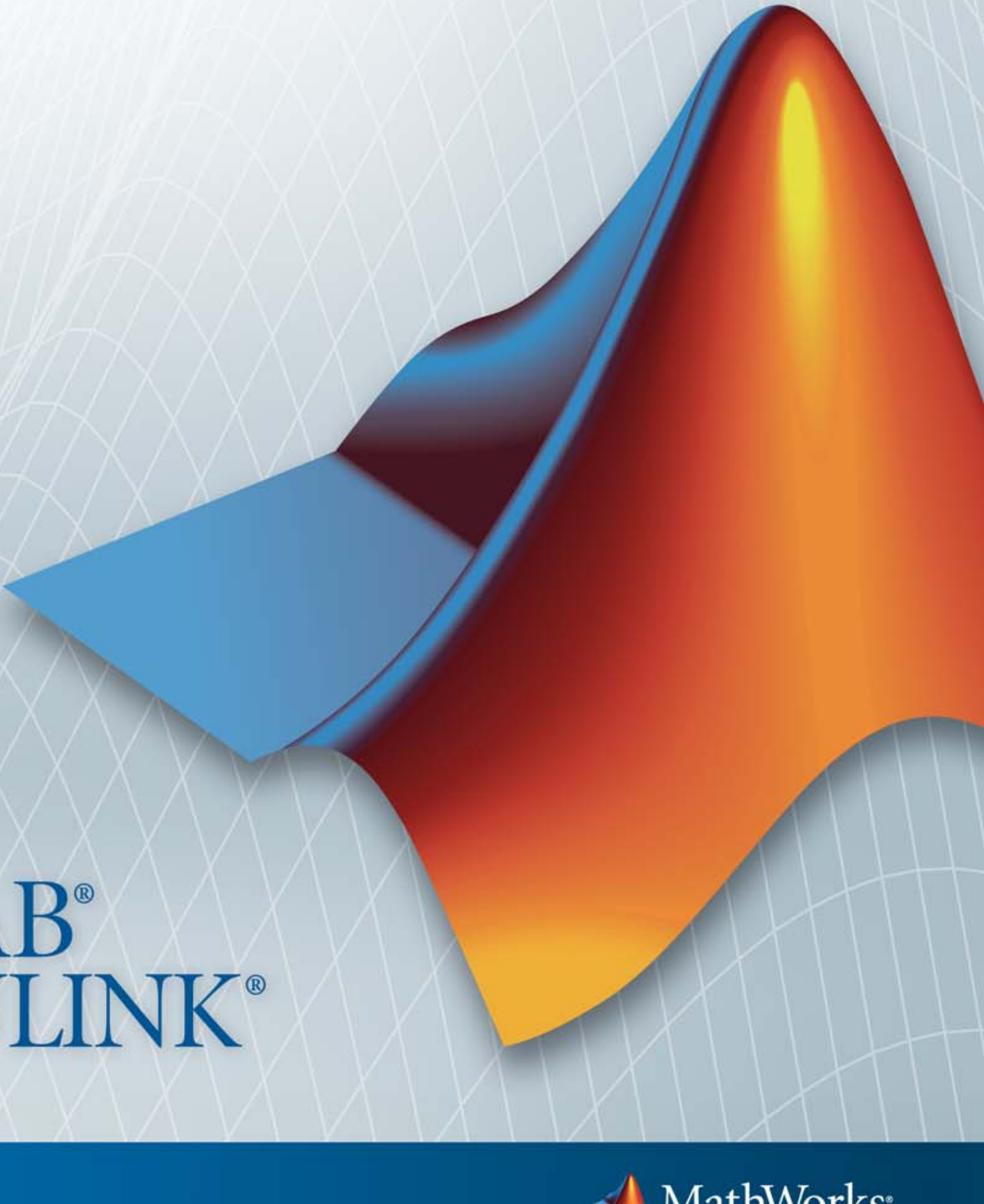
Stateflow®

User's Guide

R2012a



MATLAB® & SIMULINK®

MathWorks®

How to Contact 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® User's Guide

© COPYRIGHT 1997-2012 by The MathWorks, Inc.

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or reproduced in any form without prior written consent from The MathWorks, Inc.

FEDERAL ACQUISITION: This provision applies to all acquisitions of the Program and Documentation by, for, or through the federal government of the United States. By accepting delivery of the Program or Documentation, the government hereby agrees that this software or documentation qualifies as commercial computer software or commercial computer software documentation as such terms are used or defined in FAR 12.212, DFARS Part 227.72, and DFARS 252.227-7014. Accordingly, the terms and conditions of this Agreement and only those rights specified in this Agreement, shall pertain to and govern the use, modification, reproduction, release, performance, display, and disclosure of the Program and Documentation by the federal government (or other entity acquiring for or through the federal government) and shall supersede any conflicting contractual terms or conditions. If this License fails to meet the government's needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to The MathWorks, Inc.

Trademarks

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

Patents

MathWorks products are protected by one or more U.S. patents. Please see `www.mathworks.com/patents` 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)
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 2006a)
September 2006	Online only	Revised for Version 6.5 (Release 2006b)
March 2007	Online only	Revised for Version 6.6 (Release 2007a)
September 2007	Online only	Revised for Version 7.0 (Release 2007b)
March 2008	Online only	Revised for Version 7.1 (Release 2008a)
October 2008	Online only	Revised for Version 7.2 (Release 2008b)
March 2009	Online only	Revised for Version 7.3 (Release 2009a)
September 2009	Online only	Revised for Version 7.4 (Release 2009b)
March 2010	Online only	Revised for Version 7.5 (Release 2010a)
September 2010	Online only	Revised for Version 7.6 (Release 2010b)
April 2011	Online only	Revised for Version 7.7 (Release 2011a)
September 2011	Online only	Revised for Version 7.8 (Release 2011b)
March 2012	Online only	Revised for Version 7.9 (Release 2012a)

Contents

1 Stateflow Chart Concepts

2 Stateflow Chart Notation

3 Stateflow Chart Semantics

Creating Stateflow Charts

4

Modeling Logic Patterns and Iterative Loops Using Flow Graphs

5

6 Building Mealy and Moore Charts

Techniques for Streamlining Chart Design

7

Defining Data

8

9 Defining Events

Using Actions in Stateflow Charts

10

11

Making States Reusable with Atomic Subcharts

12 Saving and Restoring Simulations with SimState

13 Using Vectors and Matrices in Stateflow Charts

14 Using Variable-Size Data in Stateflow Charts

15 Using Enumerated Data in Stateflow Charts

16

Modeling Continuous-Time Systems in Stateflow Charts

17

Using Fixed-Point Data in Stateflow Charts

Using Complex Data in Stateflow Charts

18

19

Defining Interfaces to Simulink Models and the MATLAB Workspace

20

Working with Structures and Bus Signals in Stateflow Charts

Stateflow Design Patterns

21

Truth Table Functions for Decision-Making Logic

22

Using MATLAB Functions in Stateflow Charts

23

24 Using Simulink Functions in Stateflow Charts

25 Building Targets

Debugging and Testing Stateflow Charts

26

27

Exploring and Modifying Charts

A Semantic Rules Summary

B Semantic Examples

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, if the condition defining the change is true.

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

Finite State Machine Representations

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

Stateflow Chart Representations

A Stateflow chart 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 and offspring object structure [2]. 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 these elements:

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

For a description of default semantics, see Chapter 3, "Stateflow Chart Semantics".

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 chart consists 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 appears as a single Stateflow chart. Each Stateflow machine has its own object hierarchy. The Stateflow machine is the highest level in the Stateflow hierarchy. The object hierarchy beneath the Stateflow machine consists of combinations of graphical and nongraphical objects. See "Stateflow Hierarchy of Objects" on page 1-8.

Overview of Defining Stateflow Block Interfaces to Simulink Models

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

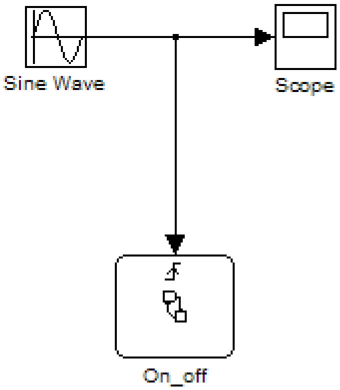
Stateflow charts are event-driven. Events can be local to the Stateflow block or can propagate to and from the Simulink model. Data can be local to the

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

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

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

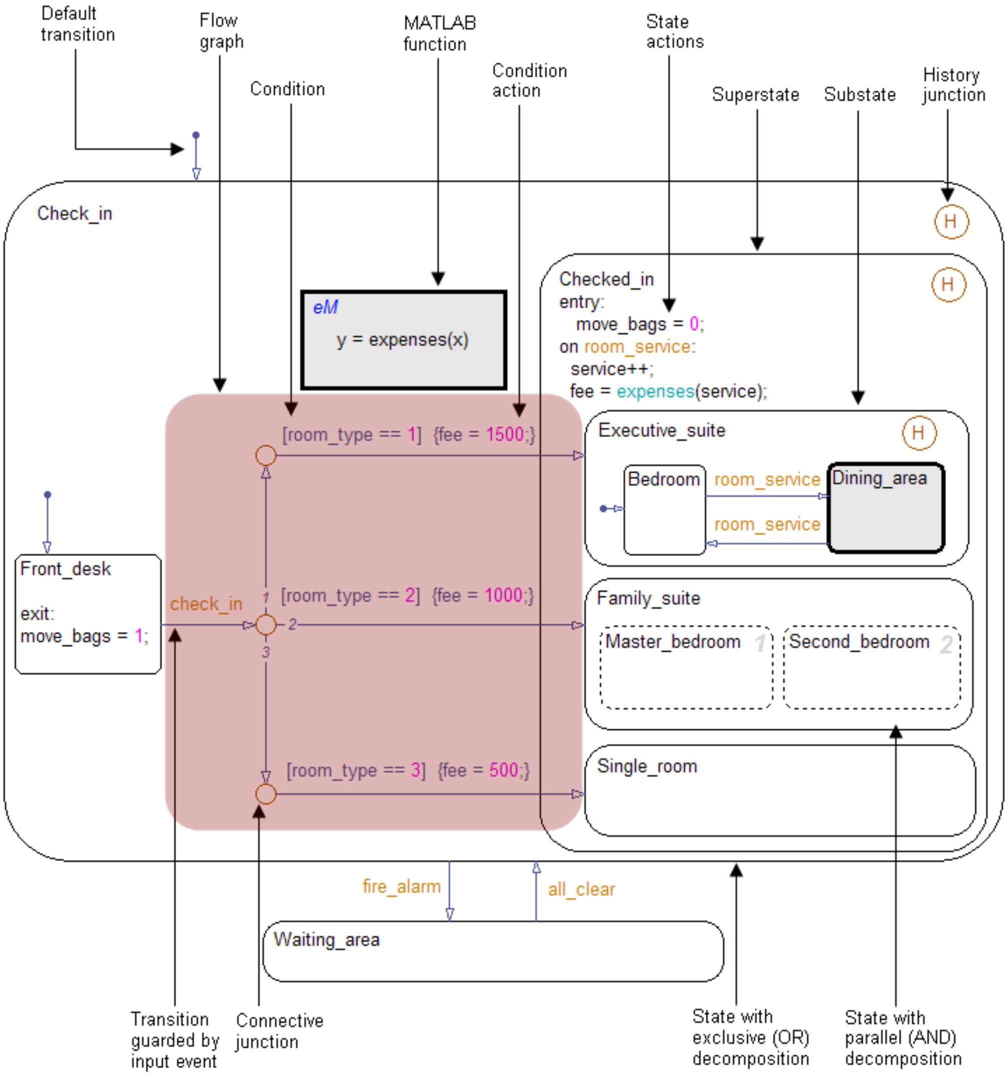
In the following example, the Simulink model consists of a Sine Wave block, a Scope block, and a single Stateflow block, titled `On_off`.



For more information, see “Using Input Events to Activate a Stateflow Chart” on page 9-11 and Chapter 19, “Defining Interfaces to Simulink Models and the MATLAB Workspace”.

Stateflow Chart Objects

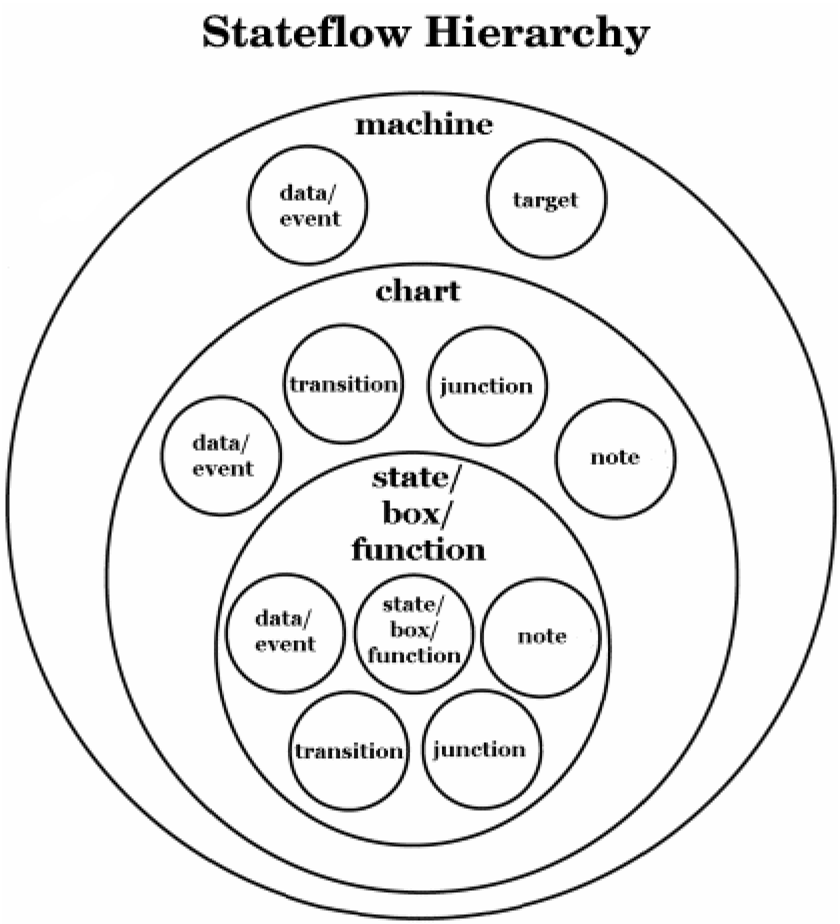
Stateflow charts consist of graphical and nongraphical objects:



To learn how these objects interact, see “How Chart Constructs Interact During Execution” on page 3-9.

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

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

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

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

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

Bibliography

[1] Harel, D. “Statecharts: A Visual Formalism for Complex Systems.” *Science of Computer Programming.* Vol. 8, 1987, pp. 231–274.

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

2

Stateflow Chart Notation

Overview of Stateflow Objects

In this section...
"Graphical Objects" on page 2-2
"Nongraphical Objects" on page 2-3
"For More Information on Stateflow Objects" on page 2-4

Graphical Objects

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

Type of Graphical Object	Toolbar Icon
State	▣
Transition	Not applicable
History junction	H
Default transition	↳
Connective junction	●
Truth table function	TF
Graphical function	f()
MATLAB® function	M

Type of Graphical Object	Toolbar Icon
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 27-2.

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

A target is a program that executes a Stateflow chart or a Simulink model containing a Stateflow machine.

This type of target...	Does this...
Simulation	Executes a simulation of your model
Embeddable code generation	Executes the Simulink model on a supported processor environment
Custom	Pinpoints your application to a specific environment

For more information, see Chapter 25, “Building Targets”.

For More Information on Stateflow Objects

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

Rules for Naming Stateflow Objects

In this section...
"Characters You Can Use" on page 2-5
"Restriction on Name Length" on page 2-5
"Keywords to Avoid When Naming Chart Objects" on page 2-5

Characters You Can Use

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

Restriction on Name Length

Name length should comply with the maximum identifier length enforced by Simulink Coder™ software. You can set this parameter in the **Code Generation > Symbols** pane of the Configuration Parameters dialog box. The default is 31 characters and the maximum length you can specify is 256 characters.

Keywords to Avoid When Naming Chart Objects

You cannot use reserved keywords to name chart objects. These keywords are part of the Stateflow action language (see Chapter 10, "Using Actions in Stateflow Charts" for details).

Usage in Action Language	Keywords	Syntax References
Boolean symbols	• `true` • `false`	"Boolean Symbols, true and false" on page 10-28
Change detection	• `hasChanged` • `hasChangedFrom` • `hasChangedTo`	"Detecting Changes in Data Values" on page 10-83

Usage in Action Language	Keywords	Syntax References
Complex data	• `complex` • `imag` • `real`	"Using Operators to Handle Complex Numbers" on page 18-9
Data types	• `boolean` • `double` • `int8` • `int16` • `int32` • `single` • `uint8` • `uint16` • `uint32`	"Setting Data Properties in the Data Dialog Box" on page 8-5
Data type operations	• `cast` • `fixdt` • `type`	"Type Cast Operations" on page 10-25
Explicit events	• `send`	"Broadcasting Events to Synchronize States" on page 10-59
Implicit events	• `change` • `chg` • `tick` • `wakeup`	"Using Implicit Events" on page 9-40
Literal symbols	• `inf` • `t`	"Supported Symbols in Actions" on page 10-28
MATLAB functions and data	• `matlab` • `ml`	"ml Namespace Operator" on page 10-42

Usage in Action Language	Keywords	Syntax References
State actions	• `bind` • `du` • `during` • `en` • `entry` • `ex` • `exit` • `on`	"Supported Action Types for States and Transitions" on page 10-2
State activity	• `in`	"Checking State Activity" on page 10-97
Temporal logic	• `after` • `at` • `before` • `every` • `sec` • `temporalCount`	"Using Temporal Logic in State Actions and Transitions" on page 10-63

States

In this section...

What Is a State?

A *state* describes an operating mode of a reactive system. In a Stateflow chart, states represent operating modes.

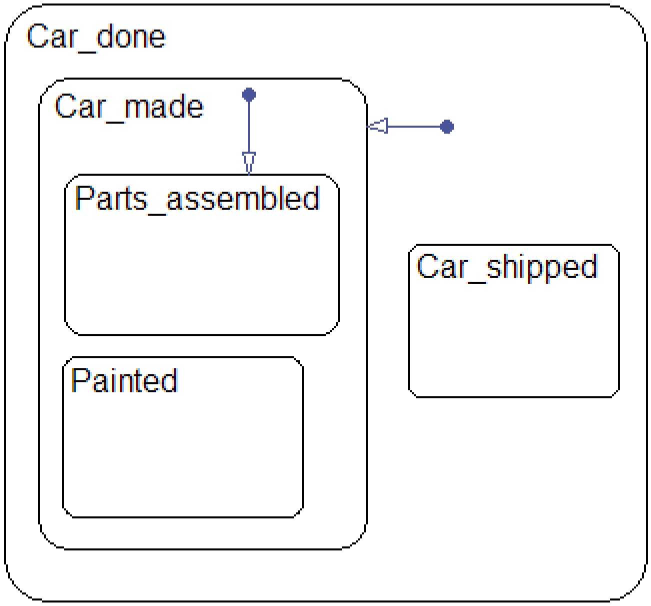
States can be active or inactive. The activity or inactivity of a state can change depending on events and conditions. The occurrence of an event drives the execution of the state chart by making states become active or inactive. At any point during chart execution, active and inactive states exist.

State Hierarchy

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

State Hierarchy Example

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



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

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

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

Objects That a State Can Contain

States can contain all other Stateflow objects 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. You define data and event containment by specifying the parent object of the data or event.

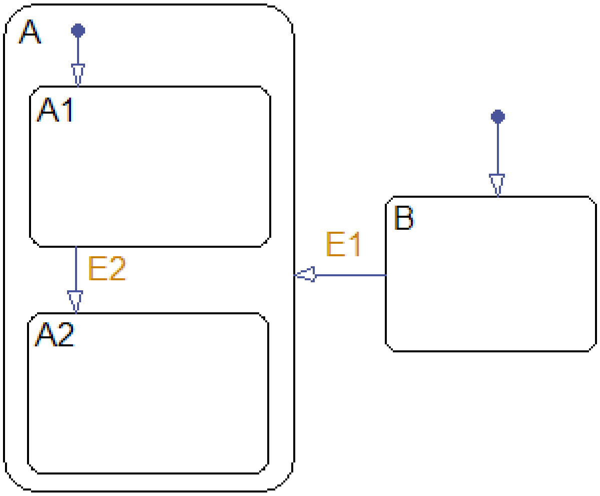
State Decomposition

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

Exclusive (OR) State Decomposition

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

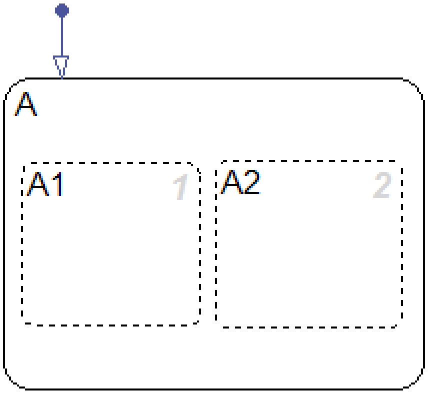
In the following example, either state `A` or state `B` can be active. If state `A` is active, either state `A1` or state `A2` can be active at a given time.



Parallel (AND) State Decomposition

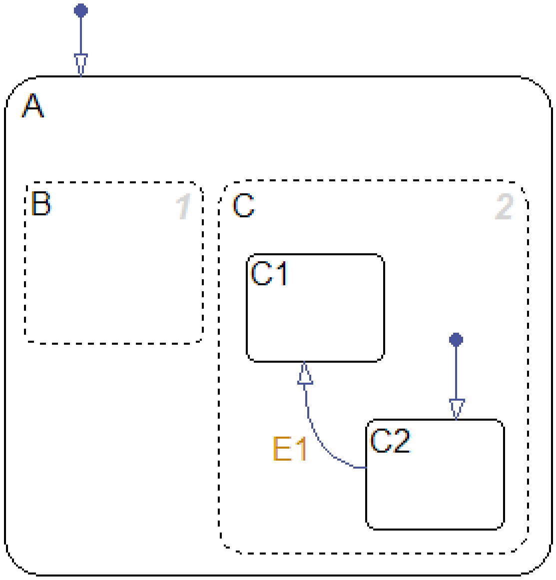
Substates with dashed borders indicate parallel (AND) decomposition. Use this decomposition to describe concurrent operating modes. When a state has parallel (AND) decomposition, all substates are active at the same time.

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



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

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

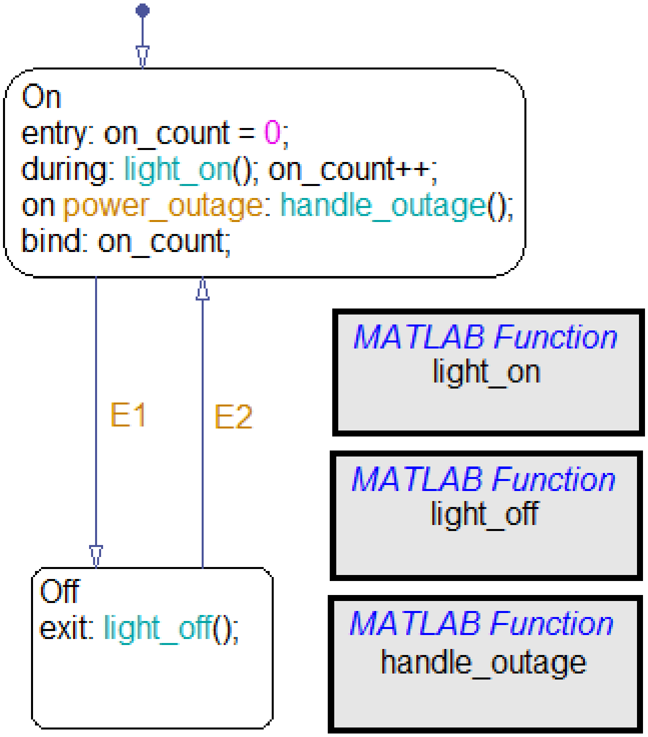


State Labels

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

```
name/
entry:entry actions
during:during actions
exit:exit actions
on event_name:on event_name actions
bind:events, data
```

The following example demonstrates the components of a state label.



Each action in the state label appears in the subtopics that follow. For more information on state actions, see:

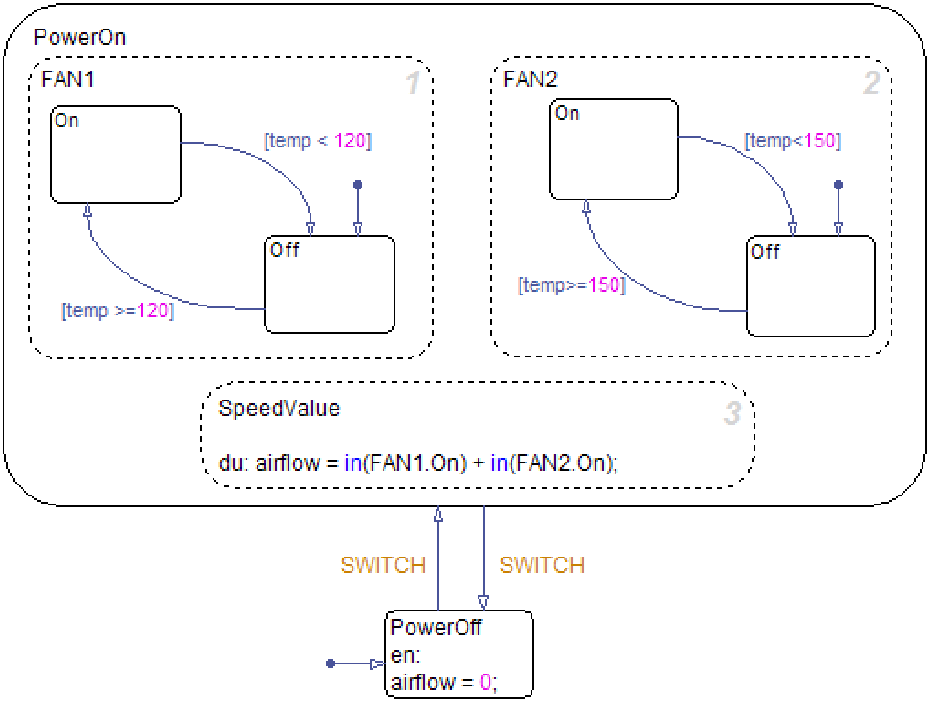
- "Process for Entering, Executing, and Exiting States" on page 3-70 — Describes how and when `entry`, `during`, `exit`, and `on` *`event_name`* actions occur.
- "State Action Types" on page 10-2 — Gives more detailed descriptions of each type of state action.

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 more information, see "Rules for Naming Stateflow Objects" on page 2-5.

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

The following example shows how unique naming of states works.



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

- `PowerOn.FAN1.On`

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

State Actions

After the name, you enter optional action statements for the state with a keyword label that identifies the type of action. You can specify none, some, or all of them. The colon after each keyword is required. The slash following the state name is optional as long as it is followed by a carriage return.

For each type of action, you can enter more than one action by separating each action with a carriage return, semicolon, or a comma. You can specify actions for more than one event 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.

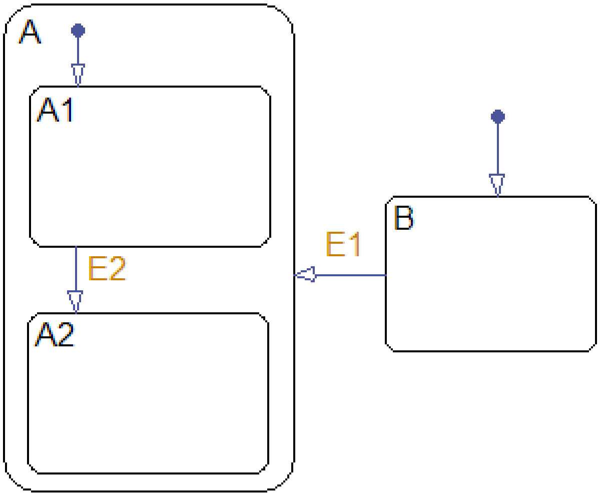
Bind Action. Preceded by the prefix `bind`. In the preceding label example, the data `on_count` is bound to the state `On`. This means that only the state `On` or a child of `On` can change the value of `on_count`. Other states, such as the state `Off`, can use `on_count` in its actions, but it cannot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

Transitions

In this section...

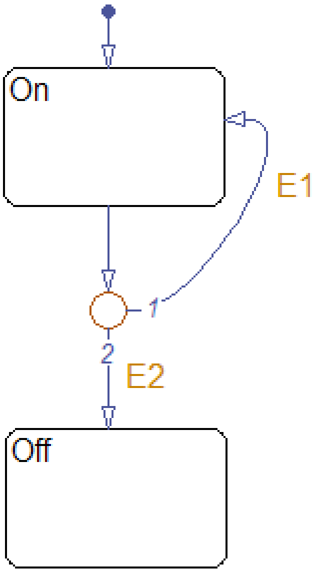
What Is a Transition?

A *transition* is a 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 typically connects a source and a destination object. The *source* object is where the transition begins and the *destination* object is where the transition ends. The following chart shows a transition from a source state, `B`, to a destination state, `A`.



Junctions divide a transition into transition segments. In this case, a full transition consists of the segments taken from the origin to the destination state. Each segment is evaluated in the process of determining the validity of a full transition.

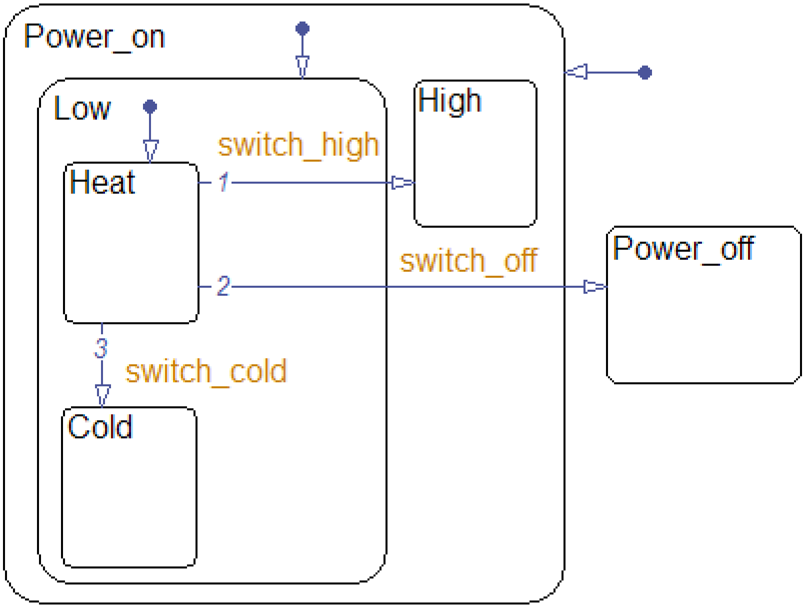
The following example has two segmented transitions: one from state `On` to state `Off`, and the other from state `On` to itself:



A default transition is a special type of transition that has no source object. See "Default Transitions" on page 2-31 for details.

Transition Hierarchy

Transitions cannot contain other objects the way that 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 `/` character represents the chart. Each level in the hierarchy of states is separated by the period (`.`) character.

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

Transition Label Notation

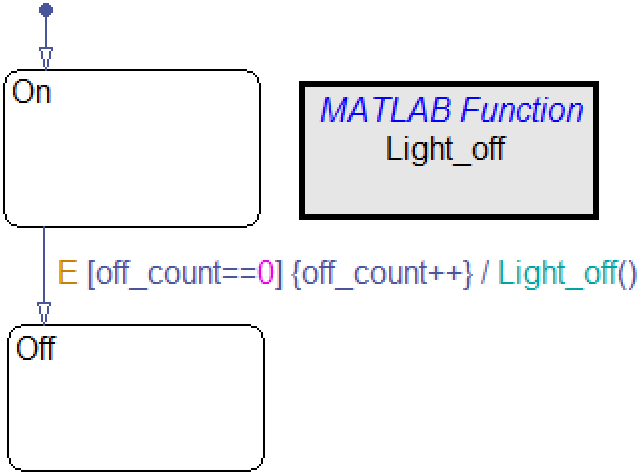
A transition is characterized by its *label*. The label can consist of an event, a condition, a condition action, and/or a transition action. The ? character is the default transition label. Transition labels have the following general format:

`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 2-20. 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. Specify multiple events using the OR logical operator (`|`).

In the preceding example, the broadcast of event `E` triggers the transition from `On` to `Off` as long as 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 10-10 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 table lists possible combinations of valid transition labels.

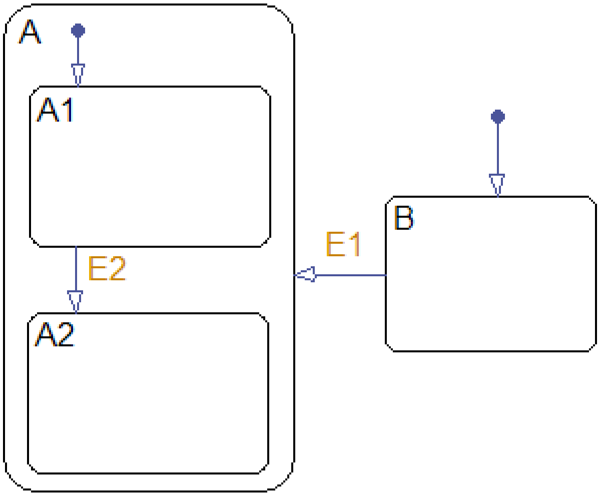
Transition Label	Is Valid If...
Event only	That event occurs
Event and condition	That event occurs and the condition is true
Condition only	Any event occurs and the condition is true
Action only	Any event occurs
Not specified	Any event occurs

Transition Connections

In this section…

Transitions to and from Exclusive (OR) States

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

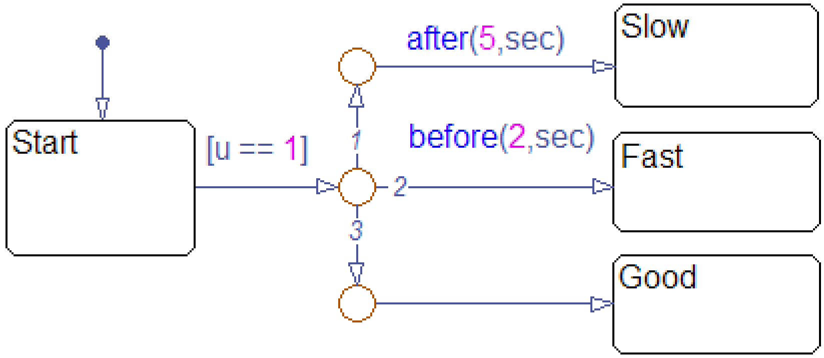


The following transition...	Is valid when...
`B` to `A`	State `B` is active and the event `E1` occurs.
`A1` to `A2`	State `A1` is active and event `E2` occurs.

See “Transitions to and from Exclusive (OR) States Examples” on page B-4 for more information on the semantics of this notation.

Transitions to and from Junctions

The following chart shows transitions to and from connective junctions.



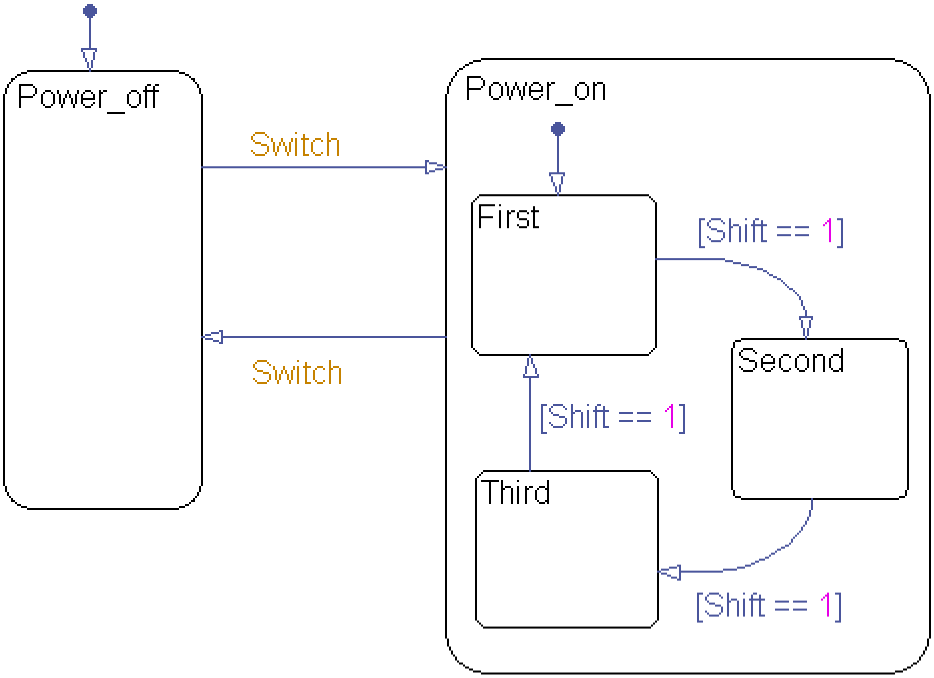
The chart uses temporal logic to determine when the input `u` equals 1.

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

For more information about temporal logic, see “Using Temporal Logic in State Actions and Transitions” on page 10-63. For more information on the semantics of this notation, see "Transitions from a Common Source to Multiple Destinations Example" on page B-42.

Transitions to and from Exclusive (OR) Superstates

This example shows transitions to and from an exclusive (OR) superstate and the use of a default transition.

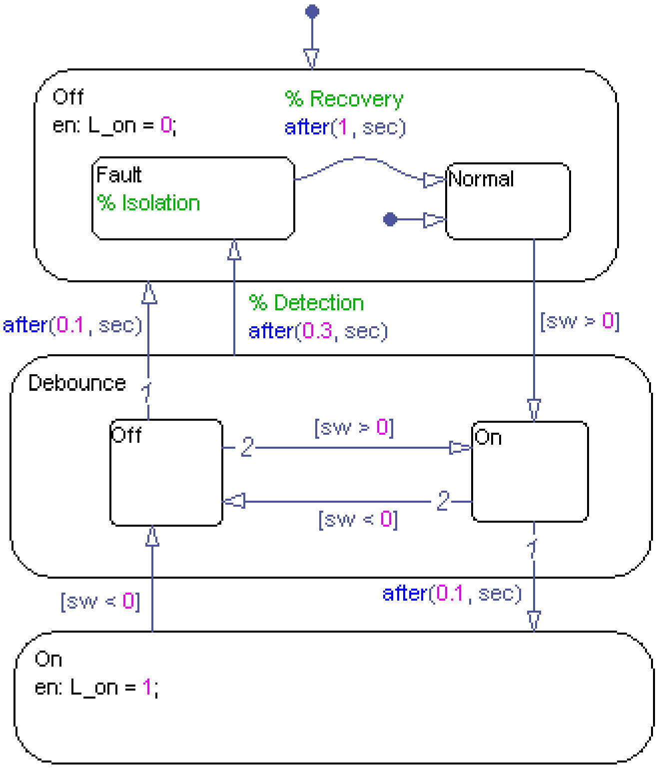


The chart has two states at the highest level in the hierarchy, `Power_off` and `Power_on`. By default, `Power_off` is active. The event `Switch` toggles the system between the `Power_off` and `Power_on` states. `Power_on` has three substates: `First`, `Second`, and `Third`. By default, when `Power_on`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

For more information on the semantics of this notation, see “Default Transition Examples” on page B-18.

Transitions to and from Substates

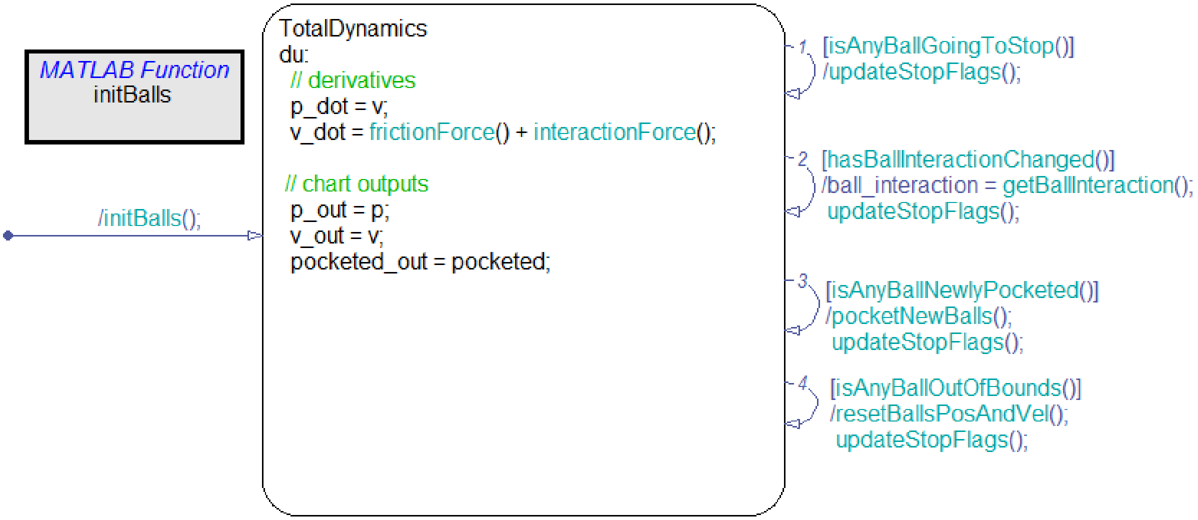
The following example shows transitions to and from exclusive (OR) substates.



For details on how this chart works, see “Key Behaviors of Debouncer Chart” on page 21-4. For information on the semantics of this notation, see “Transitioning from a Substate to a Substate with Events Example” on page B-9.

Self-Loop Transitions

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



```
/initBalls();
```

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

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

```
[isAnyBallGoingToStop()]
/updateStopFlags();

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

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

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

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

- "Self-Loop Transition Example" on page B-37
- "For-Loop Construct Example" on page B-39

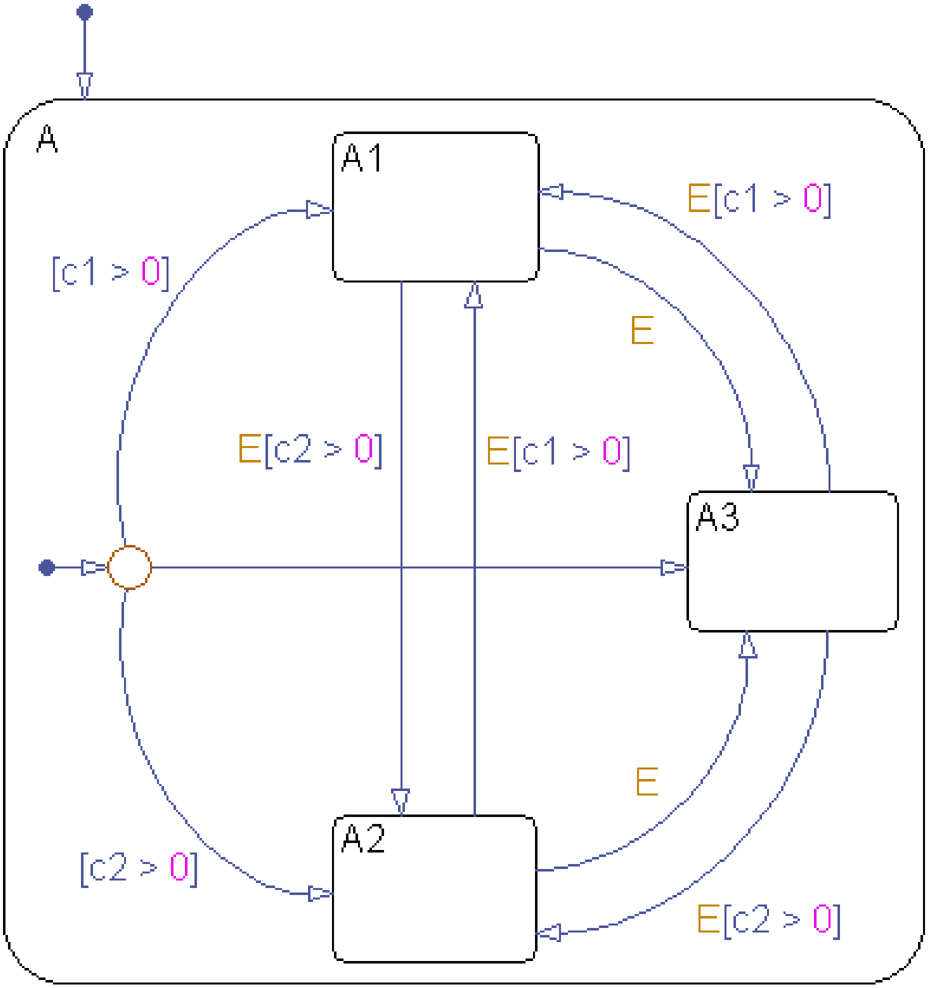
Inner Transitions

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 2-27
- "After Using an Inner Transition to a Connective Junction" on page 2-28
- "Using an Inner Transition to a History Junction" on page 2-29

Before Using an Inner Transition

This chart is an example of how you can simplify logic using an inner transition.

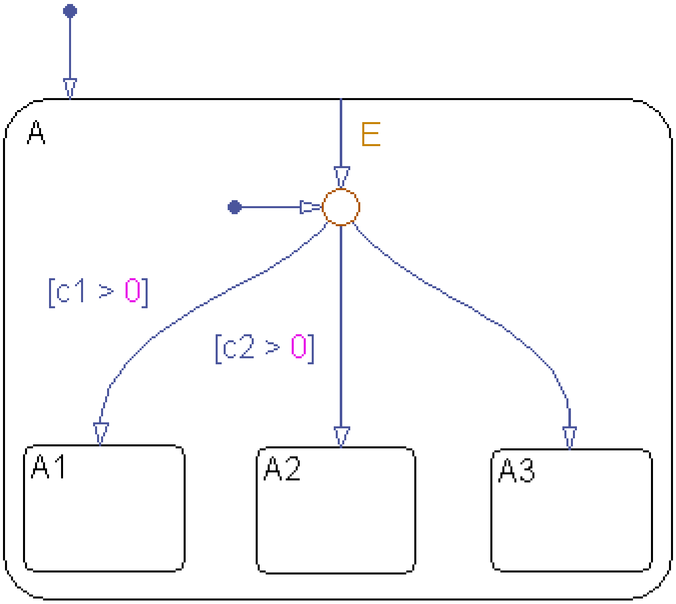


Any event occurs and awakens the Stateflow chart. The default transition to the connective junction is valid. The destination of the transition is determined by `[c1 > 0]` and `[c2 > 0]`. If `[c1 > 0]` is true, the transition to

`A1` is true. If `[c2 > 0]` is true, the transition to `A2` is valid. If neither `[c1 > 0]` nor `[c2 > 0]` is true, the transition to `A3` is valid. The transitions among `A1`, `A2`, and `A3` are determined by `E`, `[c1 > 0]`, and `[c2 > 0]`.

After Using an Inner Transition to a Connective Junction

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



An event occurs and awakens the chart. The default transition to the connective junction is valid. The destination of the transitions is determined by `[c1 > 0]` and `[c2 > 0]`.

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

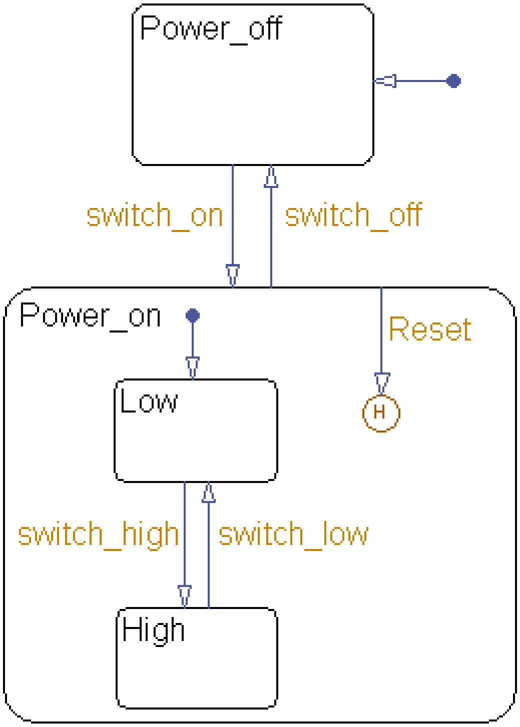
Note When you use an inner transition to a connective junction, an active substate can exit and reenter when the transition condition for that substate is valid. For example, if substate `A1` is active and `[c1 > 0]` is true, the transition to `A1` is valid. In this case:

1. Exit actions for `A1` execute and complete.
2. `A1` becomes inactive.
3. `A1` becomes active.
4. Entry actions for `A1` execute and complete.

See "Processing the First Event with an Inner Transition to a Connective Junction" on page B-28 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 2-43 for another example using a history junction.

See "Inner Transition to a History Junction Example" on page B-31 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.

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.

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

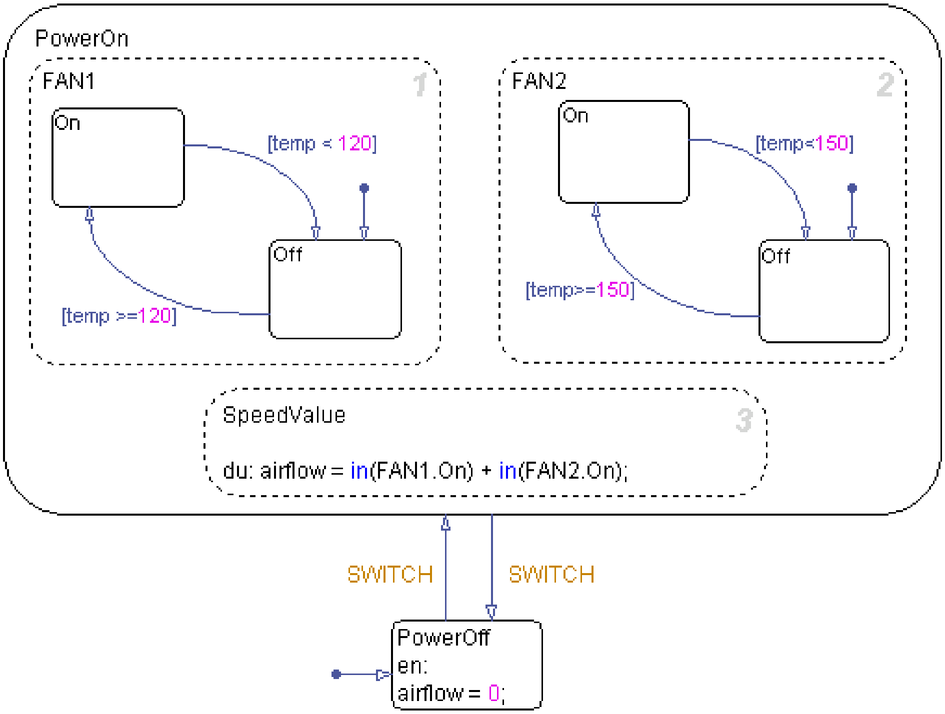
Default Transition Examples

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

- "Default Transition to a State Example" on page 2-32
- "Default Transition to a Junction Example" on page 2-33
- "Default Transition with a Label Example" on page 2-34

Default Transition to a State Example

This example shows a default transition to a state.

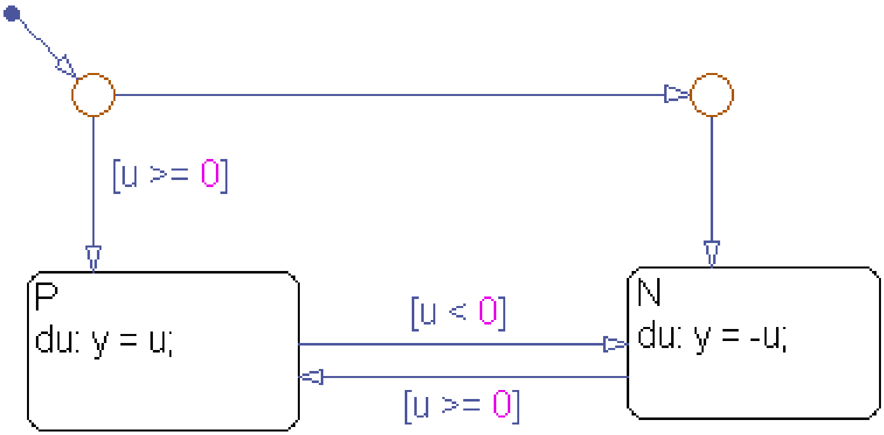


Without the default transition to state `PowerOff`, when the Stateflow chart wakes up, none of the states becomes active. You can detect this situation at run-time by checking for state inconsistencies. See “Animating Stateflow Charts in Normal Mode” on page 26-4 for more information.

See “Default Transition Examples” on page B-18 for information on the semantics of this notation.

Default Transition to a Junction Example

This example shows a default transition to a connective junction.

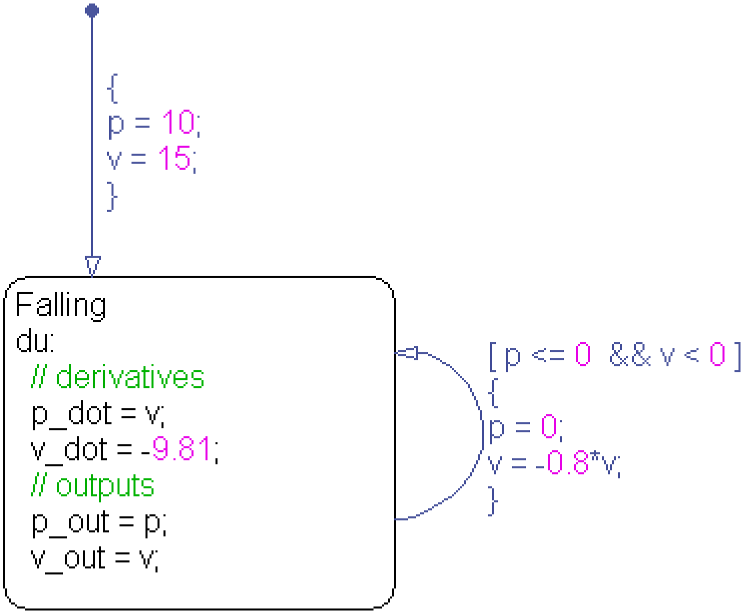


The default transition to the connective junction defines that upon entering the chart, the destination depends on the condition of each transition segment.

See “Default Transition to a Junction Example” on page B-19 for information on the semantics of this notation.

Default Transition with a Label Example

This example shows a default transition with a label.



```
{
p = 10;
v = 15;
}
```

```
Falling
du:
  // derivatives
  p_dot = v;
  v_dot = -9.81;
  // outputs
  p_out = p;
  v_out = v;
```

