on page 8-12).

Note Stateflow vectors and matrices use zero-based indexing, unlike MATLAB vectors and matrices, which use one-based indexing.

How to Specify Data Size

You specify the size of Stateflow data as a scalar value or a MATLAB vector of values in the **Size** field of the Data properties dialog box, as described in "Properties You Can Set in the General Pane" on page 8-8.

You can enter size as a constant value or an expression. Stateflow input and output data objects can also inherit their sizes from the Simulink signals that connect to them.

Sizing Data as a Constant Value

In the Data properties dialog box, you specify scalar data by setting the **Size** field to `1` or leaving it blank.

You specify MATLAB vectors and matrices as multidimensional arrays, where the number of dimensions equals the length of the vector, and the size

of each dimension corresponds to the value of each element of the vector. Enter the size of two-dimensional arrays in `[row column]` format.

One-dimensional Stateflow data arrays 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]`. To define a row vector of size 5, set the **Size** field to `[1 5]`. To define a column vector of size 6, set the **Size** field to `[6 1]` or just `6`.

Sizing Data by Expression

You can use a mathematical expression to set the size of Stateflow data. Size expressions must evaluate to a positive integer.

In the **Size** field of the Data properties dialog box, you can enter a MATLAB expression for each dimension. Expressions can contain a mix of numeric constants, Stateflow constants, arithmetic operators, parameters, and calls to functions such as `size`, `min`, and `max`. Valid size expressions include:

```
k+1
size(x)
min(size(y),k)
```

For more information about expressions, see “Entering Expressions and Parameters for Data Properties” on page 8-24.

Note You cannot size Stateflow input data with an expression that accepts frame-based data from Simulink models.

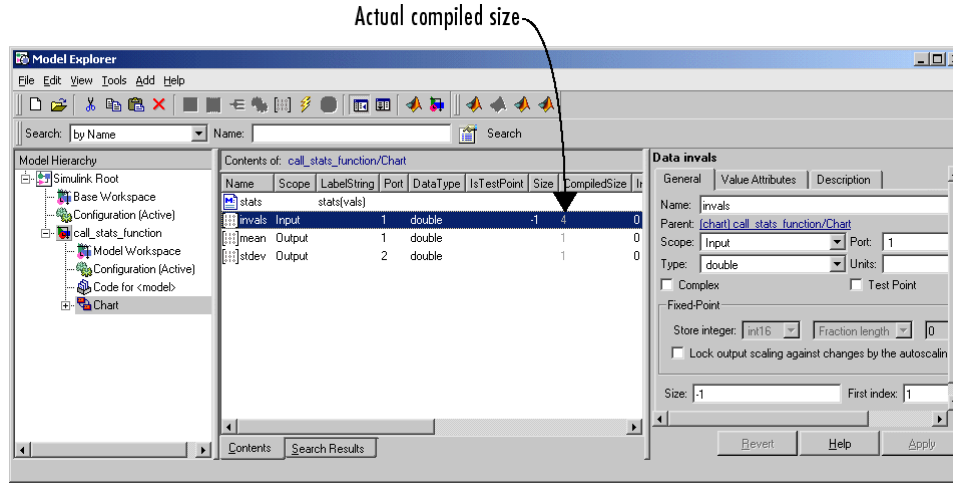
Inheriting Input and Output Data Size from Simulink Signals

To configure Stateflow input and output data objects to inherit size from the corresponding Simulink input and output signals, enter `-1` in the **Size** field of the Data properties dialog box. This default setting applies to input and output data that you add in the Stateflow Editor (see “Adding Data” on page 8-2). After you build your model, the **Compiled Size** column of the Model

Explorer displays the actual size that the compiled simulation application uses.

Note Stateflow blocks in libraries can inherit data sizes. However, multiple instances of the same library in a model must inherit the same data size.

In the following example, the input data `invals` connects to a Constant block that specifies a four-element vector. The **Compiled Size** column displays the correct inherited size of 4 for `invals`.



Inheriting the size of input data is complete for all cases. Chart actions that store values in the specified output infer the inherited size of output data. If the expected size in the Simulink model matches the inferred size, inheritance is successful. Otherwise, a mismatch occurs during build time.

Note Stateflow charts cannot inherit frame-based data sizes from Simulink models.

Defining Temporary Data

In this section...

When to Define Temporary Data

Define temporary data when you want to use data that persists only while a function executes. You can define temporary data in graphical, truth table, and Embedded MATLAB functions. 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:

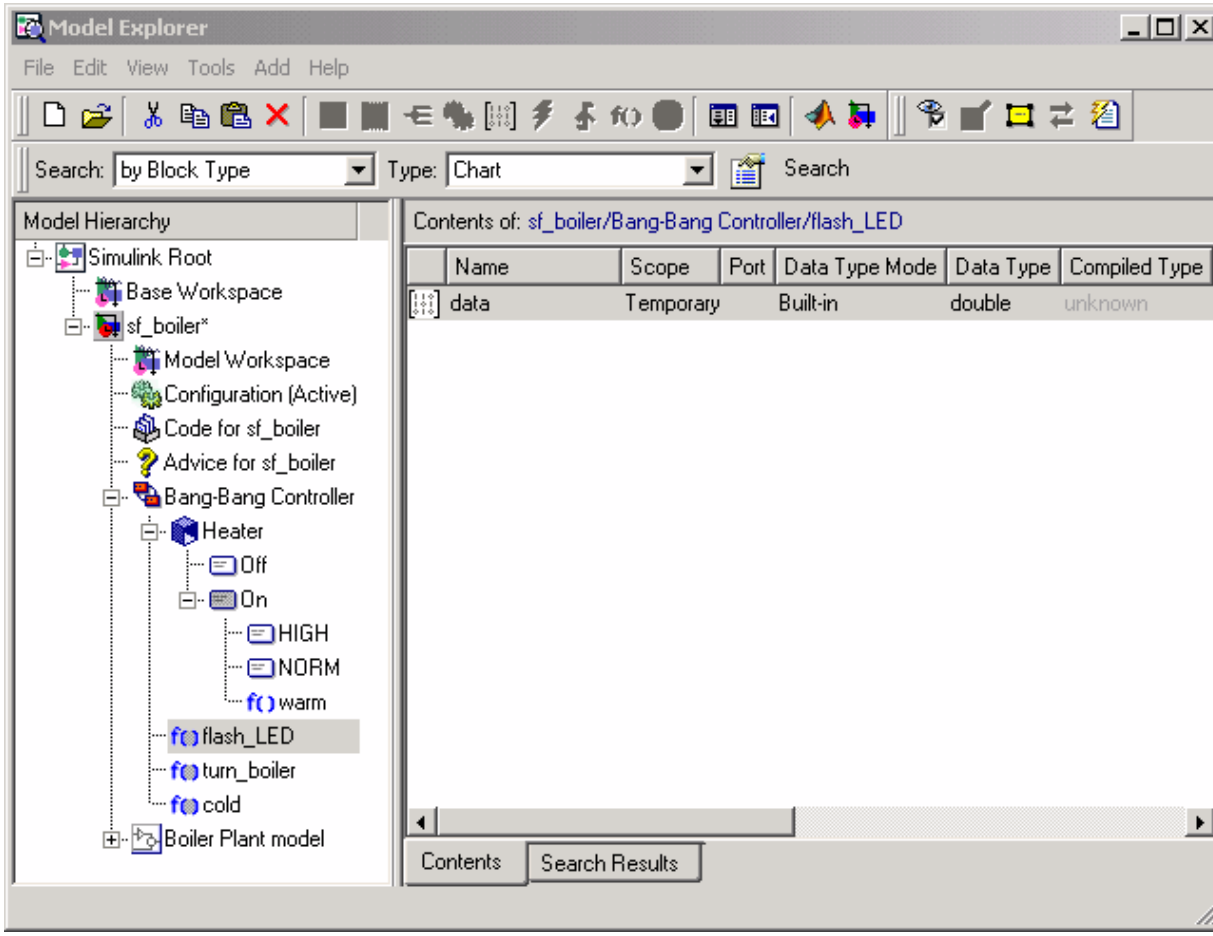
1. In the Stateflow Editor, select **Tools > Explore**.

   The Model Explorer appears.

2. In the **Model Hierarchy** pane of the Model Explorer, select the graphical, truth table, or Embedded MATLAB function that will use temporary data.

3. Select **Add > Data**, or click the **Add Data** button:

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 "Setting Data Properties in the Data Dialog Box" on page 8-6.

Resolving Data Properties from Simulink Signal Objects

In this section...

About Explicit Signal Resolution

Stateflow local and output data in Stateflow 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 "Resolving Symbols" and "Hierarchical Symbol Resolution" in the Simulink documentation.

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 Stateflow chart data in the generated code. See "Working with Data" in the Real-Time Workshop User's Guide.

Enabling Explicit Signal Resolution

To enable explicit signal resolution, follow these steps:

1. In the model workspace or base workspace, define a `Simulink.Signal` object with the properties you want your Stateflow data to inherit.

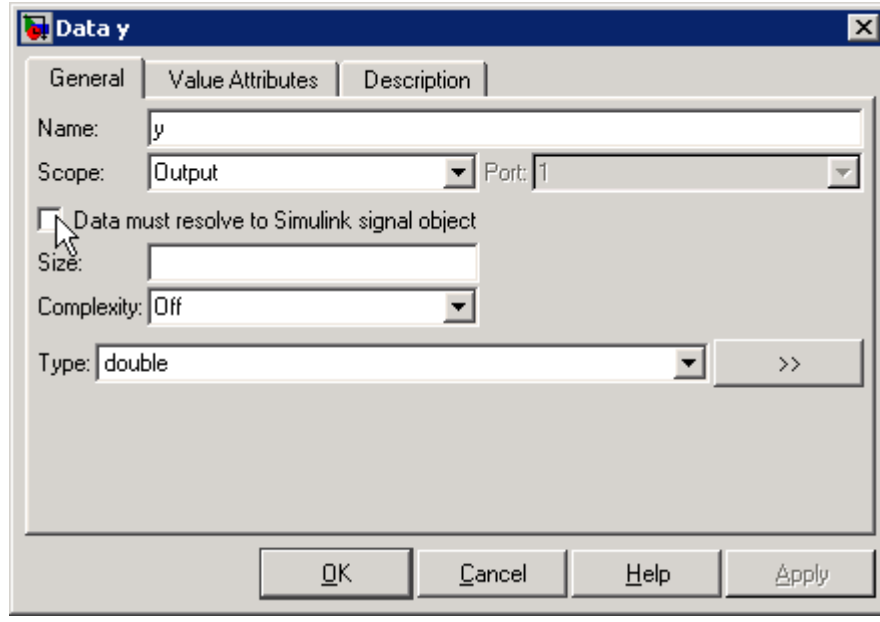
   For more information about creating Simulink signals, see Simulink.Signal in the Simulink Reference documentation.

2. Add output or local data to a Stateflow chart.

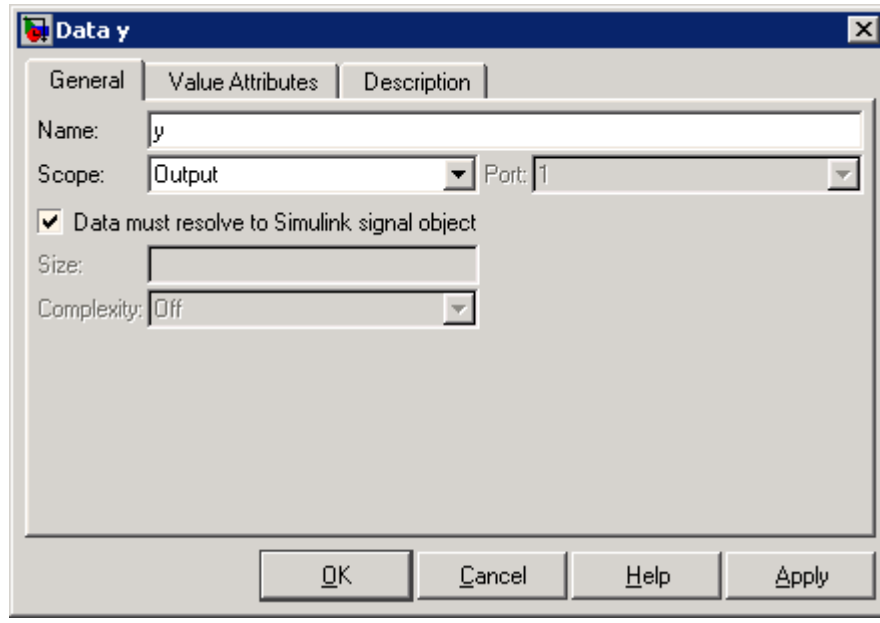
   The Data properties dialog box opens.

3. Enter a name for your data that matches the name of the `Simulink.Signal` object.

4. In the Data properties dialog box, select the **Data must resolve to Simulink signal object** check box, as in this example.



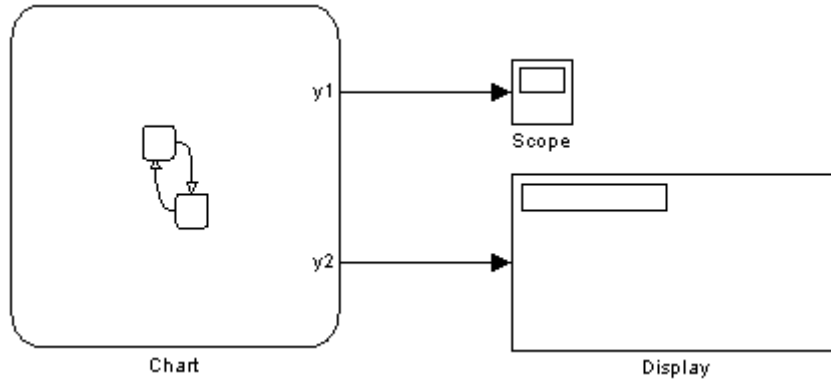
After you check this box, the dialog box removes or grays out the properties that your data will inherit from the signal.



For a list of properties that your data can inherit during signal resolution, see “Inherited Properties” on page 8-55.

A Simple Example

This topic presents an example that demonstrates how a Stateflow chart resolves local and output data to `Simulink.Signal` objects. To open the model, click sf_resolve_signal_object, or type `sf_resolve_signal_object` at the MATLAB command prompt. The model appears as shown.



In the base workspace, there are three `Simulink.Signal` objects with these properties:

Contents of: Base Workspace

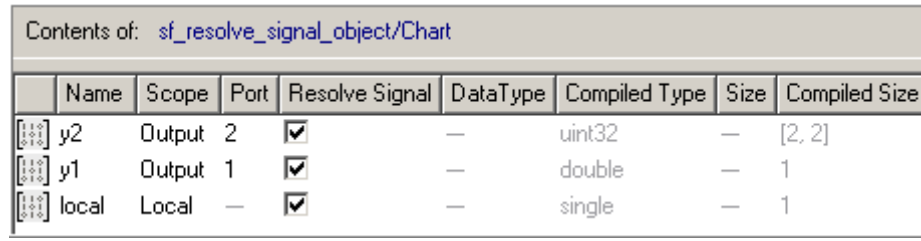
Name	DataType	Dimensions	Complexity	StorageClass	Min	Max
y2	uint32	[2 2]	auto	Auto	-Inf	Inf
y1	double	1	auto	SimulinkGlobal	-Inf	Inf
local	single	1	auto	ExportedGlobal	-Inf	Inf

The Stateflow chart contains three data objects — two outputs and a local variable — that will resolve to a signal with the same name, as follows:

Contents of: sf_resolve_signal_object/Chart

Name	Scope	Port	Resolve Signal	DataType	Compiled Type	Size	Compiled Size
y2	Output	2	☑	—	unknown	—	
y1	Output	1	☑	—	unknown	—	
local	Local	—	☑	—	unknown	—	

When you build the model, each data object inherits the properties of the identically named signal:



Contents of: sf_resolve_signal_object/Chart

	Name	Scope	Port	Resolve Signal	DataType	Compiled Type	Size	Compiled Size
	y2	Output	2	☑	—	uint32	—	[2, 2]
	y1	Output	1	☑	—	double	—	1
	local	Local	—	☑	—	single	—	1

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 the `local` as an exported global variable:

```
/*
 * Exported States
 *
 * Note: Exported states are block states with an exported global
 * storage class designation.  RTW declares the memory for these states
 * and exports their symbols.
 *
 */
extern real32_T local;                 /* '<Root>/Chart' */
```

Best Practices for Using Data in Stateflow Charts

In this section...

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.

In the Stateflow action language, inherited properties of outputs are determined solely by external information from Simulink models and *not* from the code. By contrast, within the Embedded MATLAB language subset — used in truth tables and Embedded MATLAB functions — inherited properties of outputs are determined solely from the code and the properties of the inputs.

Note Stateflow blocks in libraries can inherit data properties. However, multiple instances of the same library in a model must inherit the same values for those properties.

Tips for Using Data Stores in Stateflow Charts

When Binding to Data Stores in Stateflow Charts

When you bind a Stateflow data object to a data store, the Stateflow object inherits all properties from the data store. To ensure that properties propagate correctly when you access data stores, follow these guidelines to create data stores:

- Specify the signal type as `real`.

- Specify a data type other than `auto`.
- Minimize the use of automatic-mode properties.

When Enforcing Writes Before Reads in Unconnected Blocks

To enforce writes before reads when unconnected blocks share global data in Stateflow charts, follow these guidelines:

- Segregate reads into separate blocks from writes.
- Assign priorities to blocks so that your model invokes write blocks before read blocks.

  For instructions on how to set block execution order, see "Controlling and Displaying the Sorted Order" in the Simulink software documentation.

Transferring Data Across Models

In this section...

Copying Data Objects

When you copy a Stateflow 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.

Moving 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.

9

Defining Events

How Events Work in Stateflow Charts

In this section...

What Is an Event?

An *event* is a Stateflow object that can trigger actions in one of these objects:

- A Simulink triggered subsystem
- A Simulink function-call subsystem
- A Stateflow chart

When to Use Events

Use events when you want to do one of the following:

- Activate a Simulink triggered subsystem (see "Using Edge Triggers to Activate a Simulink Block" on page 9-16)
- Activate a Simulink function-call subsystem (see "Using Function Calls to Activate a Simulink Block" on page 9-21)
- Trigger actions in parallel states of a Stateflow chart (see "Broadcasting Events in Actions" on page 10-50)

Types of Events

An explicit event is an event that you define and can have one of these scopes.

Scope	Description
Local	Event that can occur anywhere in a Stateflow machine but is visible only in the parent object (and descendants of the parent).

Scope	Description
Input from Simulink	Event that occurs in a Simulink block but is broadcast to a Stateflow chart. See "Using Input Events to Activate a Stateflow Chart" on page 9-11.
Output to Simulink	Event that occurs in a Stateflow chart but is broadcast to a Simulink block. See "Using Output Events to Activate a Simulink Block" on page 9-16.
Exported	Event that can be broadcast by external code built into a standalone or code generation target. You can define exported events only for a Stateflow machine. See "Exporting Events to Stateflow External Code" on page 9-28.
Imported	Externally defined event that can be broadcast anywhere within the hierarchy of a Stateflow machine. You can define imported events only for a Stateflow machine. See "Importing Events from Stateflow External Code" on page 9-29.

An implicit event is a built-in event that broadcasts automatically during chart execution (see "Using Implicit Events" on page 9-31).

Where You Can Use Events

You can define explicit events at these levels of the Stateflow hierarchy.

An event you define in a...	Is visible to...
Machine **Note** Avoid defining events at this level of the hierarchy. For details, see "Best Practices for Using Events in Stateflow Charts" on page 9-36.	All charts in the model and all states and substates
Chart	The chart and all states and substates
Subchart	The subchart and all states and substates
State	The state and all substates

How to Define Events

In this section...

Adding Events Using the Stateflow Editor

In the Stateflow Editor, you can add events to your Stateflow chart. Follow these steps:

1. In the Stateflow Editor, select **Add** > **Event**.
2. In the resulting submenu, select the scope for the event.

   The Stateflow Editor adds a default definition of the new event to the Stateflow hierarchy, and the Event properties dialog box appears.
3. Specify properties for the event in the Event properties dialog box, as described in "Setting Properties for an Event" on page 9-7.

Adding Events Using the Model Explorer

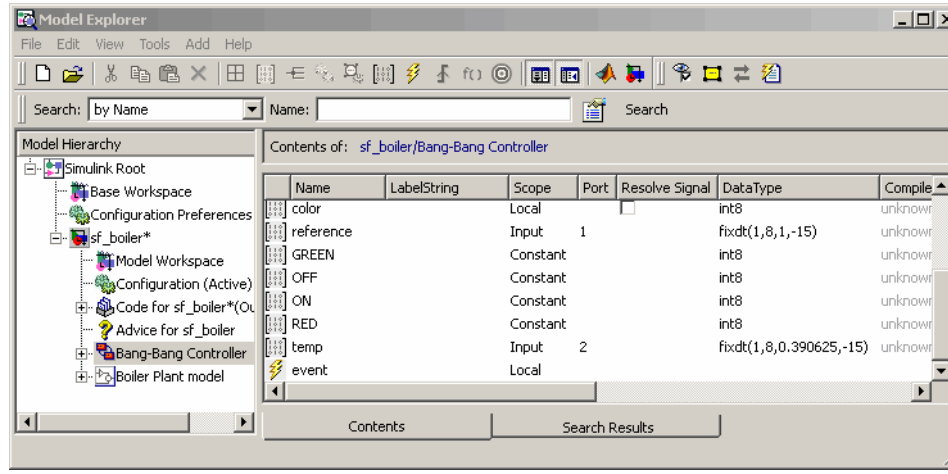
To add events using the Model Explorer, follow these steps:

1. In the Stateflow Editor, select **Tools** > **Explore**.

   The Model Explorer appears.
2. In the Model Explorer, select the object in the Stateflow hierarchy where you want the new event to be visible.

   The object you select becomes the *parent* of the event.
3. Select **Add** > **Event**, or click the **Add Event** button:

The Model Explorer adds a default definition for the new event in the hierarchy and displays an entry row for the new event in the **Contents** pane, as in this example.



4 Change the properties of the event you add in one of these ways:

- Right-click the event row and select **Properties** to open the Event properties dialog box.

  See “Setting Properties for an Event” on page 9-7 for a description of each property for an event.

- Click individual cells in the entry row to set specific properties such as **Name**, **Scope**, and **Port**.

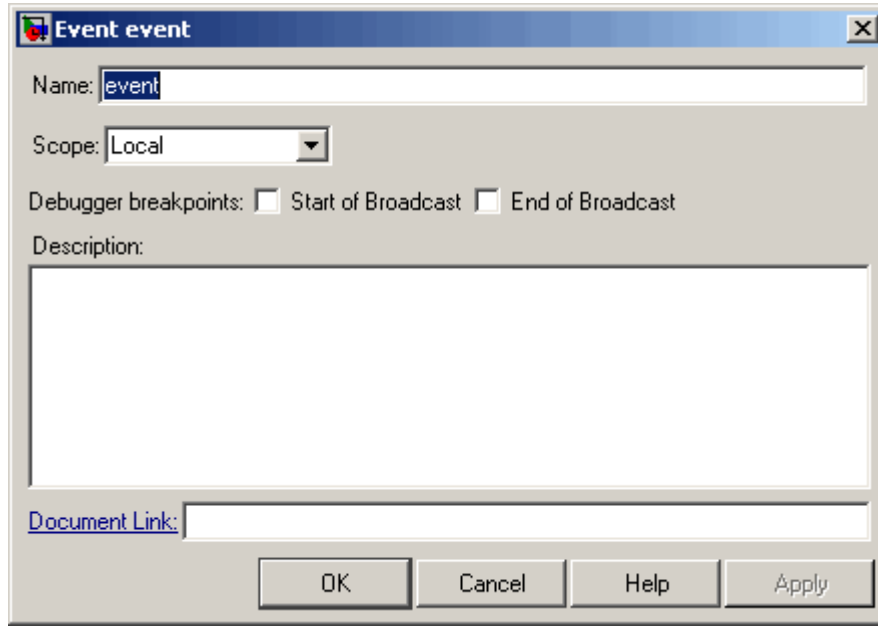
Setting Properties for an Event

In this section...

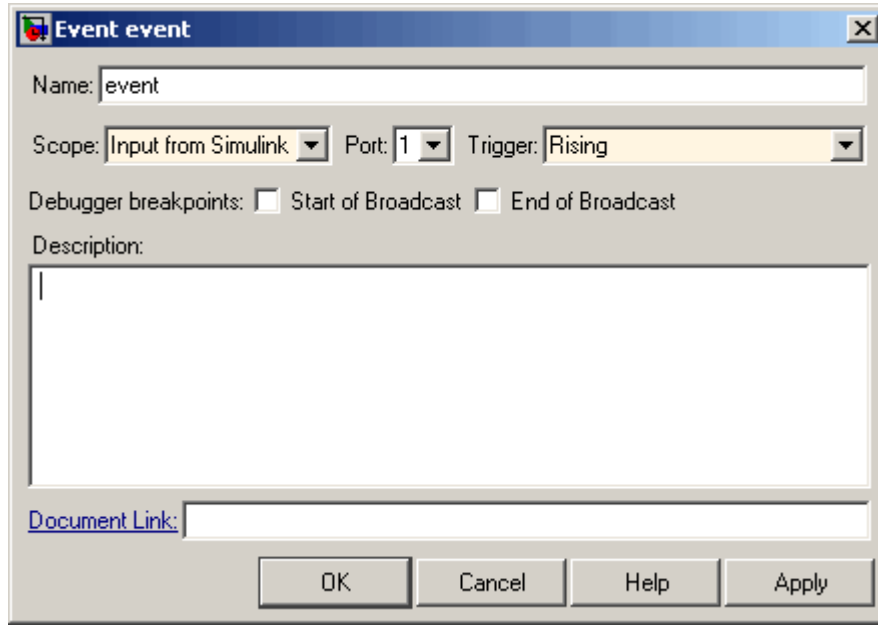
When to Use the Event Properties Dialog Box

Use the Event properties dialog box when you want to modify properties of an event, which can vary based on the scope of the event. The Event properties dialog box displays only the property fields that apply to the event you are modifying.

For example, the dialog box displays these properties and default values for an event whose scope is **Local**.



For input events, the dialog box displays these properties and defaults.



Accessing the Event Properties Dialog Box

To access the Event properties dialog box, use one of these methods:

- Add a new event from the Stateflow Editor.

  The Event properties dialog box appears, as described in "Adding Events Using the Stateflow Editor" on page 9-5.

- Open the Event properties dialog box in the Model Explorer in one of these ways:
  - Double-click the event in the **Contents** pane.
  - Right-click the event in the **Contents** pane and select **Properties**.
  - Select the event in the **Contents** pane and then select **View** > **Dialog View**.

    The Event properties dialog box opens inside the Model Explorer.

See “Adding Events Using the Model Explorer” on page 9-5.

Property Fields

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.

Scope

Scope of the event. The scope specifies where the event occurs relative to the parent object. For information about types of scope, see “Types of Events” on page 9-2.

Port

Property that applies to *input* and *output* events.

- For input events, port is the index of the input signal that triggers the event.
- For output events, port is the index of the signal that outputs this event.

You assign input and output events to ports in the order in which you add the events. For example, you assign the first input event to input port 1 and the third output event to output port 3.

You can change port assignments in the Model Explorer or the Event properties dialog box. When you change the number of one port, the numbers of other ports adjust automatically to preserve the relative order. See “Associating Input Events with Control Signals” on page 9-14 and “Associating Output Events with Output Ports” on page 9-26.

Trigger

Type of signal that triggers an input or output event. See “Using Input Events to Activate a Stateflow Chart” on page 9-11 or “Using Output Events to Activate a Simulink Block” on page 9-16.

Debugger Breakpoints

Option for setting debugger breakpoints at the start or end of an event broadcast.

Description

Description of this event. You can enter brief descriptions of events in the hierarchy.

Document Link

Link to online documentation for events in a Stateflow chart. To document a particular event, set the **Document Link** property to a Web URL address or MATLAB expression that displays documentation in a suitable online format (for example, an HTML file or text in the MATLAB Command Window). When you click the blue **Document Link** text, the chart evaluates the expression.

Using Input Events to Activate a Stateflow Chart

In this section...

What Is an Input Event?

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

You can activate a Stateflow chart via a change in control signal (an edge-triggered input event) or a function call from a Simulink block (a function-call input event). The sections that follow describe when and how to use each type of input event.

Note You cannot mix edge-triggered and function-call input events in a Stateflow chart. If you try to mix these input events, an error message appears during simulation.

Using Edge Triggers to Activate a Stateflow Chart

An edge-triggered input event causes a Stateflow chart to execute during the current time step of simulation. This type of input event works only when a change in control signal acts as a trigger.

When to Use an Edge-Triggered Input Event

Use an edge-triggered input event to activate a chart when your model requires chart execution at regular (or periodic) intervals.

How to Define an Edge-Triggered Input Event

To define an edge-triggered input event, follow these steps:

1 Add an event to the Stateflow chart, as described in "How to Define Events" on page 9-5.

Note You must add an input event to the chart and not to one of its objects.

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 Set the **Trigger** property to one of these edge triggers.

Edge Trigger Type	Description
`Rising`	Rising edge trigger, where the control signal changes from either 0 or a negative value to a positive value.
`Falling`	Falling edge trigger, where the control signal changes from either 0 or a positive value to a negative value.
`Either`	Either rising or falling edge trigger.

In all cases, the signal must cross 0 to be a valid edge trigger. For example, a signal that changes from -1 to 1 is a valid rising edge, but a signal that changes from 1 to 2 is not valid.

Example of Using an Edge-Triggered Input Event

The demo model `sf_loop_scheduler` shows how to use an edge-triggered input event to activate a Stateflow chart at regular intervals. For information on running this model and how it works, see "Scheduling One Subsystem in a Single Time Step Using a Loop Scheduler" on page 18-12.

Using Function Calls to Activate a Stateflow Chart

A function-call input event causes a Stateflow chart to execute during the current time step of simulation.

Note When you use 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.

How to Define a Function-Call Input Event

To define a function-call input event, follow these steps:

1. Add an event to the Stateflow chart, as described in "How to Define Events" on page 9-5.

   Note You must add an input event to the chart and not to one of its objects.

   ---

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. Set the **Trigger** property to `Function call`.

Example of Using a Function-Call Input Event

The demo model `sf_loop_scheduler` shows how to use a function-call input event to activate a Stateflow chart. For information on running this model and how it works, see "Scheduling One Subsystem in a Single Time Step Using a Loop Scheduler" on page 18-12.

Associating Input Events with Control Signals

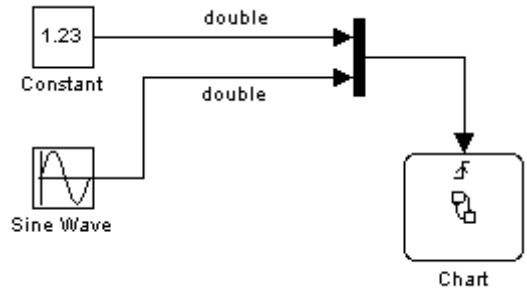
When you define one or more input events for a chart, a single trigger port to the chart block appears. External Simulink blocks can trigger the input events via a signal or vector of signals connected to the trigger port. The **Port** property of an input event associates the event with a specific element of a control signal vector that connects to the trigger port (see "Port" on page 9-9).

The number of the port that you assign to the input event acts as an index into the control signal vector. For example, the first element of the signal vector triggers the input event assigned to input port 1, the fourth element triggers the input event assigned to input port 4, and so on. You assign port numbers in the order in which you add the events. However, you can change these assignments by setting the **Port** property of an event to the index of the signal that you use to trigger the event.

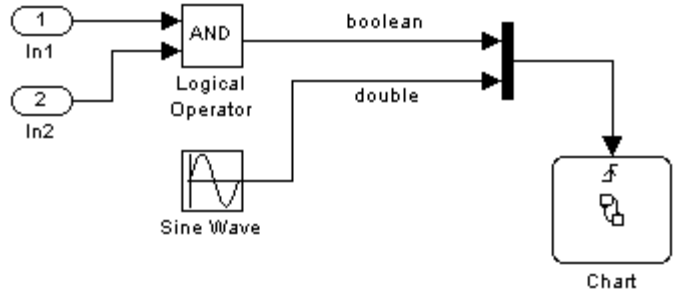
Data Types Allowed for Input Events

For multiple input events to a trigger port, the data types of all signals must be identical. If you use signals of different data types as input events, an error message appears when you try to simulate your model.

For example, you can mux two input signals of type `double` to use as input events to a chart.



However, you cannot mux two input signals of different data types, such as `boolean` and `double`.



Behavior of 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 per valid event. For edge-triggered input events, multiple edges can occur in the same time step, which wake the chart more than once in that time step. In this situation, events occur (and wake the chart) in an ascending order based on their port numbers.

Behavior of 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 can be valid in a single time step.

Using Output Events to Activate a Simulink Block

In this section...

What Is an Output Event?

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 allows a chart to notify other blocks in a model about events that occur in the chart.

You use output events to activate other blocks in the same model. You can define multiple output events in a chart, where each output event maps to an output port (see "Port" on page 9-9).

Note Output events must be scalar.

Using Edge Triggers to Activate a Simulink Block

An edge-triggered output event activates a Simulink block to execute during the current time step of simulation. This type of output event works only when a change in control signal acts as a trigger.

When to Use an Edge-Triggered Output Event

Use an edge-triggered output event to activate a Simulink subsystem when your model requires subsystem execution at regular (or periodic) intervals.

How to Define an Edge-Triggered Output Event

To define an edge-triggered output event, follow these steps:

1 Add an event to the Stateflow chart, as described in "How to Define Events" on page 9-5.

2 Set the **Scope** property for the event to **Output to Simulink**.

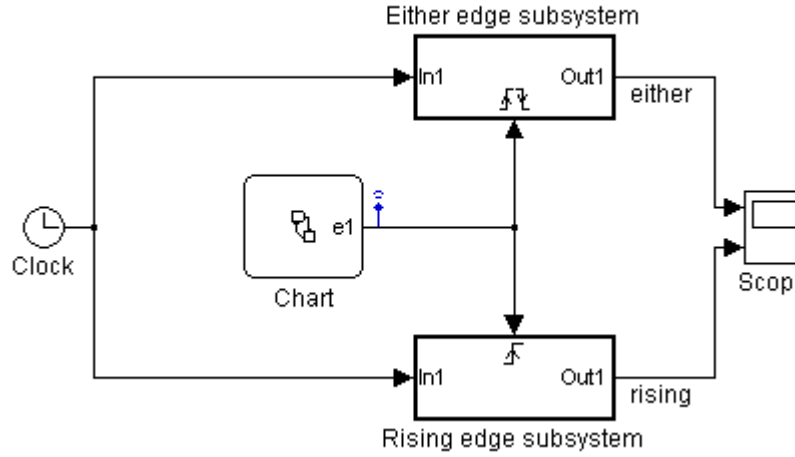
   For each output event you define, an output port appears on the Stateflow block.

3 Set the **Trigger** property of the output event to `Either Edge`.

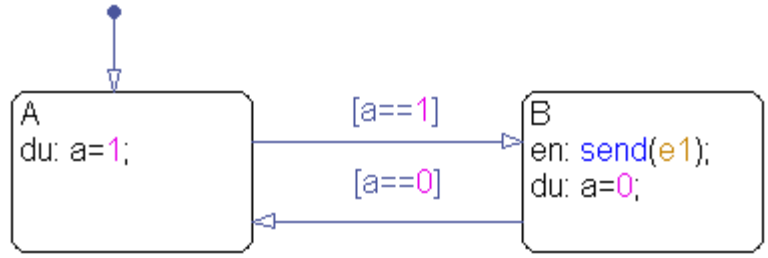
Note Unlike edge-triggered input events, you cannot specify a `Rising` or `Falling` edge trigger.

Example of Using an Edge-Triggered Output Event

The following model shows how to use an edge-triggered output event to activate triggered subsystems at regular intervals.



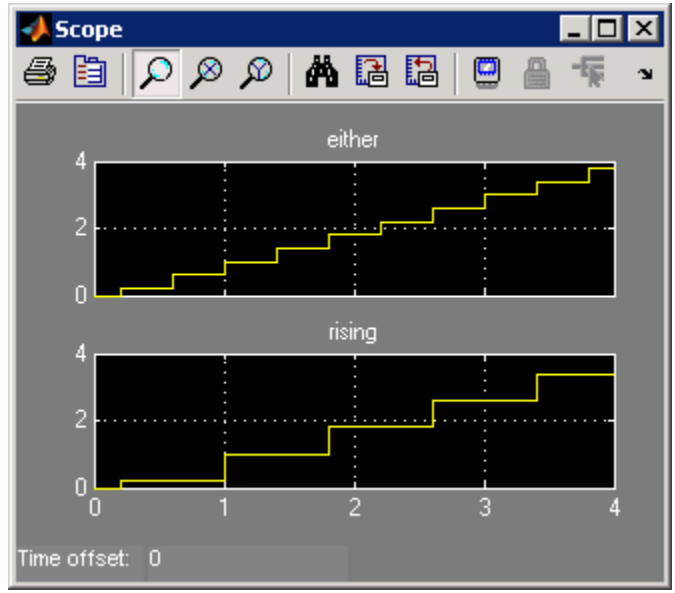
The chart contains the edge-triggered output event `e1` and the local data `a`, which switches between 0 and 1 during simulation.



In a Stateflow chart, the **Trigger** property of an edge-triggered output event is always `Either Edge`. However, Simulink triggered subsystems can have a `Rising`, `Falling`, or `Either Edge` trigger. This model shows the difference between triggering a rising edge subsystem and an either edge subsystem.

The output event `e1` triggers the...	**On...**	**When the data `a`...**
Either edge subsystem	Every event broadcast	Switches from 0 to 1 or from 1 to 0
Rising edge subsystem	Every other event broadcast	Switches from 0 to 1

When you simulate the model, the scope shows these results.



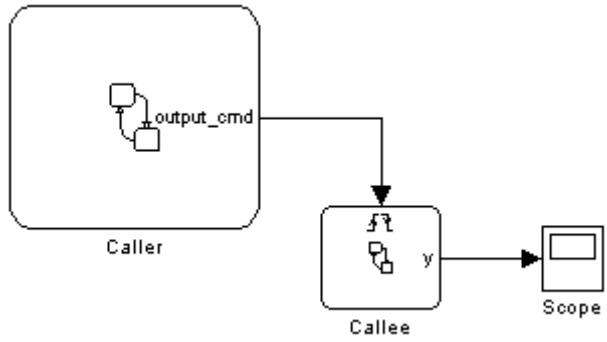
Queuing Behavior for Broadcasting an Edge-Triggered Output Event Multiple Times

If a chart tries to broadcast the same edge-triggered output event multiple times in a single time step, the chart dispatches only one of these broadcasts in the present time step. However, the chart *queues up* any pending broadcasts for dispatch — that is, one at a time in successive time steps. Each time the chart wakes up in successive time steps, if any pending broadcasts exist for the output event, the chart signals the edge-triggered subsystem for execution. Based on the block sorted order of the Simulink model, the edge-triggered subsystem executes. (For details, see "Controlling and Displaying the Sorted Order" in the Simulink documentation.)

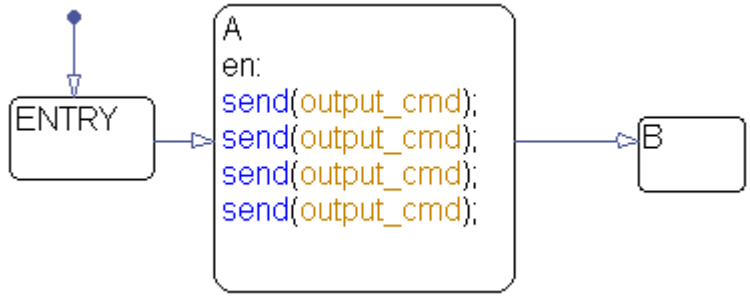
Note For information on what happens for function-call output events, see "Interleaving Behavior for Broadcasting a Function-Call Output Event Multiple Times" on page 9-24.

Example of Queuing Behavior for Edge-Triggered Output Events

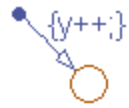
In this example, the chart Caller uses the edge-triggered output event `output_cmd` to activate the chart Callee.



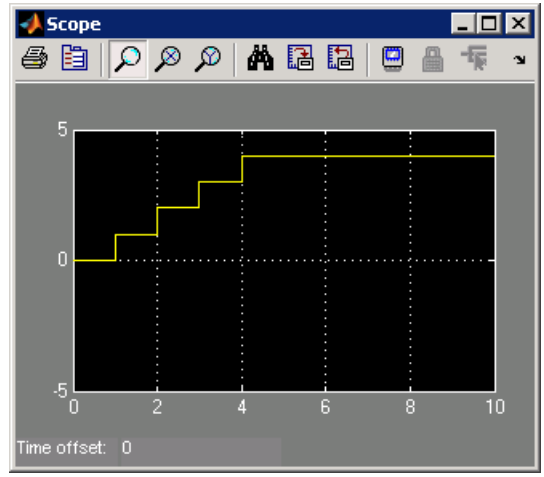
The chart Caller tries to broadcast the same edge-triggered output event four times in a single time step, as shown.



Each time the chart Callee is activated, the output data `y` increments by one.



When you simulate the model, you see this output in the scope.



At t = 1, the chart Caller dispatches only one of the four output events. Therefore, the chart Callee executes once during that time step. However, the chart Caller *queues up* the other three event broadcasts for future dispatch — that is, one at a time for t = 2, 3, and 4. Each time Caller wakes up in successive time steps, it activates Callee for execution. Therefore, the action `y++` occurs once per time step at t = 1, 2, 3, and 4. During simulation, Callee executes based on the block sorted order of the Simulink model.

Using Function Calls to Activate a Simulink Block

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.

When to Use a Function-Call Output Event

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.

How to Define a Function-Call Output Event

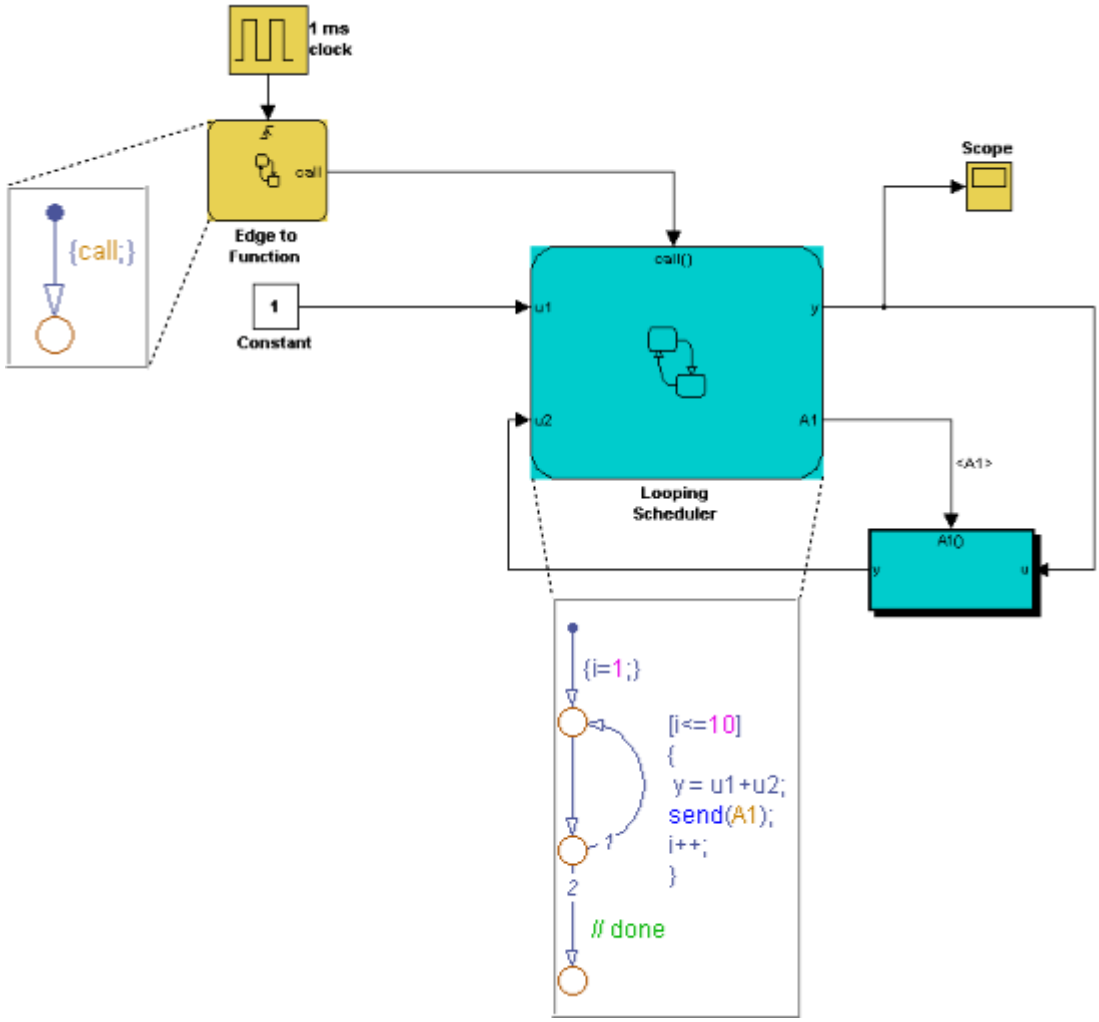
To define a function-call output event, follow these steps:

1. Add an event to the Stateflow chart, as described in "How to Define Events" on page 9-5.
2. Set the **Scope** property for the event to **Output to Simulink**.

   For each output event you define, an output port appears on the Stateflow block.
3. Set the **Trigger** property of the output event to `Function call`.

Example of Using a Function-Call Output Event

The demo model `sf_loop_scheduler` shows how to use a function-call output event to activate a Simulink block. For information on running this model and how it works, see "Scheduling One Subsystem in a Single Time Step Using a Loop Scheduler" on page 18-12.



The function-call output event...	Of the chart...	Activates...
`call`	Edge to Function	The chart Looping Scheduler
`A1`	Looping Scheduler	The function-call subsystem A1

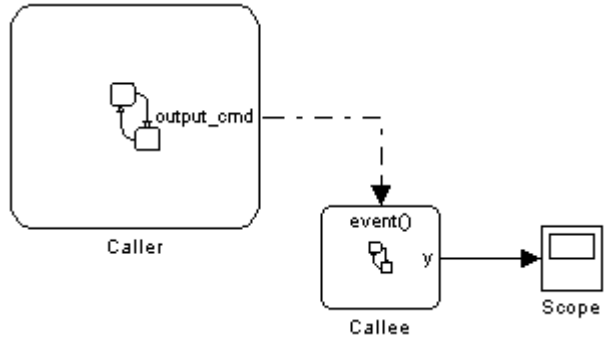
Interleaving Behavior for Broadcasting a Function-Call Output Event Multiple Times

If a chart tries to broadcast the same function-call output event multiple times 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 function-call initiator so that output from the function-call subsystem is available right away in the function-call initiator. (For details, see "Function-Call Subsystems" in the Simulink documentation.)

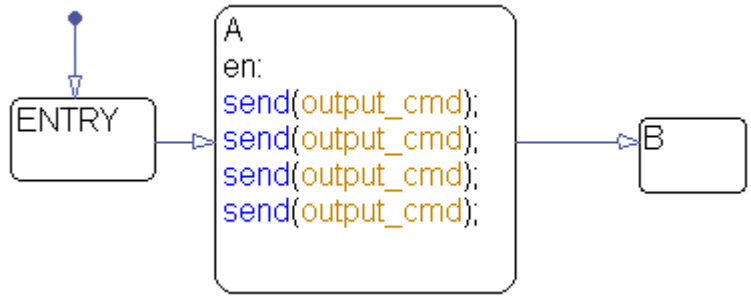
Note For information on what happens for edge-triggered output events, see "Queuing Behavior for Broadcasting an Edge-Triggered Output Event Multiple Times" on page 9-19.

Example of Interleaving Behavior for Function-Call Output Events

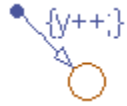
In this example, the chart Caller uses the function-call output event `output_cmd` to activate the chart Callee.



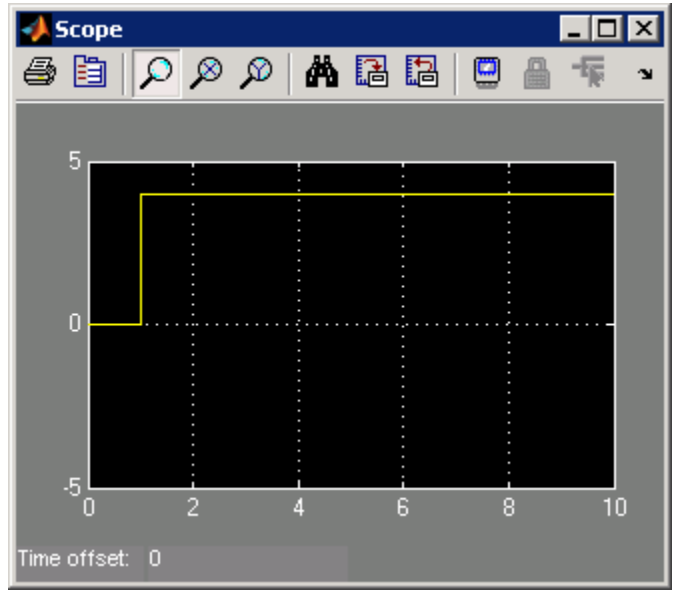
The chart Caller tries to broadcast the same function-call output event four times in a single time step, as shown.



Each time the chart Callee is activated, the output data y increments by one.



When you simulate the model, you see this output in the scope.



At t = 1, the chart Caller dispatches all four output events. Therefore, the chart Callee executes four times during that time step. Therefore, the action y++ also occurs four times in that time step. During simulation, execution of Callee is interleaved with execution of Caller so that output from Callee is available right away.

Associating Output Events with Output Ports

The **Port** property associates an output event with an output port on the chart block that owns the event. This property specifies the position of the output port relative to others.

All output ports appear sequentially from top to bottom. Output data ports appear sequentially above output event ports on the right side of a chart block. As you add output events, their default **Port** properties appear sequentially at the end of the current port list.

You can change the default port assignment of an event by resetting its **Port** property. When you change the **Port** property for an output event, the ports for the remaining output events automatically renumber, preserving the original order. For example, assume you have three output events, `OE1`, `OE2`, and `OE3`, which associate with the output ports 4, 5, and 6, respectively. If you change the **Port** property for `OE2` to 6, the ports for `OE1` and `OE3` renumber to 4 and 5, respectively.

Accessing Simulink Subsystems Triggered By Output Events

To access the Simulink subsystem associated with a Stateflow output event, follow these steps:

1. In the Stateflow Editor, right-click the state that contains the event of interest.
2. Select **Explore**.

   Using the Explore menu, you can access all events defined in the selected state.
3. Select the desired event.

   The Simulink subsystem associated with the event appears.

Sharing Events with Stateflow External Code

In this section...

Exporting Events to Stateflow External Code

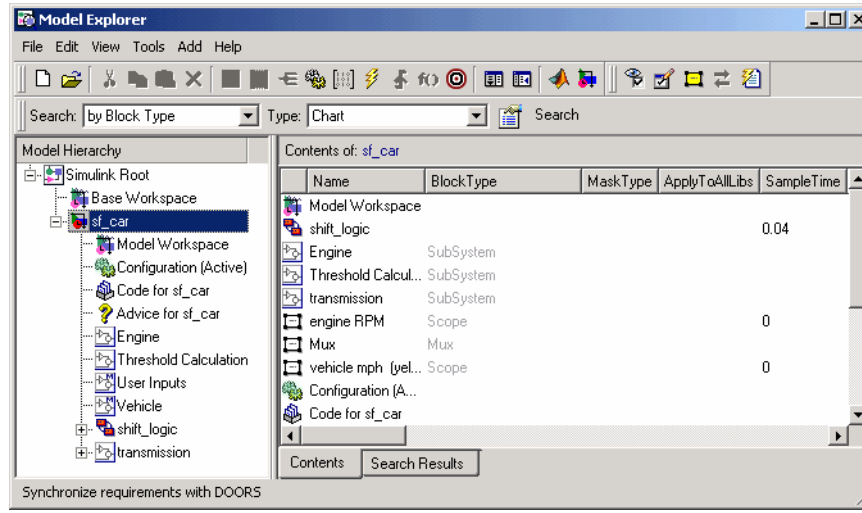
Stateflow machines can export events that trigger Stateflow charts in the model. Exported events are children of the Stateflow machine. You cannot define exported events at any other level in the Stateflow hierarchy.

How to Export Events

To export an event, follow these steps:

1 Add an event to the Stateflow machine, as described in "Adding Events Using the Model Explorer" on page 9-5.

In the **Model Hierarchy** pane of the Model Explorer, the Stateflow machine has the same name as the Simulink model. For example, consider the model `sf_car`.



In this model, the Stateflow machine is `sf_car` (highlighted) and not `shift_logic`, which is the Stateflow chart.

2. Set the **Scope** property of the event to **Exported**, as described in "Setting Properties for an Event" on page 9-7.

Format of Exported Events in External Code

The Stateflow code generator generates a function for each exported event. The C prototype for the exported event function takes the form

```
void external_broadcast_EVENT()
```

where `EVENT` is the name of the exported event. External code built into a target can trigger the event by invoking the event function. If you define an exported event named `switch_on`, external code can trigger this event by invoking the generated function `external_broadcast_switch_on`.

See "Exported Events" on page 16-33 for examples of how to trigger an exported event and how to export a Stateflow event to Stateflow external code.

Importing Events from Stateflow External Code

The Stateflow machine is the parent of imported events defined by external code. You can import an event to build a custom target, which triggers the imported event in external code.

How to Import Events

To import an event, follow these steps:

1. Add an event to the Stateflow machine, as described in "Adding Events Using the Model Explorer" on page 9-5.

2. Set the **Scope** property of the event to **Imported**, as described in "Setting Properties for an Event" on page 9-7.

Note You must use the Model Explorer to add imported events to the Stateflow machine (see "Adding Events Using the Model Explorer" on page 9-5).

Format of Imported Events in External Code

Stateflow software assumes that external code defines each imported event as a function of the form

```
void external_broadcast_EVENT
```

where `EVENT` is the name of the imported event. If the Stateflow machine imports an external event named `switch_on`, Stateflow software assumes that external code defines a function named `external_broadcast_switch_on` that broadcasts the event to external code. When you build a target for the Stateflow machine, the Stateflow code generator encodes actions that signal imported events as calls to external broadcast event functions in the external code.

See "Imported Events" on page 16-35 for an example of a Stateflow external code event imported into the Stateflow hierarchy.

Using Implicit Events

In this section...

What Are 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.

Referencing 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`.

Implicit Event	Meaning
`enter` (*`state_name`*) or `en(`*`state_name`*`)`	Specifies and implicitly generates a local event when the specified `state_name` is entered.
`exit` (*`state_name`*) or `ex(`*`state_name`*`)`	Specifies and implicitly generates a local event when the specified `state_name` is exited.
`tick`	Specifies and implicitly generates a local event when the chart of the action being evaluated awakens.
`wakeup`	Same as the `tick` keyword.

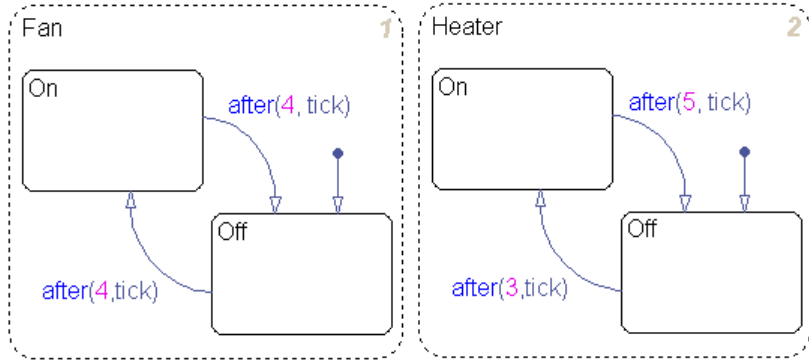
If more than one object has the same name, use the dot operator to qualify the name of the object with the name of its parent. These examples are valid references to implicit events:

```
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.

Example of an Implicit Event

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 "Using Temporal Logic in State Actions and Transitions" on page 10-56.

Counting Events

In this section...

When to Count Events

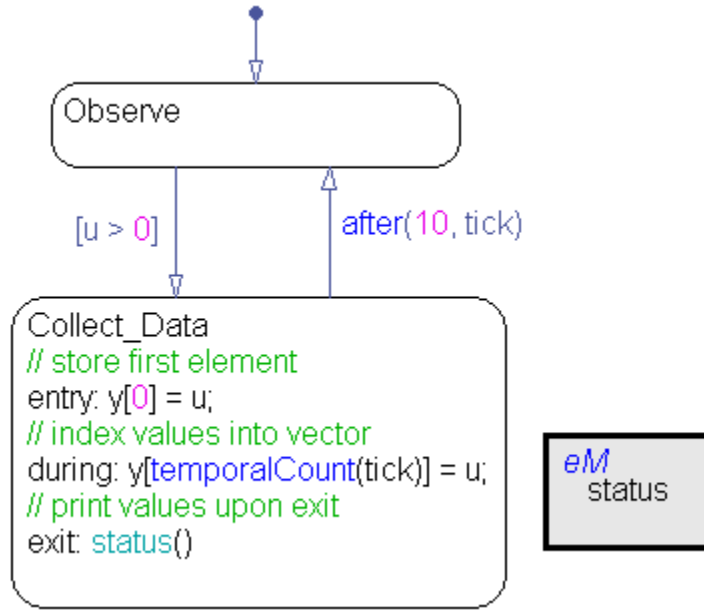
Count events when you want to keep track of explicit or implicit events in your chart.

How to Count Events

You can count occurrences of explicit and implicit events using the `temporalCount` operator. For information about the syntax of this operator, see "Using Temporal Logic in State Actions and Transitions" on page 10-56.

Example of Storing Input Data in a Vector

Suppose you want to store input data in a vector during chart simulation, as shown below.



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 data stored in the vector.

Best Practices for Using Events in Stateflow Charts

Use the send command to broadcast explicit events in actions

In state actions (`entry`, `during`, `exit`, and `on event_name`) and transition actions, use the `send` command to broadcast explicit events. Using this command enhances readability of a chart and ensures that explicit events are not mistaken for data.

Avoid defining machine-parented events

If you have multiple charts in your model, avoid defining events where the parent is the Stateflow machine. If you broadcast an event to all charts in your model, the order in which the charts awaken is unknown.

Do not mix edge-triggered input events and function-call input events in a chart

If you mix input events that use edge triggers and function calls, the chart detects this violation during parsing or code generation. An error message appears and chart execution stops.

Avoid using the enter implicit event to check state activity

Use the `in` operator instead of the `enter` implicit event to check state activity. See “Checking State Activity” on page 10-89 for details.

Transferring Events Across Models

In this section...

Copying Event Objects

When you copy a Stateflow chart from one Simulink model to another, all event objects in the chart hierarchy are copied *except* those parented by the Stateflow machine. However, you can use the Model Explorer to transfer individual event objects from machine to machine.

To *copy* objects, follow these steps:

1. In the **Contents** pane of the Model Explorer, right–click the event 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.

Moving Event Objects

To *move* objects, click the event object in the **Contents** pane of the Model Explorer and drag it to the destination Stateflow machine in the **Model Hierarchy** pane.

10

Using Actions in Stateflow Charts

Defining Action Types

In this section...

About Action Types

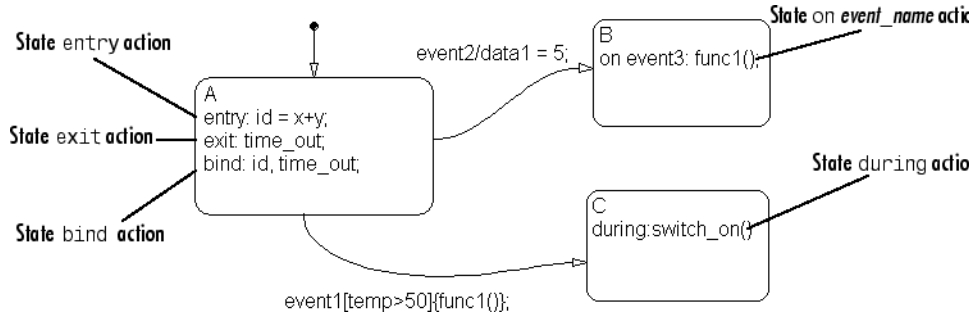
You can attach actions to states and transitions through the syntax of their labels. States specify actions through five action types: entry, during, exit, bind, and on *event_name*. Transitions specify actions through four action types: event trigger, condition, condition action, and transition action.

State Action Types

States can have different action types, which include `entry`, `during`, `exit`, `bind`, and, `on` *`event_name`* actions. The actions for states are assigned to an action type using label notation with this general format:

```
name/
entry:entry actions
during:during actions
exit:exit actions
bind:data_name, event_name
on event_name:on event_name actions
```

Different state action types appear in this diagram.



After you enter the name in the state label, enter a carriage return and specify the actions for the state.

Note The order you use to enter action types in the label does not matter.

This table summarizes the different state action types and what they do.

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*

State Action	Abbreviation	Description
`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 a full description of `entry`, `exit`, `during`, `bind`, and `on` *`event_name`* actions, see the topics that follow. For more information about the `after`, `before`, `at`, and `every` temporal logic operators, see "Using Temporal Logic in State Actions and Transitions" on page 10-56.

Note In the preceding table, the temporal logic operators use the syntax of *event-based* temporal logic. For *absolute-time* temporal logic, the operators use a different syntax. For details, see "Operators for Absolute-Time Temporal Logic" on page 10-63.

Entry Actions

Entry actions are preceded by the prefix `entry` or `en` for short, followed by a required colon (`:`), followed by one or more actions. Separate multiple actions with a carriage return, semicolon (`;`), or a comma (`,`). 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 `entry` actions only.

Entry actions are executed for a state when the state is entered (becomes active). In the preceding example in "State Action Types" on page 10-2, the entry action `id = x+y` is executed when the state A is entered by the default transition.

For a detailed description of the semantics of entering a state, see "Entering a State" on page 3-33 and "State Execution Example" on page 3-35.

Exit Actions

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

Exit actions for a state are executed when the state is active and a transition out of the state is taken.

For a detailed description of the semantics of exiting a state, see "Exiting an Active State" on page 3-35 and "State Execution Example" on page 3-35.

During Actions

During actions are preceded by the prefix `during` or `du` for short, followed by a required colon (`:`), followed by one or more actions. Separate multiple actions with a carriage return, semicolon (`;`), or a comma (`,`).

During actions are executed for a state when it is active and an event occurs and no valid transition to another state is available.

For a detailed description of the semantics of executing an active state, see "Executing an Active State" on page 3-34 and "State Execution Example" on page 3-35.

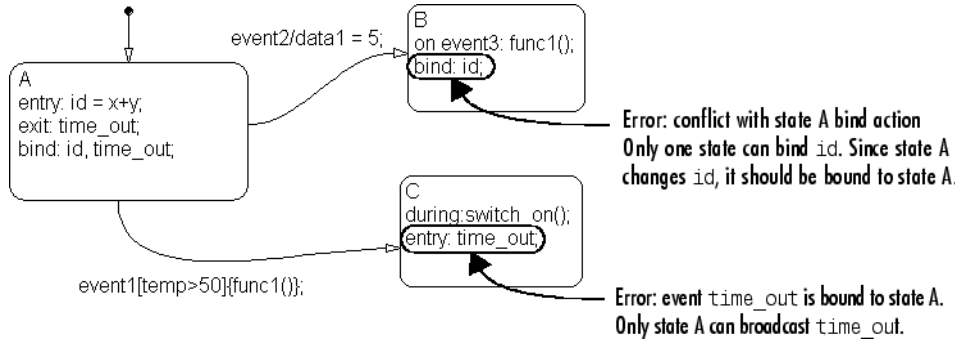
Bind Actions

Bind actions are preceded by the prefix `bind`, followed by a required colon (`:`), followed by one or more events or data. Separate multiple data/events with a carriage return, semicolon (`;`), or a comma (`,`).

Bind actions bind the specified data and events to a state. Data bound to a state can be changed by the actions of that state or its children. Other states and their children are free to read the bound data, but they cannot change it. Events bound to a state can be broadcast only by that state or its children. Other states and their children are free to listen for the bound event, but they cannot send it.

Bind actions are applicable to a Stateflow chart whether the binding state is active or not. In the preceding example in "State Action Types" on page 10-2, the bind action `bind: id, time_out` for state `A` binds the data `id` and the event `time_out` to state `A`. This forbids any other state (or its children) in the Stateflow chart from changing `id` or broadcasting event `time_out`.

If another state includes actions that change data or send events that are bound to another state, a parsing error results. The following example demonstrates a few of these error conditions:



Binding a function-call event to a state also binds the function-call subsystem that it calls. In this case, the function-call subsystem is enabled when the binding state is entered and disabled when the binding state is exited. For a detailed description of this feature, see "Using Bind Actions to Control Function-Call Subsystems" on page 10-99.

On Event_Name Actions

On *`event_name`* actions are preceded by the prefix `on`, followed by a unique event, *`event_name`*, followed by one or more actions. Separate multiple actions with a carriage return, semicolon (`;`), or a comma (`,`). You can specify actions for more than one event by adding additional `on` *`event_name`* lines for different events. If you want different events to trigger different actions, enter multiple `on` *`event_name`* action statements in the state's label, each specifying the action for a particular event or set of events, for example:

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

On *event_name* actions execute when the state is active and the event *event_name* is received by the state. This action coincides with execution of during actions for the state.

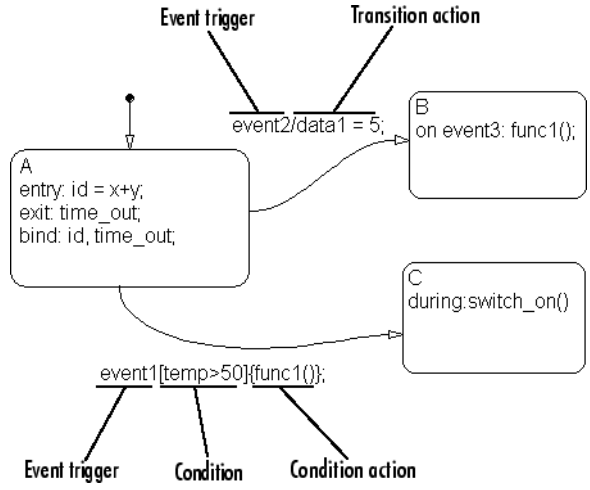
For a detailed description of the semantics of executing an active state, see "Executing an Active State" on page 3-34.

Transition Action Types

In "State Action Types" on page 10-2, you see how you can attach actions to the label for a state. You can also attach actions to a transition through its label. Transitions can have different action types, which include event triggers, conditions, condition actions, and transition actions. The actions for transitions are assigned to an action type using label notation with the following general format:

```
event_trigger[condition]{condition_action}/transition_action
```

The following example shows examples of transition action types:



Event Triggers

In transition label syntax, event triggers appear first as the name of an event. 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 10-7, 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. The absence of an event indicates that the transition is taken upon the occurrence of any event. Multiple events 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 10-7, 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 "Checking State Activity" on page 10-89)

    Note A chart cannot use the `in` condition to trigger actions based on the activity of states in other charts.

  - Temporal logic conditions (see "Using Temporal Logic in State Actions and Transitions" on page 10-56)
- The condition expression can call a graphical function, truth table function, or Embedded 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 Stateflow 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 10-7, 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 A condition is checked only if the event trigger (if any) is active.

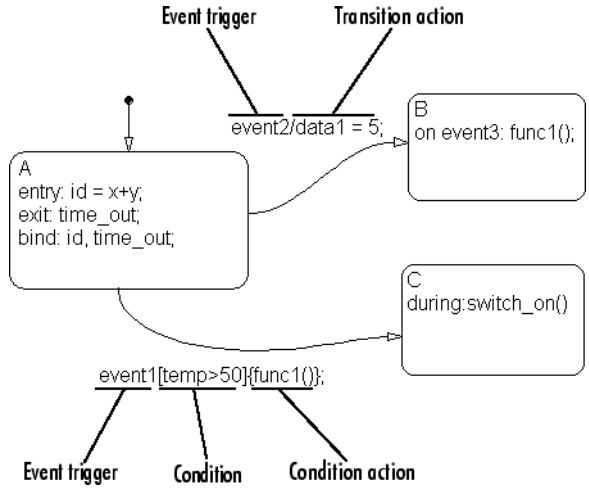
Transition Actions

In transition label syntax, transition actions are preceded with a forward slash (/). In the example in "Transition Action Types" on page 10-7, the transition from state A to state B has the transition action `data1 = 5`.

Transition actions are executed when the transition is actually taken. They are executed 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 is executed only when the entire transition path to the final destination is determined to be valid.

Example of Action Type Execution

In "State Action Types" on page 10-2 and "Transition Action Types" on page 10-7, you are introduced to the notation and meaning of the Stateflow action language types. In this topic, you see how Stateflow action language types interact when you execute this chart.



If the Stateflow chart wakes up, these steps occur:

1. The default transition to state A occurs.
2. The entry action `id = x+y` executes.
3. The event `time_out` binds to state A.
4. State A is active.

If state A is active and the Stateflow chart receives the event `event2`, these steps occur:

1. The exit action broadcast of the event `time_out` executes.
2. State A becomes inactive.
3. The transition action `data1 = 5` executes.

4 State B becomes active.

If state `A` is active and the Stateflow chart receives the event `event1`, these steps occur:

1. The condition `temp > 50` evaluates.

If the condition is...	Then the remaining steps...	And the transition...
True (temp > 50)	Execute	Occurs
False (temp <= 50)	Do not execute	Does not occur

2. The condition action call to the function `func1()` executes.
3. The exit action broadcast of the event `time_out` executes.
4. State A becomes inactive.
5. State C becomes active.

Using Operations in Actions

In this section...

Binary and Bitwise Operations

The table below summarizes the interpretation of all binary operators inStateflow action language. These operators work with the following order of precedence (1 = highest, 10 = lowest). Binary operators evaluate from left to right.

You can specify that the binary operators `&`, `^`, `|`, `&&`, and `||` are interpreted as bitwise operators in Stateflow generated C code for a chart or for all the charts in a model. See these individual operators in the table below for specific binary or bitwise operator interpretations.

Example	Precedence	Description
`a * b`	1	Multiplication
`a / b`	1	Division
`a %% b`	1	Modulus
`a + b`	2	Addition
`a - b`	2	Subtraction
`a >> b`	3	Shift operand `a` right by `b` bits. Noninteger operands for this operator are first cast to integers before the bits are shifted.

Example	Precedence	Description
`a << b`	3	Shift operand `a` left by `b` bits. Noninteger operands for this operator are first cast to integers before the bits are shifted.
`a > b`	4	Comparison of the first operand greater than the second operand
`a < b`	4	Comparison of the first operand less than the second operand
`a >= b`	4	Comparison of the first operand greater than or equal to the second operand
`a <= b`	4	Comparison of the first operand less than or equal to the second operand
`a == b`	5	Comparison of equality of two operands
`a ~= b`	5	Comparison of inequality of two operands
`a != b`	5	Comparison of inequality of two operands
`a <> b`	5	Comparison of inequality of two operands
`a & b`	6	One of the following: • Bitwise AND of two operands Enabled when you select **Enable C-bit operations** in the Chart properties dialog box. See "Specifying Chart Properties" on page 16-5. • Logical AND of two operands Enabled when you clear **Enable C-bit operations** in the Chart properties dialog box.

Example	Precedence	Description
`a ^ b`	7	One of the following: • Bitwise XOR of two operands Enabled when you select **Enable C-bit operations** in the Chart properties dialog box. See "Specifying Chart Properties" on page 16-5. • Operand `a` raised to power `b` Enabled when you clear **Enable C-bit operations** in the Chart properties dialog box.
`a | b`	8	One of the following: • Bitwise OR of two operands Enabled when you select **Enable C-bit operations** in the Chart properties dialog box. See "Specifying Chart Properties" on page 16-5. • Logical OR of two operands Enabled when you clear **Enable C-bit operations** in the Chart properties dialog box.
`a && b`	9	Logical AND of two operands
`a || b`	10	Logical OR of two operands

Unary Operations

The following unary operators are supported in Stateflow action language. Unary operators have higher precedence than binary operators and are evaluated right to left (right associative).

Example	Description
`~a`	Logical NOT of `a` Complement of `a` (if `bitops` is enabled)
`!a`	Logical NOT of `a`
`-a`	Negative of `a`

Unary Actions

The following unary actions are supported in Stateflow action language.

Example	Description
`a++`	Increment `a`
`a--`	Decrement `a`

Assignment Operations

The following assignment operations are supported in Stateflow action language.

Example	Description
`a = expression`	Simple assignment
`a := expression`	Used primarily with fixed-point numbers. See "Assignment (=, :=) Operations" on page 14-24 for a detailed description.
`a += expression`	Equivalent to `a = a + expression`
`a -= expression`	Equivalent to `a = a - expression`
`a *= expression`	Equivalent to `a = a * expression`
`a /= expression`	Equivalent to `a = a / expression`

The following assignment operations are supported in Stateflow action language when **Enable C-bit operations** is selected in the properties dialog box for the chart. See "Specifying Chart Properties" on page 16-5.

Example	Description
`a |= expression`	Equivalent to `a = a | expression` (bit operation). See operation `a | b` in "Binary and Bitwise Operations" on page 10-14.
`a &= expression`	Equivalent to `a = a & expression` (bit operation). See operation `a & b` in "Binary and Bitwise Operations" on page 10-14.
`a ^= expression`	Equivalent to `a = a ^ expression` (bit operation). See operation `a ^ b` in "Binary and Bitwise Operations" on page 10-14.

Pointer and Address Operations

The address operator is available for use with both custom code variables and Stateflow variables. The pointer operator is available for use with custom code variables only.

Note The action language parser uses a relaxed set of restrictions. As a result, many syntax errors are not trapped until compilation.

The following examples show syntax that is valid for use with *custom code* variables only.

```
varStruct.field = <expression>;
(*varPtr) = <expression>;
varPtr->field = <expression>;
myVar = varPtr->field;
varPtrArray[index]->field = <expression>;
varPtrArray[expression]->field = <expression>;
myVar = varPtrArray[expression]->field;
```

The following examples show syntax that is valid for use both with custom code variables and with Stateflow variables.

```
varPtr = &var;
ptr = &varArray[<expression>];
```

```
*(&var) = <expression>;
function(&varA, &varB, &varC);
function(&sf.varArray[<expression>]);
```

Type Cast Operations

You can use type cast operators to convert a value of one type to a value that can be represented in another type. Normally, you do not need to use type cast operators in actions because Stateflow software checks whether the types involved in a variable assignment differ and compensates by inserting the required type cast operator of the target language (typically C) in the generated code. However, external (custom) code might require data in a different type from those currently available. In this case, Stateflow software cannot determine the required type casts, and you must explicitly use a type cast operator to specify the target language type cast operator to generate.

For example, you might have a custom code function that requires integer RGB values for a graphic plot. You might have these values in Stateflow data, but only in data of type `double`. To call this function, you must type cast the original data and assign the result to integers, which you use as arguments to the function.

Stateflow type cast operations have two forms: the MATLAB type cast form and the explicit form using the `cast` operator. These operators and the special `type` operator, which works with the explicit `cast` operator, are described in the topics that follow.

MATLAB Form Type Cast Operators

The MATLAB type casting form has the general form

```
<type_op>(<expression>)
```

<type_op> is a conversion type operator that can be `double`, `single`, `int32`, `int16`, `int8`, `uint32`, `uint16`, `uint8`, or `boolean`. *<expression>* is the expression to be converted. For example, you can cast the expression `x+3` to a 16-bit unsigned integer and assign its value to the data `y` as follows:

```
y = uint16(x+3)
```

Explicit Type Cast Operator

You can also type cast with the explicit `cast` operator, which has the following general form:

```
cast(<expression>,<type>)
```

As in the preceding example, the statement

```
y = cast(x+3,uint16)
```

will cast the expression `x+3` to a 16-bit unsigned integer and assign it to `y`, which can be of any type.

type Operator

To make type casting more convenient, you can use a `type` operator that works with the explicit type cast operator `cast` to let you assign types to data based on the types of other data.

The `type` operator returns the type of an existing Stateflow data according to the general form

```
type(<data>)
```

where *<data>* is the data whose type you want to return.

The return value from a `type` operation can be used only in an explicit `cast` operation. For example, if you want to convert the data `y` to the same type as that of data `z`, use the following statement:

```
cast(y,type(z))
```

In this case, the data `z` can have any acceptable Stateflow type.

Replacing Operators with Target Functions

The target function library published by Real-Time Workshop® Embedded Coder™ code generation software allows you to replace a subset of arithmetic operators with target functions. Operator entries of the target function library can specify integral or fixed-point operand and result patterns. Operator entries may be used for the following built-in operators:

```
+
-
*
/
```

For example, you can replace an expression such as `y = u1 + u2` with a target function, provided that `u1`, `u2`, and `y` have types that permit a match with an addition entry in the target function library.

Stateflow chart semantics may limit operator entry matching because it uses the target integer size as its intermediate type in all arithmetic expressions. For example, suppose a Stateflow action contains this arithmetic expression:

```
y = (u1 + u2) % 3
```

This expression computes the intermediate addition into a target integer. If the target integer size is 32 bits, you cannot replace this expression with a target function library operator entry for addition that produces a signed 16-bit result without loss of precision.

To learn how to create and register function replacement tables in a target function library, see “Target Function Libraries” in theReal-Time Workshop Embedded Coder User’s Guide. To select and view target function libraries, see “Selecting and Viewing Target Function Libraries” in the Real-Time Workshop User’s Guide.

Using Special Symbols in Actions

In this section...

Comment Symbols

Use the symbols `%`, `//`, and `/*` to represent comments in Stateflow action language as shown in the following examples:

```
% MATLAB comment line
// C++ comment line
/* C comment line */
```

You can optionally include comments in generated code for an embedded target (see "Real-Time Workshop Pane: Comments" in the Real-Time Workshop Reference) or a Stateflow custom target (see "Configuring a Custom Target" on page 22-51). Stateflow action language comments in generated code are represented with multibyte character code. This means that you can have comments in code with characters for non-English alphabets such as Japanese Kanji characters.

Hexadecimal Notation Symbols

Stateflow action language supports 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 Stateflow action language. Calculations like `n/0`, where `n` is any nonzero real value, result in `inf`.

Line Continuation Symbol, ...

Use the characters `...` at the end of a line of action language to indicate that the expression continues on the next line.

Literal Code Symbol, $

Use `$` characters to mark action language that you want the parser to ignore but you want to appear in the generated code. For example,

```
$
ptr -> field = 1.0;
$
```

The parser is completely disabled during the processing of anything between the `$` characters. Frequent use of literals is discouraged.

MATLAB Display Symbol, ;

Omitting the semicolon after an expression displays the results of the expression in the MATLAB Command Window. If you use a semicolon, the results are not displayed.

Single-Precision Floating-Point Number Symbol, F

Use a trailing `F` to specify single-precision floating-point numbers in Stateflow action language. 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, it is assumed to be double-precision.

Time Symbol, t

Use the letter `t` to represent absolute time inherited from a Simulink signal in simulation targets. For example, the condition `[t - On_time > Duration]`

specifies that the condition is true if the value of `On_time` subtracted from the simulation time `t` is greater than the value of `Duration`.

Note The meaning of `t` for nonsimulation targets is undefined since it is dependent upon the specific application and target hardware.

Calling C Functions in Actions

In this section...

Calling C Library Functions

You can call the following small 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 "Calling the abs Function" on page 10-26.

** You can also replace calls to the C Math Library with target-specific implementations for this subset of functions. For more information, see "Replacing C Math Library Functions with Target-Specific Implementations" on page 10-27

You can call the above C Math Library functions without doing anything special as long as you are careful to call them with the right data types. In case of a type mismatch,Stateflow software replaces the input argument with a cast of the original argument to the expected type. For example, if you call the `sin` function with an integer argument, Stateflow software replaces the argument with a cast of the original argument to a floating-point number of type `double`.

If you call other C library functions not specified above, be sure to include the appropriate `#include...` statement in the **Simulation Target > Custom Code** pane of the Configuration Parameters dialog box. For details, see Chapter 22, "Building Targets".

Calling 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 other than `int32`, 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, be sure to cast its argument or return values to integer types. See "Type Cast Operations" on page 10-19.

Note If you declare `x` in custom code, the standard C `abs` function applies in all cases. For instructions on inserting custom code into Stateflow charts, see Chapter 22, "Building Targets".

Calling min and max Functions

Although `min` and `max` are not C library functions, you can use them 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 Stateflow charts in the Stateflow machine, 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.

Replacing C Math Library Functions with Target-Specific Implementations

You can use the target function library published by Real-Time Workshop Embedded Coder code generation software to replace the default implementations of a subset of C library functions with target-specific implementations (see "Supported Target Library Functions" on page 10-27). When you specify a target function library, Stateflow software generates code that calls the target implementations instead of the associated C library functions. Furthermore, Stateflow software also uses target implementations in cases where the compiler generates calls to math functions, such as in fixed-point arithmetic utilities.

Using Target Function Libraries

To learn how to create and register function replacement tables in a target function library, see "Target Function Libraries" in the Real-Time Workshop Embedded Coder User's Guide. To select and view target function libraries, see "Selecting and Viewing Target Function Libraries" in the Real-Time Workshop User's Guide.

Supported Target Library Functions

You can replace the following C Math Library functions with target-specific implementations:

- `abs`

Note See "Replacing Calls to abs" on page 10-28.

- `acos`
- `asin`
- `atan`
- `ceil`
- `cos`
- `cosh`
- `exp`
- `floor`
- `log`
- `log10`
- `sin`
- `sinh`
- `sqrt`
- `tan`
- `tanh`

Replacing Calls to abs

Stateflow software replaces calls to `abs` with target functions as follows:

For:	Action:
`abs` with floating-point arguments	Replaces with target function
`abs` with integer arguments	Replaces with ANSI C function
`abs` with fixed-point arguments (zero bias)	Replaces with ANSI C function
`abs` with fixed-point arguments (nonzero bias)	Generates error

Calling Custom C Code Functions

You can install your own C code functions for use in the Stateflow action language for simulation and for C code generation.

Specifying Custom C Functions for Simulation

To specify custom C functions for simulation, see Chapter 22, "Building Targets".

Specifying Custom C Functions for Code Generation

To specify custom C functions for code generation, follow these steps:

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

   The Configuration Parameters dialog box appears, displaying the general Real-Time Workshop configuration parameters.

2. In the left pane of the Configuration Parameters dialog box, select **Custom Code** and specify your custom C files as described in "Real-Time Workshop Pane: Custom Code".

Guidelines for Using Custom C Functions in Stateflow Action Language

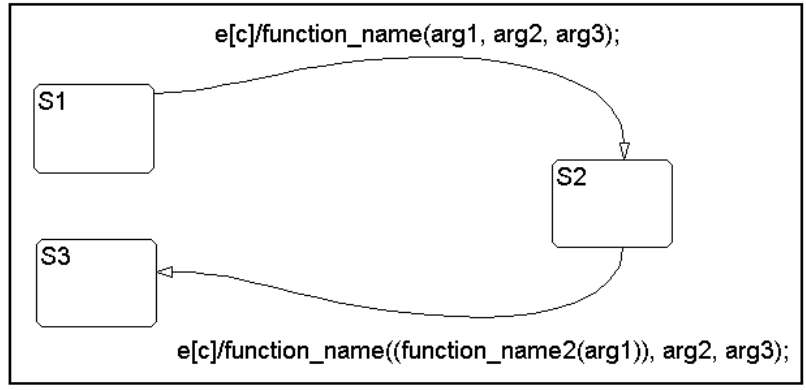
Follow these guidelines when using your own C code functions in the Stateflow action language:

- Define a function by its name, any arguments in parentheses, and an optional semicolon.

- Pass string 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).

Function Call Transition Action Example

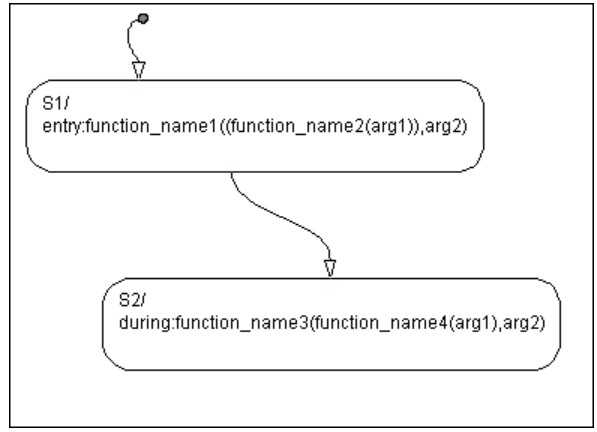
These are example formats of function calls using transition action notation.



If `S1` is active, event `e` occurs, `c` is true, and the transition destination is determined, then a function call is made to `function_name` with `arg1`, `arg2`, and `arg3`. The transition action in the transition from `S2` to `S3` shows a function call nested within another function call.

Function Call State Action Example

These are example formats of function calls using state action notation.



When the default transition into `S1` occurs, `S1` is marked active and then its `entry` action, a function call to `function_name1` with the specified arguments, is executed and completed. If `S2` is active and an event occurs, the `during` action, a function call to `function_name3` with the specified arguments, executes and completes.

Passing 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 "Specifying Chart Properties" on page 16-5).
- If the data type of `x` is `boolean`, you must turn off the coder option **Use bitsets to store state-configuration** (see "How to Optimize Generated Code for Embeddable Targets" on page 22-30).
- If `x` is an array with its first index property set to 0 (see "Setting Data Properties in the Data Dialog Box" on page 8-6), then the function must be called 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.

Using MATLAB Functions and Data in Actions

In this section...

MATLAB Functions and Stateflow Code Generation

You can call MATLAB functions and access MATLAB workspace variables in Stateflow actions, 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

The `ml` namespace operator uses standard dot `(.)` notation to reference MATLAB variables and functions in action language. 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, as shown in the preceding examples. 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 literal string.

  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

You can use the `ml` function to specify calls to MATLAB functions through a string expression in the action language. The format for the `ml` function call uses this notation:

```
ml(evalString, arg1, arg2,...);
```

`evalString` is a string 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 string 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 "Inferring Return Size for ml Expressions" on page 10-41.

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 string 'matfunc(%g, ''abcdefg'', x, %f)' is the evalString shown in the preceding format statement. Stateflow data d1 and d2 are inserted into that string with the format specifiers %g and %f, respectively. The string ''abcdefg'' is a string literal with two single pairs of quotation marks used to enclose it because it is part of the evaluation string, 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

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 is included 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 and Bitwise Operations" on page 10-14. To repeat a warning note in

that section, be sure to use parentheses around power expressions with the ^ operator when you use them in conjunction with other arithmetic operators.

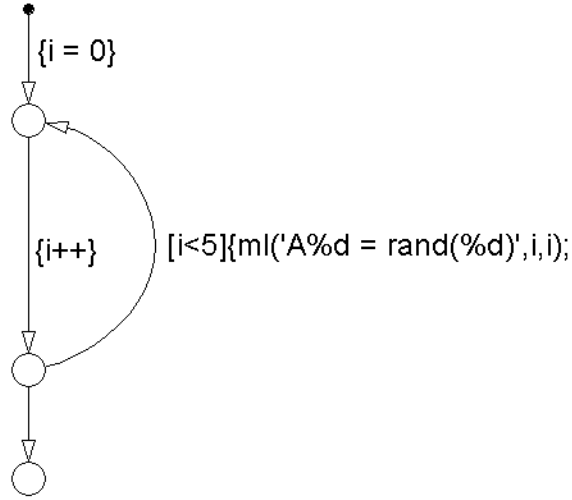
Stateflow software checks expressions for data size mismatches in your action language during parsing of your charts and during run-time. Because the return values for `ml` expressions are not known until run-time, Stateflow software must infer the size of their return values. See "Inferring Return Size for ml Expressions" on page 10-41.

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 example creates four new MATLAB matrices:



  This example demonstrates the use of a Stateflow `for` loop to create 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 is taken and 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 is taken and execution stops.

This results in 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 demonstrated by the following example:

  ```
  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`. 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 "Initializing Data from the MATLAB Base Workspace" on page 8-29).
- 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 "Inferring Return Size for ml Expressions" on page 10-41.

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 Stateflow action language.

- 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 (see Chapter 22, "Building Targets").
- If data of type `ml` contains an array, you can access the elements of the array via indexing with these rules:

  1 You can index only arrays with numerical elements.

  2 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.

  3 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 string 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 string data of type ml*/
```

- `ml` data cannot have a scope outside a Stateflow chart; that is, you cannot define the scope of `ml` data as **Input to 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 Stateflow 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 Stateflow chart and use it for MATLAB computations in Stateflow action language.

As an example, in the following Stateflow action language 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.

Inferring Return Size for ml Expressions

Stateflow expressions using the `ml` namespace operator and the `ml` function are evaluated in the MATLAB workspace at run-time. This means that 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 any of these expressions is used in action language, Stateflow code generation must create temporary Stateflow data, invisible to the user, to hold their intermediate returns for evaluation of the full expression of which they are a part. Because the size of these return values is not known until run-time, Stateflow software must employ context rules to infer their size for the 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 chosen to store it, a size mismatch error appears. To prevent these run-time errors, use these guidelines to write action language statements with MATLAB commands or `ml` data:

1. The return sizes of MATLAB commands or data in an expression must match the return sizes of peer expressions.

   For example, 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.

2. Expressions that return a scalar never produce an error.

   You can combine matrices and scalars in larger expressions because MATLAB commands practice scalar expansion. For example, in the larger expression `ml.x + y`, if `y` is a 3-by-2 matrix and `ml.x` returns a scalar, the resulting value is determined by 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. The same rule applies to subtraction (`-`), multiplication (`*`), division (`/`), and any other binary operations.

3. MATLAB commands or Stateflow data of type `ml` can be members of the following independent levels of expression, for which the return size must be resolved:

   - 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. For example, in the expression `z + func(x + ml.y)`, the size of `ml.y` has nothing to do with 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 is required to be scalar in size. For example, in the expression `x + arr[y]`, the size of `y` has nothing to do with the size of `x` because `y` and `x` are at different levels of expression, and `y` must be a scalar.

4. The return size for an indexed array element access must be a scalar.

   For example, the expression `x[1][1]`, where `x` is a 3-by-2 array, must evaluate to a scalar.

5 MATLAB command or data elements used in an expression for the input argument for a MATLAB function called through the `ml` namespace operator are resolved for size using the rule for peer expressions (preceding rule 1) for the expression itself, because there is no size definition prototype available.

   For example, 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.

6 MATLAB command or data elements used for the input argument for a graphical function in an expression are resolved for size by the function's prototype.

   For example, if the graphical function `gfunc` has the prototype `gfunc(arg1)`, where `arg1` is a 2-by-3 Stateflow data array, then 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.

7 `ml` function calls can take only scalar or string literal arguments. Any MATLAB command or data used to specify an argument for the `ml` function must return a scalar value.

8 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 a single Stateflow data of type `ml`, then the sizes of both left-hand expression and right-hand expression are determined by the right-hand expression.

   For example, 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.

9 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. For example, 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.

10 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.

For example, 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. And even if `ml.y` and `ml.z` return matching sizes at run-time, if they return nonscalar values, a size mismatch error results.

11 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.

For example, the expression `x + ml.str`, where `ml.str` is a string workspace variable, produces a run-time error stating that `ml.str` is not a numeric type.

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.

---

12 There are special cases in which no size checking is done to resolve the size of MATLAB command or data expressions that are members of larger expressions. In the cases shown, use of a singular MATLAB element such as `ml.var`, `ml.func()`, `ml(evalString, arg1, arg2,...)`, Stateflow data of type `ml`, or a graphical function returning a Stateflow data of type `ml`, does not require that size checking be enforced at run-time. In these cases, assignment of a return to the left-hand side of an assignment statement or to a function argument is made without consideration 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, string, 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`. Even though nothing is done to check the size of 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

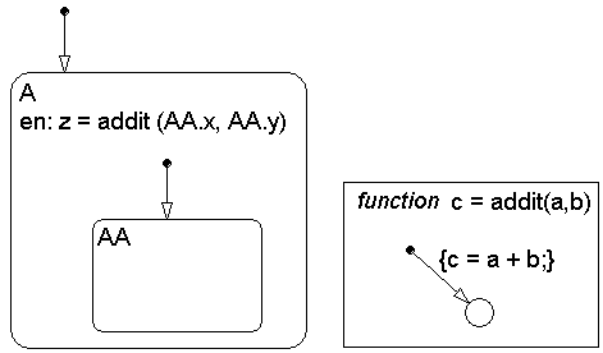
---

Note If inputs in the preceding cases are replaced with non-MATLAB numeric Stateflow data, a conversion to an `ml` type is performed.

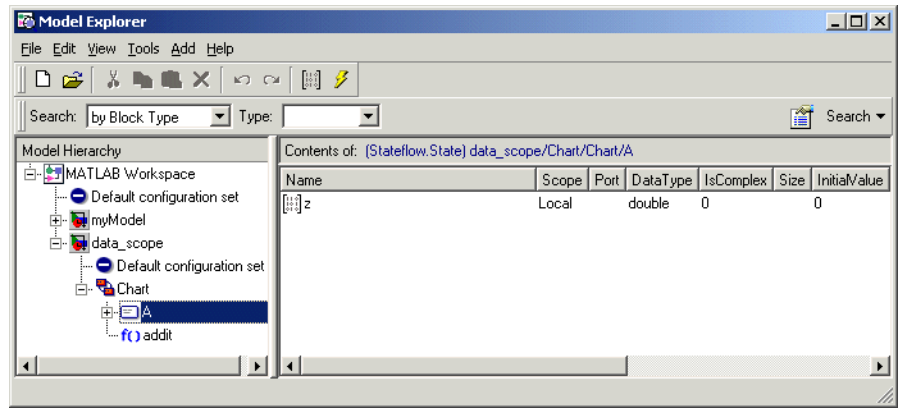
Using Data and Event Arguments in Actions

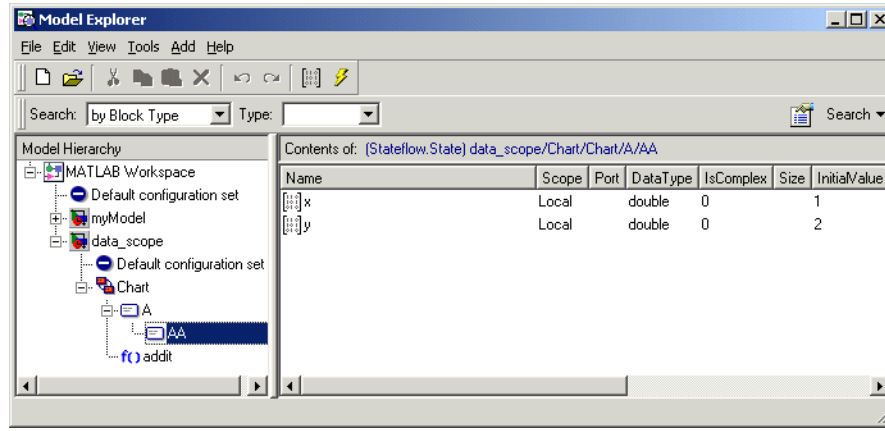
When you use data and event objects as arguments to functions that you call in action language, they are assumed to be defined at the same level in the hierarchy as the action language that references them. If they are not found at that level, Stateflow action language attempts to resolve the object name by searching up the hierarchy. Data or event object arguments that are parented anywhere else must have their path hierarchies defined explicitly.

In the following example, state A calls the graphical function `addit` to add the Stateflow data `x` and `y` and store the result in data `z`.



The following Model Explorer windows show that the data `z` is defined for state A, but the data `x` and `y` are defined for state AA, a substate of A.





The call to function `addit` from state `A` can resolve `z` because it is owned by `A`. However, it cannot resolve `x` and `y` by looking above state `A`. Therefore, the function call must reference `x` and `y` explicitly to their owner, state `AA`.

For information about functions you can call in Stateflow action language that use data as arguments, see these sections:

- "Using Graphical Functions to Extend Actions" on page 7-27
- "Calling C Functions in Actions" on page 10-25
- "Using MATLAB Functions and Data in Actions" on page 10-33

Only operators for event-based temporal logic take events as arguments. See "Using Temporal Logic in State Actions and Transitions" on page 10-56.

Using Arrays in Actions

In this section...

Array Notation

A Stateflow action 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 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 Stateflow actions 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 Stateflow charts, from Simulink models, and from custom code.

As an exception to zero-based indexing, **scalar expansion** is available within the action language. 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;
```

Note For more information, see Chapter 11, "Using Vectors and Matrices in Stateflow Charts".

Arrays and Custom Code

Stateflow action language provides the same syntax for Stateflow arrays and custom code arrays.

Note Any array variable that is referred to in a Stateflow chart but is not defined in the Stateflow hierarchy is identified as a custom code variable.

Broadcasting Events in Actions

In this section...

About Events in Actions

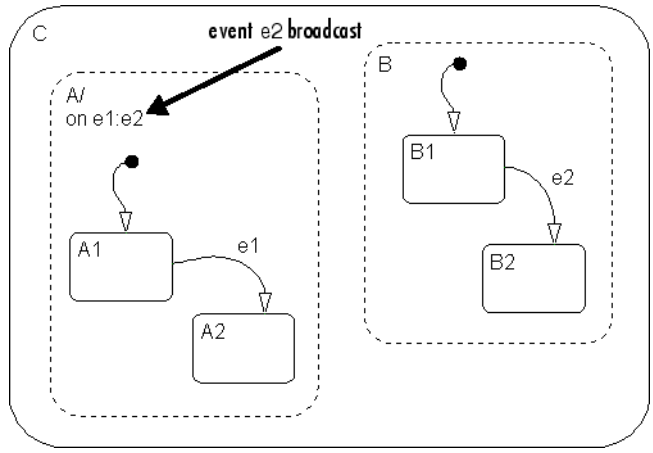
You can specify an event to be broadcast in the action language. Events have hierarchy (a parent) and scope. The parent and scope together define a range of access to events. It is primarily the event's parent that determines who can trigger on the event (has receive rights). See "How Events Work in Stateflow Charts" on page 9-2 for more information.

Event Broadcasting

You can broadcast events in the action language to synchronize AND (parallel) states. Recursive event broadcasts can lead to definition of cyclic behavior. You can detect cyclic behavior only during simulation.

Event Broadcast State Action Example

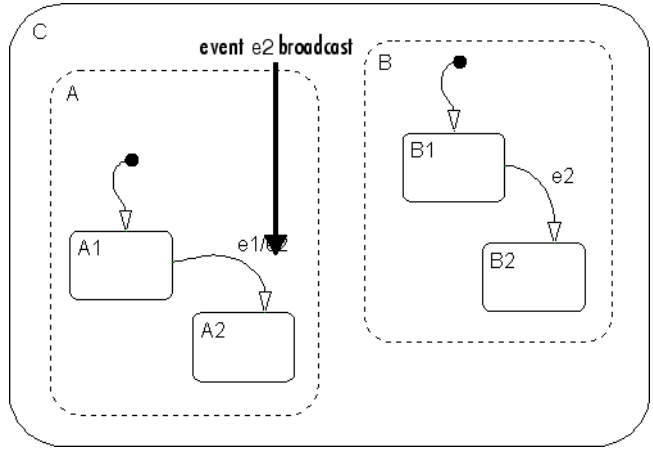
The following is an example of the state action notation for an event broadcast:



See “Event Broadcast State Action Example” on page 3-93 for information on the semantics of this notation.

Event Broadcast Transition Action Example

The following is an example of transition action notation for an event broadcast.



See “Event Broadcast Transition Action with a Nested Event Broadcast Example” on page 3-96 for information on the semantics of this notation.

Directed Event Broadcasting

You can specify a directed event broadcast in actions. Using a directed event broadcast, you can broadcast a specific event to a specific receiver state in the same chart. The receiving state must be active at the time the broadcast is executed to receive and potentially act on the directed event broadcast.

Directed event broadcasting is a more efficient means of synchronization among parallel (AND) states. Using directed event broadcasting improves the efficiency of the generated code. As is true in undirected event broadcasting, recursive event broadcasts can lead to definition of cyclic behavior.

Note An action in one chart cannot broadcast events to states defined in another chart.

Directed Event Broadcasting Using send

The format of the directed event broadcast with `send` is as follows:

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

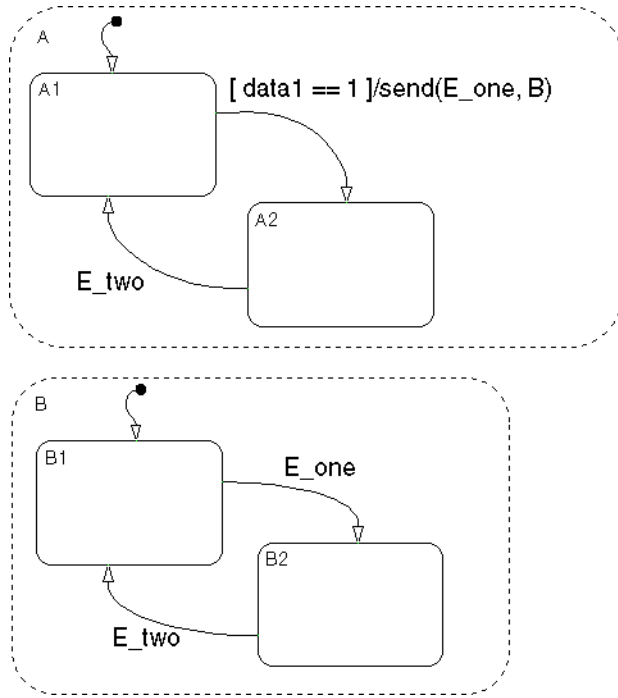
where `event_name` is broadcast to `state_name` and any offspring of that state in the hierarchy. The event sent must be visible to both the sending state and the receiving state (`state_name`).

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)
```

Note Do not use the chart name in the full hierarchy path to a state. Formal chart names include the subsystem in which they are located. For example, in the demo model `fuelsys`, the chart `control logic` is in the subsystem `fuel rate controller`. This means that the formal name for the chart `control logic` is `fuel rate controller/control logic`. This name includes the forward slash character (`'/'`), which is not a valid character in Stateflow identifiers.

This example of a directed event broadcast uses the `send(event_name,state_name)` transition action.



In this example, event `E_one` must be visible in both `A` and `B`. See "Directed Event Broadcast Using Send Example" on page 3-105 for information on the semantics of this notation.

Directed Event Broadcasting Using Qualified Event Names

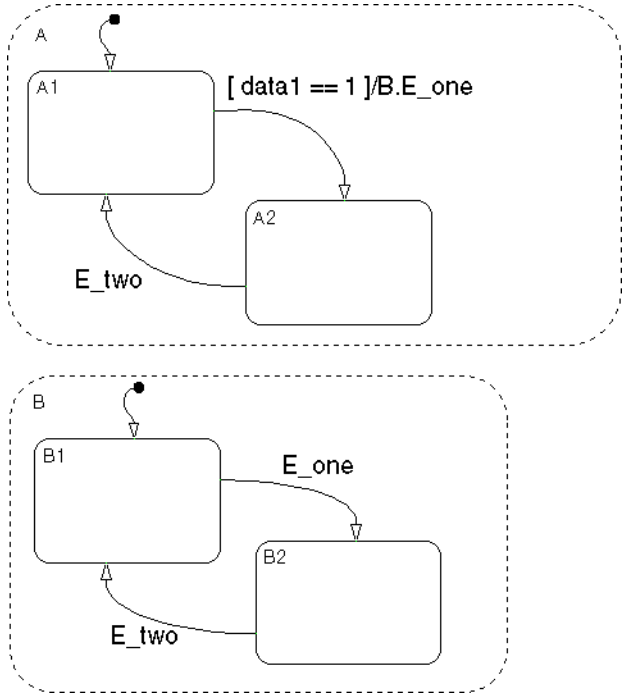
The format of the direct event broadcast using qualified event names is as follows:

```
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 sent is visible only to the receiving state (`state_name`).

The `state_name` argument can also include a full hierarchy path to the receiving state. Again, do not use the chart name in the full path name of the state.

The following example illustrates the use of a qualified event name in a directed event broadcast.



In this example, event `E_one` is visible only to state `B`. See “Directed Event Broadcasting Using Qualified Event Names Example” on page 3-107 for information on the semantics of this notation.

Using Temporal Logic in State Actions and Transitions

In this section...

What Is Temporal Logic?

Temporal logic controls execution of a Stateflow 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 keeps track of recurring events, and 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

These rules apply to the use of temporal logic operators:

- 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.
- Temporal logic operators can appear only in state actions and in transitions that originate from states.

  Note This restriction means that you cannot use temporal logic operators in conditions on default transitions or flow graph transitions.

  Every temporal logic operator has an associated state: the state in which the action appears or from which the transition originates.
- You must use event notation (see "Notations for Event-Based Temporal Logic" on page 10-61) to express event-based temporal logic in state actions.

Operators for Event-Based Temporal Logic

For event-based temporal logic, use the operators as described below.

Operator	Syntax	Description
`after`	`after(n, E)` where `E` is the base event for the `after` operator and `n` is one of the following: • 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.

Operator	Syntax	Description
`before`	`before(n, E)` where `E` is the base event for the `before` operator and `n` is one of the following: • 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.
`at`	`at(n, E)` where `E` is the base event for the `at` operator and `n` is one of the following: • A positive integer • An expression that evaluates to a positive integer value	Returns true only at the 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, `at(n, tick)` or `at(n, wakeup)` returns true if the chart has woken up for the n^{th} time since activation of the associated state. Resets the counter for `E` to 0 each time the associated state reactivates.

Operator	Syntax	Description
`every`	`every(n, E)` where `E` is the base event for the `every` operator and `n` is one of the following: • 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 `n` times since activation of the associated state. Resets the counter for `E` to 0 each time the associated state reactivates. Therefore, this operator is useful only in state actions and not in transitions.
`temporalCount`	`temporalCount(E)` where `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. Resets the counter for `E` to 0 each time the associated state reactivates.

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.

Notations 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.

Notation	Usage	Example	Description
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.
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 the use of conditional notation.

Operators for Absolute-Time Temporal Logic

For absolute-time temporal logic, use the operators as described below.

Operator	Syntax	Description
`after`	`after(n, sec)` where `n` is any positive number or expression and `sec` is a keyword that denotes the simulation time elapsed since activation of the associated state.	Returns true if `n` seconds of simulation time have elapsed since activation of the associated state. Otherwise, the operator returns false. Resets the counter for `sec` to 0 each time the associated state reactivates.

Operator	Syntax	Description
`before`	`before(n, sec)` where `n` is any positive number or expression and `sec` is a keyword that denotes the simulation time elapsed since activation of the associated state.	Returns true if fewer than `n` seconds of simulation time have elapsed since activation of the associated state. Otherwise, the operator returns false. Resets the counter for `sec` to 0 each time the associated state reactivates.
`temporalCount`	`temporalCount(sec)` where `sec` is a keyword that denotes the simulation time elapsed since activation of the associated state.	Counts and returns the number of seconds of simulation time that have elapsed since activation of the associated state. Resets the counter for `sec` to 0 each time the associated state reactivates.

Defining Time Delays

Use the keyword `sec` to define simulation time that has elapsed since activation of a state. For example, the continuous-time chart below defines two absolute time delays in transitions. (See Chapter 13, "Modeling Continuous-Time Systems in Stateflow Charts" for information about modeling continuous-time systems.)

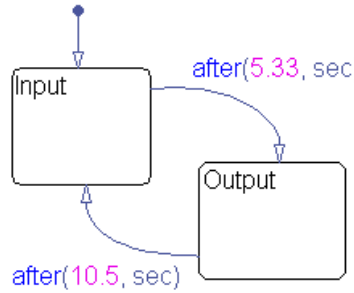


Chart execution occurs as follows:

1. When the chart awakens, the state `Input` activates first.
2. After 5.33 seconds of simulation time pass, the transition from `Input` to `Output` occurs.
3. The state `Input` deactivates, and the state `Output` activates.
4. After another 10.5 seconds of simulation time pass, the transition from `Output` to `Input` occurs.
5. The state `Output` deactivates, and the state `Input` activates.
6. Steps 2–5 repeat, until the simulation ends.

Note Use the keyword `sec` only in state actions and in transitions that originate from states.

Examples of Absolute-Time Temporal Logic

These examples illustrate usage of 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;`	The `temp` variable becomes `LOW` after 12.3 seconds of simulation time have passed, since activation of the state.
`after`	Transition	`after(12.34, sec)`	A transition out of the associated state occurs after 12.34 seconds of simulation time have passed, since activation of the state.

Operator	Usage	Example	Description
`before`	Transition	`[temp > 75 && before(12.34, sec)]`	A transition out of the associated state occurs if the variable `temp` exceeds 75 and fewer than 12.34 seconds have elapsed since activation of the state.
`temporalCount`	State action (exit)	`ex:  y = temporalCount(sec);`	This action counts and returns the number of seconds of simulation time that pass between activation and deactivation of the state.

Running a Model That Demonstrates Absolute-Time Temporal Logic

The `sf_boiler` demo illustrates the use of absolute-time temporal logic to implement a bang-bang controller. To run the model, follow these steps:

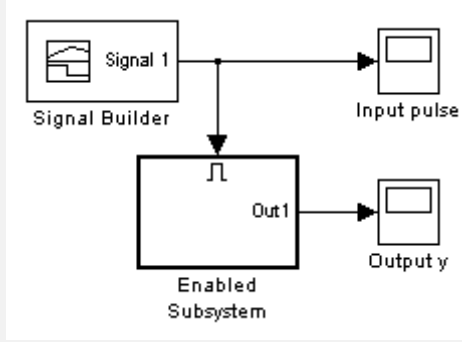
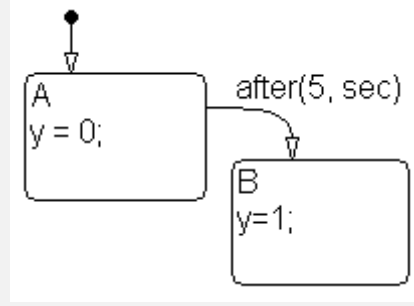
1. Type `sf_boiler` at the MATLAB command prompt.
2. Select **Simulation > Start** in the Simulink model window.

Using Absolute-Time Temporal Logic in a Conditionally Executed Subsystem

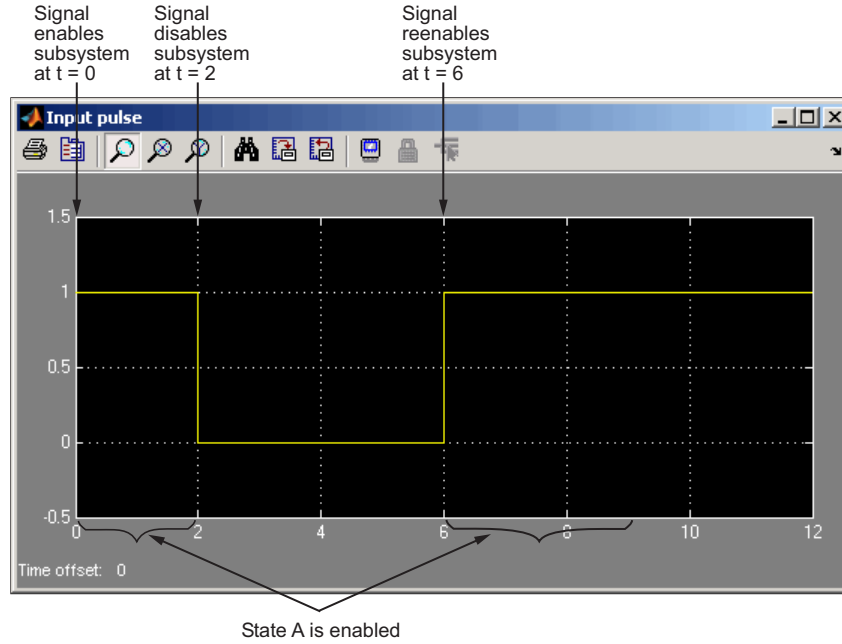
You can use absolute-time temporal logic in a chart that resides in a *conditionally executed* subsystem. (See "Creating Conditional Subsystems" in the Simulink documentation for details.) When the subsystem is disabled, the chart becomes inactive and the temporal logic operator pauses while the chart is asleep. The operator does not continue to count simulation time until the subsystem is reenabled and the chart is awake.

Example of Absolute-Time in an Enabled Subsystem

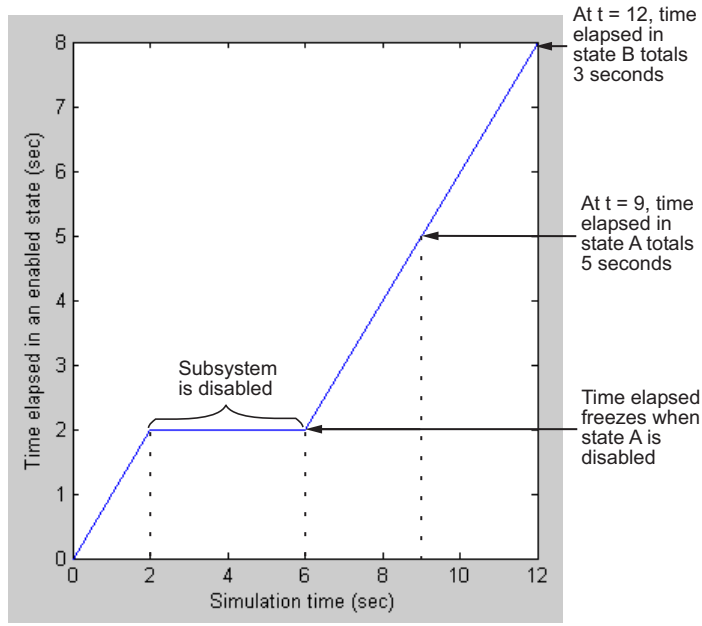
Suppose you have an enabled subsystem that contains a chart with the `after` operator. In the subsystem, the parameter **States when enabling** is set to `held`.

Model with Enabled Subsystem	Chart in Enabled Subsystem
	

The Signal Builder block provides this input signal to the subsystem.


Signal enables subsystem at t = 0
Signal disables subsystem at t = 2
Signal reenables subsystem at t = 6
Input pulse
1.5
1
0.5
0
-0.5
0
2
4
6
8
10
12
Time offset: 0
State A is enabled

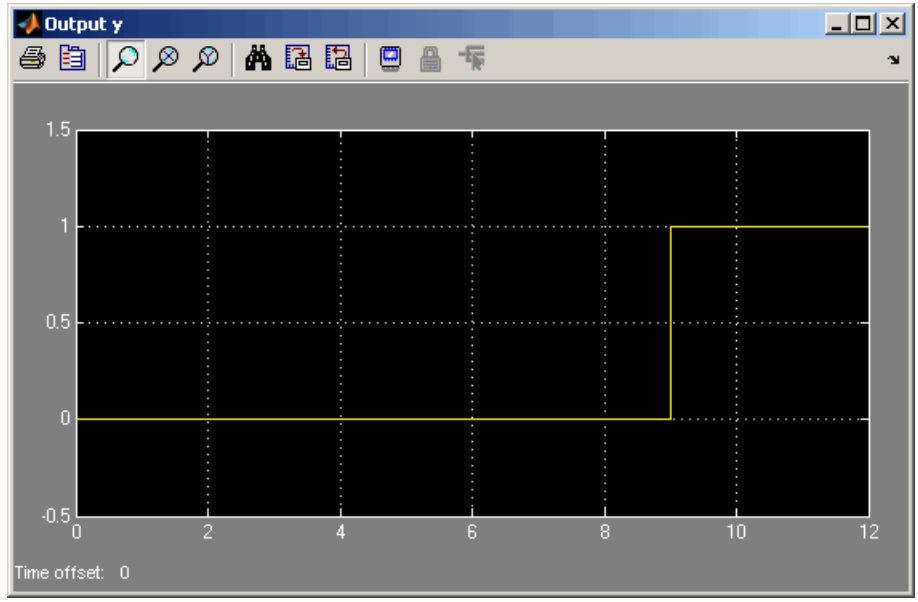
The total time elapsed in an enabled state (both A and B) is as follows.



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. Note that 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 Stateflow chart reinitializes completely when the subsystem is reenabled. In other words, default transitions execute and any temporal logic counters reset to 0.

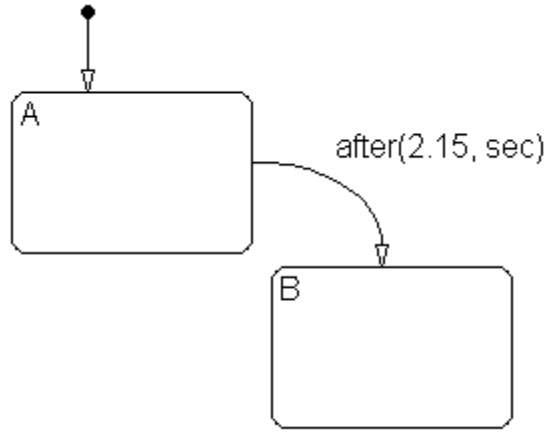
Note These semantics also apply to the `before` operator.

How Sample Time Affects Chart Execution

If a Stateflow chart has a discrete sample time, any action in the chart occurs at integer multiples of this sample time.

A Simple Example

Suppose you have a chart with a discrete sample time of 0.1 seconds:



State `A` becomes active at time t = 0, and the transition to state `B` occurs at t = 2.2 seconds. This behavior applies because the Simulink solver does not wake the chart at exactly t = 2.15 seconds. Instead, the solver wakes the chart at integer multiples of 0.1 seconds, such as t = 2.1 and 2.2 seconds.

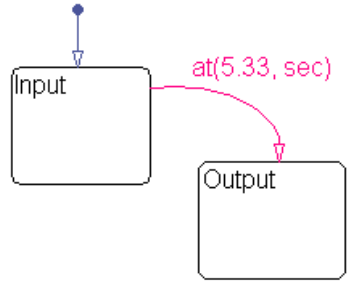
Note This behavior also applies to the `before` operator.

Tips for Using Absolute-Time Temporal Logic

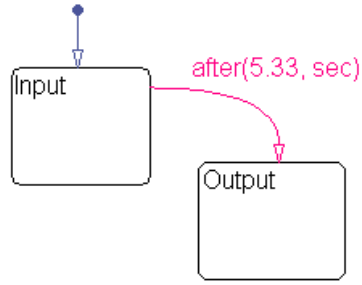
Use the after Operator to Replace the at Operator

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)`.



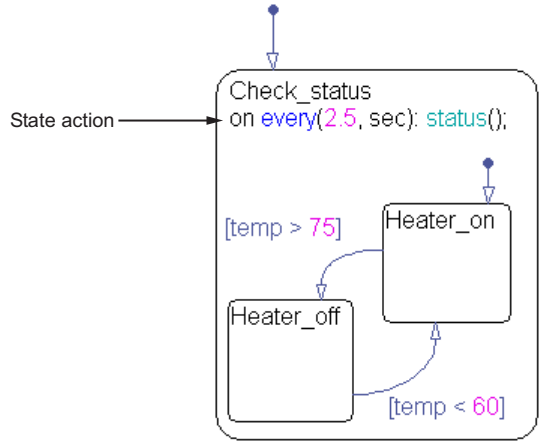
Change the transition to `after(5.33, sec)`, as shown below.



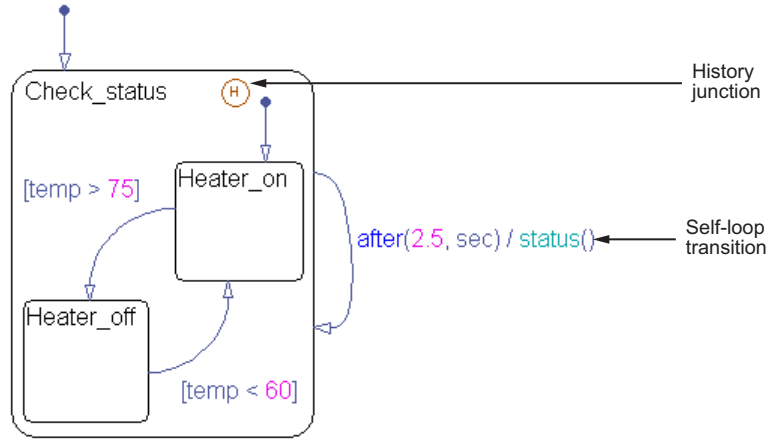
Use an Outer Self-Loop Transition with the after Operator to Replace the every Operator

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 below.



You must also add a history junction in the state so that the chart remembers the state settings prior to each self-loop transition. (See “Using History Junctions to Extend Charts and States” on page 7-2.)

Using Change Detection in Actions

In this section...

About Change Detection

A Stateflow chart can detect changes in the following types of chart data from one time step to the next:

- Inputs
- Outputs
- Local variables
- Data bound to Simulink data store memory

  (For more information, see "Sharing Global Data with Simulink Models" on page 8-32.)

For each of these types of data, you can use operators that detect the following changes:

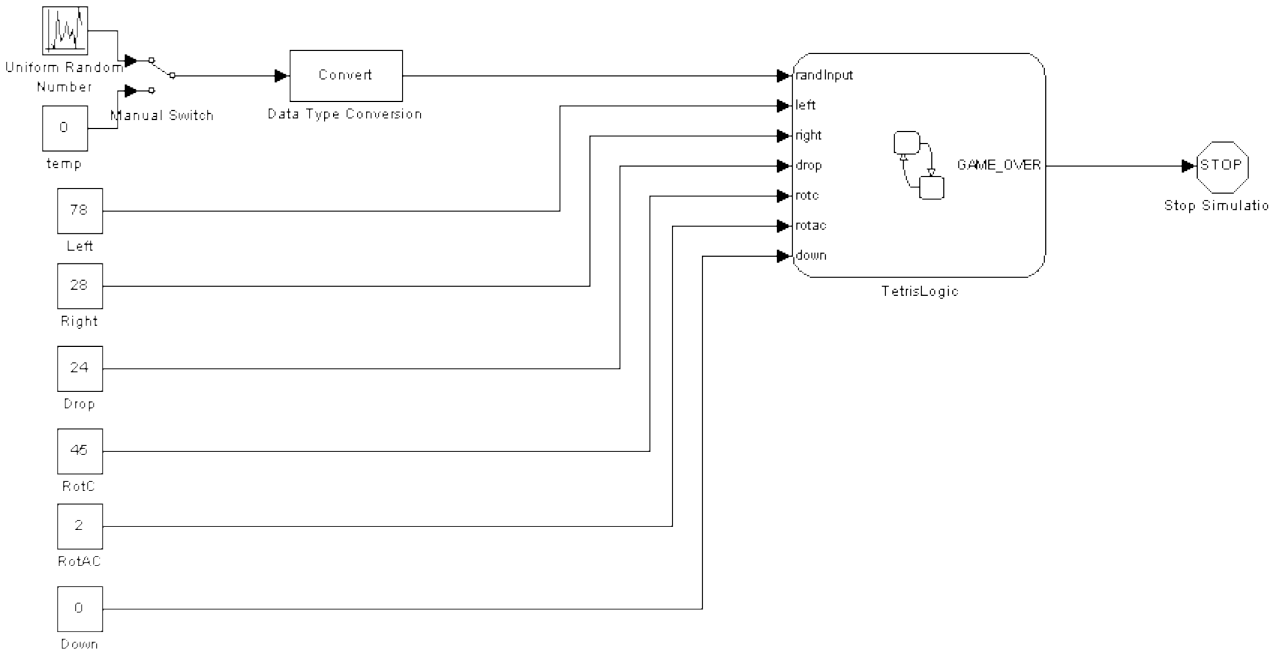
Type of Change	Operator
Data changes value from the beginning of the last time step to the beginning of the current time step.	See "hasChanged Operator" on page 10-82.

Type of Change	Operator
Data changes from a specified value at the beginning of the last time step to a different value at the beginning of the current time step.	See "hasChangedFrom Operator" on page 10-83.
Data changes to a specified value at the beginning of the current time step from a different value at the beginning of the last time step.	See "hasChangedTo Operator" on page 10-84.

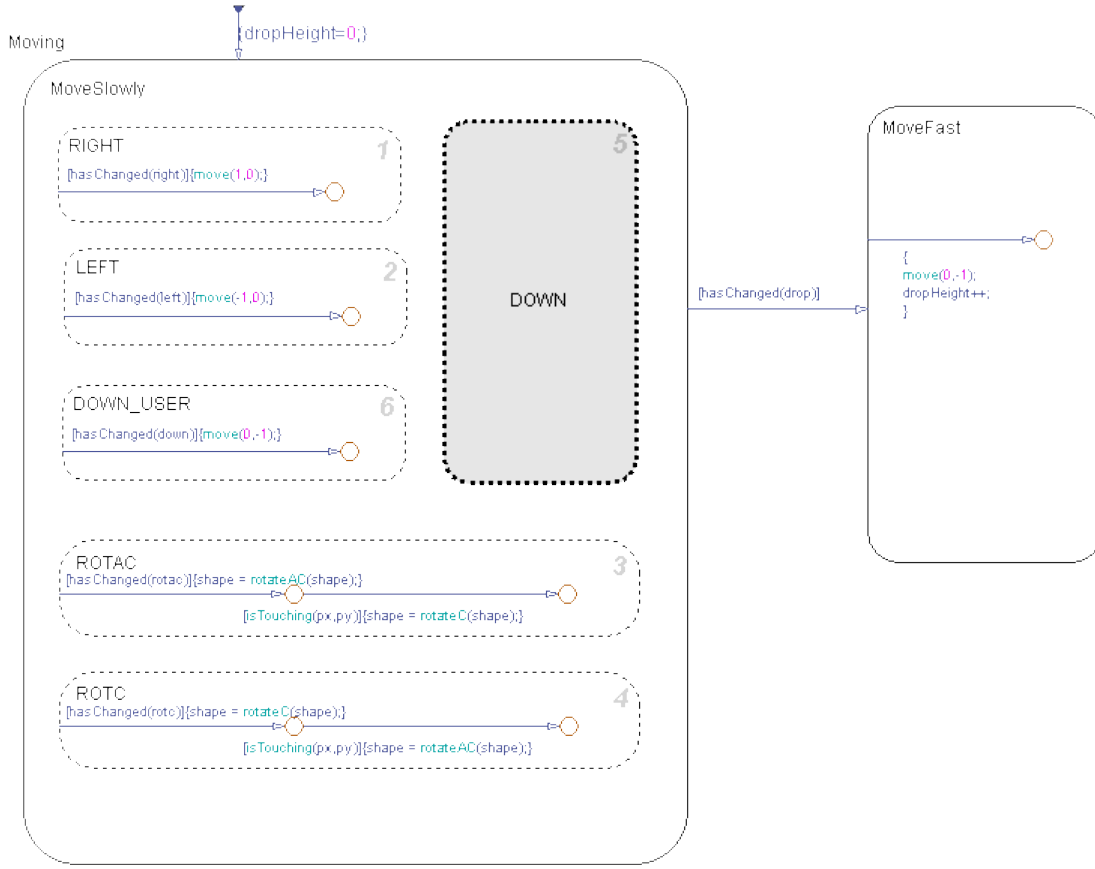
Change detection operators return 1 if the data value changes or 0 if there is no change. See "Change Detection Operators" on page 10-81.

Running a Model That Demonstrates Change Detection

Stateflow software ships with a model `sf_tetris2` that demonstrates how you can detect asynchronous changes in inputs — in this case, user keystrokes — to manipulate a Tetris shape as it moves through the playing field. The Stateflow chart TetrisLogic implements this logic:



TetrisLogic contains a subchart called Moving that calls the operator `hasChanged` to determine when users press any of the Tetris control keys, and then moves the shape accordingly. Here is a look inside the subchart:



To run the demo model from the MATLAB workspace, follow these steps:

1. At the MATLAB command prompt, type:

   ```
   demos
   ```

   The MATLAB Help Browser opens the Demos tab in the Help Navigator pane.

2. In the Help Navigator pane, navigate to **Simulink > Stateflow > General Applications**.

3 In the right contents pane, click **Tetris**.

A description of the Tetris demo model appears.

4 In the upper right corner of the contents pane, click the link **Open this model**.

The model opens on your desktop.

Tip You can also open the model by typing `sf_tetris2` at the MATLAB command prompt.

How Change Detection Works

A Stateflow chart detects changes in chart data by evaluating values at time step boundaries. That is, 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 automatically double-buffers these values in local variables, as follows:

Local Buffer:	Stores:
var_name`_prev`	Value of data at the beginning of the last time step
var_name`_start`	Value of data at the beginning of the current time step

---

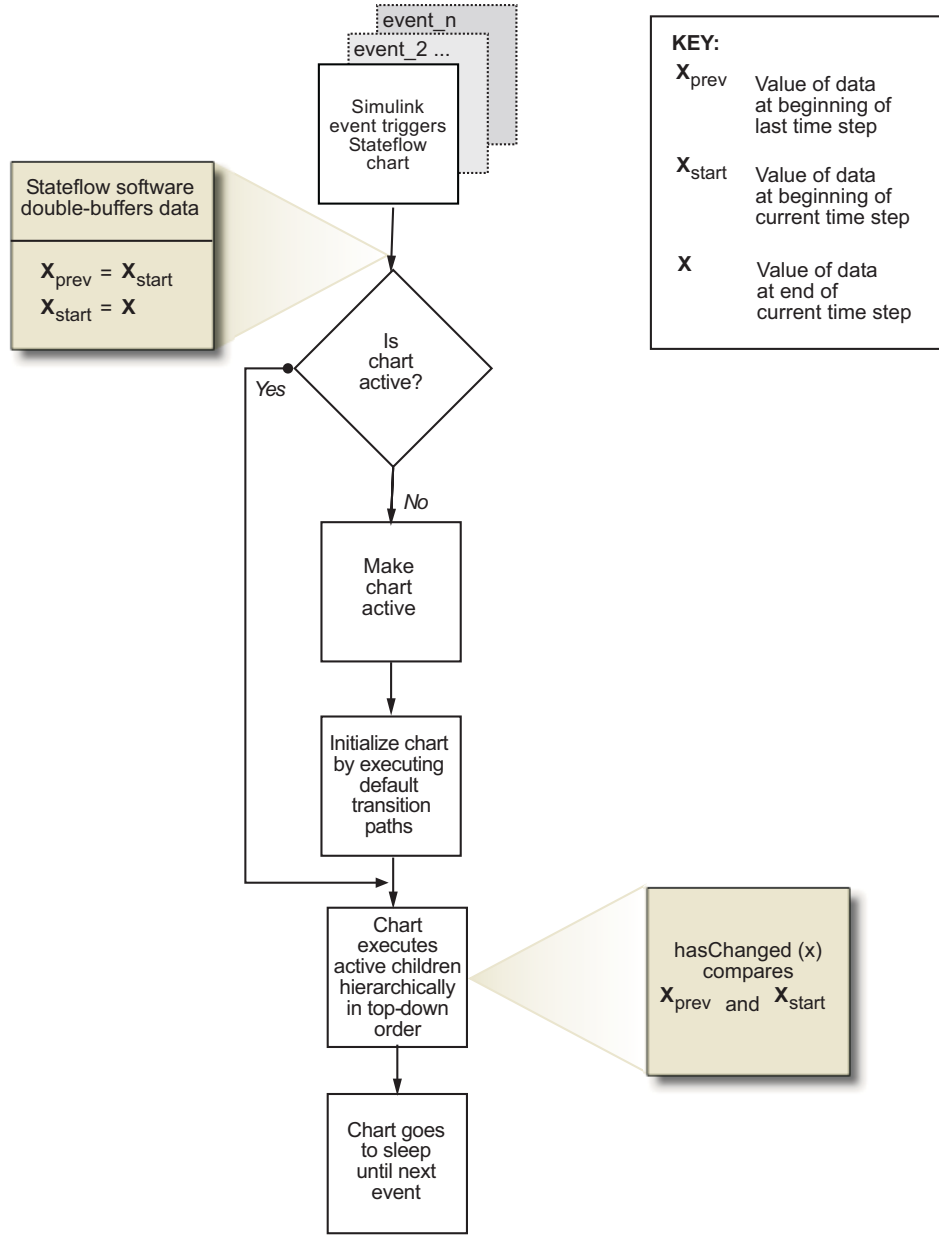
Note Double-buffering occurs once per time step except if multiple input events occur in the same time step. Then, double-buffering occurs once per input event (see "Handling Changes When Multiple Input Events Occur" on page 10-81).

When you invoke change detection operations in an action, Stateflow software performs the following operations:

1. Double-buffers data values *after* a Simulink event triggers the chart, but *before* the chart begins execution.

2. Compares values in `_prev` and `_start` buffers. If the values match, the change detection operator returns 0 (no change); otherwise, it returns 1 (change).

The following diagram places these tasks in the context of the chart life cycle:



The fact that buffering occurs before chart execution has implications for change detection in the following scenarios:

- "Handling Transient Changes in Local Variables" on page 10-81
- "Handling Changes When Multiple Input Events Occur" on page 10-81

Handling Transient Changes in Local Variables

Stateflow software attempts to filter out transient changes in local chart variables by evaluating their values only at time boundaries (see "How Change Detection Works" on page 10-78). This behavior means that the software evaluates the specified local variable only once at the end of the execution step and, therefore, returns a consistent result. That is, the return value 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. In this case, the operator `hasChanged(temp)` returns 0 for the next time step, indicating that no change occurred. For more information, see "Change Detection Operators" on page 10-81.

Handling Changes When Multiple Input Events Occur

When multiple input events occur in the same time step, Stateflow software updates the `_prev` and `_start` buffers once per event. In this way, a chart detects changes between input events, even if the changes occur more than once in a given time step.

Change Detection Operators

Change detection operators check for changes in chart inputs, outputs, and local variables, and in Stateflow data that is bound to Simulink data store memory.

You can invoke change detection operators wherever you call built-in functions in a chart — in state actions, transition actions, condition actions, and conditions. There are three change detection operators:

- "hasChanged Operator" on page 10-82

hasChanged Operator

The `hasChanged` operator detects any change in Stateflow data since the last time step, using the following heuristic:

$$hasChanged(x) = \begin{cases} 1 \; if \; x_{prev} \neq x_{start} \\ 0 \; otherwise \end{cases}$$

where x_{start} represents the value at the beginning of the current time step and x_{prev} represents the value at the beginning of the previous time step.

Syntax.

```
hasChanged ( u )
hasChanged ( m [ expr ] )
hasChanged ( s [ expr ] )
```

where `u` is a scalar or matrix variable, `m` is a matrix, and `s` is aggregate data.

The arguments *u*, *m*, and `s` must be one of the following data types:

- Input, output, or local variable in a Stateflow chart

> **Note** If you enable the chart option **Initialize Outputs Every Time Chart Wakes Up**, do not use an output as the argument of the `hasChanged` operator. With this option enabled, the operator always returns `0` (or `false`) for outputs, so there is no reason to use change detection.

- Stateflow data that is bound to Simulink data store memory

The arguments cannot be expressions or custom code variables.

Description. `hasChanged ( u )` returns `1` if *u* changes value since the last time step. If *u* is a matrix, `hasChanged` returns `1` if *any* element of *u* changes value since the last time step.

`hasChanged ( m [ expr ] )` returns `1` if the value at location `expr` of matrix `m` changes value since the last time step. `expr` can be an arbitrary expression that evaluates to a scalar value.

`hasChanged ( s [ expr ] )` returns `1` if the value at location `expr` of aggregate data `s` has changed since the last time step. `s` must be a fully qualified name, such as `u.foo.bar`, which resolves to an aggregate data type such as a structure or bus signal. `expr` can be an arbitrary expression that evaluates to a scalar value.

All forms of `hasChanged` return zero if a chart writes to the data, but does not change its value.

hasChangedFrom Operator

The `hasChangedFrom` operator detects when Stateflow data changes *from* a specified value since the last time step, using the following heuristic:

$$hasChangedFrom(x, x_0) = \begin{cases} 1 \; if \; x_{prev} \neq x_{start} \; and \; x_{prev} = x_0 \\ 0 \; otherwise \end{cases}$$

where x_{start} represents the value at the beginning of the current time step and x_{prev} represents the value at the beginning of the previous time step.

Syntax.

```
hasChangedFrom ( u , v )
hasChangedFrom ( m [ expr ], v )
hasChangedFrom ( s [ expr ], v )
```

where `u` is a scalar or matrix variable, `m` is a matrix, and `s` is aggregate data.

The arguments `u`, `m`, and `s` must be one of the following data types:

- Input, output, or local variable in a Stateflow chart

Note If you enable the chart option **Initialize Outputs Every Time Chart Wakes Up**, do not use an output as the first argument of the `hasChangedFrom` operator. With this option enabled, the operator always returns `0` (or `false`) for outputs, so there is no reason to use change detection.

- Stateflow data that is bound to Simulink data store memory

Note The first arguments *u*, *m*, and *s* cannot be expressions or custom code variables. The second argument *v* can be an expression. However, if the first argument is a matrix variable, then *v* must resolve to a scalar value or a matrix value with the same dimensions as the first argument.

Description. `hasChangedFrom ( u, v )` returns 1 if *u* changes from the value specified by *v* since the last time step. If *u* is a matrix variable whose elements all equal the value specified by *v*, `hasChangedFrom` returns 1 if one or more elements of the matrix changes to a different value in the current time step.

`hasChangedFrom ( m [ expr ], v)` returns `1` if the value at location *expr* of matrix *m* changes from the value specified by *v* since the last time step. *expr* can be an arbitrary expression that evaluates to a scalar value.

`hasChangedFrom ( s [ expr ], v)` returns `1` if the value at location *expr* of aggregate data *s* changes from the value specified by *v* since the last time step. *s* must be a fully qualified name, such as `u.foo.bar`, which resolves to an aggregate data type such as a structure or bus signal. *expr* can be an arbitrary expression that evaluates to a scalar value.

hasChangedTo Operator

The `hasChangedTo` operator detects when Stateflow data changes to a specified value since the last time step, using the following heuristic:

$$hasChangedTo(x, x_0) = \begin{cases} 1 \; if \; x_{prev} \neq x_{start} \; and \; x_{start} = x_0 \\ 0 \; otherwise \end{cases}$$

where x_{start} represents the value at the beginning of the current time step and x_{prev} represents the value at the beginning of the previous time step.

Syntax.

```
hasChangedTo ( u , v )
hasChangedTo ( m [ expr ], v )
hasChangedTo ( s [ expr ], v )
```

where *u* is a scalar or matrix variable, *m* is a matrix, and *s* is aggregate data.

The arguments *u*, *m*, and *s* must be one of the following data types:

- Input, output, or local variable in a Stateflow chart

  Note If you enable the chart option **Initialize Outputs Every Time Chart Wakes Up**, do not use an output as the first argument of the `hasChangedTo` operator. With this option enabled, the operator always returns `0` (or `false`) for outputs, so there is no reason to use change detection.

- Stateflow data that is bound to Simulink data store memory

Note The first arguments *u*, *m*, and *s* cannot be expressions or custom code variables. The second argument *v* can be an expression. However, if the first argument is a matrix variable, then *v* must resolve to either a scalar value or a matrix value with the same dimensions as the first argument.

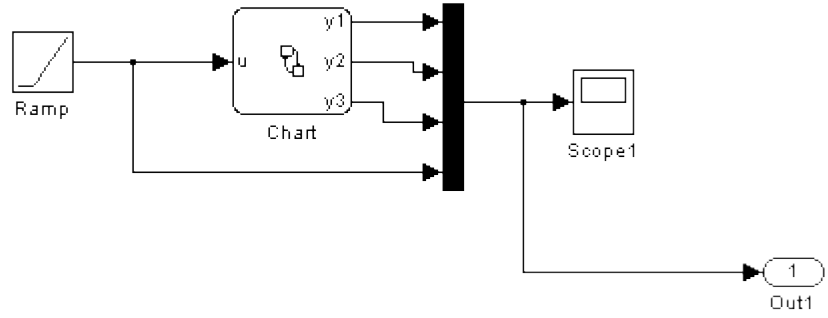
Description. `hasChangedTo ( u, v )` returns 1 if *u* changes to the value specified by *v* in the current time step. If *u* is a matrix variable, `hasChangedTo` returns 1 if any its of its elements changes value so that all elements of the matrix equal the value specified by *v* in the current time step.

`hasChangedTo ( m [ expr ], v)` returns `1` if the value at location *expr* of matrix *m* changes to the value specified by *v* in the current time step. *expr* can be an arbitrary expression that evaluates to a scalar value.

`hasChangedTo ( s [ expr ], v)` returns `1` if the value at location *`expr`* of aggregate data *`s`* changes to the value specified by *`v`* in the current time step. *`s`* must be a fully qualified name, such as `u.foo.bar`, which resolves to an aggregate data type such as a structure or bus signal. *`expr`* can be an arbitrary expression that evaluates to a scalar value.

Change Detection Example

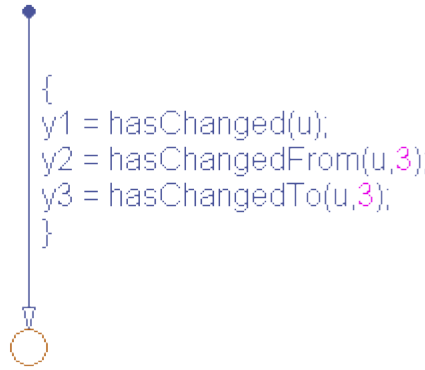
The following model shows how to use the `hasChanged`, `hasChangedFrom`, and `hasChangedTo` operators to detect specific changes in an input signal. In this example, a Ramp block sends a discrete, increasing time signal to a Stateflow chart, as follows:



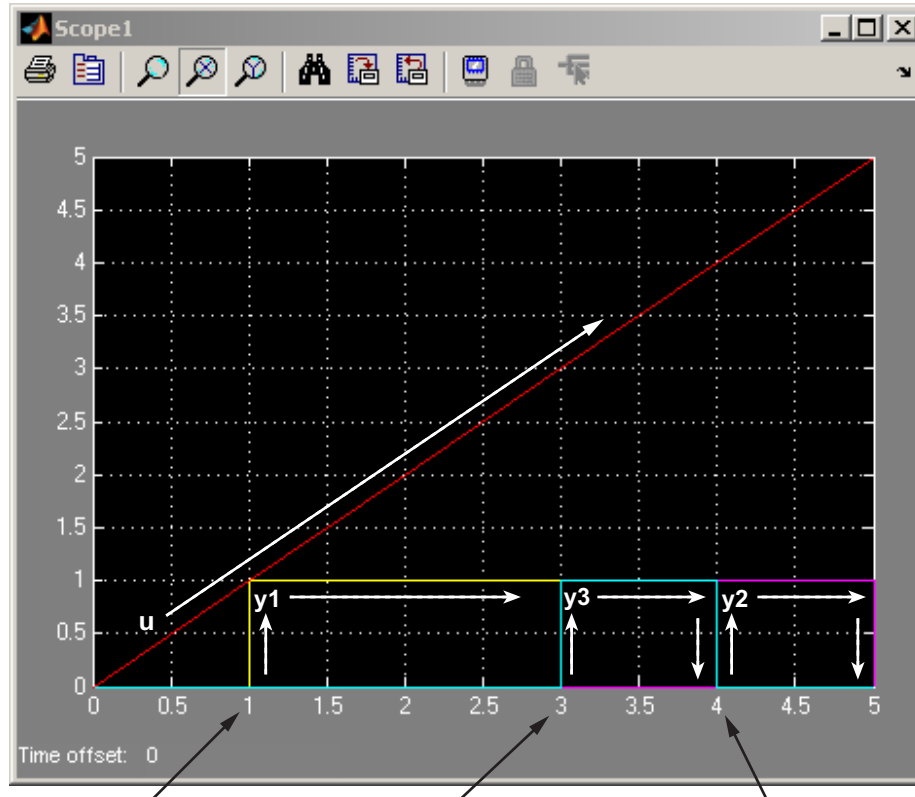
The model uses a fixed-step solver with a step size of 1. The signal increments by 1 at each time step. The chart analyzes the input signal for the following changes at each time step:

- Any change from the previous time step
- Change to the value 3
- Change from the value 3

To check the signal, the chart calls three change detection operators in a transition action, and outputs the return values as `y1`, `y2`, and `y3`, as follows:



During simulation, the outputs `y1`, `y2`, and `y3` represent changes in the input signal, as shown in this scope:



*y*1
transitions to 1 at T1;
stays at 1
because u keeps
increasing

*y*3
transitions to 1 at T3
when u changes to 3;
transitions back to 0
at T4 when u increases
from 3 to 4

*y*2
transitions to 1 at T4
when u changes from 3 to 4;
transitions back to 0
at T5 when u increases
from 4 to 5

Checking State Activity

In this section...

When to Check State Activity

Check state activity when you have substates in parallel states that can be active at the same time. For example, checking state activity allows you to synchronize substates in two parallel states.

How to Check State Activity

Use the `in` operator to check if a state is active. You can use this operator in state actions and transitions that originate from states.

The in Operator

Purpose

Checks if a state is active in a given time step during chart execution.

Syntax

```
in(S)
```

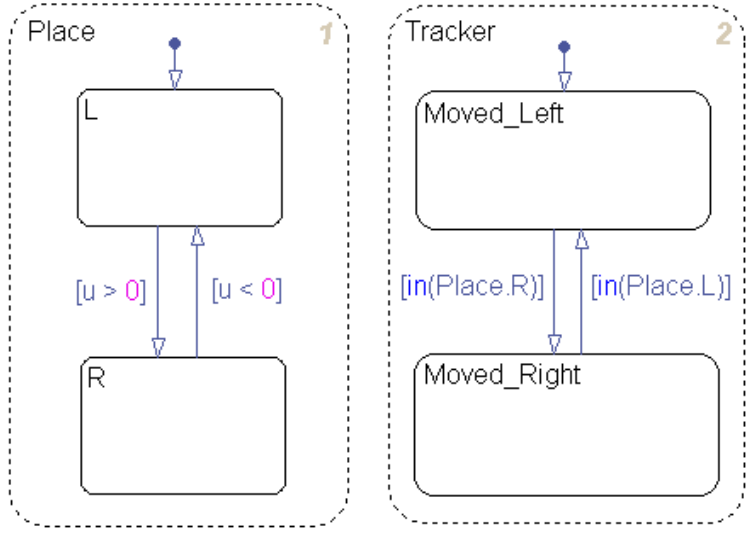
where `S` is a fully qualified state.

Description

The `in` operator is true and returns a value of 1 whenever state `S` is active; otherwise, it returns a value of 0.

Example

This example illustrates the use of the `in` operator in transition conditions.



In this chart, using the `in` operator to check state activity synchronizes substates in the parallel states `Place` and `Tracker`. For example, when the input position `u` becomes positive, the state transition from `Place.L` to `Place.R` occurs. This transition makes the condition `[in(Place.R)]` true, and the transition from `Tracker.Moved_Left` to `Tracker.Moved_Right` occurs.

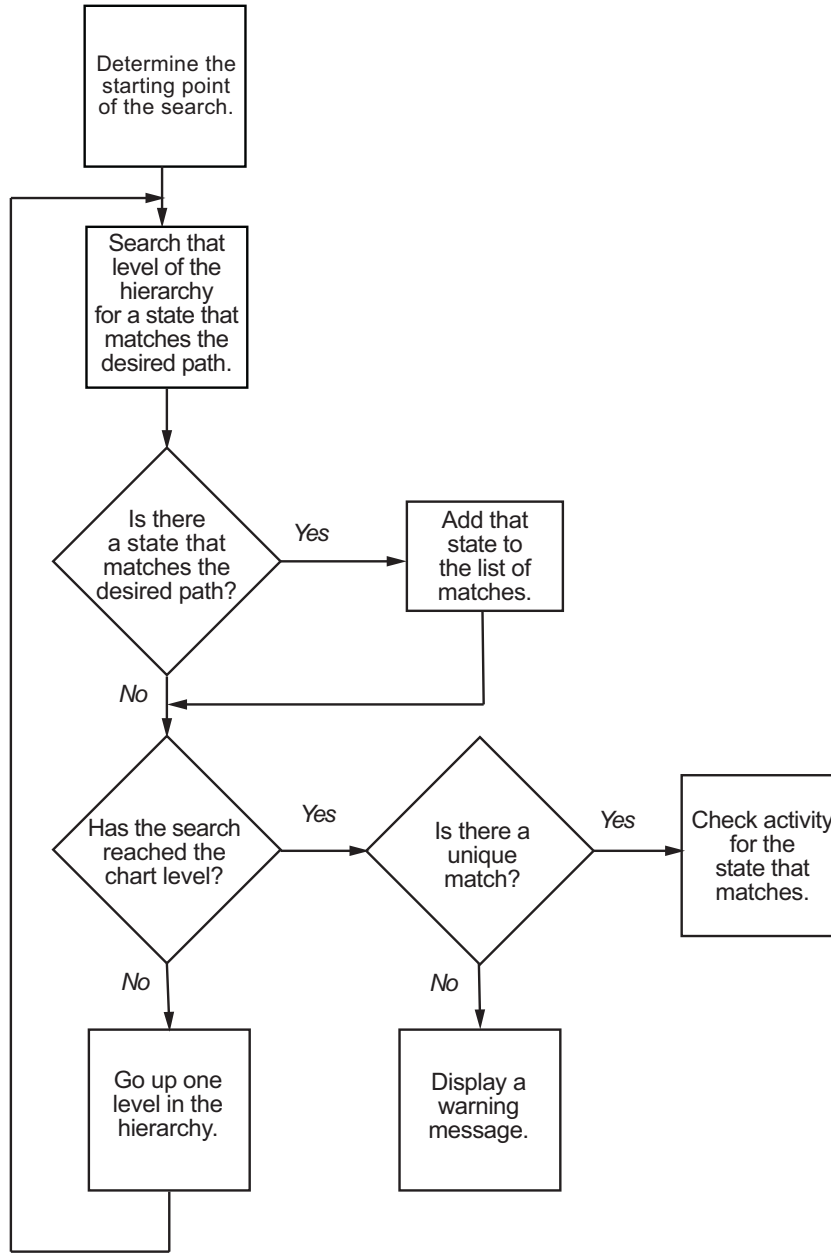
How Checking State Activity Works

Checking state activity is a two-stage process. First, the `in` operator must find the desired state. Then, the operator determines if the desired state is active.

- The `in` operator does not perform an exhaustive search for all states in a chart that can match the argument. It performs a localized search and stops.

- The `in` operator does not stop searching after it finds one match. It continues to search until it reaches the chart level.

This diagram shows the detailed process of checking state activity.


Determine the starting point of the search.
Search that level of the hierarchy for a state that matches the desired path.
Is there a state that matches the desired path?
Yes
Add that state to the list of matches.
No
Has the search reached the chart level?
Yes
Is there a unique match?
Yes
Check activity for the state that matches.
No
No
Go up one level in the hierarchy.
Display a warning message.

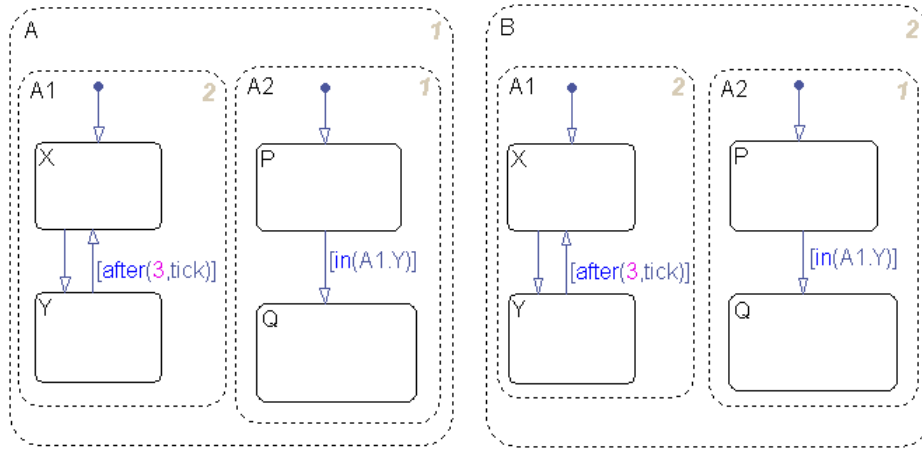
When you use the `in` operator to check state activity, these actions take place:

1. The search begins in the state where you use the `in` operator.
   - If you use the `in` operator in a state action, then that state is the starting point.
   - If you use the `in` operator in a transition label, then the parent of the source state is the starting point.
2. The `in` operator searches at that level of the hierarchy for a path to a state that matches the desired state. If the operator finds a match, it adds that state to the list of possible matches.
3. The operator moves up to the next highest level of the hierarchy. At that level, the operator searches for a path to a state that matches the desired state. If the operator finds a match, it adds that state to the list of possible matches.
4. The `in` operator repeats the previous step until it reaches the chart level.
5. At the chart level, the operator searches for a path to a state that matches the desired state. If the operator finds a match, it adds that state to the list of possible matches. Then, the search ends.
6. After the search ends, one of the following occurs:
   - If a unique search result is found, the `in` operator checks if that state is active and returns a value of 0 or 1.
   - If the operator finds no matches or multiple matches for the desired state, a warning message appears.

State Resolution for Identically Named Substates

For identically named substates in parallel superstates, the scope of the `in` operator remains local with respect to its chart-level superstate. When the `in` operator checks activity of a substate, it does not automatically detect an identically named substate that resides in a parallel superstate.

This example shows how the `in` operator works in a chart with identically named substates.



- Superstates `A` and `B` have identical substates `A1` and `A2`.
- The condition `in(A1.Y)` guards the transition from `P` to `Q` in the states `A.A2` and `B.A2`.
- For the state `A.A2`, the condition `in(A1.Y)` refers to the state `A.A1.Y`.
- For the state `B.A2`, the condition `in(A1.Y)` refers to the state `B.A1.Y`.

For the transition condition of `A.A2`, the `in` operator performs these search actions:

Step	Action of the `in` Operator	Finds a Match?
1	Picks `A.A2` as the starting point and searches for the state `A.A2.A1.Y`	No
2	Moves up to the next level of the hierarchy and searches for the state `A.A1.Y`	Yes
3	Moves up to the chart level and searches for the state `A1.Y`	No

The search ends, with the single state `A.A1.Y` found. The `in` operator checks if that state is active and returns a value of 0 or 1.

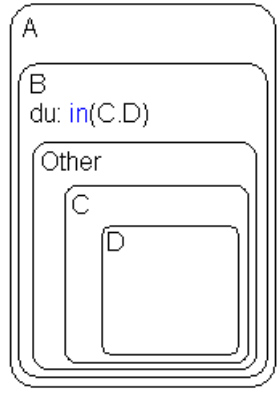
Localizing the scope of the `in` operator produces a unique search result. For example, the `in` operator of `A.A2` does not detect the state `B.A1.Y`, because the search algorithm localizes the scope of the operator. Similarly, the `in` operator of `B.A2` detects only the state `B.A1.Y` and does not detect the state `A.A1.Y`.

Best Practices for Checking State Activity

Use a Specific Search Path

Be specific when defining the path of the state whose activity you want to check. See the examples that follow for details.

Example of No States Matching the Argument of the in Operator.



In the state `A.B`, the `during` action invokes the `in` operator. Assume that you want to check the activity of the state `A.B.Other.C.D`. The `in` operator performs these search actions:

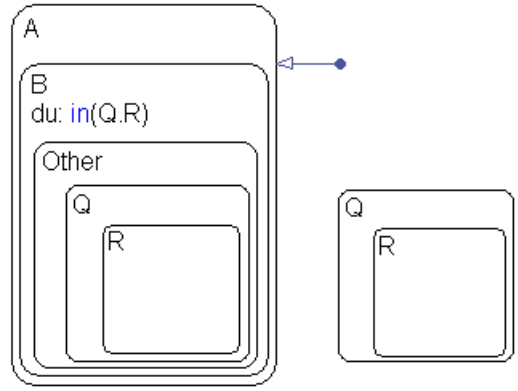
Step	Action of the `in` Operator	Finds a Match?
1	Picks `A.B` as the starting point and searches for the state `A.B.C.D`	No

Step	Action of the `in` Operator	Finds a Match?
2	Moves up to the next level of the hierarchy and searches for the state `A.C.D`	No
3	Moves up to the chart level and searches for the state `C.D`	No

The search ends, and a warning message appears because no match exists.

To eliminate the warning message, use a more specific path to check state activity: `in(Other.C.D)`.

Example of the Wrong State Matching the Argument of the in Operator.



In the state `A.B`, the `during` action invokes the `in` operator. Assume that you want to check the activity of the state `A.B.Other.Q.R`. The `in` operator performs these search actions:

Step	Action of the `in` Operator	Finds a Match?
1	Picks `A.B` as the starting point and searches for the state `A.B.Q.R`	No

Step	Action of the `in` Operator	Finds a Match?
2	Moves up to the next level of the hierarchy and searches for the state `A.Q.R`	No
3	Moves up to the chart level and searches for the state `Q.R`	Yes

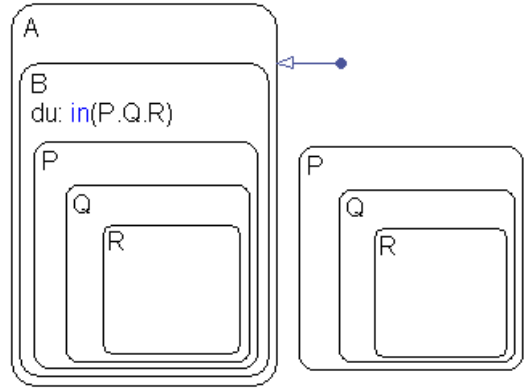
The search ends, with the single state `Q.R` found. The `in` operator checks if that state is active and returns a value of 0 or 1.

In this example, the `in` operator checks the status of the wrong state. To prevent this error, use a more specific path to check state activity: `in(Other.Q.R)`.

Use Unique State Names

Use unique names when you name the states in a chart.

Example of Multiple States Matching the Argument of the in Operator.



In the state `A.B`, the `during` action invokes the `in` operator. Assume that you want to check the activity of the state `A.B.P.Q.R`. The `in` operator performs these search actions:

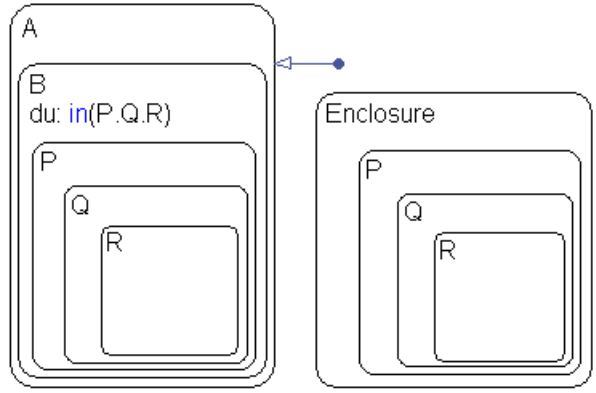
Step	Action of the `in` Operator	Finds a Match?
1	Picks `A.B` as the starting point and searches for the state `A.B.P.Q.R`	Yes
2	Moves up to the next level of the hierarchy and searches for the state `A.P.Q.R`	No
3	Moves up to the chart level and searches for the state `P.Q.R`	Yes

The search ends, and a warning message appears because multiple matches exist.

To eliminate the warning message, do one of these corrective actions:

- Rename one of the matching states.
- Use a more specific path to the desired state: `in(B.P.Q.R)`.
- Enclose the outer state `P.Q.R` in a box or another state, as shown below.

  Adding an enclosure prevents the `in` operator of state `A.B` from detecting that outer state.



Using Bind Actions to Control Function-Call Subsystems

In this section...

About Bind Actions

Bind actions in a state bind specified data and events to that state. 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 it triggers. The function-call subsystem enables when the state with the bound event is entered and disables when that state is exited. This means that the execution of the function-call subsystem is fully bound to the activity of the state that calls it.

Binding a Function-Call Subsystem to a State

By default, a function-call subsystem is controlled by the Stateflow 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 is deactivated. 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 10-5).

Bind actions 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.

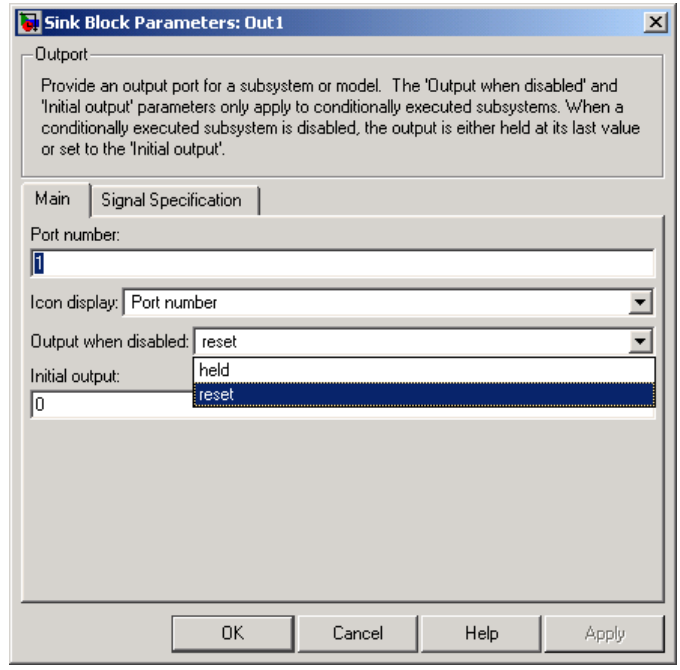
When you bind a function-call subsystem to a state, you can fine-tune the behavior of the subsystem when it is enabled and disabled, as described in the following sections:

- "Handling Outputs When the Subsystem is Disabled" on page 10-100
- "Controlling Behavior of States When the Subsystem is Enabled" on page 10-101

Handling Outputs When the Subsystem is Disabled

Although function-call subsystems do not execute while they are 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 its Block Parameters dialog box, as in this example:



2 Select an option for the field **Output when disabled**, as follows:

Select:	To:
held	Maintain most recent output value
reset	Reset output to its initial value

3 Click **OK** to record the settings.

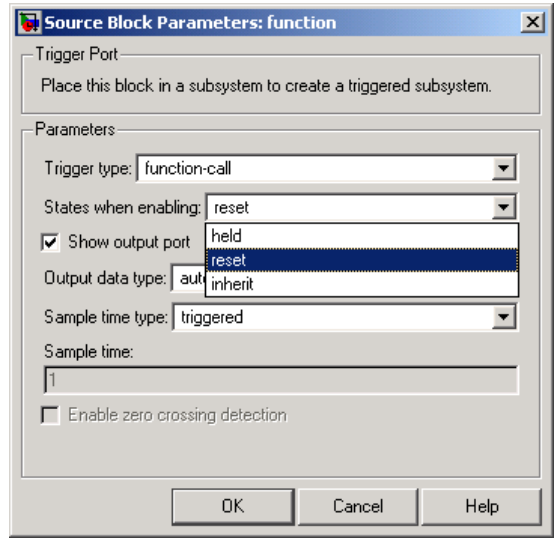
Note Setting **Output when disabled** is meaningful only when the function-call subsystem is bound to a state, as described in "Binding a Function-Call Subsystem to a State" on page 10-99.

Controlling 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 its Block Parameters dialog box, as in this example:



2 Select an option for the field **States when enabling**, as follows:

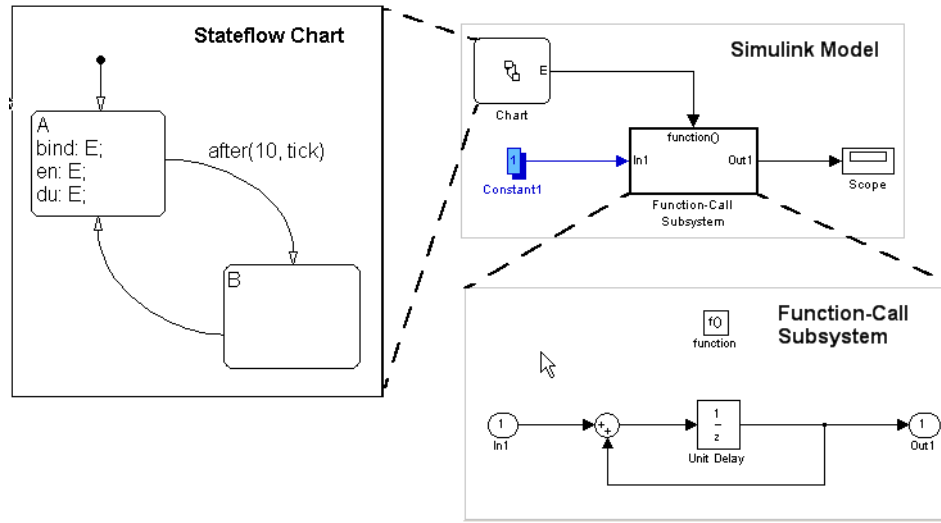
Select:	To:
held	Maintain most recent values of the states of the subsystem that contains the trigger port
reset	Revert to the initial conditions of the states of the subsystem that contains this trigger port
inherit	Inherit this setting from the function-call initiator's parent subsystem. If the parent 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 "Binding a Function-Call Subsystem to a State" on page 10-99.

Example of How to Bind a Function-Call Subsystem to a State

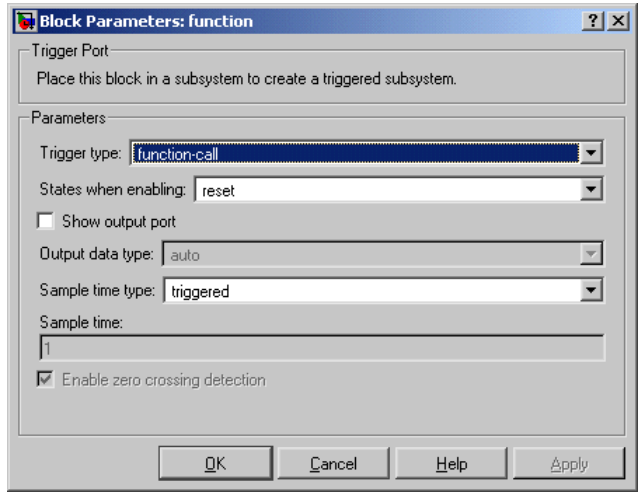
The control of a Stateflow state that binds a function-call subsystem trigger is best understood through the creation and execution of an example model. In the following example, a Simulink model triggers a function-call subsystem with a function-call trigger event `E` bound to state `A` of a Stateflow chart.



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 count by 1 each time, using a Unit Delay block to store the count.

The Stateflow chart contains two states, `A` and `B`, and connecting transitions, along with some actions. Notice that event `E` is bound to state A with the binding action `bind:E`. Event `E` is defined for the Stateflow chart in the example with a scope of **Output to Simulink** and a trigger type of **function-call**.

The Block Parameters dialog box for the trigger port appears.



Notice that the **States when enabling** field is set to the default value **reset**. This resets the state values for the function-call subsystem to zero when it is enabled.

Notice also that the **Sample time type** field is set to the default value **triggered**. This value sets the function-call subsystem to execute only when it is triggered by a calling event while it is enabled.

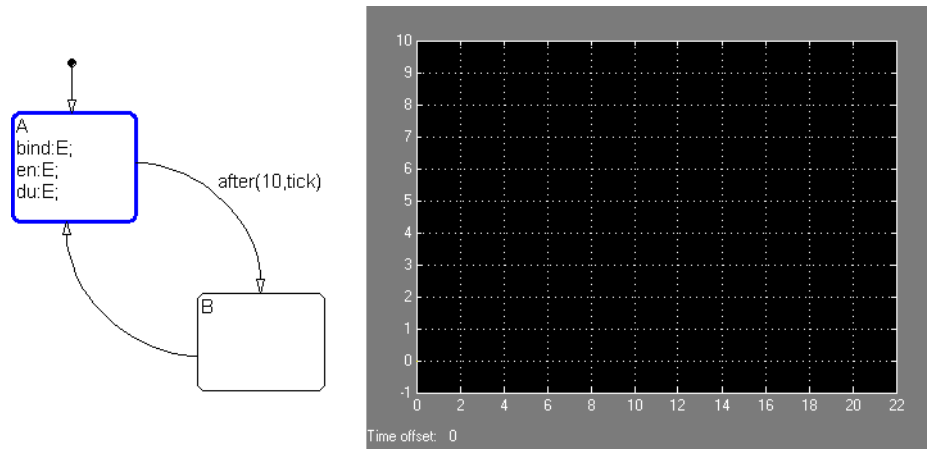
Setting **Sample time type** to **periodic** enables the **Sample time** field below it, which defaults to 1. These settings force the function-call subsystem to execute for each time step specified in the **Sample time** field while it is enabled. To accomplish this, the state that binds the calling 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. Therefore, for fixed-step sampling, the sample time you enter in the **Sample time** field must be an integer multiple of the fixed-step size. For variable-step sampling, there are no limits on what you can enter in the **Sample time** field.

Simulating a Bound Function-Call Subsystem

To see the control that a state can have over the function-call subsystem whose trigger event it binds, begin simulating the example model in "Example of How to Bind a Function-Call Subsystem to a State" on page 10-103. For the purposes of display, the simulation parameters for this model specify a fixed-step solver with a fixed-step size of 1. Take note of model behavior in the following steps, which record the simulating Stateflow chart and the output of the subsystem.

1 The default transition to state `A` is taken.

2 State `A` becomes active as shown.

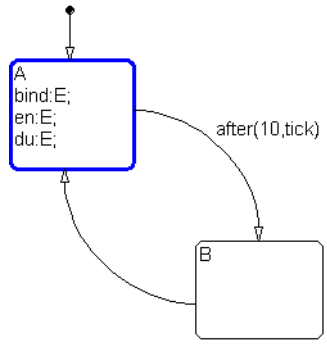


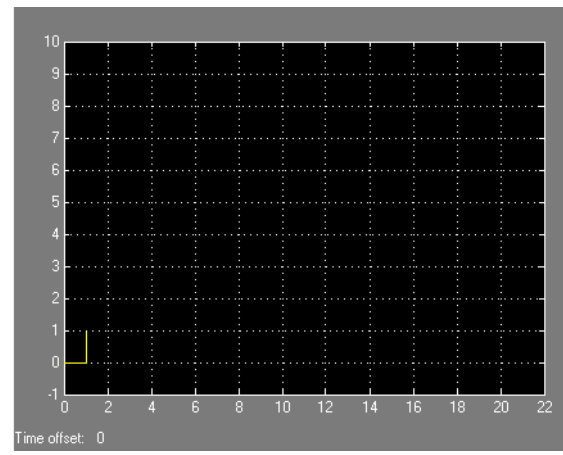
When state `A` becomes active, it executes its bind and entry actions. The binding action, `bind:E`, binds event `E` to state `A`. This enables the function-call subsystem and resets its state variables to 0.

State `A` also executes its entry action, `en:E`, which sends an event `E` to trigger the function-call subsystem and execute its block diagram. The block diagram increments a count by 1 each time using a Unit Delay block. Since the previous content of the Unit Delay block is 0 after the reset, the starting output point is 0 and the current value of 1 is held for the next call to the subsystem.

3 The next update event from the Simulink model tests state `A` for an outgoing transition.

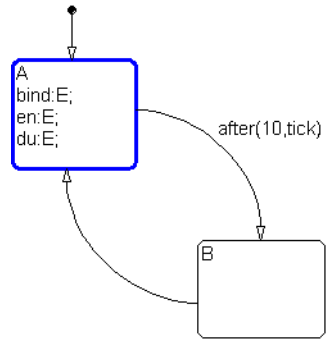
The temporal operation on the transition to state `B`, `after(10, tick)`, allows the transition to be taken only after ten update events are received. This means that for the second update, the during action of state `A`, `du:E`, is executed, which sends an event to trigger the function-call subsystem. The held content of the Unit Delay block, 1, is output to the scope as shown.

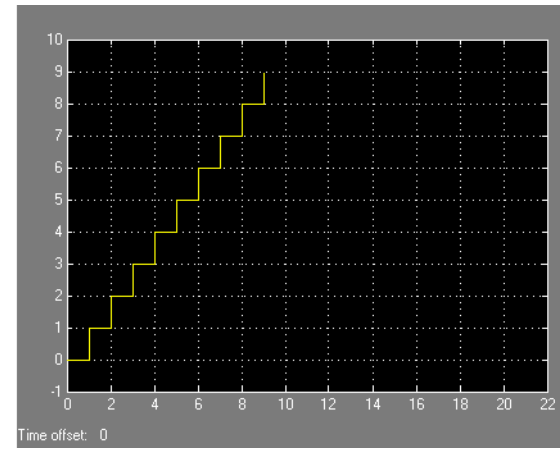




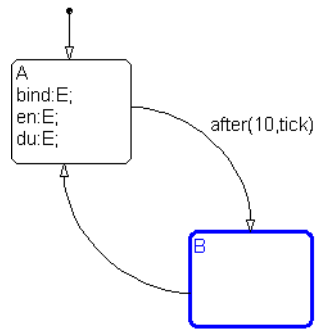
The subsystem also adds 1 to the held value to produce the value 2, which is held by the Unit Delay block for the next triggered execution.

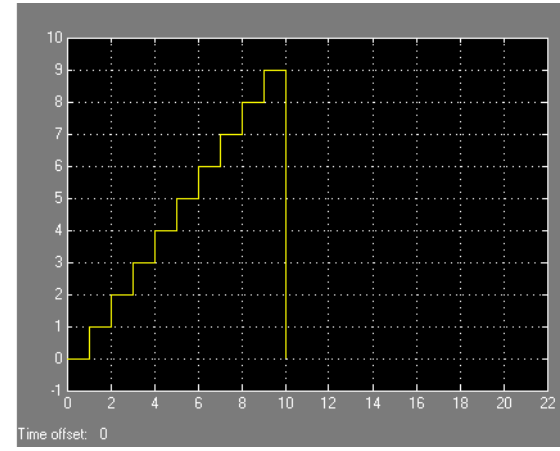
4 The next eight update events repeat step 2, which increment the subsystem output by 1 each time as shown.





5 On the 11th update event, the transition to state `B` is taken as shown.

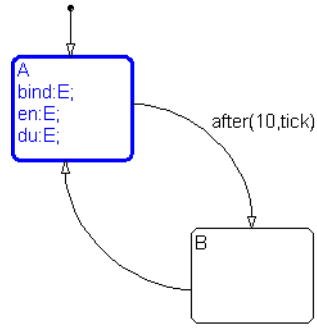


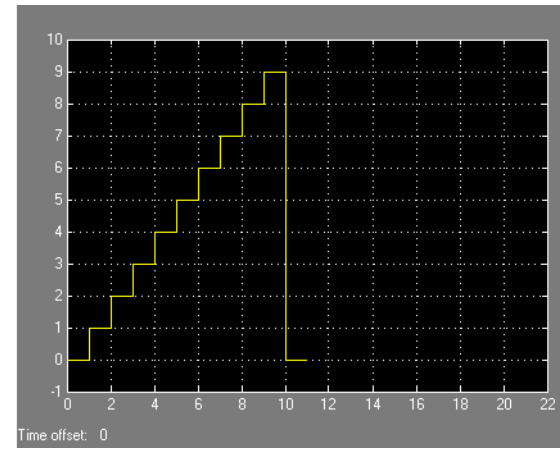


This makes state `B` active. Since the binding state `A` is no longer active, the function-call subsystem is disabled, and its output drops to 0.

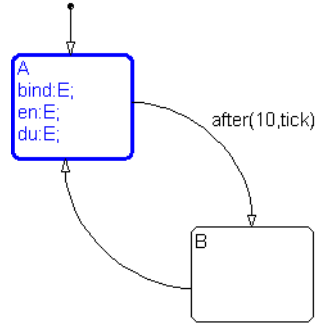
6 When the next sampling event occurs, the transition from state `B` to state `A` is taken.

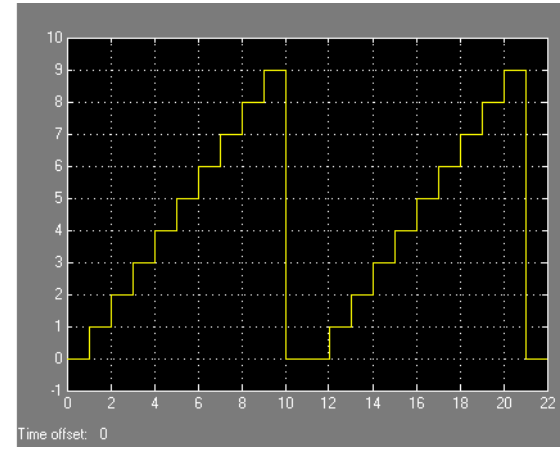
Once again, the binding action, `bind: E`, enables the function-call subsystem and resets its output to 0 as shown.





7 With the next 10 update events, steps 2 through 5 repeat, producing the following output:

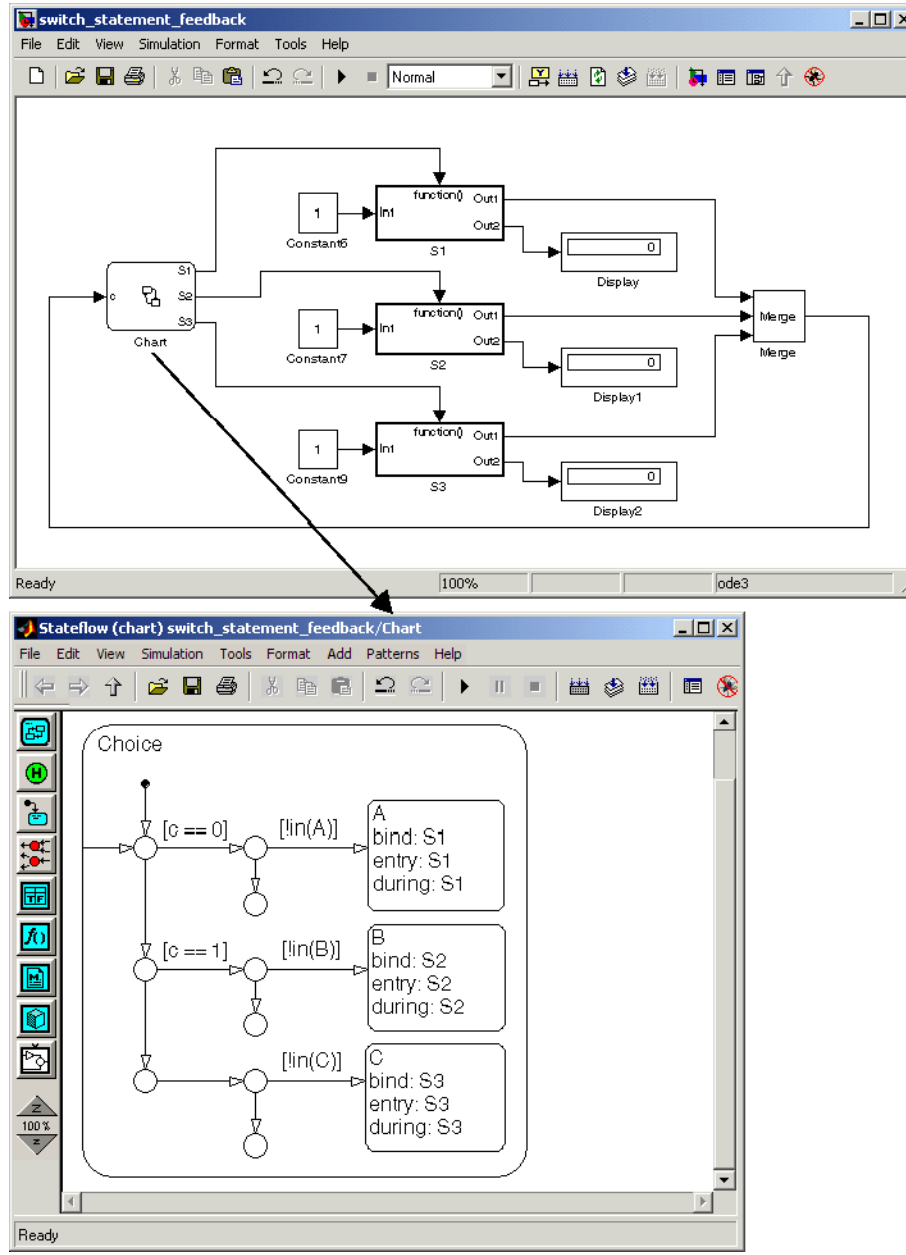




Using Stateflow Logic with Binding

You can use Stateflow logic to control function-call subsystems that model C-like `switch`, `if-else`, `for`, and `while` statements in Simulink models. Although you can model switch behavior in a Stateflow chart, the generated code approximates the switch logic by using `if-else` statements.

For example, the following model demonstrates a Simulink `switch` statement with subsystems controlled by bind actions:


switch_statement_feedback
File Edit View Simulation Format Tools Help
Normal
function()
In1
Out1
Out2
Constant6
S1
Display
Chart
S2
S3
Constant7
Display1
Merge
Constant9
Display2
Ready
100%
ode3
Stateflow (chart) switch_statement_feedback/Chart
File Edit View Simulation Tools Format Add Patterns Help
Choice
[c == 0]
[!in(A)]
A
bind: S1
entry: S1
during: S1
[c == 1]
[!in(B)]
B
bind: S2
entry: S2
during: S2
[!in(C)]
C
bind: S3
entry: S3
during: S3
100 %
Ready

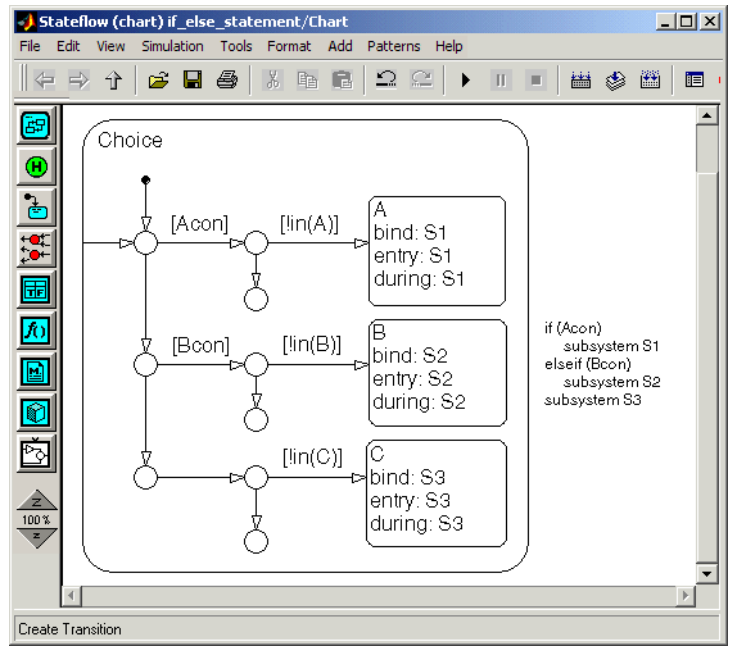
In this model, the Stateflow chart controls three subsystems, `S1`, `S2`, and `S3`, through the bind actions for three states, `A`, `B`, and `C`, respectively. In this example, the value of the case argument `c` determines the subsystem to execute. State `A` becomes active and stays active when `c` is 0. State `B` becomes active and stays active when `c` is 1. State `C` becomes active and stays active when `c` has any other value.

When state `A` is active, the event `S1` is bound to state `A`, which enables subsystem `S1`. The entry and during actions for `A` broadcast the event `S1` whenever the model is updated for sampling. This means that while `A` is active, the subsystem `S1` is executed for each sample time. The same applies to subsystem `S2` for state `B`, and to subsystem `S3` for state `C`.

The generated code for this model does not contain `switch` statements. Instead, it uses `if-else` logic, as represented by the following pseudocode:

```
if (c==0)
   if (!in(A))
      subsystem S1
else if (c==1)
   if(!in(B))
      subsystem S2
else
   if (!in(C))
       subsystem S3
```

You can modify the previous Stateflow chart to control a Simulink model with an `if-else` statement, as shown.



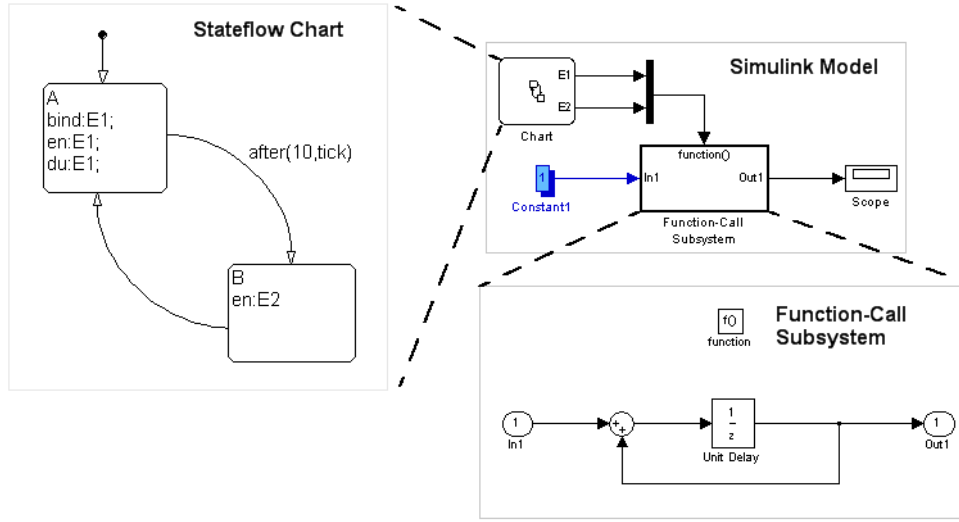
In this example, State `A` becomes active and stays active when the condition `Acon` is true. State `B` becomes active and stays active when the condition `Bcon` is true and the condition `Acon` is false. State `C` becomes active and stays active when both conditions `Acon` and `Bcon` are false. This creates the following `if-else` statement in the Simulink model:

```
if (Acon)
     subsystem S1
elseif (Bcon)
     subsystem S2
subsystem S3
```

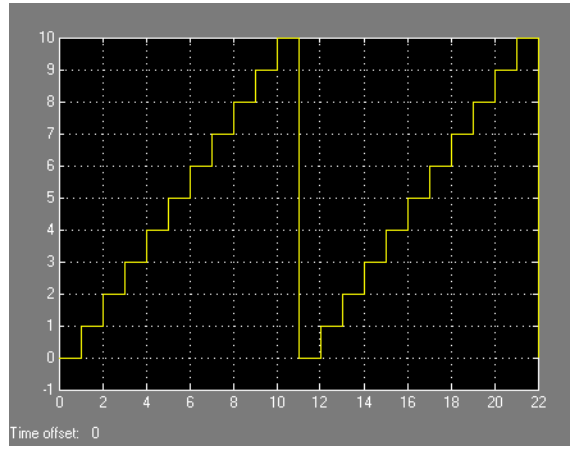
Avoiding Muxed Trigger Events with Binding

The simulated example in “Simulating a Bound Function-Call Subsystem” on page 10-105 shows how binding events gives control of a function-call subsystem to a single state in a Stateflow chart. This control can be undermined if you allow other events to trigger the function-call subsystem

through a mux. For example, the following Simulink model defines two function-call events to trigger a function-call subsystem through a mux:



In the Stateflow chart, `E1` is bound to state `A`, but `E2` is not. This means that state `B` is free to send the triggering event `E2` in its entry action. When you simulate this model, you receive the following output:



Notice that broadcasting `E2` in state `B` changes the output, which now rises to a height of 10 before the binding action in state `A` resets the data.

Note Binding is not recommended when users provide multiple trigger events to a function-call subsystem through a mux. Muxed trigger events can interfere with event binding and cause undefined behavior.

11

Using Vectors and Matrices in Stateflow Charts

How Vectors and Matrices Work in Stateflow Charts

In this section...

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 more information, see "Examples of Vectors and Matrices in Stateflow Charts" on page 11-15.

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
- Embedded MATLAB functions (see Chapter 20, “Using Embedded MATLAB Functions in Stateflow Charts”)
- Truth table functions (see Chapter 19, “Truth Table Functions”)
- Graphical functions (see “Using Graphical Functions to Extend Actions” on page 7-27)
- Simulink functions (see Chapter 21, “Using Simulink Functions in Stateflow Charts”)
- Change detection operators

For more information, see “Operations That Work with Vectors and Matrices in Stateflow Action Language” on page 11-9 and “Rules for Using Vectors and Matrices in Stateflow Charts” on page 11-11.

How to Define Vectors and Matrices

In this section...

Defining a Vector

Define a vector in a Stateflow chart as follows:

1. In the Stateflow Editor, select **Add > Data** and choose a scope.

   The Data properties dialog box appears.

2. In the **General** pane, 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 object.

   Note Vectors cannot have the base type `ml`. See "Rules for Using Vectors and Matrices in Stateflow Charts" on page 11-11.

4. Set initial values for the vector in the **Value Attributes** pane.

   - 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**.

Defining a Matrix

Define a matrix in a Stateflow chart as follows:

1 In the Stateflow Editor, select **Add > Data** and choose a scope.

The Data properties dialog box appears.

2 In the **General** pane, 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 object.

Note Matrices cannot have the base type `ml`. See "Rules for Using Vectors and Matrices in Stateflow Charts" on page 11-11.

---

4 Set initial values for the matrix in the **Value Attributes** pane.

- 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**.

How to Assign and Access Values of Vectors and Matrices

In this section...

Notation for Vectors and Matrices

Index notation for vectors and matrices in a Stateflow chart differs from the notation you use in a MATLAB script. You use zero-based indexing for each dimension of a vector or matrix in Stateflow action language. However, you use one-based indexing in a MATLAB script.

To refer to...	In Stateflow action language, use...	In a MATLAB script, use...
The first element of a vector `test`	`test[0]`	`test(1)`
The i^{th} element of a vector `test`	`test[i-1]`	`test(i)`
The element in row `4` and column `5` of a matrix `test`	`test[3][4]`	`test(4,5)`
The element in row `i` and column `j` of a matrix `test`	`test[i-1][j-1]`	`test(i,j)`

Assigning and Accessing Values of Vectors

The following examples show how to assign the value of an element in a vector in Stateflow action language.

If you enter...	You assign the value...	To...
`test[0] = 10;`	10	The first element
`test[i] = 77;`	77	The `(i+1)`th element

The following examples show how to access the value of an element in a vector in Stateflow action language.

If you enter...	You access the value of...
`old = test[1];`	The second element of a vector `test`
`new = test[i+5];`	The `(i+6)`th element of a vector `test`

Assigning and Accessing Values of Matrices

The following examples show how to assign the value of an element in a matrix in Stateflow action language.

If you enter...	You assign the value...	To the element in...
`test[0][8] = 10;`	10	Row 1, column 9
`test[i][j] = 77;`	77	Row `i+1`, column `j+1`

The following examples show how to access the value of an element in a matrix in Stateflow action language.

If you enter...	You access the value of...
`old = test[1][8];`	The matrix `test` in row 2, column 9
`new = test[i][j];`	The matrix `test` in row `i+1`, column `j+1`

Using Scalar Expansion to Assign Values of a Vector or Matrix

You can use scalar expansion in Stateflow action language to set all elements of a vector or matrix to the same value. This method works for a vector or matrix of any size.

This action sets all elements of a vector to 10.

```
test_vector = 10;
```

This action sets all elements of a matrix to 20.

```
test_matrix = 20;
```

Note You cannot use scalar expansion on a vector or matrix in the MATLAB base workspace. If you try to use scalar expansion, the vector or matrix in the base workspace converts to a scalar.

Operations That Work with Vectors and Matrices in Stateflow Action Language

In this section...

Binary Operations

You can perform element-wise binary operations on vector and matrix operands of equal dimensions in the following order of precedence (1 = highest, 3 = lowest). For operations with equal precedence, they evaluate in order from left to right.

Example	Precedence	Description
`a * b`	1	Multiplication
`a / b`	1	Division
`a + b`	2	Addition
`a - b`	2	Subtraction
`a == b`	3	Comparison, equality
`a != b`	3	Comparison, inequality

The multiplication and division operators in Stateflow action language perform element-wise operations, not standard matrix multiplication and division. For more information, see "Using Embedded MATLAB Functions to Perform Matrix Multiplication and Division" on page 11-12.

Unary Operations and Actions

You can perform element-wise unary operations and actions on vector and matrix operands.

Example	Description
`~a`	Unary minus
`!a`	Logical NOT
`a++`	Increments all elements of the vector or matrix by 1
`a--`	Decrements all elements of the vector or matrix by 1

Assignment Operations

You can perform element-wise assignment operations on vector and matrix operands.

Example	Description
`a = expression`	Simple assignment
`a += expression`	Equivalent to `a = a + expression`
`a -= expression`	Equivalent to `a = a - expression`
`a *= expression`	Equivalent to `a = a * expression`
`a /= expression`	Equivalent to `a = a / expression`

Rules for Using 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 “Operations That Work with Vectors and Matrices in Stateflow Action Language” on page 11-9.

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 10-38.

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

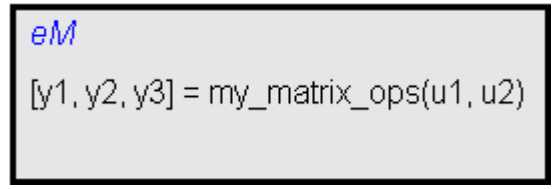
In this section...

Using Embedded MATLAB Functions to Perform Matrix Multiplication and Division

In Stateflow action language, the multiplication and division operators perform element-wise multiplication and division. Use an Embedded 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. Add this Embedded MATLAB function to your chart.



   The Embedded MATLAB action language supports a subset of MATLAB functions and operations. Therefore, you can perform standard multiplication and division on matrix operands.

2. Double-click the function box to open the Embedded MATLAB Editor.

3. In the Embedded MATLAB Editor, enter the code below.

```
1     function [y1, y2, y3] = my_matrix_ops(u1, u2)
2
3 -   y1 = u1 * u2;    % matrix multiplication
4 -   y2 = u1 \ u2;    % matrix division from the right
5 -   y3 = u1 / u2;    % matrix division from the left
```

- `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 In the Stateflow Editor, select **View > Model Explorer**.

The Model Explorer appears.

b In the **Model Hierarchy** pane, navigate to the level of the Embedded MATLAB function.

c In the **Contents** pane, set properties for each data object.

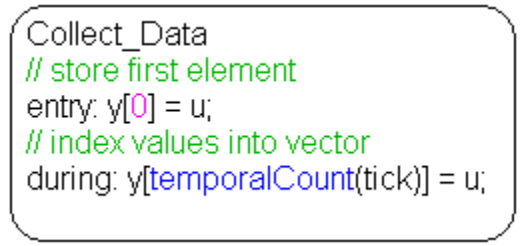
Note To initialize a matrix, see "Defining a Matrix" on page 11-5.

Using the temporalCount Operator to Index a Vector

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 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 **Add > Data** and the scope **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 **Add > Data** and the scope **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.

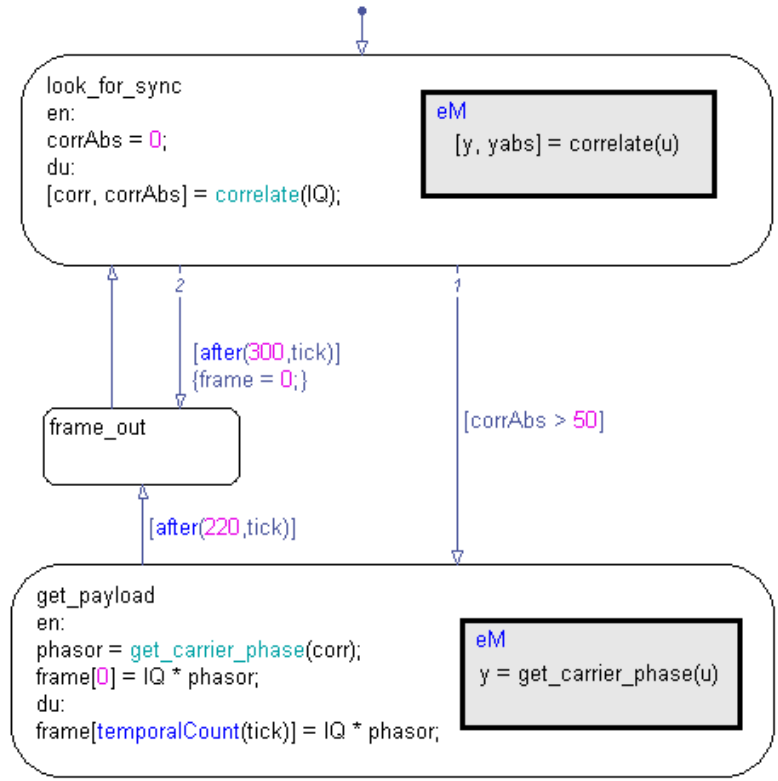
For information about the `temporalCount` operator, see "Using Temporal Logic in State Actions and Transitions" on page 10-56.

Examples of Vectors and Matrices in Stateflow Charts

In this section...

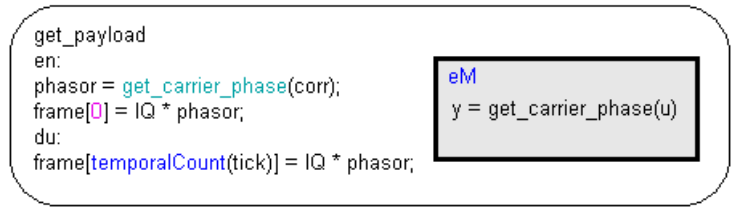
Communications Example

The demo 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.



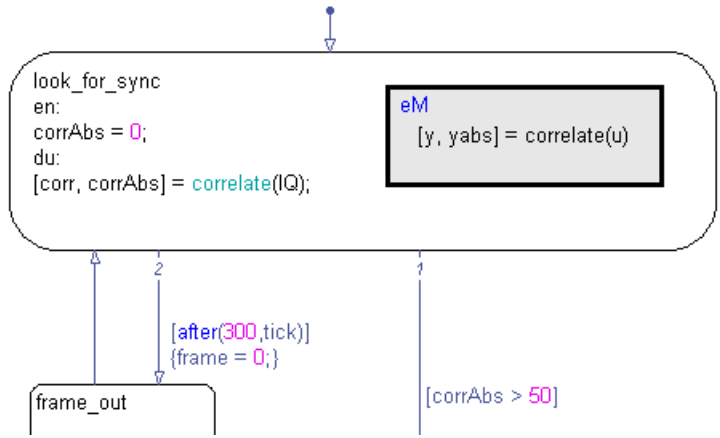
For details of how the chart works, see "Implementing a Frame Synchronization Controller Using a Stateflow Chart" on page 15-17.

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 "Using the temporalCount Operator to Index a Vector" on page 11-13.)

Scalar Expansion of a Vector

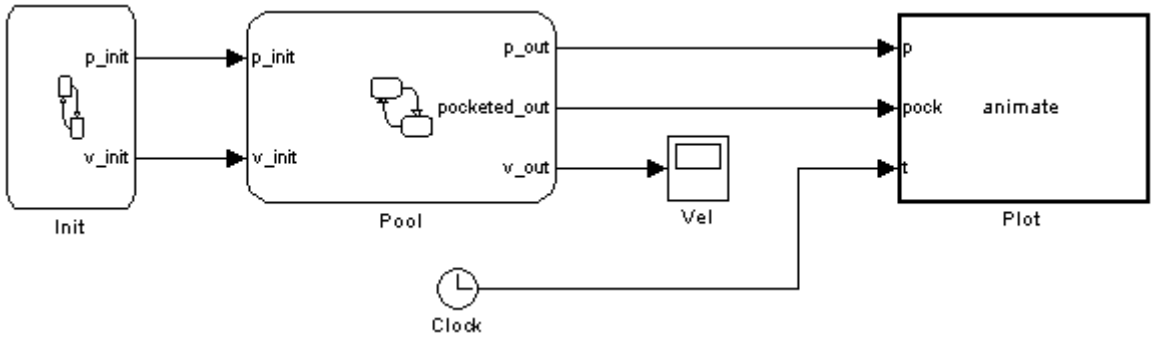


In the second outgoing transition of the state `look_for_sync`, the transition action `frame = 0` resets all elements of the vector `frame` to 0 via scalar

expansion. (For more information, see "Using Scalar Expansion to Assign Values of a Vector or Matrix" on page 11-8.)

Physics Example

The demo 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.

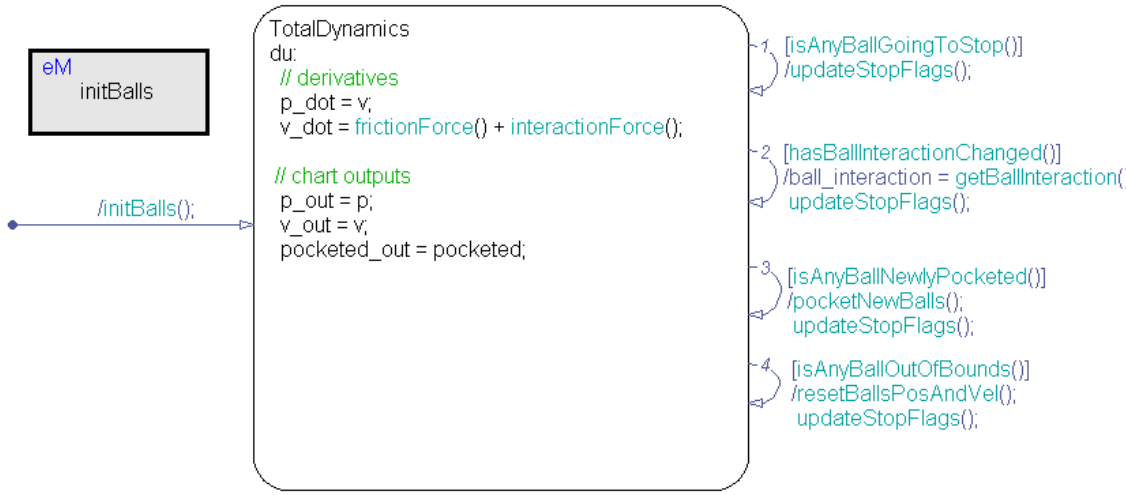
Storage of Two-Dimensional Data in Matrices

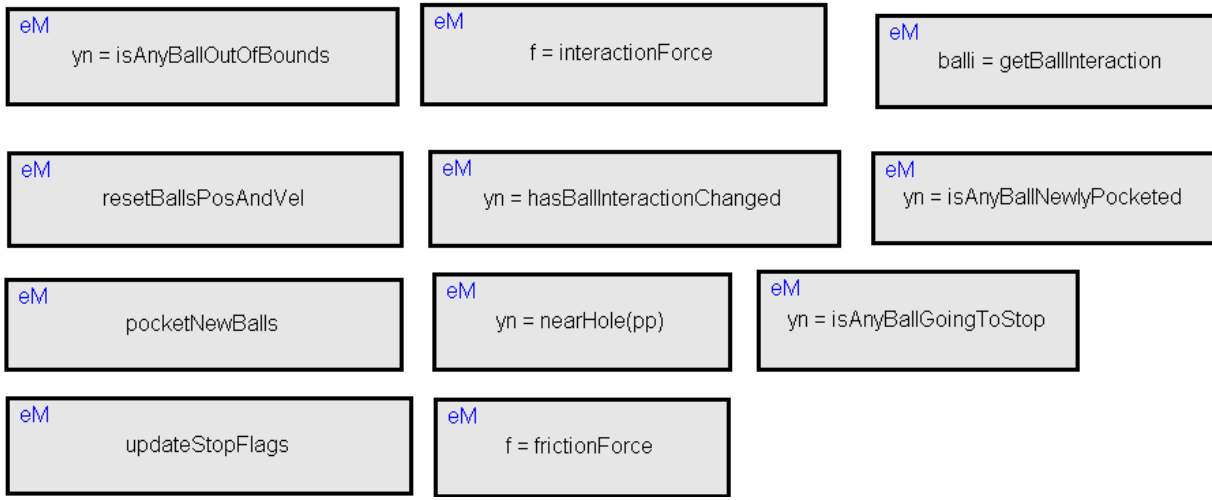
To simulate the opening shot, the Pool chart stores two-dimensional data in matrices.

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 Embedded MATLAB functions that perform matrix calculations.





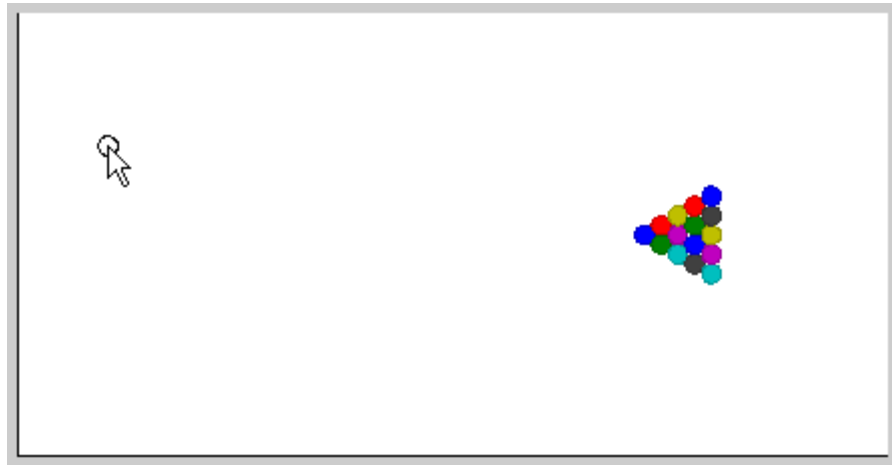
Embedded MATLAB Function	Description
`frictionForce`	Calculates the friction force acting on each ball.
`getBallInteraction`	Returns a matrix of Boolean data on whether any two balls are in contact.
`hasBallInteractionChanged`	Returns 1 if ball interactions have changed and 0 otherwise.
`initBalls`	Initializes 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.

Running the Demo Model

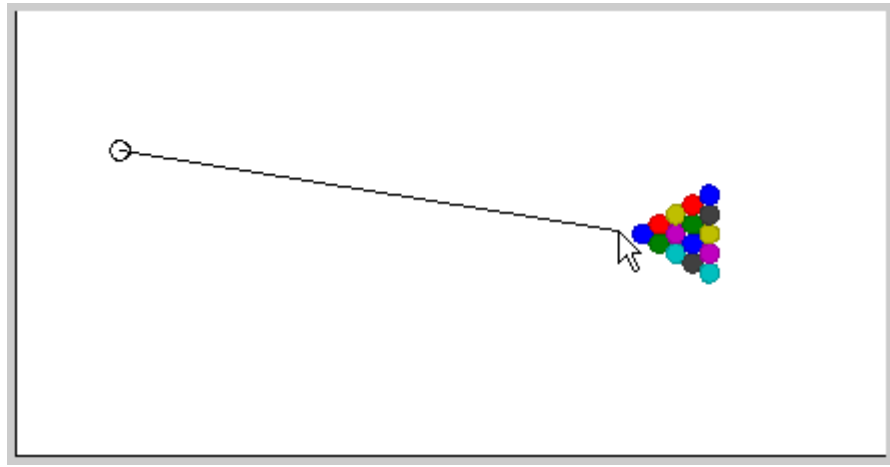
To run the demo model, follow these steps:

1. Type `sf_pool` at the MATLAB command prompt.
2. Select **Simulation > Start** in the model window.

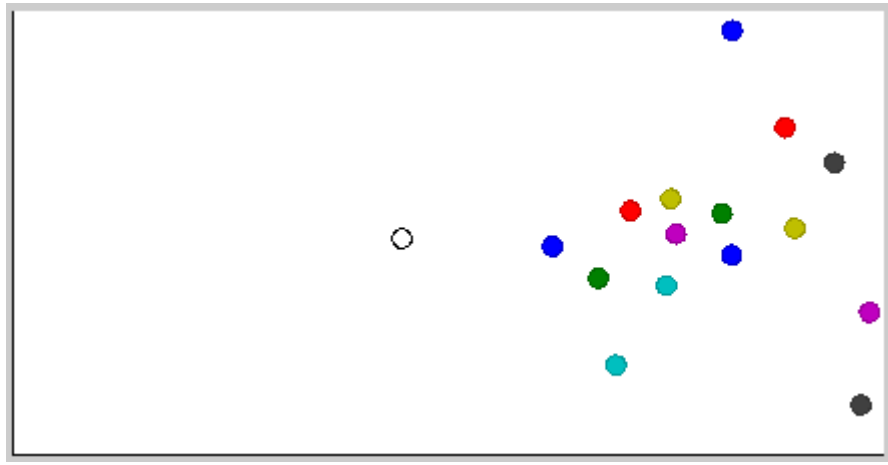
3 Click anywhere in the animated pool table to specify the initial position of the cue ball.



4 Click a different spot to specify the initial velocity of the cue ball.



5 Watch the balls move across the pool table.



12

Using Enumerated Data in Stateflow Charts

What Is Enumerated Data?

Enumerated data has a finite set of values. An enumerated data type consists of values that you allow for that type, or *enumerated values*. For integer-based enumerated types, each enumerated value consists of a name and an underlying numeric value.

For example, the following MATLAB script restricts an integer-based enumerated data type named `BasicColors` to three enumerated values.

```
classdef(Enumeration) BasicColors < Simulink.IntEnumType
  enumeration
    Red(0)
    Yellow(1)
    Green(2)
  end
end
```

Enumerated Value	Enumerated Name	Underlying Numeric Value
`Red(0)`	`Red`	`0`
`Yellow(1)`	`Yellow`	`1`
`Green(2)`	`Green`	`2`

For information on defining an enumerated data type, see "How to Define Enumerated Data in a Stateflow Chart" on page 12-8.

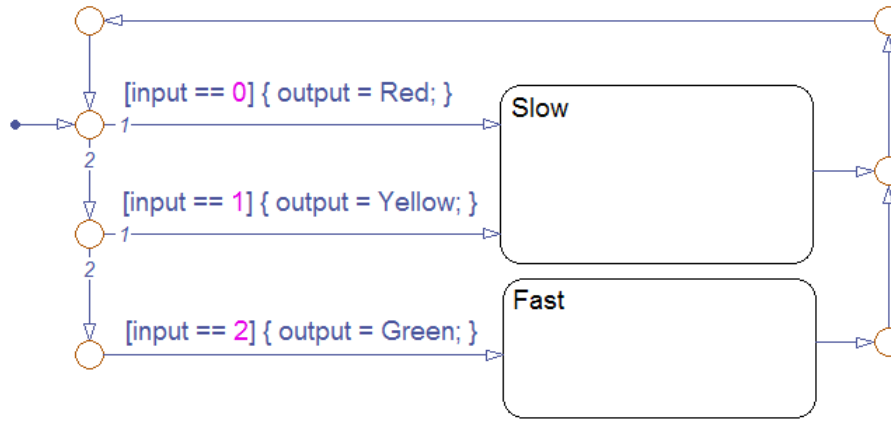
For information on using enumerated data in other blocks of a Simulink model, see "Using Enumerated Data" in the *Simulink User's Guide*.

Benefits of Using Enumerated Data in a Chart

Use enumerated data in a Stateflow chart to:

- Model a physical system with a finite number of states
- Restrict data to a finite set of values
- Refer to these values by name

For example, this chart uses enumerated data to refer to a set of colors.



- The chart models a system with two discrete states: `Slow` and `Fast`.
- The enumerated data `output` is restricted to a finite set of values: 0, 1, and 2.
- You can refer to these values by their enumerated names: `Red`, `Yellow`, and `Green`.

In large-scale models, use enumerated data for these benefits:

- Enhance readability of data in a chart.
- Avoid defining a long list of constants.

  For example, you can group related values into separate data types.

Where to Use Enumerated Data in a Chart

You can use enumerated data at these levels of the Stateflow hierarchy:

- Chart
- Subchart
- State

You can use enumerated data as arguments for:

- State actions
- Condition and transition actions
- Vector and matrix indexing
- Simulink functions (see Chapter 21, "Using Simulink Functions in Stateflow Charts")
- Graphical functions (see "Using Graphical Functions to Extend Actions" on page 7-27)
- Truth table functions that use Stateflow action language (see Chapter 19, "Truth Table Functions")

Note Embedded MATLAB functions and truth table functions that use Embedded MATLAB action language do not support enumerated data as arguments.

You can use enumerated data for simulation and Real-Time Workshop code generation. However, custom targets do not support enumerated data. For more information, see "Rules for Using Enumerated Data in a Stateflow Chart" on page 12-18.

Elements of an Enumerated Data Type Definition

This M-file shows the elements of an enumerated data type definition.

```
classdef(Enumeration) BasicColors < Simulink.IntEnumType
  enumeration
    Red(0)
    Yellow(1)
    Green(2)
  end

  methods (Static = true)
    function retVal = getDefaultValue()
      % GETDEFAULTVALUE  Returns the default enumerated value.
      %   This value must be an instance of the enumerated type.
      %   Used by Simulink when an instance of this class is
      %   needed but the value is not known (e.g., initializing
      %   ground values or casting an invalid numeric value to
      %   an enumerated type). If this method is not defined,
      %   the first value is used.
      retVal = BasicColors.Green;
    end

    function retVal = getDescription()
      % GETDESCRIPTION  Optional string to describe data type.
      retVal = 'This defines an enumerated type for colors';
    end

    function retVal = getHeaderFile()
      % GETHEADERFILE  File where type is defined for generated
      %   code. If specified, this file is #included as needed
      %   in the code. Otherwise, the type is written out in
      %   the generated code.
      retVal = 'imported_enum_type.h';
    end

    function retVal = addClassNameToEnumNames()
      % ADDCLASSNAMETOENUMNAMES  Specify if class name is added
      %   as a prefix to enumerated names in the generated code.
      %   By default we do not add the prefix.
```

```
        retVal = true;
      end
    end
end
```

The data type definition consists of three sections of code.

Section of Code	Required?	Purpose	Reference
`classdef`	Yes	Gives the name of the enumerated data type	"Defining an Enumerated Data Type in an M-file" on page 12-8
`enumeration`	Yes	Lists the enumerated values that the data type allows	"Defining an Enumerated Data Type in an M-file" on page 12-8
`methods`	No	Provides methods that customize the data type: • `getDefaultValue` Specifies a default enumerated value other than the first one in the list of allowed values • `getDescription` Gives a description of the data type for Real-Time Workshop generated code • `getHeaderFile` Enables importing of custom header files that contain enumerated type definitions for Real-Time Workshop generated code • `addClassNameToEnumNames` Prevents name conflicts with identifiers in Real-Time Workshop generated code and enhances readability	"Using Enumerated Data" in the Simulink User's Guide "Enumerated Data Types in Generated Code" in the Real-Time Workshop User's Guide

In the example, the `methods` section of code customizes the data type as follows:

- 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 Real-Time Workshop generated code
- Uses a custom header file to prevent the data type from being written out in Real-Time Workshop generated code
- Adds the name of the data type as a prefix to each enumerated name in Real-Time Workshop generated code

How to Define Enumerated Data in a Stateflow Chart

In this section...

Tasks for Defining Enumerated Data in a Chart

To use enumerated data in a chart, you must follow these steps:

1 Define an enumerated data type in an M-file on the MATLAB path.

This data type is a MATLAB class definition, which belongs in an M-file. For details, see *Object-Oriented Programming* in the MATLAB documentation.

Note For each enumerated type, you must create a new M-file.

2 Add data of the enumerated type to a chart.

Defining an Enumerated Data Type in an M-file

To define an enumerated data type in an M-file, follow these steps:

1 Create a class definition file.

In the MATLAB Command Window, select **File > New > Class M-File**.

2 Define enumerated values in an `enumeration` section.

```
classdef(Enumeration) EnumTypeName < Simulink.IntEnumType
  enumeration
    EnumName(N)
    ...
  end
end
```

EnumTypeName is a case-sensitive string that must be unique among data type names and workspace variable names. An enumerated type can define any number of values. Each enumerated value consists of a string *EnumName* and an integer *N*. Each *EnumName* must be unique within its type, but can also appear in other enumerated types.

For example, you can enter the following lines in the MATLAB Editor:

```
classdef(Enumeration) BasicColors < Simulink.IntEnumType
  enumeration
    Red(0)
    Yellow(1)
    Green(2)
  end
end
```

The `classdef` section defines an integer-based enumerated data type with the name `BasicColors` and derives it from the built-in type `Simulink.IntEnumType`. The `enumeration` section is the set of values that this data type allows. The default value is the first one in the list, unless you specify otherwise in the next step.

3 (Optional) Customize the data type using a `methods` section.

For details, see "Elements of an Enumerated Data Type Definition" on page 12-5 or "Overriding Default Methods (Optional)" in the *Simulink User's Guide*.

4 Save the M-file on the MATLAB path.

The name of your M-file must match exactly the name of your data type. For example, the data type `BasicColors` must reside in an M-file named `BasicColors.m`.

Note To add a directory to the MATLAB search path, type `addpath` *`pathname`* at the command prompt.

Adding Enumerated Data to a Chart

To add enumerated data to a chart, follow these steps:

1 In the Stateflow Editor, select **Add > Data** and choose a scope other than `Constant`.

   The Data properties dialog box appears.

2 In the **General** pane, enter a name and data type for the enumerated data.

   a In the **Name** field, enter a name.

   b In the **Type** field, select `Enum: <class name>`.

   Note The **Complexity** field disappears when you select `Enum: <class name>` because enumerated data does not support complex values.

   c Replace `<class name>` with the name of the data type that you defined in your M-file.

   For example, you can enter `Enum: BasicColors` in the **Type** field. (See "Defining an Enumerated Data Type in an M-file" on page 12-8.)

   d Click **Apply**.

3 (Optional) In the **Value Attributes** pane, enter an initial value for the enumerated data.

   a In the **Initial value** field, enter a prefixed identifier that refers to an enumerated value for this data type. (For details, see "Rules for Using Enumerated Data in a Stateflow Chart" on page 12-18.)

   For example, `BasicColors.Red` is an identifier that uses prefixed notation. (See "Prefixed Notation for Enumerated Values" on page 12-14.)

   Note If you leave this field empty, the default enumerated value applies—that is, the first value in the data type definition. To specify the default value explicitly, see "Elements of an Enumerated Data Type Definition" on page 12-5 or "Overriding Default Methods (Optional)" in the *Simulink User's Guide*.

b Click **OK**.

Ensuring That Changes in Data Type Definition Take Effect

When you update an enumerated data type definition for an open model, the changes do not take effect right away. To see the effects of updating a data type definition, follow these steps:

1. Save the model.
2. Close the model.
3. Delete instances of the data type from the MATLAB base workspace.

   Tip To find these instances, type `whos` at the command prompt.

4. Open the model.
5. Start simulation or generate Real-Time Workshop code.

Notation for Referring to Enumerated Values in a Chart

In this section...

Nonprefixed Notation for Enumerated Values

To minimize identifier length when referring to enumerated values, use nonprefixed notation. This notation is a string of the form *`Name`*, where *`Name`* is the name of an enumerated value.

If your Stateflow chart uses data types that contain identical enumerated names (such as `Colors.Red` and `Temp.Red`), consider using prefixed notation to prevent name conflicts among identifiers. For details, see "Prefixed Notation for Enumerated Values" on page 12-14.

Requirements for Using Nonprefixed Notation

The requirements for using nonprefixed notation are:

- The enumerated data type definition is in an M-file on the MATLAB search path.
- One of the following is true:
  - Enumerated data of this type exists in the chart.
  - A prefixed identifier for this data type exists in the chart.

Example of Nonprefixed Notation in Stateflow Action Language

Suppose that you have an identifier with nonprefixed notation: `Red`. The enumerated name `Red` belongs to the data type `TrafficColors`.

You can meet the requirements for nonprefixed notation as follows:

- Define `TrafficColors` as an enumerated data type in an M-file on the MATLAB search path.

- Verify that one of the following is true:
  - Enumerated data of this type exists in the chart.
  - A prefixed identifier for this data type exists in the chart, such as `TrafficColors.Yellow` or `TrafficColors.Green`.

Prefixed Notation for Enumerated Values

To prevent name conflicts when referring to enumerated values, use prefixed notation. This notation is a string of the form *`Type`*`.`*`Name`*, where *`Type`* is an enumerated data type and *`Name`* is the name of an enumerated value.

Suppose that you have 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`.

Requirement for Using Prefixed Notation

The only requirement for using prefixed notation is that the enumerated data type definition is in an M-file on the MATLAB search path.

Example of Prefixed Notation in Stateflow Action Language

Suppose that you have an identifier with prefixed notation: `TrafficColors.Red`. The enumerated name `Red` belongs to the data type `TrafficColors`.

You can meet the requirement for prefixed notation by defining `TrafficColors` as an enumerated data type in an M-file on the MATLAB search path.

Operations on Enumerated Data in Stateflow Action Language

These operations work with enumerated operands.

Example	Description
`a = exp`	Assignment of `exp`, which must evaluate to an enumerated value
`a == b`	Comparison, equality
`a != b`	Comparison, inequality

How to View Enumerated Values in a Stateflow Chart

In this section...

Viewing Values of Enumerated Data During Simulation

To view the values of enumerated data during simulation, follow these steps:

1. Open the Stateflow Debugger.

   For example, select **Tools > Debug** in the Stateflow Editor.

2. In the Stateflow Debugger, select breakpoints.

3. Click **Start** to simulate the model.

4. During simulation, select **Browse Data**.

   In the Stateflow Debugger, the values of enumerated data appear by name. (For more information, see "Watching Data in the Stateflow Debugger" on page 23-30.)

Viewing Values of Enumerated Data After Simulation

To view the values of enumerated data after simulation, follow these steps:

1. Open the Model Explorer.

   For example, select **View > Model Explorer** in the Stateflow Editor.

2. In the **Model Hierarchy** pane, select a chart with enumerated data.

3. In the **Contents** pane, right-click an enumerated data and select **Properties**.

   The Data properties dialog box appears.

4 In the **Value Attributes** pane, select **Save final value to base workspace**.

5 Click **OK** to close the Data properties dialog box.

6 Repeat steps 2 through 5 if you want to save the final value of another enumerated data.

7 Simulate the model.

8 After simulation ends, view enumerated data in the base workspace.

In the MATLAB Command Window, the final values of enumerated data appear by underlying numeric value.

Rules for Using Enumerated Data in a Stateflow Chart

These rules apply when you use enumerated data in a chart.

Use the name of the enumerated data type as the name of the M-file that contains the type definition

This rule enables resolution of enumerated data types for Simulink models.

Use a unique name for an enumerated data type

The name of an enumerated data type cannot match the name of another data type or a variable in the MATLAB base workspace. Otherwise, a name conflict occurs.

Do not define enumerated data at the machine level of the hierarchy

Machine-parented data is not supported for enumerated types.

Do not use enumerated data for inputs and outputs of exported functions

This rule applies to graphical functions, truth table functions, and Simulink functions.

Do not assign enumerated values to constant data

Since enumerated values are constants by nature, assigning these values to constant data is redundant and unnecessary. If you try to assign enumerated values to constant data, an error message appears.

Ensure unique name resolution for nonprefixed identifiers

If you use nonprefixed identifiers to refer to enumerated values in a chart, ensure unique name resolution in each case. For requirements, see “Nonprefixed Notation for Enumerated Values” on page 12-13.

Assign to enumerated data only expressions that evaluate to enumerated values

Examples of valid assignments to enumerated data include:

- `y = BasicColors(3)`
- `y = BasicColors.Red`

Use a prefixed identifier to set the initial value of enumerated data

If you choose to set an initial value for enumerated data, you must use a prefixed identifier in the **Initial value** field of the Data properties dialog box. For example, `BasicColors.Red` is a valid identifier, but not `Red`. This rule applies because the initial value must evaluate to a valid MATLAB expression.

For information about prefixed notation, see "Prefixed Notation for Enumerated Values" on page 12-14.

Note This rule also applies if you use the Model Explorer to define enumerated data.

Do not define minimum or maximum values for enumerated data

How the **Minimum** and **Maximum** fields appear in the **Value Attributes** pane of the Data properties dialog box depends on which option you use to define enumerated data.

If you select this option in the Type field of the General pane...	The Minimum and Maximum fields in the Value Attributes pane appear...
`Enum: <class name>`	Grayed out
`<data type expression>` or `Inherit from Simulink`	Available

Leave the **Minimum** and **Maximum** fields empty for enumerated data. Any values you enter in these fields are ignored.

Do not use the ml namespace operator to access enumerated data from the MATLAB base workspace

This operator does not support enumerated data. For more information, see “ml Namespace Operator” on page 10-33.

Do not build custom targets for models with enumerated data

This rule applies because only simulation and Real-Time Workshop code generation support enumerated data. If you use Stateflow® Coder™ software to build a standalone custom target for a model that contains enumerated data, an error message appears.

For information about custom targets, see “How to Build a Stateflow Custom Target” on page 22-50.

Best Practices for Using Enumerated Data in a Stateflow Chart

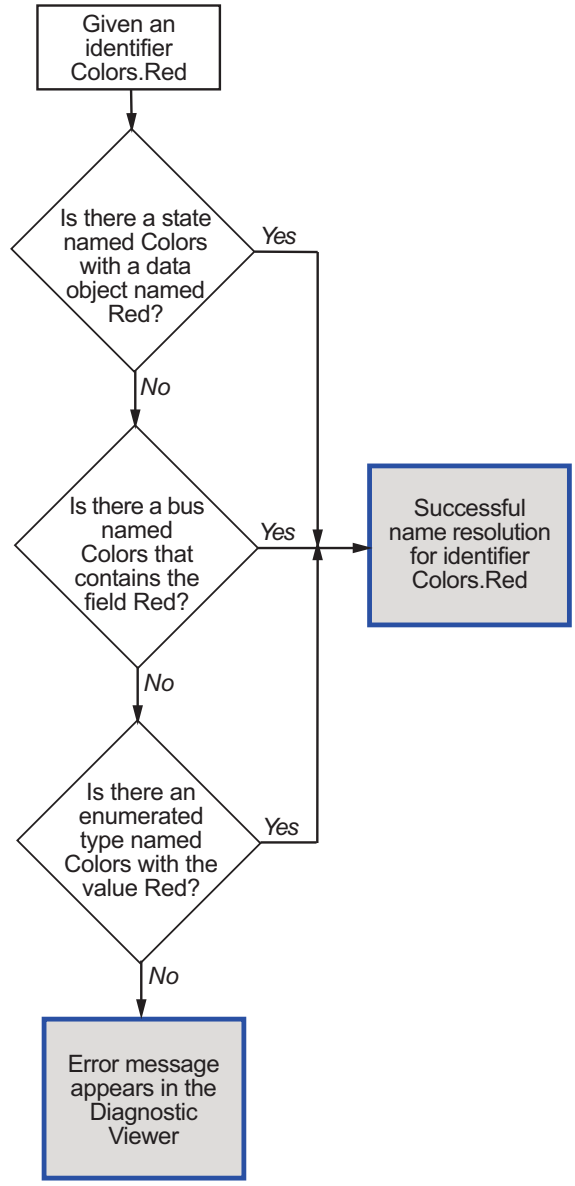
Add prefixes to enumerated names to enhance readability of Real-Time Workshop generated code

If you add prefixes to enumerated names in Real-Time Workshop generated code, you enhance readability and avoid name conflicts with global symbols. For details, see "Enumerated Data Types in Generated Code" in the Real-Time Workshop User's Guide.

Use unique identifiers to refer to enumerated values in Stateflow action language

This guideline prevents name conflicts with other objects in a chart. If an enumerated value uses the same identifier as a data object in a state or a bus field in a chart, the chart does not resolve the identifier as an enumerated value.

For example, the following diagram shows the stages in which a chart tries to resolve the identifier `Colors.Red`.

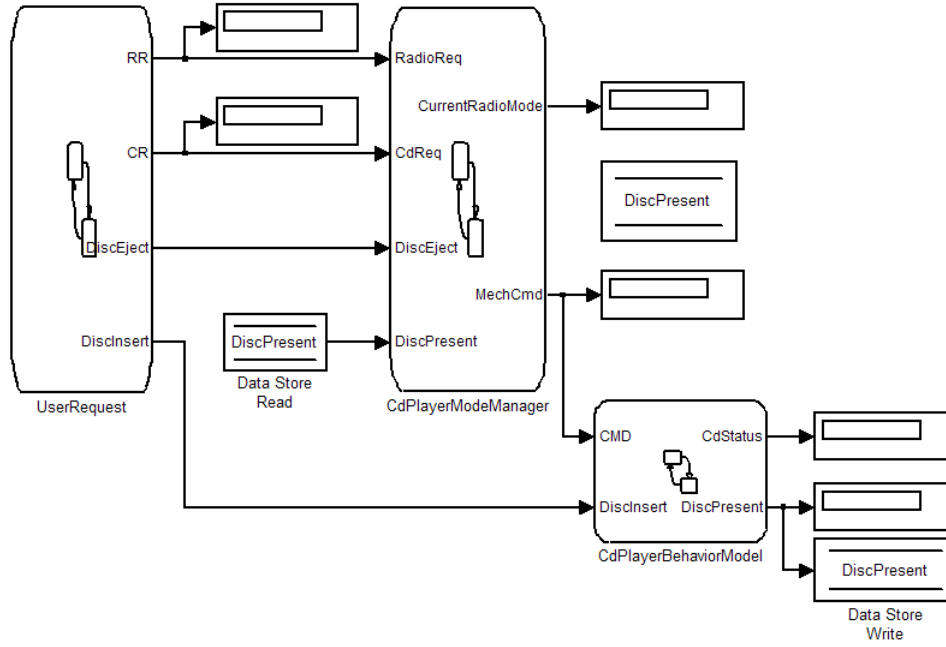

Given an identifier Colors.Red
Is there a state named Colors with a data object named Red?
Yes
No
Is there a bus named Colors that contains the field Red?
Yes
No
Is there an enumerated type named Colors with the value Red?
Yes
No
Successful name resolution for identifier Colors.Red
Error message appears in the Diagnostic Viewer

CD Player Model That Uses Enumerated Data

In this section...

Overview of CD Player Model

This Simulink model implements a basic CD player using enumerated data in three Stateflow charts.



Model Component	Description	Details
UserRequest chart	Reads and stores user inputs	"How the UserRequest Chart Works" on page 12-29
CdPlayerModeManager chart	Determines whether the CD player operates in CD or radio mode	"How the CdPlayerModeManager Chart Works" on page 12-30
CdPlayerBehaviorModel chart	Describes behavior of the CD player mechanism	"How the CdPlayerBehaviorModel Chart Works" on page 12-32

Benefits of Using Enumerated Types in This Model

This model uses two enumerated data types: `RadioRequestMode` and `CdRequestMode`.

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)`

By grouping related values into separate data types, you get these benefits:

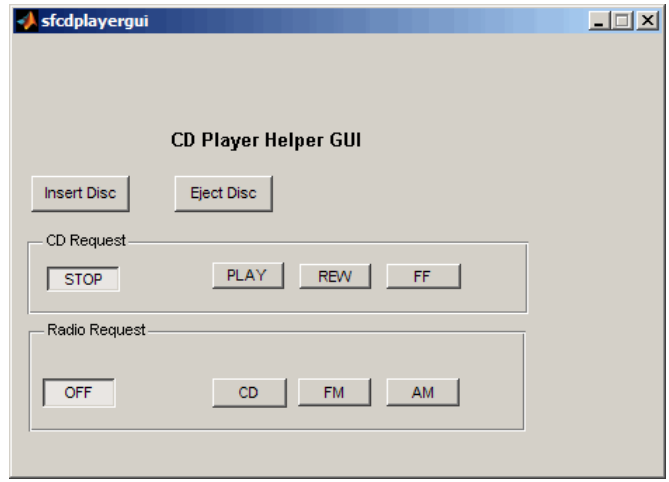
- Enhance readability of data values in each chart.
- Avoid defining a long list of constants, which reduces the amount of data in your model.

Running the CD Player Model

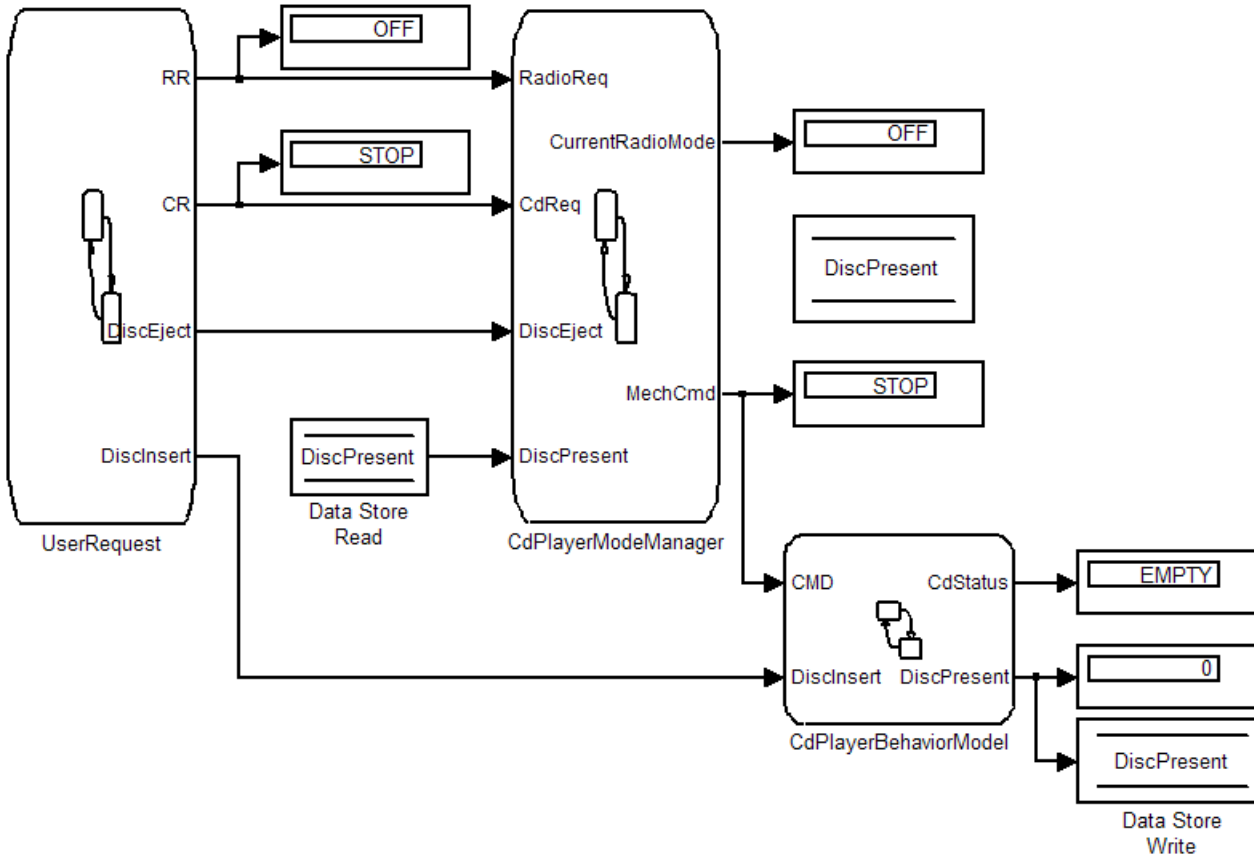
Follow these steps to run the model:

1. Type `sf_cdplayer` at the MATLAB command prompt.
2. Start simulation of the model.

   The CD Player Helper GUI appears.

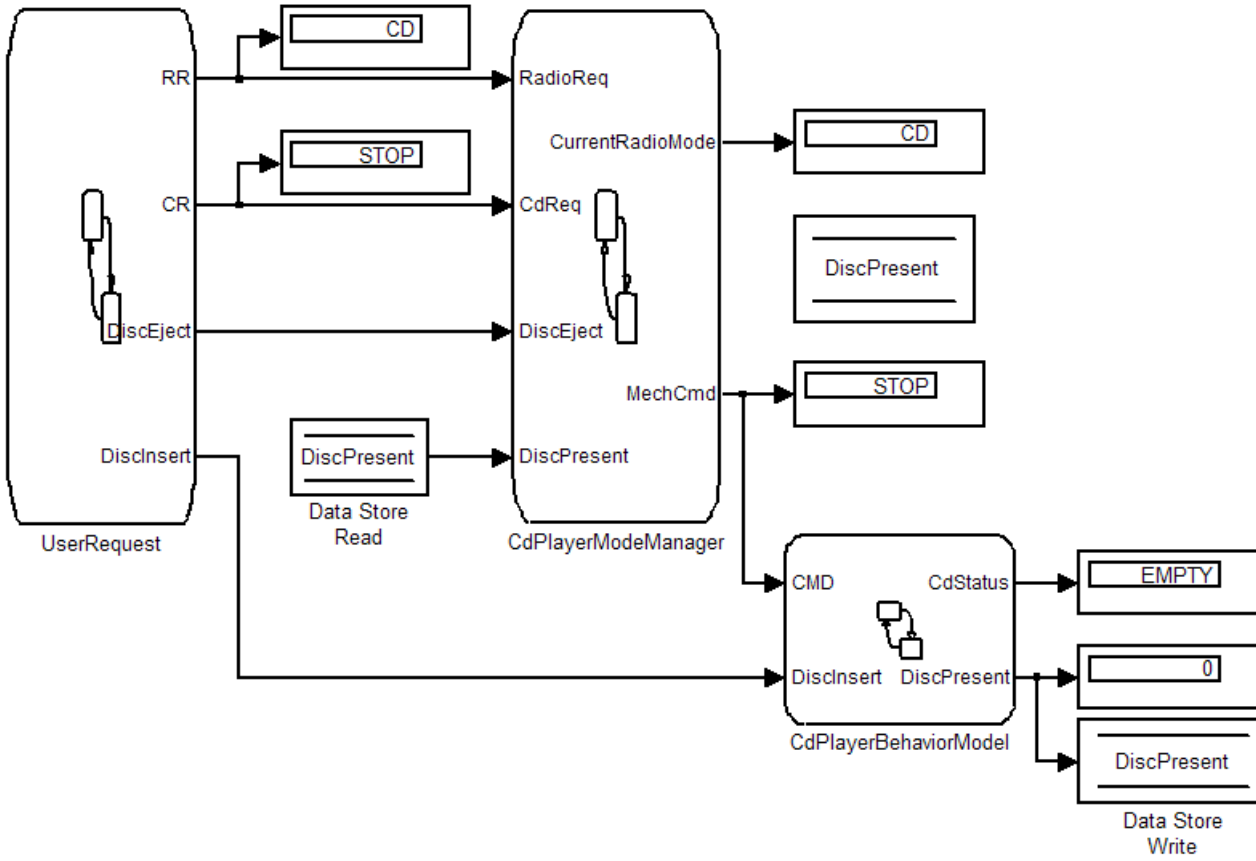


The Display blocks in the model show the default settings of the CD player.



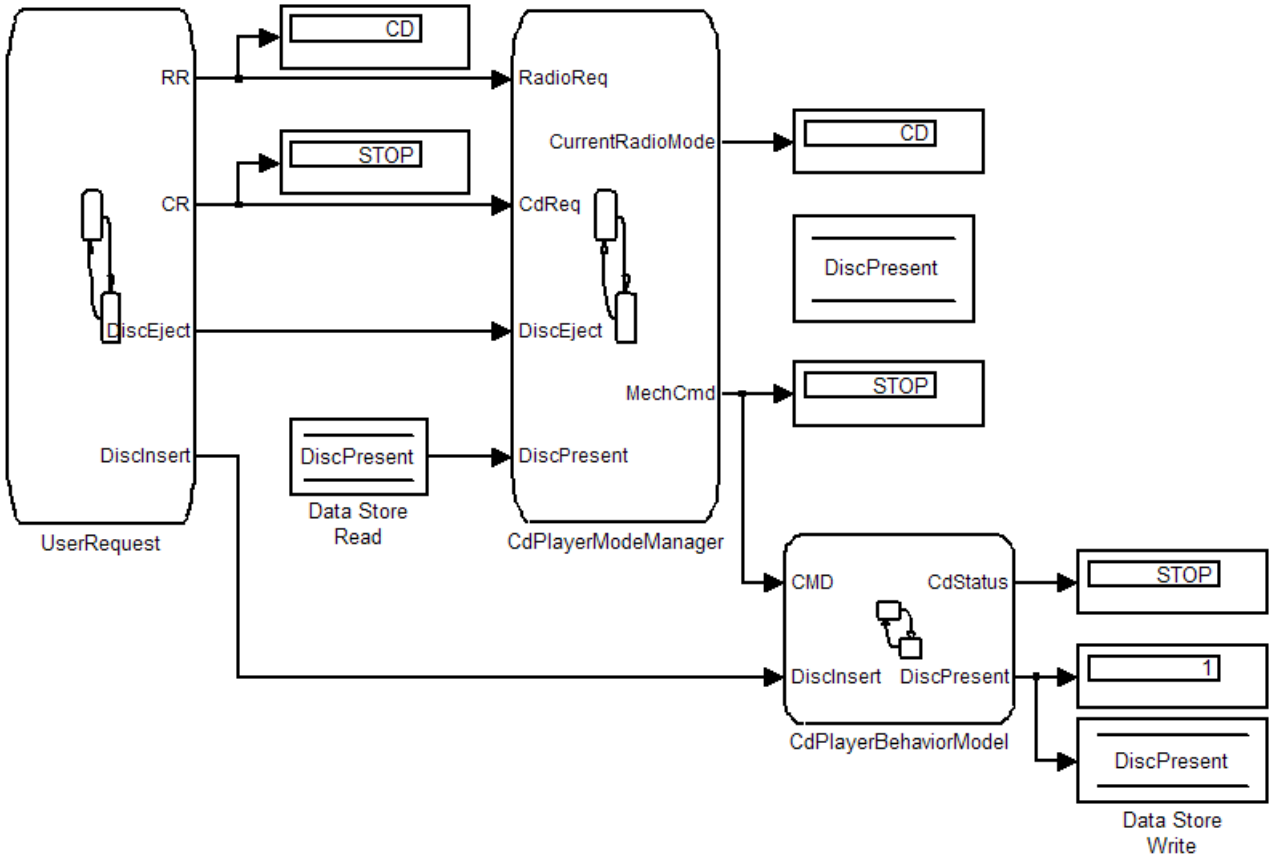
3 In the CD Player Helper GUI, click **CD** in the **Radio Request** section.