```
[ p <= 0  && v < 0 ]
{
p = 0;
v = -0.8*v;
}
```

When the chart wakes up, the data p and v initialize to 10 and 15, respectively.

See "Labeled Default Transitions Example" on page B-22 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 B-34 for a summary of the semantics of connective junctions.

Flow Graph Notation with Connective Junctions

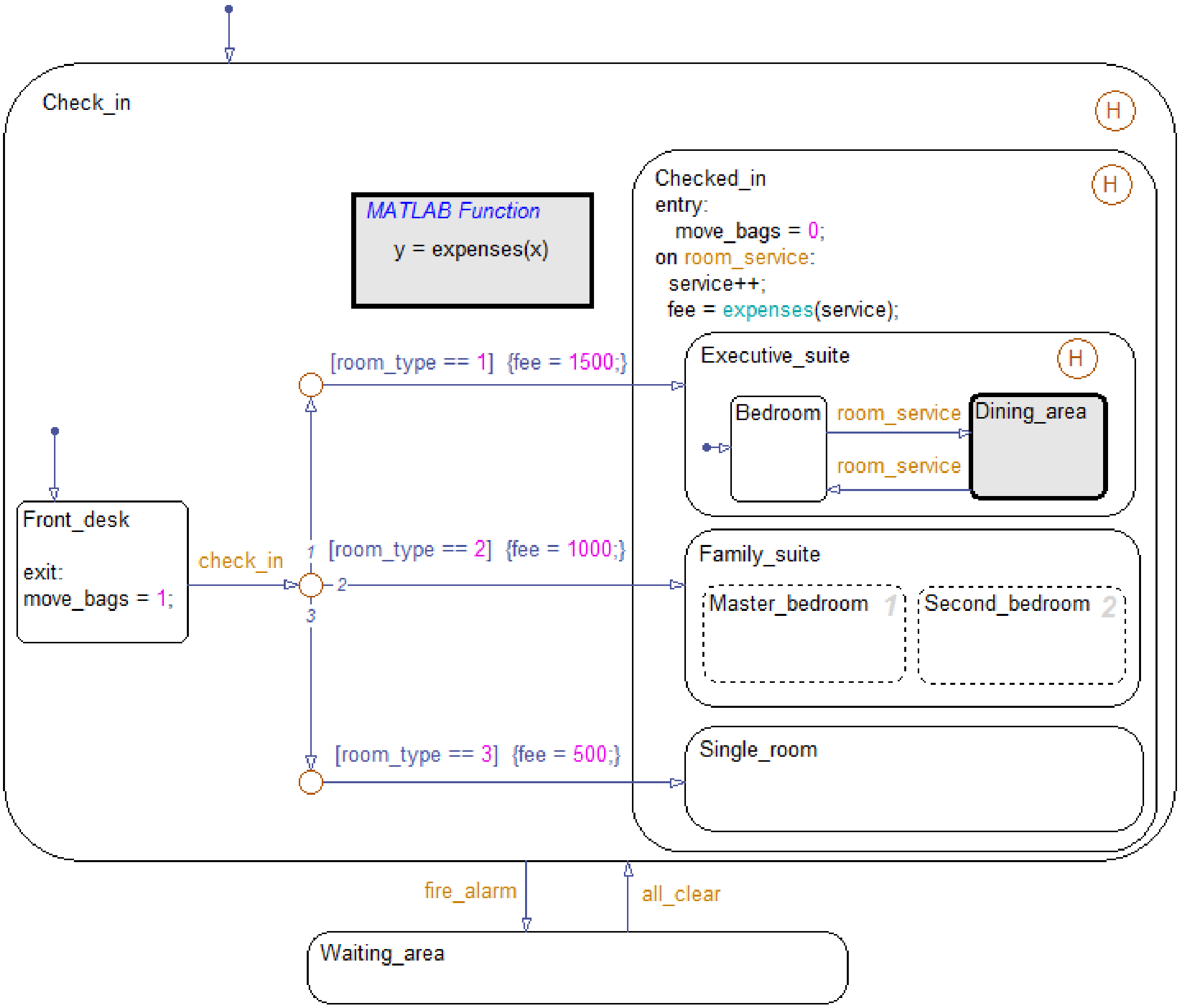
Flow graph notation uses connective junctions to represent common code structures like `for` loops and `if-then-else` constructs without the use of states. And by reducing the number of states in your Stateflow charts, flow graph notation produces efficiently generated code that helps optimize memory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

Flow graph notation uses 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 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.

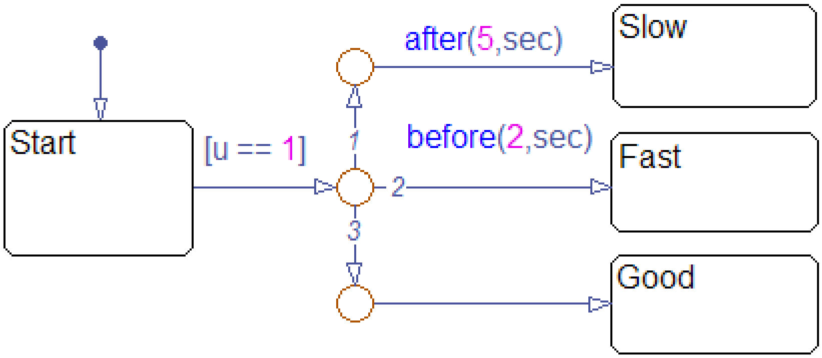
Connective Junction with All Conditions Specified Example



A transition from the `Front_desk` state to a connective junction is labeled by the `check_in` event. Transitions from the connective junction to the destination states are labeled with conditions. If `Front_desk` is active when `check_in` occurs, the transition from `Front_desk` to the connective junction occurs first. The transition from the connective junction to a destination state depends on which of the `room_type` conditions is true. If none of the conditions is true, no transition occurs and `Front_desk` remains active.

For more information about this chart, see “Phases of Chart Execution” on page 3-14. For more information on the semantics of this notation, see “If-Then-Else Decision Construct Example” on page B-36.

Connective Junction with One Unconditional Transition Example



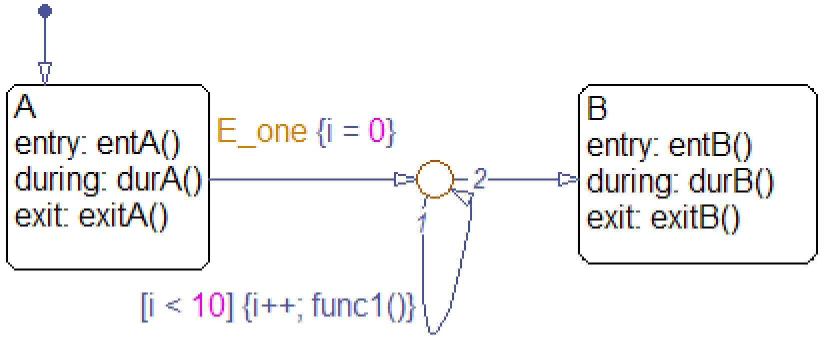
The chart uses temporal logic to determine when the input `u` equals 1.

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

For more information about temporal logic, see “Using Temporal Logic in State Actions and Transitions” on page 10-63. For more information on the semantics of this notation, see “If-Then-Else Decision Construct Example” on page B-36.

Connective Junction and For Loops Example

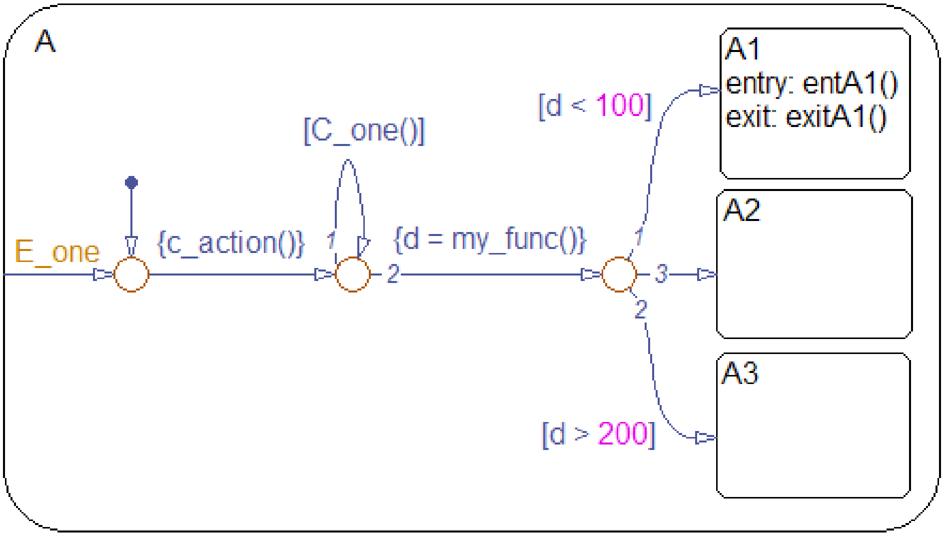
This example shows a combination of flow graph notation and state transition notation. Self-loop transitions to connective junctions can represent `for` loop constructs. The chart uses implicit ordering of outgoing transitions (see "Implicit Ordering of Outgoing Transitions" on page 3-59).



See "For-Loop Construct Example" on page B-39 for information on the semantics of this notation.

Flow Graph Notation Example

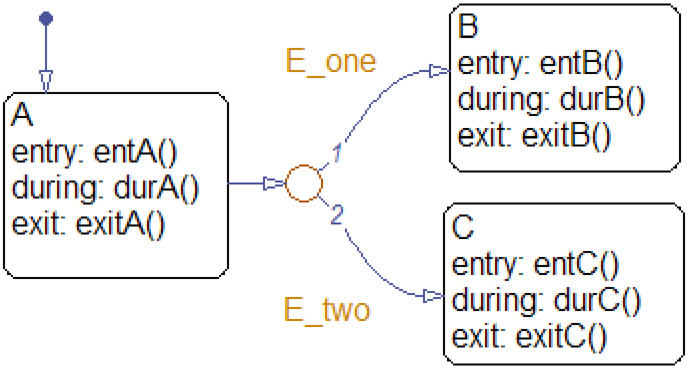
This example shows the use of flow graph notation. The chart uses implicit ordering of outgoing transitions (see "Implicit Ordering of Outgoing Transitions" on page 3-59).



See “Flow Graph Notation Example” on page B-40 for information on the semantics of this notation.

Connective Junction from a Common Source to Multiple Destinations Example

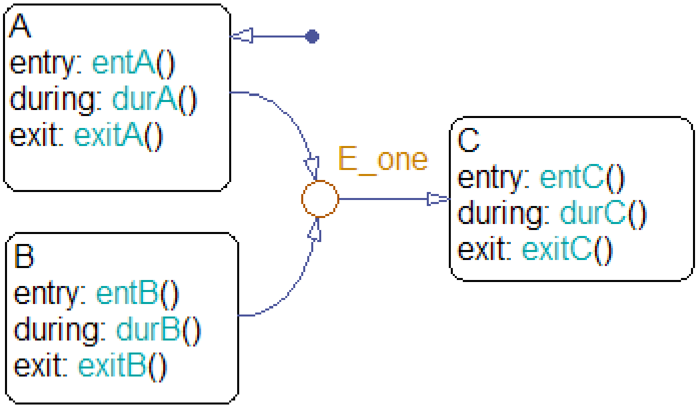
This example shows transition segments from a common source to multiple conditional destinations using a connective junction. The chart uses implicit ordering of outgoing transitions (see “Implicit Ordering of Outgoing Transitions” on page 3-59).



See "Transitions from a Common Source to Multiple Destinations Example" on page B-42 for information on the semantics of this notation.

Connective Junction Common Events Example

This example shows transition segments from multiple sources to a single destination based on the same event using a connective junction.



See "Transitions from a Source to a Destination Based on a Common Event Example" on page B-45 for information on the semantics of this notation.

History Junctions

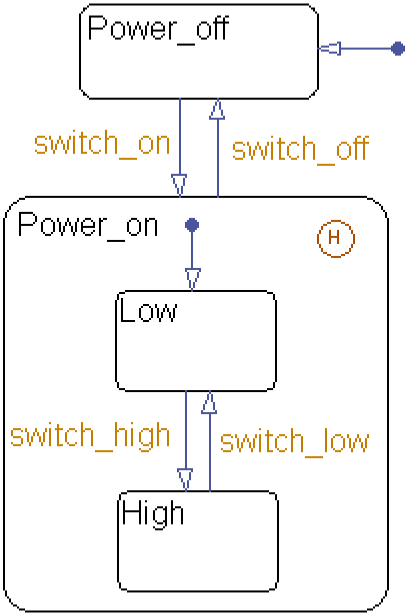
In this section...
"What Is a History Junction?" on page 2-43
"History Junctions and Inner Transitions" on page 2-44

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

See "Default Transition and a History Junction Example" on page B-20 for more information on the semantics of this notation.

History Junctions and Inner Transitions

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

See "Using an Inner Transition to a History Junction" on page 2-29 for an example of this notation.

See "Inner Transition to a History Junction Example" on page B-31 for more information on the semantics of this notation.

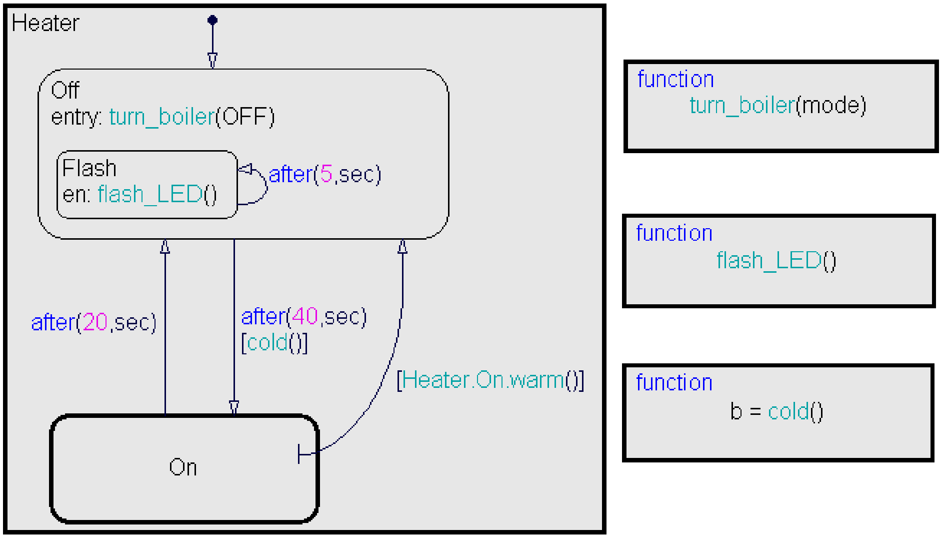
Boxes

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 `Heater` groups together related states `Off` and `On`.



For rules of using boxes and other examples, see "Grouping Chart Objects with Boxes" on page 7-50.

When to Use Reusable Functions in State Charts

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

- Flow graph — Encapsulate flow graphs containing if-then-else, switch-case, for, while, or do-while patterns.
- MATLAB — Write matrix-oriented algorithms; call MATLAB functions for data analysis and visualization.
- Simulink — Call Simulink function-call subsystems directly to streamline design and improve readability.
- Truth table — Represent combinational logic for decision-making applications such as fault detection and mode switching.

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

If the four standard types of Stateflow functions do not work, you can write your own C or C++ code for integration with your state chart. For more information about custom code integration, see Chapter 25, “Building Targets”.

Stateflow Chart Semantics

What Do Semantics Mean for Stateflow Charts?

In this section…

What Are Chart Semantics?

Chart semantics describe execution behavior according to the interaction of graphical and nongraphical constructs.

Graphical Constructs

Graphical constructs consist of objects that appear graphically in a chart. You use the object palette in the Stateflow Editor to build graphical constructs (see "Editor Operations" on page 4-32).

Graphical Constructs	Types	References
Flow graphs	• Decision logic patterns • Loop logic patterns	Chapter 5, "Modeling Logic Patterns and Iterative Loops Using Flow Graphs"
Functions	• Graphical functions • MATLAB functions • Truth table functions • Simulink functions	• "What Is a Graphical Function?" on page 7-30 • "Example of a MATLAB Function in a Stateflow Chart" on page 23-5 • "What Is a Truth Table?" on page 22-2 • "What Is a Simulink Function?" on page 24-2

Graphical Constructs	Types	References
Junctions	• Connective junctions • History junctions	• “Connective Junctions” on page 2-36 • “History Junctions” on page 2-43
States	• States with exclusive (OR) decomposition • States with parallel (AND) decomposition • Substates and superstates	• “Exclusive (OR) State Decomposition” on page 2-10 • “Parallel (AND) State Decomposition” on page 2-11 • “Creating Substates and Superstates” on page 4-11
Transitions	• Default transitions • Object-to-object transitions • Inner transitions • Self-loop transitions	• “Default Transitions” on page 2-31 • “Transition Connections” on page 2-22

Nongraphical Constructs

Nongraphical constructs appear textually in a chart and often refer to data and events (see Chapter 8, “Defining Data” and Chapter 9, “Defining Events”). You follow syntax rules of the Stateflow action language to build nongraphical constructs (see Chapter 10, “Using Actions in Stateflow Charts”).

Examples of nongraphical constructs include:

- Conditions and condition actions
- Function calls
- State actions
- Temporal logic statements

Common Graphical and Nongraphical Constructs

The following chart shows commonly used graphical and nongraphical constructs.

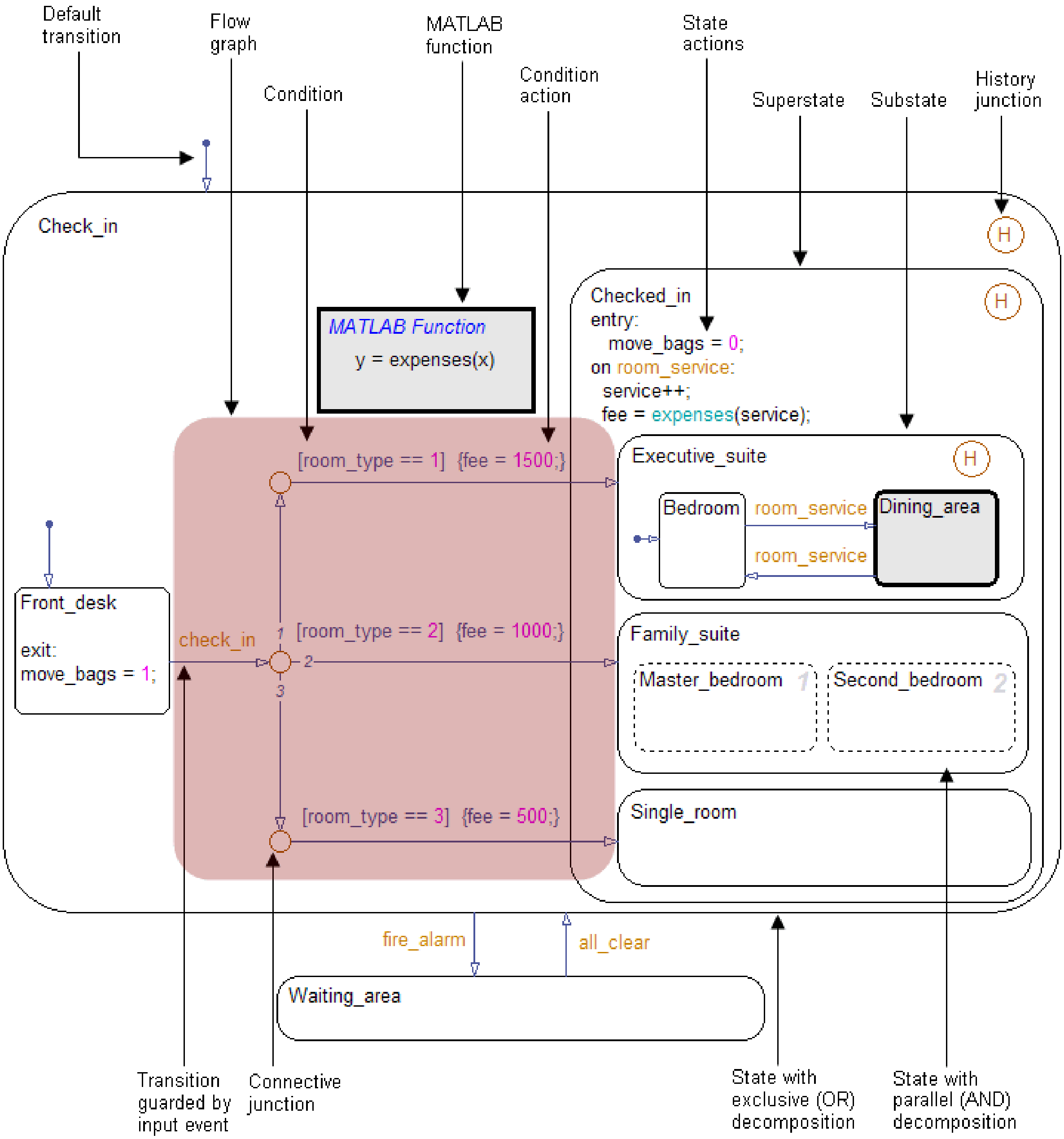

Default transition
Flow graph
Condition
MATLAB function
Condition action
State actions
Superstate
Substate
History junction
Check_in
H
Checked_in
entry:
move_bags = 0;
on room_service:
service++;
fee = expenses(service);
H
MATLAB Function
y = expenses(x)
[room_type == 1] {fee = 1500;}
Executive_suite
H
Bedroom
room_service
Dining_area
room_service
Front_desk
exit:
move_bags = 1;
check_in
1
2
3
[room_type == 2] {fee = 1000;}
Family_suite
Master_bedroom 1
Second_bedroom 2
[room_type == 3] {fee = 500;}
Single_room
fire_alarm
all_clear
Waiting_area
Transition guarded by input event
Connective junction
State with exclusive (OR) decomposition
State with parallel (AND) decomposition

Chart Construct	Graphical or Nongraphical?	Description	Reference
Condition	Nongraphical	Boolean expression that specifies that a transition path is valid if the expression is true; part of a transition label	"Transition Label Notation" on page 2-19 and "Conditions" on page 10-10
Condition action	Nongraphical	Action that executes as soon as the condition evaluates to true; part of a transition label	"Transition Label Notation" on page 2-19 and "Condition Actions" on page 10-11
Connective junction	Graphical	Object that enables representation of different possible transition paths in a flow graph	"Connective Junctions" on page 2-36
Default transition	Graphical	Object that specifies which state to enter when two or more exclusive (OR) states exist at the same level of hierarchy	"Default Transitions" on page 2-31
Flow graph	Graphical	Construct that models logic patterns by using connective junctions and transitions	Chapter 5, "Modeling Logic Patterns and Iterative Loops Using Flow Graphs"
History junction	Graphical	Object that remembers the previously active state at the level of hierarchy in which it appears	"History Junctions" on page 2-43

Chart Construct	Graphical or Nongraphical?	Description	Reference
MATLAB function	Graphical	Method of performing computations using a subset of the MATLAB language	Chapter 23, "Using MATLAB Functions in Stateflow Charts"
State actions	Nongraphical	Expressions that specify actions to take when a state is active, such as initializing or updating data; part of a state label	"State Labels" on page 2-12 and "State Action Types" on page 10-2
State with exclusive (OR) decomposition	Graphical	State where no more than one substate can be active at a time	"Exclusive (OR) State Decomposition" on page 2-10
State with parallel (AND) decomposition	Graphical	State where all substates can be active at the same time	"Parallel (AND) State Decomposition" on page 2-11
Substate	Graphical	State that resides inside another state	"Creating Substates and Superstates" on page 4-11
Superstate	Graphical	State that contains one or more states	"Creating Substates and Superstates" on page 4-11
Transition guarded by input event	Graphical	Decision path that occurs if the chart receives a specific event broadcast from another block in the model	"Transition Action Types" on page 10-7

For details on how these graphical and nongraphical constructs interact during chart execution, see "How Chart Constructs Interact During Execution" on page 3-9.

References for Chart Semantics

For detailed information on types of chart semantics, see these references.

Topic	Reference
How do events affect chart execution?	"How Events Drive Chart Execution" on page 3-37
How does a chart switch between being active and inactive?	"Types of Chart Execution" on page 3-40
In what order do flow graphs execute?	"Process for Grouping and Executing Transitions" on page 3-51
In what order do outgoing transitions from a single source execute?	"Evaluation Order for Outgoing Transitions" on page 3-55
What happens when you enter, execute, or exit a state?	"Process for Entering, Executing, and Exiting States" on page 3-70
How do parallel (AND) states work?	"Execution Order for Parallel States" on page 3-75
How does early return logic affect chart execution?	"Early Return Logic for Event Broadcasts" on page 3-85

For detailed examples of chart semantics, see Appendix B, "Semantic Examples".

How Chart Constructs Interact During Execution

In this section...

Overview of the Example Model

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

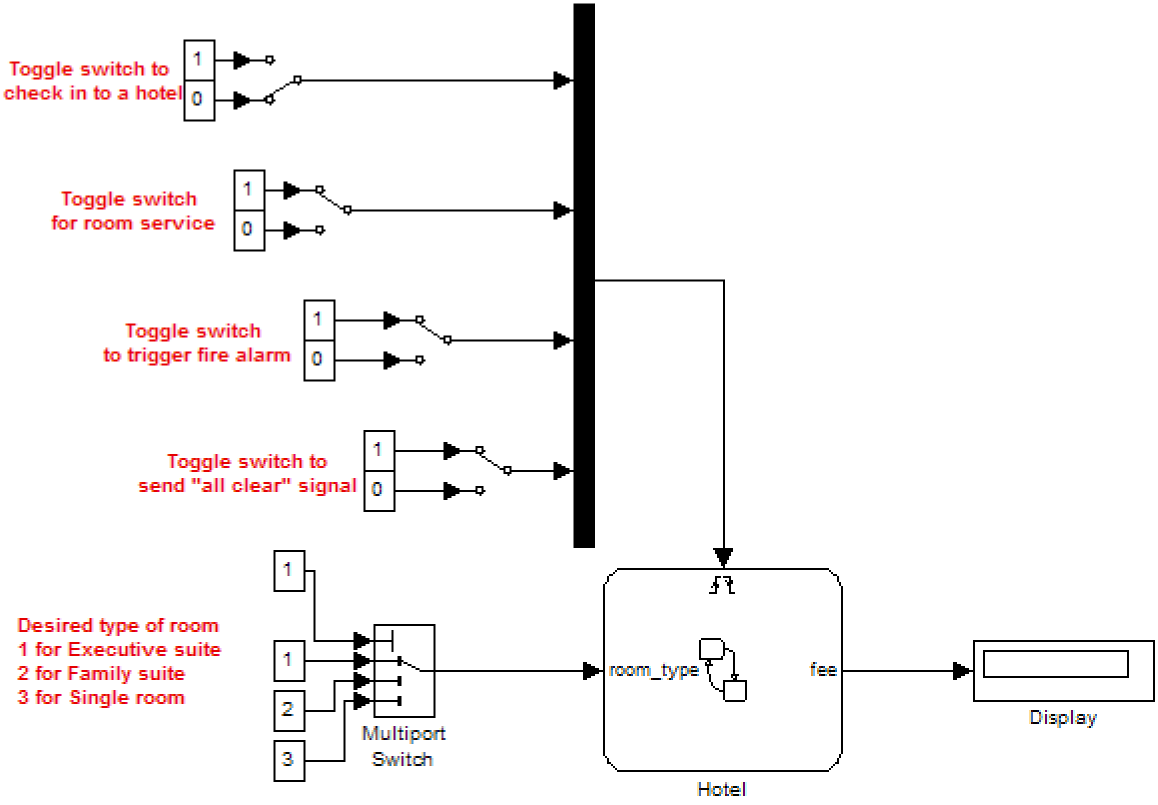
- Conditions and condition actions
- Exclusive (OR) states
- Flow graphs
- Function calls
- History junctions
- Parallel (AND) states
- State actions
- Transitions guarded by input events

For details of the chart semantics, see "Phases of Chart Execution" on page 3-14.

Model of the Check-In Process for a Hotel

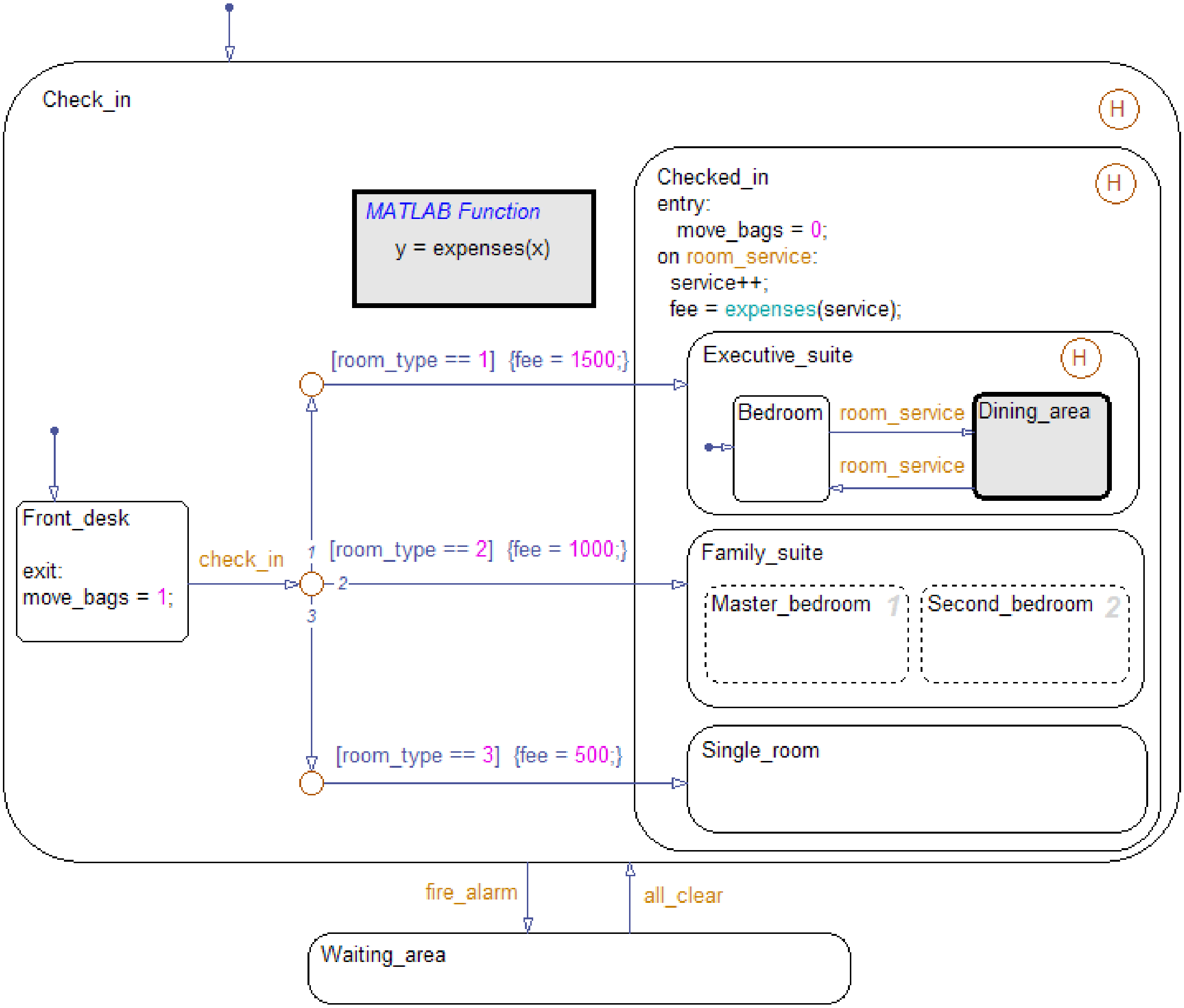
This example uses the hotel check-in process to explain Stateflow chart semantics. To open the model, type `sf_semantics_hotel_checkin` at the MATLAB command prompt.

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



The model uses this block...	To...	Because...
Manual Switch	Enable toggling between two settings during simulation without having to pause or restart.	During simulation, you can interactively trigger the chart by sending one of these input events: • Checking in to a hotel • Calling room service • Triggering a fire alarm • Sending an all-clear signal after a fire alarm
Mux	Combine multiple input signals into a vector.	A chart can support multiple input events only if they connect to the trigger port of a chart as a vector of inputs.
Multiport Switch	Enable selection among more than two inputs.	This block provides a value for the chart input data `room_type`, where each room type corresponds to a number (1, 2, or 3). A Manual Switch block cannot support more than two inputs, but a Multiport Switch block can.
Display	Show up-to-date numerical value for input signal.	During simulation, any change to the chart output data `fee` appears in the display.

The Hotel chart contains graphical constructs, such as states and history junctions, and nongraphical constructs, such as conditions and condition actions.



For a mapping of constructs to their locations in the chart, see “Common Graphical and Nongraphical Constructs” on page 3-4.

How the Chart Interacts with Simulink Blocks

Chart Initialization

When simulation starts, the chart wakes up and executes its default transitions because the **Execute (enter) Chart At Initialization** option is on (see “Execution of a Chart at Initialization” on page 3-49). Then the chart goes to sleep.

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

Chart Interaction with Other Blocks

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

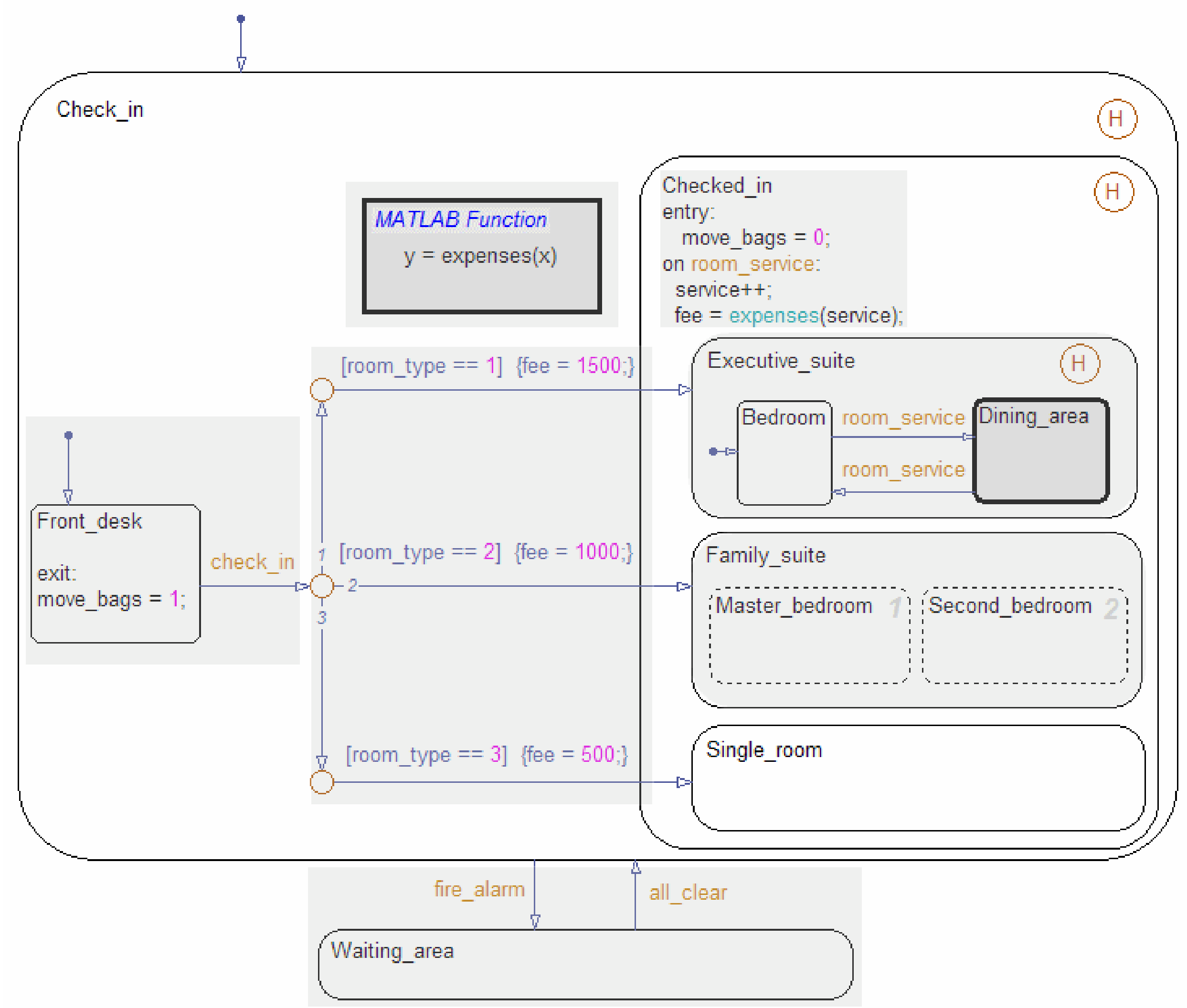
- The Multiport Switch block provides a value for the chart input data `room_type`.
- The Display block shows any change in value for the chart output data `fee`.

Chart Inactivity

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

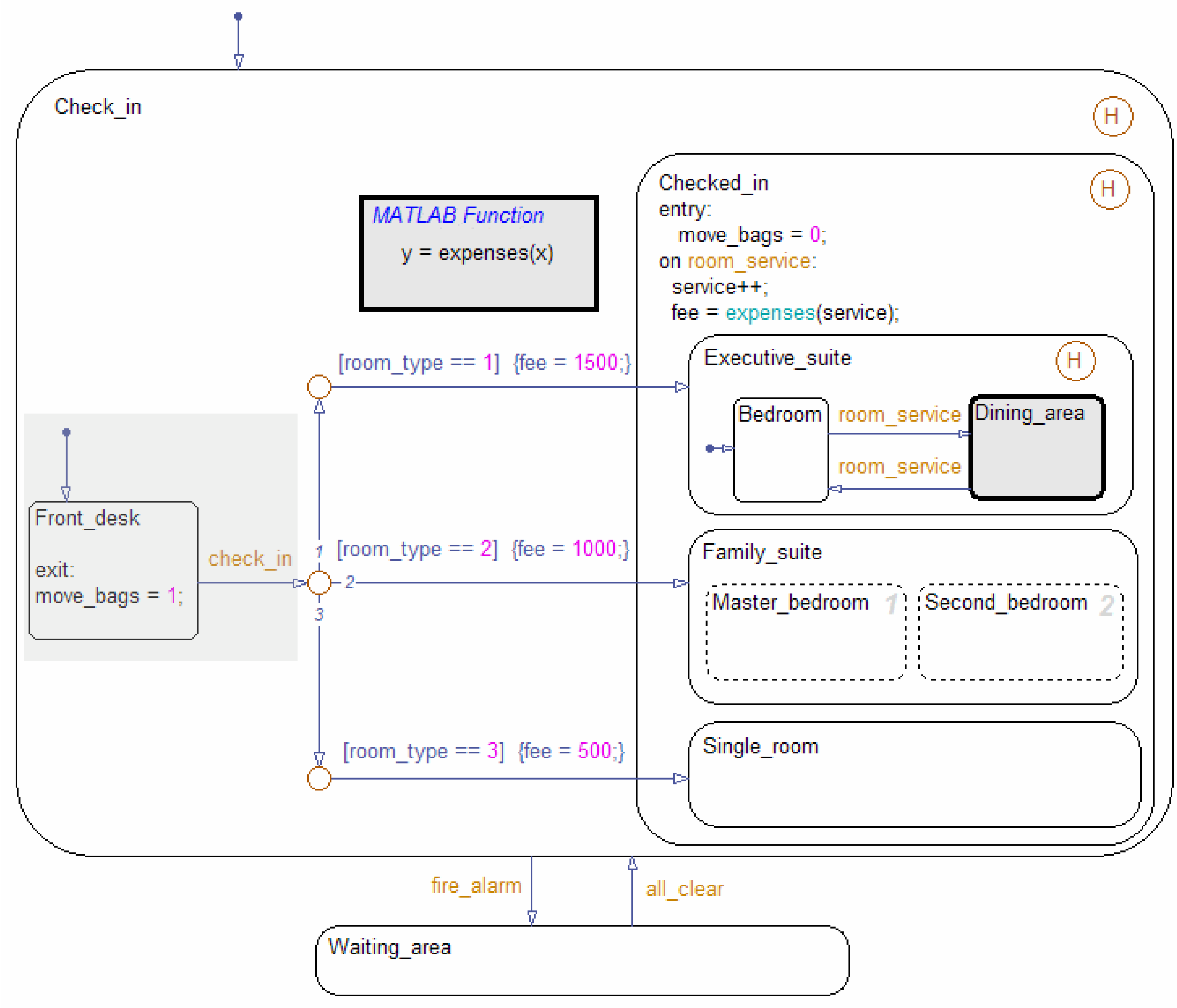
Phases of Chart Execution

The following sections explain chart execution for each shaded region of the Hotel chart.



Phase: Chart Initialization

This section describes what happens in the `Front_desk` state just after the chart wakes up.



Stage	Hotel Scenario	Chart Behavior
1	Your first stop is at the front desk of the hotel.	At the chart level, the default transition to `Check_in` occurs, making that state active. Then, the default transition to `Front_desk` occurs, making that state active. For reference, see "Steps for Entering a State" on page 3-70.
2	You leave the front desk after checking in to the hotel.	The `check_in` event guards the outgoing transition from `Front_desk`. When the chart receives an event broadcast for `check_in`, the transition becomes valid. For reference, see "How Charts Process Events" on page 3-38.
3	Just before you leave the front desk, you pick up your bags to move to your room.	Just before the transition occurs, the `exit` action of `Front_desk` sets the `move_bags` local data to 1. Then, `Front_desk` becomes inactive. For reference, see "Steps for Exiting an Active State" on page 3-72.

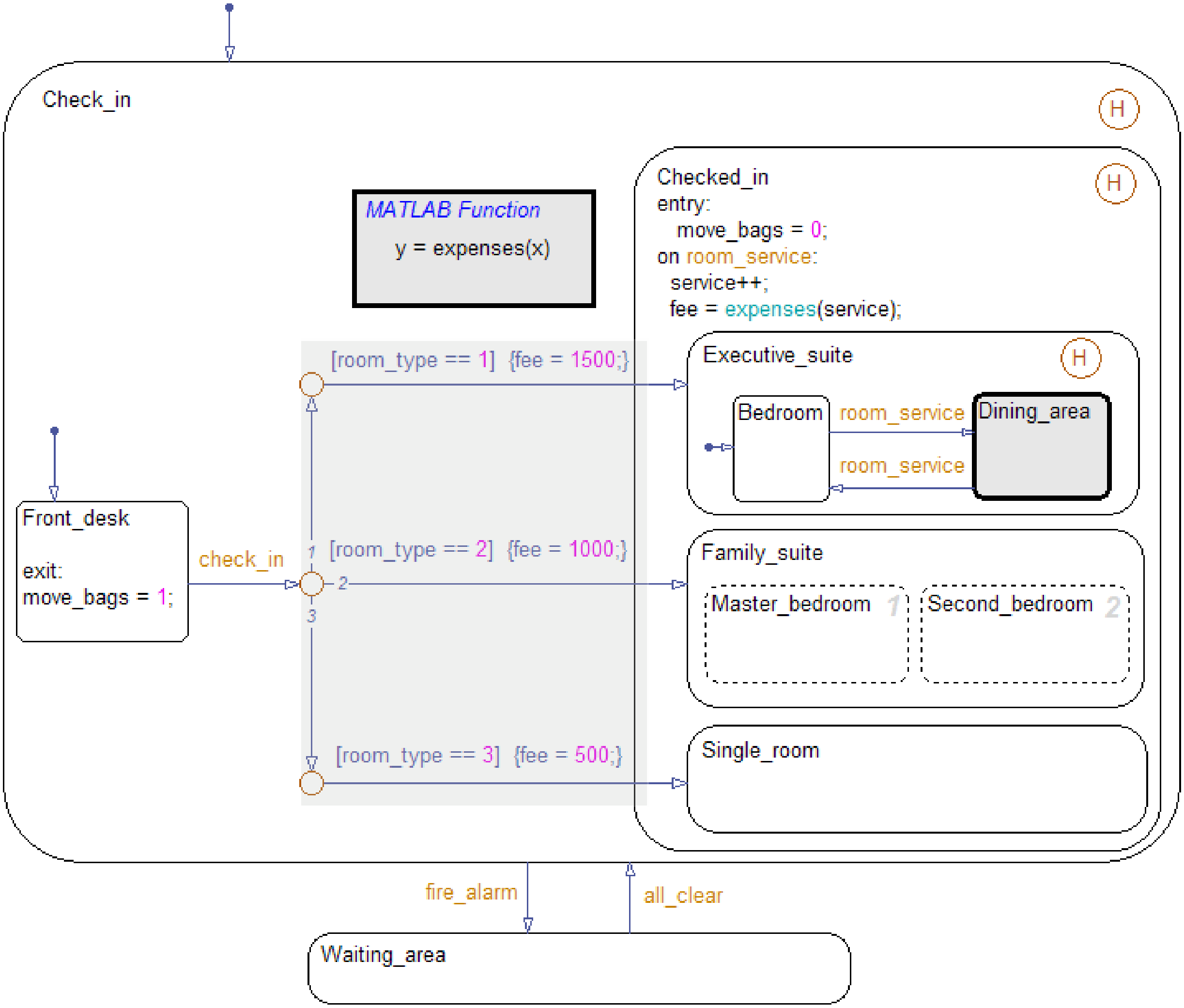
Modeling Guidelines for Chart Initialization. The following guidelines apply to chart initialization.

Modeling Guideline	Why This Guideline Applies	Reference
Use exclusive (OR) decomposition when no two states at a level of the hierarchy can be active at the same time.	This guideline ensures proper chart execution. For example, `Check_in` and `Waiting_area` are exclusive (OR) states, because you cannot be inside and outside the hotel at the same time.	• "State Decomposition" on page 2-10 • "Specifying Substate Decomposition" on page 4-14
Use a default transition to mark the first state to become	This guideline prevents state inconsistency errors during	• "Default Transitions" on page 2-31

Modeling Guideline	Why This Guideline Applies	Reference
active among exclusive (OR) states.	chart execution.	• "State Inconsistencies in a Chart" on page 26-27
Use events, instead of conditions, to guard transitions that depend on occurrences without inherent numerical value.	Since you cannot easily quantify the numerical value of checking into a hotel, model such an occurrence as an event.	• "Using Input Events to Activate a Stateflow Chart" on page 9-11
Use an `exit` action to execute a statement once, just before a state becomes inactive.	Other types of state actions execute differently and do not apply: • `Entry` actions execute once, just after a state becomes active. • `During` actions execute at every time step (except the first time step after a state becomes active). Execution continues as long as the chart remains in that state and no valid outgoing transitions exist. • `On` *`event_name`* actions execute only after receiving an event broadcast.	• "State Action Types" on page 10-2

Phase: Evaluation of Outgoing Transitions from a Single Junction

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



Stage	Hotel Scenario	Chart Behavior
1	You can move to one of three types of rooms.	After the `check_in` event triggers a transition out of `Front_desk`, three transition paths are available based on the type of room you select with the Multiport Switch block. Transition testing occurs based on the priority you assign to each path. For reference, see "Order of Execution for a Set of Flow Graphs" on page 3-52.
2	If you choose an executive suite, the base fee is 1500.	If the `room_type` input data equals 1, the top transition is valid. If this condition is true, the condition action executes by setting the `fee` output data to 1500. **Note** If the top transition is not valid, control flow backtracks to the central junction so that testing of the next transition can occur. This type of backtracking is intentional. To learn about *unintentional* backtracking and how to avoid it, see "Backtracking Behavior in Flow Graphs Example" on page B-46 and "Best Practices for Creating Flow Graphs" on page 5-30.
3	If you choose a family suite, the base fee is 1000.	If `room_type` equals 2, the middle transition is valid. If this condition is true, the condition action executes by setting `fee` to 1000.
4	If you choose a single room, the base fee is 500.	If `room_type` equals 3, the bottom transition is valid. If this condition is true, the condition action executes by setting `fee` to 500.

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

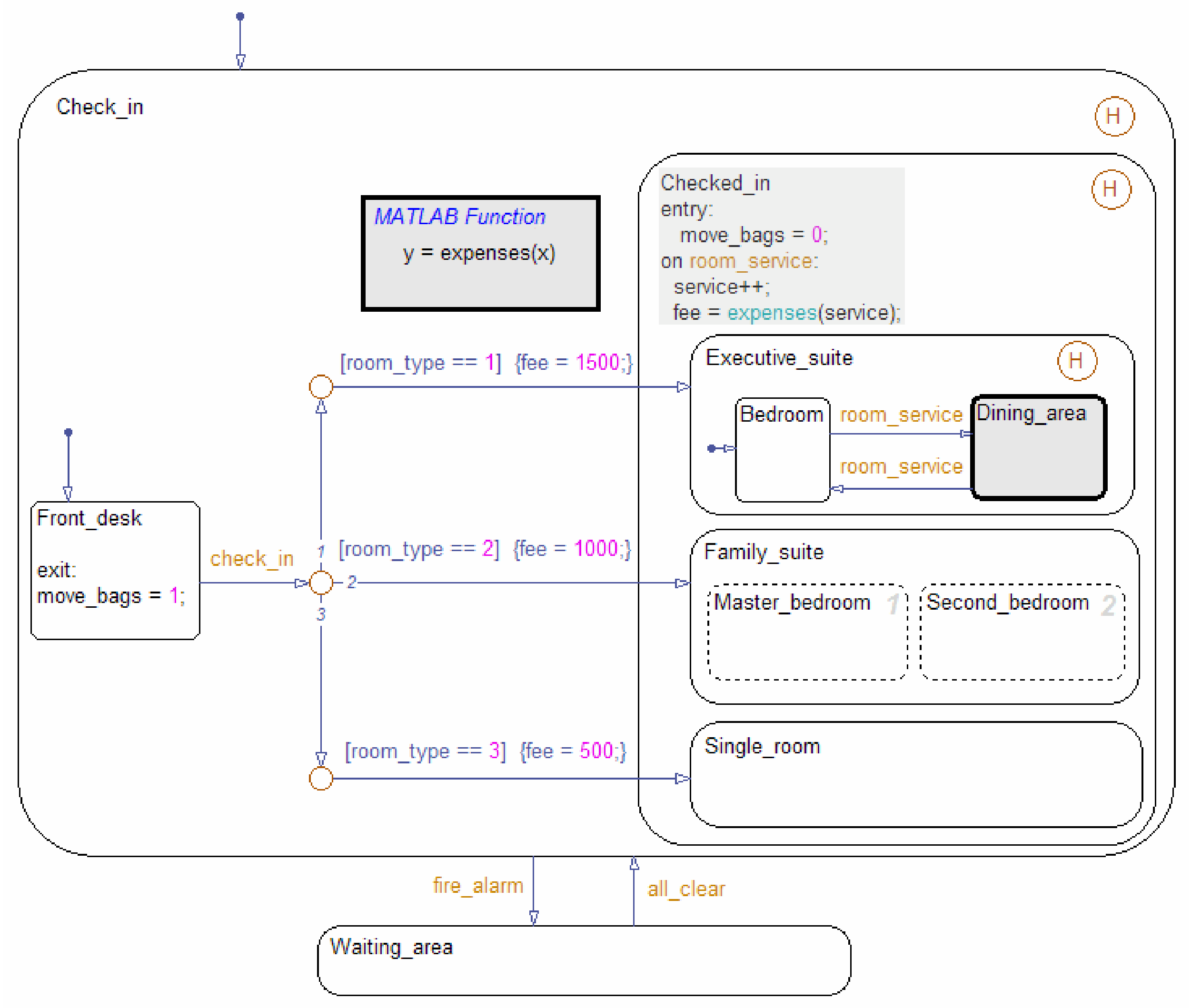
Because the Multiport Switch block outputs only 1, 2, or 3, `room_type` cannot have any other values. However, if `room_type` has a value other than 1, 2, or 3, the chart stays in the `Front_desk` state. This behavior applies because no transition path out of that state is valid.

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

Modeling Guideline	Why This Guideline Applies	Reference
Use conditions, instead of events, to guard transitions that depend on occurrences with numerical value.	Because you can quantify a type of hotel room numerically, express the choice of room type as a condition.	Chapter 5, "Modeling Logic Patterns and Iterative Loops Using Flow Graphs"
Use condition actions instead of transition actions whenever possible.	Condition actions execute as soon as the condition evaluates to true. Transition actions do not execute until after the transition path is complete, to a terminating junction or a state. Unless an execution delay is necessary, use condition actions instead of transition actions.	"Supported Action Types for States and Transitions" on page 10-2
Use explicit ordering to control the testing order of a group of outgoing transitions.	You can specify *explicit* or *implicit* ordering of transitions. By default, a chart uses explicit ordering. If you switch to implicit ordering, the transition testing order can change when graphical objects move.	"Evaluation Order for Outgoing Transitions" on page 3-55

Phase: Execution of State Actions for a Superstate

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



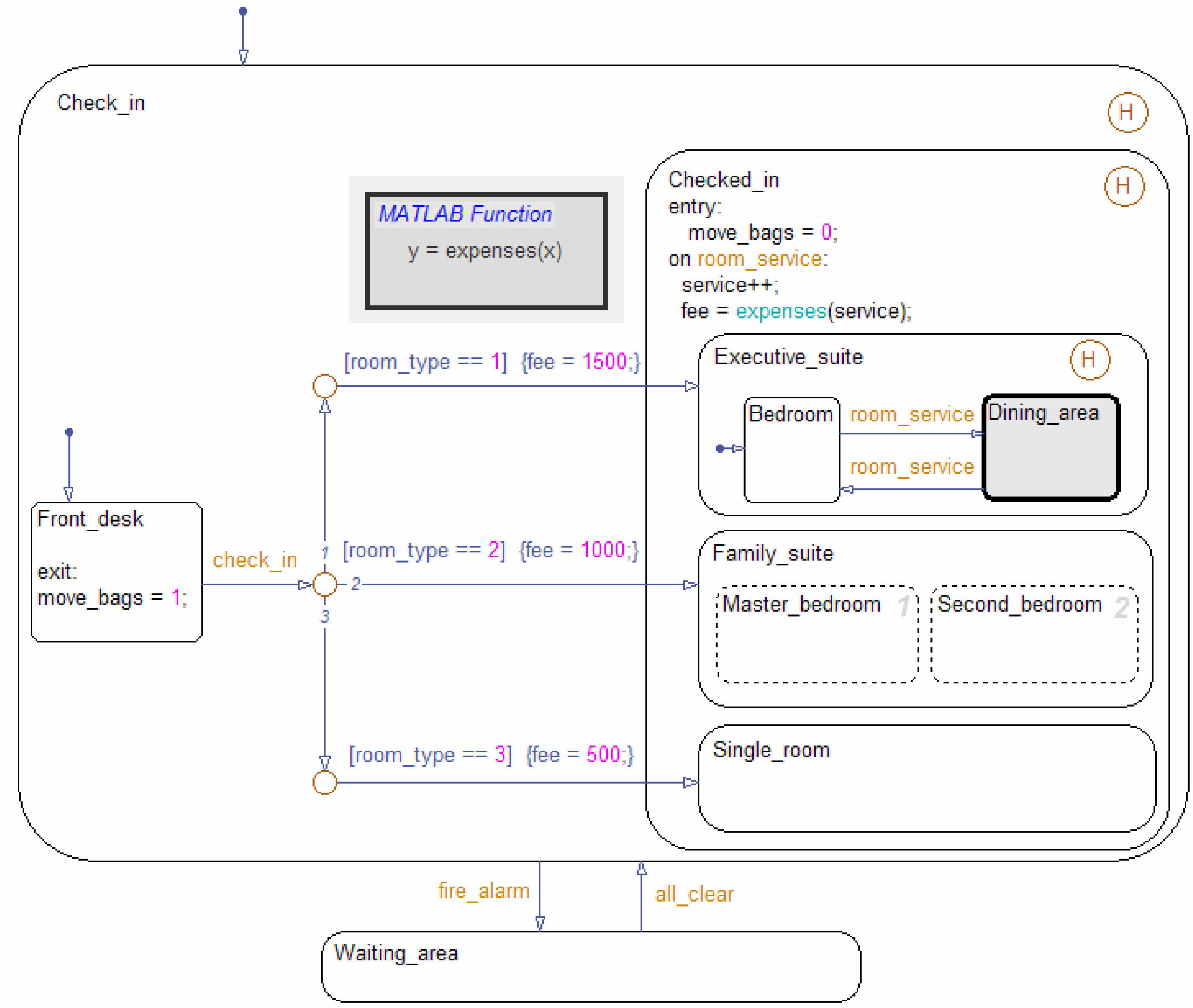
Stage	Hotel Scenario	Chart Behavior
1	After reaching your desired room, you finish moving your bags.	The `entry` action executes by setting the `move_bags` local data to 0.
2	If you order room service, your hotel bill increases by a constant amount.	If the chart receives an event broadcast for `room_service`, these actions occur: **1** The counter for the `service` local data increments by 1. **2** A function call to `expenses` occurs, which returns the value of the hotel bill stored by the `fee` output data. For reference, see "How Charts Process Events" on page 3-38.

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

Modeling Guideline	Why This Guideline Applies	Reference
Use an `entry` action to execute a statement once, right after a state becomes active.	Other types of state actions execute differently and do not apply: • `During` actions execute at every time step until there is a valid transition out of the state. • `Exit` actions execute once, just before a state becomes inactive.	"State Action Types" on page 10-2
Use an `On` *`event_name`* action to execute a statement only after receiving an event broadcast.		
Use a superstate to enclose multiple substates that share the same state actions.	This guideline enables reuse of state actions that apply to multiple substates. You write the state actions only *once*, instead of writing them separately in each substate.	"Creating Substates and Superstates" on page 4-11

Phase: Function Call from a State Action

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



Stage	Hotel Scenario	Chart Behavior
1	Based on your room type and the total number of room service requests, you can track your hotel bill.	`expenses` is a MATLAB function that takes the total number of room service requests as an input and returns the current hotel bill as an output. If you double-click the function box, you see this script in the function editor:

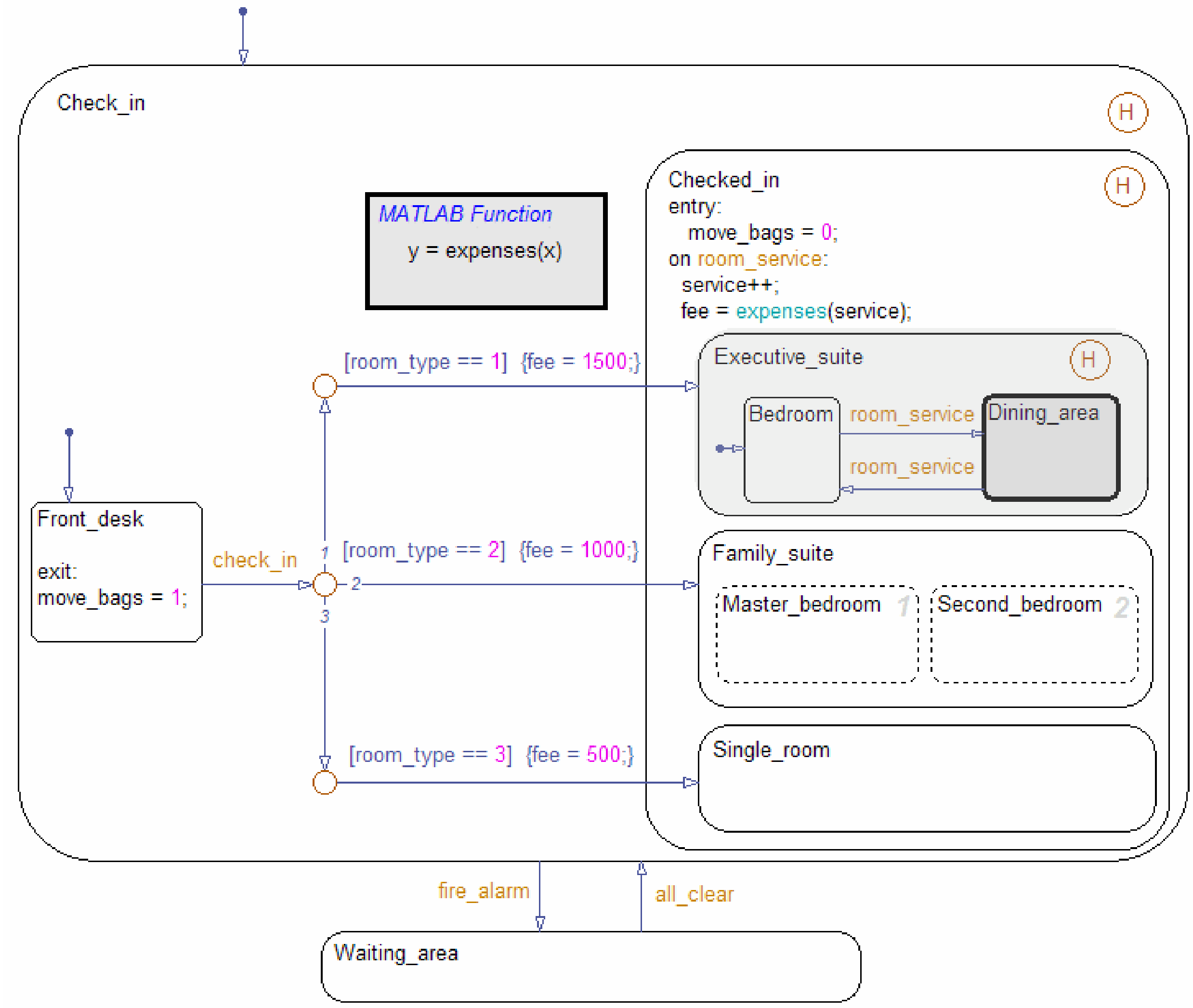
```
function y = expenses(x)

if (room_type == 1)
   y = 1500 + (x*50);
else
   if (room_type == 2)
      y = 1000 + (x*25);
   else
      y = 500 + (x*5);
   end
end
```

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

Modeling Guideline	Why This Guideline Applies	Reference
Use MATLAB functions for performing numerical computations in a chart.	MATLAB functions are better at handling numerical computations than graphical functions, truth tables, or Simulink functions.	Chapter 23, "Using MATLAB Functions in Stateflow Charts"
Use descriptive names in function signatures.	Descriptive function names enhance readability of chart objects.	

Phase: Execution of State with Exclusive Substates

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



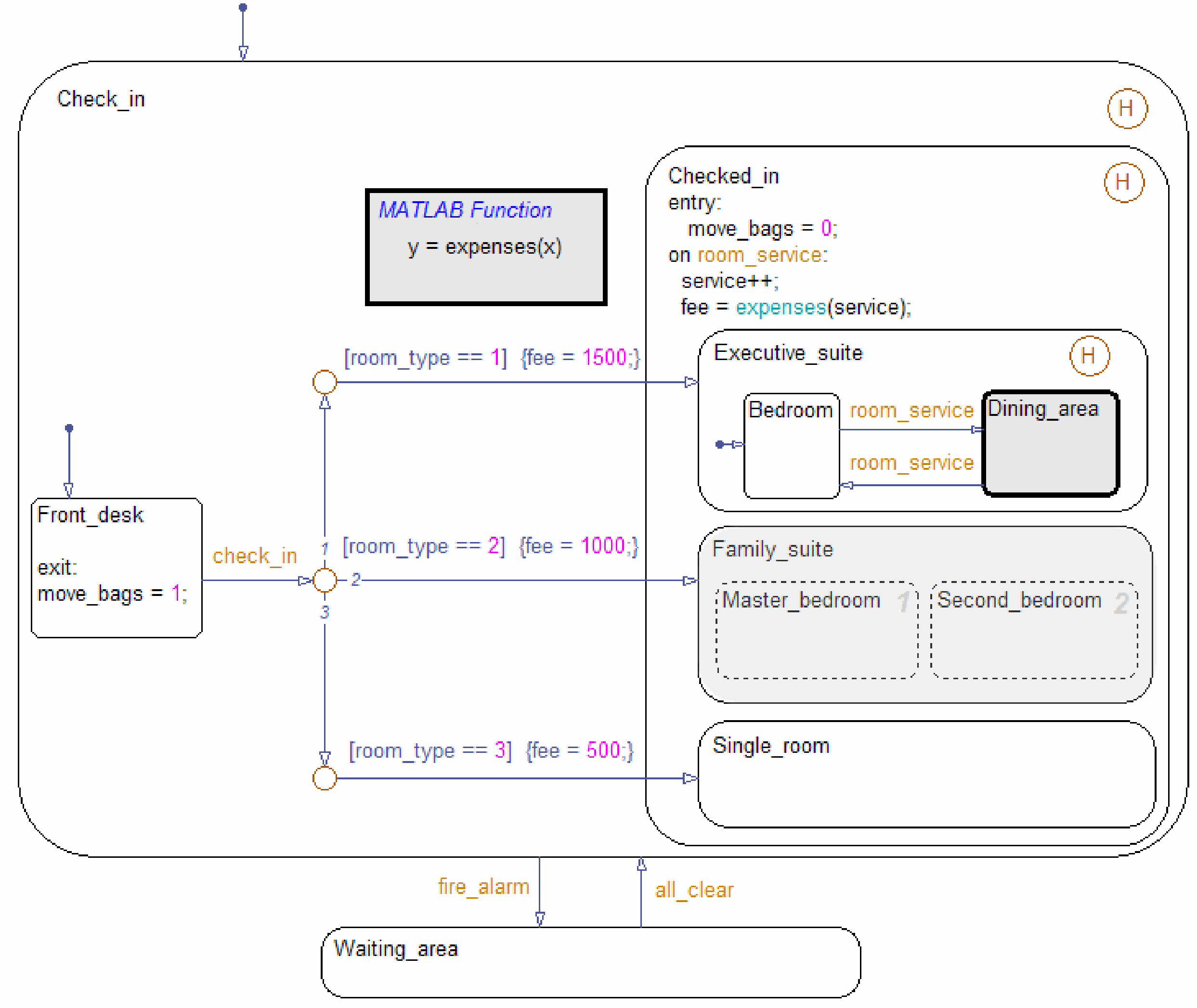
Stage	Hotel Scenario	Chart Behavior
1	When you reach the executive suite, you enter the bedroom first. **Note** The executive suite has separate bedroom and dining areas. Therefore, you can be in only one area of the suite at any time.	When the condition `room_type == 1` is true, the condition action `fee = 1500` executes. Completion of that transition path triggers these state initialization actions: **1** `Checked_in` becomes active and executes its entry action. **2** `Executive_suite` becomes active. **3** The default transition to `Bedroom` occurs, making that state active. For reference, see "Steps for Entering a State" on page 3-70.
2	When you order room service, you enter the dining area to eat.	When the `room_service` event occurs, the transition from `Bedroom` to `Dining_area` occurs.
3	When you want the food removed from the dining area, you order room service again and then return to the bedroom.	When the `room_service` event occurs, the transition from `Dining_area` to `Bedroom` occurs.
4	If you leave the executive suite because of a fire alarm, you return to your previous room after the all-clear signal.	If a transition out of `Executive_suite` occurs, the history junction records the last active substate, `Bedroom` or `Dining_area`. For details on how this transition can occur, see "Phase: Events Guard Transitions Between States" on page 3-31.

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

Modeling Guideline	Why This Guideline Applies	Reference
Use exclusive (OR) decomposition when no two states at that level of the hierarchy can be active at the same time.	This guideline ensures proper chart execution. For example, `Bedroom` and `Dining_area` are exclusive (OR) states, because you cannot be in both places at the same time.	• "State Decomposition" on page 2-10 • "Specifying Substate Decomposition" on page 4-14
If reentry to a state with exclusive (OR) decomposition depends on the previously active substate, use a history junction. This type of junction records the active substate when the chart exits the state.	If you do not record the previously active substate, the default transition occurs and the wrong substate can become active upon state reentry. For example, if you were eating when a fire alarm sounded, you would return to the bedroom instead of the dining room.	• "History Junctions" on page 2-43

Phase: Execution of State with Parallel Substates

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



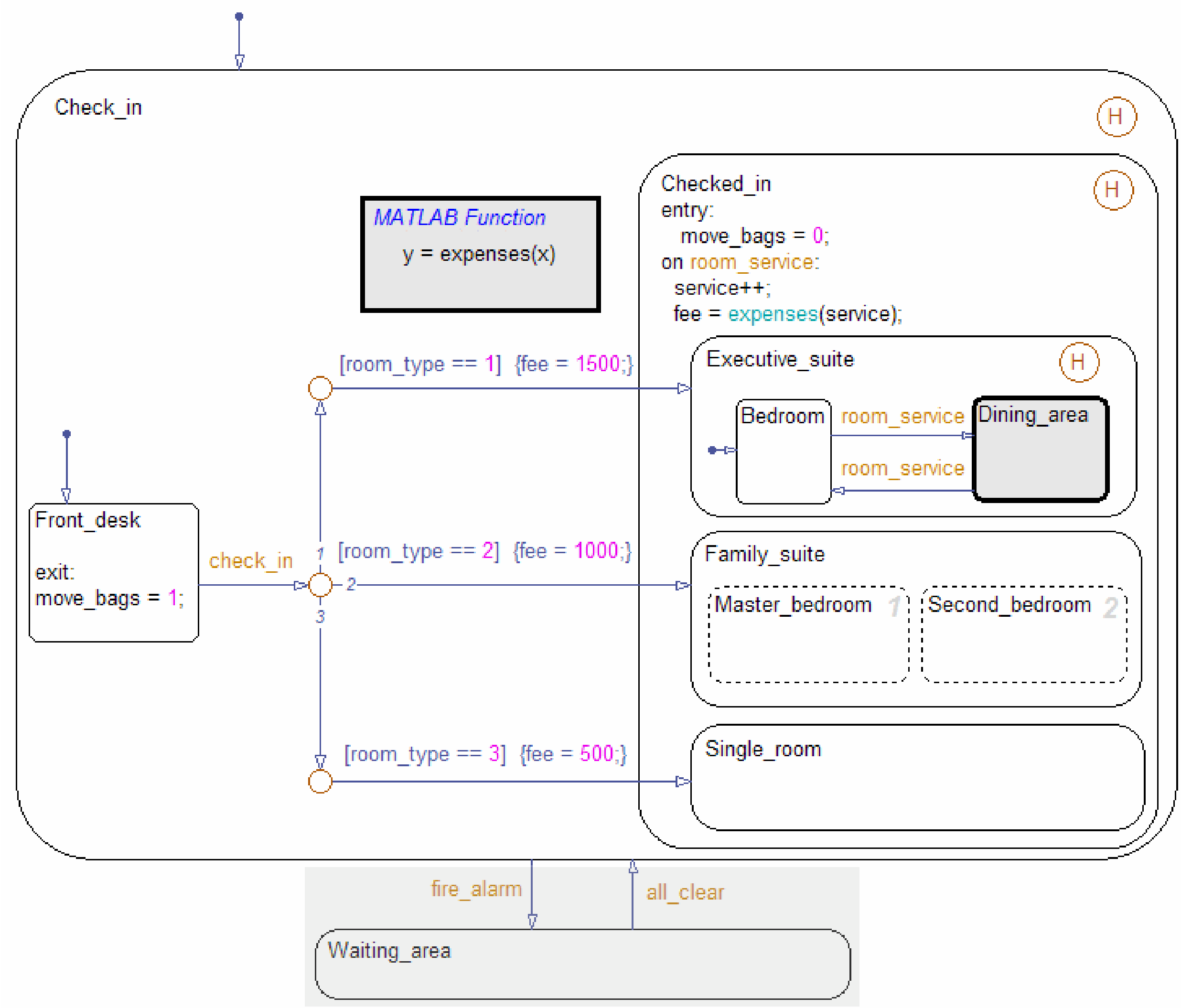
Stage	Hotel Scenario	Chart Behavior
1	When your family reaches the suite, family members can be in both bedrooms (for example, parents in the master bedroom and children in the second bedroom). A default room choice does not apply.	When the condition `room_type == 2` is true, the condition action `fee = 1000` executes. Completion of that transition path triggers these state initialization actions: **1** `Checked_in` becomes active and executes its entry action. **2** `Family_suite` becomes active. **3** The parallel states wake up in the order given by the number in the upper right corner of each state: `Master_bedroom`, then `Second_bedroom`. **How do I specify the order?** To specify the order: **a** Verify that the chart uses explicit ordering. In the Chart properties dialog box, select the **User specified state/transition execution order** check box. **b** Right-click in a parallel state and select a number from the **Execution Order** menu. For reference, see “Steps for Entering a State” on page 3-70.
2	You can occupy both rooms at the same time.	`Master_bedroom` and `Second_bedroom` remain active at the same time.

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

Modeling Guideline	**Why This Guideline Applies**	**Reference**
Use parallel (AND) decomposition when all states at that level of the hierarchy can be active at the same time.	This guideline ensures proper chart execution. For example, `Master_bedroom` and `Second_bedroom` are parallel states, because you can occupy both rooms at the same time.	• "State Decomposition" on page 2-10 • "Specifying Substate Decomposition" on page 4-14
Use *no* history junctions in states with parallel (AND) decomposition.	This guideline prevents parsing errors. Since all parallel states at a level of hierarchy are active at the same time, history junctions have no meaning.	• "History Junctions" on page 2-43
Use explicit ordering to control the execution order of parallel (AND) states.	You can specify *explicit* or *implicit* ordering of parallel states. By default, a chart uses explicit ordering. If you switch to implicit ordering, the execution order can change when parallel states move.	• "Execution Order for Parallel States" on page 3-75

Phase: Events Guard Transitions Between States

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



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

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

Modeling Guideline	Why This Guideline Applies	Reference
Use events, instead of conditions, to guard transitions that depend on occurrences without numerical value.	Because you cannot easily quantify the numerical value of a fire alarm or an all-clear signal, model such an occurrence as an event.	"Using Input Events to Activate a Stateflow Chart" on page 9-11

Modeling Guidelines for Stateflow Charts

These guidelines promote efficient modeling of charts with events, states, and transitions.

Use signals of the same data type for input events

When you use multiple input events to trigger a chart, verify that all input signals use the same data type. Otherwise, simulation stops and an error message appears. For more information, see "Data Types Allowed for Input Events" on page 9-14.

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

This guideline prevents state inconsistency errors during chart execution.

Use condition actions instead of transition actions whenever possible

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

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

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

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

Verify intended backtracking behavior in flow graphs

If your chart contains unintended backtracking behavior, a warning message appears with instructions on how to avoid that problem. For more information, see "Best Practices for Creating Flow Graphs" on page 5-30.

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

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

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

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

Use MATLAB functions for performing numerical computations in a chart

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

Use descriptive names in function signatures

Descriptive function names enhance readability of chart objects.

Use history junctions to record state history

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

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

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

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

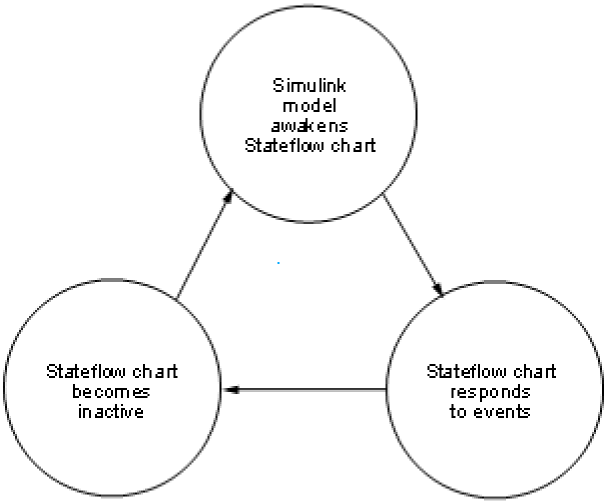
You can specify *explicit* or *implicit* ordering of parallel states. By default, a chart uses explicit ordering. If you switch to implicit ordering, the execution order can change when parallel states move.

How Events Drive Chart Execution

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. All activity caused by the event in the chart finishes before execution returns to the activity that was taking place before receiving the event. Once an event initiates an action, the action completes unless interrupted by an early return.

Sources for Stateflow Events

Simulink events awaken Stateflow charts. You can use events to control the processing of your charts by broadcasting events in the action language, as described in "Broadcasting Events to Synchronize States" on page 10-59. For examples using event broadcasting and directed event broadcasting, see:

- "Condition Actions to Broadcast Events to Parallel (AND) States Example" on page B-16
- "Cyclic Behavior to Avoid with Condition Actions Example" on page B-17
- "Event Broadcast State Action Example" on page B-50
- "Event Broadcast Transition Action with a Nested Event Broadcast Example" on page B-53
- "Event Broadcast Condition Action Example" on page B-56
- Directed Event Broadcasting

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

How Charts Process Events

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

1. Executes `during` and `on` *`event_name`* actions for the active state
2. Checks for valid transitions in substates

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

1. If the *receiver* of the event is active, then it executes (see "Execution of an Active Chart" on page 3-41 and "Steps for Executing an Active State" on page 3-71). (The event *receiver* is the parent of the event unless a directed event broadcast occurs using the `send()` function.)

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

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

   To learn about early return logic, see “Early Return Logic for Event Broadcasts” on page 3-85.

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 19-19.

Types of Chart Execution

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 "How Events Drive Chart Execution" on page 3-37.

Execution of an Inactive Chart

When a chart is inactive and first triggered by an event from a Simulink model, it first executes its set of default flow graphs (see "Order of Execution for a Set of Flow Graphs" on page 3-52). If this action does not cause an entry into a state and the chart has parallel decomposition, then each parallel state becomes active (see "Steps for Entering a State" on page 3-70).

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

Execution of an Active Chart

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

Execution of a Chart with Super Step Semantics

What Is Super Step Semantics?

By default, Stateflow charts execute once for each active input event. If no input events exist, 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 3-42).

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

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

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

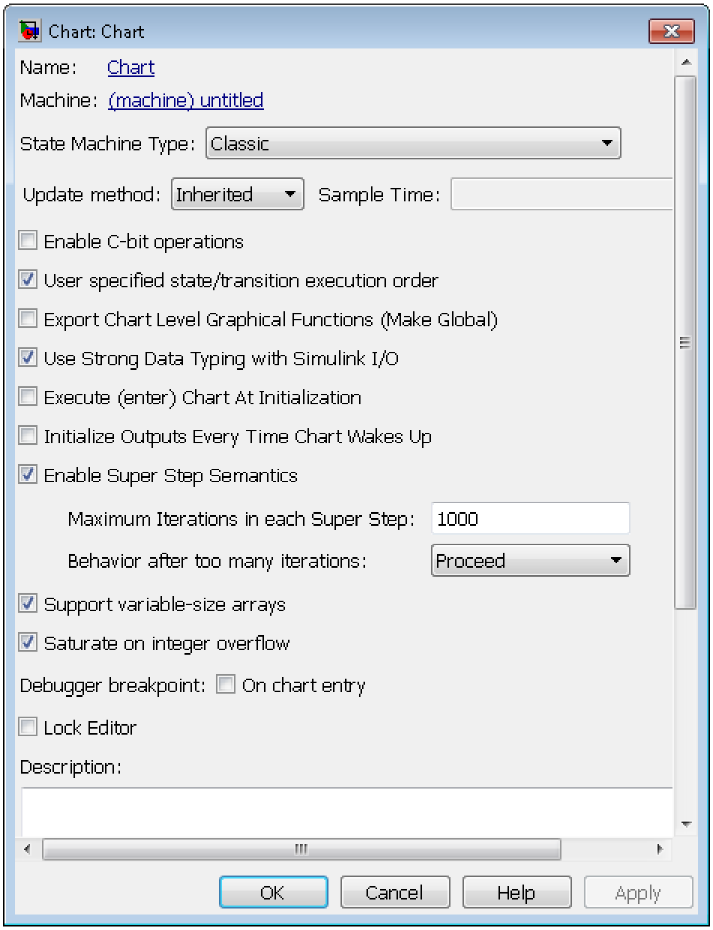
Enabling Super Step Semantics

To enable super step semantics:

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

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

Two additional fields appear below that check box.



Name: Chart

Machine: (machine) untitled

State Machine Type: Classic

Update method: Inherited   Sample Time:

☐ Enable C-bit operations

☑ 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

Maximum Iterations in each Super Step: 1000

Behavior after too many iterations: Proceed

☑ Support variable-size arrays

☑ Saturate on integer overflow

Debugger breakpoint: ☐ On chart entry

☐ Lock Editor

Description:

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

This value is the maximum number of transitions a 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**.

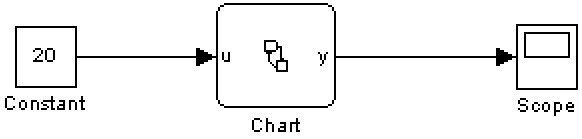
Your selection determines how the chart behaves during simulation if it exceeds the maximum number of iterations in the super step before reaching a stable state.

Behavior	Description
Proceed	The chart goes back to sleep with the last active state configuration, that is, after updating local data at the last valid transition in the super step.
Throw Error	Simulation stops and the chart generates an error, indicating that too many iterations occurred while trying to reach a stable state. **Note** Selecting **Throw Error** can help detect infinite loops in transition cycles (see "Detection of Infinite Loops in Transition Cycles" on page 3-48.

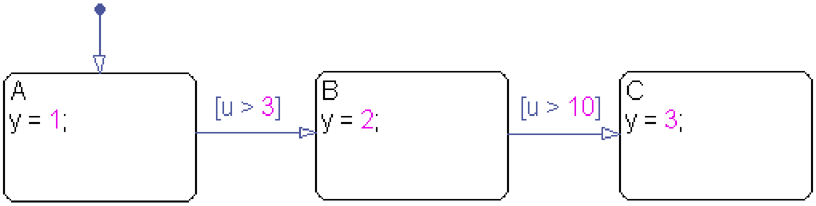
Note This option is relevant only for simulation targets. For embedded targets, code generation goes to the next time step rather than generating an error.

Super Step Example

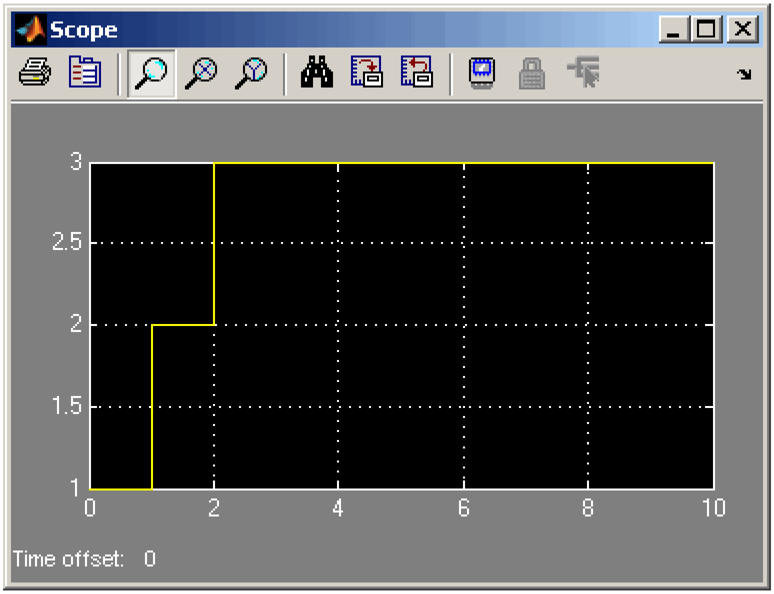
The following model shows how super step semantics differs from Classic Stateflow chart semantics:



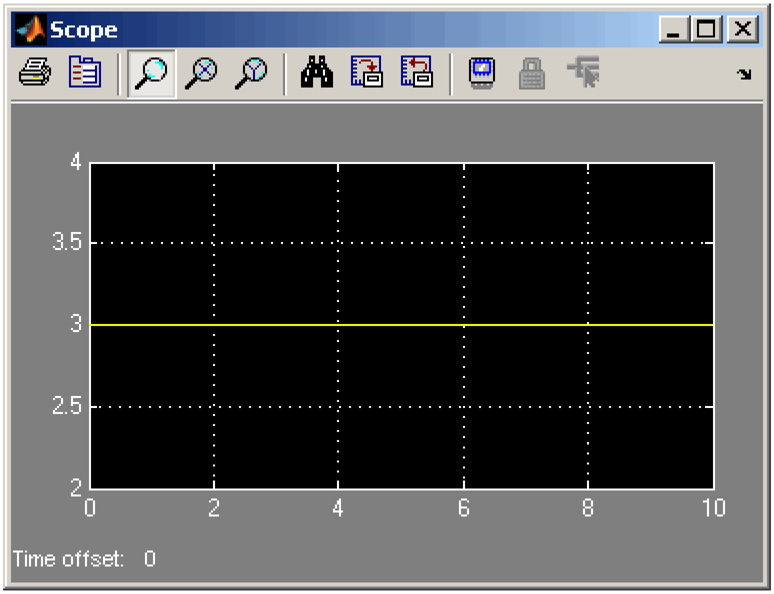
In this model, a Constant block outputs a constant value of 20 to input `u` in a Stateflow chart. Because the value of `u` is always 20, each transition in the chart is valid:



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

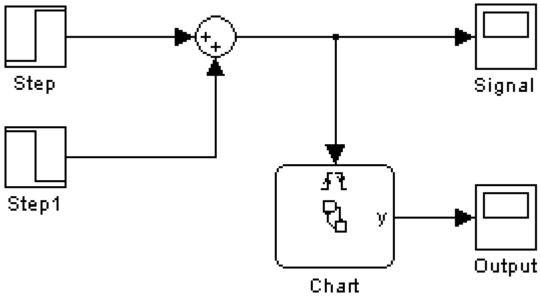


When you enable super step semantics, the chart takes all valid transitions in each time step, stopping at state `C` with `y` = 3.

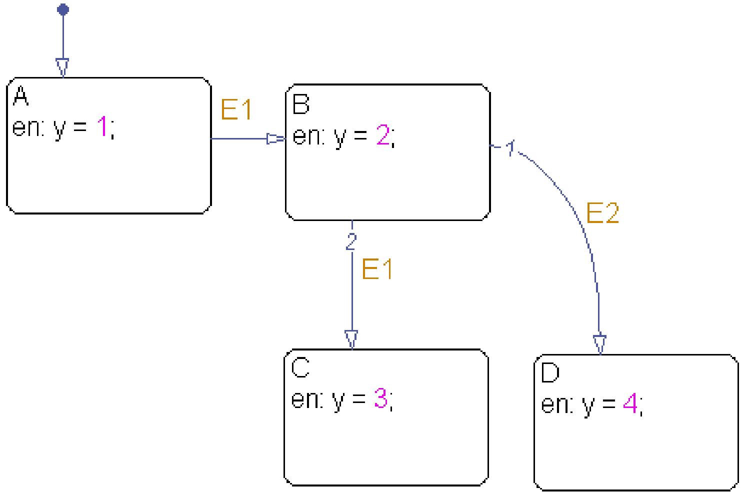


How Super Step Semantics Works with Multiple Input Events

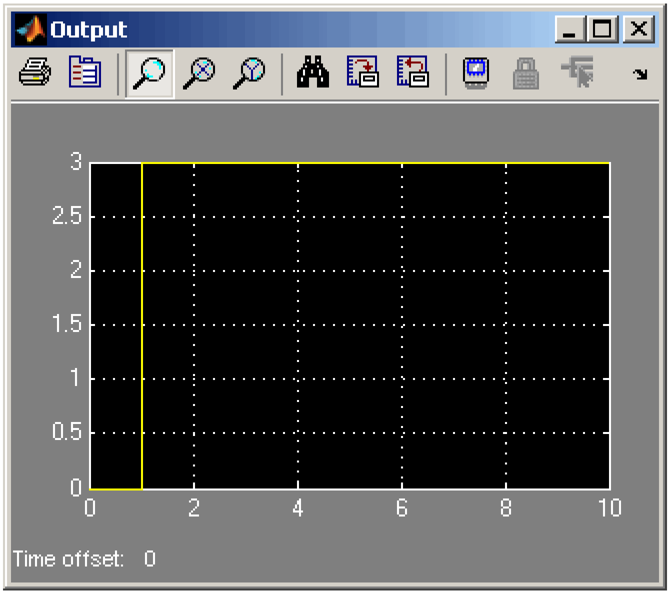
When you enable super step semantics for a chart with multiple active input events, the chart takes all valid transitions for the first active event before it begins processing the next active event. For example, consider the following model:



In this model, the Sum block produces a 2-by-1 vector signal that goes from [0,0] to [1,1] at time `t` = 1. As a result, when the model wakes up the chart, events `E1` and `E2` are both active:



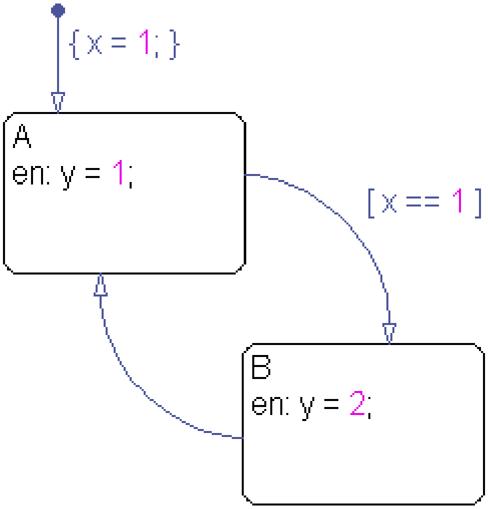
If you enable super step semantics, the chart takes all valid transitions for event `E1`. The chart takes transitions from state `A` to `B` and then from state `B` to `C` in a single super step. The scope shows that `y` = 3 at the end of the super step:



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

Detection of Infinite Loops in Transition Cycles

If your chart contains transition cycles, taking multiple transitions in a single time step can cause infinite loops. Consider the following example:



In this example, the transitions between states `A` and `B` cycle and produce an infinite loop because the value of `x` remains constant at 1. One way to detect infinite loops is to configure your chart to generate an error if it reaches a maximum number of iterations in a super step. See "Enabling Super Step Semantics" on page 3-42.

Execution of a Chart at Initialization

By default, the first time a chart wakes up, it executes the default transition paths. At this time, the chart can access inputs, write to outputs, and broadcast events. If you want your chart to begin executing from a known configuration, you can enable the option to *execute at initialization*. When you turn on this option, the state configuration of a chart initializes at time 0 instead of the first occurrence of an input event. The default transition paths of the chart execute during the model initialization phase at time 0, corresponding to the mdlInitializeConditions() phase for S-functions.

You select the **Execute (enter) Chart At Initialization** check box in the Chart properties dialog box, as described in "Setting Properties for a Single Chart" on page 19-4.

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

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

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

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

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

Process for Grouping and Executing Transitions

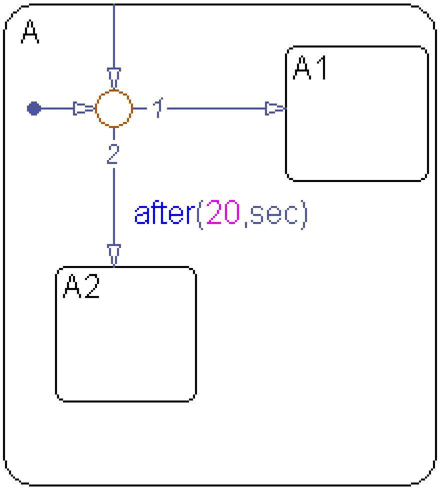
In this section...

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, the set of default flow graphs for state `A` includes:

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

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

Order of Execution for a Set of Flow Graphs

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

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

   An active state can have several possible outgoing transitions. The chart orders these transitions before checking them for validity. See "Evaluation Order for Outgoing Transitions" on page 3-55.

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

3. Test the transition segment for validity.

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

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

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

   b. The states that are the immediate children of the parent of the transition path exit (see "Steps for Exiting an Active State" on page 3-72).

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

   d. The destination state becomes active (see "Steps for Entering a State" on page 3-70).

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

   a. Testing stops without any state exits or entries.

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

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

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

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

The set of flow 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 3-56).

- Implicit ordering

  Override explicit ordering by letting a Stateflow chart use internal rules to order transitions (see "Implicit Ordering of Outgoing Transitions" on page 3-59).

Note You can order transitions only within their type (inner, outer, or default). For more information, see "Transition Flow Graph Types" on page 3-51.

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 in the same Simulink model.

Detection of Transition Shadowing

Transition shadowing occurs when a chart contains multiple unconditional transitions that originate from the same state or the same junction. To avoid transition shadowing, ensure that no more than one unconditional transition exists for each group of outgoing transitions from a state or junction.

You can control the behavior of the Stateflow diagnostic that detects transition shadowing. On the **Diagnostics > Stateflow** pane of the Configuration Parameters dialog box, set **Transition shadowing** to `none`, `warning`, or `error`. For information about other diagnostics, see "Diagnostics Pane: Stateflow" in the *Simulink Graphical User Interface* documentation.

Explicit Ordering of Outgoing Transitions

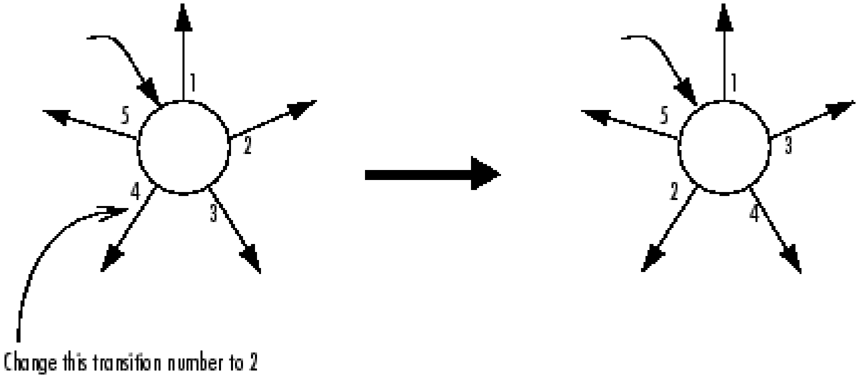
By default, a Stateflow chart orders outgoing transitions explicitly based on evaluation priorities you set.

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:

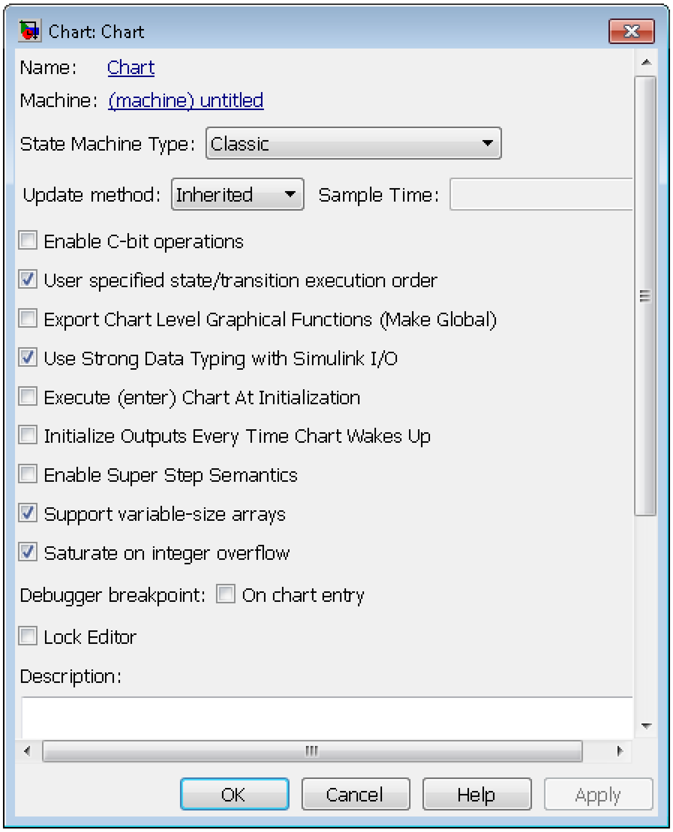
1. "Enabling Explicit Ordering at the Chart Level" on page 3-57
2. "Setting Evaluation Order for Transitions Individually" on page 3-58

Enabling Explicit Ordering at the Chart Level. To enable explicit ordering for transitions:

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

   The Chart properties dialog box appears.

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



Name: Chart

Machine: (machine) untitled

State Machine Type: Classic

Update method: Inherited Sample Time:

☐ Enable C-bit operations

☑ 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

☑ Support variable-size arrays

☑ Saturate on integer overflow

Debugger breakpoint: ☐ On chart entry

☐ Lock Editor

Description:

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 3-57.

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 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 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 3-60)

2. Label (see “Ordering by Label” on page 3-61)

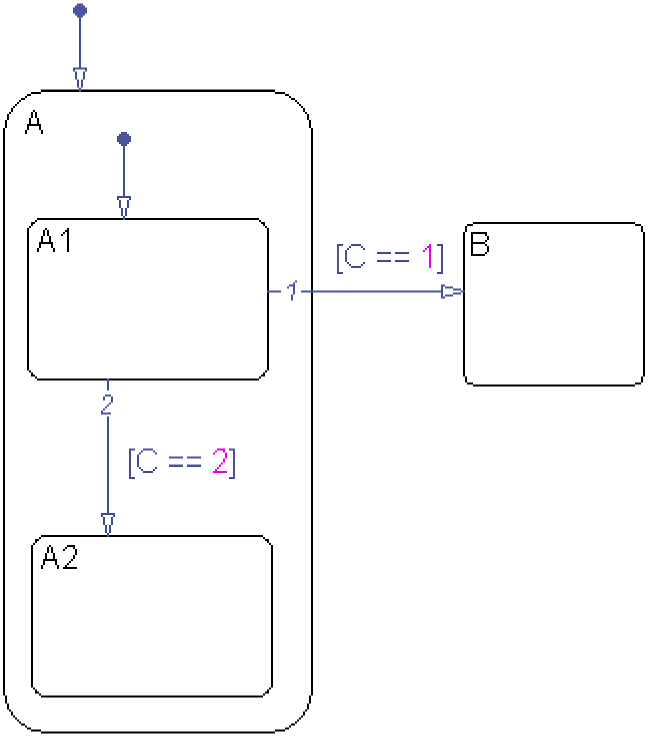
3. Angular surface position of transition source (see "Ordering by Angular Position of Source" on page 3-62)

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.

Because the chart is at a higher level in the 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:

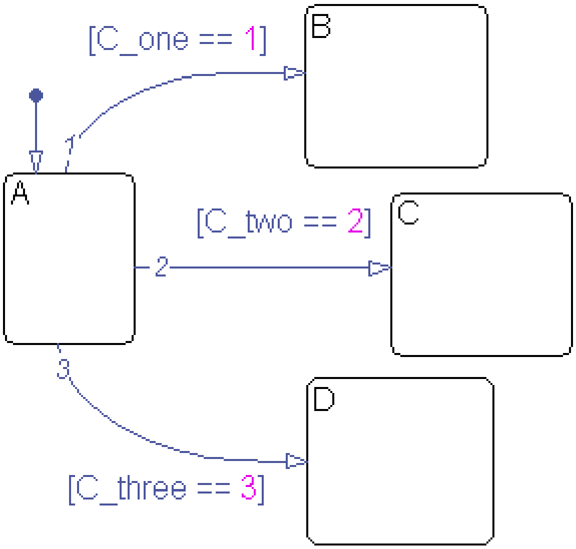
1. Labels with events and conditions
2. Labels with events
3. Labels with conditions
4. No label

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.

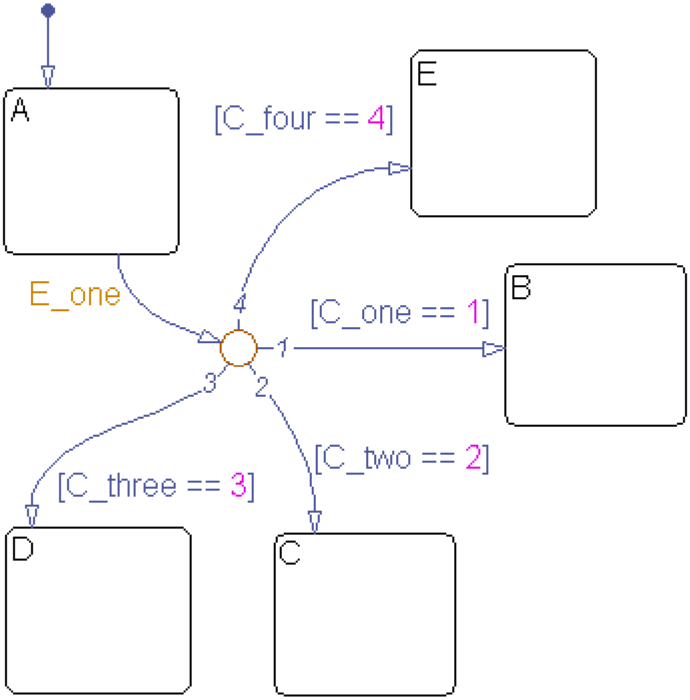


- For each outgoing transition from state `A`, the parent is the chart and the label contains a condition. Therefore, the outgoing transitions have equal hierarchical and label priority.
- The conditions [`C_one` == 1] and [`C_two` == 2] are false, and the condition [`C_three` == 3] is true.

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

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 Simulation Diagnostics Viewer displays warnings about the changes in transition evaluation order.

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

Transition Testing Order in Multilevel State Hierarchy

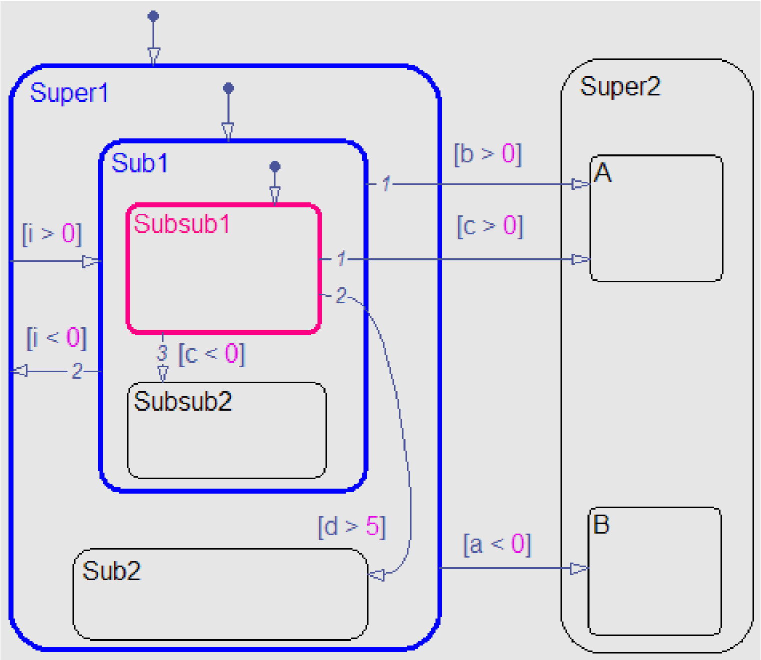
How Multilevel Transition Testing Order Works

By default, charts use explicit ordering for transitions. In this mode, you have explicit control over the testing priority, as described in "Explicit Ordering of Outgoing Transitions" on page 3-56.

If you use implicit ordering for transitions, the following testing order applies. For each group of transitions that originate from the same state, tiebreaking criteria apply in this order: hierarchy, label, and angular position.

Testing Order	Chart Action	Order by Hierarchy	Order by Label	Order by Angular Position
1	Tests transitions that originate from the highest-level active state (superstate).	**1** Outer transitions **2** Inner transitions	**1** Events and conditions **2** Events **3** Conditions **4** No label	The transition with the smallest clock position has the highest priority. A transition with a 12 o'clock source position has the lowest priority.
2	Tests transitions that originate from the next lower-level active state.	**1** Outer transitions that cross the border of the highest-level active state (superstate) **2** Outer transitions that stay within the parent of the state **3** Inner transitions		
3	Repeats step 2 until transition testing is complete.			

The following chart shows the behavior of multilevel transition testing. Assume that the `Super1.Sub1.Subsub1` state is active.

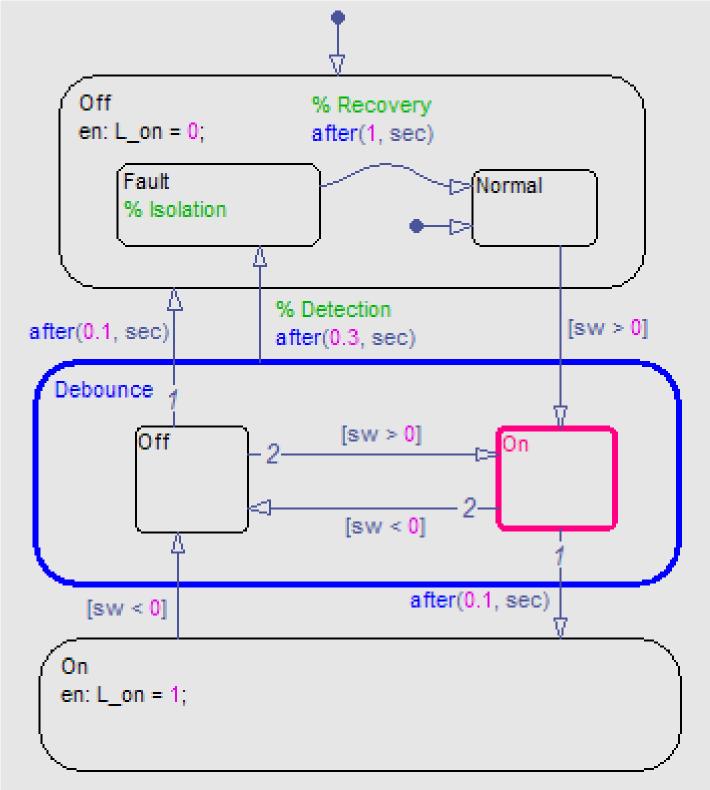


Because the chart uses implicit ordering, the following transition testing order applies:

This priority...	**Applies to the label...**	**For this transition...**
1	`[a < 0]`	`Super1` to `Super2.B`
2	`[i > 0]`	`Super1` to `Super1.Sub1`
3	`[b > 0]`	`Super1.Sub1` to `Super2.A`
4	`[i < 0]`	`Super1.Sub1` to `Super1`
5	`[c > 0]`	`Super1.Sub1.Subsub1` to `Super2.A`
6	`[d > 5]`	`Super1.Sub1.Subsub1` to `Super1.Sub2`
7	`[c < 0]`	`Super1.Sub1.Subsub1` to `Super1.Sub1.Subsub2`

Example Model with Multilevel Transition Testing

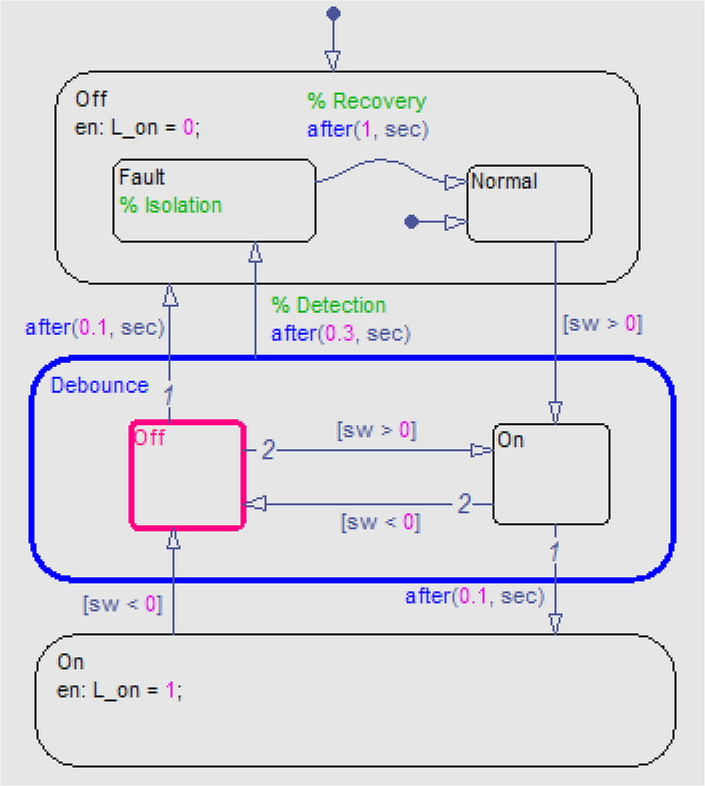
Suppose that you open the `sf_debouncer` model and reach the following point in the simulation.



Because the chart uses implicit ordering, the following transition testing order applies:

This priority...	Applies to the label...	For this transition...
1	`after(0.3, sec)`	`Debounce` to `Off.Fault`
2	`after(0.1, sec)`	`Debounce.On` to `On`
3	`[sw < 0]`	`Debounce.On` to `Debounce.Off`

Now suppose that the transition from `Debounce.On` to `Debounce.Off` occurs.



Because the chart uses implicit ordering, the following transition testing order applies:

This priority...	Applies to the label...	For this transition...
1	`after(0.3, sec)`	`Debounce` to `Off.Fault`
2	`after(0.1, sec)`	`Debounce.Off` to `Off`
3	`[sw > 0]`	`Debounce.Off` to `Debounce.On`

For more information on how this model works, see "Key Behaviors of Debouncer Chart" on page 21-4.

Process for Entering, Executing, and Exiting States

In this section…

Steps for Entering a State

A state becomes active in one of these ways:

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

A state performs its `entry` action (if specified) when it becomes active. The state becomes active before its `entry` action executes and completes.

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 3-76 and "Implicit Ordering of Parallel States" on page 3-77.

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 is an active child of this state at some point after the most recent chart initialization, perform the entry actions for that child. Otherwise, execute the default flow paths for the state.
   b. If this state has children that are parallel states (parallel decomposition), perform entry steps 1 through 5 for each state according to its entry order.
   c. If this state has only one child substate, the substate becomes active when the parent becomes active, regardless of whether a default transition is present. Entering the parent state automatically makes the substate active. The presence of any inner transition has no effect on determining the active substate.

6. If 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.

Steps for Executing an Active State

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

1. Execute the set of outer flow graphs (see “Order of Execution for a Set of Flow Graphs” on page 3-52).

   If this action causes a state transition, execution stops.

   Note This step never occurs 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 execute, starting at step 1. Parallel states execute in the same order that they become active.

Steps for Exiting an Active State

A state becomes inactive in one of these 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 "Steps for Entering a State" on page 3-70.

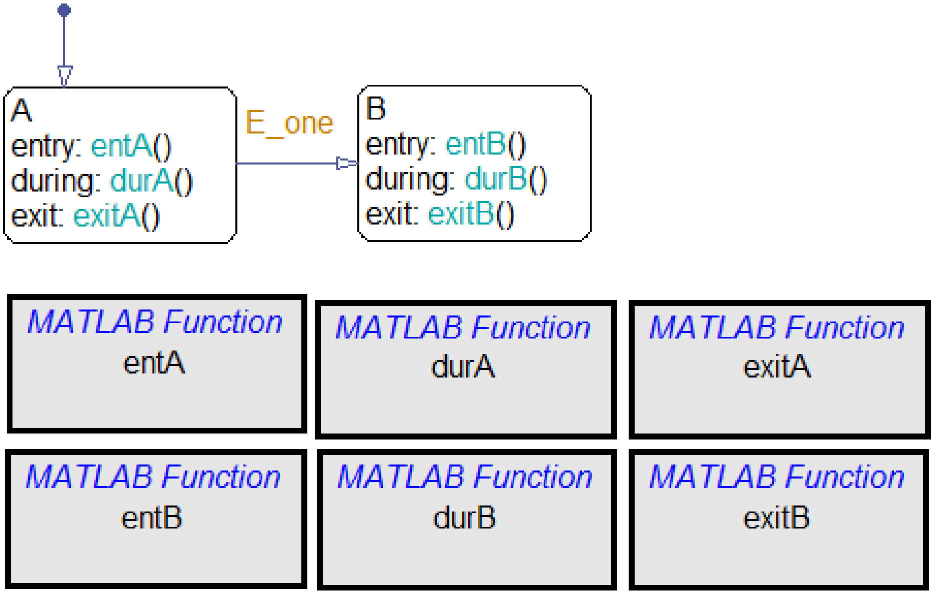
2. If a state has active children, performs the exit actions of the child states in the reverse order from when they became active.

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 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` exists.
3. State `A` becomes 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 chart wakes up.

   State `A` is active from the preceding steps 1 through 5.

2. The chart root checks to see if there is a valid transition as a result of `E_one`. A valid transition from state `A` to state `B` exists.

3. State `A` exit actions (`exitA()`) execute and complete.

4. State `A` becomes inactive.

5. State `B` becomes active.

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

   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 “Process for Entering, Executing, and Exiting States” on page 3-70.

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 3-76).

- 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 3-77).

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 3-77. 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 3-79.

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 3-76

2. "Setting Execution Order for Parallel States Individually" on page 3-77

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

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

   The Chart properties dialog box appears.

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

3. Click **OK**.

   If your chart already contains parallel states that have been ordered implicitly, the existing priorities are preserved until you explicitly change them. When you add new parallel states in explicit mode, your chart automatically assigns priorities based on order of creation (see "How Explicit Ordering Works" on page 3-76). 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 3-77.

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.

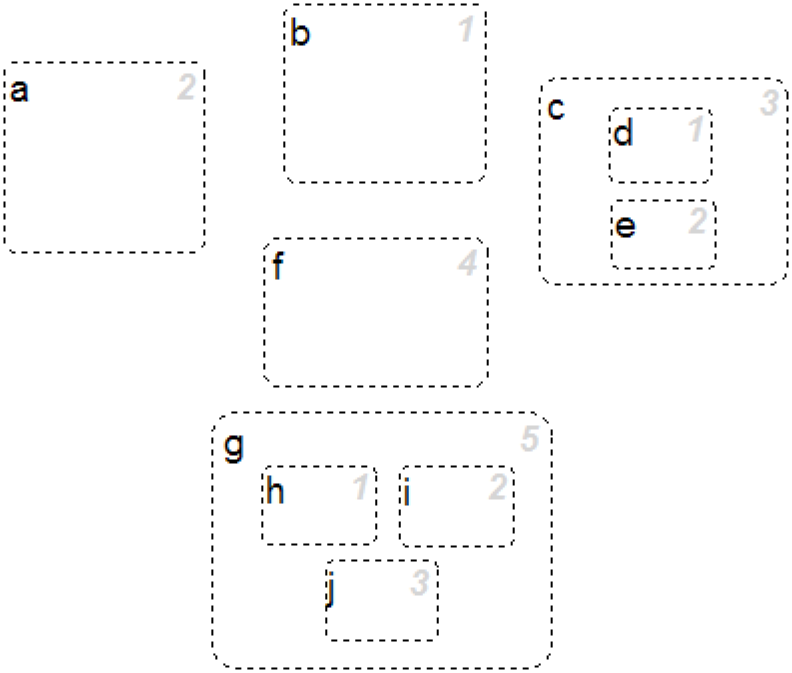
Implicit Ordering of Parallel States

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



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

Using Implicit Ordering for Parallel States

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

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

3 Click **OK**.

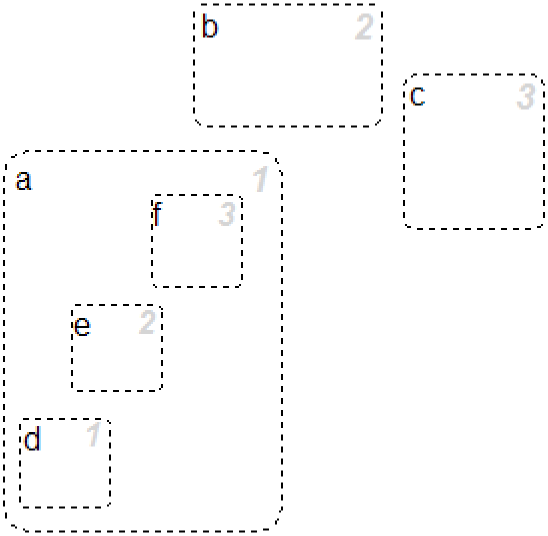
How a Chart Maintains Order of Parallel States

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:

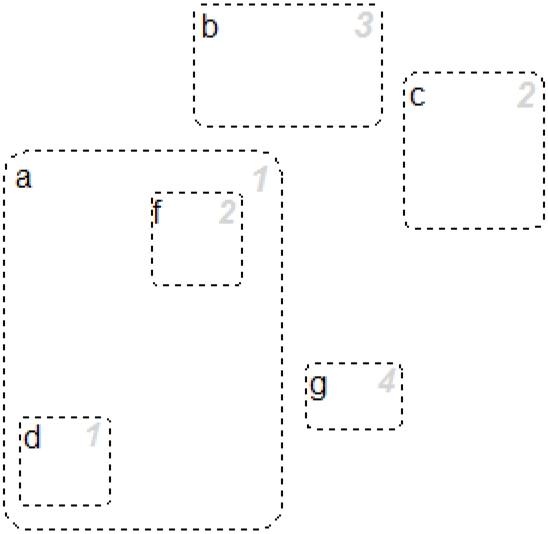


Because of explicit ordering, the priority of each state and substate matches the order of creation in the chart. The chart reprioritizes the parallel states and substates when you perform these actions:

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

2 Add a top-level state `g`.

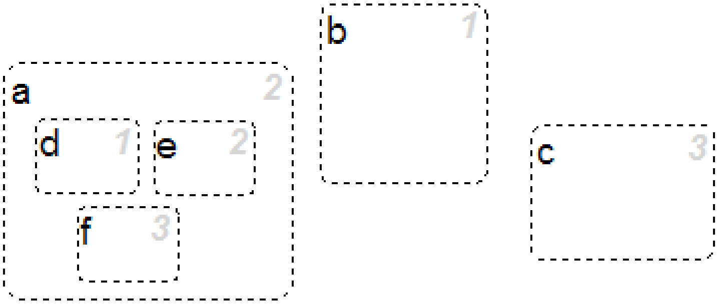
3 Remove substate `e`.



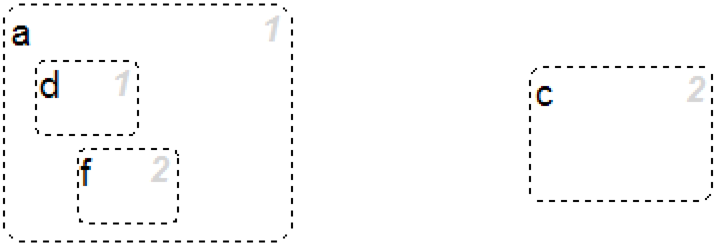
The chart preserves the priority set explicitly for top-level state `b`, but renumbers all other parallel states to preserve their prior relative order.

How a Chart Preserves Relative Priorities in Implicit Mode

For implicit ordering, a chart preserves the intended relative priority based on geometry. Consider this example of implicit ordering:



If you remove top-level state `b` and substate `e`, the chart automatically reprioritizes the remaining parallel states and substates to preserve implicit geometric order:



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.

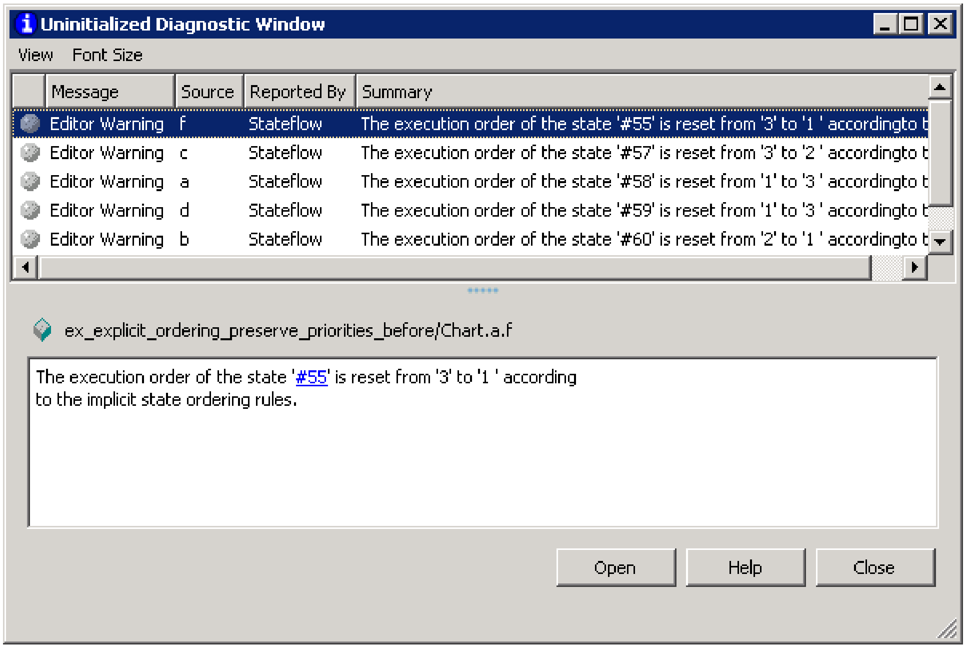
However, in explicit ordering mode, a chart cannot always reinstate the original execution priority to a restored state. It depends on *how* you restore the state.

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.
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.



Message	Source	Reported By	Summary
Editor Warning	f	Stateflow	The execution order of the state '#55' is reset from '3' to '1 ' accordingto
Editor Warning	c	Stateflow	The execution order of the state '#57' is reset from '3' to '2 ' accordingto
Editor Warning	a	Stateflow	The execution order of the state '#58' is reset from '1' to '3 ' accordingto
Editor Warning	d	Stateflow	The execution order of the state '#59' is reset from '1' to '3 ' accordingto
Editor Warning	b	Stateflow	The execution order of the state '#60' is reset from '2' to '1 ' accordingto

ex_explicit_ordering_preserve_priorities_before/Chart.a.f

The execution order of the state '#55' is reset from '3' to '1 ' according
to the implicit state ordering rules.

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

Guidelines for Proper Chart Behavior

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

1. When 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.

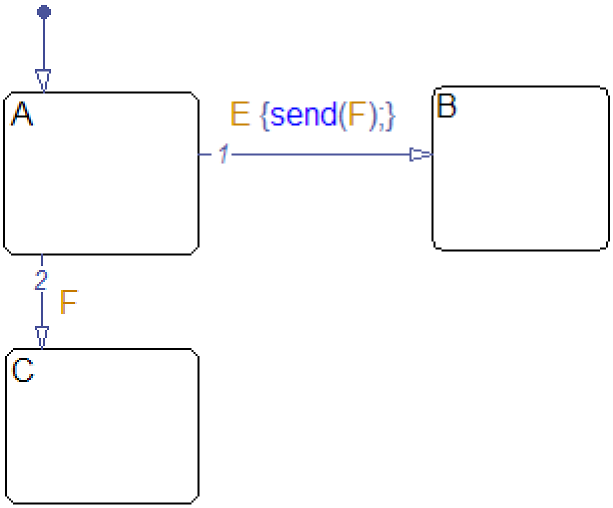
How Early Return Logic Works

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

Action Type	Early Return Logic
Entry	If the state is no longer active at the end of the event broadcast, the chart does not perform remaining steps for entering a state.
Exit	If the state is no longer active at the end of the event broadcast, the chart does not perform remaining `exit` actions or transitions from state to state.
During	If the state is no longer active at the end of the event broadcast, the chart does not perform remaining steps for executing an active state.

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.

Example of Early Return Logic



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

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

2. A valid transition to state `B` exists.

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

   Event `F` interrupts the transition from `A` to `B`.

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

5. A valid transition to state `C` exists.

6. State `A` executes its `exit` action.

7. State `A` becomes inactive.

8. State `C` becomes active.

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

State `C` is now the only active 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 guideline that a state (or chart) with exclusive (OR) decomposition can never have more than one active child. Therefore, the chart uses early return logic and halts the transition from state `A` to state `B`.

Creating Stateflow Charts

Basic Workflow for Building a State Chart

In this section...

Identify System Attributes

Before you build the state chart, identify your system attributes by answering these questions:

1. What are your interfaces?
   - **a** What are the event triggers to which your system reacts?
   - **b** What are the inputs to your system?
   - **c** What are the outputs from your system?
2. Does your system have any operating modes?
   - **a** If the answer is yes, what are the operating modes?
   - **b** Between which modes can you transition? Are there any operating modes that run in parallel?

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

Select a State Machine Type

After identifying your system attributes, the first step is to create a new chart and select one of the following state machine types:

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

For more information about Classic charts, see Chapter 3, "Stateflow Chart Semantics". For more information about Mealy and Moore charts, see Chapter 6, "Building Mealy and Moore Charts".

Specify State Actions and Transition Conditions

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

1. For each state, what are the actions you want to perform?
2. What are the rules for transitioning between your states? If your chart has no states, what are the rules for transitioning between branches of your flow logic?

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

1. Draw states to represent your operating modes, if any. See "Working with States in Charts" on page 4-10.
2. Implement the state actions by adding state labels that use the appropriate syntax. See "State Action Types" on page 10-2.
3. Draw transitions to represent the direction of flow logic, between states or between branches of your flow graph. See "Working with Transitions in Charts" on page 4-24.
4. Implement the transition conditions by adding transition labels that use the appropriate syntax. See "Transition Action Types" on page 10-7.

Define Persistent Data to Store State Variables

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

1. Add local data to the appropriate level of the chart hierarchy. See “Adding Data” on page 8-2.

2. Specify the type, size, complexity, and other data properties. See “Setting Data Properties in the Data Dialog Box” on page 8-5.

Simplify State Actions and Transition Conditions with Function Calls

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

- Flow graph — Encapsulate flow graphs containing if-then-else, switch-case, for, while, or do-while patterns.
- MATLAB — Write matrix-oriented algorithms; call MATLAB functions for data analysis and visualization.
- Simulink — Call Simulink function-call subsystems directly to streamline design and improve readability.
- Truth table — Represent combinational logic for decision-making applications such as fault detection and mode switching.

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

- “Graphical Functions for Reusing Logic Patterns and Iterative Loops” on page 7-30
- Chapter 23, “Using MATLAB Functions in Stateflow Charts”
- Chapter 24, “Using Simulink Functions in Stateflow Charts”
- Chapter 22, “Truth Table Functions for Decision-Making Logic”

If the four types of Stateflow functions do not work, you can write your own C or C++ code for integration with your state chart. For more information about custom code integration, see Chapter 25, “Building Targets”.

Check That Your System Representation Is Complete

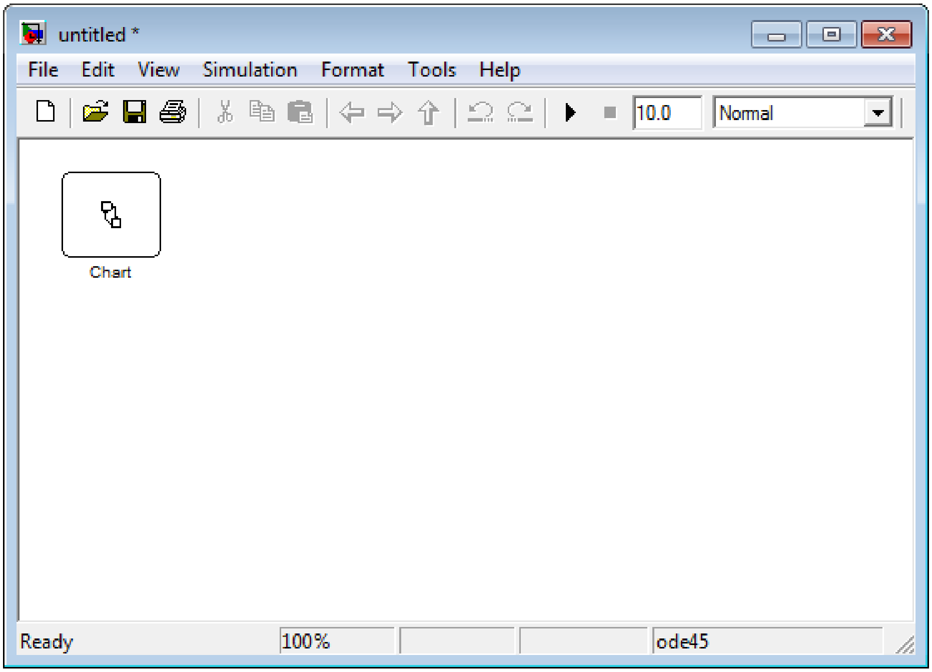
Does your Stateflow chart fully express the logical or event-driven components of your system?

- If the answer is yes, you are done.
- If the answer is no, you can create a separate chart or add hierarchy to your current chart.
    - To create a new chart, repeat all the steps in this basic workflow.
    - To add hierarchy, repeat the previous three steps on lower levels of the current chart.

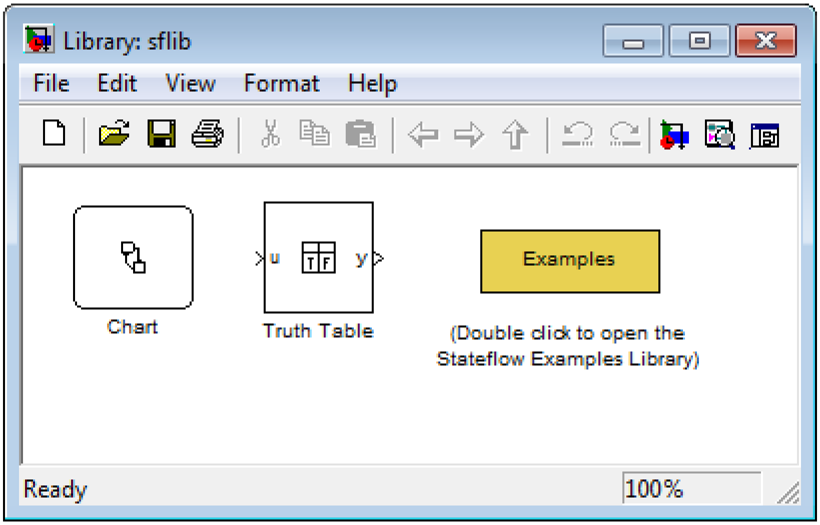
Creating an Empty State Chart

To create a new Stateflow chart, follow these steps:

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



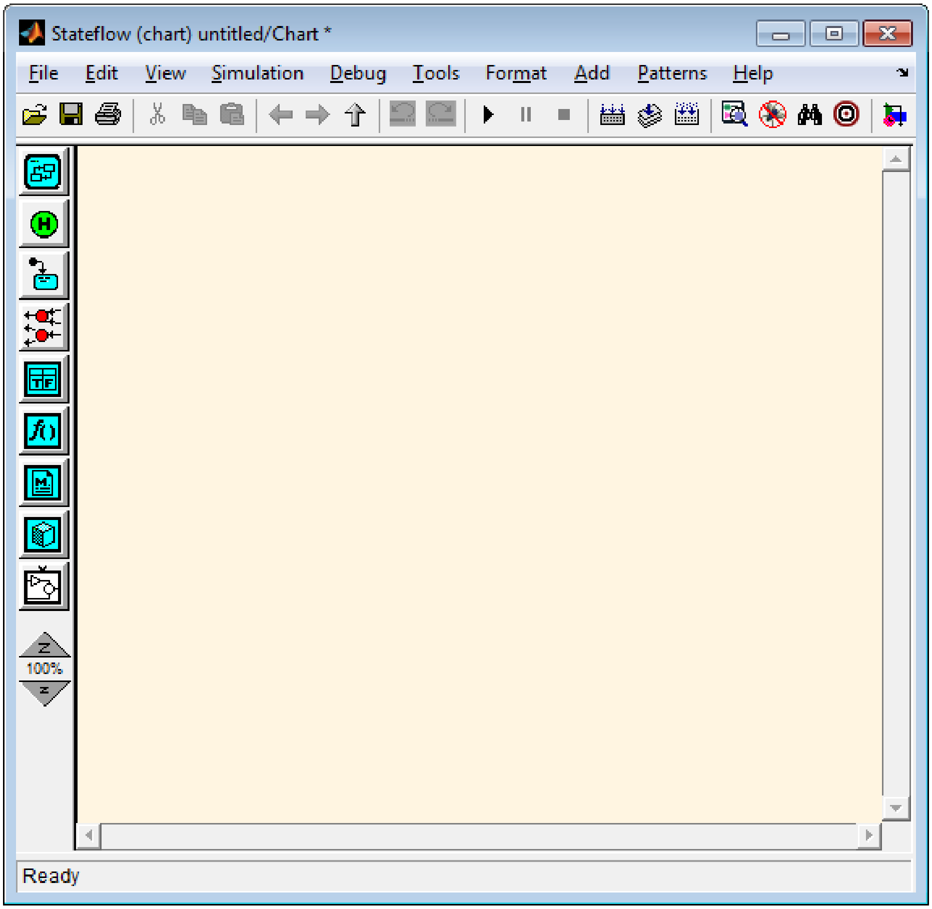
The `stateflow` command also displays the Stateflow block library.



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 Specialized Chart Libraries for Large-Scale Modeling” on page 19-20.

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

The empty chart appears in the editor.



3 Open the Chart properties dialog box.

See “Setting Properties for a Single Chart” on page 19-4.

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").
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, select 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 "Editor Operations" on page 4-32 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 19, "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 Charts

In this section...

Creating a State

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

1. Select the State tool:

2. Move your pointer into the drawing area.

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

3. Click in a particular location to create a state.

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

4. Click the question mark.

   A text cursor appears in place of the question mark.

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

The label for a state specifies its required name and optional actions. See "Labeling States" on page 4-20 for more detail.

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 Encapsulate Modal Logic" on page 7-6 for details.

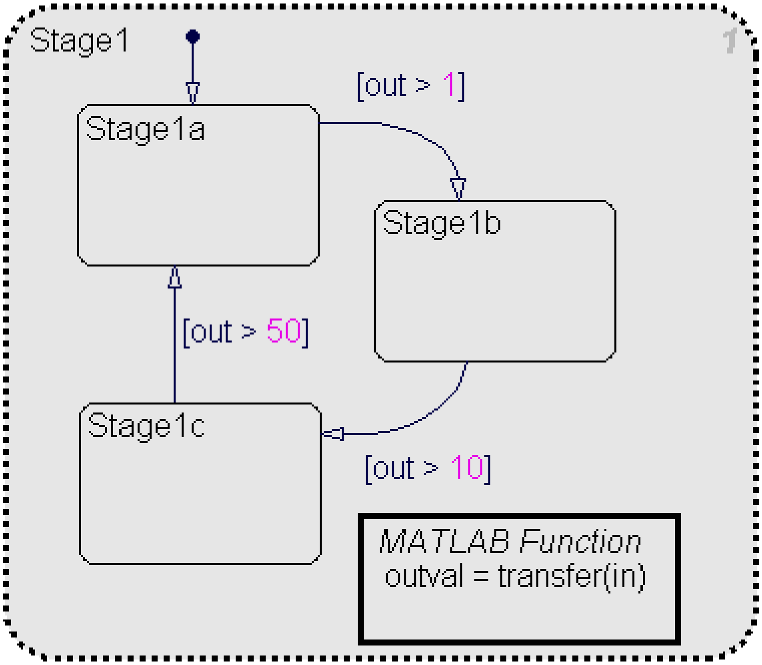
Grouping States

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

How to Group a State

You can group a state by right-clicking it and then selecting **Make Contents > Grouped** in the context menu. The state appears shaded in gray to indicate that it is now grouped.



When to Ungroup a State

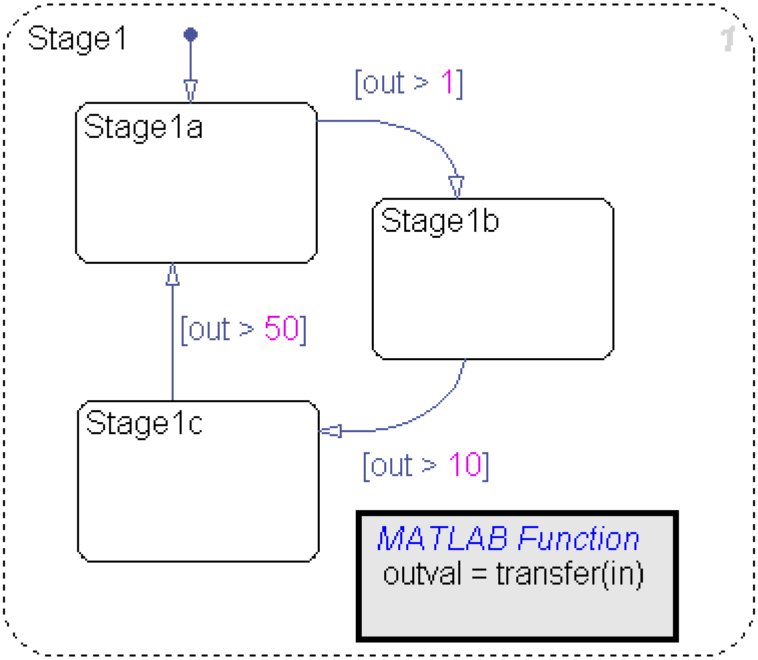
You must ungroup a state before performing these actions:

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

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

How to Ungroup a State

You can ungroup a state by right-clicking it and then clearing **Make Contents > Grouped** in the context menu. The background of the state no longer appears gray.



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 **Decomposition** context menu, and select 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 **Decomposition** context menu, and select 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 3-76 and "Implicit Ordering of Parallel States" on page 3-77.

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:

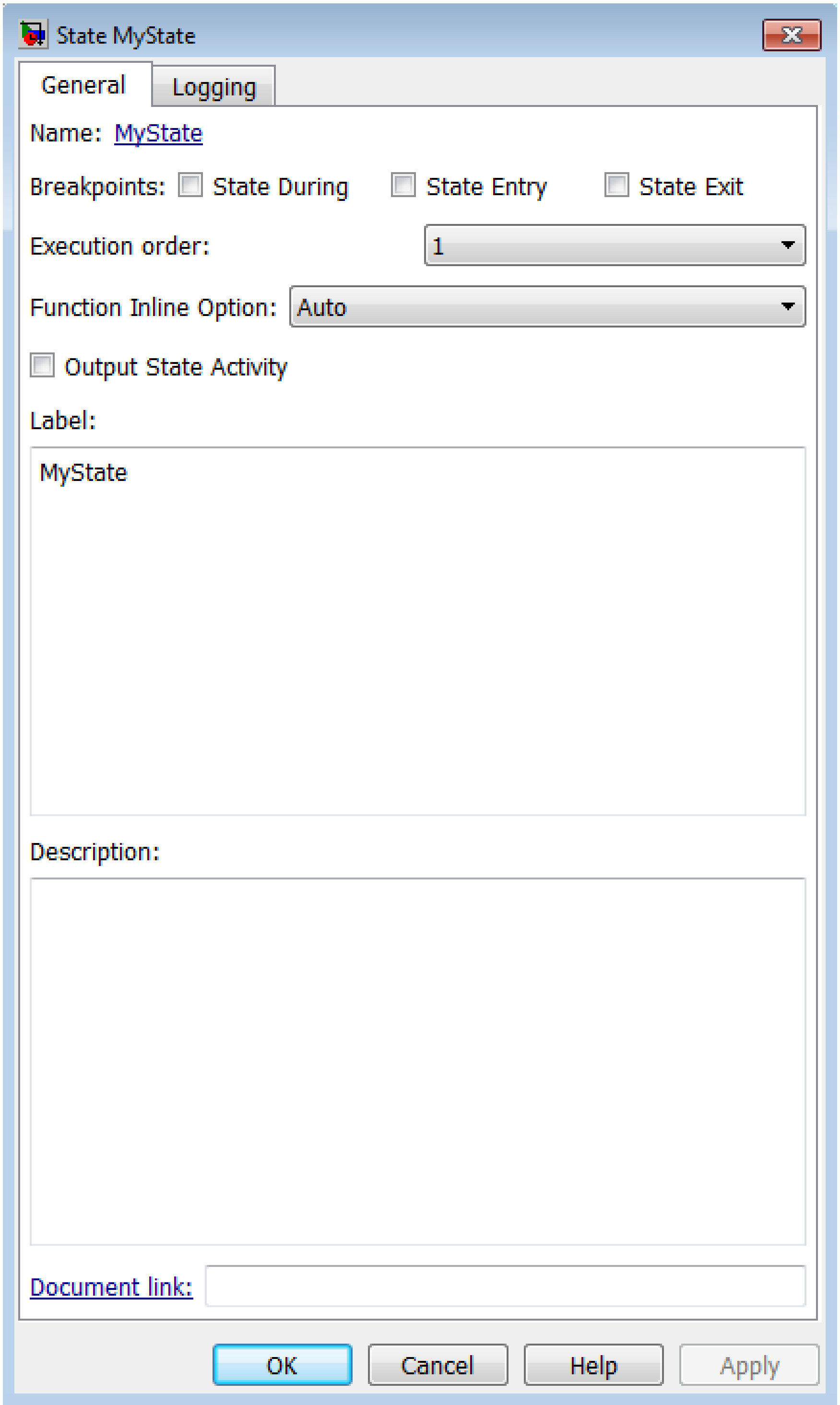
1. Right-click the state and select **Properties**.

   The State properties dialog box appears. For descriptions of properties, see "Properties You Can Set in the General Pane" on page 4-15 and "Properties You Can Set in the Logging Pane" on page 4-18.

2. Modify property settings and then click one of these buttons:
   - **Apply** to save the changes and keep the State dialog box open
   - **Cancel** to return to the previous settings
   - **OK** to save the changes and close the dialog box
   - **Help** to display the documentation in an HTML browser window

Properties You Can Set in the General Pane

The **General** pane of the State properties dialog box appears as shown.



State MyState

Name: MyState

Breakpoints: ☐ State During ☐ State Entry ☐ State Exit

Execution order: 1

Function Inline Option: Auto

☐ Output State Activity

Label:

MyState

Description:

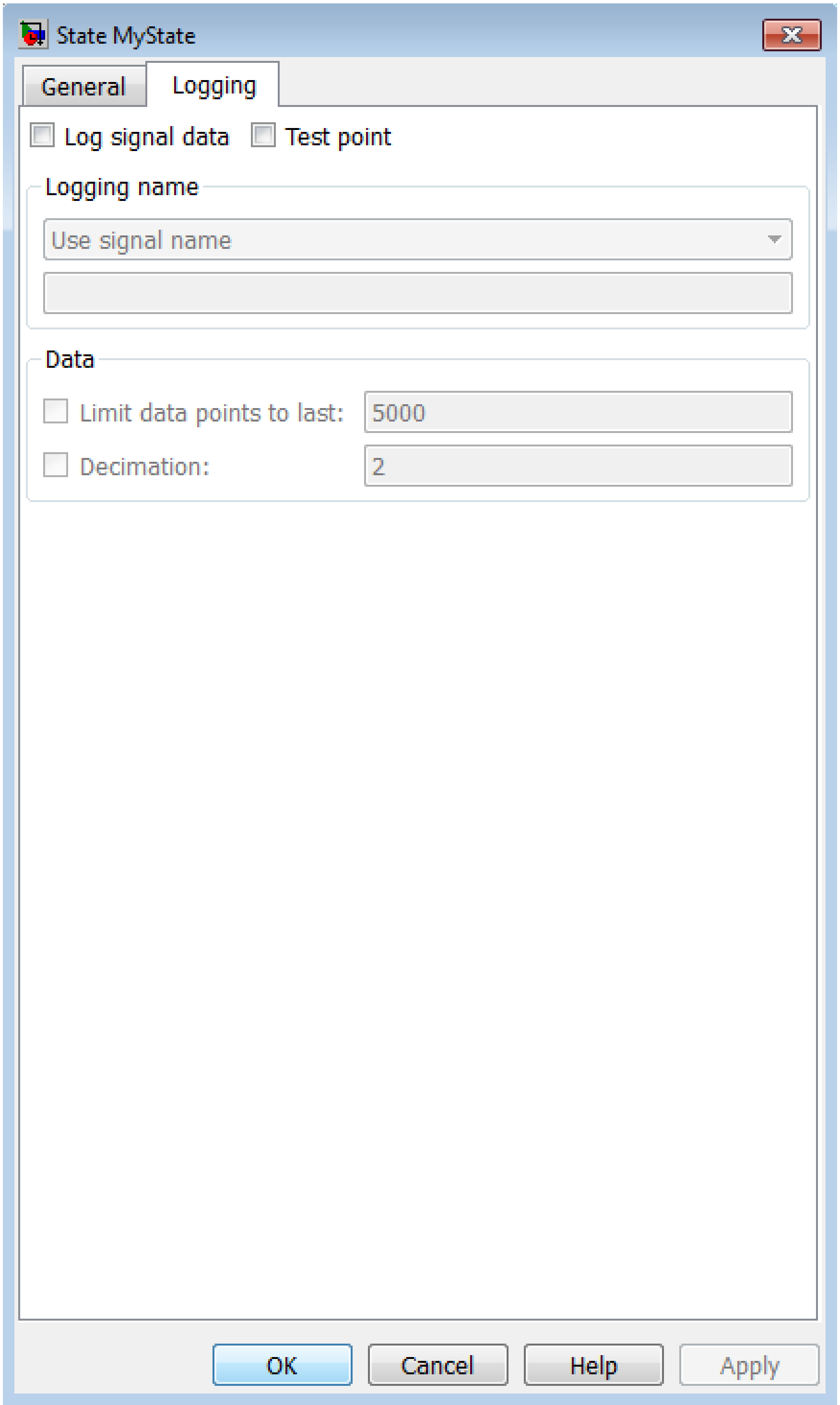
Document link:

Property	Description
Name	Stateflow chart name; read-only; click this hypertext link to bring the state to the foreground.
Breakpoints	Select the check boxes to set debugging breakpoints on the execution of state `entry`, `during`, or `exit` actions during simulation. See Chapter 26, "Debugging and Testing Stateflow Charts".
Execution order	Set the execution order of a parallel (AND) state. This property does not appear for exclusive (OR) states. See "Execution Order for Parallel States" on page 3-75.
Function Inline Option	Select one of these options to control the inlining of state functions in generated code: • **Auto** Inlines state functions based on an internal heuristic. • **Inline** Always inlines state functions in the parent function, as long as the function is not part of a recursion. See "What Happens When You Force Inlining" on page 25-95. • **Function** Creates separate static functions for each state. See "What Happens When You Prevent Inlining" on page 25-95. See "Controlling Inlining of State Functions in Generated Code" on page 25-95.
Output State Activity	Select this check box to output the activity status of this state to a Simulink model via a data output port on the parent chart. See "Outputting State Activity to a Simulink Model" on page 4-23 for more information.

Property	Description
Label	The label for the state, which includes the name of the state and its associated actions. See "Labeling States" on page 4-20.
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`.

Properties You Can Set in the Logging Pane

The **Logging** pane of the State properties dialog box appears as shown.



State MyState

General | Logging

☐ Log signal data ☐ Test point

Logging name

Use signal name

Data

☐ Limit data points to last: 5000

☐ Decimation: 2

Property	Description
Log signal data	Saves the state activity value to the MATLAB workspace during simulation.
Test point	Designates the state as a test point that can be monitored with a floating scope during model simulation. You can also log test point values into MATLAB workspace objects. See "Monitoring Test Points in Stateflow Charts" on page 26-48.
Logging name	Specifies the name associated with the logged state activity. Simulink software uses the signal name as its logging name by default. To specify a custom logging name, select `Custom` from the list box and enter the new name in the adjacent edit field.
Limit data points to last	Limits the state activity logged to the most recent samples.
Decimation	Limits state activity logged by skipping samples. For example, a decimation factor of 2 saves every other sample.

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 label, you can enter actions for any of the following action types:

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

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

- **Exit Actions** — begin on a new line with the keyword `exit` or `ex`, followed by a colon, followed by one or more action statements on one or more lines. To separate multiple actions on the same line, use a comma or a semicolon.
- **During Actions** — begin on a new line with the keyword `during` or `du`, followed by a colon, followed by one or more action statements on one or more lines. To separate multiple actions on the same line, use a comma or a semicolon.
- **Bind Actions** — begin on a new line with the keyword `bind` followed by a colon, followed by one or more data or events on one or more lines. To separate multiple actions on the same line, use a comma or a semicolon.
- **On <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 label. Each block specifies the action for a specific event or set of events, for example:

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

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

Tip You can also edit the label in the properties dialog box for the state. See "Changing State Properties" on page 4-15.

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 4-21), if unnamed, and then select the **Output State Activity** check box on the State properties dialog box (see "Changing State Properties" on page 4-15). 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 Output Data with Simulink" on page 8-31 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 Charts

In this section...

Creating a Transition

Follow these steps to create transitions between states and junctions:

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

   The pointer changes to crosshairs.

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

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

Attached transitions obey the following rules:

- Transitions do not attach to the corners of states. Corners are used exclusively for resizing.
- Transitions exit a source and enter a destination at angles perpendicular to the source or destination surface.
- Newly created transitions have smart behavior. See "Setting Smart Behavior in Transitions" on page 7-20.

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

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

- “Creating Self-Loop Transitions” on page 4-29
- “Creating Default Transitions” on page 4-29

Straight and Curved Transitions

By default, a transition maintains a straight line whenever possible. To create a curved transition, while clicking and dragging from one state or junction to another, do one of the following:

- Press the **S** key.
- Right-click the mouse.

For curved transitions, the source point remains stationary regardless of where you move the end point.

Labeling Transitions

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

- “Editing Transition Labels” on page 4-25
- “Transition Label Format” on page 4-26

For more information on transition concepts, see “Transition Label Notation” on page 2-19.

For more information on transition label contents, see “Transition Action Types” on page 10-7.

Editing Transition Labels

Label unlabeled transitions as follows:

1. Select (left-click) the transition.

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

2 Left-click the ? to edit the label.

You can now type a label.

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

Transition Label Format

Transition labels have the following general format:

```
event [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 4-27
- "Moving Transition Attach Points" on page 4-27
- "Moving Transition Labels" on page 4-28

In addition, transitions move along with the movements of states and junctions. See "Setting Smart Behavior in Transitions" on page 7-20 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 chart 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:

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

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

3. Select an arrowhead size from the menu.

Creating Self-Loop Transitions

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

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

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

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

For the semantics of self-loops, see “Self-Loop Transitions” on page 2-26.

Creating Default Transitions

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

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

The size of the endpoint of the default transition is proportional to the arrowhead size. See “Changing Transition Arrowhead Size” on page 4-28.

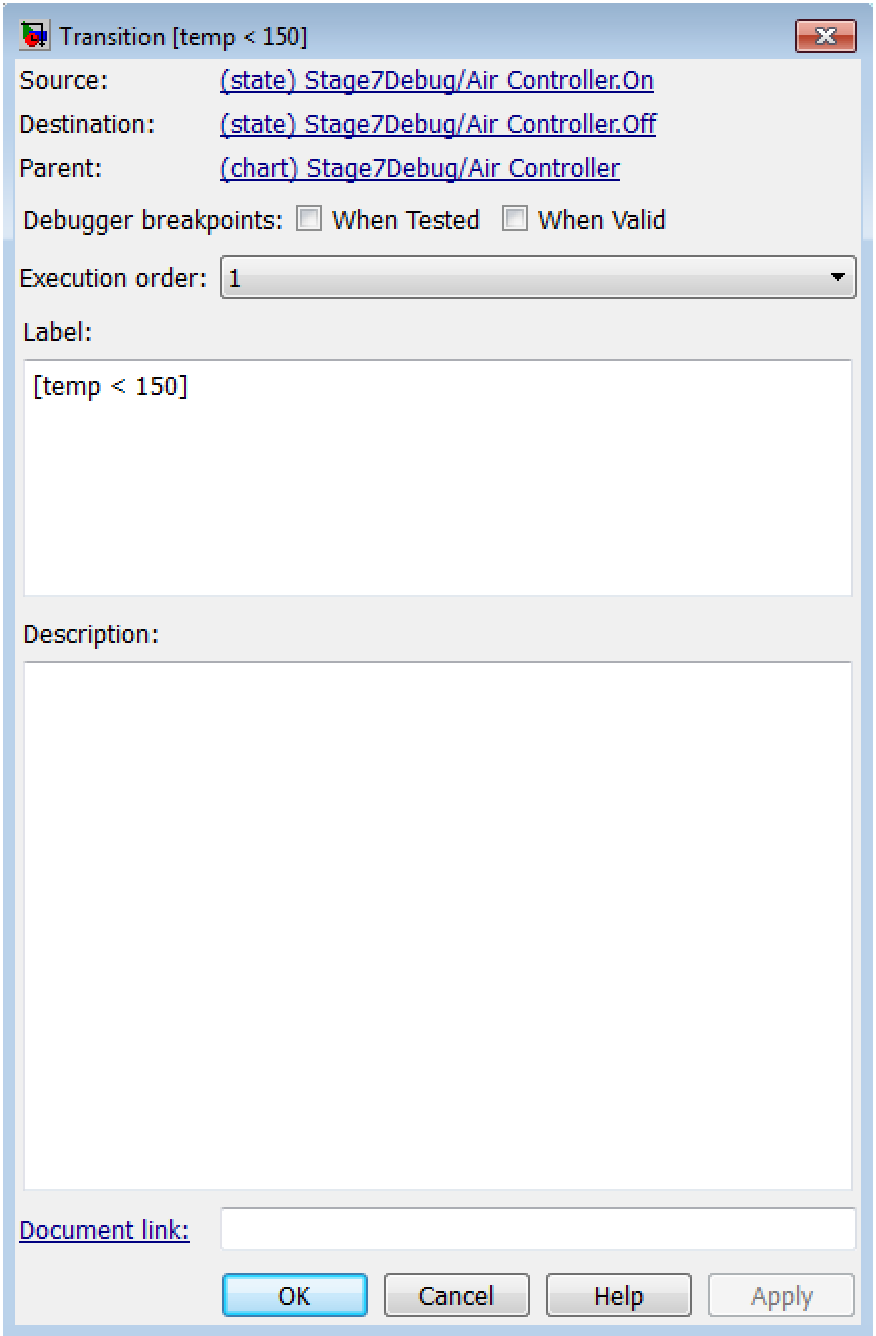
Default transitions can be labeled just like other transitions. See “Labeling Default Transitions” on page 2-31 for an example.

Changing Transition Properties

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

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

The Transition properties dialog box appears.


Transition [temp < 150]

Source: (state) Stage7Debug/Air Controller.On

Destination: (state) Stage7Debug/Air Controller.Off

Parent: (chart) Stage7Debug/Air Controller

Debugger breakpoints: ☐ When Tested ☐ When Valid

Execution order: 1

Label:

[temp < 150]

Description:

Document link:

The following transition properties appear in the dialog box:

Field	Description
Source	Source of the transition; read-only; click the hypertext link to bring the transition source to the foreground.
Destination	Destination of the transition; read-only; click the hypertext link to bring the transition destination to the foreground.
Parent	Parent of this state; read-only; click the hypertext link to bring the parent to the foreground.
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 chart executes the transition.
Label	The transition's label. See "Transition Label Notation" on page 2-19 for more information on valid label formats.
Description	Textual description or comment.
Document link	Enter a Web URL address or a general MATLAB command. Examples are `www.mathworks.com`, `mailto:email_address`, and `edit/spec/data/speed.txt`.

2 After making changes, click one of these buttons:

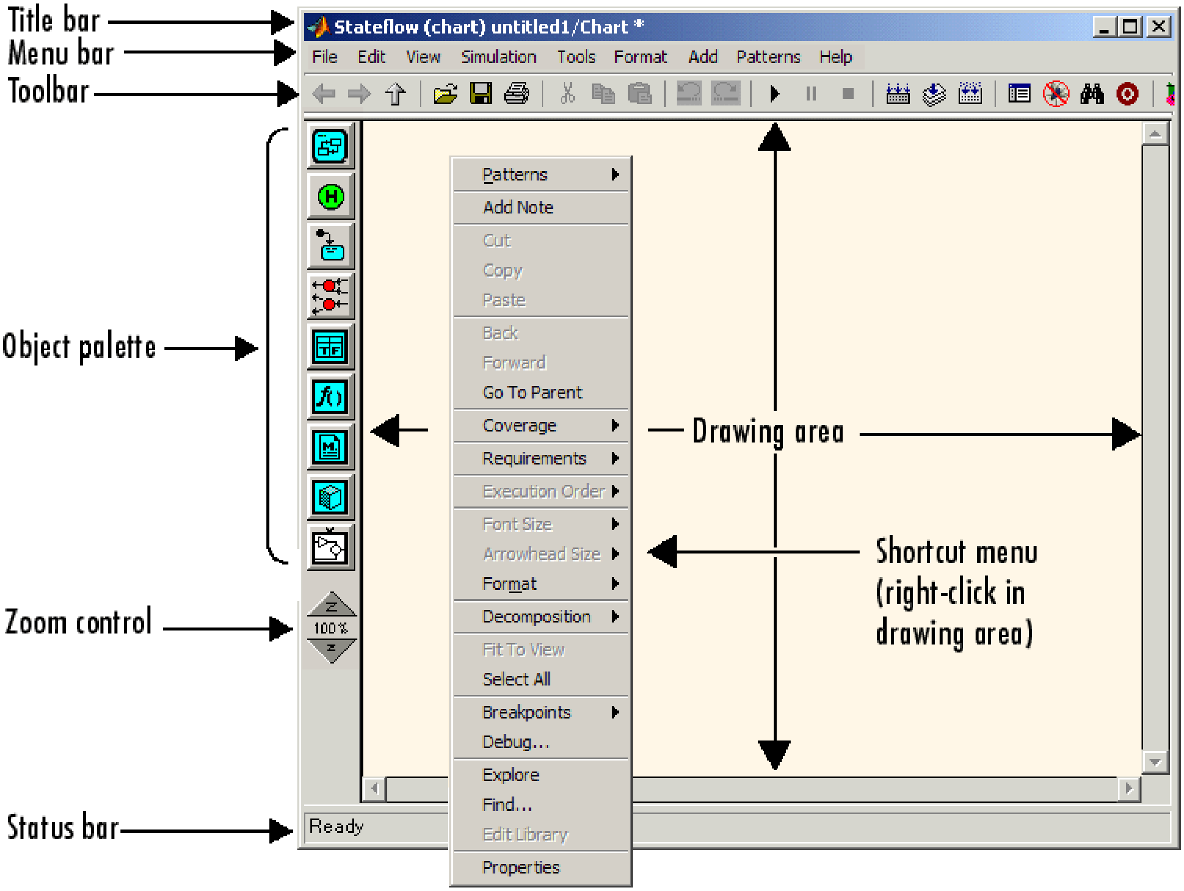
- **Apply** to save the changes and keep the Transition dialog box open.
- **Cancel** to return to the previous settings for the dialog box.
- **OK** to save the changes and close the dialog box.
- **Help** to display Stateflow online help in an HTML browser window.

Editor Operations

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 7-11).

- 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
- Shortcut menus

  These menus pop up from the drawing area when you right-click an object. They display commands that apply only to that object. If you right-click an empty area of the chart, the shortcut menu applies to the chart object. See "Displaying the Context Menu for Objects" on page 4-37 for more information.

- Status bar

  Displays tool tips and status information.

Keyboard Shortcuts for Stateflow Charts

You can use the following keyboard shortcuts in 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**

Task	**Windows® platform**	**UNIX® platform**
Fit chart to screen.	**0** or **Space Bar**	**0** or **Space Bar**
Zoom to normal view.	**1**	**1**
Move the current view down within the full chart.	**2**	**2**
Move the current view down and right within the full chart.	**3**	**3**
Move the current 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 view right within the full chart.	**6**	**6**
Move the current view up and left within the full chart.	**7**	**7**
Move the current view up within the full chart.	**8**	**8**
Move the current 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 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**
Disable (or enable) smart transitions. To create self-loop transitions, disable smart mode. For details, see "Creating Self-Loop Transitions" on page 4-29. To maintain straight lines from transition sources, enable smart mode (the default). For details, see "What Smart Transitions Do" on page 7-20.	**s** (smart) or **S**	**s** (smart) or **S**
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**

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:

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:

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 a Chart

You can specify the color and font for items in a chart, for a single item or all items in the chart.

Changing Fonts for a Single Item

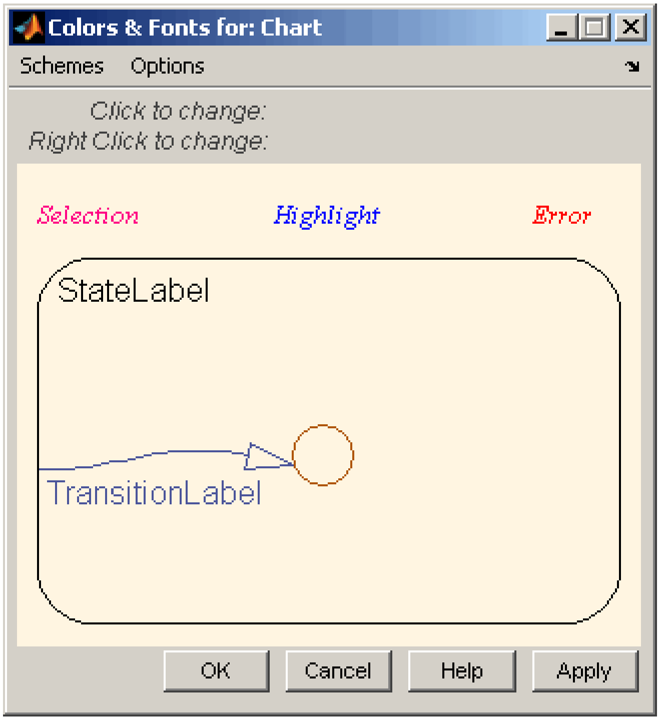
You can change the font for a single item as follows:

1. Right-click the item.
2. In the context menu, select **Font Size** > *size of font*.

Using the Colors & Fonts Dialog Box

The Colors & Fonts dialog box helps you specify a color scheme for the 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.



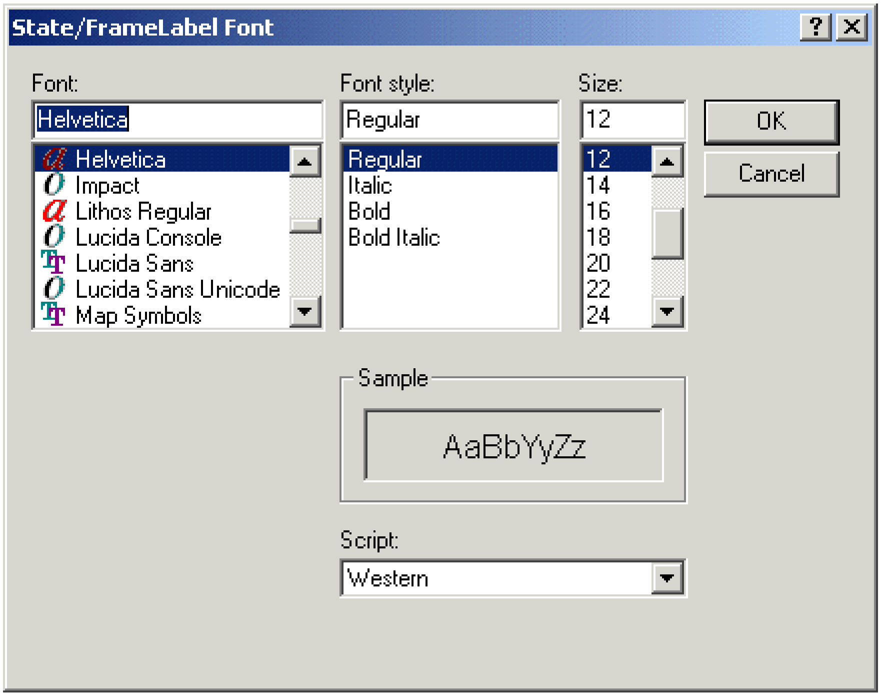
The drawing area of the dialog box displays examples of the types of objects whose colors and font labels you can specify. The examples use the colors and label fonts specified by the current color scheme for the chart. To choose another color scheme, select the scheme from the dialog box's **Schemes**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

You can use color highlighting to differentiate the following syntax elements in the Stateflow action language:

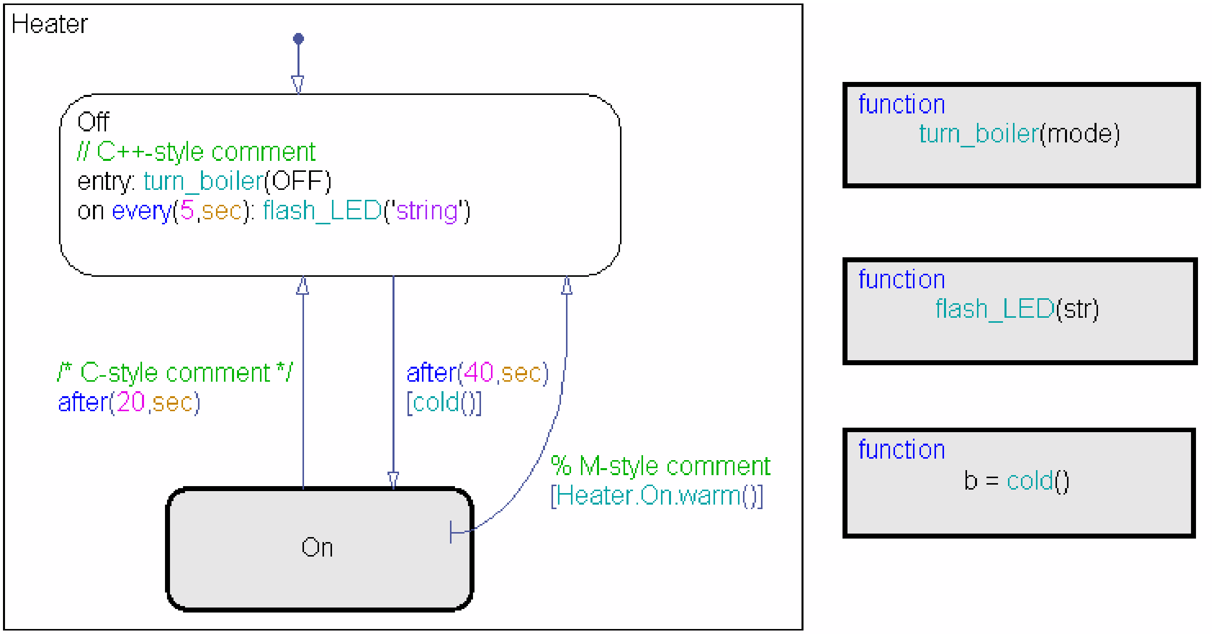
- Keyword

- Comment
- Event
- Function
- String
- Number

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

Default Syntax Highlighting

The following chart illustrates the default highlighting for each language element:



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

To modify color assignments, see "Editing Syntax Highlighting" on page 4-44. To disable syntax highlighting, see "Enabling and Disabling Syntax Highlighting" on page 4-44.

Editing Syntax Highlighting

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

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

   The Syntax Highlight Preferences dialog box appears.

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

This step adds objects to the list of already selected objects unless an object was already selected, in which case, the object is deselected. This type of multiple object selection is useful for selecting objects within a state without selecting the state itself when you select a state and all of its objects and then Shift-click inside the containing state to deselect it.

- 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 appears highlighted in the color set as the selection color (blue by default; see "Specifying Colors and Fonts in a Chart" on page 4-39 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 chart to another. The chart retains a selection list of the most recently cut objects. The objects are pasted in the drawing area location closest to the current pointer location.

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

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 chart's clipboard.

Note If you copy and paste a state in the chart, 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 chart to another, select **Copy** and then **Paste** from the right-click context menu.

Formatting Chart Objects

To enhance readability of objects in a 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

Basic Steps for Aligning, Distributing, or Resizing Chart Objects

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

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

   To set explicit ordering, 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 change from occurring. For more information, see "Execution Order for Parallel States" on page 3-75 and "Evaluation Order for Outgoing Transitions" on page 3-55.

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

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

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

To set an object as the reference, right-click the object. Brackets appear around the reference object, similar to this:

MATLAB Function

balli = getBallInteraction

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

---

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

Options for Aligning Chart Objects

This option...	Aligns selected objects...
Align Top Edges	Along the top edges
Align Centers Horizontally	So that the centers fall on the same horizontal line
Align Bottom Edges	Along the bottom edges
Align Left Edges	Along the left edges
Align Centers Vertically	So that the centers fall on the same vertical line
Align Right Edges	Along the right edges

Options for Distributing Chart Objects

This option...	Distributes selected objects so that...
Distribute Items Horizontally	The center-to-center horizontal distance between any two objects is the same. **Note** The horizontal space for distribution is the distance between the left edge of the leftmost object and the right edge of the rightmost object. If the total width of the objects you select exceeds the horizontal space available, objects can overlap after distribution.
Distribute Items Vertically	The center-to-center vertical distance between any two objects is the same. **Note** The vertical space for distribution is the distance between the top edge of the highest object and the bottom edge of the lowest object. If the total height of the objects you select exceeds the vertical space available, objects can overlap after distribution.

This option...	Distributes selected objects so that...
Make Horizontal Gaps Even	The horizontal white space between any two objects is the same. **Note** The space restriction for **Distribute Items Horizontally** applies.
Make Vertical Gaps Even	The vertical white space between any two objects is the same. **Note** The space restriction for **Distribute Items Vertically** applies.

Options for Resizing Chart Objects

This option...	Makes selected objects have...
Make Items Same Height	The same height
Make Items Same Width	The same width
Make Items Same Size	The same height and width

Example of Aligning Chart Objects

Suppose that you open the `sf_pool` demo model and see a chart with multiple MATLAB functions.

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

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

```
/initBalls();
```

```
1 [isAnyBallGoingToStop()]
  /updateStopFlags();

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

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

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

To align the three MATLAB functions on the right:

1 Type `sf_pool` at the MATLAB command prompt to open the model.

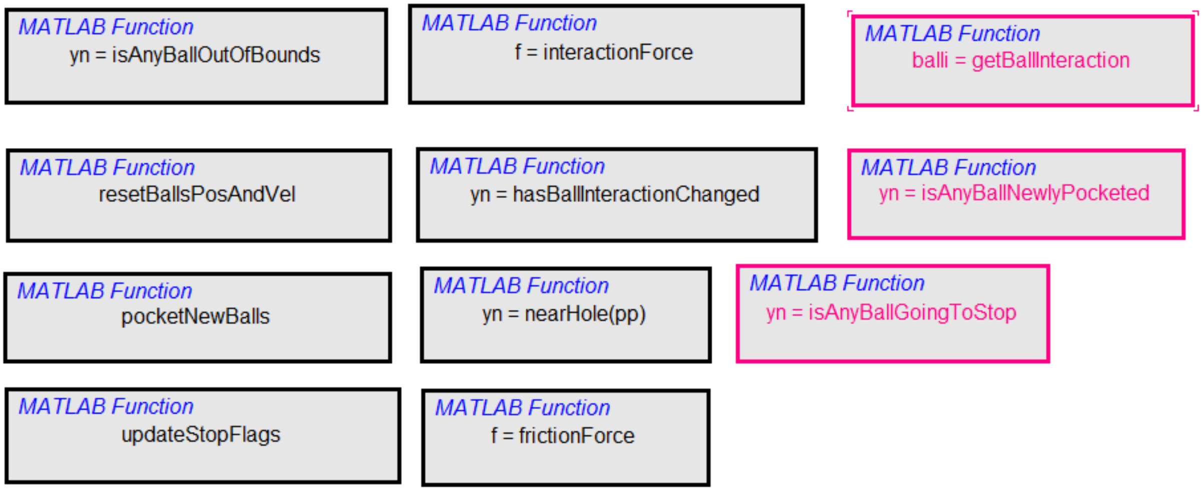
Tip Expand the Stateflow Editor to see the entire chart.

2 Click the function `isAnyBallGoingToStop`.

3 Shift-click the function `isAnyBallNewlyPocketed`.

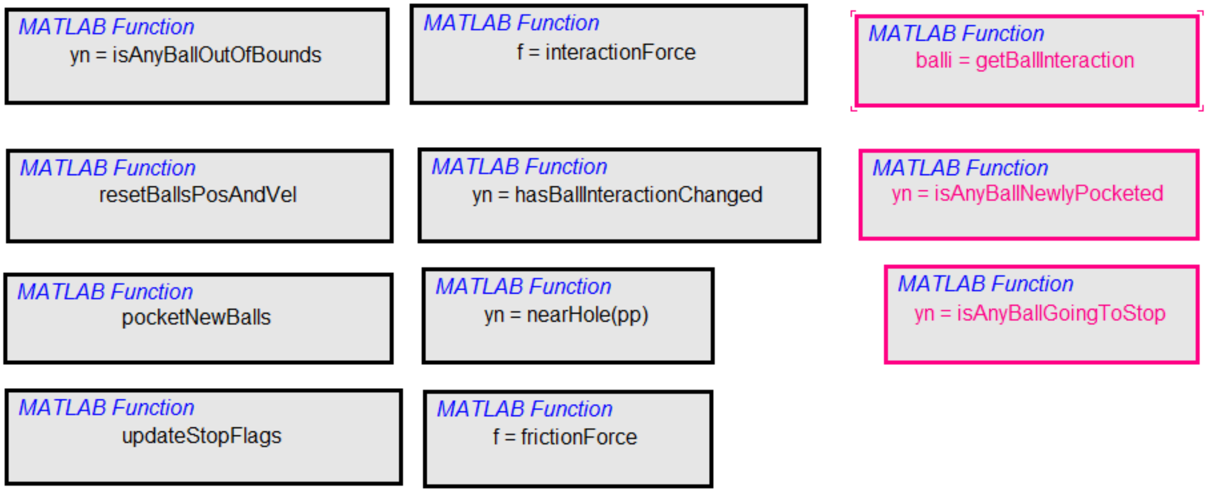
4 Shift-click the function `getBallInteraction`.

This object is the reference (or anchor) for aligning the three functions. Brackets appear around the function.



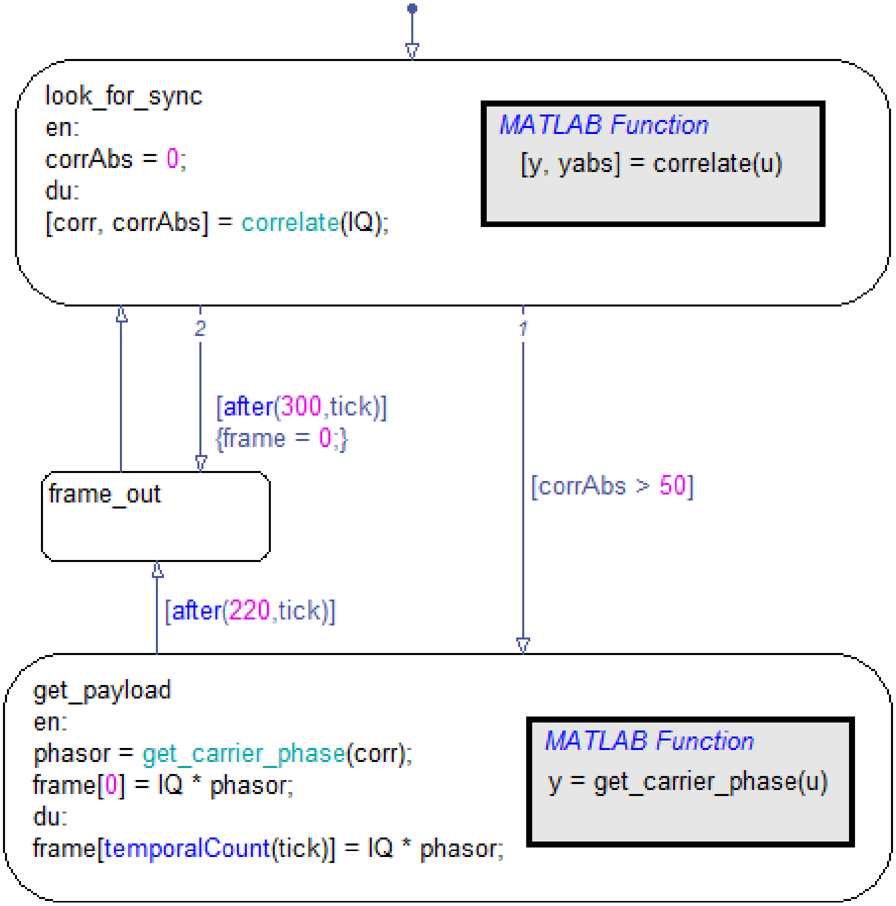
5 Select **Format > Align Items > Align Right Edges**.

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



Example of Distributing Chart Objects

Suppose that you open the `sf_frame_sync_controller` demo model and see a chart with three states.



To distribute the three states vertically:

1. Type `sf_frame_sync_controller` at the MATLAB command prompt to open the model.

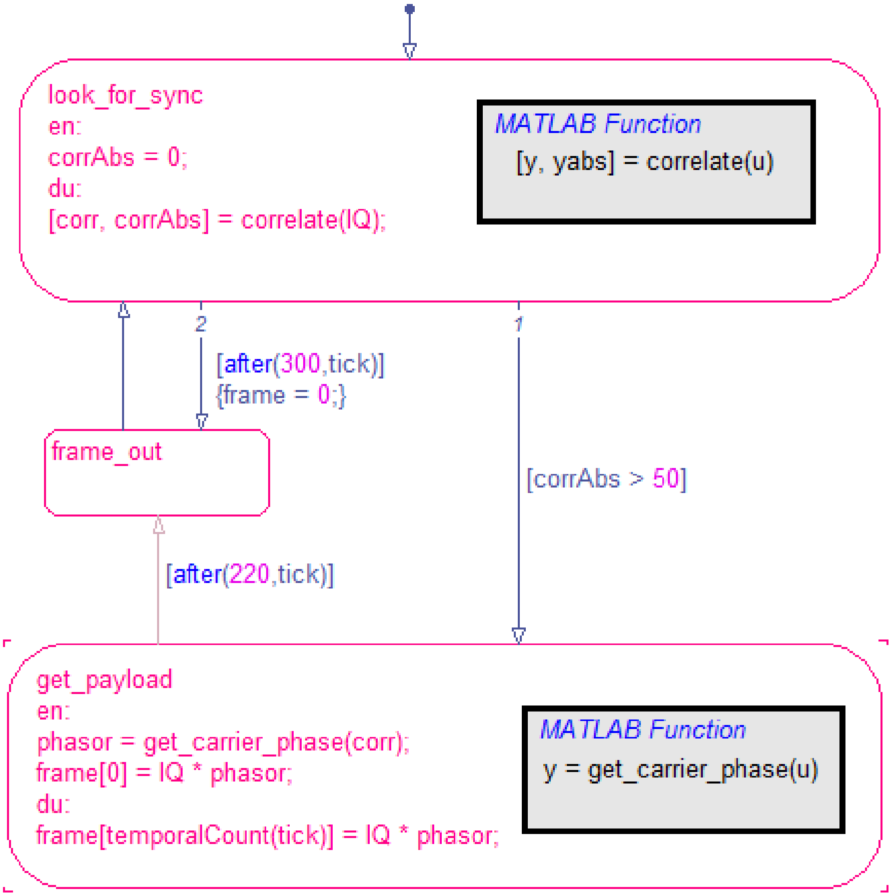
Tip Double-click the Frame Sync Controller block to open the chart.

2 Select the three states in any order.

  Shift-click to select more than one state.

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

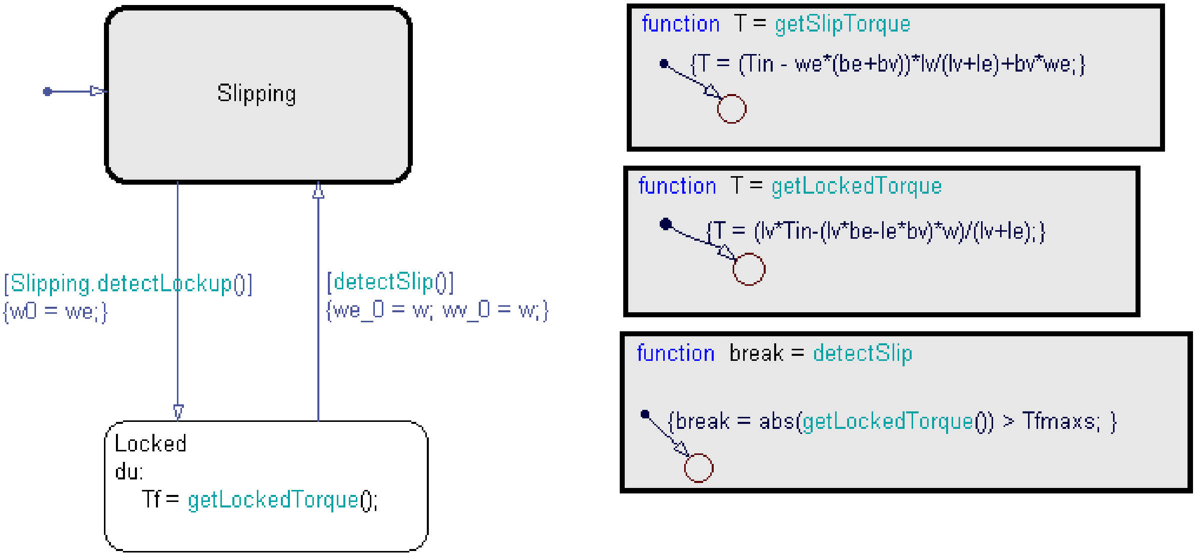
This step ensures that the vertical white space between any two states is the same.



Note When you select the three states in any order, your reference object might differ from the one shown. This difference does not affect distribution of vertical white space.

Example of Resizing Chart Objects

Suppose that you open the `sf_clutch` demo model and see a chart with graphical functions of different sizes.



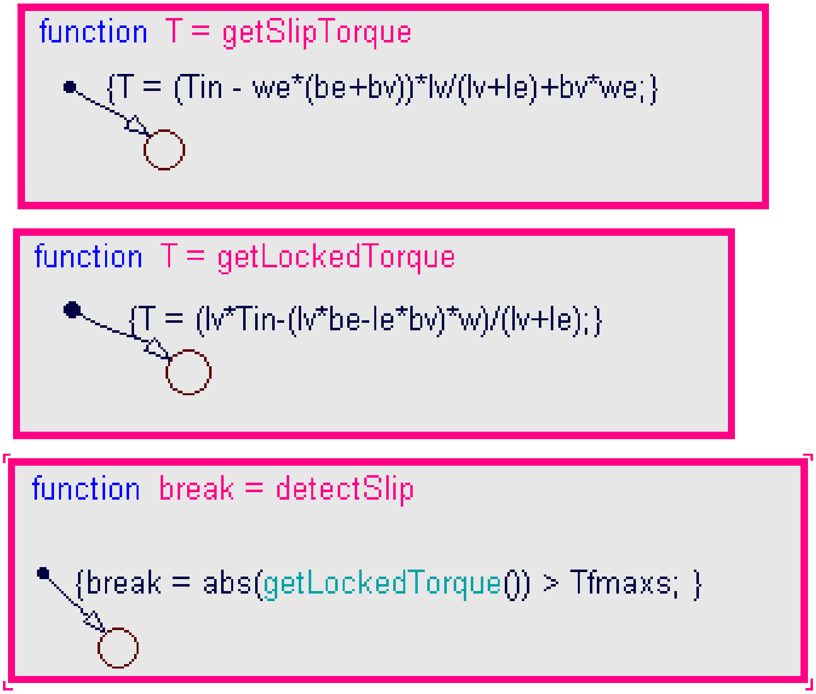
To resize the graphical functions so that they all match the size of `detectSlip`:

1. Type `sf_clutch` at the MATLAB command prompt to open the model.

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

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

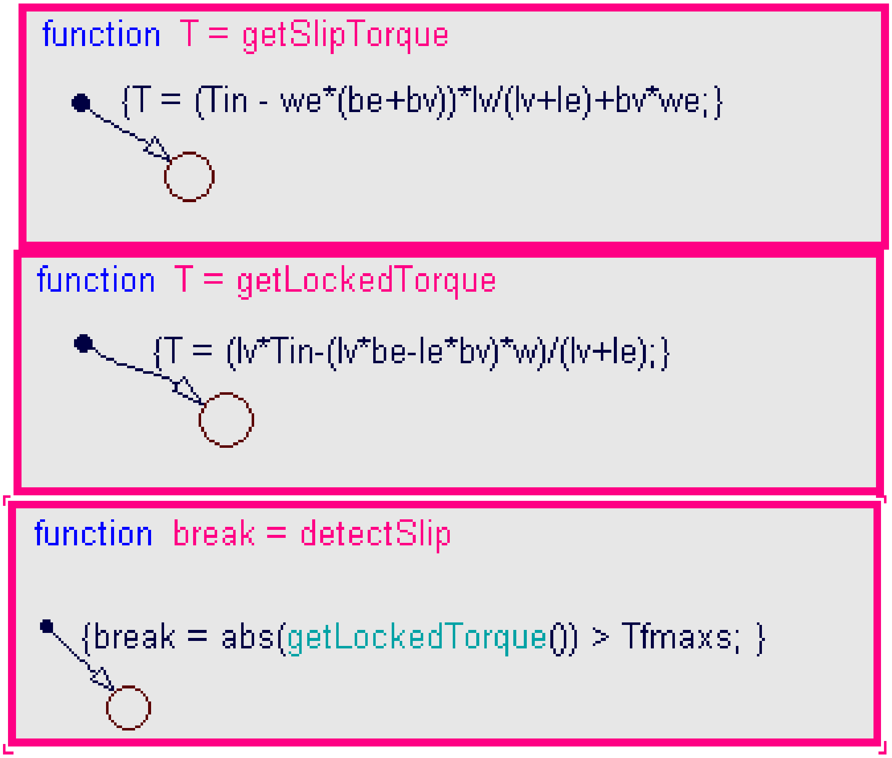
Right-click the function to mark it with brackets.



```
function  T = getSlipTorque
{T = (Tin - we*(be+bv))*lv/(lv+le)+bv*we;}

function  T = getLockedTorque
{T = (lv*Tin-(lv*be-le*bv)*w)/(lv+le);}

function  break = detectSlip
{break = abs(getLockedTorque()) > Tfmaxs; }
```

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

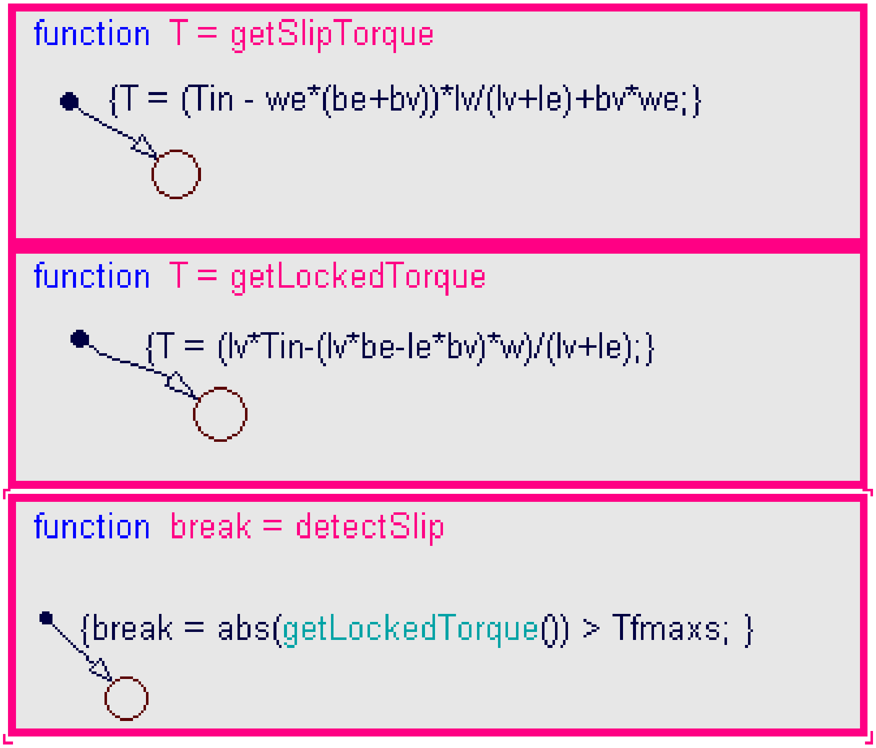
This step ensures that the three functions are the same size.