The Display blocks for enumerated data `RR` and `CurrentRadioMode` change from `OFF` to `CD`.



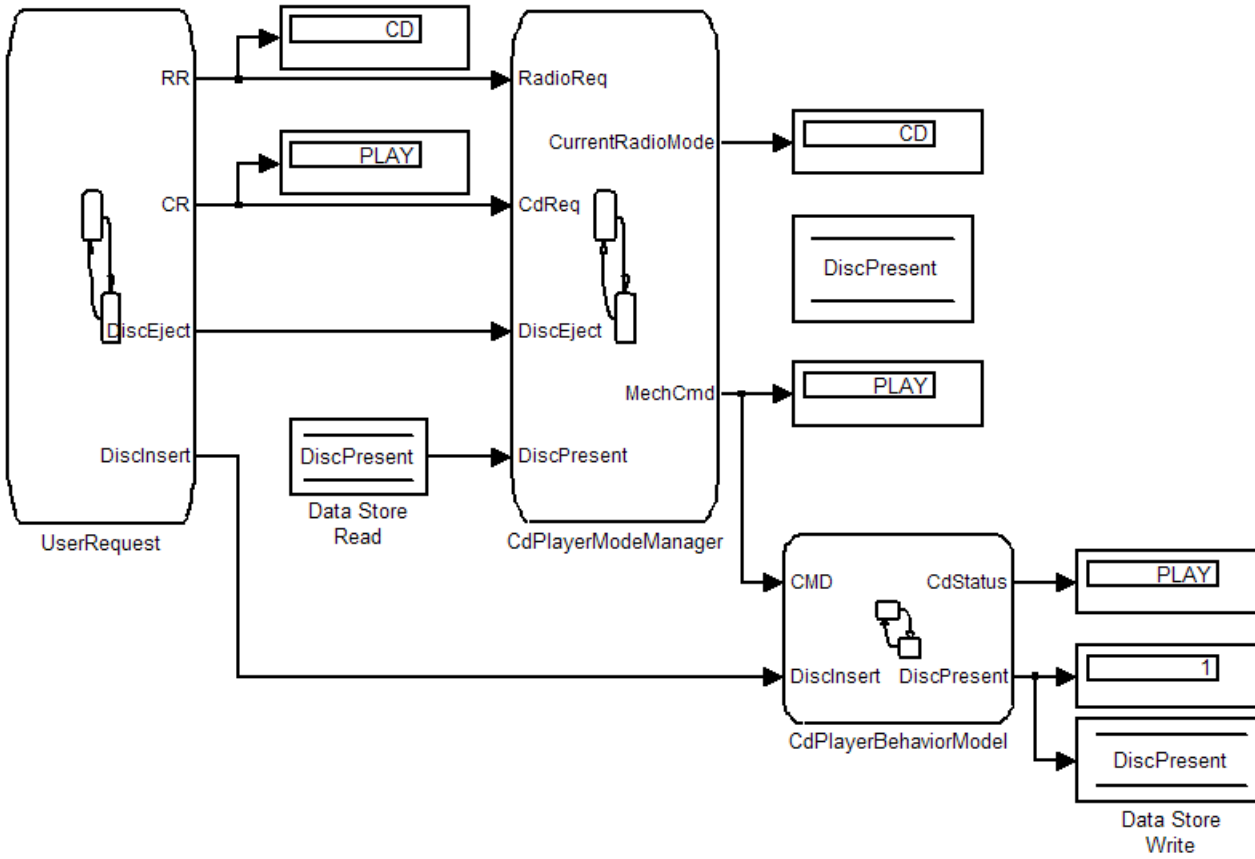
4 In the CD Player Helper GUI, click **Insert Disc**.

The Display block for enumerated data `CdStatus` changes from `EMPTY` to `DISCINSERT` to `STOP`.



5 In the CD Player Helper GUI, click **PLAY** in the **CD Request** section.

The Display blocks for enumerated data `CR`, `MechCmd`, and `CdStatus` change from `STOP` to `PLAY`.

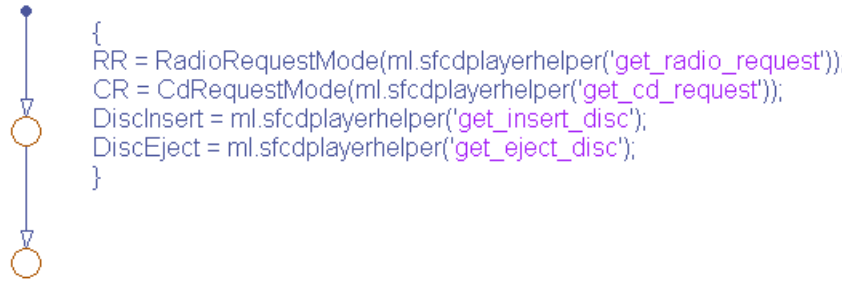


Note To see other changes in the Display blocks, you can select other operating modes for the CD player, such as FM or AM radio.

How the UserRequest Chart Works

Key features of the UserRequest chart include:

- Enumerated data
- `ml` namespace operator (see “ml Namespace Operator” on page 10-33)



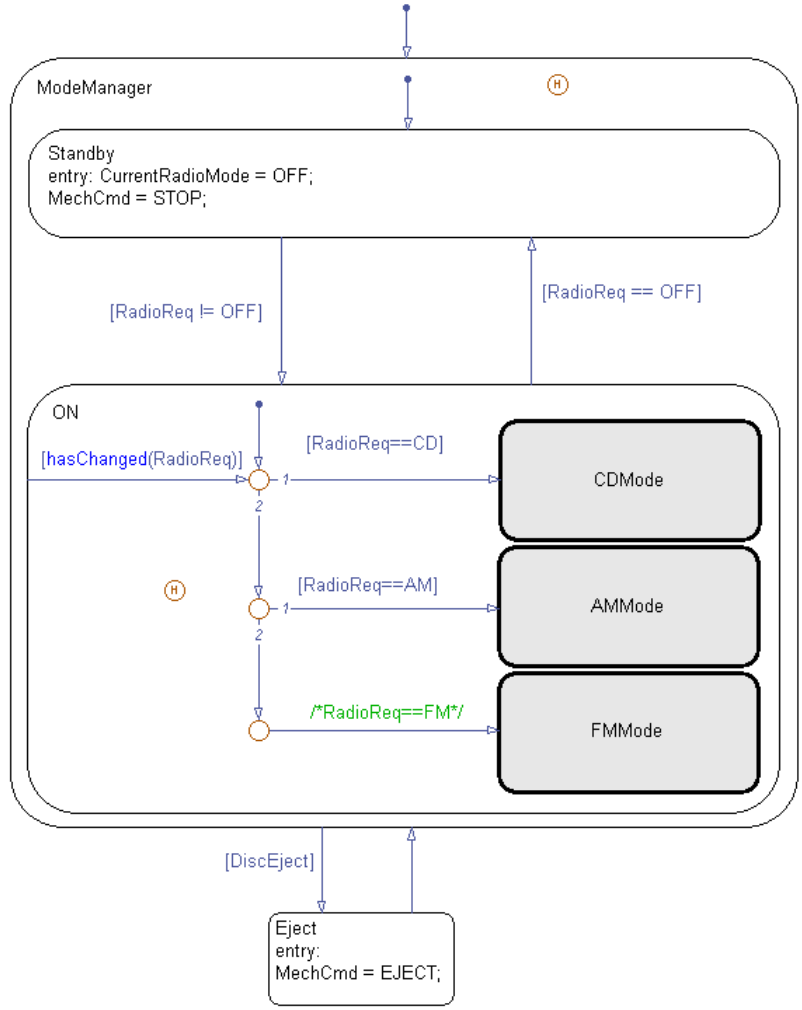
This chart reads user inputs from the CD Player Helper GUI and stores the information as output data.

Output Data Name	Data Type	Description
`RR`	Enumerated	Operating mode of the radio component
`CR`	Enumerated	Operating mode of the CD component
`DiscInsert`	Boolean	Setting for CD insertion
`DiscEject`	Boolean	Setting for CD ejection

How the CdPlayerModeManager Chart Works

Key features of the CdPlayerModeManager chart include:

- Enumerated data
- Subcharts (see “Using Subcharts to Extend Charts” on page 7-5)
- Change detection (see “Using Change Detection in Actions” on page 10-74)



Behavior of the CdPlayerModeManager Chart

1. When the chart wakes up, the `ModeManager` state is entered.
2. The previously active substate recorded by the history junction becomes active: `Standby` or `ON`.

Note Transitions between the `Standby` and `ON` substates occur as follows.

- If the enumerated data `RadioReq` is `OFF`, the `Standby` substate is entered.
- If the enumerated data `RadioReq` is not `OFF`, the `ON` substate is entered. (For details, see "Control of CD Player Operating Mode" on page 12-32.)

3. 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.

4. Steps 2 and 3 repeat until the chart goes to sleep.

Control of CD Player Operating Mode

In the `ON` substate, three subcharts represent the operating modes of a CD player: CD, AM radio, and FM radio. Each subchart corresponds to a different value of enumerated data `RadioReq`.

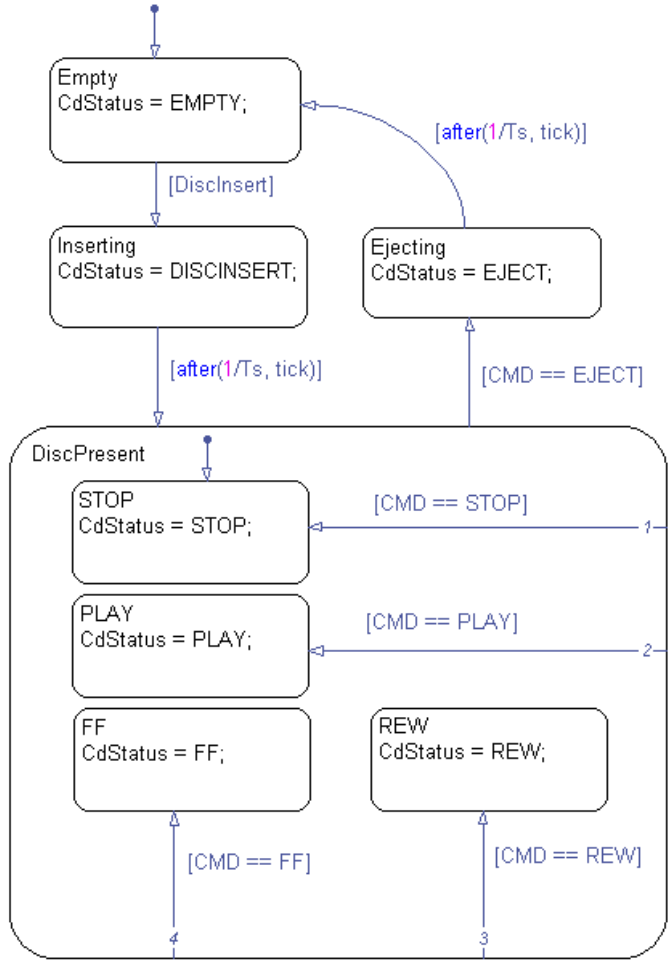
Value of Enumerated Data `RadioReq`	Active Subchart	Purpose of Subchart
`CD`	`CDMode`	Outputs play, rewind, fast forward, and stop commands to the CdPlayerBehaviorModel chart
`AM`	`AMMode`	Sets the CD player to AM radio mode
`FM`	`FMMode`	Sets the CD player to FM radio mode

Note The `hasChanged` operator detects changes in the value of `RadioReq` via an inner transition.

How the CdPlayerBehaviorModel Chart Works

Key features of the CdPlayerBehaviorModel chart include:

- Enumerated data
- Temporal logic (see "Using Temporal Logic in State Actions and Transitions" on page 10-56)



Behavior of the CdPlayerBehaviorModel Chart

1 When the chart wakes up, the `Empty` state is entered.

2 If the Boolean data `DiscInsert` is 1 (or `true`), a transition to the `Inserting` state occurs.

3 After a short time delay, a transition to the `DiscPresent` state occurs.

4 The `DiscPresent` state remains active until the data `CMD` becomes `EJECT`.

5 If the enumerated data `CMD` is `EJECT`, a transition to the `Ejecting` state occurs.

6 After a short time delay, a transition to the `Empty` state occurs.

7 Steps 2 through 6 repeat until the chart goes to sleep.

Update of CD Player Behavior

Whenever a state transition occurs, the enumerated data `CdStatus` changes value to reflect the behavior of the CD player.

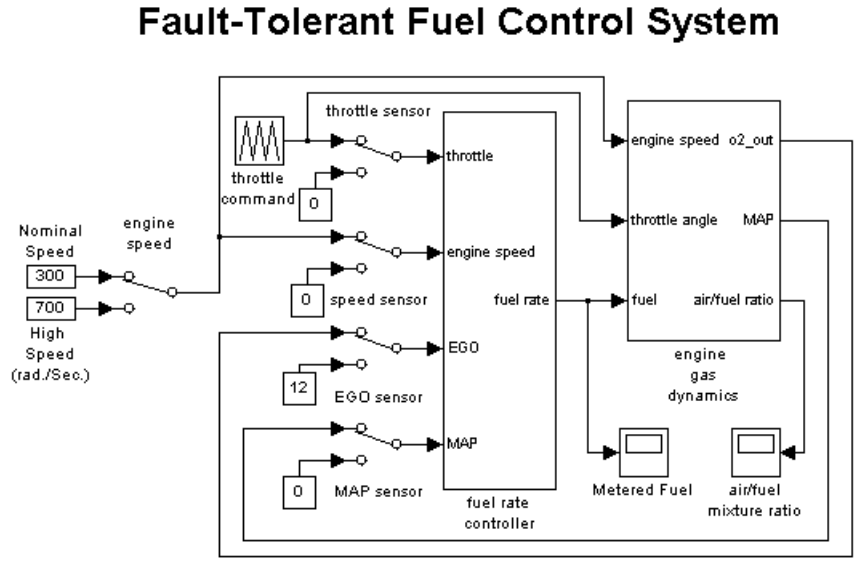
Active State	Value of Enumerated Data `CdStatus`	Behavior of CD Player
`Empty`	`EMPTY`	CD player is empty.
`Inserting`	`DISCINSERT`	CD is being inserted into the player.
`DiscPresent.STOP`	`STOP`	CD is present and stopped.
`DiscPresent.PLAY`	`PLAY`	CD is present and playing.
`DiscPresent.REW`	`REW`	CD is present and rewinding.
`DiscPresent.FF`	`FF`	CD is present and fast forwarding.
`Ejecting`	`EJECT`	CD is being ejected from the player.

Example of Using Enumerated Values for Indexing a Vector

In this section...

Goal of the Example

The goal of this example is to update a model that uses constant data for indexing a vector.



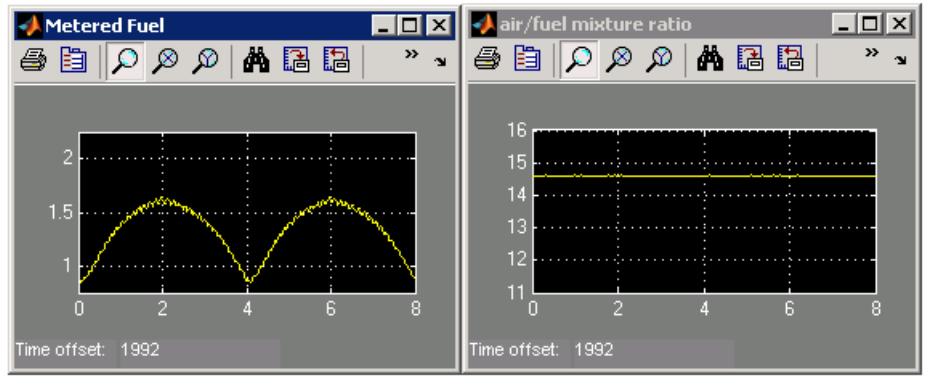
Overview of the Model and Simulation Results

Follow these steps to open the model and view the simulation results:

1 Type `fuelsys` at the command prompt.

For details on how this model works, see “Exploring a Typical Stateflow Application” on page 1-23.

2 Simulate the model to see these results in the scopes.

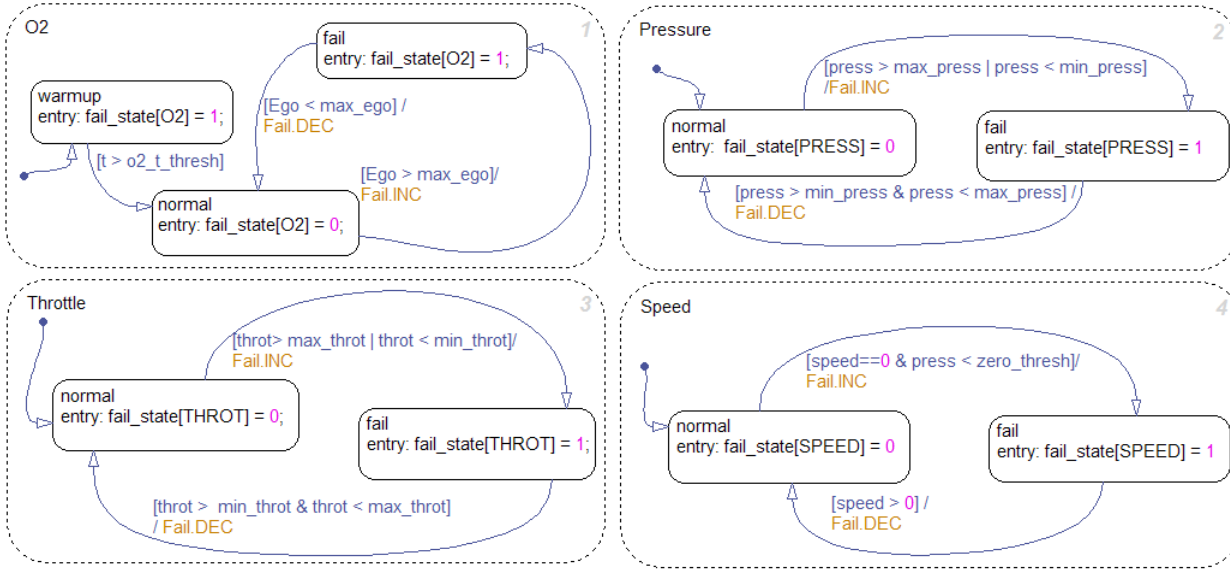


Rationale for Updating the Model

In the subsystem `fuel rate controller`, the chart `control logic` contains four constants:

- `THROT`
- `SPEED`
- `O2`
- `PRESS`

To reduce the amount of data in the model, you can convert the four constants to enumerated values.



Editing a Model to Use an Enumerated Data Type

The sections that follow describe how to convert constants to enumerated values. This procedure reduces the amount of data in the model while retaining the same simulation results.

Task	Description	Reference
1	Define an enumerated data type for the chart.	"Defining an Enumerated Data Type for the Chart" on page 12-38
2	Ensure that identifiers for enumerated values resolve correctly.	"Ensuring Unique Name Resolution for Nonprefixed Identifiers" on page 12-38
3	Remove unused items in the model.	"Deleting Constant Data from the Chart" on page 12-39

Defining an Enumerated Data Type for the Chart

Follow these steps to define an enumerated data type for the Stateflow chart named `control logic`.

1. Open a new M-file (for example, by selecting **File > New > Class M-File** in the MATLAB Command Window).

2. Enter these lines in the MATLAB Editor:

```
classdef(Enumeration) OperatingModes < Simulink.IntEnumType
  enumeration
    THROT(1)
    SPEED(2)
    O2(3)
    PRESS(4)
  end
end
```

   The `classdef` section defines an integer-based enumerated data type named `OperatingModes` that is derived from the built-in type `Simulink.IntEnumType`. The `enumeration` section is the set of enumerated values that this data type allows. Each enumerated name is followed by the underlying numeric value.

3. Save your M-file as `OperatingModes.m` in a directory on the MATLAB search path.

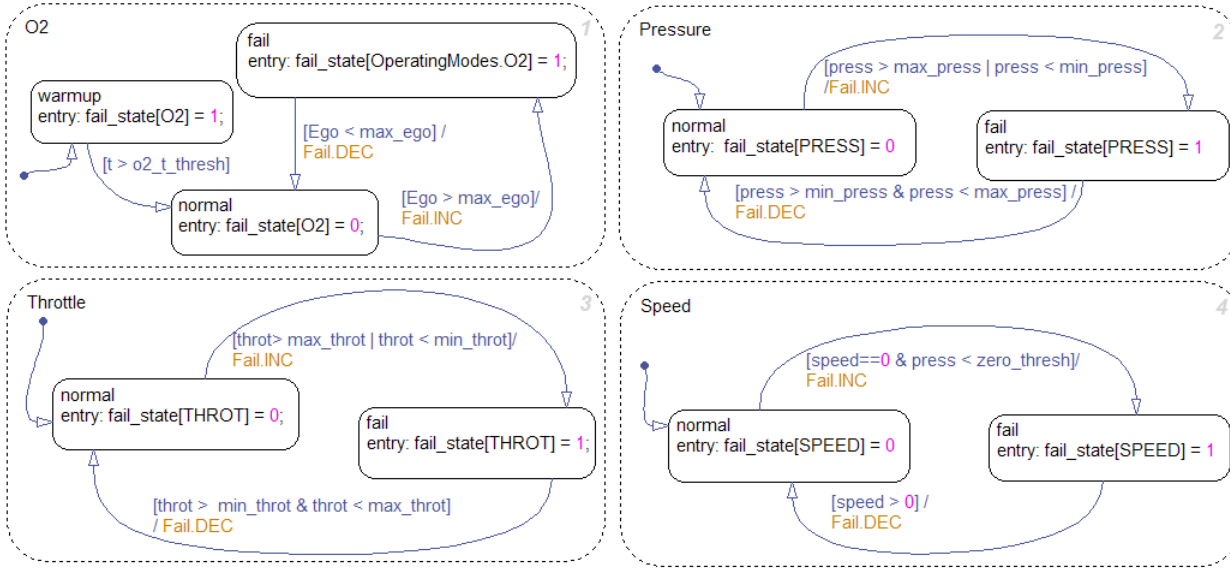
   The name of your M-file must match exactly the name of your data type. Therefore, you must use `OperatingModes.m` as the name of your M-file.

Note To add a directory to the MATLAB search path, type `addpath` *`pathname`* at the command prompt.

Ensuring Unique Name Resolution for Nonprefixed Identifiers

In the chart `control logic`, the states contain nonprefixed identifiers (`THROT`, `SPEED`, `O2`, and `PRESS`) that refer to enumerated values of the data type `OperatingModes`. For the chart to resolve these nonprefixed identifiers correctly, add a prefix to one of them. (See “Requirements for Using

Nonprefixed Notation" on page 12-13.) For example, you can change the `O2` identifier to `OperatingModes.O2` in the state `O2.fail`.



Deleting Constant Data from the Chart

In the Model Explorer, delete the constant data `THROT`, `SPEED`, `O2`, and `PRESS` from the chart `control logic`.

Simulating the New Model

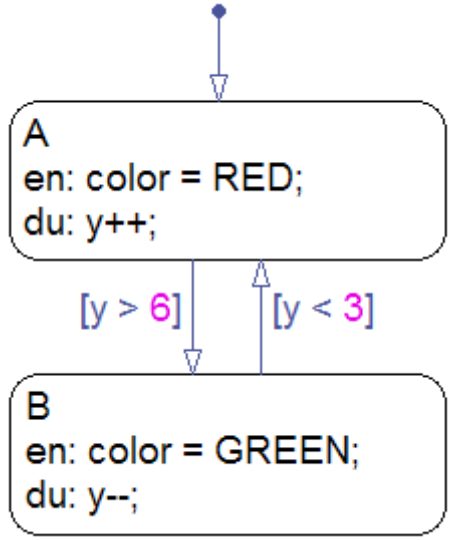
If you simulate the new model, the results match those of the original design.

Example of Using Enumerated Values for Assignment

In this section...

Goal of the Example

The goal of this example is to build a chart that uses enumerated values in assignment statements.



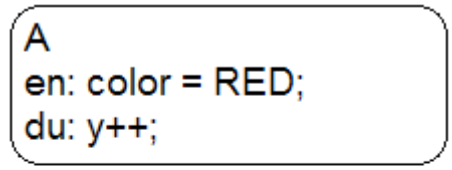
Building the Chart

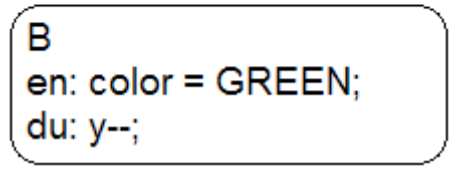
To build the chart, follow these steps.

Adding States and Transitions to the Chart

You can add states and transitions to the chart as follows.

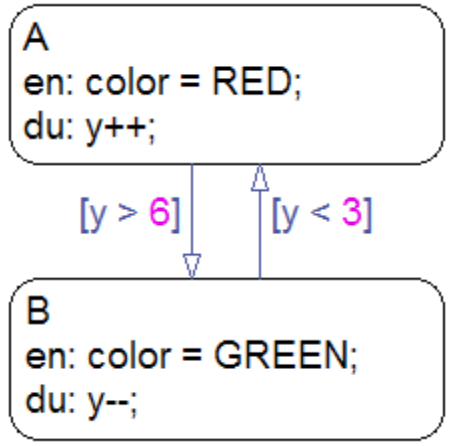
1. Type `sfnew` at the command prompt to create a new model with a chart inside.
2. In the Stateflow Editor, add states `A` and `B` to the chart.



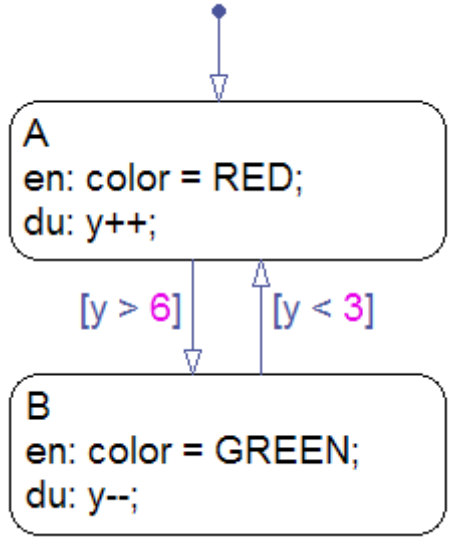


Note You will define the data `color` and `y` in the sections that follow.

3 Add transitions between states A and B.



4 Add a default transition to state A.



Defining an Enumerated Data Type for the Chart

To use enumerated data in the chart, you must define the data type in an M-file.

1. Open a new M-file (for example, by selecting **File > New > Class M-File** in the MATLAB Command Window).

2. Enter these lines in the MATLAB Editor:

```
classdef(Enumeration) TrafficColors < Simulink.IntEnumType
  enumeration
    RED(0)
    YELLOW(5)
    GREEN(10)
  end
end
```

   The `classdef` section defines an integer-based enumerated data type named `TrafficColors` that is derived from the built-in type `Simulink.IntEnumType`. The `enumeration` section is the set of enumerated values that this data type allows. Each enumerated name is followed by the underlying numeric value.

3. Save your M-file as `TrafficColors.m` in a directory on the MATLAB search path.

   The name of your M-file must match exactly the name of your data type. Therefore, you must use `TrafficColors.m` as the name of your M-file.

Note To add a directory to the MATLAB search path, type `addpath` *`pathname`* at the command prompt.

Adding Enumerated Data to the Chart

You can add the enumerated data `color` to the chart as follows.

1. In the Stateflow Editor, select **Add > Data > Output to Simulink**.

   The Data properties dialog box appears.

2 In the **General** pane, enter `color` in the **Name** field.

3 In the **Type** field, select `Enum: <class name>`.

4 Replace `<class name>` with `TrafficColors`, the name of the data type that you defined in an M-file in "Defining an Enumerated Data Type for the Chart" on page 12-43.

5 Click **OK**.

Adding Integer Data to the Chart

You can add the integer data `y` to the chart as follows.

1 In the Stateflow Editor, select **Add > Data > Output to Simulink**.

The Data properties dialog box appears.

2 In the **General** pane, enter `y` in the **Name** field.

3 In the **Type** field, select `uint8`.

4 Click **OK**.

Viewing Results for Simulation

To view results for simulation, follow these steps.

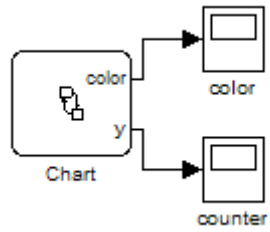
Adding Scopes to View Output

You can add two scopes to your model as follows.

1 Open the Simulink Library Browser.

2 In the Simulink/Sinks library, select the Scope block.

3 Add two scopes to your model as shown.



Setting the Sample Time for Simulation

You can set a discrete sample time for simulation using a fixed-step solver. (For details, see "Solvers" in the *Simulink User's Guide*.)

1 Open the Configuration Parameters dialog box (for example, by selecting **Simulation > Configuration Parameters** in the Stateflow Editor).

2 In the **Solver** pane, select `Fixed-step` in the **Type** field.

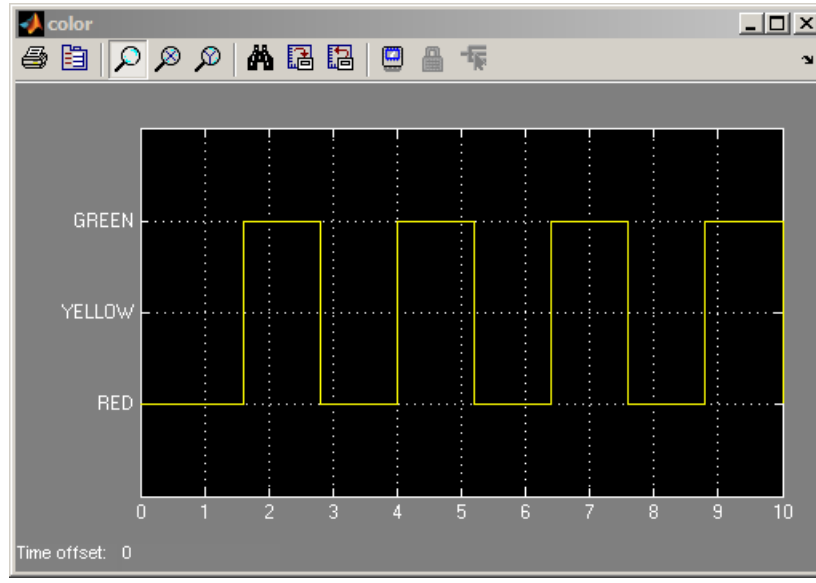
3 Select `Discrete (no continuous states)` in the **Solver** field.

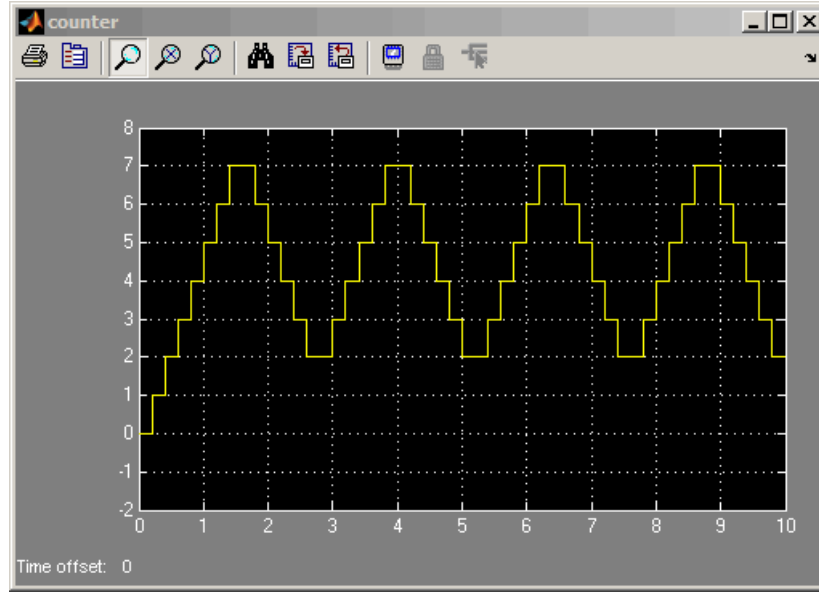
4 Enter `0.2` in the **Fixed-step size (fundamental sample time)** field.

5 Click **OK**.

Starting Simulation

If you start simulation, you see these results in the scopes.


color
GREEN
YELLOW
RED
0
1
2
3
4
5
6
7
8
9
10
Time offset: 0


counter
8
7
6
5
4
3
2
1
0
-1
-2
0
1
2
3
4
5
6
7
8
9
10
Time offset: 0

How the Chart Works

During simulation, the chart works as follows.

Stage 1: Execution of State A

1. After the chart wakes up, state `A` is entered.
2. State `A` executes the `entry` action by assigning the value `RED` to the enumerated data `color`.
3. The data `y` increments once per time step (every 0.2 seconds) until the condition `[y > 6]` is true.
4. The chart takes the transition from state `A` to state `B`.

Stage 2: Execution of State B

1. After the transition from state `A` occurs, state `B` is entered.
2. State `B` executes the `entry` action by assigning the value `GREEN` to the enumerated data `color`.
3. The data `y` decrements once per time step (every 0.2 seconds) until the condition `[y < 3]` is true.
4. The chart takes the transition from state `B` to state `A`.

Stage 3: Repeat of State Execution

States `A` and `B` take turns executing until the simulation ends.

13

Modeling Continuous-Time Systems in Stateflow Charts

About Continuous-Time Modeling

In this section...

What Is Continuous-Time Modeling?

Continuous-time modeling allows you to simulate hybrid systems that use mode logic — that is, systems that respond to both continuous and discrete mode changes. 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 changes direction and velocity due to a sudden loss of energy. A later exercise shows you how to model a bouncing ball in continuous-time using a Stateflow chart (see "Modeling a Bouncing Ball in Continuous-Time" on page 13-15).

When you configure Stateflow charts for continuous-time simulation, they interact with the Simulink solver in the same way as other continuous blocks, as follows:

- Maintain mode in minor time steps.

  Stateflow charts do not update mode in minor time steps. This behavior ensures that outputs computed in a minor time step are based on the state of the chart during the last major time step.

- Compute the state of the chart at each time step and expose the state derivative to the Simulink solver.

  You can define local continuous variables to hold state information. Stateflow charts automatically provide programmatic access to the derivatives of state variables. Continuous solvers in Simulink models use this data to compute the chart's continuous states at the current time step, based on values from the previous time steps and the state derivatives.

Note For more information on how solvers work, see "Solvers" in the Simulink User's Guide documentation.

- Can register zero crossings on state transitions.

  Stateflow charts can register a zero-crossings function with a Simulink model to help determine when a state transition occurs. When the Simulink solver detects a change of mode, it searches forward from the previous major time step to detect when the zero crossing — or state transition — occurred.

Note For more information about how a Simulink model uses zero-crossing detection to simulate discontinuities in continuous states, see "Zero-Crossing Detection" in the Simulink User's Guide documentation.

When To Use Stateflow Charts for Continuous-Time Modeling

Use Stateflow charts for modeling hybrid systems with modal behavior — that is, systems that transition from one mode to another in response to physical events and conditions, where each mode is governed by continuous-time dynamics.

In Stateflow charts, you can represent mode logic succinctly and intuitively as a series of states, transitions, and flow graphs. You can also easily represent state information as continuous local variables with automatic access to time derivatives, as described in "About Continuous-Time Variables" on page 13-12.

If your continuous or hybrid system does not contain mode logic, consider using a Simulink model (see "Modeling a Continuous System" in the Simulink software documentation).

Running Demos of Continuous-Time Modeling

You can run the following demos of continuous-time modeling with zero-crossing detection.

Demo	Description
Modeling a Rectifier with Zero Crossings	Rectifier takes a single (scalar) input and converts it to its absolute value. Illustrates how Stateflow charts register zero-crossing variables with Simulink models for accurate detection of mode changes.
Modeling a Bouncing Ball	Demonstrates how to model the dynamics of a bouncing ball by defining continuous-time state variables and their derivatives in Stateflow charts. To try it yourself, see "Modeling a Bouncing Ball in Continuous-Time" on page 13-15.
Modeling Newton's Cradle	Demonstrates how to model elastic collisions between balls in Newton's Cradle, a device that demonstrates conservation of momentum and energy. Uses vector assignment to continuous-time state variables.
Modeling a Clutch	Implements the Simulink clutch demo model purely in a Stateflow chart. Represents the modal nature of the clutch using two states, `Locked` and `Slipping`.
Modeling the Opening Shot in Pool	Demonstrates how to model continuous systems that have a large number of discontinuous events which rapidly (and unpredictably) change the dynamics.

To run these continuous-time demos:

1 At the MATLAB prompt, type:

```
demo simulink stateflow
```

A list of Stateflow software demos appears in the MATLAB Help browser.

2. In the Help Navigator pane, click **Zero Crossings and Derivatives in Stateflow**.

3. In the right pane, select the demo of interest and follow the instructions.

Workflow for Creating Continuous-Time Charts

Here are the tasks for modeling hybrid systems containing mode logic in continuous-time using Stateflow charts:

Step	Task	Example in Bouncing Ball Model
1	Configure the chart to update in continuous-time.	"Task 1: Configure the Bouncing Ball Chart for Continuous Updating" on page 13-16
2	Decide whether to detect zero crossings.	"Task 2: Decide Whether to Enable Zero-Crossing Detection for the Bouncing Ball" on page 13-16
3	Define continuous-time variables.	"Task 3: Define Continuous-Time Variables for Position and Velocity" on page 13-16
4	Choose a solver that supports continuous states (see "Choosing a Solver" in the Simulink User's Guide documentation).	"Task 4: Choose a Solver for the Bouncing Ball Chart" on page 13-17
5	Add system dynamics.	"Task 5: Add Dynamics for a Free-Falling Ball" on page 13-18
6	Expose continuous states to a Simulink model.	"Task 6: Expose Ball Position and Velocity to the Simulink Model" on page 13-20
7	Validate semantics, based on design considerations for continuous-time simulation.	"Task 7: Validate Semantics of Bouncing Ball Chart" on page 13-20
8	Simulate the chart.	"Task 8: Simulate Bouncing Ball Chart" on page 13-20
9	Debug and revise.	"Task 9: Check for the Bounce" on page 13-23

Configuring a Stateflow Chart to Update in Continuous-Time

Continuous updating is a Stateflow chart property. To set this property, follow these steps:

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

   The Chart properties dialog box appears.

2. In the Chart properties dialog box, set update method to **Continuous**.

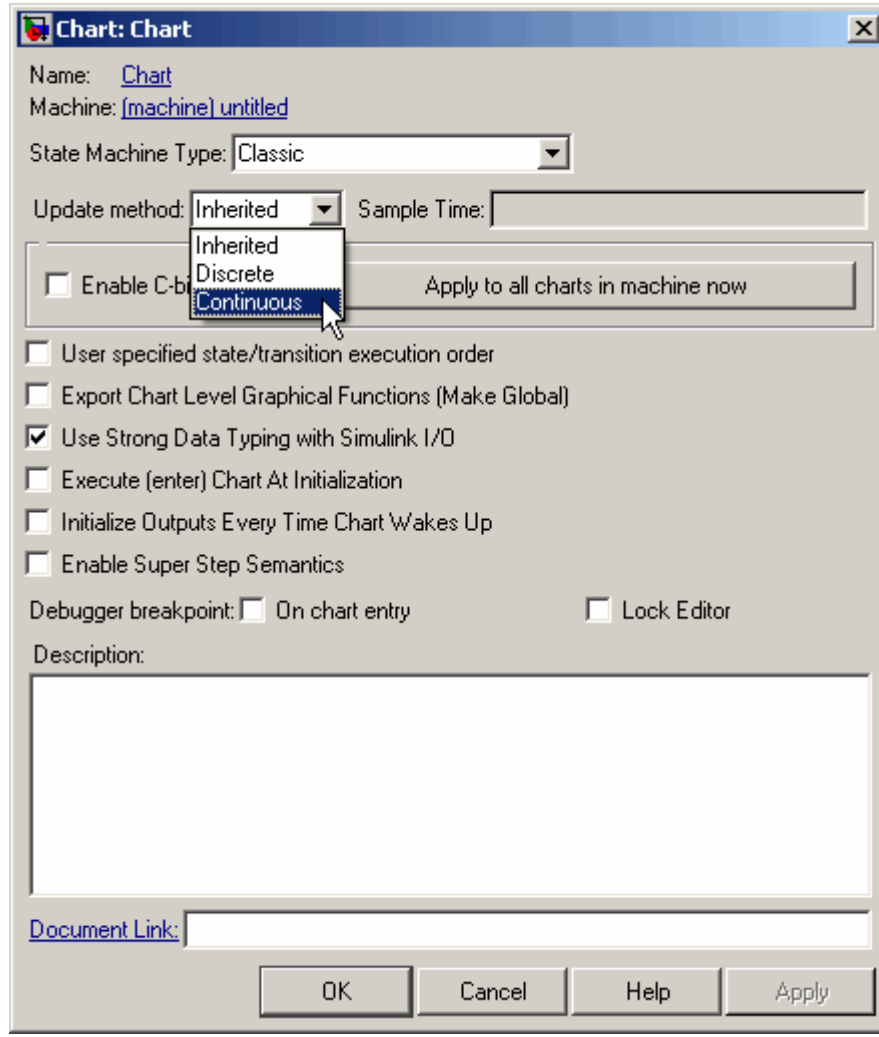
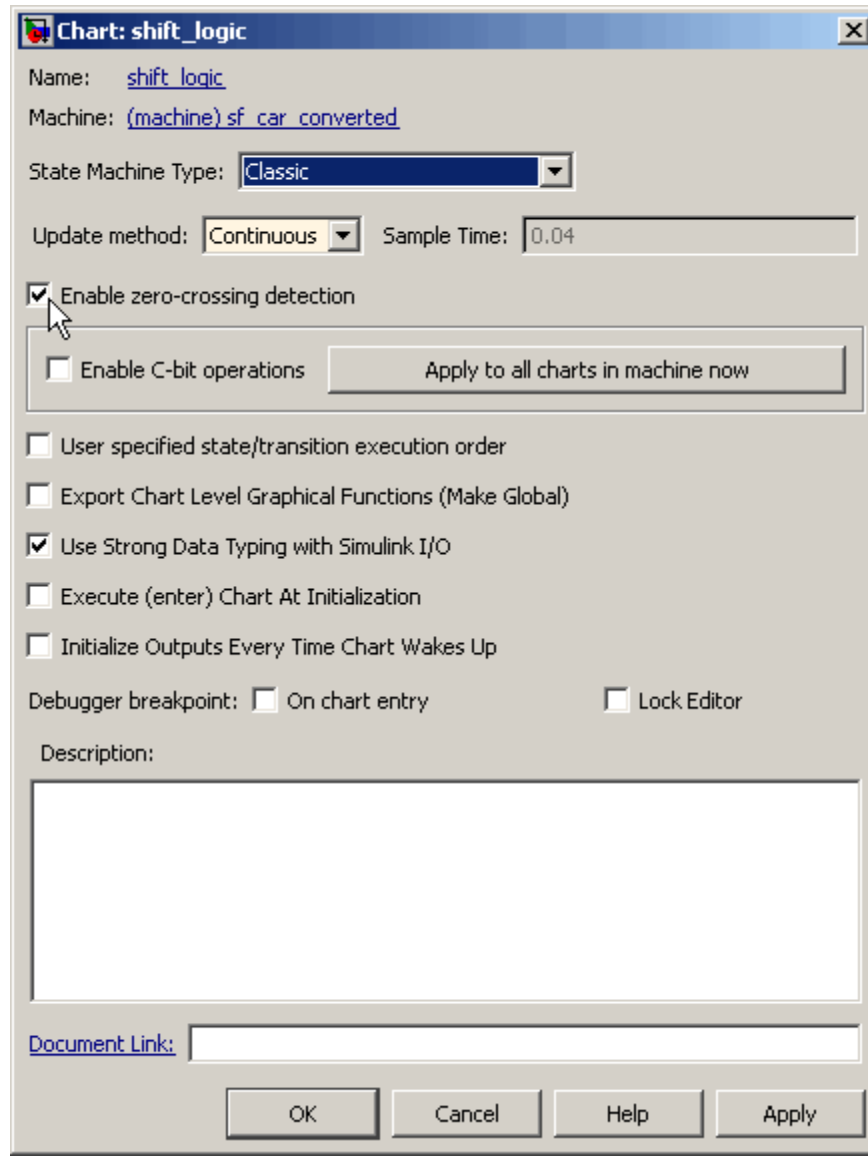


Chart: Chart
Name: Chart
Machine: (machine) untitled
State Machine Type: Classic
Update method: Inherited
Sample Time:
Inherited
Discrete
Continuous
Enable C-b
Apply to all charts in machine now
User specified state/transition execution order
Export Chart Level Graphical Functions (Make Global)
Use Strong Data Typing with Simulink I/O
Execute (enter) Chart At Initialization
Initialize Outputs Every Time Chart Wakes Up
Enable Super Step Semantics
Debugger breakpoint:
On chart entry
Lock Editor
Description:
Document Link:
OK
Cancel
Help
Apply

Note When you set update method to **Continuous**, the Stateflow chart automatically:

- Enables zero-crossing detection



- Disables super step semantics

.

After you select the continuous update method, note that zero-crossing detection is enabled by default.

3 Decide whether or not to enable zero-crossing detection, based on considerations described in "When to Enable Zero-Crossing Detection" on page 13-11.

Note You can choose from different zero-crossing detection algorithms in the **Solver** pane of the Configuration Parameters dialog box. See "Zero-Crossing Algorithms" in the Simulink User's Guide documentation for details.

---

4 Click **OK**.

When to Enable Zero-Crossing Detection

Whether or not to enable zero-crossing detection on state transitions can be a trade-off between accuracy and performance. Generally when detecting zero crossings, a Simulink model accurately simulates mode changes without unduly reducing step size. However, for systems that exhibit *chattering* — frequent fluctuations between two modes of continuous operation — enabling zero-crossing detection may impact simulation time. Chattering requires a Simulink model to check for zero crossings in rapid succession, resulting in excessively small step sizes which can slow simulation. In these situations, you can disable zero-crossing detection, choose a different zero-crossing detection algorithm for your chart, or modify parameters that control the frequency of zero crossings in your Simulink model. See "Preventing Excessive Zero Crossings" in the Simulink User's Guide documentation.

Defining Continuous-Time Variables

In this section...

About Continuous-Time Variables

To compute a continuous state, you must determine its rate of change, or derivative. You can represent this information using **local** variables that update in continuous-time. In a Stateflow chart, continuous-time variables are always `double` type. You cannot change the type, but you can change the size.

Implicit Time Derivatives

For each continuous variable you define, a Stateflow chart implicitly creates a variable to represent its time derivative. A chart denotes time derivative variables as *variable_name*`_dot`. For example, the time derivative of continuous variable `x` is `x_dot`. You can write to the time derivative variable in the `during` action of a state. The time derivative variable does not appear in the Model Explorer.

Note You should **not** explicitly define variables with the suffix `_dot` in a Stateflow chart.

Rules for Using Continuous-Time Variables

Follow these rules when defining and using continuous-time variables:

- Scope must be `local`.
- Define continuous-time variables at the chart level or below in the Stateflow object hierarchy.

- Expose continuous state by assigning the local variable to a Stateflow output (see “Exposing Continuous States to a Simulink Model” on page 13-14).

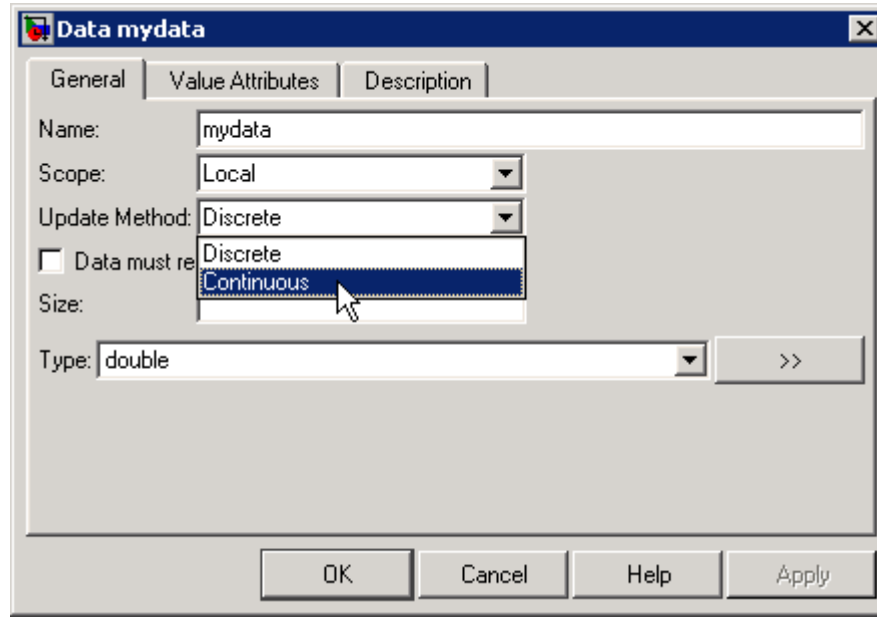
How to Define Continuous-Time Variables

To define continuous-time variables, follow these steps:

1. Configure your chart to update in continuous-time, as described in “Configuring a Stateflow Chart to Update in Continuous-Time” on page 13-7.

2. Add local data to your Stateflow chart in the Stateflow Editor or Model Explorer.

3. In the properties dialog box for your local data, set Update Method to **Continuous**.



In this example, the Stateflow chart automatically creates the variable `mydata_dot` to represent the time derivative of this data.

Note When you set a variable to update in continuous-time, you cannot bind that data to a Simulink signal.

Exposing Continuous States to a Simulink Model

In a Stateflow chart, you represent continuous state using local variables, not inputs or outputs (see "About Continuous-Time Variables" on page 13-12). To expose the continuous states to a Simulink model, you must explicitly assign the local variables to Stateflow outputs in the `during` action of the state. For examples, see "Modeling a Bouncing Ball in Continuous-Time" on page 13-15.

Modeling a Bouncing Ball in Continuous-Time

In this section...

Try It

The following topics give you step-by-step instructions for modeling a bouncing ball as a Stateflow chart in continuous-time using the workflow described in "Workflow for Creating Continuous-Time Charts" on page 13-6.

Dynamics of a Bouncing Ball

The dynamics of a bouncing ball describes the ball as it falls, when it hits the ground, and when it bounces back up.

You can specify how the ball falls freely under gravity using the following second-order differential equation:

$$\ddot{p} = -g$$

In this equation, p describes the position of the ball as a function of time, and $g = 9.81 m/s^2$, which is the acceleration due to gravity.

A Stateflow chart requires that you specify system dynamics as first-order differential equations. You can describe the dynamics of the free-falling ball in terms of position `p` and velocity `v` using the following first-order differential equations:

Equation	Description
$\dot{p} = v$	Derivative of position is velocity
$\dot{v} = -9.81$	Derivative of velocity is acceleration

The bounce occurs after the ball hits the ground at position p <= 0. At this point in time, you can model the bounce by updating position and velocity as follows:

- Reset position to 0
- Reset velocity to the negative of its value just before the ball hit the ground
- Multiply the new velocity by a coefficient of distribution (-0.8) that reduces the speed just after the bounce

Modeling the Bouncing Ball

The following steps take you through the workflow for modeling a bouncing ball in continuous-time. To view the completed model, open the bouncing ball demo.

Task 1: Configure the Bouncing Ball Chart for Continuous Updating

1. Create an empty Stateflow chart and open its properties dialog box.

   If you need instructions, see "Creating a Stateflow Chart" on page 4-2.

2. In the General panel of the properties dialog box, set the update method to **Continuous**.

Task 2: Decide Whether to Enable Zero-Crossing Detection for the Bouncing Ball

For this example, enable zero-crossing detection (the default) so that the Simulink model can determine exactly when the ball hits the ground at p <= 0. Otherwise, the Simulink model may not be able to simulate the physics accurately. For example, the ball may appear to descend below ground.

Task 3: Define Continuous-Time Variables for Position and Velocity

1. Define two continuous-time variables, p for position and v for velocity. For each variable, follow these steps:

a In the Stateflow Editor, select **Add > Data > Local**.

b Enter the name for the local data.

c Set the update method to **Continuous**.

d Leave all other properties at their default values and click **OK**.

Note For each continuous local variable that you define, the Stateflow chart implicitly creates its time derivative as a variable of the same name with the suffix `_dot`. In this example, the Stateflow chart defines `p_dot` as the derivative of position `p` and `v_dot` as the derivative of velocity `v`.

---

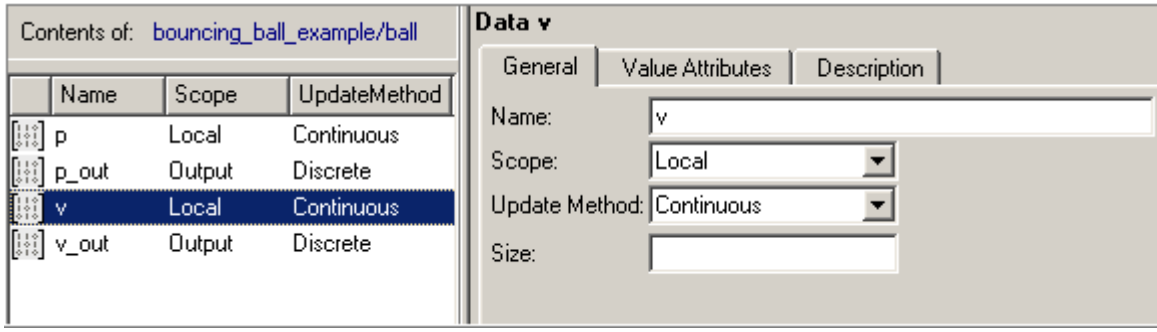
2 Define two outputs, `p_out` and `v_out` for exposing continuous state to the Simulink model. For each variable, follow these steps:

a In the Stateflow Editor, select **Add > Data > Output to Simulink**.

b Enter the name for the output data.

c Leave all other properties at their default values and click **OK**.

Your Stateflow chart should contain the following data (as viewed in the Model Explorer):

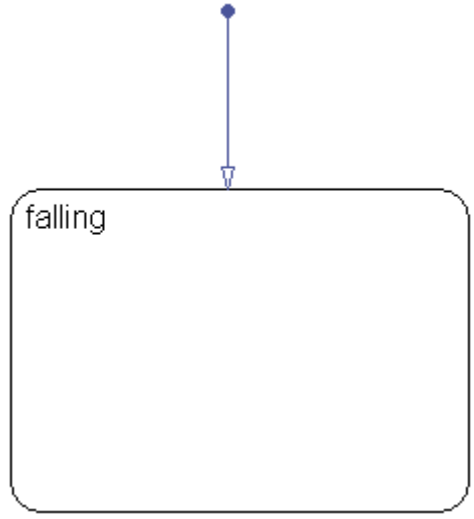


Task 4: Choose a Solver for the Bouncing Ball Chart

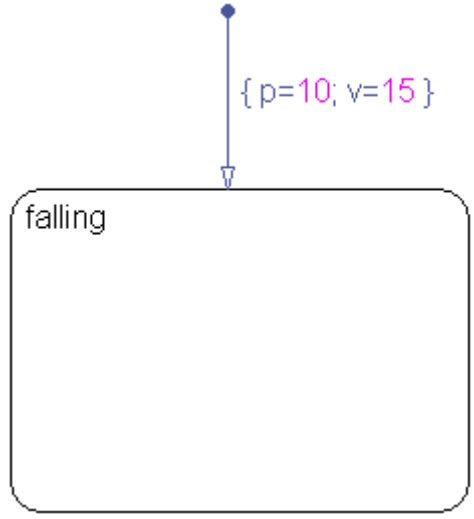
For this example, you can use `ode45 (Dormand-Prince)`, the default variable-step solver used by Simulink models with continuous states.

Task 5: Add Dynamics for a Free-Falling Ball

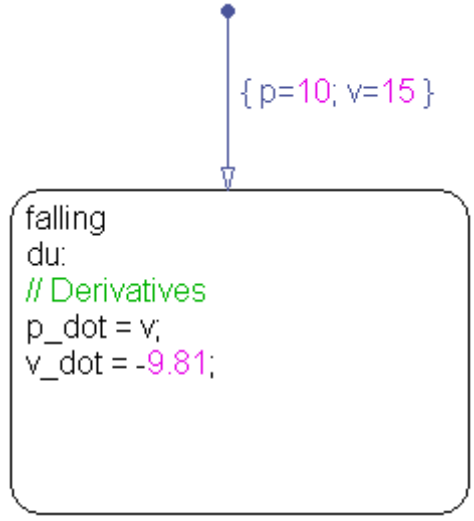
1 Add a state called `falling` with a default transition.



2 In the default transition, set initial position to 10 meters and initial velocity to 15 meters/second. Use a transition action, as follows.



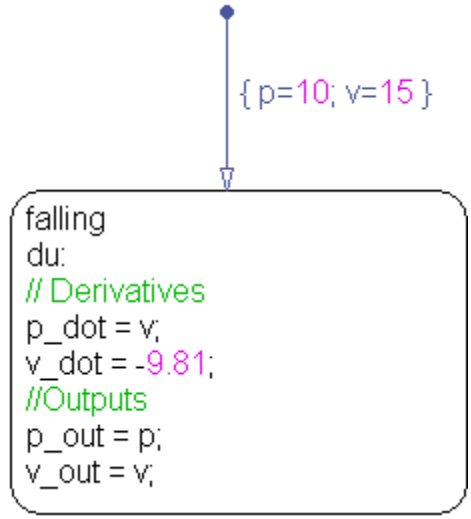
3 Add a `during` action to the `falling` state that defines the derivatives of position and velocity, as follows.



The derivative of position is velocity and the derivative of velocity is acceleration due to gravity (`-g`).

Task 6: Expose Ball Position and Velocity to the Simulink Model

In the `during` action, assign position to the output `p_out` and assign velocity to the output `v_out`, as follows.



Task 7: Validate Semantics of Bouncing Ball Chart

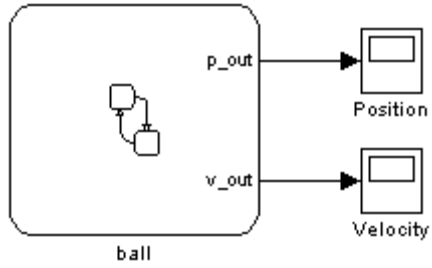
Check semantics against the requirements defined in "Design Considerations for Continuous-Time Modeling in Stateflow Charts" on page 13-27.

This chart meets design requirements, as follows:

- Assigns values to derivatives `p_dot` and `v_dot` in a `during` action
- Writes to local variables `p` and `v` in a transition action
- Initializes local variables on the default transition
- Does not contain events, inner transitions, event-based temporal logic, or change detection operators

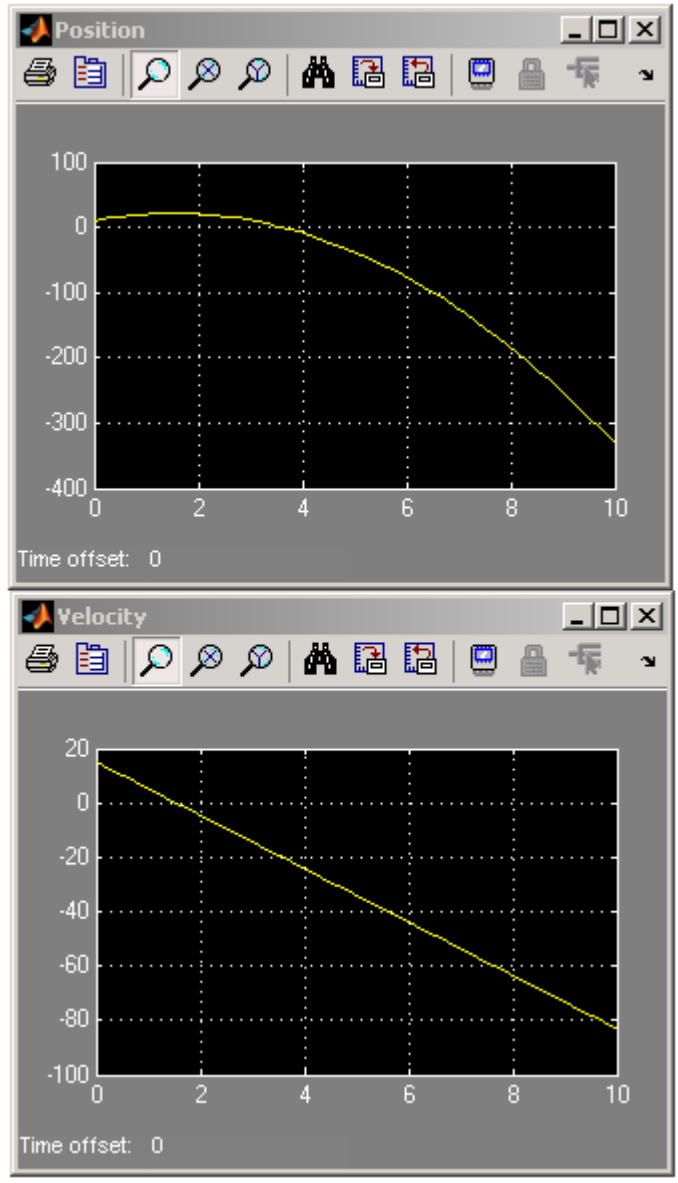
Task 8: Simulate Bouncing Ball Chart

1 Attach a scope to each output.



2 Simulate the chart and view the output in the scopes.

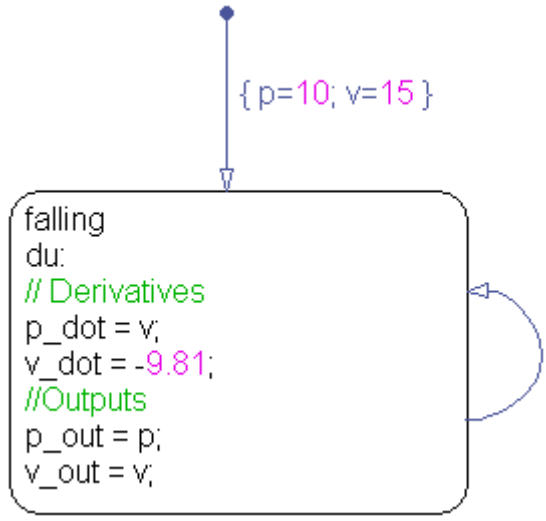
After autoscaling, the scopes show a pattern of a free-falling ball.



Note that the ball appears to fall below the ground (below position p = 0) because the chart does not yet include a check for the bounce.

Task 9: Check for the Bounce

1 Add a self-loop transition to state `falling`.

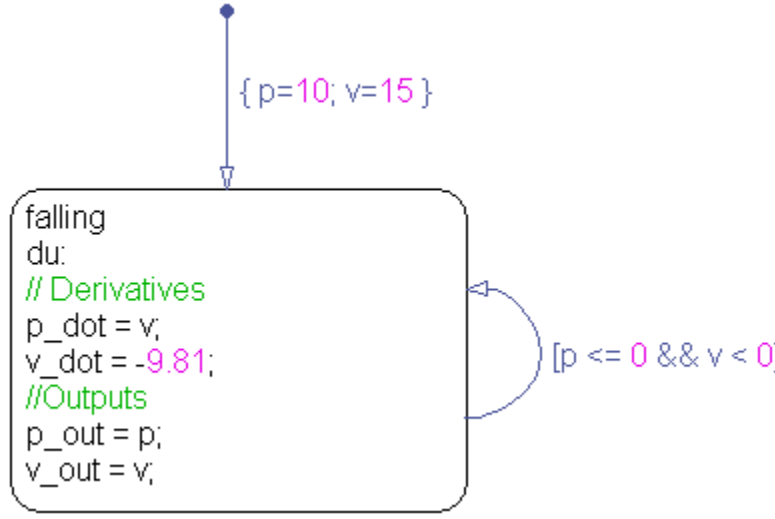


Note The chart uses a self-loop transition so it can model the bounce as an instantaneous mode change — where the ball suddenly reverses direction — rather than as a finite time collision.

---

2 Add a condition on the transition that indicates when the ball hits the ground.

The condition should check for position `p <= 0` and velocity `v < 0`, as follows.



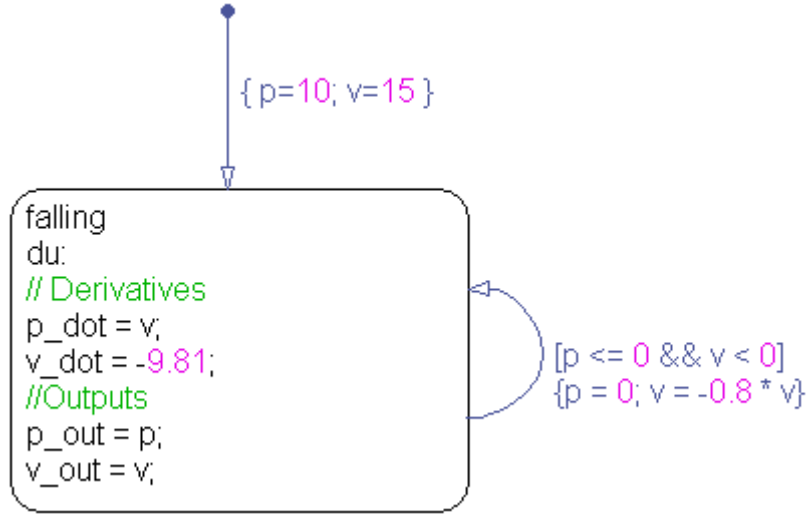
Why not just check for p == 0?

Physically, the ball hits the ground when position `p` is exactly zero. However, by relaxing the condition, you increase the tolerance within which the Simulink model can detect when the continuous variable changes sign (see "How Blocks Work with Zero-Crossing Detection" in the Simulink User's Guide documentation).

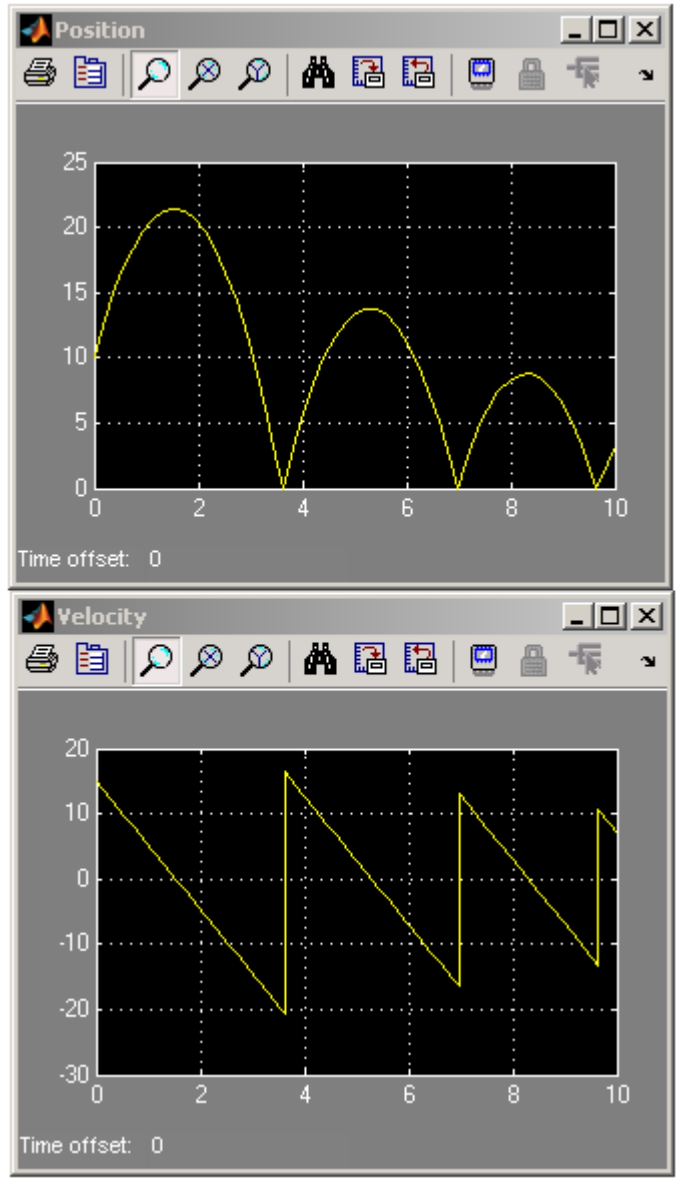
Why add the second check for v < 0?

The second check helps maintain the efficiency of the Simulink solver by minimizing the frequency of zero crossings. Without the second check, the condition becomes true immediately following the state transition, resulting in two successive zero crossings.

3 When the ball hits the ground, reset position and velocity in a condition action, as follows.



4 Simulate the chart again. This time, the scopes illustrate the expected bounce pattern (after autoscaling).


Position
25
20
15
10
5
0
0
2
4
6
8
10
Time offset: 0
Velocity
20
10
0
-10
-20
-30
0
2
4
6
8
10
Time offset: 0

Design Considerations for Continuous-Time Modeling in Stateflow Charts

In this section...

Rationale for Design Considerations

To guarantee the integrity — or *smoothness* — of the results in continuous-time modeling, you must constrain your charts to a restricted subset of Stateflow chart semantics. The restricted semantics ensure that inputs do not depend on unpredictable factors — or *side effects* — such as:

- Simulink solver's guess for number of minor intervals in a major time step
- Number of iterations required to stabilize the integration loop or zero crossings loop

By minimizing side effects, a Stateflow chart can maintain its state at minor time steps and, therefore, update state only during major time steps when mode changes occur. Using this heuristic, a Stateflow chart can always compute outputs based on a constant state for continuous-time.

A Stateflow chart generates informative errors to help you correct semantic violations.

Summary of Rules for Continuous-Time Modeling

Here are the rules for modeling continuous-time Stateflow charts:

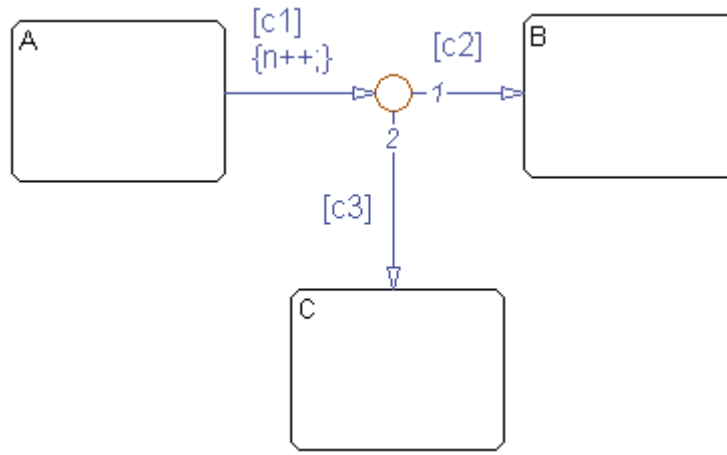
Update local data *only* in transition, `entry`, and `exit` actions

To maintain precision in continuous-time simulation, you should update local data (continuous or discrete) only during physical events at major time steps.

In Stateflow charts, physical events cause state transitions. Therefore, write to local data only in actions that execute during transitions, as follows:

- State `exit` actions, which execute before leaving the state at the beginning of the transition
- Transition actions, which execute during the transition
- State `entry` actions, which execute after entering the new state at the end of the transition
- Condition actions on a transition, but only if the transition directly reaches a state

  Consider this Stateflow chart.



  In this example, the action `{n++}` executes even when conditions `c2` and `c3` are false. In this case, `n` gets updated in a minor time step because there is no state transition.

Do not write to local continuous data in `during` actions because these actions execute in minor time steps.

Do not call Simulink functions in state `during` actions or transition conditions

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 Chapter 21, "Using Simulink Functions in Stateflow Charts".

Compute derivatives only in `during` actions

A Simulink model reads continuous-time derivatives during minor time steps. The only part of a Stateflow chart that executes during minor time steps is the `during` action. Therefore, you should compute derivatives in `during` actions to give your Simulink model the most current calculation.

Do not read outputs and derivatives in states or transitions

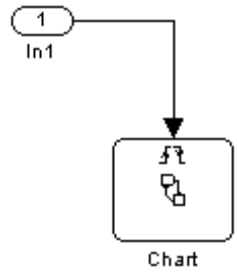
This restriction ensures smooth outputs in a major time step because it prevents a Stateflow chart from using values that may no longer be valid in the current minor time step. Instead, a Stateflow chart always computes outputs from local discrete data, local continuous data, and chart inputs.

Use discrete variables to govern conditions in `during` actions

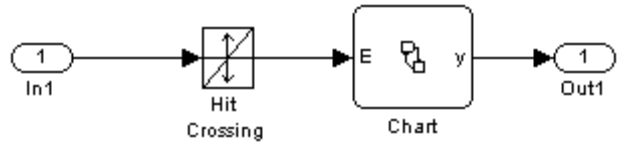
This restriction prevents mode changes from occurring between major time steps. When placed in `during` actions, conditions that affect control flow should be governed by discrete variables because they do not change between major time steps.

Do not use input events in continuous-time Stateflow charts

The presence of input events makes Stateflow charts behave like a triggered subsystem and therefore unable to simulate in continuous-time. For example, the following model generates an error if the Stateflow chart uses a continuous update method.



To model the equivalent of an input event, pass the input signal through a Hit Crossing block as an input to the continuous chart, as in this example.



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. If inner transitions are taken, the `entry` action is never executed.

Limit use of temporal logic

Do not use event-based temporal logic. Use only absolute-time temporal logic for continuous-time simulation. See “Operators for Absolute-Time Temporal Logic” on page 10-63 for details.

Event-based temporal logic has no meaning because there is no concept of a tick during a continuous-time simulation.

The chart must have at least one substate

In continuous-time simulation, the `during` action of a state updates the outputs. A chart with no state produces no output. To simulate the behavior

of a stateless chart in continuous-time, create a single state which calls a graphical function in its `during` action.

Do not use change detection operators in continuous charts

To implement change detection, Stateflow software buffers variables in a way that affects the behavior of charts between a minor time step and the next major time step.

14

Using Fixed-Point Data in Stateflow Charts

What Is Fixed-Point Data?

In this section...

Before You Begin

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.

See "Tips for Using Fixed-Point Data" on page 14-8.

Fixed-Point Numbers

Fixed-point numbers use integers and integer arithmetic to represent real numbers and arithmetic with the following encoding scheme:

$$V \approx \tilde{V} = SQ + B$$

where

- 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 fixed-point representation.
- Q is an integer that encodes $\tilde{V}$. It is referred to as the *quantized integer*.

  Q is the actual stored integer value used in representing the fixed-point number; that is, if a fixed-point number changes, its quantized integer, Q, changes – S and B remain unchanged.
- S is a coefficient of Q referred to as the *slope*.
- B is an additive correction referred to as the *bias*.

Fixed-point numbers encode real quantities (for example, 15.375) using the stored integer Q. You set the value of Q by solving the preceding equation $\tilde{V} = SQ + B$ for Q and rounding the result to an integer value as follows:

$$Q = round((V - B)/S)$$

For example, suppose you want to represent the number 15.375 in a fixed-point type with the slope $S = 0.5$ and the bias $B = 0.1$. This means that

$$Q = round((15.375 — 0.1)/0.5) = 30$$

However, because Q is rounded to an integer, you have lost some precision in representing the number 15.375. If you calculate the number that Q actually represents, you now get a slightly different answer.

$$V = \tilde{V} = SQ + B = 0.5 \cdot 30 + 0.1 = 15.1$$

Using fixed-point numbers to represent real numbers with integers involves the loss of some precision. However, if you choose S and B correctly, you can minimize this loss to acceptable levels.

Fixed-Point Operations

Now that you can express fixed-point numbers as $\tilde{V} = SQ + B$, you can define operations between two fixed-point numbers.

The general equation for an operation between fixed-point operands is as follows:

```
c = a <op> b
```

where `a`, `b`, and `c` are all fixed-point numbers, and `<op>` refers to one of the binary operations: addition, subtraction, multiplication, or division.

The general form for a fixed-point number `x` is $S_xQ_x + B_x$ (see “Fixed-Point Numbers” on page 14-2). Substituting this form for the result and operands in the preceding equation yields this expression:

$$(S_cQ_c + B_c) = (S_aQ_a + B_a) \text{ <op> } (S_bQ_b + B_b)$$

The values for S_c and B_c are chosen by Stateflow software for each operation (see "Promotion Rules for Fixed-Point Operations" on page 14-18) and are based on the values for S_a, S_b, B_a and B_b that you enter for each fixed-point data (see "Specifying Fixed-Point Data" on page 14-6).

Note You can be more precise in choosing the values for S_c and B_c when you use the `:=` assignment operator (that is, `c := a <op> b`). See "Assignment (=, :=) Operations" on page 14-24.

Using the values for S_a, S_b, S_c, B_a, B_b, and B_c, you can solve the preceding equation for Q_c for each binary operation as follows:

- The operation `c=a+b` implies that

$$Q_c = ((S_a/S_c)Q_a + (S_b/S_c)Q_b + (B_a + B_b - B_c)/S_c)$$

- The operation `c=a-b` implies that

$$Q_c = ((S_a/S_c)Q_a - (S_b/S_c)Q_b - (B_a - B_b - B_c)/S_c)$$

- The operation `c=a*b` implies that

$$Q_c = ((S_aS_b/S_c)Q_aQ_b + (B_aS_b/S_c)Q_a + (B_bS_a/S_c)Q_a + (B_aB_b - B_c)/S_c)$$

- The operation `c=a/b` implies that

$$Q_c = ((S_aQ_a + B_a)/(S_c(S_bQ_b + B_b)) - (B_c/S_c))$$

The fixed-point approximations of the real number result of the operation `c = a <op> b` are given by the preceding solutions for the value Q_c. In this way, all fixed-point operations are performed using only the stored integer Q for each fixed-point number and integer operation.

How Fixed-Point Data Works in Stateflow Charts

In this section...

How Stateflow Software Defines Fixed-Point Data

The preceding example in "What Is Fixed-Point Data?" on page 14-2 does not answer the question of how the values for the slope, S, the quantized integer, Q, and the bias, B, are implemented as integers. These values are implemented as follows:

- Stateflow software defines a fixed-point data type from values that you specify.

  You specify values for S, B, and the base integer type for Q. The available base types for Q are the unsigned integer types `uint8`, `uint16`, and `uint32`, and the signed integer types `int8`, `int16`, and `int32`. For specific instructions on how to enter fixed-point data, see "Specifying Fixed-Point Data" on page 14-6.

  Notice that 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.

- Stateflow software implements an integer variable for the Q value of each fixed-point data in generated code.

  This is the only part of a fixed-point number that varies in value. The quantities S and B are constant and appear only as literal numbers or expressions in generated code.

- The slope, S, is factored into an integer power of two, E, and a coefficient, F, such that $S = F*2^E$ and $1 \le F < 2$.

The powers of 2 are implemented as bit shifts, which are more efficient than multiply instructions. Setting $F = 1$ avoids the computationally expensive multiply instructions for values of $F > 1$. This *binary-point-only* scaling is implemented with bit shifts only and is recommended.

- Operations for fixed-point types are implemented with solutions for the quantized integer as described in "Fixed-Point Operations" on page 14-3.

  To generate efficient code, the fixed-point promotion rules choose values for S_c and B_c that conveniently cancel out difficult terms in the solutions. See "Addition (+) and Subtraction (-)" on page 14-22 and "Multiplication (*) and Division (/)" on page 14-22.

  You can use a special assignment operator (`:=`) and context-sensitive constants to maintain as much precision as possible in your fixed-point operations. See "Assignment (=, :=) Operations" on page 14-24 and "Fixed-Point Context-Sensitive Constants" on page 14-7.

- Any remaining numbers, such as the fractional slope, F, that cannot be expressed as a pure integer or a power of 2, are converted into fixed-point numbers.

  These remaining numbers can be computationally expensive in multiplication and division operations. Therefore, using binary-point-only scaling in which $F = 1$ and $B = 0$ is recommended.

- Simulation can detect when the result of a fixed-point operation *overflows* the capacity of its fixed-point type. See "Overflow Detection for Fixed-Point Types" on page 14-10.

Specifying Fixed-Point Data

You can specify fixed-point data in Stateflow charts as follows:

1. From the Stateflow Editor, select **Add > Data**, and then select the scope for the new data object. (See "Scope" on page 9-9 for a description of each type of scope.)

   Doing so adds a default definition of the new data object to the Stateflow hierarchy, and the Data properties dialog box appears.

2. Click the Show data type assistant button  to display the Data Type Assistant.

3 In the **Mode** field of the Data Type Assistant, select `Fixed point`.

4 Specify the fixed-point data properties as described in "Fixed-Point Data Properties" on page 8-13.

5 Specify the name, size, and other properties for the new data object as described in "Setting Data Properties in the Data Dialog Box" on page 8-6.

Note You can also specify a fixed-point constant indirectly in action language by using a fixed-point context-sensitive constant. See "Fixed-Point Context-Sensitive Constants" on page 14-7.

Fixed-Point Context-Sensitive Constants

You can use fixed-point constants without using the Data properties dialog box or Model Explorer, by using context-sensitive constants. These constants infer their types from the context in which they occur. They are written like ordinary constants, but have the suffix `C` or `c`. For example, the numbers `4.3C` and `123.4c` are valid fixed-point context-sensitive constants you can use in action language operations.

These rules apply to context-sensitive constants:

- If any type in the context is a double, then the context-sensitive constant is cast to type double.
- In an addition or subtraction operation, the type of the context-sensitive constant is the type of the other operand.
- In a multiplication or division operation with a fixed-point number, they obtain the best possible precision for a fixed-point result.

  The Simulink Fixed Point function `fixptbestexp` provides this functionality.
- In a cast, the context is the type to which the constant is being cast.
- As an argument in a function call, the context is the type of the formal argument. In an assignment, the context is the type of the left-hand operand.

- You cannot use context-sensitive constants on the left-hand side of an assignment.
- You cannot use context-sensitive constants as both operands of a binary operation.

While you can use fixed-point context-sensitive constants in context with any types (for example, `int32` or `double`), their main use is with fixed-point numbers. The algorithm that computes the type to assign to a fixed-point context-sensitive constant depends on the operator, the types in the context, and the value of the constant. It provides a "natural" type, providing maximum accuracy without overflow.

Tips for Using Fixed-Point Data

When you use fixed-point numbers, follow these guidelines:

1. Develop and test your application 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. You need this while you are building your application.

2. Once your application works well, start substituting fixed-point data for double-precision data during the simulation phase, as follows:

   a. Set the integer word size for the simulation environment to the integer size of the intended target environment.

      Stateflow generated code uses this integer size to select result types for your fixed-point operations. See “Setting the Integer Word Size for a Target” on page 14-19.

   b. Add the suffix ’C’ to literal numeric constants.

      This suffix casts a literal numeric constant in the type of its context. For example, if `x` is fixed-point data, the expression `y = x/3.2C` first converts the numerical constant 3.2 to the fixed-point type of `x` and then performs the division with a fixed-point result. See “Fixed-Point Context-Sensitive Constants” on page 14-7 for more information.

Note If you do not use context-sensitive constants with fixed-point types, noninteger numeric constants (for example, constants that have a decimal point) can force fixed-point operations to produce floating-point results.

3 When you simulate, use overflow detection.

See "Overflow Detection for Fixed-Point Types" on page 14-10 for instructions on how to set overflow detection in simulation.

4 If you encounter overflow errors in fixed-point data, you can do one of the following to add range to your data.

- Increase the number of bits in the overflowing fixed-point data.

  For example, change the base type for *Q* from `int16` to `int32`.

- Increase the range of your fixed-point data by increasing the power of 2 value, *E*.

  For example, you can increase *E* from -2 to -1. This action decreases the available precision in your fixed-point data.

5 If you encounter problems with model behavior stemming from inadequate precision in your fixed-point data, you can do one of the following to add precision to your data:

- Increase the precision of your fixed-point data by decreasing the value of the power of 2 binary point *E*.

  For example, you can decrease *E* from -2 to -3. This action decreases the available range in your fixed-point data.

- If you decrease the value of *E*, you can prevent overflow by increasing the number of bits in the base data type for *Q*.

  For example, you can change the base type for *Q* from `int16` to `int32`.

6 If you cannot avoid overflow for lack of precision, try using the `:=` assignment operator in place of the = operator for assigning the results of multiplication and division operations.

You can use the := operator to increase the range and precision of the result of fixed-point multiplication and division operations at the expense of computational efficiency. See "Assignment Operator :=" on page 14-25.

Overflow Detection for Fixed-Point Types

Overflow occurs when the magnitude of a result assigned to a data exceeds the numeric capacity of that data. You can detect overflow of integer and fixed-point operations during simulation with these steps:

1. In the Stateflow Editor, select **Tools > Open Simulation Target**.

   The **Simulation Target** pane of the Configuration Parameters dialog box appears.

2. Select both the **Enable debugging/animation** and **Enable overflow detection (with debugging)** options.

   For descriptions of these options, see "Speeding Up Simulation" on page 22-17.

3. Click **Execute** to build the simulation target.

4. In the Stateflow Editor toolbar, select Debug  to open the Debugging window.

5. In the Debugging window, select **Data Range**.

   See "Setting Error Checking in the Debugging Window" on page 23-10 for a description of this option.

6. In the Debugging window, click **Start** to begin simulating the model.

Simulation stops when an overflow occurs.

Sharing Fixed-Point Data with Simulink Models

To share fixed-point data with Simulink models, use one of these methods:

- Define identically in both Stateflow charts and Simulink models the data that you input from or output to Simulink blocks.

The values that you enter for the **Stored Integer** and **Scaling** fields in the shared data's properties dialog box in a Stateflow chart (see "Specifying Fixed-Point Data" on page 14-6) must match similar fields that you enter for fixed-point data in a Simulink model. See "Fixed-Point "Bang-Bang Control" Example" on page 14-12 for an example of this method of sharing input data from a Simulink model using a Gateway In block.

For some Simulink blocks, you can specify the type of input or output data directly. For example, you can set fixed-point output data directly in the Block Parameters dialog box of the Simulink Constant block when you select **Specify via dialog** for the **Output data type mode** field (under **Show additional parameters**).

- Define the data as **Input** or **Output** in the Data properties dialog box in the Stateflow chart and instruct the sending or receiving block in the Simulink model to inherit its type from the chart data.

  Many blocks allow you to set their data types and scaling 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 Simulink Constant block to inherit its type from the Stateflow **Input to Simulink** port that it supplies by selecting **Inherit via back propagation** for the **Output data type mode** field in its Block Parameters dialog box (under **Show additional parameters**).

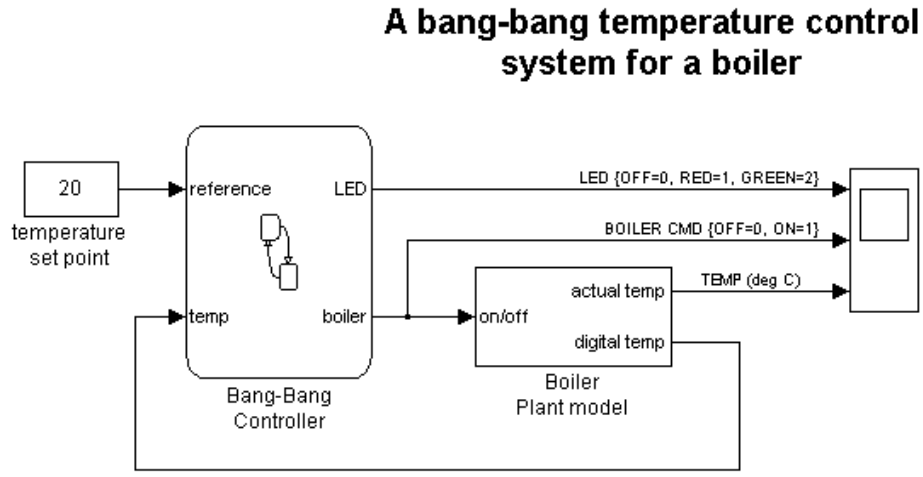
Fixed-Point "Bang-Bang Control" Example

In this section...

Opening the Fixed-Point "Bang-Bang Control" Example

Stateflow software includes demo models with applications of fixed-point data. For this example, load the demo model by typing `sf_boiler` at the MATLAB command prompt.

The model appears as shown.



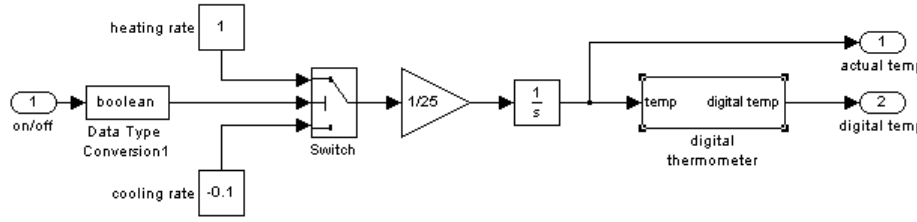
The Stateflow block performs almost all the logic of the bang-bang boiler model, except for the Boiler Plant model subsystem block.

Exploring the Fixed-Point "Bang-Bang Control" Example

To explore the model, follow these steps:

1. In the Simulink model window, double-click the Boiler Plant model subsystem block.

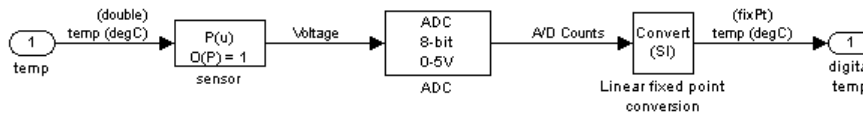
   The subsystem appears.



   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.

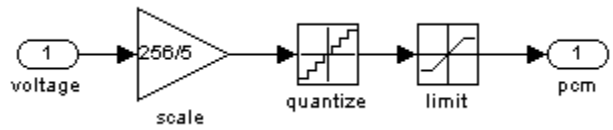
sensor Block

The sensor block converts input boiler temperature *(T)* to an intermediate analog voltage output *V* with a first-order polynomial that gives this output:

$$V = 0.05*T + 0.75$$

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:

$$T_{digital} = (256/5)*V = (256*0.05/5)*T+(256/5)*0.75$$

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:

$$V = S*Q + B$$

Therefore,

$$Q = (V - B)/S = (1/S)*V + (-1/S)*B$$

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:

$$(1/S)*V + (-1/S)*B = (256*0.05/5)*T + (256/5)*0.75$$

Since T is the real-world value for the environment temperature, the above equation implies these relations:

$$V = T$$

and

$$1/S = (256*0.05)/5$$

$$S = 5/(256*0.05) = 0.390625$$

and

$$(-1/S)*B = (256/5)*0.75$$

$$B = -(256/5)*0.75*5/(256*0.05) = -0.75/0.05 = 15$$

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.

Operations with Fixed-Point Data

In this section...

Supported Operations with Fixed-Point Operands

Binary Operations

These binary operations work with fixed-point operands in the following order of precedence (1 = highest, 8 = lowest). For operations with equal precedence, they evaluate in order from left to right:

Example	Precedence	Description
`a * b`	1	Multiplication
`a / b`	1	Division
`a + b`	2	Addition
`a - b`	2	Subtraction
`a > b`	3	Comparison, greater than
`a < b`	3	Comparison, less than
`a >= b`	3	Comparison, greater than or equal to
`a <= b`	3	Comparison, less than or equal to
`a == b`	4	Comparison, equality
`a ~= b`	4	Comparison, inequality
`a != b`	4	Comparison, inequality
`a <> b`	4	Comparison, inequality

Example	Precedence	Description
`a & b`	5	One of the following: • Bitwise AND Enabled when **Enable C-bit operations** is selected in the Chart properties dialog box. See "Specifying Chart Properties" on page 16-5. Operands are cast to integers before the operation is performed. • Logical AND Enabled when **Enable C-bit operations** is cleared in the Chart properties dialog box.
`a | b`	6	One of the following: • Bitwise OR Enabled when **Enable C-bit operations** is selected in the Chart properties dialog box. See "Specifying Chart Properties" on page 16-5. Operands are cast to integers before the operation is performed. • Logical OR Enabled when **Enable C-bit operations** is cleared in the Chart properties dialog box.
`a && b`	7	Logical AND
`a || b`	8	Logical OR

Unary Operations and Actions

These unary operations and actions work with fixed-point operands:

Example	Description
`~a`	Unary minus

Example	Description
`!a`	Logical NOT
`a++`	Increment
`a--`	Decrement

Assignment Operations

These assignment operations work with fixed-point operands:

Example	Description
`a = expression`	Simple assignment
`a := expression`	See "Assignment Operator :=" on page 14-25.
`a += expression`	Equivalent to `a = a + expression`
`a -= expression`	Equivalent to `a = a - expression`
`a *= expression`	Equivalent to `a = a * expression`
`a /= expression`	Equivalent to `a = a / expression`
`a |= expression`	Equivalent to `a = a | expression` (bit operation). See operation `a | b` in "Binary Operations" on page 14-16.
`a &= expression`	Equivalent to `a = a & expression` (bit operation). See operation `a & b` in "Binary Operations" on page 14-16.

Promotion Rules for Fixed-Point Operations

Operations with at least one fixed-point operand require rules for selecting the type of the intermediate result for that operation. For example, in the action statement `c = a + b`, where `a` or `b` is a fixed-point number, an intermediate result type for `a + b` must first be chosen before the result is calculated and assigned to `c`.

The rules for selecting the numeric types used to hold the results of operations with a fixed-point number are called *fixed-point promotion rules*. The goal of these rules is to maintain computational efficiency and usability.

Note You can use the `:=` assignment operator to override the fixed-point promotion rules and obtain greater accuracy. However, in this case, greater accuracy can require more computational steps. See "Assignment Operator :=" on page 14-25.

The following topics describe the process of selecting an intermediate result type for binary operations with at least one fixed-point operand.

Default Selection of the Number of Bits of the Result Type

A fixed-point number with $S = 1$ and $B = 0$ is treated as an integer. In operations with integers, the C language promotes any integer input with fewer bits than the type `int` to the type `int` and then performs the operation.

The type `int` is the *integer word size* for C on a given platform. Result word size is increased to the integer word size because processors can perform operations at this size efficiently.

To maintain consistency with the C language, this default rule applies to assigning the number of bits for the result type of an operation with fixed-point numbers:

When both operands are fixed-point numbers, the number of bits in the result type is the maximum number of bits in the input types or the number of bits in the integer word size for the target machine, whichever is larger.

Note The preceding rule is a default rule for selecting the bit size of the result for operations with fixed-point numbers. This rule is overruled for specific operations as described in the sections that follow.

Setting the Integer Word Size for a Target. The preceding default rule for selecting the bit size of the result for operations with fixed-point numbers relies on the definition of the integer word size for your target. You can set the integer word size for the targets that you build in Simulink models with these steps:

1. Right-click inside the root Simulink model and select **Configuration Parameters**.

   The Configuration Parameters dialog box opens.

2. Select **Hardware Implementation** in the left navigation panel.

   The right panel displays configuration parameters for embedded hardware (simulation and code generation) and emulation hardware (code generation only).

3. To set integer word size for embedded hardware, follow these steps:
   - In the drop-down menu for the **Device type** field, select **Custom**.
   - In the **int** field, enter a word size in bits.

4. To set integer word size for emulation hardware, follow these steps:
   - If no configuration fields appear, clear the **None** check box.
   - In the drop-down menu for the **Device type** field, select **Custom**.
   - In the **int** field, enter a word size in bits.

5. Click **OK** to accept the changes.

When you build any target after making this change, the generated code uses this integer size to select result types for your fixed-point operations.

Note It is recommended that you set all the available sizes because they affect code generation, although they do not affect the implementation of the fixed-point promotion rules in generated code.

Unary Promotions

Only the unary minus (-) operation requires a promotion of its result type. The word size of the result is given by the default procedure for selecting the bit size of the result type for an operation involving fixed-point data. See "Default Selection of the Number of Bits of the Result Type" on page 14-19. The bias, B, of the result type is the negative of the bias of the operand.

Binary Operation Promotion for Integer Operand with Fixed-Point Operand

Integers as operands in binary operations with fixed-point numbers are treated as fixed-point numbers of the same word size with slope, *S*, equal to 1, and a bias, *B*, equal to 0. The operation now becomes a binary operation between two fixed-point operands. See "Binary Operation Promotion for Two Fixed-Point Operands" on page 14-21.

Binary Operation Promotion for Double Operand with Fixed-Point Operand

When one operand is of type `double` in a binary operation with a fixed-point type, the result type is `double`. In this case, the fixed-point operand is cast to type `double`, and the operation is performed.

Binary Operation Promotion for Single Operand with Fixed-Point Operand

When one operand is of type `single` in a binary operation with a fixed-point type, the result type is `single`. In this case, the fixed-point operand is cast to type `single`, and the operation is performed.

Binary Operation Promotion for Two Fixed-Point Operands

Operations with both operands of fixed-point type produce an intermediate result of fixed-point type. The resulting fixed-point type is chosen through the application of a set of operator-specific rules. The procedure for producing an intermediate result type from an operation with operands of different fixed-point types is summarized in these topics:

Addition (+) and Subtraction (-). The output type for addition and subtraction is chosen so that the maximum positive range of either input can be represented in the output while preserving maximum precision. The base word type of the output follows the rule in "Default Selection of the Number of Bits of the Result Type" on page 14-19. 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.

Note Mixing signed and unsigned operands can yield unexpected results and is not recommended.

Multiplication (*) and Division (/). The output type for multiplication and division is chosen to yield the most efficient code implementation. You cannot use nonzero biases for multiplication and division in Stateflow charts (see note).

The slope for the result type of the product of the multiplication of two fixed-point numbers is the product of the slopes of the operands. Similarly, the slope of the result type of the quotient of the division of two fixed-point numbers is the quotient of the slopes. The base word type is chosen to conform to the rule in "Default Selection of the Number of Bits of the Result Type" on page 14-19.

Note Because nonzero biases are computationally very expensive, they are not supported for multiplication and division.

Relational Operations (>, <, >=, <=, ==, -=, !=, <>). You can use the following relational (comparison) operations on all fixed-point types: >, <, >=, <=, ==, -=, !=, <>. See "Supported Operations with Fixed-Point Operands" on page 14-16 for an example and description of these operations. Both operands in a comparison must have equal biases (see note).

Comparing fixed-point values of different types can yield unexpected results because each operand must convert to a common type for comparison. Because of rounding or overflow errors during the conversion, values that do not appear equal might be equal and values that appear to be equal might not be equal.

Note To preserve precision and minimize unexpected results, both operands in a comparison operation must have equal biases.

For example, compare these two unsigned 8-bit fixed-point numbers, `a` and `b`, in an 8-bit target environment:

Fixed-Point Number a	Fixed-Point Number b
$S_a = 2^{-4}$	$S_b = 2^{-2}$
$B_a = 0$	$B_b = 0$
$V_a = 43.8125$	$V_b = 43.75$
$Q_a = 701$	$Q_b = 175$

By rule, the result type for comparison is 8-bit. Converting `b`, the least precise operand, to the type of a, the most precise operand, could result in overflow. Consequently, `a` is converted to the type of `b`. Because the bias values for both operands are 0, the conversion occurs as follows:

$$S_b\ (newQ_a) = S_aQ_a$$

$$newQ_a = (S_aS_b)\ Q_a = (2^{-4}/2^{-2})\ 701 = 701/4 = 175$$

Although they represent different values, `a` and `b` are considered equal as fixed-point numbers.

Logical Operations (&, |, &&, ||). If `a` is a fixed-point number used in a logical operation, it is interpreted with the equivalent substitution `a != 0.0C` where `0.0C` is an expression for zero in the fixed-point type of `a` (see "Fixed-Point Context-Sensitive Constants" on page 14-7). For example, if `a` is a fixed-point number in the logical operation `a && b`, this operation is equivalent to the following:

```
(a != 0.0C) && b
```

The preceding operation is not a check to see whether the quantized integer for a, Q_a, is not 0. If the real-world value for a fixed-point number `a` is 0,

this implies that $V^a = S_a Q_a + B_a = 0.0$. Therefore, the expression `a != 0`, for fixed-point number `a`, is equivalent to this expression:

$$Q_a \; ! = -B_a / S_a$$

For example, if a fixed-point number, `a`, has a slope of 2^{-2}, and a bias of 5, the test `a != 0` is equivalent to the test `if` $Q_a \; ! = -20$.

Assignment (=, :=) Operations

You can use the assignment operations `LHS = RHS` and `LHS := RHS` between a left-hand side (`LHS`) and a right-hand side (`RHS`). See these topics for examples that contrast the two assignment operations:

Assignment Operator =

An assignment statement of the type `LHS = RHS` is equivalent to casting the right-hand side to the type of the left-hand side. You can use any assignment between fixed-point types and therefore, implicitly, any cast.

A cast converts the stored integer Q from its original fixed-point type while preserving its value as accurately as possible using the online conversions (see "Fixed-Point Conversion Operations" on page 14-32). Assignments are most efficient when both types have the same bias, and slopes that are equal or both powers of 2.

Assignment Operator :=

Ordinarily, the fixed-point promotion rules determine the result type for an operation. Using the := assignment operator overrides this behavior by using the type of the `LHS` as the result type of the `RHS` operation.

These rules apply to the `:=` assignment operator:

- The `RHS` can contain at most one binary operator.
- If the `RHS` contains anything other than an addition (+), subtraction (-), multiplication (*), or division (/) operation, or a constant, then the `:=` assignment behaves like regular assignment (=).
- Constants on the `RHS` of an `LHS := RHS` assignment are converted to the type of the left-hand side using offline conversion (see "Fixed-Point Conversion Operations" on page 14-32). Ordinary assignment always casts the `RHS` using online conversions.

When to Use the := Operator Instead of the = Operator

Use the := assignment operator instead of the = assignment operator in these cases:

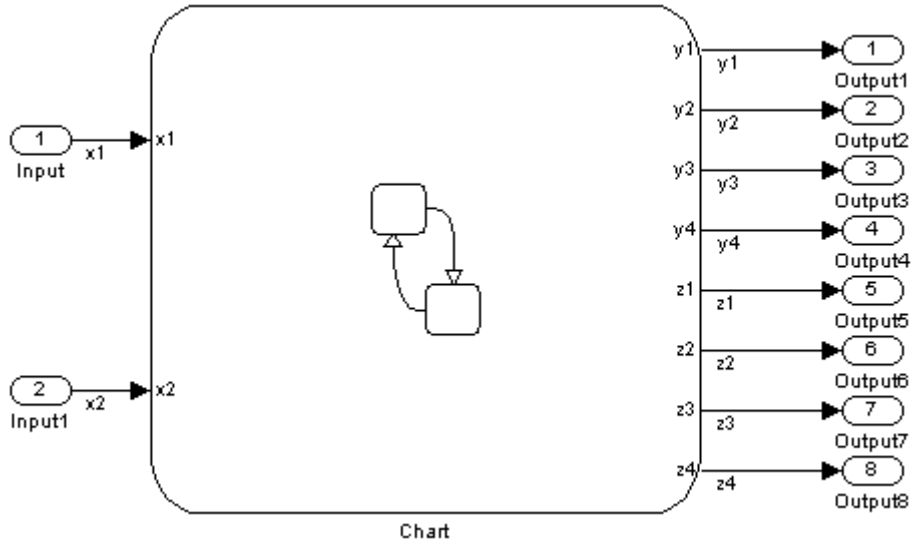
- Arithmetic operations where you want to avoid overflow
- Multiplication and division operations where you want to retain precision

---

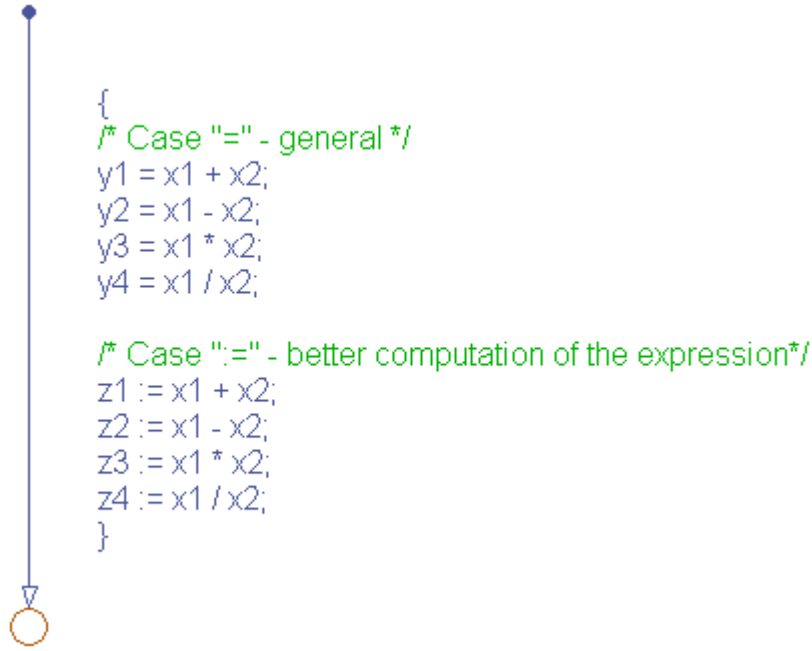
Caution Using the := assignment operator to produce a more accurate result can generate code that is less efficient than the code you generate using the normal fixed-point promotion rules.

Example of Using the := Operator for Addition and Subtraction

This Simulink model contains a Stateflow chart with two inputs and eight outputs.



The chart contains a graphical function that compares the use of the = and := assignment operators.



If you generate code for this model, you see code similar to this.

```
/* Exported block signals */
int16_T x1;                            /* '<Root>/Input' */
int16_T x2;                            /* '<Root>/Input1' */
int32_T y1;                            /* '<Root>/Chart' */
int32_T y2;                            /* '<Root>/Chart' */
int32_T z1;                            /* '<Root>/Chart' */
int32_T z2;                            /* '<Root>/Chart' */
int16_T y3;                            /* '<Root>/Chart' */
int16_T y4;                            /* '<Root>/Chart' */
int16_T z3;                            /* '<Root>/Chart' */
int16_T z4;                            /* '<Root>/Chart' */
```

```
/* Model step function */
void sf_colonequal_operator_step(void)
{
  /*  Case "=" - general  */
  y1 = (int32_T)(x1 + x2);
  y2 = (int32_T)(x1 - x2);
  y3 = x1 * x2 >> 3;
  y4 = div_s16_floor(x1, x2) << 3U;

  /*  Case ":=" - better computation of the expression */
  z1 = (int32_T)x1 + (int32_T)x2;
  z2 = (int32_T)x1 - (int32_T)x2;
  z3 = (int16_T)((int32_T)x1 * (int32_T)x2 >> 3);
  z4 = (int16_T)(((int32_T)x1 << 3U) / (int32_T)x2);
}
```

The inputs `x1` and `x2` are signed 16-bit integers with 3 fraction bits. For addition and subtraction, the outputs are signed 32-bit integers with 3 fraction bits.

You can avoid overflow if you use the := operator instead of the = operator. For example, assume that the inputs have these values:

- `x1` = $2^{15} - 1$
- `x2` = 1

The operator...	Performs addition by...	And produces the result...	Because the sum...
=	Adding the inputs in 16 bits before casting the sum to 32 bits	`y1` = -2^{15}	Overflows
:=	Casting the inputs to 32 bits before adding them	`z1` = $+2^{15}$	Does not overflow

Similarly, you can avoid overflow for subtraction if you use the := operator instead of the = operator.

Example of Using the := Operator for Multiplication

The following example contrasts the := and = assignment operators for multiplication. You can use the := operator to avoid overflow in the multiplication `c = a * b`, where `a` and `b` are two fixed-point operands. The operands and result for this operation are 16-bit unsigned integers with these assignments:

Fixed-Point Number a	Fixed-Point Number b	Fixed-Point Number c
$S_a = 2^{-4}$	$S_b = 2^{-4}$	$S_c = 2^{-5}$
$B_a = 0$	$B_b = 0$	$B_c = 0$
$V_a = 20.1875$	$V_b = 15.3125$	$V_c = ?$
$Q_a = 323$	$Q_b = 245$	$Q_c = ?$

where S is the slope, B is the bias, V is the real-world value, and Q is the quantized integer.

`c = a*b`. In this case, first calculate an intermediate result for `a*b` in the fixed-point type given by the rules in the section "Fixed-Point Operations" on page 14-3, and then cast that result into the type for `c`.

The intermediate value is calculated as follows:

$$Q_{iv} = Q_a Q_b$$

$$= 323 \cdot 245 = 79135$$

Because the maximum value of a 16-bit unsigned integer is $2^{16} - 1 = 65535$, the preceding result overflows its word size. An operation that overflows its type produces an undefined result.

You can capture overflow errors like the preceding example during simulation with the Debugger window. See "Overflow Detection for Fixed-Point Types" on page 14-10.

c := a*b. In this case, calculate `a*b` directly in the type of `c`. Use the solution for Q_c given in "Fixed-Point Operations" on page 14-3 with the requirement of zero bias, which is as follows:

$$Q_c = ((S_a S_b / S_c) Q_a Q_b)$$

$$= (2^{-4} \cdot 2^{-4} / 2^{-5})(323 \cdot 245)$$

$$= 79135/8 = 9892 \text{ (rounded to floor)}$$

No overflow occurs in this case, and the approximate real-world value is as follows:

$$\tilde{V}_c = S_c Q_c = 2^{-5} \cdot 9892 = 9892/32 = 309.125$$

This value is very close to the actual real-world result of 309.121.

Example of Using the := Operator for Division

The following example contrasts the := and = assignment operators for division. You can use the := operator to obtain a more precise result for the division of two fixed-point operands, `a` and `b`, in the statement `c := a/b`.

This example uses the following fixed-point numbers, where S is the slope, B is the bias, V is the real-world value, and Q is the quantized integer:

Fixed-Point Number a	Fixed-Point Number b	Fixed-Point Number c
$S_a = 2^{-4}$	$S_b = 2^{-3}$	$S_c = 2^{-6}$
$B_a = 0$	$B_b = 0$	$B_c = 0$
$V_a = 2$	$V_b = 3$	$V_c = ?$
$Q_a = 32$	$Q_b = 24$	$Q_c = ?$

c = a/b. In this case, first calculate an intermediate result for `a/b` in the fixed-point type given by the rules in the section "Fixed-Point Operations" on page 14-3, and then cast that result into the type for `c`.

The intermediate value is calculated as follows:

$$Q_{iv} = Q_a/Q_b$$

$$= 32/24 = 1$$

The intermediate value is then cast to the result type for `c` as follows:

$$S_bQ_c = S_{iv}Q_{iv}$$

$$Q_c = (S_{iv}/S_c)Q_{iv}$$

The slope of the intermediate value for a division operation is calculated as

$$S_{iv} = S_a/S_b = 2^{-4-3}/2 = 2^{-1}$$

Substitution of this value into the preceding result yields the final result.

$$Q_c = 2^{-1}/2^{-6} = 2^5 = 32$$

In this case, the approximate real-world value is $\tilde{V}c = 32/64 = 0.5$, which is not a very good approximation of the actual result, $2/3 \approx 0.667$.

`c := a/b`. In this case, calculate `a/b` directly in the type of `c`. Use the solution for Q_c given in "Fixed-Point Operations" on page 14-3 with the simplification of zero bias, which is as follows:

$$Q_c = (S_aQ_a) / (S_c(S_bQ_b))$$

$$= (S_a/(S_b \cdot S_c)) \cdot Q_a/Q_b$$

$$= (2^{-4}/(2^{-3} \cdot 2^{-6})) \cdot 32/24$$

$$= 2^5 \cdot 32/24 = 42$$

In this case, the approximate real-world value $\tilde{V}c = 42/64 = 0.6563$, a much better approximation to the precise result, $2/3 = 0.667$.

:= Assignment and Context-Sensitive Constants

In a := assignment operation, the type of the left-hand side (`LHS`) determines part of the context used for inferring the type of a right-hand side (`RHS`) context-sensitive constant.

These rules apply to `RHS` context-sensitive constants in assignments with the := operator:

- If the `LHS` is a floating-point data (type `double` or `single`) , the `RHS` context-sensitive constant becomes a floating-point constant.
- For addition and subtraction, the type of the `LHS` determines the type of the context-sensitive constant on the `RHS`.
- For multiplication and division, the type of the context-sensitive constant is chosen independently of the `LHS`.

Fixed-Point Conversion Operations

Real numbers are converted into fixed-point data during data initialization and as part of casting operations in the application. These conversions compute a quantized integer, *Q*, from a real number input. Offline conversions initialize data, and online conversions perform casting operations in the running application. The topics that follow describe each conversion type and give examples of the results.

Offline Conversions for Initialized Data

Offline conversions are performed during code generation and are designed to maximize accuracy. These conversions round the resulting quantized integer to its nearest integer value. If the conversion overflows, the result saturates the value for *Q*.

Offline conversions are performed for these operations:

- Initialization of data (both variables and constants) in the Stateflow hierarchy
- Initialization of constants or variables from the MATLAB workspace

Online Conversions for Casting Operations

Online conversions are performed for casting operations that take place during execution of the application. Designed to maximize computational efficiency, they are faster and more efficient than offline conversions, but less precise. Instead of rounding Q to its nearest integer, online conversions round to the floor (with the exception of division, which can round to 0, depending on the C compiler you have). If the conversion overflows the type to which you convert, the result is undefined.

Offline and Online Conversion Examples

The following examples show the difference in the results of offline and online conversions of real numbers to a fixed-point type defined by a 16-bit word size, a slope (S) equal to 2^{-4}, and a bias (B) equal to 0:

		Offline Conversion	Online Conversion		
V	***V/S***	***Q***	$\tilde{V}$	***Q***	$\tilde{V}$
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

In the preceding example,

- V is the real-world value represented as a fixed-point value.
- V/S is the floating-point computation for the quantized integer Q.
- Q is the rounded value of V/S.
- $\tilde{V}$ is the approximate real-world value resulting from Q for each conversion.

Autoscaling of Stateflow Fixed-Point Data

The Simulink autoscaling tool autoscales Stateflow fixed-point data. See "Automatic Scaling" in the Simulink Fixed Point software documentation for instructions on autoscaling fixed-point data.

You can prevent Stateflow fixed-point data from being autoscaled by selecting the **Lock output scaling against changes by the autoscaling tool** check box in the Data properties dialog box for fixed-point data. Selecting this option prevents the current fixed-point type from being replaced with a Simulink chosen type in the autoscaling tool. See "Setting Data Properties in the Data Dialog Box" on page 8-6 for a description of the properties for data.

15

Using Complex Data in Stateflow Charts

How Complex Data Works in Stateflow Charts

In this section...

What Is Complex Data?

Complex data is data whose value is a complex number. For example, an input signal with the value `3 + 5i` is complex. See "Complex Signals" in the Simulink documentation for details.

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 Stateflow charts to implement mode logic for frame synchronization.
3. Let the charts access complex input and output data so that nested Embedded MATLAB functions can drive the mode logic.

For an example of modeling a frame synchronization algorithm, see "Implementing a Frame Synchronization Controller Using a Stateflow Chart" on page 15-17.

Note Continuous-time variables of complex type are *not* supported. For more information, see "Defining Continuous-Time Variables" on page 13-12.

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
- Embedded MATLAB functions (see Chapter 20, "Using Embedded MATLAB Functions in Stateflow Charts")
- Truth table functions (see Chapter 19, "Truth Table Functions")
- Graphical functions (see "Using Graphical Functions to Extend Actions" on page 7-27)
- Change detection operators (see "Using Change Detection in Actions" on page 10-74)

Note Exported functions do not support complex data as arguments.

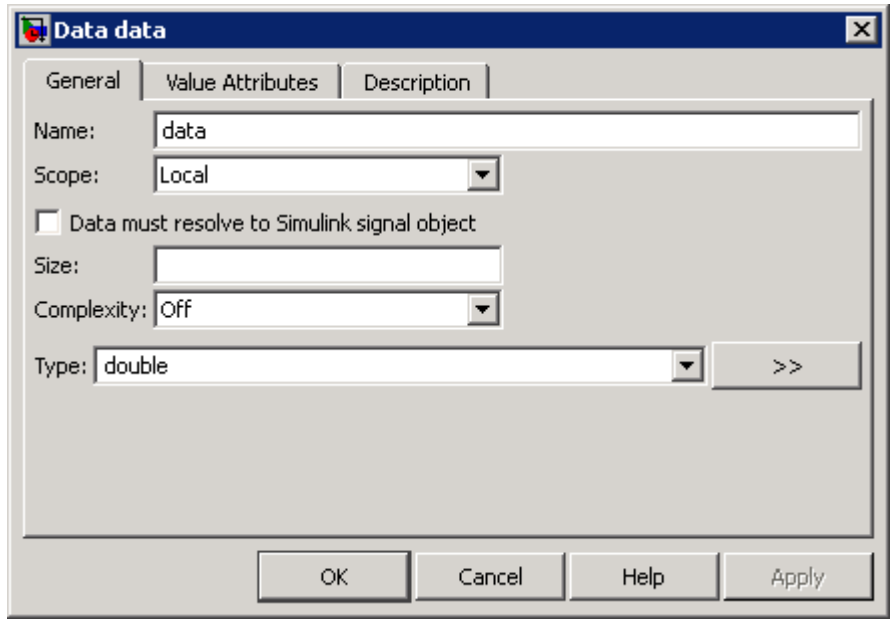
For more information, see "Operations on Complex Data in Stateflow Action Language" on page 15-6 and "Rules for Using Complex Data in Stateflow Charts" on page 15-11.

How to Define Complex Data

Define complex data in a Stateflow chart as follows:

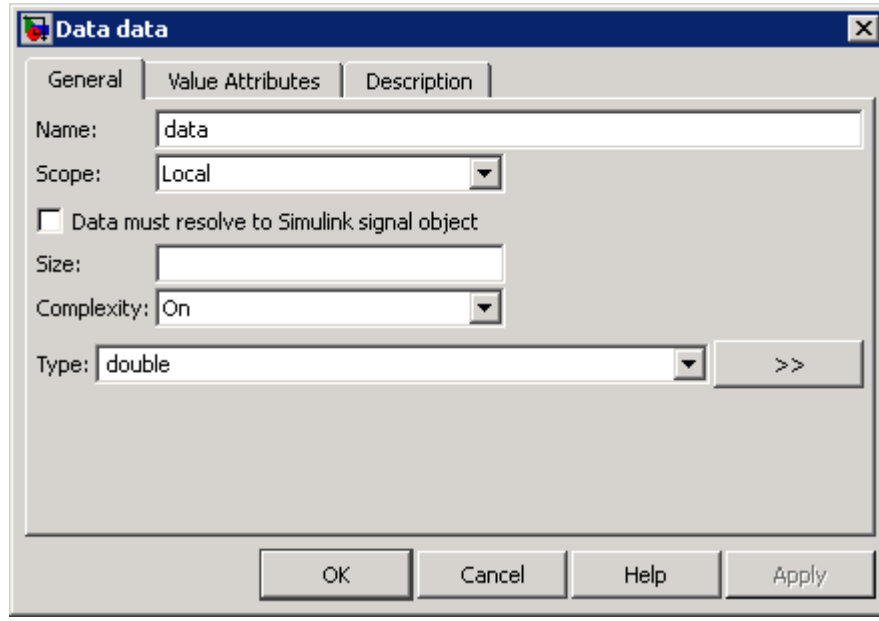
1. In the Stateflow Editor, select **Add > Data**, and then select the scope for the new data object.

   A default definition of the new data object appears in the Stateflow hierarchy, and the Data properties dialog box appears.



Note Complex data does not support the scopes `Constant` and `Data Store Memory`.

2. In the **Complexity** field of the Data properties dialog box, select `On`.



3 Specify the name, size, base type, and other properties for the new data object as described in "Setting Data Properties in the Data Dialog Box" on page 8-6.

Note Complex data does not support the base types `ml`, `struct`, and `boolean`. See "Built-In Data Types" on page 8-46 for more information.

---

4 Click **Apply**.

Operations on Complex Data in Stateflow Action Language

In this section...

Binary Operations

These binary operations work with complex operands in the following order of precedence (1 = highest, 3 = lowest). For operations with equal precedence, they evaluate in order from left to right.

Example	Precedence	Description
`a * b`	1	Multiplication
`a + b`	2	Addition
`a - b`	2	Subtraction
`a == b`	3	Comparison, equality
`a != b`	3	Comparison, inequality

Stateflow action language does not support division of complex operands because this operation requires a numerically stable implementation, especially when the base type of the complex data is fixed-point.

To perform complex division, use an Embedded MATLAB function, which provides a numerically accurate and stable result. For details, see "Performing Complex Division with an Embedded MATLAB Function" on page 15-15.

Unary Operations and Actions

These unary operations and actions work with complex operands.

Example	Description
`~a`	Unary minus
`!a`	Logical NOT
`a++`	Increment
`a--`	Decrement

Assignment Operations

These assignment operations work with complex operands.

Example	Description
`a = expression`	Simple assignment
`a += expression`	Equivalent to `a = a + expression`
`a -= expression`	Equivalent to `a = a - expression`
`a *= expression`	Equivalent to `a = a * expression`

Using Operators to Handle Complex Numbers

In this section...

Why Use Operators for Complex Numbers?

Use operators to handle complex numbers because Stateflow action language does not support complex number notation `(a + bi)`, where `a` and `b` are real numbers.

Defining a Complex Number

To define a complex number based on two real values, use the `complex` operator described below.

complex Operator

Syntax.

```
complex(realExp, imagExp)
```

where `realExp` and `imagExp` are arguments that define the real and imaginary parts of a complex number, respectively. The two arguments must be real values or expressions that evaluate to real values, where the numeric types of both arguments are identical.

Description. The `complex` operator returns a complex number based on the input arguments.

Example.

```
complex(3.24*pi, -9.99)
```

This expression returns the complex number `10.1788    9.9900i`.

Accessing Real and Imaginary Parts of a Complex Number

To access the real and imaginary parts of a complex number, use the operators `real` and `imag` described below.

real Operator

Syntax.

```
real(compExp)
```

where `compExp` is an expression that evaluates to a complex number.

Description. The `real` operator returns the value of the real part of a complex number.

Note If the input argument is a purely imaginary number, the `real` operator returns a value of 0.

Example.

```
real(frame(200))
```

If the expression `frame(200)` evaluates to the complex number `8.23 + 4.56i`, the `real` operator returns a value of 8.2300.

imag Operator

Syntax.

```
imag(compExp)
```

where `compExp` is an expression that evaluates to a complex number.

Description. The `imag` operator returns the value of the imaginary part of a complex number.

Note If the input argument is a real number, the `imag` operator returns a value of 0.

Example.

```
imag(frame(200))
```

If the expression `frame(200)` evaluates to the complex number `8.23 + 4.56i`, the `imag` operator returns a value of 4.5600.

Working with Vector Arguments

The operators `complex`, `real`, and `imag` also work with vector arguments.

Example	If the input `x` is...	Then the output `y` is...
`y = real(x)`	An n-dimensional vector of complex values	An n-dimensional vector of real values
`y = imag(x)`	An n-dimensional vector of real values	An n-dimensional vector of zeros
`y = complex(real(x), imag(x))`	An n-dimensional vector of complex or real values	An n-dimensional vector identical to the input argument

Rules for Using Complex Data in Stateflow Charts

These rules apply when you use complex data in Stateflow charts.

Do not use complex number notation in actions

Stateflow action language does 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 Stateflow action language statements.

To define a complex number, use the `complex` operator described in "Using Operators to Handle Complex Numbers" on page 15-8.

Do not perform math function operations on complex data in Stateflow action language

Math operations such as `sin`, `cos`, `min`, `max`, and `abs` do not work with complex data in Stateflow action language. However, you can use Embedded MATLAB functions for these operations.

For more information, see "Performing Math Function Operations with an Embedded MATLAB Function" on page 15-14.

Mix complex and real operands only for addition, subtraction, and multiplication

If you mix operands for any other math operations in Stateflow action language, an error message appears when you try to simulate your model.

To mix complex and real operands for division, you can use an Embedded MATLAB function as described in "Performing Complex Division with an Embedded MATLAB Function" on page 15-15.

Tip Another way to mix operands for division is to use the `complex`, `real`, and `imag` operators in Stateflow action language.

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 "Using Operators to Handle Complex Numbers" on page 15-8.

Do not define complex data with constant or data store memory scope

If you define complex data with `Constant` or `Data Store Memory` scope, an error message 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 message 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 "Properties You Can Set in the Value Attributes Pane" on page 8-20 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 **Value Attributes** pane of 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 data type

If you assign complex values to real data types, an error message appears when you try to simulate your model.

Note You can assign both real and complex values to complex data types.

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.

Tips for Using Complex Data in Stateflow Charts

In this section...

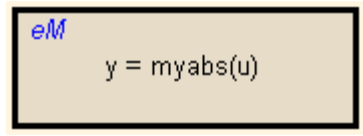
Performing Math Function Operations with an Embedded MATLAB Function

Math functions such as `sin`, `cos`, `min`, `max`, and `abs` do not work with complex data in Stateflow action language. However, you can use an Embedded MATLAB function in your chart to perform math function operations on complex data.

A Simple Example

Suppose that you want to find the absolute value of a complex number. Follow these steps:

1. Add this Embedded MATLAB function to your chart.



2. Double-click the function box to open the Embedded MATLAB Editor.

3. In the Embedded MATLAB Editor, enter the code below.

```
1    function y = myabs(u)
2
3 -  y = abs(u);
4
5
```

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 In the Stateflow Editor, select **View > Model Explorer**.

   b In the **Model Hierarchy** pane of the Model Explorer, navigate to the Embedded 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**.

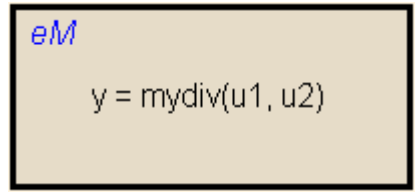
Performing Complex Division with an Embedded MATLAB Function

Division with complex operands is not available as a binary or assignment operation in Stateflow action language. However, you can use an Embedded MATLAB function in your chart to perform division on complex data.

A Simple Example

Suppose that you want to divide two complex numbers. Follow these steps:

1 Add this Embedded MATLAB function to your chart.



2 Double-click the function box to open the Embedded MATLAB Editor.

3 In the Embedded MATLAB Editor, enter the code below.

```
1    function y = mydiv(u1, u2)
2
3 -  y = u1 / u2;
4
```

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 In the Stateflow Editor, select **View > Model Explorer**.

   b In the **Model Hierarchy** pane of the Model Explorer, navigate to the Embedded 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**.

Implementing a Frame Synchronization Controller Using a Stateflow Chart

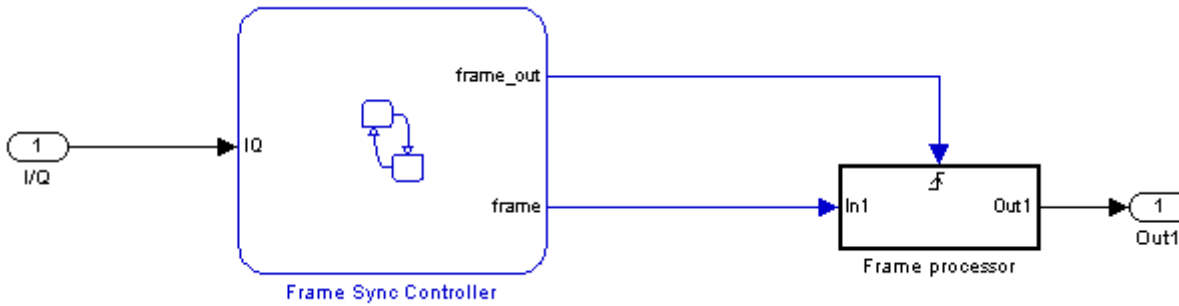
In this section...

What Is Frame Synchronization?

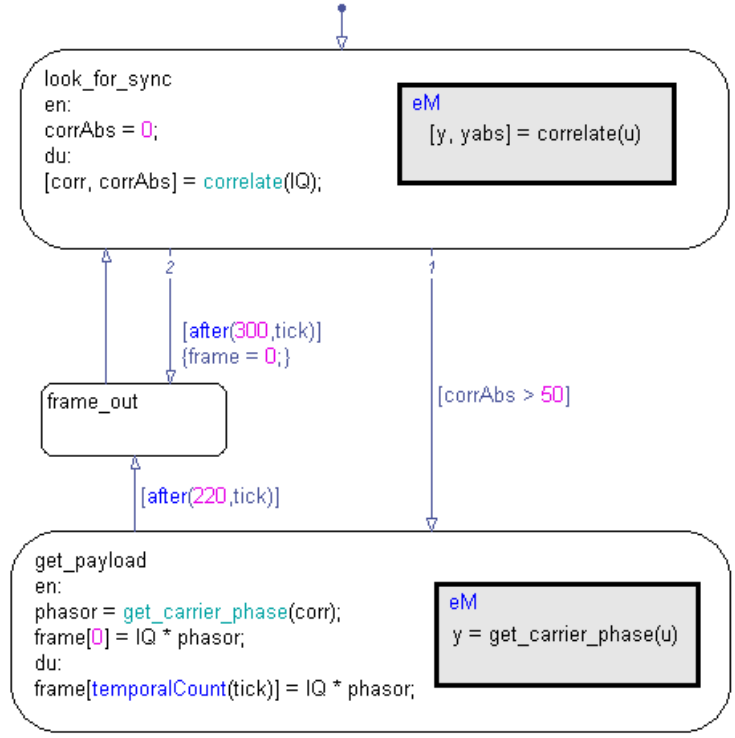
In communication systems, frame synchronization is a method of finding valid data in a transmission that consists of *data frames*. To aid frame synchronization, the transmitter inserts a fixed data pattern at the start of each data frame to mark the start of valid data. The receiver searches for the fixed pattern in each data frame and achieves frame synchronization when the correlation between the input data and the fixed pattern is high.

A Frame Synchronization Controller Chart

This Simulink subsystem is part of a larger model that illustrates the use of Communications Blockset™ software to model a communication system. The chart Frame Sync Controller models a frame synchronization algorithm.



The chart contains these states, transitions, and Embedded MATLAB functions:



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.

For more information, see “How the Chart Works” on page 15-19.

Key Features of the Chart

Key features of the 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`. (See "Using Temporal Logic in State Actions and Transitions" on page 10-56 for information about the `temporalCount` operator.)

- Embedded MATLAB functions with complex arguments

  The Embedded MATLAB functions `correlate` and `get_carrier_phase` have complex input and output arguments.

Opening the Model

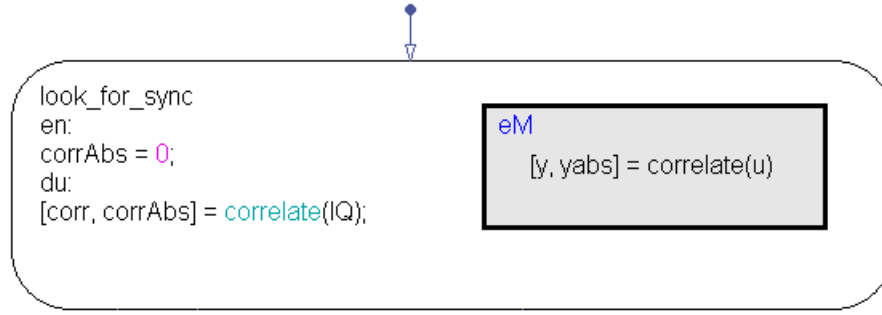
To open the model, type `sf_frame_sync_controller` at the MATLAB command prompt.

Note You cannot simulate this model by itself. This example is available only to illustrate the use of complex data in a Stateflow chart.

How the Chart Works

Stage 1: Activation of the Frame Synchronization Algorithm

When the chart wakes up, the state `look_for_sync` activates to start the frame synchronization algorithm.



Stage 2: Calculation of Correlation Between the Input Signal and the Fixed Pattern

The Embedded 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`.

Code for the function `correlate` appears below.

```
1    function [y, yabs] = correlate(u)
2
3    persistent zi;
4
5    B = conj(flipud(trainSig));
6    A = 1;
7
8    if(isempty(zi))
9        zi = zeros(max(length(A),length(B))-1,1) + complex(0,0);
10   end
11
12   [y,zi] = filter(B,A,u,zi);
13
14   yabs = abs(y);
```

Stage 3: Calculation of Absolute Value of the Complex Correlation

The Embedded 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.

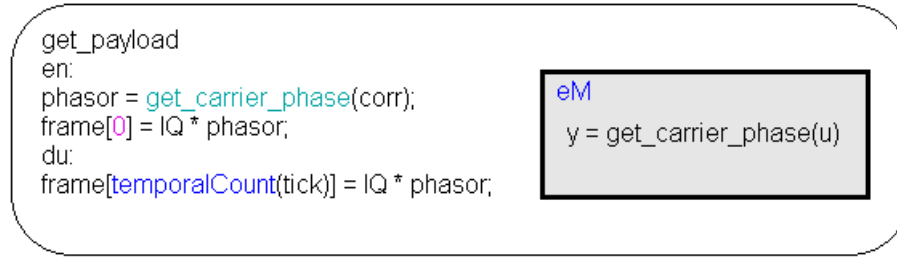
Stage 4: Identification of Valid Data in a Data Frame

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. See "Stage 7: Restart of the Frame Synchronization Algorithm" on page 15-22.

Stage 5: Storage of Valid Data in a Complex Vector

When the correlation is high, the state `get_payload` activates.



The Embedded 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`.

Code for the function `get_carrier_phase` appears below.

```
1    function y = get_carrier_phase(u)
2
3 -  y = exp(-1i*angle(u));
4
5
```

Stage 6: Output of Valid Data from a Data Frame

After collecting 220 data points, the chart outputs the vector `frame` to the next block in the model.

Stage 7: Restart of the Frame Synchronization Algorithm

The state `look_for_sync` reactivates, and the frame synchronization algorithm restarts for the next data frame.

Implementing a Spectrum Analyzer Using a Stateflow Chart

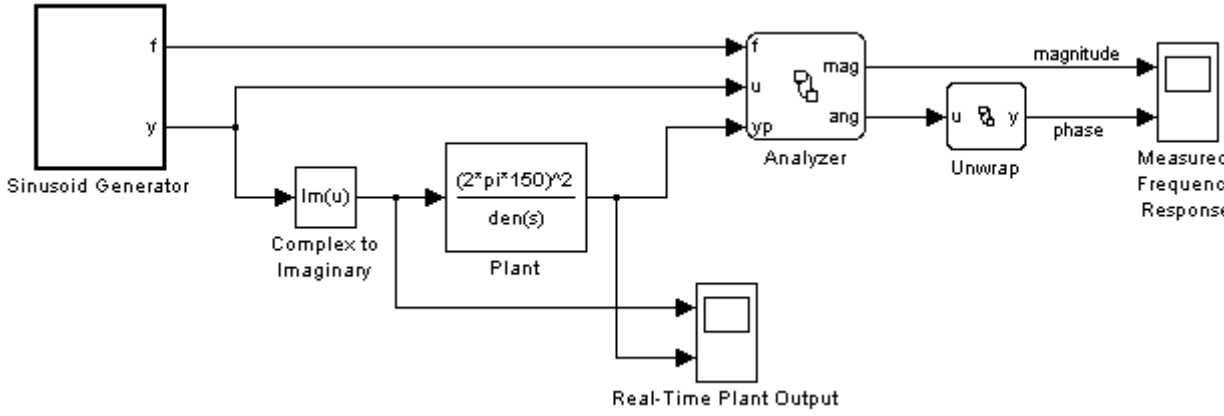
In this section...

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.

A Spectrum Analyzer Model

This Simulink model measures 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.



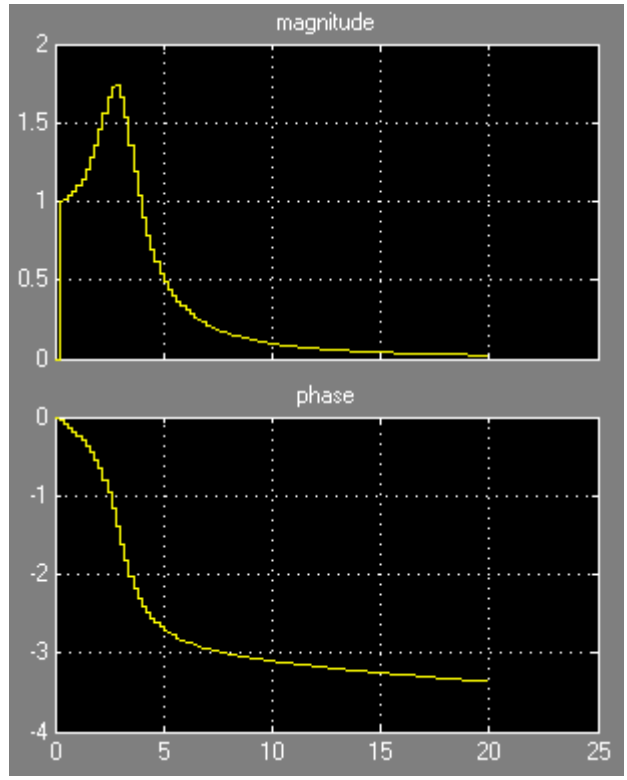
Model Component	Description	Details
Sinusoid Generator block	Generates a complex sinusoidal signal of increasing frequency and supplies this signal to other blocks.	"How the Sinusoid Generator Block Works" on page 15-26
Complex to Imaginary block	Extracts the imaginary part of the complex signal from the Sinusoid Generator block so that a sine wave of increasing frequency can drive the Plant block.	None
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.	None
Analyzer chart	Calculates the frequency response of the second-order system defined by the Plant block.	"How the Analyzer Chart Works" on page 15-28
Unwrap chart	Processes the phase angle output of the Analyzer chart.	"How the Unwrap Chart Works" on page 15-30

Running the Spectrum Analyzer Model

Follow these steps to run the model:

1. Type `sf_spectrum_analyzer` at the MATLAB command prompt.
2. Double-click the Measured Frequency Response scope.
3. Select **Simulation > Start** in the Simulink model window and watch the output in the scope.
4. In the scope display, right-click and select **Autoscale** from the context menu.

   The scope shows a set of discrete Bode plots.



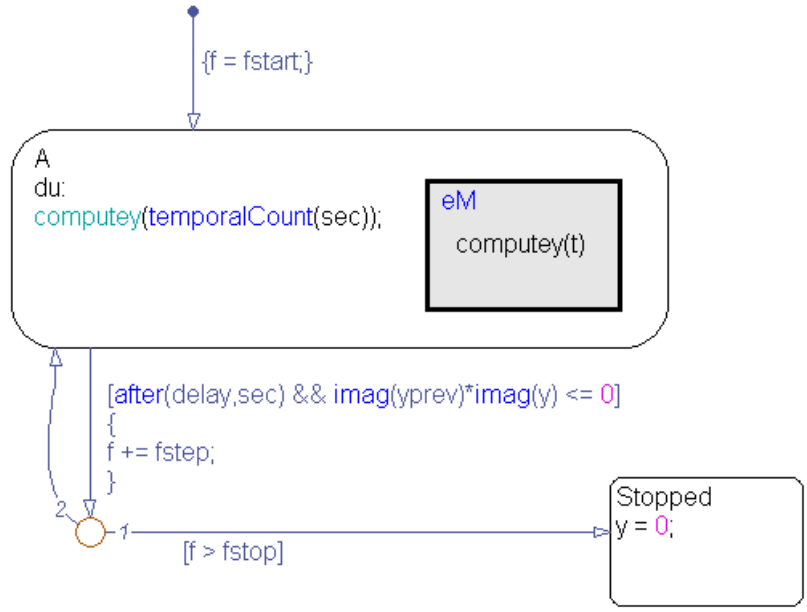
- In the magnitude plot, the sharp peak is the response of the Plant block to a resonant frequency.
- In the phase plot, the angle changes from 0 to -π radians (-180 degrees). Each complex pole in the Plant block adds -π/2 radians to the phase angle.

How the Sinusoid Generator Block Works

This block is a masked subsystem that contains a Stateflow chart. To access the chart, right-click the Sinusoid Generator block and select **Look Under Mask** from the context menu.

Key features of the signal generator chart include:

- Absolute-time temporal logic for controlling changes in frequency (see "Operators for Absolute-Time Temporal Logic" on page 10-63)
- Embedded MATLAB function that generates a complex signal (see Chapter 20, "Using Embedded MATLAB Functions in Stateflow Charts")
- Transition condition that contains complex operands (see "Transition Action Types" on page 10-7)



Stage 1: Definition of Signal Frequency

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.

Stage 2: Generation of Complex Signal

While state `A` is active, the Embedded MATLAB function `computey` generates the complex signal `y` based on frequency `f` and simulation time `t`.

Code for the function appears below.

```
1     function computey(t)
2 -   yprev = y;
3 -   y = exp(2*PI*f*t*1j);
```

Stage 3: Update of Frequency and Complex Signal

If `delay` seconds have elapsed since activation of state `A`, the frequency `f` increases by an amount `fstep` and the Embedded MATLAB function `computey` 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.

Stage 4: Termination of Complex Signal

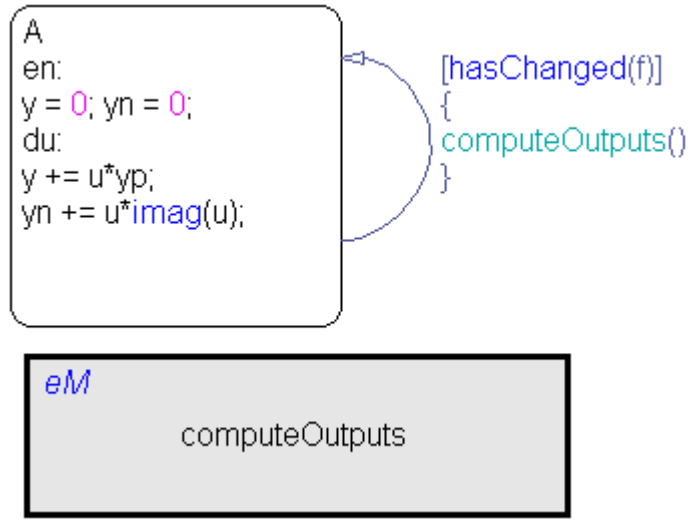
When the frequency `f` reaches the value `fstop`, the state `Stopped` becomes active. The complex signal terminates and the simulation ends.

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 features of the Analyzer chart include:

- Change detection of input frequency (see "Using Change Detection in Actions" on page 10-74)
- Embedded MATLAB function that processes complex data (see Chapter 20, "Using Embedded MATLAB Functions in Stateflow Charts")
- State `during` action that contains complex operands (see "State Action Types" on page 10-2)



Stage 1: Activation of State A

When the chart awakens, 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.

Stage 2: Calculation of Frequency Response

For a given frequency, the Embedded MATLAB function `computeOutputs` finds the magnitude and phase angle of the system response.

Code for the function appears below.

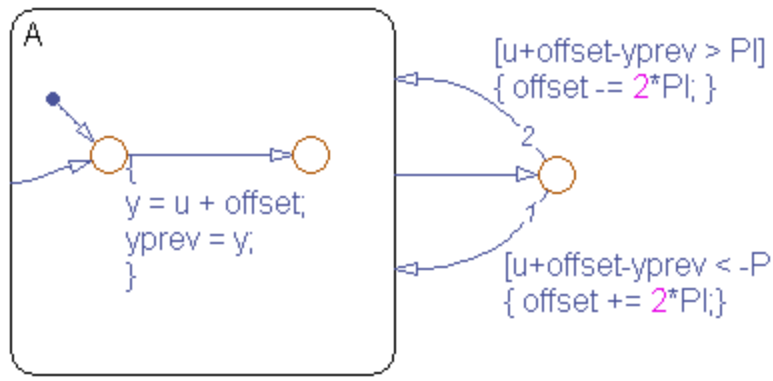
```
1     function computeOutputs
2
3 -   mag = abs(y)/abs(yn);
4 -   ang = -angle(y) + pi/2;
```

Stage 3: Change Detection of Input Frequency

The `hasChanged` operator detects if the input frequency `f` has changed since the previous time step. If so, the Embedded MATLAB function 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.

16

Defining Interfaces to Simulink Models and the MATLAB Workspace

Overview of Stateflow Block Interfaces

In this section...

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.

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
- The properties of a Stateflow chart
- Graphical functions exported from a chart

  See "Exporting Chart-Level Graphical Functions" on page 7-35 for more details.
- The MATLAB workspace

  See "Using MATLAB Functions and Data in Actions" on page 10-33 for more details.
- Definitions in external code sources

Typical Tasks to Define Stateflow Block Interfaces

Defining the interface for a Stateflow block in a Simulink model involves some or all the tasks described in the following topics:

- Specify the update method for a Stateflow block in a Simulink model.

  This task is described in "Setting the Stateflow Block Update Method" on page 16-15.

- Define the input and output data and events that you need.

  See the following topics for detailed information:

  - "Using Input Events to Activate a Stateflow Chart" on page 9-11
  - "Using Output Events to Activate a Simulink Block" on page 9-16
  - "Sharing Input and Output Data with Simulink Models" on page 8-27

- Add and define any nonlocal data and events with which your Stateflow chart must interact.
- Define relationships with any external sources.

  See the topics "MATLAB Workspace Interfaces" on page 16-28 and "Interface to External Sources" on page 16-30.

The preceding task list is a typical sequence. You may find that another sequence better complements your model development.

See "Implementing Update Interfaces to Simulink Models" on page 16-17 for examples of implemented interfaces to Simulink models.

Where to Find More Information on Events and Data

See the following references for defining the interface of a Stateflow Chart block in a Simulink model:

- "Using Input Events to Activate a Stateflow Chart" on page 9-11
- "Using Output Events to Activate a Simulink Block" on page 9-16
- "Importing Events from Stateflow External Code" on page 9-29
- "Exporting Events to Stateflow External Code" on page 9-28

- "Sharing Input and Output Data with Simulink Models" on page 8-27
- "Sharing Stateflow Data with External Modules" on page 8-39

Specifying Chart Properties

In this section...

About Chart Properties

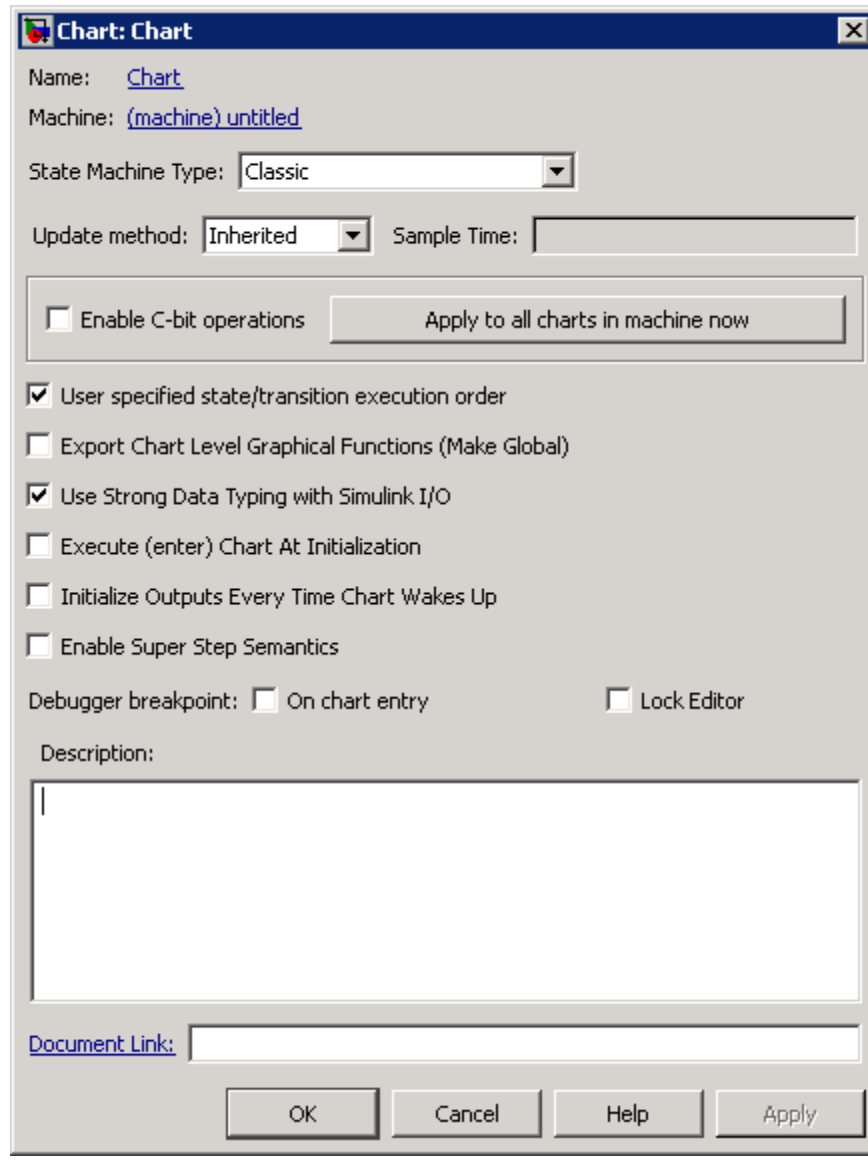
You set part of the interface for a Stateflow block to its Simulink model when you specify the properties for the chart of a Stateflow block. You can specify properties for individual charts or for all charts in a model.

Setting Properties for Individual Charts

To specify properties for an individual Stateflow chart, follow these steps:

1. Double-click on a Stateflow chart to open it in the Stateflow Editor.
2. Right-click an open area of the Stateflow chart.
3. From the context menu, select **Properties**.

   The properties dialog box for the chart appears.



4 Enter properties for the chart based on these descriptions:

Field	Description
Name	Stateflow chart name; read-only; click this hypertext link to bring the chart to the foreground.
Machine	Simulink subsystem name; read-only; click this hypertext link to bring the Simulink subsystem to the foreground.
State Machine Type	Type of state machine to create. Choose from: • **Classic**: Default state machine. Provides full set of Stateflow chart semantics (see Chapter 3, "Stateflow Chart Semantics") . • **Mealy**: State machine in which output is a function of inputs *and* state. • **Moore**: State machine in which output is a function *only* of state. Mealy and Moore charts use a subset of Stateflow chart semantics. For more information, see Chapter 6, "Building Mealy and Moore Charts".
Update method	Method by which a simulation updates (wakes up) a chart in a Simulink model (see "Setting the Stateflow Block Update Method" on page 16-15). Choose from **Inherited**, **Discrete**, or **Continuous**. For more information about continuous updating, see Chapter 13, "Modeling Continuous-Time Systems in Stateflow Charts".
Sample Time	If **Update method** is **Discrete**, enter a sample time.
Enable zero-crossing detection	If **Update method** is **Continuous**, zero-crossing detection is enabled by default. See "When to Enable Zero-Crossing Detection" on page 13-11 in Chapter 13, "Modeling Continuous-Time Systems in Stateflow Charts".

Field	Description
Enable C-bit operations	Select this check box to recognize C bitwise operators (~, &, |, ^, >>, and so on) in action language statements and encode them as C bitwise operations. If you clear this check box, the following occurs: • & and | are interpreted as logical operators. • ^ is interpreted as the power operator (for example, 2^3 = 8). • The remaining expressions (>>, <<, and so on) result in parse errors. To specify this interpretation for all charts in the model (machine), click the **Apply to all charts in machine now** button.
User specified state/transition execution order	Select this check box to use explicit ordering of parallel states and transitions. In this mode, you have complete control of the order in which parallel states are executed and transitions originating from a source are tested for execution. For more information, see "Execution Order for Parallel States" on page 3-39 and "Evaluation Order for Outgoing Transitions" on page 3-21.
Export Chart Level Graphical Functions	Exports graphical functions defined at the chart's root level. See "Exporting Chart-Level Graphical Functions" on page 7-35 for more information.

Field	Description
Use Strong Data Typing with Simulink I/O	If you select this check box, the Chart block for this chart can accept input signals of any data type supported by Simulink software, provided that the type of the input signal matches the type of the corresponding chart input data item (see "Sharing Input and Output Data with Simulink Models" on page 8-27). If the types do not match, a type mismatch error occurs. If this item is cleared, the chart accepts and outputs only signals of type `double`. In this case, Stateflow software converts Simulink input signals to the data types of the corresponding chart input data items. Similarly, Stateflow software converts chart output data (see "Sharing Input and Output Data with Simulink Models" on page 8-27) to type `double` if this option is not selected. For fixed-point data, see the note following this table.
Execute (enter) Chart At Initialization	Select this check box if you want a chart's state configuration to be initialized at time 0 instead of at the first occurrence of an input event (see "Executing a Chart at Initialization" on page 3-16).

Field	Description
Initialize Outputs Every Time Chart Wakes Up	Interprets the initial value of outputs every time a chart wakes up, not only at time 0. When you set an initial value for an output data object, the output will be reset to that value. Outputs are reset whenever a chart is triggered, whether by function call, edge trigger, or clock tick. Enable this option to • Ensure all outputs are defined in every chart execution • Prevent latching of outputs (carrying over values of outputs computed in previous executions) • Give all chart outputs a meaningful initial value
Enable Super Step Semantics	Select to enable Stateflow charts to take multiple transitions in each time step until it reaches a stable state. For more information, see “Executing a Chart with Super Step Semantics” on page 3-6.
Maximum Iterations in Each Super Step	If you enable super step semantics, specify the maximum number of transitions the chart should take in each time step.

Field	Description
Behavior after too many iterations	If you enable super step semantics, specify what the chart should do after it reaches the maximum number of transitions before taking all valid transitions. The options are: • **Proceed** — Chart execution continues to the next time step • **Throw Error** — Simulation stops and an error message appears **Note** The Throw Error option is valid only for simulation. In generated code, chart execution always proceeds.
Debugger breakpoint: On chart entry	Select to set a debugging breakpoint on entry to this chart.
Editor: Locked	Select to mark the Stateflow chart as read-only and prohibit any write operations.
Description	Textual description/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`.

Note For fixed-point data, the **Use Strong Data Typing with Simulink I/O** option is always on. Therefore, if an input or output fixed-point data in a Stateflow chart does not match its counterpart data in a Simulink model, a mismatch error results.

5 Select one of the following buttons:

- **Apply** to save the changes

- **Cancel** to cancel any changes since the last apply
- **OK** to save the changes and close the dialog box
- **Help** to display the online help in an HTML browser window

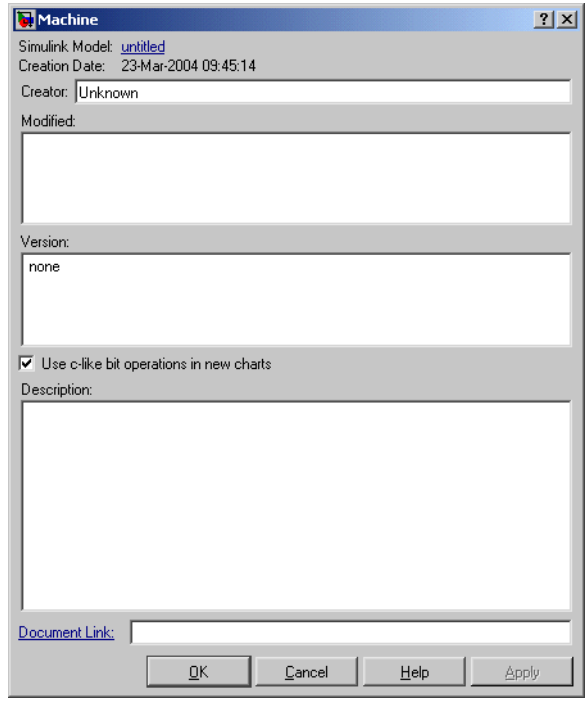
Setting Properties for All Charts in the Model

You can set some properties for all charts in the model by setting properties for the Stateflow machine for a model. The Stateflow machine for a model represents all of the Stateflow blocks in a model.

To set properties for the Stateflow machine, do the following:

1 In the Chart properties dialog box for a particular Stateflow chart, select the **Machine** link at the top of the dialog box.

The Machine properties dialog box appears.



See "Setting Properties for Individual Charts" on page 16-5 for instructions on how to access the Chart properties dialog box for a Stateflow chart.

2 Enter information in the fields provided as described below.

Field	Description
Simulink Model	Name of the Simulink model that defines this Stateflow machine, which is read-only. You change the model name in the Simulink window when you save the model under a chosen file name.
Creation Date	Date on which this machine was created.
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.
Use C-like bit operations in new charts	If you select this check box, all new charts recognize C bitwise operators (~, &, |, ^, >>, and so on) in action language statements and encode these operators as C bitwise operations. You can enable or disable this option for individual charts or all charts in the model in an individual chart's property dialog box. See "Setting Properties for Individual Charts" on page 16-5 for a detailed explanation of this property.
Description	Brief description of this Stateflow machine, which is stored with the model that defines it.
Document Link	MATLAB expression that, when evaluated, displays documentation for this Stateflow machine.

3 Click one of the following:

- **Apply** saves the changes.

- **Cancel** closes the dialog box without making any changes.
- **OK** saves the changes and closes the dialog box.
- **Help** displays the online help in an HTML browser window.

Setting the Stateflow Block Update Method

Stateflow blocks are Simulink subsystems. Simulink events wake up subsystems for execution. To specify a wakeup method for a chart, set the chart's **Update method** property in the Chart dialog box for the chart (see "Specifying Chart Properties" on page 16-5). Choose from the following wakeup methods:

- **Inherited**

  This is the default update method. Specifying this method causes input from the Simulink model to determine when the chart wakes up during a simulation.

  If you define input events for the chart, the Stateflow block is explicitly triggered by a signal on its trigger port originating from a connected Simulink block. This trigger input event can be set in the Model Explorer to occur in response to a Simulink signal that is **Rising**, **Falling**, or **Either** (rising and falling), or in response to a **Function Call**. See "Using Input Events to Activate a Stateflow Chart" on page 9-11.

  If you do not define input events, the Stateflow block implicitly inherits triggers from the Simulink model. These implicit events are the sample times (discrete or continuous) of the Simulink signals providing inputs to the chart. If you define data inputs (see "Sharing Input and Output Data with Simulink Models" on page 8-27), 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 its parent subsystem's execution behavior.

- **Discrete**

  The Simulink model awakens (samples) the Stateflow block at the rate you specify as the block's **Sample Time** property. An implicit event is generated at regular time intervals corresponding to the specified rate. The sample time is in the same units as the Simulink simulation time. Note that other blocks in the Simulink model can have different sample times.

- **Continuous**

  Stateflow charts maintain mode in minor time steps and can define continuous states and their derivatives. In addition, charts can register zero crossings, allowing Simulink models to sample Stateflow charts whenever state changes occur. See Chapter 13, “Modeling Continuous-Time Systems in Stateflow Charts”.

Implementing Update Interfaces to Simulink Models

In this section...

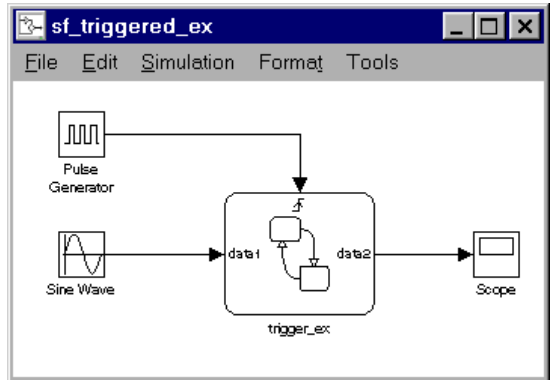
Defining a Triggered Stateflow Block

These are essential conditions that define an edge-triggered Stateflow block:

- The chart **Update method** (set in the Chart properties dialog box) is set to **Triggered** or **Inherited**. (See "Specifying Chart Properties" on page 16-5.)
- The chart has an **Input from Simulink** event defined and an edge-trigger type specified. (See "Using Input Events to Activate a Stateflow Chart" on page 9-11.)

Triggered Stateflow Block Example

A Pulse Generator block connected to the trigger port of the Stateflow block is an example of an edge-triggered Stateflow block.



The **Input from Simulink** event has a **Rising 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.

Defining a Sampled Stateflow Block

There are two ways you can define a sampled Stateflow block. Setting the chart **Update method** (set in the Chart properties dialog box) to **Sampled** and entering a **Sample Time** value define a sampled Stateflow block. (See "Specifying Chart Properties" on page 16-5.)

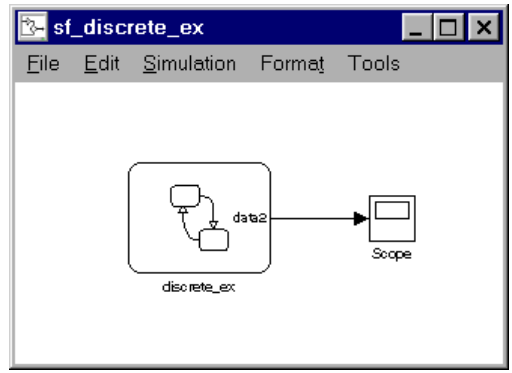
Alternatively, you can add and define an **Input from Simulink** data object. Data is added and defined using either the Stateflow Editor **Add** menu or the Model Explorer. (See "Sharing Input and Output Data with Simulink Models" on page 8-27.) The Simulink model determines the chart sample time to be consistent with the rate of the incoming data signal.

The **Sample Time** (set in the Chart properties dialog box) takes precedence over the sample time of any **Input from Simulink** data.

Sampled Stateflow Block Example

You specify a discrete sample rate to have a Simulink model trigger a Stateflow block that is not explicitly triggered via the trigger port. You can specify a **Sample Time** for the Stateflow chart in the Chart properties dialog

box. The Stateflow block is then called by the Simulink model at the defined, regular sample times.



The outputs of a sampled Stateflow block are held after the execution of the block.

Defining an Inherited Stateflow Block

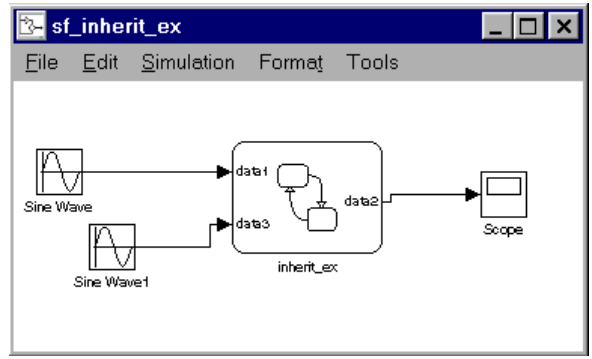
These are essential conditions that define an inherited trigger Stateflow block:

- The chart **Update method** (set in the Chart properties dialog box) is set to **Triggered** or **Inherited**. (See "Specifying Chart Properties" on page 16-5)
- The chart has an **Input from Simulink** data object defined (added and defined using either the Stateflow Editor **Add** menu or the Model Explorer). (See "Sharing Input and Output Data with Simulink Models" on page 8-27.) The Simulink model determines the chart sample time to be consistent with the rate of the incoming data signal.

Inherited Stateflow Block Example

A Simulink model can trigger a Stateflow block that is not explicitly triggered by a trigger port or a specified discrete sample time. In this case, the Simulink model calls the Stateflow block at a sample time determined by the model.

In this example, more than one **Input from Simulink** data object is defined. The Simulink model 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.

Defining a Continuous Stateflow Block

To define a continuous Stateflow block, set the chart **Update method** in the Chart properties dialog box to **Continuous**. See Chapter 13, "Modeling Continuous-Time Systems in Stateflow Charts".

Defining 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. It assumes that you already have in place a programmed function-call subsystem and a Stateflow block in the Simulink model. Use the following steps to connect the Stateflow block to the function-call subsystem and trigger it during simulation.

1. In the Stateflow Editor, select **Add > Event**.

   A context menu of different event scopes appears.

2. From the context menu, select **Output to Simulink**.

   The Event properties dialog box appears with a default name of `event` and a **Scope** of **Output to Simulink**.

3. In the **Trigger** field, select **Function Call**.

4 Name the event appropriately and select **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.

5 From the Simulink Library Browser **Ports & Subsystems** library, place a function-call subsystem in the Simulink model.

You can also create a function-call subsystem by adding a subsystem to the model and adding a Trigger port to the subsystem. In the TriggerPort parameters dialog box for the Trigger block, set the **Trigger type** field to **function-call**.

6 Connect the output port on the Stateflow block for the **Function Call** trigger **Output to Simulink** event you add to the function-call trigger input port of the subsystem.

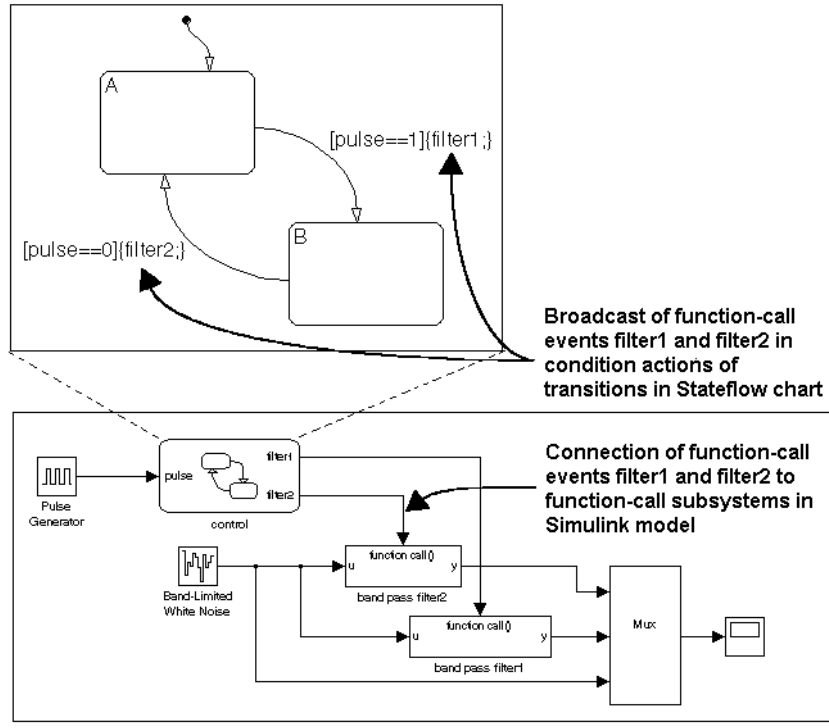
You should avoid placing any other blocks in the connection lines between the Stateflow block and the function-call subsystem for Stateflow blocks that have feedback loops from a block triggered by a function-call event.

Note You cannot connect a function-call output event from a Stateflow chart to a Simulink Demux block in order to trigger multiple subsystems.

---

7 To execute the function-call subsystem, include an event broadcast of the function-call output event in the actions of the Stateflow chart as shown in the following example.

Function-Call Output Events Example



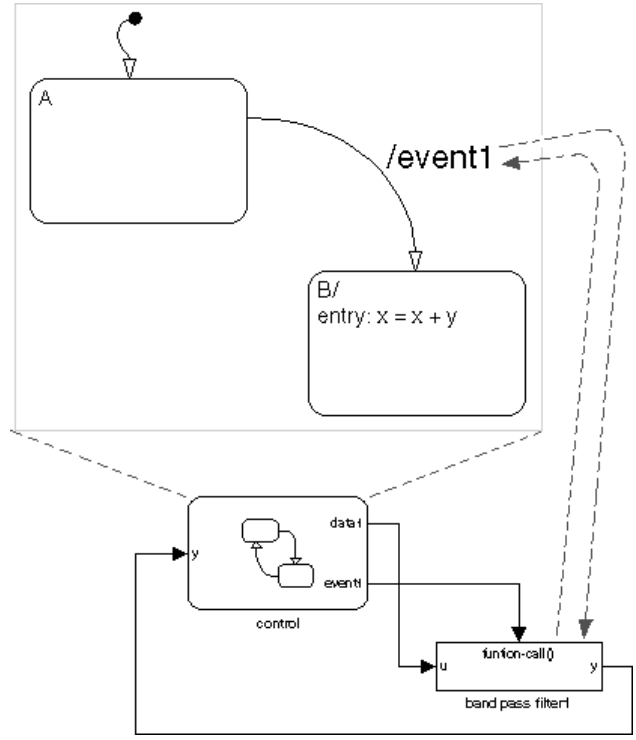
The control Stateflow block has one data input called `pulse` and two function-call output events called `filter1` and `filter2`. A pulse generator provides input data to the control block. Each function-call output event is attached to a subsystem in the Simulink model that is set to trigger by a function call.

Each transition in the control chart has a condition based on the size of the input pulse. When taken, each transition broadcasts a function-call output event that determines whether to make a function call to `filter1` or `filter2`. If the **Output to Simulink** function-call event `filter1` is broadcast, the `band pass filter1` subsystem executes. If the **Output to Simulink** function-call event `filter2` is broadcast, the `band pass filter2` subsystem executes. When either of these subsystems is finished executing, control is returned to the control Stateflow block for the next execution step. In this way, the

Stateflow block controls the execution of `band pass filter1` and `band pass filter2`.

Function-Call Semantics Example

In this example the transition from state `A` to state `B` (in the Stateflow chart) has a transition action that specifies the broadcast of `event1`. `event1` is an **Output to Simulink** event with a **Function Call** trigger type. The Stateflow block output port for `event1` is connected to the trigger port of the `band pass filter1` Simulink block. The `band pass filter1` block has its **Trigger type** field set to **Function Call**.



This sequence occurs when state `A` is active and the transition from state `A` to state `B` is valid and is taken:

1. State `A` exit actions execute and complete.

2. State `A` is marked inactive.

3. The transition action executes and completes.

   In this case the transition action is a broadcast of `event1`. Because `event1` is an event output to the Simulink subsystem with a function-call trigger, the `band pass filter1` block executes and completes, and then returns to the next statement in the execution sequence. The value of `y` is fed back to the Stateflow chart.

4. State `B` is marked active.

5. State `B` entry actions execute and complete (`x = x + y`). The value of `y` is the updated value from the `band pass filter1` block.

6. The Stateflow chart goes back to sleep, waiting to be awakened by another event.

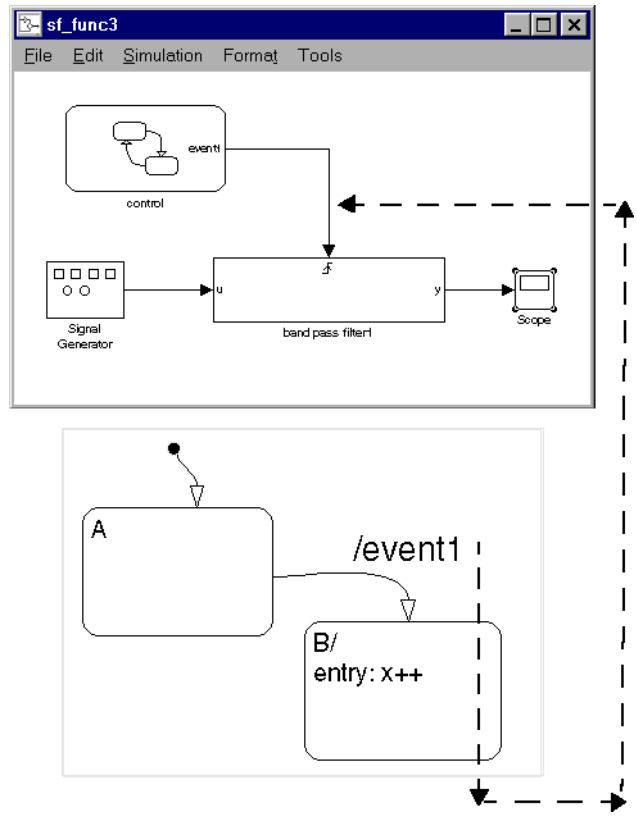
Defining Edge-Triggered Output Events

A Simulink model controls the execution of edge-triggered subsystems with output events. These are essential conditions that define this use of triggered output events:

- The chart has an **Output to Simulink** event with the trigger type **Either Edge**. See "Using Output Events to Activate a Simulink Block" on page 9-16.
- The Simulink block connected to the edge-triggered **Output to Simulink** event has its own trigger type set to the equivalent edge triggering.

Edge-Triggered Semantics Example

In this example, the transition from state `A` to state `B` (in the Stateflow chart) has a transition action that specifies the broadcast of `event1`. `event1` is an **Output to Simulink** event with an **Either edge** trigger type. The Stateflow block output port for `event1` is connected to the trigger port of the `band pass filter1` Simulink block. The `band pass filter1` block has its **Trigger type** field set to **Either edge**.



This sequence occurs when state A is active and the transition from state A to state B is valid and is taken:

1. State A exit actions execute and complete.
2. State A is marked inactive.
3. The transition action, an edge-triggered **Output to Simulink** event broadcast, registers (but does not execute). The Simulink model is controlling the execution, and execution control does not shift until the Stateflow block completes.
4. State B is marked active.

5 State `B` entry actions execute and complete (`x = x++`).

6 The Stateflow chart goes back to sleep, waiting to be awakened by another event.

7 The `band pass filter1` block is triggered, executes, and completes.

Creating Chart Libraries

A Stateflow chart library is a Simulink block library that contains Stateflow Chart blocks (and, optionally, other types of Simulink blocks as well). Just as Simulink libraries serve as repositories of commonly used blocks, chart libraries serve as repositories of commonly used charts.

You create a chart library in the same way you create other types of Simulink libraries. First, create an empty chart library by selecting **File > New > Library** in the Simulink window. Then create or copy Chart blocks into the library just as you would create or copy Chart blocks into a Simulink model.

You use chart libraries in the same way you use other types of Simulink libraries. To include a chart from a library in your model, copy or drag the chart from the library to the model. Simulink software creates a link from the instance in your model to the instance in the library. You can update all instances of the chart simply by updating the library instance.

Note Events parented by a library Stateflow machine are invalid. You can define such events, but they will be flagged as errors when you parse a model.

MATLAB Workspace Interfaces

In this section...

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.

Examining 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` at the MATLAB command line.

See the MATLAB software documentation for more information.

Interfacing the MATLAB Workspace with Stateflow Charts

Stateflow charts have the following access to the MATLAB workspace:

- You can access MATLAB data or MATLAB functions in Stateflow action language with the `ml` namespace operator or the `ml` function.

  See "Using MATLAB Functions and Data in Actions" on page 10-33 for more information.

- You can use the MATLAB workspace to initialize chart data at the beginning of a simulation.

See “Entering Expressions and Parameters for Data Properties” on page 8-24.

- You can save chart data to the workspace at the end of a simulation.

  See “Saving Data to the MATLAB Workspace” on page 8-31 for more information.

Interface to External Sources

In this section...

Introduction

Any source of data, events, or code that is outside a Stateflow chart, its Stateflow machine, or its Simulink model, is considered external to that Stateflow chart. You can interface data and events from external sources to your Stateflow chart. See Chapter 8, "Defining Data" and Chapter 9, "Defining Events" for information on defining data and events.

You can include external source code in the **Simulation Target > Custom Code** pane of the Configuration Parameters dialog box. (For details, see Chapter 22, "Building Targets".)

Exported Data

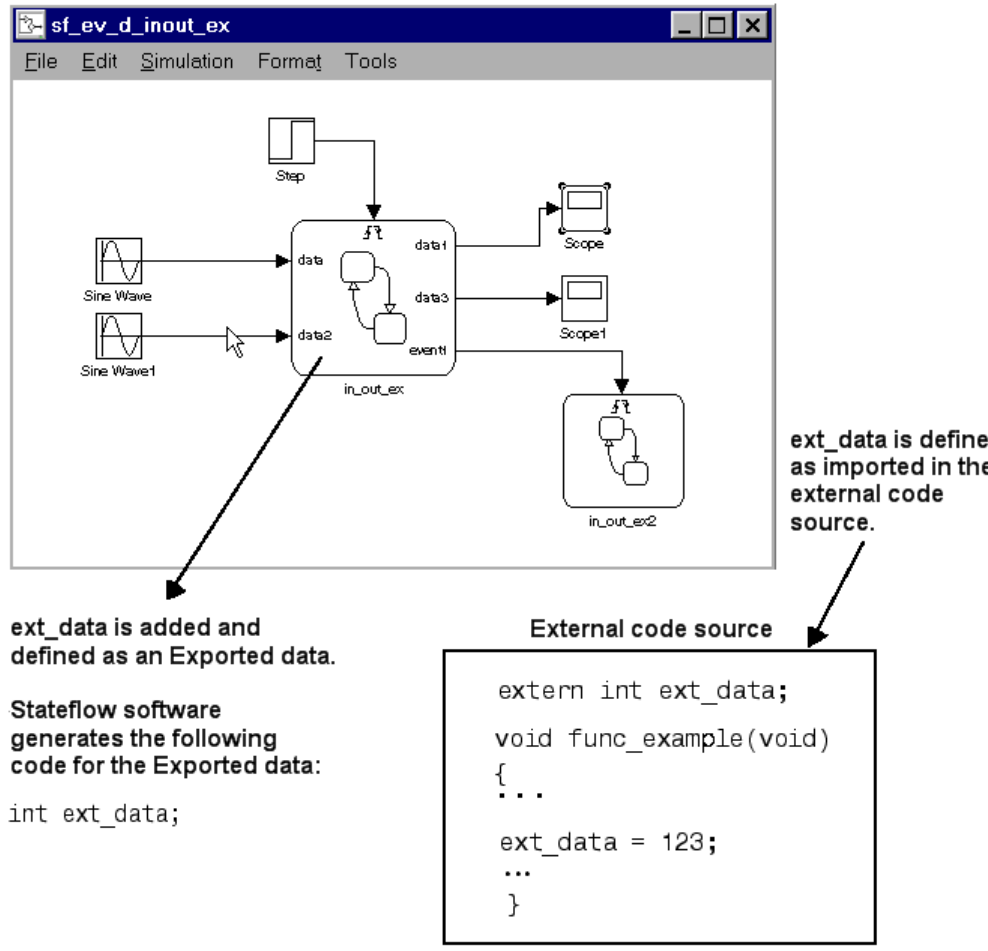
You might want an external source (outside the Stateflow chart, its Stateflow machine, and its Simulink model) to be able to access a data object. By defining a data object's scope as **Exported**, you make it accessible to external sources. Exported data must be parented by the Stateflow machine, because the machine has the highest level in the Stateflow hierarchy and can interface to external sources. The Stateflow machine also retains the ability to access the exported data object. Exporting the data object does not imply anything about what the external source does with the data. It is the responsibility of the external source to include the exported data object (in the manner appropriate to the source) to make use of the right to access the data.

If the external source is another Stateflow machine, then that machine defines an exported data object, and the other machine defines the same data

object as **Imported**. Stateflow software generates the appropriate export and import data code for both machines.

Exported Data Example

The following example shows the format required in the external code source (custom code) to import a Stateflow exported data object:



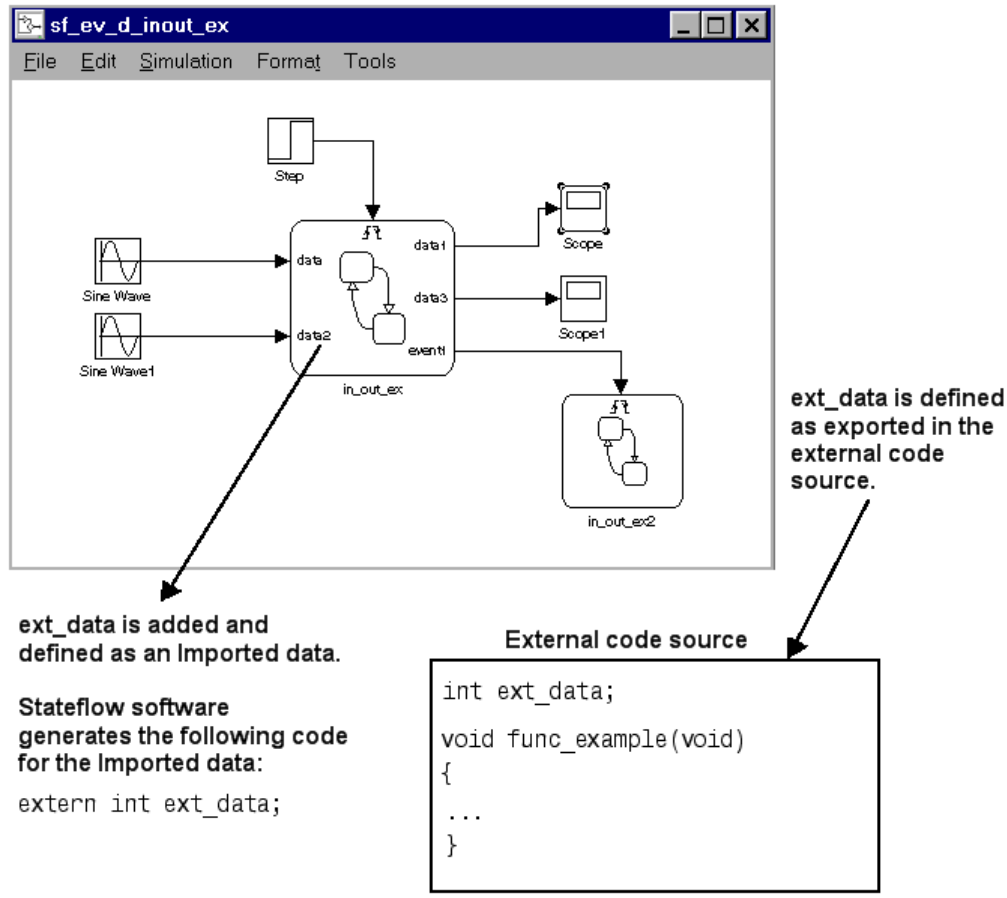
Imported Data

Similarly, you might want to access a data object that is externally defined outside the Stateflow chart, its Stateflow machine, and its Simulink model. If you define the data's scope as **Imported**, the data can be accessed anywhere within the hierarchy of the Stateflow machine (including any offspring of the machine). An imported data object's parent is external. However, the data object needs an adoptive parent to resolve symbols for code generation. An imported data object's adoptive parent must be the Stateflow machine, because the machine has the highest level in the Stateflow hierarchy and can interface to external sources. It is the responsibility of the external source to make the imported data object available (in the manner appropriate to the source).

If the external source for the data is another Stateflow machine, that machine must define the same data object as **Exported**. Stateflow software generates the appropriate import and export data code for both machines.

Imported Data Example

This example shows the format required to retrieve imported data from an external code source (custom code).



Exported Events

You might want an external source (outside the Stateflow chart, its Stateflow machine, and its Simulink model) to be able to broadcast an event. By defining an event's scope to be **Exported**, you make that event available to external sources for broadcast purposes. Exported events must be parented by the Stateflow machine, because the machine has the highest level in the Stateflow hierarchy and can interface to external sources. The Stateflow machine also retains the ability to broadcast the exported event. Exporting the event does not imply anything about what the external source does with the information. It is the responsibility of the external source to include the

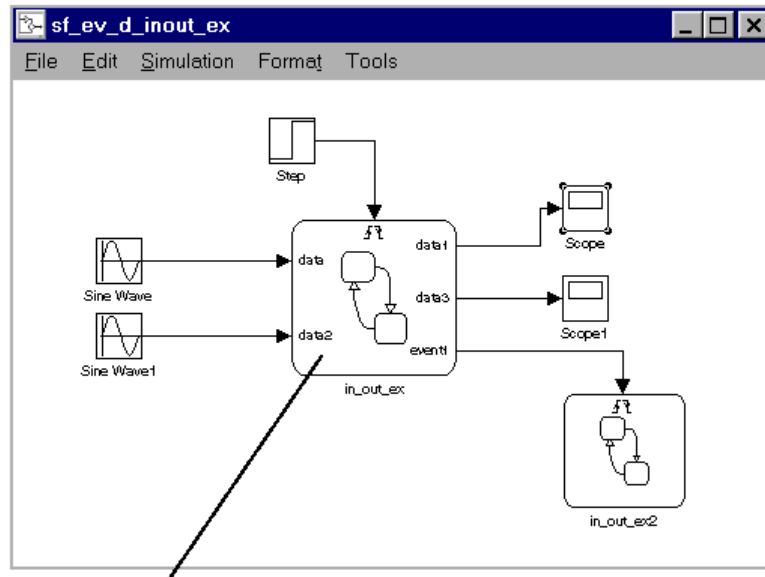
Exported event (in the manner appropriate to the source) to make use of the right to broadcast the event.

If the external source for the event is another Stateflow machine, then that machine must define the event as an **Exported** event, and the other machine must define the same event as **Imported**. Stateflow software generates the appropriate export and import event code for both machines.

Consider a real-world example to clarify when to define an **Exported** event. You have purchased a communications pager. There are a few people you want to be able to page you, so you give those people your personal pager number. These people now know your pager number and can call that number and page you at any time. You do not usually page yourself, but you can do so. Telling someone the pager number does not mean they have heard and recorded the number. It is the other person's responsibility to retain the number.

Exported Event Example

This example shows the format required in the external code source (custom code) to take advantage of an **Exported** event.



e is added and defined as an Exported event.

Stateflow software generates the following code for the Exported event:

```
void broadcast_e (void)
{
/* code based on
   event definition
*/
 } ...
```

e is imported in the external code source.

External code source

```
void func_example(void)
{
extern void broadcast_e (void);
 ...
external_broadcast_e();
 ...
 }
```

Imported Events

You might want to broadcast an event that is defined externally (outside the Stateflow chart, its Stateflow machine, and its Simulink model). By defining an event's scope to be **Imported**, you can broadcast the event anywhere within the hierarchy of that machine (including any offspring of the machine).

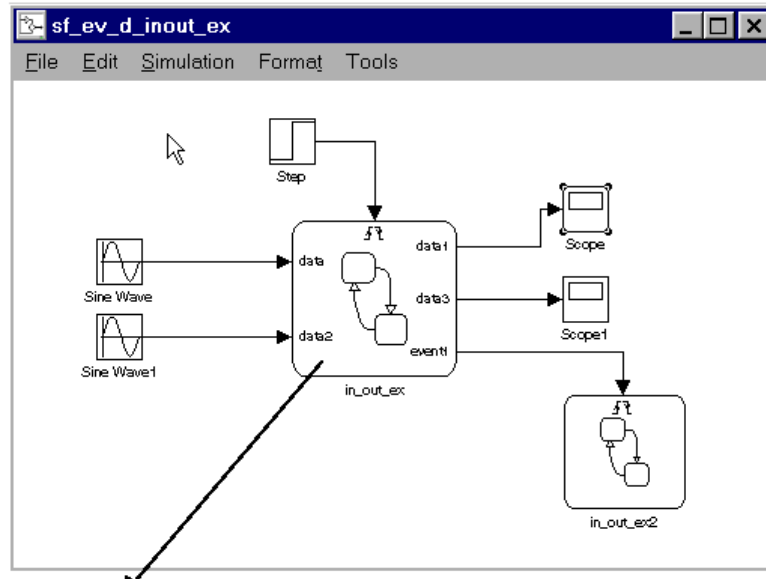
An imported event's parent is external. However, the event needs an adoptive parent to resolve symbols for code generation. An imported event's adoptive parent must be the Stateflow machine, because the machine has the highest level in the Stateflow hierarchy and can interface to external sources. It is the responsibility of the external source to make the imported event available (in the manner appropriate to the source).

If the external source is another Stateflow machine, the source machine must define the same event as **Exported**. Stateflow software generates the appropriate import and export event code for both machines.

The preceding pager example for exported events can clarify the use of imported events. For example, someone buys a pager and tells you that you might want to use this number to page them in the future, and they give you the pager number to record. You can then use that number to page that person.

Imported Event Example

The following example shows the format required in an external code source (custom code) to generate an **Imported** event.



e is exported in the external code source.

e is added and defined as an Imported event.

Stateflow software generates the following code for the Imported event:

```
extern void broadcast_e (void);
```

External code source

```
void broadcast_e (void)
{
...
}
```

17

Working with Structures and Bus Signals in Stateflow Charts

About Stateflow Structures

In this section...

What is a Stateflow Structure?

A Stateflow structure is a data type that you define as a `Simulink.Bus` object. The elements of a Stateflow structure data type are called *fields*. The fields can be any combination of individual signals, muxed signals, vectors, and buses. Each field has its own data type, which need not match that of any other field.

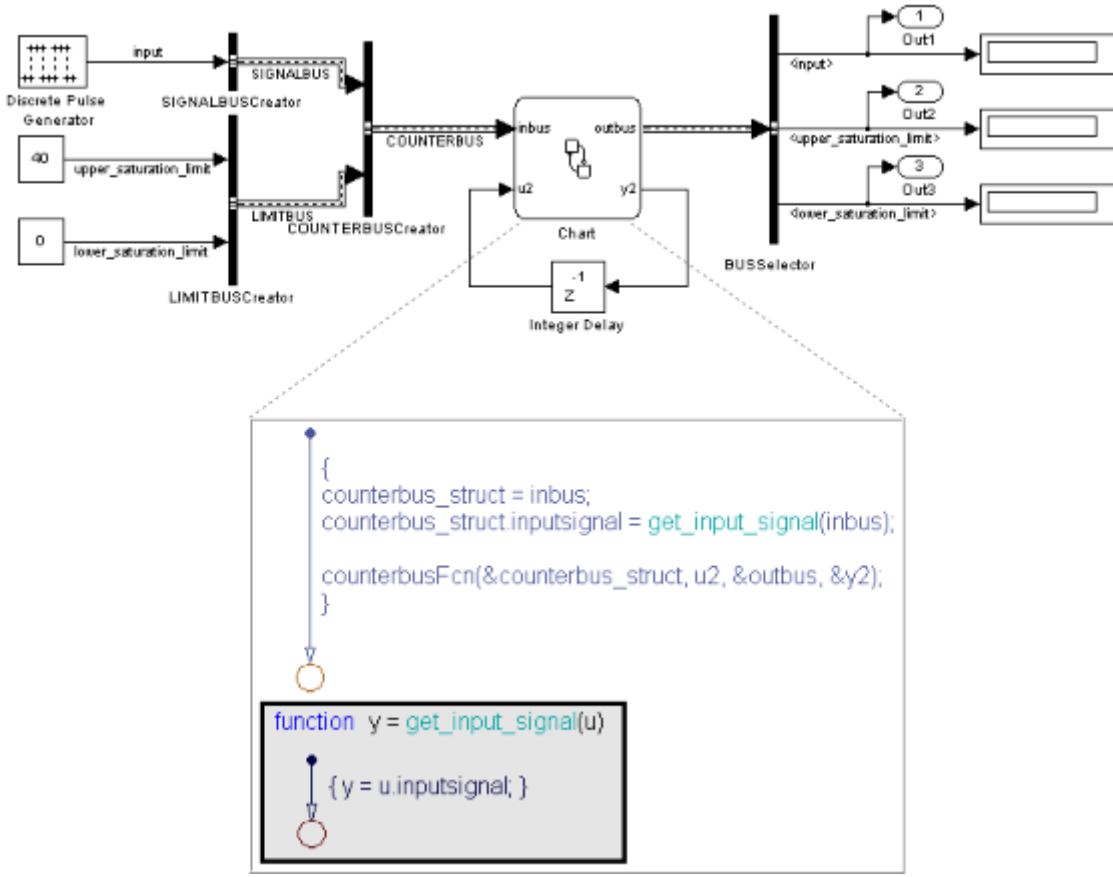
What You Can Do with Structures

With the Stateflow structure data type, you can create

- Inputs and outputs for accessing Simulink bus signals from Stateflow charts, Truth Table blocks, and Embedded MATLAB function blocks (see "Defining Structure Inputs and Outputs" on page 17-7)
- Local structure data in Stateflow charts, truth tables, graphical functions, Embedded MATLAB functions, and boxes (see "Defining Local Structures" on page 17-11)
- Temporary structure data in Stateflow graphical functions, truth tables, and Embedded MATLAB functions (see "Defining Temporary Structures" on page 17-12)

Example of Stateflow Structures

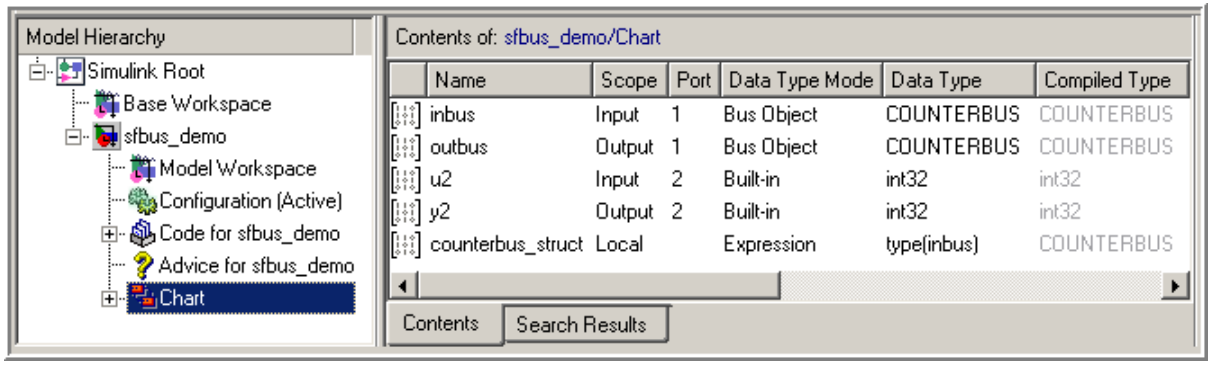
The model sfbus_demo provides examples of structures in a Stateflow chart and graphical function, as follows:



In this model, the Stateflow chart receives a bus input signal using the structure `inbus` at input port 1 and outputs a bus signal from the structure `outbus` at output port 1. The input signal comes from the Simulink Bus Creator block `COUNTERBUSCreator`, which bundles signals from two other Bus Creator blocks: `SIGNALBUSCreator` and `LIMITBUSCreator`. The structure `outbus` connects to a Simulink Bus Selector block `BUSSelector`. The Stateflow chart also contains a local structure `counterbus_struct` and a graphical function `get_input_signal` that contains an input structure `u` and output structure `y`.

Structure Definitions in sfbus_demo Stateflow Chart

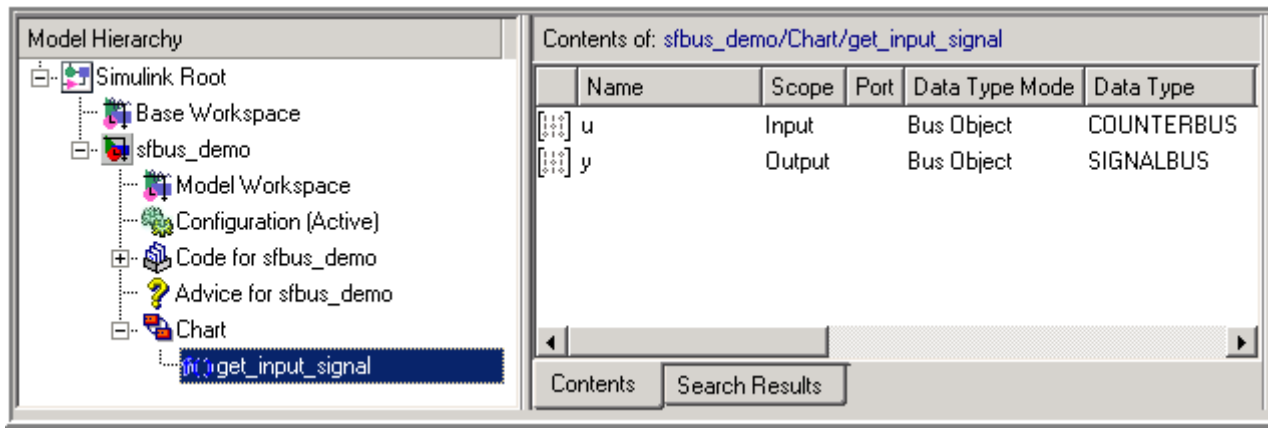
Here are the definitions of the structures in the Stateflow chart of the `sfbus_demo` model, as they appear in the Model Explorer:



Note The local structure `counterbus_struct` is defined using the `type` operator in an expression, as described in "Defining Structure Types with Expressions" on page 17-13.

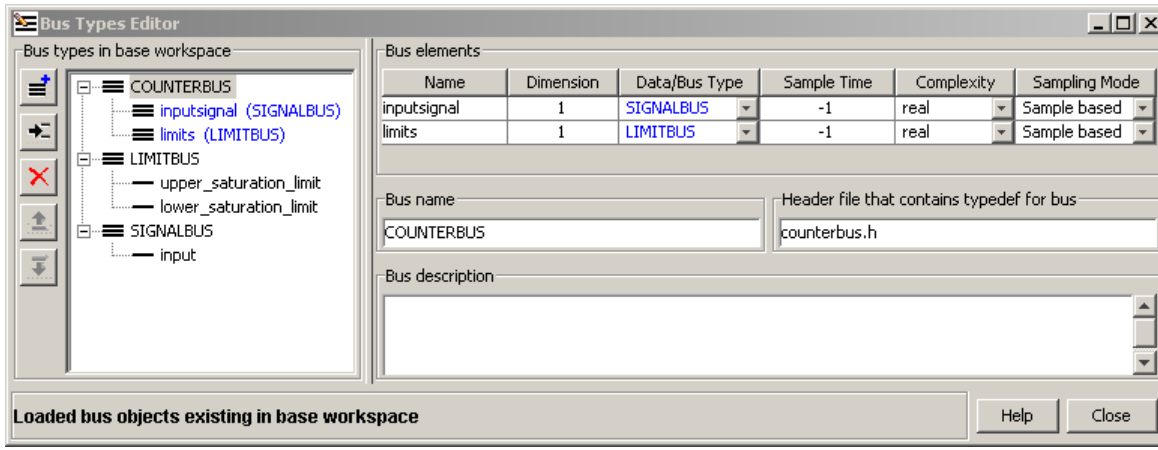
Structure Definitions in sfbus_demo Stateflow Graphical Function

Here are the definitions of the structures in the graphical function `get_input_signal` as they appear in the Model Explorer:



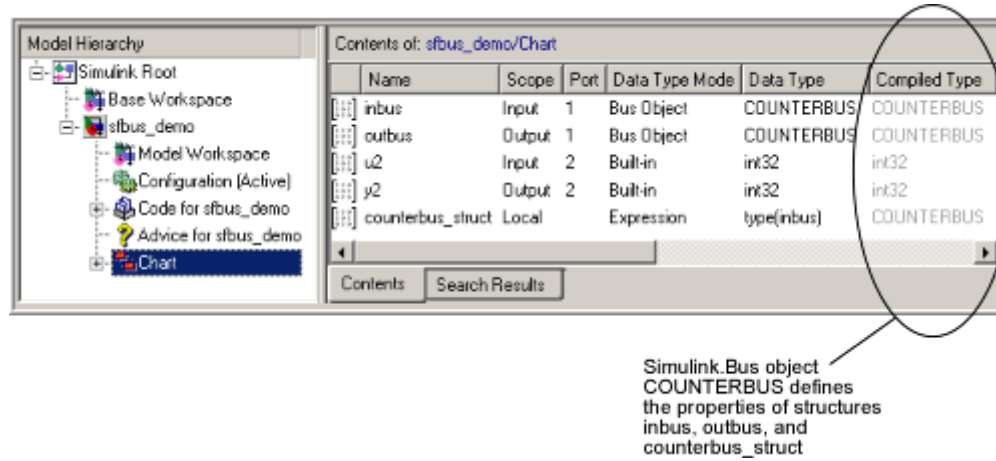
Simulink Bus Objects Define Stateflow Structures

Each Stateflow structure must be defined by a `Simulink.Bus` object in the base workspace. This means that the structure shares the same properties as the bus object, including number, name, and type of fields. For example, the `sfbus_demo` model defines the following bus objects in the base workspace:



You can find the bus object that defines a Stateflow structure by looking in the Data Type and Compiled Type columns in the Contents pane of the Model Explorer. For example, the structures `inbus`, `outbus`, and

`counterbus_struct` are all defined in `sfbus_demo` by the same Simulink bus object, `COUNTERBUS`, as shown in this view of the Model Explorer:



Based on these definitions, `inbus`, `outbus`, and `counterbus_struct` have the same properties as `COUNTERBUS`. For example, these Stateflow structures in `sfbus_demo` reference their fields by the same names as the elements in `COUNTERBUS`, as follows:

Structure	First Field	Second Field
`inbus`	`inbus.inputsignal`	`inbus.limits`
`outbus`	`outbus.inputsignal`	`outbus.limits`
`counterbus_struct`	`counterbus_struct.inputsignal`	`counterbus_struct.limits`

To learn how to define structures in Stateflow charts using `Simulink.Bus` objects, see “Defining Stateflow Structures” on page 17-7.

If you define a custom structure in C for your Stateflow chart, you must make sure that the structure's `typedef` declaration in your header file matches the properties of the `Simulink.Bus` object that defines the structure, as described in “Integrating Custom Structures in Stateflow Charts” on page 17-20.

Defining Stateflow Structures

In this section...

Rules for Defining Structure Data Types in Stateflow Charts

Follow these rules when defining structures in Stateflow charts:

- You must define each structure as a `Simulink.Bus` object in the base workspace.
- You cannot define structures for Stateflow machines.

Note TheStateflow machine is the object that contains all other Stateflow objects in a Simulink model (see "Stateflow Hierarchy of Objects" on page 1-20).

- Stateflow structures cannot have scopes defined as Constant, Parameter, or Data Store Memory.
- Stateflow structures cannot contain arrays of buses.
- Stateflow data array objects cannot contain structures.

Defining Structure Inputs and Outputs

- "Interfacing Stateflow Structures with Simulink Bus Signals" on page 17-8
- "Working with Virtual and Nonvirtual Buses" on page 17-10

Interfacing Stateflow Structures with Simulink Bus Signals

You can drive Stateflow structure inputs by using any Simulink bus signal that has matching properties. Similarly, Stateflow charts can output structures to Simulink blocks that accept bus signals.

To create inputs and outputs in Stateflow charts:

1. Create a Simulink bus object in the base workspace to define the structure type for your Stateflow chart.

   For information about how to create Simulink bus objects, see `Simulink.Bus` in the Simulink Reference documentation.

2. Select **Tools > Explore** in the Stateflow Editor to open the Model Explorer.

3. In the Model Explorer, add a data object as described in "Adding Data Using the Model Explorer" on page 8-3.

   The Model Explorer adds a data object and opens a Properties dialog box in its right-hand Dialog pane.

4. In the **Name** field of the Properties dialog box, enter the name of the structure data.

5. In the **Scope** field, select either `Input` or `Output`.

6. In the **Type** field, select `Inherit: Same as Simulink`, `Bus: <bus object name>`, or `<data type expression>` according to these guidelines:

Type	Works with Scope	Requirements
Inherit: Same as Simulink	**Input**	You do not need to specify a value. The data type is inherited from previously-defined data, based on the scope you selected for the data object. There must be a Simulink bus signal in your model that connects to the Stateflow structure input. The Simulink bus signal must be a nonvirtual bus (see "Working with Virtual and Nonvirtual Buses" on page 17-10). You must specify a `Simulink.Bus` object in the base workspace with the same properties as the bus signal in your model that connects to the Stateflow structure input. The following properties must match: • Number, name, and type of inputs • Dimension • Sample Time • Complexity • Sampling Mode If your input signal comes from a Bus Creator block, you must check the option **Specify properties via bus object** in the Bus Creator properties dialog box. When you enable this option, the Simulink model verifies that the properties of the `Simulink.Bus` object in the base workspace match the properties of the Simulink bus signal.

Type	Works with Scope	Requirements
Bus: <bus object name>	**Input** or **Output**	Replace "*<bus object name>*" in the **Type** field with the name of the `Simulink.Bus` object in the base workspace that defines the Stateflow structure. For example: `Bus: inbus`. **Note** You are not required to specify a bus signal in your Simulink model that connects to the Stateflow structure input or output. However, if you do specify a bus signal, its properties must match the `Simulink.Bus` object that defines the Stateflow structure input or output.
<date type expression>	**Input** or **Output**	Replace "*<data type expression>*" in the **Type** field with an expression that evaluates to a data type.Enter the expression according to these guidelines: • For structure inputs, you can use the Stateflow `type` operator to assign the type of your structure based on the type of another structure defined in the Stateflow chart, as described in "Defining Structure Types with Expressions" on page 17-13. **Note** You cannot use the `type` operator for structure outputs (structures of scope **Output**). • For structure inputs or outputs, you can enter the name of the `Simulink.Bus` object in the base workspace that defines the Stateflow structure.

7 Click **Apply**.

Working with Virtual and Nonvirtual Buses

Simulink models support virtual and nonvirtual buses. Virtual buses read their inputs from noncontiguous memory, while nonvirtual buses read their inputs from data structures stored in contiguous memory (see "Virtual and Nonvirtual Buses" in the Simulink documentation).

Stateflow charts support nonvirtual buses only. When Simulink models contain Stateflow structure inputs and outputs, a hidden converter block converts bus signals for use with Stateflow charts, as follows:

- Converts incoming virtual bus signals to nonvirtual buses for Stateflow structure inputs
- Converts outgoing nonvirtual bus signals from Stateflow charts to virtual bus signals, if necessary

Even though this conversion process allows Stateflow charts to accept virtual and nonvirtual buses as input, Stateflow structures cannot inherit properties from virtual bus input signals. If the input to a chart is a virtual bus, you must set the data type mode of the Stateflow bus input to **Bus Object**, as described in "Interfacing Stateflow Structures with Simulink Bus Signals" on page 17-8.

Defining Local Structures

To define local structures:

1. Create a Simulink bus object in the base workspace to define the structure type for your Stateflow chart.

   For information about how to create Simulink bus objects, see `Simulink.Bus` in the Simulink Reference documentation.

2. Select **Tools > Explore** in the Stateflow Editor to open the Model Explorer.

3. In the Model Explorer, add a data object as described in "Adding Data Using the Model Explorer" on page 8-3.

   The Model Explorer adds a data object and opens a Properties dialog box in its right-hand Dialog pane.

4. In the **Name** field of the Properties dialog box, enter the name of the structure data.

5. In the **Scope** field, select `Local`.

6. In the **Type** field, select either `Bus: <bus object name>`, or `<data type expression>`, and then specify the expression as follows:

Type	What to Specify
Bus: <bus object name>	Replace "*<bus object name>*" in the **Type** field with the name of the `Simulink.Bus` object in the base workspace that defines the Stateflow structure. For example: `Bus: inbus`.
<date type expression>	Replace "*<data type expression>*" in the **Type** field with an expression that evaluates to a data type.You can enter any of the following expressions: • Use the Stateflow `type` operator to assign the type of your structure based on the type of another structure defined in the Stateflow chart, as described in "Defining Structure Types with Expressions" on page 17-13 • Enter the name of the `Simulink.Bus` object in the base workspace that defines the Stateflow structure.

7 Click **Apply**.

Defining Temporary Structures

You can define temporary structures in Stateflow truth tables, graphical functions, and Embedded MATLAB functions.

To define a temporary structure:

1 Create a Simulink bus object in the base workspace to define the structure type for your Stateflow chart.

For information about how to create Simulink bus objects, see `Simulink.Bus` in the Simulink Reference documentation.

2 Select **Tools > Explore** in the Stateflow Editor to open the Model Explorer.

3 In the Model Explorer, add a data object *to your function* as described in "Adding Data Using the Model Explorer" on page 8-3.

The Model Explorer adds a data object and opens a Properties dialog box in its right-hand Dialog pane.

4 In the **Name** field of the Properties dialog box, enter the name of the structure data.

5 In the **Scope** field, select `Temporary`.

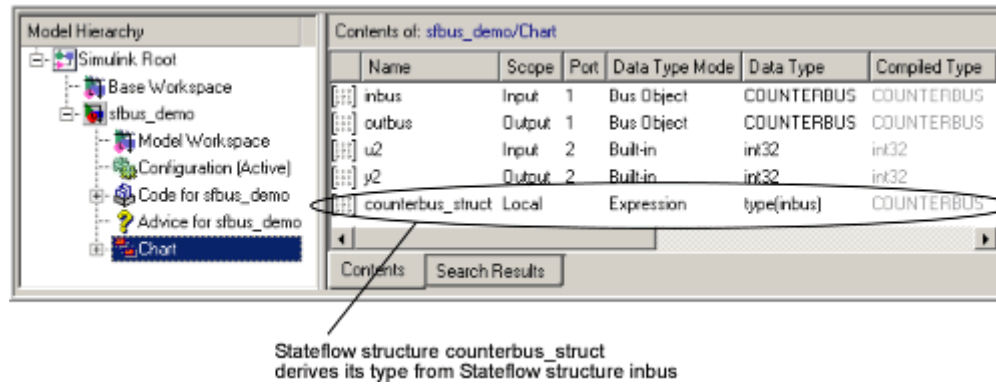
6 In the **Type** field, select either `Bus: <bus object name>`, or `<data type expression>`, and then specify the expression as follows:

Type	What to Specify
Bus: <bus object name>	Replace "*<bus object name>*" in the **Type** field with the name of the `Simulink.Bus` object in the base workspace that defines the Stateflow structure. For example: `Bus: inbus`.
<date type expression>	Replace "*<data type expression>*" in the **Type** field with an expression that evaluates to a data type.You can enter any of the following expressions: • Use the Stateflow `type` operator to assign the type of your structure based on the type of another structure defined in the Stateflow chart, as described in "Defining Structure Types with Expressions" on page 17-13 • Enter the name of the `Simulink.Bus` object in the base workspace that defines the Stateflow structure.

7 Click **Apply**.

Defining Structure Types with Expressions

You can define structure types with expressions that call the Stateflow `type` operator. This operator assigns the type of your structure based on the type of another structure defined in the Stateflow chart. For example, the model sfbus_demo contains a local structure whose type is defined using a `type` operator expression, as follows:



In this case, the structure `counterbus_struct` derives its type from structure `inbus`, which is defined by the `Simulink.Bus` object `COUNTERBUS`. Therefore, the structure `counterbus_struct` is also defined by the bus object `COUNTERBUS`.

To learn how to use the Stateflow `type` operator, see “Deriving Data Types from Previously Defined Data” on page 8-47.

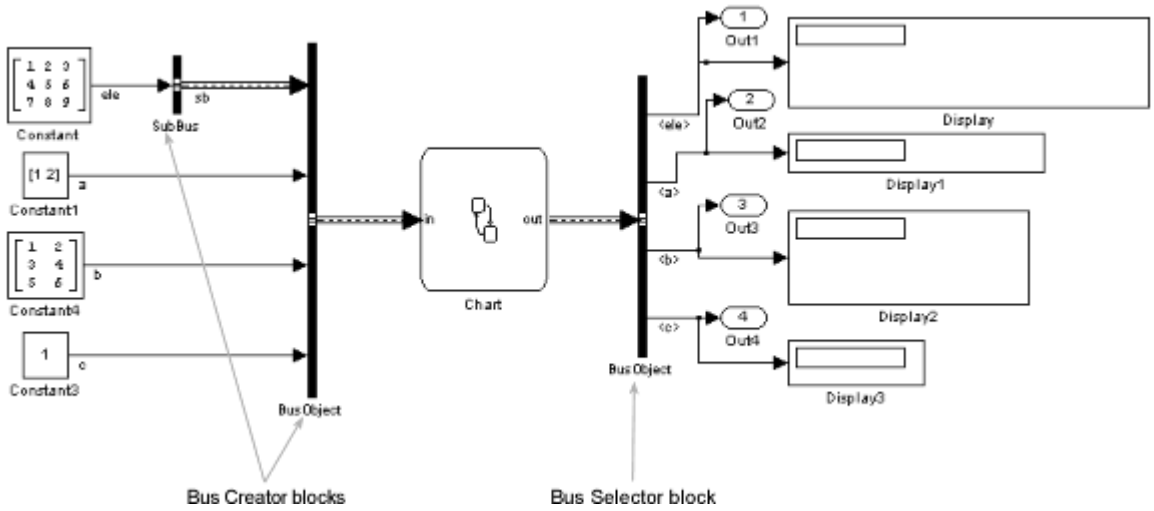
Structure Operations

In this section...

Indexing Sub-Structures and Fields

You index substructures and fields of Stateflow structures by using dot notation. With dot notation, the first text string identifies the parent object, and subsequent text strings 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). By default, the names of the fields of a Stateflow structure match the names of the elements of the `Simulink.Bus` object that defines the structure.

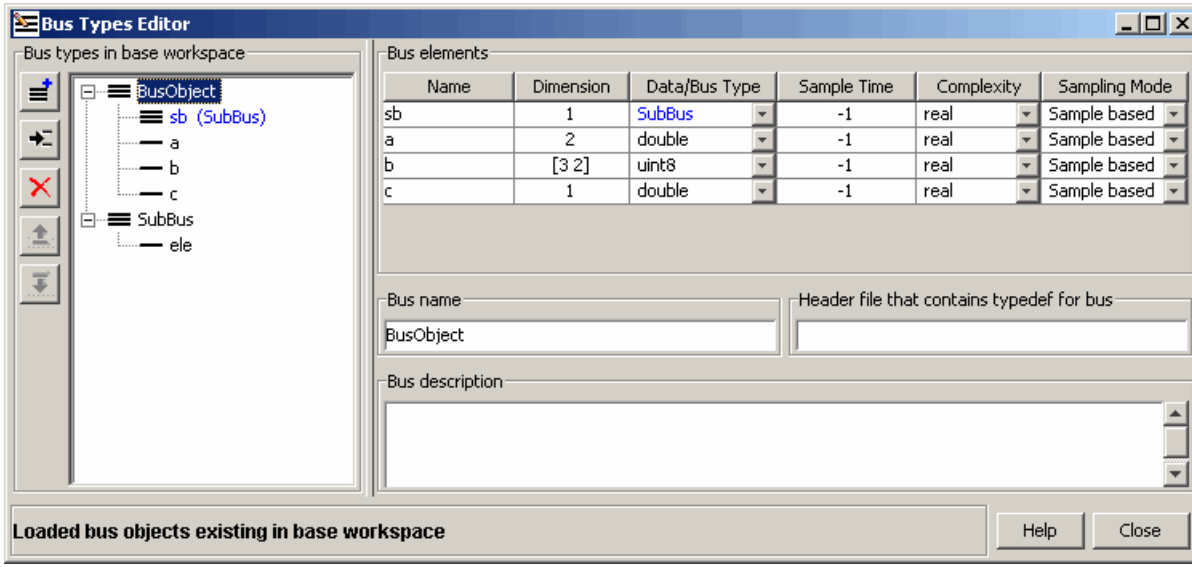
For example, consider the following model:



In this example, the following structures are defined in the Stateflow chart:

Name of Structure	Scope	Defined By Simulink.Bus Object
`in`	Input	`BusObject`
`out`	Output	`BusObject`
`subbus`	Local	`SubBus`

The `Simulink.Bus` objects that define these structures have the following elements:



By default, Stateflow structures `in` and `out` have the same fields — `sb`, `a`, `b`, and `c` — as the elements of `Simulink.Bus` object `BusObject`. Similarly, the Stateflow structure `subbus` has the same field `ele` as the element of `Simulink.Bus` object `SubBus`. Based on these specifications, the following table shows how the Stateflow chart resolves symbols in dot notation for indexing fields of the structures in this example:

Dot Notation	Symbol Resolution
`in.c`	Field `c` of input structure `in`

Dot Notation	Symbol Resolution
`in.a[1]`	Second value of the vector field `a` of input structure `in`
`out.sb`	Substructure `sb` of output structure `out`
`in.sb.ele[2][3]`	Value in the third row, fourth column of field `ele` of substructure `sb` of input structure `in`
`subbus.ele[1][1]`	Value in the second row, second column of field `ele` of local structure `subbus`

Assigning Values

You can assign values to any Stateflow structure *except* input structures — that is, a structures with scope equal to **Input**. Here are the guidelines for assigning values to output, local, and temporary structures:

Operation	Conditions
Assign one structure to another structure	You must define both structures with the same `Simulink.Bus` object in the base workspace.
Assign one structure to a substructure of a different structure and vice versa	You must define the structure with the same `Simulink.Bus` object in the base workspace as the substructure.
Assign a field of one structure to a field of another structure	The fields must have the same type and size. **Note** In this case, you do not need to define the Stateflow structures with the same `Simulink.Bus` object in the base workspace.

For example, the following table presents valid and invalid structure assignments based on the specifications for the model sfbus_demo, as described in “Example of Stateflow Structures” on page 17-2:

Assignment	Valid or Invalid?	Rationale
`outbus = inbus;`	Valid	Both `outbus` and `inbus` are defined by the same `Simulink.Bus` object, `COUNTERBUS`.
`inbus = outbus;`	Invalid	You cannot write to input structures.
`inbus.limits = outbus.limits;`	Invalid	You cannot write to fields of input structures.
`counterbus_struct = inbus;`	Valid	Both `counterbus_struct` and `inbus` are defined by the same `Simulink.Bus` object, `COUNTERBUS`.
`counterbus_struct.inputsignal = inbus.inputsignal;`	Valid	Both `counterbus_struct.inputsignal` and `inbus.inputsignal` have the same type and size because they each reference field `inputsignal`, a substructure of the `Simulink.Bus` object `COUNTERBUS`.
`outbus.limits.upper_saturation_limit = inbus.inputsignal.input;`	Valid	The field `upper_saturation_limit` from `limits`, a substructure of `COUNTERBUS`, has the same type and size as the field `input` from `inputsignal`, a different substructure of `COUNTERBUS`.
`outbus.limits = inbus.inputsignal;`	Invalid	The substructure `limits` is defined by a different `Simulink.Bus` object than the substructure `inputsignal`.

Getting Addresses

When you write custom functions that take structure pointers as arguments, you must pass the structures by address. To get addresses of Stateflow structures and structure fields, use the `&` operator, as in the following examples:

- `&in` — Address of Stateflow structure `in`
- `&in.b` — Address of field `b` in Stateflow structure `in`

The model sfbus_demo contains a custom C function `counterbusFcn` that takes structure pointers as arguments, defined as follows in a custom header file:

```
...
extern void counterbusFcn
            (COUNTERBUS *u1, int u2, COUNTERBUS *y1, int *y2);
...
```

To call this function, you must pass addresses to two structures defined by the `Simulink.Bus` object `COUNTERBUS`, as in this example:

```
counterbusFcn(&counterbus_struct, u2, &outbus, &y2);
```

See "Example of Stateflow Structures" on page 17-2 for a description of the structures defined in sfbus_demo.

Integrating Custom Structures in Stateflow Charts

You can define custom structures in C code, which you can then integrate with your Stateflow chart for simulation and Real-Time Workshop code generation. Follow these steps:

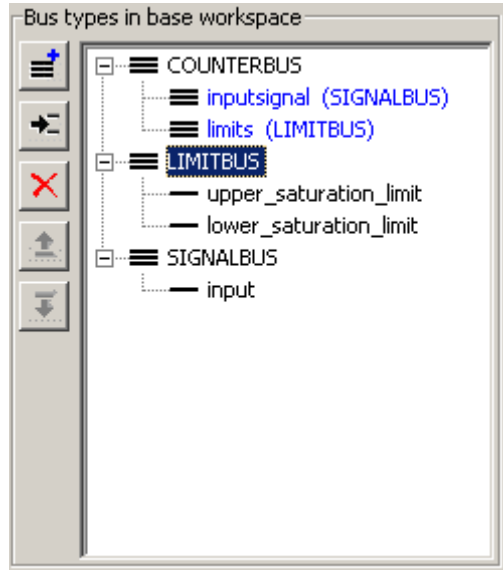
1. Define your structure in C, creating custom source and header files.

   The header file must contain the `typedef` statements for your structures. For example, the model sfbus_demo uses custom structures, defined in a custom header file as follows:

   ```
   ...
   #include "tmwtypes.h"

   typedef struct {
     int input;
   } SIGNALBUS;

   typedef struct {
     int upper_saturation_limit;
     int lower_saturation_limit;
   } LIMITBUS;

   typedef struct {
     SIGNALBUS inputsignal;
     LIMITBUS limits;
   } COUNTERBUS;
   ...
   ```

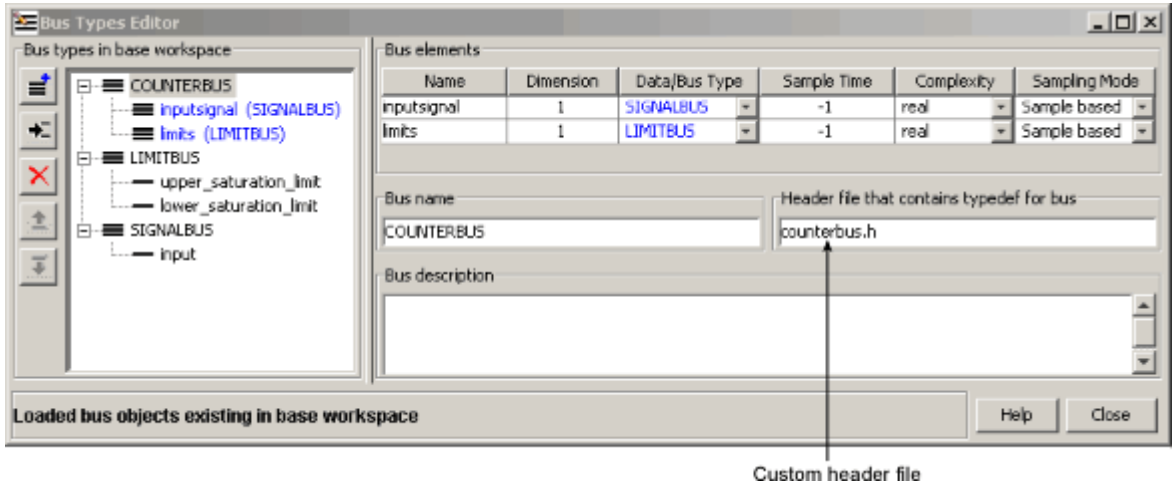
2. Define a `Simulink.Bus` object in the base workspace that matches each custom structure `typedef`.

   For example, the model sfbus_demo, defines the following `Simulink.Bus` objects to match each `typedef` in the custom header file:



3 Open the Bus Editor and for each bus object in the base workspace defined in custom code, add the name of the header file that contains the matching `typedef`.

For example, the model sfbus_demo specifies the custom header file **counterbus.h** for the bus object `COUNTERBUS`:



4 Configure your Stateflow chart to include your custom C code, as follows.

To Include Custom C Code:	Do This:
In code generated for simulation	Follow these steps: **1** Open the Stateflow chart that uses your custom C structures. The Stateflow Editor appears. **2** In the Stateflow Editor, select **Tools > Open Simulation Target**. The Configuration Parameters dialog box appears. **3** In the Configuration Parameters dialog box, select **Simulation Target > Custom Code** in the Select tree. Custom code options appear in the right pane. **4** Specify your custom code as described in "Task 1: Include Custom C Code in the Simulation Target" on page 22-9. For more information, see Chapter 22, "Building Targets".
In code generated for real-time applications	Follow these steps: **1** Open the Stateflow chart that uses your custom C structures. The Stateflow Editor appears. **2** In the Stateflow Editor, select **Tools > Open RTW Target**. The Configuration Parameters dialog box appears. **3** In the Configuration Parameters dialog box, select **Real-Time Workshop > Custom Code** in the Select tree. Custom code options appear in the right pane. **4** Follow instructions in "Configuring Custom Code" in the Real-Time Workshop User's Guide.

5 Build your model and fix errors (see “Debugging Structures” on page 17-25).

6 Run your model.

Debugging Structures

You debug structures as you would other Stateflow chart data, as described in Chapter 23, “Debugging and Testing Stateflow Charts”. Using the Stateflow Debugger, you can examine the values of structure fields during simulation, either from the graphical debugging window or from the command line, as described in “Watching Data Values with Debuggers” on page 23-30. To view the values of structure fields at the command line, use dot notation to index into the structure, as described in “Indexing Sub-Structures and Fields” on page 17-15.

18

Stateflow Design Patterns

This chapter describes Stateflow patterns that you can use to address design challenges that occur when developing and implementing embedded software. Think of these design patterns as templates that you can customize for your own applications.

Debouncing Signals

In this section...

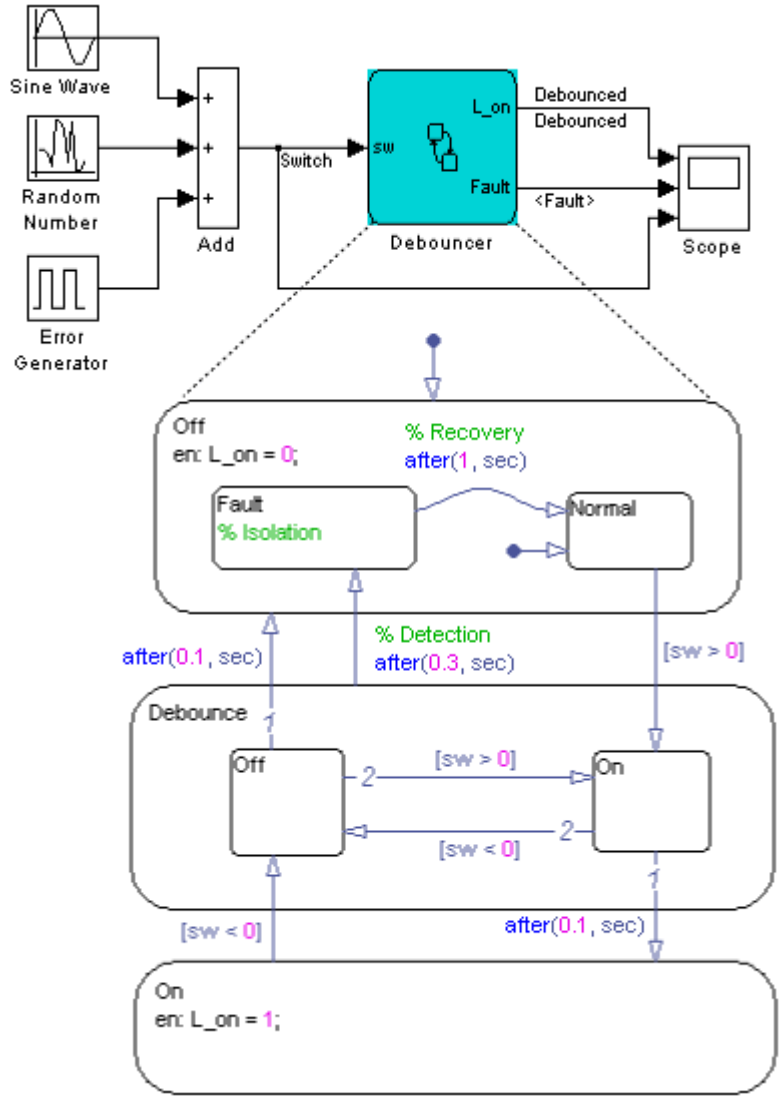
Why Debounce Signals

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 using a process called *debouncing*.

For example, 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. Instead, you can design a Stateflow debouncer that uses temporal logic to determine whether the switch is really on or off.

The Debouncer Model

The model `sf_debouncer` illustrates a design pattern that uses temporal logic to isolate transient signals.



Key Behaviors of Debouncer Chart

The key behaviors of the Debouncer chart are:

Intermediate Debounce State Isolates Transients

In addition to the states On and Off, 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 of time. The logic works as follows.

If the input signal...	Then this state...	Transitions to...	And the...
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
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

Temporal Logic Determines True State

The debouncer design pattern uses temporal logic to:

- Determine whether the input signal is normal or transient
- Give transient signals time to recover and return to normal state

Using Absolute-Time Temporal Logic. The debouncer design uses the `after(n, sec)` operator to implement absolute-time temporal logic (see "Operators for Absolute-Time Temporal Logic" on page 10-63). The keyword `sec` defines simulation time that has elapsed since activation of a state.

Using 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)` operator (see "Operators for Event-Based Temporal Logic" on page 10-57). The keyword `tick` specifies and implicitly generates a local event when the chart awakens (see "Using Implicit Events" on page 9-31).

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.

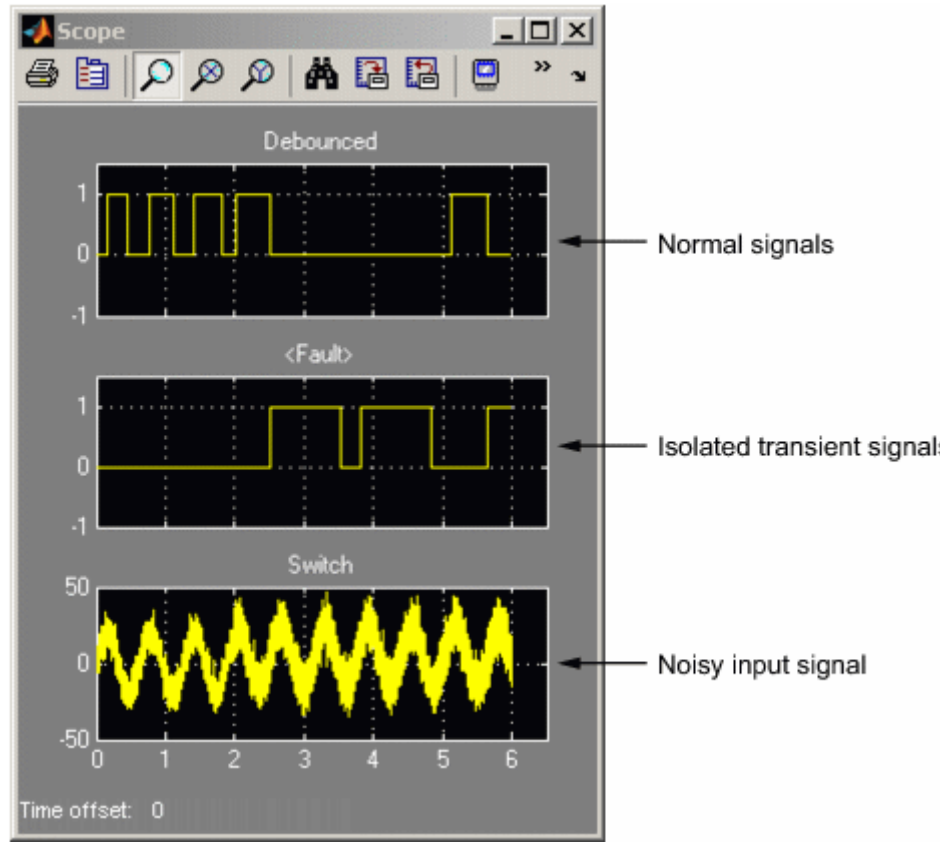
Absolute Time-Based Logic	Event-Based Logic
`after ( 0.1, sec )`	`after ( 100, tick )`
`after ( 0.3, sec )`	`after ( 300, tick )`
`after ( 1, sec )`	`after ( 1000, tick )`

Running the Debouncer

To run the `sf_debouncer` model, follow these steps:

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



Note To debounce the signals using event-based logic, change the Debouncer chart as described in "Using Event-Based Temporal Logic" on page 18-5 and simulate the chart again. You should get the same results.

Scheduling Execution of Simulink Subsystems

In this section...

When to Implement Schedulers Using Stateflow Charts

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.

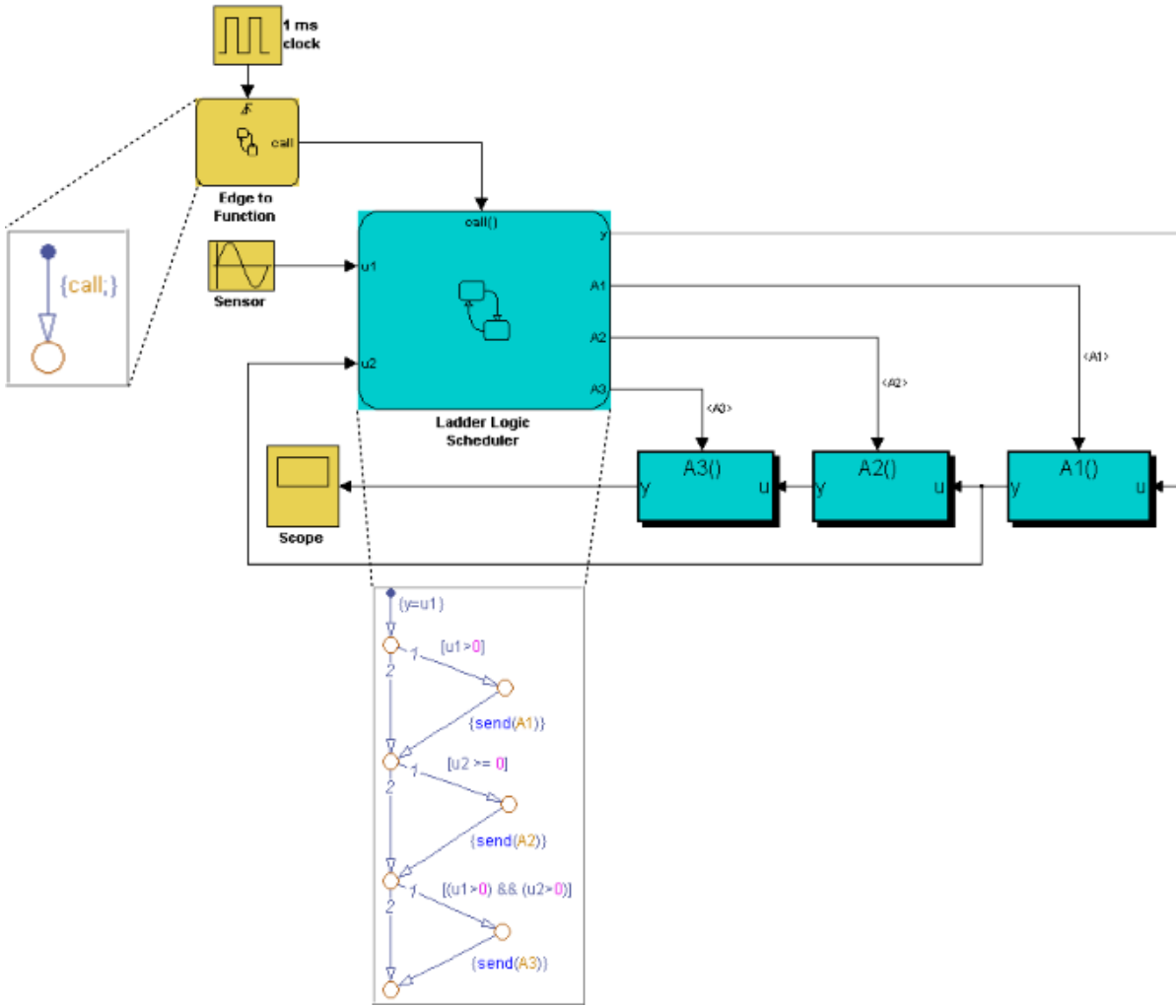
Types of Scheduler Patterns

You can implement the following types of schedulers using Stateflow charts.

Scheduler Design Pattern	Description
Ladder logic scheduler	Schedules multiple Simulink subsystems to execute in a single time step
Loop scheduler	Schedules one Simulink subsystem to execute multiple times in a single time step
Temporal logic scheduler	Schedules Simulink subsystems to execute at specific times

Scheduling Multiple Subsystems in a Single Time Step Using a Ladder Logic Scheduler

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.



Key Behaviors of Ladder Logic Scheduler

The key behaviors of the ladder logic scheduler are:

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 Graph Determines Order of Execution" on page 18-9).

Flow Graph 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 graph 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 Event Broadcasting Using send" on page 10-52).

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.

Running the Ladder Logic Scheduler

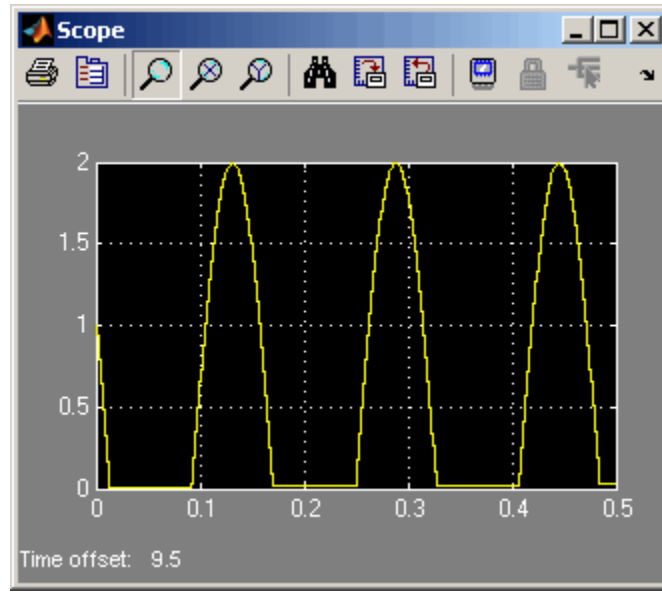
To run the sf_ladder_logic_scheduler model, follow these steps:

1 Open the model by typing sf_ladder_logic_scheduler at the MATLAB command prompt.

2 Open the Stateflow chart Ladder Logic Scheduler and the Scope block.

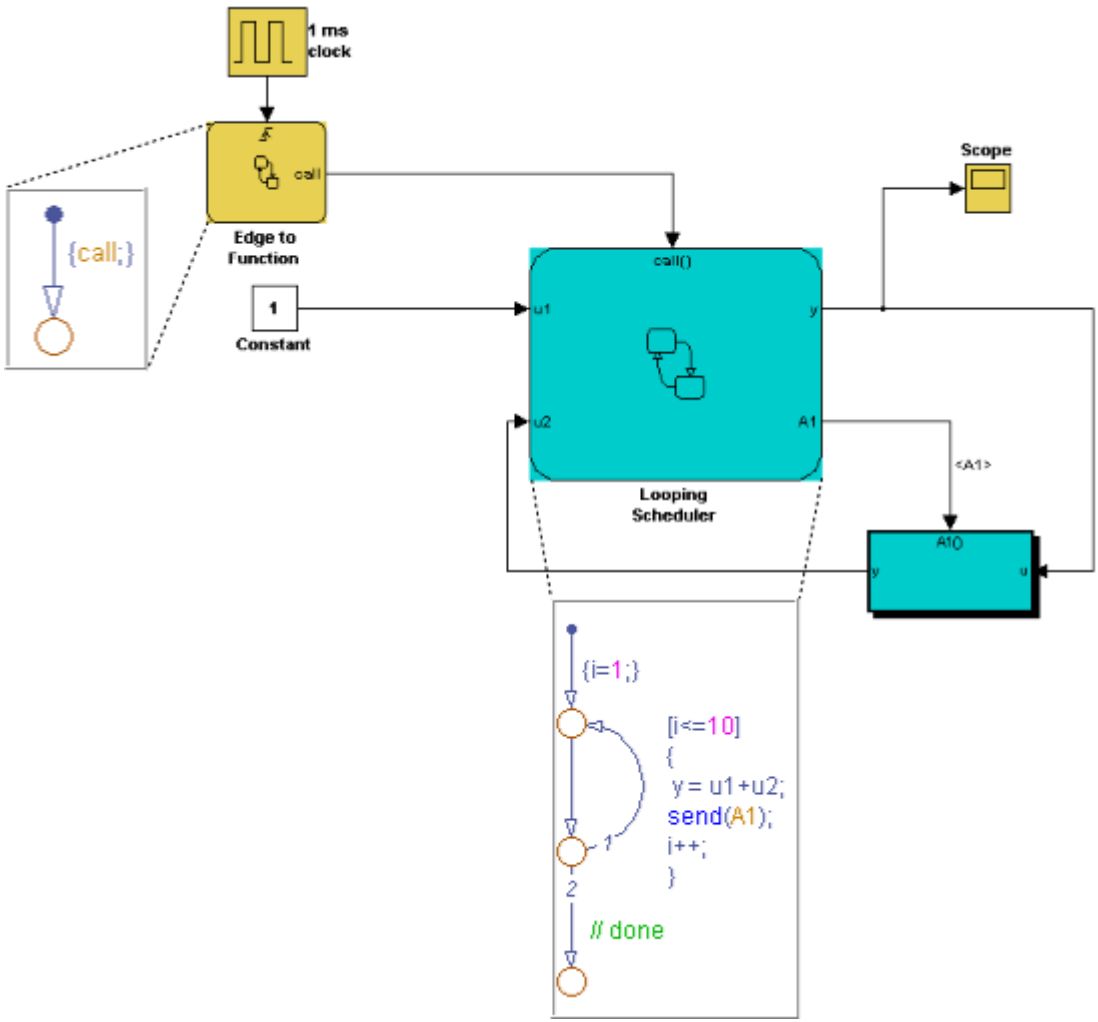
3 Simulate the chart.

The scope shows how output y changes, depending on which subsystems the Ladder Logic Scheduler chart calls during each time step.



Scheduling One Subsystem in a Single Time Step Using a Loop Scheduler

The **loop scheduler** design pattern allows you to schedule one Simulink subsystem to execute multiple times in a single time step. The model `sf_loop_scheduler` illustrates this design pattern.