```
function T = getSlipTorque
{T = (Tin - we*(be+bv))*lv/(lv+le)+bv*we;}

function T = getLockedTorque
{T = (lv*Tin-(lv*be-le*bv)*w)/(lv+le);}

function break = detectSlip
{break = abs(getLockedTorque()) > Tfmaxs; }
```

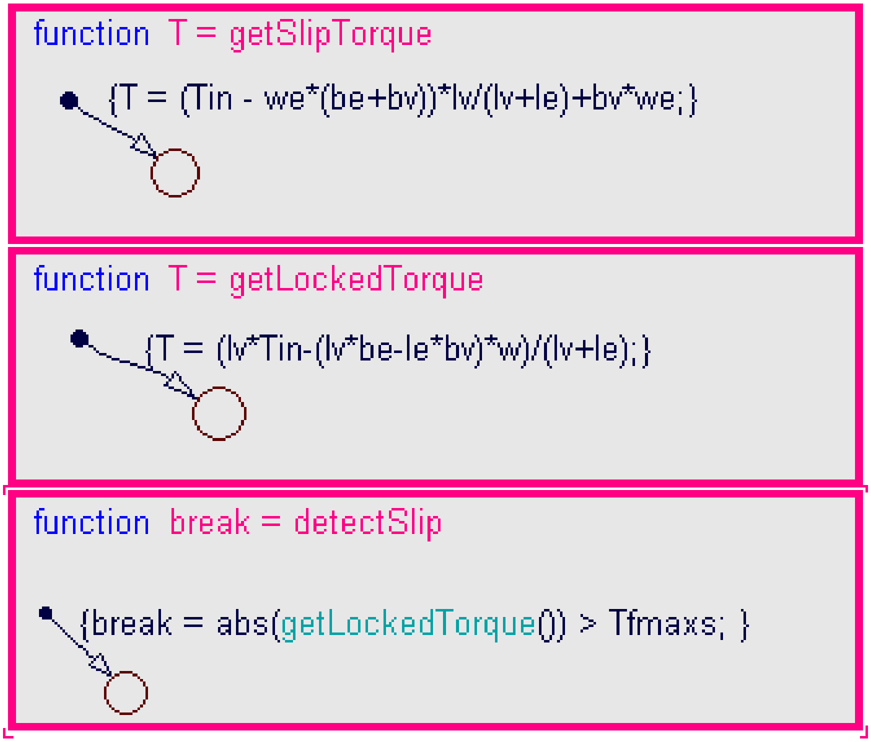
5 Adjust the function boxes to correct the format:

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



```
function T = getSlipTorque
{T = (Tin - we*(be+bv))*lv/(lv+le)+bv*we;}

function T = getLockedTorque
{T = (lv*Tin-(lv*be-le*bv)*w)/(lv+le);}

function break = detectSlip
{break = abs(getLockedTorque()) > Tfmaxs; }
```

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



```
function T = getSlipTorque
{T = (Tin - we*(be+bv))*lv/(lv+le)+bv*we;}

function T = getLockedTorque
{T = (lv*Tin-(lv*be-le*bv)*w)/(lv+le);}

function break = detectSlip
{break = abs(getLockedTorque()) > Tfmaxs; }
```

Zooming a Chart

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

- **Zoom In** button. Zooms in by the current zoom factor.
- **Zoom Out** button. Zooms out by the current zoom factor.

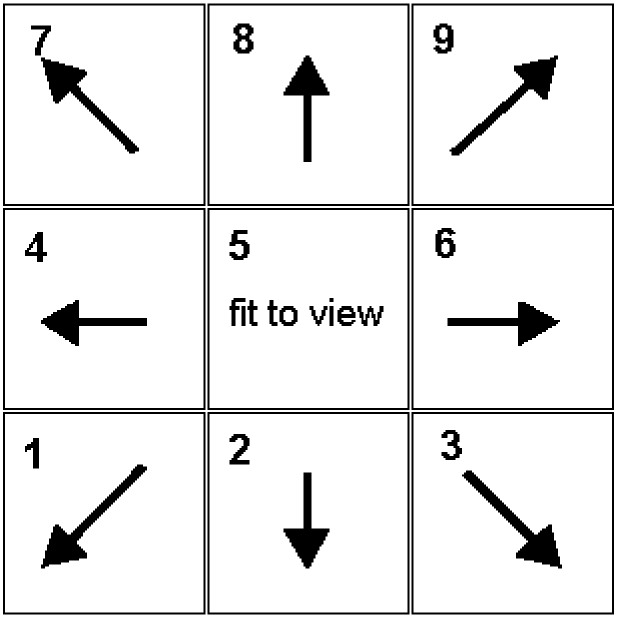
Zooming with Shortcut Keys

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.

Undoing and Redoing Editor Operations

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

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

Exceptions for Undo

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

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

  To understand subcharting, see “Using Subcharts to Encapsulate Modal Logic” on page 7-6.
- 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 7-14 and “Drawing a Supertransition Out of a Subchart” on page 7-17.
- You cannot undo any changes made to the chart through the Stateflow API.

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

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

Printing Stateflow Charts

Printing Scaled Charts

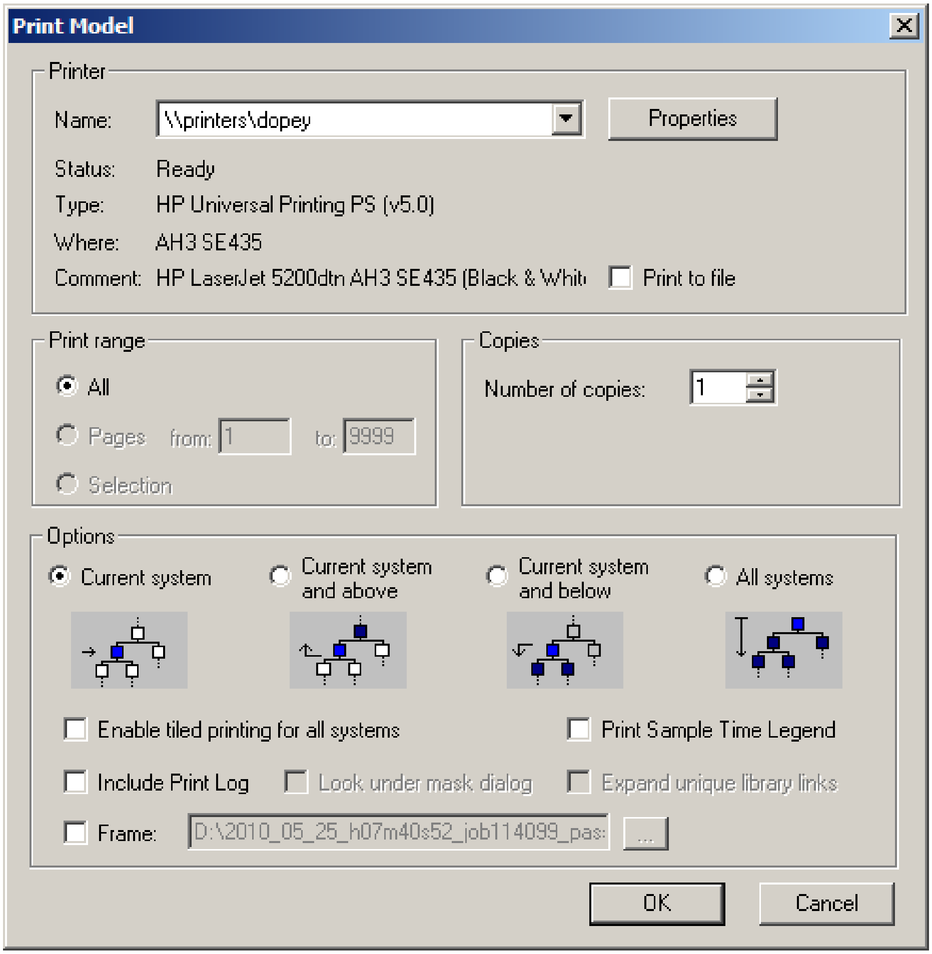
By default, Stateflow software scales each chart that you print to fit on a single page. If you prefer to print charts without scaling to preserve clarity and detail, you can use tiled printing, as described in “Using Tiled Printing for Stateflow Charts” on page 4-66.

To print scaled charts, follow these steps:

1. Open the chart or subchart you want to print.

2. In the editor, select **File > Print**.

   The Print Model dialog box appears:



Print Model

Printer

Name: \\printers\dopey

Status: Ready

Type: HP Universal Printing PS (v5.0)

Where: AH3 SE435

Comment: HP LaserJet 5200dtn AH3 SE435 (Black & Whit

☐ Print to file

Print range

● All

○ Pages from: 1 to: 9999

○ Selection

Copies

Number of copies: 1

Options

● Current system ○ Current system and above ○ Current system and below ○ All systems

☐ Enable tiled printing for all systems

☐ Print Sample Time Legend

☐ Include Print Log

☐ Look under mask dialog

☐ Expand unique library links

☐ Frame: D:\2010_05_25_h07m40s52_job114099_pas:

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
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 4-66.
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**.
Print Sample Time Legend	Prints the Sample Time Legend on a separate page from your model. The legend contains sample time information for your entire model, including any subsystems.
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 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 4-63.

To print charts on tiled pages, follow these steps:

1 Open the chart or subchart you want to print.

2 In the 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 Model 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 editor displays the page boundaries as tiles in a checkerboard pattern. When a state extends beyond the page boundary, you can select and drag the 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

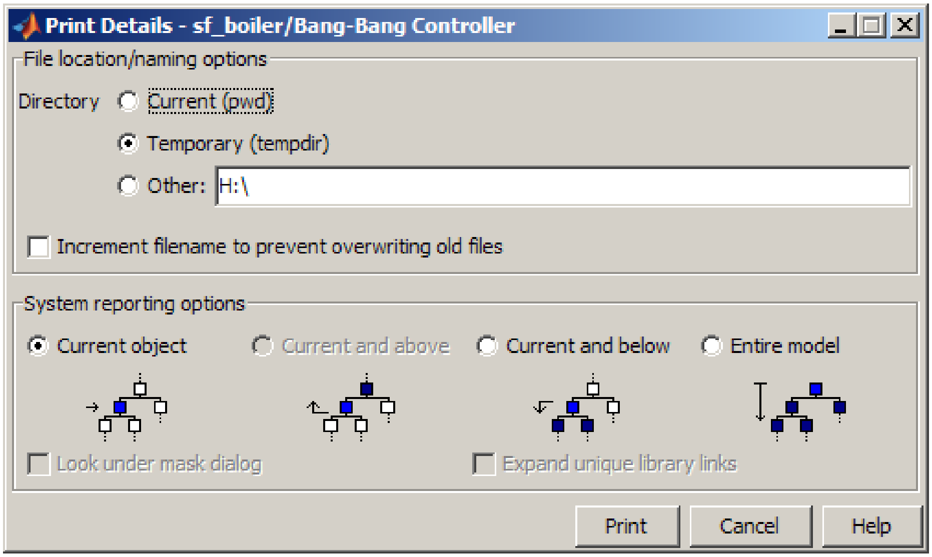
The **Print Details** report is an extension of the **Print Details** report in the Simulink model window. It provides a report of Stateflow and Simulink objects relative to the chart currently in view from which you select the report.

To generate a model report on chart objects:

1 Open the chart or subchart for which you want a report.

2 In the editor, select **File > Print Details**.

The Print Details dialog box appears as follows:



Print Details - sf_boiler/Bang-Bang Controller

File location/naming options

Directory ○ Current (pwd) ● Temporary (tempdir) ○ Other: H:\

☐ Increment filename to prevent overwriting old files

System reporting options

● Current object ○ Current and above ○ Current and below ○ Entire model

☐ Look under mask dialog ☐ Expand unique library links

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 4-69 and “Report Format” on page 4-69.

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

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

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

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

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

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

- An appendix tabulates the covered Stateflow and Simulink objects in the report.

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 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.

Tip You can also print the current view from the MATLAB command line using the `sfprint` function.

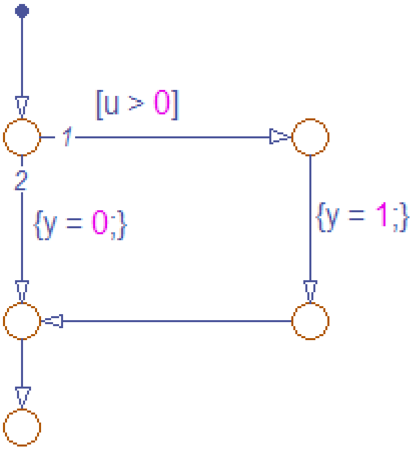
5

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 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 chart. For more information about graphical functions, see "Graphical Functions for Reusing Logic Patterns and Iterative Loops" on page 7-30.

Creating Flow Graphs with the Pattern Wizard

In this section...

Why Use the Pattern Wizard?

The Pattern Wizard is a utility 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 chart.

How do I create and open a new Stateflow chart?

a Type `sfnew` or `stateflow` at the MATLAB command prompt.

A model opens, containing an empty chart.

b Double-click the chart to open it.

2. Select a flow graph pattern:

To Create:	Select:	Reference
`if` decision patterns	**Patterns > Add Decision**	"Decision Logic Patterns in Flow Graphs" on page 5-9
`for`, `while`, and `do while` loop patterns	**Patterns > Add Loop**	"Iterative Loop Patterns in Flow Graphs" on page 5-14
`switch` patterns	**Patterns > Add Switch**	"Switch Patterns in Flow Graphs" on page 5-16

   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 to your chart.

4. Specify conditions and actions (optional).

   You can also add or change conditions and actions directly in the chart.

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 5-20.

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 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 editor.

Guidelines for Creating a Pattern Folder

The Pattern Wizard uses a single, flat folder for saving and retrieving flow graph patterns. Follow these guidelines when creating your pattern folder:

- Store all flow graphs at the top level of the pattern folder; do not create subfolders.
- Make sure all flow graph files have a `.mdl` extension.

How to Save Flow Graph Patterns for Easy Retrieval

1. Create a folder for storing your patterns according to "Guidelines for Creating a Pattern Folder" on page 5-7.
2. In your 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 folder for storing custom patterns.

   The Pattern Wizard stores your flow graphs in the pattern folder as an `.mdl` file. The patterns that you save in this folder 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 designated folder (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 folder.

How to Change Your Pattern Folder

1 Rename your existing pattern folder.

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 folder.

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 chart.

See "Creating a Graphical Function" on page 7-31.

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.

4 In the menu bar, select **Patterns > Add Custom**.

The Select a Custom Pattern dialog box appears, displaying all of your saved patterns.

Why does my dialog box not display any patterns?

You have not 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 the menu bar, 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:

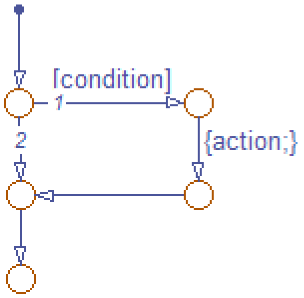
   - Connect the flow graphs to the appropriate transitions.
   - Ensure that there is only one default transition for exclusive (OR) states at each level of hierarchy.
   - Define all necessary inputs, outputs, and local data.

MAAB-Compliant Patterns from the Pattern Wizard

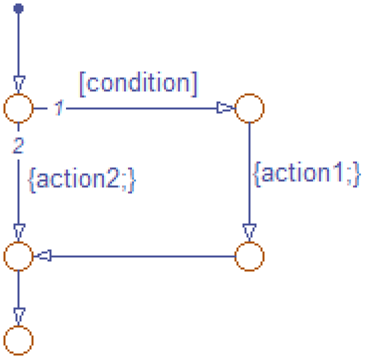
The Pattern Wizard generates MAAB-compliant flow graphs.

Decision Logic Patterns in Flow Graphs

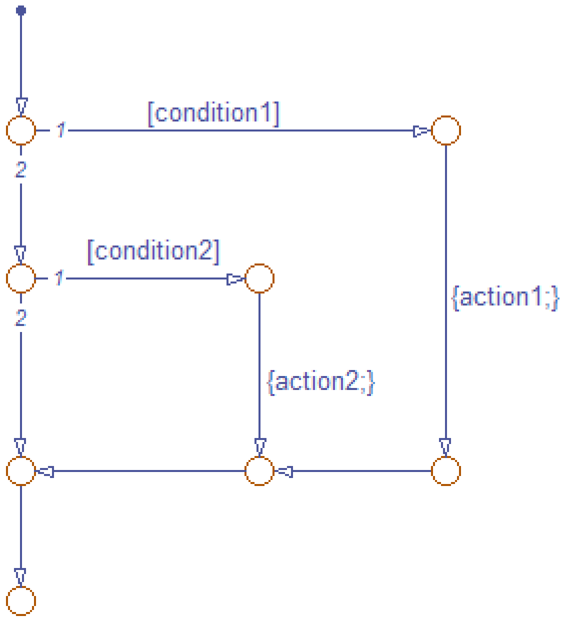
The Pattern Wizard generates the following MAAB-compliant decision logic patterns:



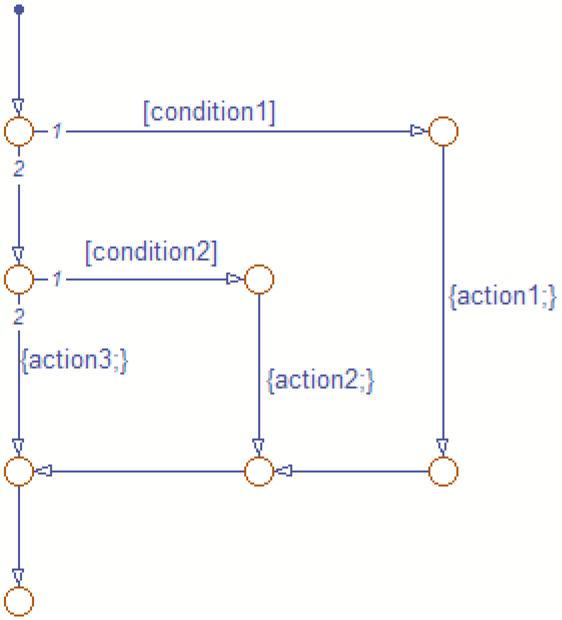
if



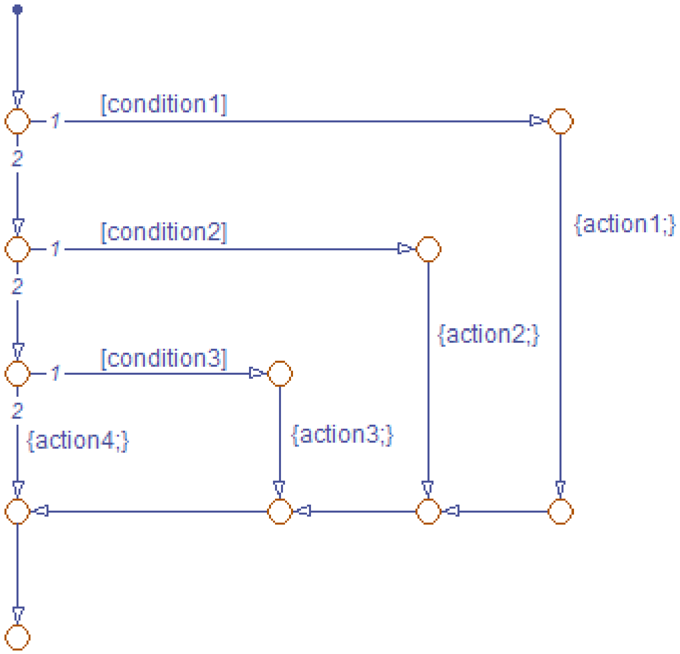
if-else



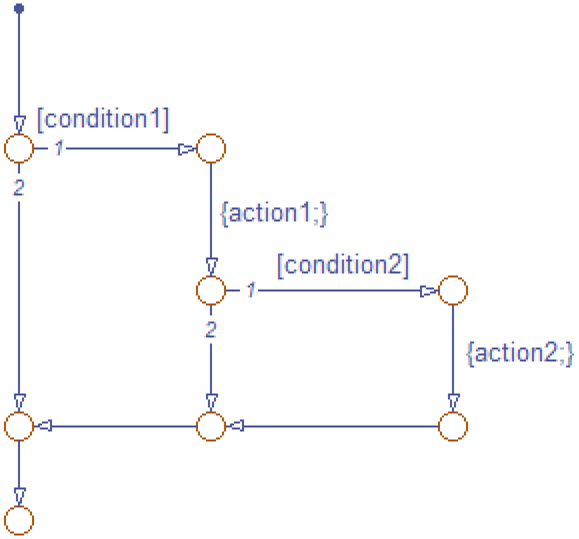
if-elseif



if-elseif-else



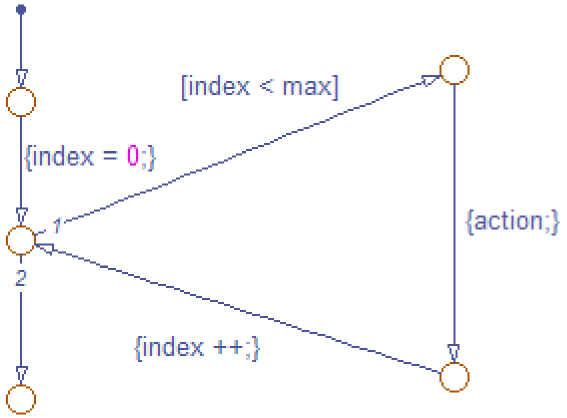
if-elseif-elseif-else



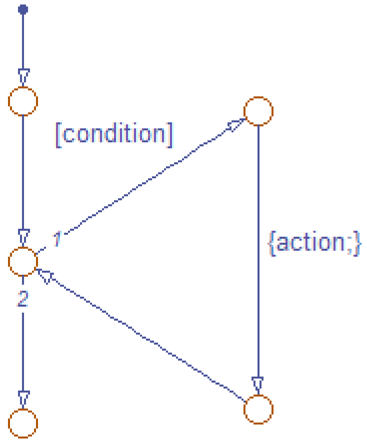
Nested if

Iterative Loop Patterns in Flow Graphs

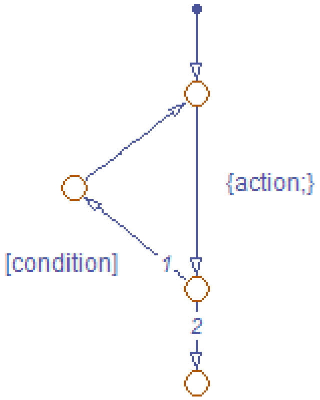
The Pattern Wizard generates the following MAAB-compliant iterative loop patterns:



for



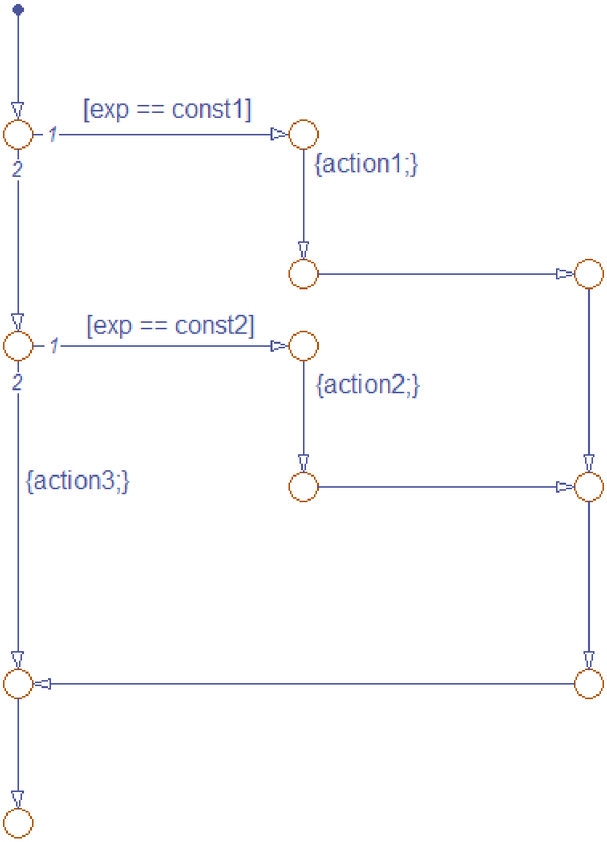
while



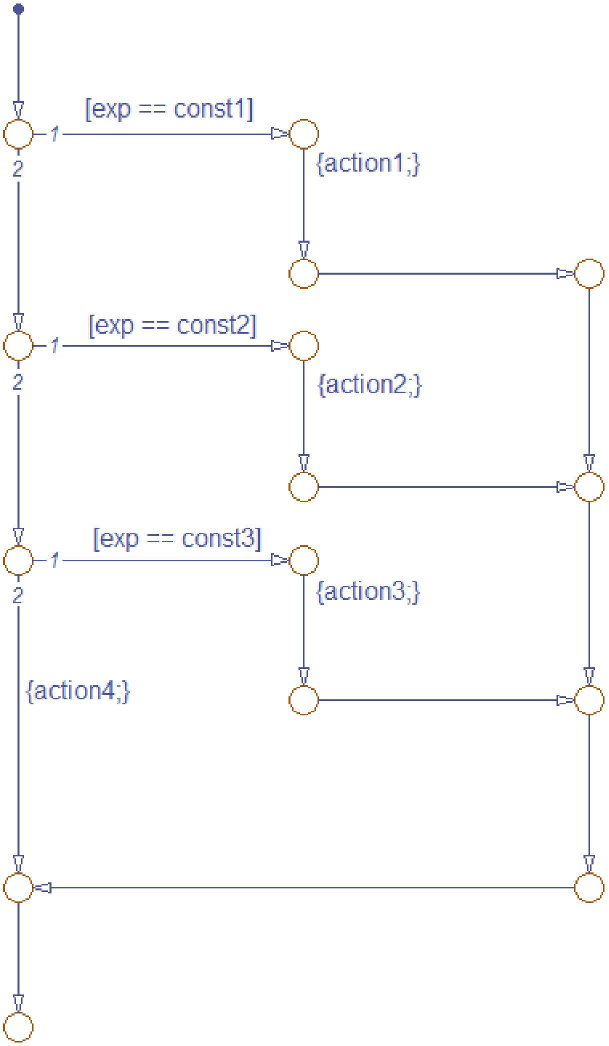
do-while

Switch Patterns in Flow Graphs

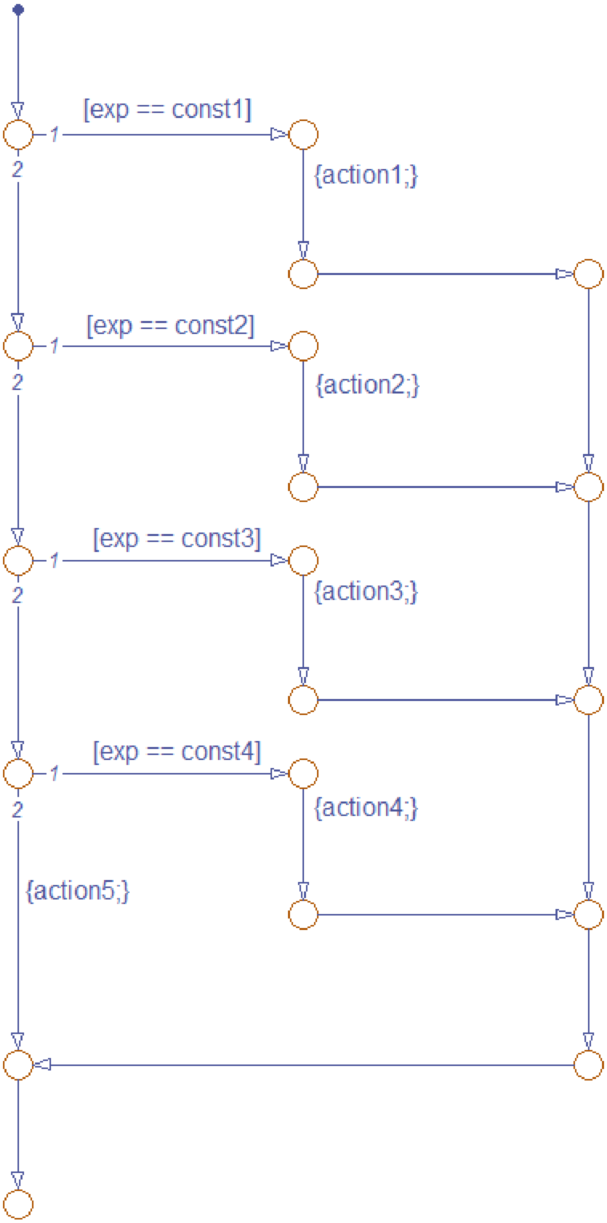
The Pattern Wizard generates the following MAAB-compliant switch patterns:



switch with two cases and default



switch with three cases and default



switch with four cases and default

Try It: Creating and Reusing a Custom Pattern with the Pattern Wizard

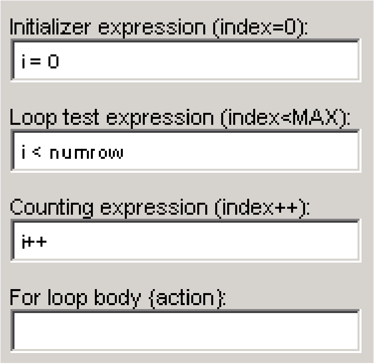
This exercise shows how to create, modify, and save a custom flow graph pattern for iterating over the upper triangle of a two-dimensional matrix. In the upper triangle, the row index *i* is always less than or equal to column index *j*. This flow graph pattern uses nested `for`-loops to ensure that *i* never exceeds *j*.

Creating the Upper Triangle Iterator Pattern

1. Open a new (empty) chart.
2. Select **Patterns > Add Loop > For**.

   The Stateflow Patterns dialog box appears.

3. Enter the initializer, loop test, and counting expressions for iterating through the first dimension of the matrix, as follows:

   Initializer expression (index=0): i = 0

   Loop test expression (index<MAX): i < numrow

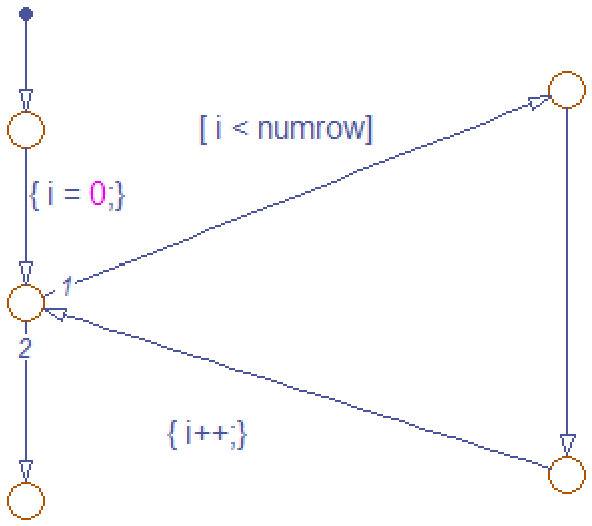
   Counting expression (index++): i++

   For loop body {action}:

   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:

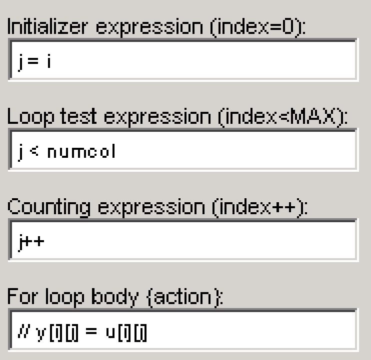
- Conforms to all best practices for creating flow graphs, as described in "Best Practices for Creating Flow Graphs" on page 5-30.
- Provides the correct syntax for conditions and condition actions.

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.



Initializer expression (index=0):

j = i

Loop test expression (index<MAX):

j < numcol

Counting expression (index++):

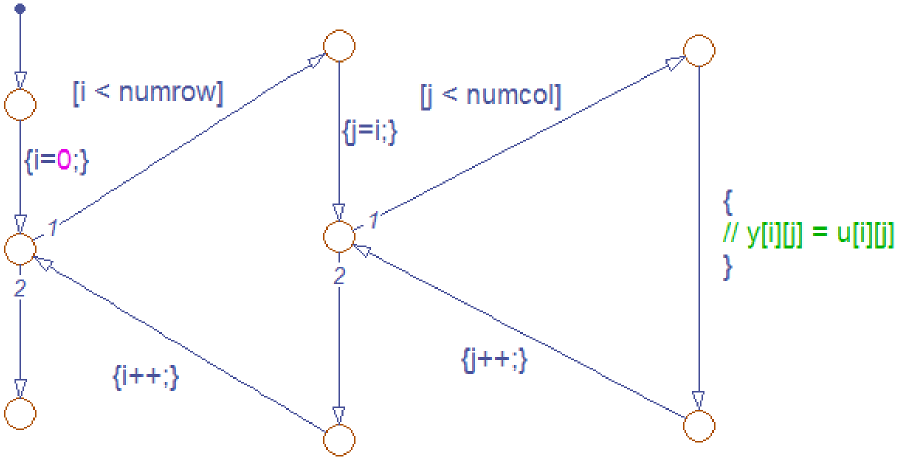
j++

For loop body {action}:

// y[i][j] = u[i][j]

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:

   e. Right-click the transition and select **Execution Order**.

   f. 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 5-23).

Saving the Upper Triangle Iterator Pattern for Reuse

1. Create a folder for storing flow graph patterns, as described in "Guidelines for Creating a Pattern Folder" on page 5-7.
2. Open the 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 editor, select **Patterns > Save Pattern** and take one of these actions:

If you have...	Then Pattern Wizard...	Action
Not yet designated the pattern folder	Prompts you to create or select a pattern folder	Select the folder you just created. See "How to Save Flow Graph Patterns for Easy Retrieval" on page 5-7.
Already designated the pattern folder	Prompts you to save your pattern to the designated folder	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 chart.

2 Drag a graphical function into the chart from the object palette and enter the following function signature:

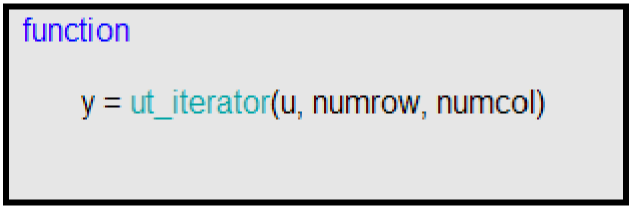
```
function y = ut_iterator(u, numrow, numcol)
```

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**.

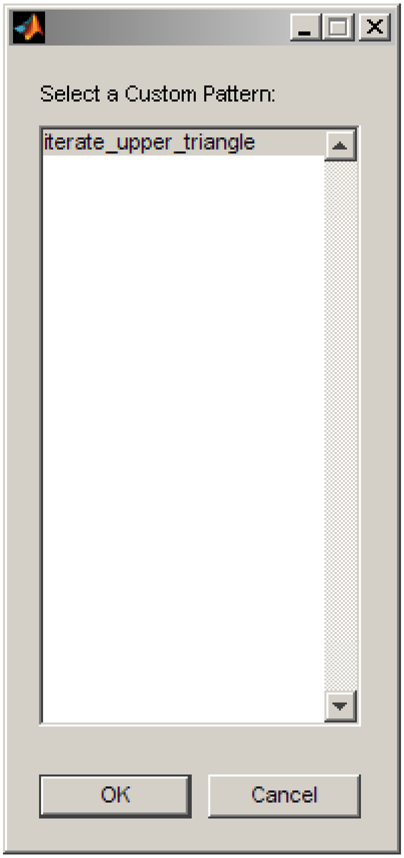
The function should look like this:



```
function

    y = ut_iterator(u, numrow, numcol)
```

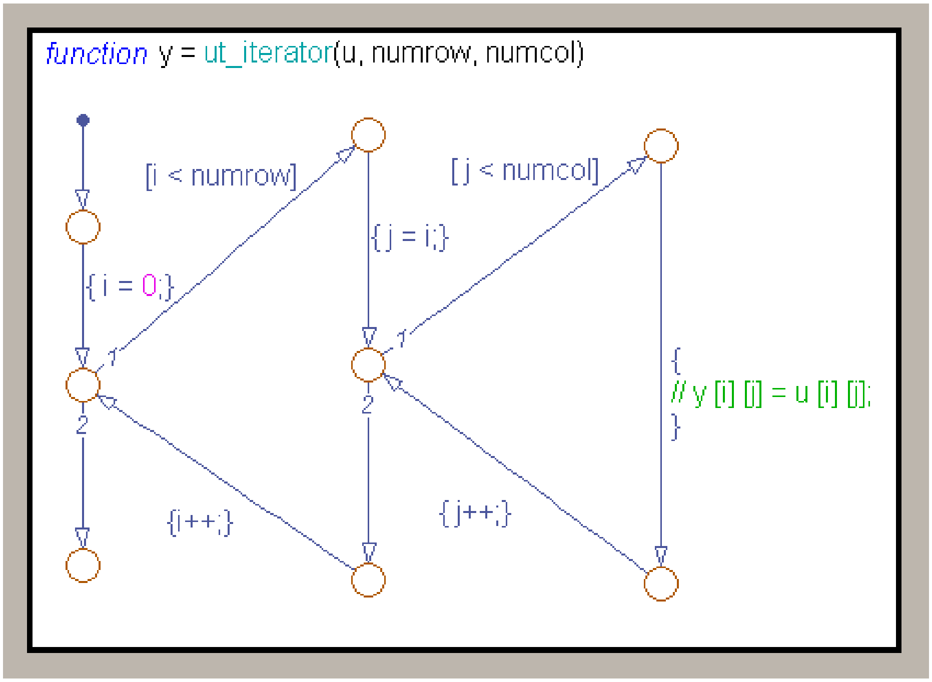
4 Double-click to open the subcharted function and select **Patterns > Add Custom**.

The Select a Custom Pattern dialog box appears, listing all the patterns you saved in your pattern folder.



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 chart, be sure to modify data names, types, and sizes as necessary and substitute an appropriate action.

Drawing and Customizing Flow Graphs By Hand

In this section...

How to Draw a Flow Graph

You can draw and customize flow graphs manually by using connective junctions as branch points between alternate transition paths:

1. Open a chart.
2. From the editor toolbar, drag one or more connective junctions into the chart 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.

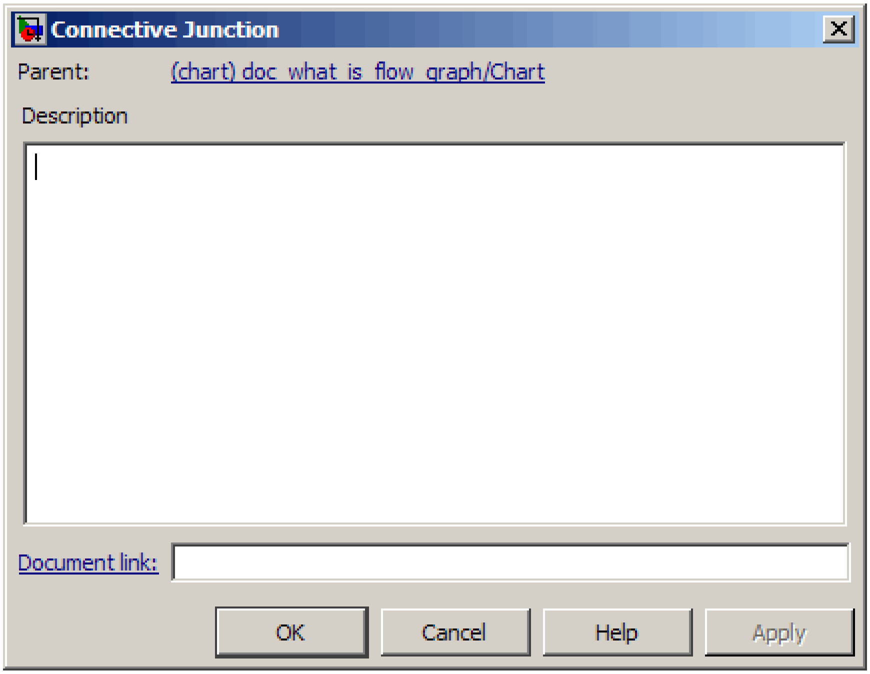
   A menu of junction sizes appears.
3. Select a junction size.

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 reaches the termination point.

Provide an unconditional transition from every junction except the terminating junction

This guideline ensures that unintended backtracking behavior does not occur in a flow graph. If unintended backtracking occurs during simulation, a warning message appears.

You can control the level of diagnostic action for unintended backtracking in the **Diagnostics > Stateflow** pane of the Configuration Parameters dialog box. For more information, see the documentation for the "Unexpected backtracking" diagnostic.

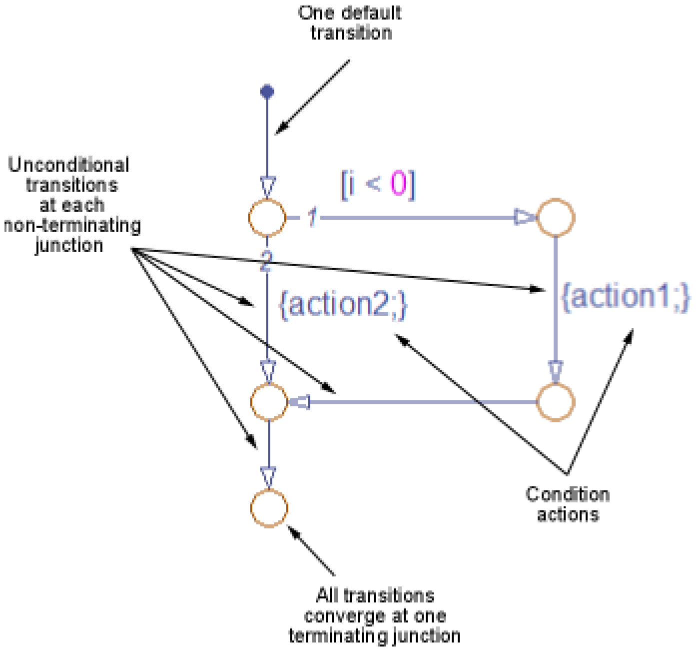
Unintended backtracking can occur at a junction under these conditions:

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

Use condition actions to process updates, not transition actions

Flow graphs test transitions, but do not execute them (and, therefore, never execute transition actions).

The following example illustrates these best practices:



Enhancing Readability of Generated Code for Flow Graphs

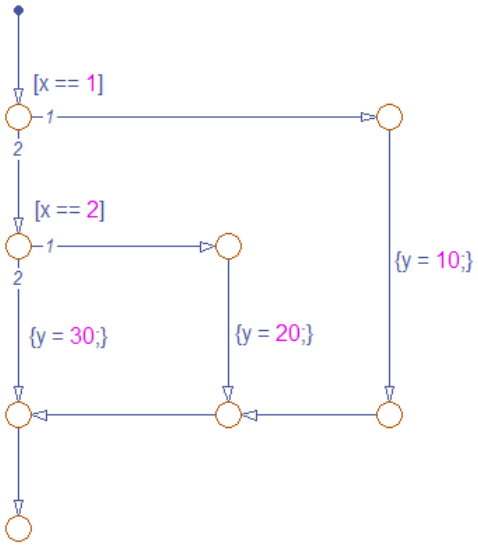
In this section...

Appearance of Generated Code for Flow Graphs

When you use Embedded Coder™ software to generate code for embedded real-time (`ert`) targets, the code from a flow graph resembles the samples that follow.

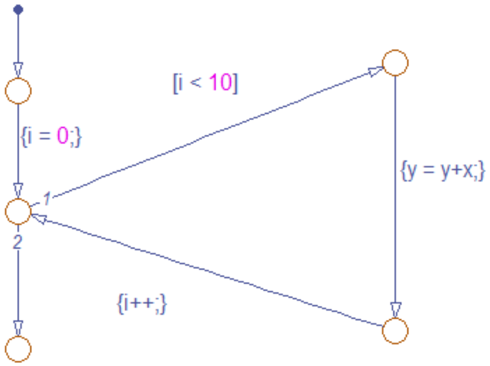
The following characteristics apply:

- By default, the generated code uses `if-elseif-else` statements to represent `switch` patterns. To convert the code to use `switch-case` statements, see "Converting If-Elseif-Else Code to Switch-Case Statements" on page 5-36.
- By default, variables that appear in the flow graph do not retain their names in the generated code. Modified identifiers guarantee that no naming conflicts occur.
- Traceability comments for the transitions appear between each set of `/*` and `*/` markers. To learn more about traceability, see "Traceability of Stateflow Objects in Generated Code" on page 25-79.



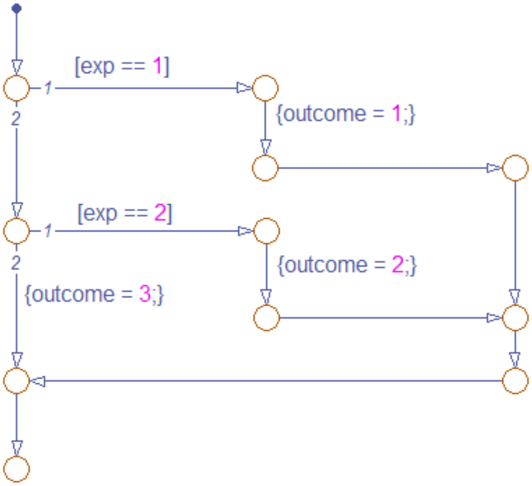
```
if (modelname_U.In1 == 1.0) {
  /* Transition: '<S1>:11' */
  /* Transition: '<S1>:12' */
  modelname_Y.Out1 = 10.0;

  /* Transition: '<S1>:15' */
  /* Transition: '<S1>:16' */
} else {
  /* Transition: '<S1>:10' */
  if (modelname_U.In1 == 2.0) {
    /* Transition: '<S1>:13' */
    /* Transition: '<S1>:14' */
    modelname_Y.Out1 = 20.0;

    /* Transition: '<S1>:16' */
  } else {
    /* Transition: '<S1>:17' */
    modelname_Y.Out1 = 30.0;
  }
}
```

Sample Code for a Decision Logic Pattern



```
for (sf_i = 0; sf_i < 10; sf_i++) {
  /* Transition: '<S1>:40' */
  /* Transition: '<S1>:41' */
  modelname_B.y = modelname_B.y +
    modelname_U.In1;

  /* Transition: '<S1>:39' */
}
```

Sample Code for an Iterative Loop Pattern



```
if (modelname_U.In1 == 1.0) {
  /* Transition: '<S1>:149' */
  /* Transition: '<S1>:150' */
  modelname_Y.Out1 = 1.0;

  /* Transition: '<S1>:151' */
  /* Transition: '<S1>:152' */
  /* Transition: '<S1>:158' */
  /* Transition: '<S1>:159' */
} else {
  /* Transition: '<S1>:156' */
  if (modelname_U.In1 == 2.0) {
    /* Transition: '<S1>:153' */
    /* Transition: '<S1>:154' */
    modelname_Y.Out1 = 2.0;

    /* Transition: '<S1>:155' */
    /* Transition: '<S1>:158' */
    /* Transition: '<S1>:159' */
  } else {
    /* Transition: '<S1>:161' */
    modelname_Y.Out1 = 3.0;
  }
}
```

Sample Code for a Switch Pattern

Converting If-Elseif-Else Code to Switch-Case Statements

When you generate code for embedded real-time targets, you can choose to convert `if-elseif-else` code to `switch-case` statements. This conversion can enhance readability of the code. For example, when a flow graph contains a long list of conditions, the `switch-case` structure:

- Reduces the use of parentheses and braces
- Minimizes repetition in the generated code

How to Convert If-Elseif-Else Code to Switch-Case Statements

The following procedure describes how to convert generated code for the flow graph from `if-elseif-else` to `switch-case` statements.

Step	Task	Reference
1	Verify that your flow graph follows the rules for conversion.	"Verifying the Contents of the Flow Graph" on page 5-41
2	Enable the conversion.	"Enabling the Conversion" on page 5-42
3	Generate code for your model.	"Generating Code for Your Model" on page 5-43
4	Troubleshoot the generated code. • If you see `switch-case` statements for your flow graph, you can stop. • If you see `if-elseif-else` statements for your flow graph, update the chart and repeat the previous step.	"Troubleshooting the Generated Code" on page 5-43

Rules of Conversion

For the conversion to occur, the following rules must hold. LHS and RHS refer to the left-hand side and right-hand side of a condition, respectively.

Construct	Rules to Follow
Flow graph	Must have two or more *unique* conditions, in addition to a default. For more information, see “How the Conversion Handles Duplicate Conditions” on page 5-37.
Each condition	Must test equality only.
	Must use the same variable or expression for the LHS. **Note** You can reverse the LHS and RHS.
Each LHS	Must be a single variable or expression.
	Cannot be a constant.
	Must have an integer or enumerated data type.
	Cannot have any side effects on simulation. For example, the LHS can read from but not write to global variables.
Each RHS	Must be a constant.
	Must have an integer or enumerated data type.

How the Conversion Handles Duplicate Conditions

If a flow graph has duplicate conditions, the conversion preserves only the first condition. The code discards all other instances of duplicate conditions.

After removal of duplicates, two or more unique conditions must exist. If not, no conversion occurs and the code contains all duplicate conditions.

Example of Generated Code

```
if (x == 1) {
    block1
} else if (x == 2) {
    block2
} else if (x == 1) {  // duplicate
    block3
} else if (x == 3) {
    block4
} else if (x == 1) {  // duplicate
    block5
} else {
    block6
}
```

Code After Conversion

```
switch (x) {
    case 1:
     block1; break;
    case 2:
     block2; break;
    case 3:
     block4; break;
    default:
     block6; break;
}
```

Example of Generated Code

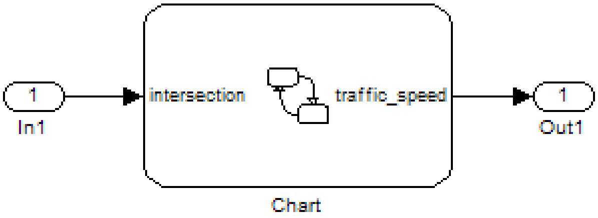
```
if (x == 1) {
    block1
} else if (x == 1) {  // duplicate
    block2
} else {
    block3
}
```

Code After Conversion

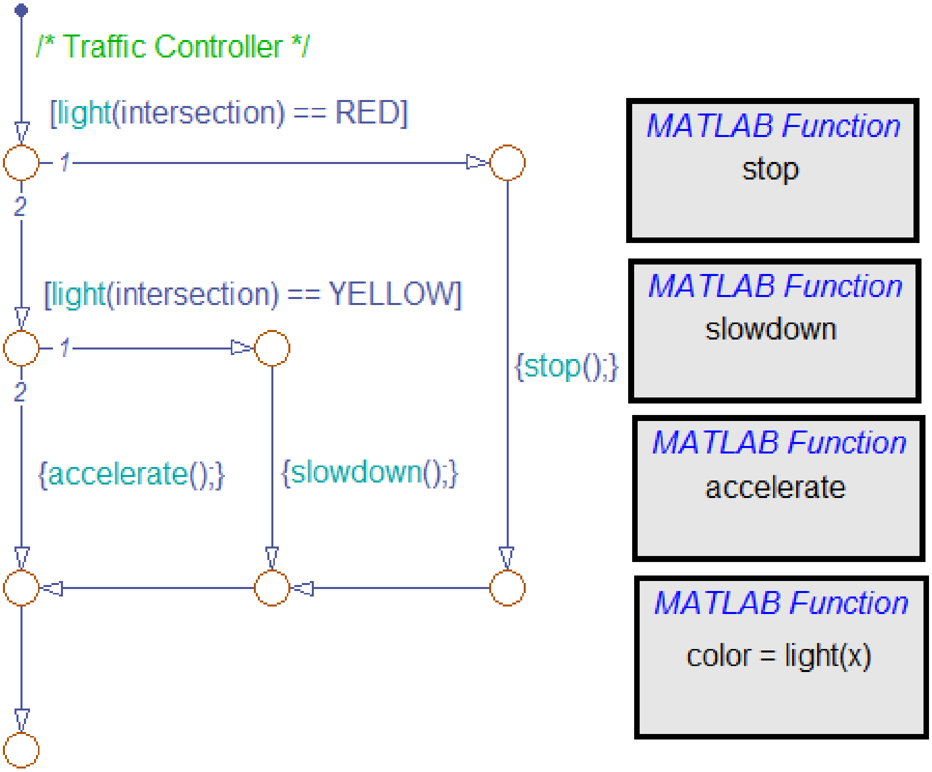
No change, because only one unique condition exists

Example of Converting Code for If-Elseif-Else Decision Logic to Switch-Case Statements

Suppose that you have the following model with a single chart.



The chart contains a flow graph and four MATLAB functions:



The MATLAB functions in the chart contain the code in the following table. In each case, the **Function Inline Option** is `Auto`. For more information about function inlining, see "Specifying Graphical Function Properties" on page 7-47.

MATLAB Function	Code

`stop`

```
function stop
%#codegen
coder.extrinsic('disp');
disp('Not moving.')

traffic_speed = 0;
```

`slowdown`

```
function slowdown
%#codegen
coder.extrinsic('disp')
disp('Slowing down.')

traffic_speed = 1;
```

`accelerate`

```
function accelerate
%#codegen
coder.extrinsic('disp');
disp('Moving along.')

traffic_speed = 2;
```

`light`

```
function color = light(x)
%#codegen
if (x < 20)
    color = TrafficLights.GREEN;
elseif (x >= 20 && x < 25)
    color = TrafficLights.YELLOW;
else
    color = TrafficLights.RED;
end
```

The output `color` of the function `light` uses the enumerated type `TrafficLights`. The enumerated type definition in `TrafficLights.m` is:

```
classdef(Enumeration) TrafficLights < Simulink.IntEnumType
  enumeration
    RED(0)
    YELLOW(5)
    GREEN(10)
  end
end
```

For more information, see "How to Define Enumerated Data in a Stateflow Chart" on page 15-8.

Verifying the Contents of the Flow Graph

Check that the flow graph in your chart follows all the rules in "Rules of Conversion" on page 5-36.

Construct	How the Construct Follows the Rules
Flow graph	Two unique conditions exist, in addition to the default: • `[light(intersection) == RED]` • `[light(intersection) == YELLOW]`
Each condition	Each condition: • Tests equality • Uses the same function call `light(intersection)` for the LHS

Construct	How the Construct Follows the Rules
Each LHS	Each LHS: • Contains a single expression • Is the output of a function call and therefore not a constant • Is of enumerated type `TrafficLights`, which you define in `TrafficLights.m` on the MATLAB path (see "How to Define Enumerated Data in a Stateflow Chart" on page 15-8) • Uses a function call that has no side effects
Each RHS	Each RHS: • Is an enumerated value and therefore a constant • Is of enumerated type `TrafficLights`

Enabling the Conversion

1. Open the Configuration Parameters dialog box.
2. In the **Code Generation** pane, select `ert.tlc` for the **System target file**.

   This step specifies an ERT-based target for your model.
3. In the **Code Generation > Code Style** pane, select the **Convert if-elseif-else patterns to switch-case statements** check box.

Tip This conversion works on a per-model basis. If you select this check box, the conversion applies to:

- Flow graphs in all charts of a model
- MATLAB functions in all charts of a model
- All MATLAB Function blocks in that model

Generating Code for Your Model

In the **Code Generation** pane of the Configuration Parameters dialog box, click **Build** in the lower right corner.

Troubleshooting the Generated Code

The generated code for the flow graph appears something like this:

```
if (sf_color == RED) {
  /* Transition: '<S1>:11' */
  /* Transition: '<S1>:12' */
  /* MATLAB Function 'stop': '<S1>:23' */
  /* '<S1>:23:6' */
  rtb_traffic_speed = 0;

  /* Transition: '<S1>:15' */
  /* Transition: '<S1>:16' */
} else {
  /* Transition: '<S1>:10' */
  /* MATLAB Function 'light': '<S1>:19' */
  if (ifelse_using_enums_U.In1 < 20.0) {
    /* '<S1>:19:3' */
    /* '<S1>:19:4' */
    sf_color = GREEN;
  } else if ((ifelse_using_enums_U.In1 >= 20.0) &&
             (ifelse_using_enums_U.In1 < 25.0)) {
    /* '<S1>:19:5' */
    /* '<S1>:19:6' */
    sf_color = YELLOW;
  } else {
    /* '<S1>:19:8' */
    sf_color = RED;
  }

  if (sf_color == YELLOW) {
    /* Transition: '<S1>:13' */
    /* Transition: '<S1>:14' */
    /* MATLAB Function 'slowdown': '<S1>:24' */
    /* '<S1>:24:6' */
    rtb_traffic_speed = 1;
```

```
    /* Transition: '<S1>:16' */
  } else {
    /* Transition: '<S1>:17' */
    /* MATLAB Function 'accelerate': '<S1>:25' */
    /* '<S1>:25:6' */
    rtb_traffic_speed = 2;
  }
}
```

Because the MATLAB function `light` appears inlined, inequality comparisons appear in these lines of code:

```
if (ifelse_using_enums_U.In1 < 20.0) {
....
} else if ((ifelse_using_enums_U.In1 >= 20.0) &&
             (ifelse_using_enums_U.In1 < 25.0)) {
....
```

Because inequalities appear in the body of the `if-elseif-else` code for the flow graph, the conversion to `switch-case` statements does not occur. To prevent this behavior, do one of the following:

- Specify that the function `light` does not appear inlined. See "Changing the Inlining Property for the Function" on page 5-44.
- Modify the flow graph. See "Modifying the Flow Graph to Ensure Switch-Case Statements" on page 5-46.

Changing the Inlining Property for the Function. If you do not want to modify your flow graph, change the inlining property for the function `light`:

1 Right-click the function box for `light` and select **Properties**.

  The properties dialog box appears.

2 For **Function Inline Option**, select `Function`.

3 Click **OK** to close the dialog box.

Note You do not have to change the inlining property for the other three MATLAB functions in the chart. Because the flow graph does not call those functions during evaluation of conditions, the inlining property for those functions can remain `Auto`.

When you regenerate code for your model, the code for the flow graph now appears something like this:

```
switch (ifelse_using_enums_light(ifelse_using_enums_U.In1)) {
 case RED:
  /* Transition: '<S1>:11' */
  /* Transition: '<S1>:12' */
  /* MATLAB Function 'stop': '<S1>:23' */
  /* '<S1>:23:6' */
  ifelse_using_enums_Y.Out1 = 0.0;

  /* Transition: '<S1>:15' */
  /* Transition: '<S1>:16' */
  break;

 case YELLOW:
  /* Transition: '<S1>:10' */
  /* Transition: '<S1>:13' */
  /* Transition: '<S1>:14' */
  /* MATLAB Function 'slowdown': '<S1>:24' */
  /* '<S1>:24:6' */
  ifelse_using_enums_Y.Out1 = 1.0;

  /* Transition: '<S1>:16' */
  break;

 default:
  /* Transition: '<S1>:17' */
  /* MATLAB Function 'accelerate': '<S1>:25' */
  /* '<S1>:25:6' */
  ifelse_using_enums_Y.Out1 = 2.0;
  break;
}
```

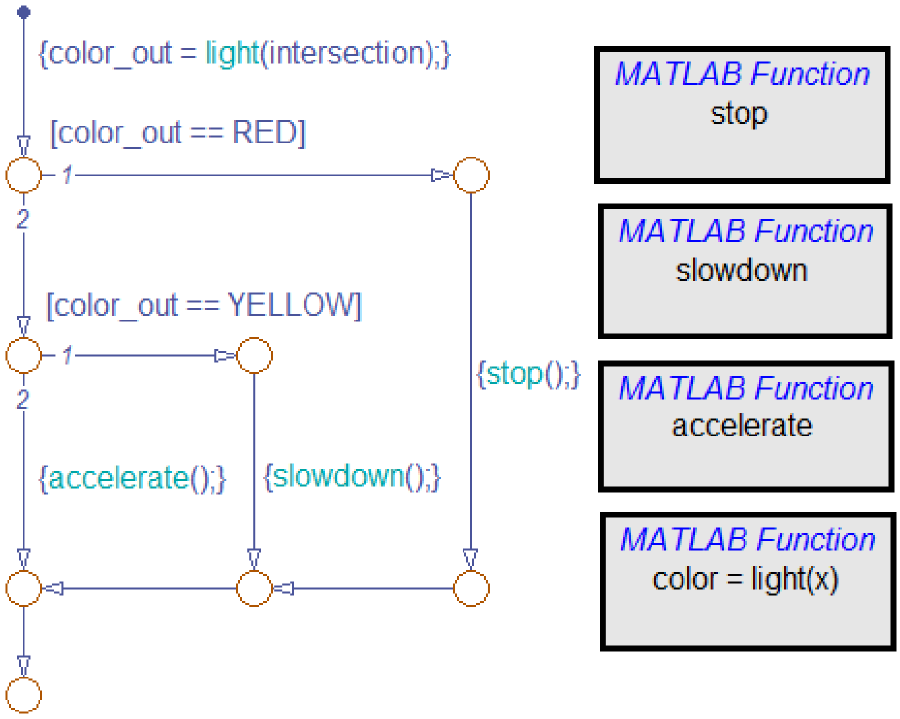
Because the MATLAB function `light` no longer appears inlined, the conversion to `switch-case` statements occurs. The `switch-case` statements provide the following benefits to enhance readability:

- The code reduces the use of parentheses and braces.
- The LHS expression `ifelse_using_enums_light(ifelse_using_enums_U.In1)` appears only once, minimizing repetition in the code.

Modifying the Flow Graph to Ensure Switch-Case Statements. If you do not want to change the inlining property for the function `light`, modify your flow graph:

1. Add chart local data `color_out` with the enumerated type `TrafficLights`.
2. Replace each instance of `light(intersection)` with `color_out`.
3. Add the action `{color_out = light(intersection)}` to the default transition of the flow graph.

The chart should now look something like this:



When you regenerate code for your model, the code for the flow graph uses `switch-case` statements.

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 HDL.

Note To generate HDL code from Stateflow charts, you must use HDL Coder™ software, which is available separately.

Running a Demo of Mealy and Moore Machines

Stateflow software ships with a model that shows how to use Mealy and Moore machines for sequence recognition in signal processing. To run the demo:

1. At the MATLAB prompt, type:

   ```
   demo simulink stateflow
   ```

   The Help browser shows a list of demos you can access.

2. Click the demo titled **Sequence Recognition Using Mealy and Moore Charts**.

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

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 summary describes the earliest time at which Mealy and Moore charts can compute outputs:

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

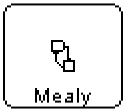
To create a new Mealy or Moore chart, follow these steps:

1. Add a new Chart block to a Simulink model; then double-click the block to open the Stateflow Editor.
2. Right-click in an empty area of the chart and select **Properties**.

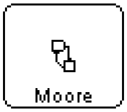
   The Chart Properties dialog box opens.
3. From the **State Machine Type** drop-down menu, select `Mealy` or `Moore`.
4. Click **OK**.

   The chart icon updates to display the selected chart type:

Mealy



Moore



5. Design your chart according to the guidelines for the chart type (see "Design Considerations for Mealy Charts" on page 6-7 and "Design Considerations for Moore Charts" on page 6-13.

Design Considerations for Mealy Charts

In this section...
"Mealy Semantics" on page 6-7
"Design Rules for Mealy Charts" on page 6-7
"Example: Mealy Vending Machine" on page 6-10

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 chart provides one time base for input and clock (see "Calculate Output and State Using One Time Base" on page 6-10).

- 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" on page 6-8
- "Do Not Use State Actions or Transition Actions" on page 6-8
- "Restrict Use of Data" on page 6-8
- "Restrict Use of Events" on page 6-9
- "Calculate Output and State Using One Time Base" on page 6-10

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” on page 6-8
- “Do Not Define Data Store Memory” on page 6-9

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

Limit the use of events in Mealy charts as follows:

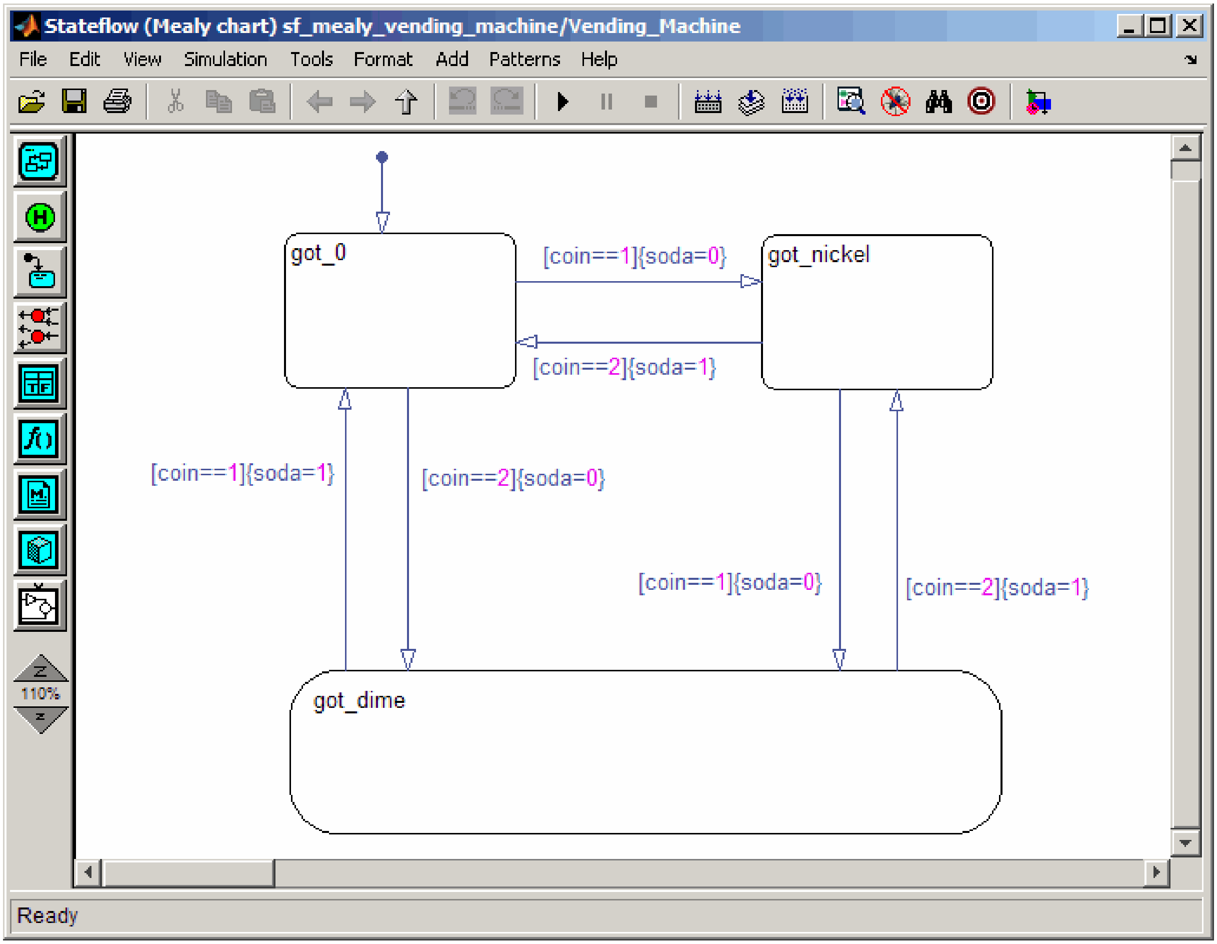
Do:	**Do Not:**
Use input events to trigger the chart	Broadcast any type of event
Use event-based temporal logic to guard transitions You can use event-based temporal logic in Mealy charts because it behaves synchronously (see "Operators for Event-Based Temporal Logic" on page 10-64). Think of the change 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 "Keywords for Implicit Events" on page 9-40).	Use local events to guard transitions You cannot use local 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.

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 6-14
- "Restrict Data to Inputs, Outputs, and Constants" on page 6-16
- "Reference Input Only in Conditions" on page 6-17
- "Do Not Use Actions on Transitions" on page 6-19

- “Do Not Use Graphical Functions” on page 6-19
- “Do Not Use Truth Tables, MATLAB Functions, or Simulink Functions” on page 6-19
- “Restrict Use of Events” on page 6-19

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 6-14
- “Allow Actions in Leaf States Only” on page 6-15
- “Do Not Label State Actions” on page 6-16

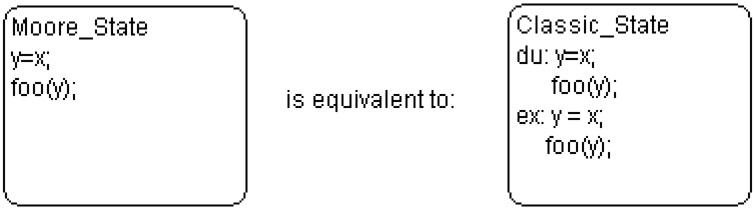
You cannot define actions on transitions because transitions almost always depend on inputs. For example, if you compute outputs in a condition action on a transition, the chart updates outputs whenever there is a change on the input — a violation of Moore semantics.

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.

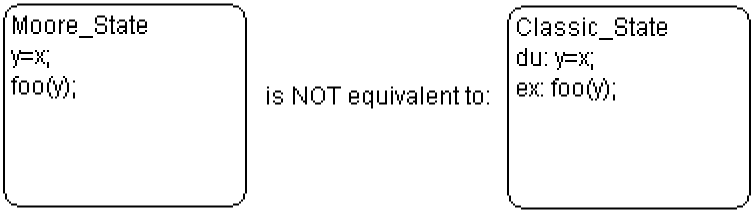
To implement the duality of execution, the `during` and `exit` actions must be identical, as in this example.



```
Moore_State
y=x;
foo(y);
```

```
Classic_State
du: y=x;
    foo(y);
ex: y = x;
    foo(y);
```

Moore states do *not* differentiate between `during` and `exit` actions, as shown here.



```
Moore_State
y=x;
foo(y);
```

```
Classic_State
du: y=x;
ex: foo(y);
```

Note There are no labels on state actions in Moore charts (see "Do Not Label State Actions" on page 6-16).

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 "Types of Chart Execution" on page 3-40 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 6-14. 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" on page 6-16
- "Restrict Machine-Parented Data to Constants and Parameters" on page 6-17
- "Do Not Define Data Store Memory" on page 6-17

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.

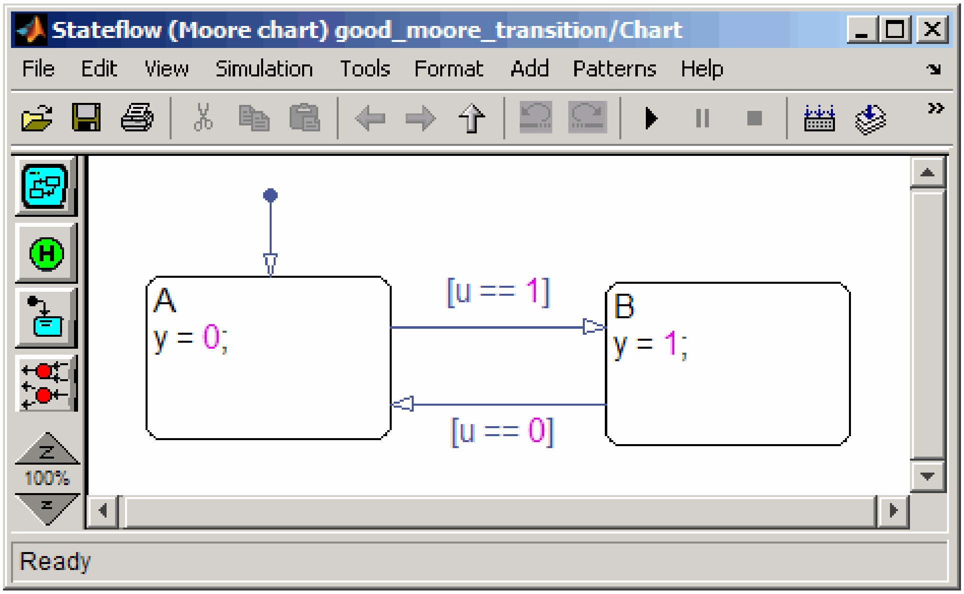
Restrict Machine-Parented Data to Constants and Parameters. Machine-parented data is data that you define for a Stateflow machine, which is the collection of Stateflow blocks in a Simulink model. The Stateflow machine is the highest level of the Stateflow hierarchy. When you define data at this level, 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.

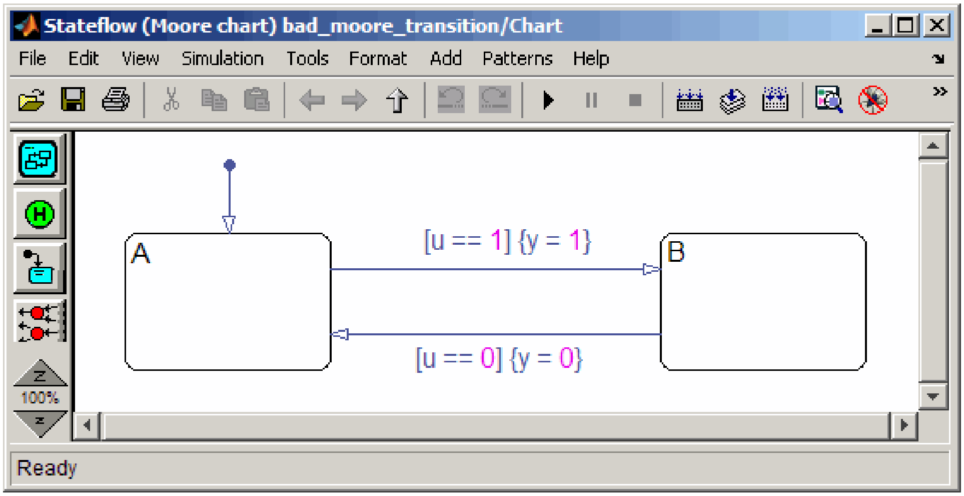
Reference Input Only in Conditions

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 6-17).

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, MATLAB Functions, or Simulink Functions

You cannot use truth tables, MATLAB functions, or Simulink functions in Moore charts. These restrictions prevent violations of Moore semantics during chart execution.

Restrict Use of Events

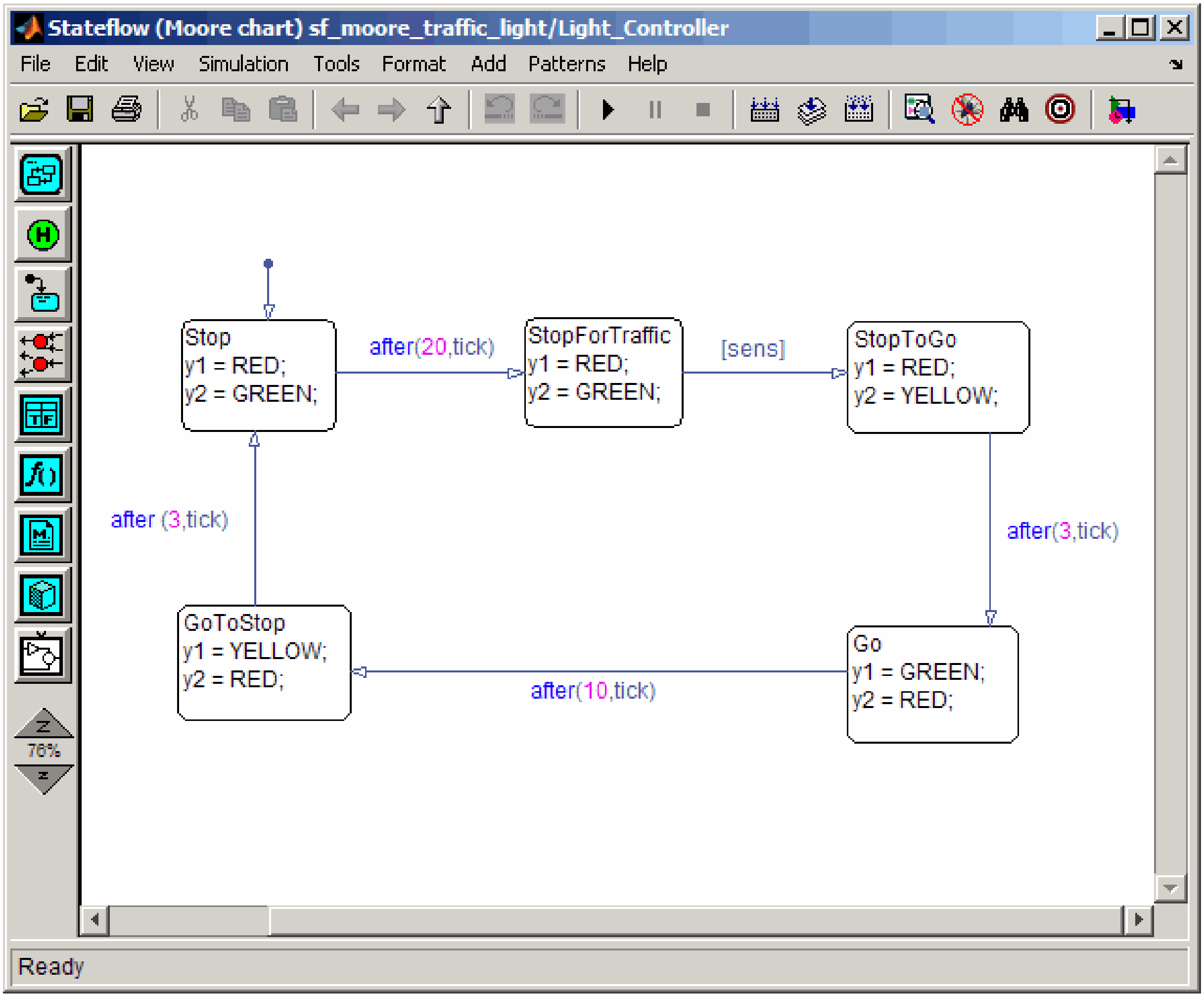
Limit the use of events in Moore charts as follows:

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 10-64). Think of the change in value of a	Use local events to guard transitions You cannot use local 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

Do:	Do Not:
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 "Keywords for Implicit Events" on page 9-40).	compute outputs whenever input changes.

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`).

Effects of Changing the 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 6-25). 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.

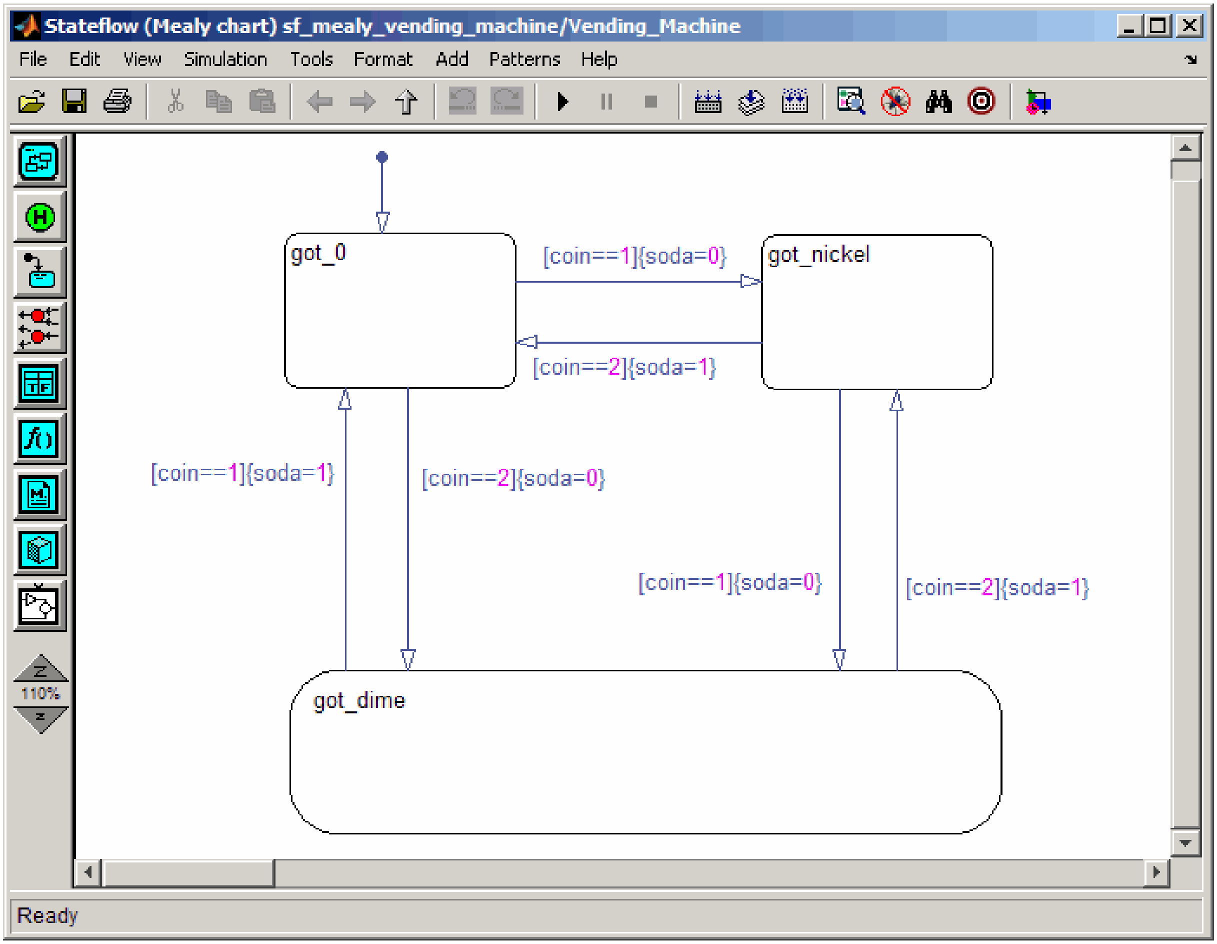
From	**To**	**Result**
Mealy	Classic	Mealy charts retain their semantics when changed to Classic type.
Classic	Mealy	If the Classic chart confirms to Mealy semantic rules, the Mealy chart exhibits equivalent behavior, provided that output is defined at every time step.
Moore	Classic	State actions in 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 6-10.



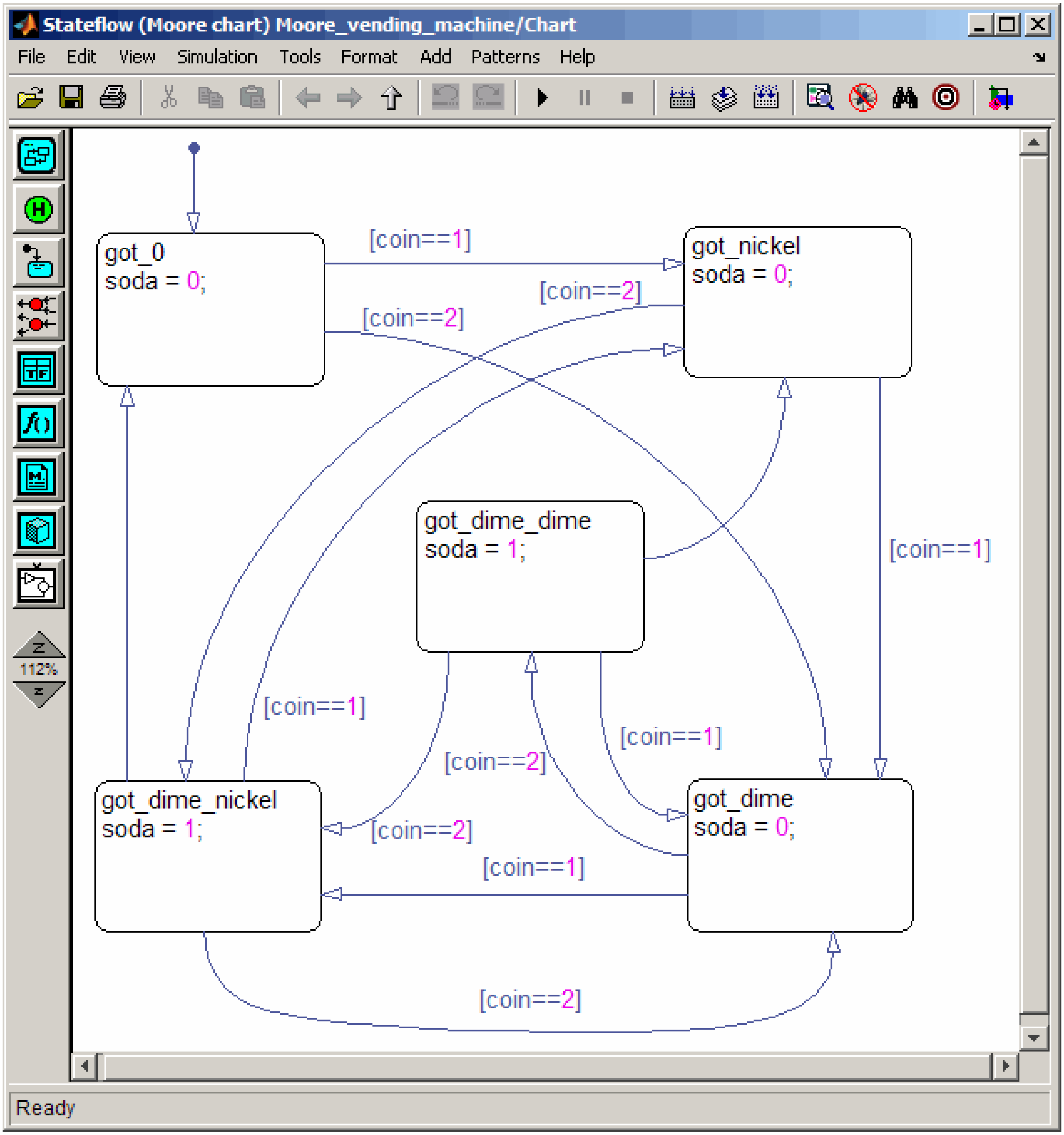
If you change the chart type to **Moore** and rebuild, you get the following diagnostic message:

```
Stateflow Moore chart cannot have condition or transition actions.
```

This indicates 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 6-19). According to Moore semantics, you must instead compute outputs in state actions (see “Design Rules for Moore Charts” on page 6-13).

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.


Stateflow (Moore chart) Moore_vending_machine/Chart
File Edit View Simulation Tools Format Add Patterns Help
112%
got_0
soda = 0;
got_nickel
soda = 0;
got_dime_dime
soda = 1;
got_dime_nickel
soda = 1;
got_dime
soda = 0;
[coin==1]
[coin==2]
[coin==2]
[coin==1]
[coin==1]
[coin==2]
[coin==1]
[coin==2]
[coin==1]
[coin==2]
Ready

The semantics of the two charts differ as follows:

Mealy Vending Machine	**Moore Vending Machine**
Uses 3 states	Uses 5 states
Computes outputs in condition actions	Computes outputs in state actions
Updates output based on input	Updates output before evaluating input, requiring an extra time step to produce the soda

For this vending machine, Mealy is a better modeling paradigm because there is no delay in releasing 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.

Techniques for Streamlining Chart Design

Recording State Activity with History Junctions

In this section…

What Is a History Junction?

A history junction records the activity of substates inside superstates. Use a history junction in a chart or superstate to indicate that its last active substate becomes active when the chart or superstate becomes active.

Creating a History Junction

To create a history junction, do the following:

1. In the editor toolbar, click the History Junction icon:

2. Move your pointer into the chart.
3. Click to place a history junction inside the state whose last active substate it records.

To create multiple history junctions, do the following:

1. In the editor toolbar, double-click the History Junction icon.

   The button is now in multiple-object mode.

2. Click anywhere in the drawing area to place a history junction.
3. Move to and click another location to create an additional history junction.

4. Click the History Junction icon or press the **Esc** key to cancel the operation.

To move a history junction to a new location, click and drag it to the new position.

Changing History Junction Size

To change the size of junctions:

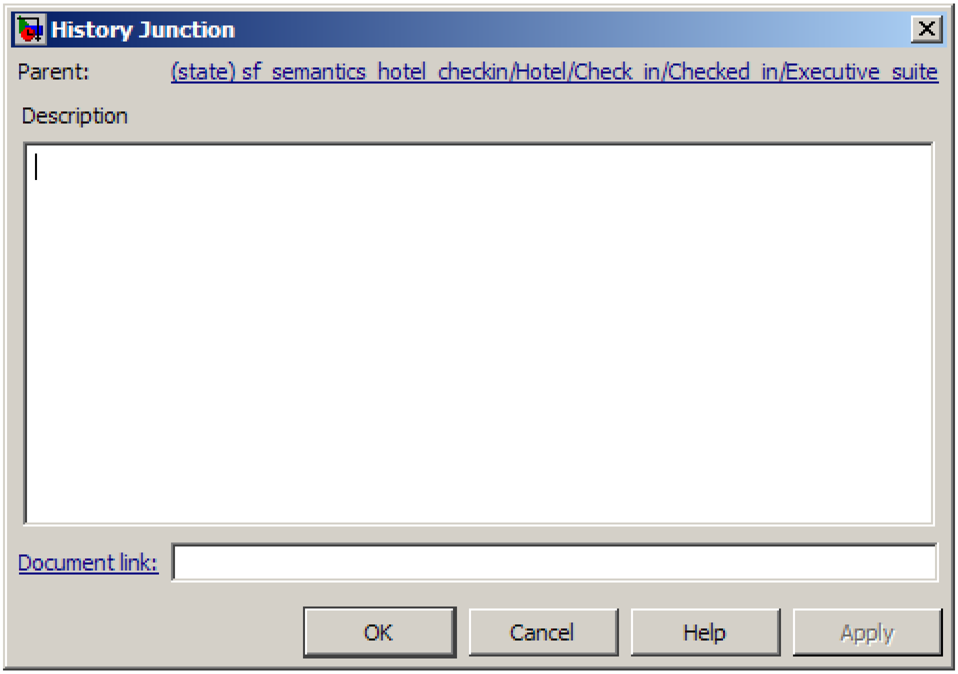
1. Select the history junctions whose size you want to change.
2. Right-click one of the junctions and select **Junction Size**.
3. Select a size from the list of junction sizes.

Changing History Junction Properties

To edit the properties for a junction:

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

   The History Junction dialog box appears.



Parent: (state) sf_semantics_hotel_checkin/Hotel/Check_in/Checked_in/Executive_suite

2 Edit the fields in the properties dialog box.

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

3 When finished editing, click one of the following buttons:

- **Apply** to save the changes
- **Cancel** to cancel any changes

- **OK** to save the changes and close the dialog box
- **Help** to display the Stateflow online help in an HTML browser window

Using Subcharts to Encapsulate Modal Logic

In this section...

What Is a Subchart?

A subchart is a graphical object that can contain anything a top-level chart can, including other subcharts. A subchart, or a subcharted state, is a superstate of the states that it contains. You can nest subcharts to any level in your chart design.

Using subcharts, you can reduce a complex chart to a set of simpler, hierarchically organized units. This design makes the chart easier to understand and maintain, without changing the chart behavior. Subchart boundaries do not apply during simulation and code generation.

The subchart appears as a block with its name in the block center. However, you can define actions and default transitions for subcharts just as you can for superstates. You can also create transitions to and from subcharts just as you can create transitions to and from superstates. You can create transitions between states residing outside a subchart and any state within a subchart. The term *supertransition* refers to a transition that crosses subchart boundaries in this way. See "Moving Between Different Levels of Hierarchy with Supertransitions" on page 7-12 for more information.

Subcharts define a containment hierarchy within a top-level chart. A subchart or top-level chart is the *parent* of the states it contains at the first level and an *ancestor* of all the subcharts contained by its children and their descendants at lower levels.

Some subcharts can be *atomic* if they meet certain modeling requirements. For more information, see Chapter 11, “Making States Reusable with Atomic Subcharts”.

Creating a Subchart

You create a subchart by converting an existing state, box, or graphical function into the subchart. The object to convert can be one that you create for making a subchart or an existing object whose contents you want to turn into a subchart.

To convert a new or existing state, box, or graphical function to a subchart:

1 Right-click the object and select **Make Contents > Subcharted**.

2 Confirm that the object now appears as a subchart.

To convert the subchart back to its original form, right-click the subchart. In the context menu, select **Make Contents > Subcharted**.

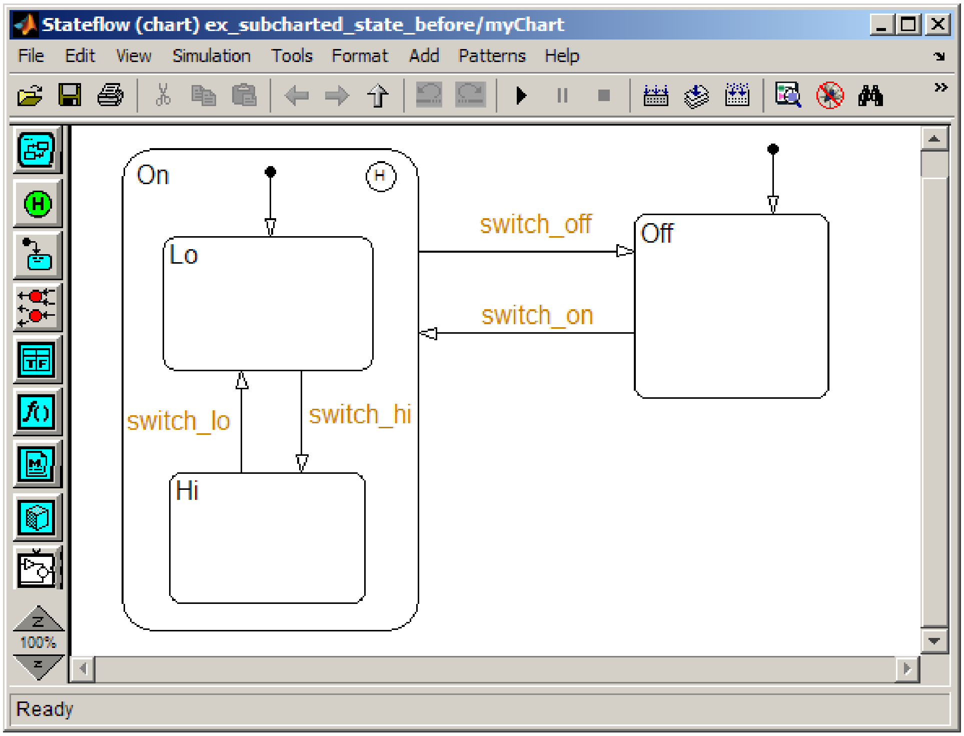
Rules of Subchart Conversion

When you convert a box to a subchart, the subchart retains the attributes of a box. For example, the position of the resulting subchart determines its activation order in the chart if implicit ordering is enabled (see “Grouping Chart Objects with Boxes” on page 7-50 for more information).

You cannot undo the operation of converting a subchart back to its original form. When you perform this operation, the undo and redo buttons are disabled from undoing and redoing any prior operations.

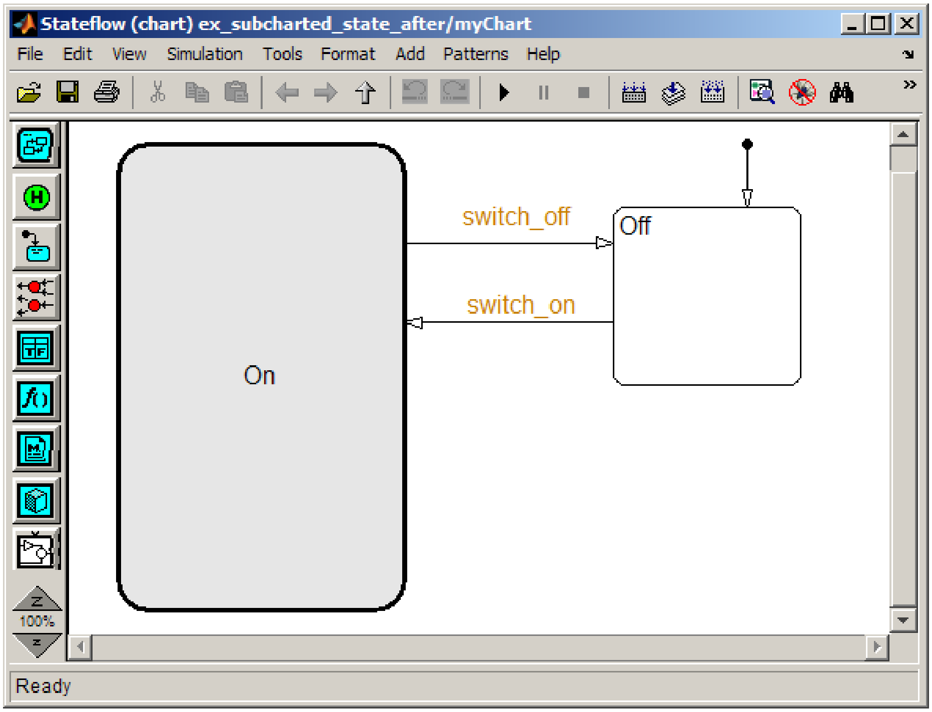
Example of Converting a State to a Subchart

Suppose that you have the following chart:



1 To convert the `On` state to a subchart, right-click the state and select **Make Contents > Subcharted**.

2 Confirm that the `On` state now appears as a subchart.



Manipulating Subcharts as Objects

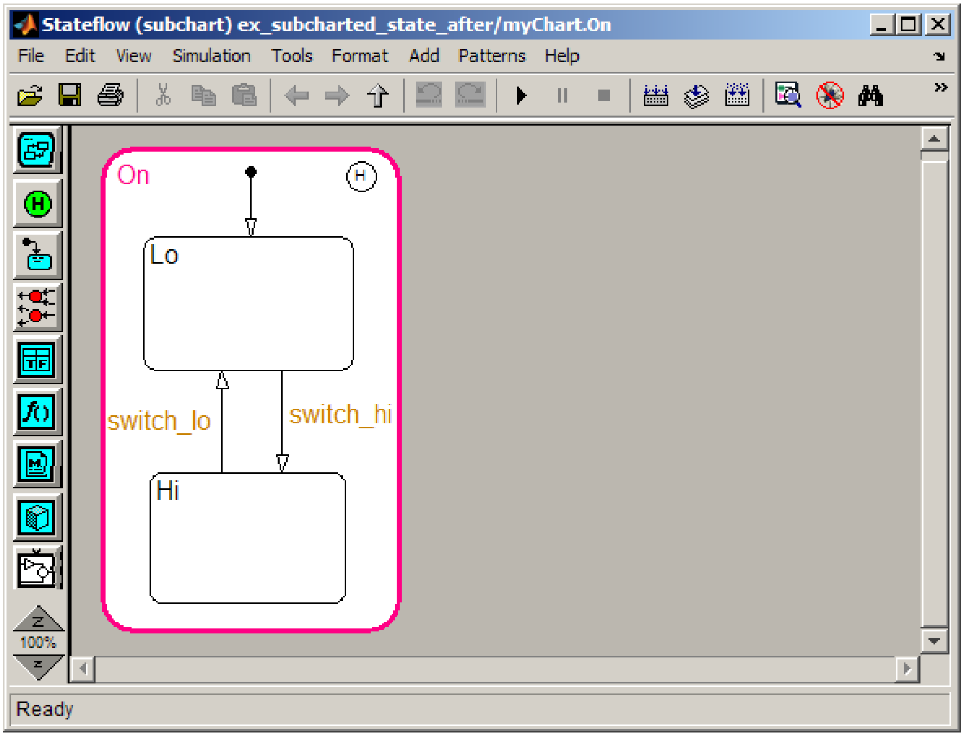
Subcharts also act as individual objects. You can move, copy, cut, paste, relabel, and resize subcharts as you would states and boxes. You can also draw transitions to and from a subchart and any other state or subchart at the same or different levels in the chart hierarchy (see “Moving Between Different Levels of Hierarchy with Supertransitions” on page 7-12).

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.



A shaded border surrounds the contents of the subchart. This border displays supertransitions.

Editing a Subchart

After you open a subchart (see "Opening a Subchart" on page 7-9), 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 “Moving Between Different Levels of Hierarchy with Supertransitions” on page 7-12 for information on creating supertransitions.

Navigating Subcharts

The Stateflow Editor toolbar contains a set of buttons for navigating the subchart hierarchy of a chart.

Tool	**Description**
⇧	If the Stateflow Editor is displaying a subchart, clicking this button replaces the subchart with the subchart’s parent in the Stateflow Editor. If the Stateflow Editor is displaying a top-level chart, clicking this button replaces the chart with the Simulink model window containing that chart.
⇦	Clicking this button shows the chart that you visited before the current chart, so that you can navigate up the hierarchy.
⇨	Clicking this button shows the chart that you visited after visiting the current chart, so that you can navigate down the hierarchy.

Note You can also use the key sequence **..** (that is, press the period key twice) to navigate up to the parent object for a subcharted state, box, or function.

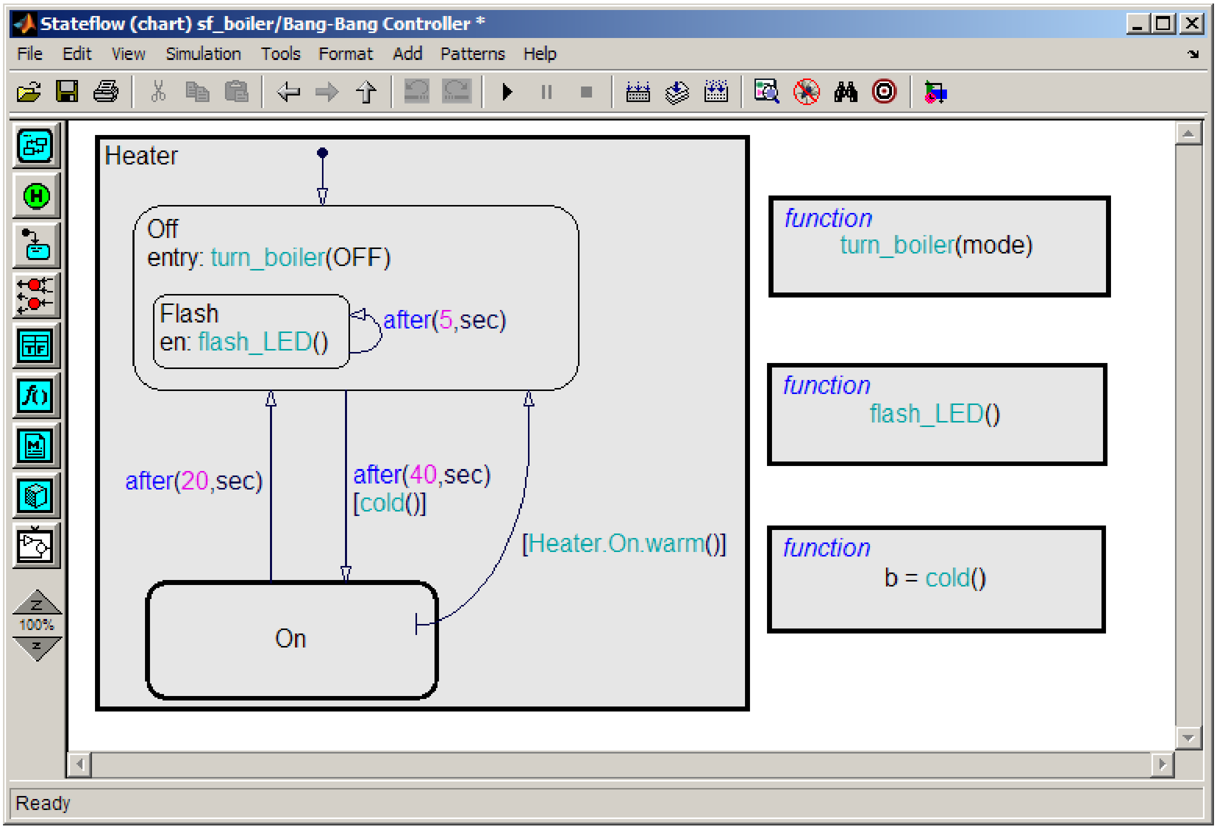
Moving Between Different Levels of Hierarchy with Supertransitions

In this section...

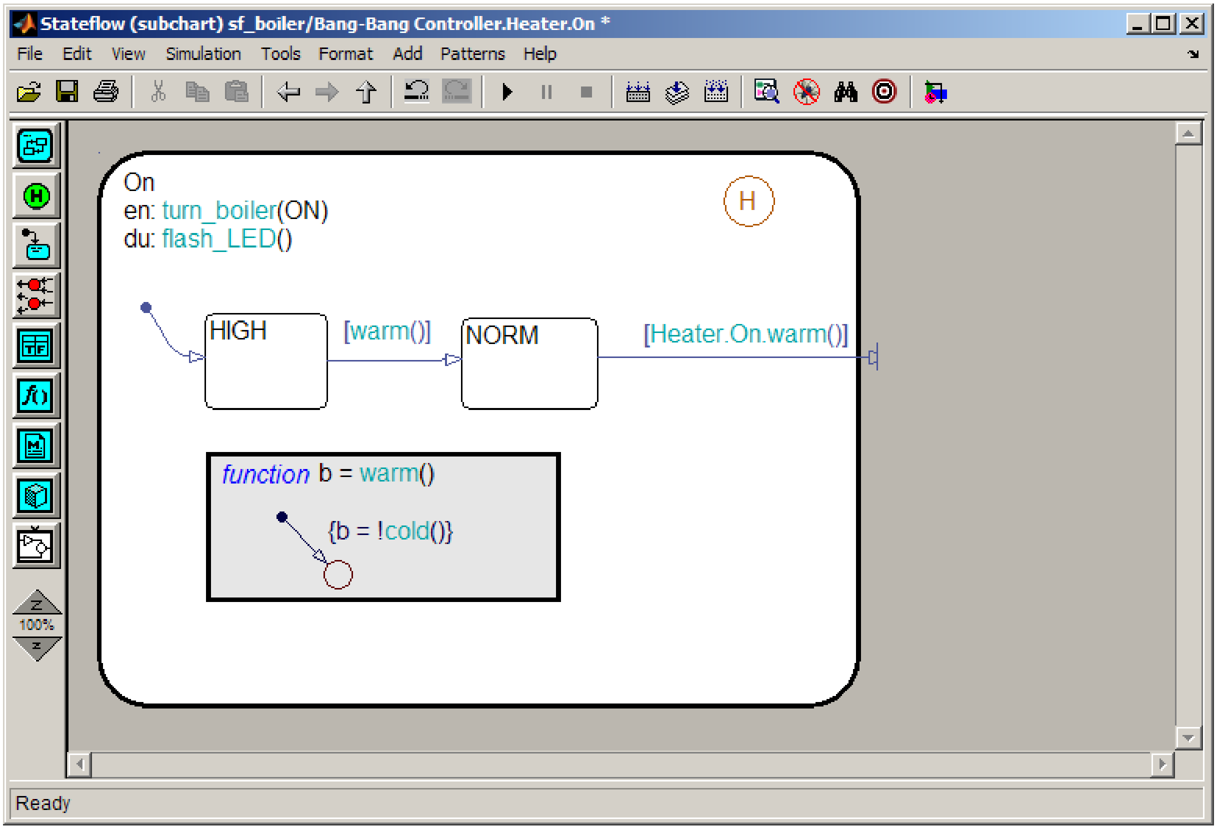
What Is a Supertransition?

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 a supertransition leaving the `On` subchart:



The same supertransition appears inside the subchart as follows:



In this example, supertransition `[Heater.On.warm()]` goes from NORM in the On subchart to the Off state in the parent chart. Both segments of the supertransition have the same label.

Drawing a Supertransition Into a Subchart

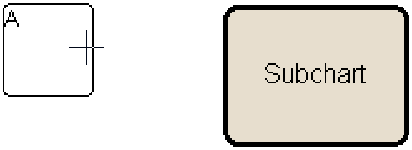
Use the following steps to draw a supertransition from an object outside a subchart to an object inside the subchart.

Note You cannot undo the operation of drawing a supertransition. When you perform this operation, the undo and redo buttons are disabled from undoing and redoing any prior operations.

---

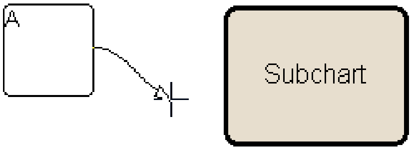
1. Position your pointer over the border of the state.

   The pointer assumes the crosshairs shape.



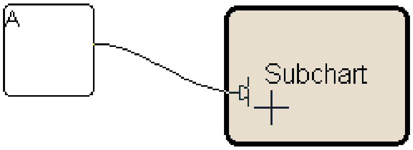
2. Drag the mouse.

   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.

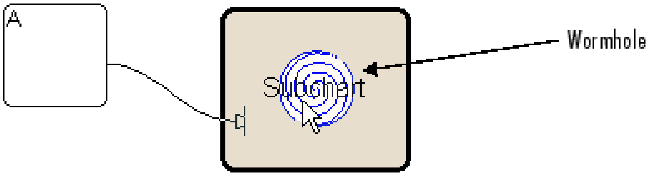
   The arrowhead now penetrates the slit.



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.

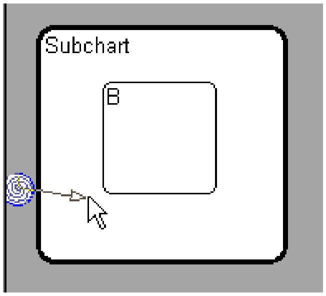
   A wormhole appears in the center of the subchart.



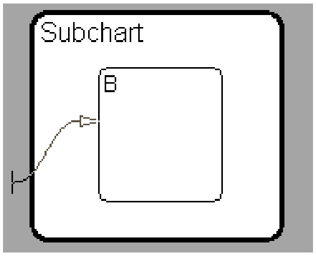
   A *wormhole* allows you to open a subchart while drawing a supertransition.

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.

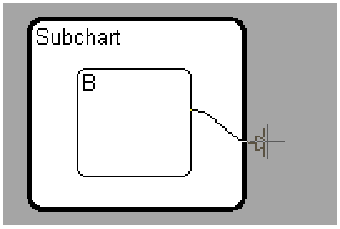
Drawing a Supertransition Out of a Subchart

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.

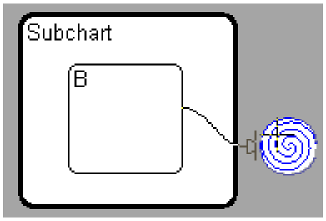
1. Draw an inner transition segment from the source object anywhere just outside the border of the subchart

   A slit appears as shown.

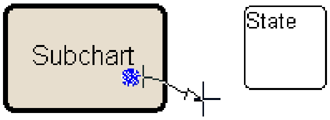
2. Keep dragging the transition away from the border of the subchart.

   A wormhole appears.

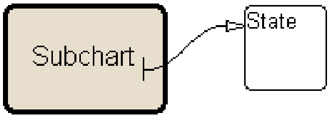


3 Drag the transition down the wormhole.

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. 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 4-25). 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.

Maintaining Transition Shapes with Smart Behavior

In this section…

What Are Smart Transitions?

Smart transitions attach their ends to the surfaces of Stateflow objects and maintain their shapes and uniqueness when you rearrange chart objects.

Setting Smart Behavior in Transitions

By default, new transitions have 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 steps:

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 no check mark appears for **Smart**, select it to enable smart behavior.

   To disable smart transition behavior, select **Smart** so that no check mark appears.

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.

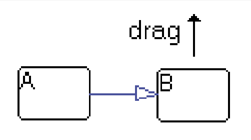
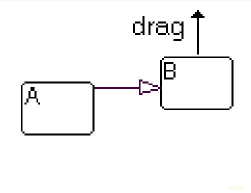
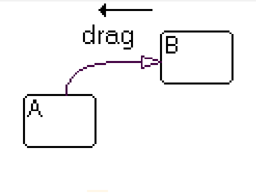
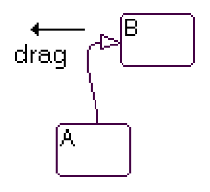
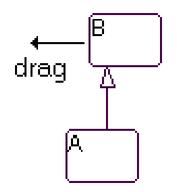
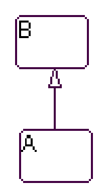
What Smart Transitions Do

The following topics discuss some of the behaviors of smart transitions:

- “Smart Transitions Slide Around Surfaces” on page 7-21
- “Smart Transitions Slide and Maintain Shape” on page 7-22
- “Smart Transitions Connect States to Junctions at 90 Degree Angles” on page 7-23
- “Smart Transitions Snap to an Invisible Grid” on page 7-25
- “Smart Transitions Bow Symmetrically” on page 7-26
- “Smart Transitions Prefer Straight Lines from Junctions” on page 7-27

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 as you drag it counterclockwise around the upper right corner of state A. During this process, state B turns to its selection color and the transition turns to a light shade of gray. The arrows show the dragging direction.

1	**2**	**3**
 	 	
4	**5**	**6**
 	 	

The following behavior applies to the preceding example:

1 The first capture shows states A and B at the beginning of movement.

2 As B moves upward, the back end of the transition slides upward on A, keeping the transition straight.

3 As B moves around the corner of A, the back end of the transition suddenly hops around the upper right-hand corner of A. The transition appears curved from the top surface of A to the left side of B. This shape maintains perpendicularity with each attached state side.

Note A hop around the corner of a state is necessary because transitions cannot attach at corners of states.

---

4 As B moves on top of A, the transition stays curved but its front end slides down to the lower left-hand corner of B.

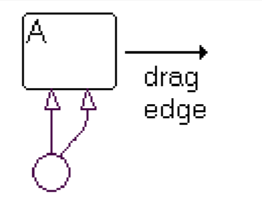
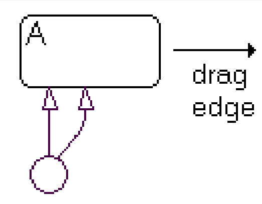
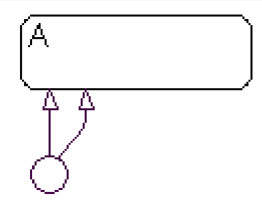
5 As B continues to move to the left over A, the front end of the transition hops around the lower left-hand corner of B.

6 Finally, as B moves directly over A, the front end of the transition slides over the bottom edge of B.

As B continues to circle A, steps 1 through 6 repeat for each remaining side of A.

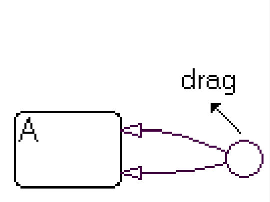
Smart Transitions Slide and Maintain Shape

While smart transitions allow their ends to slide around surfaces of connected objects, they also try 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.

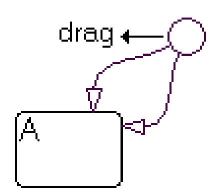
1	2	3
		

In the following example, the ends of a pair of transitions with smart behavior originate from a junction and terminate in a state. As the junction moves around the state, the ends slide around the state and maintain the same relative spacing between each other. The arrows indicate the direction of movement.

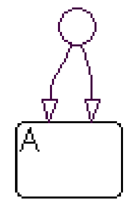
1



2



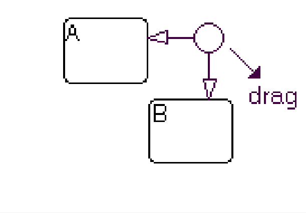
3



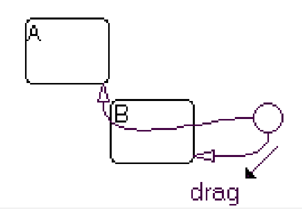
Smart Transitions Connect States to Junctions at 90 Degree Angles

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 connects to two states, A and B. Watch the behavior of two straight smart transitions as the junction moves to different locations.

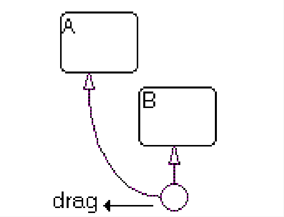
1



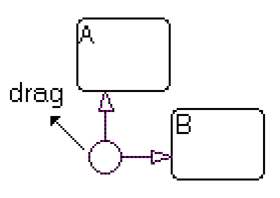
2



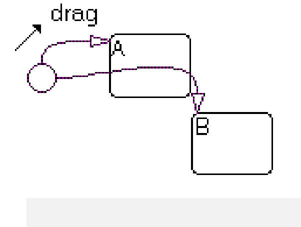
3



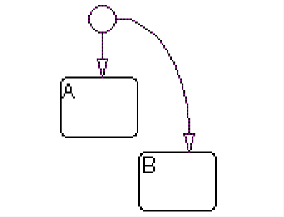
4



5



6



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.

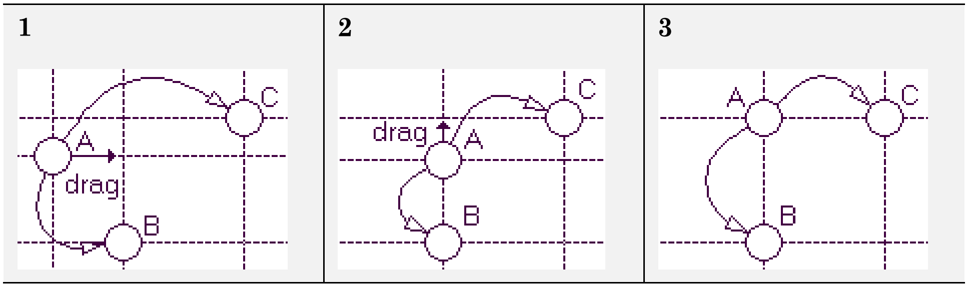
4 At this location (under A, to the left of B), straight-line transitions to A and B are possible from the junction's top and right connection points, respectively.

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:

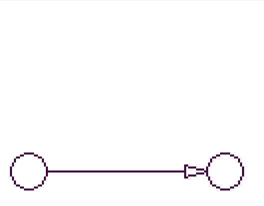
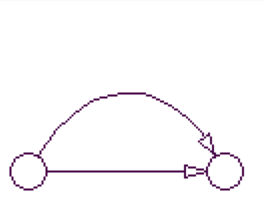
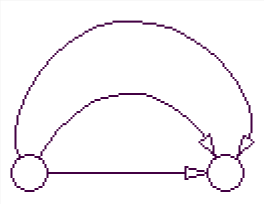
1 In the first scene, the snap grid for each junction does not overlap. The arrow indicates that junction A is being moved toward the vertical snap line for junction B.

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:

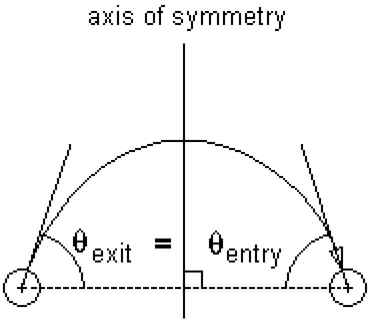
1	2	3
		

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.

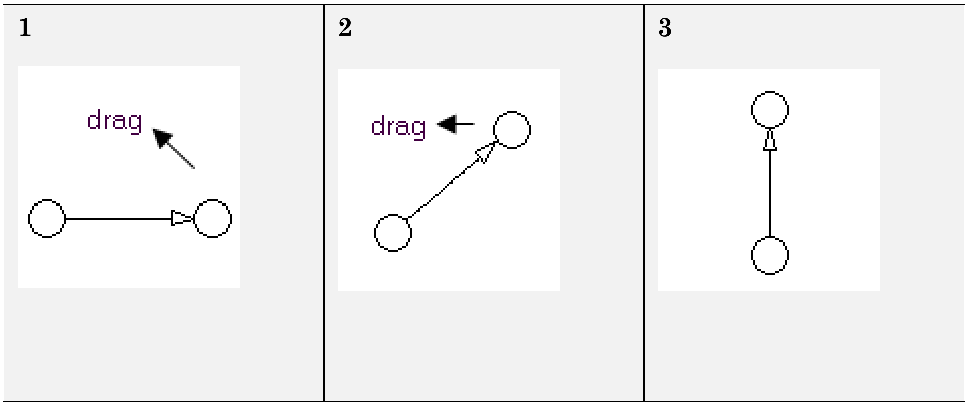


You can bow a smart transition between two junctions to any degree. Place your pointer anywhere on the transition (except end points) and click and drag in a direction perpendicular to a straight line connecting the two junctions. You can move the mouse in any direction to bow the transition but only the component perpendicular to the straight line applies.

Disabling smart behavior for a transition allows you to distort the transition asymmetrically (see “Nonsmart Transitions Distort Asymmetrically” on page 7-29). However, if you enable smart behavior again, the transition returns to its previous bowed shape.

Smart Transitions Prefer Straight Lines from Junctions

Transitions with smart behavior prefer straight lines coming from junctions. In the following example, the terminating junction moves in a radial direction around another junction.



The smart transition maintains a straight line, because the end on the originating junction follows the tip of the transition.

What Nonsmart Transitions Do

The following topics describe some of the behavior exhibited by transitions without smart behavior.

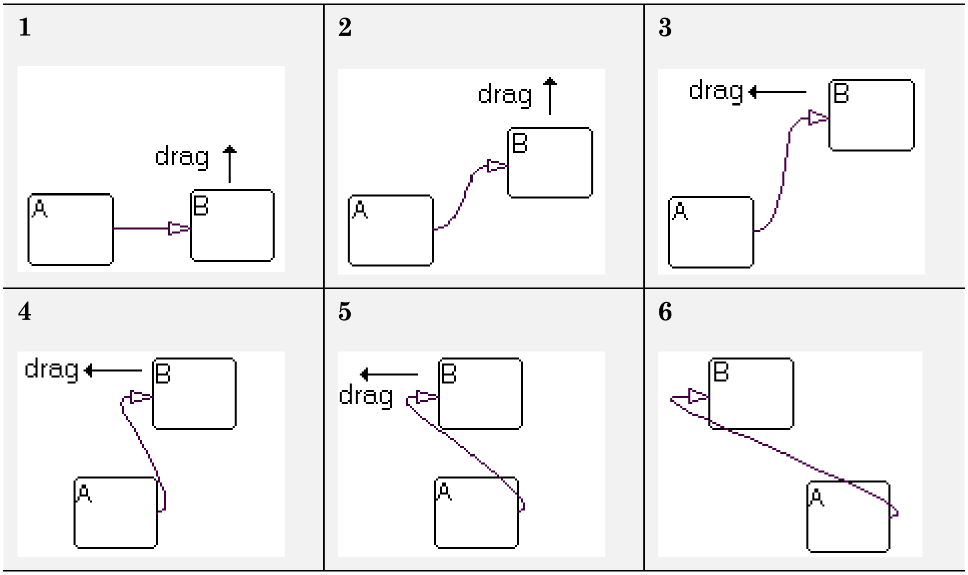
- “Nonsmart Transitions Anchor Connection Points” on page 7-28

- “Nonsmart Transitions Distort Asymmetrically” on page 7-29

You can disable and enable smart behavior in transitions. See the section “Setting Smart Behavior in Transitions” on page 7-20.

Nonsmart Transitions Anchor Connection Points

Contrast the example in the section “Smart Transitions Slide Around Surfaces” on page 7-21 with the following example.

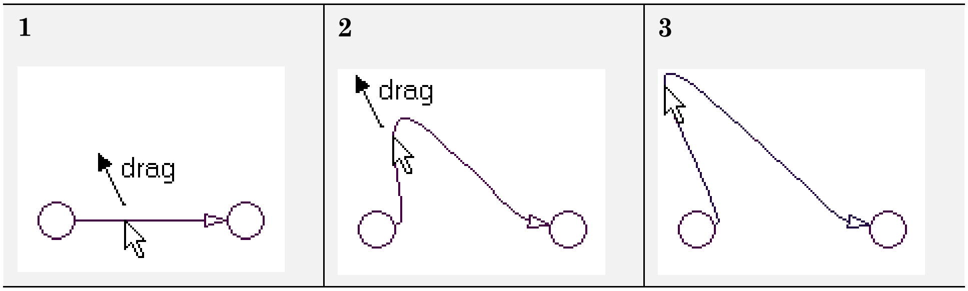


A nonsmart transition connects state A to state B. The pointer appears over state B and clicks and drags to new locations counterclockwise around A. During this process, state B turns to its highlight color but the transition remains unchanged, a sign of a nonsmart transition.

As B moves around A, the transition changes into a distorted curve that maintains the original attachment points. These points remain unchanged in position, although the angle of attachment is always perpendicular to the side of the state.

Nonsmart Transitions Distort Asymmetrically

By clicking and dragging on different locations along a nonsmart transition, you can reshape it into an asymmetric curve suited to your needs. Consider 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** to disable smart behavior.
3. Place your pointer anywhere on the transition.
4. Click and drag your pointer up and to the left.

Graphical Functions for Reusing Logic Patterns and Iterative Loops

In this section...

What Is a Graphical Function?

A graphical function in a Stateflow chart is a graphical element that helps you reuse control-flow logic and iterative loops. This function is a program you write with flow graphs using connective junctions and transitions. You create a graphical function, fill it with a flow graph, and call the function in the actions of states and transitions.

Why Use a Graphical Function in a Stateflow Chart?

This function helps you to:

- Create modular, reusable logic that you can call anywhere in your chart.
- Track simulation behavior visually during chart animation.

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. Follow these guidelines:

- If you want to call the function only within one state or subchart and its substates, put your graphical function in that state or subchart. That function overrides any other functions of the same name in the parents and ancestors of that state or subchart.
- If you want to call the function anywhere in that chart, put your graphical function at the chart level.
- If you want to call the function from any chart in your model, put your graphical function at the chart level and enable exporting of chart-level graphical functions. For instructions, see “Exporting Chart-Level Graphical Functions” on page 7-39.

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 editor toolbar:

   f()

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:

   $\texttt{[}r_1\texttt{, } r_2\texttt{,..., } r_n\texttt{] = func(}a_1\texttt{,}a_2\texttt{,..., }a_n\texttt{)}$

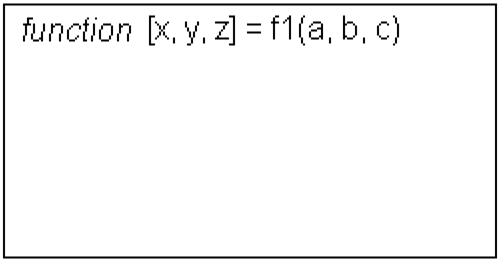
   where `func` is the name of your function, a_1, a_2, ..., a_n are formal names for its arguments, and r_1, r_2, ..., r_n are formal names for its return values.

   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`).



function [x, y, z] = f1(a, b, c)

Note In the chart, you can change the signature of your graphical function at any time. After you edit the signature, the Model Explorer updates to reflect the changes.

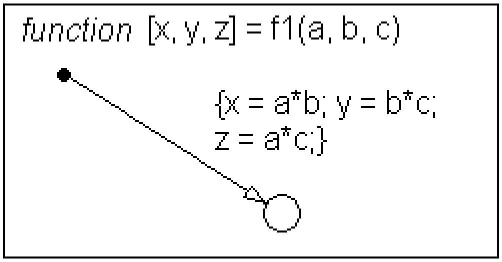
Programming a Graphical Function

To program a graphical function, follow these steps:

1. Click the default transition icon in the 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.

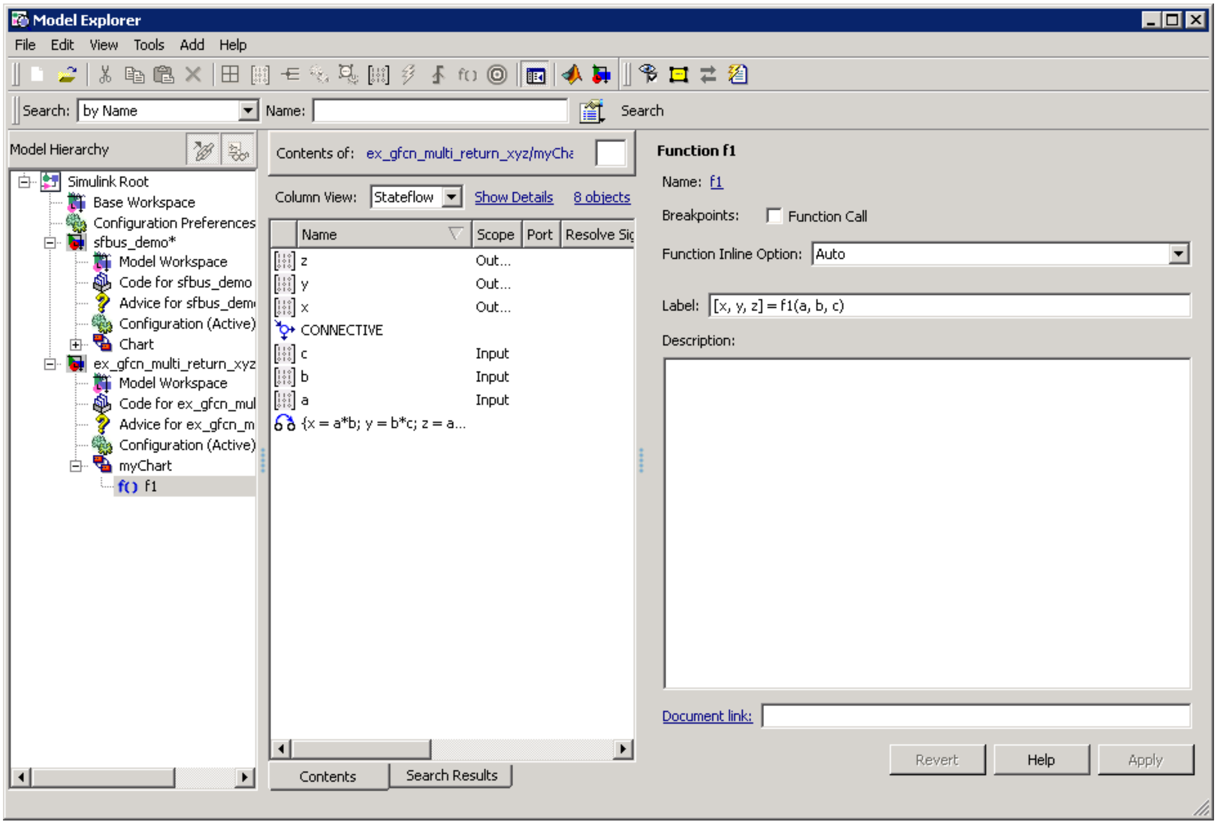


Defining Graphical Function Data

You must define the data in your graphical function:

1 Open the Model Explorer.

2 Expand the chart 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.



Name	Scope	Port	Resolve Sig
z	Out...		
y	Out...		
x	Out...		
CONNECTIVE			
c	Input		
b	Input		
a	Input		
{x = a*b; y = b*c; z = a...			

Function f1

Name: f1

Breakpoints: ☐ Function Call

Function Inline Option: Auto

Label: [x, y, z] = f1(a, b, c)

Description:

Document link:

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`

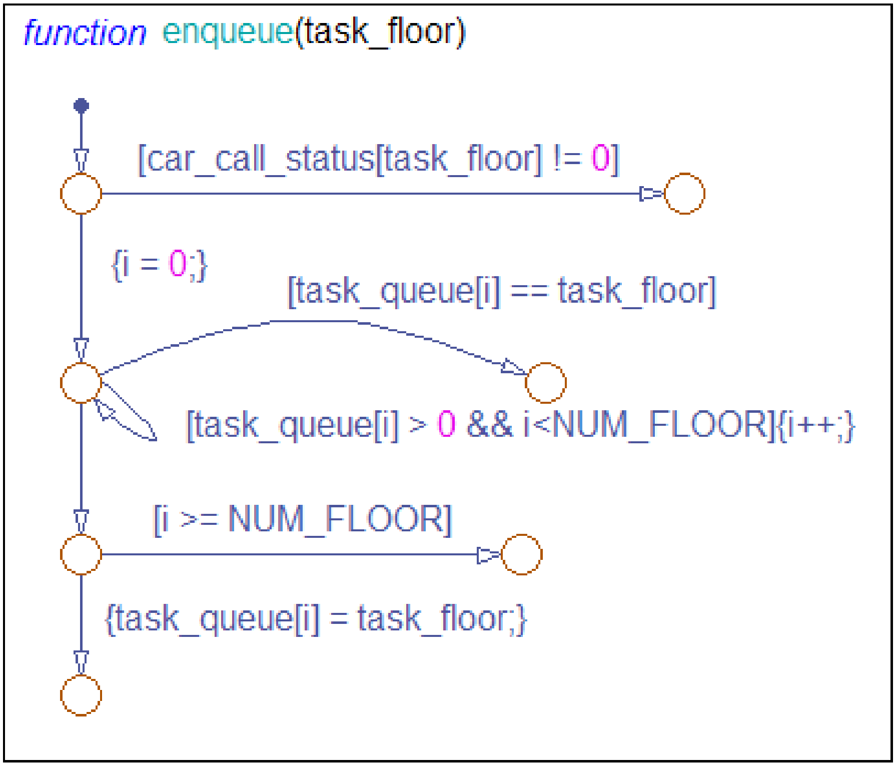
  Temporary data initializes at the start of every function call.
- `Constant`

  Constant data retains its initial value through all function calls.

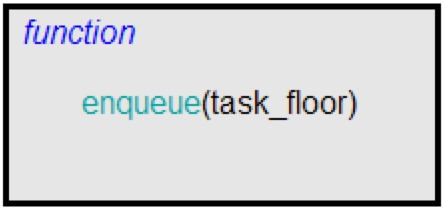
Note You can initialize your function data (other than arguments and return values) from the MATLAB workspace. However, you can save only local items to this workspace.

Managing Large Graphical Functions

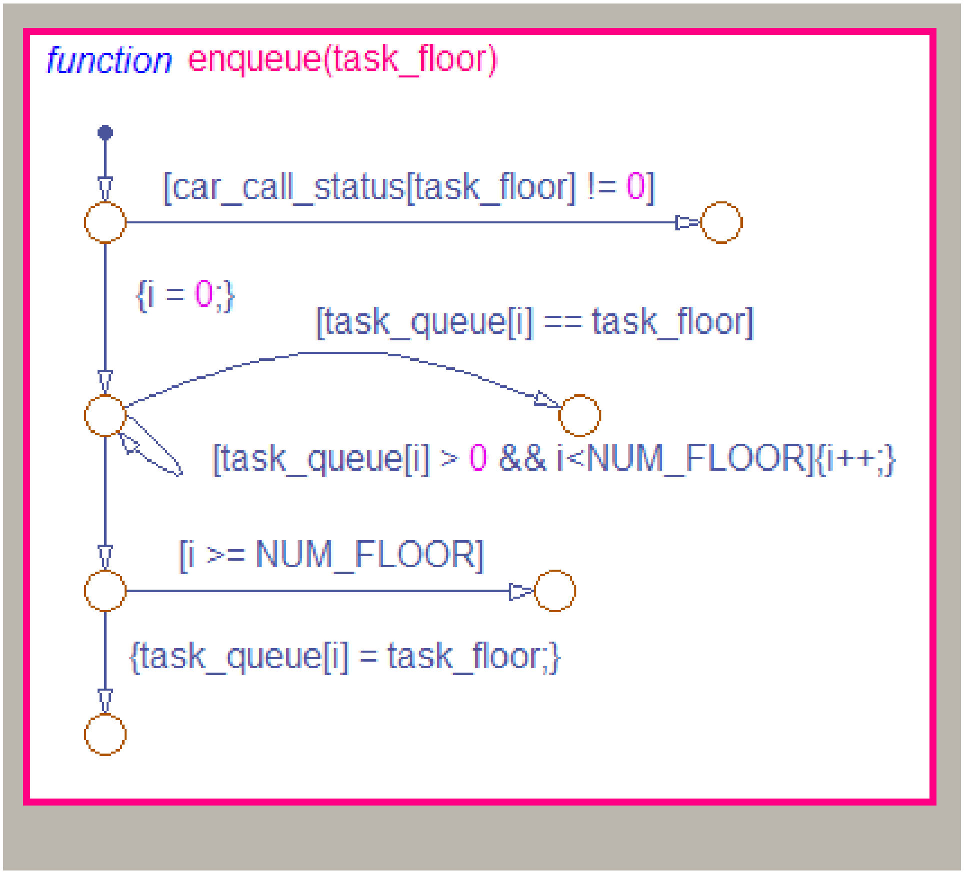
You can make your graphical function as large as you want, as shown below.



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 States and Transitions

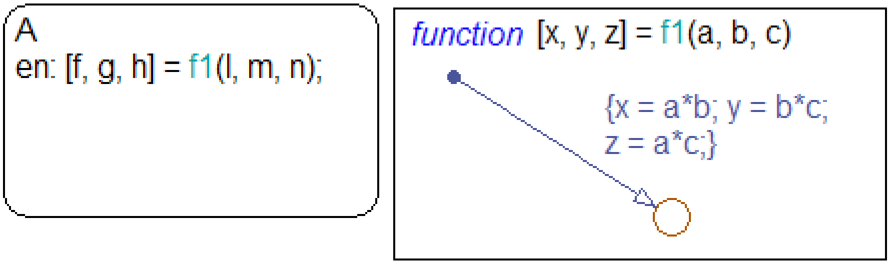
Syntax

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 7-31 for information about syntax for a function signature.

Tip If the formal arguments of a function signature are scalars, verify that inputs and outputs of function calls follow the rules of scalar expansion. For more information, see “How Scalar Expansion Works for Functions” on page 13-6.

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

To export graphical functions to your main model:

1. Open the chart where your graphical function resides.
2. Open the Chart properties dialog box.
3. 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, a simulation error occurs.

Do not export graphical functions that contain unsupported inputs or outputs

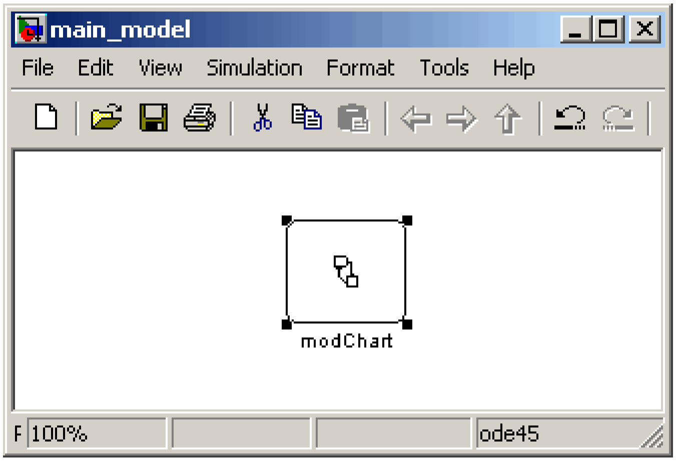
You cannot export a graphical function when inputs or outputs have any of the following properties:

- Fixed-point data type with word length greater than 32 bits
- Variable size

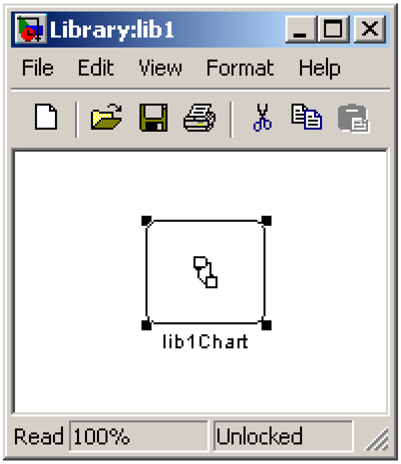
Example of Exporting Chart-Level Graphical Functions

This example describes how to export graphical functions in library charts to your main model.

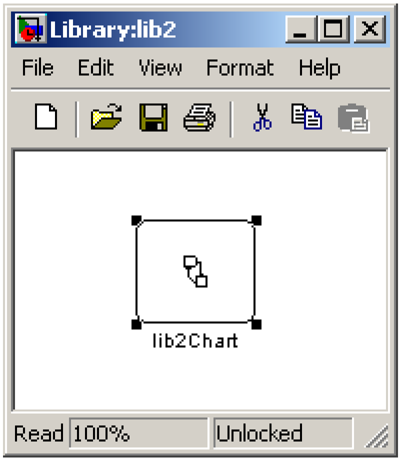
1. Create these objects:
    - Add a model named `main_model`, with a chart named `modChart`.



- Add a library model named `lib1`, with a chart named `lib1Chart`.

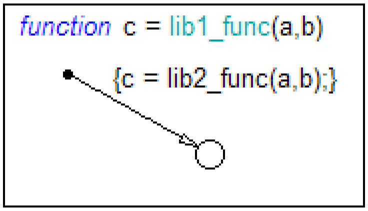


- Add a library model named `lib2`, with a chart named `lib2Chart`.

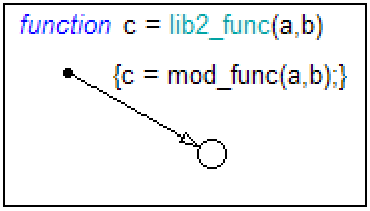


2 Create these graphical functions in the library charts:

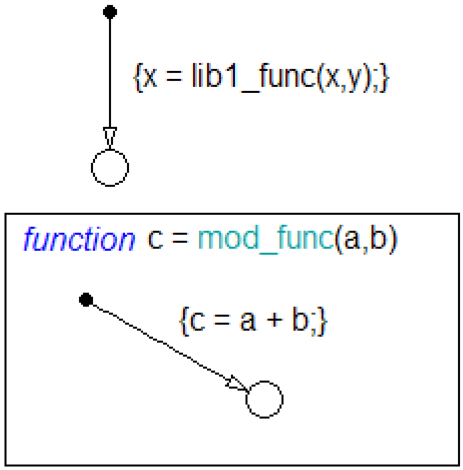
- For `lib1Chart`, add this graphical function.



- For `lib2Chart`, add this graphical function.

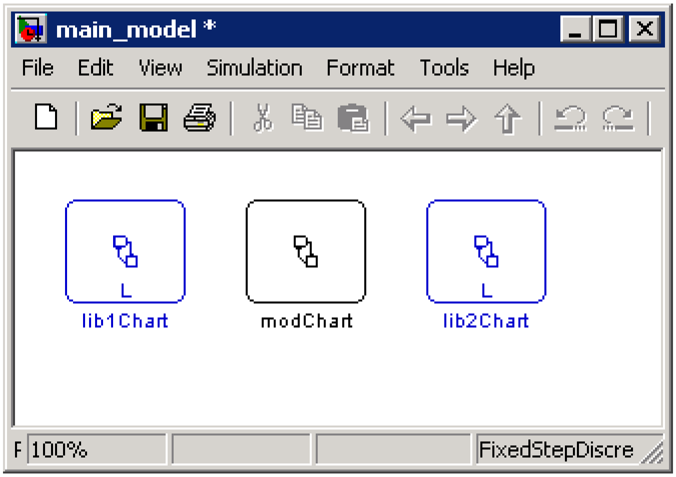


3. For `modChart`, add a graphical function and a default transition with a `lib1_func` action.



4. For each chart, follow these steps:
   - **a** Open the Chart properties dialog box.
   - **b** In the Chart properties dialog box, select **Export Chart Level Graphical Functions (Make Global)**.
   - **c** Click **OK**.

5 Drag `lib1Chart` and `lib2Chart` into `main_model` from `lib1` and `lib2`, respectively. Your main model should look something like this:



Each chart now defines a graphical function that any chart in `main_model` can call.

6 Open the Model Explorer.

7 In the **Model Hierarchy** pane of the Model Explorer, navigate to `main_model`.

8 Add the data `x` and `y` to the Stateflow machine:

  a Select **Add > Data**.

  b In the **Name** column, enter `x`.

  c In the **Initial Value** column, enter `0`.

  d Use the default settings for other properties of `x`.

  e Select **Add > Data**.

  f In the **Name** column, enter `y`.

  g In the **Initial Value** column, enter `1`.

  h Use the default settings for other properties of `y`.

This step ensures that input and output data are defined globally to support exported graphical functions.

9. Open the Configuration Parameters dialog box.

10. In the Configuration Parameters dialog box, go to the **Solver** pane.

11. In the **Solver options** section, make these changes:
    a. For **Type**, select `Fixed-step`.
    b. For **Solver**, select `Discrete (no continuous states)`.
    c. For **Fixed-step size**, enter `1`.
    d. Click **OK**.

    This step ensures that when you simulate your model, a discrete solver is used. For more information, see "Solvers" in the Simulink 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. Open the Simulink Library Browser.

2 From the Simulink/Sinks Library, select the Scope block and add it to `main_model`.

3 Open the Model Explorer.

4 In the **Model Hierarchy** pane, navigate to `modChart`.

5 Add the output data `z` to the chart:

   a Select **Add > Data**.

   b In the **Name** column, enter `z`.

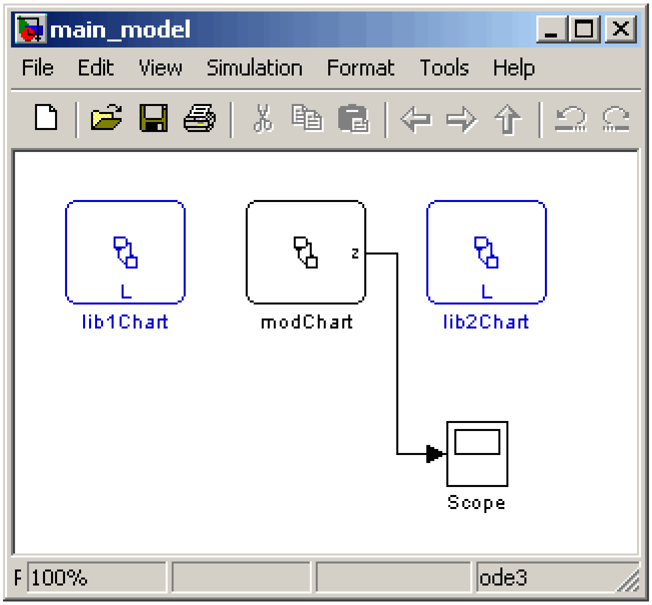
   c In the **Scope** column, select `Output`.

   d Use the default settings for other properties.

6 For `modChart`, update the default transition action to read as follows:

```
{x = lib1_func(x,y); z = x;}
```

7 In the model window, connect the outport from `modChart` to the inport of the Scope block.

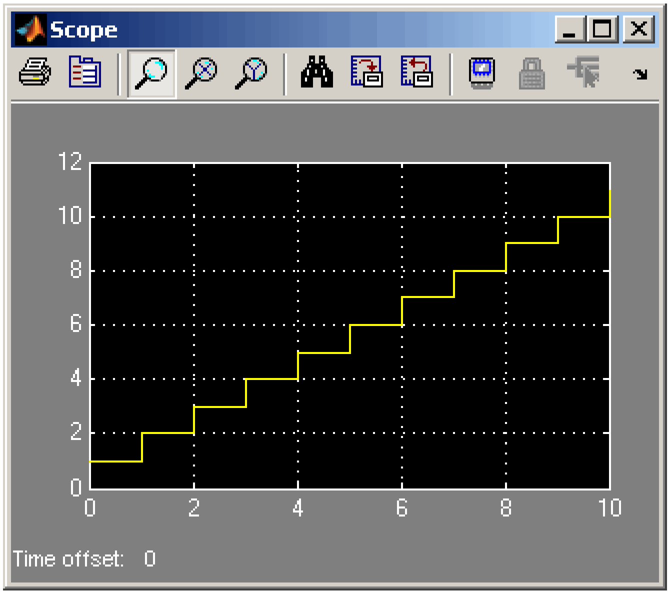


8 Double-click the Scope block to open the display.

9 Start simulation.

10 After the simulation ends, right-click in the scope display and select **Autoscale**.

The results look something like this:



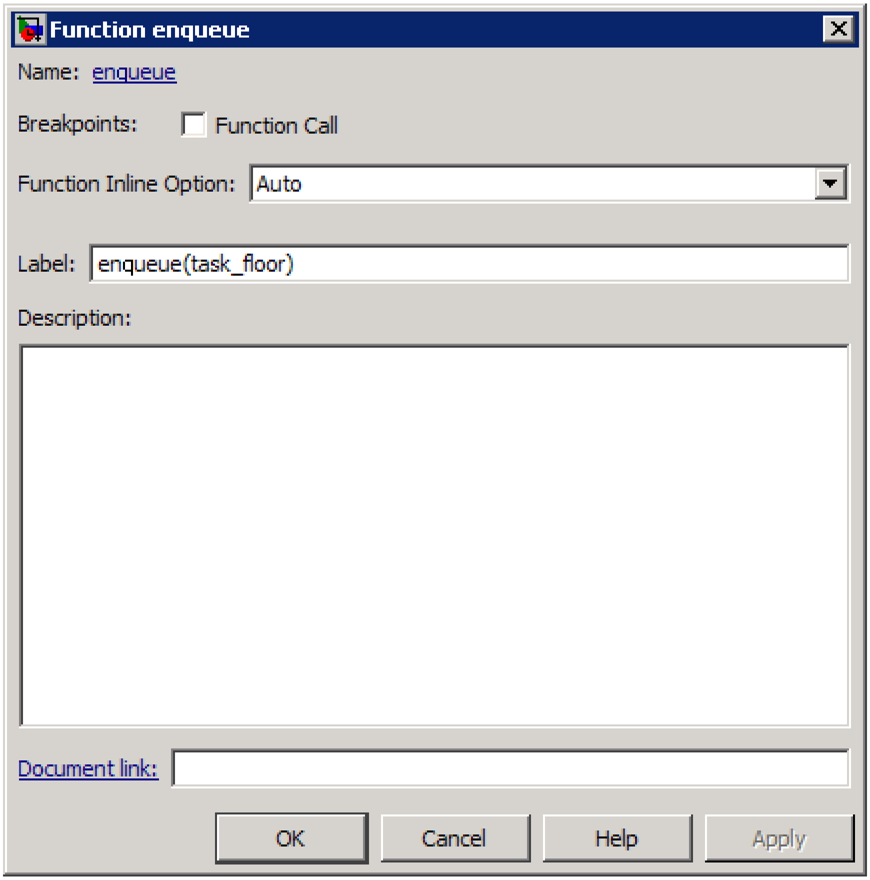
Specifying Graphical Function Properties

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.



Name: enqueue

Function Inline Option: Auto

Label: enqueue(task_floor)

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 7-31 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`.

Grouping Chart Objects with Boxes

In this section...

When to Use Boxes

Use a Stateflow box to organize graphical objects in your chart.

Semantics of Stateflow Boxes

Visibility of Graphical Objects in Boxes

Boxes add a level of hierarchy to Stateflow charts. This property affects visibility of functions and states inside a box to objects that reside outside of the box. If you refer to a box-parented function or state from a location outside of the box, you must include the box name in the path. See "Using a Box to Group Functions" on page 7-53.

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 7-54.

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 chart. For details, see "Explicit Ordering of Parallel States" on page 3-76.

Rules for Using Boxes

When you use a box, these rules apply:

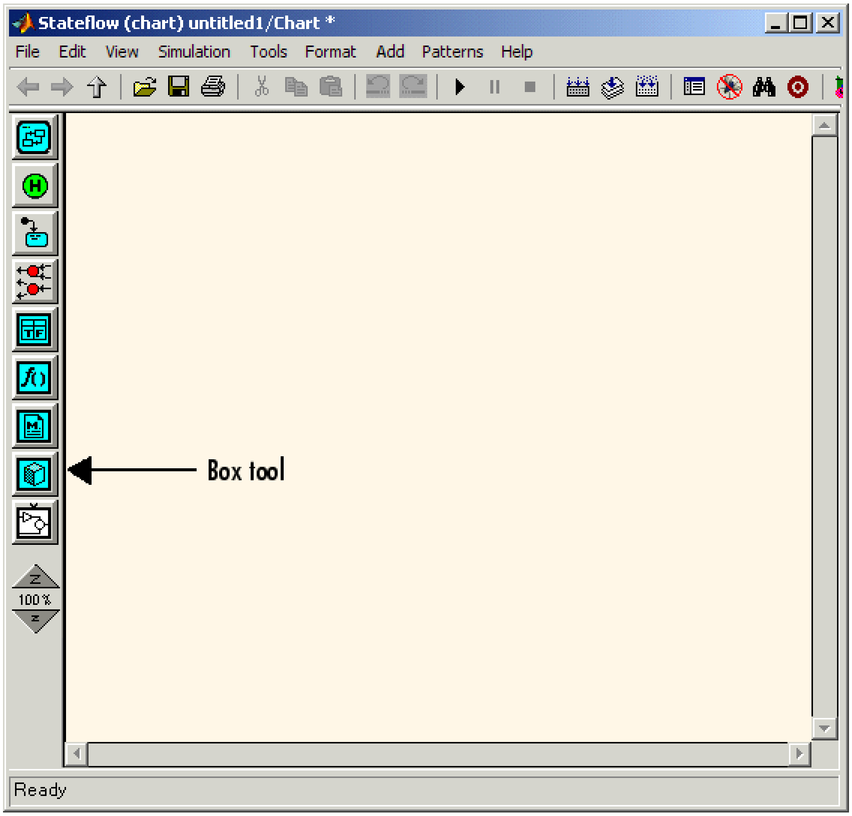
- Include the box name in the path when you use dot notation to refer to a box-parented function or state from a location outside of the box.
- You can move or draw graphical objects inside a box, such as functions and states.

  You can draw a state around the objects you want inside it and then convert that state to a box. See .
- 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 4-12.
- You can subchart a box to hide its elements. See "Using Subcharts to Encapsulate Modal Logic" on page 7-6.
- 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 in your chart by using the box tool shown below.



1 Click the Box tool.

2 Move your pointer into the drawing area.

3 Click in any location to create a box.

The new box appears with a question mark (`?`) name in its upper left corner.

4 Click the question mark label.

5 Enter a name for the box and then click outside of the box.

Deleting a Box

To delete a box, click to select it and press the **Delete** key.

Examples of Using Boxes

Using a Box to Group Functions

This chart shows a box named `Status` that groups together MATLAB functions.

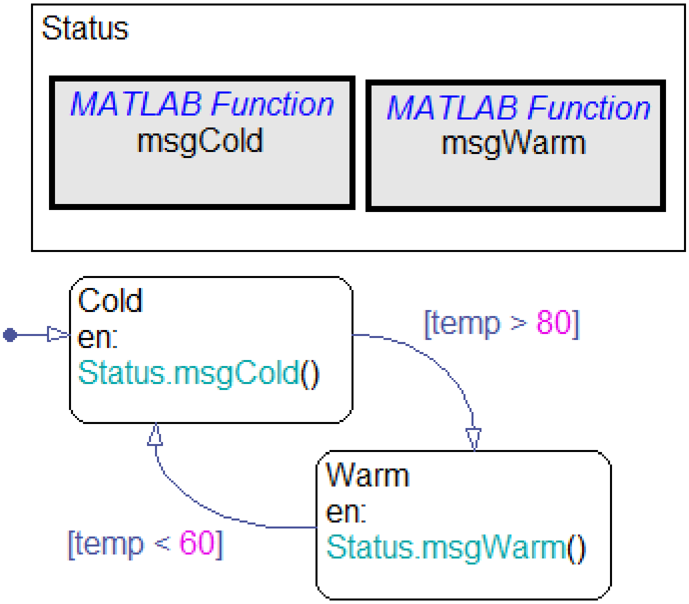


Chart execution takes place as follows:

1. The state `Cold` activates first.
2. Upon entry, the state `Cold` invokes the function `Status.msgCold`.

   This function displays a status message that the temperature is cold.

Note Because the MATLAB function resides inside a box, the path of the function call must include the box name `Status`. If you omit this prefix, an error message appears.

---

3. If the value of the input data `temp` exceeds 80, a transition to the state `Warm` occurs.
4. Upon entry, the state `Warm` invokes the function `Status.msgWarm`.

   This function displays a status message that the temperature is warm.

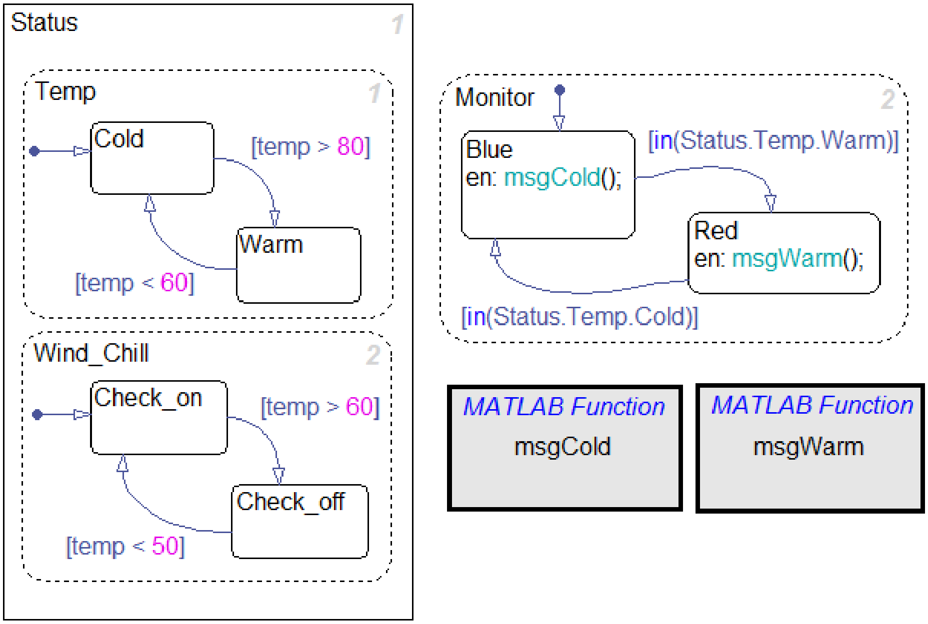
Note Because the MATLAB function resides inside a box, the path of the function call must include the box name `Status`. If you omit this prefix, an error message appears.

---

5. If the value of the input data `temp` drops below 60, a transition to the state `Cold` occurs.
6. Steps 2 through 5 repeat until the simulation ends.

Using a Box to Group States

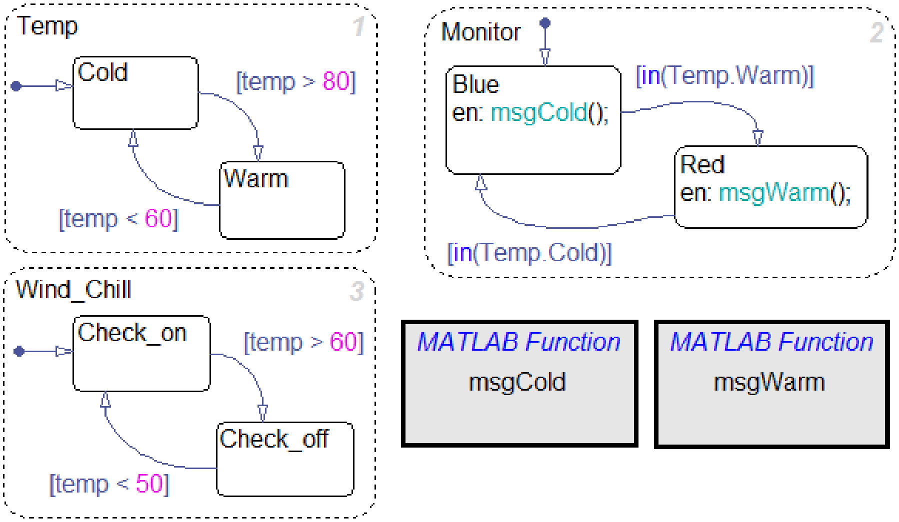
This chart shows a box named `Status` that groups together related states. The chart uses implicit ordering for parallel states, instead of the default explicit mode. (For details, see "Implicit Ordering of Parallel States" on page 3-77.)



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 10-97.

Using Descriptive Comments in a Chart

In this section…

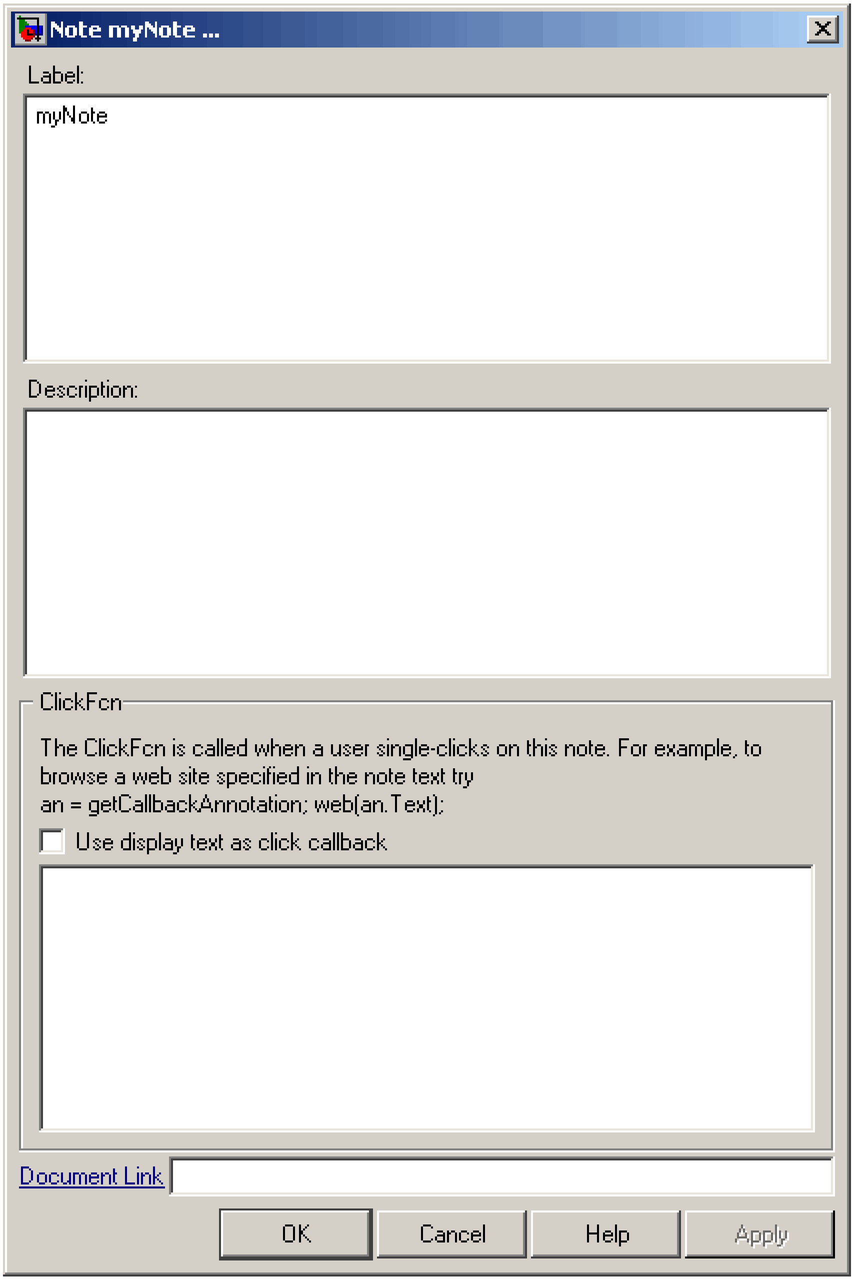
Creating Notes

You can enter comments or notes in any location on the chart.

1. Place your pointer at the desired location in the chart.
2. Right-click and select **Add Note**.
3. Begin typing your comments.
4. Press the **Return** key to start a new line.
5. When finished typing, click outside the note text.

Changing Note Properties

You can use the Note properties dialog box to edit note properties.



myNote

The ClickFcn is called when a user single-clicks on this note. For example, to browse a web site specified in the note text try
an = getCallbackAnnotation; web(an.Text);

☐ Use display text as click callback

The Note properties dialog box contains the following properties:

Field	Description
Label	The label for the note. This includes the name of the note and its associated actions.
Description	Textual description/comment.
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. Selecting this option disables the **ClickFcn** edit field.
ClickFcn	Specifies MATLAB code to be executed when a user single-clicks this annotation. The 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 documentation for a description of the **ClickFcn** edit field.

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 a Chart" on page 4-39.

You can also change your note text to bold or italic:

1. Right-click the note text and select **Text Format**.
2. In the submenu, select **Bold** or **Italic** (default).

TeX Instructions

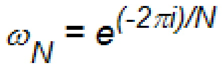
In your notes, you can use a subset of TeX commands embedded in the string to produce special characters. For example, you can embed Greek letters and mathematical symbols.

1. Right-click the text of a note and select **Text Format**.
2. In the submenu, make sure that **TeX instructions** has a check mark in front of it. Otherwise, select it.
3. Click the note text.
4. Replace the existing note text with the following expression.

   ```
   \it{\omega_N = e^{(-2\pii)/N}}
   ```

5. Click outside the note.

   The note in your chart looks something like this:



$$\omega_N = e^{(-2\pi i)/N}$$

Defining Data

Adding Data

In this section...

When to Add Data

Add data when you want to store values that are visible at 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. Actions in your chart can refer to internal and external data.

Diagnostic for Detecting Unused Data

If you have unused data in your chart, a warning appears during simulation with a list of data you can remove. By removing objects that have no effect on simulation, you can reduce the size of your model. This diagnostic checks for usage of Stateflow data, except for the following types:

- Machine-parented data
- Inputs and outputs of MATLAB functions
- Data of parameter scope in a chart that contains atomic subcharts

After you select data for removal, a dialog box confirms your choice. In this dialog box, you can specify that other deletions occur without confirmation. If you prevent the confirmation dialog box from appearing, you can reenable it at any time by typing at the command prompt:

```
sfpref('showDeleteUnusedConfGui', 1)
```

You can control the level of diagnostic action for unused data in the **Diagnostics > Stateflow** pane of the Configuration Parameters dialog box. For more information, see the documentation for the "Unused data and events" diagnostic.

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 8-5.

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**.

   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.

4. Change the properties of the data, as described in "Setting Data Properties in the Data Dialog Box" on page 8-5.

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:

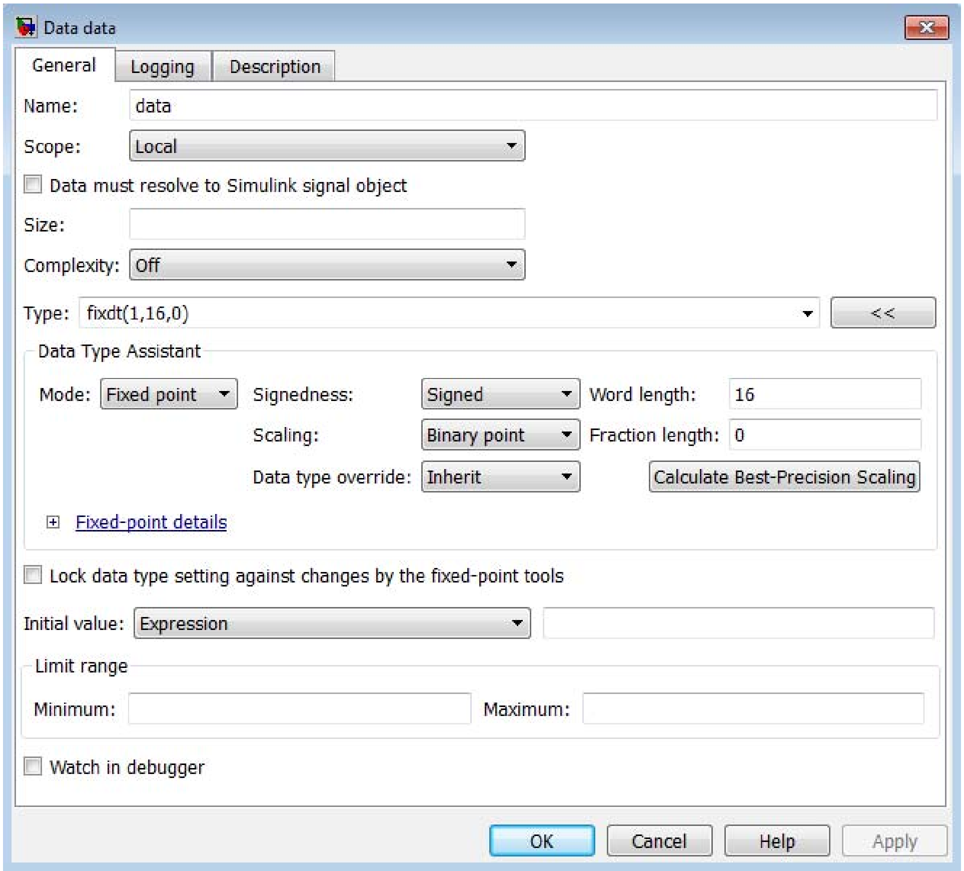
- Stateflow machine
- Stateflow chart
- Box
- State
- Subchart
- Substate
- Function

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 `Local` and type is `Fixed point`.



Name: data

Scope: Local

☐ Data must resolve to Simulink signal object

Size:

Complexity: Off

Type: fixdt(1,16,0)

Data Type Assistant

Mode: Fixed point

Signedness: Signed

Word length: 16

Scaling: Binary point

Fraction length: 0

Data type override: Inherit

☐ Lock data type setting against changes by the fixed-point tools

Initial value: Expression

Limit range

Minimum:

Maximum:

☐ Watch in debugger

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, type, initial value, and limit range of a data object. See “Properties You Can Set in the General Pane” on page 8-8.

- Use the **Logging** pane to enable logging for local data. See "Properties You Can Set in the Logging Pane" on page 8-24.

Note You can log states and local data in Stateflow charts.

- Use the **Description** pane to index into a data object array and enter a description about the data object. See "Properties You Can Set in the Description Pane" on page 8-26.

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 8-3.

  After you add the data object, the Data properties dialog box appears.

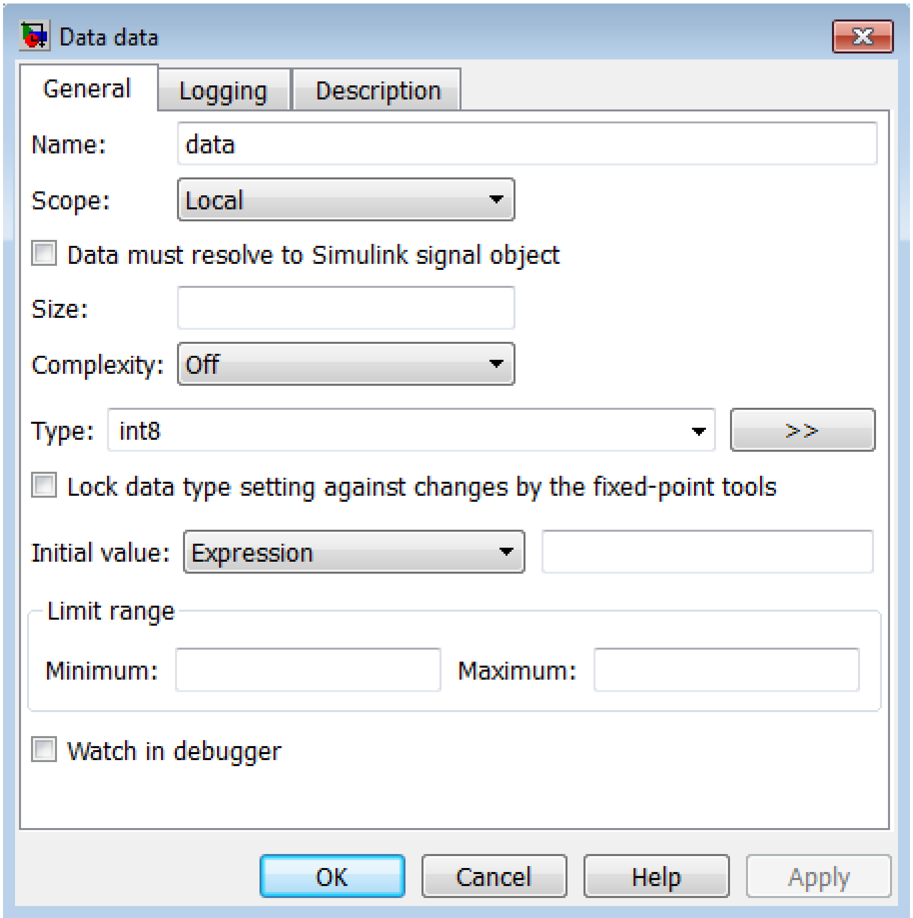
- Open the Data properties dialog box from the Model Explorer for a data object that already exists in the Stateflow hierarchy. Use one of these techniques:
  - Double-click the data object in the **Contents** pane.
  - Right-click the data object in the **Contents** pane and select **Properties**.
  - Select the data object in the **Contents** pane and then select **View > Show Dialog Pane**.

    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.



Data data

Name: data

Scope: Local

☐ Data must resolve to Simulink signal object

Size:

Complexity: Off

Type: int8

☐ Lock data type setting against changes by the fixed-point tools

Initial value: Expression

Limit range

Minimum:

Maximum:

☐ Watch in debugger

You can set these properties in the **General** pane.

Name

Name of the data object. For more information, see "Rules for Naming Stateflow Objects" on page 2-5.

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 “Working with Block Masks” in Simulink software documentation for information on how to assign a parameter to a masked subsystem. See “Sharing Simulink Parameters with Charts” on page 8-32 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 MATLAB function. Otherwise, the Simulink model provides the data to the Stateflow chart via an input port on the Stateflow block. See “Sharing Output Data with Simulink” on page 8-31.
Output	Return value of a function if the parent is a graphical, truth table, or MATLAB function. Otherwise, the Stateflow chart provides the data to the Simulink model via an output port on the Stateflow block. See “Sharing Output Data with Simulink” on page 8-31.

Scope Value	Description
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 Multiple Charts” on page 8-35.
Temporary	Data that persists only during the execution of a function. You can define temporary data only for a graphical, truth table, or MATLAB function, as described in “Defining Temporary Data” on page 8-65.
Exported	Data from the Simulink model that is made available to external code defined in the Stateflow hierarchy, as described in “Sharing Chart Data with External Modules” on page 8-42. 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 Chart Data with External Modules” on page 8-42. 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 Output Data with Simulink” on page 8-31.

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 the generated code)
- Sampling mode (for Truth Table block output data)

For more information, see "Resolving Data Properties from Simulink Signal Objects" on page 8-72.

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 8-54.

Variable size

Option that specifies whether the data object changes dimensions during simulation. This check box is available only for input and output data. For more information, see Chapter 14, "Using Variable-Size Data in Stateflow Charts".

Complexity

Option that specifies whether 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 18-2.

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 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 8-45.

Lock data type setting against changes by the fixed-point tools

Select this check box to prevent replacement of the current data type with a type that the Fixed-Point Tool or Fixed-Point Advisor chooses. See "Automatic Data Typing Tools" in the *Simulink Fixed Point™ User's Guide* for instructions on autoscaling fixed-point data.

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
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 resolves.

For more information, see "Initializing Data from the MATLAB Base Workspace" on page 8-32 and "Sharing Simulink Parameters with Charts" on page 8-32.

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:

- **Minimum** — The smallest value allowed for the data item during simulation. You can enter an expression or parameter that evaluates to a numeric scalar value.
- **Maximum** — The largest value allowed for the data item during simulation. You can enter an expression or parameter that evaluates to a numeric scalar value.

The smallest value you can set for **Minimum** is `-inf` and the largest value you can set for **Maximum** 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 minimum or maximum value before you can select **Calculate Best-Precision Scaling** in the **General** pane. For more information, see "Calculate Best-Precision Scaling" on page 8-18.

For more information on entering values for **Limit range** properties, see "Entering Expressions and Parameters for Data Properties" on page 8-27.

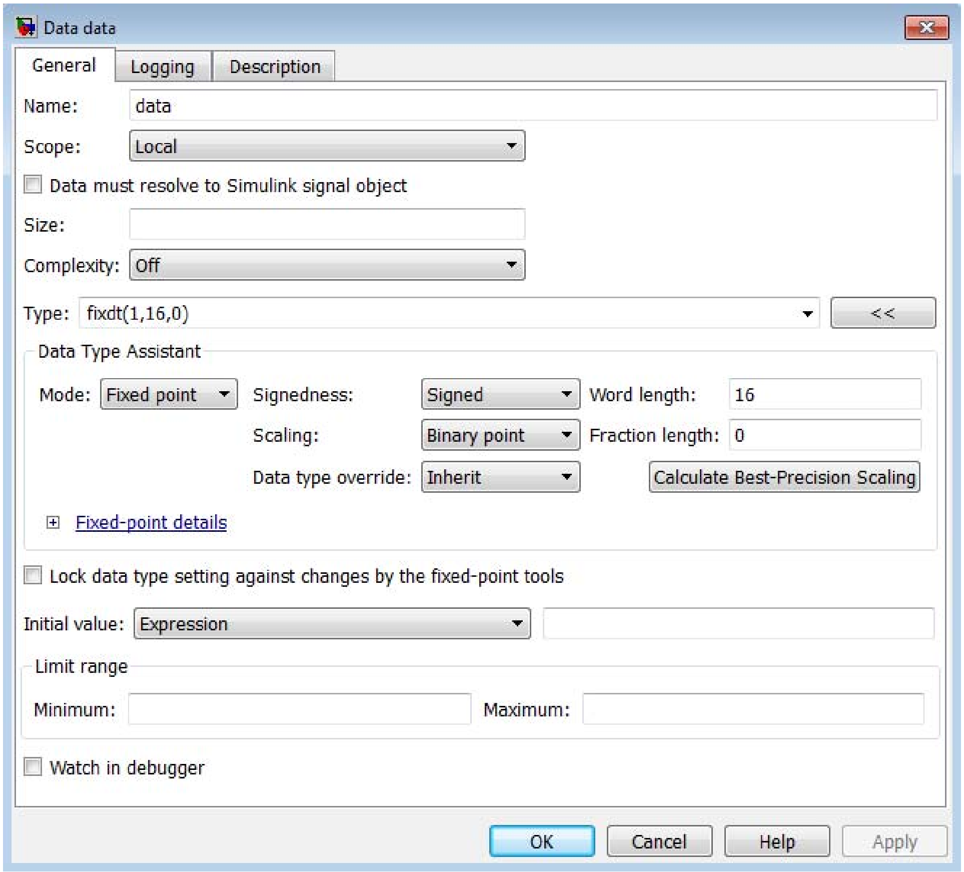
Watch in debugger

Option that enables you to watch the data values in the Stateflow Debugger (see "Watching Data in the Stateflow Debugger" on page 26-37).

Fixed-Point Data Properties

Properties that apply 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.



Name: data

Scope: Local

☐ Data must resolve to Simulink signal object

Size:

Complexity: Off

Type: fixdt(1,16,0)

Data Type Assistant

Mode: Fixed point

Signedness: Signed

Word length: 16

Scaling: Binary point

Fraction length: 0

Data type override: Inherit

⊞ Fixed-point details

☐ Lock data type setting against changes by the fixed-point tools

Initial value: Expression

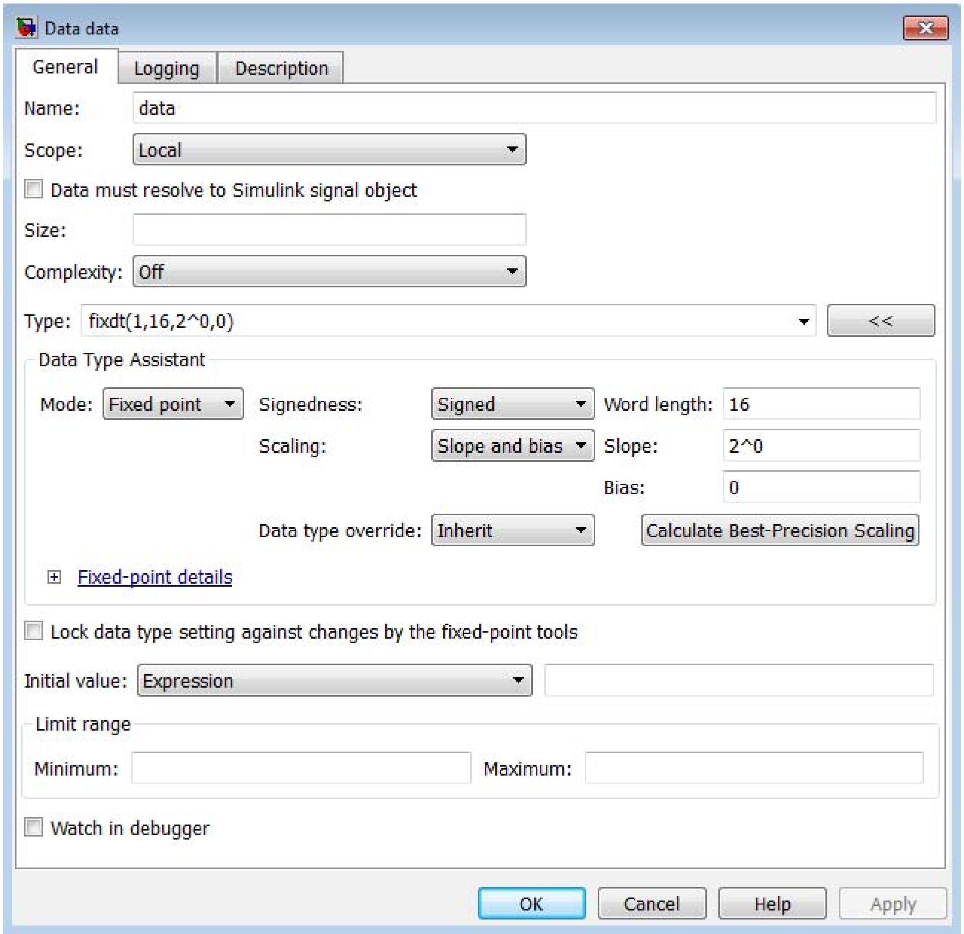
Limit range

Minimum:

Maximum:

☐ Watch in debugger

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.



Name: data

Scope: Local

☐ Data must resolve to Simulink signal object

Size:

Complexity: Off

Type: fixdt(1,16,2^0,0)

Data Type Assistant

Mode: Fixed point

Signedness: Signed

Word length: 16

Scaling: Slope and bias

Slope: 2^0

Bias: 0

Data type override: Inherit

☐ Lock data type setting against changes by the fixed-point tools

Initial value: Expression

Limit range

Minimum:

Maximum:

☐ Watch in debugger

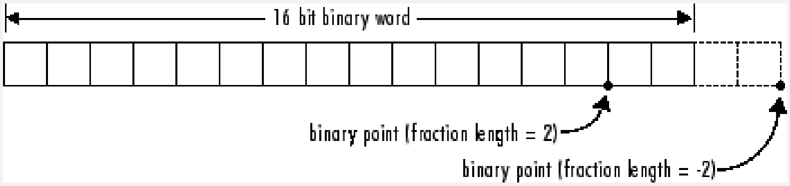
You can use the Data Type Assistant to set these fixed-point properties:

Signedness. 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 holds the quantized integer. Large word sizes represent large values with greater precision than small word sizes. The default bit size is 16.

- For chart-level data of the following scopes, word length can be any integer between 0 and 128.
  - Input
  - Output
  - Parameter
  - Data Store Memory
- For other Stateflow data, word length can be any integer between 0 and 32.

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: 

Scaling Mode	Description
	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*.

Data type override. Specify whether or not to inherit the data type override setting of the Fixed-Point Tool that applies to this model. If the data does not inherit the model-wide setting, the specified data type applies. For more information about the Fixed-Point Tool, see `fxptdlg` in the Simulink documentation.

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 **General** tab of the Data properties dialog box.

To automatically calculate best precision scaling values:

1. In the Data properties dialog box, click the **General** tab.

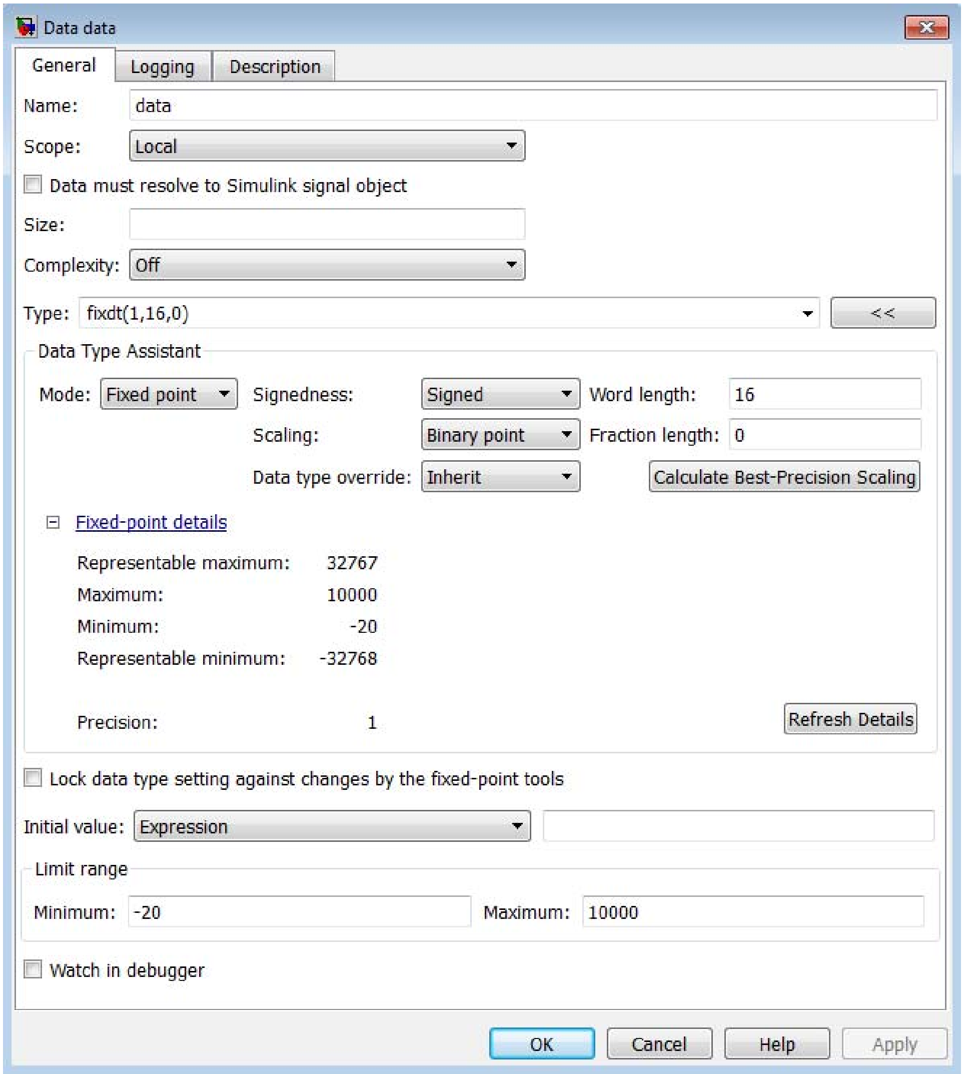
2. Specify **Limit range** properties.

3 Click **Calculate Best-Precision Scaling**.

Simulink software calculates the scaling values and displays them in the **Fraction length** field or the **Slope** and **Bias** fields. For more information, see "Constant Scaling for Best Precision" 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 data of `Parameter` scope.

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.



Name: data

Scope: Local

☐ Data must resolve to Simulink signal object

Size:

Complexity: Off

Type: fixdt(1,16,0)

Data Type Assistant

Mode: Fixed point

Signedness: Signed

Word length: 16

Scaling: Binary point

Fraction length: 0

Data type override: Inherit

⊟ Fixed-point details

Representable maximum:	32767
Maximum:	10000
Minimum:	-20
Representable minimum:	-32768
Precision:	1

☐ Lock data type setting against changes by the fixed-point tools

Initial value: Expression

Limit range

Minimum: -20

Maximum: 10000

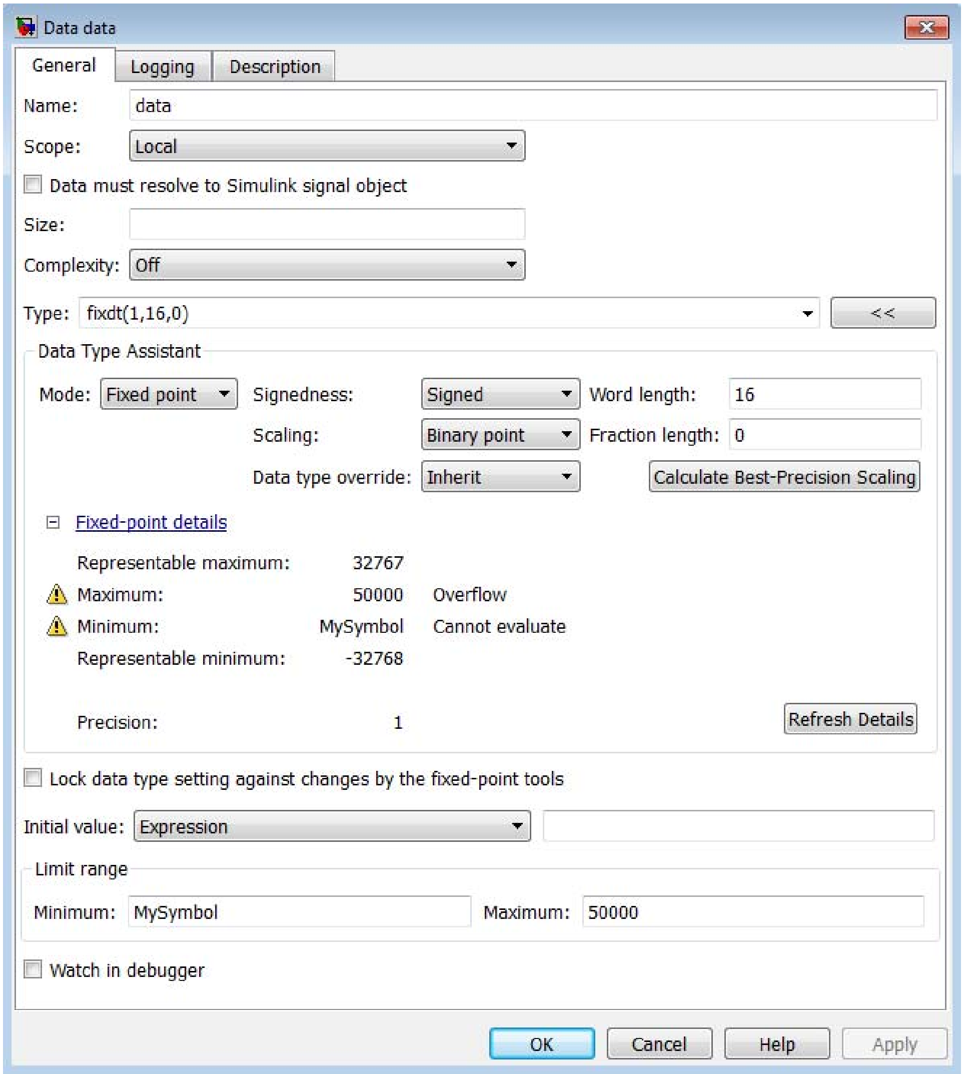
☐ Watch in debugger

The rows labeled `Minimum` and `Maximum` show the same values that appear in the corresponding **Minimum** and **Maximum** fields in the **Limit range** section. See "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.



Data data

General | Logging | Description

Name: data

Scope: Local

☐ Data must resolve to Simulink signal object

Size:

Complexity: Off

Type: fixdt(1,16,0)

Data Type Assistant

Mode: Fixed point

Signedness: Signed

Word length: 16

Scaling: Binary point

Fraction length: 0

Data type override: Inherit

⊟ Fixed-point details

Representable maximum:	32767	
⚠ Maximum:	50000	Overflow
⚠ Minimum:	MySymbol	Cannot evaluate
Representable minimum:	-32768	
Precision:	1	

☐ Lock data type setting against changes by the fixed-point tools

Initial value: Expression

Limit range

Minimum: MySymbol

Maximum: 50000

☐ Watch in debugger

The row labeled `Minimum` shows the error `Cannot evaluate` because evaluating the expression `MySymbol`, specified in the **Minimum** field of the **Limit range** section, cannot return a numeric value. When an expression does not evaluate successfully, the **Fixed-point details** subpane shows the unevaluated expression (truncating to 10 characters as needed) in place of the unavailable value.

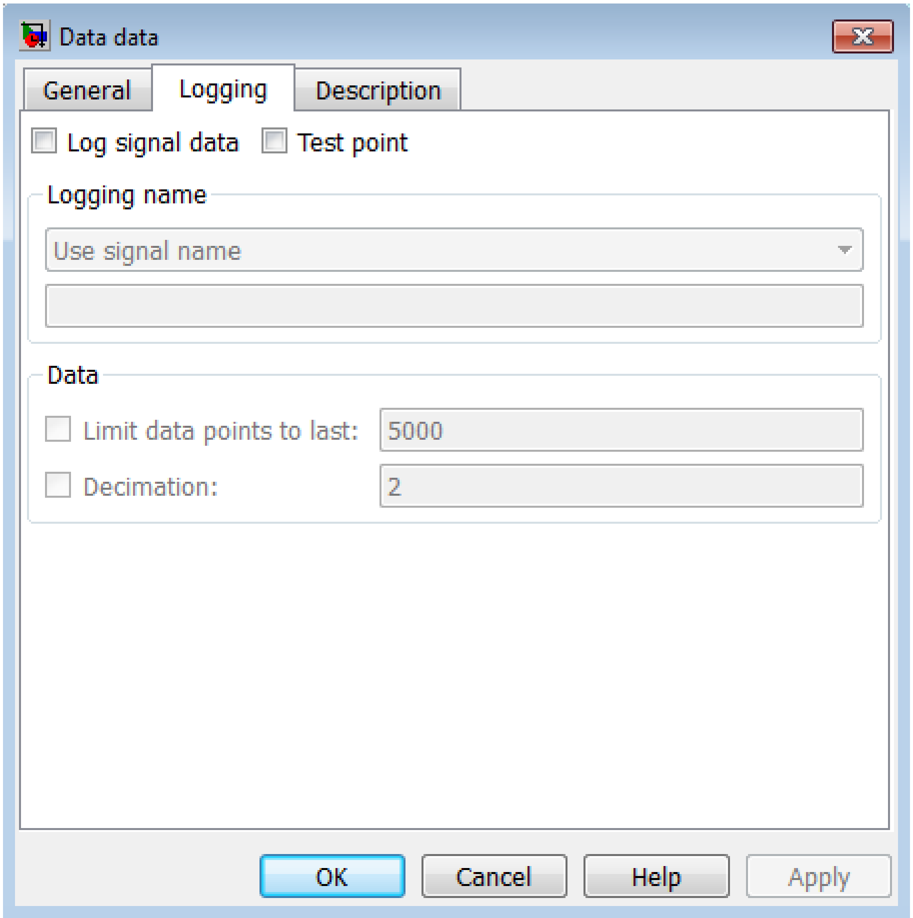
To correct this error, define `MySymbol` in the base workspace to provide a numeric value. If you click **Refresh Details**, the value of `MySymbol` appears in place of the unevaluated text, and the error indicator and description disappear.

To correct the overflow error for `Maximum`, perform one or more of the following changes so that the fixed-point data type can represent the maximum value you specify:

- Decrease the value in the **Maximum** field of the **Limit range** section.
- Increase **Word length**.
- Decrease **Fraction length**.

Properties You Can Set in the Logging Pane

The **Logging** pane of the Data properties dialog box appears as shown.



☐ Log signal data ☐ Test point

Logging name

Use signal name

Data

☐ Limit data points to last: 5000

☐ Decimation: 2

You can set these properties in the **Logging** pane.

Log signal data

Saves the data value to the MATLAB workspace during simulation.

Test point

Designates the data as a test point. A test point is a signal you can observe in a Floating Scope block in a model (see “Working with Test Points” in the Simulink documentation). Data objects can be test points if:

- Scope is **Local**
- Parent is not a Stateflow machine
- Data type is not **ml**

Logging name

Specifies the name associated with logged signal data. Simulink software uses the signal name as its logging name by default. To specify a custom logging name, select `Custom` from the list box and enter the new name in the adjacent edit field.

Limit data points to last

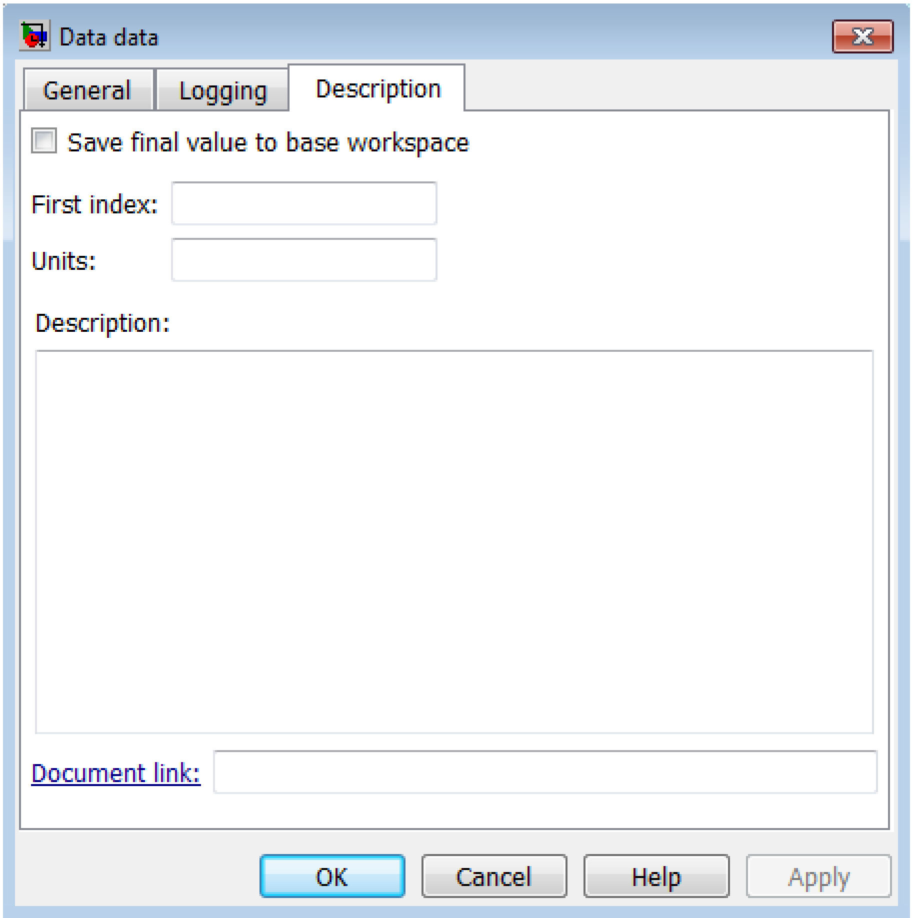
Limits the amount of data logged to the most recent samples.

Decimation

Limits the amount of data logged by skipping samples. For example, a decimation factor of 2 saves every other sample.

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.

Save final value to base workspace

Option that assigns the value of the data item to a variable of the same name in the base workspace at the end of simulation (see “Using Model Workspaces” in the Simulink documentation).

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.

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 8-11
- “Type” on page 8-12
- “Initial value” on page 8-13
- Minimum and Maximum (see “Limit range properties” on page 8-14)
- “Fixed-Point Data Properties” on page 8-14

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
First index	0
Size	• `1` (inherited), for inputs, parameters, and function outputs • `1` (scalar), for 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 8-32)
- Derive from a Simulink block parameter that you define and initialize in the parent masked subsystem (see "Sharing Simulink Parameters with Charts" on page 8-32)

You can mix both types of parameters in an expression.

Using Constants in Expressions

For expressions in the Data properties dialog box, you can use numeric constants of the appropriate type and size. Do not use Stateflow constants in these expressions.

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
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 Inputs, Outputs, and Parameters with Simulink and the MATLAB Workspace

In this section...

Sharing Input Data with Simulink

Data flows from Simulink into a chart via input ports on the Stateflow chart block.

To add input data to a chart:

1 Add a data object to the chart, as described in "Adding Data Using the Stateflow Editor" on page 8-3.

Note Add the data to the chart itself, 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 8-7.

3 Set the **Scope** property to `Input`.

An input port appears on the Stateflow chart block in the model.

You assign inputs to ports in the order in which you add the data. For example, you assign the second input to input port 2. 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 data, as described in “Typing Stateflow Data” on page 8-45.
5. Set the size of the input data, as described in “Sizing Stateflow Data” on page 8-54.

Note You cannot type or size Stateflow input data to accept frame-based data from Simulink.

Sharing Output Data with Simulink

Data flows from a chart into Simulink via output ports on the Stateflow chart block.

To add output data to a chart:

1. Add a data object to the chart, as described in “Adding Data Using the Stateflow Editor” on page 8-3.

   Note Add data to the chart itself, 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 8-7.
3. Set the **Scope** property to `Output`.

   An output port appears on the Stateflow chart block in the model.

   You assign outputs to ports in the order in which you add the data. For example, you assign 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 output data, as described in “Typing Stateflow Data” on page 8-45.

5. Set the size of the output data, as described in "Sizing Stateflow Data" on page 8-54.

Sharing Simulink Parameters with Charts

When to Share Simulink Parameters

Share Simulink parameters with Stateflow charts to maintain consistency with your Simulink model. By using parameters, you can also avoid hard-coding data sizes and types.

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 "Working with Block Masks" in the Simulink 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 chart defines data of scope **Parameter** as a constant. You cannot change a parameter value during model execution.

When simulation starts, Simulink tries to resolve the Stateflow data object to a parameter at the lowest level masked subsystem. If unsuccessful, Simulink moves up the model hierarchy to resolve the data object to a parameter at higher level masked subsystems.

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

Data Parent	Scope	When Initialized
Function (graphical, truth table, and 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 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 **Description** pane of the Data properties dialog box, select **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 Multiple Charts

In this section...

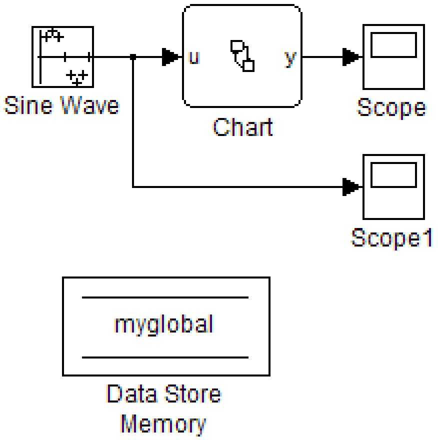
About Data Stores

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.

You can use data stores with buses, but not with arrays of buses. For more information about using data stores with buses, see "Using Data Stores with Buses and Arrays of Buses" in the Simulink documentation.

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:



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 8-36). 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 (see “Reading and Writing Global Data Programmatically” on page 8-38).

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 "Best Practices for Using Data Stores in Stateflow Charts" on page 8-41.

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

In the Model Explorer, follow these steps:

1. Select **Add** > **Data**.

   The Model Explorer adds a data object to the Stateflow chart.

2. 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.

3. Click **OK**.

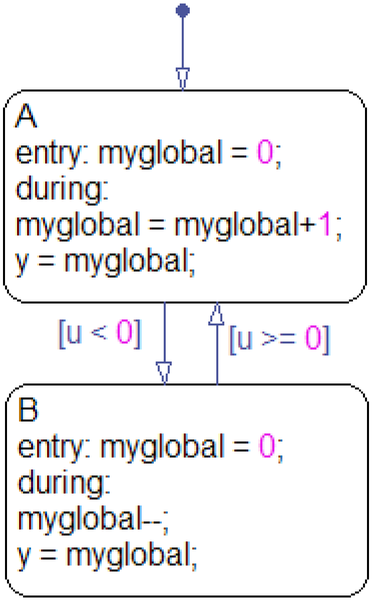
Resolving Data Store Bindings

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. 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.

The following chart 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 “Best Practices for Using Data Stores in Stateflow Charts” on page 8-41.

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**.

Creating a Global Data Store Across Multiple Models

To create read/write references to a global data store that you can share across multiple models:

1. Define data store memory objects that reside in each chart that shares the data.
   1. Use the Model Explorer to add a data object to each chart, 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`.

2. Verify that your models do not contain any Data Store Memory blocks.

   However, you can include Data Store Read and Data Store Write blocks.

3. Create a `Simulink.Signal` object in the MATLAB base workspace.
   1. In the Model Explorer, navigate to **Simulink Root > Base Workspace** in the **Model Hierarchy** pane.
   2. Select **Add > Simulink.Signal**.
   3. Give the object the same name as the data store memory objects in your charts.

4. Verify that these settings apply to the `Simulink.Signal` object:
   1. Set **Data type** to an explicit data type.

      The data type cannot be `auto`.
   2. Set **Dimensions** to be fully specified.

      The signal dimensions cannot be –1, or inherited.

c Set **Complexity** to `real`.

d Set **Sample mode** to `Sample based`.

e Set **Storage class** to `ExportedGlobal`.

Best Practices for Using Data Stores in Stateflow Charts

When Binding to Data Stores in 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 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 documentation.

Sharing Chart Data with External Modules

In this section...

Methods of Sharing Chart 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 19-25 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 19-26 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 8-7.

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 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 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	Enumerated	Expression	Bus Object
Local		yes	yes	yes	yes	yes
Constant		yes	yes		yes	yes
Parameter	yes	yes	yes	yes	yes	yes
Input	yes	yes	yes	yes	yes	yes
Output	yes	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 Output Data with Simulink” on page 8-31). • If scope is **Output**, you inherit the data type from the Simulink output signal on the designated output port (see “Sharing Output Data with Simulink” on page 8-31).

Mode	What To Specify
	Note Avoid inheriting data types from output signals. See “Avoid inheriting output data properties from Simulink blocks” on page 8-78. • 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 Inputs, Outputs, and Parameters with Simulink and the MATLAB Workspace” on page 8-30). • 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 Multiple Charts” on page 8-35).
Built in	Select a data type from the drop-down list of supported data types, as described in “Built-In Data Types” on page 8-49.
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 8-14.
Enumerated	Specify the class name for the enumerated data type. For more information, see Chapter 15, “Using Enumerated Data in Stateflow Charts”.

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 8-51 • `type` operator to specify the type of previously defined data, as described in "Deriving Data Types from Previously Defined Data" on page 8-50 • `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 8-27.
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 Editor. **Note** You can also inherit bus object properties from Simulink signals. See "Using Composite Signals" in the Simulink documentation. For more information about Stateflow bus object structures, see Chapter 20, "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 10-47. **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:

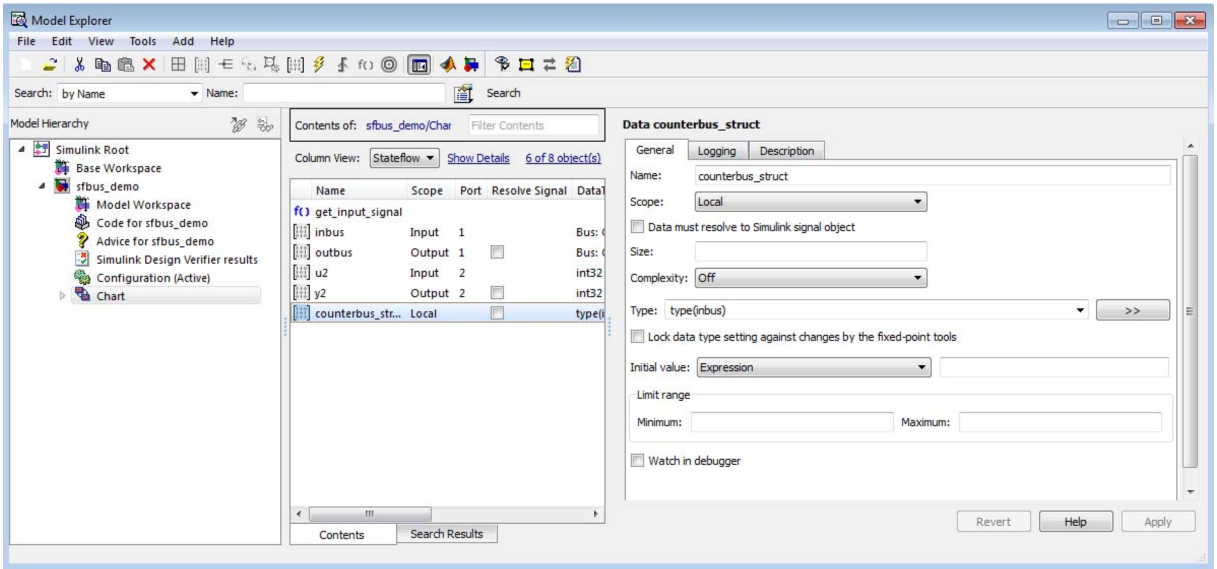
Scope:	Can inherit type from:
Input	Simulink input signal connected to corresponding input port in Stateflow chart
Output	Simulink output signal connected to corresponding output port in Stateflow chart **Note** Avoid inheriting data types from output signals. See “Avoid inheriting output data properties from Simulink blocks” on page 8-78.
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 8-45.

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.

Deriving Data Types from Previously Defined Data

You can use the `type` operator to derive data types from previously defined data. In the following example, the expression `type(inbus)` specifies the data type of the Stateflow structure `counterbus_struct`, where `inbus` is defined by the `Simulink.Bus` object `COUNTERBUS`. Therefore, the structure `counterbus_struct` also derives its data type from the bus object `COUNTERBUS`.



Name	Scope	Port	Resolve Signal	DataT
f() get_input_signal				
inbus	Input	1		Bus: (
outbus	Output	1	☐	Bus: (
u2	Input	2		int32
y2	Output	2	☐	int32
counterbus_str...	Local		☐	type(i

Data counterbus_struct

Name: counterbus_struct

Scope: Local

☐ Data must resolve to Simulink signal object

Size:

Complexity: Off

Type: type(inbus) >>

☐ Lock data type setting against changes by the fixed-point tools

Initial value: Expression

Limit range

Minimum: Maximum:

☐ Watch in debugger

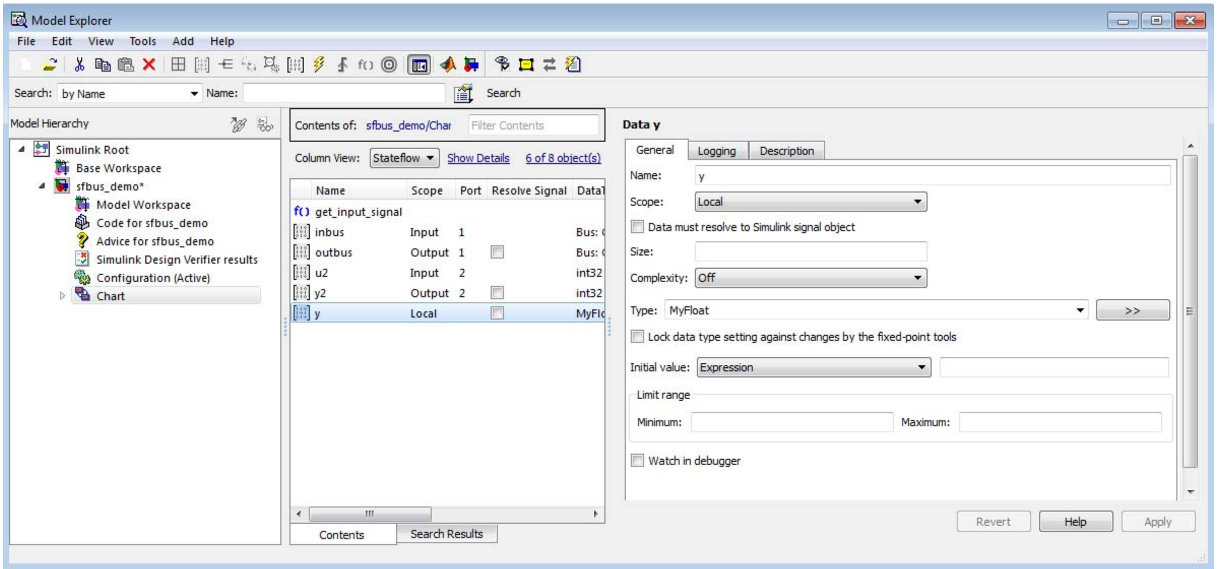
After you build your model, the **Compiled Type** column of the Model Explorer shows the type of each data object in the compiled simulation application. For more information, see “type Operator” on page 10-26.

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). Suppose that you define a data type alias named `MyFloat` as follows:

```
MyFloat = Simulink.AliasType;
MyFloat.BaseType = 'single';
```

In the following example, the data `y` has the same type as `MyFloat`.



Name	Scope	Port	Resolve Signal	DataT
f() get_input_signal				
inbus	Input	1		Bus: (
outbus	Output	1	☐	Bus: (
u2	Input	2		int32
y2	Output	2	☐	int32
y	Local		☐	MyFlo

Name: y

Scope: Local

☐ Data must resolve to Simulink signal object

Size:

Complexity: Off

Type: MyFloat

☐ Lock data type setting against changes by the fixed-point tools

Initial value: Expression

Limit range

Minimum: Maximum:

☐ Watch in debugger

After you build your model, the **Compiled Type** column of the Model Explorer shows the type used in the compiled simulation application.

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, select **Use Strong Data Typing with Simulink I/O** in the Chart properties dialog box (see "Specifying Chart Properties" on page 19-4). When you select this check box, the chart accepts input signals of any data type that Simulink supports, as long as the data type of the input signal matches the type of the corresponding Stateflow data object. Otherwise, you receive a type mismatch error.

Note For fixed-point data, select **Use Strong Data Typing with Simulink I/O** 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...

Methods for Sizing Stateflow Data

You can specify the size of Stateflow data by:

- Inheriting the size from a Simulink signal
- Using numeric values
- Using MATLAB expressions

Support for a sizing method depends on the scope of your data:

Scope of Data	Method for Sizing Data		
	Inherit the Size	Use Numeric Values	Use MATLAB Expressions
Local	No	Yes	Yes
Constant	No	Yes	Yes
Parameter	No	Yes	Yes
Input	Yes	Yes	Yes

Scope of Data	Method for Sizing Data		
	Inherit the Size	Use Numeric Values	Use MATLAB Expressions
Output	Yes	Yes	Yes
Data store memory	Yes	No	No

Stateflow data store memory inherits all data properties, including size, from the Simulink data store to which it resolves. You cannot specify any properties explicitly for data store memory.

How to Specify Data Size

Using the Size Field of the Data Properties Dialog Box

To specify the size of Stateflow data in the Data properties dialog box, you use the **Size** field, as described in "Properties You Can Set in the General Pane" on page 8-8. For more information, see:

- "Inheriting Input or Output Size from Simulink Signals" on page 8-55
- "Guidelines for Sizing Data with Numeric Values" on page 8-56
- "Guidelines for Sizing Data with MATLAB Expressions" on page 8-57

Setting the Stateflow.Data Object Property

To specify the size of Stateflow data using API commands, you set the `Props.Array.Size` property to a numeric value or a MATLAB expression that represents a scalar, vector, matrix, or n-dimensional array. For more information on using the API, see "Data Properties" in the Stateflow API documentation.

Inheriting Input or Output Size from Simulink Signals

To configure Stateflow input and output data to inherit size from the corresponding Simulink input and output signals, enter –1 in the **Size** field of the Data properties dialog box. This default setting applies to input and output data that you add to your chart. After you build your model, the

Compiled Size column of the Model Explorer displays the actual size that the compiled simulation application uses.

The equivalent API command for specifying an inherited data size is:

```
data_handle.Props.Array.Size = '-1';
```

Chart actions that store values in the specified output infer the inherited size of output data. If the expected size in the Simulink signal matches the inferred size, inheritance is successful. Otherwise, a mismatch occurs during build time.

Note Stateflow charts cannot inherit frame-based data sizes from Simulink signals.

Guidelines for Sizing Data with Numeric Values

When you specify data size using numeric values in the **Size** field of the Data properties dialog box, follow these guidelines:

Dimensionality	What to Specify in the Dialog Box	Equivalent API Command
Scalar	`1` (or leave the field blank)	`data_handle.Props.Array.Size = '1';` `data_handle.Props.Array.Size = '';`
Vector	The number of elements in the row or column vector	`data_handle.Props.Array.Size = 'number_of_elements';`

Dimensionality	What to Specify in the Dialog Box	Equivalent API Command
Matrix	An expression of the format `[r c]`, where: - *`r`* is the number of rows - *`c`* is the number of columns	`data_handle.Props.Array.Size = '[r c]';`
N-dimensional array	An expression of the format `[Size_of_dim1 Size_of_dim2 ... Size_of_dimN]`, where: - *`Size_of_dim1`* is the size of the first dimension - *`Size_of_dim2`* is the size of the second dimension - *`Size_of_dimN`* is the size of the `N`-th dimension	`data_handle.Props.Array.Size = '[Size_of_dim1 Size_of_dim2 ... Size_of_dimN];`

One-dimensional Stateflow vectors are compatible with Simulink row or column vectors of the same size. For example, Stateflow input or output data of size `3` is compatible with a Simulink row vector of size `[1 3]` or column vector of size `[3 1]`.

Guidelines for Sizing Data with MATLAB Expressions

When you specify data size using MATLAB expressions, follow the same guidelines that apply to sizing with numeric values (see "Guidelines for Sizing Data with Numeric Values" on page 8-56). The following guidelines also apply.

- Expressions that specify the size of a dimension:
  - Can contain a mix of numeric values, variables, arithmetic operators, parameters, and calls to MATLAB functions.
  - Must evaluate to a positive integer value.

- To specify inherited data size, you must enter –1 in the **Size** field or set the `Props.Array.Size` property for the data to –1. Expressions cannot evaluate to a value of –1.
- If the expression contains an enumerated value, you must include the type prefix for consistency with MATLAB naming rules.

  For example, `Colors.Red` is valid but `Red` is not.
- You cannot size Stateflow input data with an expression that accepts frame-based data from Simulink.

Examples of Valid Data Size Expressions

The following examples are valid MATLAB expressions for sizing data in your chart:

- `K+3`, where `K` is a chart-level Stateflow data
- `N/2`, where `N` is a variable in the MATLAB base workspace
- `2*Colors.Red`, where `Red` is an enumerated value of type `Colors`
- `[fi(2,1,16,2) fi(4,1,16,2)]`, which specifies a data size of `[2 4]` using a signed fixed-point type with word length of 16 and fraction length of 2

Name Conflict Resolution for Variables in Size Expressions

When multiple variables with identical names exist in a model, the variable with the highest priority applies:

1. Mask parameters
2. Model workspace
3. MATLAB base workspace
4. Stateflow data

Best Practices for Sizing Stateflow Data

Avoid use of variables that can lead to naming conflicts

For example, if a variable named `off` exists in the MATLAB base workspace and as local chart data, do not use `off` in the **Size** field of the Data properties dialog box.

Avoid use of size(u) expressions

Instead of using a `size(u)` expression, use a MATLAB expression that evaluates directly to the size of Stateflow data.

Handling Integer Overflow for Chart Data

In this section...

When Integer Overflow Can Occur

For some arithmetic operations, a processor might need to take an n-bit fixed-point value and store it in m bits, where $m \neq n$. If $m < n$, the reduced range of the value can cause an overflow for an arithmetic operation. Some processors identify this overflow as `Inf` or `NaN`. Other processors, especially digital signal processors (DSPs), handle overflows by saturating or wrapping the value.

For more information about saturation and wrapping for integer overflow, see “What Are Saturation and Wrapping?” in the Simulink Fixed Point documentation.

Support for Handling Integer Overflow in Charts

For Stateflow charts, you can control whether or not saturation occurs for integer overflow. Use the chart property, **Saturate on integer overflow**, to control overflow handling.

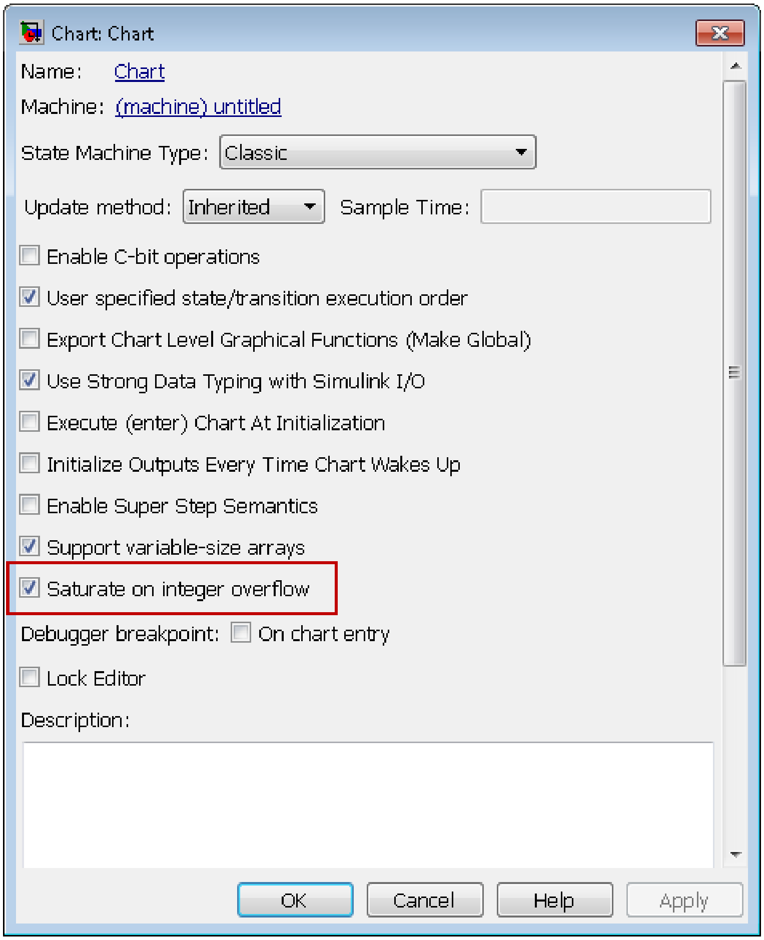


Chart: Chart

Name: Chart

Machine: (machine) untitled

State Machine Type: Classic

Update method: Inherited   Sample Time:

☐ Enable C-bit operations

☑ 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

☑ Support variable-size arrays

☑ Saturate on integer overflow

Debugger breakpoint: ☐ On chart entry

☐ Lock Editor

Description:

Check Box	When to Use This Setting	Overflow Handling	Example of the Result
Selected	Overflow is possible for data in your chart and you want explicit saturation protection in the generated code.	Overflows saturate to either the minimum or maximum value that the data type can represent.	An overflow associated with a signed 8-bit integer saturates to –128 or +127 in the generated code.
Cleared	You want to optimize efficiency of the generated code.	The handling of overflows depends on the C compiler that you use for generating code.	The number 130 does not fit in a signed 8-bit integer and wraps to –126 in the generated code.

Arithmetic operations for which you can enable saturation protection are:

- Unary minus: `–a`
- Binary operations: `a + b`, `a – b`, `a * b`, `a / b`, `a ^ b`
- Assignment operations: `a += b`, `a –=b`, `a *= b`, `a /= b`

Keep the following considerations in mind when you select **Saturate on integer overflow**:

- Saturation applies to all intermediate operations, not just the output or final result.
- The code generator can detect cases when overflow is not possible. In these cases, the generated code does not include saturation protection.

Effect of Integer Promotion Rules on Saturation

Stateflow charts use ANSI® C rules for integer promotion.

- All arithmetic operations use a data type that has the same word length as the target word size. Therefore, the intermediate data type in a chained arithmetic operation can be different from the data type of the operands or the final result.

- For operands with integer types smaller than the target word size, promotion to a larger type of the same word length as the target size occurs. This implicit cast occurs before any arithmetic operations take place.

  For example, when the target word size is 32 bits, an implicit cast to `int32` occurs for operands with a type of `uint8`, `uint16`, `int8`, or `int16` before any arithmetic operations occur.

Suppose that you have the following expression, where `y`, `u1`, `u2`, and `u3` are of `uint8` type:

```
y = (u1 + u2) - u3;
```

Based on integer promotion rules, that expression is equivalent to the following statements:

```
uint8_T u1, u2, u3, y;
int32_T tmp, result;
tmp = (int32_T) u1 + (int32_T) u2;
result = tmp - (int32_T) u3;
y = (uint8_T) result;
```

For each calculation, the following data types and saturation limits apply.

Calculation	Data Type	Saturation Limits
`tmp`	`int32`	`(MIN_INT32, MAX_INT32)`
`result`	`int32`	`(MIN_INT32, MAX_INT32)`
`y`	`uint8`	`(MIN_UINT8, MAX_UINT8)`

Suppose that `u1`, `u2`, and `u3` are equal to 200. Because the saturation limits depend on the intermediate data types and not the operand types, you get the following values:

- `tmp` is 400.
- `result` is 200.
- `y` is 200.

Impact of Saturation on Debugger Checks

Suppose that you select **Enable overflow detection (with debugging)** in the **Simulation Target** pane of the Configuration Parameters dialog box. When you select **Saturate on integer overflow**, the Stateflow debugger does not flag cases of integer overflow during simulation. However, the debugger continues to flag the following situations:

- Out-of-range data violations based on minimum and maximum range checks
- Division-by-zero operations

Defining Temporary Data

In this section...
"When to Define Temporary Data" on page 8-65
"How to Define Temporary Data" on page 8-65

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 MATLAB functions in your chart. For example, you can designate a loop counter to have **Temporary** scope if the counter value does not need to persist after the function completes.

How to Define Temporary Data

To define temporary data for a Stateflow function, follow these steps:

1. Open the Model Explorer.

2. In the Model Explorer, select the graphical, truth table, or MATLAB function that will use temporary data.

3. Select **Add > Data**.

   The Model Explorer adds a default definition for the data in the Stateflow hierarchy, with a scope set to **Temporary** by default.

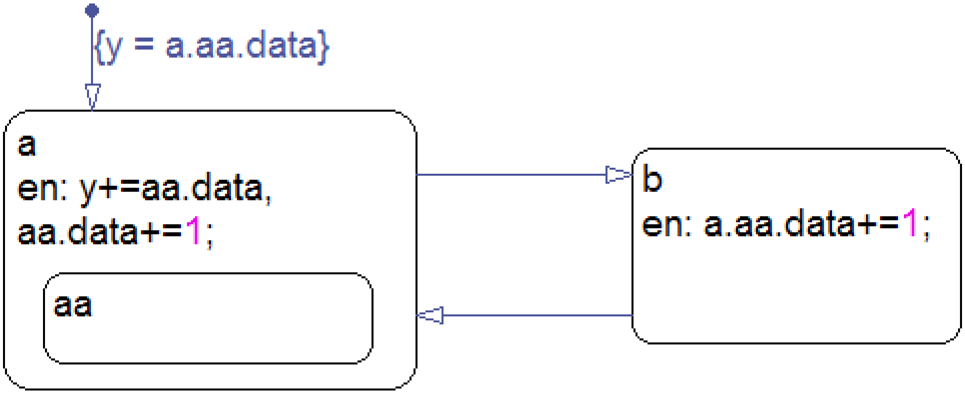
4. Change other properties of the data if necessary, as described in "Setting Data Properties in the Data Dialog Box" on page 8-5.

Using Dot Notation to Identify Data in a Chart

In this section...

What Is Dot Notation?

Dot notation is a way to identify data at a specific level of the Stateflow chart hierarchy. For example, you can use dot notation for data identifiers in state actions and transitions.



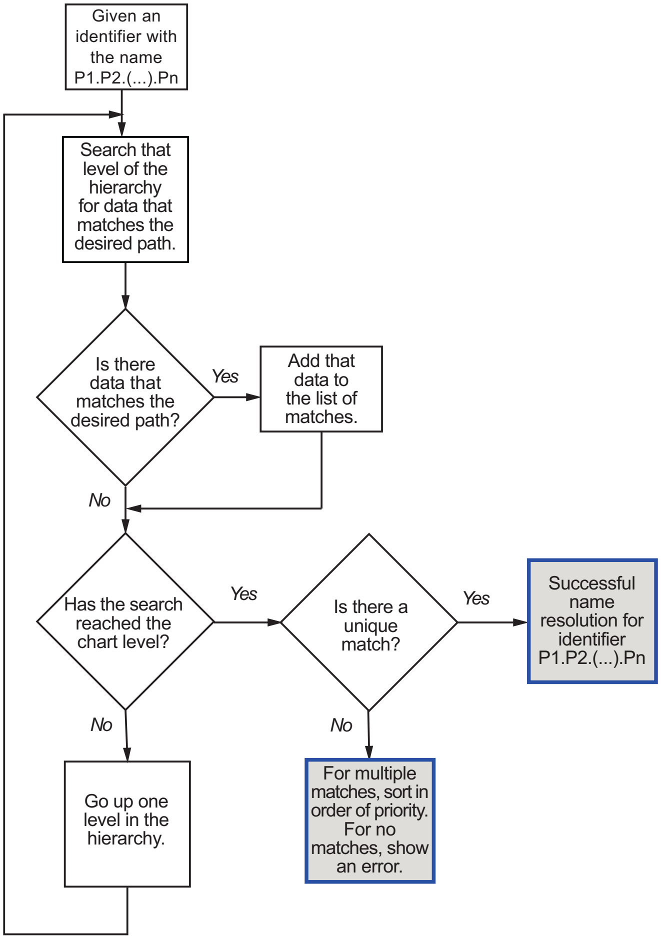
In this chart, `data` resides in the state `aa`. Identifiers in state actions and transitions use dot notation to refer to this data.

- In state `a`, the entry action contains the identifier `aa.data`.
- In state `b`, the entry action contains the identifier `a.aa.data`.
- In the default transition, the action contains the identifier `a.aa.data`.

Resolution of Data Identifiers with Dot Notation

During simulation, the chart searches for data that matches the identifier with dot notation. These rules apply:

- The chart does not do an exhaustive search of all data.
- The chart does not stop searching after finding one match. The search continues until it reaches the chart level.



Process for Resolving Data Identifiers with Dot Notation

The flow chart describes the following search process.

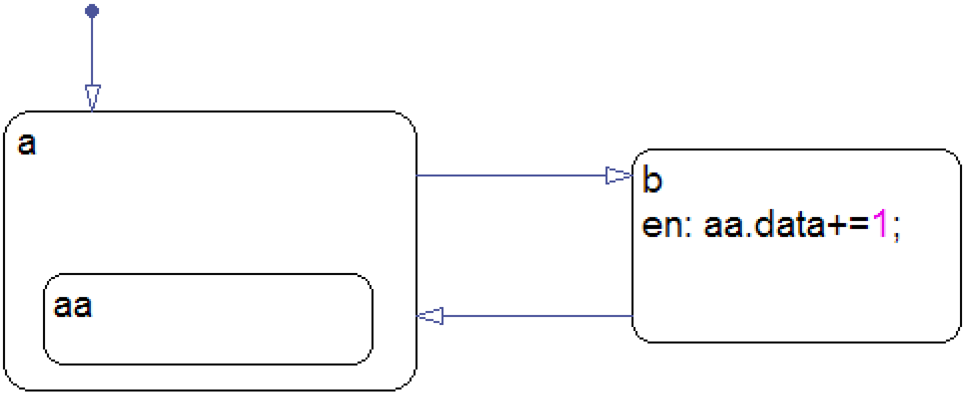
Stage	Action
1	The search begins at the level of the hierarchy where the identifier appears. • For a state action, that state is the starting point. • For a transition label, the parent of the source object is the starting point.
2	The chart searches at that level of the hierarchy for a path to the data. If the chart finds a match, it adds that path to the list of possible matches.
3	The chart moves up to the next highest level of the hierarchy. At that level, the chart searches for a path to the data. If the chart finds a match, it adds that path to the list of possible matches.
4	The previous step repeats until the search reaches the chart level.
5	At the chart level, one more search occurs for a path to the data. If a match exists, that path becomes part of the list of possible matches. Then, the search ends.
6	After the search ends, one of the following occurs: • If a unique match exists, the statement containing the data identifier executes. • If multiple matches exist, the chart sorts them in this order of priority: **1** Local data in a state, subchart, or function **2** Field name of a bus object (see Chapter 20, “Working with Structures and Bus Signals in Stateflow Charts”) **3** Value of an enumerated data type (see Chapter 15, “Using Enumerated Data in Stateflow Charts”) The statement containing the data identifier executes using the match of highest priority. • If no matches exist, an error message appears.

Best Practices for Using Dot Notation in Data Identifiers

These examples show how to avoid problems when using dot notation in data identifiers.

Use a Specific Path in the Identifier

Be specific when defining the path to the data.



Suppose that state `aa` contains `data`. In state `b`, the entry action contains the `aa.data` identifier that the chart cannot resolve. This search process occurs:

Stage	Action	Finds a Match?
1	Chooses state `b` as the starting point and searches at that level for an object `aa` that contains `data`.	No
2	Moves up to the next level of the hierarchy and searches at the chart level for an object `aa` that contains `data`.	No

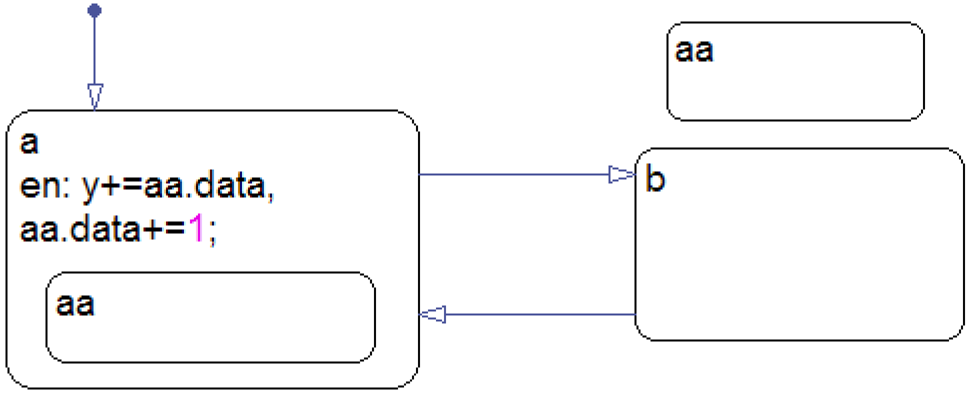
The search ends, and an error message appears because no match exists for the `aa.data` identifier.

To avoid this message, use a specific path in the identifier for the entry action in state `b`:

```
en: a.aa.data+=1;
```

Use Unique State Names

Use unique names when you name the states in a chart.



Suppose that both states named `aa` contain a data object named `data`. In state `a`, the entry action contains two `aa.data` identifiers that the chart cannot resolve. This search process occurs:

Stage	Action	Finds a Match?
1	Chooses state `a` as the starting point and searches at that level for an object `aa` that contains `data`.	Yes
2	Moves up to the next level of the hierarchy and searches at the chart level for an object `aa` that contains `data`.	Yes

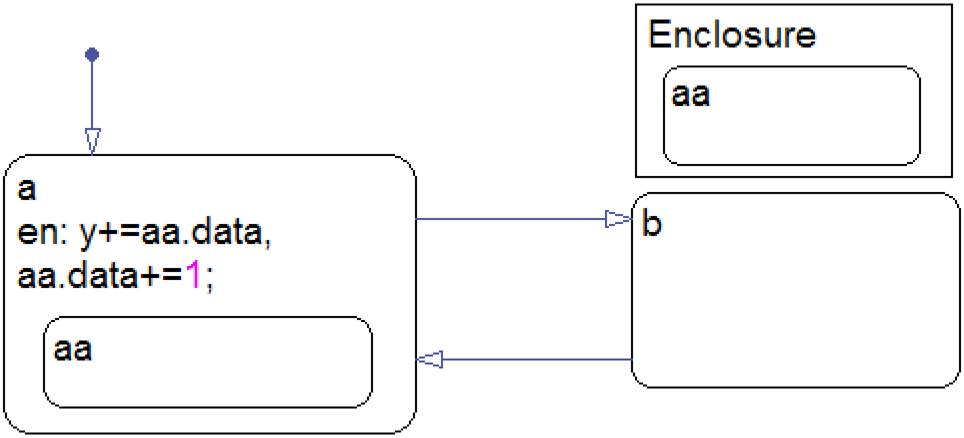
The search ends, and a warning message appears because multiple matches of equal priority exist for the `aa.data` identifiers.

To avoid this message, perform one of these corrective actions:

- Rename one of the two states named `aa`.
- Use a more specific path in the identifiers for the entry action in state `a`:

  ```
  en: y+=a.aa.data, a.aa.data+=1;
  ```

- Enclose the outer state `aa` in a box or another state. Adding an enclosure prevents the search process from detecting that outer state.



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 "Custom Storage Classes" in the Embedded Coder 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.

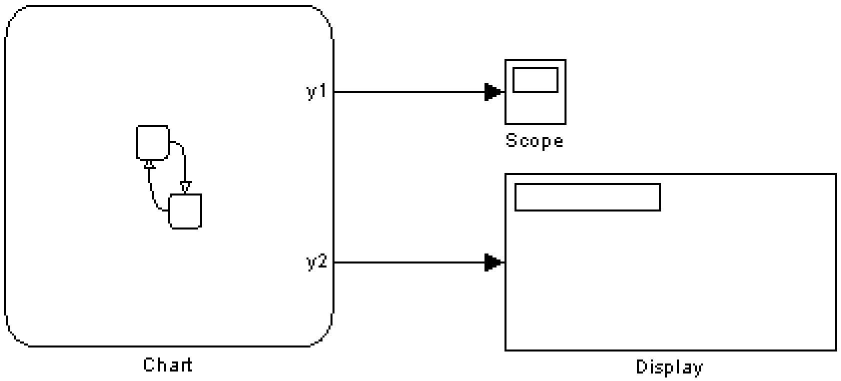
3. Enter a name for your data that matches the name of the `Simulink.Signal` object.

4. In the Data properties dialog box, select the **Data must resolve to Simulink signal object** check box.

   After you select this check box, the dialog box removes or grays out the properties that your data inherits from the signal. For a list of properties that your data can inherit during signal resolution, see "Inherited Properties" on page 8-72.

A Simple Example

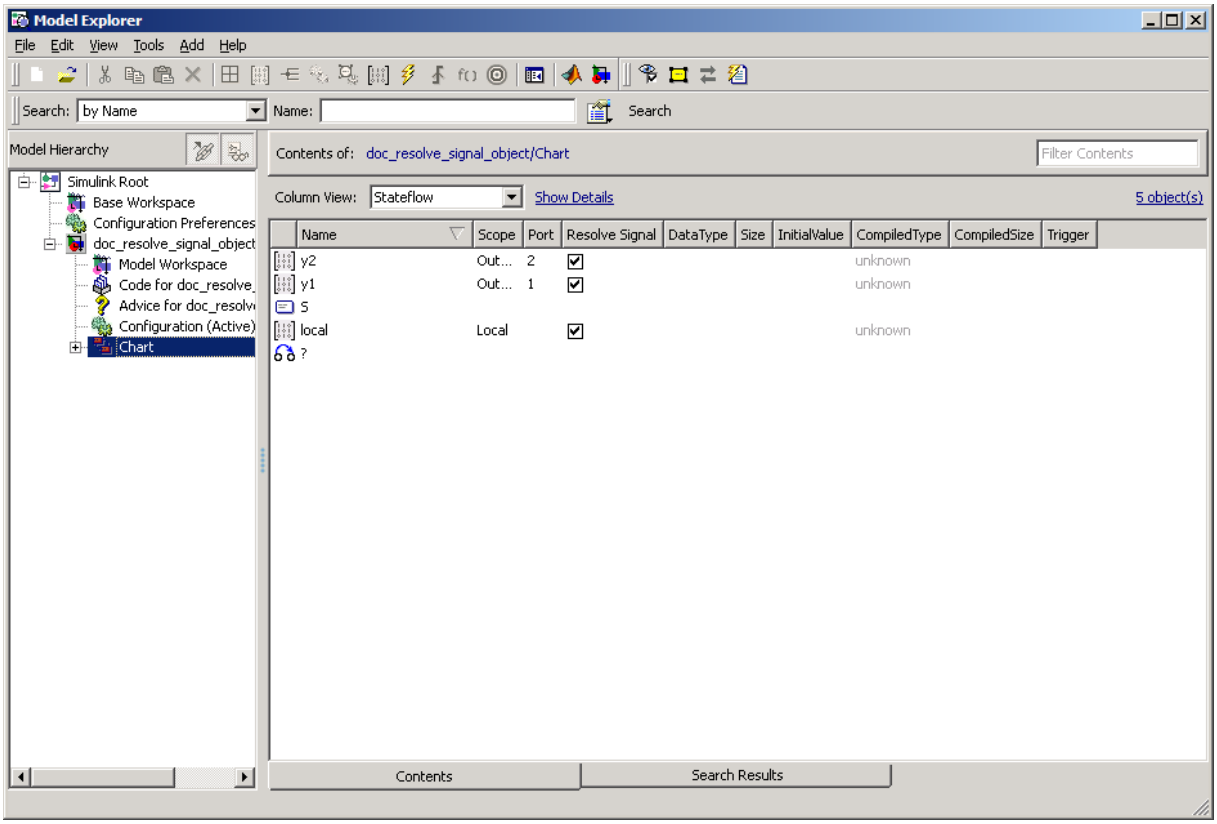
The following model shows how a Stateflow chart resolves local and output data to `Simulink.Signal` objects.



In the base workspace, there are three `Simulink.Signal` objects with these properties:

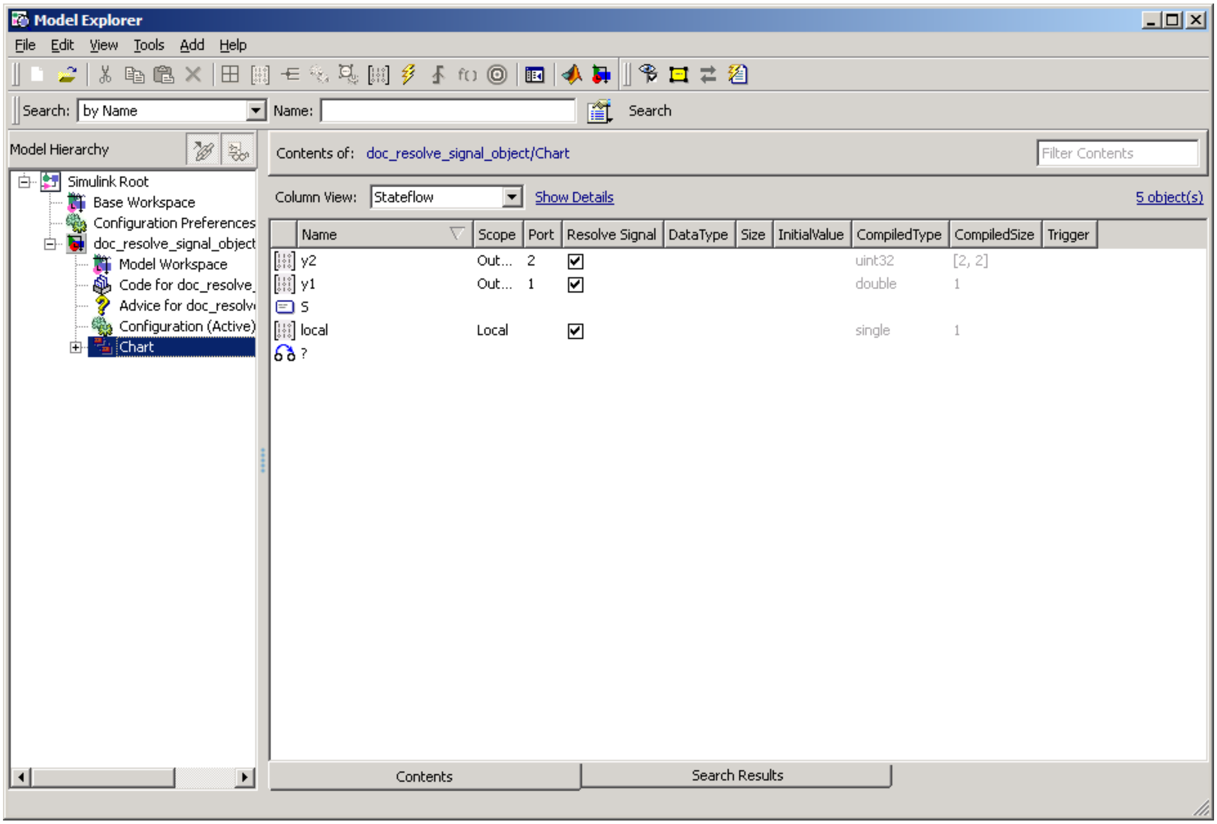
Name	Data Type	Dimensions	Storage Class
`y1`	`double`	1	`SimulinkGlobal`
`y2`	`uint32`	[2 2]	`Auto`
`local`	`single`	1	`ExportedGlobal`

The chart contains three data objects — two outputs and a local variable — that will resolve to a signal with the same name, as follows:



Name	Scope	Port	Resolve Signal	DataType	Size	InitialValue	CompiledType	CompiledSize	Trigger
y2	Out...	2	☑				unknown		
y1	Out...	1	☑				unknown		
S									
local	Local		☑				unknown		
?									

When you build the model, each data object inherits the properties of the identically named signal:



Search: by Name

Name:

Contents of: doc_resolve_signal_object/Chart

Column View: Stateflow

Name	Scope	Port	Resolve Signal	DataType	Size	InitialValue	CompiledType	CompiledSize	Trigger
y2	Out...	2	☑				uint32	[2, 2]	
y1	Out...	1	☑				double	1	
s									
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 `local` to be an exported global variable:

```
/*
 * Exported States
 *
 * Note: Exported states are block states with an exported
 * global storage class designation.
 *
 */
extern real32_T local;                  /* '<Root>/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 and *not* from the code. However, in the MATLAB action language — used in truth tables and MATLAB functions — inherited properties of outputs are determined solely from the code and the properties of the inputs.

Restrict use of machine-parented data

Use machine-parented data when you want to use global data definitions for Mealy and Moore charts, which do not support data store memory (see Chapter 6, "Building Mealy and Moore Charts" for details). Otherwise, avoid using machine-parented data. The presence of machine-parented data in a model prevents reuse of generated code and other code optimizations. This type of data is also incompatible with many Simulink and Stateflow features.

For example, the following features do not support machine-parented data:

- Enumerated data (see Chapter 15, "Using Enumerated Data in Stateflow Charts")
- Simulink functions (see Chapter 24, "Using Simulink Functions in Stateflow Charts")
- Chart SimState (see Chapter 12, "Saving and Restoring Simulations with SimState")

- Implicit change events (see “Keywords for Implicit Events” on page 9-40)
- Detection of unused data (see “Diagnostic for Detecting Unused Data” on page 8-2)
- Model referencing (see “Limitations on All Model Referencing” in the Simulink documentation)
- Analysis by Simulink Design Verifier™ software
- Code generation by Simulink PLC Coder™ software

To make Stateflow data accessible to other charts and blocks in a model, use data store memory. For details, see “Sharing Global Data with Multiple Charts” on page 8-35.

Transferring Data Across Models

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.

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:

- Activate a Simulink triggered subsystem (see “Using Edge Triggers to Activate a Simulink Block” on page 9-24)
- Activate a Simulink function-call subsystem (see “Using Function Calls to Activate a Simulink Block” on page 9-33)
- Trigger actions in parallel states of a Stateflow chart (see “Broadcasting Events to Synchronize States” on page 10-59)

Although Stateflow software does not limit the number of events you can use in a chart, the underlying C compiler enforces a theoretical limit of (2^31)-1 events for the generated code.

When should I use conditions instead of events?

Use conditions on transitions when you want to:

- Represent conditional statements, for example, `x < 1` or `x == 0`
- Represent a change of input value from a Simulink block

For more information about using conditions on transitions, see "Transition Action Types" on page 10-7.

Types of Events

An explicit event is an event that you define and can have one of the following 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). See "Directed Event Broadcasting" on page 10-59.
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 9-24.

An implicit event is a built-in event that broadcasts automatically during chart execution (see "Using Implicit Events" on page 9-40).

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...**
Chart	The chart and all states and substates
Subchart	The subchart and all states and substates
State	The state and all substates

Diagnostic for Detecting Unused Events

If you have unused events in your chart, a warning message appears during simulation with a list of events you can remove. By removing objects that have no effect on simulation, you can reduce the size of your model. This diagnostic checks for usage of Stateflow events, except for the following types:

- Function-call input events
- Edge-triggered input events

After you select an event for removal, a dialog box confirms your choice. In this dialog box, you can specify that other deletions occur without confirmation. If you prevent the confirmation dialog box from appearing, you can reenable it at any time by typing at the command prompt:

```
sfpref('showDeleteUnusedConfGui', 1)
```

You can control the level of diagnostic action for unused events in the **Diagnostics > Stateflow** pane of the Configuration Parameters dialog box. For more information, see the documentation for the “Unused data and events” diagnostic.

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:

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**.

   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.

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**.

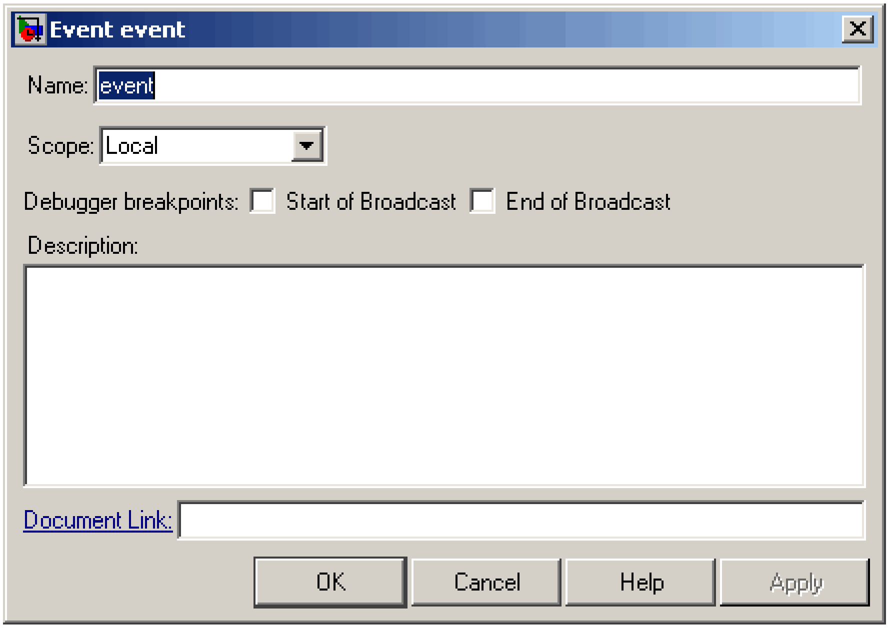
Setting Properties for an Event

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**.



Event event

Name: event

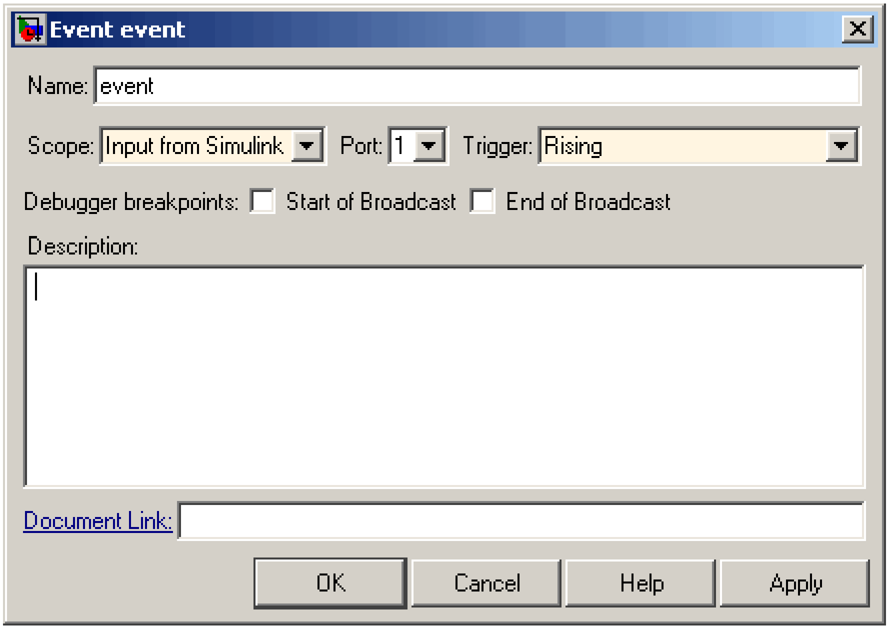
Scope: Local

Debugger breakpoints: ☐ Start of Broadcast ☐ End of Broadcast

Description:

Document Link:

For input events, the dialog box displays these properties and defaults.



Event event

Name: event

Scope: Input from Simulink Port: 1 Trigger: Rising

Debugger breakpoints: ☐ Start of Broadcast ☐ End of Broadcast

Description:

Document Link:

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 > Show Dialog Pane**.

    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 9-3.

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 “Association of Input Events with Control Signals” on page 9-14 and “Association of Output Events with Output Ports” on page 9-38.

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 9-24.

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:

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" on page 21-14.

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:

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” on page 21-14.

Association of Input Events with Control Signals

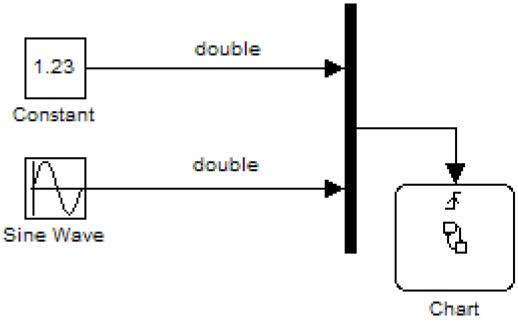
When you define one or more input events for a chart, a single trigger port to the chart block appears. External Simulink blocks can trigger the input events via a signal or vector of signals connected to the trigger port. The **Port** property of an input event associates the event with a specific element 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.

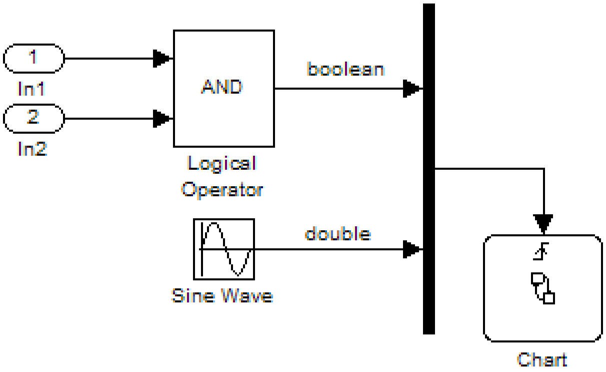
Data Types Allowed for Input Events

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.

Controlling States When Function-Call Inputs Reenable Charts

In this section...
"Setting Behavior for a Reenabled Chart" on page 9-16
"Behavior When the Parent Is the Model Root" on page 9-17
"Behavior When the Chart Is Inside a Model Block" on page 9-20

Setting Behavior for a Reenabled Chart

If you define a function-call input event for a chart, you can control the behavior of states when this event reenables the chart:

1. Open the Chart properties dialog box.

2. For **States When Enabling**, select one of these options:

   - `Held` — Maintain most recent values of the states.
   - `Reset` — Revert to the initial values of the states.
   - `Inherit` — Inherit this setting from the parent subsystem.

If...	The inherited setting is...
The parent of the chart is the model root	`Held`
The chart is inside a Model block	`Reset` For more information, see "Referencing a Model" in the Simulink documentation.

   For new charts, the default setting is `Held`.

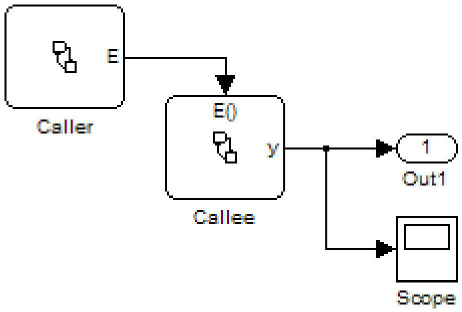
Behavior When the Parent Is the Model Root

When the parent of your chart is the model root, the following types of behavior can occur when a function-call input event reenables the chart.

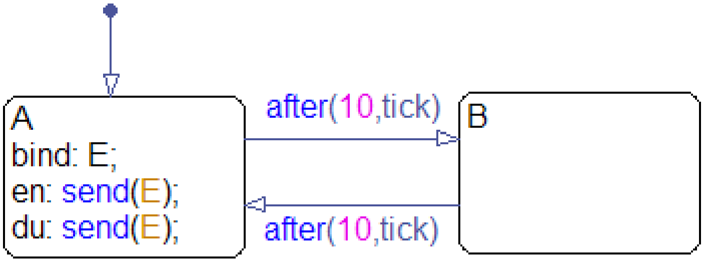
The chart...	When you set the States When Enabling property to...
Maintains the most recent values of the states	`Inherit` or `Held`
Reverts to the initial values of the states	`Reset`

What Happens When the Setting Is Inherit or Held

In the following model, the `Caller` chart uses the event `E` to wake up and execute the `Callee` chart.



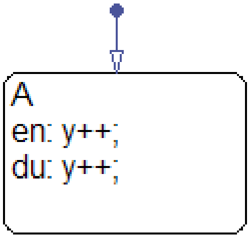
The `Caller` chart contains two states, `A` and `B`.



When you bind `E` to `A`:

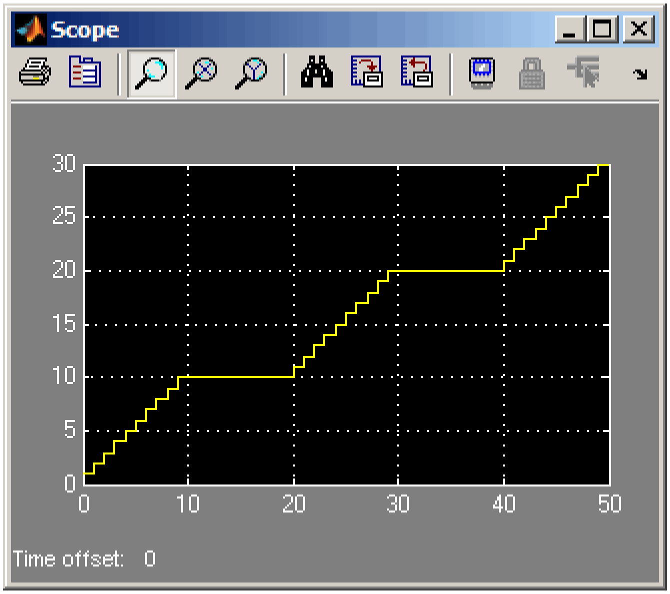
- Entering `A` enables the `Callee` chart.
- Exiting `A` disables the `Callee` chart.
- Reentering `A` reenables the `Callee` chart.

Each time the `Callee` chart executes, the output data `y` increments by one.



In the Chart properties dialog box for `Callee`, **States When Enabling** is `Inherit`. Because the parent of this chart is the model root, the chart maintains the most recent values of all states when reenabled.

During simulation, `Callee` maintains the most recent value of its state when the function-call input event reenables the chart at t = 20 and 40.



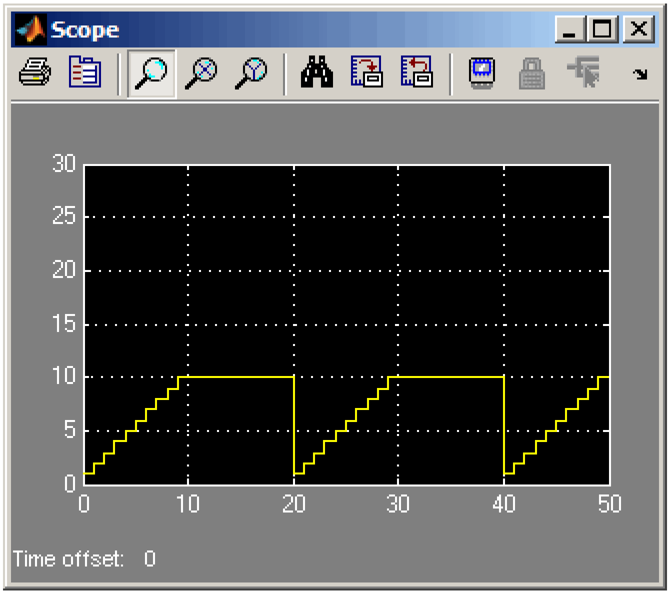
The key behaviors are:

Time Interval	Caller Chart	Callee Chart
0 – 10	State `A` is active and enables `Callee`.	State `A` executes by incrementing `y`.
10 – 20	State `B` is active and disables `Callee`.	State `A` does not execute.
20 – 30	State `A` is active and reenables `Callee`.	State `A` executes by incrementing `y`.
30 – 40	State `B` is active and disables `Callee`.	State `A` does not execute.
40 – 50	State `A` is active and reenables `Callee`.	State `A` executes by incrementing `y`.

If **States When Enabling** is `Held`, the output is the same.

What Happens When the Setting Is Reset

Suppose that the **States When Enabling** property is `Reset` for `Callee`. During simulation, `Callee` reverts to the initial value of its state when the function-call input event reenables the chart at t = 20 and 40.



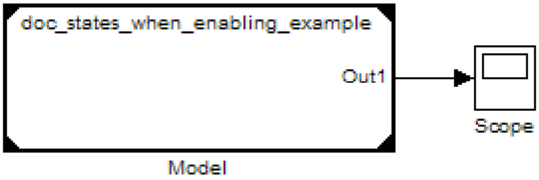
Behavior When the Chart Is Inside a Model Block

When your chart is inside a Model block, the following types of behavior can occur when a function-call input event reenables the chart.

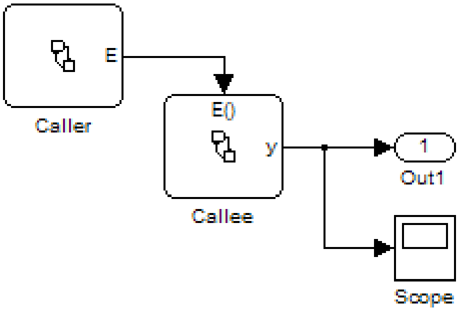
The chart...	When you set the States When Enabling property to...
Maintains the most recent values of the states	`Held`
Reverts to the initial values of the states	`Inherit` or `Reset`

What Happens When the Setting Is Inherit or Reset

The following model contains a Model block and a scope. (For more information about using Model blocks, see “Referencing a Model” in the Simulink documentation.)