Key Behaviors of Loop Scheduler

The key behaviors of the loop scheduler are:

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 Graph Implements For Loop" on page 18-13).

Flow Graph 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 graph 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.

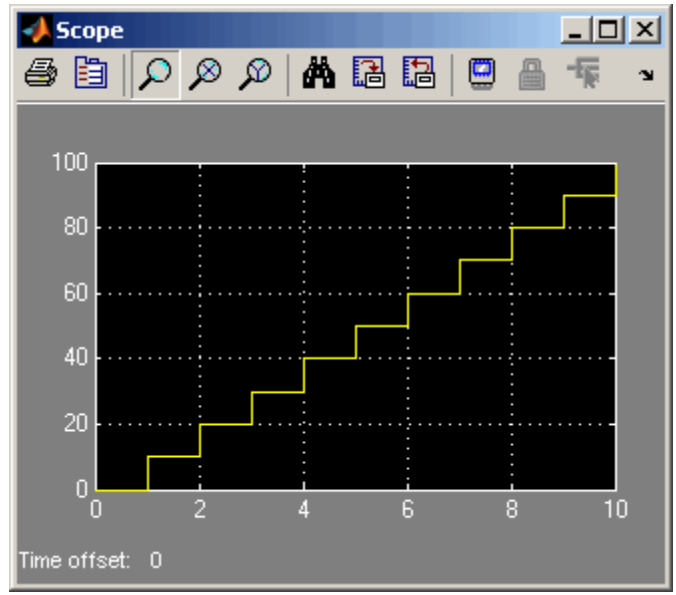
Running the Loop Scheduler

To run the `sf_loop_scheduler` model, follow these steps:

1. Open the model by typing `sf_loop_scheduler` at the MATLAB command prompt.
2. Open the Stateflow chart Looping Scheduler and the Scope block.

3 Simulate the chart.

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`).

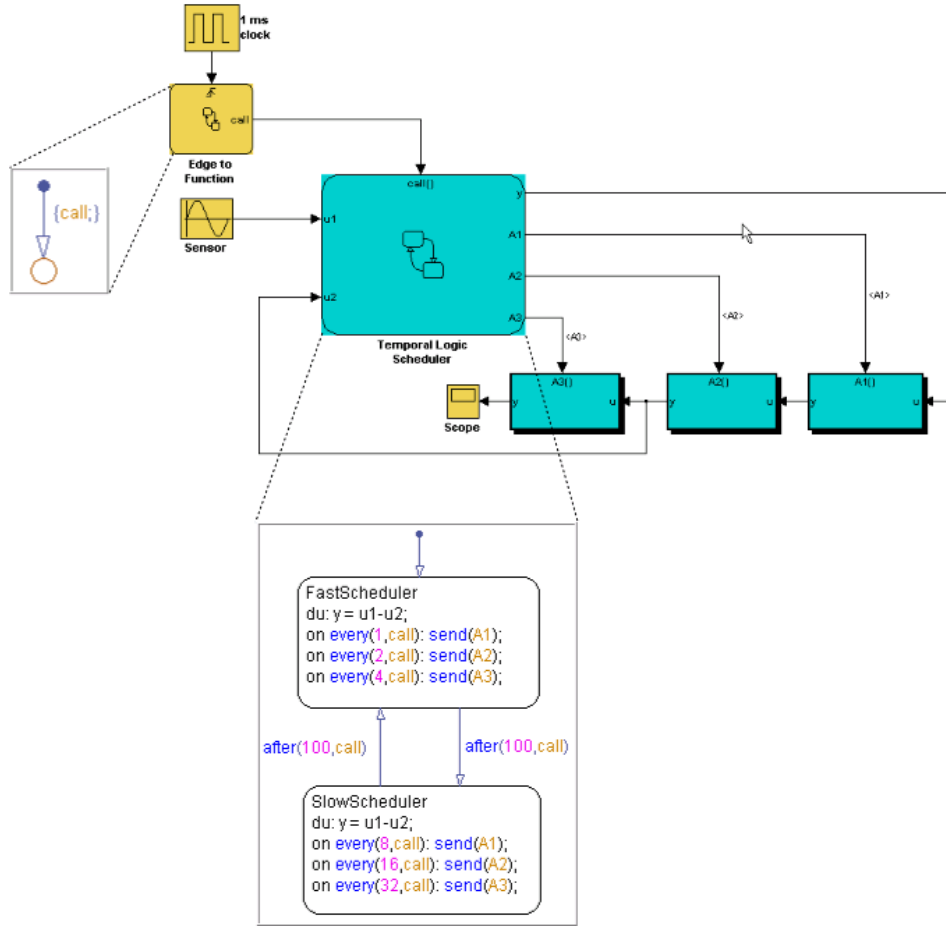
2 The chart broadcasts a function-call output event that executes subsystem `A1`.

3 Subsystem `A1` multiplies `y` by a gain of 1.

4 Control returns to the chart.

Scheduling Subsystems to Execute at Specific Times Using a Temporal Logic Scheduler

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.



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 10-57).

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

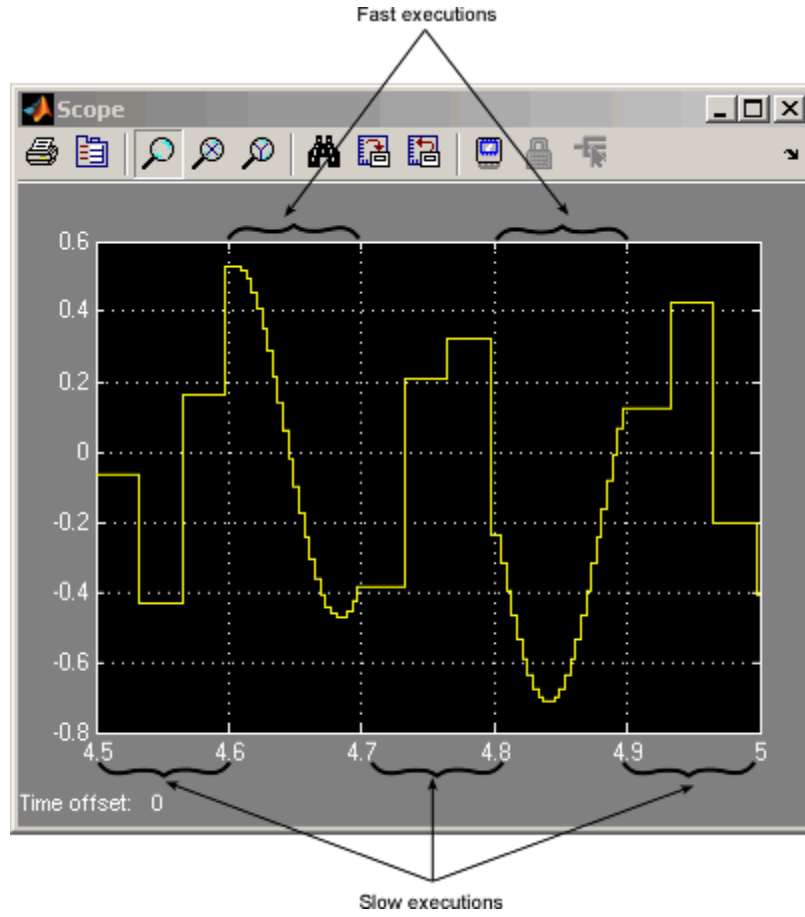
The SlowScheduler state schedules function calls less frequently — at 8, 16, and 32 times slower than the base rate. The chart switches between fast and slow executions after every 100 invocations of the `call` event.

Running the Temporal Logic Scheduler

To run the `sf_temporal_logic_scheduler` model, follow these steps:

1. Open the model by typing `sf_temporal_logic_scheduler` at the MATLAB command prompt.
2. Open the Stateflow chart Temporal Logic Scheduler and the Scope block.
3. Simulate the chart.

The scope illustrates the different rates of execution.



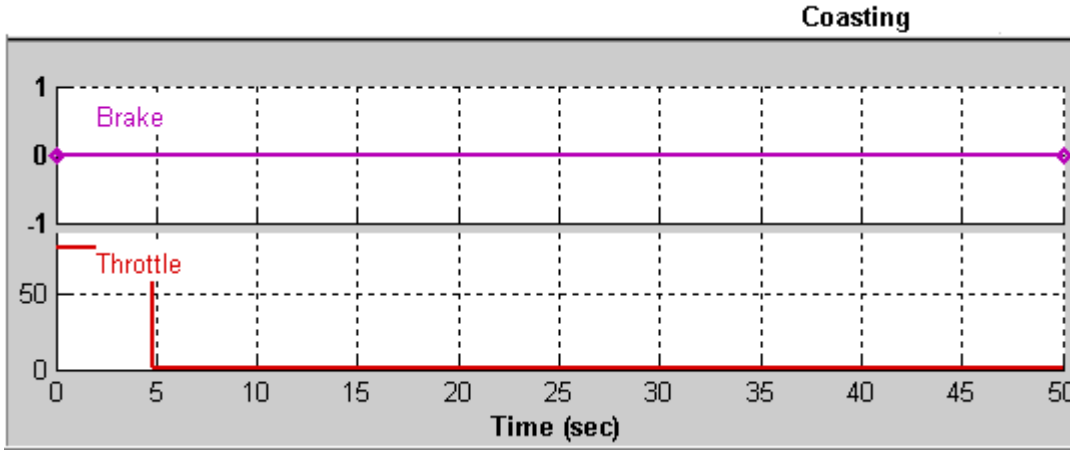
Implementing Dynamic Test Vectors

In this section...

When to Implement Test Vectors Using Stateflow Charts

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.



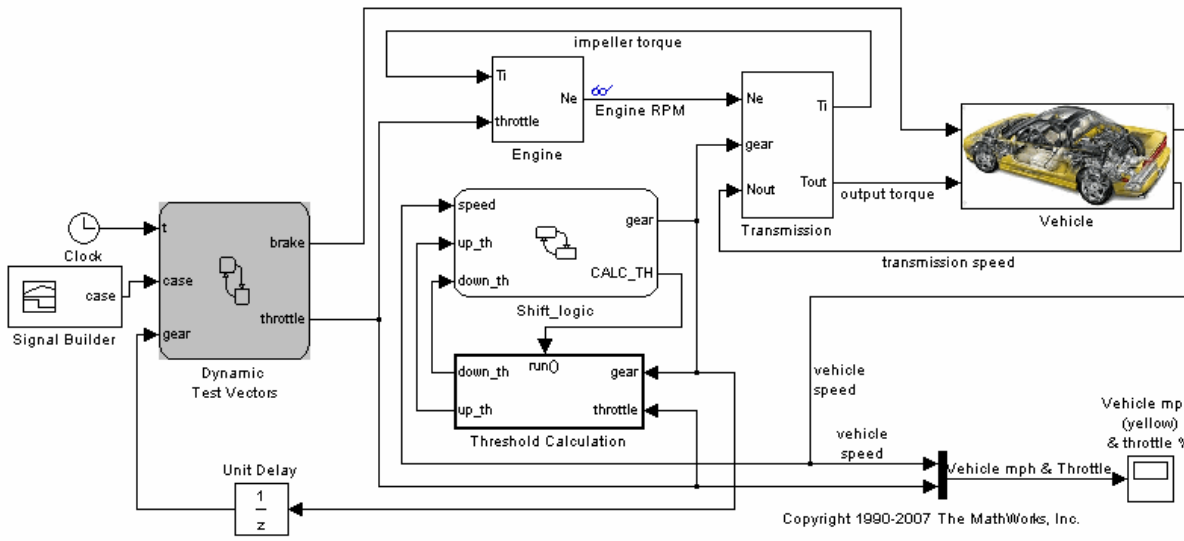
However, this approach has limitations. The signal changes value based on time, but cannot respond dynamically to changes in the system that

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.

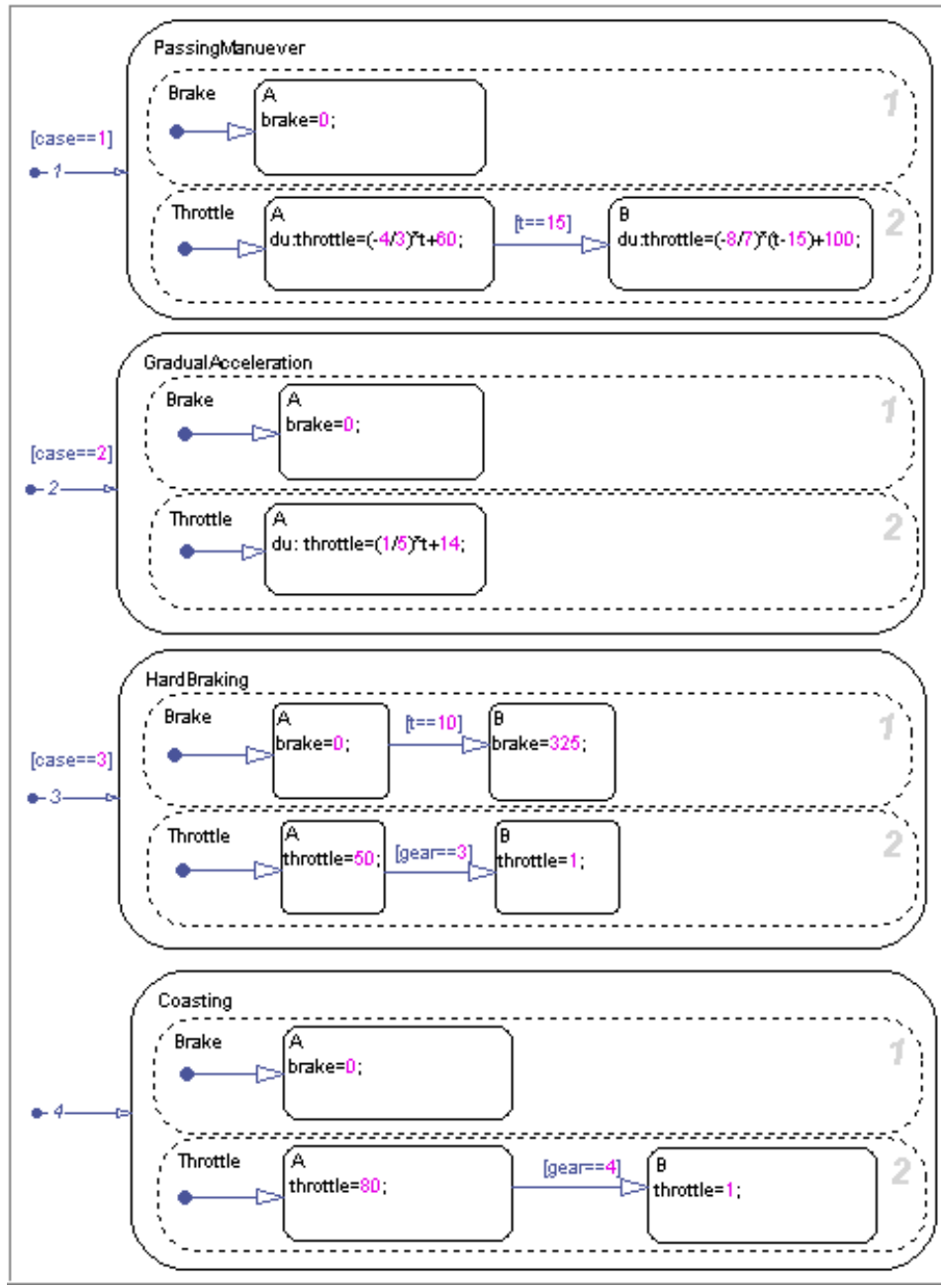
By contrast, you can use Stateflow charts to develop test vectors that use conditional logic to evaluate and respond to changes in system state as they occur. For example, to test the coasting scenario, the chart can evaluate an output that represents the gear range and reduce speed only after the transmission 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. For a detailed look at this type of chart, see “A Dynamic Test Vector Chart” on page 18-19.

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 Simulink 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.

Note This model is based on the Simulink demo model `sldemo_autotrans`.

Key Behaviors of the Test Vector Chart and Model

The key behaviors of the test vector chart and model are:

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:

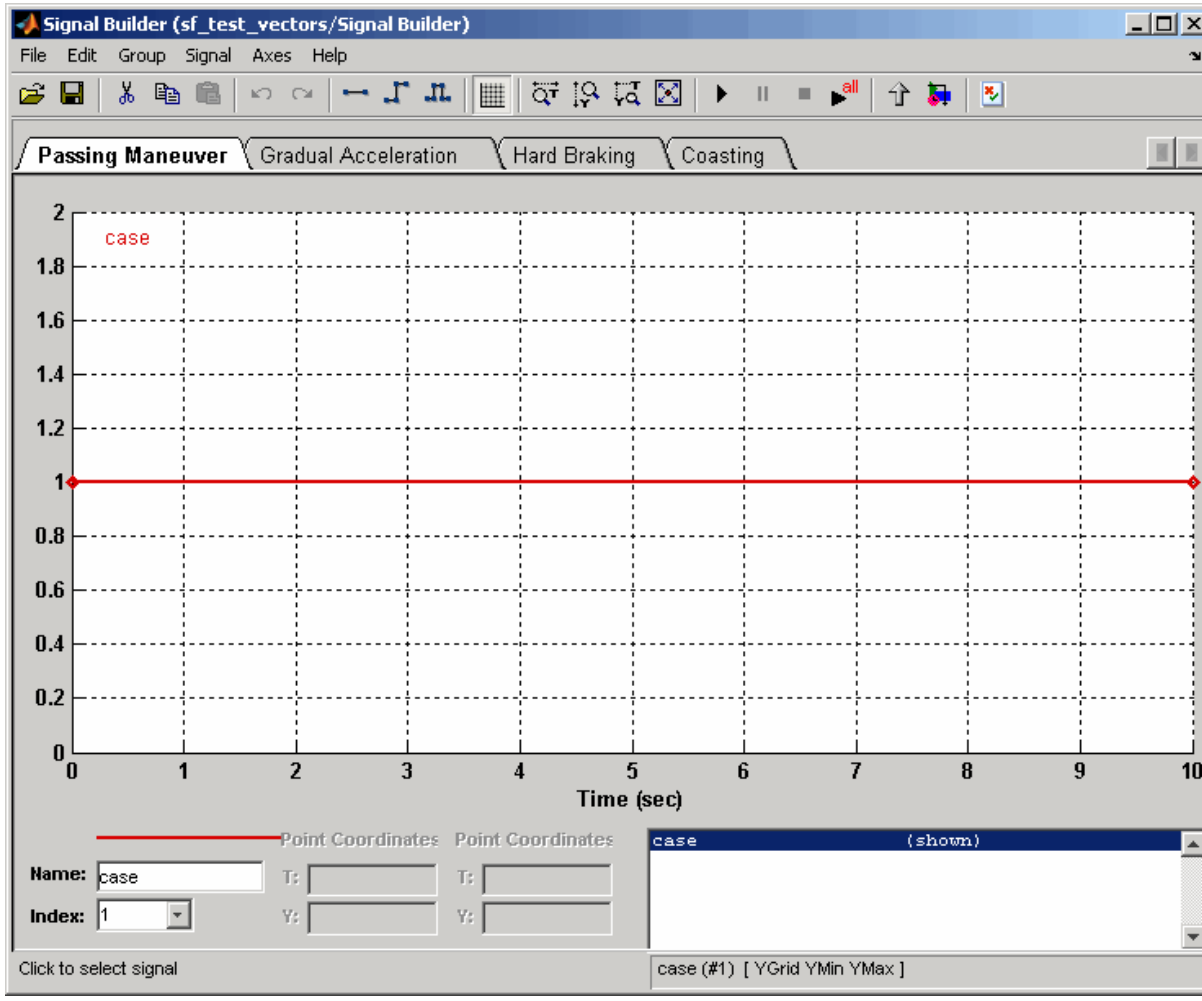
- Conditional logic (see "State Action Types" on page 10-2 and "Transition Action Types" on page 10-7)
- Temporal logic (see "Using Temporal Logic in State Actions and Transitions" on page 10-56)

- Change detection operators (see "Using Change Detection in Actions" on page 10-74)
- MATLAB functions (see "Using MATLAB Functions and Data in Actions" on page 10-33)

For more information, see Chapter 10, "Using Actions in Stateflow Charts".

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.



Selecting and Running Test Cases. In the Signal Builder, select and run test cases as follows:

To Test:	Do This:
One case	Select 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® Verification and Validation™ software license*)	Click the **Run all and produce coverage** button: 

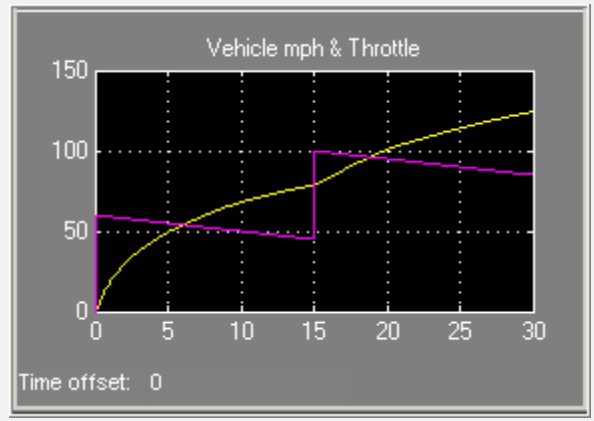
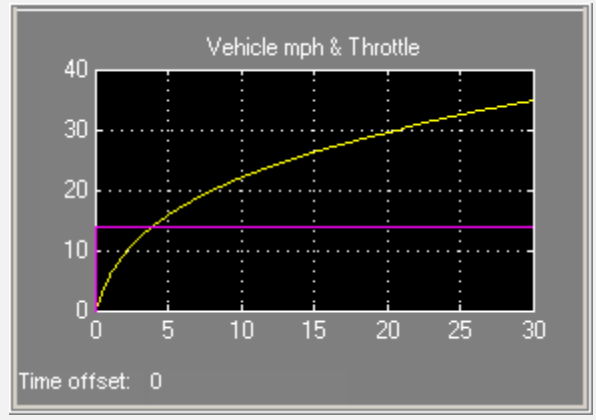
The Signal Builder block sends to the Dynamic Test Vectors chart one or more constant signal values that correspond to the driving scenarios you select. The chart uses these values to activate the appropriate test cases.

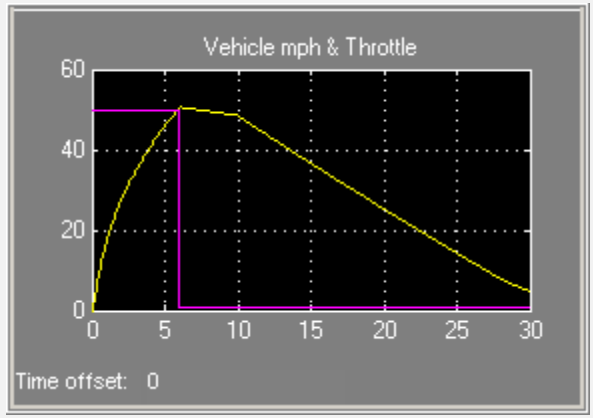
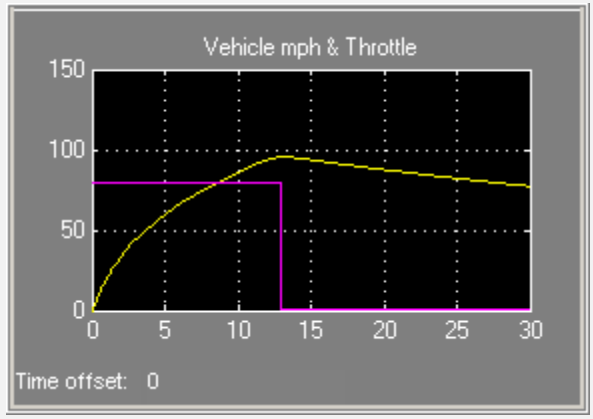
Running the Model with Stateflow Test Vectors

To run the `sf_test_vectors` model, follow these steps:

1. Open the model by typing `sf_test_vectors` at the MATLAB command prompt.
2. Open the Stateflow chart Dynamic Test Vectors, the Signal Builder block, and the Scope block.
3. Select and simulate a driving scenario from the Signal Builder block, as described in “Selecting and Running Test Cases” on page 18-23.

   The scope illustrates 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.

19

Truth Table Functions

What Is a Truth Table?

Truth table functions implement logical decision-making behavior that you call in an action language. Stateflow truth tables contain conditions, decisions, and actions arranged as follows:

Condition	Decision 1	Decision 2	Decision 3	Default Decision
x == 1	T	F	F	-
y == 1	F	T	F	-
z == 1	F	F	T	-
Action	t = 1	t = 2	t = 3	t = 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, that is referred to as a decision.

You evaluate a truth table one decision at a time, starting with Decision 1. If one of the decisions is true, you perform its action and truth table execution is complete. For example, if conditions 1 and 2 are false and condition 3 is true, Decision 3 is true and the variable `t` is set equal to 3. The remaining decisions are not tested and evaluation of the truth table is finished.

The last decision in the preceding example, Default Decision, covers all possible remaining decisions. If Decisions 1, 2, and 3 are false, then the Default Decision is automatically true and its action (t = 4) is executed. You can see this behavior when you examine the following equivalent pseudocode for the evaluation of the preceding truth table example:

Description	Pseudocode
Decision 1 Decision 1 Action	`if ((x == 1) & !(y == 1) & !(z == 1))` `  t = 1;`

Description	Pseudocode
Decision 2 Decision 2 Action	`elseif (!(x == 1) & (y == 1) !(z == 1))` `  t = 2;`
Decision 3 Decision 3 Action	`elseif (!(x == 1) & !(y == 1) (z == 1))` `  t = 3;`

Default Decision
Default Decision Action
	`else`
	`  t = 4;`
	`endif`

Language Options for Stateflow Truth Tables

In this section...

Stateflow Classic Truth Tables

Using Stateflow Classic truth tables, you can specify conditions and actions using the Stateflow action language, which supports basic C constructs and provides access to MATLAB functions using the `ml` namespace operator or `ml` function. For more information about the Stateflow action language, see Chapter 10, "Using Actions in Stateflow Charts".

Stateflow Classic mode is the default setting for Stateflow truth tables.

Embedded MATLAB Truth Tables

You can specify conditions and actions for Embedded MATLAB truth tables by using Embedded MATLAB action language, a restricted subset of the MATLAB language that provides optimizations for code generation.

Embedded MATLAB truth tables offer several advantages over Stateflow Classic truth tables:

- The Embedded MATLAB language subset 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 Embedded MATLAB library functions (for example MATLAB `sin` and `fft` functions) and generate code for these functions using Real-Time Workshop code generation software.

- You can create temporary or persistent variables during simulation or in code directly without having to define them in the Model Explorer.
- Embedded MATLAB language subset uses a better debugging scheme. It is easier to set breakpoints on lines of code, step through code, and watch data values through tool tips.
- You can use persistent variables in truth table actions. This feature allows you to define data that persists across multiple calls to the truth table function during simulation.
- You get more comprehensive model coverage. Embedded MATLAB truth tables generate coverage reports on branches in conditions *and* actions. Stateflow Classic truth tables provide coverage reports for conditions only. For more information, see "Model Coverage for Truth Tables" on page 19-56.

Selecting a Language for Stateflow Truth Tables

To specify an action language for your Stateflow truth table, follow these steps:

1 Double-click the truth table to open the Truth Table Editor.

2 Select **Language** from the **Settings** menu.

3 Choose a language from the drop-down menu.

Migrating from Stateflow Classic to Embedded MATLAB Truth Tables

When you migrate from a Stateflow Classic truth table to an Embedded MATLAB truth table, you must verify that the code used to program the actions conforms to Embedded MATLAB syntax. Between the two action languages, these differences exist:

- In the Embedded MATLAB action language, indices are one-based; in the Stateflow action language, you can specify the first index.
- In the Embedded MATLAB action language, the expression for *not equal to* is ~=; in the Stateflow action language, the equivalent syntax is `!=`.

You can check for syntax errors by using the Run Diagnostics command in the Truth Table Editor, as described in "Checking Truth Tables for Errors" on page 19-43.

Workflow for Using Truth Tables

Here is the recommended workflow for using truth tables in Simulink models:

1. Add a truth table to your Simulink model using one of the methods described in "Building a Simulink Model with a Stateflow Truth Table" on page 19-7.
2. Specify properties of the truth table function, as described in "Specifying Properties of Truth Table Functions in Stateflow Charts" on page 19-13.
3. Select an action language and program the conditions and actions in the truth table, as described in "Programming a Truth Table" on page 19-22.
4. Debug the truth table for syntax errors and for error during simulation, as described in "Debugging a Truth Table" on page 19-43.
5. Check coverage of conditions and actions in the truth table, as described in "Model Coverage for Truth Tables" on page 19-56.
6. Simulate the model and check the generated content for the truth tables, as described in "How Stateflow Software Implements Truth Tables" on page 19-60.

Building a Simulink Model with a Stateflow Truth Table

In this section...

Methods for Adding Truth Tables to Simulink Models

There are several ways to add a Stateflow truth table to a Simulink model:

Procedure	Action Languages Supported	How To Do It
Add a Truth Table block directly to the model	Embedded MATLAB only	See Truth Table.
Add a Stateflow block that calls a truth table function	Stateflow Classic and Embedded MATLAB	See "Adding a Stateflow Block that Calls a Truth Table Function" on page 19-7.

Adding a Stateflow Block that Calls a Truth Table Function

This section describes how to add a Stateflow block to your Simulink model, and then create a chart that calls a truth table function. These topics are covered:

- "Creating a Simulink Model" on page 19-8
- "Creating a Stateflow Truth Table" on page 19-10
- "Specifying Properties of Truth Table Functions in Stateflow Charts" on page 19-13
- "Calling a Truth Table in a Stateflow Action" on page 19-15
- "Creating Truth Table Data in Stateflow Charts and Simulink Models" on page 19-18

Once you build a model in this section, finish it by programming the truth table with its behavior in “Programming a Truth Table” on page 19-22.

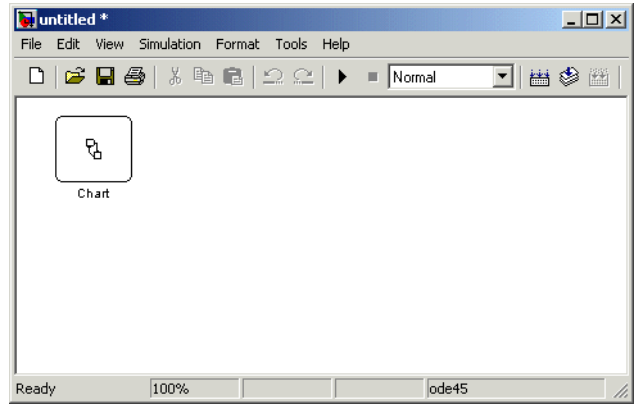
Creating a Simulink Model

To execute a truth table, you first need a Simulink model that calls a Stateflow block. Later, you will create a Stateflow chart for the Stateflow block that calls a truth table function. In this section, you create a Simulink model that calls a Stateflow block with the following procedure:

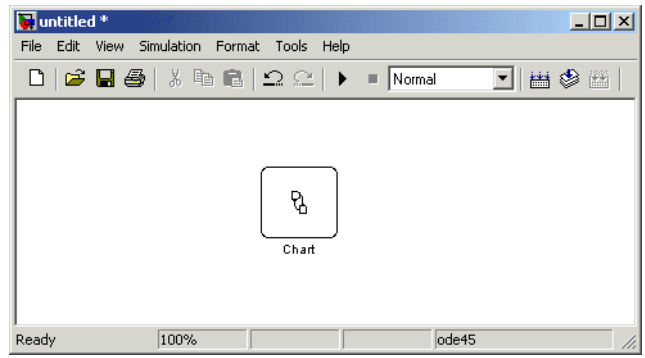
1 At the MATLAB prompt, enter the following command:

```
sfnew
```

An untitled Simulink model with a Stateflow block appears.



2 Click and drag the Stateflow block to the center of the Simulink window.



This action makes room for the blocks you add in the steps that follow.

3 In the Simulink window, select **View > Library Browser**.

The Simulink Library Browser window opens with the **Simulink** node expanded.

4 Under the **Simulink** node, select the **Sources** library.

The right pane of the Simulink Library Browser window displays the block members of the **Sources** library.

5 From the right pane of the Simulink Library Browser window, click and drag the Constant block to the left of the Stateflow block in the Simulink model.

6 Add two more Constant blocks to the left of the Chart block, and add a Display block (from the Sinks library) to the right of the Chart block.

7 In the Simulink model window, double-click the middle Constant block.

8 In the resulting Block Parameters dialog box, change the **Constant value** field to 0.

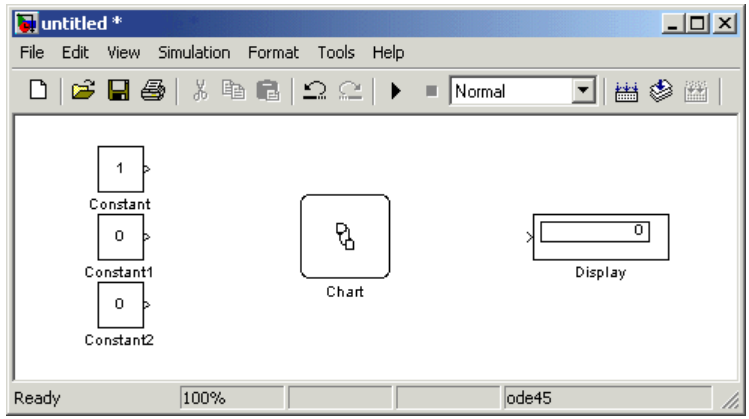
9 Click **OK** to close the dialog box.

10 In the Simulink model window, double-click the bottom Constant block.

11 In the resulting Block Parameters dialog box, change the **Constant value** field to 0.

12 Click **OK** to close the dialog box.

Your model should now have the following appearance.



13 In the Simulink model window, from the **Simulation** menu, select **Configuration Parameters**.

The Configuration Parameters dialog box opens.

14 Set

- Solver Options **Type** field to `Variable-step`
- **Stop Time** to `inf`

15 Click **OK** to accept these values and close the Configuration Parameters dialog box.

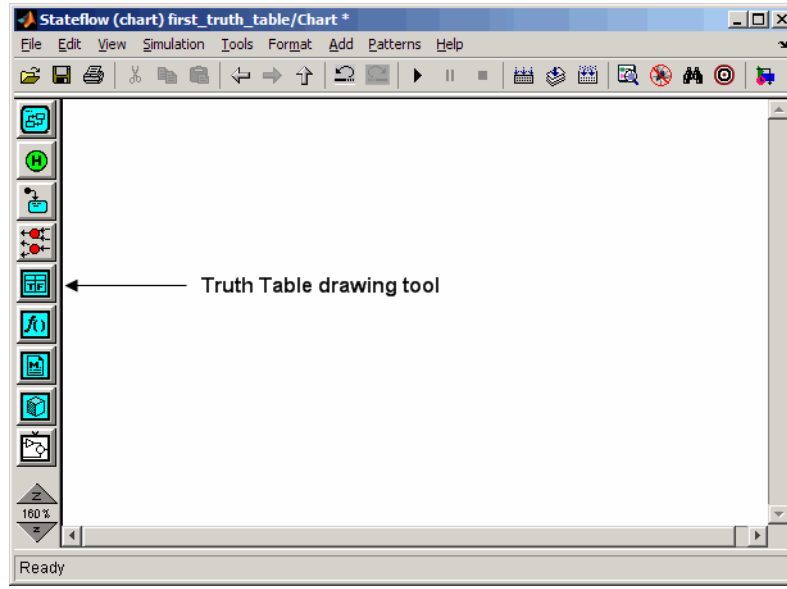
16 Save the model as `first_truth_table.mdl`.

Creating a Stateflow Truth Table

You created a Simulink model in "Creating a Simulink Model" on page 19-8 that contains a Stateflow block. Now you need to open the Stateflow chart for the block and specify a truth table for it in these steps:

1 In the Simulink model, double-click the Stateflow block named Chart.

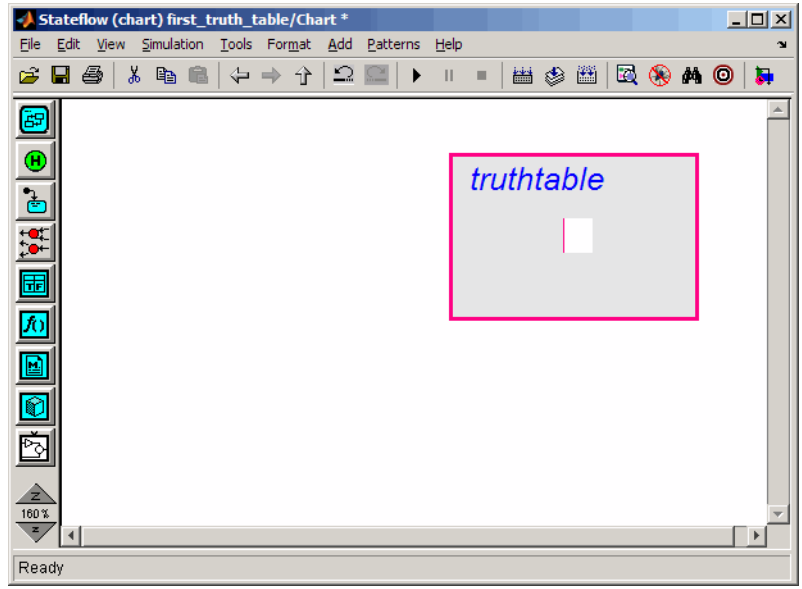
An empty Stateflow Editor appears.



2 In the Stateflow Editor, click the Truth Table drawing tool:

3 Move your pointer into the empty chart area and notice that it appears in the shape of a box.

4 Click to place a new truth table as shown.

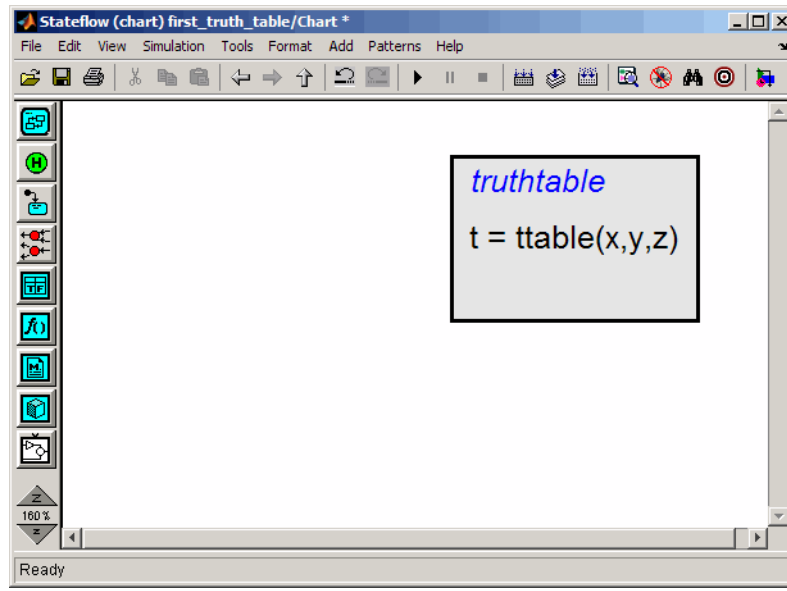


A shaded box appears with the title `truthtable` and a flashing text cursor in the middle of the box.

5 Enter the signature label

```
t = ttable(x,y,z)
```

and click outside the truth table box.



You must label a truth table 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 functions with only one return value, you can omit the brackets in the signature label.

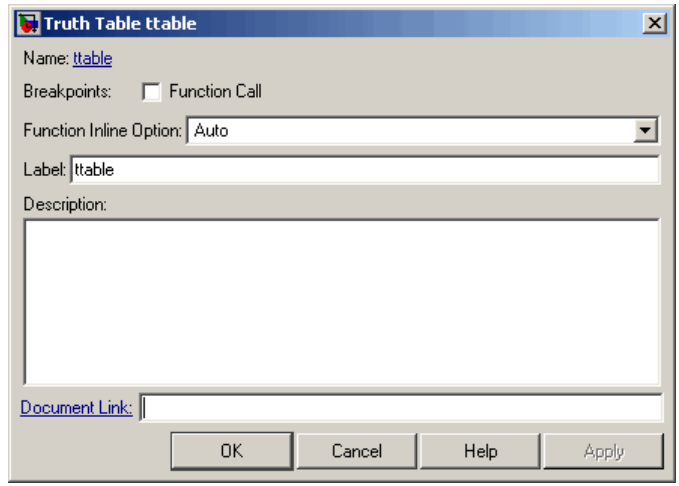
Specifying Properties of Truth Table Functions in Stateflow Charts

After you add a truth table function to a Stateflow chart, you can specify its properties by following these steps:

1 Right-click the truth table function box.

2 Select **Properties** from the context menu.

The Truth Table properties dialog box for the truth table function appears.



The fields in the Truth Table properties dialog box are as follows:

Field	Description
Name	Function name; read-only; click this hypertext link to bring the truth table function to the foreground in its native Stateflow chart.
Breakpoints	Select **Function Call** to set a breakpoint to pause execution during simulation when the truth table function is called.

Field	Description
Function Inline Option	This option controls the inlining of the truth table function in generated code through the following selections: • **Auto** Decides whether or not to inline the truth table function based on an internal calculation. • **Inline** Inlines the truth table function as long as it is not exported to other charts and is not part of a recursion. A recursion exists if the function calls itself either directly or indirectly through another called function. • **Function** Does not inline the function.
Label	You can specify the signature label for the function through this field. See "Creating a Stateflow Truth Table" on page 19-10 for more information.
Description	Textual description/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`.

Calling a Truth Table in a Stateflow Action

In "Creating a Stateflow Truth Table" on page 19-10, you created the truth table function `ttable` with the signature

```
t = ttable(x,y,z)
```

Now you need to specify a call to the truth table function in the Stateflow chart. Later, when the chart executes during simulation, it calls the truth table.

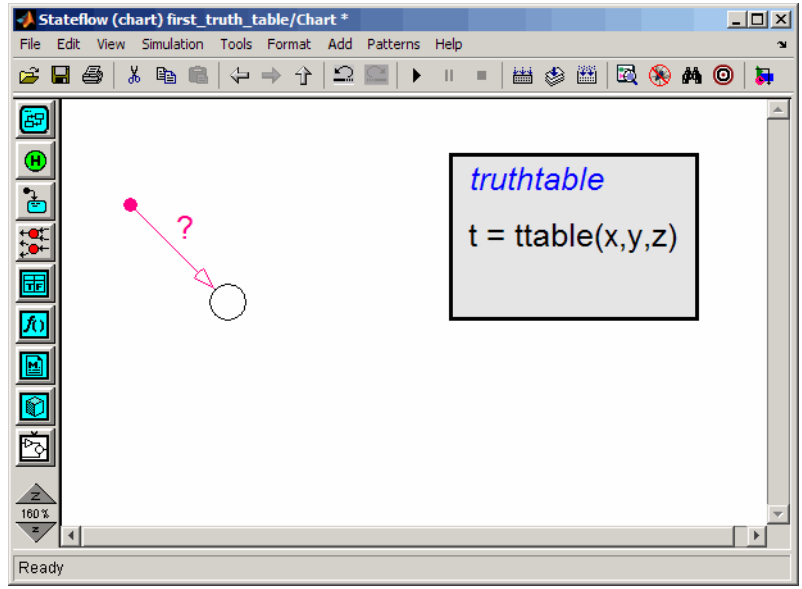
You can call truth table functions from the actions of any state or transition. You can also call truth tables from other functions, including graphical functions and other truth tables. Also, if you export a truth table, you can call it from any Stateflow chart in the model.

Use these steps to call the `ttable` function from the default transition of its own Stateflow chart.

1. Select the Default Transition button  from the drawing toolbar.

2. Move your pointer to a location left of the truth table function and notice that it appears in the shape of a downward-pointing arrow.

3. Click to place a default transition into a terminating junction.



4. Click the question mark character `(?)` that appears on the highlighted default transition.

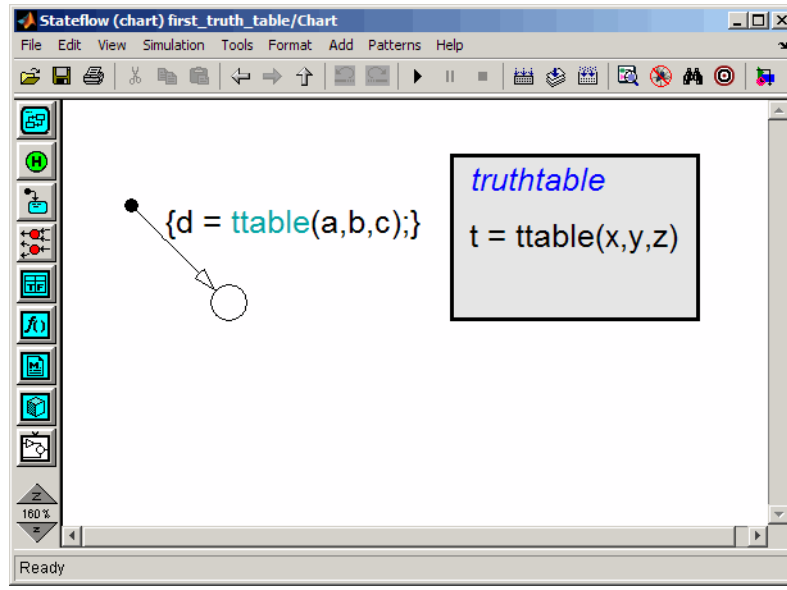
   A blinking cursor in a text field appears for entering the label of the default transition.

5 Enter the text

```
{d = ttable(a,b,c);}
```

and click outside the transition label to finish editing it.

You might want to adjust the label's position by clicking and dragging it to a new location. The finished Stateflow chart should have the following appearance:



The label on the default transition that you entered provides a condition action that calls the truth table with arguments and a return value. When the Simulink model triggers the Stateflow block during simulation, the default transition is taken and a call to the truth table `ttable` is made.

The call to the truth table in Stateflow action language must match the truth table signature. This means that the type of the return value `d` must match the type of the signature return value `t`, and the type of the arguments `a`, `b`, and `c` must match the type of the signature arguments `x`, `y`, and `z`. You ensure this with a later step in this section when you create the data that you use in the Stateflow chart.

6 From the **File** menu, select **Save** to save the model.

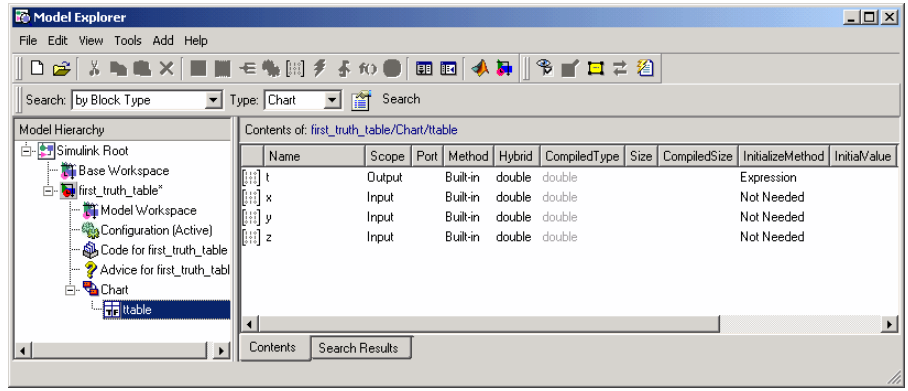
Creating Truth Table Data in Stateflow Charts and Simulink Models

When you create a truth table with its own signature, you specify data for it in the form of a return value (`t`) and argument values (`x`, `y`, `z`). When you specify a call to a truth table, as you did in "Calling a Truth Table in a Stateflow Action" on page 19-15, you specify data that you pass to the return and argument values of the truth table (`d`, `a`, `b`, and `c`). Now you must define this data for the Stateflow chart in these steps:

1 Double-click the truth table to open the Truth Table Editor.

2 In the Truth Table Editor, select the **Edit Data/Ports** button  .

The Model Explorer appears.



In the **Model Hierarchy** pane, the node for the function `ttable` is highlighted, and the **Contents** pane displays the output (t) and inputs (`x`, `y`, `z`) for `ttable`. By default, these data are defined as scalars of type `double`. If you want to redefine these data with a different array size and type, you do it in the Model Explorer. However, no changes are necessary for this example.

Notice also in the **Model Hierarchy** pane that the node above the function `ttable` is `Chart`, the name of the Stateflow chart that contains the truth table `ttable`.

3 In the **Model Hierarchy** pane, select **Chart**.

Notice that `Chart` contains no data in the **Contents** pane. You need to add the return and argument data used in calling `ttable`.

4 Select **Add** > **Data**.

A scalar data is added to the chart in the **Contents** pane of the Model Explorer with the default name `data`. This data matches argument `x` in type and size.

Tip To verify that the properties match, right-click `data` in the **Contents** pane and select **Properties**. The property sheet shows that the type is **double** and the size is scalar (the default when there is no entry in the **Size** field).

---

5 In the **Contents** pane, double-click the entry `data` in the **Name** column.

A small text field opens with the name `data` highlighted.

6 In the text field, change the name to `a` and click **Enter**.

7 Click the entry `Local` under the **Scope** column.

A drop-down menu of selectable scopes appears with `Local` selected.

8 Select `Input`.

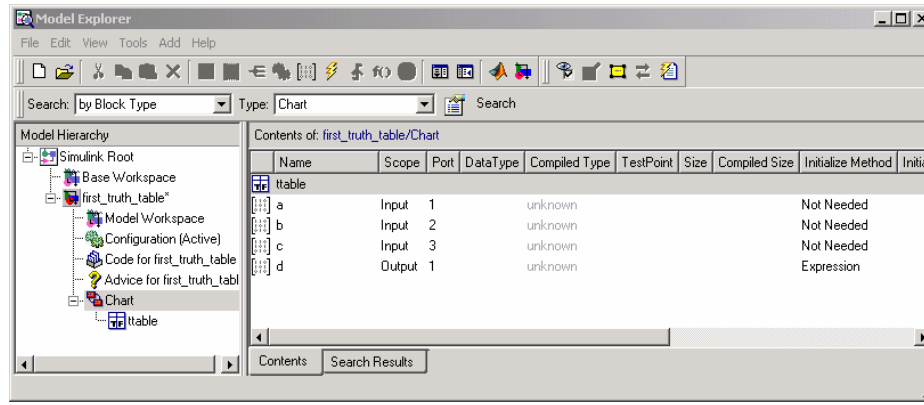
The scope `Input` means that the Simulink model provides the value for this data, which it passes to the Stateflow chart through an input port on the Stateflow block.

You should now see the new data input `a` in the **Contents** pane.

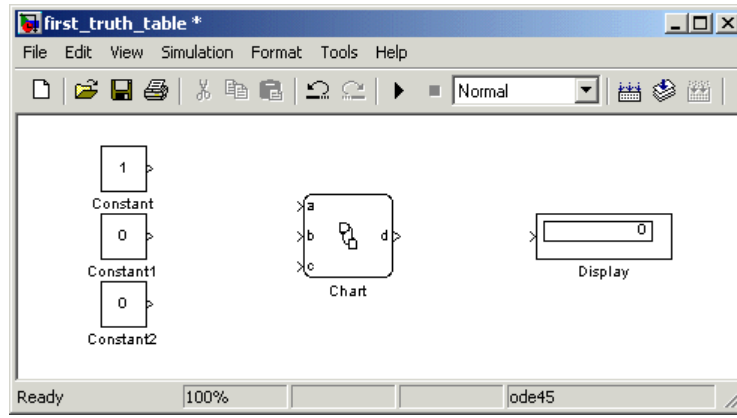
9 Repeat steps 3 through 7 to add the data `b` and `c` with the scope `Input`, and data `d` with a scope of `Output`.

The scope `Output` means that the Stateflow chart provides this data and passes it to the Simulink model through an output port on the Stateflow block.

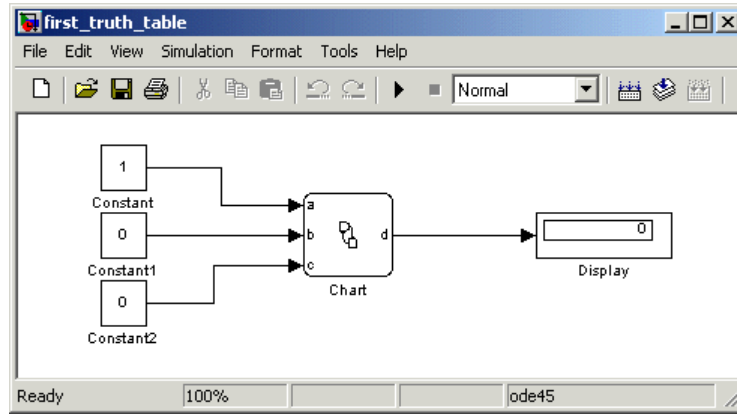
You should now see the following data in the Model Explorer.



The data `a`, `b`, `c`, and `d` match their counterparts `x`, `y`, `z`, and `t` in the truth table signature in size (scalar) and type (double), but have sources outside the Stateflow block. Notice that input ports for `a`, `b`, and `c`, and an output port for `d` appear on the Stateflow block in the Simulink model.



10 Complete connections to the Simulink blocks as shown.



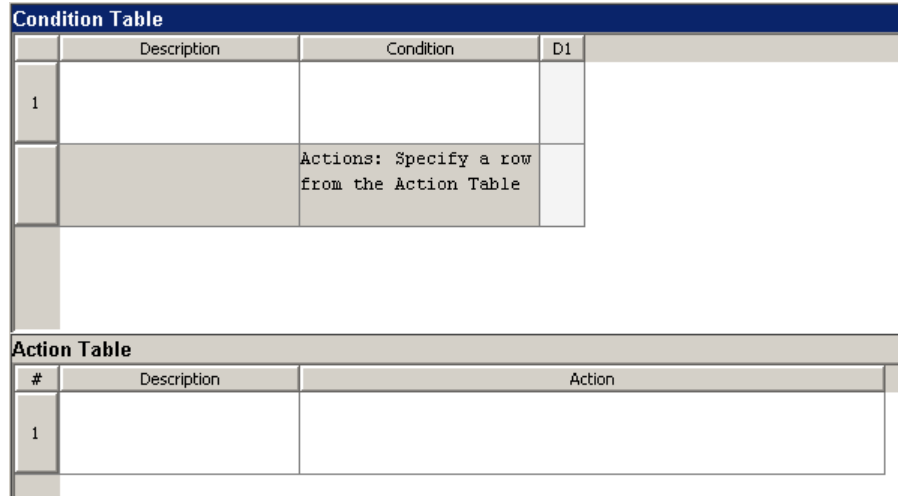
11 To save the model, select **File > Save**.

Programming a Truth Table

In this section...

Opening a Truth Table for Editing

After you create and label a truth table in a Stateflow chart, you specify its logical behavior. Double-click the truth table function to open the Truth Table Editor.



The Truth Table Editor is titled in <*model name*>/<*truth table name*> format in its header. An empty default truth table contains a **Condition Table** and

an **Action Table**, each with one row. The **Condition Table** also contains a single decision column, **D1**, and a single action row.

Selecting An Action Language

Select the language you want to use for programming conditions and actions in your truth table by following these steps:

1. In the Truth Table Editor, select **Language** from the **Settings** menu.
2. Choose a language from the drop-down menu.

Entering Truth Table Conditions

Conditions are the starting point for specifying logical behavior in a truth table. You open the truth table `ttable` for editing in "Opening a Truth Table for Editing" on page 19-22. In this topic, you start programming the behavior of `ttable` by specifying its conditions.

You enter conditions in the **Condition** column of the **Condition Table**. For each condition that you enter, you can also enter an optional description in the **Description** column. Use the following procedure to enter the conditions of the truth table `ttable`:

1. Click anywhere in the **Condition Table** to select it.
2. Click the Append Row button  twice.

   Two rows are appended to the bottom of the **Condition Table**.
3. Click and drag the bar separating 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.

   The cell is highlighted and a flashing text cursor appears in the cell.
5. Enter the following text:

   ```
   x is equal to 1
   ```

Condition descriptions are optional, but appear as comments in the generated code for the truth table.

6 Press the **Tab** key to select the next cell on the right in the **Condition** column.

Tip You can use **Shift+Tab** to select the next cell on the left.

---

7 In the first row cell of the **Condition** column, enter the following text:

```
XEQ1:
```

This text is an optional label you can include with the condition. In the generated code for a truth table, the condition label becomes the name of a temporary data variable that stores the outcome of its condition. If you do not enter a label, a temporary variable appears.

Note Condition 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 (_).

---

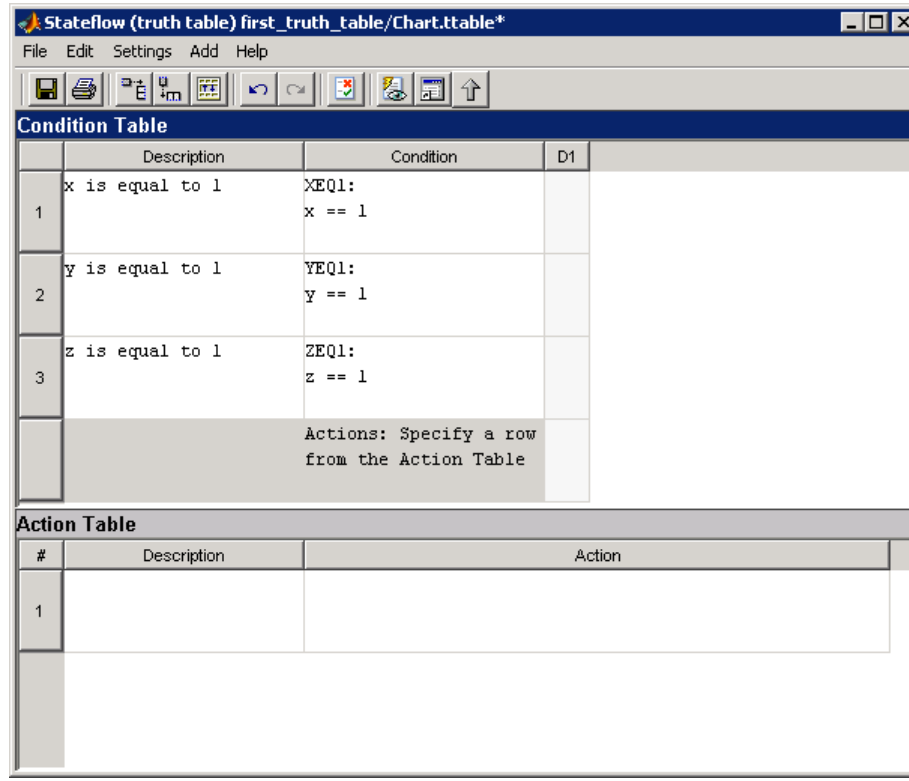
8 Press **Enter** and enter the following text:

```
x == 1
```

This text is the actual condition. Each condition you enter must evaluate to zero (false) or nonzero (true). You can use optional brackets in the condition (for example, `[x == 1]`) as you would in Stateflow action language.

You can use data passed to the truth table function through its arguments in truth table conditions. 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 Stateflow chart.

9 Repeat the preceding steps to enter the other two conditions.



Entering 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 allowed values for the condition outcomes in a decision are `T` (true), `F` (false), and - (true or false). In "Entering Truth Table Conditions" on page 19-23 you entered conditions for the truth table `ttable`. Continue by entering decisions in the decision columns with these steps:

1. Click anywhere in the **Condition Table** to make sure it is selected.
2. Click the Append Column toolbar button  three times to add three columns to the right end of the **Condition Table**.
3. Click the top cell in decision column **D1**.

The cell is highlighted and a flashing text cursor appears in the cell.

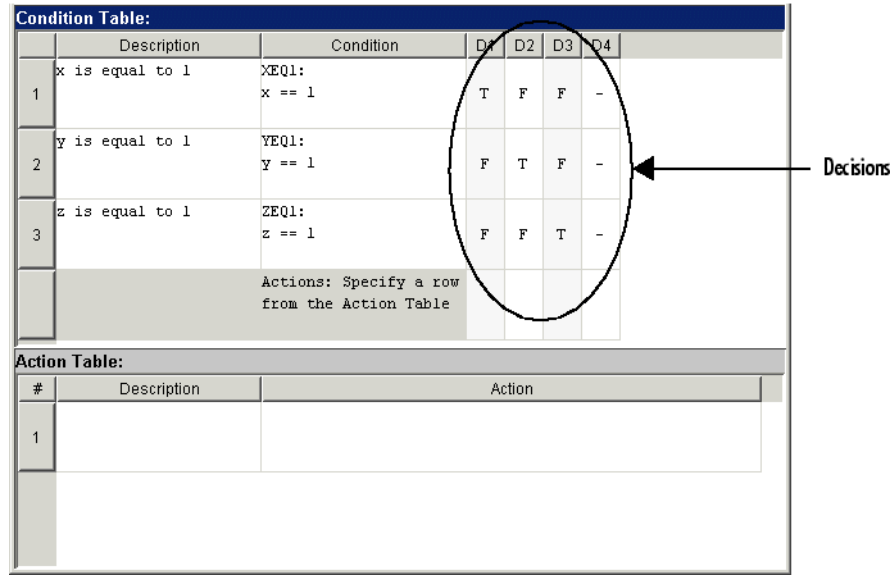
4 Press the space bar until a value of `T` appears.

Pressing the space bar toggles through the possible values of `F`, `-`, and `T`. You can also enter these characters directly. All other entries are rejected.

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 **Tab** and **Shift+Tab** to advance left or right in these cells.

6 Enter the remaining values for the decision columns, as shown here.



During execution of the truth table, decisions are tested 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, you must not depend on a particular 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 remaining 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, where - represents any outcome for the condition.

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

Note The default decision column must be the last column on the right in the **Condition Table**.

Entering Truth Table Actions

During execution of the truth table, decisions are tested in left to right order. When a decision is realized during execution of the truth table, the action in the **Action Table** specified in the **Actions** row for that decision column is executed and the truth table is exited.

In "Entering Truth Table Decisions" on page 19-25, you entered decisions in the Truth Table Editor. The next step is to enter the 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**.

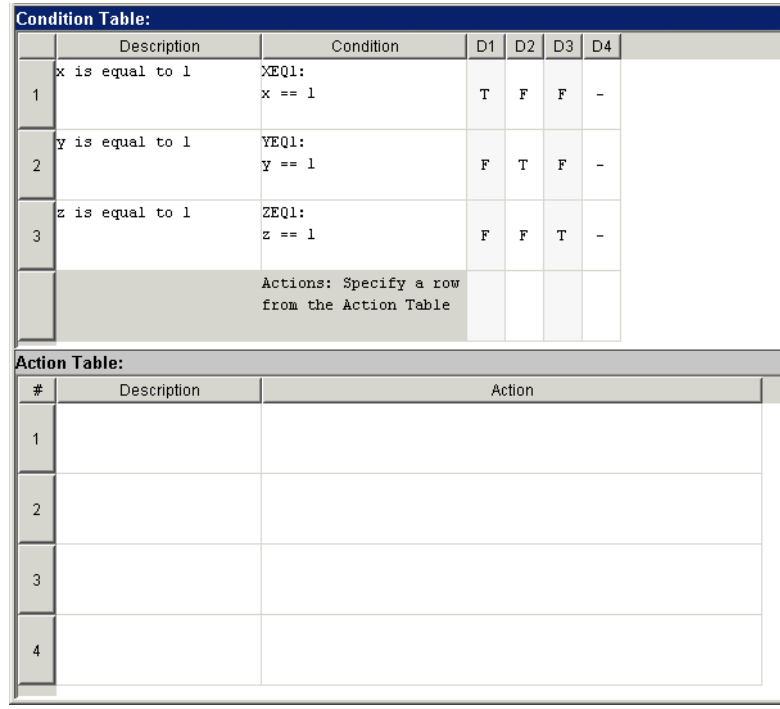
This section describes how to program truth table actions with these topics:

- "Setting Up the Action Table" on page 19-28 — Shows you how to set up the Action table in truth table `ttable`.

- "Programming Actions in Stateflow Classic Action Language" on page 19-29 — Provides sample code in Stateflow action language to program actions in `ttable`. Choose this section if you selected **Stateflow Classic** as the language for this truth table.
- "Programming Actions in Embedded MATLAB Action Language" on page 19-31 — Provides sample M-code to program actions in `ttable`. Choose this section if you selected **Embedded MATLAB** as the language for this truth table.

Setting Up the Action Table

1. Click anywhere in the **Action Table** to select it.
2. Click the Append Row toolbar button  three times to add three rows to the bottom of the **Action Table**.
3. Click and drag the bottom border of the Truth Table Editor window down to enlarge it and clearly show all rows of the **Action Table**, as shown.



4 Program the actions using the language you selected for the truth table.

If you selected...	Use this procedure...
Stateflow Classic	"Programming Actions in Stateflow Classic Action Language" on page 19-29
Embedded MATLAB	"Programming Actions in Embedded MATLAB Action Language" on page 19-31

Programming Actions in Stateflow Classic Action Language

Follow this procedure to program your actions in Stateflow action language:

1 Click the top cell in the **Description** column of the **Action Table.**

The cell is highlighted and a flashing text cursor appears in the cell.

2 Enter the following description:

```
set t to 1
```

Action descriptions are optional, but appear as comments in the generated code for the truth table.

3 Press **Tab** 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:

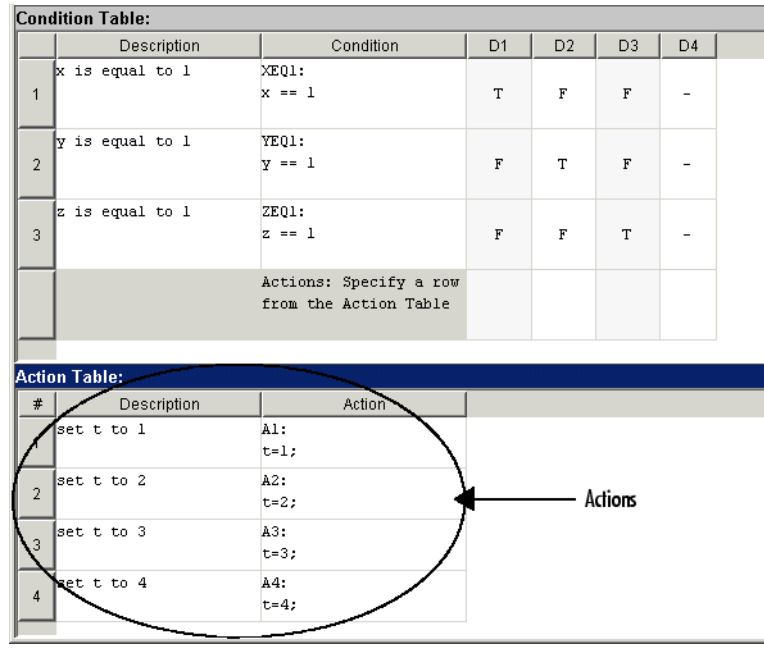
```
t=1;
```

You can use data passed to the truth table function through its arguments and return value in truth table actions. The preceding action, `t=1`, sets the value of the return value `t`. You can also specify actions with data defined for a parent object of the truth table, including the Stateflow 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 is echoed to the MATLAB Command Window when it is executed during simulation. Use this echoing option as a debugging tool.

---

6 Enter the remaining actions in the **Action Table**, as shown here.



Now you are ready to assign actions to decisions, as described in "Assigning Truth Table Actions to Decisions" on page 19-35.

Programming Actions in Embedded MATLAB Action Language

If you selected Embedded MATLAB action language, you can write M-code to program your actions. M-code allows you to add control flow logic and to call MATLAB functions directly. In the following procedure, you will program an action in the truth table `ttable`, using the following features of the Embedded MATLAB syntax:

- Persistent variables
- `if ... else ... end` control flows
- `for` loop
- Ability to call the MATLAB function `plot` directly

Follow these steps:

1 Click the top cell in the **Description** column of the **Action Table.**

The cell is highlighted and a flashing text cursor appears in the cell.

2 Enter this description:

```
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.
```

Action descriptions are optional, but appear as comments in the generated code for the truth table.

3 Press **Tab** 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;

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);
```

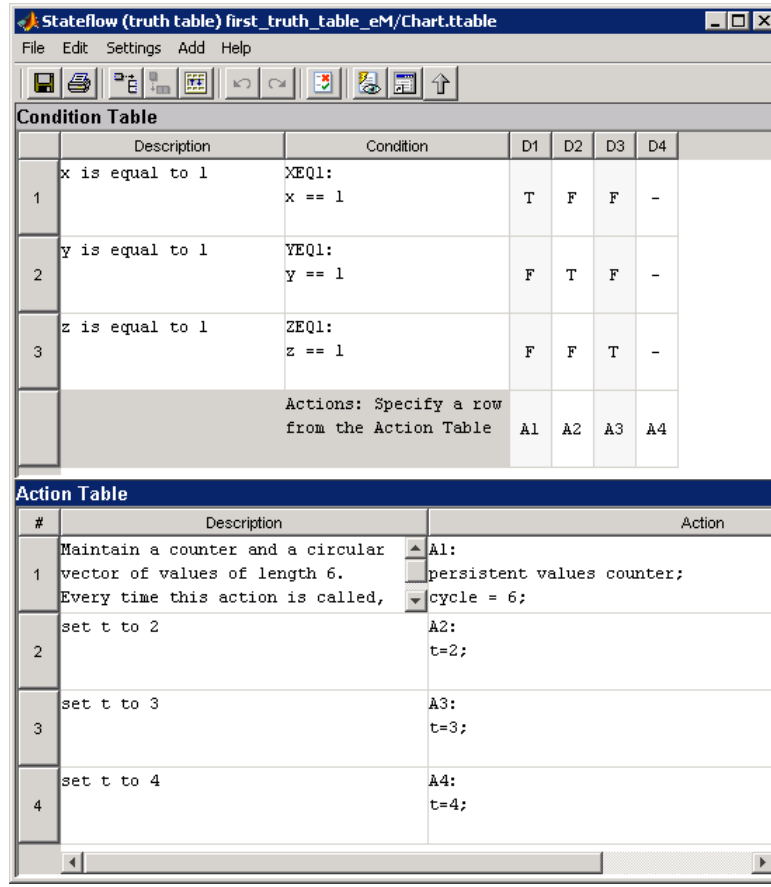
```
   for i = 1:cycle
      values(i) = i;
   end

   % For debugging purposes, call the MATLAB
   %  function "plot" to show values
   plot(values);
end

% Output t takes the next value in values vector
t = values( mod(counter, cycle) + 1);
```

You can use data passed to the truth table function through its arguments and return value in truth table actions. The preceding action sets the return value `t` 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 Stateflow 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.

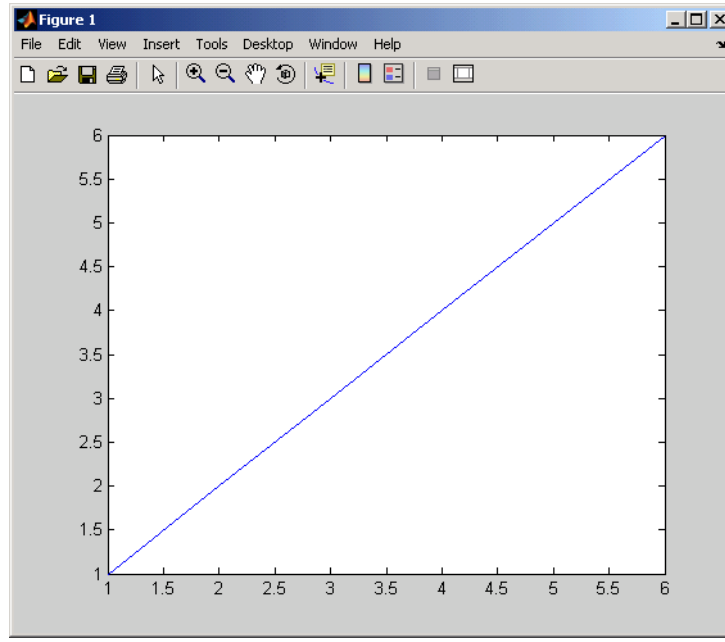
Note If you omit the semicolon at the end of an action, the result of the action echoes to the MATLAB Command Window when it is executed during simulation. Use this echoing option as a debugging tool.

---

6 Enter the remaining actions in the **Action Table**, as shown.



Note that if you simulate this model, condition `D1` will always be true, causing action `A1` to execute and display a plot of the content of the vector `values`, as follows:



Now you are ready to assign actions to decisions, as described in "Assigning Truth Table Actions to Decisions" on page 19-35.

Assigning 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 can use this association to determine what action to execute when a decision tests as true.

In this section, you will learn how to link actions to decisions.

Rules for Assigning Actions to Decisions

You can be creative in assigning actions. Here is a list of rules for assigning 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 be entered with the action in the **Action Table**.

- You must specify at least one action for each decision.

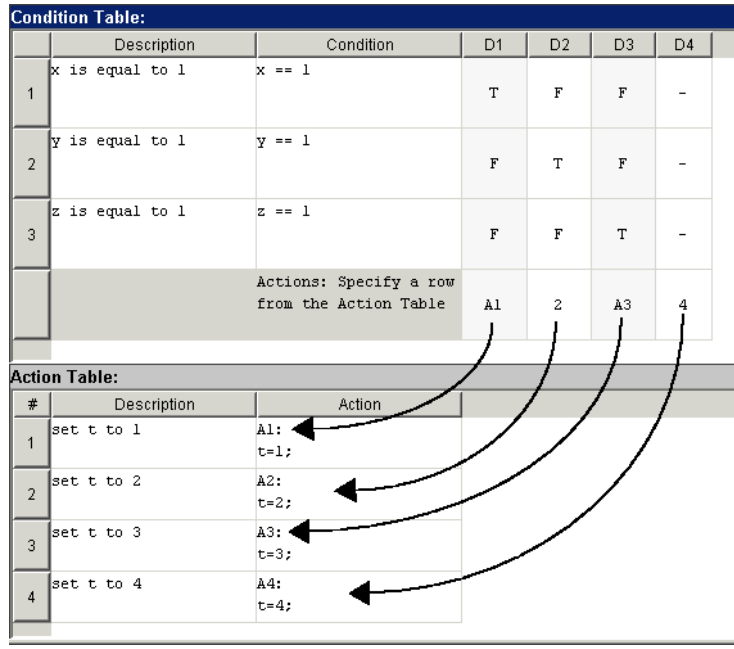
  Actions for decisions are not optional. Each decision must have at least one action specifier that points to an action in the **Action Table**. If you want to specify no action for a decision, specify a row that contains no action statements.

- You can specify multiple actions for a decision with multiple specifiers separated by a comma.

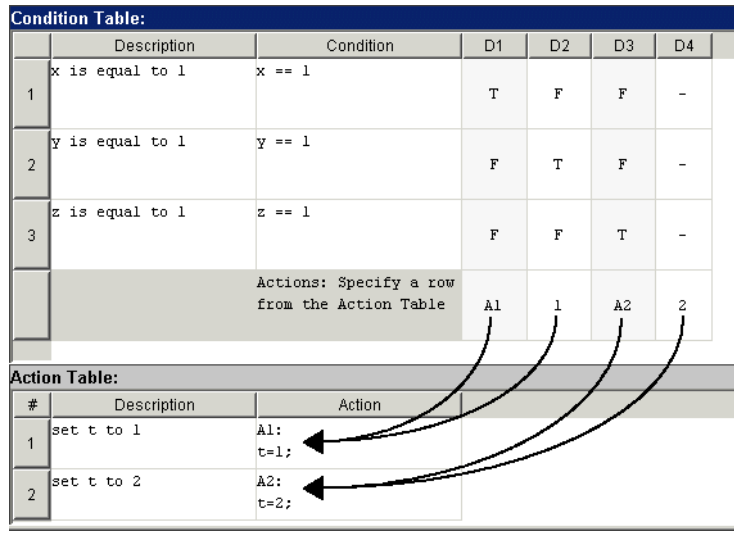
  For example, for the decision column **D1** you can specify A1, A2, A3 or 1, 2, 3 to execute the first three actions if decision **D1** is true.

- You can mix row number and label action specifiers interchangeably in any order.

  The following example uses both row and label action specifiers.

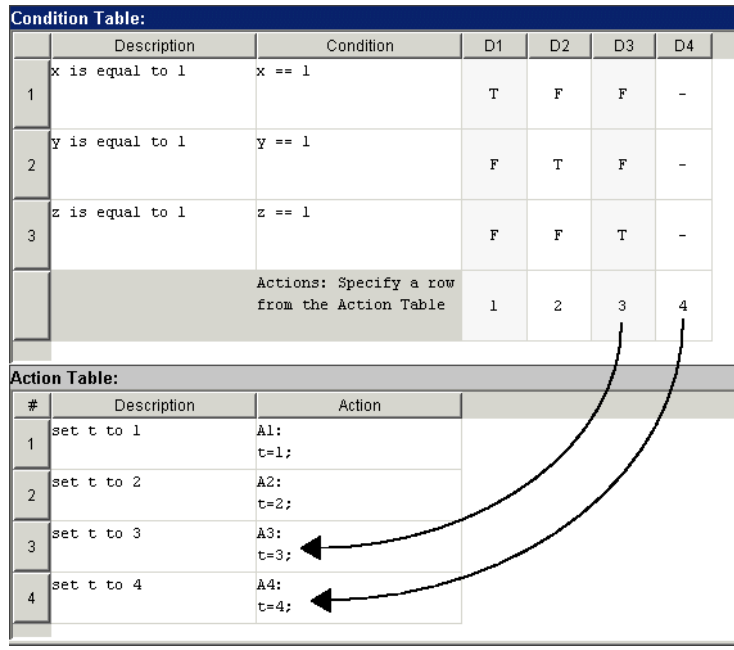


- You can specify the same action for more than one decision, as shown.

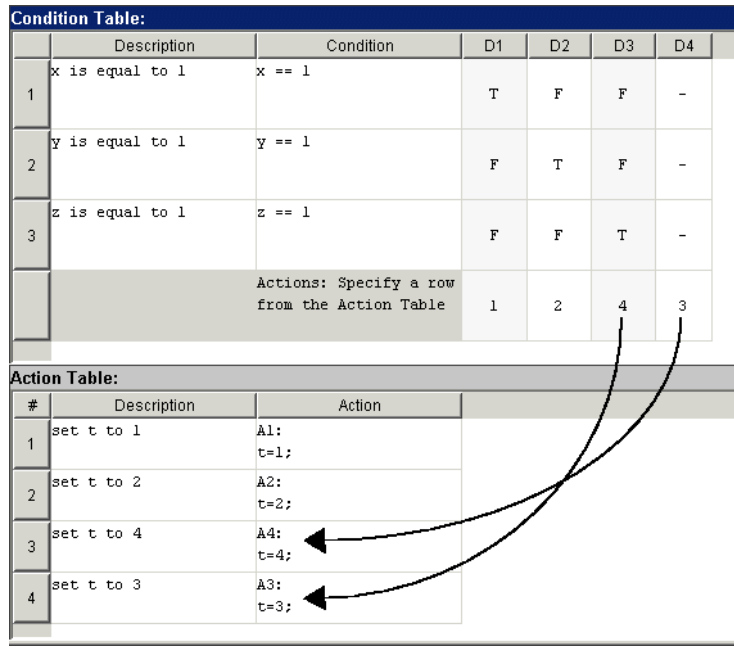


- Row number action specifiers in the **Actions** row of the **Condition Table** automatically adjust to changes in the row order of the **Actions Table**.

  In the following example, decisions **D3** and **D4** are assigned the actions in rows 3 and 4 of the **Action Table**, respectively.



Select row 4 in the **Action Table** and select the Move Row Up tool  to reverse rows 3 and 4, and notice the change in the action specifiers for columns **D3** and **D4**, as shown.



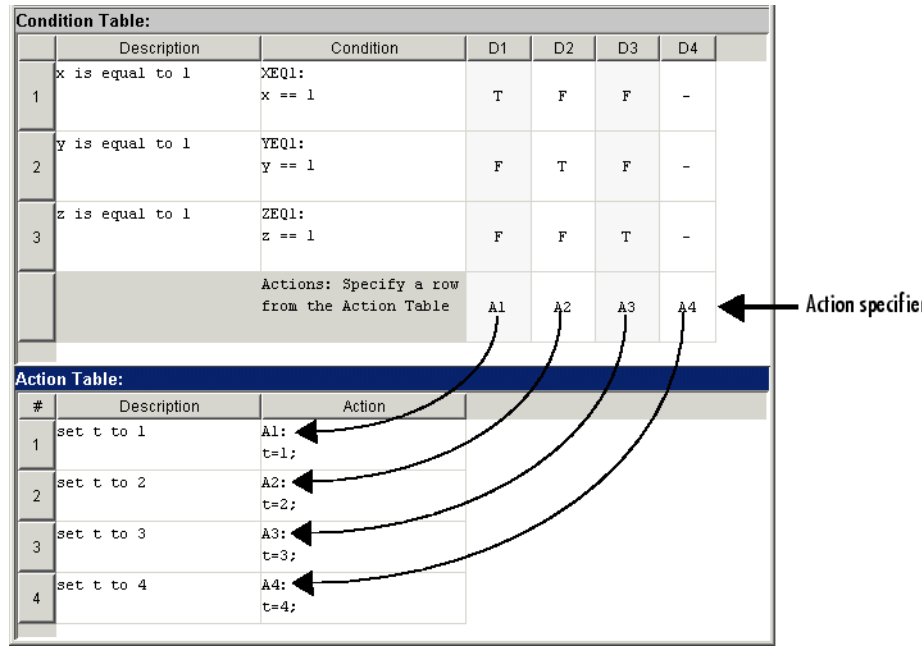
How to Assign Actions to Decisions

This section describes how to assign actions to decisions in the example 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**. Decision **D1** is assigned the action `t=1`, decision **D2** is assigned the action `t=2`, and so on. 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**, that links the action labeled `A1` in the **Action Table** to decision **D1**.

3. Enter the action specifiers for the remaining decision columns as shown in the following:



Now you are ready to perform the final step in programming a truth table, “Adding Initial and Final Actions” on page 19-40.

Adding Initial and Final Actions

In addition to the actions for decisions, you can add initial and final actions to the truth table function. Initial actions specify an action that executes before any decisions are tested. Final actions specify an action that executes as the last action before the truth table is exited. 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 to display diagnostic messages in the MATLAB Command Window before and after the execution of the truth table `ttable`:

1. In the Truth Table Editor for the truth table `ttable`, right-click row 1 of the **Action Table**.

   A context menu appears.

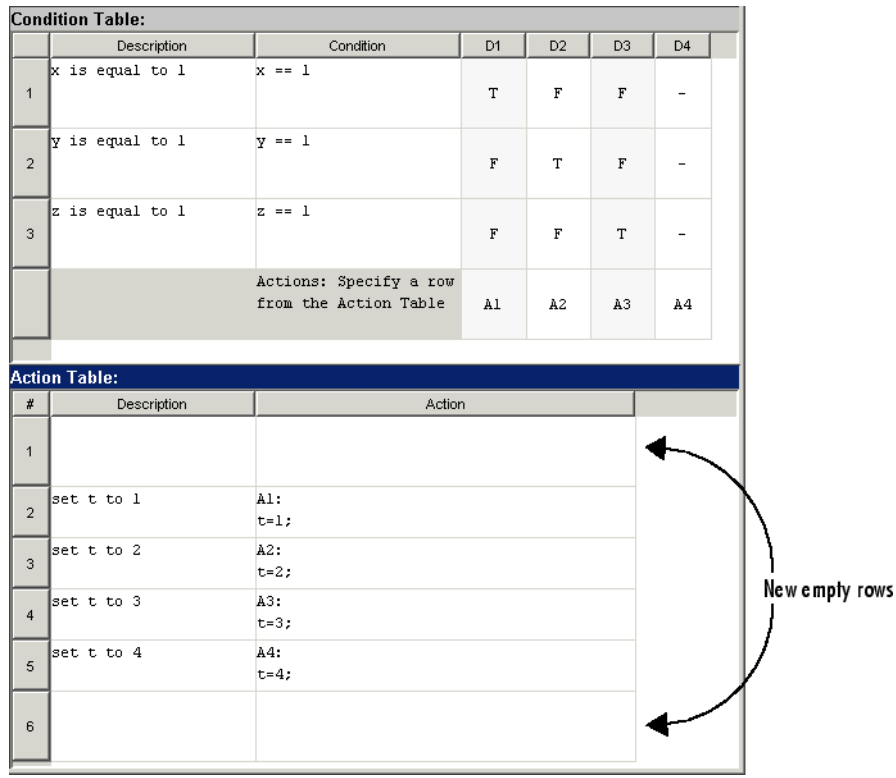
2 From the context menu, select **Insert Row**.

A blank row is inserted at the beginning of the **Action Table**.

3 Click the Append Row tool  .

A blank row is appended to the bottom of the **Action Table**.

4 Click and drag the bottom border of the Truth Table Editor to expose all six rows of the **Action Table**, as shown.



5 Add the initial action in row 1 as follows:

Truth Table Type	Description	Action
Stateflow Classic	Initial action:Display message	`INIT:ml.disp('truth table ttable entered');`
Embedded MATLAB	Initial action:Display message	`INIT:disp('truth table ttable entered');`

6 Add the final action in row 6 as follows:

Truth Table Type	Description	Action
Stateflow Classic	Final action:Display message	`FINAL:ml.disp('truth table ttable exited');`
Embedded MATLAB	Final action:Display message	`FINAL:disp('truth table ttable exited');`

Even though the initial and final actions for the preceding truth table example are shown in the first and last rows of the **Action Table**, you can enter these actions in any row. You can also explicitly assign the initial and final actions to decisions by using the action specifier `INIT` or `FINAL` in the **Actions** row of the **Condition Table**.

Debugging a Truth Table

In this section...

Checking Truth Tables for Errors

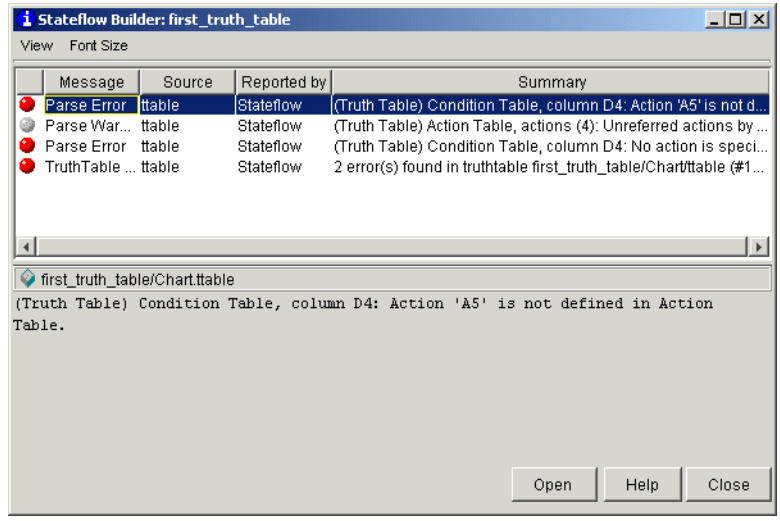
Once you completely specify your truth tables, you must begin the process of debugging them. The first step is to run diagnostics to check truth tables for syntax errors including overspecification and underspecification, as described in "Correcting Overspecified and Underspecified Truth Tables" on page 19-53.

To check for syntax errors, follow these steps:

1. Double-click the truth table to open its editor.

2. In the Truth Table Editor toolbar, click the **Run Diagnostics** button  .

   If there are no errors or warnings, the Builder window appears and reports a message of success. If errors are found, the Builder window lists them. For example, if you change the action for decision column **D4** to an action that does not exist, the Builder window appears.



Each detected error appears with a red button, and each warning appears with a gray button. The first error message appears highlighted in the top pane, and the diagnostic message appears in the bottom pane.

Truth table diagnostics run automatically when you start simulation of the model with a new or modified truth table. If no errors are found, the Builder window does not appear and simulation commences immediately.

Debugging a Truth Table During Simulation

There are several ways to debug truth tables during simulation:

Method	Use With	How To Do It
Use Stateflow debugging tools to step through each condition and action, and monitor data values during simulation.	Stateflow Classic truth table and Embedded MATLAB truth table	See "Using Stateflow Debugging Tools" on page 19-45.
Use Embedded MATLAB debugging tools to step through Embedded MATLAB code generated by the truth table.	Embedded MATLAB truth table only	See "Using Embedded MATLAB Debugging Tools" on page 19-52.

Using Stateflow Debugging Tools

When you use Stateflow debugging tools to debug truth tables, you must perform these tasks:

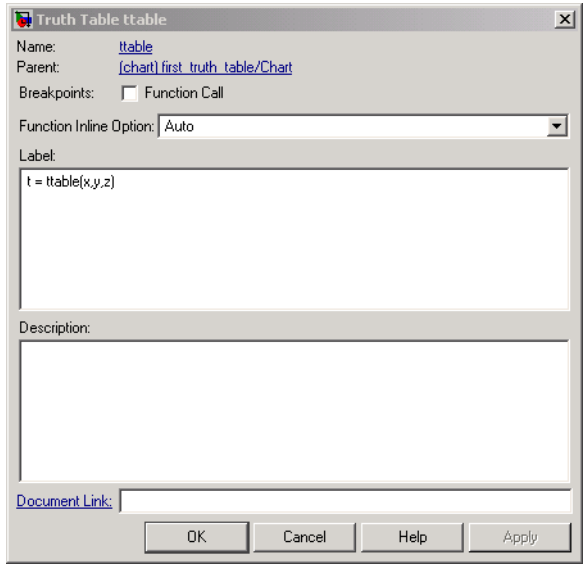
1. Specify a breakpoint for the call to the truth table.
2. Step through the conditions and actions.

Specifying a Breakpoint for the Call to a Truth Table. Before you debug the truth table during simulation, you must set a breakpoint for the truth table in its properties dialog box. This breakpoint pauses execution during simulation so that you can debug each execution step of a truth table using the Stateflow Debugger.

Follow these steps:

1. In the Stateflow Editor, right-click the truth table.
2. In the context menu, select **Properties**.

   The Truth Table properties dialog box appears.



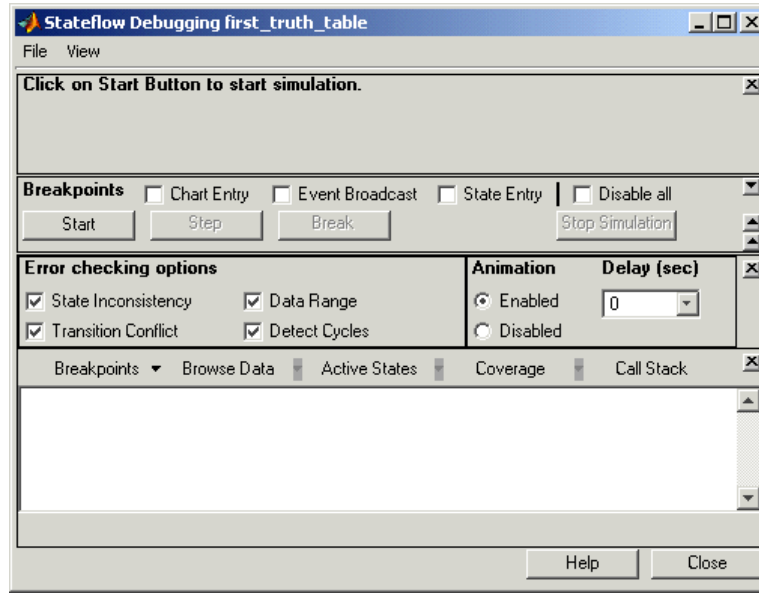
3 For **Breakpoints**, select **Function Call**.

This option sets a breakpoint to occur when this truth table function is called in the Stateflow chart during simulation.

4 Select **OK** to save settings and close the Truth Table properties dialog box.

Stepping Through Conditions and Actions of a Truth Table. After setting a breakpoint for the truth table function call, you can step through the conditions and actions by following these steps:

1 Select the Debug button  in the Stateflow Editor toolbar to start the Stateflow Debugging window as shown.



2 From the Stateflow Debugging window, select the **Start** button to begin simulation of your model.

When you simulate your model, the Stateflow Debugger checks automatically for syntactical errors if the truth tables have changed since the last simulation. If you receive errors or warnings, make corrections before you try to simulate again.

If there are no syntactical errors in the truth table, a simulation application builds and the simulation of your model begins.

3 Wait until the breakpoint for the call to the truth table is reached.

When this breakpoint is encountered, the truth table `ttable` appears and the **Start** button in the Stateflow Debugger changes to the **Continue** button.

4 In the Stateflow Debugging window, click the **Step** button three times to advance simulation through the call to the truth table.

The `INIT` action of the truth table is highlighted prior to its execution.

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');

5 Click **Step** to execute the `INIT` action and advance truth table execution to the first condition, as shown.

Condition Table:

	Description	Condition	D1	D2	D3	D4
1	x is equal to 1	x == 1	T	F	F	-
2	y is equal to 1	y == 1	F	T	F	-
3	z is equal to 1	z == 1	F	F	T	-

6 Click **Step** to evaluate the first condition and advance truth table execution to the second condition.

Condition Table:

	Description	Condition	D1	D2	D3	D4
1	x is equal to 1	x == 1	T	F	F	-
2	y is equal to 1	y == 1	F	T	F	-
3	z is equal to 1	z == 1	F	F	T	-

7 Click **Step** to evaluate the second condition and advance truth table execution to the third condition.

Condition Table:

	Description	Condition	D1	D2	D3	D4
1	x is equal to 1	x == 1	T	F	F	-
2	y is equal to 1	y == 1	F	T	F	-
3	z is equal to 1	z == 1	F	F	T	-

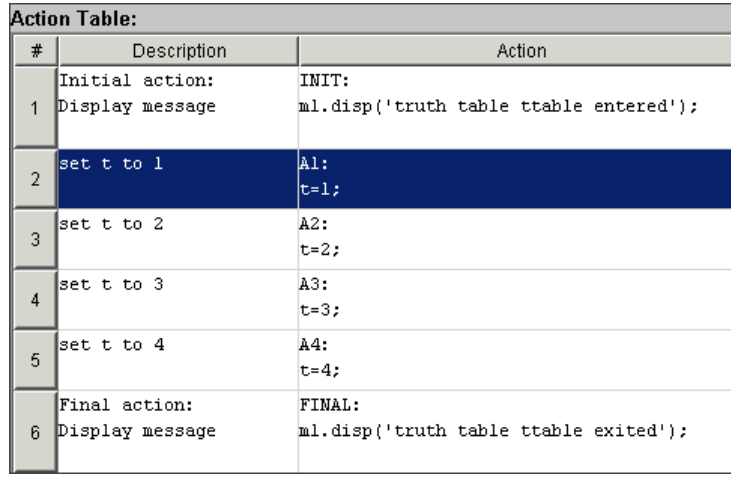
8 Click **Step** to evaluate the third condition and advance truth table execution to the first decision.

Condition Table:

	Description	Condition	D1	D2	D3	D4
1	x is equal to 1	x == 1	T	F	F	-
2	y is equal to 1	y == 1	F	T	F	-
3	z is equal to 1	z == 1	F	F	T	-
		Actions: Specify a row from the Action Table	A1	A2	A3	A4

9 Click **Step** twice.

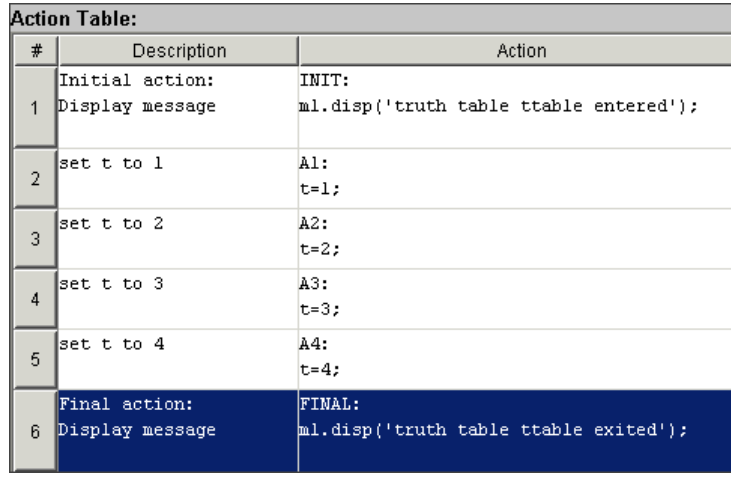
Because the first decision is true, truth table execution advances to its action, which is labeled `A1`.



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');

10 Click **Step** three times to execute action `A1` and advance to the `FINAL` action.

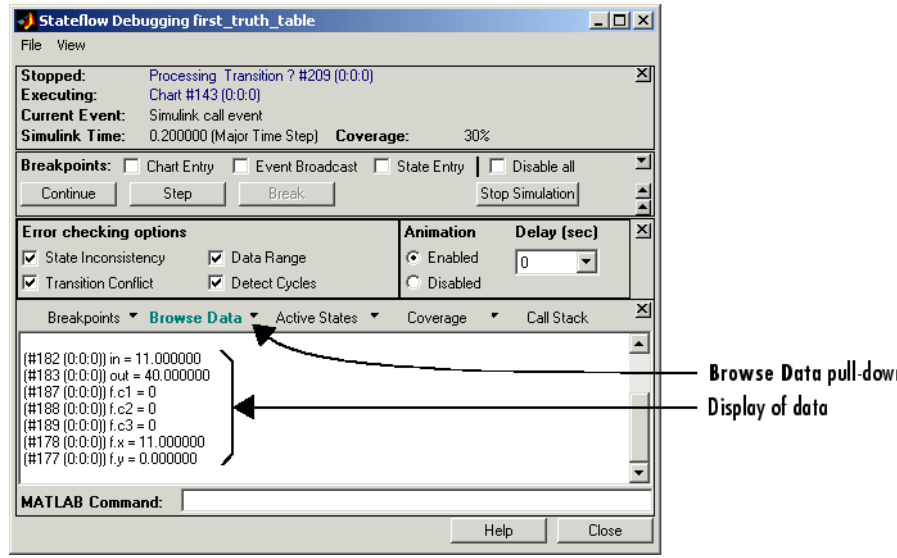


Action Table:

#	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');

11 In the Stateflow Debugging window, from the **Browse Data** pull-down, select **All Data (Current Chart)**.

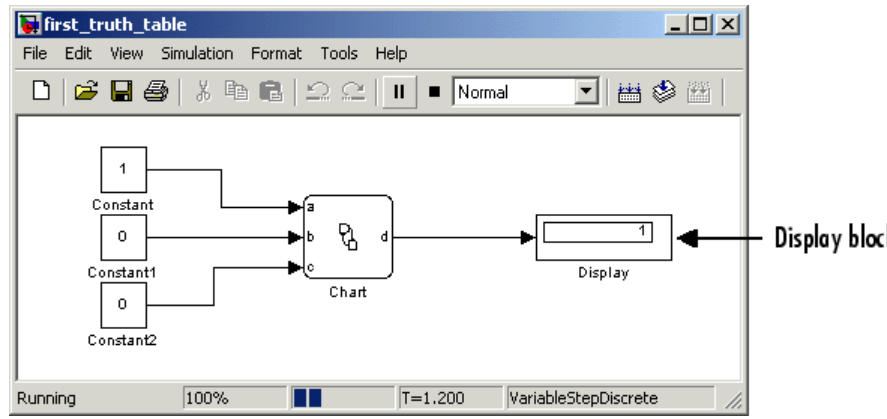
A continuously updated display appears in the bottom pane of the Stateflow Debugging window, as shown.



You can use this display to monitor Stateflow data during simulation.

12 In the Stateflow Debugging window, click **Step**.

This step executes the final action and exits the truth table. The Display block in the Simulink window should now display the number 1, as shown.



13 Change the values of the Constant blocks and continue stepping through the simulation.

For example, you might want to set the Constant1 block to 1 (sets `b` to 1) and the other Constant blocks to 0 (sets `a` and `c` to 0). Or you might want to set all the Constant blocks to 0. To enter a new value for a Constant block, double-click it. In the resulting Block Parameters dialog box, enter the new value in the **Constant value** field.

Using Embedded MATLAB Debugging Tools

Embedded MATLAB truth tables generate content as Embedded MATLAB code, a format that offers advantages for debugging. You can set breakpoints on any line of generated code (whereas you cannot set breakpoints directly on a truth table). You can debug code generated by Embedded MATLAB truth tables the same way you debug an Embedded MATLAB function, as described in "Debugging an Embedded MATLAB Function Block".

For more information about how to generate content for truth tables, see "How Stateflow Software Implements Truth Tables" on page 19-60.

Correcting Overspecified and Underspecified Truth Tables

In this section...

Defining an Overspecified Truth Table

An overspecified truth table contains at least one decision that will never be executed because it is already specified in a previous decision in the **Condition Table**. This example shows the **Condition Table** of an overspecified truth table.

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

The decision in column **D3** (`-TT`) specifies the decisions `FTT` and `TTT`. These decisions have already been specified by decisions **D1** (`FTT`) and **D2** (`TTT` and `TFT`). Therefore, column **D3** is an overspecification.

This example shows the **Condition Table** of a truth table that appears to be overspecified, but is not.

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

In this case, the decision **D4** specifies two decisions (TTT and FTT). FTT is specified by decision **D1**, but TTT is not specified in a previous decision column. Therefore, this **Condition Table** is not overspecified.

Defining an Underspecified Truth Table

An underspecified truth table lacks one or more possible decisions that might require an action to avoid undefined behavior. This example shows the **Condition Table** of an underspecified truth table:

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

Complete coverage of the conditions in the preceding truth table requires a **Condition Table** with every possible decision, like this example.

Condition Table:

	Description	Condition	D1	D2	D3	D4	D5	D6	D7	D8
1	Condition C1	C1: x == 0	T	T	T	F	F	T	F	F
2	Condition C2	C2: y == 0	T	T	F	T	F	F	T	F
3	Condition C3	C3: z == 0	T	F	T	T	F	F	F	T
		Actions: Specify a row from the Action Table	A1	A2	A3	A4	A5	A6	A7	A8

A possible workaround is to specify an action for all other possible decisions through a default decision, as in this example.

Condition Table:

	Description	Condition	D1	D2	D3	D4
1	Condition C1	C1: x == 0	T	T	T	-
2	Condition C2	C2: y == 0	T	T	F	-
3	Condition C3	C3: z == 0	T	F	T	-
		Actions: Specify a row from the Action Table	A1	A2	A3	DA

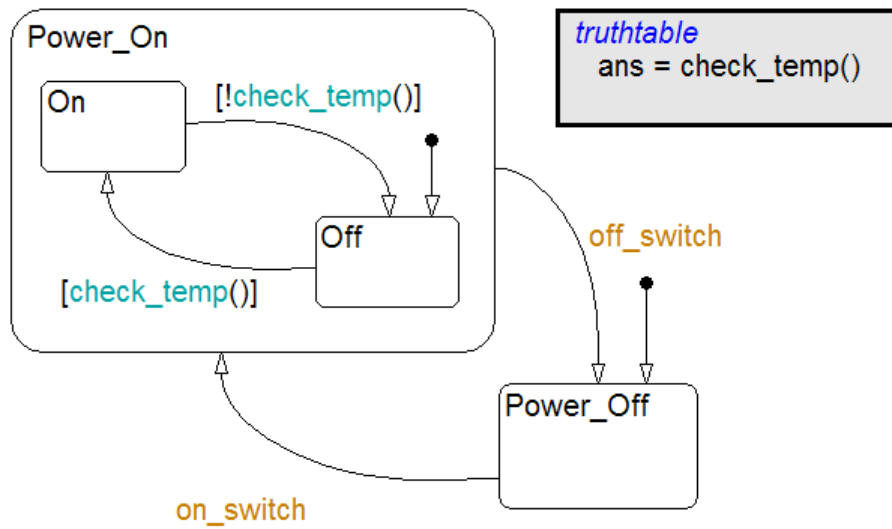
The last decision column is the default decision for the truth table. The default decision covers any remaining decisions not tested for in the preceding decision columns. See “The Default Decision Column” on page 19-27 for an example and more complete description of the default decision column for a **Condition Table**.

Model Coverage for Truth Tables

Stateflow software reports model coverages for the decisions made by the objects in a Stateflow chart during model simulation. The model coverage report includes coverage for the decisions made by truth table functions, as follows:

Type of Truth Table	Type of Coverage
Stateflow Classic	Coverage reports generated for conditions only.
Embedded MATLAB	Coverage reports generated for conditions and actions because you can use the Embedded MATLAB action language to specify decision points in actions using control flow constructs, such as loops and switch statements.

This section examines model coverage for an example Stateflow Classic truth table, `check_temp`, which is tested during simulation in this Stateflow chart.



The contents of the `check_temp` truth table are shown here.

Condition Table:

	Description	Condition	D1	D2
1	Check ambient temp for cooling time	CAMTSC: amb_temp > min_amb_temp	T	-
2	Check attic temp setting for cooling	attic_temp > attic_temp_set	T	-
3	Check ambient temp for cooling capable	amb_temp < attic_temp - 5	T	-
		Actions: Specify a row from the Action Table	ON	OFF

Action Table:

#	Description	Action
1	Stay on or turn on	ON: ans = 1;
2	Stay off or turn off	OFF: ans = 0;

You generate model coverage reports for a model during simulation. You first specify the creation of the reports in the Simulink model window and then simulate the model. When simulation ends, a model coverage report appears in a browser window. See “Making Model Coverage Reports” on page 23-52 for information on how to set up a model coverage report.

Note The Model Coverage tool requires a Simulink Verification and Validation software license.

The following is the part of a model coverage report that reports on the `check_temp` truth table.

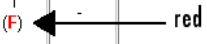
4. Function "check_temp"

Parent: attic_fan/On_Off

Metric	Coverage (this object)	Coverage (inc. descendants)
Cyclomatic Complexity	0	3
Decision (D1)	NA	100% (2/2) decision outcomes
Condition (C1)	NA	83% (5/6) condition outcomes
MCDC (C1)	NA	67% (2/3) conditions reversed the outcome

Predicate table analysis (missing values are in parentheses)

Check ambient temp for cooling time	CAMTSC: amb_temp > min_amb_temp	T (ok)	-
Check attic temp setting for cooling	attic_temp > attic_temp_set	T (ok)	-
Check ambient temp for cooling capable	amb_temp < attic_temp - 5	T (F)	-
	Actions	ON (ok)	OFF



Coverage for the truth table function in the **Coverage (this object)** column shows no valid coverage values. The reason for this is that the container object for the truth table function, the chart, makes no decision on whether to execute the `check_temp` truth table or not.

Stateflow software implements a Stateflow Classic truth table by generating a graphical function for it. The decision logic of the truth table is implemented internally in the transitions of the graphical function generated for the truth table. See “How Stateflow Software Implements Truth Tables” on page 19-60 for a description of the generated graphical function for a truth table.

The transitions of the generated graphical function for a truth table contain the decisions and conditions of the truth table. Coverage for the descendants in the **Coverage (inc. descendants)** column includes coverage for these conditions and decisions, which are tested when the truth table function is called.

In the case of the `check_temp` truth table, the only decision covered in the model coverage report is the **D1** decision. There is no model coverage for the default decision, **D2**.

Note All logic that leads to taking a default decision is based on a false outcome for all preceding decisions. This means that no logic is required for the default decision, which receives no model coverage.

Coverages for the **D1** decision and its individual conditions in the `check_temp` truth table function are as follows:

- Decision coverage for the **D1** decision is 100% because this decision was tested both true and false during simulation.
- Condition coverage for the three conditions of the **D1** decision indicate that 5 of 6 possible T/F values were tested.

  Because each condition can have an outcome value of `T` or `F`, three conditions can have 6 possible values. During simulation, only 5 of 6 were tested. The **D1** decision coverage column shows that the last condition received partial condition coverage by not evaluating to false (F) during simulation. The missing occurrence of the false (F) condition outcome is indicated by the appearance of a red `F` character.
- MCDC coverage looks for decision reversals that occur because one condition outcome changes from `T` to `F` or from `F` to `T`. The **D1** decision reverses when any of the conditions changes from `T` to `F`. This means that the outcomes `FTT`, `TFT`, and `TTF` reverse this decision by a change in the value of one condition.

  The top two conditions for the D1 decision tested both true (`T`) and false (`F`) with a resulting reversal in the decision from true (`T`) to false (`F`). However, the bottom condition tested only a true (`T`) outcome but no false (`F`) outcome (appearance of red `F` character). Therefore, two of a possible three reversals were observed and coverage is 2/3 = 67%.
- The `(ok)` next to the `ON` action label indicates that its decision realized both true (`T`) and false (`F`) during simulation. Because the default decision is based on no logic of its own, it does not receive the `(ok)` mark.

How Stateflow Software Implements Truth Tables

In this section...

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
Stateflow Classic	Graphical function
Embedded MATLAB	Embedded MATLAB code

Viewing 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 to open its editor.
3. Select the **View Generated Content**  button.

   Stateflow Classic truth tables display generated content as described in "How Stateflow Software Generates Graphical Functions for Truth Tables" on page 19-61.

How Stateflow Software Generates Graphical Functions for Truth Tables

This section describes how Stateflow software translates the logic of a Stateflow Classic truth table into a graphical function.

In this example, a Stateflow Classic truth table has three conditions, four decisions and actions, and initial and final actions.

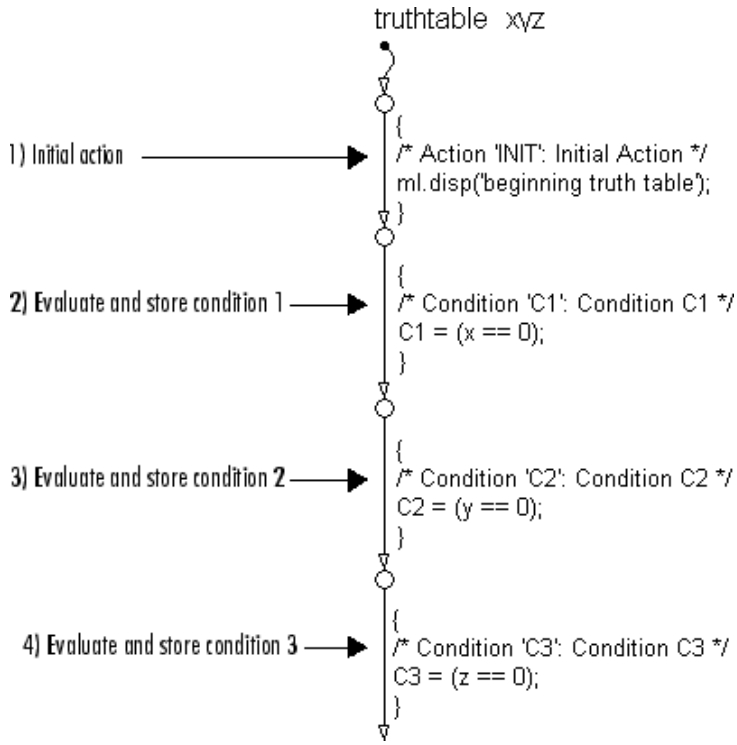
Condition Table

	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 graph is as follows, where the numbered steps show the order of execution.

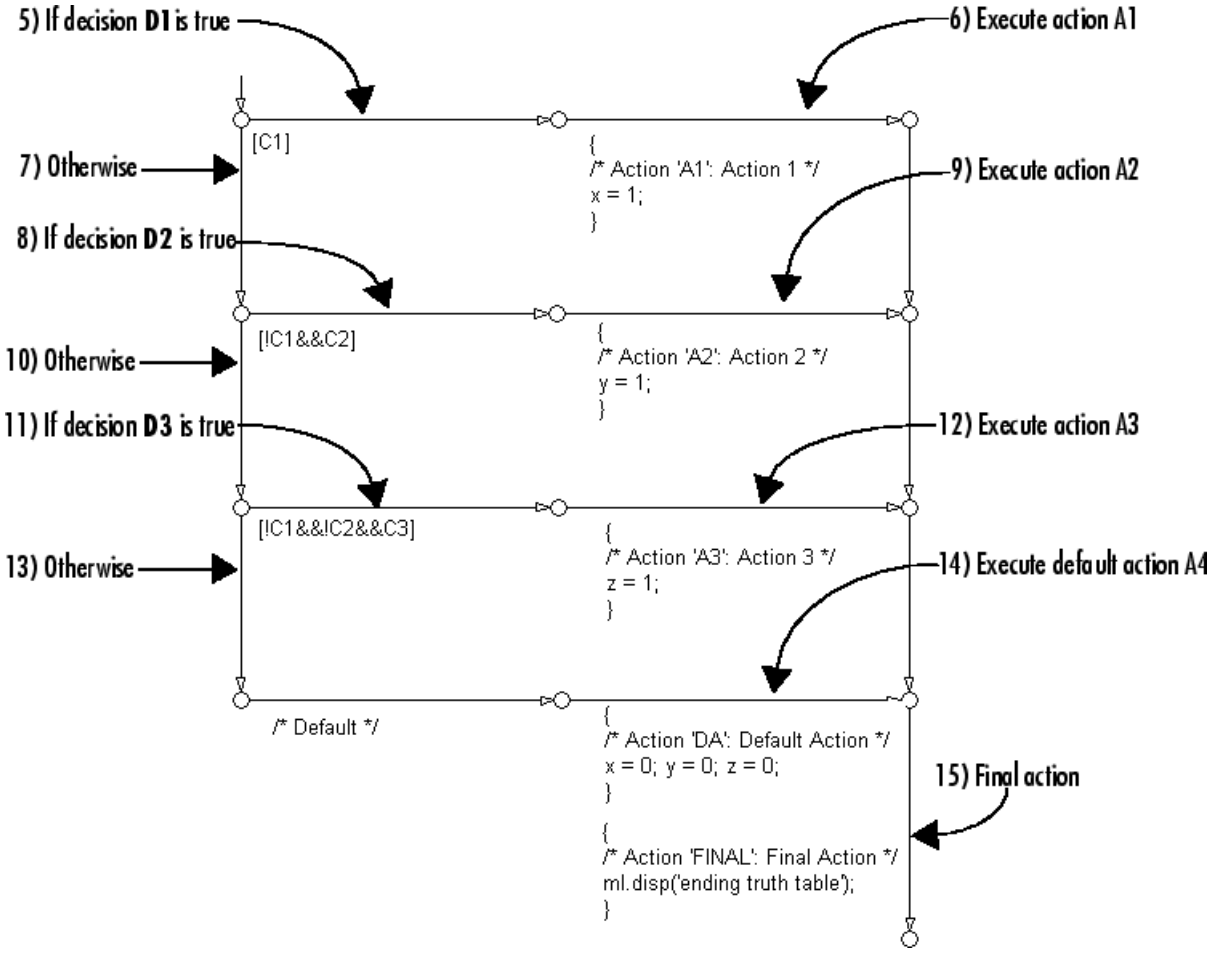


The top half of the flow graph does the following:

- 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 graph is as follows, where the numbered steps show the order of execution for each condition and action.



In the bottom half of the flow graph, the stored values for the conditions are used to make decisions on which 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 is specified as a condition action that leads to the `FINAL` action and termination of the flow graph

- 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 specified decision is matched, the action specified for that decision is executed as a condition action of its transition segment. Once the action is performed, the flow graph 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 implies that no data dependencies are possible between different decisions.

How Stateflow Software Generates Embedded MATLAB Code for Truth Tables

Stateflow software generates the content of Embedded MATLAB truth tables as Embedded MATLAB code that represents each action as a *nested* function inside the main truth table function.

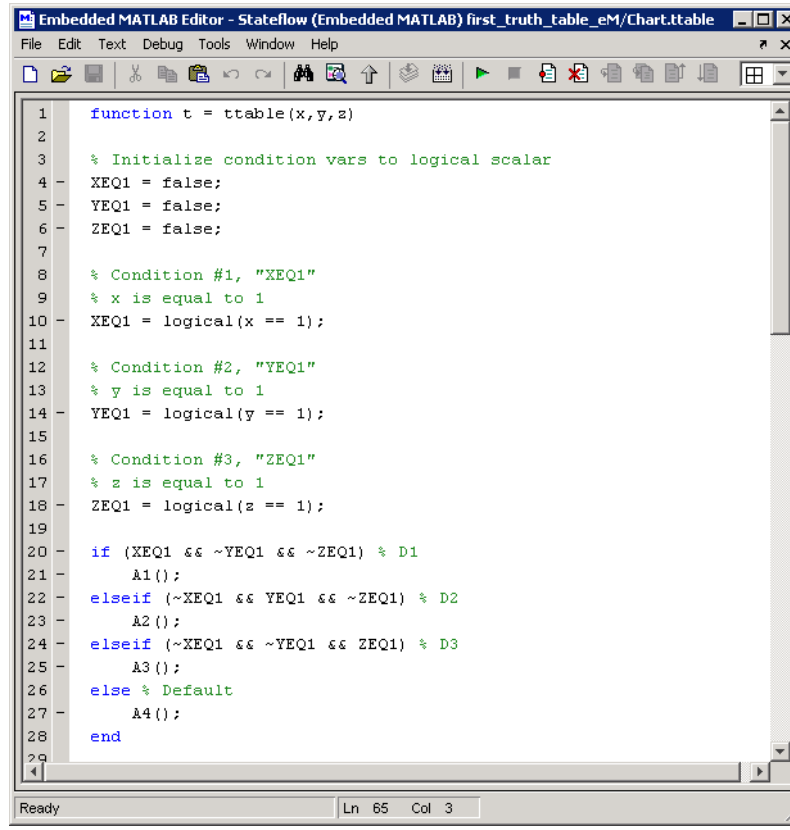
Nested functions offer several advantages over subfunctions:

- Nested functions are independent of each other. Therefore, variables are local to each function and not subject to naming conflicts.
- Nested functions can access all data from the main truth table function.

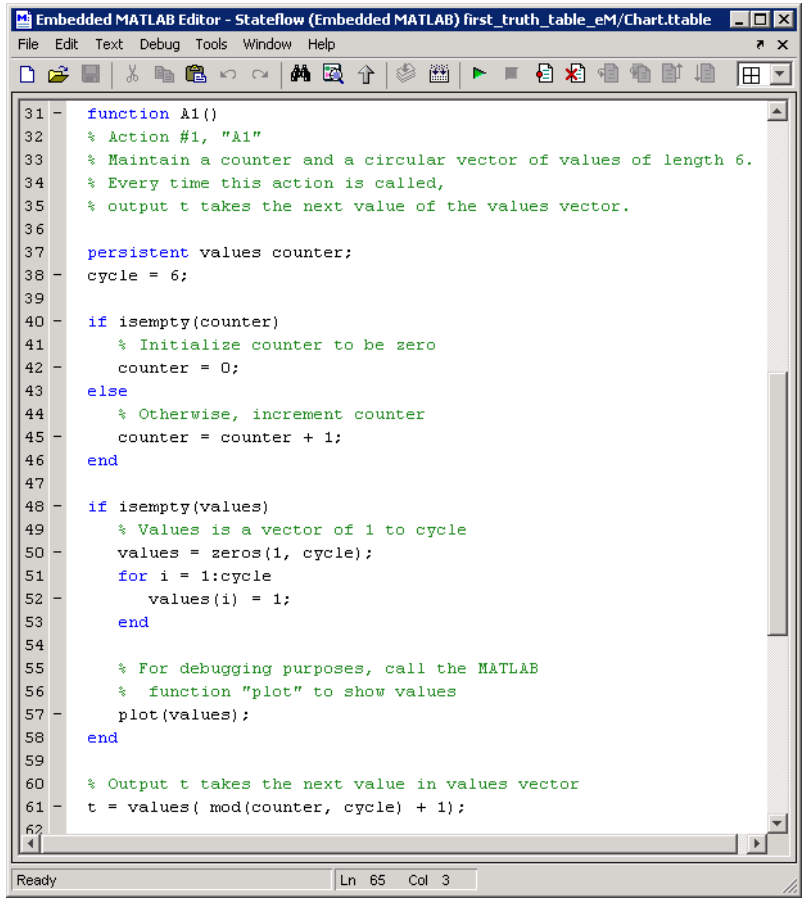
The generated content appears in an Embedded MATLAB Editor, which provides tools for simulation and debugging, as described in "Debugging an Embedded MATLAB Function Block".

Here is the generated content for the Embedded MATLAB truth table described in "Programming Actions in Embedded MATLAB Action Language" on page 19-31:

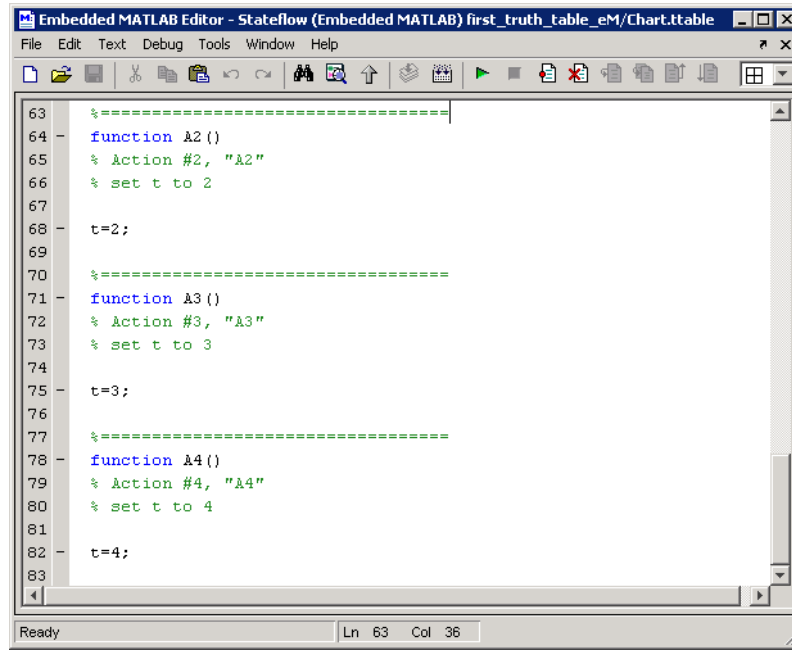
- Main truth table function


Embedded MATLAB Editor - Stateflow (Embedded MATLAB) first_truth_table_eM/Chart.ttable
File Edit Text Debug Tools Window Help
1 function t = ttable(x,y,z)
2
3 % Initialize condition vars to logical scalar
4 XEQ1 = false;
5 YEQ1 = false;
6 ZEQ1 = false;
7
8 % Condition #1, "XEQ1"
9 % x is equal to 1
10 XEQ1 = logical(x == 1);
11
12 % Condition #2, "YEQ1"
13 % y is equal to 1
14 YEQ1 = logical(y == 1);
15
16 % Condition #3, "ZEQ1"
17 % z is equal to 1
18 ZEQ1 = logical(z == 1);
19
20 if (XEQ1 && ~YEQ1 && ~ZEQ1) % D1
21 A1();
22 elseif (~XEQ1 && YEQ1 && ~ZEQ1) % D2
23 A2();
24 elseif (~XEQ1 && ~YEQ1 && ZEQ1) % D3
25 A3();
26 else % Default
27 A4();
28 end
Ready
Ln 65 Col 3

- Action A1



- Actions A2, A3, and A4


Embedded MATLAB Editor - Stateflow (Embedded MATLAB) first_truth_table_eM/Chart.ttable
File Edit Text Debug Tools Window Help
63 %==================================
64 function A2()
65 % Action #2, "A2"
66 % set t to 2
67
68 t=2;
69
70 %==================================
71 function A3()
72 % Action #3, "A3"
73 % set t to 3
74
75 t=3;
76
77 %==================================
78 function A4()
79 % Action #4, "A4"
80 % set t to 4
81
82 t=4;
83
Ready
Ln 63 Col 36

Truth Table Editor Operations

In this section...

Truth Table Editor Reference

This section describes the operations you can perform in the Truth Table Editor.

Adding or Modifying Stateflow Data

	**Edit Data/Ports** lets you add or modify Stateflow data with the Model Explorer.

Appending 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.

Compacting the Table

	**Compact Table** removes the empty rows and columns of the selected table.

Deleting Text, Rows, and Columns

To delete the contents of a cell:

1 Right-click the cell.

2 From the context menu, select **Delete Cell**.

To delete an entire row or column:

1 Right-click the row or column header.

2 From the context menu, select **Delete Row** or **Delete Column**.

You can also click the row or column header to select the entire row or column and press the **Delete** key.

Diagnosing the Truth Table

	**Run Diagnostics** checks the truth table for syntax errors. See "Debugging a Truth Table" on page 19-43.

Viewing Generated Content

	**View Generated Content** displays the code generated for the truth table. Stateflow Classic truth tables generate graphical functions; Embedded MATLAB truth tables generate Embedded MATLAB code. See "How Stateflow Software Implements Truth Tables" on page 19-60.

Editing Tables

Both the default **Condition Table** and the default **Action Table** have one empty row. Click a cell to edit its text contents. Use **Tab** and **Shift+Tab** to move horizontally between cells. To add rows and columns to either table, see "Appending Rows and Columns " on page 19-68.

You set the Truth Table Editor to display only one of the two tables by double-clicking the header of the table 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 `-`. Once you select one of these cells, you can use the spacebar 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.

Inserting Rows and Columns

To insert a blank row above an existing table row:

1. Right-click any cell in the row (including the row header).
2. From the context menu, select **Insert Row**.

To insert a blank decision column to the left of an existing decision column:

1. Right-click any cell in the existing decision column (including the column header).
2. From the context menu, select **Insert Column**.

Moving Rows and Columns

To move a condition or action row up or down:

1. Click the row header to select the row.
2. Drag the row to a new position.

   The Truth Table Editor renumbers the rows automatically.

To move a decision column up or down:

1. Click the column header to select the column.
2. Drag the column to a new position.

   The Truth Table Editor renumbers the decision columns automatically.

Printing Tables

	**Print** makes a printed copy or an online viewable copy (HTML file) of the truth table.

Selecting and Deselecting Table Elements

- To select a cell for editing, click the cell.
- To select text in a cell, click and drag your pointer over the text.
- To select a row, click the header for the row.
- To select a decision column in the **Condition Table**, click the column header (**D1**, **D2**, and so on).
- To deselect a selected cell, row, or column, press **Esc**, or click another table, cell, row, or column.

Undoing and Redoing Edit Operations

	Select the **Undo** tool or press **Ctrl+Z (Command+Z)** to reverse the effects of the preceding operation.
	Select the **Redo** tool or press **Ctrl+Y (Command+Y)** to reverse the effects of the most recently undone edit operation.

Viewing the Stateflow Chart for the Truth Table

	**Go to Stateflow Editor** displays the current truth table function in its native Stateflow chart.

Searching and Replacing Text in Truth Tables

You can use the Search & Replace tool to search for text in the **Description**, **Condition**, and **Action** columns of a truth table and replace it with a substitute string.

A Simple Example

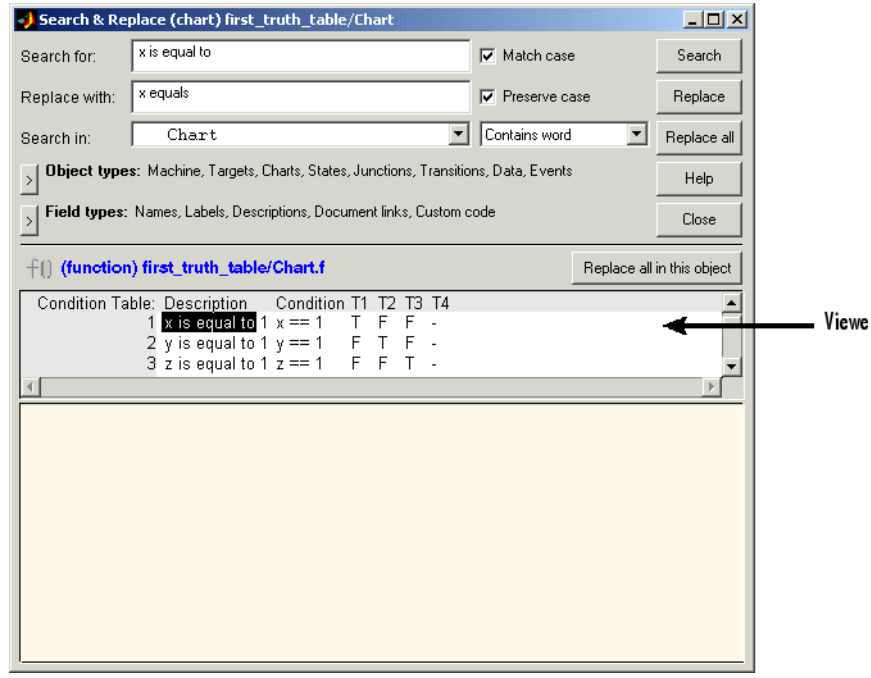
Suppose you want to search a model for the string `x is equal to` and replace it with the string `x equals`. Follow these steps:

1 In the Stateflow Editor, select **Search & Replace** from the **Tools** menu.

2 In the Search & Replace dialog box, enter the text `x is equal to` in the **Search** field, and the text `x equals` in the **Replace** field.

3 Select the **Search** button.

You see something like this in the Search & Replace window.



In the Viewer pane of the Search & Replace window, the first occurrence of the string `x is equal to` is highlighted normally and the other matches are highlighted lightly.

4 Select **Replace** to replace the first match with `x equals`.

5 Select **Replace All** to replace all matches in the model (not just in the truth table) with `x equals`.

Note For more information, see "Using the Stateflow Search & Replace Tool" on page 24-13.

Using Row and Column Tooltip Identifiers

Row and column header tooltips appear to aid truth table navigation when you scroll to other columns or rows while editing a large truth table. When you place your pointer over row or column headers, these tooltips appear:

Table	Row or Column	Tooltip
Condition	Condition row	Condition entered for this row
Condition	Decision column (**D1**, **D2**,...)	Row or label entered for this decision in the **Actions** row
Condition	**Actions** row	Actions: specify a row from the Action Table
Action	Any row	Description entered for this action

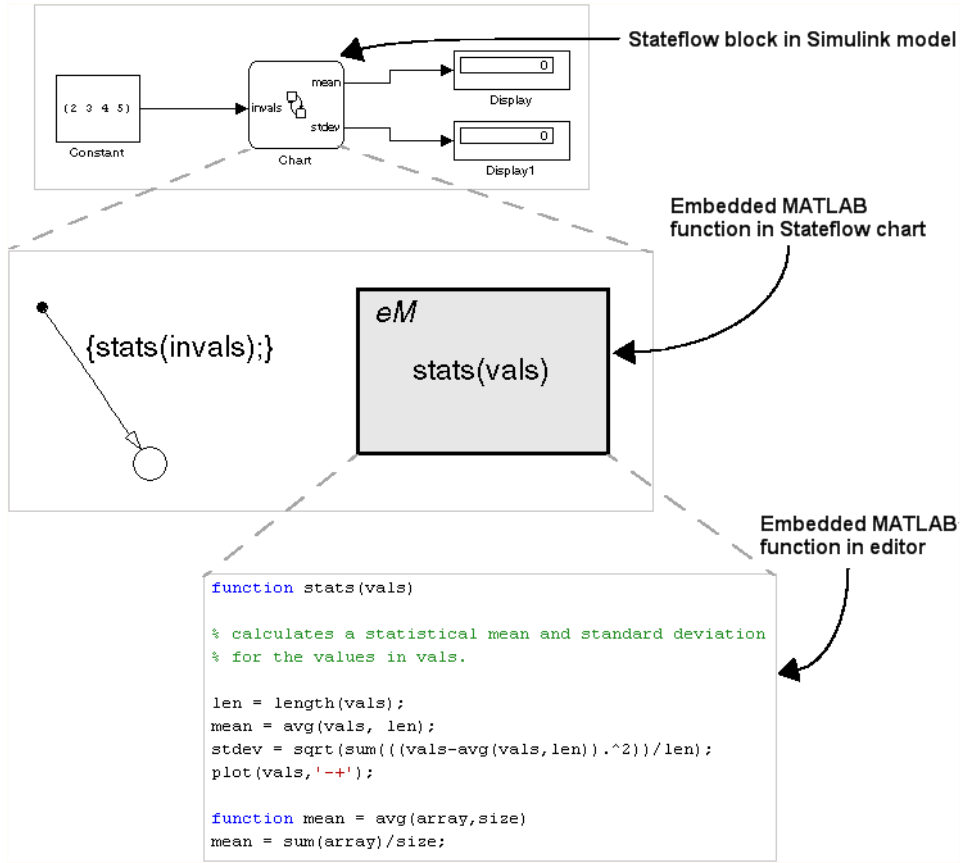
20

Using Embedded MATLAB Functions in Stateflow Charts

Introduction to Embedded MATLAB Functions

You can use Embedded MATLAB functions to add MATLAB functions to a Stateflow chart. This capability is useful for coding algorithms that are better expressed in the textual MATLAB language than in the graphical Stateflow action language. Embedded MATLAB functions work with a subset of the MATLAB language called the Embedded MATLAB subset, which provides optimizations for generating efficient, production-quality C code for embedded applications. For more information, see "Working with the Embedded MATLAB Subset".

This example shows a Simulink model with a Stateflow chart that contains an Embedded MATLAB function.



You will build this model in “Building a Simulink Model with an Embedded MATLAB Function” on page 20-5.

Note in this example that the Embedded MATLAB function can call any of these types of functions:

- Subfunctions

Subfunctions are defined in the body of the Embedded MATLAB function. In the preceding example, `avg` is a subfunction. See "Calling Subfunctions" in the Embedded MATLAB User's Guide.

- Embedded MATLAB run-time library functions

  Embedded MATLAB run-time library functions are a subset of the functions that you can call in the MATLAB workspace. They generate C code for building targets that conform to the memory and data type requirements of embedded environments. In the preceding example, `length`, `sqrt`, and `sum` are examples of Embedded MATLAB run-time library functions. See "Calling Embedded MATLAB Library Functions" in the Embedded MATLAB User's Guide.

- Stateflow functions

  Graphical, truth table, and other Embedded MATLAB functions can be called from an Embedded MATLAB function in a Stateflow chart.

- Some MATLAB functions

  Functions that cannot be resolved as subfunctions, Embedded MATLAB run-time library functions, or Stateflow functions are resolved in the MATLAB workspace. These functions do not generate code; they execute only in the MATLAB workspace during simulation of the model. See "Calling MATLAB Functions" in the Embedded MATLAB User's Guide.

- Fixed-Point Toolbox™ run-time library functions

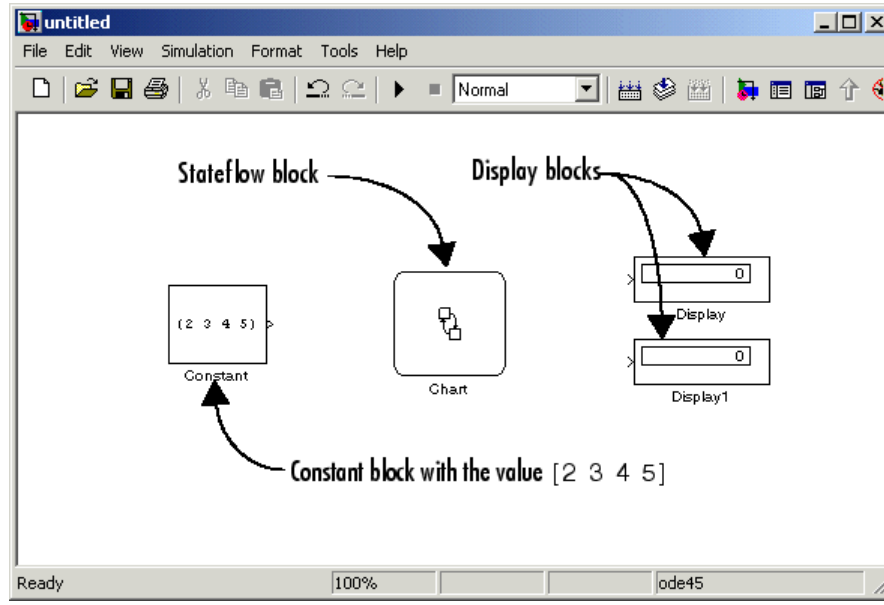
  For more information on fixed-point support in Embedded MATLAB functions, refer to "Working with the Fixed-Point Embedded MATLAB Subset" in the Fixed-Point Toolbox User's Guide.

Building a Simulink Model with an Embedded MATLAB Function

This section explains how to create a Simulink model with a Stateflow block that calls two Embedded MATLAB functions, `meanstats` and `stdevstats`. `meanstats` calculates a mean and `stdevstats` calculates a standard deviation for the values in `vals` and outputs them to the Stateflow data `mean` and `stdev`, respectively.

Follow these steps:

1. Create a new Simulink model with the following blocks:



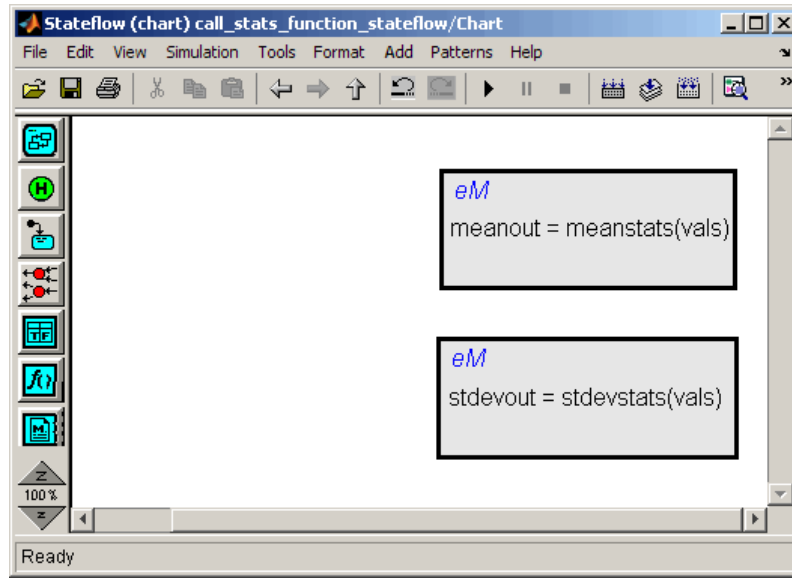
2. Save the model as `call_stats_function_stateflow`.

3. In the Simulink model, double-click the Stateflow block to open the Stateflow Editor.

4. In the Stateflow Editor, drag two Embedded MATLAB functions into the empty Stateflow chart using this icon from the tool palette:

A text field with a flashing cursor appears in the middle of each Embedded MATLAB function.

5 Label each function as shown:



You must label an Embedded 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.

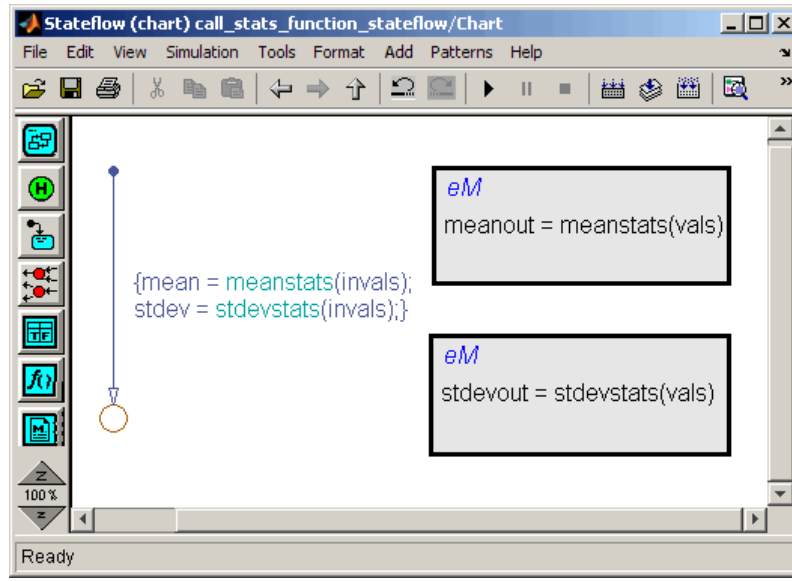
Note For Embedded MATLAB functions with only one return value, you can omit the brackets in the signature label.

---

6 In the Stateflow chart, draw a default transition into a terminating junction with this condition action:

```
{mean = meanstats(invals);
stdev = stdevstats(invals);}
```

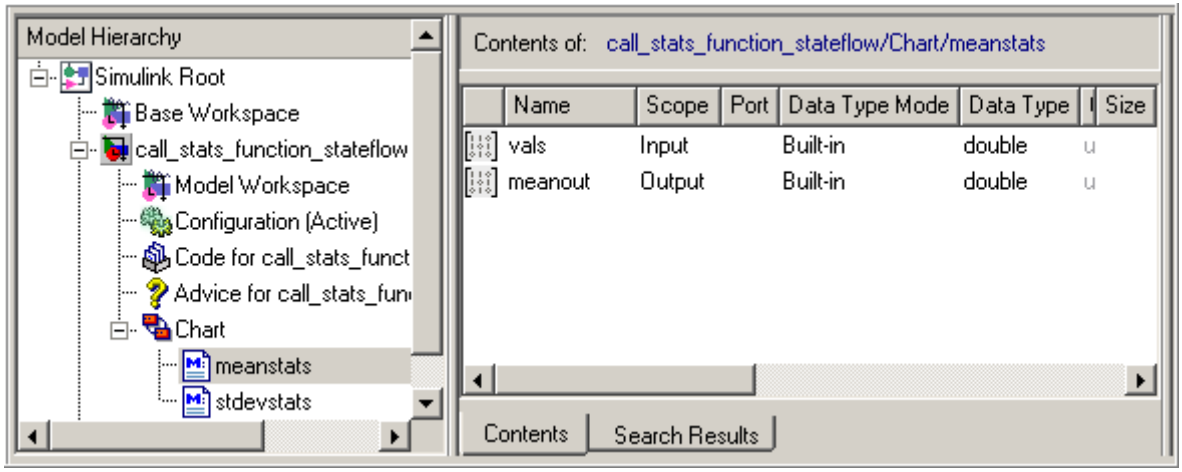
The Stateflow chart should look like this figure.



7 In the Stateflow chart, double-click the function `meanstats` to edit its function body in the Embedded MATLAB Editor.

8 In the Embedded MATLAB Editor, select **Tools > Model Explorer**.

The Model Explorer appears.



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.

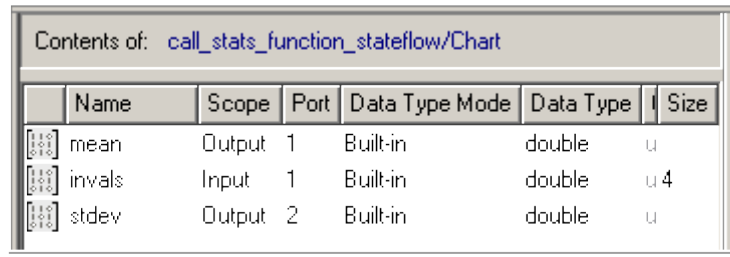
9 Double-click the `vals` row under the **Size** column to set the size of `vals` to `4`.

10 Back in the Stateflow chart, double-click the function `stdevstats` and repeat steps 8 and 9.

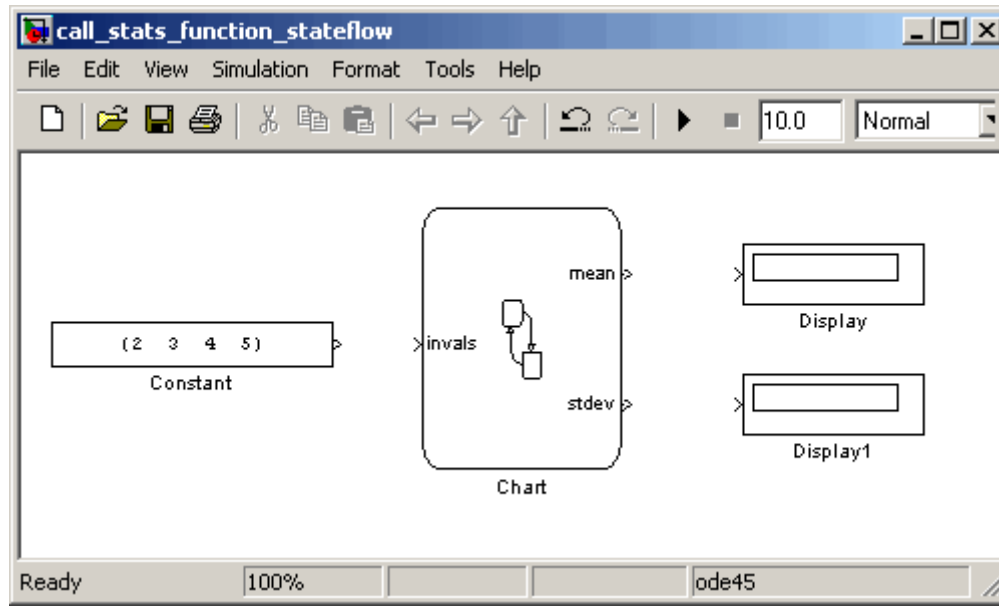
11 Back in the **Model Hierarchy** pane of the Model Explorer, select **Chart** and add the following data:

Name	Scope	Size
`invals`	`Input`	4
`mean`	`Output`	Scalar (no change)
`stdev`	`Output`	Scalar (no change)

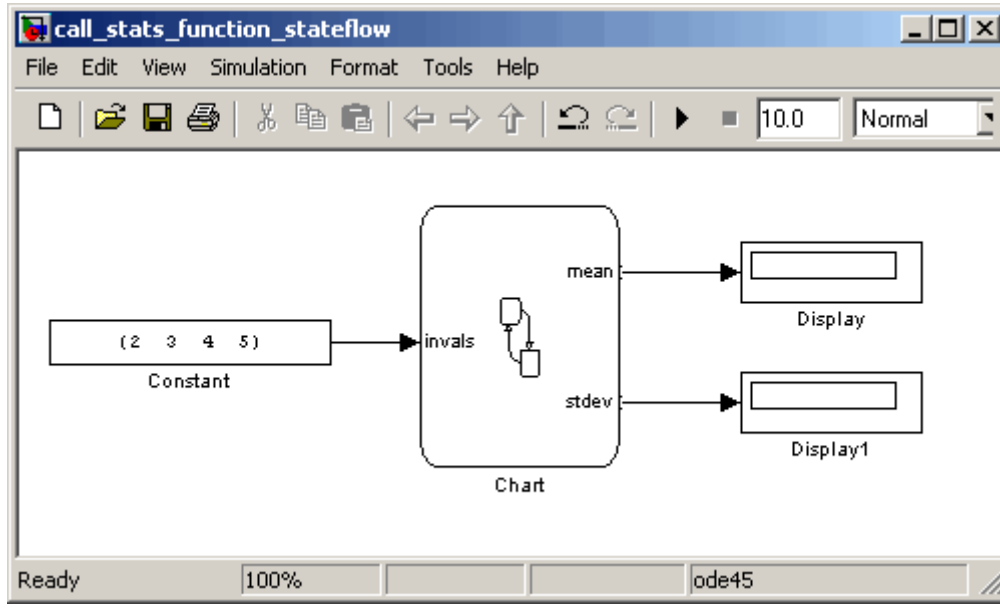
You should now see the following data in the Model Explorer.



After you add the data `invals`, `mean`, and `stdev` to the Stateflow chart, the corresponding input and output ports appear on the Stateflow block in the Simulink model.



12 Connect the Constant block and the Display block to the ports of the Stateflow block and save the model.



The section "Debugging an Embedded MATLAB Function" on page 20-15 shows you how to program the functions `meanstats` and `stdevstats`.

Programming an Embedded MATLAB Function

To program the functions `meanstats` and `stdevstats` that you created in "Building a Simulink Model with an Embedded MATLAB Function" on page 20-5, follow these steps:

1. Open the Stateflow chart in the model `call_stats_function_stateflow`.

2. In the Stateflow chart, open the Embedded MATLAB function `meanstats`.

   The Embedded MATLAB Editor appears with the function header as shown.

   ```
   function meanout = meanstats(vals)
   ```

   This function header is taken from the label that you added to the function in the Stateflow chart. You can edit it directly in the Embedded MATLAB Editor, and your changes will be reflected in the Stateflow Editor when you close the window or click the **Update Diagram** icon in the toolbar:

3. After the function header, enter a line space and this comment:

   ```
   % Calculates a statistical mean for vals
   ```

4. Now enter this statement:

   ```
   eml.extrinsic('plot');
   ```

   The function `plot` is a MATLAB function that is not supported by the Embedded MATLAB subset. To call unsupported MATLAB functions, you must first declare them to be extrinsic, as described in "Calling MATLAB Functions" in the Embedded MATLAB User's Guide.

5. Add the line:

   ```
   len = length(vals);
   ```

The function `length` is an example of a built-in MATLAB function that is supported by the Embedded MATLAB subset. 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. MATLAB functions supported by the Embedded MATLAB subset appear in "Embedded MATLAB Function Library Reference" in the Embedded MATLAB User's Guide.

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`. You can change the size and type of `len` as described in in the Embedded MATLAB User's Guide.

The Embedded MATLAB function treats implicitly declared local data as temporary data. It comes into existence only when the function is called and disappears when the function exits. You can declare local data for an Embedded MATLAB function to be persistent by using the `persistent` construct (see "Declaring Persistent Variables" in the Embedded MATLAB User's Guide).

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` . Since these data are defined for the parent Stateflow chart, you can use them directly in the Embedded MATLAB function.

Two-dimensional arrays with a single row or column of elements are treated as vectors or matrices in Embedded 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 Embedded MATLAB function uses the functions `avg` and `sum` to compute the value of `mean`. `sum` is an Embedded MATLAB run-time library function. `avg` is a subfunction that you define later. When resolving function names, Embedded MATLAB functions look for subfunctions first, followed by Embedded MATLAB run-time library functions.

Note If you call a function that the Embedded MATLAB function cannot resolve as a subfunction or Embedded MATLAB runtime library function, you must declare the function to be extrinsic so it can be resolved as a MATLAB function, as described in "Calling MATLAB Functions" in the Embedded MATLAB User's Guide.

7 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 in the Embedded MATLAB runtime library. When the Embedded MATLAB function encounters an extrinsic function, it sends the call to the MATLAB workspace for execution during simulation.

8 Now, define the subfunction `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 subfunction `avg` calculates the average of the elements in `array` by dividing their sum by the value of argument `size`.

For more information on creating subfunctions, see "Subfunctions" in MATLAB software documentation.

The complete code for the Embedded MATLAB function `meanstats` looks like this:

```
function meanout = meanstats(vals)

% Calculates a statistical mean for vals

eml.extrinsic('plot');
len = length(vals);
meanout = avg(vals,len);

plot(vals,'-+');
```

```
function mean = avg(array,size)
mean = sum(array)/size;
```

9 Save the model (`call_stats_function_stateflow`).

10 Back in the Stateflow chart, open the second Embedded MATLAB 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)

%Calculate 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;
```

Debugging an Embedded MATLAB Function

In this section...

Checking Embedded MATLAB Functions for Syntax Errors

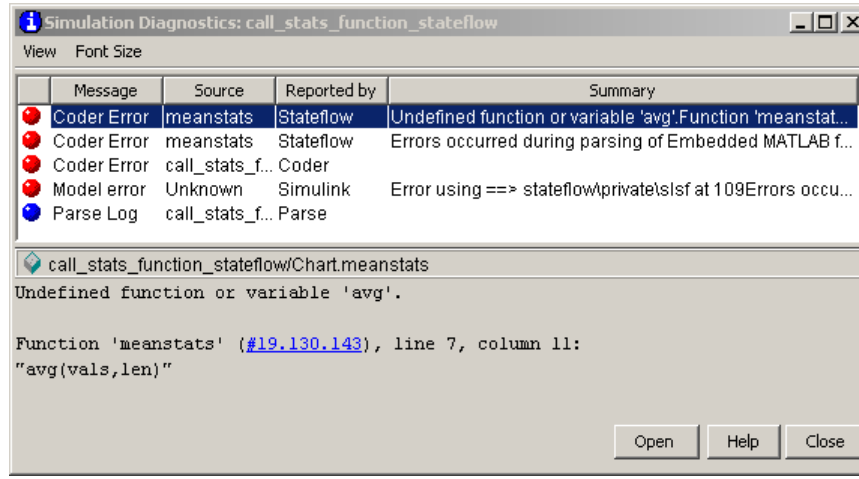
Before you can build a simulation application for a model, you must fix syntax errors. Follow these steps to check the Embedded MATLAB function `meanstats` for syntax violations:

1. Open the Embedded MATLAB function `meanstats` inside the Stateflow chart in the `call_stats_function_stateflow` model that you updated in "Programming an Embedded MATLAB Function" on page 20-11.

   The Embedded MATLAB Editor uses the MATLAB M-Lint Code Analyzer to automatically check your function code for errors and recommend corrections (see "Using M-Lint with Embedded MATLAB Code" in the Embedded MATLAB User's Guide).

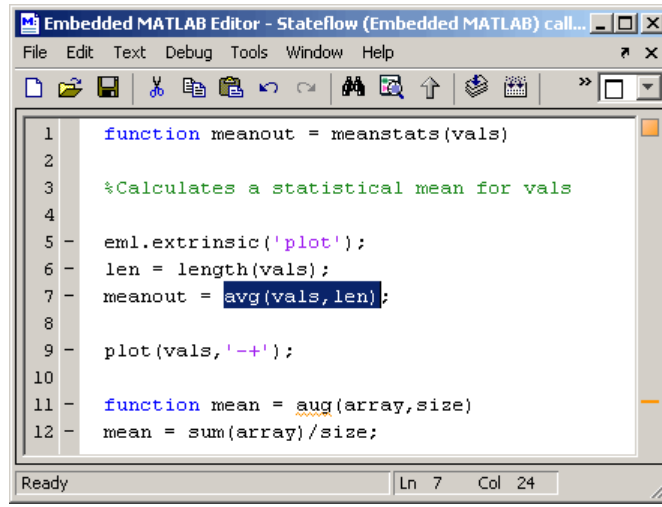
2. In the Embedded MATLAB Editor, select the Build tool  to build a simulation application for the example Simulink model.

   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 subfunction `avg` to a nonexistent subfunction `aug` in `meanstats`, the Builder reports these errors:



Each error message appears with a red button. The selected error message displays diagnostic information in the bottom pane.

3 Click the link in the diagnostic message to display the offending line of code, as shown.



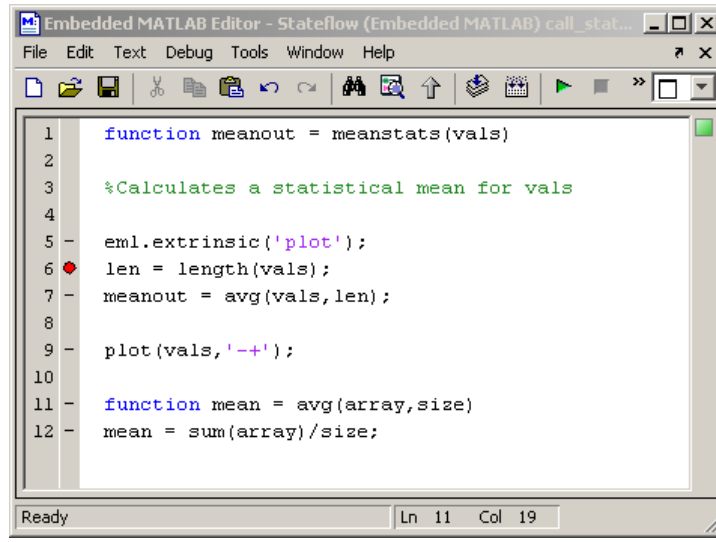
Run-Time Debugging for Embedded MATLAB Functions

You use simulation to test your Embedded MATLAB functions for run-time errors that are not detectable by the Stateflow Debugger. When you simulate your model, your Embedded MATLAB functions undergo diagnostic tests for missing or undefined information and possible logical conflicts as described in "Checking Embedded MATLAB Functions for Syntax Errors" on page 20-15. If no errors are found, the simulation of your model begins.

Follow these steps to simulate and debug the `meanstats` Embedded MATLAB function during run-time conditions:

1. In the Embedded MATLAB Editor, click the dash (-) character in the left margin of line 6.

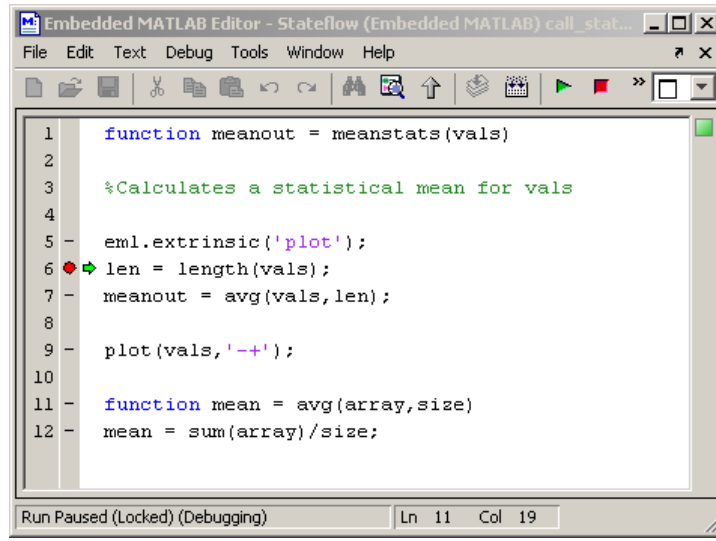
   A small red ball appears next to line 6, indicating that you set a breakpoint.



2. Click the **Start Simulation** icon  to begin simulating 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. This pause is indicated by a small green arrow in the left margin as shown.



3 Click the **Step** icon  to advance execution one line to line 7.

Notice that line 7 calls the subfunction `avg`. If you click **Step** here, execution advances to line 9, past the execution of the subfunction `avg`. In order to track execution of the lines in the subfunction `avg`, you must click the **Step In** icon.

4 Click the **Step In** icon  to advance execution to the first line of the called subfunction `avg`.

Once you are in the subfunction, you can advance through the subfunction one line at a time with the Step tool. If the subfunction calls another subfunction, use the Step In tool to step into it. If you want to continue through the remaining lines of the subfunction and go back to the line after the subfunction call, click the **Step Out** icon  .

5 Click the **Step** icon to execute the only line in the subfunction `avg`.

When the subfunction `avg` finishes its execution, you see a green arrow pointing down under its last line.

6 Click the **Step** icon to return to the function `meanstats`.

Execution advances to the line after to the call to the subfunction `avg`, line 9.

7 To display the value of the variable `len`, place your pointer over the text `len` in line 6 for at least a second.

The value of `len` appears adjacent to your pointer.

You can display the value for any data in the Embedded MATLAB block 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` in line 6, or as an argument in the function header.

You can also report the values for Embedded 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 might have to press **Enter** to see it). At this prompt, you can inspect data defined for the Embedded MATLAB function by entering the name of the data, as shown in this example:

```
debug>> len
len =
     4
debug>>
```

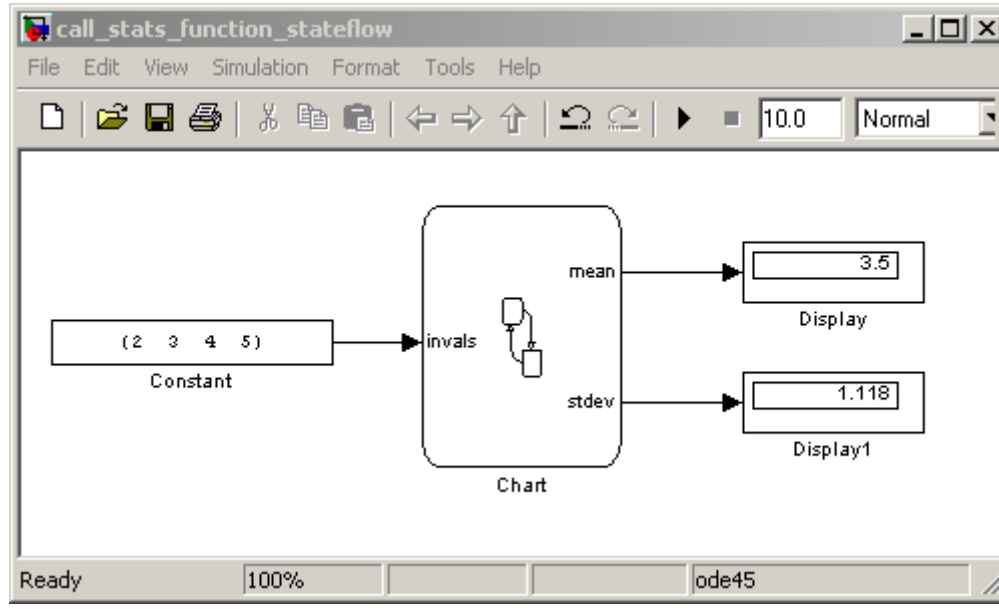
As another debugging alternative, you can display the execution result of an Embedded MATLAB function line by omitting the terminating semicolon. If you do, execution results for the line are echoed to the MATLAB Command Window during simulation.

8 Click the **Continue** icon  to leave the function until it is called again and the breakpoint on line 6 is reached.

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, clicking the **Step** icon accomplishes the same thing.

9 Click the breakpoint at line 6 to remove it and click the green arrow to complete the simulation.

In the Simulink window, the computed values of `mean` and `stdev` now appear in the Display blocks.



Checking for Data Range Violations

During debugging, Embedded MATLAB functions automatically check input and output data for data range violations.

Specifying a Range

To specify a range for input and output data, follow these steps:

1 In the Model Explorer, select the Embedded 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, select the **Value Attributes** tab and enter a limit range, as described in "Limit range properties" on page 8-21.

Controlling Data Range Checking

To control data range checking, follow these steps:

1 Open the Stateflow Debugger, as described in "Opening the Stateflow Debugger" on page 23-2.

2 In the **Error checking options** pane, perform one of these actions:

To:	Do This:
Enable data range checking	Select the **Data Range** check box
Disable data range checking	Clear the **Data Range** check box

Model Coverage for an Embedded MATLAB Function

In this section...

About Model Coverage

The **Model Coverage** tool reports model coverages for the decisions and conditions of Embedded MATLAB functions. For example, the Embedded MATLAB function `if` statement

```
if (x > 0 || y > 0)
 reset = 1;
```

contains a decision with two conditions (`x > 0` and `y > 0`). You use the **Model Coverage** tool for Embedded MATLAB functions to make sure that all decisions and conditions are taken during simulation of the model.

For a description of model coverage for other Stateflow objects, see "Understanding Model Coverage for Stateflow Charts" on page 23-51.

Note The Model Coverage tool requires a Simulink Verification and Validation software license.

Types of Model Coverage in Embedded MATLAB Functions

During simulation, these Embedded MATLAB function statements are tested for Decision Coverage:

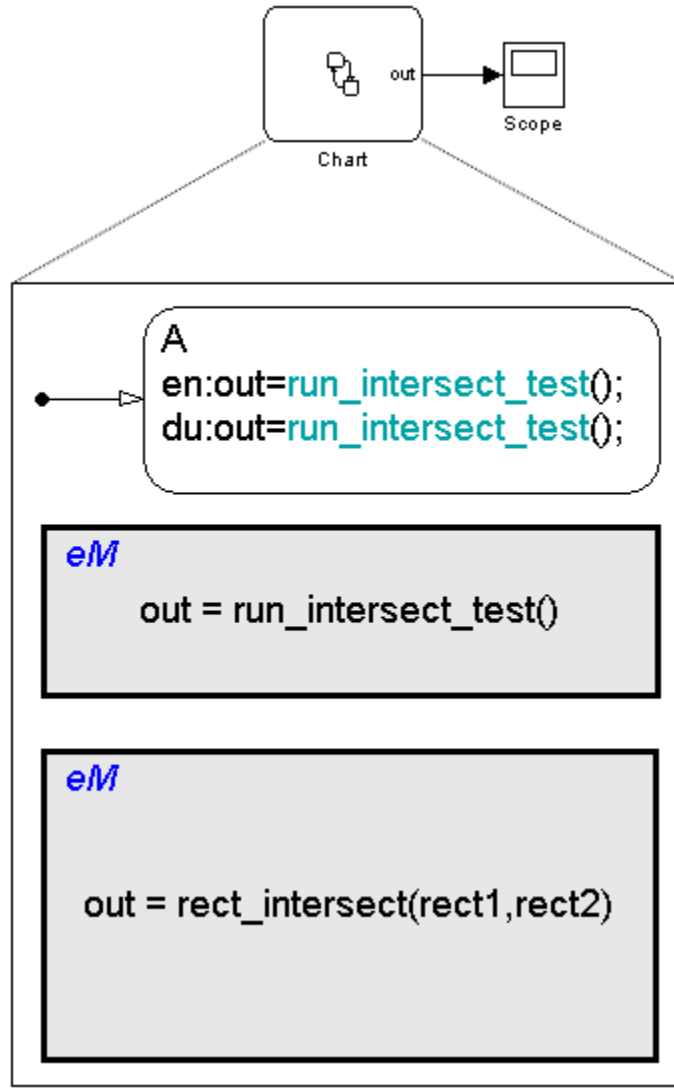
- function header — Decision coverage is 100% if the function or subfunction is executed.
- `if` — Decision coverage is 100% if the `if` expression evaluates to true at least once and false at least once.
- `switch` — Decision coverage is 100% if every `switch` case is taken, including the fall-through case.
- `for` — Decision coverage is 100% if the equivalent loop condition evaluates to true at least once, and false at least once.
- `while` — Decision coverage is 100% if the equivalent loop condition evaluates to true at least once, and false at least once.

During simulation, these logical conditions are tested for Condition Coverage and MCDC in the Embedded MATLAB function:

- `if` statement conditions
- `while` statement conditions, if present

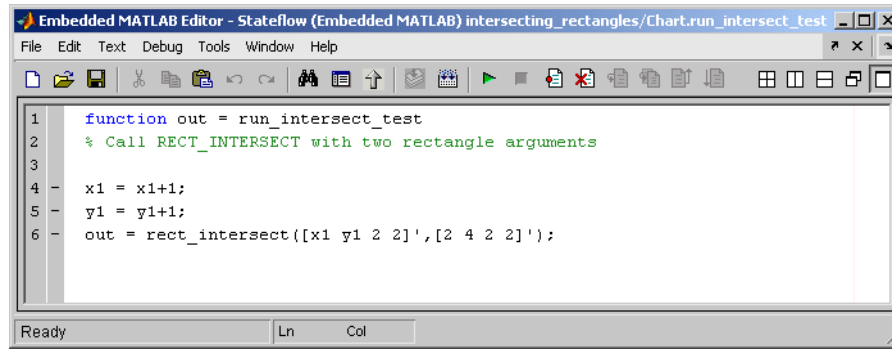
Creating a Model with Embedded MATLAB Function Decisions

In this topic, you examine an example model you can use to generate a model coverage report for two Embedded MATLAB functions. The following model is named `intersecting_rectangles` and contains a single Stateflow chart with output data sent to a Scope block as shown.

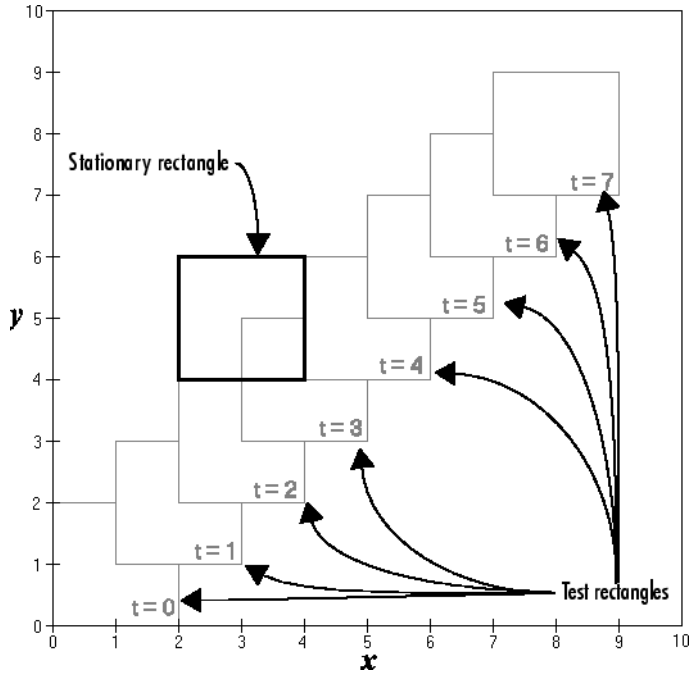


The preceding Stateflow chart has a state with a default transition and entry and during actions. The state executes its entry action the first time that it is entered for the first time sample. Each succeeding time sample calls the during action of the active state.

The entry and during actions of state A call the Embedded MATLAB function `run_intersect_test`, which appears as follows in the Embedded MATLAB Editor:

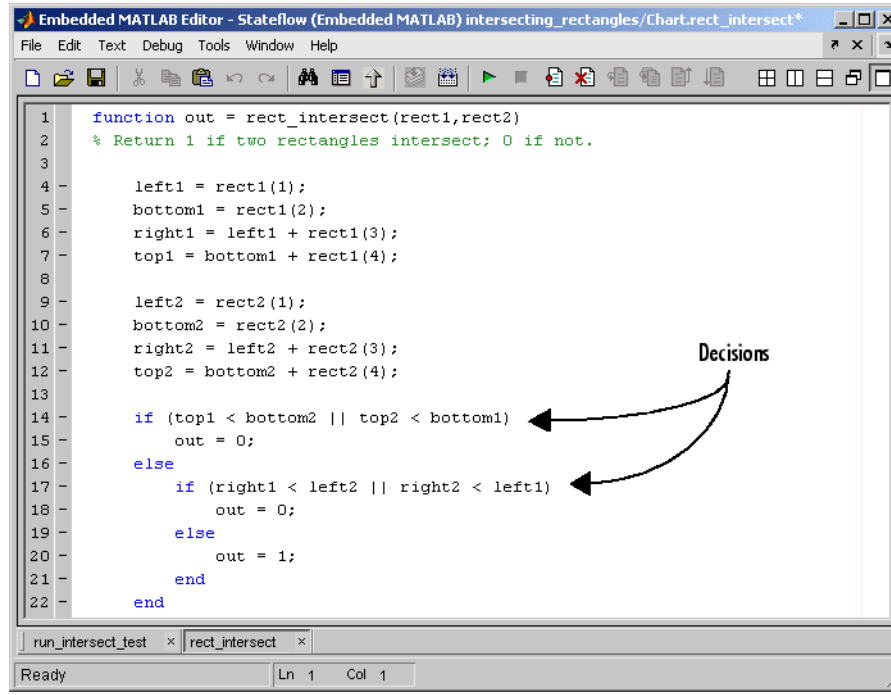


`run_intersect_test` calls the function `rect_intersect` with two rectangle arguments that each consist of coordinates for the lower left corner of the rectangle (origin), and its width and height. The first rectangle is a test rectangle, and the second is a stationary rectangle. The coordinates for the origin of the test rectangle are represented by the Stateflow data `x1` and `y1`, which are both initialized to -1. This means that `x1` and `y1` are 0 for the first sample. The progression of rectangle arguments during simulation is as follows:



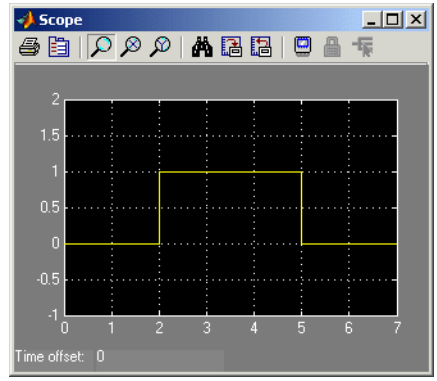
In the preceding display, the stationary rectangle is shown in bold with a lower left origin of `(2,4)` and a width and height of 2. At time `t = 0`, the first test rectangle has an origin of `(0,0)` and a width and height of 2. For each succeeding sample, the origin of the test rectangle increments by `(1,1)`. The rectangles at sample times `t = 2`, `3`, and `4` intersect with the test rectangle.

The function `rect_intersect`, as shown, checks to see if two rectangles intersect.



`rect_intersect` receives the two rectangle arguments from `run_intersect_test`. It first calculates horizontal (`x`) coordinates for the left and right sides, and vertical (`y`) values for the top and bottom sides for each rectangle and compares them in the nested `if-else` decisions shown. The function returns a logical value of 1 if the rectangles intersect and 0 if they do not.

Scope output during simulation, which plots the return value against the sample time, confirms the intersecting rectangles for sample `2`, `3`, and `4`.



Understanding Embedded MATLAB Function Model Coverage

Model coverage reports are generated during simulation if you specify them (see "Making Model Coverage Reports" on page 23-52). When simulation is finished, the model coverage report appears in a browser window. After the summary for the model, the Details section reports on each of the parts of the model. Model coverage for the parts of the example model in "Creating a Model with Embedded MATLAB Function Decisions" on page 20-23 appears in this order:

- Model `intersecting_rectangles`
- Subsystem `Chart`
- Chart `Chart`
  - Function `rect_intersect`

    #1: `function out = rect_intersect(rect1,rect2)`

    #14: `if (top1 < bottom2 || top2 < bottom1)`

    #17: `if (right1 < left2 || right2 < left1`
  - Function `run_intersect_test`

    #1: `function out = run_intersect_test`

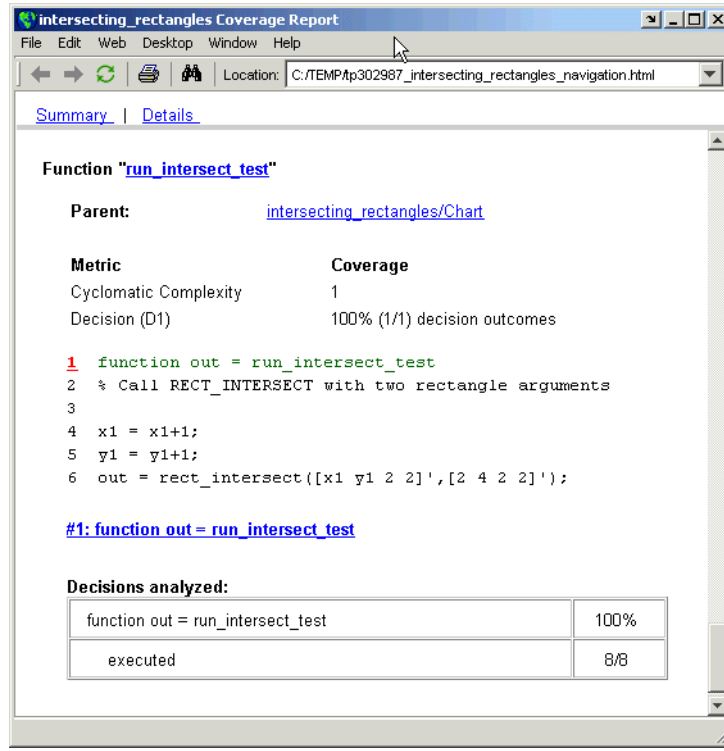
The reports for the Embedded MATLAB functions `rect_intersect` and `run_intersect_test` appear in alphabetical order as part of the report on

their parent, the Stateflow block Chart. The reports on individual decisions for each function appear in numerical line order. Line numbers are indicated by the # character.

The following subtopics examine the model coverage report for the example model in reverse order of the report. Reversing the order helps you make sense of the summary information that appears at the top of each section.

Model Coverage for Embedded MATLAB Function run_intersect_test

Model coverage for the Embedded MATLAB function `run_intersect_test`, which sends test rectangles to the function `rect_intersect`, appears in the model coverage report as shown.

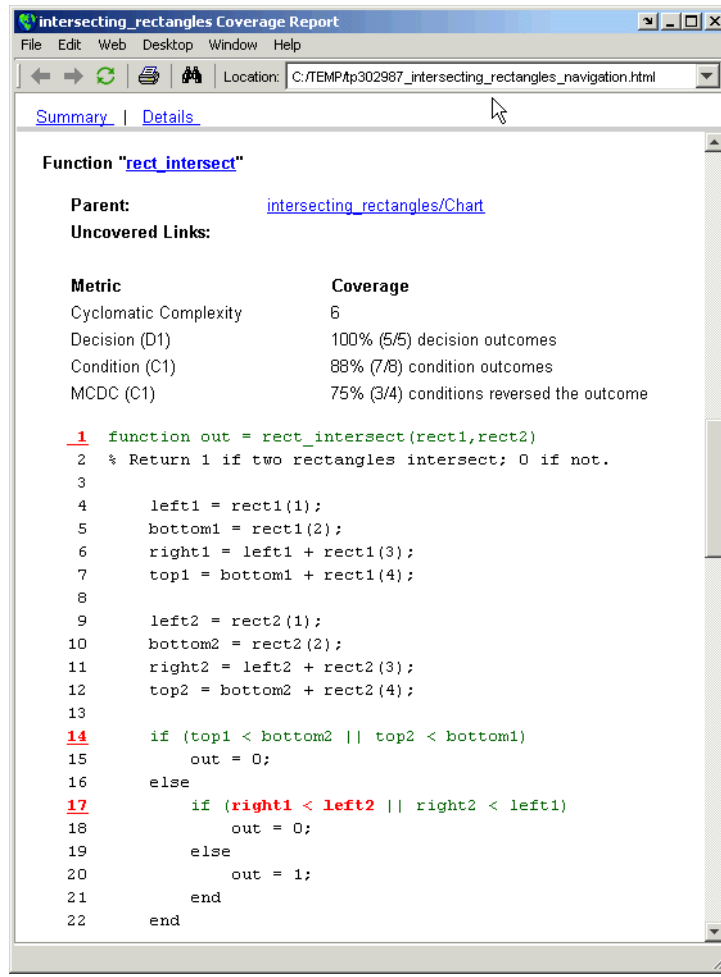


The report on `run_intersect_test` begins with the function name `run_intersect_test`, which links to an Embedded MATLAB Editor for the function. Following the name is a link to the model coverage report for the parent of `run_intersect_test`, the Stateflow chart `Chart`. Coverage continues with a summary of the coverage metrics for the function followed by a code listing. The line number for the first line of the listing is highlighted in bold red, which is actually a link to an analysis for that decision below.

The first line of every function receives coverage analysis indicative of the decision to run the function. Its coverage here indicates that the function `run_intersect_test` executed 8 out of 8 times for the samples taken at 0 through 7 seconds. Because this is the only decision in the function `run_intersect_test`, coverage for it in the metrics table above indicates the occurrence of 1 out of 1 possible outcomes. In this case, the only possible outcome is function execution. In other words, the function was executed during simulation.

Coverage for Embedded MATLAB Function rect_intersect

Model coverage for the Embedded MATLAB function `rect_intersect`, which tests rectangles for intersection, appears first in the model coverage report as shown.

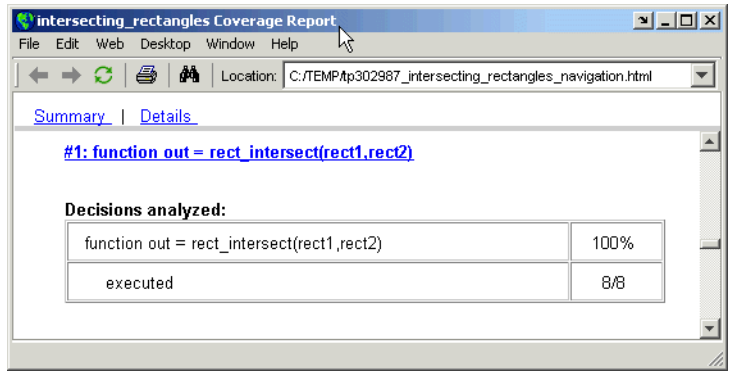


The coverage metrics for `rect_intersect` include Decision, Condition, and MCDC coverage. These metrics are best understood after you examine the coverage for the decisions in `rect_intersect` highlighted in the listing below.

The listing for `rect_intersect` includes three highlighted line numbers. The first line is highlighted as the decision on whether or not to execute the function. Highlighted line numbers 14 and 17 indicate decisions in a nested `if-else` statement. Notice that the condition `right1 < left2` in line 17 is

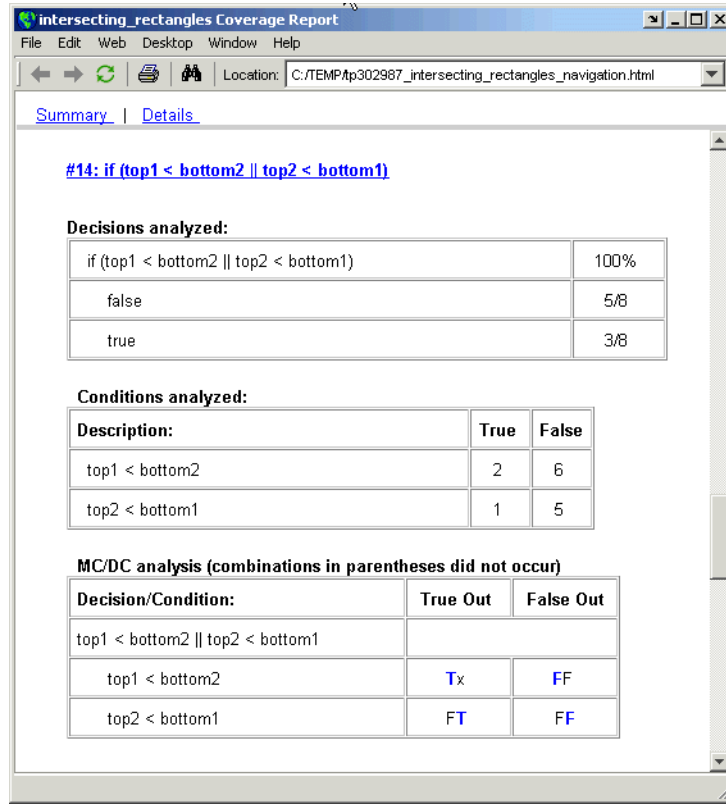
highlighted in red. This means that this condition did not test both the true and false possible outcomes for this decision during simulation. Exactly which of the outcomes was not tested is answered by the metrics for the decision in line 17. The metrics for line 17 and the remaining decisions appear below in numerical line order, and are accessed through the line number links in the listing.

Coverage for Line 1. Coverage metrics for line 1 appear directly below the listing for `rect_intersect` as shown.



The first line of every function receives coverage analysis indicative of the decision to run the function in response to a call. The preceding table indicates that `rect_intersect` executed. Coverage for this decision, which is equivalent to decision D1 in the metrics table for `rect_intersect`, is therefore 100%.

Coverage for Line 14. Coverage metrics for line 14 appear directly below the coverage metrics for line 1 as shown.

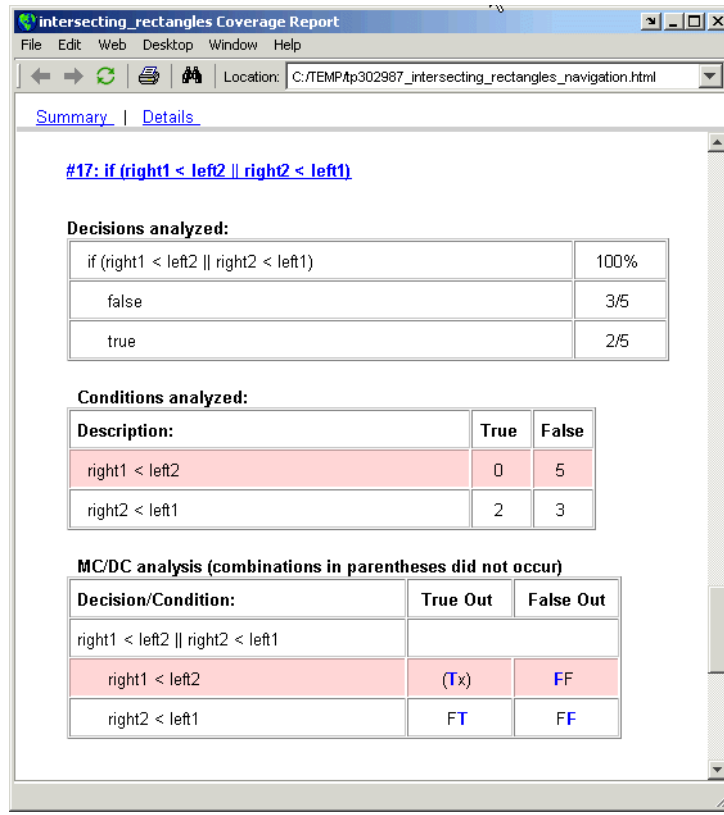


The **Decisions analyzed** table indicates that there are two possible outcomes for the decision in line 14: false and true. 5 of 8 times the decision evaluated false, and the remaining times (3) it evaluated true. Because both the true and false outcome occurred during simulation, Decision Coverage is 100%.

The following **Conditions analyzed** table sheds some light on the decision in line 14. Because this decision consists of two conditions linked by a logical or (`||`) operation, only one condition must evaluate true for the decision to be true. Also, if the first condition evaluates to true, there is no need to evaluate the second condition. The first condition, `top1 < bottom2`, was evaluated 8 times, and was true twice. This means it was necessary to evaluate the second condition only 6 times. In only one case was the second condition true, so that the total true occurrences for the decision equals 3, as reported in the **Decisions analyzed** table.

MCDC coverage looks for decision reversals that occur because one condition outcome changes from `T` to `F` or from `F` to `T`. The table identifies all possible combinations of outcomes for the conditions that lead to a reversal in the decision. The character `x` is used to indicate a condition outcome that is irrelevant to the decision reversal. Reversing condition outcomes that are not achieved during simulation are marked with a set of parentheses. There are no parentheses, so all decision reversing outcomes occurred, and MCDC Coverage is complete for this decision.

Coverage for rect_intersect Line 17. Coverage metrics for line 17 appear directly below the coverage metrics for line 14, as shown.

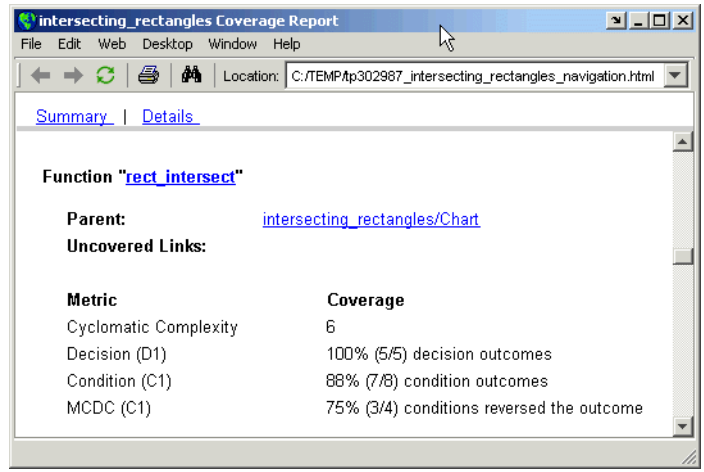


The line 17 decision `if (right1 < left2 || right2 < left1)` is nested in the `if` statement of the line 14 decision and is evaluated only if the line 14 decision is false. Because the line 14 decision evaluated false 5 times, line 17 is evaluated 5 times, 3 of which were false. Because both the true and false outcomes were achieved, Decision Coverage for line 17 is 100%.

Because line 17, like line 14, has two conditions related by a logical OR operator (||), condition 2 is tested only if condition 1 is false. Because condition 1 tests false 5 times, condition 2 is tested 5 times. Of these, condition 2 tests true 2 times and false 3 times, which accounts for the two occurrences of the true outcome for this decision.

Because the first condition of the line 17 decision does not test true, both outcomes did not occur for that condition, and the Condition Coverage for the first condition is highlighted with a rose color. Because the first condition of the line 17 decision does not test true, MCDC coverage is also highlighted in the same way for a decision reversal based on the true outcome for that condition.

Coverage for All rect_intersect Lines. Reviewing the coverage report for each line of `rect_intersect` in the previous topics makes sense of the coverage metrics reported at the beginning of the model coverage report for `rect_intersect`, which is as shown.



Based on the model coverage reports for each line of `rect_intersect`, you can draw these conclusions:

- There are five decision outcomes reported for `rect_intersect` in the line reports: one for line 1 (execute), two for line 14 (true and false), and two for line 17 (true and false). The decision coverage for each line shows 100% coverage. This means that decision coverage for `rect_intersect` is 5 of 5, or 100%.
- There are four conditions reported for `rect_intersect` in the line reports. Lines 14 and 17 each have two conditions, and each condition has two condition outcomes (true and false), for a total of eight condition outcomes in `rect_intersect`. All conditions tested for both the true and false outcome, except for the first condition of line 17 (`right1 < left2`). This means that condition coverage for `rect_intersect` is 7 of 8, or 88%.
- The MCDC tables for each line list two cases of decision reversal for each condition. Only the decision reversal from changing the condition 1 outcome from true to false did not occur during simulation. This means that 3 of 4, or 75%, of the possible reversal cases were tested for during simulation. Therefore, only three of a possible four reversals were observed and coverage is 75%.

Working with Structures and Bus Signals in Embedded MATLAB Functions

In this section...

About Structures in Embedded MATLAB Functions

Embedded MATLAB functions support MATLAB structures. You can create structures in top-level Embedded MATLAB functions in Stateflow charts to interface with Simulink bus signals at input and output ports. Simulink buses appear inside the Embedded MATLAB function as structures; structure outputs from the Embedded MATLAB function appear as buses.

You can also create structures as local and persistent variables in top-level functions and subfunctions of Embedded MATLAB functions.

Defining Structures in Embedded MATLAB Functions

This section describes how to define structures in Embedded MATLAB functions.

- "Rules for Defining Structures in Embedded MATLAB Functions" on page 20-37
- "Defining Structure Inputs and Outputs to Interface with Bus Signals" on page 20-38
- "Defining Local and Persistent Structure Variables" on page 20-39

Rules for Defining Structures in Embedded MATLAB Functions

Follow these rules when defining structures in Embedded MATLAB functions in Stateflow charts:

- For each structure input or output in an Embedded MATLAB function, you must define a `Simulink.Bus` object in the base workspace to specify its type to the Simulink signal.

- Embedded MATLAB structures cannot inherit their type from Simulink signals.
- Embedded MATLAB functions support nonvirtual buses only (see "Virtual and Nonvirtual Buses" in the Simulink User's Guide).
- Structures cannot have scopes defined as **Parameter** or **Constant**.

Defining Structure Inputs and Outputs to Interface with Bus Signals

When you create structure inputs in Embedded 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 Embedded MATLAB function.

You can connect Embedded 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 Embedded MATLAB functions

To define structure inputs and outputs in Embedded 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 Embedded MATLAB function.

   For information about how to create Simulink bus objects, see `Simulink.Bus` in the Simulink Reference.

2. In the Embedded MATLAB Editor, open the Model Explorer by selecting **Tools > Model Explorer**.

3. In the Model Explorer, follow these steps:

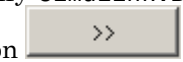
   a. In the **Model Hierarchy** pane, select the Embedded MATLAB function in your Stateflow chart.

b Add a data object, as described in "Adding Data Using the Model Explorer" on page 8-3.

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 Embedded MATLAB function. This name does not have to match the name of the bus object in the base workspace.
Scope	Select `Input` or `Output`.
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, click the Show data type assistant button 

to display the Data Type Assistant. Then, click the **Edit** button to bring up the Simulink Bus Types Editor (see "Using Bus Objects" in the Simulink User's Guide).

e Click **Apply**.

4 If your structure is an output (has scope of **Output**), define the output implicitly in the Embedded MATLAB function to have the same type, size, and complexity as its `Simulink.Bus` object, as described in "About Structures in the Embedded MATLAB Subset" in the Embedded MATLAB User's Guide.

Defining Local and Persistent Structure Variables

You can define structures as local or persistent variables inside Embedded MATLAB functions (see "Types of Structures in the Embedded MATLAB Subset" in the Embedded MATLAB User's Guide).

21

Using Simulink Functions in Stateflow Charts

What Is 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. This function provides an efficient model design and improves 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" in *Simulink User's Guide*)
- Scheduling execution of multiple controllers

In a Stateflow chart, a Simulink function behaves like a function-call subsystem block of a Simulink model (see "Function-Call Subsystems" in *Simulink User's Guide*). However, these differences apply.

Behavior	Function-Call Subsystem	Simulink Function
Requires function-call output events for execution	Yes	No
Requires signal lines in the model	Yes	No
Supports frame-based input and output signals	Yes	No

The scope of a Simulink function in a Stateflow chart depends on where the function resides.

If the function resides in a...	Then you can call the function...
State	In that state and all its substates (see "How a Simulink Function Binds to a State" on page 21-13)
Chart	Anywhere in the chart

When to Use a Simulink Function in a Stateflow Chart

In this section...

Advantages of Using Simulink Functions in a Stateflow Chart

When you define a function that uses Simulink 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

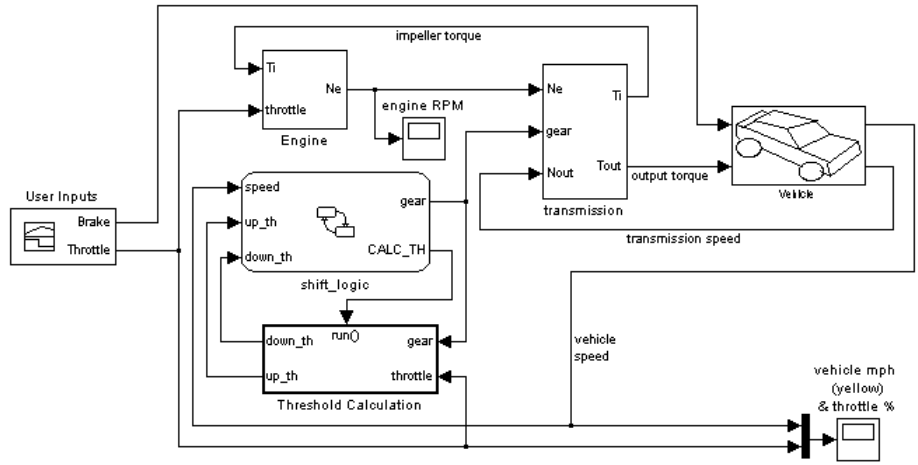
For details about each modeling method, see "Benefits of Using a Simulink Function to Access Simulink Blocks" on page 21-4 and "Benefits of Using a Simulink Function to Schedule Execution of Multiple Controllers" on page 21-6.

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.

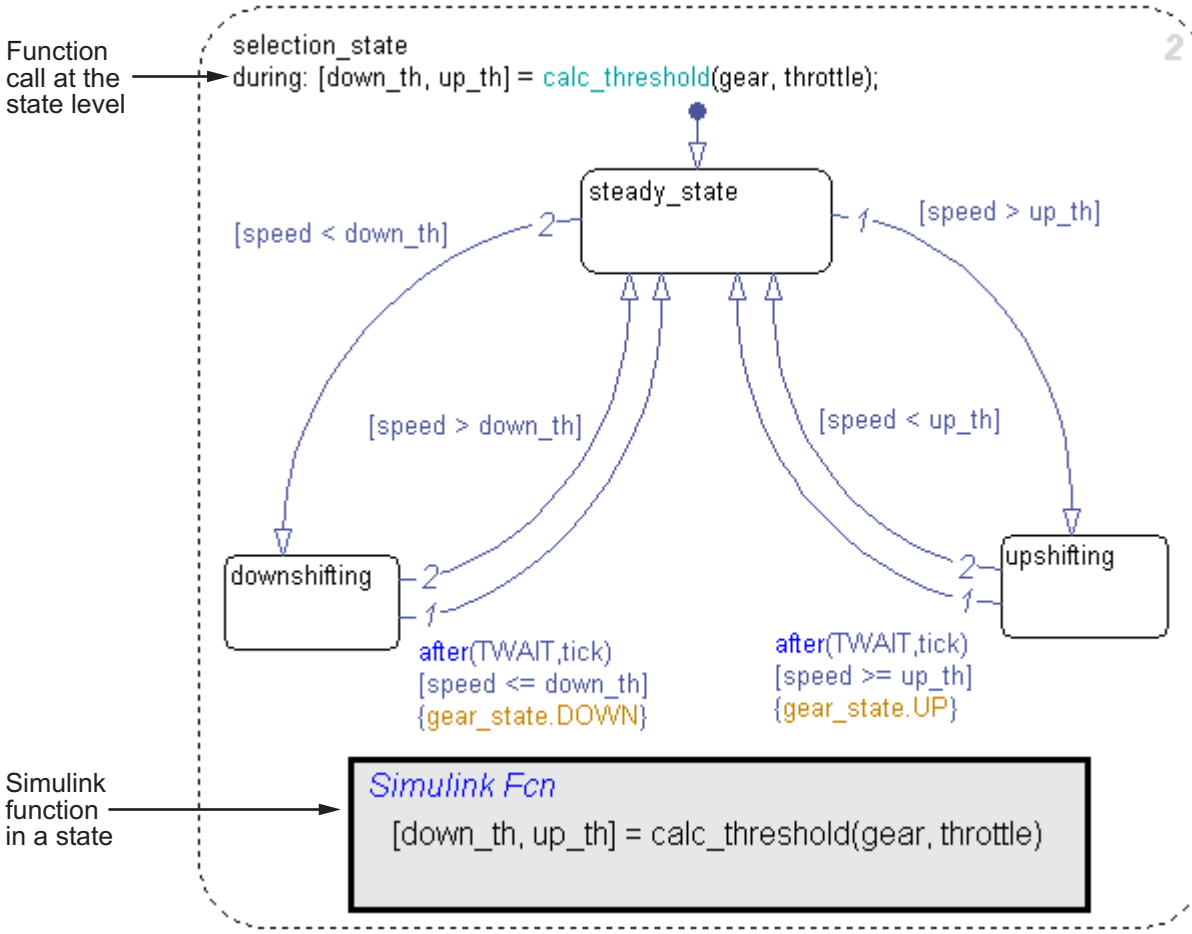
Modeling Method Without a Simulink Function

You define a function-call subsystem in the Simulink model (see "Function-Call Subsystems" in *Simulink User's Guide*). Use an output event in a Stateflow chart to call the subsystem, as shown below.

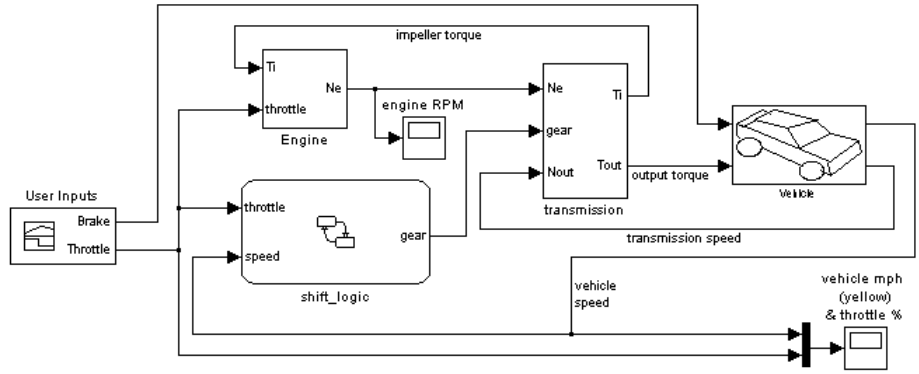


Modeling Method With a Simulink Function

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 below.



This modeling method minimizes the objects in your model.



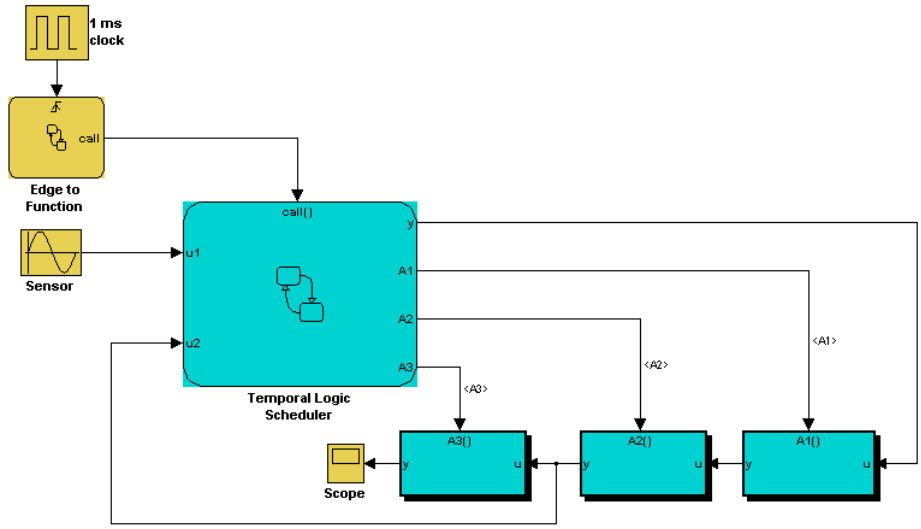
For more information, see “Example of Defining a Function That Uses Simulink Blocks” on page 21-25.

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.

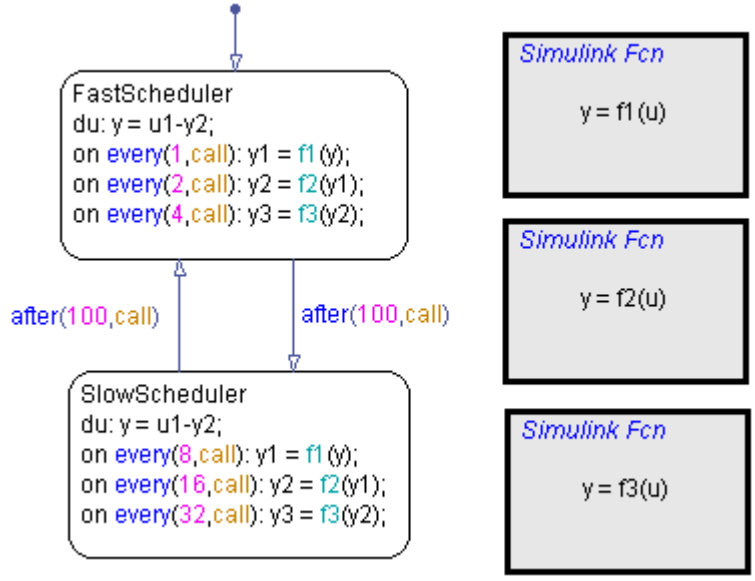
Modeling Method Without Simulink Functions

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 below.

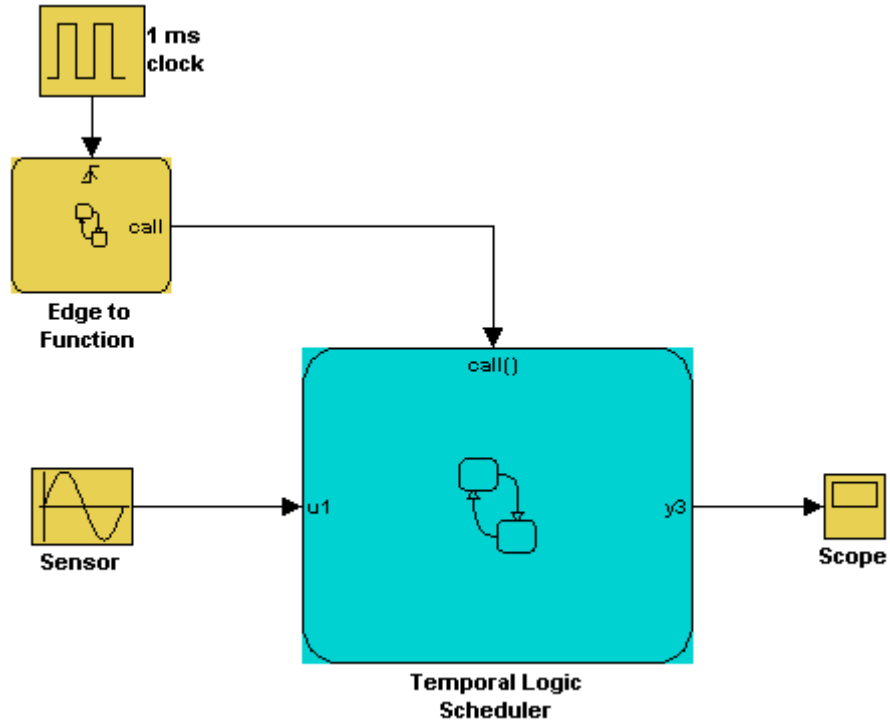


Modeling Method With Simulink Functions

You define each controller as a Simulink function in a Stateflow chart and use function calls to schedule execution of the subsystems, as shown below.



This modeling method minimizes the objects in your model.



For more information, see “Example of Scheduling Execution of Multiple Controllers” on page 21-31.

How to Define a Simulink Function in a Stateflow Chart

In this section...

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:

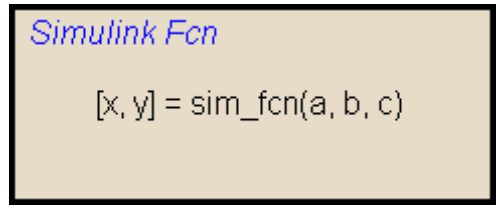
```
[r_1, r_2,..., r_n] = simfcn(a_1, a_2,..., a_n)
```

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 Embedded 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`).



Note You can also create and edit a Simulink function by using API methods. See "API Object Reference" for more information.

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.

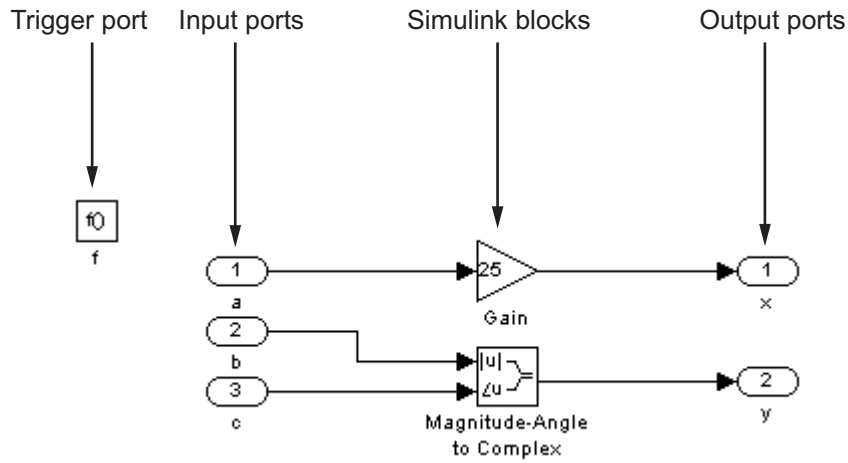
The contents of the subsystem appear: input and output ports that match the function signature and a single function-call trigger port.

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 must 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. For more information, see "Best Practices for Using Simulink Functions" on page 21-24.

How a Simulink Function Binds to a State

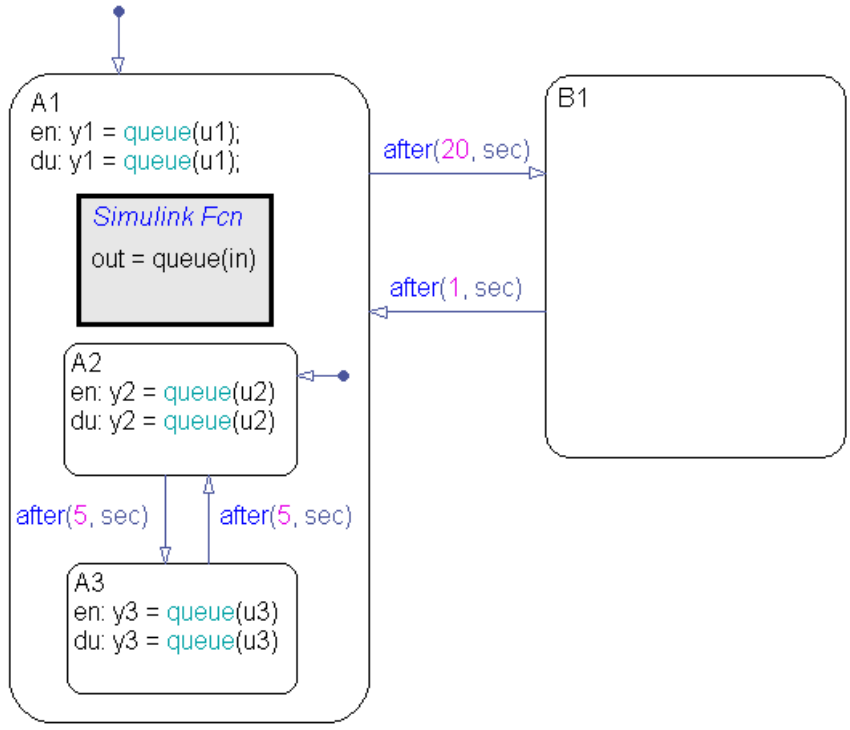
In this section...

Binding Behavior of a Simulink Function

When a Simulink function resides inside a state, the function is bound 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 is bound to a state.



Because the function `queue` resides in state `A1`, the function is bound 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.

Controlling Subsystem Variables When the Simulink Function Is Disabled

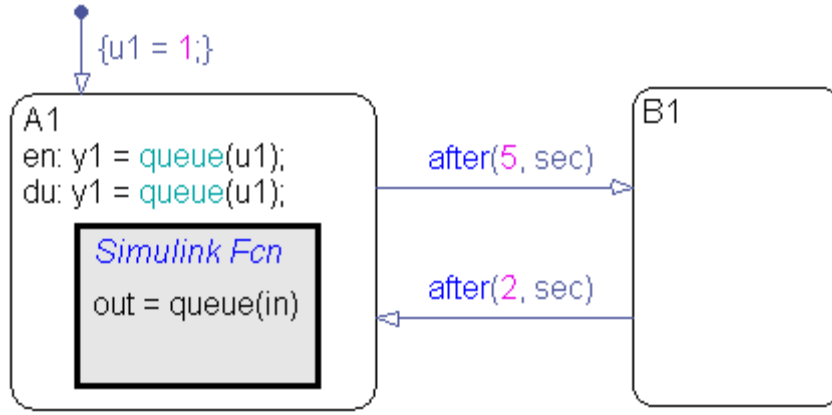
If a Simulink function is bound to a state, you can hold the subsystem variables at their values from the previous execution or reset the variables to their initial values. Follow these steps:

1. In the Simulink function, double-click the trigger port to open the Block Parameters dialog box.
2. Select an option for the field **States when enabling**.

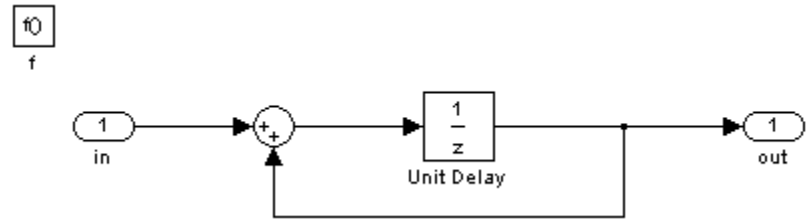
Option	Description	Reference Section
`held`	Holds the values of the subsystem variables from the previous execution	"How the Function Behaves When Variables Are Held" on page 21-18
`reset`	Resets the subsystem variables to their initial values	"How the Function Behaves When Variables Are Reset" on page 21-19

Example of Binding a Simulink Function to a State

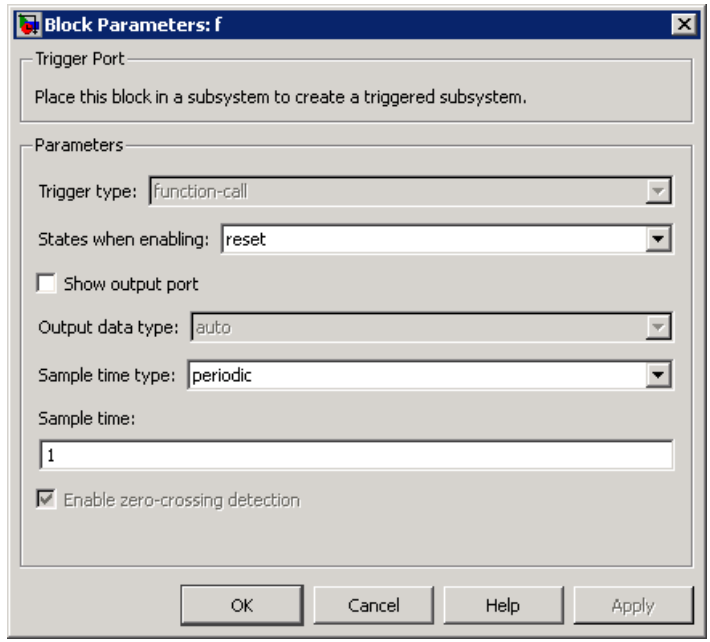
This example shows how a Simulink function behaves when bound to a state.



The function `queue` contains a block diagram that increments a counter by 1 each time the function executes.



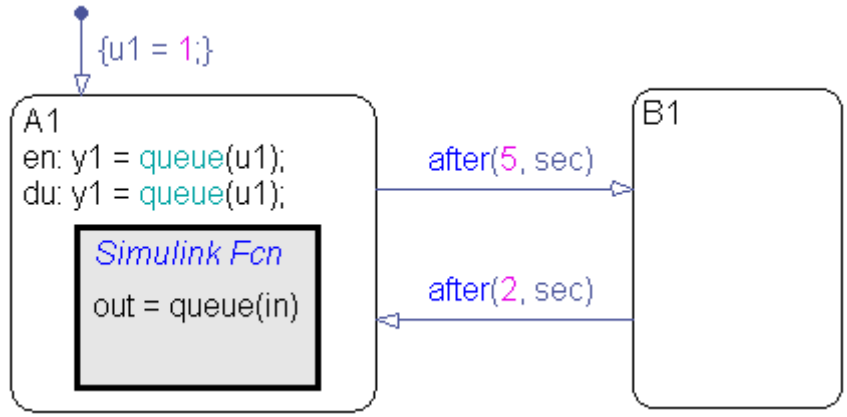
The Block Parameters dialog box for the trigger port appears as follows.



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.

Note If you use a fixed-step solver, the value in the **Sample time** field must be an integer multiple of the fixed-step size. This restriction does not apply to variable-step solvers. (See "Solvers" in the Simulink documentation.)

Simulation Behavior of the Chart

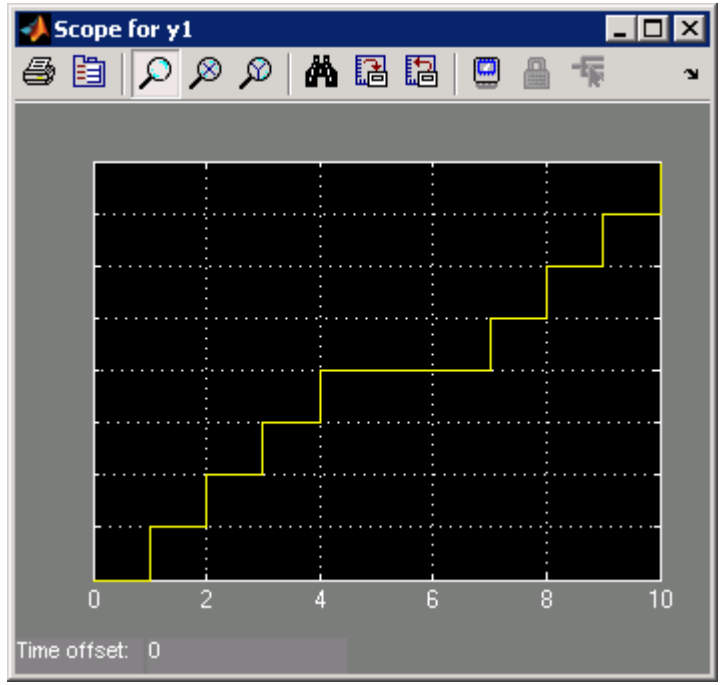


When you simulate the chart, the following actions occur.

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

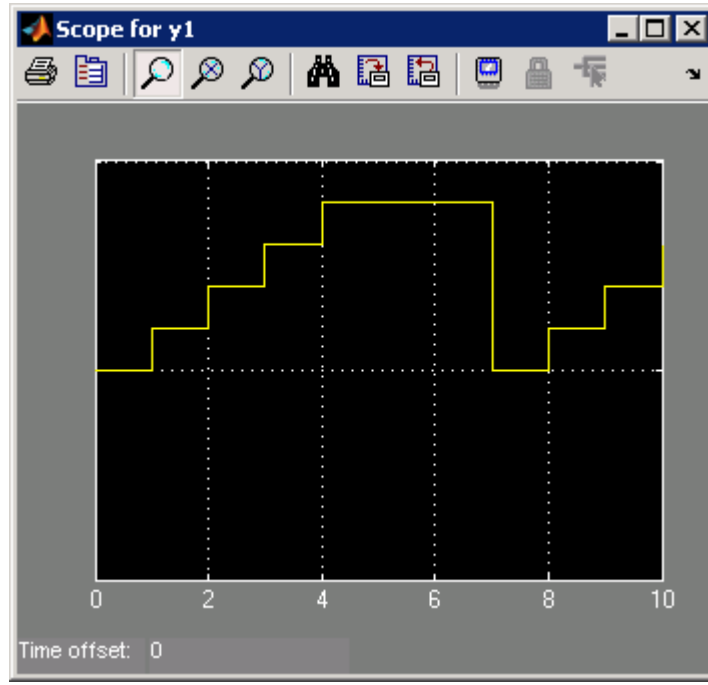
If you set the option **States when enabling** to `held`, the output `y1` is as follows.



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

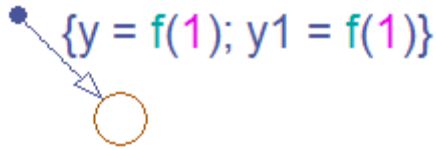
If you set the option **States when enabling** to `reset`, the output `y1` is as follows.

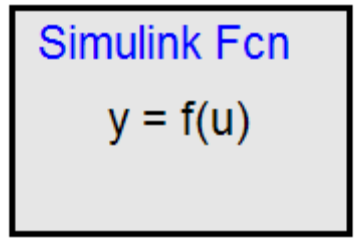


When state A1 becomes inactive at t = 5, the Simulink function does *not* hold the counter value. When A1 is active again at t = 7, the counter resets to zero. Therefore, the output `y1` resets too.

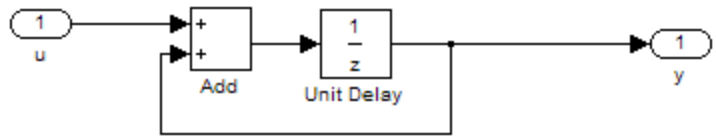
How a Simulink Function Behaves When Called from Multiple Sites

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 "Function-Call Subsystems" in the Simulink documentation.

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 Chapter 13, "Modeling Continuous-Time Systems in Stateflow Charts".

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 in the Simulink Reference documentation.

Do not export Simulink functions

If you try to export Simulink functions, an error message appears when you simulate your model. To avoid this problem, clear the **Export Chart Level Graphical Functions (Make Global)** 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. 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. See “Design Rules for Moore Charts” on page 6-15 for more information.

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. Simulink functions are not supported for use with HDL code generation.

Best Practices for Using Simulink Functions

Place a Simulink function at the lowest possible level of the Stateflow hierarchy

This guideline enables binding of a Simulink function only to the state and substates that require access. You also enhance readability of the chart.

Set properties of input ports explicitly for a Simulink function

The input ports of a Simulink function cannot inherit their sizes and data types. Therefore, you must set sizes and types explicitly when the inputs are not scalar data of type `double`.

The output ports of a Simulink function can inherit sizes and data types based on connections inside the subsystem. Therefore, you can specify sizes and types of outputs as inherited.

Tip To minimize updates required for changes in input port properties, you can specify sizes and data types as parameters.

Avoid using machine-parented data with Simulink functions

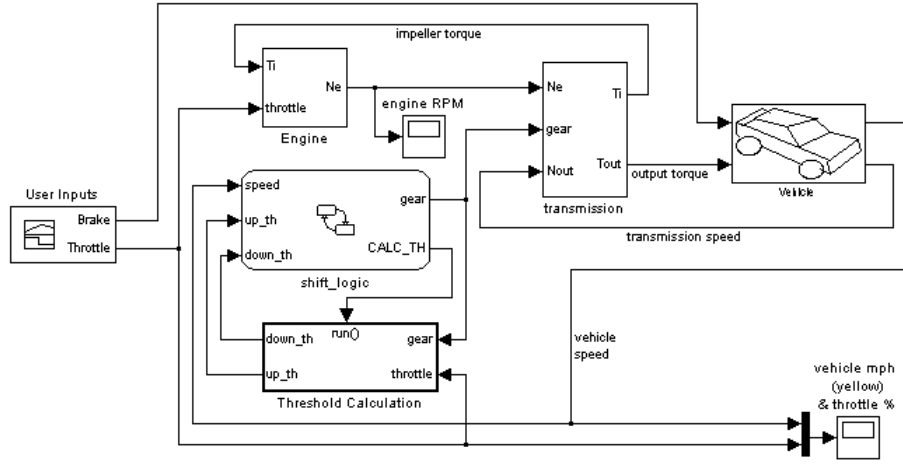
Use data store memory instead of machine-parented data. For more information, see Chapter 8, “Defining Data”.

Example of Defining a Function That Uses Simulink Blocks

In this section...

Goal of the Example

The goal of this example is to use a Simulink function in a Stateflow chart to improve the design of a model named `old_sf_car`.

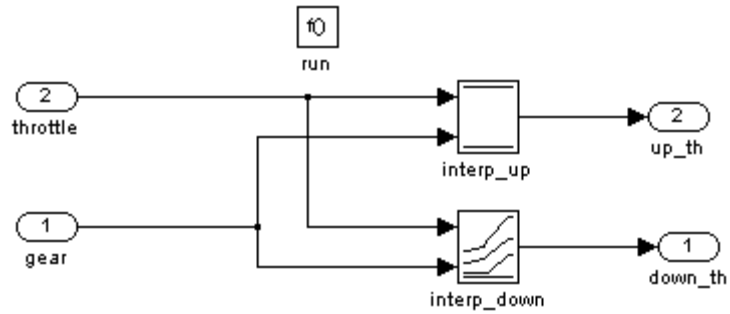


Rationale for Improving the Model Design

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, as shown.



- The subsystem outputs (`up_th` and `down_th`) feed directly into the chart as inputs.

  Note that 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.

Editing a Model to Use a Simulink Function

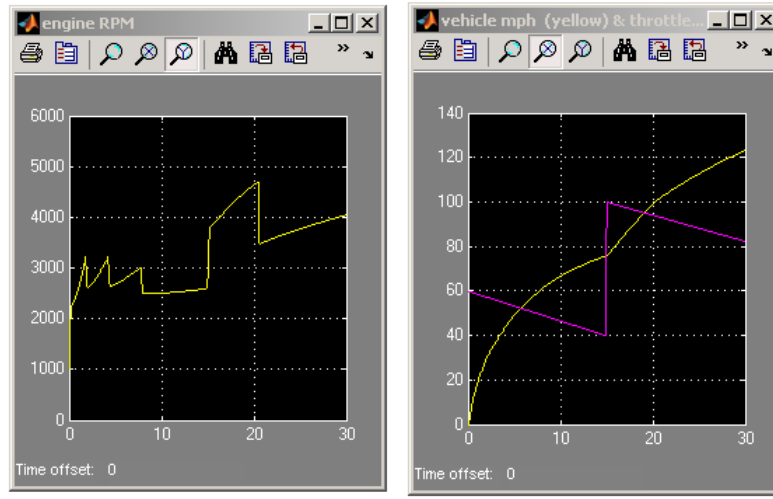
The sections that follow describe how to replace a function-call subsystem in a Simulink model with a Simulink function in a Stateflow chart. This procedure reduces the number of objects in the model while retaining the same simulation results.

Task	Description	Reference Section
1	Open the model.	"Open the Model" on page 21-27
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 21-27
3	Change the scope of specific chart-level data to `Local`.	"Change the Scope of Chart Data" on page 21-29

Task	Description	Reference Section
4	Replace the event broadcast with a function call.	"Update State Action in the Chart" on page 21-29
5	Verify that function inputs and outputs are defined.	"Add Data to the Chart" on page 21-30
6	Remove unused items in the model.	"Remove Unused Items in the Model" on page 21-30

Open the Model

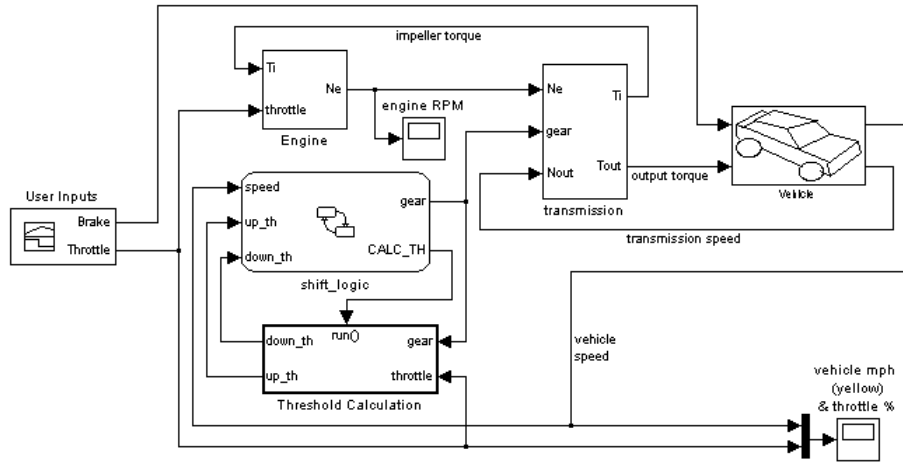
Type `old_sf_car` at the MATLAB command prompt. If you simulate the model, you see these results in the two scopes.



Add a Simulink Function to the Chart

Follow these steps to add a Simulink function to the shift_logic chart.

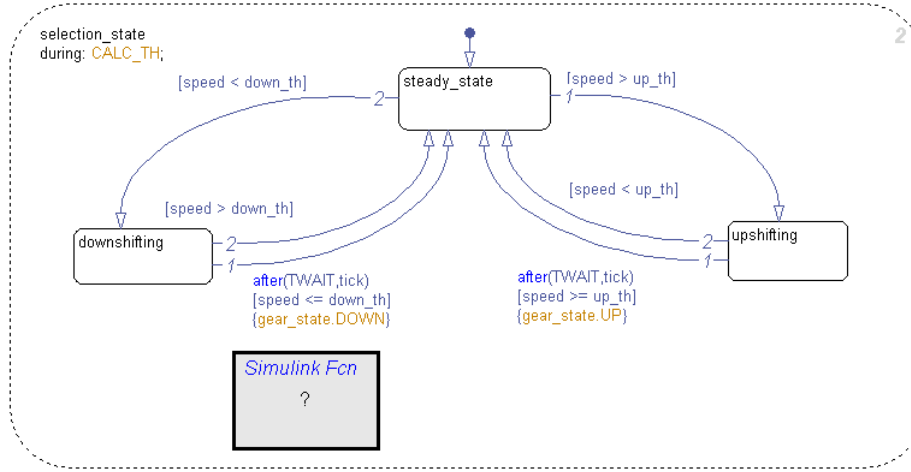
1 In the Simulink model, double-click the Threshold Calculation block in the lower left corner.



2 Copy all subsystem elements except for the trigger port.

3 Open the shift_logic chart.

4 In the Stateflow Editor, add a Simulink function to the state `selection_state`.



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 "How a Simulink Function Binds to a State" on page 21-13.

5. Double-click the Simulink function and paste the subsystem elements of the Threshold Calculation block.

6. Rename the Simulink function from the default `simfcn` to `calc_threshold`.

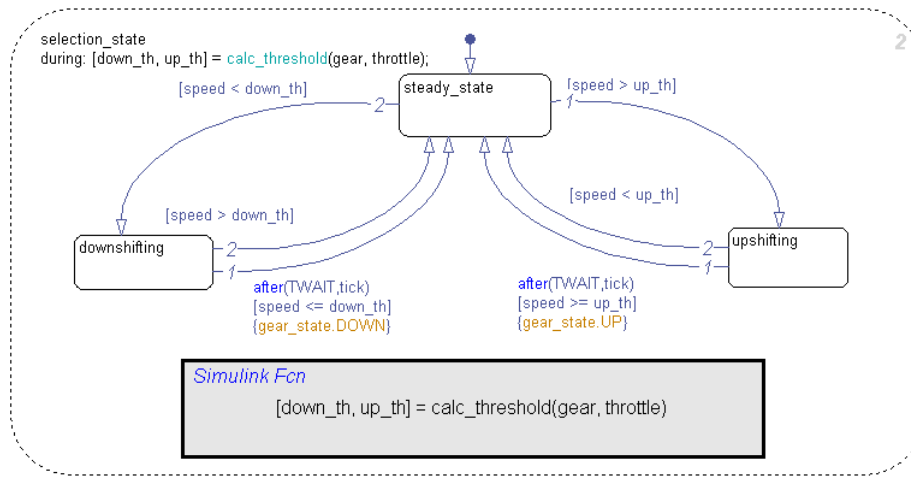
Change the Scope of Chart Data

In the Model Explorer, you must 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 "Adding Data" on page 8-2.)

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.

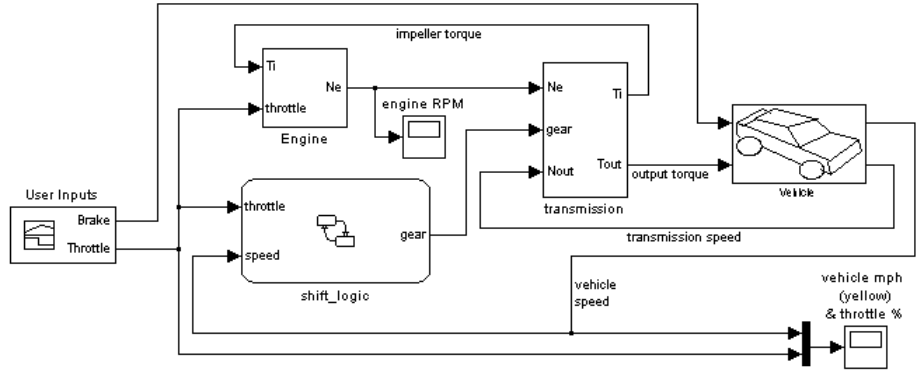
Remove Unused Items in the Model

1. In the Simulink model, delete the Threshold Calculation block and any dashed signal lines.

2. In the Model Explorer, you can delete the function-call output event `CALC_TH` because the Threshold Calculation block no longer exists.

Running the New Model

Your new model should look something like this.



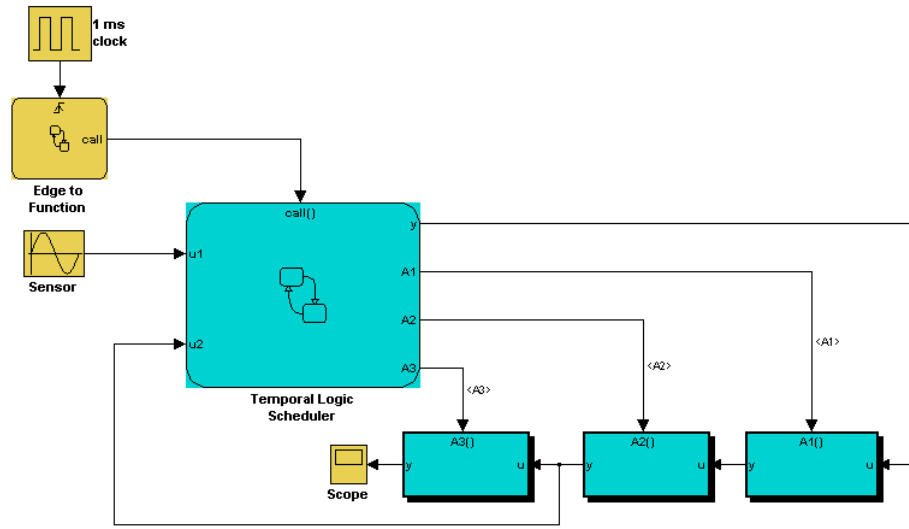
If you simulate the new model, the results match those of the original design.

Example of Scheduling Execution of Multiple Controllers

In this section...

Goal of the Example

The goal of this example 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.

  Note that 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.

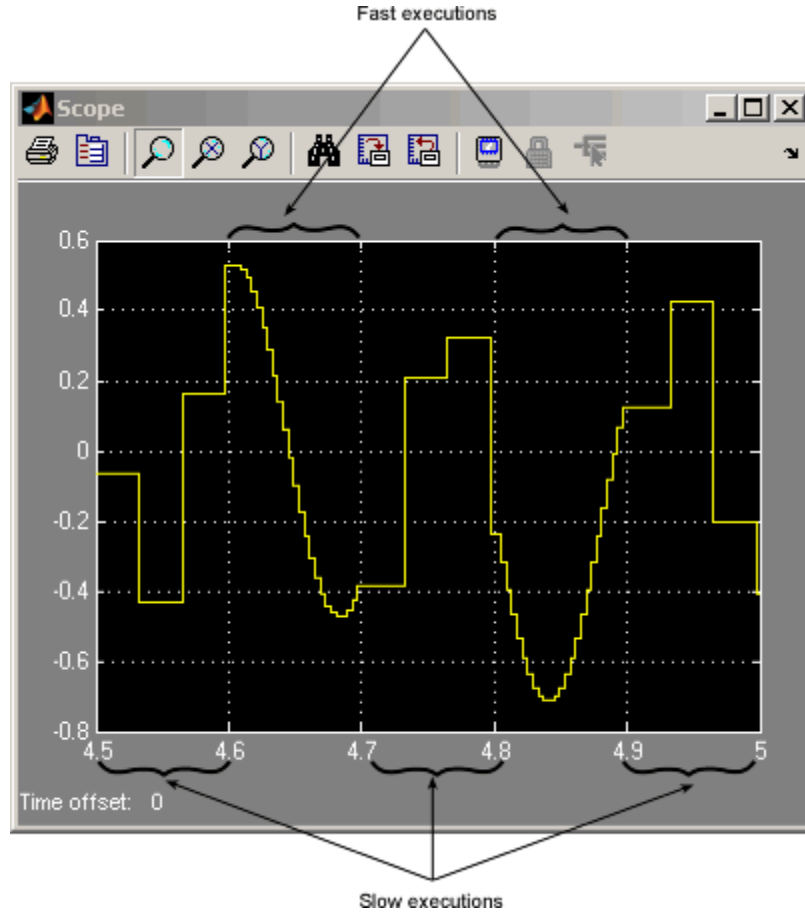
Editing a Model to Use Simulink Functions

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.

Task	Description	Reference Section
1	Open the model.	"Open the Demo Model" on page 21-33
2	Move the contents of the function-call subsystems into Simulink functions in the chart.	"Add Simulink Functions to the Chart" on page 21-33
3	Change the scope of specific chart-level data to `Local`.	"Change the Scope of Chart Data" on page 21-35
4	Replace event broadcasts with function calls.	"Update State Actions in the Chart" on page 21-36
5	Verify that function inputs and outputs are defined.	"Add Data to the Chart" on page 21-36
6	Remove unused items in the model.	"Remove Unused Items in the Model" on page 21-37

Open the Demo Model

Type `sf_temporal_logic_scheduler` at the MATLAB command prompt. If you simulate the model, you see this result in the scope.

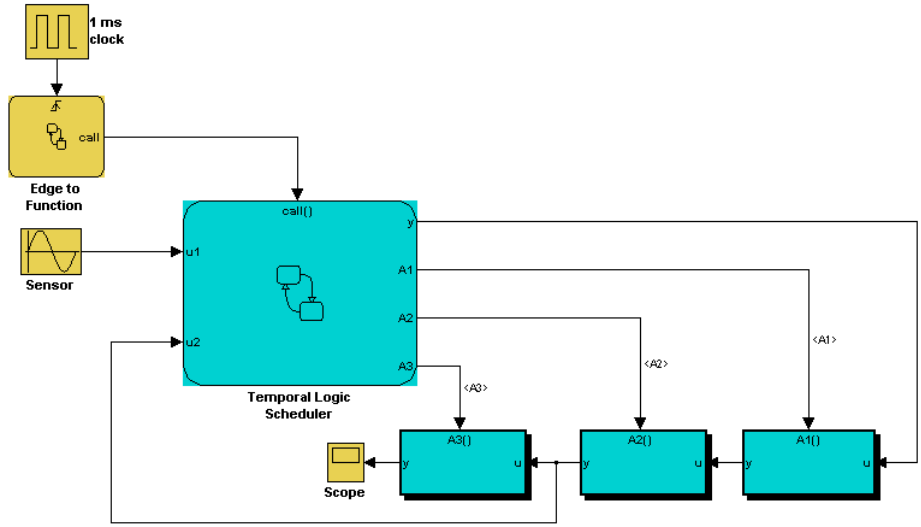


For more information, see “Scheduling Subsystems to Execute at Specific Times Using a Temporal Logic Scheduler” on page 18-15.

Add Simulink Functions to the Chart

Follow these steps to add Simulink functions to the Temporal Logic Scheduler chart.

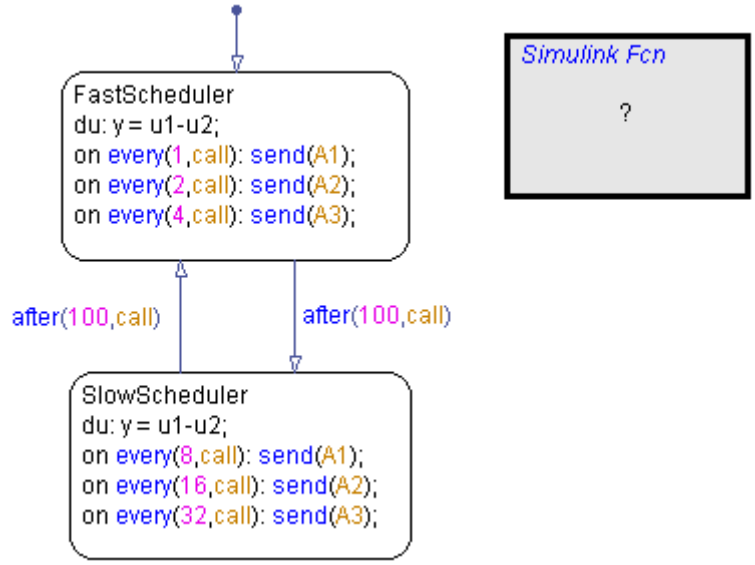
1 In the Simulink model, double-click the A1 block in the lower right corner.



2 Copy all subsystem elements except for the trigger port.

3 Open the Temporal Logic Scheduler chart.

4 In the Stateflow Editor, add a Simulink function to the chart, outside of any states.



Note The function resides at the chart level because both states `FastScheduler` and `SlowScheduler` require access to the function output.

5. Double-click the Simulink function and paste the subsystem elements of the A1 block.

6. Rename the Simulink function from the default `simfcn` to `f1`.

7. Repeat steps 1 through 6 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`.

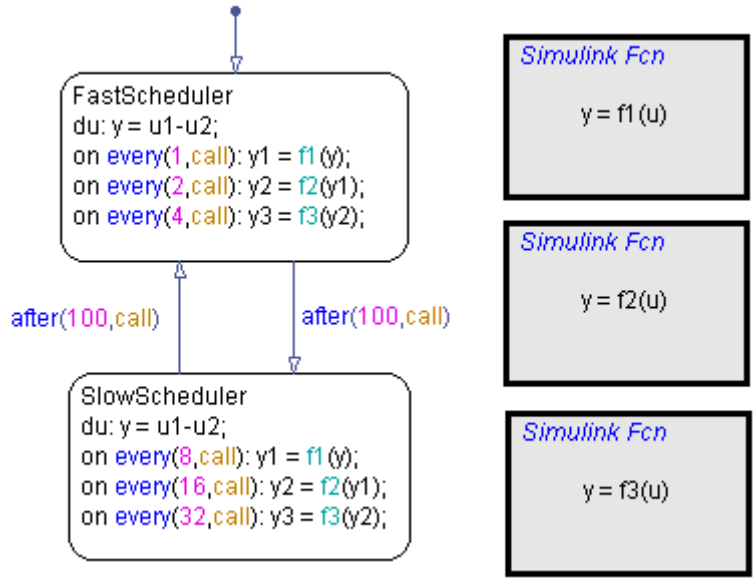
Change the Scope of Chart Data

In the Model Explorer, you must 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`, you must define three data. (For details, see "Adding Data" on page 8-2.)

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.

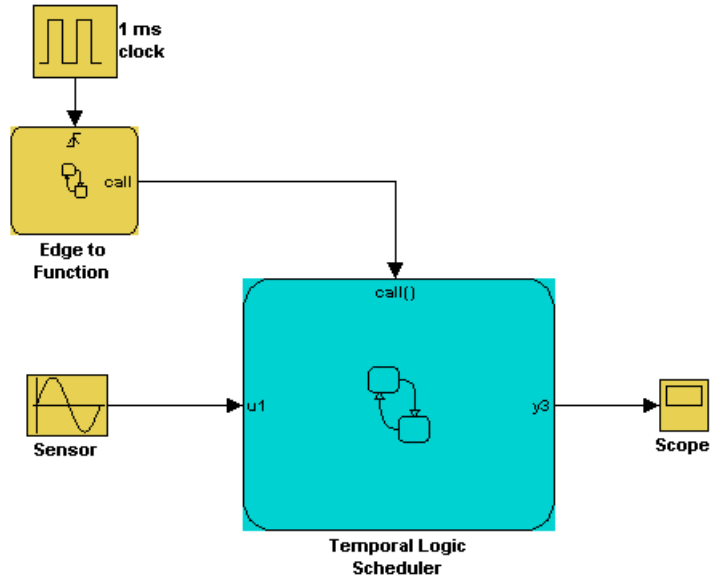
Note To flip the Scope block, right-click and select **Format > Flip Block** from the context menu.