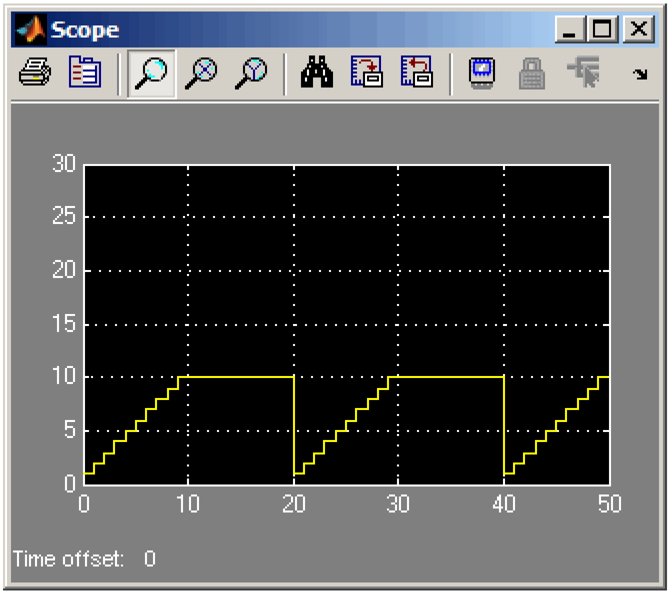


The Model block contains the `Caller` and `Callee` charts from “Behavior When the Parent Is the Model Root” on page 9-17.



In the Chart properties dialog box for `Callee`, **States When Enabling** is `Inherit`. Because this chart is inside a Model block, the chart reverts to the initial values of all states when reenabled.

During simulation, `Callee` reverts to the initial value of its state when the function-call input event reenables the chart at t = 20 and 40.



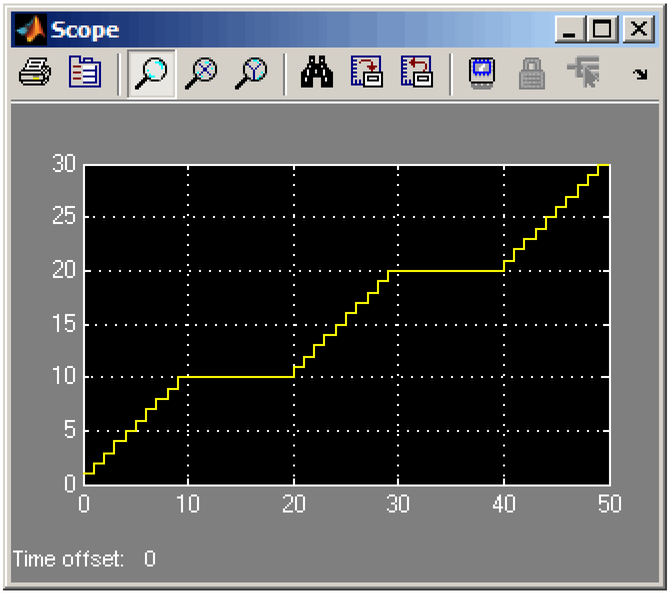
The key behaviors are:

Time Interval	Caller Chart	Callee Chart
0 – 10	State `A` is active and enables `Callee`.	State `A` executes by incrementing `y`.
10 – 20	State `B` is active and disables `Callee`.	State `A` does not execute.
20 – 30	State `A` is active and reenables `Callee`.	State `A` executes by incrementing `y`.
30 – 40	State `B` is active and disables `Callee`.	State `A` does not execute.
40 – 50	State `A` is active and reenables `Callee`.	State `A` executes by incrementing `y`.

If **States When Enabling** is `Reset`, the output is the same.

What Happens When the Setting Is Held

Suppose that the **States When Enabling** property is `Held` for `Callee`. During simulation, `Callee` maintains the most recent value of its state when the function-call input event reenables the chart at t = 20 and 40.



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:

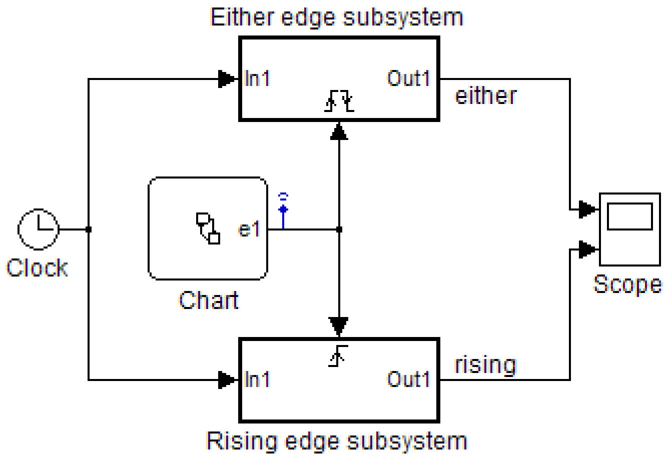
1. Add an event to the Stateflow chart, as described in “How to Define Events” on page 9-5.
2. Set the **Scope** property for the event to **Output to Simulink**.

   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`.

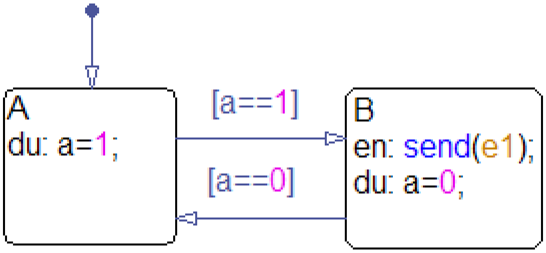
Note Unlike edge-triggered input events, you cannot specify a `Rising` or `Falling` edge trigger.

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.



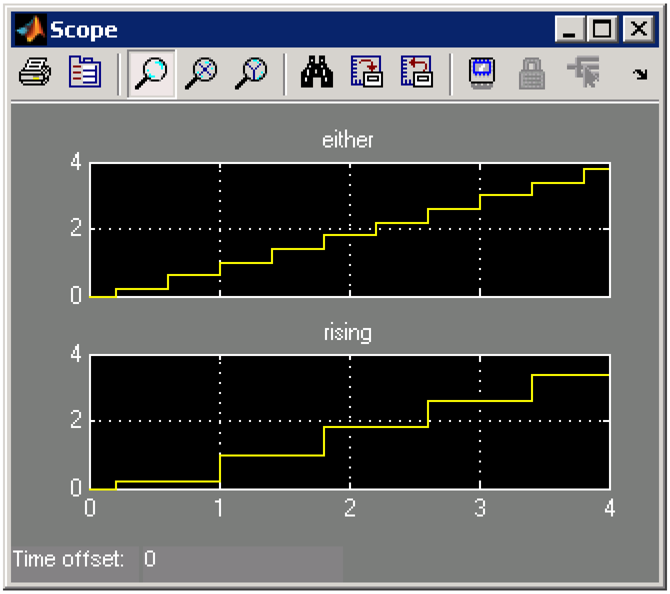
The chart contains the edge-triggered output event `e1` and the local data `a`, which switches between 0 and 1 during simulation.



In a 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` switches...**
Either edge subsystem	Every event broadcast	From 0 to 1, or from 1 to 0
Rising edge subsystem	Every other event broadcast	From 0 to 1

When you simulate the model, the scope shows these results.



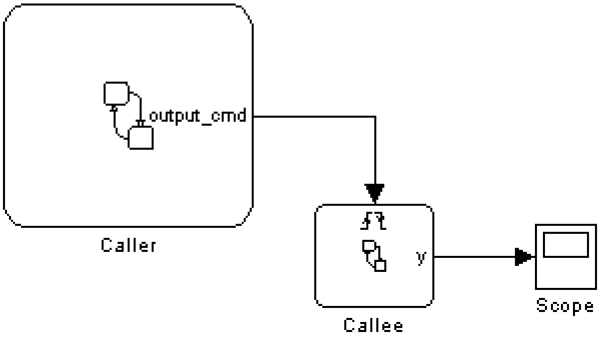
Queuing Behavior for Broadcasting an Edge-Triggered Output Event Multiple Times

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.)

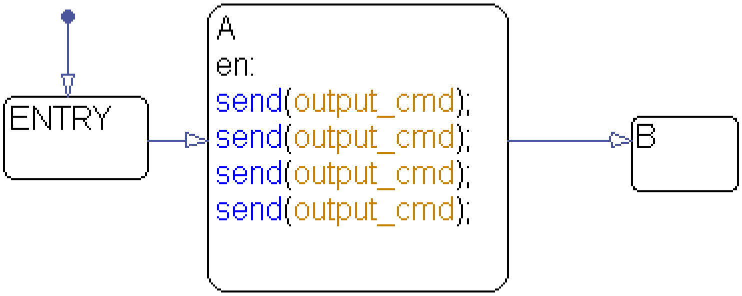
Note For information on what happens for function-call output events, see “Interleaving Behavior for Broadcasting a Function-Call Output Event Multiple Times” on page 9-36.

Example of Queuing Behavior for Edge-Triggered Output Events

In this model, the chart Caller uses the edge-triggered output event `output_cmd` to activate the chart Callee.

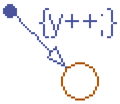


The chart Caller tries to broadcast the same edge-triggered output event four times in a single time step, as shown.

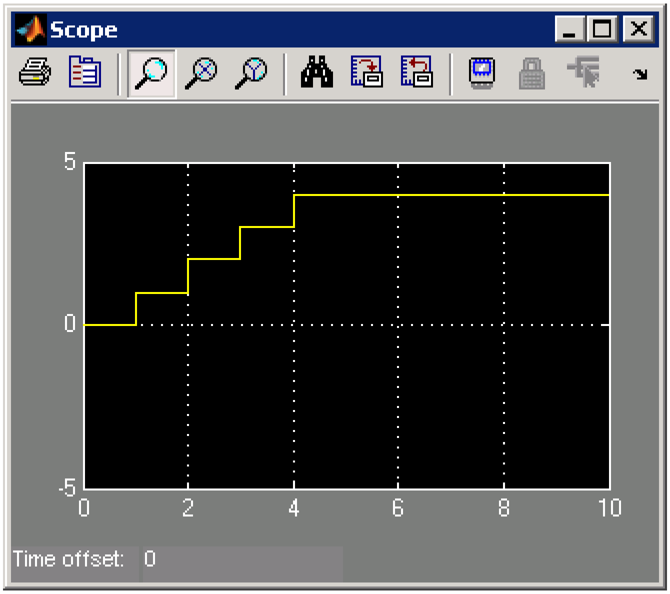


```
en:
send(output_cmd);
send(output_cmd);
send(output_cmd);
send(output_cmd);
```

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.

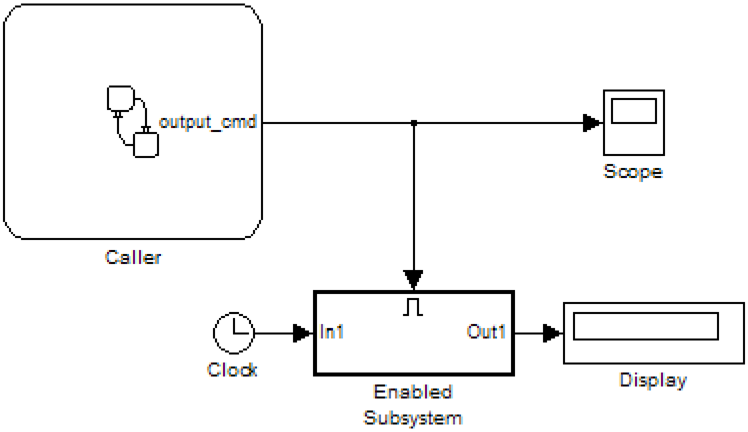
Example of Using Queuing Behavior to Approximate a Function Call

When you cannot use a function-call output event, such as for HDL code generation, you can approximate a function call by using:

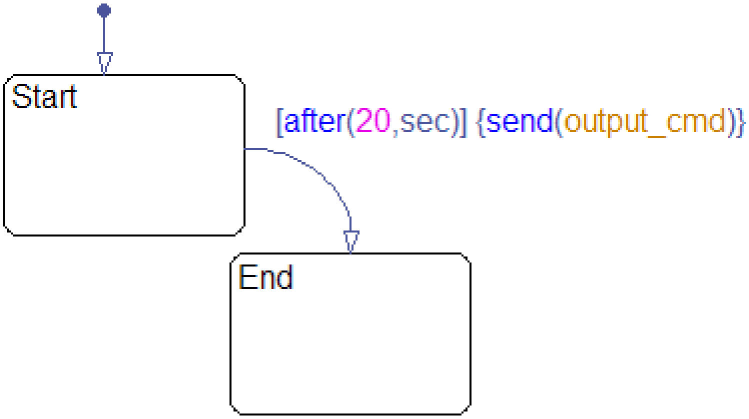
- An edge-triggered output event
- An enabled subsystem
- Two consecutive event broadcasts

Note While you can approximate a function call, a subtle difference in execution behavior exists. Execution of a function-call subsystem occurs *during* execution of the chart action that provides the trigger. However, execution of an enabled subsystem occurs *after* execution of that chart action is complete.

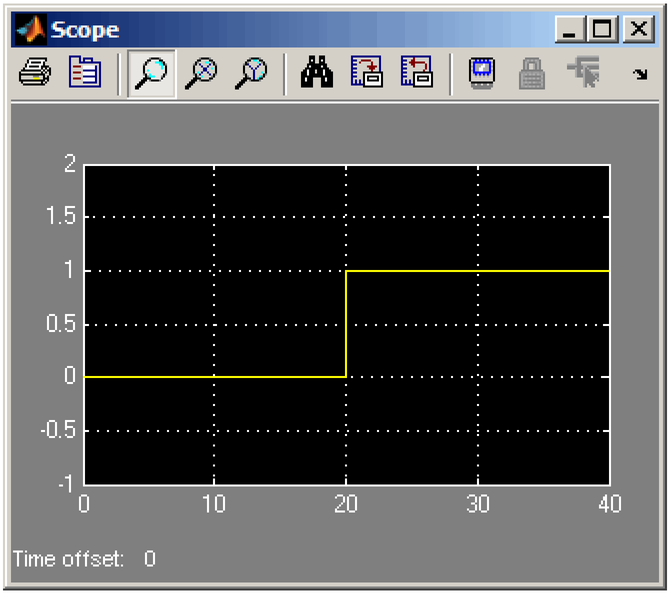
In the following model, the chart Caller uses the edge-triggered output event `output_cmd` to activate the enabled subsystem. The scope shows the value of the output event during simulation.



The chart Caller broadcasts the edge-triggered output event using a `send` action.

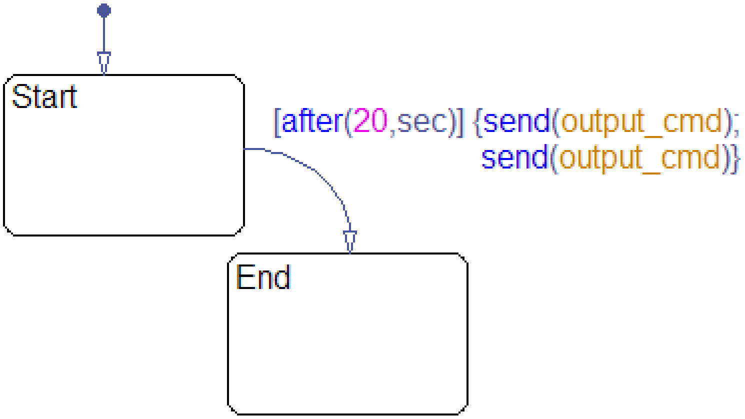


When you simulate the model, you see the following output in the scope. The simulation runs for 40 seconds.

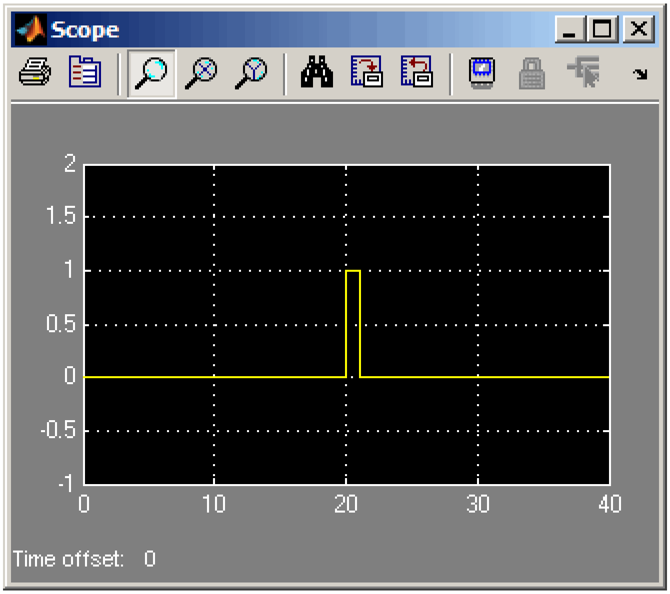


When simulation starts, the value of `output_cmd` is 0. At t = 20, the chart dispatches `output_cmd`. Because this value changes to 1, the enabled subsystem becomes active and executes during that time step. Because no other event broadcasts occur, the enabled subsystem continues to execute at every time step until simulation ends. Therefore, the Display block shows a final value of 40.

To approximate a function call, add a second event broadcast in the *same* action.



When you simulate the model, you see the following output in the scope. The simulation runs for 40 seconds.



When simulation starts, the value of `output_cmd` is 0. At t = 20, the chart dispatches `output_cmd`. Because this value changes to 1, the enabled subsystem becomes active and executes during that time step. The chart also *queues up* the second event for dispatch at the next time step. At t = 21, the chart dispatches the second output event, which changes the value of `output_cmd` to 0. Therefore, the enabled subsystem stops executing and the Display block shows a final value of 20.

The queuing behavior of consecutive edge-triggered output events enables you to approximate a function call with an enabled subsystem.

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

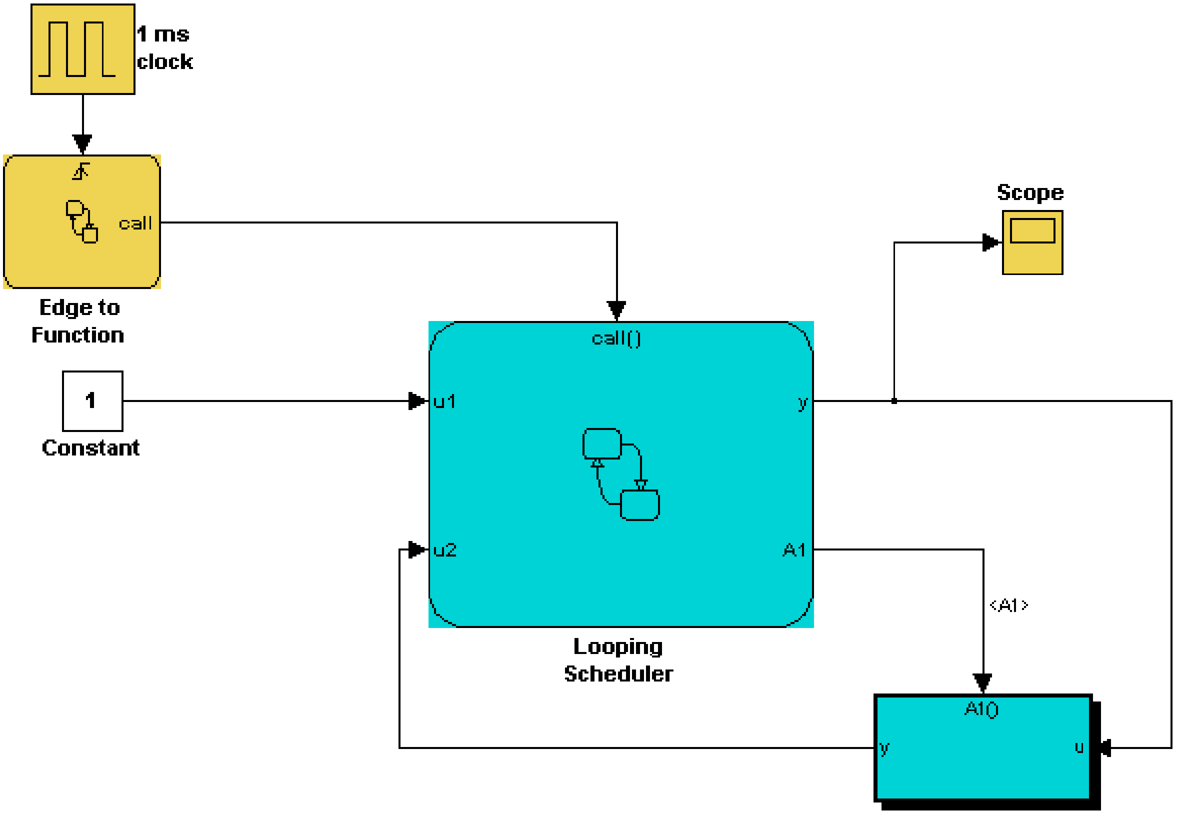
To define a function-call output event:

1. Add an event to the 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 **Scope** 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" on page 21-14.



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

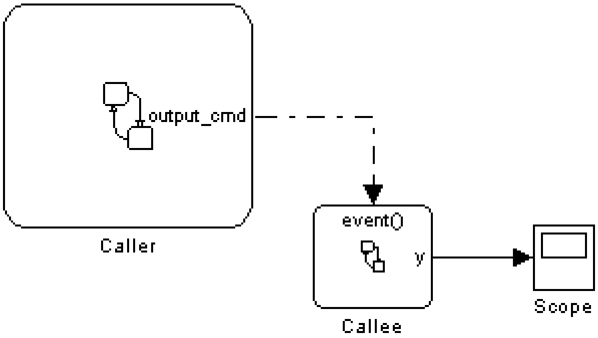
Interleaving Behavior for Broadcasting a Function-Call Output Event Multiple Times

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.)

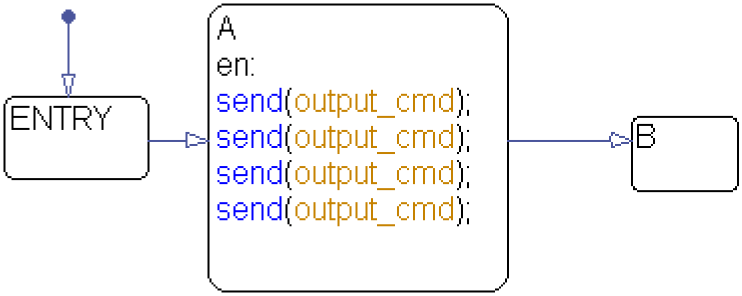
Note For information on what happens for edge-triggered output events, see "Queuing Behavior for Broadcasting an Edge-Triggered Output Event Multiple Times" on page 9-27.

Example of Interleaving Behavior for Function-Call Output Events

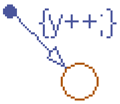
In this model, the chart Caller uses the function-call output event `output_cmd` to activate the chart Callee.



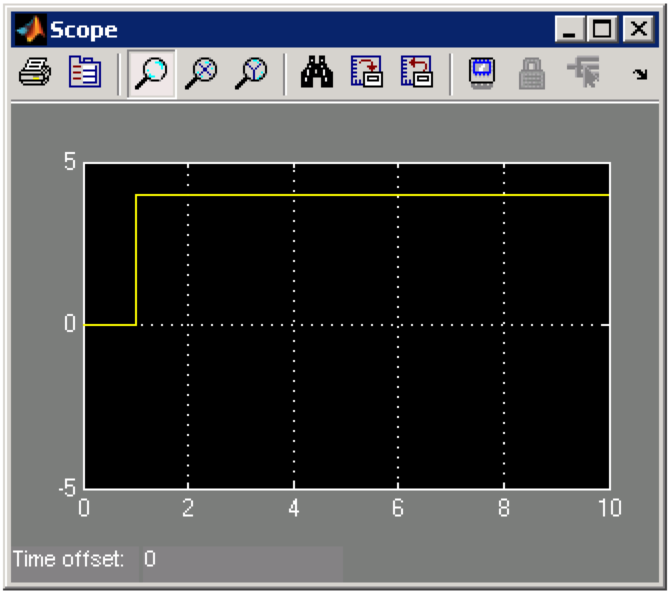
The chart Caller tries to broadcast the same function-call output event four times in a single time step, as shown.



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.

Association of 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:

1. In your chart, 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.

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.

Keywords for Implicit Events

To reference implicit events, action statements use this syntax:

```
event(object)
```

where `event` is the name of the implicit event and `object` is the state or data in which the event occurs.

Each keyword below generates implicit events in the action language notation for states and transitions.

Implicit Event	Meaning
`change(`*`data_name`*`)` or `chg(`*`data_name`*`)`	Specifies and implicitly generates a local event when Stateflow software writes a value to the variable *`data_name`*. The variable *`data_name`* cannot be machine-parented data. This implicit event works only with data that is at the chart level or lower in the hierarchy. For machine-parented data, use change detection operators to determine when the data value changes. For more information, see "Detecting Changes in Data Values" on page 10-83.
`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.

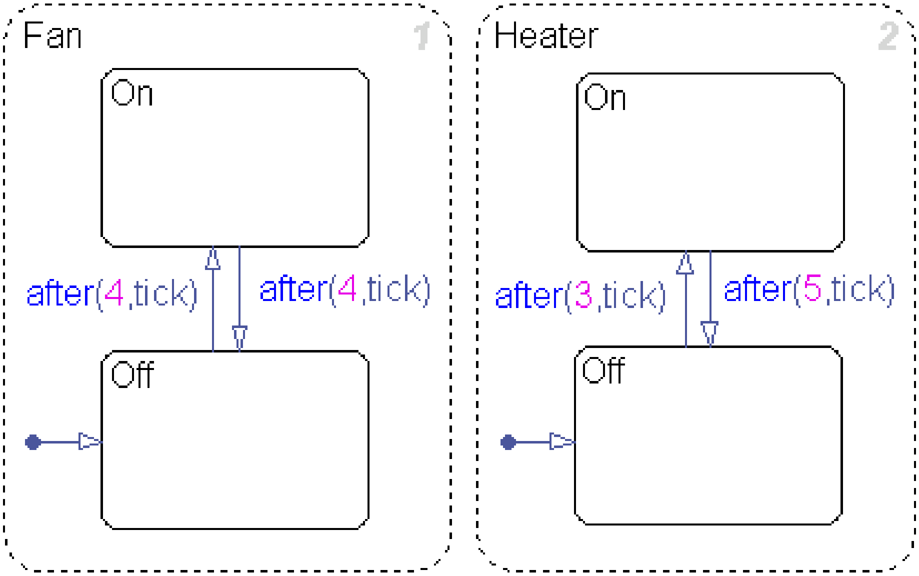
If more than one object has the same name, use the dot operator to qualify the name of the object with the name of its parent. These examples are valid references to implicit events:

```
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.

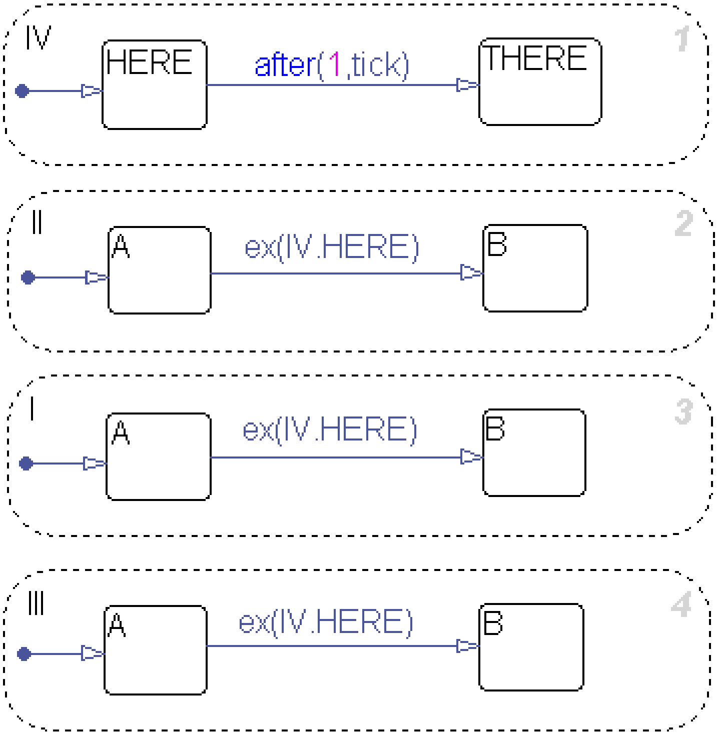
For information about the `after` operator, see "Using Temporal Logic in State Actions and Transitions" on page 10-63.

Execution Order of Transitions with Implicit Events

Suppose that:

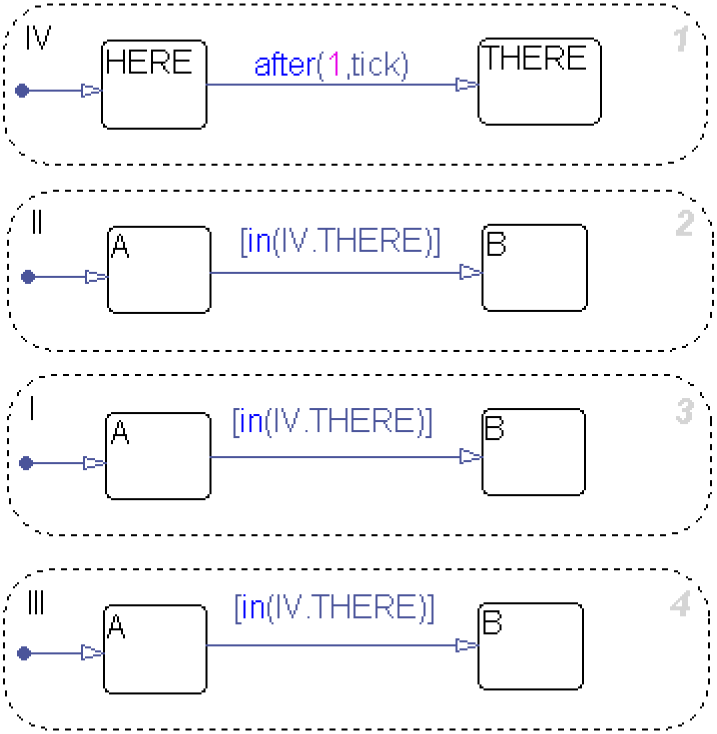
- Your chart contains parallel states.
- In multiple parallel states, the same implicit event is used to guard a transition from one substate to another.

When multiple transitions are valid in the same time step, the transitions execute based on the order in which they were created in the chart. This order does not necessarily match the activation order of the parallel states that contain the transitions. For example, consider the following chart:



When the transition from `IV.HERE` to `IV.THERE` occurs, the condition `ex(IV.HERE)` is valid for the transitions from A to B for the parallel states I, II, and III. The three transitions from A to B execute in the order in which they were created: in state I, then II, and finally III. This order does not match the activation order of those states.

To ensure that valid transitions execute in the same order that the parallel states become active, use the `in` operator instead of implicit `enter` or `exit` events:



With this modification, the transitions from A to B occur in the same order as activation of the parallel states. For more information about the `in` operator, see “Checking State Activity” on page 10-97.

Counting Events

In this section...
"When to Count Events" on page 9-45
"How to Count Events" on page 9-45
"Example of Collecting and Storing Input Data in a Vector" on page 9-45

When to Count Events

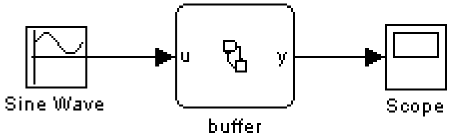
Count events when you want to keep track of explicit or implicit events in your chart.

How to Count Events

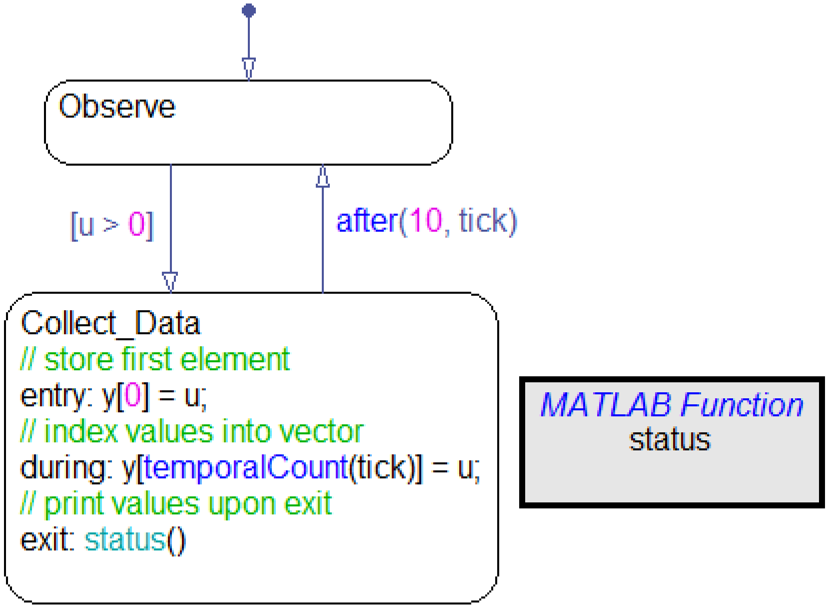
You can count occurrences of explicit and implicit events using the `temporalCount` operator. For information about the syntax of this operator, see "Operators for Event-Based Temporal Logic" on page 10-64.

Example of Collecting and Storing Input Data in a Vector

The following model collects and stores input data in a vector during chart simulation:



The chart contains two states and one MATLAB function:



Stage 1: Observation of Input Data

The chart awakens and remains in the `Observe` state, until the input data `u` is positive. Then, the transition to the state `Collect_Data` occurs.

Stage 2: Storage of Input Data

After the state `Collect_Data` becomes active, the value of the input data `u` is assigned to the first element of the vector `y`. While this state is active, each subsequent value of `u` is assigned to successive elements of `y` using the `temporalCount` operator.

Stage 3: Display of Data Stored in the Vector

After 10 ticks, the data collection process ends, and the transition to the state `Observe` occurs. Just before the state `Collect_Data` becomes inactive, a function call to `status` displays the vector data at the MATLAB prompt.

For more information about ticks in a Stateflow chart, see “Using Implicit Events” on page 9-40.

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 condition 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. See "Directed Event Broadcasting" on page 10-59 for details.

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 10-97 for details.

10

Using Actions in Stateflow Charts

Supported Action Types for States and Transitions

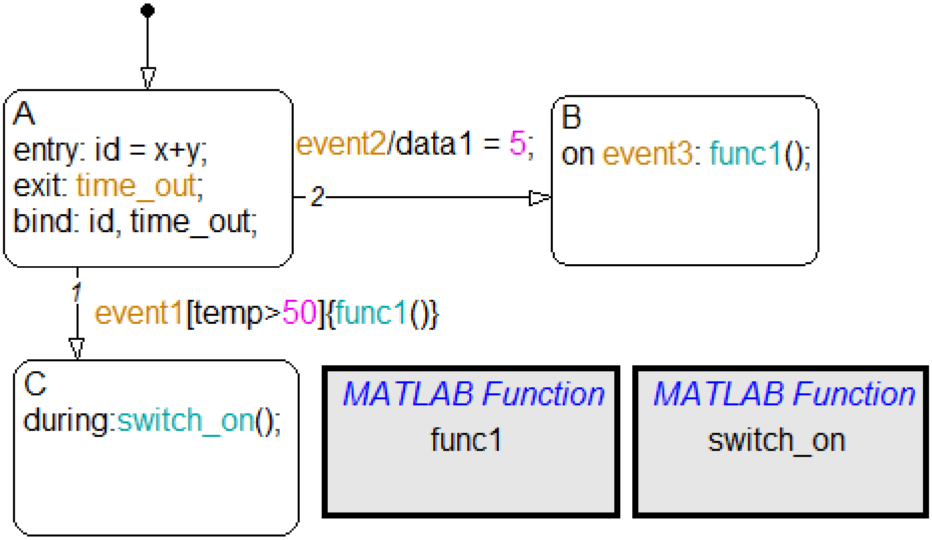
In this section...

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
```

For example, different state action types appear in the following chart.



After you enter the name in the state label, enter a carriage return and specify the actions for the state. The order you use to enter action types in the label does not matter.

Note If you do *not* specify the action type explicitly for a statement, the chart treats that statement as an entry action.

This table summarizes the different state action types.

State Action	Abbreviation	Description
`entry`	`en`	Executes when the state becomes active
`exit`	`ex`	Executes when the state is active and a transition out of the state occurs

State Action	Abbreviation	Description
`during`	`du`	Executes when the state is active and a specific event occurs
`bind`	none	Binds an event or data object so that only that state and its children can broadcast the event or change the data value
`on` *`event_name`*	none	Executes when the state is active and it receives a broadcast of *`event_name`*
`on 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 10-63.

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 10-70.

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 for a state execute 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` executes when the state A is entered by the default transition.

For a detailed description of the semantics of entering a state, see “Steps for Entering a State” on page 3-70 and “State Execution Example” on page 3-72.

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 execute when the state is active and a transition out of the state occurs.

For a detailed description of the semantics of exiting a state, see “Steps for Exiting an Active State” on page 3-72 and “State Execution Example” on page 3-72.

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 for a state execute when the state 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 “Steps for Executing an Active State” on page 3-71 and “State Execution Example” on page 3-72.

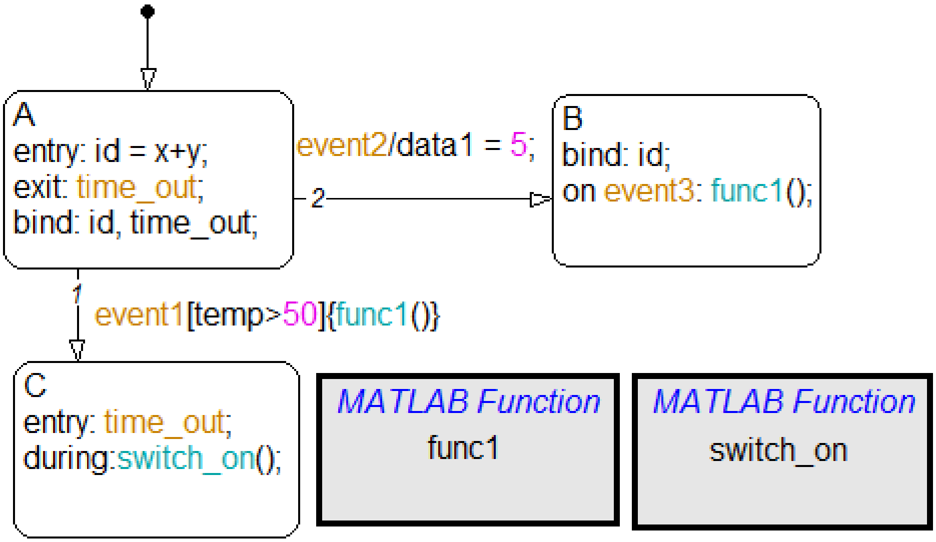
Bind Actions

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 apply 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 binding prevents any other state (or its children) in the chart from changing `id` or broadcasting event `time_out`.

If another state includes actions that change data or broadcast events that bind to another state, a parsing error occurs. The following example shows a few of these error conditions:



The following action...	Causes a parsing error because...
`bind:  id` in state `B`	Only one state can change the data `id`, which binds to state A
`entry:  time_out` in state `C`	Only one state can broadcast the event `time_out`, which binds to state A

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 more information about this behavior, see "Using Bind Actions to Control Function-Call Subsystems" on page 10-108.

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.

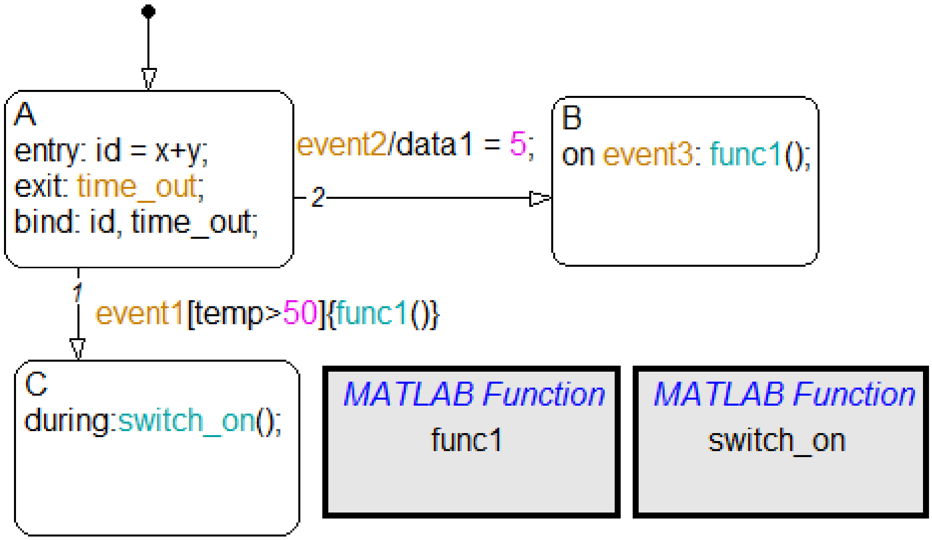
For a detailed description of the semantics of executing an active state, see "Steps for Executing an Active State" on page 3-71.

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 on its label. Transitions can have different action types, which include event triggers, conditions, condition actions, and transition actions. The action types follow the label notation with this general format:

`event_trigger[condition]{condition_action}/transition_action`

The following example shows typical transition label syntax:



Transition	Event Trigger	Condition	Condition Action	Transition Action
State A to state C	`event1`	`temp > 50`	`func1()`	None
State A to state B	`event2`	None	None	`data1 = 5`

Event 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 10-97)

    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 10-63)
- The condition expression can call a graphical function, truth table function, or MATLAB function that returns a numeric value.

  For example, `[test_function(x, y) < 0]` is a valid condition expression.

Note If the condition expression calls a function with multiple return values, only the first value applies. The other return values are not used.

- The condition expression should not call a function that causes the 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.

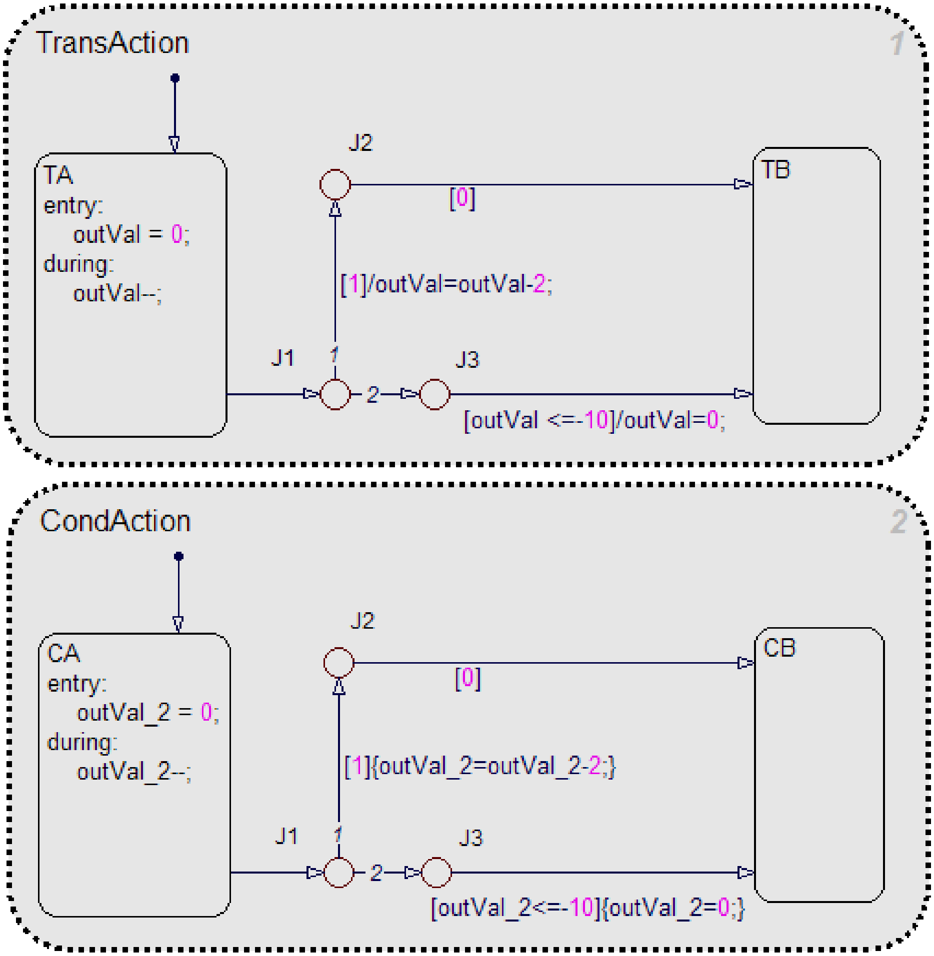
Transition Actions

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`.

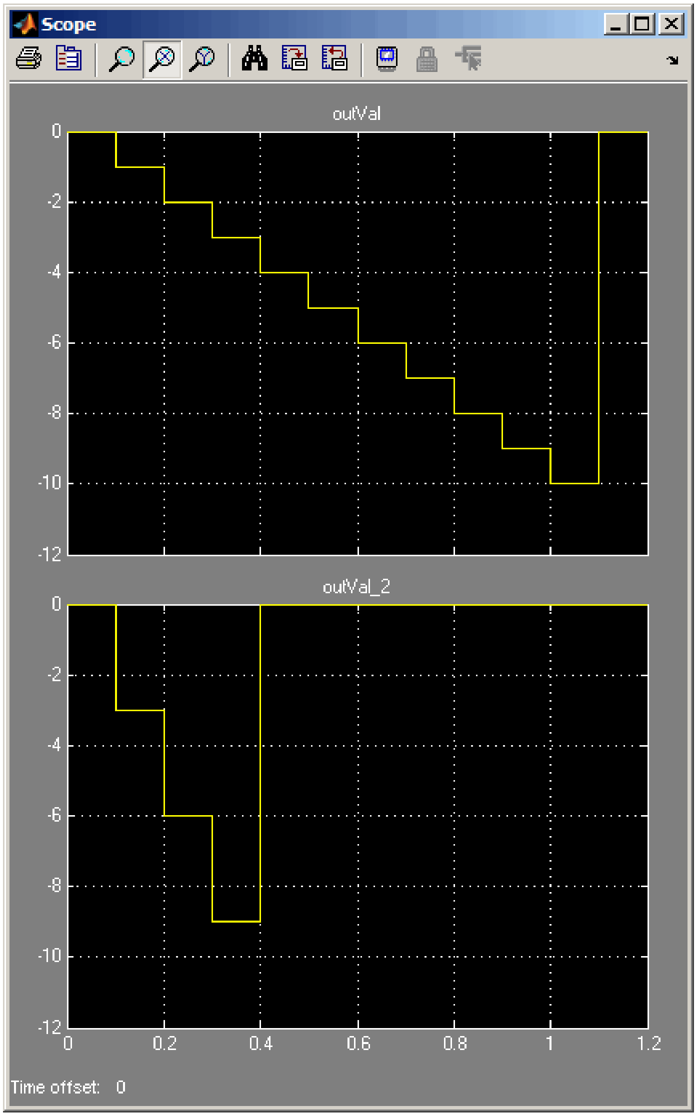
Transition actions execute only after the complete transition path is taken. They execute after the transition destination has been determined to be valid, and the condition, if specified, is true. If the transition consists of multiple segments, the transition action executes only after the entire transition path to the final destination is determined to be valid.

Execution of Actions in States and Transitions

The following chart shows how action language constructs interact during simulation:



When you simulate the model, you get the following results:



The following actions occur in the `TransAction` state:

Time	What Happens in the TransAction State
0.0	State TA becomes active.
	In TA, the `entry` action executes by setting the value of `outVal` to 0.
0.1	The transition from TA to junction J1 occurs, because the path is unconditional.
	Evaluation of the condition between J1 and J2 occurs, which returns true.
	The transition action does *not* execute, because the full transition path from J1 to TB is not complete.
	Evaluation of the condition between J2 and TB occurs, which returns false. Therefore, execution returns to J1.
	The transition from J1 to J3 occurs, because the path is unconditional.
	Evaluation of the condition between J3 and TB occurs, which returns false. Therefore, execution returns to TA.
	In TA, the `during` action executes by decrementing the value of `outVal` by 1.
0.2 – 1.0	The execution pattern from t = 0.1 repeats for each time step.
1.1	The transition from TA to junction J1 occurs, because the path is unconditional.
	Evaluation of the condition between J1 and J2 occurs, which returns true.
	The transition action does *not* execute, because the full transition path from J1 to TB is not complete.
	Evaluation of the condition between J2 and TB occurs, which returns false. Therefore, execution returns to J1.
	The transition from J1 to J3 occurs, because the path is unconditional.
	Evaluation of the condition between J3 and TB occurs, which returns true.
	State TB becomes active.
	Because the transition from J3 to TB is now complete, the transition action executes by setting the value of `outVal` to 0.

The following actions occur in the `CondAction` state:

Time	What Happens in the CondAction State
0.0	State CA becomes active.
	In CA, the `entry` action executes by setting the value of `outVal_2` to 0.
0.1	The transition from CA to junction J1 occurs, because the path is unconditional.
	Evaluation of the condition between J1 and J2 occurs, which returns true.
	The condition action executes by decrementing the value of `outVal_2` by 2.
	Evaluation of the condition between J2 and CB occurs, which returns false. Therefore, execution returns to J1.
	The transition from J1 to J3 occurs, because the path is unconditional.
	Evaluation of the condition between J3 and CB occurs, which returns false. Therefore, execution returns to CA.
	In CA, the `during` action executes by decrementing the value of `outVal_2` by 1.
0.2 – 0.3	The execution pattern from t = 0.1 repeats for each time step.
0.4	The transition from CA to junction J1 occurs, because the path is unconditional.
	Evaluation of the condition between J1 and J2 occurs, which returns true.
	The condition action executes by decrementing the value of `outVal_2` by 2.
	Evaluation of the condition between J2 and CB occurs, which returns false. Therefore, execution returns to J1.
	The transition from J1 to J3 occurs, because the path is unconditional.
	Evaluation of the condition between J3 and CB occurs, which returns true.
	The condition action executes by setting the value of `outVal_2` to 0.
	State CB becomes active.

Combining State Actions to Eliminate Redundant Code

In this section...

State Actions You Can Combine

You can combine `entry`, `during`, and `exit` actions that execute the same tasks in a state.

Why Combine State Actions

By combining state actions that execute the same tasks, you eliminate redundant code. For example:

<table>
<tr><th>Separate Actions</th><th>Equivalent Combined Actions</th></tr>
<tr><td>

```
entry:
  y = 0;
  y++;
during: y++;
```

</td><td>

```
entry: y = 0;
entry, during: y++;
```

</td></tr>
<tr><td>

```
en:
  fcn1();
  fcn2();
du: fcn1();
ex: fcn1();
```

</td><td>

```
en, du, ex: fcn1();
en: fcn2();
```

</td></tr>
</table>

Combining state actions this way produces the same chart execution behavior (semantics) and generates the same code as the equivalent separate actions.

See Also

- "How to Combine State Actions" on page 10-17
- "Order of Execution of Combined Actions" on page 10-18
- "Rules for Combining State Actions" on page 10-19

How to Combine State Actions

Combine a set of `entry`, `during`, and/or `exit` actions that perform the same task as a comma-separated list in a state. Here is the syntax:

```
entry, during, exit: task1; task2;...taskN;
```

You can also use the equivalent abbreviations:

```
en, du, ex: task1; task2;...taskN;
```

Valid Combinations

You can use any combination of the three actions. For example, the following combinations are valid:

- `en, du:`
- `en, ex:`
- `du, ex:`
- `en, du, ex:`

You can combine actions in any order in the comma-separated list. For example, `en, du:` gives the same result as `du, en:`. See "Order of Execution of Combined Actions" on page 10-18.

Invalid Combinations

You cannot combine two or more actions of the same type. For example, the following combinations are invalid:

- `en, en:`
- `ex, en, ex:`

- `du, du, ex:`

If you combine multiple actions of the same type, you receive a warning that the chart executes the action only once.

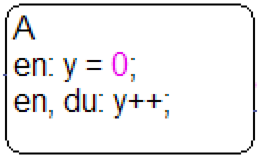
Order of Execution of Combined Actions

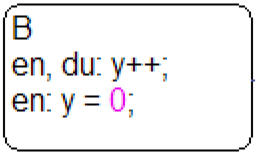
States execute combined actions in the same order as they execute separate actions:

1. `Entry` actions first, from top to bottom in the order they appear in the state
2. `During` actions second, from top to bottom
3. `Exit` actions last, from top to bottom

The order in which you combine actions does not affect state execution behavior. For example:

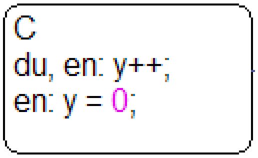
This state...	**Executes actions in this order...**

 A
en: y = 0;
en, du: y++;
	1 `en:   y = 0;`
	2 `en:   y++;`
	3 `du:   y++;`

 B
en, du: y++;
en: y = 0;
	1 `en:   y++;`
	2 `en:   y = 0;`
	3 `du:   y++;`

This state...

Executes actions in this order...

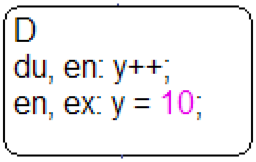


```
C
du, en: y++;
en: y = 0;
```

1 `en:   y++;`

2 `en:   y = 0;`

3 `du:   y++;`



```
D
du, en: y++;
en, ex: y = 10;
```

1 `en:   y++;`

2 `en:   y = 10;`

3 `du:   y++;`

4 `ex:   y = 10;`

Rules for Combining State Actions

- Do not combine multiple actions of the same type.
- Do not create data or events that have the same name as the action keywords: `entry`, `en`, `during`, `du`, `exit`, `ex`.

Supported Operations on Chart Data

In this section...

Binary and Bitwise Operations

The table below summarizes the interpretation of all binary operators in Stateflow action language. These operators work with the following order of precedence (0 = 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`	0	Operand `a` raised to power `b` Enabled when you clear **Enable C-bit operations** in the Chart properties dialog box. See "Specifying Chart Properties" on page 19-4.
`a * b`	1	Multiplication
`a / b`	1	Division
`a %% b`	1	Modulus
`a + b`	2	Addition

Example	Precedence	Description
`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.
`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 19-4. • 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	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 19-4.
`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 19-4. • 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 17-34 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 19-4.

Example	Description
`a |= expression`	Equivalent to `a = a | expression` (bit operation). See operation `a | b` in "Binary and Bitwise Operations" on page 10-20.
`a &= expression`	Equivalent to `a = a & expression` (bit operation). See operation `a & b` in "Binary and Bitwise Operations" on page 10-20.
`a ^= expression`	Equivalent to `a = a ^ expression` (bit operation). See operation `a ^ b` in "Binary and Bitwise Operations" on page 10-20.

Pointer and Address Operations

The address operator (&) is available for use with both Stateflow and custom code variables. The pointer operator (*) is available for use only with custom code variables.

Note The action language parser uses a relaxed set of restrictions and does not catch syntax errors until compile time.

The following examples show syntax that is valid for both Stateflow and custom code variables. The prefix `cc_` shows the places where you can use only custom code variables, and the prefix `sfcc_` shows the places where you can use either Stateflow or custom code variables.

```
cc_varPtr = &sfcc_var;
cc_ptr = &sfcc_varArray[<expression>];
cc_function(&sfcc_varA, &sfcc_varB, &sfcc_varC);
cc_function(&sfcc_sf.varArray[<expression>]);
```

The following examples show syntax that is valid only for custom code variables.

```
varStruct.field = <expression>;
(*varPtr) = <expression>;
varPtr->field = <expression>;
```

```
myVar = varPtr->field;
varPtrArray[index]->field = <expression>;
varPtrArray[expression]->field = <expression>;
myVar = varPtrArray[expression]->field;
```

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-Specific Implementations

Using the code replacement library published by Embedded Coder code generation software, you can replace a subset of arithmetic operators with target-specific implementations. Operator entries of the code replacement library can specify integral or fixed-point operand and result patterns. Operator entries can be used for the following built-in operators:

```
+
-
*
/
```

For example, you can replace an expression such as `y = u1 + u2` with a target-specific implementation, as long as `u1`, `u2`, and `y` have types that permit a match with an addition entry in the code replacement library.

Stateflow chart semantics might limit operator entry matching because the chart 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 an addition operator from the code replacement library to produce a signed 16-bit result, without a loss of precision.

To learn how to create and register code replacement tables, see “Code Replacement” in the Embedded Coder documentation. To select and view code replacement libraries, see “Selecting and Viewing Code Replacement Libraries” in the Simulink Coder documentation.

Supported Symbols in Actions

In this section...

Boolean Symbols, true and false

Use the symbols `true` and `false` to represent Boolean constants. You can use these symbols as scalars in expressions. Examples include:

```
cooling_fan = true;
heating_fan = false;
```

Tip These symbols are case-sensitive. Therefore, `TRUE` and `FALSE` are not Boolean symbols.

Do not use `true` and `false` in the following cases. Otherwise, error messages appear.

- Left-hand side of assignment statements
  - `true++;`
  - `false += 3;`
  - `[true, false] = my_function(x);`

- Argument of the `change` implicit event (see “Using Implicit Events” on page 9-40)
  - `change(true);`
  - `chg(false);`
- Indexing into a vector or matrix (see “How to Assign and Access Values of Vectors and Matrices” on page 13-8)
  - `x = true[1];`
  - `y = false[1][1];`

Note If you define `true` and `false` as Stateflow data objects, your custom definitions of `true` and `false` override the built-in Boolean constants.

Comment Symbols, %, //, /*

Use the symbols `%`, `//`, and `/*` to represent comments as shown in these examples:

```
% MATLAB comment line
// C++ comment line
/* C comment line */
```

You can also include comments in generated code for an embedded target (see “Code Generation Pane: Comments” in the *Simulink Coder Reference*) or a Stateflow custom target (see “Configuring a Custom Target” on page 25-55). Stateflow action language comments in generated code use multibyte character code. Therefore, you can have code comments with characters for non-English alphabets, such as Japanese Kanji characters.

Hexadecimal Notation Symbols, 0xFF

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`.

Note If you define `inf` as a Stateflow data object, your custom definition of `inf` overrides the built-in value.

Line Continuation Symbol, ...

Use the characters `...` at the end of a line to indicate that the expression continues on the next line. For example, you can use the line continuation symbol in a state action:

```
entry: total1 = 0, total2 = 0, ...
  total3 = 0;
```

Literal Code Symbol, $

Use `$` characters to mark actions that you want the parser to ignore but you want to appear in the generated code. For example, the parser does not process any text between the `$` characters below.

```
$
ptr -> field = 1.0;
$
```

Note Avoid frequent use of literal symbols.

MATLAB Display Symbol, ;

Omitting the semicolon after an expression displays the results of the expression in the MATLAB Command Window. If you use a semicolon, the results do not appear.

Single-Precision Floating-Point Number Symbol, F

Use a trailing `F` to specify single-precision floating-point numbers in 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, double precision applies.

Time Symbol, t

Use the letter `t` to represent absolute time that the chart inherits from a Simulink signal in simulation targets. For example, the condition `[t - On_time > Duration]` specifies that the condition is true if the difference between the simulation time `t` and `On_time` is greater than the value of `Duration`.

The letter `t` has no meaning for nonsimulation targets, since `t` depends on the specific application and target hardware.

Note If you define `t` as a Stateflow data object, your custom definition of `t` overrides the built-in value.

Calling C Functions in Actions

In this section...

Calling C Library Functions

You can call this subset of the C Math Library functions:

`abs`* **	`acos`**	`asin`**	`atan`**	`atan2`**	`ceil`**
`cos`**	`cosh`**	`exp`**	`fabs`	`floor`**	`fmod`**
`labs`	`ldexp`**	`log`**	`log10`**	`pow`**	`rand`
`sin`**	`sinh`**	`sqrt`**	`tan`**	`tanh`**	

* The Stateflow `abs` function goes beyond that of its standard C counterpart with its own built-in functionality. See "Calling the abs Function" on page 10-33.

** You can also replace calls to the C Math Library with target-specific implementations for this subset of functions. For more information, see "Replacement of C Math Library Functions with Target-Specific Implementations" on page 10-34.

When you can call these math functions, double precision applies unless the first input argument is explicitly single precision. When a type mismatch occurs, a cast of the input argument to the expected type replaces the original argument. For example, if you call the `sin` function with an integer argument, a cast of the input argument to a floating-point number of type `double` replaces the original argument.

If you call other C library functions not listed above, include the appropriate `#include...` statement in the **Simulation Target > Custom Code** pane of the Configuration Parameters dialog box. For details, see Chapter 25, “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, cast its argument or return values to integer types. See “Type Cast Operations” on page 10-25.

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 25, “Building Targets”.

Calling min and max Functions

You can call `min` and `max` by emitting the following macros automatically at the top of generated code.

```
#define min(x1,x2) ((x1) > (x2) ? (x2):(x1))
#define max(x1,x2) ((x1) > (x2) ? (x1):(x2))
```

To allow compatibility with user graphical functions named `min()` or `max()`, generated code uses a mangled name of the following form: `<prefix>_min`. However, if you export `min()` or `max()` graphical functions to other charts in your model, the name of these functions can no longer be emitted with mangled names in generated code and conflict occurs. To avoid this conflict, rename the `min()` and `max()` graphical functions.

Replacement of C Math Library Functions with Target-Specific Implementations

You can use the code replacement library published by Embedded Coder code generation software to replace the default implementations of a subset of C library functions with target-specific implementations (see "Supported Functions for Code Replacement" on page 10-34). When you specify a code replacement library, Stateflow software generates code that calls the target implementations instead of the associated C library functions. Stateflow software also uses target implementations in cases where the compiler generates calls to math functions, such as in fixed-point arithmetic utilities.

Use of Code Replacement Libraries

To learn how to create and register code replacement tables in a library, see "Map Math Functions to Target-Specific Implementations" in the Embedded Coder documentation. To select and view code replacement libraries, see "Selecting and Viewing Code Replacement Libraries" in the Simulink Coder documentation.

Supported Functions for Code Replacement

You can replace the following math functions with target-specific implementations:

Function	Data Type Support
`abs` **Note** See also “Replacement of Calls to abs” on page 10-36.	Floating-point and integer
`acos`	Floating-point
`asin`	Floating-point
`atan`	Floating-point
`atan2`	Floating-point
`ceil`	Floating-point
`cos`	Floating-point
`cosh`	Floating-point
`exp`	Floating-point
`floor`	Floating-point
`fmod`	Floating-point
`ldexp`	Floating-point
`log`	Floating-point
`log10`	Floating-point
`max`	Floating-point and integer
`min`	Floating-point and integer
`pow`	Floating-point
`sin`	Floating-point
`sinh`	Floating-point
`sqrt`	Floating-point
`tan`	Floating-point
`tanh`	Floating-point

Replacement of Calls to abs

Replacement of calls to `abs` can occur as follows:

Type of Argument for `abs`	Result
Floating-point	Replacement with target-specific implementation
Integer	Replacement with target-specific implementation
Fixed-point with zero bias	Replacement with ANSI C function
Fixed-point with nonzero bias	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" on page 10-36
- "Specifying Custom C Functions for Code Generation" on page 10-37
- "Guidelines for Calling Custom C Functions in Stateflow Action Language" on page 10-37
- "Guidelines for Writing Custom C Functions That Access Input Vectors" on page 10-37
- "Example of Function Call in Transition Action" on page 10-38
- "Example of Function Call in State Action" on page 10-40
- "Passing Arguments by Reference" on page 10-41

Specifying Custom C Functions for Simulation

To specify custom C functions for simulation:

1. Open the Configuration Parameters dialog box.
2. Select **Simulation Target > Custom Code**.

3. Specify your custom C files, as described in "Integrating Custom C Code for Nonlibrary Charts for Simulation" on page 25-8.

Specifying Custom C Functions for Code Generation

To specify custom C functions for code generation:

1. Open the Configuration Parameters dialog box.
2. Select **Code Generation > Custom Code**.
3. Specify your custom C files, as described in "Integrating Custom C Code for Nonlibrary Charts for Code Generation" on page 25-23.

Guidelines for Calling Custom C Functions in 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).

Guidelines for Writing Custom C Functions That Access Input Vectors

- Use the `sizeof` function to determine the length of an input vector.

  For example, your custom function can include a for-loop that uses `sizeof` as follows:

  ```
  for(i=0; i < sizeof(input); i++) {
  ......
  }
  ```

- If your custom function uses the value of the input vector length multiple times, include an input to your function that specifies the input vector length.

  For example, you can use `input_length` as the second input to a `sum` function as follows:

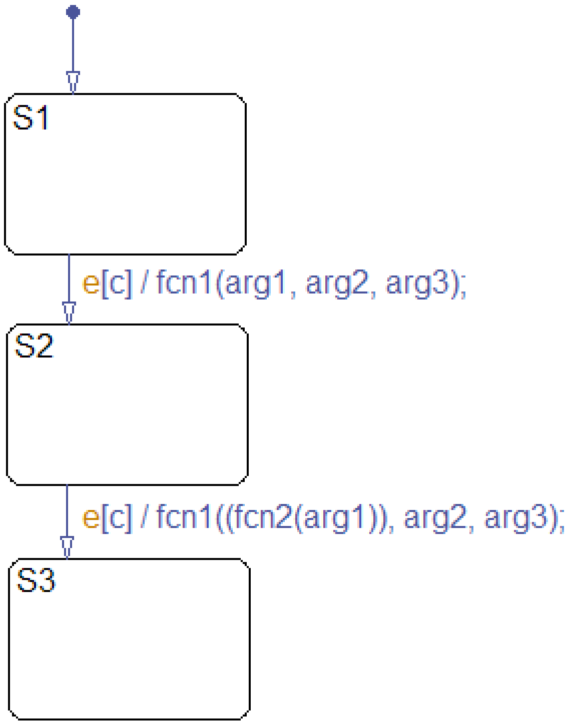
  ```
  int sum(double *input, double input_length)
  ```

  Your `sum` function can include a for-loop that iterates over all elements of the input vector:

  ```
  for(i=0; i < input_length; i++) {
  ......
  }
  ```

Example of Function Call in Transition Action

Example formats of function calls using transition action notation appear in the following chart.



A function call to `fcn1` occurs with `arg1`, `arg2`, and `arg3` if the following are true:

- `S1` is active.
- Event `e` occurs.
- Condition `c` is true.
- The transition destination `S2` is valid.

The transition action in the transition from `S2` to `S3` shows a function call nested within another function call.

Example of Function Call in State Action

Example formats of function calls using state action notation appear in the following chart.

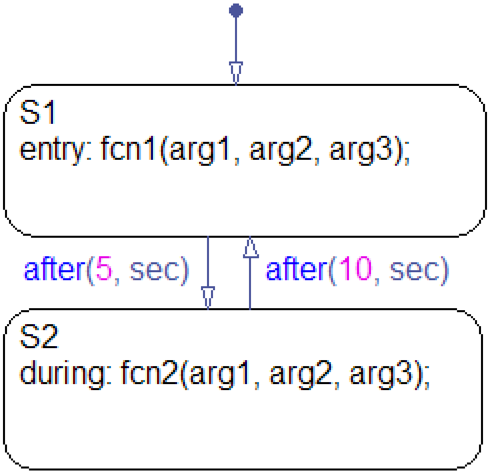


Chart execution occurs as follows:

1. When the default transition into `S1` occurs, `S1` becomes active.
2. The `entry` action, a function call to `fcn1` with the specified arguments, executes.
3. After 5 seconds of simulation time, `S1` becomes inactive and `S2` becomes active.
4. The `during` action, a function call to `fcn2` with the specified arguments, executes.
5. After 10 seconds of simulation time, `S2` becomes inactive and `S1` becomes active again.
6. Steps 2 through 5 repeat until the simulation ends.

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 19-4).
- If the data type of `x` is `boolean`, you must turn off the coder option **Use bitsets for storing state configuration** (see "How to Optimize Generated Code for Embeddable Targets" on page 25-29).
- If `x` is an array with its first index property set to 0 (see "Setting Data Properties in the Data Dialog Box" on page 8-5), then you must call the function as follows.

  ```
  f(&(x[0]));
  ```

  This passes a pointer to the first element of `x` to the function.
- If `x` is an array with its first index property set to a nonzero number (for example, 1), the function must be called in the following way:

  ```
  f(&(x[1]));
  ```

  This passes a pointer to the first element of `x` to the function.

Calling Built-In MATLAB Functions and Accessing Workspace Data

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. For example, the statement `a = ml.x` returns the value of the MATLAB workspace variable `x` to the Stateflow data `a`.

For functions, the syntax is as follows:

```
[return_val1, return_val2,...] = ml.matfunc(arg1, arg2,...)
```

For example, the statement `[a, b, c] = ml.`*`matfunc`*`(x, y)` passes the return values from the MATLAB function *`matfunc`* to the Stateflow data `a`, `b`, and `c`.

If the MATLAB function you call does not require arguments, you must still include the parentheses. If you omit the parentheses, Stateflow software interprets the function name as a workspace variable, which, when not found, generates a run-time error during simulation.

Examples

In these examples, `x`, `y`, and `z` are workspace variables and `d1` and `d2` are Stateflow data:

- `a = ml.sin(ml.x)`

  In this example, the MATLAB function `sin` evaluates the sine of `x`, which is then assigned to Stateflow data variable `a`. However, because `x` is a workspace variable, you must use the namespace operator to access it. Hence, `ml.x` is used instead of just `x`.

- `a = ml.sin(d1)`

  In this example, the MATLAB function `sin` evaluates the sine of `d1`, which is assigned to Stateflow data variable `a`. Because `d1` is Stateflow data, you can access it directly.

- `ml.x = d1*d2/ml.y`

  The result of the expression is assigned to `x`. If `x` does not exist prior to simulation, it is automatically created in the MATLAB workspace.

- `ml.v[5][6][7] = ml.`*`matfunc`*`(ml.x[1][3], ml.y[3], d1, d2, '`*`string`*`')`

  The workspace variables `x` and `y` are arrays. `x[1][3]` is the `(1,3)` element of the two-dimensional array variable `x`. `y[3]` is the third element of the one-dimensional array variable `y`. The last argument, `'`*`string`*`'`, is a 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. 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 "How Charts Infer the Return Size for ml Expressions" on page 10-50.

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 enclosed with two single pairs of quotation marks 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 appears in the *`evalString`* expression as `X`. The squaring of each operand is performed with the MATLAB `power` function, which takes two arguments: the value to square, and the power value, 2.

Expressions using the `ml` namespace operator and the `ml` function can be used as arguments for `ml` namespace operator and `ml` function expressions. The following example nests `ml` expressions at three different levels:

```
a = ml.power(ml.sin(ml.X + ml('cos(Y)')),2)
```

In composing your `ml` expressions, follow the levels of precedence set out in "Binary and Bitwise Operations" on page 10-20. Use parentheses around power expressions with the ^ operator when you use them in conjunction with other arithmetic operators.

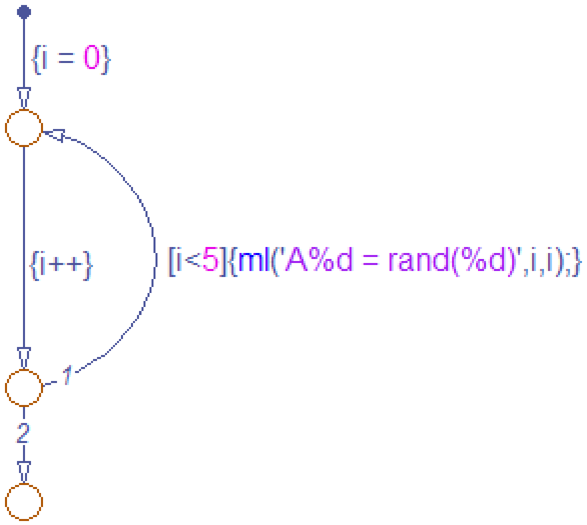
Stateflow software checks expressions for data size mismatches in your actions during parsing of charts and during run time. Because the return values for `ml` expressions are not known until run time, Stateflow software must infer the size of their return values. See "How Charts Infer the Return Size for ml Expressions" on page 10-50.

Which ml Should I Use?

In most cases, the notation of the `ml` namespace operator is more straightforward. However, using the `ml` function call does offer a few advantages:

- Use the `ml` function to dynamically construct workspace variables.

  The following flow graph creates four new MATLAB matrices:

  The `for` loop creates four new matrix variables in the MATLAB workspace. The default transition initializes the Stateflow counter `i` to 0, while the transition segment between the top two junctions increments it by 1. If `i` is less than 5, the transition segment back to the top junction evaluates the `ml` function call `ml('A%d = rand(%d)',i,i)` for the current value of `i`. When `i` is greater than or equal to 5, the transition segment between the bottom two junctions occurs and execution stops.

The transition executes the following MATLAB commands, which create a workspace scalar (`A1`) and three matrices (`A2`, `A3`, `A4`):

```
A1 = rand(1)
A2 = rand(2)
A3 = rand(3)
A4 = rand(4)
```

- Use the `ml` function with full MATLAB notation.

  You cannot use full MATLAB notation with the `ml` namespace operator, as 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 8-32).
- Any numerical scalar or array of `ml` data in the Stateflow hierarchy can participate in any kind of unary operation and any kind of binary operation with any other data in the hierarchy.

  If `ml` data participates in any numerical operation with other data, the size of the `ml` data must be inferred from the context in which it is used, just as return data from the `ml` namespace operator and `ml` function are. See "How Charts Infer the Return Size for ml Expressions" on page 10-50.

  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 25, "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 statements, `mldata1` and `mldata2` are Stateflow data of type `ml`:

```
mldata1 = ml.rand(3);
mldata2 = ml.transpose(mldata1);
```

In the first line of this example, `mldata1` receives the return value of the MATLAB function `rand`, which, in this case, returns a 3-by-3 array of random numbers. Note that `mldata1` is not specified as an array or sized in any way. It can receive any MATLAB workspace data or the return of any MATLAB function because it is defined as a Stateflow data of type `ml`.

In the second line of the example, `mldata2`, also of Stateflow data type `ml`, receives the transpose matrix of the matrix in `mldata1`. It is assigned the return value of the MATLAB function `transpose` in which `mldata1` is the argument.

Note the differences in notation if the preceding example were to use MATLAB workspace data (`wsdata1` and `wsdata2`) instead of Stateflow `ml` data to hold the generated matrices:

```
ml.wsdata1 = ml.rand(3);
ml.wsdata2 = ml.transpose(ml.wsdata1);
```

In this case, each workspace data must be accessed through the `ml` namespace operator.

How Charts Infer the Return Size for ml Expressions

Stateflow expressions using the `ml` namespace operator and the `ml` function evaluate in the MATLAB workspace at run time. The actual size of the data returned from the following expression types is known only at run time:

- MATLAB workspace data or functions using the `ml` namespace operator or the `ml` function call

  For example, the size of the return values from the expressions `ml.`*`var`*, `ml.`*`func`*`()`, or `ml(`*`evalString, arg1, arg2,...`*`)`, where *`var`* is a MATLAB workspace variable and *`func`* is a MATLAB function, cannot be known until run-time.
- Stateflow data of type `ml`
- Graphical functions that return Stateflow data of type `ml`

When these expressions appear in actions, Stateflow code generation creates temporary data to hold intermediate returns for evaluation of the full expression of which they are a part. Because the size of these return values is unknown until run time, Stateflow software must employ context rules to infer the sizes for creation of the temporary data.

During run time, if the actual returned value from one of these commands differs from the inferred size of the temporary variable that stores it, a size mismatch error appears. To prevent run-time errors, use the following guidelines to write actions with MATLAB commands or `ml` data:

Guideline		**Example**
Return sizes of MATLAB commands or data in an expression must match return sizes of peer expressions.		In the expression `ml.func() * (x + ml.y)`, if `x` is a 3-by-2 matrix, then `ml.func()` and `ml.y` are also assumed to evaluate to 3-by-2 matrices. If either returns a value of different size (other than a scalar), an error results during run-time.
Expressions that return a scalar never produce an error. You can combine matrices and scalars in larger expressions because MATLAB commands use scalar expansion.		In the expression `ml.x + y`, if `y` is a 3-by-2 matrix and `ml.x` returns a scalar, the resulting value is the result of adding the scalar value of `ml.x` to every member of `y` to produce a matrix with the size of `y`, that is, a 3-by-2 matrix. The same rule applies to subtraction (`-`), multiplication (`*`), division (`/`), and any other binary operations.
MATLAB commands or Stateflow data of type `ml` can be members of these independent levels of expression, for which resolution of return size is necessary:	Arguments The expression for each function argument is a larger expression for which the return size of MATLAB commands or Stateflow data of type `ml` must be determined.	In the expression `z + func(x + ml.y)`, the size of `ml.y` is independent of the size of `z`, because `ml.y` is used at the function argument level. However, the return size for `func(x + ml.y)` must match the size of `z`, because they are both at the same expression level.
	Array indices The expression for an array index is an independent level of expression that must be scalar in size.	In the expression `x + array[y]`, the size of `y` is independent of the size of `x` because `y` and `x` are at different levels of expression. Also, `y` must be a scalar.

Guideline	Example
The return size for an indexed array element access must be a scalar.	The expression `x[1][1]`, where `x` is a 3-by-2 array, must evaluate to a scalar.
MATLAB command or data elements used in an expression for the input argument of a MATLAB function called through the `ml` namespace operator are resolved for size. This resolution uses the rule for peer expressions (preceding rule 1) for the expression itself, because no size definition prototype is available.	In the function call `ml.`*`func`*`(x + ml.y)`, if `x` is a 3-by-2 array, `ml.y` must return a 3-by-2 array or a scalar.
MATLAB command or data elements used for the input argument for a graphical function in an expression are resolved for size by the function prototype.	If the graphical function *`gfunc`* has the prototype *`gfunc`*`(`*`arg1`*`)`, where *`arg1`* is a 2-by-3 Stateflow data array, the calling expression, *`gfunc`*`(ml.y + x)`, requires that both `ml.y` and `x` evaluate to 2-by-3 arrays (or scalars) during run-time.
`ml` function calls can take only scalar or string literal arguments. Any MATLAB command or data that specifies an argument for the `ml` function must return a scalar value.	In the expression `a = ml('sin(x)')`, the `ml` function calls the MATLAB function `sin` to evaluate the sine of `x` in the MATLAB workspace. Stateflow data variable `a` stores that result.
In an assignment, the size of the right-hand expression must match the size of the left-hand expression, with one exception. If the left-hand expression is a single MATLAB variable, such as `ml.x`, or Stateflow data of type `ml`, the right-hand expression determines the sizes of both expressions.	In the expression `s = ml.`*`func`*`(x)`, where `x` is a 3-by-2 matrix and `s` is scalar Stateflow data, `ml.`*`func`*`(x)` must return a scalar to match the left-hand expression, `s`. However, in the expression `ml.y = x + s`, where `x` is a 3-by-2 data array and `s` is scalar, the left-hand expression, workspace variable `y`, is assigned the size of a 3-by-2 array to match the size of the right-hand expression, `x+s`, a 3-by-2 array.

Guideline	Example
In an assignment, Stateflow column vectors on the left-hand side are compatible with MATLAB row or column vectors of the same size on the right-hand side. A matrix you define with a row dimension of 1 is considered a row vector. A matrix you define with one dimension or with a column dimension of 1 is considered a column vector.	In the expression `s = ml.func()`, where `ml.func()` returns a 1-by-3 matrix, if `s` is a vector of size `3`, the assignment is valid.
If you cannot resolve the return size of MATLAB command or data elements in a larger expression by any of the preceding rules, they are assumed to return scalar values.	In the expression `ml.x = ml.y + ml.z`, none of the preceding rules can be used to infer a common size among `ml.x`, `ml.y`, and `ml.z`. In this case, both `ml.y` and `ml.z` are assumed to return scalar values. Even if `ml.y` and `ml.z` return matching sizes at run-time, if they return nonscalar values, a size mismatch error results.
The preceding rules for resolving the size of member MATLAB commands or Stateflow data of type `ml` in a larger expression apply only to cases in which numeric values are expected for that member. For nonnumeric returns, a run-time error results. **Note** Member MATLAB commands or data of type `ml` in a larger expression are limited to numeric values (scalar or array) only if they participate in numeric expressions.	The expression `x + ml.str`, where `ml.str` is a string workspace variable, produces a run-time error stating that `ml.str` is not a numeric type.

Special cases exist, in which no size checking occurs to resolve the size of MATLAB command or data expressions that are part of larger expressions. Use of the following expressions does not require enforcement of size checking at run-time:

- `ml.var`
- `ml.func()`

- `ml(`*`evalString`*`, `*`arg1`*`, `*`arg2`*`,...)`
- Stateflow data of type `ml`
- Graphical function returning a Stateflow data of type `ml`

In these cases, assignment of a return to the left-hand side of an assignment statement or a function argument occurs without checking for a size mismatch between the two:

- An assignment in which the left-hand side is a MATLAB workspace variable

  For example, in the expression `ml.x = ml.y`, `ml.y` is a MATLAB workspace variable of any size and type (structure, cell array, 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`. Although size checking does not occur for the input type, if the passed-in data is not of the expected type, an error results from the function call `ml.`*`func`*`()`.

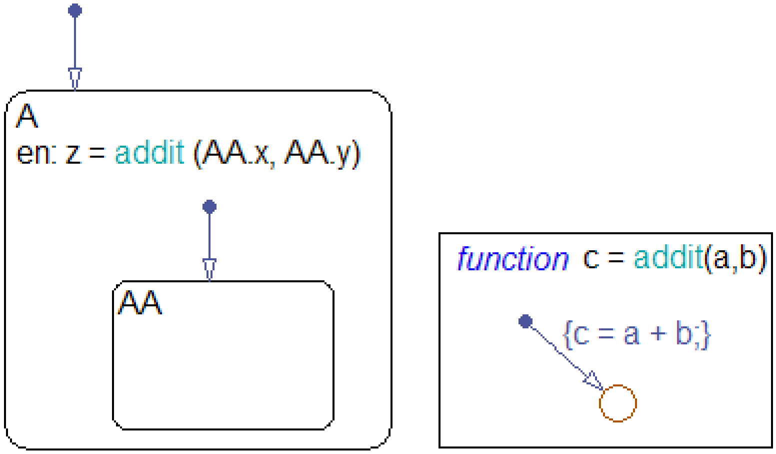
- Arguments for a graphical function that are specified as Stateflow data of type `ml` in its prototype statement

Note If you replace the inputs in the preceding cases with non-MATLAB numeric Stateflow data, conversion to an `ml` type occurs.

Using Data and Event Arguments in Actions

When you use data and event objects as arguments to functions that you call in actions, the chart assumes that these arguments appear at the same level in the hierarchy. If the data and event arguments are not at that level, object name resolution occurs by searching up the hierarchy. Data or event object arguments 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 call to function `addit` from state `A` can resolve `z` because that data object belongs to state `A`. However, the function call cannot resolve `x` and `y` by looking above state `A` in the chart hierarchy. Therefore, the function call must reference `x` and `y` explicitly to the parent state `AA`.

For information about functions you can call in actions that use data as arguments, see these sections:

- "Graphical Functions for Reusing Logic Patterns and Iterative Loops" on page 7-30
- "Calling C Functions in Actions" on page 10-32

- “Calling Built-In MATLAB Functions and Accessing Workspace Data” on page 10-42

Only operators for event-based temporal logic take events as arguments. See “Using Temporal Logic in State Actions and Transitions” on page 10-63.

Using Arrays in Actions

In this section...
"Array Notation" on page 10-57
"Arrays and Custom Code" on page 10-58

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. 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 13, “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 to Synchronize States

In this section...

Directed Event Broadcasting

You can broadcast events directly from one state to another to synchronize parallel (AND) states in the same chart. The following rules apply:

- The receiving state must be active during the event broadcast.
- An action in one chart cannot broadcast events to states in another chart.

Using a directed event broadcast provides the following benefits over an undirected broadcast:

- Prevents unwanted recursion during simulation.
- Improves the efficiency of generated code.

For information about avoiding unwanted recursion, see "Guidelines for Avoiding Unwanted Recursion in a Chart" on page 26-36.