Remove Unused Items in the Model

1. In the Simulink model, delete the A1, A2, and A3 subsystem blocks and any dashed signal lines.
2. In the Model Explorer, you can delete output events `A1`, `A2`, and `A3` and input data `u2` because the function-call subsystems no longer exist.

Running the New Model

Your new model should look something like this.



If you simulate the new model, the results match those of the original design.

22

Building Targets

Targets You Can Build

In this section...

Code Generation for Stateflow Charts and Truth Table Blocks

You can generate code for models with Stateflow charts and Truth Table blocks for these uses:

- Simulation
- Production and rapid prototyping

Code Generation for Simulation

A *simulation target* is a specification of the generated code, custom code, and build type you use for generating simulation code for Chart and Truth Table blocks in a model.

Whenever you simulate a model that contains Stateflow blocks, Stateflow software generates code that compiles into an S-function MEX file (for details, see "S-Function MEX-Files" on page 22-73). This code enables the Stateflow blocks to interface with other blocks in a Simulink model, the MATLAB base workspace, and the Stateflow Debugger. This code is not suitable for production or rapid prototyping.

Code Generation for Production and Rapid Prototyping

An *embeddable target* is a specification of the generated code, custom code, and build type you use for generating production code for Chart and Truth Table blocks in a model.

Stateflow Coder and Real-Time Workshop software can work together to generate embeddable code for Stateflow blocks. This code is optimized for production and rapid prototyping, but does not contain code to interface with

other blocks in a Simulink model, the MATLAB base workspace, and the Stateflow Debugger.

Software Requirements for Building Targets

To build targets for models with Stateflow charts or Truth Table blocks, you must have a license for the software listed:

Target to Build	Software to Use
Simulation target	Stateflow
Embeddable target	Stateflow Coder and Real-Time Workshop

The default target type of Real-Time Workshop code generation is generic real-time (`grt`). To build other embeddable targets, you must have the appropriate license. See “Available Targets” in the Real-Time Workshop User’s Guide for more information.

Choosing a Procedure to Simulate a Model

In this section...

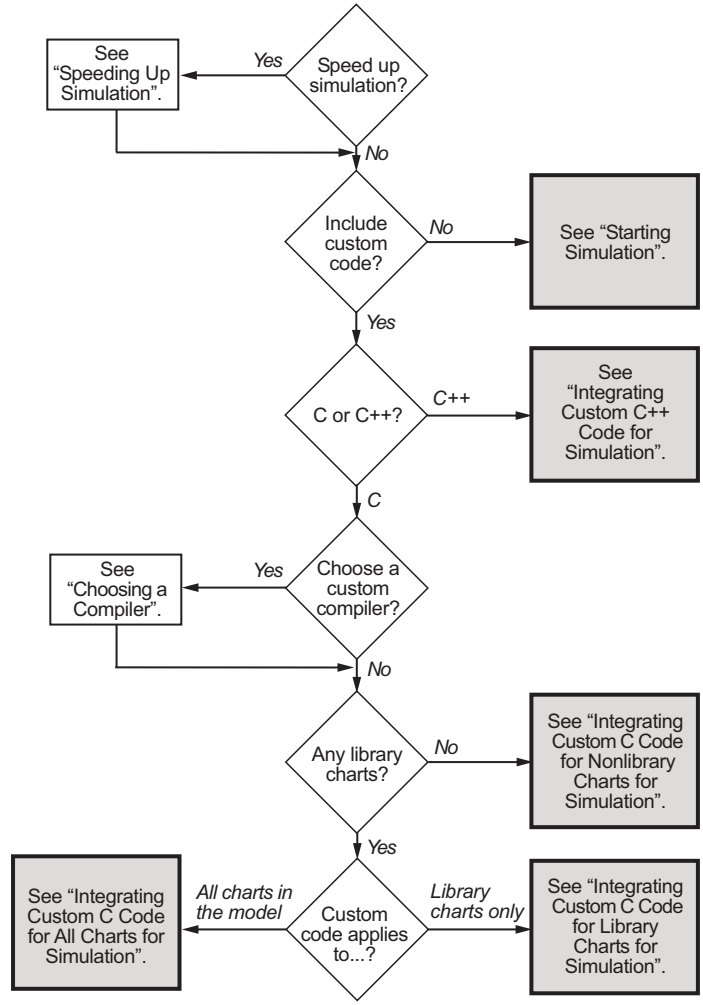
Guidelines for Simulation

When you simulate a 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 "Speeding Up Simulation" on page 22-17.
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.
Choose a custom compiler	You use the UNIX version of Stateflow software or do not wish to use the default `lcc` compiler. See "Choosing a Compiler" on page 22-42.
Include custom code only for library charts	You want to provide custom code in a portable, self-contained library for use in multiple models.

Choosing the Right Procedure for Simulation

To choose the right procedure for simulation, find the highlighted block that describes your goal and see the corresponding section in "Procedures for Simulation" on page 22-7. These procedures apply to models that contain Chart or Truth Table blocks.


Speed up simulation?
Yes
See "Speeding Up Simulation".
No
Include custom code?
No
See "Starting Simulation".
Yes
C or C++?
C++
See "Integrating Custom C++ Code for Simulation".
C
Choose a custom compiler?
Yes
See "Choosing a Compiler".
No
Any library charts?
No
See "Integrating Custom C Code for Nonlibrary Charts for Simulation".
Yes
Custom code applies to...?
All charts in the model
See "Integrating Custom C Code for All Charts for Simulation".
Library charts only
See "Integrating Custom C Code for Library Charts for Simulation".

Procedures for Simulation

In this section...

Starting Simulation

Simulate your model in one of these ways:

- Click the play button in the toolbar of the Simulink model window or the Stateflow Editor.
- Select **Simulation > Start** in the Simulink model window or the Stateflow Editor.

See "Generated Code Files for Targets You Build" on page 22-73 for details about the simulation code you generate for your model and its directory structure.

For information on setting simulation options using the command-line API, see "Using the Command-Line API to Set Parameters for Simulation and Embeddable Code Generation" on page 22-32.

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.

Integrating Custom C++ Code for Simulation

To integrate custom C++ code for simulation, perform the tasks that follow.

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. In the Stateflow Editor, select **Tools > Open Simulation Target**.

2. In the Configuration Parameters dialog box, select the **Simulation Target > Custom Code** 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 Configuration Parameters dialog box, select the **Real-Time Workshop** pane.

6 Select `C++` from the **Language** menu.

7 Click **OK**.

Task 3: Choose a C++ Compiler

For instructions, see "Choosing a Compiler" on page 22-42.

Task 4: Simulate the Model

For instructions, see "Starting Simulation" on page 22-7.

Integrating Custom C Code for Nonlibrary Charts for Simulation

To integrate custom C code that applies to nonlibrary charts for simulation, perform the tasks that follow.

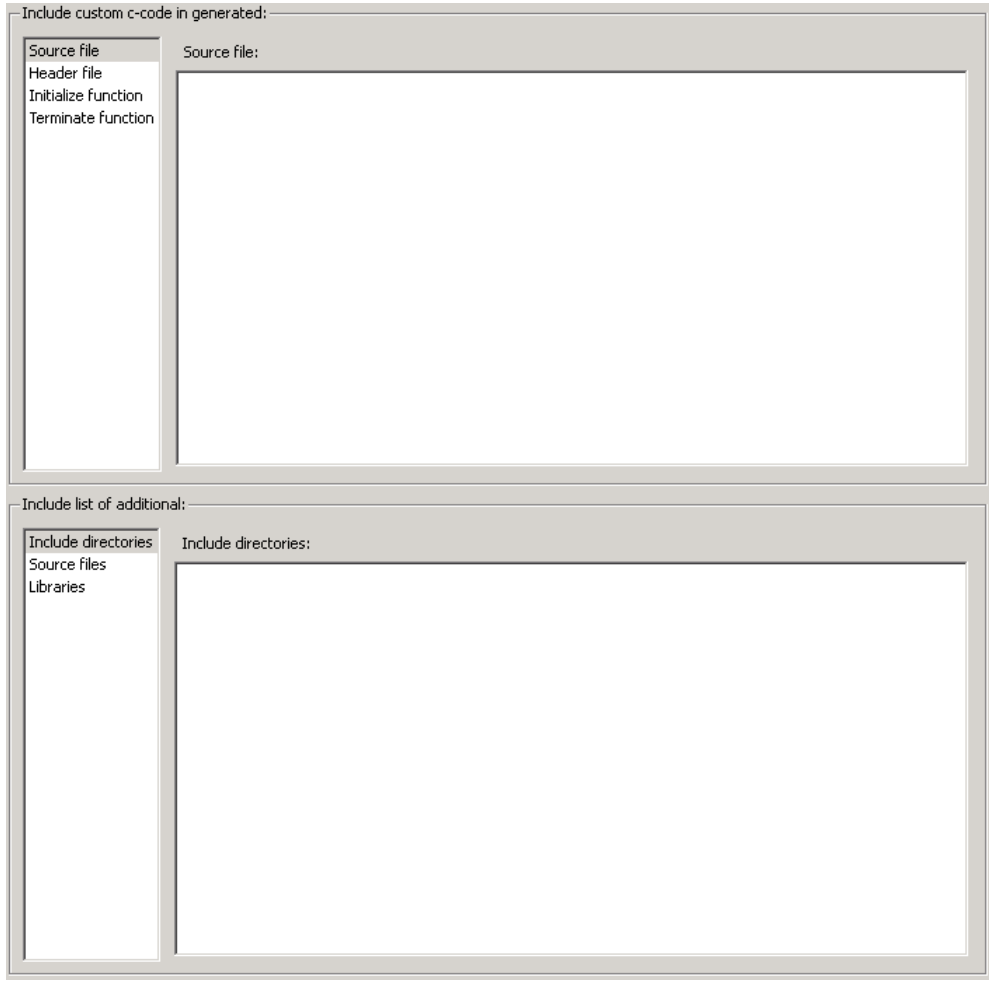
Task 1: Include Custom C Code in the Simulation Target

Specify custom code options in the simulation target for your model:

1 In the Stateflow Editor, select **Tools > Open Simulation Target**.

2 In the Configuration Parameters dialog box, select the **Simulation Target > Custom Code** pane.

The custom code options appear.



3 Specify your custom code in the subpanes.

Follow the guidelines in "Specifying Relative Paths for Custom Code" on page 22-40.

- **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 directory 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 **Tools > Rebuild All** in the Stateflow Editor.

If you want to build the target only for the parts of a chart that have changed since the previous build, select **Tools > Build Diagram** in the Stateflow Editor.

Task 2: Simulate the Model

For instructions, see "Starting Simulation" on page 22-7.

Integrating Custom C Code for Library Charts for Simulation

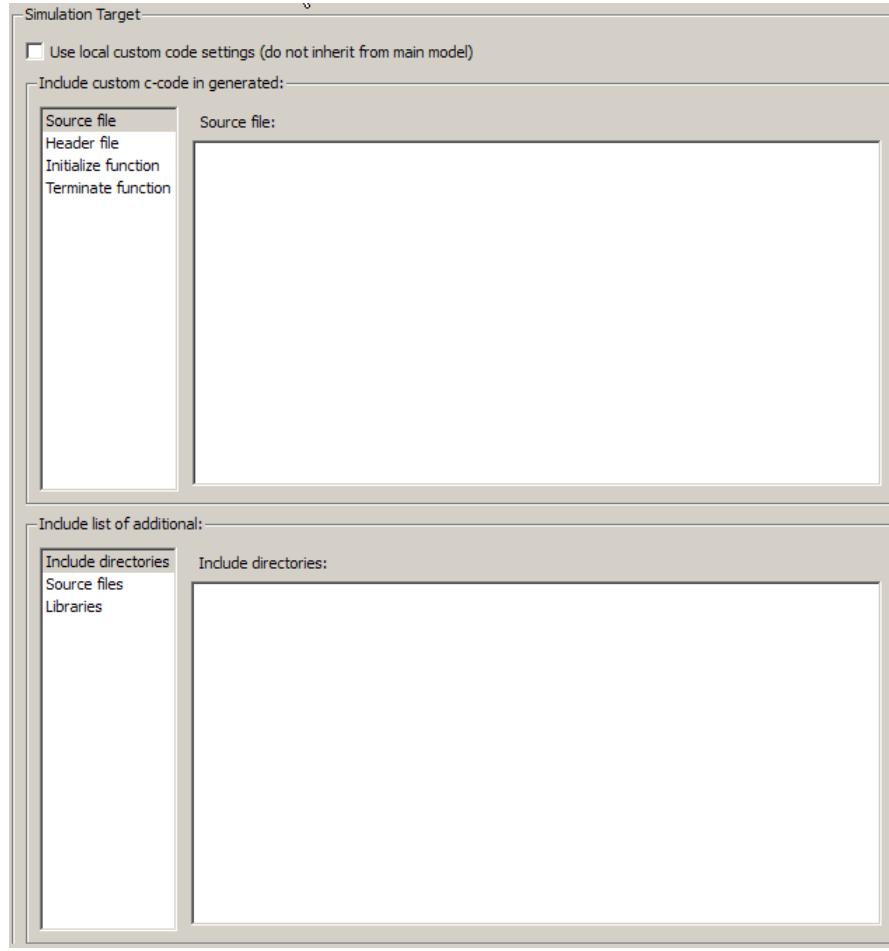
To integrate custom C code that applies only to library charts for simulation, perform the tasks that follow.

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 **Tools > Open Simulation Target**.

The Configuration Parameters dialog box appears.



2 In the **Simulation Target** pane, select **Use local custom code settings (do not inherit from main model)**.

This action 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 "Specifying Relative Paths for Custom Code" on page 22-40.

Note See "Task 1: Include Custom C Code in the Simulation Target" on page 22-9 for descriptions of the custom code options.

---

4 Click **OK**.

Task 2: Simulate the Model

For instructions, see "Starting Simulation" on page 22-7.

Integrating Custom C Code for All Charts for Simulation

To integrate custom C code that applies to all charts for simulation, perform the tasks that follow.

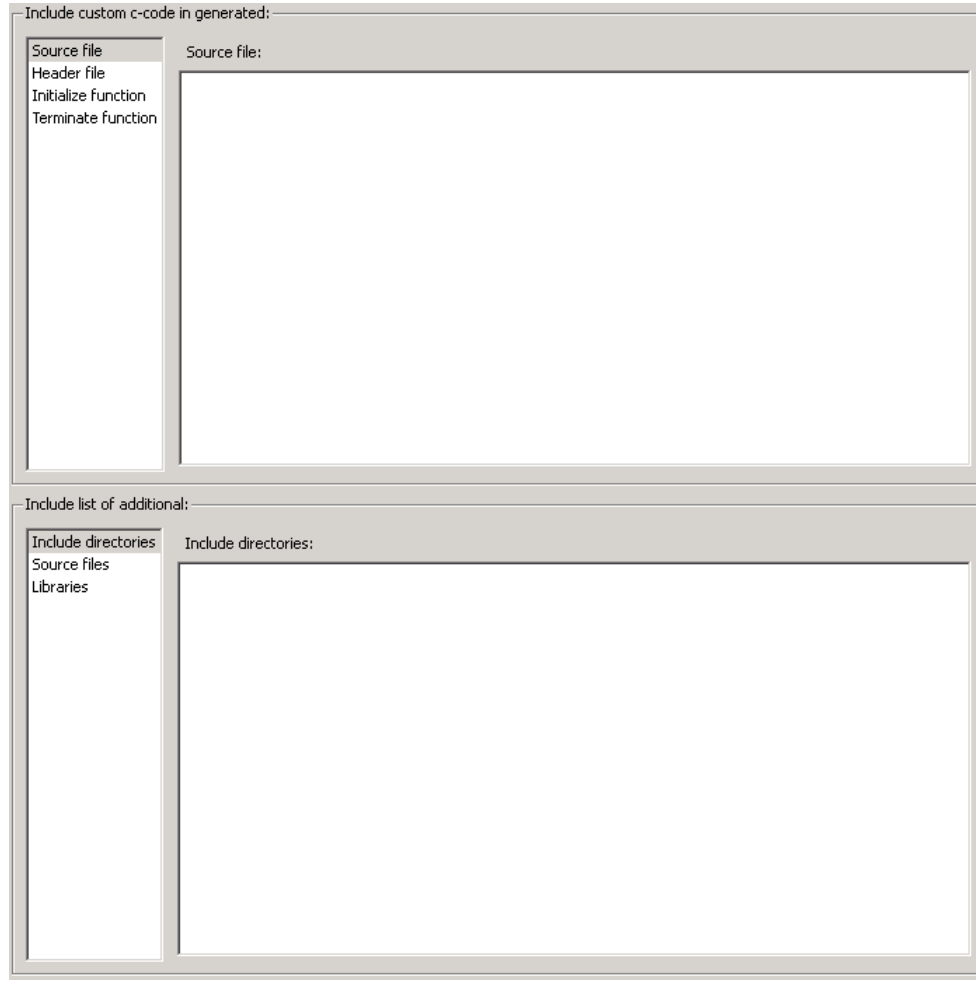
Task 1: Include Custom C Code in the Simulation Target for the Main Model

Specify custom code options in the simulation target for your main model:

1 In the Stateflow Editor, select **Tools > Open Simulation Target**.

2 In the Configuration Parameters dialog box, select the **Simulation Target > Custom Code** pane.

The custom code options appear.



3 Specify your custom code in the subpanes.

Follow the guidelines in "Specifying Relative Paths for Custom Code" on page 22-40.

Note See "Task 1: Include Custom C Code in the Simulation Target" on page 22-9 for descriptions of the custom code options.

4 Click **OK**.

By default, settings in the **Simulation Target > Custom Code** 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 **Tools > Rebuild All** in the Stateflow Editor.

If you want to build the target only for the parts of a chart that have changed since the previous build, select **Tools > Build Diagram** 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 **Tools > Open Simulation Target**.

The Configuration Parameters dialog box appears.

2 In the **Simulation Target** pane, clear the **Use local custom code settings (do not inherit from main model)** check box.

This action ensures that library charts inherit the custom code settings of your main model.

3 Click **OK**.

Task 3: Simulate the Model

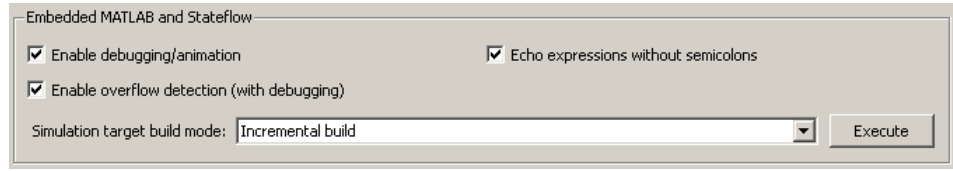
For instructions, see "Starting Simulation" on page 22-7.

Speeding Up Simulation

To simulate your model more quickly, disable options as described in the steps that follow:

1 In the Stateflow Editor, select **Tools > Open Simulation Target**.

The **Simulation Target** pane appears in the Configuration Parameters dialog box.



2 Clear any of these options:

- **Enable debugging/animation** — Clear this check box to disable chart animation and debugging.

  This option enables automatically when you use the Stateflow Debugger to start a model simulation. You can also control chart animation separately in the Debugger. (The Stateflow Debugger works only with simulation targets. Therefore, you cannot generate debugging/animation code for embeddable targets, even if you enable this option.)

- **Enable overflow detection (with debugging)** — Clear this check box to disable overflow detection of Stateflow data in the generated code. Overflow occurs for data when a value is assigned to it that exceeds the numeric capacity of its type.

Note The **Enable overflow detection (with debugging)** option is important for fixed-point data. For more information, see "Overflow Detection for Fixed-Point Types" on page 14-10.

To detect overflow in data during simulation, you must also select the **Data Range** check box in the Debugger window. See "Debugging Data Range Violations in a Chart" on page 23-24 for more details.

- **Echo expressions without semicolons** — Clear this check box to disable run-time output in the MATLAB Command Window, such as actions that do not terminate with a semicolon.

3 Click **OK**.

Choosing a Procedure to Generate Embeddable Code for a Model

In this section...

Guidelines for Embeddable Code Generation

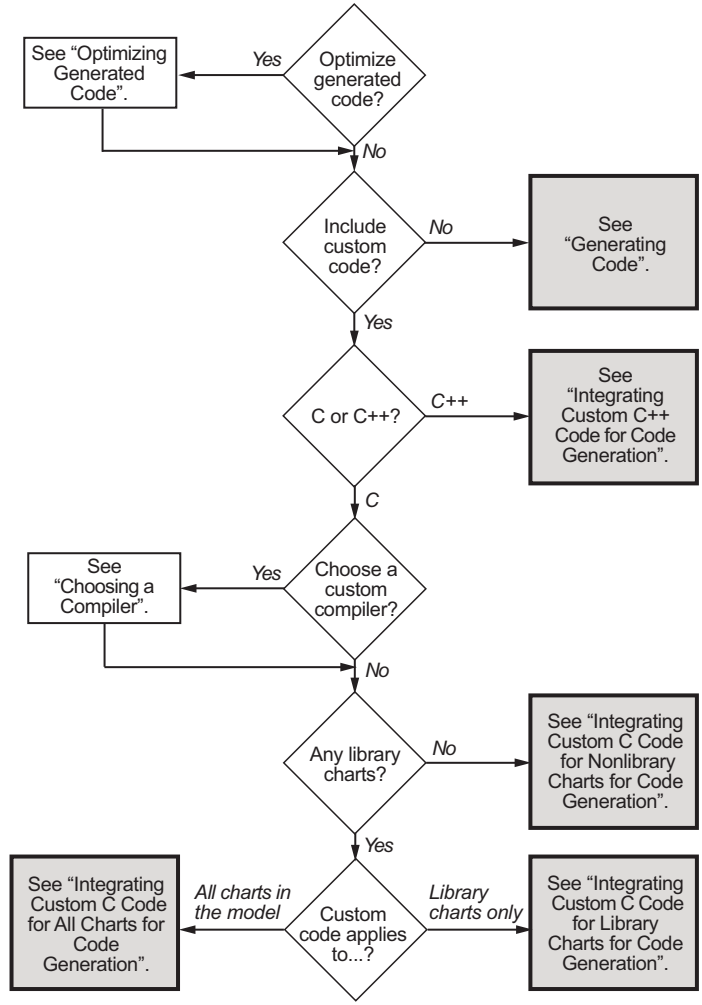
When you generate embeddable code for a model, use these guidelines to choose the right procedure.

Do this step...	When...
Optimize generated code	You want to improve readability of the code and reduce the amount of memory storage required. See "Optimizing Generated Code" on page 22-30.
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.
Choose a custom compiler	You use the UNIX version of Stateflow software or do not wish to use the default `lcc` compiler. See "Choosing a Compiler" on page 22-42.
Include custom code only for library charts	You want to provide custom code in a portable, self-contained library for use in multiple models.

Choosing the Right Procedure for Embeddable Code Generation

To choose the right procedure for embeddable code generation, find the highlighted block that describes your goal and see the corresponding section

in “Procedures for Embeddable Code Generation” on page 22-21. These procedures apply to models that contain Chart or Truth Table blocks.



Procedures for Embeddable Code Generation

In this section...

Generating Code

Generate embeddable code for your model in one of these ways:

- Use the keyboard shortcut **Ctrl-B** or **Command-B**.
- Click **Build** in the **Real-Time Workshop** pane of the Configuration Parameters dialog box.

See "Generated Code Files for Targets You Build" on page 22-73 for details about the embeddable code you generate for your model and its directory structure.

For information on setting code generation options using the command-line API, see "Using the Command-Line API to Set Parameters for Simulation and Embeddable Code Generation" on page 22-32.

Note You cannot generate embeddable code only for the Stateflow blocks in a library model. You must first create a link to the library block in your main model and then generate code for the main model.

Integrating Custom C++ Code for Code Generation

To integrate custom C++ code for embeddable code generation, perform the tasks that follow.

Task 1: Prepare Code Files

Prepare your custom C++ code for Real-Time Workshop code generation.

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 Real-Time Workshop Code Generation

To include custom C++ code for Real-Time Workshop code generation, perform these steps:

1. In the Stateflow Editor, select **Simulation > Configuration Parameters**.
2. In the Configuration Parameters dialog box, select the **Real-Time Workshop** pane.
3. Select `C++` from the **Language** menu. Click **Apply**.
4. Select the **Real-Time Workshop > Custom Code** pane.
5. Add your custom header file in the **Header file** subpane. Click **Apply**.
6. Add your custom C++ files in the **Source files** subpane.
7. Click **OK**.

Task 3: Choose a C++ Compiler

For instructions, see "Choosing a Compiler" on page 22-42.

Task 4: Generate Code

For instructions, see "Generating Code" on page 22-21.

Integrating Custom C Code for Nonlibrary Charts for Code Generation

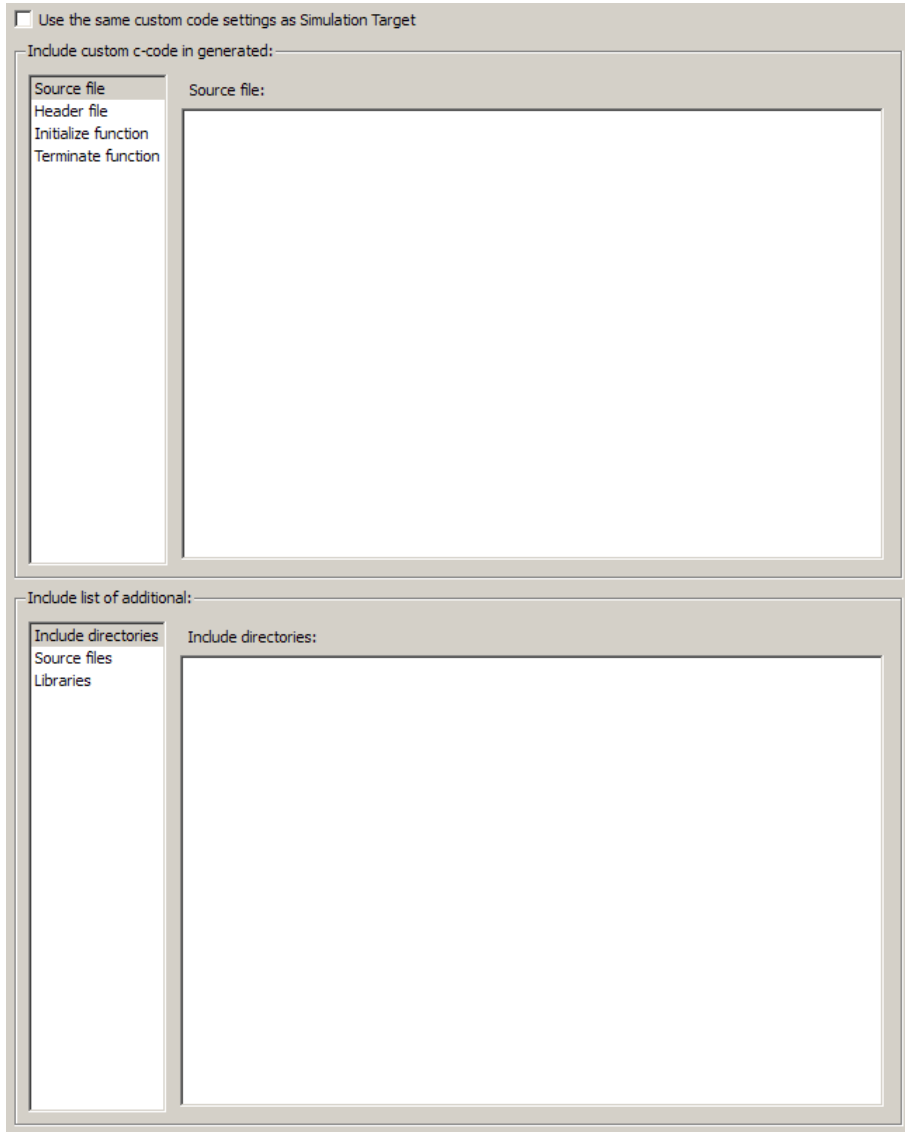
To integrate custom C code that applies to nonlibrary charts for embeddable code generation, perform the tasks that follow.

Task 1: Include Custom C Code for Embeddable Code Generation

Specify custom code options for Real-Time Workshop code generation of your model:

1. In the Stateflow Editor, select **Simulation > Configuration Parameters**.
2. In the Configuration Parameters dialog box, select **Real-Time Workshop > Custom Code**.

The custom code options appear.



3 Specify your custom code in the subpanes.

Follow the guidelines in "Specifying Relative Paths for Custom Code" on page 22-40.

Note If you specified custom code settings for simulation, you can apply these settings to Real-Time Workshop code generation. To avoid entering the same information twice, select **Use the same custom code settings as Simulation Target**.

Task 2: Generate Code

For instructions, see "Generating Code" on page 22-21.

Integrating Custom C Code for Library Charts for Code Generation

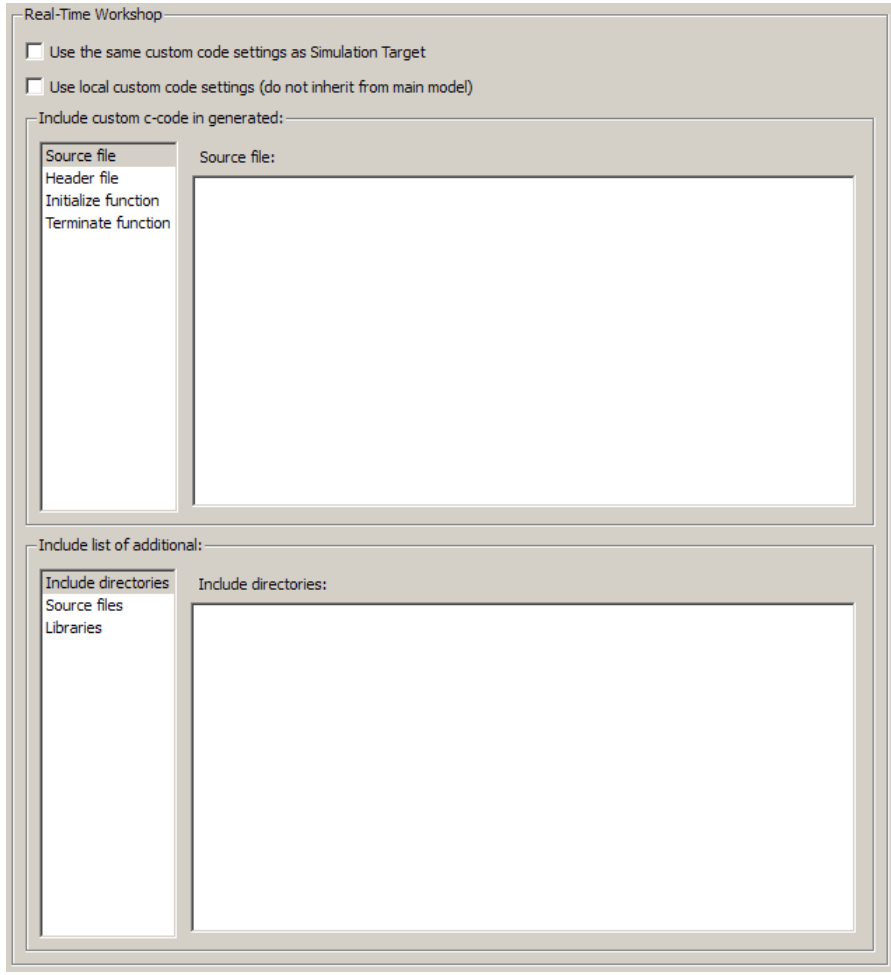
To integrate custom C code that applies only to library charts for embeddable code generation, perform the tasks that follow.

Task 1: Include Custom C Code in Embeddable Targets for Library Models

Specify custom code options in the embeddable target for each library model that contributes a chart to your main model:

1 In the Stateflow Editor, select **Tools > Open RTW Target**.

The Configuration Parameters dialog box appears.



2 In the **Real-Time Workshop** pane, select **Use local custom code settings (do not inherit from main model)**.

This action ensures that each library model retains its own custom code settings during Real-Time Workshop code generation.

3 Specify your custom code in the subpanes.

Follow the guidelines in "Specifying Relative Paths for Custom Code" on page 22-40.

Note If you specified custom code settings for simulation, you can apply these settings to Real-Time Workshop code generation. To avoid entering the same information twice, select **Use the same custom code settings as Simulation Target**.

---

4 Click **OK**.

Task 2: Generate Code

For instructions, see "Generating Code" on page 22-21.

Integrating Custom C Code for All Charts for Code Generation

To integrate custom C code that applies to all charts for embeddable code generation, perform the tasks that follow.

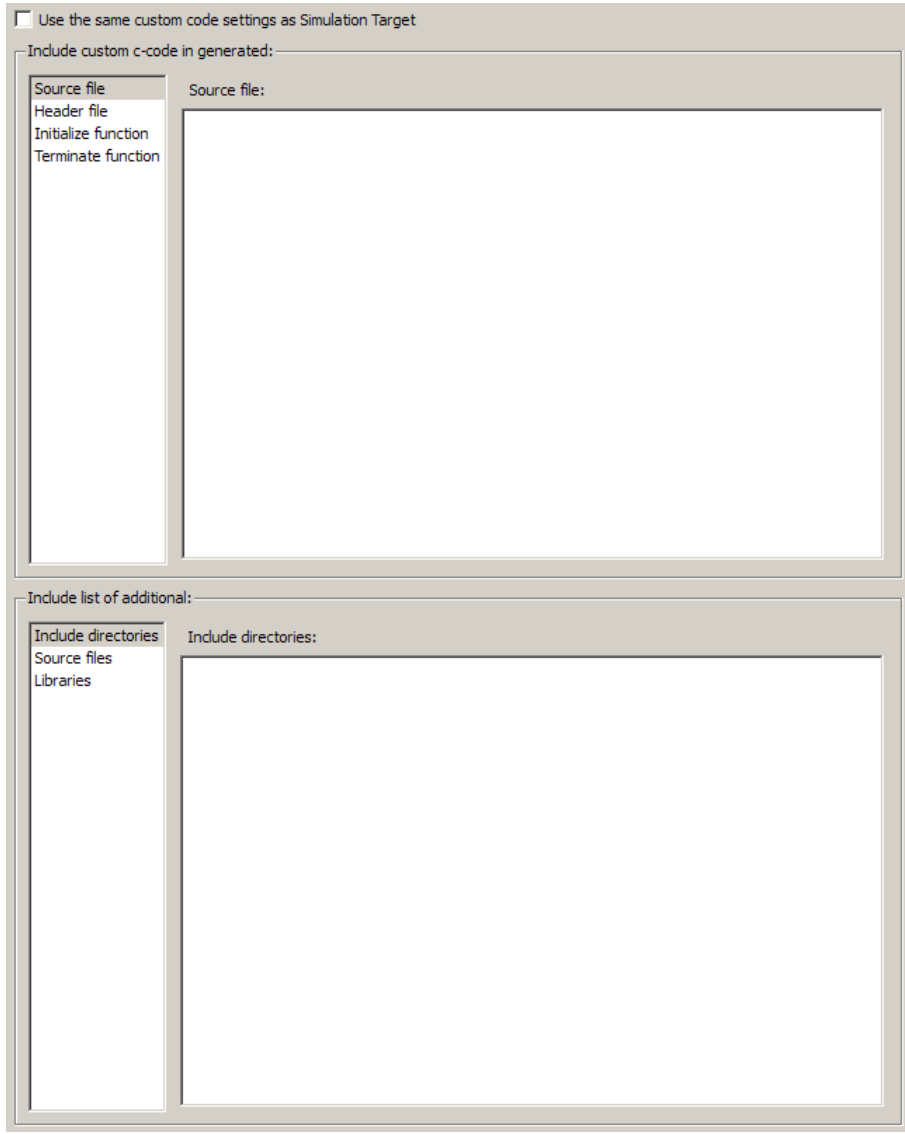
Task 1: Include Custom C Code for Embeddable Code Generation of the Main Model

Specify custom code options for Real-Time Workshop code generation of your main model:

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

2 In the Configuration Parameters dialog box, select **Real-Time Workshop > Custom Code**.

The custom code options appear.



3 Specify your custom code in the subpanes.

Follow the guidelines in "Specifying Relative Paths for Custom Code" on page 22-40.

Note If you specified custom code settings for simulation, you can apply these settings to Real-Time Workshop code generation. To avoid entering the same information twice, select **Use the same custom code settings as Simulation Target**.

Task 2: Ensure That Custom C Code for the Main Model Applies to Library Charts

Configure the embeddable target for each library model that contributes a chart to your main model:

1. In the Stateflow Editor, select **Tools > Open RTW Target**.
2. In the **Real-Time Workshop** pane, clear the **Use local custom code settings (do not inherit from main model)** check box.

   This action ensures that library charts inherit the custom code settings of your main model.
3. Click **OK**.

Task 3: Generate Code

For instructions, see "Generating Code" on page 22-21.

Optimizing Generated Code

In this section...

How to Optimize Generated Code for Embeddable Targets

To optimize Real-Time Workshop code generation for your model, perform these steps:

1. In the Stateflow Editor, select **Simulation > Configuration Parameters**.
2. In the Configuration Parameters dialog box, select the **Optimization** pane.
3. In the **Stateflow** section of the **Optimization** pane, choose from these options:
   - **Use bitsets for storing state configuration** — Reduces the amount of memory that stores state configuration variables. However, it can increase the amount of memory that stores target code if the target processor does not include instructions for manipulating bitsets.
   - **Use bitsets for storing boolean data** — Reduces the amount of memory that stores Boolean variables. However, it can increase the amount of memory that stores target code if the target processor does not include instructions for manipulating bitsets.

Note You cannot use bitsets when you generate code for these cases:

- an external mode simulation
- a target that specifies an explicit structure alignment

Design Tips for Optimizing Generated Code

The following design tips can help optimize generated code.

Do not access machine-parented data in a graphical function

This restriction prevents long parameter lists from appearing in the code generated for a graphical function. You can access local data that resides in the same chart as the graphical function.

For more information, see "Using Graphical Functions to Extend Actions" on page 7-27.

Be explicit about the inline option of a graphical function

When you use a graphical function in a Stateflow chart, select `Inline` or `Function` for the property **Function Inline Option**. Otherwise, the code generated for a graphical function may not appear as you want.

For more information, see "Specifying Graphical Function Properties" on page 7-44.

Avoid using multiple edge-triggered events in Stateflow charts

If you use more than one edge trigger, you generate multiple source code files to handle rising or falling edge detections. If multiple triggers are required, use function-call events instead.

For more information, see Chapter 9, "Defining Events".

Combine input signals of a chart into a single bus object

When you use a bus object, you reduce the number of parameters in the parameter list of a generated function. This guideline also applies to output signals of a chart.

For more information, see Chapter 17, "Working with Structures and Bus Signals in Stateflow Charts".

Using the Command-Line API to Set Parameters for Simulation and Embeddable Code Generation

In this section...

How to Set Parameters at the Command Line

To programmatically set options in the Configuration Parameters dialog box for simulation and embeddable code generation, you can use the command-line API.

1. At the MATLAB command prompt, type:

   ```
   object_name = getActiveConfigSet(gcs)
   ```

   This command returns an object handle to the model settings in the 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 Reference documentation.

Simulation Parameters for Nonlibrary Models

The following table summarizes the parameters and values in the Configuration Parameters dialog box that you can set for simulation of nonlibrary models using the command-line API. The parameters are listed in the order that they appear in the Configuration Parameters dialog box.

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`SFSimEnableDebug` string – `'off'`, **`on`**	**Simulation Target > Enable debugging / animation**	Enables debugging and animation of a model during simulation and also enables the Stateflow Debugger.
`SFSimOverflowDetection` string – `'off'`, **`on`**	**Simulation Target > Enable overflow detection (with debugging)**	Enables overflow detection of data during simulation. Overflow occurs for data when a value assigned to it exceeds the numeric capacity of the data type. **Note** To enable this option, you must also select the **Data Range** check box in the Stateflow Debugger window.

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`SFSimEcho` string – `'off'`, **`on`**	**Simulation Target > Echo expressions without semicolons**	Enables run-time output to appear in the MATLAB Command Window during simulation.
`SimBuildMode` string – **`sf_incremental_build`**, `'sf_nonincremental_build'`, `'sf_make'`, `'sf_make_clean'`, `'sf_make_clean_objects'`	**Simulation Target > Simulation target build mode**	Specifies how you build the simulation target for a model.
`SimReservedNameArray` *string array* – **`{}`**	**Simulation Target > 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.
`SimCustomSourceCode` *string* –	**Simulation Target > Custom Code > Source file**	Enter code lines to appear near the top of a generated source code file.
`SimCustomHeaderCode` *string* –	**Simulation Target > Custom Code > Header file**	Enter code lines to appear near the top of a generated header file.
`SimCustomInitializer` *string* –	**Simulation Target > Custom Code > Initialize function**	Enter code statements that execute once at the start of simulation.
`SimCustomTerminator` *string* –	**Simulation Target > Custom Code > Terminate function**	Enter code statements that execute at the end of simulation.
`SimUserIncludeDirs` *string* –	**Simulation Target > Custom Code > Include directories**	Enter a space-separated list of directory paths that contain files you include in the compiled target.

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`SimUserSources` *string* –	**Simulation Target > Custom Code > Source files**	Enter a list of source files to compile and link into the target.
`SimUserLibraries` *string* –	**Simulation Target > Custom Code > 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. The parameters are listed in the order that they appear in the Configuration Parameters dialog box.

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`SimUseLocalCustomCode` string – **`off`** , `'on'`	**Simulation Target > 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* –	**Simulation Target > Source file**	Enter code lines to appear near the top of a generated source code file.
`SimCustomHeaderCode` *string* –	**Simulation Target > Header file**	Enter code lines to appear near the top of a generated header file.
`SimCustomInitializer` *string* –	**Simulation Target > Initialize function**	Enter code statements that execute once at the start of simulation.
`SimCustomTerminator` *string* –	**Simulation Target > Terminate function**	Enter code statements that execute at the end of simulation.

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`SimUserIncludeDirs` *string* –	**Simulation Target > Include directories**	Enter a space-separated list of directory paths that contain files you include in the compiled target.
`SimUserSources` *string* –	**Simulation Target > Source files**	Enter a list of source files to compile and link into the target.
`SimUserLibraries` *string* –	**Simulation Target > Libraries**	Enter a space-separated list of static libraries that contain custom object code to link into the target.

Code Generation Parameters for Nonlibrary Models

The following table is a partial list of the parameters and values in the Configuration Parameters dialog box that you can set for embeddable code generation using the command-line API. The parameters are listed in the order that they appear in the Configuration Parameters dialog box.

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`UseSimReservedNames` string – **`off`** , `'on'`	**Real-Time Workshop > Symbols > Use the same reserved names as Simulation Target**	Specify whether to use the same reserved names as those specified for simulation. (Applies only if the model contains Embedded MATLAB Function blocks, Stateflow charts, or Truth Table blocks.)
`ReservedNameArray` *string array* – **`{}`**	**Real-Time Workshop > 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.

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`RTWUseSimCustomCode` string – **`off`** , `'on'`	**Real-Time Workshop > Custom Code > Use the same custom code settings as Simulation Target**	Specify whether to use the same custom code settings as those specified for simulation. (Applies only if the model contains Embedded MATLAB Function blocks, Stateflow charts, or Truth Table blocks.)
`CustomSourceCode` *string* –	**Real-Time Workshop > Custom Code > Source file**	Enter code lines to appear near the top of a generated source code file.
`CustomHeaderCode` *string* –	**Real-Time Workshop > Custom Code > Header file**	Enter code lines to appear near the top of a generated header file.
`CustomInitializer` *string* –	**Real-Time Workshop > Custom Code > Initialize function**	Enter code statements that execute once at the start of simulation.
`CustomTerminator` *string* –	**Real-Time Workshop > Custom Code > Terminate function**	Enter code statements that execute at the end of simulation.
`CustomInclude` *string* –	**Real-Time Workshop > Custom Code > Include directories**	Enter a space-separated list of directory paths that contain files you include in the compiled target.
`CustomSource` *string* –	**Real-Time Workshop > Custom Code > Source files**	Enter a list of source files to compile and link into the target.
`CustomLibrary` *string* –	**Real-Time Workshop > Custom Code > Libraries**	Enter a space-separated list of static libraries that contain custom object code to link into the target.

Code Generation Parameters for Library Models

The following table summarizes the code generation parameters that apply to library models. The parameters are listed in the order that they appear in the Configuration Parameters dialog box.

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`RTWUseSimCustomCode` string – **`off`** , `'on'`	**Real-Time Workshop > Use the same custom code settings as Simulation Target**	Specify whether to use the same custom code settings as those specified for simulation. (Applies only if the model contains Embedded MATLAB Function blocks, Stateflow charts, or Truth Table blocks.)
`RTWUseLocalCustomCode` string – **`off`** , `'on'`	**Real-Time Workshop > 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.
`CustomSourceCode` *`string`* –	**Real-Time Workshop > Source file**	Enter code lines to appear near the top of a generated source code file.
`CustomHeaderCode` *`string`* –	**Real-Time Workshop > Header file**	Enter code lines to appear near the top of a generated header file.
`CustomInitializer` *`string`* –	**Real-Time Workshop > Initialize function**	Enter code statements that execute once at the start of simulation.
`CustomTerminator` *`string`* –	**Real-Time Workshop > Terminate function**	Enter code statements that execute at the end of simulation.
`CustomInclude` *`string`* –	**Real-Time Workshop > Include directories**	Enter a space-separated list of directory paths that contain files you include in the compiled target.

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`CustomSource` *`string`* —	**Real-Time Workshop > Source files**	Enter a list of source files to compile and link into the target.
`CustomLibrary` *`string`* —	**Real-Time Workshop > Libraries**	Enter a space-separated list of static libraries that contain custom object code to link into the target.

For more information about parameters and values you can specify for embeddable code generation, see "Parameter Command-Line Information Summary" in the Real-Time Workshop Reference documentation.

Specifying Relative Paths for Custom Code

In this section...

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.

Searching Relative Paths

Search paths exist relative to these directories:

- The current directory
- The model directory (if different from the current directory)
- The custom list of directories that you specify
- All the directories on the MATLAB search path, excluding the toolbox directories

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 parses these strings and 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 string 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 (-).

Choosing a Compiler

You must use a C or C++ compiler for compiling code that you generate. The Windows version of Stateflow software ships with a C compiler (`lcc.exe`) and a make utility (`lccmake`). Both tools reside in the directory `matlabroot\sys\lcc`. If you do not install any other compiler, `lcc` is the default compiler that builds your targets.

If you use the UNIX version of Stateflow software or do not wish to use the default `lcc` compiler, you must install your own target compiler. You can use any compiler supported by MATLAB software.

Note For an updated list of supported compilers, go to:

`http://www.mathworks.com/support/compilers/current_release/`

To install your own target compiler:

1. At the MATLAB prompt, type:

   ```
   mex -setup
   ```

2. Follow the prompts for entering information about your compiler.

Note If you select an unsupported compiler, this warning message appears when you start a build that requires compilation:

```
The mex compiler specified using 'mex -setup' is not supported
for simulation builds.  Using the lcc compiler instead.
```

Examples of Integrating Custom C Code in Nonlibrary Models

In this section...

Accessing Examples of Custom C Code Integration

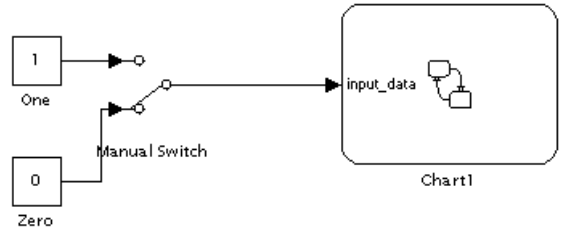
Follow these steps to access the example models:

1. Download the zipped file from this location: www.mathworks.com/company/newsletters/digest/june99/stateflow/integration.zip
2. Extract the contents of the zipped file and store them in a single directory on your local hard drive.
3. In the MATLAB Command Window, change the current working directory to the one where your new files reside.

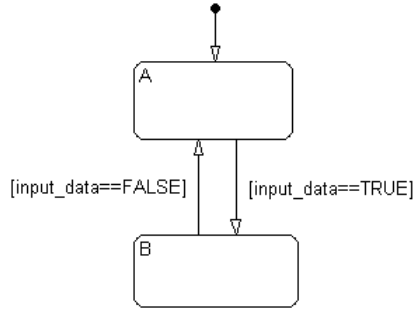
Example of Using Custom C Code to Define Global Constants

This example describes how to use custom C code to define constants that apply to all charts in your model.

1. Open the file named `example1.mdl` in the Simulink model window.



This chart also appears in the Stateflow Editor.



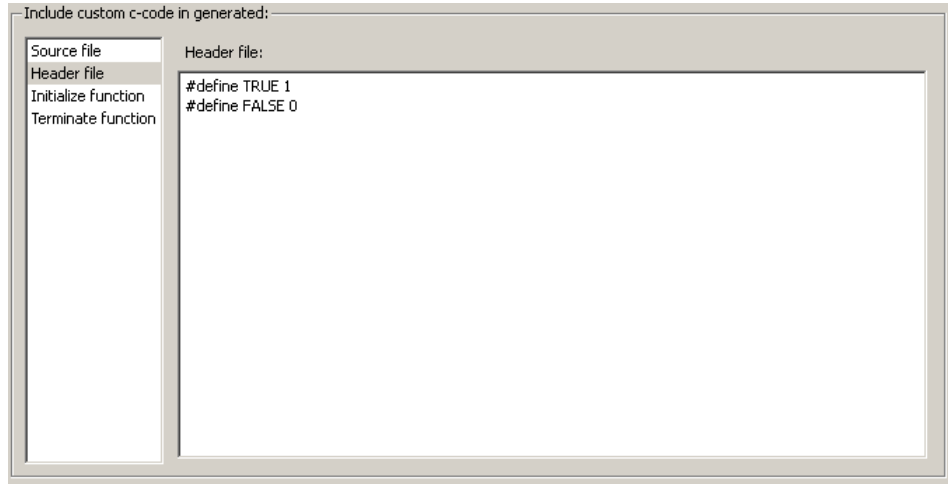
The chart contains two states `A` and `B` and one Simulink input named `input_data`, which you can set to 0 or 1 by flipping the Manual Switch in the model during simulation.

2 In the Stateflow Editor, select **Tools > Open Simulation Target**.

3 In the Configuration Parameters dialog box, select the **Simulation Target > Custom Code** pane.

4 Select the **Header file** subpane.

In this subpane, you can enter `#define` and `#include` statements.



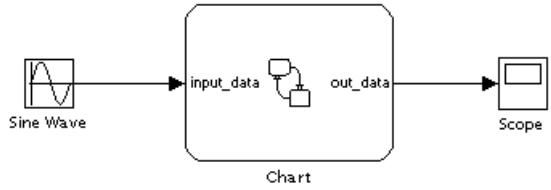
In this example, you define two constants named `TRUE` and `FALSE` to move between states in your chart, instead of using the values 1 and 0. These custom definitions improve the readability of your chart actions. Note that `TRUE` and `FALSE` are not Stateflow data objects.

Because the two custom definitions appear at the top of your generated machine header file (`example1_sfun.h`), you can use `TRUE` and `FALSE` in all charts that belong to this model.

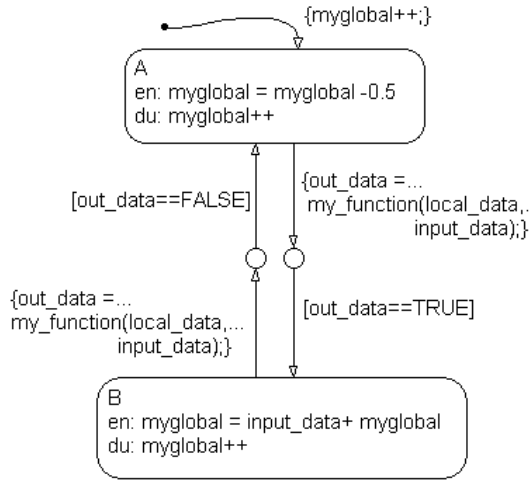
Example of Using Custom C Code to Define Global Constants, Variables, and Functions

This example describes how to use custom C code to define constants, variables, and functions that apply to all charts in your model.

1 Open the file named `example2.mdl` in the Simulink model window.



This chart also appears in the Stateflow Editor.



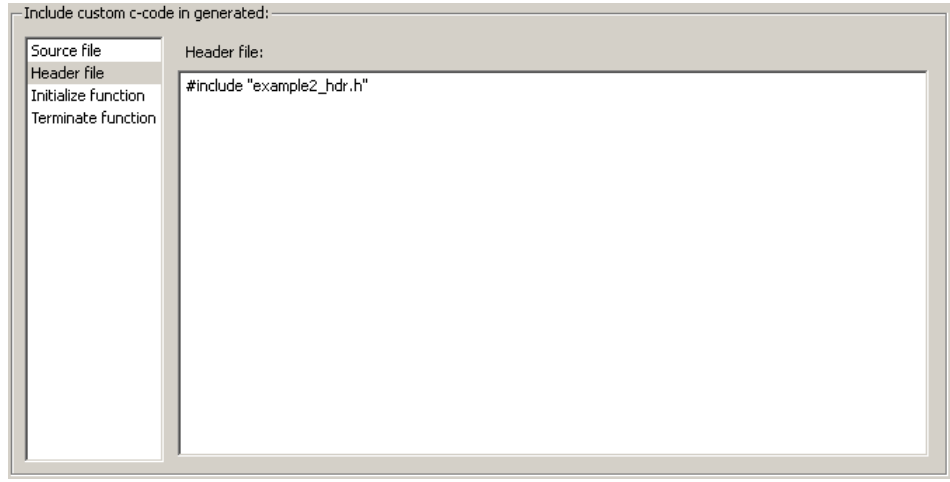
The chart contains two states `A` and `B` and three data objects: `input_data`, `local_data`, and `out_data`. The chart calls a custom function named `my_function` and accesses a custom variable named `myglobal`.

2 In the Stateflow Editor, select **Tools > Open Simulation Target**.

3 In the Configuration Parameters dialog box, select the **Simulation Target > Custom Code** pane.

4 Select the **Header file** subpane.

In this subpane, you can enter `#define` and `#include` statements.



Note When you include a custom header file, you must enclose the file name in double quotes.

The custom header file `example2_hdr.h` contains definitions of three constants:

```
#define TRUE 1
#define FALSE 0
#define MAYBE 2
```

This header file also contains declarations for the function `my_function` and the variable `myglobal`:

```
extern int myglobal;
extern int my_function(int var1, double var2);
```

5 Select the **Include directories** subpane.

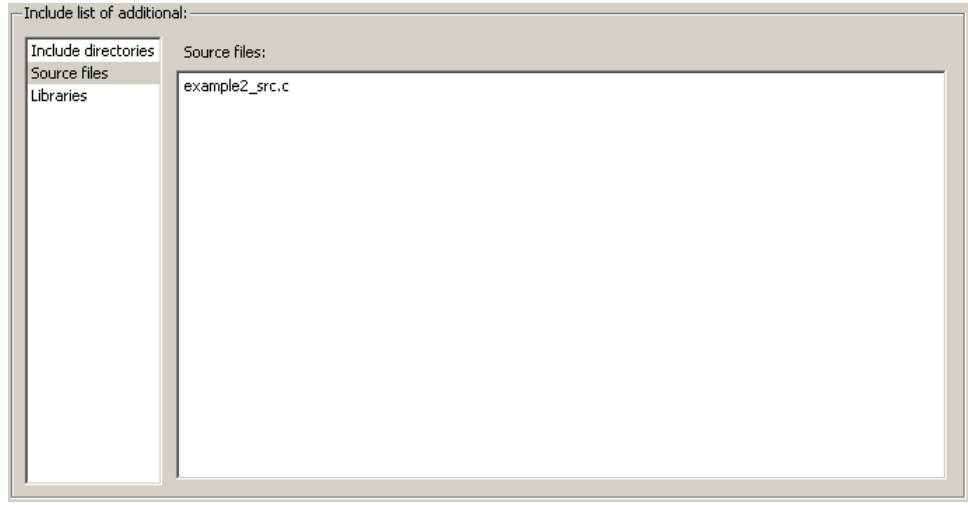
The single period (`.`) indicates that all your custom code files reside in the same directory as `example2.mdl`.

Tip To direct your makefile to look for header or source files in a subdirectory relative to the model directory, use this relative path name:

```
.\subdirectory_name
```

---

6 Select the **Source files** subpane.



The custom source file `example2_src.c` compiles along with the Stateflow generated code into a single S-function MEX file. See "S-Function MEX-Files" on page 22-73 for details.

Tip To include a source file that resides in a subdirectory relative to the model directory, use this relative path name:

```
.\subdirectory_name\source_file.c
```

In this example, you define three constants, a variable, and a function via custom code options. Because the custom definitions appear at the top of your generated machine header file (`example2_sfun.h`), you can access them in all charts that belong to this model.

How to Build a Stateflow Custom Target

In this section...

When to Build a Custom Target

If you want to generate standalone code for applications other than production or rapid prototyping, you can use Stateflow Coder code generation software to build a custom target. This code does not include optimizations provided by Real-Time Workshop code generation.

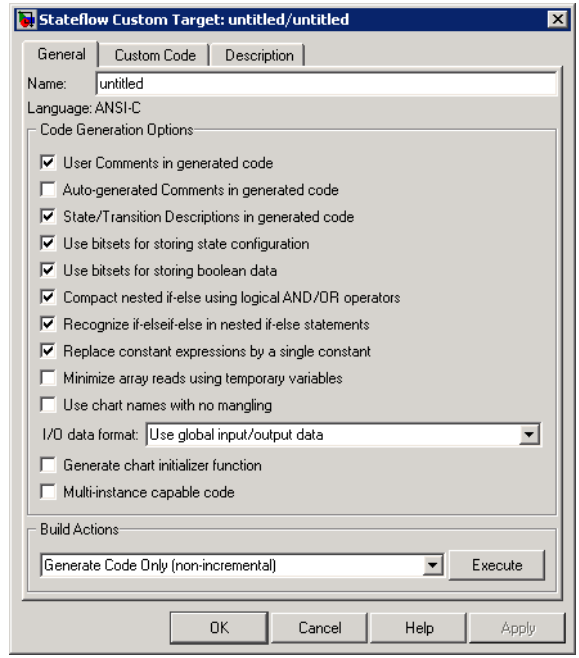
For information on setting custom target options programmatically, see "Target Properties" and "Target Methods" in the *Stateflow and Stateflow Coder API*.

Adding a Stateflow Custom Target to Your Model

To add a custom target to your model, perform these steps:

1 In the Stateflow Editor, select **Add** > **Target**.

The Stateflow Custom Target dialog box appears.



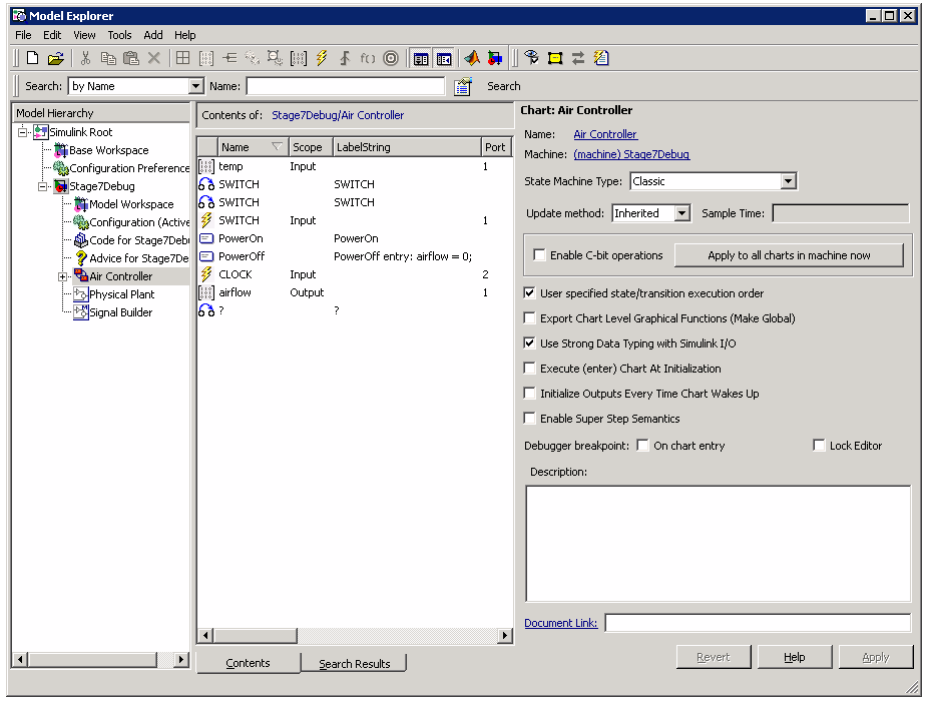
2 In the **Name** field, enter any name except the reserved names `sfun` and `rtw`, and click **OK**.

Configuring a Custom Target

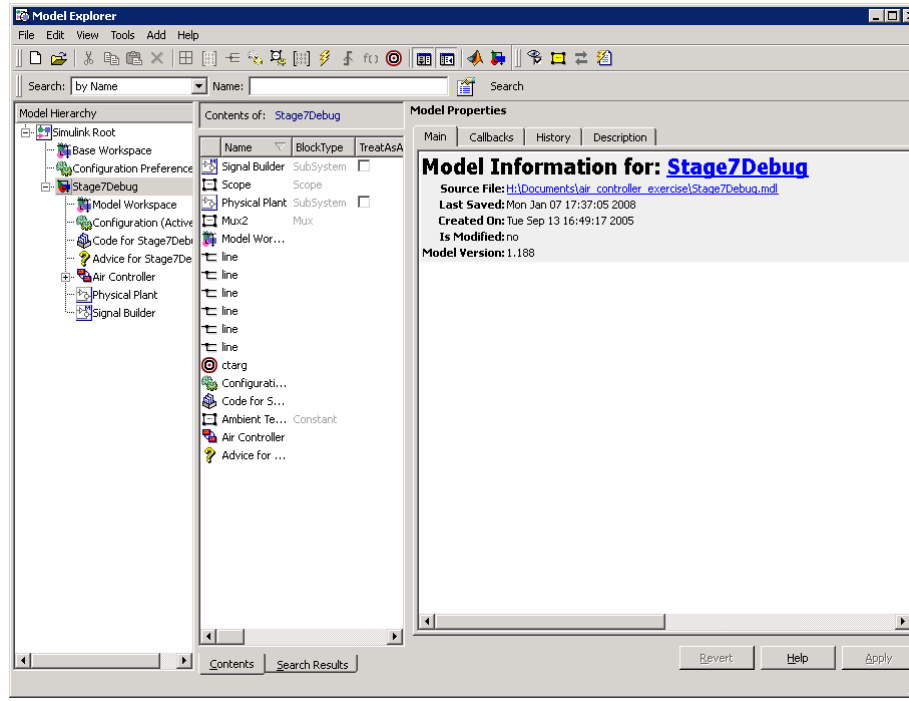
To configure a custom target, perform these steps:

1 From the Stateflow Editor toolbar, click the **Explore** icon:

The Model Explorer appears with the Stateflow chart highlighted in the **Model Hierarchy** pane.



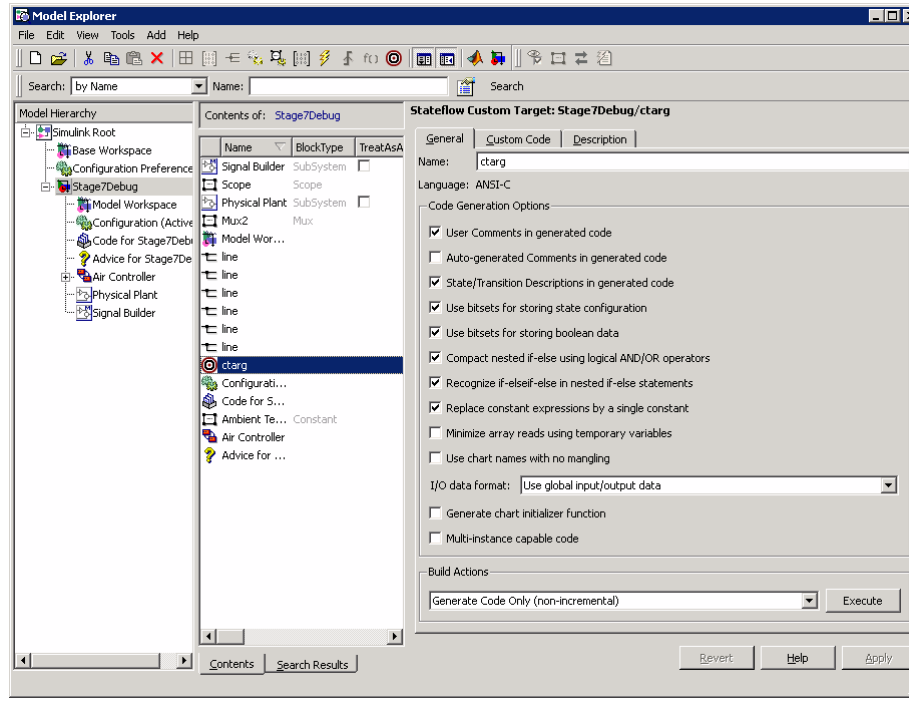
2 In the **Model Hierarchy** pane, select the main model with the custom target.



The custom target (in this example, `ctarg`) appears as an object of the main model.

3 In the **Contents** pane, click the row for the custom target.

The Stateflow Custom Target dialog box appears in the dynamic dialog box on the right.



4 In the **General** pane of the Stateflow Custom Target dialog box, specify options for your custom target:

- **User Comments in generated code** — Includes user-defined comments in the generated code.
- **Auto-generated Comments in generated code** — Includes auto-generated comments in the generated code.
- **State/Transition Descriptions in generated code** — Includes descriptions of states and transitions in the generated code.
- **Use bitsets for storing state configuration** — Reduces the amount of memory that stores the variables. However, it can increase the amount of memory that stores target code if the target processor does not include instructions for manipulating bitsets.
- **Use bitsets for storing boolean data** — Reduces the amount of memory that stores Boolean variables. However, it can increase the

amount of memory that stores target code if the target processor does not include instructions for manipulating bitsets.

Note You cannot use bitsets when you generate code for these cases:

- an external mode simulation
- a target that specifies an explicit structure alignment

- **Compact nested if-else using logical AND/OR operators** — Improves readability of generated code by compacting nested if-else statements using logical AND (`&&`) and OR (`||`) operators.

  For example, the generated code

  ```
  if(c1) {
      if(c1) {
          a1();
      }
  }
  ```

  becomes

  ```
  if(c1 && c2) {
      a1();
  }
  ```

  and the generated code

  ```
  if(c1) {
      /* fall through to do a1() */
  }else if(c2) {
      /* fall through to do a1() */
  }else{
      /* skip doing a1() */
      goto label1;
  }
  a1();
  label1:
      a2();
  ```

becomes

```
if(c1 || c2) {
    a1();
}
a2();
```

- **Recognize if-elseif-else in nested if-else statements** — Improves readability of generated code by creating an `if-elseif-else` construct in place of deeply nested `if-else` statements.

  For example, the generated code

```
if(c1) {
    a1();
}else{
    if(c2) {
        a2();
    }else{
        if(c3) {
            a3();
        }
    }
}
```

  becomes

```
if(c1) {
    a1();
}else if(c2) {
    a2();
}else if(c3) {
    a3();
}
```

- **Replace constant expressions by a single constant** — Improves readability by preevaluating constant expressions and replacing them with a single constant. This optimization also eliminates dead code.

  For example, the generated code

```
if(2+3<2) {
    a1();
}else {
    a2(4+5);
}
```

becomes

```
if(0) {
    a1();
}else {
    a2(9);
}
```

in the first phase of this optimization. The second phase eliminates the `if` statement, resulting in simply

```
a2(9);
```

- **Minimize array reads using temporary variables** — Minimizes expensive array read operations by using temporary variables when possible.

  For example, the generated code

```
a[i] = foo();
if(a[i]<10 && a[i]>1) {
    y = a[i]+5;
}else{
    z = a[i];
}
```

  becomes

```
a[i] = foo();
temp = a[i];
if(temp<10 && temp>1) {
    y = temp+5;
}else{
    z = temp;
}
```

- **Use chart names with no mangling** — (See the note below before using.) Preserves the names of chart entry functions so that you can invoke them using handwritten C code.

Note When you select this check box, the generated code does not mangle the chart names to make them unique. Because this option does not check for name conflicts in generated code, use this option only when you have unique chart names in your model. Conflicts in generated names can cause variable aliasing and compilation errors.

- **I/O data format** — Choose one of these options:

  Select **Use global input/output data** to generate chart input and output data as global variables.

  Select **Pack input/output data into structures** to generate structures for chart input data and chart output data.

- **Generate chart initializer function** — Generates a function initializer of data.
- **Multi-instance capable code** — Generates multiple instantiable chart objects instead of a static definition.

5 Select one of these build options:

- **Generate Code Only (non-incremental)** to regenerate code for all charts in the model.
- **Rebuild All (including libraries)** to rebuild the target, including chart libraries, from scratch. Use this option if you have changed your compiler or updated your object files since the last build.
- **Make without generating code** to invoke the make process without generating code. Use this option when you have custom source files that you must recompile in an incremental build mechanism that does not detect changes in custom code files.

6 Specify any custom code options in the **Custom Code** pane.

Building a Custom Target

To build a custom target, click **Execute** in the **General** pane of the Stateflow Custom Target dialog box.

Note You cannot build a custom target for a model that contains enumerated data. For details, see "Rules for Using Enumerated Data in a Stateflow Chart" on page 12-18.

See "Generated Code Files for Targets You Build" on page 22-73 for details about the code you generate for this target and its directory structure.

What Happens During the Target Building Process?

The target building process takes place as follows:

1. The charts in your model parse to ensure that their logic is valid.
2. If any errors are found, diagnostic error messages appear in the Build window, and the building process stops. See “Parsing Stateflow Charts” on page 22-61 for more details.
3. If your charts parse without error, code generation software generates C code from your charts.

   You can specify code generation options when you configure your targets.
4. Code generation software produces a makefile to build the generated source code into an executable program.

   The makefile can optionally build your custom code into the target.
5. The specified C compiler for the MATLAB environment and a make utility build the code into an application for your target.

Parsing Stateflow Charts

In this section...

How the Stateflow Parser Works

When you begin a build for a target, the parser evaluates the graphical and nongraphical objects in each Stateflow machine against the supported Stateflow chart notation and the action language syntax.

Note The parser does not check for unresolved symbol errors unless you start simulation or update the model diagram. See "Resolving Symbols" on page 22-67 for details.

Calling the Stateflow Parser

Apart from building a target, you can call the Stateflow parser to check the syntax of your Stateflow charts in one of these ways:

- Parse an individual chart in the Stateflow Editor by selecting **Tools** > **Parse Diagram**.
- Parse a Stateflow machine (that is, all the charts in a model), by selecting **Tools** > **Parse** in the Stateflow Editor.
- When you simulate a model, build a target, or generate code for a target, you automatically parse the Stateflow machine.

In all cases, the Stateflow Builder window appears when parsing is complete. If parsing is unsuccessful (that is, an error appears), the Stateflow Editor automatically appears with the highlighted object causing the first parse error. In the Stateflow Builder window, each error appears with a leading red button icon. You can double-click any error in this window to bring its source

Stateflow chart to the front with the source object highlighted. See "Parsing Chart Example" on page 22-62 for example displays of parsing results in the Stateflow Builder window.

Note Parsing informational messages also appear in the MATLAB Command Window.

Parser Error Checking

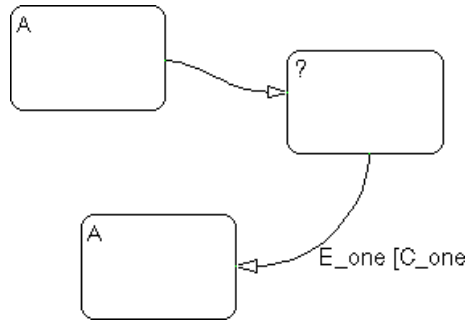
Using the Debugger, you can detect the following errors during simulation:

- State Inconsistency — Most commonly caused by the omission of a default transition to a substate in superstates with exclusive (OR) decomposition. See "Debugging State Inconsistencies in a Chart" on page 23-20.
- Transition Conflict — Occurs when there are two equally valid transition paths from the same source. See "Debugging Conflicting Transitions in a Chart" on page 23-22.
- Data Range Violation — Occurs when minimum and maximum values specified for a data in its properties dialog box exceed their limits or when fixed-point data overflows its base word size. See "Debugging Data Range Violations in a Chart" on page 23-24.
- Cyclical Behavior — Occurs when a step or sequence of steps repeats itself indefinitely. See "Debugging Cyclic Behavior in a Chart" on page 23-26.

You can modify the notation to resolve run-time errors. See Chapter 23, "Debugging and Testing Stateflow Charts" for more information on debugging run-time errors.

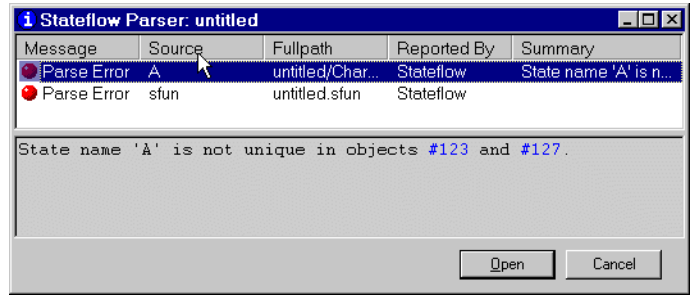
Parsing Chart Example

For this chart, the steps that follow describe the parsing process and its reported results.



1 In the Stateflow Editor, select **Tools** > **Parse Diagram** to parse the chart.

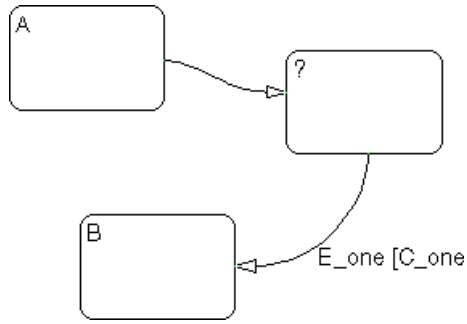
This action selects State A in the upper left corner, and this message appears in the Parser window and the MATLAB Command Window.



2 Fix the parse error.

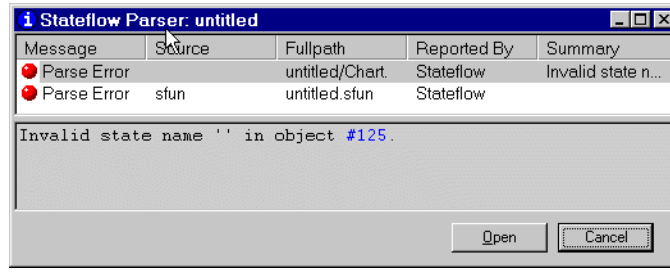
In this example, there are two states with the name A. Edit the chart and label the duplicate state with the text B.

The chart should look similar to this figure.



3 In the Stateflow Editor, select **Tools > Parse Diagram** to reparse the chart.

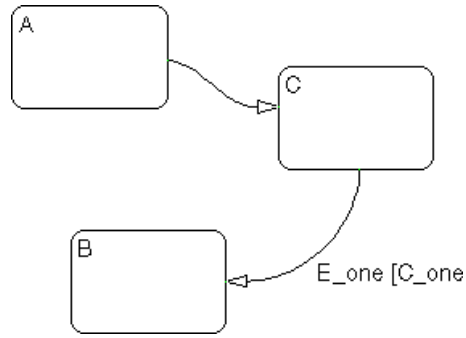
This message appears in the Parser window and the MATLAB Command Window.



4 Fix the parse error.

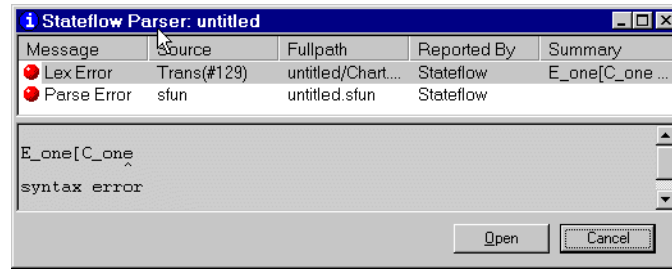
In this example, you must label the state with the question mark with at least a state name. Edit the chart and label the state with the text `C`.

The chart should look similar to this figure.



5 In the Stateflow Editor, select **Tools** > **Parse Diagram** to reparse the chart.

This message appears in the Parser window and the MATLAB Command Window.

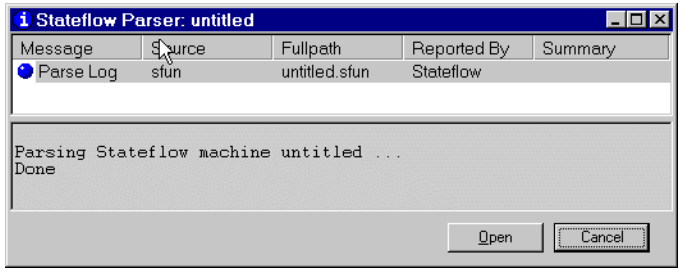


6 Fix the parse error.

In this example, the transition label contains a syntax error, where the closing bracket of the condition is missing. Edit the chart and add the closing bracket so that the label is `E_one [C_one]`.

7 In the Stateflow Editor, select **Tools** > **Parse Diagram** to reparse the chart.

This message appears in the Parser window and the MATLAB Command Window.



The chart now has no parse errors.

Resolving Event, Data, and Function Symbols in Stateflow Action Language

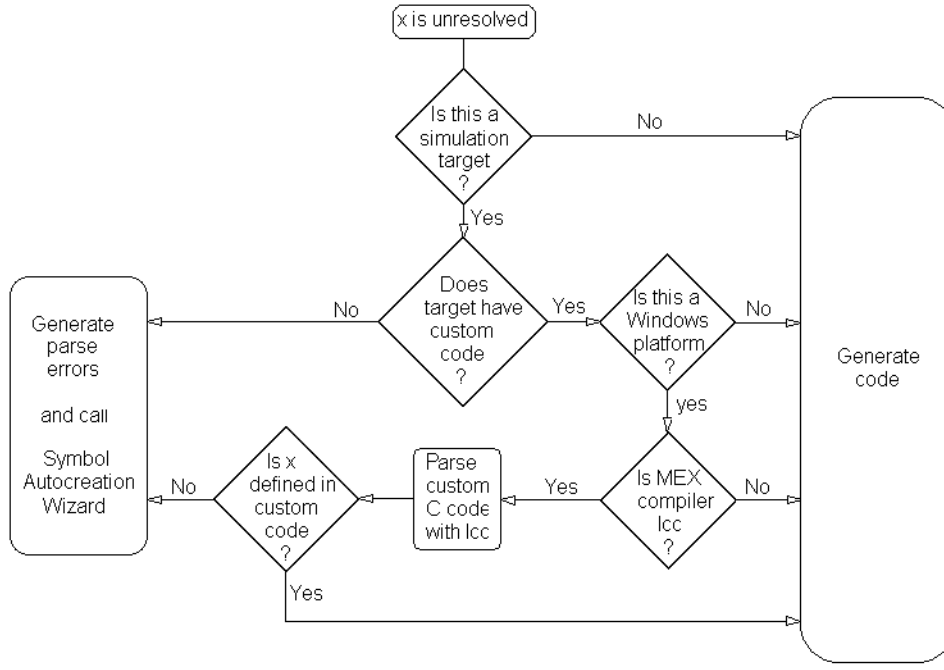
In this section...

Resolving Symbols

To check for unresolved symbol errors, you can use of these methods:

- Start simulation (for example, by selecting **Simulation > Start** in the model window)
- Update the model diagram (for example, by selecting **Edit > Update Diagram** in the model window)

Each method triggers parsing of the Stateflow machine (see "Parsing Stateflow Charts" on page 22-61). During parsing, if your chart does not resolve some of its symbols, the following process determines whether to report errors for the unresolved symbols or to continue generating code.

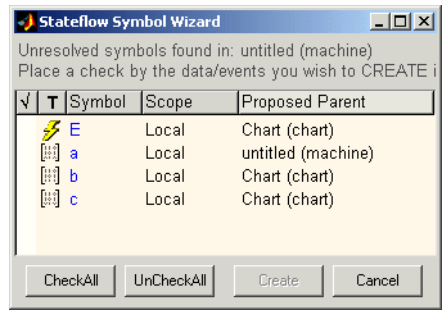


Note When you parse a chart without simulation or diagram updates, the Stateflow parser does not have access to all the information needed to check for unresolved symbols, such as exported graphical functions from other charts and enumerated data types. However, if you start simulation or update the model diagram, you invoke the model compilation process, which has full access to the information needed.

For information about Simulink symbol resolution, see “Resolving Symbols” and “Hierarchical Symbol Resolution” in the Simulink documentation.

Symbol Autocreation Wizard

You can use the Symbol Autocreation Wizard to add missing data and events to your Stateflow charts. When you parse or simulate a chart, the Wizard detects references to undefined data and events and presents a list of the recommended data or events that you must define.



To accept, reject, or change a recommended item, do one of these steps:

- To accept an item, click on the space in front of the item under the check mark column.

  To accept all items, click **CheckAll**.

- To reject an item, leave it unchecked.

- To change an item, click on the icon under the **T** (type) column, or click on the string under the **Scope** or **Proposed Parent** column for that item.

  Each time you click on an icon or a string, the Wizard replaces the entry with a different one. Keep clicking until the desired icon or string appears.

Column in the Wizard	Choices When You Toggle Between Entries
T	Data, Event
Scope	Local, Input, Output
Proposed Parent	Chart, Machine

After you finish editing the symbol definitions, click **Create** to add the symbols to the Stateflow hierarchy.

Error Messages When Parsing Charts and Generating Code

In this section...

How Error Messages Appear

Error messages appear in a dialog box and in the MATLAB Command Window. Double-clicking a message in the error dialog box zooms the source Stateflow chart to the object that caused the error.

Parser Error Messages

The Stateflow parser flags syntax errors in a chart. For example, using a backward slash (\) instead of a forward slash (/) to separate the transition action from the condition action generates a general parse error message.

Typical parse error messages include:

- `"Invalid state name xxx"` or `"Invalid event name yyy"` or `"Invalid data name zzz"`

  A state, data, or event name contains a nonalphanumeric character other than underscore.

- `"State name xxx is not unique in objects #yyy and #zzz"`

  Two or more states at the same hierarchy level have the same name.

- `"Invalid transition out of AND state xxx (#yy)"`

  A transition originates from an AND (parallel) state.

- `"Invalid intersection between states xxx and yyy"`

Neighboring state borders intersect. If the intersection is not apparent, consider the state to be a cornered rectangle instead of a rounded rectangle.

- `"Junction #x is sourcing more than one unconditional transition"`

  More than one unconditional transition originates from a connective junction.

- `"Multiple history junctions in the same state #xxx"`

  A state contains more than one history junction.

Code Generation Error Messages

Typical code generation error messages include:

- `"Failed to create file:  modelName_sfun.c"`

  Code generation software does not have permission to generate files in the current directory.

- `"Another unconditional transition of higher priority shadows transition #xx"`

  More than one unconditional inner, default, or outer transition originates from the same source.

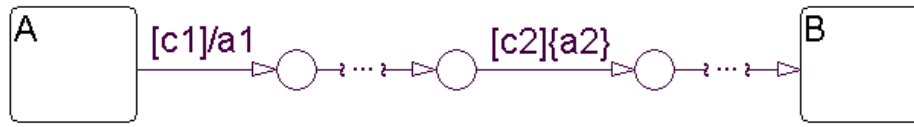
- `"Default transition cannot end on a state that is not a substate of the originating state."`

  A transition path starting from a default transition segment in one state completes at a destination state that is not a substate of the original state.

- `"Input data xxx on left hand side of an expression in yyy"`

  A Stateflow expression assigns a value to an **Input from Simulink** data object. By definition, a Stateflow expression cannot change the value of a Simulink input.

- `"Transition <number> has a condition action which is preceded by a transition <number> containing a transition action.  This is not allowed as it results in out-of-order execution, i.e., the condition action of <number> gets executed before the transition action of <number>."`



The preceding Stateflow chart flags this error. Assuming that there are no other actions than those you indicate for the labeled transition segments between state `A` and state `B`, the following pseudocode expresses the sequence of execution that takes place when state `A` is active:

```
If (c1) {
 if(c2) {
  a2;
  exit A;
  a1;
  enter B;
 }
}
```

Because condition actions evaluate when their guarding condition is true and transition actions evaluate when the transition is actually taken, condition action `a2` executes prior to transition action `a1`. This action violates the apparent graphical sequence of executing `a1` and then `a2`. In this case, the preceding chart flags the error during build time. To fix this problem, you can change `a1` and `a2` to be both condition or transition actions.

Compilation Error Messages

If compilation errors indicate undeclared identifiers, verify that variable expressions in state, condition, and transition actions are defined.

Consider, for example, an action language expression such as `a=b+c`. In addition to entering this expression in the Stateflow chart, you must create data objects for `a`, `b`, and `c` using the Model Explorer. If you do not define the data objects, the parser assumes that these unknown variables appear in the **Custom code** portion of the target, at the beginning of the generated code. Because of this assumption, error messages appear at compile time and not at code generation time.

Generated Code Files for Targets You Build

In this section...

S-Function MEX-Files

If you have a Simulink model named `mainModel.mdl`, which contains two Stateflow blocks named `chart1` and `chart2`, you have a machine named `mainModel` that parents two charts named `chart1` and `chart2`.

When you simulate the Stateflow chart for `mainModel.mdl`, you generate code for `mainModel.mdl` that compiles into an S-function MEX-file. MEX-file extensions are platform-specific, as described in the MATLAB software documentation. For example, on 32-bit Windows PC platforms, you generate a MEX-file for `mainModel` named `mainModel_sfun.mexw32`. On Linux® x86-64 platforms, you generate `mainModel_sfun.mexa64`.

S-function MEX files appear in the current MATLAB directory. You can change this location at the MATLAB command prompt with a `cd` command.

Directory Structure of Generated Files

Most of the code files that you generate reside in a subdirectory of the current MATLAB directory. This table summarizes the directory structure for different targets.

Target Type	Model Type	Directory Under `<cwd>/slprj/_sfprj/<mainModel>`
Simulation	Main (nonlibrary)	`/_self/sfun/src`

Target Type	Model Type	Directory Under <cwd>/slprj/_sfprj/<mainModel>
Simulation	Library	/<libModel>/sfun/src
Embeddable	Main (nonlibrary)	/_self/rtw/<sys_targ>/src
Embeddable	Library	/<libModel>/rtw/<sys_targ>/src
Custom	Main (nonlibrary)	/_self/<custom>/src
Custom	Library	/<libModel>/<custom>/src

These definitions apply to the table:

- <cwd> is the current MATLAB directory.
- <mainModel> is the name of the main model.
- <libModel> is the name of the library model.
- <sys_targ> is the type of embeddable target (for example, grt or ert).
- <custom> is the name of the custom target.

Note For embeddable targets, the integrated C code for the entire model resides in the subdirectory <mainModel>_<sys_targ>_rtw of the current MATLAB directory. The executable file generated for the entire model resides in the current MATLAB directory.

Code Files for a Simulation Target

For a simulation target, you generate these files:

- <model>_sfun.h is the machine header file. It contains:
  - All the defined global variables needed for the generated code
  - Type definition of the Stateflow machine-specific data structure that holds machine-parented local data

  - External declarations of any Stateflow machine-specific global variables and functions
  - Custom code strings that you specify
- `<model>_sfun.c` is the machine source file. It includes the machine header file and all the chart header files (described below) and contains:
  - All the machine-parented event broadcast functions
  - Simulink interface code
- `<model>_sfun_registry.c` is a machine registry file that contains Simulink interface code.
- `cn_<model>.h` is the chart header file for the chart `chartn`, where `n` = 1, 2, 3, and so on, depending on how many charts your model has (see the following note). This header file contains type definitions of the chart-specific data structures that hold chart-parented local data and states.
- `cn_<model>.c` is the chart source file for `chartn`, where `n` = 1, 2, 3, and so on, depending on how many charts your model has (see the following note). This source file includes the machine header file and the corresponding chart header file and also contains:
  - Chart-parented data initialization code
  - Chart execution code (state entry, during, and exit actions, and so on)
  - Chart-specific Simulink interface code

Note Every chart is assigned a unique number at creation time. This number appears as a suffix for the chart source and chart header file names for every chart (where `n` = 1, 2, 3, and so on, depending on how many charts your model has).

For library models, a static library file named `<libModel>_sfun` resides in the same directory as the source code. The file extension depends on the platform. (On a Windows operating system, it is `<libModel>_sfun.lib`, but on a UNIX operating system, it is `<libModel>_sfun.a`.)

Code Files for an Embeddable Target

For an embeddable target, you generate integrated C code for the entire model:

- `<model>.h`
- `<model>.c`

You also generate intermediate code files during the target building process:

- `<model>_rtw.tlh`
- `<model>_rtw.tlc`
- `cn_<model>.tlh`, where `n` = 1, 2, 3, and so on, depending on how many charts your model has
- `cn_<model>.tlc`, where `n` = 1, 2, 3, and so on, depending on how many charts your model has

Other auxiliary files can appear depending on the type of embeddable target you choose for Real-Time Workshop code generation.

Code Files for a Custom Target

For a custom target, you generate these files:

- `<model>_<custom>.h` where `<custom>` is the name of the custom target.
- `<model>_<custom>.c` where `<custom>` is the name of the custom target.
- `cn_<model>.h` is the chart header file for the chart `chartn`, where `n` = 1, 2, 3, and so on, depending on how many charts your model has. This file contains type definitions of the chart-specific data structures that hold chart-parented local data and states.
- `cn_<model>.c` is the chart source file for `chartn`, where `n` = 1, 2, 3, and so on, depending on how many charts your model has. This chart source file includes the machine header file and the corresponding chart header file.

Makefiles

You generate makefiles for your model that are platform and compiler-specific. On UNIX platforms, you generate a `gmake`-compatible makefile named `<mainModel>_sfun.mku` that compiles all your generated code into an

executable. On PC platforms, you generate an ANSI-C compiler-specific makefile based on your C-MEX setup:

Compiler	Makefile	Symbol Definition File
Microsoft® Visual C++®	`<mainModel>_sfun.mak`	`<mainModel>_sfun.def` (required to build S-function MEX-files)
Open Watcom	`<mainModel>_sfun.wmk`	None
`lcc-win32` (default ANSI-C compiler)	`<mainModel>_sfun.lmk`	None

Note For an updated list of supported PC compilers, go to:

`http://www.mathworks.com/support/compilers/current_release/`

Traceability of Stateflow Objects in Real-Time Workshop Generated Code

In this section...

What Is Traceability?

Traceability is the ability to navigate between a line of generated code and its corresponding object. For example, you can click a hyperlink in a traceability comment to go from that line of code to the object in the model. You can also right-click an object in your model to find the line in the code that corresponds to the object. This two-way navigation is known as *bidirectional* traceability.

See "Tracing Generated Code Back to Your Simulink Model" in the Real-Time Workshop User's Guide for information about how traceability works for Simulink blocks.

Traceability Requirements

To enable traceability comments, you must have a license for Real-Time Workshop Embedded Coder software. These comments appear only in code that you generate for an embedded real-time (`ert`) based target.

Traceable Stateflow Objects

Bidirectional traceability is supported for these Stateflow objects:

- States

- Transitions
- Embedded MATLAB functions

Note Traceability is not supported for M-files that you call from an Embedded MATLAB function.

- Truth Table blocks and truth table functions
- Graphical functions
- Simulink functions

Traceability in one direction is supported for these Stateflow objects:

- Events (code-to-model)

  Code-to-model traceability works for explicit events, but not implicit events. Clicking a hyperlink for an explicit event in the generated code highlights that item in the **Contents** pane of the Model Explorer.

- Junctions (model-to-code)

  Model-to-code traceability works for junctions with at least one outgoing transition. Right-clicking such a junction in the Stateflow Editor highlights the line of code that corresponds to the first outgoing transition for that junction.

Note Embedded MATLAB Function blocks that you insert directly in a Simulink model are also traceable. For more information, see "Using Traceability in Embedded MATLAB Function Blocks" in the Simulink software documentation.

When to Use Traceability

Comments for Large-Scale Models

Use traceability when you want to generate commented code for a large-scale model. You can identify chart objects in the code and avoid manually entering comments or descriptions.

Validation of Generated Code

Use traceability when you want to validate generated code. You can identify which chart object corresponds to a particular line of code and keep track of code from different objects that you have or have not reviewed.

Basic Workflow for Using Traceability

The basic workflow for using traceability is:

1. Open your model, if necessary.
2. Define your system target file to be an embedded real-time (`ert`) target.
3. Enable and configure the traceability options.
4. Generate the source code and header files for your model.
5. Do one or both of these steps:
   - Trace a line of generated code to the model.
   - Trace an object in the model to a line of code.

Examples of Using Traceability

Bidirectional Traceability for States and Transitions

You can see how bidirectional traceability works for states and transitions by following these steps:

1. Type `old_sf_car` at the MATLAB prompt.

2 In the Simulink model window or the Stateflow Editor, select **Simulation > Configuration Parameters**.

3 In the **Real-Time Workshop** pane, go to the **Target selection** section and enter `ert.tlc` for the system target file. Click **Apply** in the lower right corner of the window.

Note Traceability comments appear in generated code only for embedded real-time targets.

---

4 In the **Real-Time Workshop > Report** pane, select **Create code generation report**.

This action automatically selects **Launch report automatically** and **Code-to-model**.

5 Select **Model-to-code** in the **Navigation** section. Then click **Apply**.

This action automatically selects all check boxes in the **Traceability Report Contents** section.

Note For large models that contain over 1000 blocks, clear the **Model-to-code** check box to speed up code generation.

---

6 Go to the **Real-Time Workshop > Interface** pane. In the **Software environment** section, select **continuous time**. Then click **Apply**.

Note Because this demo model contains a block with a continuous sample time, you must perform this step before generating code.

---

7 In the **Real-Time Workshop** pane, click **Build** in the lower right corner.

This action generates source code and header files for the `old_sf_car` model that contains the `shift_logic` chart. After the code generation process is complete, the code generation report appears automatically.

8 Click the `old_sf_car.c` hyperlink in the report.

9 Scroll down through the code to see the traceability comments.

```
170  /* Functions for block: '<Root>/shift_logic' */
171  static void sf_car_gear_state(void)
172  {
173    /* During 'gear_state': '<S5>:2' */
174    if (!(sf_car_DWork.is_active_gear_state == 0)) {
175      switch (sf_car_DWork.is_gear_state) {
176       case sf_car_IN_first:
177        /* During 'first': '<S5>:6' */
178        if (_sfEvent_sf_car_ == sf_car_event_UP) {
179          /* Transition: '<S5>:12' */
```

Traceability comment for a state

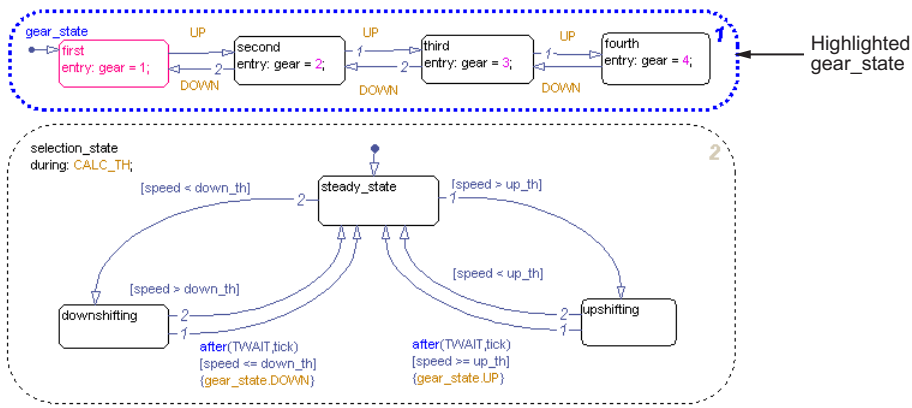
Traceability comment for a transition

Note The line numbers shown above can differ from the numbers that appear in your code generation report.

10 Click the `<S5>:2` hyperlink in this traceability comment:

```
/* During 'gear_state': '<S5>:2' */
```

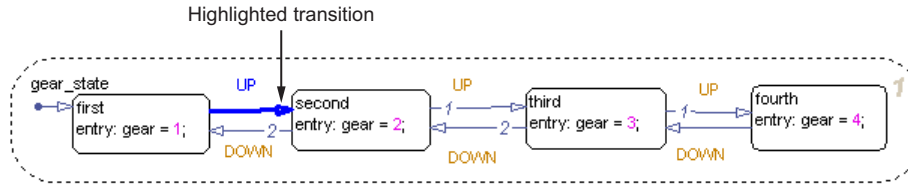
In the Stateflow Editor, the object `gear_state` appears highlighted.



11 Click the `<S5>:12` hyperlink in this traceability comment:

```
/* Transition: '<S5>:12' */
```

In the Stateflow Editor, the corresponding transition appears highlighted.



Note To remove highlighting from an object in the Stateflow Editor, select **View** > **Remove Highlighting**.

12 You can also trace an object in the model to a line of generated code. In the Stateflow Editor, right-click the object `gear_state` and select **Real-Time Workshop** > **Navigate to Code** from the context menu.

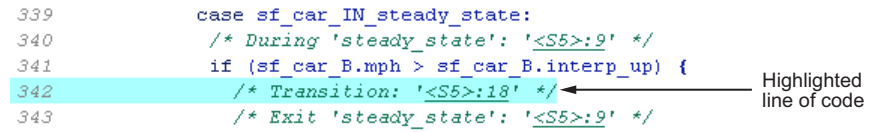
The code for that state appears highlighted in `old_sf_car.c`.

```
170   /* Functions for block: '<Root>/shift_logic' */
171   static void sf_car_gear_state(void)
172   {
173     /* During 'gear_state': '<S5>:2' */
174     if (!(sf_car_DWork.is_active_gear_state == 0)) {
175       switch (sf_car_DWork.is_gear_state) {
176        case sf_car_IN_first:
177         /* During 'first': '<S5>:6' */
178         if (_sfEvent_sf_car_ == sf_car_event_UP) {
179           /* Transition: '<S5>:12' */
```

Highlighted line of code

13 In the Stateflow Editor, right-click the transition with the condition `[speed > up_th]` and select **Real-Time Workshop** > **Navigate to Code** from the context menu.

The code for that transition appears highlighted in `old_sf_car.c`.



Note For a list of all Stateflow objects in your model that are traceable, click the `Traceability Report` hyperlink in the code generation report.

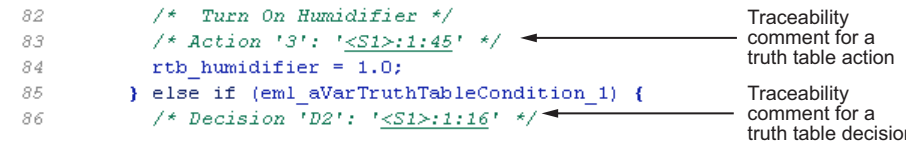
See "Creating and Using a Code Generation Report" in the Real-Time Workshop Embedded Coder User's Guide for more information about the code generation report.

Bidirectional Traceability for Truth Table Blocks

You can see how bidirectional traceability works for a Truth Table block by following these steps:

1. Type `sf_climate_control` at the MATLAB prompt.
2. Complete steps 2–5 in "Bidirectional Traceability for States and Transitions" on page 22-80.
3. In the **Real-Time Workshop** pane of the Configuration Parameters dialog box, click **Build** in the lower right corner.

   The code generation report appears automatically.
4. Click the `sf_climate_control.c` hyperlink in the report.
5. Scroll down through the code to see the traceability comments.

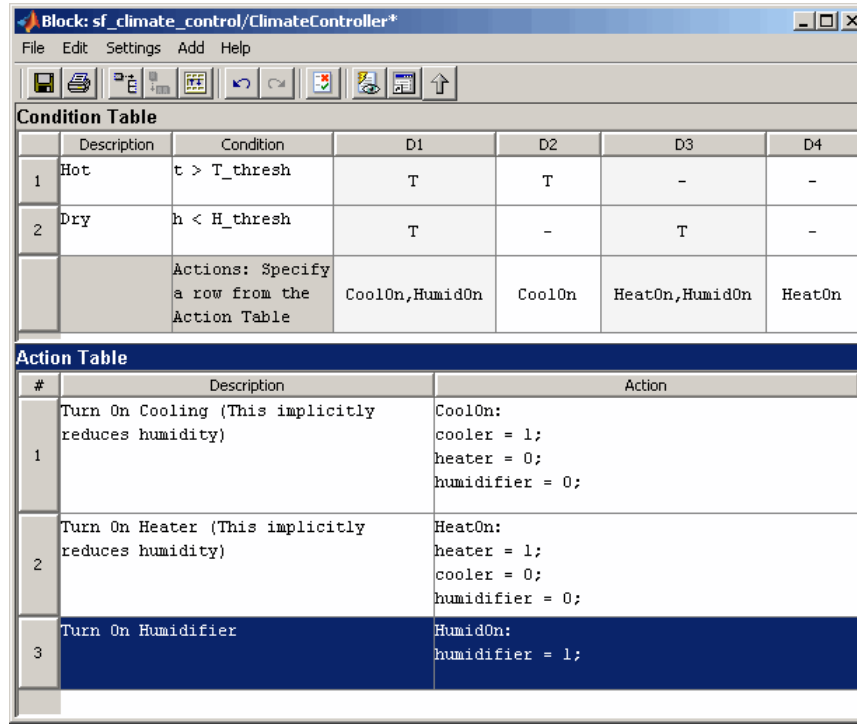


Note The line numbers shown above can differ from the numbers that appear in your code generation report.

6 Click the `<S1>:1:45` hyperlink in this traceability comment:

```
/* Action '3': '<S1>:1:45' */
```

In the Truth Table Editor, row 3 of the Action Table appears highlighted.



7 You can also trace a condition, decision, or action in the table to a line of generated code. For example, right-click a cell in the column `D2` and select **Real-Time Workshop > Navigate to Code** from the context menu.

The code for that decision appears highlighted in `sf_climate_control.c`.

```
82          /*  Turn On Humidifier */
83          /* Action '3': '<S1>:1:45' */
84          rtb_humidifier = 1.0;
85        } else if (eml_aVarTruthTableCondition_1) {
86          /* Decision 'D2': '<S1>:1:16' */
```

Highlighted line of code

Note To select **Real-Time Workshop > Navigate to Code** for a condition, decision, or action, right-click a cell in the row or column that corresponds to that truth table element.

Bidirectional Traceability for Graphical Functions

You can see how bidirectional traceability works for graphical functions by following these steps:

1. Type `sf_clutch` at the MATLAB prompt.
2. Complete steps 2–6 in "Bidirectional Traceability for States and Transitions" on page 22-80.
3. Go to the **Solver** pane in the Configuration Parameters dialog box. In the **Solver options** section, select `Fixed-step` in the **Type** field. Then click **Apply**.

   Note Because this demo model does not work with variable-step solvers, you must perform this step before generating code.

4. In the **Real-Time Workshop** pane of the Configuration Parameters dialog box, click **Build** in the lower right corner.

   The code generation report appears automatically.
5. Click the `sf_clutch.c` hyperlink in the report.
6. Scroll down through the code to see the traceability comments.

```
235            case sf_clutch_IN_Slipping:
236             /* Graphical Function 'detectLockup': '<S1>:10' */
237             /* Transition: '<S1>:28' */
238             /* Graphical Function 'getSlipTorque': '<S1>:3' */
```

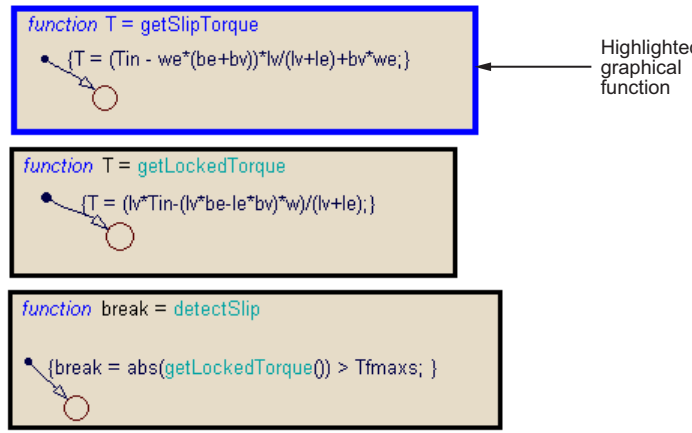
Traceability comment for a graphical function

Note The line numbers shown above can differ from the numbers that appear in your code generation report.

7 Click the `<S1>:3` hyperlink in this traceability comment:

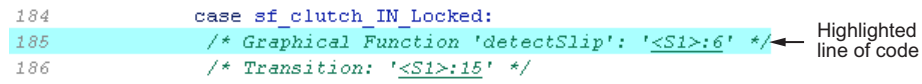
```
/* Graphical Function 'getSlipTorque': '<S1>:3' */
```

In the Stateflow Editor, the graphical function `getSlipTorque` appears highlighted.



8 You can also trace a graphical function in the Stateflow Editor to a line of generated code. For example, right-click the graphical function `detectSlip` and select **Real-Time Workshop > Navigate to Code** from the context menu.

The code for that graphical function appears highlighted in `sf_clutch.c`.

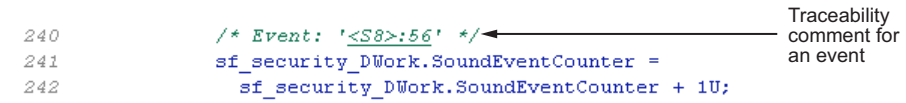


Code-to-Model Traceability for Events

You can see how code-to-model traceability works for events by following these steps:

1. Type `sf_security` at the MATLAB prompt.
2. Complete steps 2–6 in "Bidirectional Traceability for States and Transitions" on page 22-80.
3. In the **Real-Time Workshop** pane of the Configuration Parameters dialog box, click **Build** in the lower right corner.

   The code generation report appears automatically.
4. Click the `sf_security.c` hyperlink in the report.
5. Scroll down through the code to see the following traceability comment.



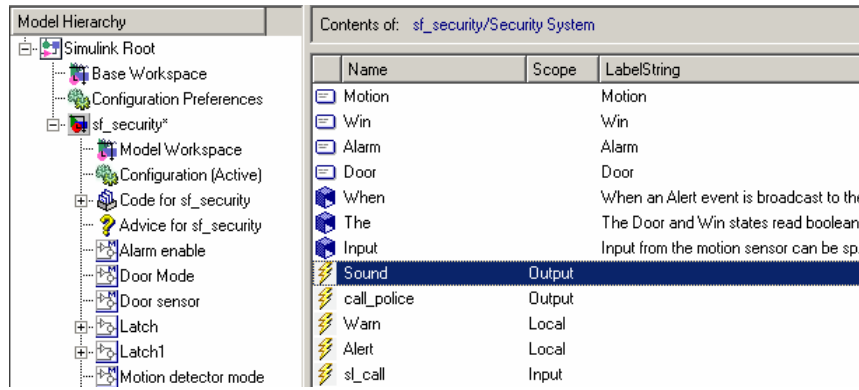
Note The line numbers shown above can differ from the numbers that appear in your code generation report.

---

6. Click the `<S8>:56` hyperlink in this traceability comment:

   ```
   /* Event: '<S8>:56' */
   ```

   In the **Contents** pane of the Model Explorer, the event `Sound` appears highlighted.



Model-to-Code Traceability for Junctions

You can see how model-to-code traceability works for junctions by following these steps:

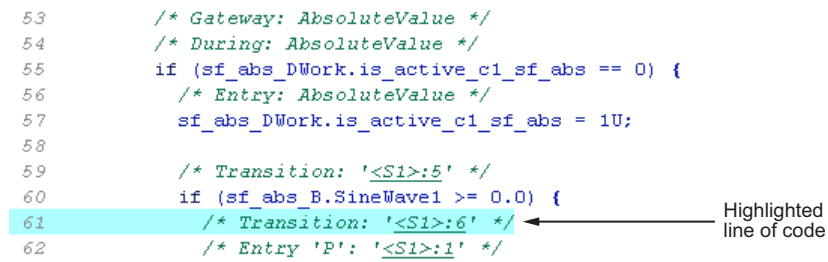
1. Type `sf_abs` at the MATLAB prompt.
2. Complete steps 2–6 in "Bidirectional Traceability for States and Transitions" on page 22-80.
3. Go to the **Solver** pane in the Configuration Parameters dialog box. In the **Solver options** section, select `Fixed-step` in the **Type** field. Then click **Apply**.

   Note Because this demo model does not work with variable-step solvers, you must perform this step before generating code.

4. In the **Real-Time Workshop** pane, click **Build** in the lower right corner.

   The code generation report appears automatically.
5. Open the `AbsoluteValue` chart in the Stateflow Editor.
6. Right-click the left junction and select **Real-Time Workshop > Navigate to Code** from the context menu.

The code for the first outgoing transition of that junction appears highlighted in `sf_abs.c`.



Format of Traceability Comments

The format of a traceability comment depends on the Stateflow object type.

State

Syntax.

```
/* <ActionType> '<StateName>': '<ObjectHyperlink>' */
```

Example.

```
/* During 'gear_state': '<S5>:2' */
```

This comment refers to the `during` action of the state `gear_state`, which has the hyperlink `<S5>:2`.

Transition

Syntax.

```
/* Transition: '<ObjectHyperlink>' */
```

Example.

```
/* Transition: '<S5>:12' */
```

This comment refers to a transition, which has the hyperlink `<S5>:12`.

Embedded MATLAB Function

Syntax.

```
/* Embedded MATLAB Function '<Name>': '<ObjectHyperlink>' */
```

Within the inlined code for an Embedded MATLAB function, comments that link to individual lines of the function have the following syntax:

```
/* '<ObjectHyperlink>' */
```

Examples.

```
/* Embedded MATLAB Function 'test_function': '<S50>:99' */

/* '<S50>:99:20' */
```

The first comment refers to the Embedded MATLAB function named `test_function`, which has the hyperlink `<S50>:99`.

The second comment refers to line 20 of the Embedded MATLAB function in your chart.

Truth Table Block

Syntax.

```
/* Truth Table Function '<Name>': '<ObjectHyperlink>' */
```

Within the inlined code for a Truth Table block, comments for conditions, decisions, and actions have the following syntax:

```
/* Condition '#<Num>': '<ObjectHyperlink>' */
/* Decision 'D<Num>': '<ObjectHyperlink>' */
/* Action '<Num>': '<ObjectHyperlink>' */
```

`<Num>` is the row or column number that appears in the Truth Table Editor.

Examples.

```
/* Truth Table Function 'truth_table_default': '<S10>:100' */
```

```
/* Condition '#1': '<S10>:100:8' */
/* Decision 'D1': '<S10>:100:16' */
/* Action '1': '<S10>:100:31' */
```

The first comment refers to a Truth Table block named `truth_table_default`, which has the hyperlink `<S10>:100`.

The other three comments refer to elements of that Truth Table block. Each condition, decision, and action in the Truth Table block has a unique hyperlink.

Truth Table Function

See "Truth Table Block" on page 22-91 for syntax and examples.

Graphical Function

Syntax.

```
/* Graphical Function '<Name>': '<ObjectHyperlink>' */
```

Example.

```
/* Graphical Function 'hello': '<S1>:123' */
```

This comment refers to a graphical function named `hello`, which has the hyperlink `<S1>:123`.

Simulink Function

Syntax.

```
/* Simulink Function '<Name>': '<ObjectHyperlink>' */
```

Example.

```
/* Simulink Function 'simfcn': '<S4>:10' */
```

This comment refers to a Simulink function named `simfcn`, which has the hyperlink `<S4>:10`.

Event

Syntax.

```
/* Event: '<ObjectHyperlink>' */
```

Example.

```
/* Event: '<S3>:33' */
```

This comment refers to an event, which has the hyperlink `<S3>:33`.

23

Debugging and Testing Stateflow Charts

Using the Stateflow Debugger

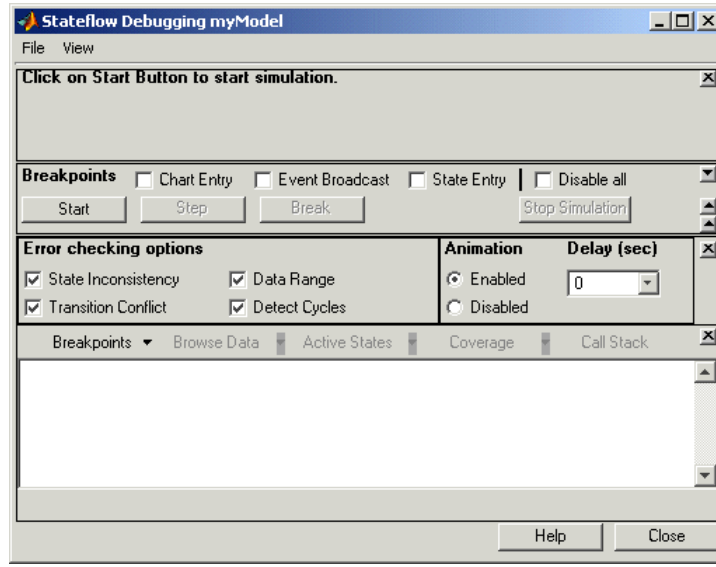
In this section...

Opening the Stateflow Debugger

To open the Stateflow Debugging window, follow these steps:

1 In the Stateflow Editor, select **Tools > Debug**.

The Stateflow Debugging window opens.



Animating Stateflow Charts

During simulation, you can animate Stateflow charts in a Simulink model to provide visual verification that your chart behaves as expected. Animation highlights objects in Stateflow charts as they execute. You can animate charts during simulation in two contexts:

- In *normal* mode on the host machine where you run MATLAB and Simulink software (see "Animating Stateflow Charts in Normal Mode" on page 23-3)
- In *external* mode on a target machine where your generated code runs (see "Animating Stateflow Charts in External Mode" on page 23-4)

Animating Stateflow Charts in Normal Mode

During simulation in normal mode on a host machine, you can animate states and transitions in a Stateflow chart. Follow these steps:

1. Open the Stateflow chart you want to animate.
2. In the Stateflow Editor, select **Tools > Debug** to open the Stateflow Debugging window.

3 In the Animation section of the Debugging window, select **Enabled**.

4 Control the speed of animation by entering a value in the **Delay** field, as follows:

- For the fastest animation, select a value of 0 seconds.
- For the slowest animation, select a value of 1 second.

5 Start simulation.

The Stateflow chart highlights states and transitions as they execute during simulation.

Animating Stateflow Charts in External Mode

You can animate Stateflow charts in external mode — the mode in which Real-Time Workshop code generation software establishes a communications link between a Simulink model and code executing on a target system (see "External Mode" in the Real-Time Workshop User's Guide). In external mode, you can animate states in Stateflow charts, and view test point signals in a floating scope or signal viewer.

- "Animating States During Simulation in External Mode" on page 23-4
- "Viewing Test Point Data in Floating Scopes and Signal Viewers" on page 23-6

Animating States During Simulation in External Mode. To animate states in Stateflow charts in external mode, follow these steps:

1 Load the Stateflow chart you want to animate to the target machine.

2 In the Stateflow Editor, select **Tools > Debug** to open the Stateflow Debugging window.

3 In the Animation section of the Debugging window, select **Enabled**.

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

5 In the left Select pane, select **Real-Time Workshop > Interface**.

6 In the Data exchange section of the right pane, select **External mode** from the drop-down menu in the Interface field and click **OK**.

7 In the Simulink model that contains the chart, select **Tools > External Mode Control Panel**.

8 In the External Mode Control Panel dialog box, click the **Signals & Triggering** button.

9 In the External Signal & Triggering dialog box, set these parameters:

In:	Select:
Signal selection pane	Stateflow chart block you want to animate
Trigger pane	**Arm when connecting to target** check box
Trigger pane	**normal** from drop-down menu in Mode field

10 Build the model to generate an executable file.

11 Start the target in the background by typing this command at the MATLAB prompt:

```
!model_name.exe -w &
```

For example, if the name of your model is `my_control_sys`, enter this command:

```
!my_control_sys.exe -w &
```

Note `-w` allows the target code to wait for the Simulink model connection

12 In the Simulink model editor, select **Simulation > External**, and then select **Simulation > Connect to Target**.

13 Start simulation.

The Stateflow chart will highlight states as they execute.

Viewing Test Point Data in Floating Scopes and Signal Viewers. When you simulate Stateflow charts in external mode, you can view test point data in floating scopes and signal viewers. In Stateflow charts, you can designate local data and states to be test points.

To view test point data during simulation in external mode, follow these steps:

1. Open the Model Explorer and for each data you want to view, follow these steps:
   a. In the left Model Hierarchy pane, select the state or local data of interest.
   b. In the right Dialog pane, select the **Test point** check box.
2. From a floating scope or signal viewer, click the signal selection button:

   The Signal Selector dialog box opens.
3. In the Signal Selector Model hierarchy pane, select the Stateflow chart.
4. In the Signal Selector Contents pane, list **Testpointed signals only** and check the test point signals you want to view.
5. Simulate the model in external mode as described in “Animating States During Simulation in External Mode” on page 23-4.

   The scope or viewer displays the values of the test point signals as the simulation runs.

Setting Breakpoints for Debugging a Chart

A breakpoint indicates a point at which the Stateflow Debugging window halts execution of a simulating Stateflow chart. At this time, you can inspect Stateflow data and the MATLAB workspace and examine the status of a simulating Stateflow chart.

The Stateflow Debugging window supports global and local breakpoints. Global breakpoints halt execution on any occurrence of the specific type of breakpoint. Local breakpoints halt execution on a specific object.

Setting Global Breakpoints

Use the **Breakpoint** controls in the Stateflow Debugging window to specify global breakpoints. When a global breakpoint is encountered during simulation, execution stops and the Debugger takes control. Select any or all of these breakpoints:

- **Chart Entry** — Simulation halts on chart entry.
- **Event Broadcast** — Simulation halts when an event is broadcast.
- **State Entry** — Simulation halts when a state is entered.

Global breakpoints can be changed during run-time and are immediately enforced. When you save the Stateflow chart, all the Stateflow Debugging window settings (including breakpoints) are saved, so that the next time you open the model, the breakpoints are as you left them.

Setting Local Breakpoints

You can set breakpoints for specific state actions, transitions, function calls, and event broadcasts in a Stateflow chart.

1. Open the properties dialog box of the object for which you want to set a breakpoint, as follows:
   a. Right-click the object from one of these sources:

Object	Right-Click In:	
	Stateflow Chart	Model Explorer
State	✓	✓
Transition	✓	
Function	✓	✓
Event		✓

b From the resulting pop-up menu, select **Properties**.

A dialog box appears for setting the properties of the object.

2 In the properties dialog box, select from the following breakpoints options:

For:	Select:
States	**State During** — Stop execution before performing the state during actions. **State Entry** — Stop execution before performing the state entry actions. **State Exit** — Stop execution before performing the state exit actions.
Transitions	**When Tested** — Stop execution before testing the transition to see if it is a valid path. **When Valid** — Stop execution after the transition tests valid, but before taking the transition.
Functions	**Function Call** — Stop execution before calling the function.
Events	**Start of Broadcast** — Stop execution before broadcasting the event. **End of Broadcast** — Stop execution after a Stateflow object reads the event.

Disabling All Breakpoints

To disable all breakpoints in the Debugger window, select the check box **Disable all**.

Clearing All Breakpoints

There is no button or check box in the Debugger window to clear breakpoints. To find and clear all breakpoints without disabling them, you must use a set of Stateflow API commands as shown below. (For more information, see the Stateflow API documentation.)

```
% get a handle for the root object
rootObj = find(sfroot,'-isa','Stateflow.Machine','Name',model);

% find all states, transitions, data, and charts
stateObjects = rootObj.find('-isa','Stateflow.State');
transitionObjects =rootObj.find('-isa','Stateflow.Transition');
dataObjects = rootObj.find('-isa','Stateflow.Data');
chartObjects = rootObj.find('-isa','Stateflow.Chart');

% for all states, clear their breakpoints
for i = 1:size(stateObjects,1)
stateObjects(i).Debug.Breakpoints.OnEntry = 0;
stateObjects(i).Debug.Breakpoints.OnDuring = 0;
stateObjects(i).Debug.Breakpoints.OnExit = 0;
stateObjects(i).Machine.Debug.BreakOn.ChartEntry = 0;
stateObjects(i).Machine.Debug.BreakOn.EventBroadcast = 0;
stateObjects(i).Machine.Debug.BreakOn.StateEntry = 0;
end

% for all transitions, clear their breakpoints
for i = 1:size(transitionObjects,1)
transitionObjects(i).Debug.Breakpoints.WhenTested = 0;
transitionObjects(i).Debug.Breakpoints.WhenValid = 0;
end

% for all data, clear their breakpoints
for i = 1:size(dataObjects,1)
dataObjects(i).Debug.Watch = 0;
end

% for all charts, clear their breakpoints
for i = 1:size(chartObjects,1)
chartObjects(i).Debug.Breakpoints.OnEntry = 0;
end
```

The first command returns a handle to the machine object that represents the top level of the Stateflow hierarchy. The next four commands use the API method `find` to specify the type of object to find. For example, the command

```
stateObjects = rootObj.find(`-isa','Stateflow.State')
```

searches through the `rootObj` and returns an array listing of all state objects in your model. (See Finding Objects and Properties in the Stateflow API documentation.)

You can also define the properties of Stateflow objects. For example, you can clear all breakpoints in your model by setting those property values to zero for all states, transitions, data, and charts as shown in the code.

Setting Error Checking in the Debugging Window

The options in the **Error checking options** section of the Stateflow Debugging window insert generated code in the simulation target to provide breakpoints to catch different types of errors that might occur during simulation. Select any or all of the following error checking options:

- **State inconsistency** — Check for state inconsistency errors that are most commonly caused by the omission of a default transition to a substate in superstates with XOR decomposition. See "Debugging State Inconsistencies in a Chart" on page 23-20 for a complete description and example.
- **Transition Conflict** — Check whether there are two equally valid transition paths from the same source at any step in the simulation. See "Debugging Conflicting Transitions in a Chart" on page 23-22 for a complete description and example.
- **Data Range** — Check whether the minimum and maximum values you specified for a data in its properties dialog box are exceeded. Also check whether fixed-point data overflows its base word size. See "Debugging Data Range Violations in a Chart" on page 23-24 for a complete description and example.
- **Detect Cycles** — Check whether a step or sequence of steps indefinitely repeats itself. See "Debugging Cyclic Behavior in a Chart" on page 23-26 for a complete description and example.

To include the supporting code designated for these debugging options in the simulation application, select the **Enable debugging/animation** check box in the **Simulation Target** pane of the Configuration Parameters dialog box. This option is described in "Speeding Up Simulation" on page 22-17.

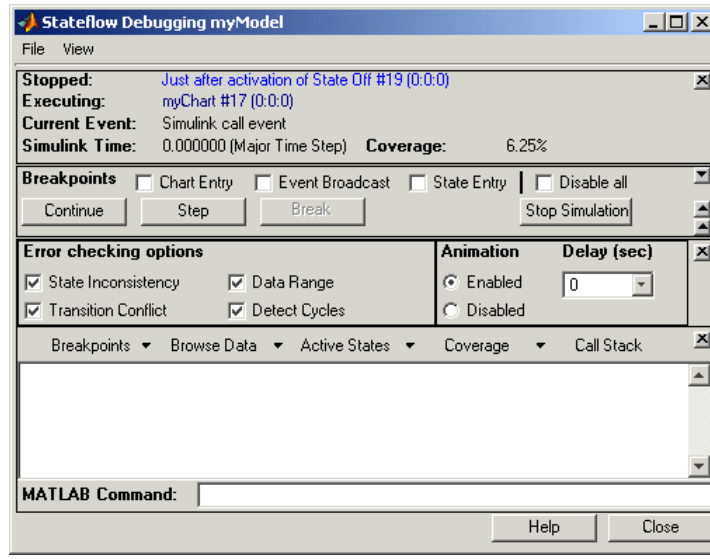
Note You must rebuild the target for any changes to the settings referenced above to take effect.

Starting Simulation in the Debugging Window

To debug the Stateflow charts in a model, you start simulation in the Debugging window with these steps:

1 Select the **Start** button.

A debugging simulation session starts. When a breakpoint that you set is encountered, the Stateflow Debugging window takes on the following appearance:



At the breakpoint, the following status items appear in the upper portion of the Debugger window:

- **Stopped** — Displays the step executed just prior to breaking execution.
- **Executing** — Displays the currently executing Stateflow chart.

- **Current Event** — Displays the event being processed by the Stateflow chart.
- **Simulink Time** — Displays the current simulation time.
- **Code Coverage** — Displays the percentage of code covered in this simulation.

During simulation, the Stateflow chart is marked read-only. The appearance of the Stateflow Editor toolbar and menus changes so that object creation is not possible. When the Stateflow Editor is in this read-only mode, its condition is referred to as *iced*.

Controlling the Execution Rate in the Debugging Window

Once you start simulation as described in "Starting Simulation in the Debugging Window" on page 23-11, and a breakpoint is reached, you can control the rate of execution of Stateflow charts to execute step-by-step or continuously until another breakpoint is reached. Use the following buttons in the Stateflow Debugging window to control the rate of execution:

- **Continue** — After simulation has been started, and a breakpoint has been encountered, the **Start** button is marked **Continue**. Press **Continue** to continue simulation.
- **Step** — Execute the next execution step, and suspend the simulation.
- **Break** — Suspend the simulation and transfer control to the Debugging window.
- **Stop Simulation** — Stop simulation altogether and relinquish debugging control. When simulation stops, the Stateflow Editor toolbar and menus return to their normal appearance and operation so that object creation is again possible.

Setting the Output Display Pane

During simulation, the Debugging window monitors a variety of execution indicators in its output display in the bottom pane of the Debugging window. You select the contents of this display with the following pull-downs located just above the display, which are enabled only after a breakpoint is reached during simulation.

- **Breakpoints** — Display a list of the set breakpoints. You can set breakpoints in the Debugger and in the properties dialogs of individual objects such as states, transitions, and functions. See "Setting Breakpoints for Debugging a Chart" on page 23-6 for details. This option lists breakpoints for the currently executing chart or for all charts in the model.
- **Browse Data** — Display the current values of defined data objects. This pull-down list lets you filter displayed data between all data and watched data. Watched data has the **Data** property **Watch in Debugger** enabled for it. Each of these categories is further filtered by data for the currently executing chart, or all charts in the model. For more details see "Watching Data in the Stateflow Debugger" on page 23-30.
- **Active States** — Display a list of active states in the display area. Double-clicking any state causes the Stateflow Editor to display that state. This pull-down lets you display active states in the current chart, or active states for all charts in the model.
- **Call Stack** — Display a sequential list of the **Stopped** and **Current Event** status items that occur with each single-step through the simulation.

Once you make a selection, the pull-down menu corresponding to the current display is highlighted. Once you select an output display button, that type of output is displayed until you choose a different display type. You can clear the display by selecting **Clear Display** from the **File** menu of the Stateflow Debugging window.

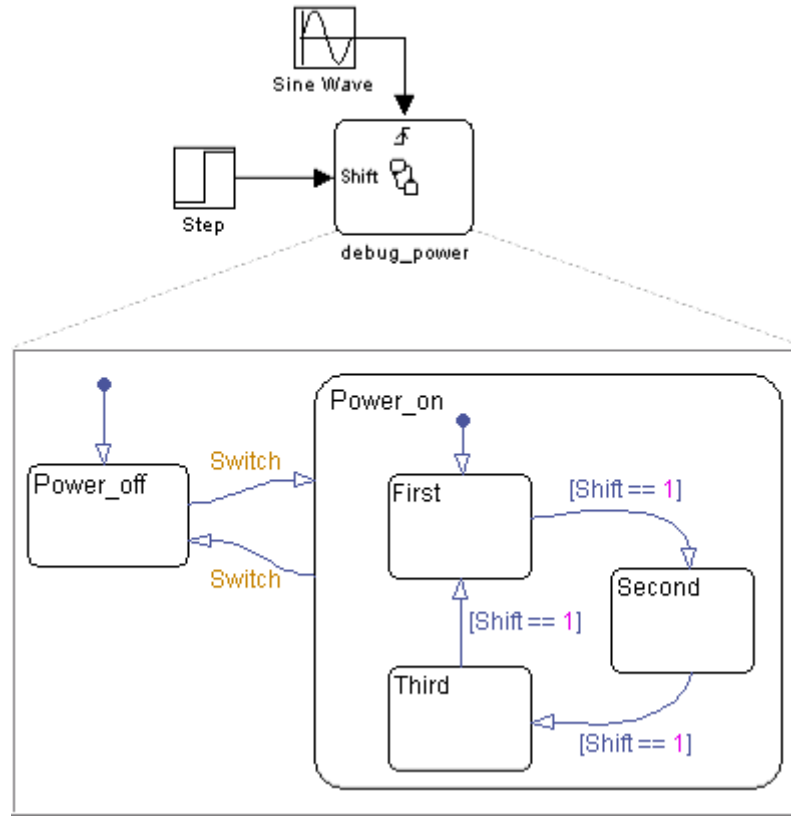
Example of Debugging Run-Time Errors in a Chart

In this section...

Creating the Model and the Stateflow Chart

In this topic, you create a model with a Stateflow chart to debug. Follow these steps:

1 Create the following Simulink model and Stateflow chart:



2. From the Stateflow Editor, add an event `Switch` with a scope of **Input from Simulink** and a **Rising Edge** trigger.

3. Also add a data `Shift` with a scope of **Input from Simulink**.

The Stateflow 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 Simulink 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`.

Debugging the Stateflow Chart

You create an example model with a Stateflow chart that needs debugging in "Creating the Model and the Stateflow Chart" on page 23-14. To debug the Stateflow chart, follow these steps:

1. In the Stateflow Editor, select **Tools > Open Simulation Target**.

   The **Simulation Target** pane of the Configuration Parameters dialog box appears.

2. Verify that **Enable debugging/animation** is selected.

3. Click **OK** in the Configuration Parameters dialog box.

4. In the Stateflow Editor, select **Tools > Debug**.

   The Stateflow Debugging window opens.

5. Select the **Chart Entry** option in the **Breakpoint** section.

6. Under **Animation**, select **Enabled** to enable animation of Stateflow charts during simulation.

7. In the Stateflow Debugging window, click **Start** to start the simulation.

   Informational messages appear in the MATLAB Command Window. The Stateflow Editor toolbar and menus change appearance to indicate read-only status. The Stateflow chart is parsed, the code is generated, and the target is built.

   Because you specified a breakpoint on chart entry, the execution stops at that point and the Debugger shows these messages:

   ```
   Stopped: Just after entering during function
   ```

```
   of Chart debug_power
Executing: sf_debug_ex_debug_power
Current Event: Input event Switch
```

8 Click **Step**.

The **Step** button executes the next step and stops.

9 Continue clicking the **Step** button and watching the animating Stateflow chart.

After each step, watch the Stateflow chart animation and the Debugger status area to see the sequence of execution.

Single-stepping shows that the Stateflow 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.

Correcting the Run-Time Error

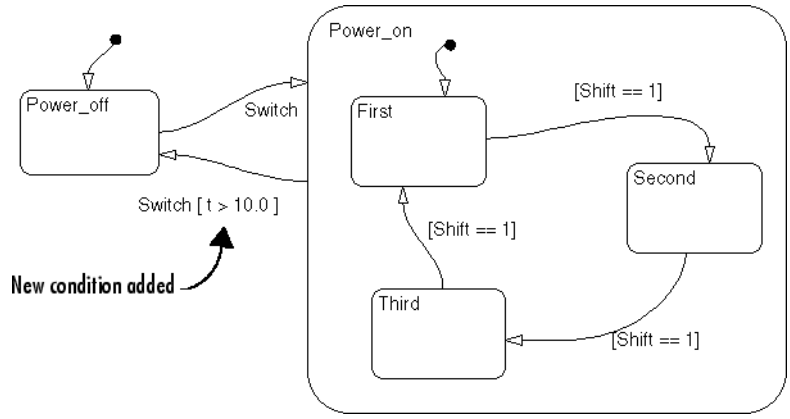
In "Debugging the Stateflow Chart" on page 23-16, you step through a simulation of an example Stateflow chart and find an error: the event `Switch` drives the simulation but the simulation time passes too quickly for the input data object `Shift` to have an effect.

Correct this error as follows:

1 In the Stateflow Editor, select **Simulation > Stop**.

The Stateflow Editor is now writable. You can edit the chart.

2 Add the condition `[t > 10.0]` 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 10.0.

3. In the Stateflow Debugging window, click **Start** to begin simulation again.

4. Click **Step** repeatedly to observe the new behavior.

Identifying Stateflow Objects in Error Messages

When an error message appears during simulation, it refers to the relevant Stateflow object using its name and ID number. An example of an error message is:

```
Unresolved event 'Switch' in transition Switch (#100)
```

The ID number of a Stateflow object is unique, but not its name. To identify an object using its ID number, perform one of these steps:

- Use these Stateflow API commands at the MATLAB prompt:

  ```
  >> theObject = find(sfroot, 'Id', <id number>);
  >> theObject.view
  ```

  The first command finds the Stateflow object corresponding to the `<id number>` that you specify. The second command highlights the chosen object in the Stateflow Editor. (See the Stateflow API documentation for information about the `find` and `view` methods.)

- Right-click an object in the chart and select **Send to Workspace** from the context menu.

  The properties of the object appear in the MATLAB Command Window. The ID number of the object appears in the list of properties.

Debugging State Inconsistencies in a Chart

In this section...

Definition of State Inconsistency

States in a Stateflow chart are inconsistent if they violate any of these rules:

- An active state (consisting of at least one substate) with exclusive (OR) decomposition has exactly one active substate.
- All substates of an active state with parallel (AND) decomposition are active.
- All substates of an inactive state with either exclusive (OR) or parallel (AND) decomposition are inactive.

Causes of State Inconsistency

State inconsistency errors are most commonly caused by the omission of a default transition to a substate in superstates with exclusive (OR) decomposition.

Design errors in complex Stateflow charts can also result in state inconsistency errors. You can detect these errors using the Debugger at runtime.

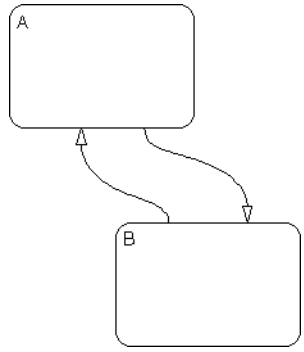
Detecting State Inconsistency

To detect the state inconsistency during a simulation,

1. Build the target with debugging enabled.
2. Invoke the Debugger and enable **State Inconsistency** checking.
3. Start the simulation.

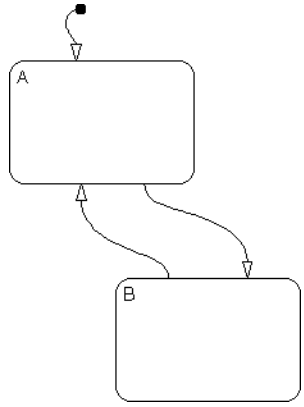
State Inconsistency Example

This Stateflow chart has a state inconsistency.



In the absence of a default transition indicating which substate is to become active, the simulation encounters a run-time state inconsistency error.

Adding a default transition to one of the substates resolves the state inconsistency.



Debugging Conflicting Transitions in a Chart

In this section...

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, Stateflow software 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 of Outgoing Transitions" on page 3-22.
- For implicit ordering, evaluation of conflicting transitions occurs based on internal rules described in "Implicit Ordering of Outgoing Transitions" on page 3-27.

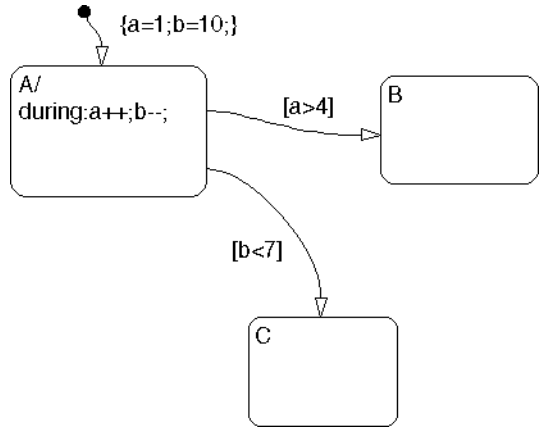
How to Detect Conflicting Transitions

To detect conflicting transitions during a simulation, follow these steps:

1. Build the target with debugging enabled.
2. Invoke the Debugger and enable **Transition Conflict** checking.
3. Start the simulation.

Example of Conflicting Transitions

This Stateflow chart has two conflicting transitions.



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.

Resolution of Transition Conflict 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 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.

Resolution of Transition Conflict 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.

Debugging Data Range Violations in a Chart

In this section...

Types of Data Range Violations

Stateflow software detects the following data range violations during simulation:

- If a data object equals a value outside the range of the values set in the **Initial**, **Minimum**, and **Maximum** fields specified in the Data properties dialog box

  See "Setting Data Properties in the Data Dialog Box" on page 8-6 for a description of the **Initial**, **Minimum**, and **Maximum** fields in the Data properties dialog box.

- If the fixed-point result of a fixed-point operation overflows its bit size

  See "Overflow Detection for Fixed-Point Types" on page 14-10 for a description of the overflow condition in fixed-point numbers.

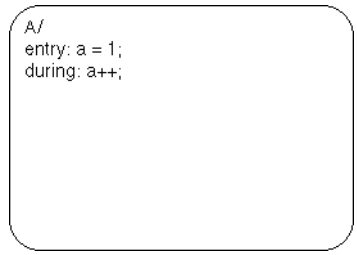
Detecting Data Range Violations

To detect data range violations during a simulation:

1. Build the target with debugging enabled.
2. Open the Debugger window.
3. In the **Error checking options** of the Debugger, select **Data Range**.
4. Start the simulation.

Data Range Violation Example

This Stateflow chart has a data range violation.



The data a is defined to have an **Initial** and **Minimum** value of 0 and a **Maximum** value of 2. Each time an event awakens this Stateflow chart and state A is active, a increments. The value of a quickly becomes a data range violation.

Debugging Cyclic Behavior in a Chart

In this section...

What Is Cyclic Behavior?

Cyclic behavior is a step or sequence of steps that is repeated indefinitely (recursive). The Stateflow Debugger uses cycle detection algorithms to detect a class of infinite recursions caused by event broadcasts.

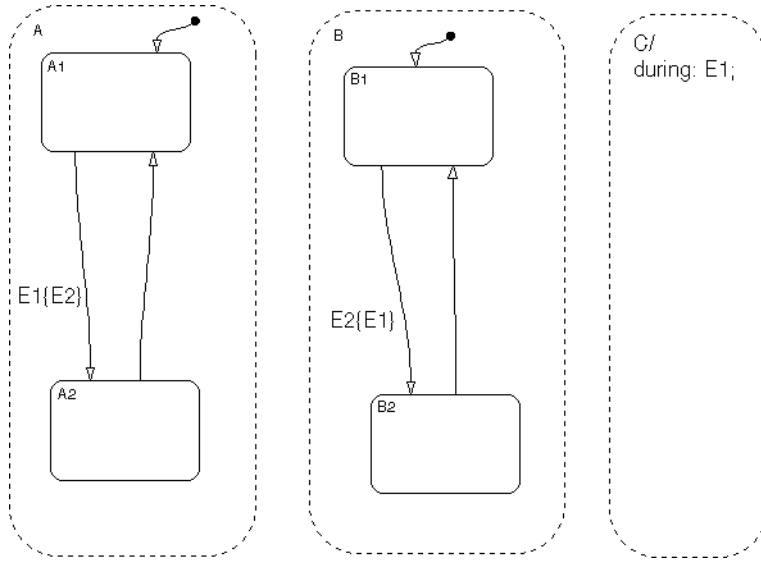
Detecting Cyclic Behavior During Simulation

To detect cyclic behavior during a simulation, do the following:

1. Build the target with debugging enabled.
2. Invoke the Debugger and enable **Detect Cycles**.
3. Start the simulation.

Cyclic Behavior Example

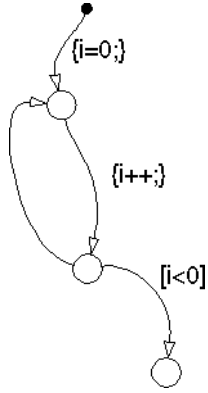
This Stateflow chart shows a typical example of a cycle created by infinite recursions caused by an event broadcast.



When the state `C` `during` action executes, event `E1` is broadcast. The transition from state `A.A1` to state `A.A2` becomes valid when event `E1` is broadcast. Event `E2` is broadcast as a condition action of that transition. The transition from state `B.B1` to state `B.B2` becomes valid when event `E2` is broadcast. Event `E1` is broadcast as a condition action of the transition from state `B.B1` to state `B.B2`. Because these event broadcasts of `E1` and `E2` are in condition actions, a recursive event broadcast situation occurs. Neither transition can complete.

Flow Cyclic Behavior Not Detected Example

This Stateflow chart shows an example of cyclic behavior in a flow graph that is not detected by the Debugger.

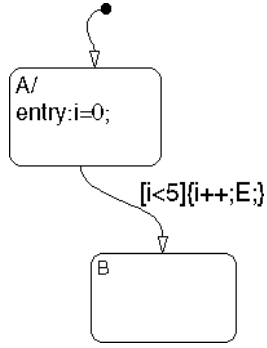


The data object `i` is set to 0 in the condition action of the default transition. `i` is incremented in the next transition segment condition action. The transition to the third connective junction is valid only when the condition `[i < 0]` is true. This condition will never be true in this flow graph and there is a cycle.

This cycle is not detected by the Debugger because it does not involve event broadcast recursion. Detecting cycles that depend on data values is not currently supported.

Noncyclic Behavior Flagged as a Cyclic Example

This Stateflow chart shows an example of noncyclic behavior that the Debugger flags as being cyclic.



State A becomes active and i is initialized to 0. When the transition is tested, the condition [i < 5] is true. The condition actions that increment i and broadcast the event E are executed. The broadcast of E when state A is active causes a repetitive testing (and incrementing of i) until the condition is no longer true. The Debugger flags this as a cycle when in reality, the apparent cycle is broken when i becomes greater than 5.

Watching Data Values with Debuggers

In this section...

Watching Data in the Stateflow Debugger

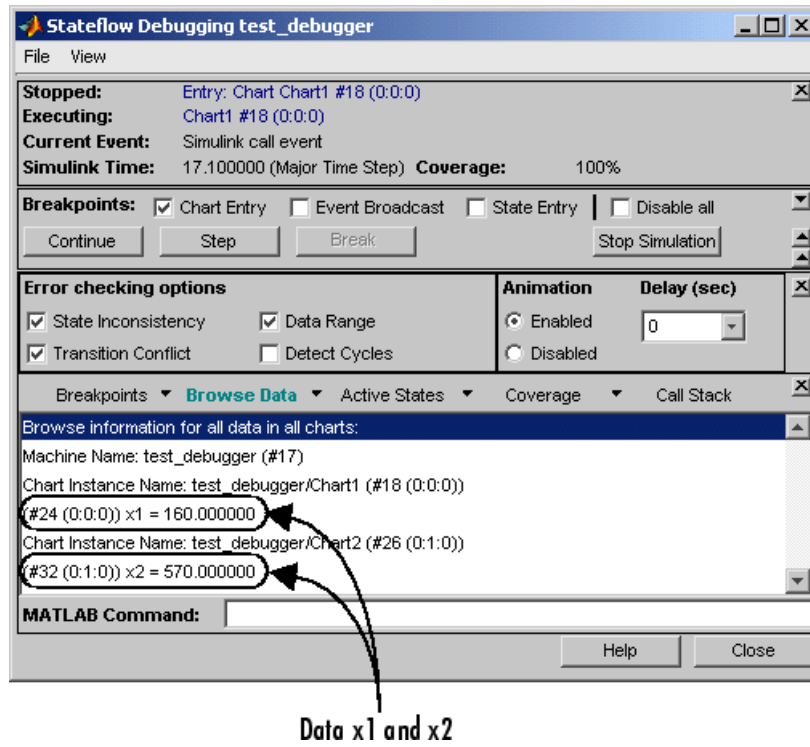
The **Browse Data** pull-down menu in the Stateflow Debugger lets you display selected data in the bottom output display pane of the Stateflow Debugger during simulation, after a breakpoint is reached. The Debugger can filter the display between:

- Watched data and all data
- Watched data in the currently executing chart and watched data for all charts in a model

---

Note You designate Stateflow data to be *watched data* by enabling the property "Watch in Stateflow Debugger" on page 8-23, as described in "Properties You Can Set in the Value Attributes Pane" on page 8-20.

The following example displays **All Data (All Charts)** for two executing charts, `Chart1` and `Chart2`, in a simulating model. Each chart has its own data value: `x1` and `x2`, respectively.

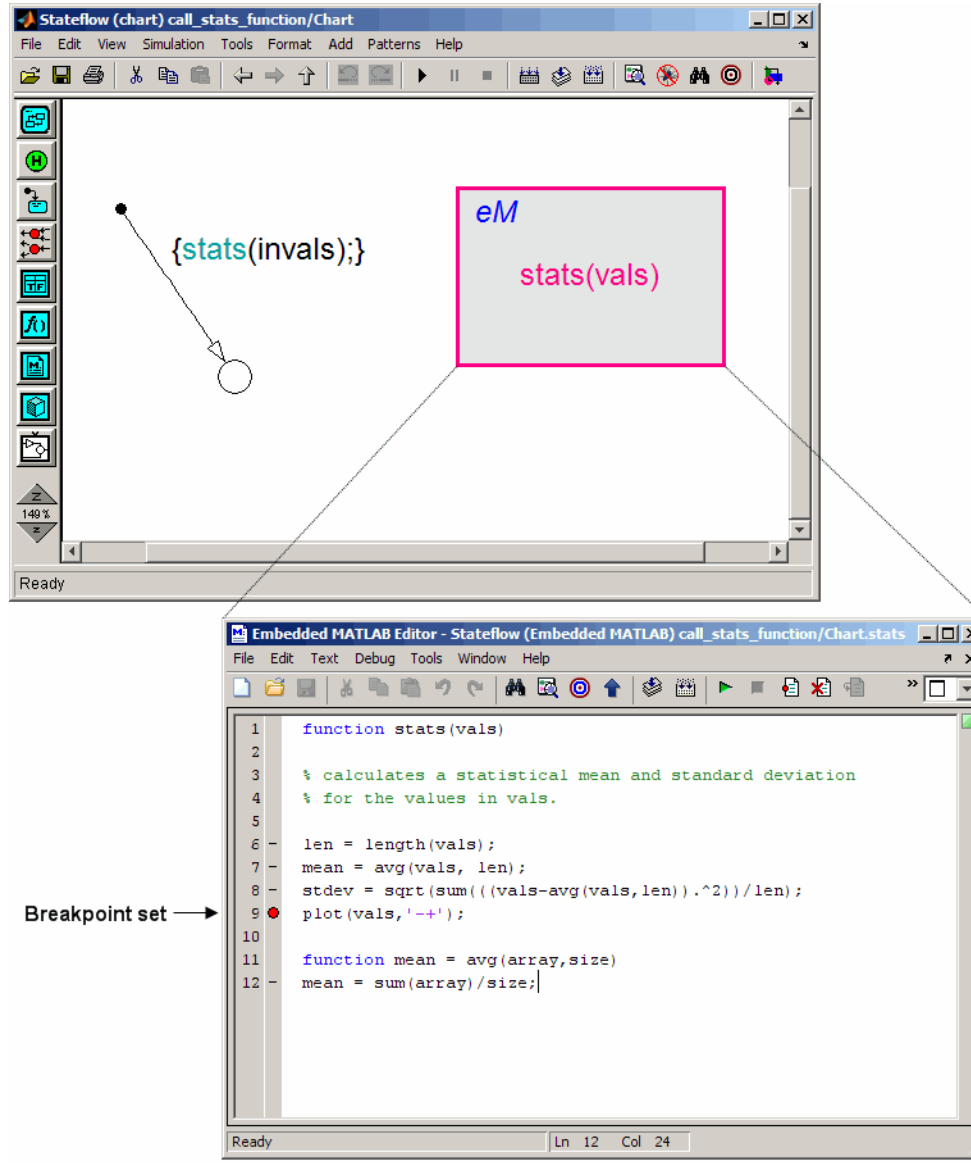


The data for each chart is headed by its owning object. Each displayed object (chart, state, data, and so on) is accompanied by a unique identifier in the form (`#id(xx:yy:zz)`), which is used in linking the listed object to its appearance in the Stateflow chart.

Note Fixed-point data appears with two values: the quantized integer value (stored integer) and the scaled real-world (actual) value. For more information, see "How Fixed-Point Data Works in Stateflow Charts" on page 14-5.

Watching 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 example, a default transition calls an Embedded MATLAB function with a breakpoint set at the last executable line of the function:


Stateflow (chart) call_stats_function/Chart
File Edit View Simulation Tools Format Add Patterns Help
{stats(invals);}
eM
stats(vals)
Ready
Embedded MATLAB Editor - Stateflow (Embedded MATLAB) call_stats_function/Chart.stats
File Edit Text Debug Tools Window Help
function stats(vals)
% 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);
plot(vals,'-+');
function mean = avg(array,size)
mean = sum(array)/size;
Breakpoint set
Ready
Ln 12 Col 24

When simulation reaches the breakpoint, you can display Stateflow data in the MATLAB Command Window. Assuming you want to watch the data variable `vals` from the previous example, follow these steps:

1. At the MATLAB prompt, press **Enter**.

   A `debug>>` prompt appears.

2. Enter the MATLAB command `whos` to view the data that is visible at the current scope.

```
debug>> whos
  Name        Size                Bytes  Class

  vals        4x1                    32  double array
  len         1x1                     8  double array
  stdev       1x1                     8  double array
  mean        1x1                     8  double array
  invals      4x1                    32  double array

Grand total is 5 data in scope

debug>>
```

3. Enter the name of data array `vals` at the prompt to display its value.

```
debug>> vals

 vals =

     2
     3
     4
     5

debug>>
```

4. Enter `vals(2:3)` to view the values of a submatrix of the array.

```
debug>> vals (2:3)
```

```
ans =

     3
     4

debug>>
```

The Command Line Debugger provides these commands during simulation:

Command	Description
`dbstep`	Advance to next executable line of code.
`dbstep [in/out]`	When debugging Embedded MATLAB functions: • `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*.

Command	Description
`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 Embedded MATLAB function.

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 string, for example, `evalin('base','whos')`.

Note To return to the MATLAB base workspace, use the `dbquit` command.

Monitoring Test Points in Stateflow Charts

In this section...

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 the following Stateflow objects as test points:

- Any state
- Local data with the following characteristics:
  - Can be scalar, one-dimensional, or two-dimensional in size
  - Can be any data type except `ml`
  - Must be a descendant of a Stateflow chart

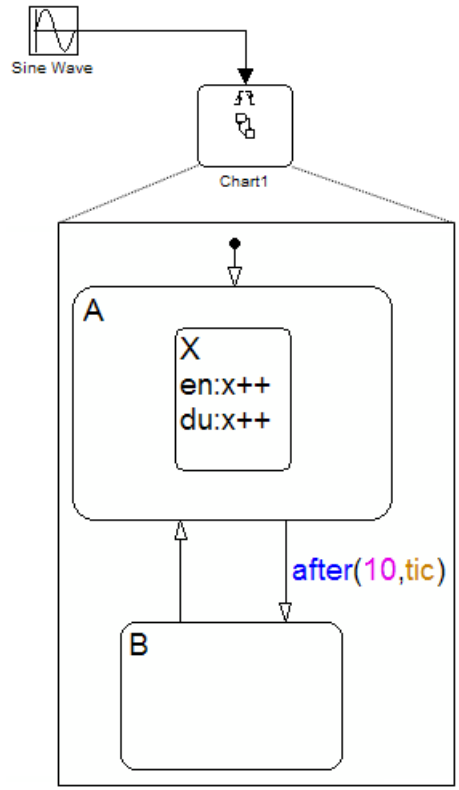
You implicitly declare all states and local data as test points by selecting the **Enable debugging/animation** option for the Stateflow simulation target in the Configuration Parameters dialog box. If you do not select this option, you can specify individual local data or states as test points by setting their **TestPoint** property via the Stateflow API or in the Model Explorer (see "Setting Test Points for Stateflow States and Local Data with the Model Explorer" on page 23-38).

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.

Setting Test Points for Stateflow States and Local Data with the Model Explorer

You can explicitly set individual states or local data as test points through the Model Explorer. Use the example you create in the following procedure to learn how to set individual test points for Stateflow states and data.

1 Create this model.



The model consists of a single Stateflow block named Chart1, which is triggered by a signal from a Sine Wave block through the input trigger event `tic`. In the Stateflow chart, the state `A` and its substate `X` are entered for 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 `x` is added to the state `X`. The `entry` and `during` actions for substate `X` increment `x` while `X` is active for 10 `tic` events. When state `B` is entered, `x` reinitializes to zero, and the cycle repeats.

2 Save the model as `myModel.mdl`.

3 Start the Model Explorer. In the Simulink model, select **View > Model Explorer**.

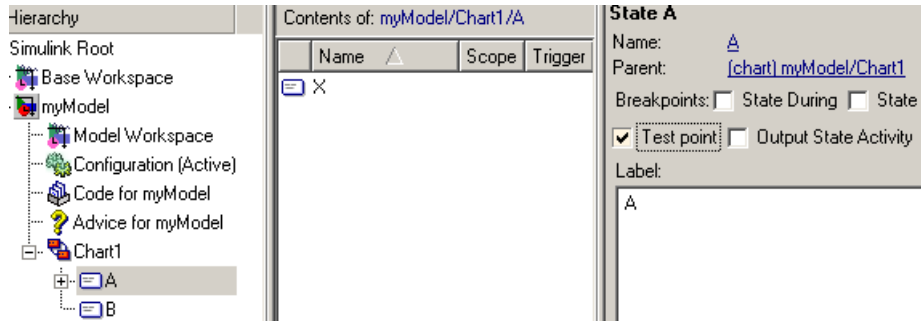
The Model Explorer appears.

4 In the Model Explorer, expand `myModel`.

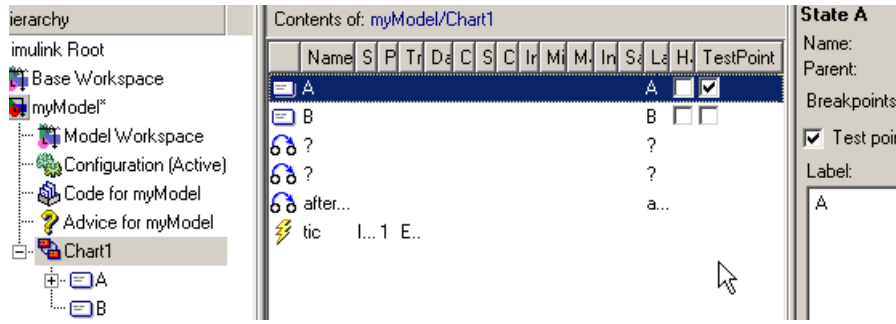
5 Expand `Chart1`, then select `A`.

6 In the rightmost pane, **State A**, select the **Test point** check box. Click **Apply**.

This action creates a test point for the state `A`.



Alternatively, you can access a test point through the middle pane. By default, the Model Explorer displays event and data child objects in the **Contents** pane for the selected object in the **Model Hierarchy** pane. You can set the test point for the state `A` through this pane by selecting the parent of `A`. If the states do not appear in the middle pane, select the **States/Functions/Boxes/Etc.** check box in the **View > List View Options** for **All Stateflow Objects**.



7 Repeat step 6 for state `X`. Click **Apply**.

8 Select `X` again. Select the local data `x` in the **Contents** pane.

9 In the rightmost pane for that data, select the **Value Attributes** tab and then select the **Test point** check box. Click **Apply**.

10 Repeat step 6 for state `B`. Click **Apply** and save the model.

You can now log these test points. See "Logging Data Values and State Activity" on page 23-40 for instructions on using the Signal Logging dialog box. See "Logging Data Values Using the Command Line API" on page 23-45 for instructions on logging signals at the MATLAB command line.

Logging Data Values and State Activity

During simulation, you can log values for data and state activity into Simulink objects. After simulation, you can access these objects in the MATLAB workspace and use them to report and plot the values.

You can use the following procedure to learn how to access logged Stateflow data and state activity. This procedure uses the model, `myModel`, from the preceding topic, "Setting Test Points for Stateflow States and Local Data with the Model Explorer" on page 23-38.

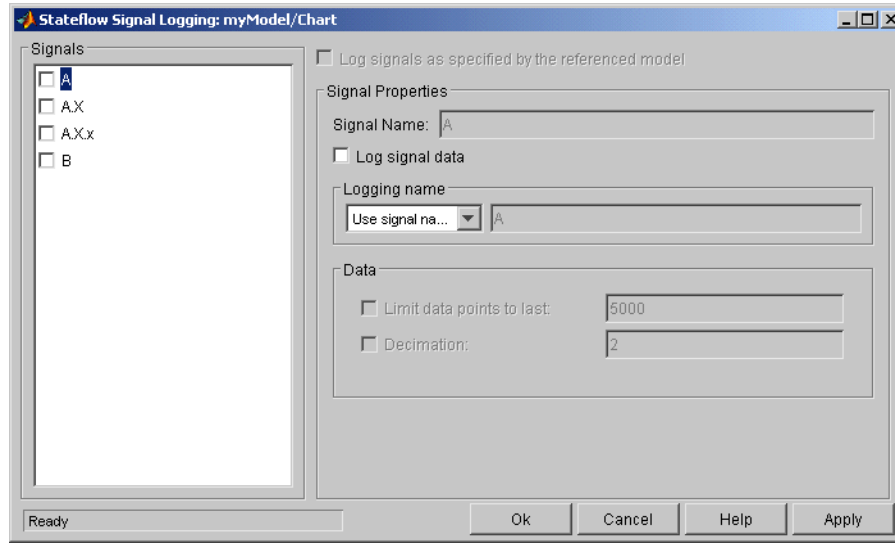
1 If `myModel` is not already open, at the MATLAB prompt, type

```
>> myModel
```

The model appears.

2 In the Simulink model window, right-click the Stateflow block and select **Log Chart Signals**.

The Signal Logging dialog box appears, as shown.



3 Select the check box next to A.

This box is the state activity signal for state A. When A is active, its value is 1. When A is inactive, its value is 0.

After checking A, notice these properties in the right pane of the Signal Logging dialog box:

Signal Properties	Description
Signal Name	Name of the highlighted state or data.
Log signal data	Checking this selects the highlighted signal in the **Signals** pane.

Signal Properties	Description
Logging name	Name of the signal logged. By default, this is set to the name of the selected/highlighted state or data. You can select `Custom` for this property to rename the selected/highlighted signal in the adjacent field to the right.
Limit data points to last	Select this property to enter the number of most recent sample values to log in the adjacent field to the right for the selected/highlighted signal.
Decimation	Select this property to enter the level of decimation for the signal values logged for the selected/highlighted signal.

4. Select all the signals in the **Signal** pane and click **OK** to close the Signal Logging dialog box.

5. Simulate the model.

   During simulation, the Simulink model data log object `logsout` is generated in the MATLAB workspace.

6. After simulation, enter this string at the MATLAB prompt:

```
>> logsout
```

   You see this result:

```
logsout =

Simulink.ModelDataLogs (myModel):
Name                    Elements  Simulink Class

Chart1                     4      StateflowDataLogs
```

   The display identifies `logsout` as a Simulink object of type `ModelDataLogs`. This type is the highest level logging object. The object `Chart1` appears as the only contents of `logsout` and represents logged data for the Stateflow block. `Chart1` is identified as a Simulink object of type `StateflowDataLogs`.

7 At the MATLAB prompt, enter this string:

```
>> logsout.Chart1
```

You see this result:

```
ans =

Simulink.StateflowDataLogs (Chart1):
  Name                    Elements  Simulink Class

  ('A.X.x')                  1      Timeseries
  A                          1      Timeseries
  ('A.X')                    1      Timeseries
  B                          1      Timeseries
```

The signals that you selected in the Signal Logging dialog box appear as Simulink objects of type `Timeseries`. Notice that the signals for the activity of state `X` and the value of data `x` appear as `('A.X')` and `('A.X.x')`, respectively. Because of the way that logged signals are stored, you must use dot notation to access logged data for Stateflow objects below chart level in the Stateflow chart.

8 At the MATLAB prompt, enter this string:

```
>> logsout.Chart1.('A.X.x')
```

You see this result:

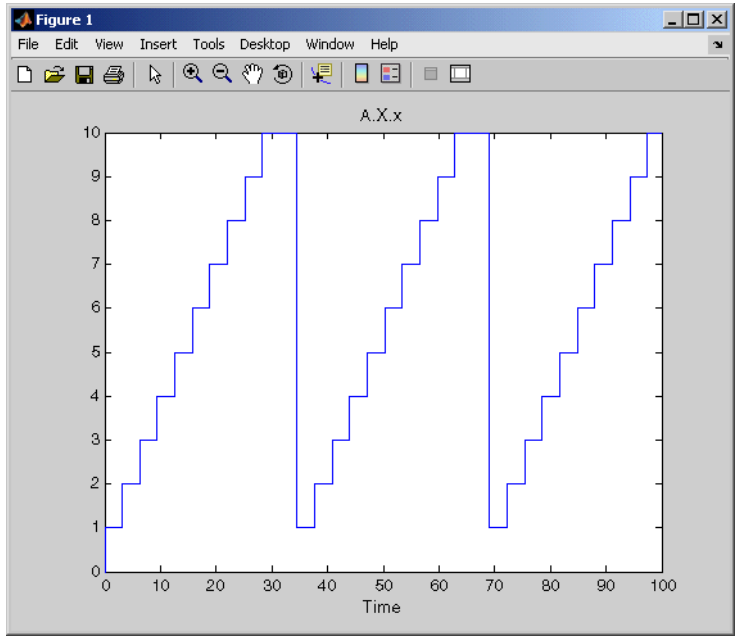
```
ans =

          Name: 'A.X.x'
     BlockName: 'StateflowChart/A.X.x'
     PortIndex: 1
    SignalName: 'A.X.x'
    ParentName: 'A.X.x'
      TimeInfo: [1x1 Simulink.TimeInfo]
          Time: [114x1 double]
          Data: [114x1 double]
```

The logging object for the data `x`, `('A.X.x')`, is actually a structure of logged data pertinent to `x`. The actual logged signal values for `x` are contained in the `Data` object, a vector of 114 values. For example, if you were to enter the MATLAB command `logsout.Chart1.('A.X.x').Data`, a long stream of data would appear. A better way to see the logged values of `x` is to use the `plot` method shown in the next step.

9 At the MATLAB prompt, plot the values of `x` with this command:

```
>> logsout.Chart1.('A.X.x').plot
```

You see this result:



The preceding plot exhibits the expected results for the value of `x`. It increments for 10 time steps before resetting to 0 when states `X` and `A` are exited and state `B` is entered in the Stateflow chart.

The preceding example shows some capabilities you have for reporting logged Stateflow data. Stateflow data conforms to the general rules for handling logging signals in Simulink models.

Logging Data Values Using the Command Line API

You can also specify which signals to log by using API commands at the MATLAB prompt. The following procedure uses the model, `myModel`, from the previous topic "Setting Test Points for Stateflow States and Local Data with the Model Explorer" on page 23-38.

1. If `myModel` is not already open, at the MATLAB prompt, type:

   ```
   >> myModel
   ```

   The model appears.

2. To define a Simulink object of type `SigPropNode` for the Stateflow chart, use this command:

   ```
   >> signal_properties = ...
   get_param('myModel/Chart1','AvailSigsInstanceProps')
   ```

   You see this result:

   ```
   signal_properties =

   Simulink.SigPropNode
   ```

3. To retrieve the contents of this object, use the API method `get`:

   ```
   >> signal_properties.get
   ```

   You see this result:

   ```
      Path: 'StateflowChart'
      Name: 'Chart1@StateflowChart'
      Type: 'Stateflow'
   Signals: [4x1 Simulink.SigProp]
   ```

   You can log four signals in the chart.

4. To view the properties of the first signal, type:

   ```
   >> signal_properties.Signals(1).get
   ```

   This list appears:

```
          SigName: 'A.X.x'
        BlockPath: 'StateflowChart/A.X.x'
        PortIndex: 1
        LogSignal: 0
    UseCustomName: 0
          LogName: 'A.X.x'
  LimitDataPoints: 0
        MaxPoints: 5000
         Decimate: 0
       Decimation: 2
    LogFramesIndv: 0
         Children: [0x1 double]
```

By default, `LogName` is identical to `SigName`. If you want to use another name for the logged signal, instead of `SigName` itself, change `LogName` to a different string:

```
>> signal_properties.Signals(1).LogName = 'new_name';
```

Then enable the custom string for this signal:

```
>> signal_properties.Signals(1).UseCustomName = 1;
```

5 The value of `LogSignal` is 0 if the signal is not logged, and it equals 1 if the signal is logged. In this case, the signal is not logged.

To enable logging of the signal `A.X.x`, change the value of `LogSignal`:

```
>> signal_properties.Signals(1).LogSignal = 1;
```

6 Reset the chart parameters using the updated `signal_properties` object:

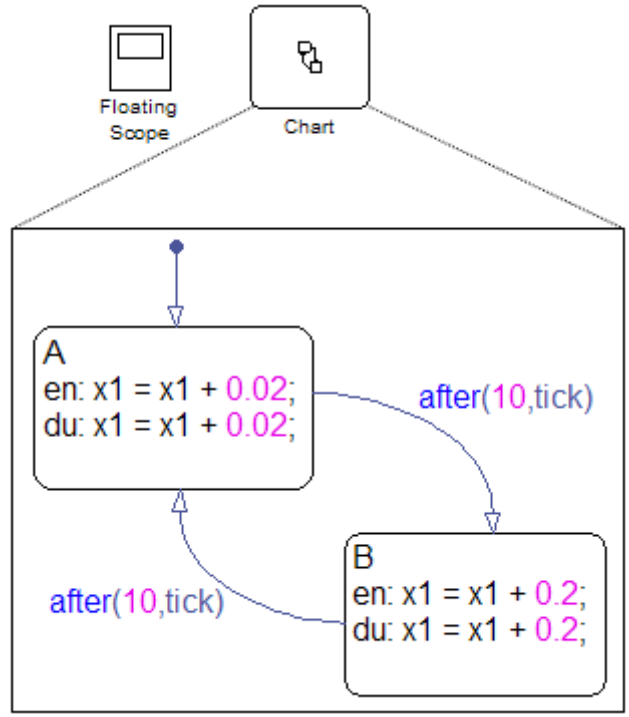
```
>> set_param('myModel/Chart1','AvailSigsInstanceProps', ...
signal_properties)
```

In the Simulink model window, right-click the Stateflow block and select **Log Chart Signals**. In the Signal Logging dialog box, you see that the box next to the signal `A.X.x` is now checked. Also, the custom string `new_name` appears in the **Logging name** field for that signal.

For more information on how you can use and manipulate logged data with MATLAB commands and scripts, see “Logging Signals” in the Simulink software documentation.

Using a Floating Scope to Monitor Data Values and State Activity

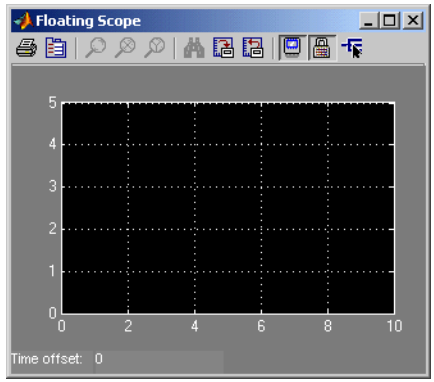
In the steps of this topic, you configure a Floating Scope block to monitor a data value and the activity of a state in this example model:



The model consists of a Floating Scope block and a Stateflow block. The chart for the Stateflow block starts by adding an increment of 0.02 for 10 samples to the data `x1`. For the next 10 samples, `x1` increments by 0.2, and the cycle repeats.

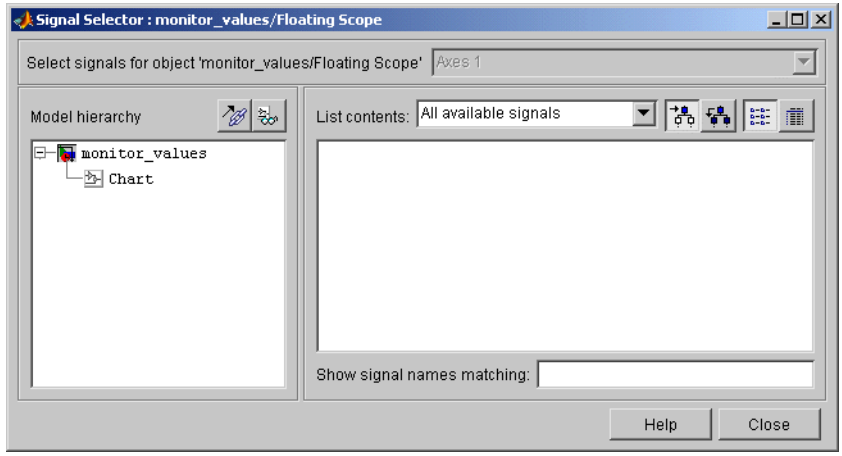
1 Double-click the Floating Scope block.

A Floating Scope window appears, already scaled for this example.

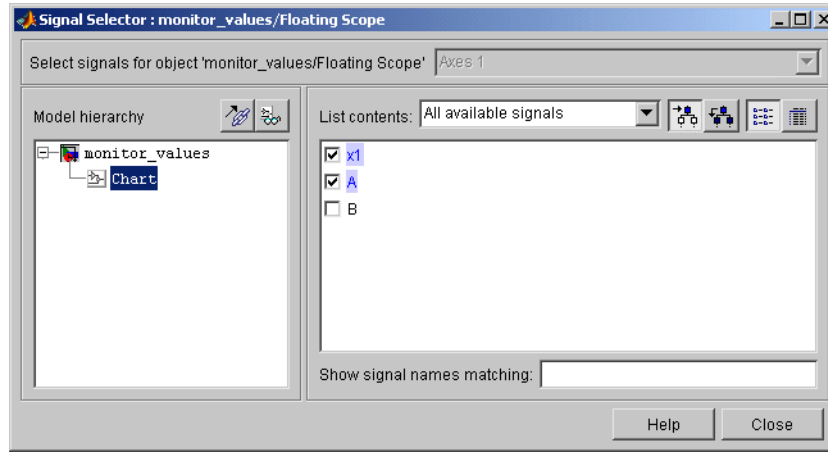


2 In the Floating Scope window, select the Signal Selection tool  .

The Signal Selector dialog box appears with a hierarchy of Simulink blocks for the model.



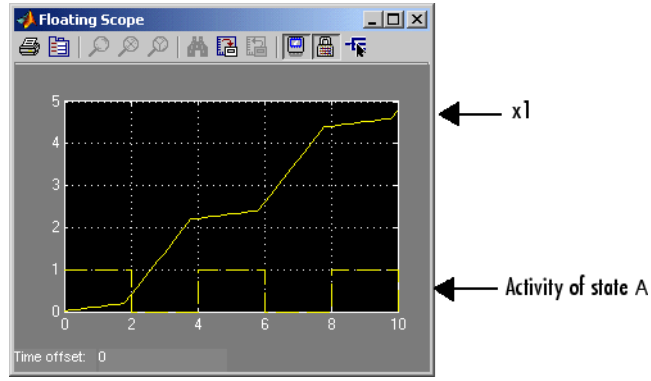
3 In the **Model hierarchy** pane, select the Stateflow block whose signals you want to monitor and, in the **List contents** pane, select the data you want to monitor.



In the preceding example, the block named Chart is selected in the **Model hierarchy** pane, and the data `x1` and the activity of state `A` are selected in the **Contents** pane.

4 Simulate the model.

When you simulate the example model, you receive a signal trace for `x1` and for the activity of state `A`, as shown.



When state A is active, its activity signal value is 1, and when it is inactive, its signal value is 0. Because this value is very low or very high compared to some data, you might want to put it in a second Floating Scope block to compare it with other data.

Understanding Model Coverage for Stateflow Charts

In this section...

About Model Coverage

Model coverage is a measure of how thoroughly a model is tested. The Model Coverage tool helps you to validate your model tests by measuring model coverage for your tests. It determines the extent to which a model test case exercises simulation control flow paths through a model. The percentage of paths that a test case exercises is called its *model coverage*.

You can also use model coverage for:

- Truth tables (see "Model Coverage for Truth Tables" on page 19-56)
- Embedded MATLAB functions (see "Model Coverage for an Embedded MATLAB Function" on page 20-22)

Note The Model Coverage tool requires a license for Simulink Verification and Validation software.

Making Model Coverage Reports

Model Coverage reports are generated during simulation if you specify them (see "Specifying Coverage Report Settings" on page 23-52). For Stateflow charts, the Model Coverage tool records the execution of the chart itself and the execution of its states, transition decisions, and the individual conditions that compose each decision. When simulation is finished, the Model Coverage report tells you how thoroughly a model has been tested, in terms of how many times each exclusive substate is entered, executed, and exited based on the history of the superstate, how many times each transition decision has been evaluated as true or false, and how many times each condition (predicate) has been evaluated as true or false.

Specifying Coverage Report Settings

To specify coverage report settings, select **Tools > Coverage Settings** in a Simulink model window.

By selecting the **Generate HTML Report** option in the Coverage Settings dialog box, you can create an HTML report containing the coverage data generated during simulation of the model. The report appears in the MATLAB Help browser at the end of simulation.

By selecting the **Generate HTML Report** option, you also enable the selection of different coverages that you can specify for your reports. The following sections address only coverage metrics that affect reports for Stateflow charts. These metrics include decision coverage, condition coverage, and MCDC coverage. For a complete discussion of all dialog box fields and entries, see "Specifying Model Coverage Reporting Options" in the Simulink Verification and Validation documentation.

Cyclomatic Complexity

Cyclomatic complexity is a measure of the complexity of a software module based on its edges, nodes, and components within a control-flow graph. It provides an indication of how many times you need to test the module.

The calculation of cyclomatic complexity is as follows:

```
CC = E - N + p
```

where `CC` is the cyclomatic complexity, `E` is the number of edges, `N` is the number of nodes, and `p` is the number of components.

Within the Model Coverage tool, each decision is exactly equivalent to a single control flow node, and each decision outcome is equivalent to a control flow edge. Any additional structure in the control-flow graph is ignored since it contributes the same number of nodes as edges and therefore has no effect on the complexity calculation. This allows cyclomatic complexity to be expressed as follows:

```
CC = OUTCOMES - DECISIONS + p
```

For analysis purposes, each chart is considered to be a single component.

Decision Coverage

Decision coverage interprets a model execution in terms of underlying decisions where behavior or execution must take one outcome from a set of mutually exclusive outcomes.

Note Full coverage for an object of decision means that every decision has had at least one occurrence of each of its possible outcomes.

Decisions belong to an object making the decision based on its contents or properties. The following table lists the decisions recorded for model coverage for the Stateflow objects owning them. The sections that follow the table describe these decisions and their possible outcomes.

Object	Possible Decisions
Chart	If a chart is a triggered Simulink block, it must decide whether or not to execute its block. See “Chart as a Triggered Simulink Block Decision” on page 23-54. If a chart contains exclusive (OR) substates, it must decide which of its states to execute. See “Chart Containing Exclusive OR Substates Decision” on page 23-54.

Object	Possible Decisions
State	If a state is a superstate containing exclusive (OR) substates, it must decide which substate to execute. See "Superstate Containing Exclusive OR Substates Decision" on page 23-54. If a state has `on event name` actions (which might include temporal logic operators), the state must decide whether or not to execute the actions. See "State with On Event_Name Action Statement Decision" on page 23-57.
Transition	If a transition is a conditional transition, it must decide whether or not to exit its active source state or junction and enter another state or junction. See "Conditional Transition Decision" on page 23-57.

Chart as a Triggered Simulink Block Decision

If the chart is a triggered block in a Simulink model, the decision to execute the block is tested. If the block is not triggered, there is no decision to execute the block, and the measurement of decision coverage is not applicable (`NA`).

See "Chart as Subsystem Details Report Section" on page 23-60.

Chart Containing Exclusive OR Substates Decision

If the chart contains exclusive (OR) substates, the decision on which substate to execute is tested. If the chart contains only parallel AND substates, this coverage measurement is not applicable (`NA`).

See "Chart as Superstate Details Report Section" on page 23-60.

Superstate Containing Exclusive OR Substates Decision

Since a chart is hierarchically processed from the top down, procedures such as exclusive (OR) substate entry, exit, and execution are sometimes decided by the parenting superstate.

Note Decision coverage for superstates applies to exclusive (OR) substates only. A superstate makes no decisions for its parallel (AND) substates.

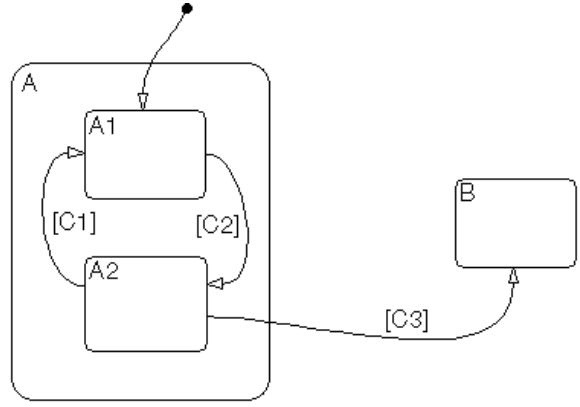
Since a superstate must decide which of its exclusive (OR) substates to process, the number of decision outcomes for the superstate is equal to the number of exclusive (OR) substates that it contains. In the examples following, the choice of which substate to process is made in one of three possible contexts.

Note Implicit transitions are shown as dashed lines in the following examples.

---

1 Active Call

In the following example, states `A` and `A1` are active.



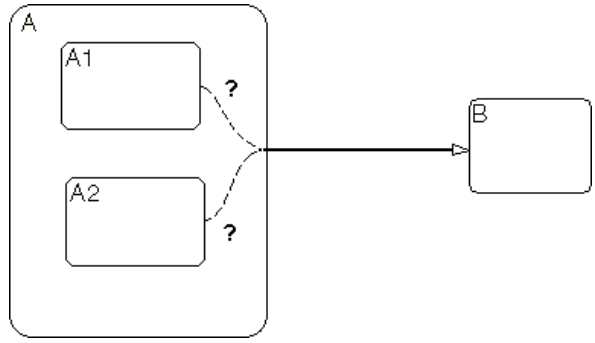
This gives rise to the following superstate/substate decisions:

- The parent of states A and B must decide which of these states to process. This decision belongs to the parent. Since A is active, it is processed.
- State A, the parent of states `A1` and `A2`, must decide which of these states to process. This decision belongs to state A. Since `A1` is active, it is processed.

  During processing of state `A1`, all its outgoing transitions are tested. This decision belongs to the transition and not to its parent state A. In this case, the transition marked by condition `C2` is tested and a decision is made whether to take the transition to `A2` or not. See "Conditional Transition Decision" on page 23-57.

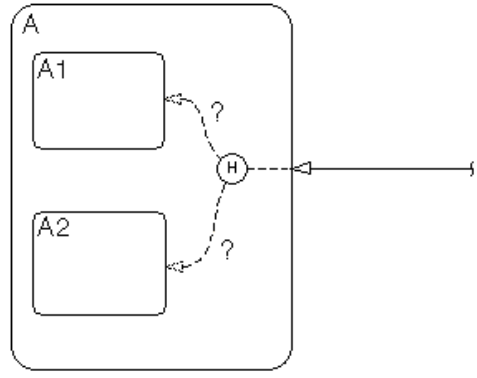
1 Implicit Substate Exit Context

In the following example, a transition takes place whose source is superstate A and whose destination is state B. If the superstate has two exclusive (OR) substates, it is the decision of superstate A as to which of these substates will perform the implicit transition from substate to superstate.



2 Substate Entry with a History Junction

A history junction, similar to the one shown in the example following, provides a superstate with the means of recording which of its substates was last active before the superstate was exited. If that superstate now becomes the destination of one or more transitions, the history junction provides it the means of deciding which previously active substate to enter.



See “State Details Report Section” on page 23-61.

State with On Event_Name Action Statement Decision

A state that has an `on event_name` action statement must decide whether to execute that statement based on the reception of a specified event or on an accumulation of the specified event when using temporal logic operators.

See “State Labels” on page 2-8 and “Using Temporal Logic in State Actions and Transitions” on page 10-56.

Conditional Transition Decision

A conditional transition is a transition with a triggering event and/or a guarding condition (see “Transition Label Notation” on page 2-14). In a conditional transition from one state to another, the decision to exit one state and enter another is credited to the transition itself.

See “Transition Details Report Section” on page 23-63.

Note Only conditional transitions receive decision coverage. Transitions without decisions are not applicable to decision coverage.

Condition Coverage

Condition coverage reports on the extent to which all possible outcomes are achieved for individual subconditions composing a transition decision.

Note Full condition coverage means that all possible outcomes occurred for each subcondition in the test of a decision.

For example, for the decision [A & B & C] on a transition, condition coverage reports on the true and false occurrences of each of the subconditions A, B, and C. This results in six possible outcomes: true and false for each of three subconditions.

See “Transition Details Report Section” on page 23-63.

MCDC Coverage

The Modified Condition Decision Coverage (MCDC) option reports a test's coverage of occurrences in which changing an individual subcondition within a transition results in changing the entire transition trigger expression from true to false or false to true.

Note If matching true and false outcomes occur for each subcondition, coverage is 100%.

For example, if a transition executes on the condition [`C1 & C2 & C3 | C4 & C5`], the MCDC report for that transition shows actual occurrences for each of the five subconditions (`C1, C2, C3, C4, C5`) in which changing its result from true to false is able to change the result of the entire condition from true to false.

See "Transition Details Report Section" on page 23-63.

Coverage Reports for Stateflow Charts

The following sections of a Model Coverage report were generated by simulating the `sf_boiler` demo model, which includes the Stateflow Chart block Bang-Bang Controller. The coverage metrics for **Decision Coverage**, **Condition Coverage**, and **MCDC Coverage** are enabled for this report. The **Look-up Table Coverage** and **Signal Range Coverage** metrics are dependent on the Simulink model and not relevant to the coverage of Stateflow charts.

These subtopics follow:

For information on the model coverage of truth tables, see “Model Coverage for Truth Tables” on page 19-56.

Summary Report Section

Summary

Model Hierarchy/Complexity:		Test 1		
		D1	C1	MCDC
1. sf_boiler	20	89%	71%	43%
2. . . . Bang-Bang Controller	16	95%	71%	43%
3. . . . . SF: Bang-Bang Controller	15	95%	71%	43%
4. . . . . SF: Heater	12	94%	71%	43%
5. . . . . SF: Off	2	100%	75%	50%
6. . . . . SF: On	4	88%	NA	NA
7. . . . . SF: flash_LED	1	100%	NA	NA
8. . . . . SF: turn_boiler	1	100%	NA	NA
9. . . . Boiler Plant model	3	67%	NA	NA
10. . . . . digital thermometer	2	50%	NA	NA
11. . . . . ADC	2	50%	NA	NA

The Summary section shows coverage results for the entire test and appears at the beginning of the Model Coverage report.

Each line in the hierarchy summarizes the coverage results at its level and the levels below it. You can click a hyperlink to a later section in the report with the same assigned hierarchical order number that details that coverage and the coverage of its children.

The top level, sf_boiler, is the Simulink model itself. The second level, Bang-Bang Controller, is the Stateflow chart. The next levels are superstates within the Stateflow chart in order of hierarchical containment. Each superstate uses an SF: prefix. The bottom level, Boiler Plant model, is an additional subsystem in the model.

Chart as Subsystem Details Report Section

2. Subsystem "Bang-Bang Controller"

Parent:	/sf_boiler
Child Systems:	Bang-Bang Controller

Metric	Coverage (this object)	Coverage (inc. descendants)
Cyclomatic Complexity	**1**	**16**
Decision (D1)	NA	95% (21/22) decision outcomes
Condition (C1)	NA	71% (10/14) condition outcomes
MCDC (C1)	NA	43% (3/7) conditions reversed the outcome

The Subsystem report sees the chart as a block in a Simulink model, instead of a chart with states and transitions. If you click the hyperlink of the subsystem name in the title, you see a highlighted Bang-Bang Controller block in the Simulink block diagram.

Chart as Superstate Details Report Section

3. Chart "Bang-Bang Controller"

Parent:	sf_boiler/Bang-Bang Controller
Child Systems:	Heater, flash_LED, turn_boiler

Metric	Coverage (this object)	Coverage (inc. descendants)
Cyclomatic Complexity	1	15
Decision (D1)	100% (2/2) decision outcomes	95% (21/22) decision outcomes
Condition (C1)	NA	71% (10/14) condition outcomes
MCDC (C1)	NA	43% (3/7) conditions reversed the outcome

Decisions analyzed:

Substate executed	100%
State "Off"	1160/1400
State "On"	240/1400

The Chart report sees a Stateflow chart as the superstate container of its states and transitions. If you click the hyperlink of the chart name in the title, you see the chart in the Stateflow Editor.

Cyclomatic complexity and decision coverage appear for the chart and its descendants. Condition coverage and MCDC are not applicable (NA) for a chart, but apply to the descendants.

State Details Report Section

6. State "On"

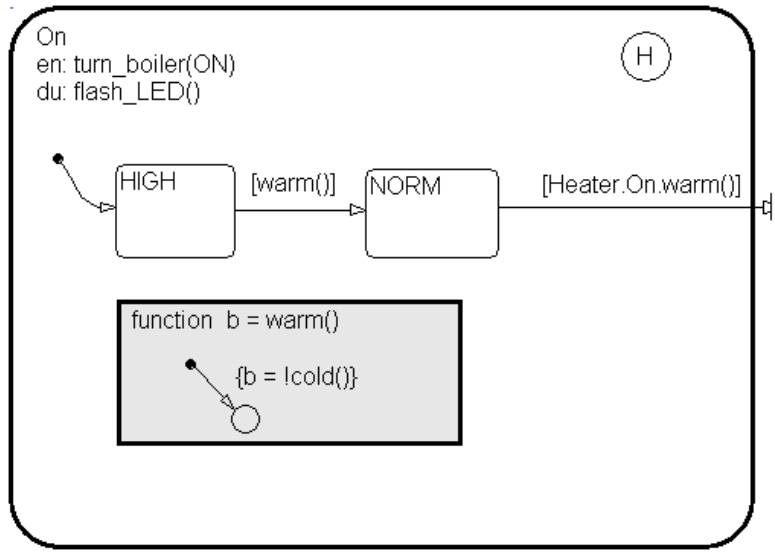
Parent: sf_boiler/Bang-Bang Controller.Heater
Uncovered Links: 

Metric	Coverage (this object)	Coverage (inc. descendants)
Cyclomatic Complexity	3	4
Decision (D1)	83% (5/6) decision outcomes	88% (7/8) decision outcomes

Decisions analyzed:

Substate executed	100%
State "HIGH"	150/233
State "NORM"	83/233
Substate exited when parent exits	50%
State "HIGH"	7/7
State "NORM"	0/7
Previously active substate entered due to history	100%
State "HIGH"	7/28
State "NORM"	21/28

The example state section contains a report on the state On, which appears as follows:



`On` resides in the box `Heater`, which has its own details report (not shown) because it contains other Stateflow objects. However, because `On` is a superstate containing the two states `HIGH` and `NORM` along with a history junction and the function `warm`, `On` has its own numbered report in the Details section.

The decision coverage for the `On` state tests the decision of which substate to execute. The results indicate that five of six possible outcomes were tested during simulation. The decisions include:

1. The choice of which substate to execute when `On` is executed
2. The choice of which state to exit when `On` is exited
3. The choice of which substate to enter when `On` is entered and the History junction has a record of the previously active substate

Because each decision above can result in processing either HIGH or NORM, the total possible outcomes are 3 x 2 = 6.

The decision coverage tables display the number of occurrences for each decision and the number of times each state was chosen. For example, the

first decision was made 233 times. Of these, the HIGH state was executed 150 times and the NORM state was executed 83 times.

Cyclomatic complexity and decision coverage also apply to descendants of the `On` state. The decision required by the condition `[warm()]` for the transition from HIGH to NORM brings the total possible decision outcomes to 8. Condition coverage and MCDC are not applicable (`NA`) for a state.

Note Nodes and edges that make up the cyclomatic complexity calculation have no direct relationship with model objects (states, transitions, and so on). Instead, this calculation requires a graph representation of the equivalent control flow.

Transition Details Report Section

Reports for transitions appear under the report sections of their owning objects. Transitions do not appear in the model hierarchy of the Summary section, since the hierarchy is based on superstates owning other Stateflow objects.

Transition "after(40,sec) [cold()]" from "Off" to "On"

Parent: sf_boiler/Bang-Bang Controller.Heater
Uncovered Links:

Metric	Coverage
Cyclomatic Complexity	3
Decision (D1)	100% (2/2) decision outcomes
Condition (C1)	67% (4/6) condition outcomes
MCDC (C1)	33% (1/3) conditions reversed the outcome

Decisions analyzed:

Transition trigger expression	100%
false	1131/1160
true	29/1160

Conditions analyzed:

Description:	True	False
Condition 1, "sec"	1160	0
Condition 2, "after(40,sec)"	29	1131
Condition 3, "cold()"	29	0

MC/DC analysis (combinations in parentheses did not occur)

Decision/Condition:	True Out	False Out
Transition trigger expression		
Condition 1, "sec"	TTT	(Fxx)
Condition 2, "after(40,sec)"	TTT	TFx
Condition 3, "cold()"	TTT	(TTF)

The decision for this transition depends on the time delay of 40 seconds and the condition `[cold()]`. If, after a 40 second delay, the environment is cold (`cold() = 1`), the decision to execute this transition and turn the Heater on is made. For other time intervals or environment conditions, the decision is made not to execute.

For decision coverage, both true and false outcomes occurred. Because two of two decision outcomes occurred, coverage was full or 100%.

Condition coverage shows that only 4 of 6 condition outcomes were tested. The temporal logic statement `after(40,sec)` represents two conditions:

the occurrence of `sec` and the time delay `after(40,sec)`. Therefore, three conditions on the transition exist: `sec`, `after(40,sec)`, and `cold()`. Since each of these decisions can be true or false, six possible condition outcomes exist.

The **Conditions analyzed** table shows each condition as a row with the recorded number of occurrences for each outcome (true or false). Decision rows in which a possible outcome did not occur are shaded. For example, the first and the third rows did not record an occurrence of a false outcome.

In the MC/DC report, all sets of occurrences of the transition conditions are scanned for a particular pair of decisions for each condition in which the following are true:

- The condition varies from true to false.
- All other conditions contributing to the decision outcome remain constant.
- The outcome of the decision varies from true to false, or the reverse.

For three conditions related by an implied AND operator, these criteria can be satisfied by the occurrence of these conditions.

Condition Tested	True Outcome	False Outcome
1	**T**TT	**F**xx
2	T**T**T	T**F**x
3	TT**T**	TT**F**

Notice that in each line, the condition tested changes from true to false while the other condition remains constant. Irrelevant contributors are coded with an "x" (discussed below). If both outcomes occur during testing, coverage is complete (100%) for the condition tested.

The preceding report example shows coverage only for condition 2. The false outcomes required for conditions 1 and 3 did not occur, and are indicated by parentheses for both conditions. Therefore, condition rows 1 and 3 are shaded. While condition 2 has been tested, conditions 1 and 3 have not and MCDC is 33%.

For some decisions, the values of some conditions are irrelevant under certain circumstances. For example, in the decision [`C1 & C2 & C3 | C4 & C5`] the left side of the `|` is false if any one of the conditions `C1`, `C2`, or `C3` is false. The same applies to the right side result if either `C4` or `C5` is false. When searching for matching pairs that change the outcome of the decision by changing one condition, holding some of the remaining conditions constant is irrelevant. In these cases, the MC/DC report marks these conditions with an "x" to indicate their irrelevance as a contributor to the result. These conditions appear as shown.

Transition "[c1&c2&c3 | c4&c5]" . . .

MC/DC analysis (combinations in parentheses did not occur)

Decision/Condition:	#1 True Out	#1 False Out
Transition trigger expression		
Condition 1, "c1"	TTTxx	FxxFx
Condition 2, "c2"	TTTxx	TFxFx
Condition 3, "c3"	TTTxx	TTFFx
Condition 4, "c4"	FxxTT	FxxFx
Condition 5, "c5"	FxxTT	FxxTF

Consider the first matched pair. Since condition 1 is true in the **True** outcome column, it must be false in the matching **False** outcome column. This makes the conditions `C2` and `C3` irrelevant for the false outcome since `C1 & C2 & C3` is always false if `C1` is false. Also, since the false outcome is required to evaluate to false, the evaluation of `C4 & C5` must also be false. In this case, a match was found with `C4` = F, making condition `C5` irrelevant.

Colored Stateflow Chart Coverage Display

The Model Coverage tool displays model coverage results for individual blocks directly in Simulink diagrams. If you enable this feature, the Model Coverage tool does the following:

- Highlights (colors) Stateflow objects that have received model coverage during simulation

- Provides a context-sensitive display of summary model coverage information for each object

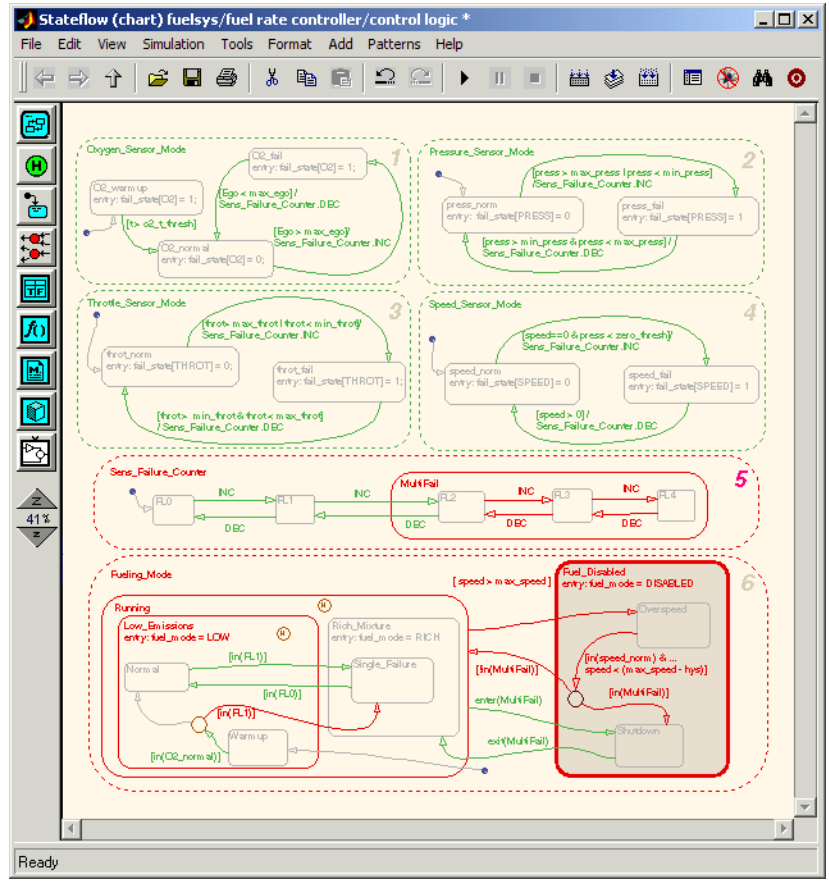
---

Caution The coverage tool changes colors only for open Stateflow charts at the time coverage information is reported. When you interact with the Stateflow chart, such as selecting a transition or a state, colors revert to their default values.

For details on enabling and selecting this feature in the Simulink window, see "Enabling the Colored Diagram Display" in the Simulink Verification and Validation documentation.

Displaying Model Coverage with Model Coloring

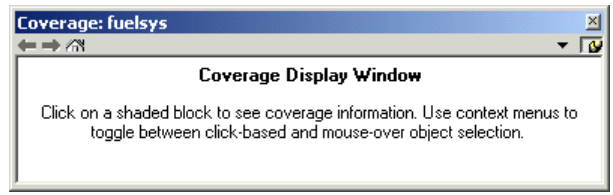
Once you enable display coverage with model coloring, anytime that the model generates a model coverage report, individual Stateflow objects receiving coverage are highlighted with light green or light red.



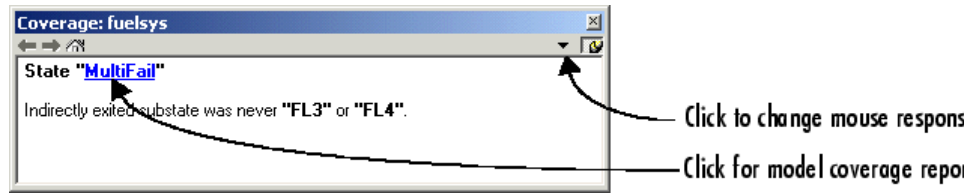
Objects highlighted in light green received full coverage during testing. Objects highlighted in light red received incomplete coverage. Objects with no color highlighting receive no coverage at all.

Note To revert the Stateflow chart to show original colors, select and deselect its objects.

Along with the highlighted Stateflow chart, a Coverage Display Window appears, as shown.



If you click a highlighted Stateflow object, its summarized coverage appears in the Coverage Display Window. In the preceding example, the following summary report appears when you click the `MultiFail` state:



Summary coverage information appears in the Coverage Display Window for the Stateflow object, whose hyperlinked name appears at the top of the window. Click the hyperlink to access the appropriate section of the coverage report for this object.

You can set the Coverage Display Window to appear for a block in response to a hovering mouse cursor instead of a mouse click in one of two ways:

- Select the downward arrow on the right side of the Coverage Display Window, and, from the resulting menu, select **Focus**.
- Right-click a colored block and select **Coverage display on mouse-over** from the resulting context menu.

24

Exploring and Modifying Charts

Using the Model Explorer with Stateflow Objects

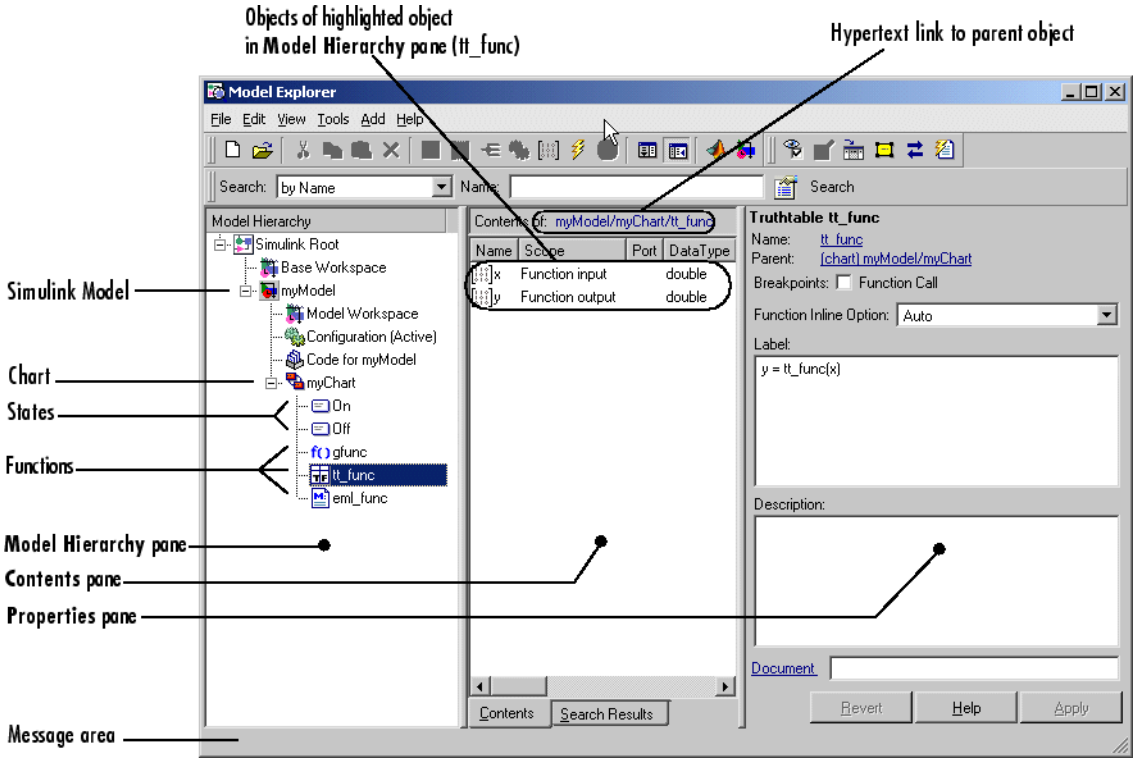
In this section...

Viewing Stateflow Objects in the Model Explorer

Depending on what you edit, you can use one of these methods for opening the Model Explorer:

- From the toolbar menu of the Stateflow Editor, Truth Table Editor, or Embedded MATLAB Editor, click the **Explore** icon  .
- In the Stateflow Editor or the Truth Table Editor, select **Tools > Explore**.
- In the Stateflow Editor, select **View > Model Explorer**.
- Right-click an empty area in the Stateflow chart. From the resulting pop-up menu, select **Explore**.

The Model Explorer appears similar to the following:



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 (chart, truth table, or Embedded MATLAB function) is highlighted in the **Model Hierarchy** pane and its objects are displayed in the **Contents** pane. In the preceding example, the Model Explorer was opened from the Truth Table Editor for the truth table `tt_func` in the Stateflow chart `myChart`.

The **Model Hierarchy** pane displays the elements of all loaded Simulink models, which includes Stateflow charts, and their states, boxes, and functions. 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. For convenience, a hypertext link to the currently selected object in the **Model Hierarchy** pane is included following 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, selecting the link

```
(Stateflow.TruthTable) myModel/myChart/myChart/tt_func
```

displays the truth table `tt_func` in the Truth Table Editor.

By default, the Model Explorer displays event and data child objects in the **Contents** pane for the selected object in the **Model Hierarchy** pane. To display additional or different child Stateflow objects in the **Contents** pane, do the following:

1. From the Model Explorer, select **View** > **List View Options**.
2. In the submenu, select any or all of the following individual options: **States/Functions/Boxes/Etc.**, **Transitions**, **Junctions**, **Events**, or **Data**.

   To display all of the preceding Stateflow child objects, select **All Stateflow Objects**.

Each type of object, whether in the **Object Hierarchy** or **Contents** pane, appears with an adjacent icon. Objects that are subcharted (states, boxes, and graphical functions) have their appearance altered by shading.

Object	Icon	Icon for Subcharted Object
Chart		Not applicable
State		
Box		
Graphical function		
Truth table function		Not applicable

Object	Icon	Icon for Subcharted Object
Embedded MATLAB function		Not applicable
Simulink function		Not applicable
Data		Not applicable
Event		Not applicable
Target		Not applicable

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 "Setting Properties for Stateflow Objects in the Model Explorer" on page 24-9 for more details.

Editing States or Charts in the Model Explorer

To edit a state or chart displayed in the **Object Hierarchy** pane of the Model Explorer, do the following:

1. Right-click the object.
2. Select **Edit** from the resulting menu.

The selected object appears highlighted in the Stateflow Editor in the context of its parent.

Adding Data and Events in the Model Explorer

State, box, and function Stateflow objects can parent data and events. You can also add data and events to the Simulink model to make them globally available to all Stateflow objects in the model.

To add a data or an event to a Stateflow object or to the Simulink model, do the following:

1. In the **Model Hierarchy** pane of the Model Explorer, select a Simulink model or a Stateflow object.
2. From the **Add** menu, select **Data** or **Event**.

   A data or event is added to the Model Explorer **Contents** pane with the default name `data` or `event`. If you continue adding more data, each new data or event is named with an integer suffix (`data1`, `event1`, `data2`, `event2`, and so on).

   You can change the displayed properties for a data or event directly in the Model Explorer. You can also access the complete list of properties for a data or event from the Model Explorer. See "Setting Properties for Stateflow Objects in the Model Explorer" on page 24-9.

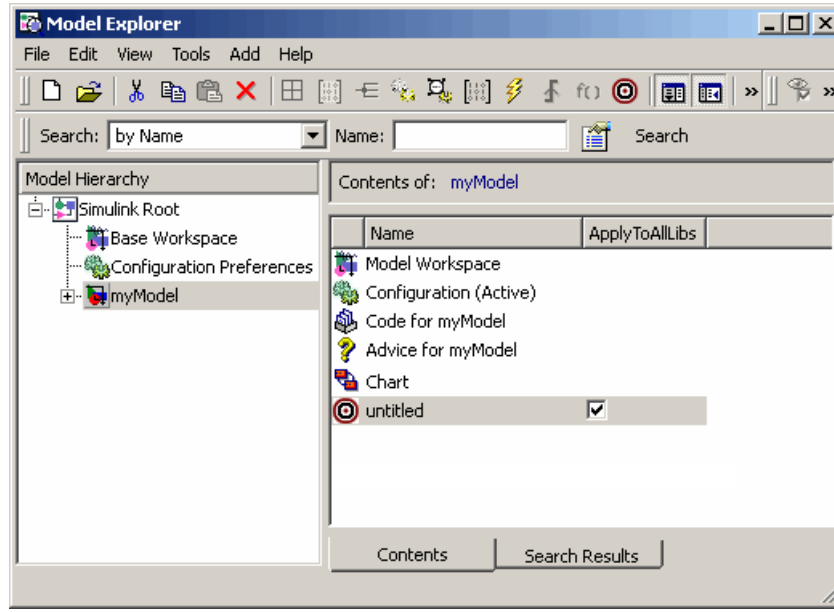
For more detailed examples of creating data and events in the Model Explorer, see "Adding Events Using the Model Explorer" on page 9-5 and "Adding Data Using the Model Explorer" on page 8-3.

Adding Custom Targets in the Model Explorer

Custom targets are parented exclusively by a Simulink model. In the Model Explorer, you can add custom targets to a model as follows:

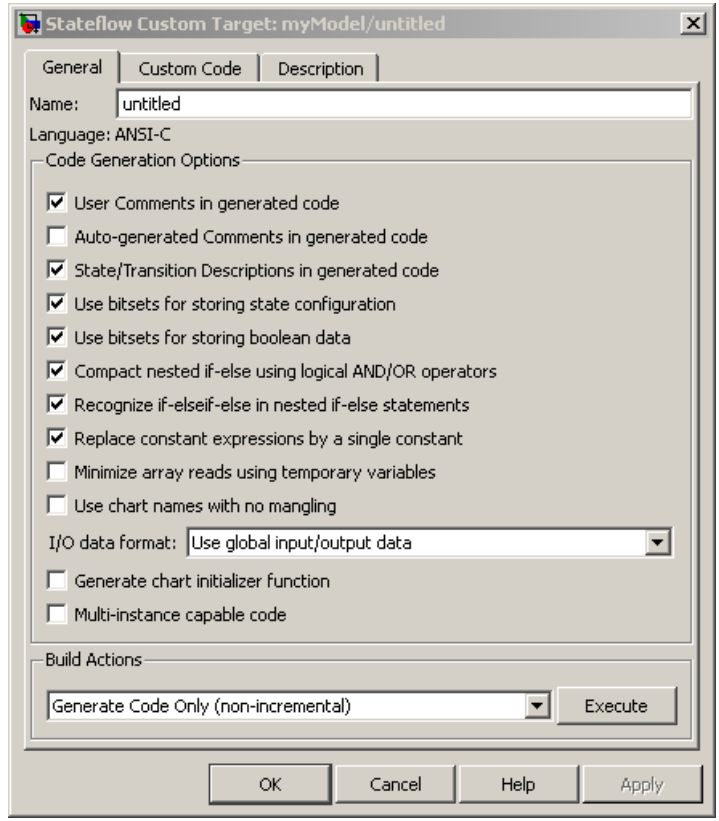
1. In the Model Explorer, in the left **Model Hierarchy** pane, select the Simulink model to receive the custom target.
2. In the Model Explorer, select **Add > Stateflow Target**.

The **Contents** pane of the Model Explorer displays the new custom target with the default name `untitled`.



3 In the **Contents** pane, right-click the row of the custom target and select **Properties** from the context menu.

The Stateflow Custom Target dialog box appears.



4 Enter the name of the custom target.

   You can use any string except the reserved names `sfun` and `rtw`.

5 Specify other properties in the panes.

6 Click **Apply**.

For more information, see "How to Build a Stateflow Custom Target" on page 22-50.

Renaming Objects in the Model Explorer

Follow these steps to rename a state, box, function, data, event, or target objects in the Model Explorer:

1. Right-click the object row in the **Contents** pane of the Model Explorer.

   A pop-up menu appears.

2. From the resulting pop-up menu, select **Rename**.

   The name of the selected object appears in a text edit box that overlays the Name property for the object row.

3. Change the name of the target in the edit box and click outside the edit box.

You can also change the name of an object in the Model Explorer by changing the value of its Name property. See "Setting Properties for Stateflow Objects in the Model Explorer" on page 24-9 for details.

Setting Properties for Stateflow Objects in the Model Explorer

To change one of the displayed properties of a displayed object in the **Contents** pane of the Model Explorer:

1. In the **Contents** pane, click anywhere in the row of the displayed object.

   This highlights the row.

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 **Return** 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.
2. Select **Properties** from the resulting menu.

   The properties dialog box for the object appears.
3. Edit the appropriate properties and select **Apply** or **OK** to apply the changes.

To display the property dialog box dynamically for the selected object in the **Model Hierarchy** or **Contents** pane of the Model Explorer:

1. Select **View** > **Dialog View**.

   The property dialog box for the selected object appears in the far right pane of the Model Explorer.

Moving and Copying Data, Events, and Targets 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** is automatically changed to **Local**.

You can move data, event, or target objects to another parent by doing the following:

1. Select the data, event, or target 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.

You can accomplish the same outcome by cutting or copying the selected events, data, and targets as follows:

1. Select the event, data, and targets to move in the **Contents** pane of the Model Explorer.

2. In the Model Explorer, select **Edit > Cut** or **Copy**.

   If you select **Cut**, the selected items are deleted and are 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 resulting 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 machine, chart, or state 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 of the Model Explorer and selecting **Paste** from the resulting menu. The Model Explorer also uses the keyboard equivalent of **Ctrl+V** (Paste) on a computer running the UNIX or Windows operating system.

Changing 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.

Deleting Data, Events, and Targets in the Model Explorer

Delete event, data, and target objects in the **Contents** pane of the Model Explorer as follows:

1 Select the object.

2 Press the **Delete** key.

You can also select **Cut** from the **Edit** menu or **Ctrl+X** from the keyboard to delete an object.

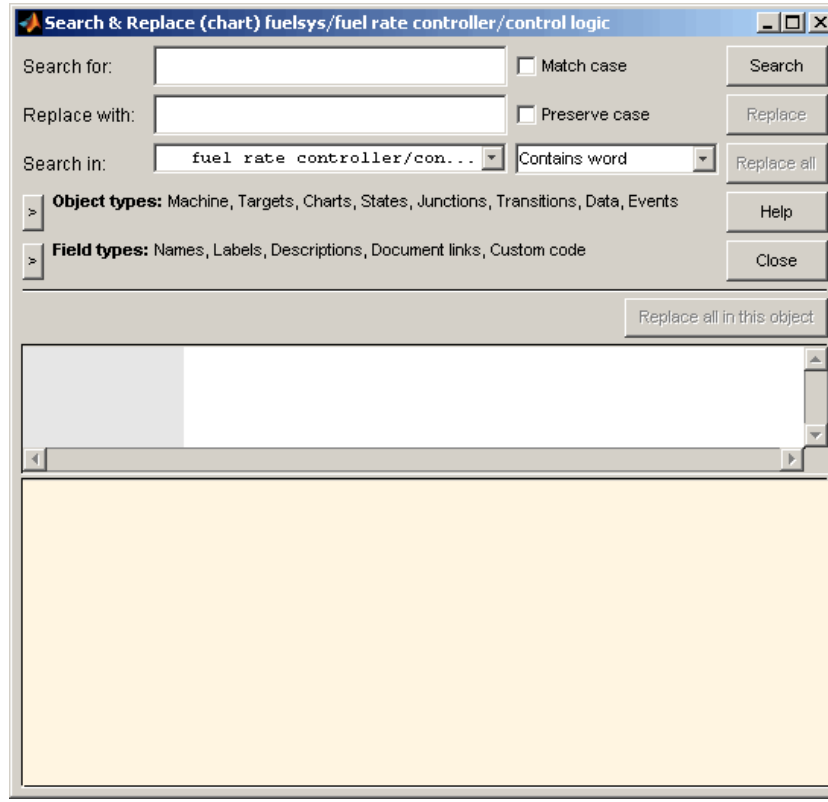
Using the Stateflow Search & Replace Tool

In this section...

Opening the Search & Replace Tool

To display the Search & Replace dialog box, do the following:

1. Open a Stateflow chart in the Stateflow Editor.
2. Select **Tools > Search & Replace**.



The window name for the Search & Replace dialog box contains a full path expression for the current Stateflow chart or machine in this form.

```
(object) Machine/Subsystem/Chart
```

The Search & Replace dialog box contains these 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. See "Match case (Case Sensitive)" on page 24-16.

- **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 "Using the Replace Buttons" on page 24-25.

- **Preserve case**

  This option modifies replacement text. For an understanding of this option, see "Replacing with Case Preservation" on page 24-24.

- **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 "Using Different Search Types" on page 24-16.

- **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. By default, these boxes are hidden; only current selections are displayed next to their titles.

  Select the right-facing arrow button  in front of the title to expand a selection box and make changes.

  Select the same button (this time with a left-facing arrow)  to compress the selection box to display the settings only, or, if you want, just leave the box expanded.

- **Search** and **Replace** buttons

  These are described in "Using the Search Button and View Area" on page 24-20 and "Using the Replace Buttons" on page 24-25.

- **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 24-21.

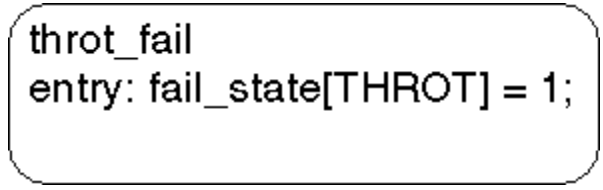
Using Different Search Types

Enter search pattern text in the **Search for** text box. You can use one of the following settings in the **Match options** field (unlabeled and just to the right of the **Search in** field) to further refine the meaning of the text entered.

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 you have a state with this label and entry action.



Searching for the string `fail` with the **Contains word** option finds two occurrences of the string `fail`.

Match case (Case Sensitive)

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 strings with the same sequence of base characters in lower- or uppercase. For example, the search string `"AnDrEw"` finds the matching text `"andrew"` or `"Andrew"` or `"ANDREW"`.

Match whole word

Select this option to specify that the search pattern text 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 the string `fail` finds no text within that state. However, searching for the string `"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 a string composed of letters, numbers, and special symbols that defines one or more string 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 the string `"fail_"` matches the `"fail_"` string that is part of the second line, character for character. Searching with the regular expression `"\w*_"` also finds the string `"fail_"`. This search string 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” in the MATLAB software documentation.

Searching with Regular Expression Tokens

Within a regular expression, you use parentheses to group characters or expressions. For example, the regular expression `"and(y|rew)"` matches the text `"andy"` or `"andrew"`. Parentheses also have the side effect of remembering what they match so that you can recall and reuse the found text with a special variable in the **Search for** field. These variables are called *tokens*.

For details on how to use tokens in the Search & Replace tool, see "Tokens" in the MATLAB software documentation.

You can also use tokens in the **Replace with** field. See "Replacing with Tokens" on page 24-24 for a description of using regular expression tokens for replacing.

Preserve case

This option modifies replacement text and not search text. For details, see "Replacing with Case Preservation" on page 24-24.

Specifying 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 Stateflow chart for searching in the **Search in** field. By default, the current Stateflow chart in which you entered 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 to text matches in the selected object types by following these steps:

1 Expand the **Object types** field.

2 Select one or more object types.

Field Types

Limit your search to text matches for the specified fields by following these steps:

1 Expand the **Field types** field.

2 Select 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.

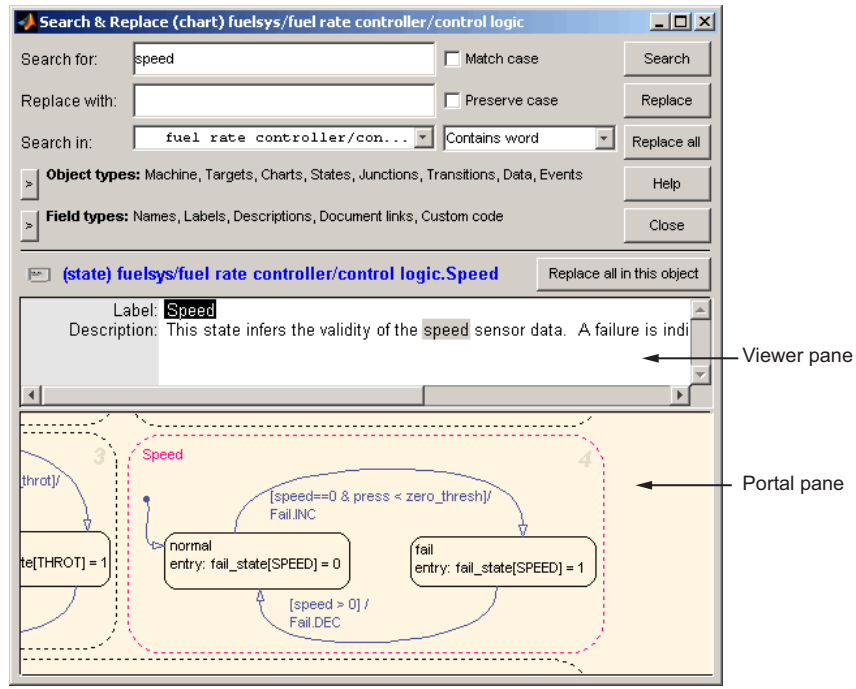
Document links. All objects have searchable **Link** fields.

Custom code. Only target objects contain custom code.

Using 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 are displayed 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 fuelsys demo model, a search for the word `"speed"` finds the **Description** field for the state `Speed`. The resulting view area display 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 “Viewing Stateflow Objects in the Model Explorer” on page 24-2.

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.

To display the object, click the mouse once on the full path name of the object. If the object is a graphical member of a Stateflow chart, it appears in the Stateflow Editor. Otherwise, it appears as a member of its Stateflow chart in the Model Explorer.

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.

Additional Display Options

Right-click anywhere in the Search & Replace dialog box to display a menu with these selections.

Selection	Result
Show portal	A toggle switch that hides or displays the portal.
Edit	Displays the object with matching text in the Stateflow Editor. Applies to states, junctions, transitions, and charts.
Explore	Displays the object with matching text in the Model Explorer. Applies to states, data, events, machines, charts, and targets.
Properties	Displays the properties dialog box for the object with matching text.

Note The **Edit**, **Explore**, and **Properties** selections are available only after a successful search.

If the portal is not visible, you can select the **Show portal** option to display it. You can also click and drag the border between the viewer and the portal (the cursor turns to a vertical double arrow), which resides just above the bottom boundary of the Search & Replace dialog box. Moving this border allows you to exchange area between the portal and the viewer. If you click and drag the border with the left mouse button, the graphic display resizes after you reposition the border. If you click and drag the border with the right mouse button, the graphic display continuously resizes as you move the border.

Specifying 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 choose one of the dynamic replacement options described below.

Replacing with Case Preservation

If you choose the **Case Preservation** 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

  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."`.

Replacing with Tokens

Within a regular expression, you use parentheses to group characters or expressions. For example, the regular expression `"and(y|rew)"` matches the text `"andy"` or `"andrew"`. Parentheses also have the side effect of remembering what they matched so that you can recall and reuse the matching text with a special variable in the **Replace with** field. These variables are called *tokens*.

Tokens outside the search pattern have the form `$1,$2,...,$n (n<17)` and are assigned left to right from parenthetical expressions in the search string.

For example, the search pattern `"(\w*)_(\w*)"` finds all word expressions with a single underscore separating the left and right sides of the word. If you specify an accompanying replacement string of `"$2_$1"`, you can replace all these expressions by their reverse expression with a single **Replace all**. For example, the expression `"Bill_Jones"` is replaced by `"Jones_Bill"`, and the expression `"fuel_system"` is replaced by `"system_fuel"`.

For details on how to use tokens in regular expression search patterns, see "Regular Expressions" in the MATLAB software documentation.

Using the 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 string in the **Search for** field is replaced by the text string you entered in the **Replace with** field. The Search & Replace tool then searches for the next occurrence of the **Search for** text string.

Replace All

When you select the **Replace All** button, all instances of text matching the **Search for** field are replaced by the text string 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.

If the search scope is set to **Search Whole Machine**, then after finishing the current Stateflow chart, replacement continues to the completion of all other charts in your Simulink model.

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 a search string 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. For example, a search on **Custom Code** fields without selecting target objects is invalid.

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.

Finding Stateflow Objects

In this section...

Types of Finder Tools

Two types of finder tools can search for Stateflow objects.

- On most platforms, when you select **Tools > Find** in the Stateflow Editor, the Simulink Find dialog box appears. You can use this tool to search for Simulink and Stateflow objects that meet criteria you specify. Any objects that meet your criteria appear in the search results pane of the dialog box.

  For details, see "The Finder" in the Simulink software documentation.

- On platforms that do not support the Simulink Find tool, the original Stateflow Finder appears when you select **Tools > Find** in the Stateflow Editor.

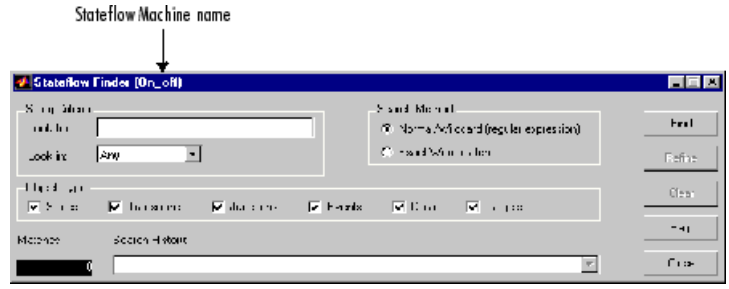
Note See the Simulink Release Notes in the online documentation for a list of platforms on which the Simulink Find tool is not available.

Opening Stateflow Finder

On platforms that do not support the Simulink Find tool, access the Stateflow Finder dialog box with one of these methods:

- Select **Tools > Find** in the Stateflow Editor.
- Select **Edit > Find** in the Simulink model window.

The Finder operates on the machine whose name appears in the window title area of the Finder dialog box.



Using Stateflow Finder

String Criteria

You specify the string by entering the text to search for in the **Look for** text box. The search is case sensitive. All text fields are included in the search by default. Alternatively, you can search in specific text fields by using the **Look in** list box to choose one of these options:

Any. Search the state and transition labels, object names, and descriptions of the specified object types for the string specified in the **Look for** field.

Label. Search the state and transition labels of the specified object types for the string specified in the **Look for** field.

Name. Search the **Name** fields of the specified object types for the string specified in the **Look for** field.

Description. Search the **Description** fields of the specified object types for the string specified in the **Look for** field.

Document Link. Search the **Document** link fields of the specified object types for the string specified in the **Look for** field.

Custom Code. Search custom code for the string specified in the **Look for** field.

Search Method

By default the **Search Method** is **Normal/Wildcard** (regular expression). Alternatively, you can click the **Exact Word match** option if you are searching for a particular sequence of one or more words.

A regular expression is a string composed of letters, numbers, and special symbols that define one or more strings. 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 \ precedes it.

Special characters supported by Stateflow Finder are as follows.

Character	Description
^	Start of string
$	End of string
.	Any character
\	Quote the next character
*	Match zero or more
+	Match one or more
[ ]	Set of characters

Object Type

Specify the object types to search by toggling the check boxes. A check mark indicates that the object is included in the search criteria. By default, all object types are included in the search criteria. **Object Types** include:

- States
- Transitions
- Junctions
- Events
- Data
- Targets

Find Button

Click the **Find** button to initiate the search operation. The results appear in the display area.

Matches

The **Matches** field displays the number of objects that match the specified search criteria.

Refine Button

After the results of a search appear, enter additional search criteria and click **Refine** to narrow the previously entered search criteria. An ampersand (&) is prefixed to the search criteria in the **Search History** field to indicate a logical AND with any previously specified search criteria.

Search History

The **Search History** text box displays the current search criteria. Click the pull-down list to display search refinements. An ampersand is prefixed to the search criteria to indicate a logical AND with any previously specified search criteria. You can undo a previously specified search refinement by selecting a previous entry in the search history. By changing the **Search History** selection, you force the Finder to use the specified criteria as the current, most refined, search output.

Clear Button

Click **Clear** to clear any previously specified search criteria. By doing so, you remove the results and reset the search criteria to the default settings.

Close Button

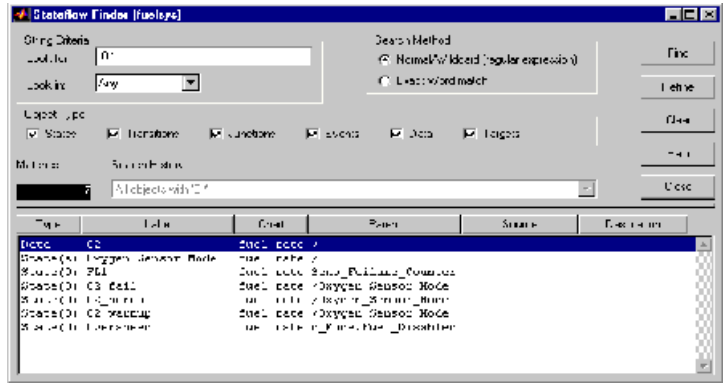
Click **Close** to close the Finder.

Help Button

Click **Help** to display the Stateflow software documentation in an HTML browser window.

Finder Display Area

The Stateflow Finder display area appears similar to this.



The display area shows matching entries with these columns:

Field	Description
Type	The object type appears in this field. States with exclusive (OR) decomposition are followed by an (O). States with parallel (AND) decomposition are followed by (A).
Label	The string label of the object appears in this field.
Chart	The title of the Stateflow chart appears in this field.

Field	Description
Parent	The parent of this object in the hierarchy.
Source	Source object of a transition.
Destination	Destination object of a transition.

All fields are truncated to maintain column widths. The **Parent**, **Source**, and **Destination** fields are truncated from the left so that the name at the end of the hierarchy is readable. The entire field contents, including the truncated portion, are used for resorting.

Each field label is also a button. Click the button to have the list sorted based on that field. If the same button is pressed twice in a row, the sort ordering is reversed.

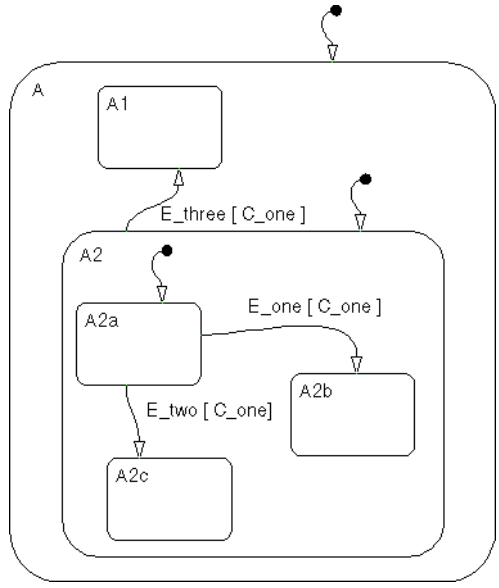
You can resize the Finder vertically to display more output rows, but you cannot expand it horizontally.

Click a graphical entry to highlight that object in the Stateflow Editor. Double-click an entry to invoke the Properties dialog box for that object. Right-click the entry to display a menu that allows you to explore, edit, or display the properties of that entry.

Representing Hierarchy

The Stateflow Finder shows **Parent**, **Source**, and **Destination** fields to represent the hierarchy. The Stateflow chart is the root of the hierarchy and is represented by the / character. Each level in the hierarchy is delimited by a period `(.)` character. The **Source** and **Destination** fields use the combination of the tilde (~) and the period `(.)` characters to denote that the state listed is relative to the **Parent** hierarchy.

Using the following Stateflow chart as an example, what are the values for the **Parent**, **Source**, and **Destination** fields for the transition from `A2a` to `A2b`?



The A2a to A2b transition is within state A2. The parent of state A2 is state A, and the parent of state A is the Stateflow chart itself. The notation for the parent of state A2a is /A.A2. State A2a is the transition source and state A2b is the destination. These states are at the same level in the hierarchy. The relative hierarchy notation for the source of the transition is ~.A2a. The full path is /A.A2.A2a. The relative hierarchy notation for the destination of the transition is ~.A2b. The full path is /A.A2.A2b.

A

Semantic Rules Summary

Entering a Chart

The set of default flow paths is executed (see “Executing a Set of Flow Graphs” on page A-3). If this action does not cause a state entry and the chart has parallel decomposition, then each parallel state is entered (see “Entering a State” on page A-2).

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

Executing an Active Chart

If the chart has no states, each execution is equivalent to initializing a chart. Otherwise, the active children are executed. Parallel states are executed in the same order that they are entered.

Entering a State

1. If the parent of the state is not active, perform steps 1-4 for the parent.
2. If this is a parallel state, check that all siblings with a higher (i.e., earlier) entry order are active. If not, perform all entry steps for these states first.
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 parallel decomposition, i.e., has children that are parallel states, perform entry steps 1-5 for each state according to its entry order.

6 If this is a parallel state, perform all entry actions 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.

Executing an Active State

1 The set of outer flow graphs is executed (see "Executing a Set of Flow Graphs" on page A-3). If this causes a state transition, execution stops. (Note that this step is never required for parallel states.)

2 During actions and valid on-event actions are performed.

3 The set of inner flow graphs is executed. If this does not cause a state transition, the active children are executed, starting at step 1. Parallel states are executed in the same order that they are entered.

Exiting an Active State

1 If this is a parallel state, make sure that all sibling states that were entered after this state have already been exited. 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 they were entered.

3 Perform any exit actions.

4 Mark the state as inactive.

Executing a Set of Flow Graphs

Flow graphs are executed by starting at step 1 below with a set of starting transitions. The starting transitions for inner flow graphs are all transition segments that originate on the respective state and reside entirely within that state. The starting transitions for outer flow graphs are all transition segments that originate on the respective state but reside at least partially outside that state. The starting transitions for default flow graphs are all default transition segments that have starting points with the same parent:

1. A set of transition segments is ordered.

2. While there are remaining segments to test, a segment is tested for validity. If the segment is invalid, move to the next segment in order. If the segment is valid, execution depends on the destination:

   States

   a. No more transition segments are tested and a transition path is formed 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 are exited (see "Exiting an Active State" on page A-3).

   c. The transition action from the final transition segment is executed.

   d. The destination state is entered (see "Entering a State" on page A-2).

   Junctions with no outgoing transition segments

   Testing stops without any states being exited or entered.

   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, back up the incoming transition segment that brought you to the junction and continue at step 2, starting with the next transition segment after the back up segment. The set of flow graphs is done executing when all starting transitions have been tested.

Executing 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 is executed (see "Executing an Active Chart" on page A-2 and "Executing an Active State" on page A-3). (The event *receiver* is the parent of the event unless the event was explicitly directed to a *receiver* using the `send()` function.)

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

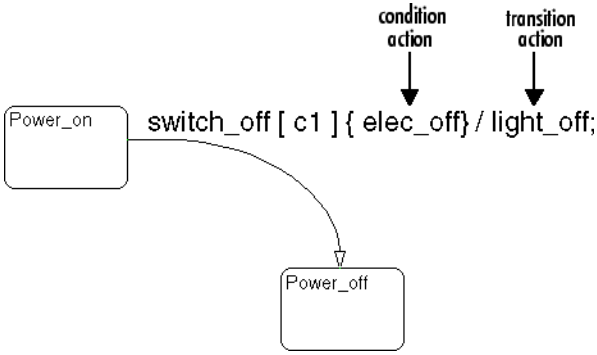
2 After broadcasting the event, the broadcaster performs early return logic based on the type of action statement that caused the event.

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 are not performed.
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 are not performed.
State During	If the state is no longer active at the end of the event broadcast, any remaining steps in executing an active state are not performed.
Condition	If the origin state of the inner or outer flow graph or parent state of the default flow graph is no longer active at the end of the event broadcast, the remaining steps in the execution of the set of flow graphs are not performed.
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 are not performed.

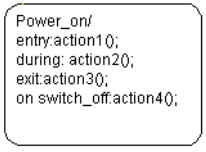
Glossary

actions

Actions take place as part of Stateflow chart execution. The action can be executed as part of a transition from one state to another, or depending on the activity status of a state. Transitions can have condition actions and transition actions as shown.



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 as shown.



An action can be a function call, a broadcast event, a variable assignment, and so on. For more information on actions and action language, see Chapter 10, "Using Actions in Stateflow Charts".

API (application programming interface)

Format you can use to access and communicate with an application program from a programming or script environment.

box

Graphical object that groups together other graphical objects in your chart. For details about how a box affects chart execution, see "Using Boxes to Extend Charts" on page 7-47.

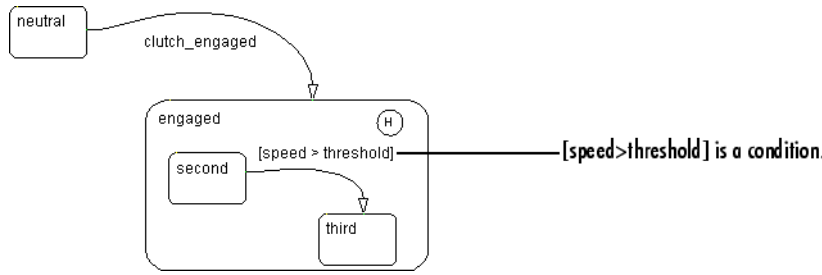
Name	Icon in the Stateflow Editor	Description
Box		Graphically organizes states, transitions, and other graphical objects in your chart.

chart instance

Link from a Stateflow 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 the instances of that chart.

condition

Boolean expression to specify that a transition occurs if the specified expression is true. For example,



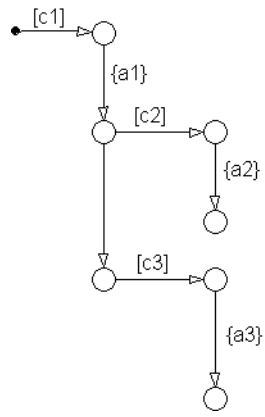
In the preceding example, assume that the state `second` is active. If an event occurs and the value for the data `speed` is greater than the value of the data `threshold`, the transition between states `second` and `third` is taken, and the state `third` becomes active.

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.

This example shows how connective junctions (displayed as small circles) represent the decision flow of an `if` code structure.




```
if [c1]{
      a1
      if [c2]{
         a2
      }else if [c3]{
        a3
      }
}
```

Name	Icon in the Stateflow Editor	Description
Connective junction		Handles situations where transitions out of one state into two or more states can occur based on the same event, but different conditions guard the transitions.

See “Connective Junctions” on page 2-30 for more information.

data

Data objects store numerical values for reference in the Stateflow chart.

See “Adding Data” on page 8-2 for more information on representing data objects.

Debugger

See **Stateflow® Debugger** on page Glossary-10.

decomposition

A state has a *decomposition* when it consists of one or more substates. A Stateflow 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 be of 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 are also used to specify that a junction should be entered by default. A default transition is represented by selecting the default transition object from the Stateflow Editor toolbar and then dropping it to attach to a destination object. The default transition object is a transition with a destination but no source object.

Name	Icon in the Stateflow Editor	Description
Default transition		Indicates, when entering this level in the hierarchy, which state becomes active by default.

See “Default Transitions” on page 2-25 for more information.

Embedded MATLAB function

A function that works with a rich subset of the MATLAB programming language. You can use an Embedded MATLAB function to call MATLAB functions in a Stateflow chart.

Name	Icon in the Stateflow Editor	Description
Embedded MATLAB function		Calls MATLAB functions in a Stateflow chart.

See Chapter 20, “Using Embedded MATLAB Functions in Stateflow Charts” for more information.

events

Events drive the Stateflow chart execution. All events that affect the Stateflow chart must be defined. The occurrence of an event causes the status of the states in the Stateflow chart to be evaluated. The broadcast of an event can trigger a transition to occur and/or can trigger an action to be executed. Events are broadcast in a top-down manner starting from the event’s parent in the hierarchy.

Events are added, removed, and edited through the Model Explorer. See “How Events Work in Stateflow Charts” on page 9-2 for more information.

Finder

Tool to search for objects in Stateflow charts on platforms that do not support the Simulink Find tool. See **Stateflow® Finder** on page Glossary-10.

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 graph
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
Function whose logic is defined by a flow graph. See "Using Graphical Functions to Extend Actions" on page 7-27.

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 "Stateflow Hierarchy of Objects" on page 1-20 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 defined to be the substate that was most recently visited. The history junction applies to the level of the hierarchy in which it appears.

Name	Icon in the Stateflow Editor	Description
History junction		Indicates, when entering this level in the hierarchy, that the last state that was active becomes the next state to be active.

See these sections for more information:

- "History Junctions" on page 2-37
- "Default Transition and a History Junction Example" on page 3-68
- "Labeled Default Transitions Example" on page 3-69
- "Inner Transition to a History Junction Example" on page 3-77

inner transitions

Transition that does not exit the source state. Inner transitions are most powerful when defined for superstates with exclusive (OR) decomposition. Use of inner transitions can greatly simplify a Stateflow chart.

See "Inner Transitions" on page 2-21 and "Inner Transition to a History Junction Example" on page 3-77 for more information.

library link

Link to a chart that is stored in a library model in a Simulink block library.

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. Thus, 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. However, when you generate an embedded or standalone custom target, you include only those charts for which there are links. You can simulate a model that includes links to a library model only if 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 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.

Mealy machine

An industry-standard paradigm for modeling finite-state machines, where output is a function of both inputs *and* state. See Chapter 6, "Building Mealy and Moore Charts" for more information.

Model Explorer

Use to add, remove, and modify data, event, and target objects in the Stateflow hierarchy. See "Using the Model Explorer with Stateflow Objects" on page 24-2 for more information.

Moore machine

An industry-standard paradigm for modeling finite-state machines, where output is a function *only* of state. See Chapter 6, "Building Mealy and Moore Charts" for more information.

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

- 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 2-7 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.

For more information, see "S-Function MEX-Files" on page 22-73.

semantics

Semantics describe how the notation is interpreted and implemented behind the scenes. A completed Stateflow chart communicates how the system will behave. A Stateflow chart contains actions associated with transitions and states. The semantics describe in what sequence these actions take place during Stateflow chart execution.

Simulink function

Graphical object in a Stateflow chart 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.

See Chapter 21, "Using Simulink Functions in Stateflow Charts" for more information.

state

A *state* describes a mode of a reactive system. A reactive system has many possible states. States in a Stateflow 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 Stateflow 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 Stateflow 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.

Name	Icon in the Stateflow Editor	Description
State		Depicts a mode of the system.

Stateflow block

Masked Simulink model that is equivalent to an empty, untitled Stateflow chart. Use the Stateflow block to include a Stateflow 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 1-5 for more information on Stateflow charts.

Stateflow Debugger

Use to debug and animate your Stateflow charts. Each state in the Stateflow chart simulation is evaluated for overall code coverage. This coverage analysis is done automatically when the target is compiled and built with the debug options. The Debugger can also be used to perform dynamic checking. The Debugger operates on the Stateflow machine.

Stateflow Editor

Use this interface to create, modify, or delete states, transitions, and other graphical objects in your Stateflow chart. The toolbar icons represent objects that you can drag and drop into your chart.

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

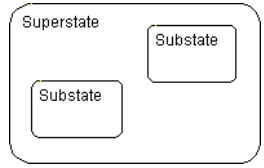
output display by clicking that object. See "Finding Stateflow Objects" on page 24-28 for more information.

subchart

Chart contained by another chart. See "Using Subcharts to Extend Charts" on page 7-5.

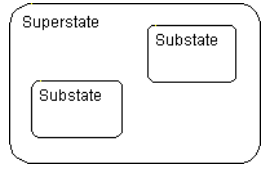
substate

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 "Using Supertransitions to Extend Transitions" on page 7-10 for more information.

target

A container object for the generated code from the Stateflow charts in a model. The collection of all Stateflow 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. See Chapter 22, "Building Targets" for more information.

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. It is often the occurrence of some event that 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. Transition segments are sometimes loosely referred to as transitions.

truth table

Function that specifies logical behavior with conditions, decisions, and actions. Truth tables are easier to program and maintain than graphical functions.

Name	Icon in the Stateflow Editor	Description
Truth table		Tabular method of specifying logical behavior with conditions, decisions, and actions.

See Chapter 19, “Truth Table Functions” for instructions on how to use truth tables in Stateflow charts.

virtual scrollbar

Use this option to set a value by scrolling through a list of choices. When you move the mouse over a menu item with a virtual scrollbar, the cursor changes to a line with a double arrowhead. Virtual scrollbars are either vertical or horizontal. The direction is indicated by the positioning of the arrowheads. Drag the mouse either horizontally or vertically to change the value.

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z