Example of Directed Event Broadcasting Using send

The format of the directed event broadcast with `send` is:

```
send(event_name,state_name)
```

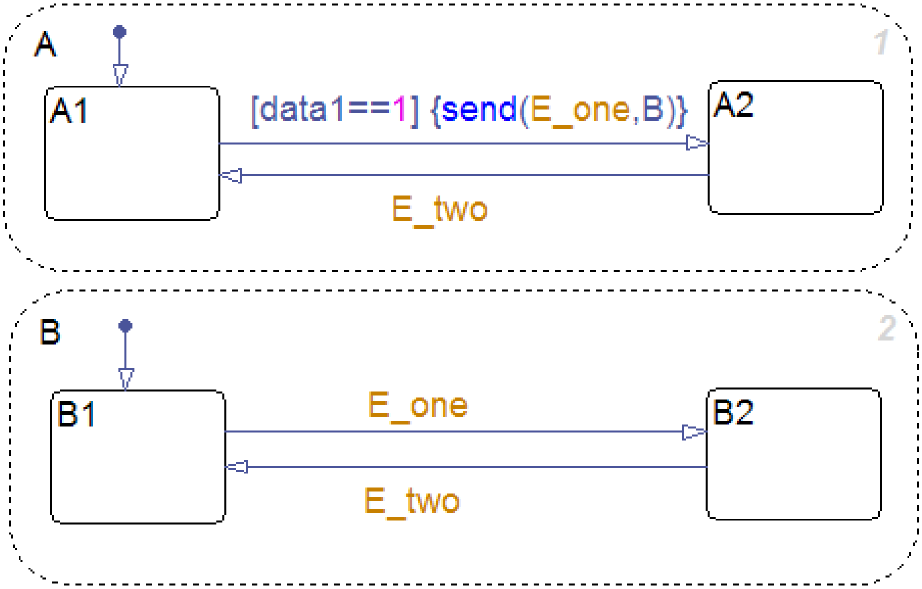
where *`event_name`* is broadcast to *`state_name`* and any offspring of that state in the hierarchy. The event you send must be visible to both the sending state and the receiving state (*`state_name`*).

The *`state_name`* argument can include a full hierarchy path to the state. For example, if the state `A` contains the state `A1`, send an event `e` to state `A1` with the following broadcast:

```
send(e, A.A1)
```

Tip Do not use the chart name in the full hierarchy path to a state. Formal chart names include the subsystem in which a chart resides. For example, in the model `sldemo_fuelsys`, the chart `control_logic` is in the subsystem `fuel_rate_control`. The formal name for the chart is `fuel_rate_control/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`*`)` syntax.



In this example, event `E_one` belongs to the chart and is visible to both `A` and `B`. See "Directed Event Broadcast Using Send Example" on page B-60 for information on the semantics of this notation.

Example of Directed Event Broadcasting Using Qualified Event Names

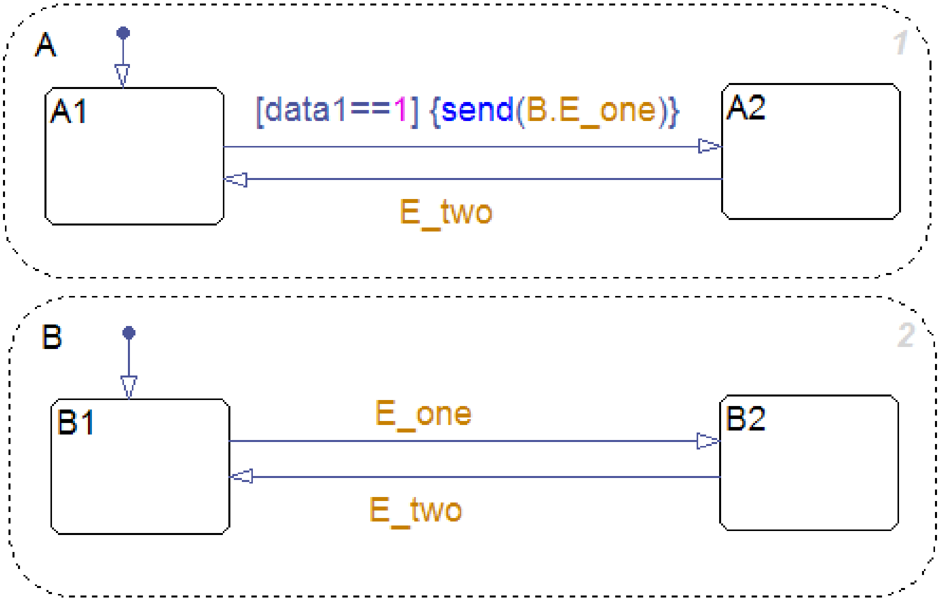
The format of the direct event broadcast using qualified event names is:

```
state_name.event_name
```

where *`event_name`* is broadcast to its owning state (*`state_name`*) and any offspring of that state in the hierarchy. The event you send is visible only to the receiving state (*`state_name`*).

The *`state_name`* argument can also include a full hierarchy path to the receiving state. Do not use the chart name in the full path name of the state.

The following example illustrates the use of a qualified event name in a directed event broadcast.



In this example, event `E_one` belongs to state `B` and is visible only to that state. See "Directed Event Broadcast Using Qualified Event Name Example" on page B-62 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
  - Transitions that originate from states
  - Transition segments that originate from junctions when the full transition path connects two states

---

Note This restriction means that you cannot use temporal logic operators in default transitions or flow 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 10-68) 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.

Operator	Syntax	Description
		Resets the counter for `E` to 0 each time the associated state reactivates.
`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.
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.
`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 with Temporal Logic

Use the keyword `sec` to define simulation time that has elapsed since activation of a state. This keyword is valid only in state actions and in transitions that originate from states.

Example of Defining Time Delays

The following continuous-time chart defines two absolute time delays in transitions. (See Chapter 16, “Modeling Continuous-Time Systems in Stateflow Charts” for information about modeling continuous-time systems.)

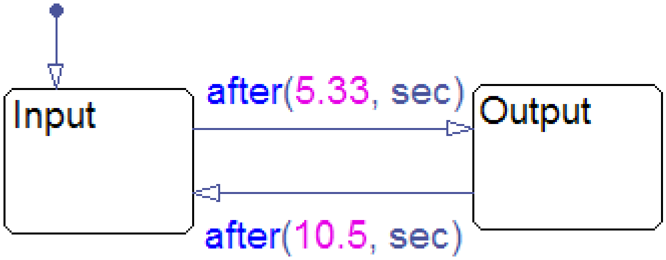
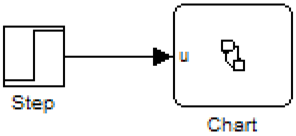


Chart execution occurs as follows:

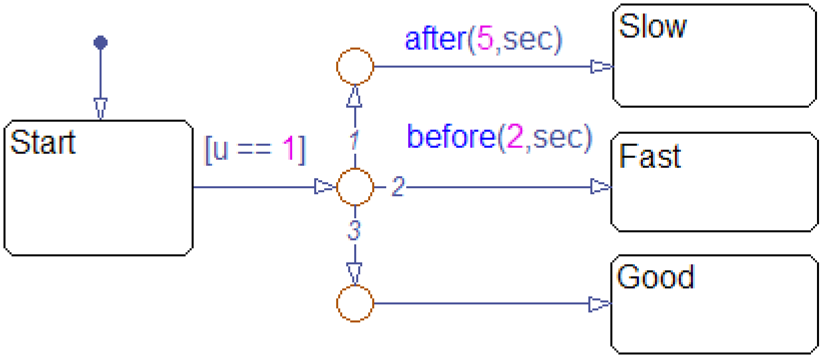
1. When the chart awakens, the state `Input` activates first.
2. After 5.33 seconds of simulation time pass, the transition from `Input` to `Output` occurs.
3. The state `Input` deactivates, and the state `Output` activates.
4. After another 10.5 seconds of simulation time pass, the transition from `Output` to `Input` occurs.
5. The state `Output` deactivates, and the state `Input` activates.
6. Steps 2 through 5 repeat, until the simulation ends.

Example of Detecting Elapsed Time

In the following model, the Step block provides a unit step input to the chart:



The chart determines when the input `u` equals 1.



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

Advantages of Using Absolute-Time Temporal Logic for Delays

Use absolute-time temporal logic instead of the implicit `tick` event for these reasons:

- Delay expressions that use absolute-time temporal logic are independent of the sample time of the model. However, the `tick` event is dependent on sample time.
- Absolute-time temporal logic works in charts that have function-call input events. However, the `tick` event does not work in charts with function-call inputs.

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.
`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 Uses Absolute-Time Temporal Logic

The `sf_boiler` model shows the use of absolute-time temporal logic to implement a bang-bang controller. To run the model:

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

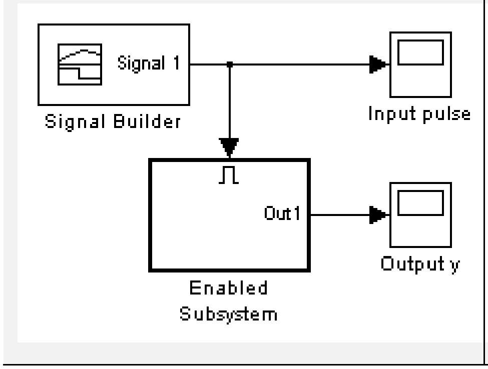
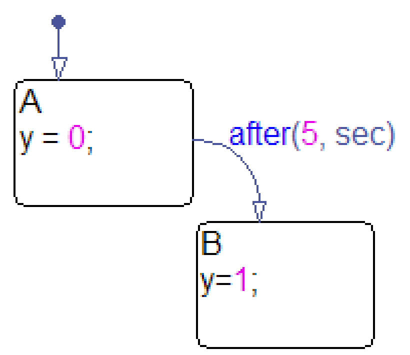
2. Start simulation of the model.

Behavior of Absolute-Time Temporal Logic in Conditionally Executed Subsystems

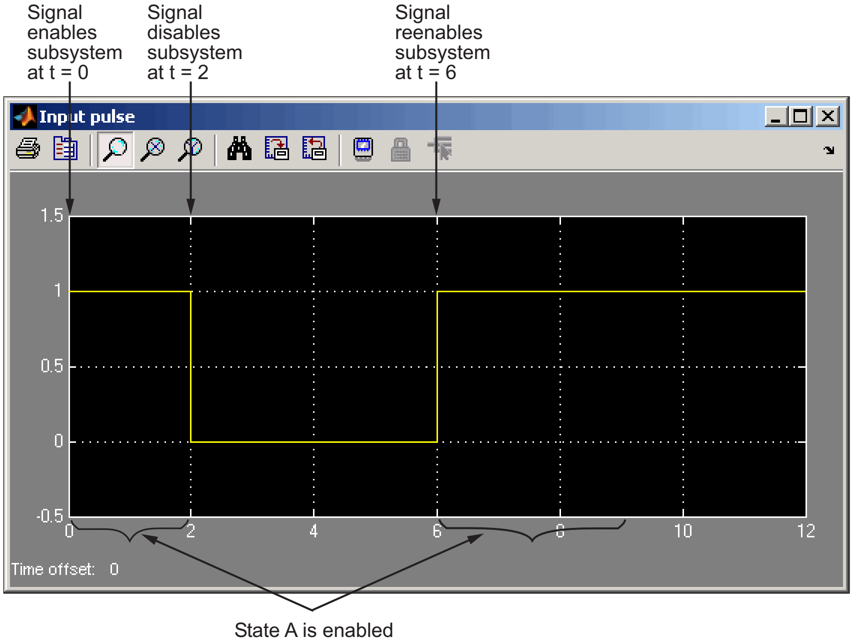
You can use absolute-time temporal logic in a chart that resides in a *conditionally executed* subsystem. (See “Creating Conditional Subsystems” in the Simulink documentation for details.) When the subsystem is disabled, the chart becomes inactive and the temporal logic operator pauses while the chart 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 Temporal Logic in an Enabled Subsystem

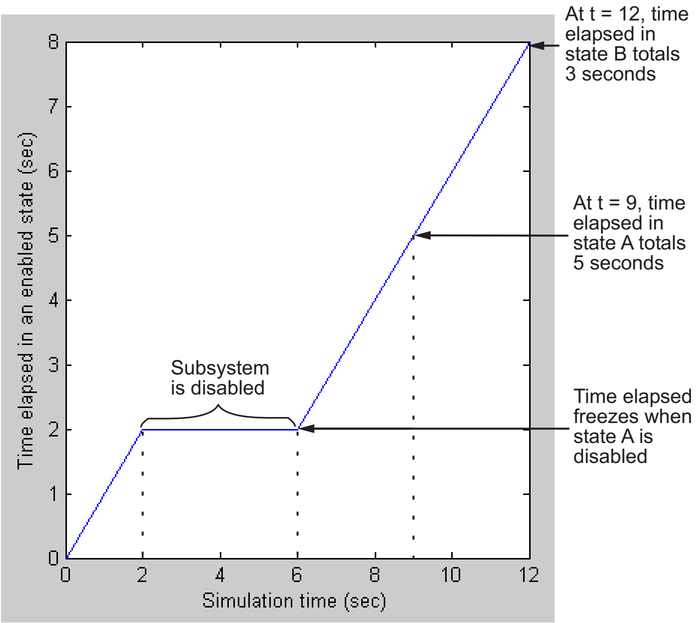
Suppose that your model has an enabled subsystem that contains a chart with the `after` operator. In the subsystem, the **States when enabling** parameter is set to `held`.

Model with Enabled Subsystem	Chart in Enabled Subsystem
	

The Signal Builder block provides the following 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

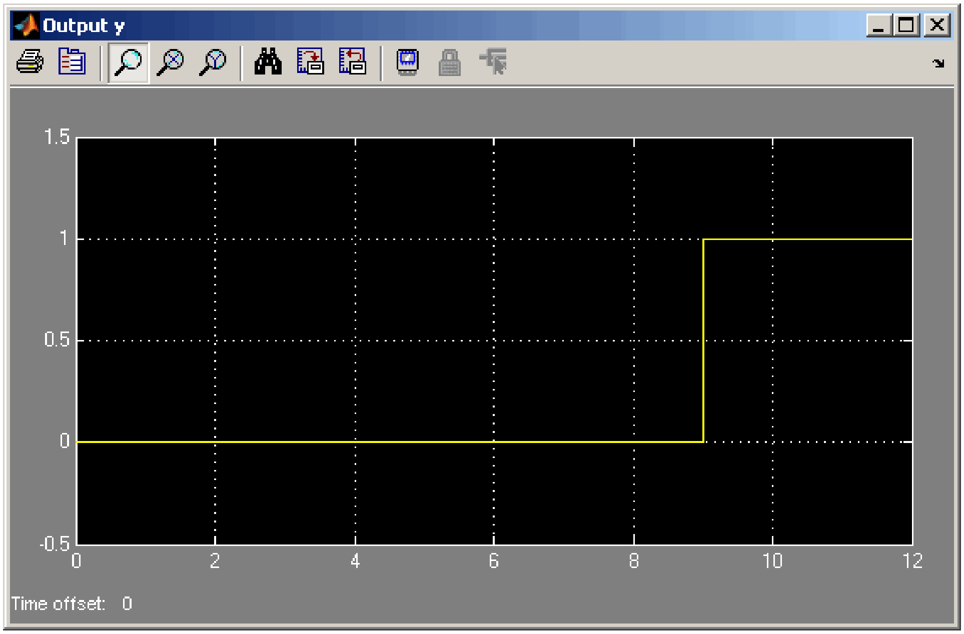
The total time elapsed in an enabled state (both A and B) is as follows.



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. 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.

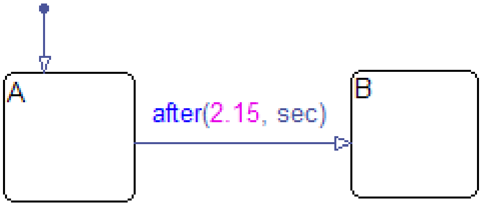
Note These semantics also apply to the `before` operator.

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.

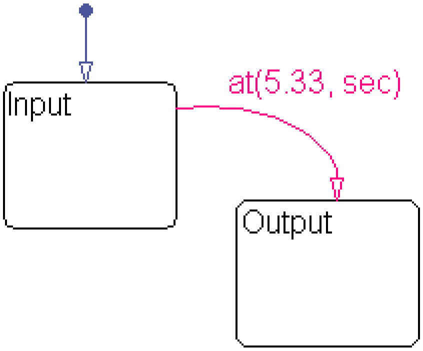
Note This behavior also applies to the `before` operator.

Best Practices for Using Absolute-Time Temporal Logic

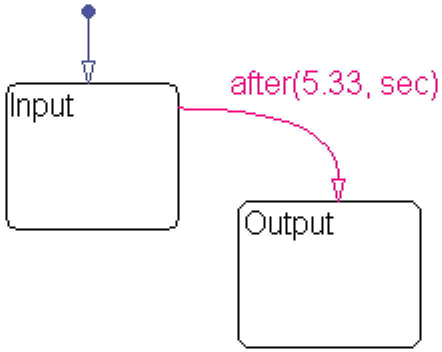
Use the after Operator to Replace the at Operator

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)`.



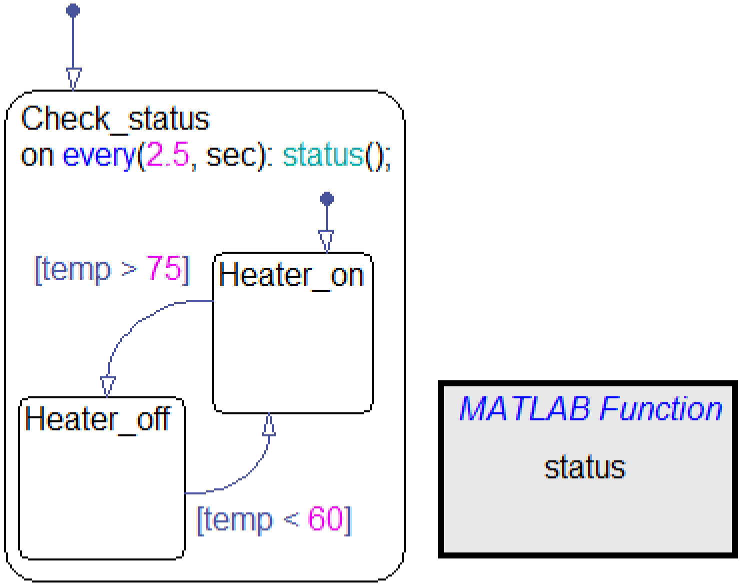
Change the transition to `after(5.33, sec)`, as shown below.



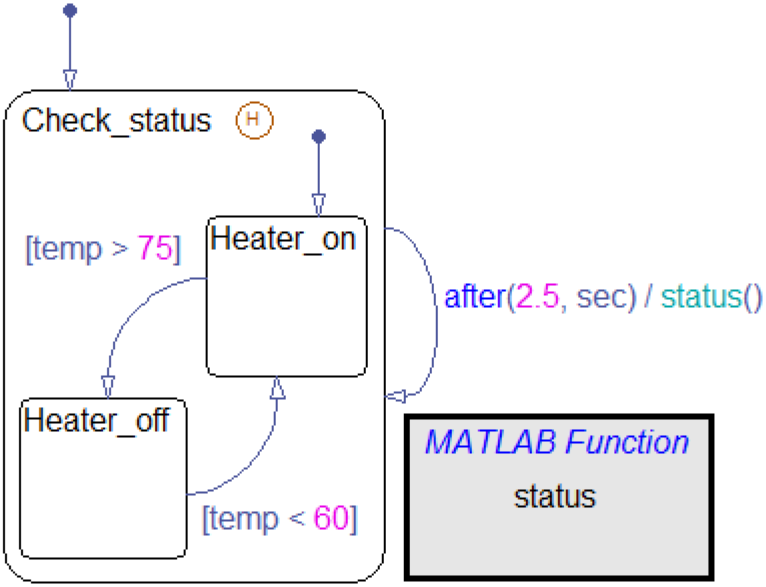
Use an Outer Self-Loop Transition with the after Operator to Replace the every Operator

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 "Recording State Activity with History Junctions" on page 7-2.)

Detecting Changes in Data Values

In this section…

Types of Data Value Changes That You Can Detect

You can detect changes in the following types of Stateflow data from one time step to the next:

- Chart inputs
- Chart outputs
- Local chart variables
- Machine-parented variables
- Data bound to Simulink data store memory

  (For more information, see "Sharing Global Data with Multiple Charts" on page 8-35.)

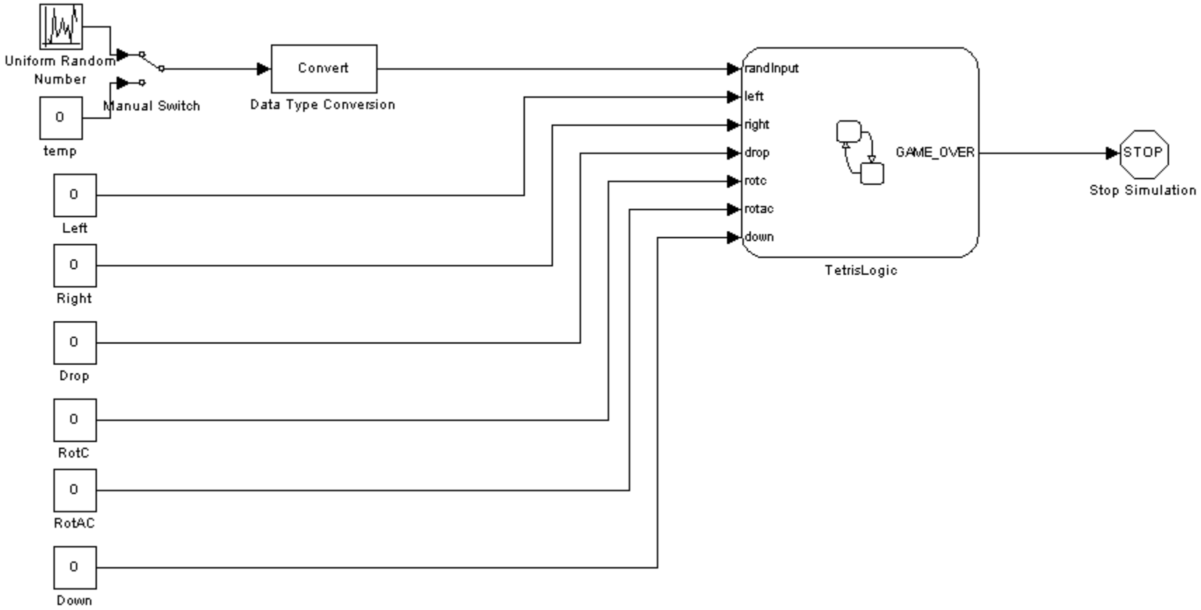
For each of these types of data, you can use operators that detect the following changes:

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 10-90.
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 10-91.
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 10-92.

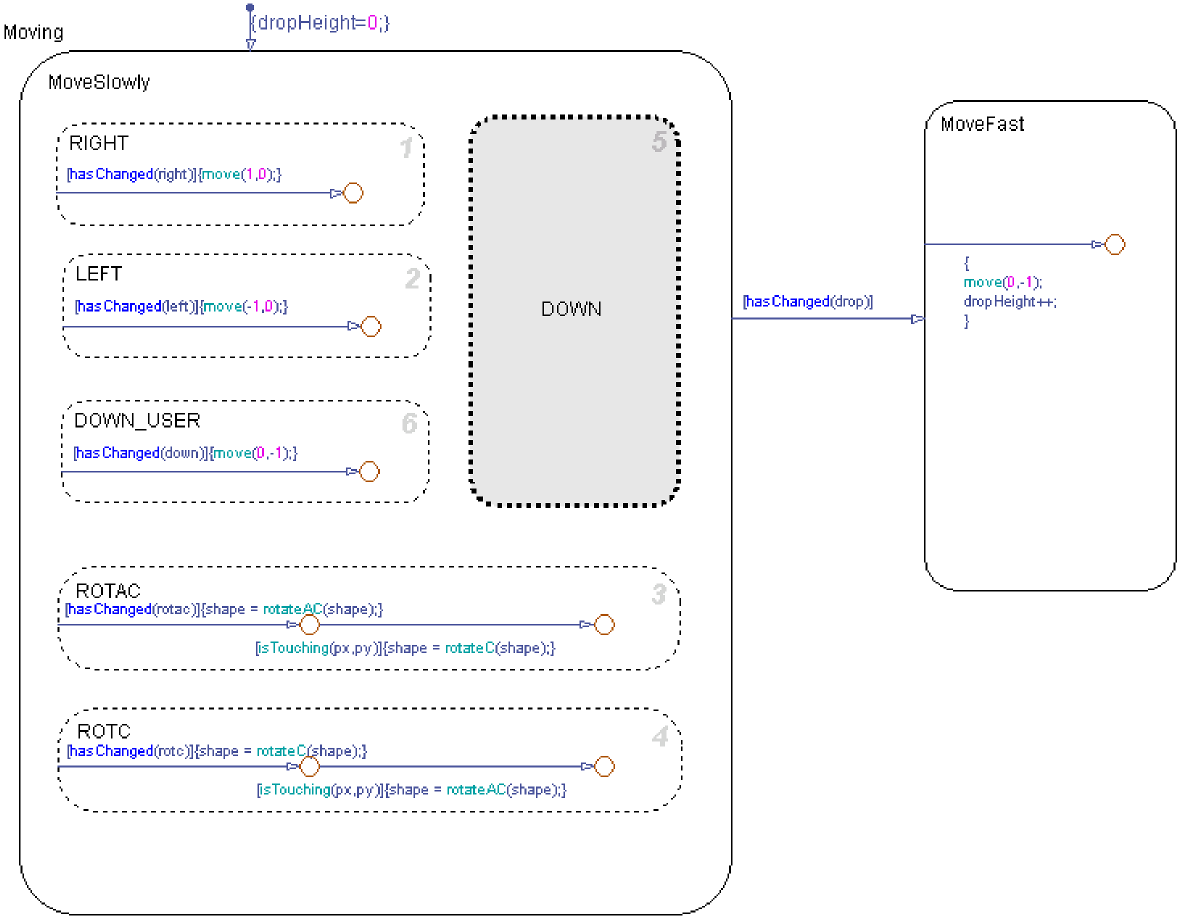
Change detection operators return 1 if the data value changes or 0 if there is no change. See "Change Detection Operators" on page 10-89.

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 model, follow these steps:

1 At the MATLAB command prompt, type:

```
demo simulink stateflow
```

2 Click the model description with the title **Tetris**.

3 Click the **Open this model** link in the upper right corner.

Tip You can also open the model by typing `sf_tetris2` at the MATLAB prompt.

---

4. Start simulation.

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

---

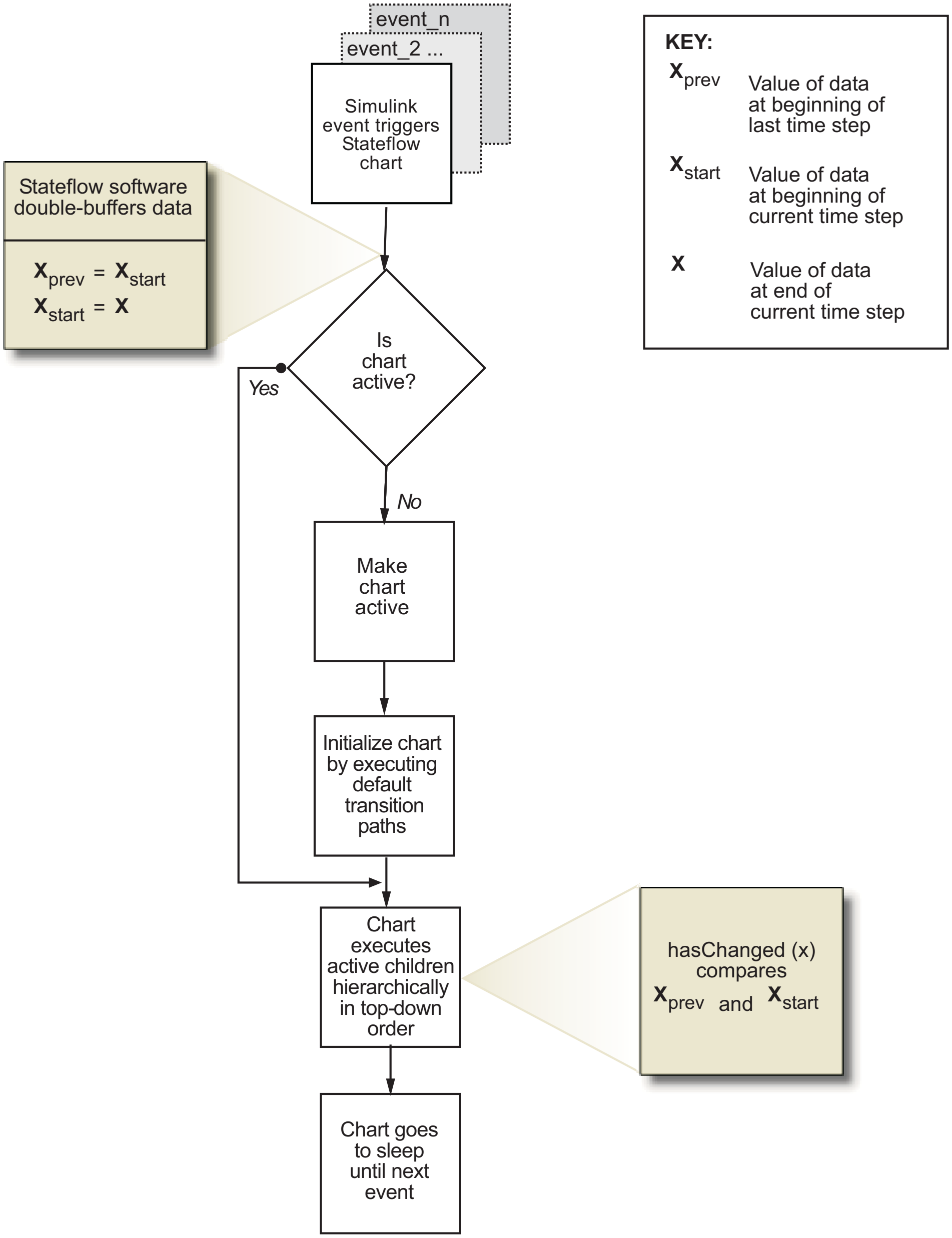
Note Double-buffering occurs once per time step except if multiple input events occur in the same time step. Then, double-buffering occurs once per input event (see "Handling Changes When Multiple Input Events Occur" on page 10-89).

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 10-89
- “Handling Changes When Multiple Input Events Occur” on page 10-89

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 10-87). 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 10-89.

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 10-90

- "hasChangedFrom Operator" on page 10-91
- "hasChangedTo Operator" on page 10-92

hasChanged Operator

The `hasChanged` operator detects any change in Stateflow data since the last time step, using the following heuristic:

$$hasChanged(x) = \begin{cases} 1 \text{ if } x_{prev} \neq x_{start} \\ 0 \text{ otherwise} \end{cases}$$

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:

$$hasChangedFrom(x, x_0) = \begin{cases} 1 \text{ if } x_{prev} \neq x_{start} \text{ and } x_{prev} = x_0 \\ 0 \text{ otherwise} \end{cases}$$

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:

$$hasChangedTo(x, x_0) = \begin{cases} 1 \text{ if } x_{prev} \neq x_{start} \text{ and } x_{start} = x_0 \\ 0 \text{ otherwise} \end{cases}$$

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.

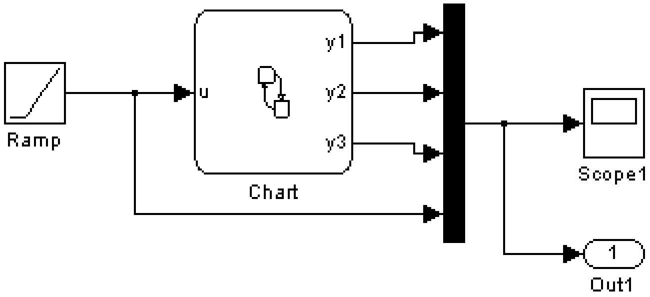
Description. `hasChangedTo ( u, v )` returns 1 if *u* changes to the value specified by *v* in the current time step. If *u* is a matrix variable, `hasChangedTo` returns 1 if any its of its elements changes value so that all elements of the matrix equal the value specified by *v* in the current time 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

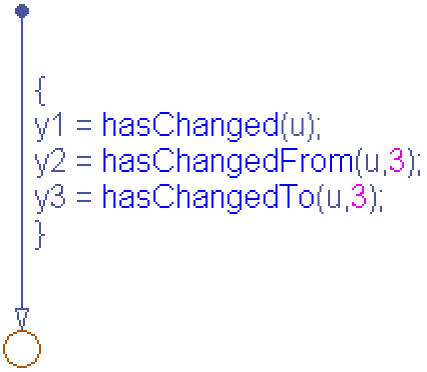
The following model shows how to use the `hasChanged`, `hasChangedFrom`, and `hasChangedTo` operators to detect specific changes in an input signal. In this example, a Ramp block sends a discrete, increasing time signal to a chart:



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:

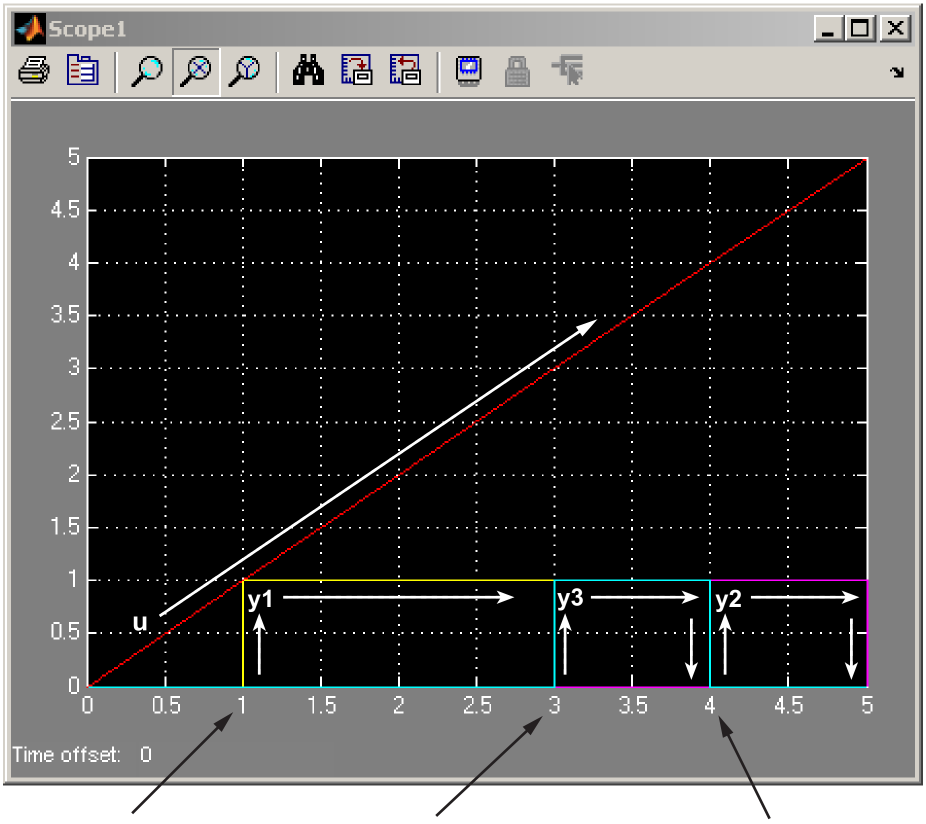
- Any change from the previous time step
- Change to the value 3
- Change from the value 3

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:



```
{
y1 = hasChanged(u);
y2 = hasChangedFrom(u,3);
y3 = hasChangedTo(u,3);
}
```

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)
```

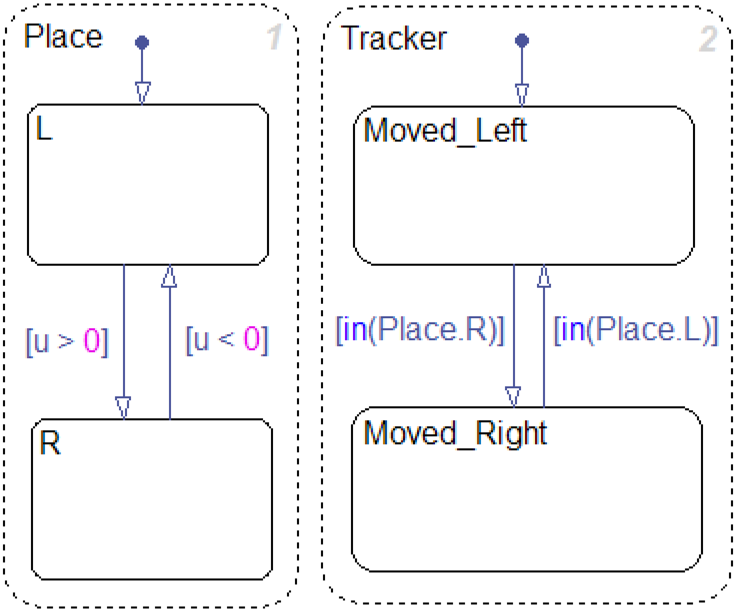
where `S` is a fully qualified state.

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.



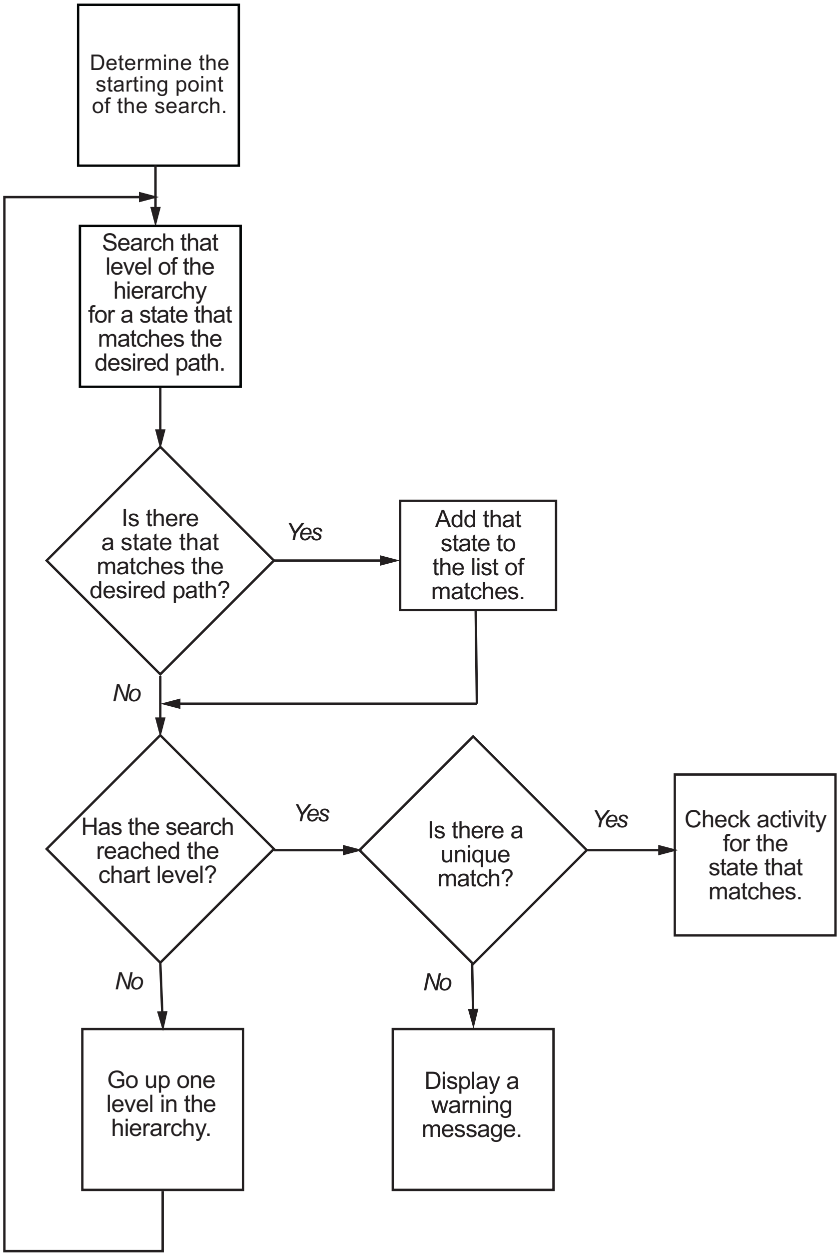
In this chart, using the `in` operator to check state activity synchronizes substates in the parallel states `Place` and `Tracker`. For example, when the input position `u` becomes positive, the state transition from `Place.L` to `Place.R` occurs. This transition makes the condition `[in(Place.R)]` true, 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.

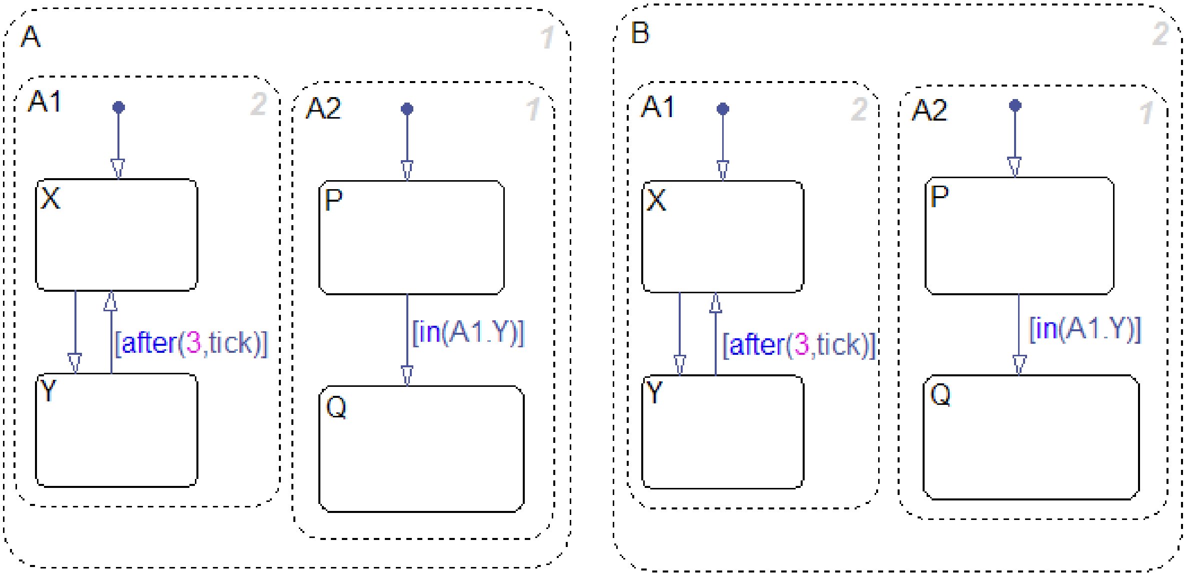
When you use the `in` operator to check state activity, these actions take place:

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.

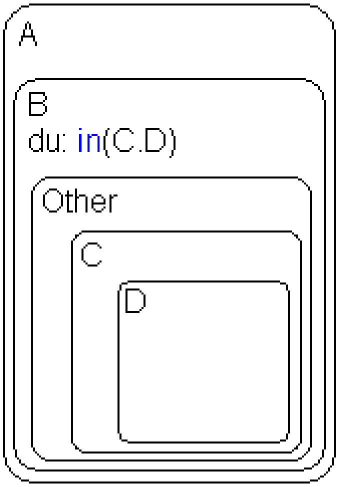
Localizing the scope of the `in` operator produces a unique search result. For example, the `in` operator of `A.A2` does not detect the state `B.A1.Y`, because the search algorithm localizes the scope of the operator. Similarly, the `in` operator of `B.A2` detects only the state `B.A1.Y` and does not 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.



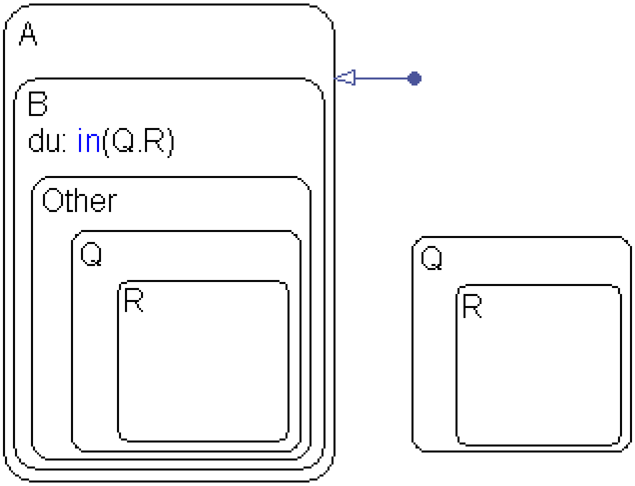
In the state `A.B`, the `during` action invokes the `in` operator. Assume that you want to check the activity of the state `A.B.Other.C.D`. The `in` operator performs these search actions:

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
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
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

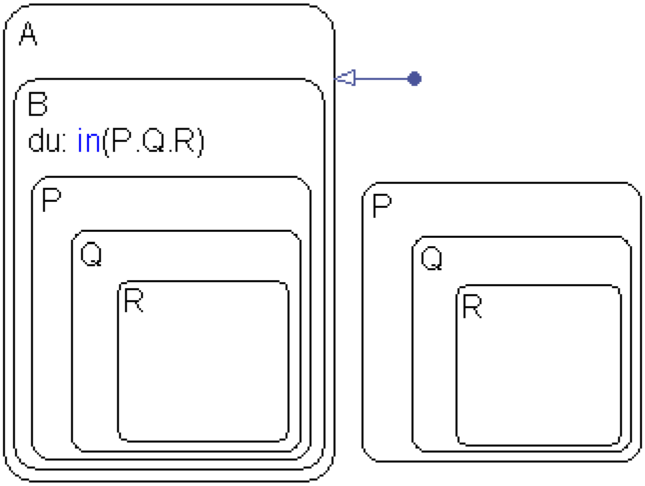
The search ends, with the single state `Q.R` found. The `in` operator checks if that state is active and returns a value of 0 or 1.

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:

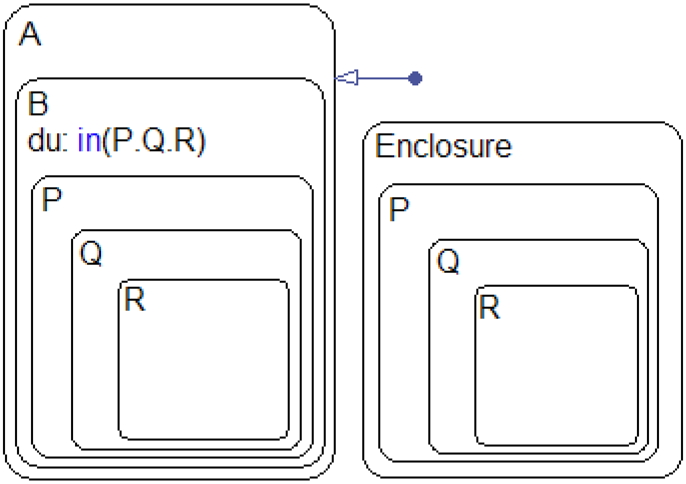
Step	Action of the `in` Operator	Finds a Match?
1	Picks `A.B` as the starting point and searches for the state `A.B.P.Q.R`	Yes
2	Moves up to the next level of the hierarchy and searches for the state `A.P.Q.R`	No
3	Moves up to the chart level and searches for the state `P.Q.R`	Yes

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...

What Are 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 the event triggers. The function-call subsystem enables when the state with the bound event is entered and disables when that state is exited. Execution of the function-call subsystem is fully bound to the activity of the state that calls it.

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 goes to sleep. To achieve a finer level of control, you can bind a function-call subsystem to a state within the chart hierarchy by using a bind action (see "Bind Actions" on page 10-6).

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.

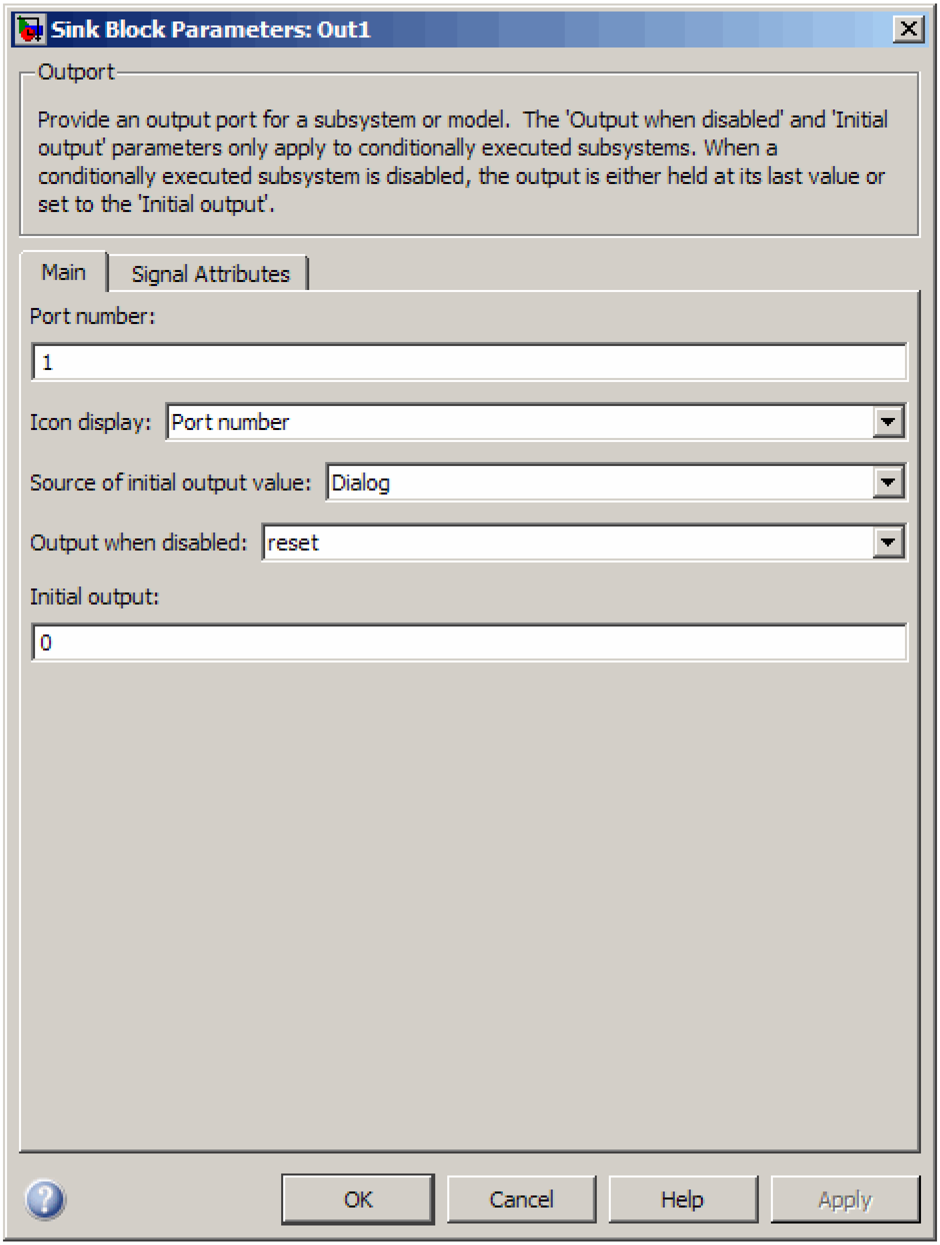
When you bind a function-call subsystem to a state, you can fine-tune the behavior of the subsystem when it is enabled and disabled, as described in the following sections:

- "Handling Outputs When the Subsystem is Disabled" on page 10-109
- "Controlling Behavior of States When the Subsystem is Enabled" on page 10-111

Handling Outputs When the Subsystem is Disabled

Although function-call subsystems do not execute while disabled, their output signals are available to other blocks in the model. If a function-call subsystem is bound to a state, you can hold its outputs at their values from the previous time step or reset the outputs to their initial values when the subsystem is disabled. Follow these steps:

1 Double-click the outport block of the subsystem to open the Block Parameters dialog box.



Outport

Provide an output port for a subsystem or model. The 'Output when disabled' and 'Initial output' parameters only apply to conditionally executed subsystems. When a conditionally executed subsystem is disabled, the output is either held at its last value or set to the 'Initial output'.

Port number: 1

Icon display: Port number

Source of initial output value: Dialog

Output when disabled: reset

Initial output: 0

2. Select an option for **Output when disabled**:

Select:	To:
`held`	Maintain most recent output value
`reset`	Reset output to its initial value

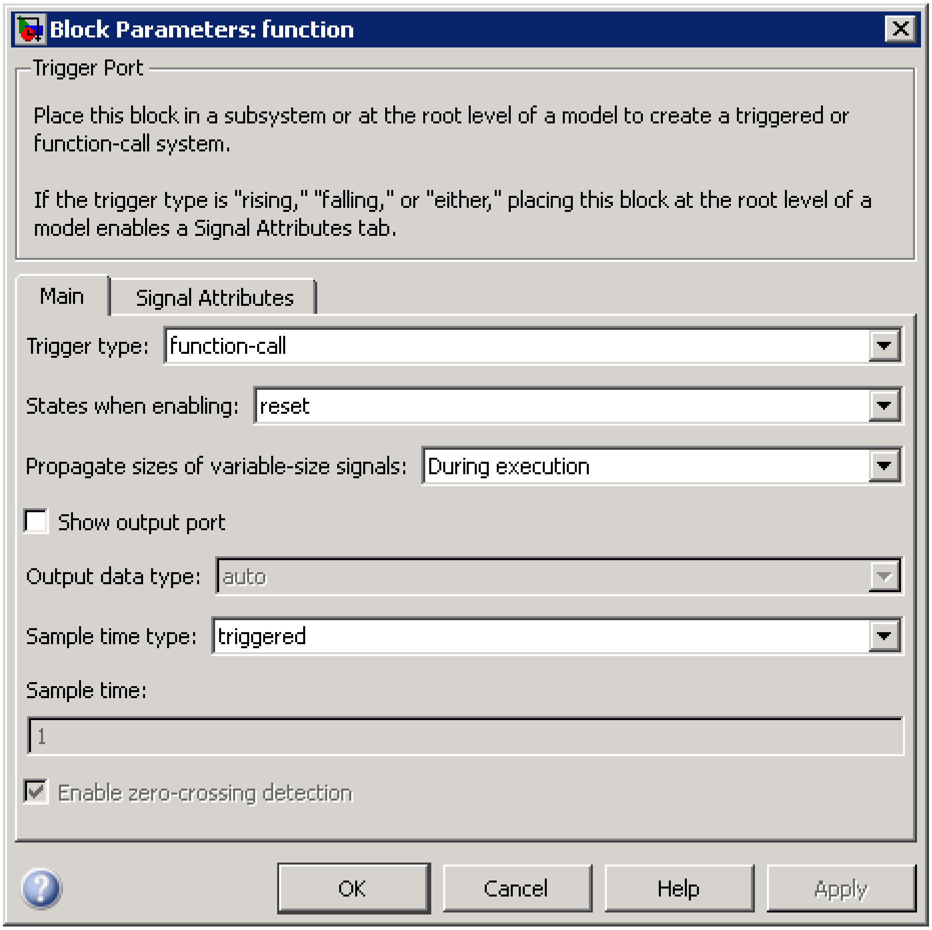
3. Click **OK** to record the settings.

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 10-108.

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 the Block Parameters dialog box.



Block Parameters: function

Trigger Port

Place this block in a subsystem or at the root level of a model to create a triggered or function-call system.

If the trigger type is "rising," "falling," or "either," placing this block at the root level of a model enables a Signal Attributes tab.

Trigger type: function-call

States when enabling: reset

Propagate sizes of variable-size signals: During execution

☐ Show output port

Output data type: auto

Sample time type: triggered

Sample time: 1

☑ Enable zero-crossing detection

2 Select an option for **States when enabling**:

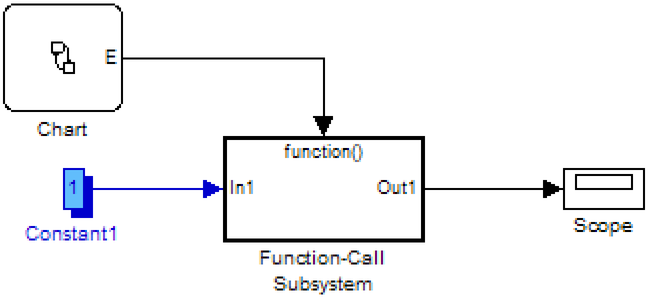
Select:	**To:**
`held`	Maintain most recent values of the states of the subsystem that contains the trigger port
`reset`	Revert to the initial conditions of the states of the subsystem that contains this trigger port
`inherit`	Inherit this setting from the function-call initiator's parent subsystem. 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 10-108.

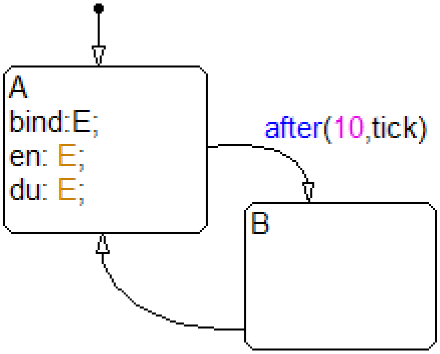
Example Model That Binds a Function-Call Subsystem to a State

The following model triggers a function-call subsystem with a trigger event `E` that binds to state `A` of a Stateflow chart:



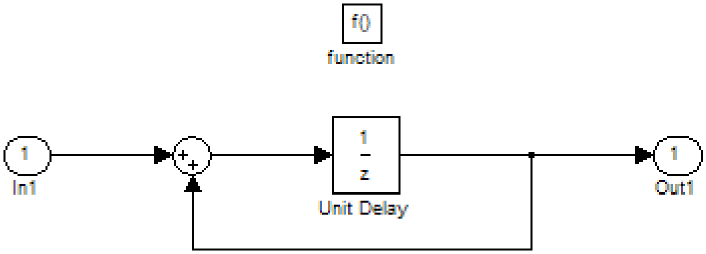
This model specifies a fixed-step solver with a fixed-step size of 1 in the **Solver** pane of the Configuration Parameters dialog box.

The Stateflow chart contains two states, `A` and `B`, and connecting transitions, along with some actions:

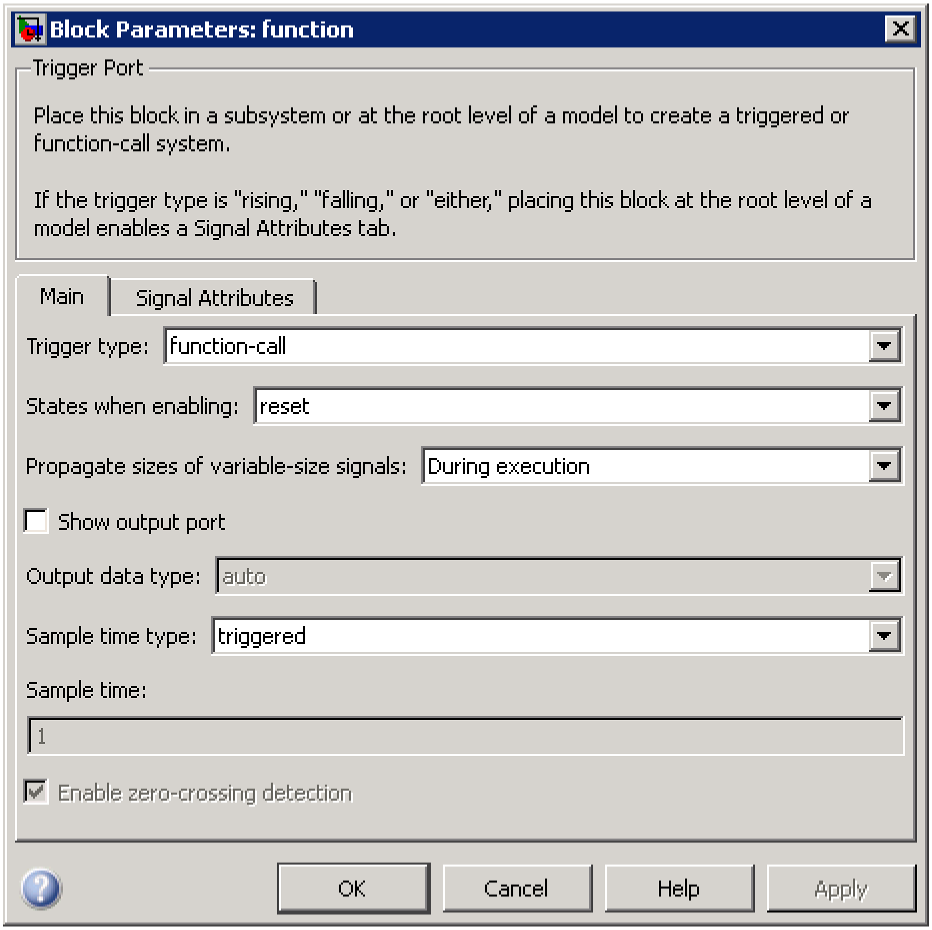


Event `E` binds to state A with the action `bind:E`. Event `E` is defined for the Stateflow chart with a scope of `Output to Simulink` and a trigger type of `function-call`.

The function-call subsystem contains a trigger port block, an input port, an output port, and a simple block diagram. The block diagram increments a counter by 1 at each time step, using a Unit Delay block:



The Block Parameters dialog box for the trigger port appears as follows.



Block Parameters: function

Trigger Port

Place this block in a subsystem or at the root level of a model to create a triggered or function-call system.

If the trigger type is "rising," "falling," or "either," placing this block at the root level of a model enables a Signal Attributes tab.

Trigger type: function-call

States when enabling: reset

Propagate sizes of variable-size signals: During execution

☐ Show output port

Output data type: auto

Sample time type: triggered

Sample time:

1

☑ Enable zero-crossing detection

The **States when enabling** parameter uses the setting `reset`. This setting resets the state values for the function-call subsystem to zero when it is enabled.

The **Sample time type** parameter uses the setting `triggered`. This setting sets the function-call subsystem to execute only when it is triggered by a calling event while it is enabled.

Setting **Sample time type** to `periodic` enables the **Sample time** field below it, which defaults to 1. These settings force the function-call subsystem to execute for each time step specified in the **Sample time** field while it is enabled. To accomplish this, the state that binds the calling event for the function-call subsystem must send an event for the time step coinciding with the specified sampling rate in the **Sample time** field. States can send events with `entry` or `during` actions at the simulation sample rate.

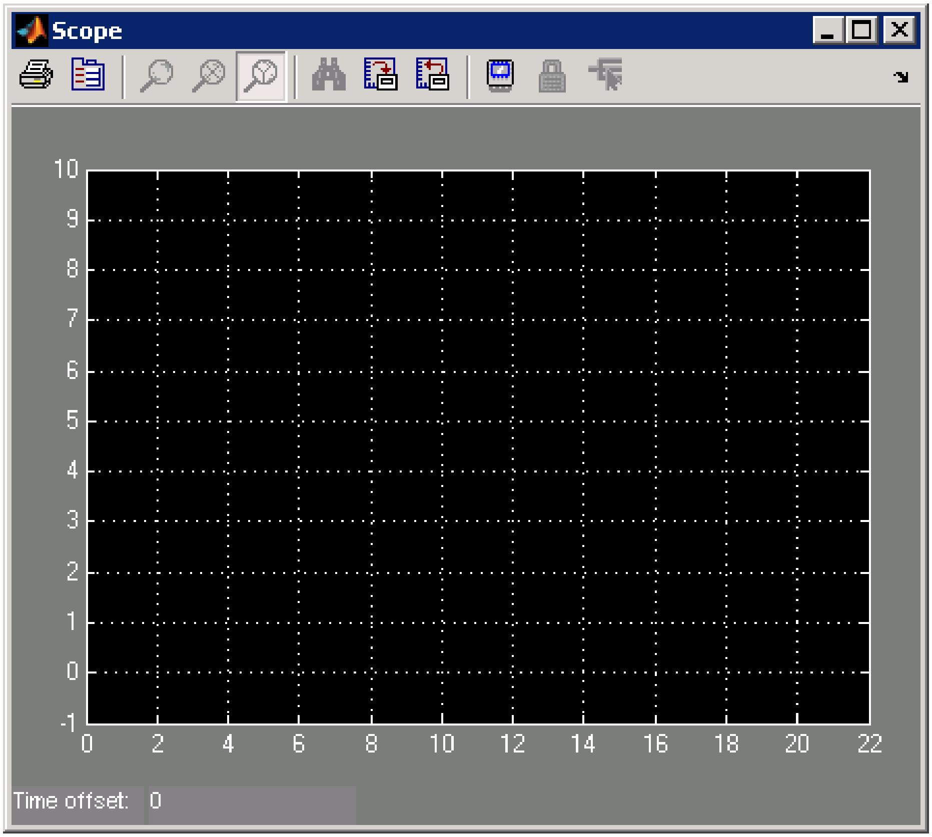
- For fixed-step sampling, the **Sample time** value must be an integer multiple of the fixed-step size.
- For variable-step sampling, the **Sample time** value has no limitations.

Behavior of a Bound Function-Call Subsystem

To see how a state controls a bound function-call subsystem, begin simulating the model in "Example Model That Binds a Function-Call Subsystem to a State" on page 10-113. The following steps describe the output of the subsystem.

1. In the chart, the default transition to state `A` occurs.

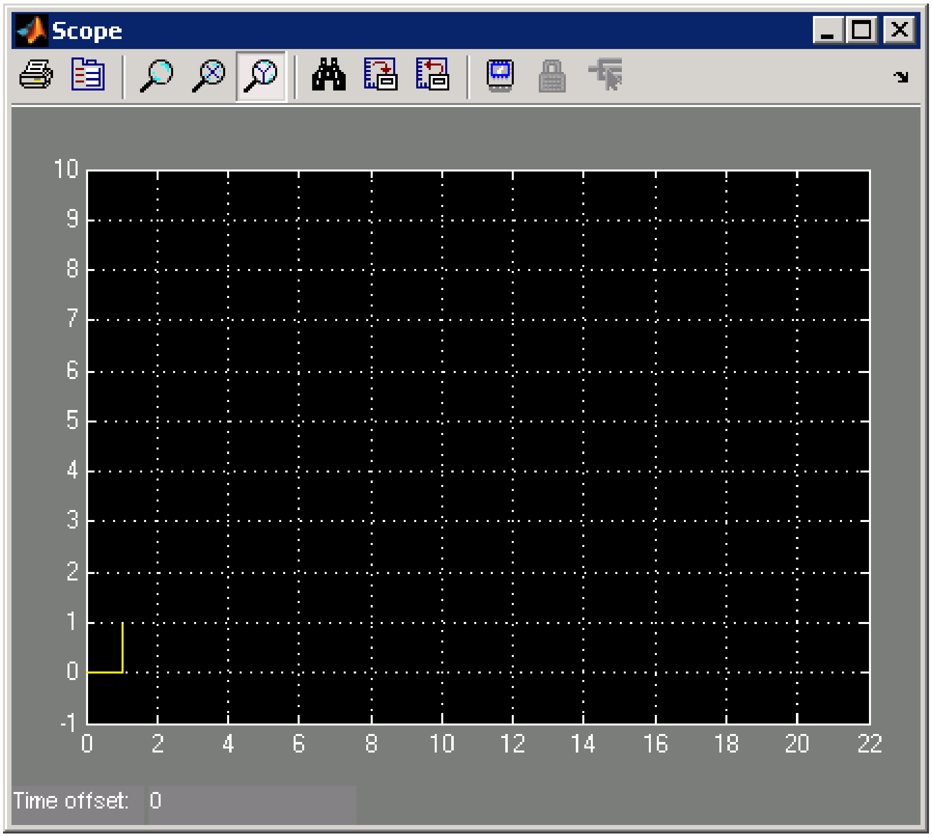
2. When state `A` becomes active, it executes its bind and entry actions. The binding action, `bind:E`, binds event `E` to state `A`. This action enables the function-call subsystem and resets its state variables to 0.



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. Because the previous content of the Unit Delay block is 0 after the reset, the initial output is 0 and the current value of 1 is held for the next call to the subsystem.

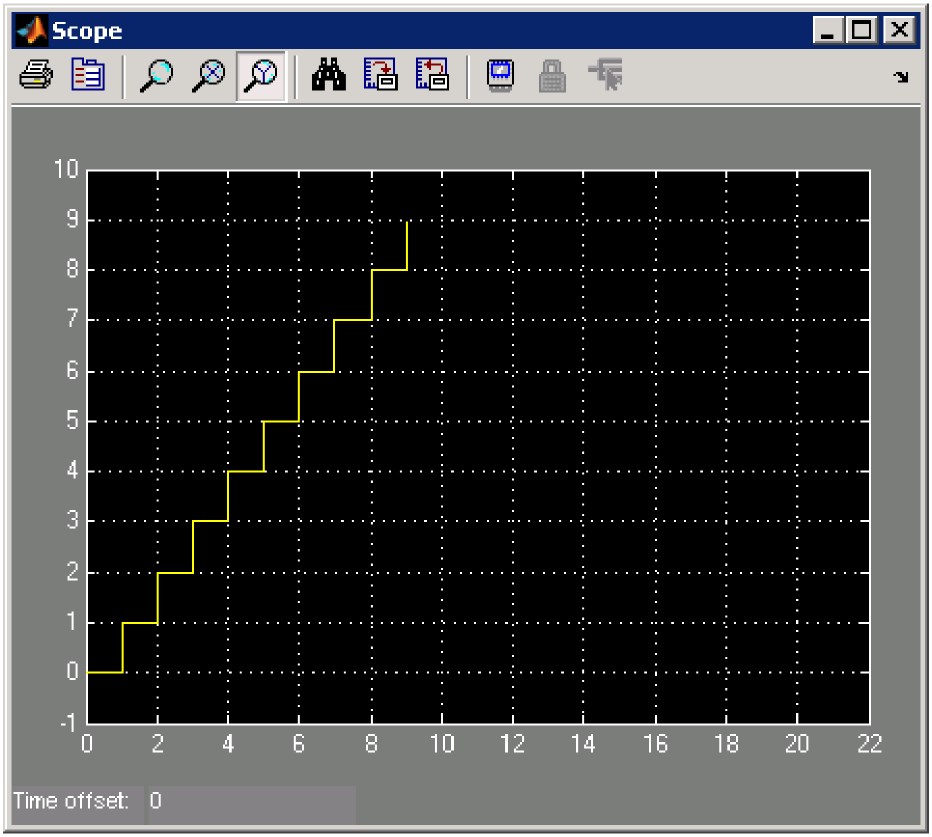
3 The next update event from the model tests state `A` for an outgoing transition.

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. For the second update, the during action of state `A`, `du:E`, executes, which sends an event to trigger the function-call subsystem. The held content of the Unit Delay block, 1, outputs to the scope.



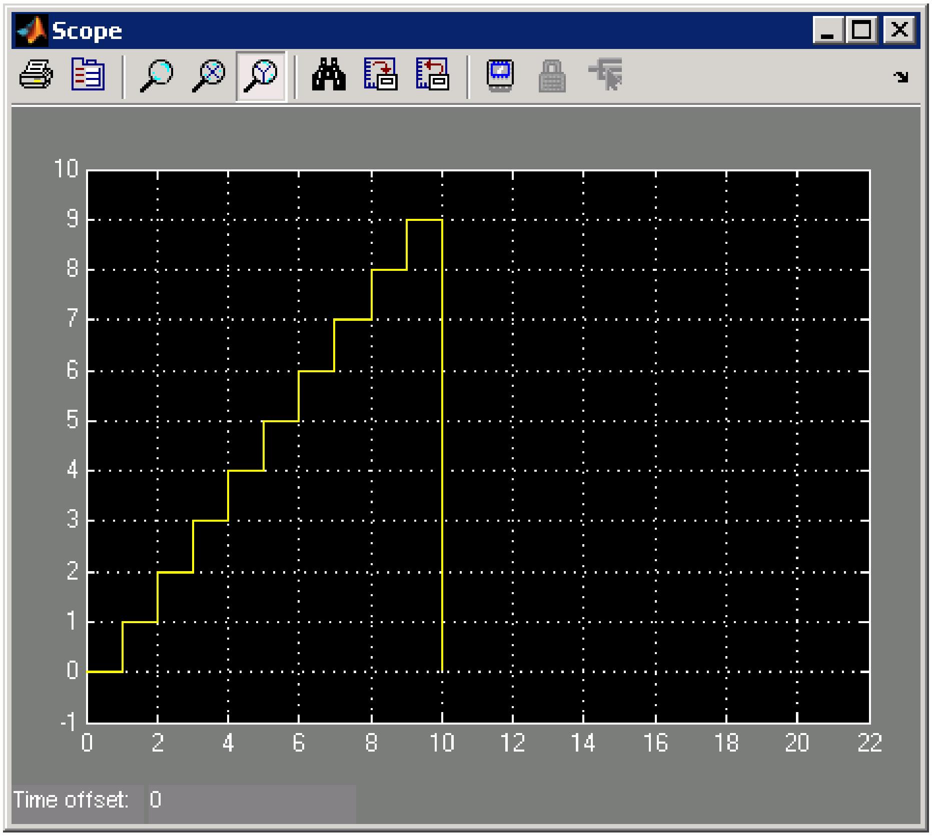
The subsystem also adds 1 to the held value to produce the value 2, which the Unit Delay block holds for the next triggered execution.

4 The next eight update events increment the subsystem output by 1 at each time step.



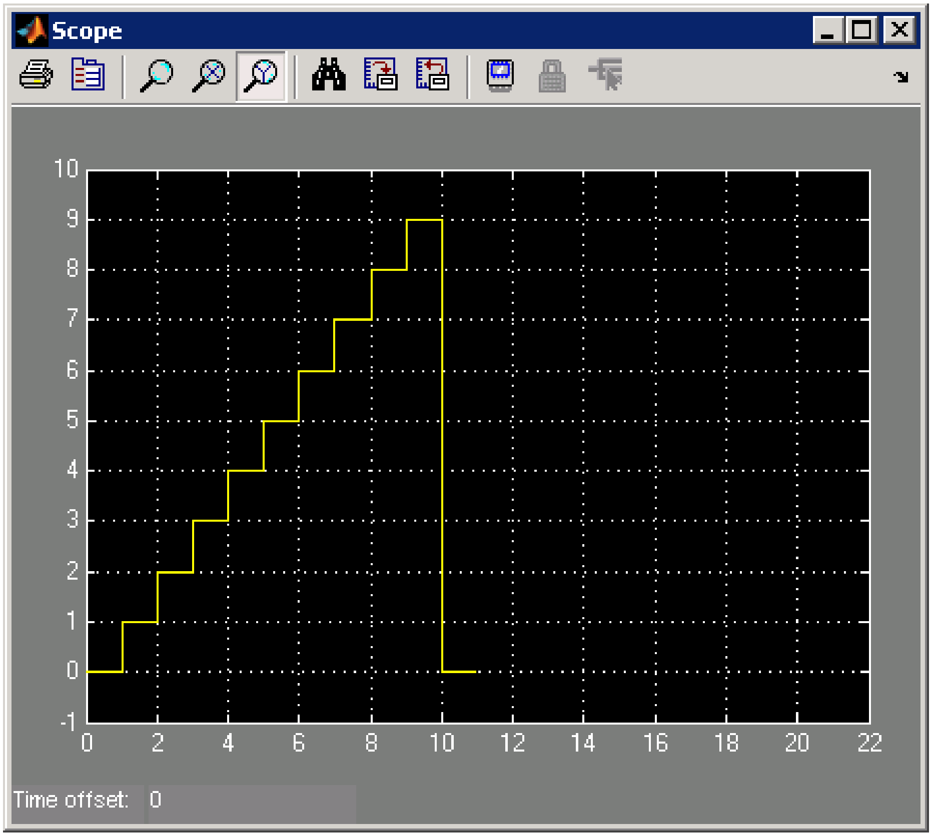
5 On the 11th update event, the transition to state `B` occurs and state `B` becomes active.

Because the binding to state `A` is no longer active, the function-call subsystem is disabled, and its output drops to 0.

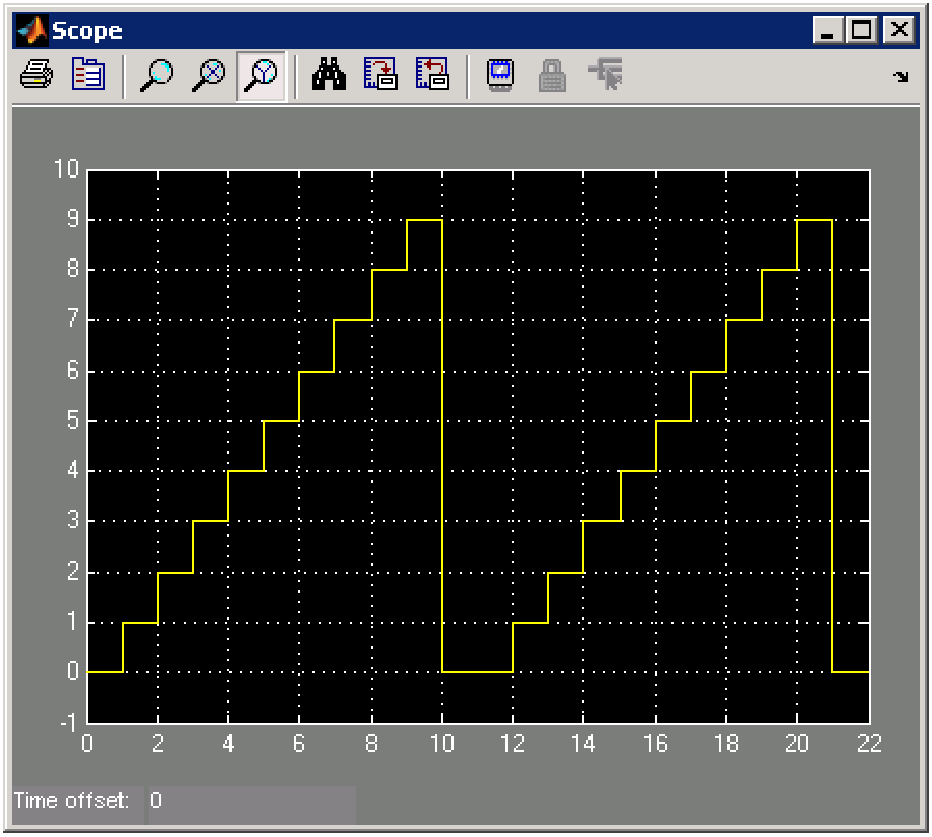


6 When the next sampling event occurs, the transition from state `B` to state `A` occurs.

Again, the binding action, `bind: E`, enables the function-call subsystem and resets its output to 0.

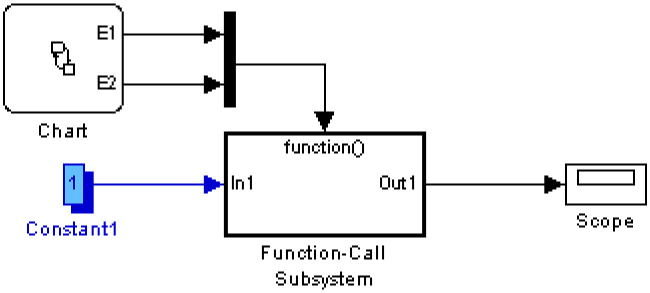


7 The next 10 update events produce the following output.

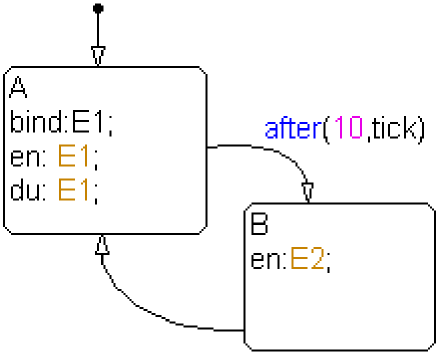


Why Avoid Muxed Trigger Events with Binding

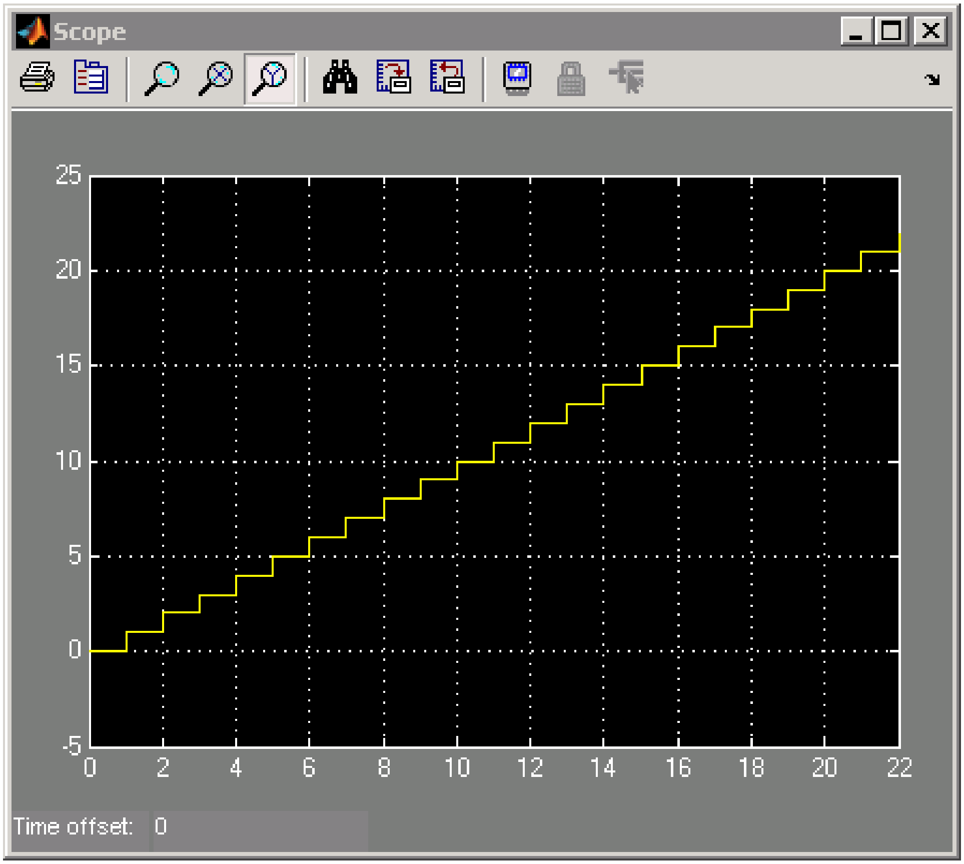
The example in "Behavior of a Bound Function-Call Subsystem" on page 10-116 shows how binding events gives control of a function-call subsystem to a single state in a Stateflow chart. This control does not work when you allow other events to trigger the function-call subsystem through a mux. For example, the following model defines two function-call events to trigger a function-call subsystem using a Mux block:



In the chart, `E1` binds to state `A`, but `E2` does not. State `B` sends the triggering event `E2` in its entry action:



When you simulate this model, you get the following output:



Broadcasting `E2` in state `B` changes the output, which no longer resets.

Note Binding is not recommended when you provide multiple trigger events to a function-call subsystem through a mux. Muxed trigger events can interfere with event binding and cause undefined behavior.

11

Making States Reusable with Atomic Subcharts

What Is an Atomic Subchart?

In a Stateflow chart, an atomic subchart is a graphical object that helps you reuse the same state or subchart across multiple charts and models. Atomic subcharts allow:

- Ease of team development for people working on different parts of the same chart
- Faster simulation after making small changes to a chart with many states or levels of hierarchy
- Manual inspection of generated code for a specific state or subchart in a chart
- Ability to animate and debug multiple charts side by side

States, subcharts, and atomic subcharts have these key similarities and differences:

Capability	State	Subchart	Atomic Subchart
Can behave as a standalone chart	No	No	Yes
Can generate reusable code	No	No	Yes
Supports access to event broadcasts outside the scope of that object	Yes	Yes	No
Supports access to data at any level of the hierarchy	Yes	Yes	No

Atomic subcharts and atomic subsystems (see Atomic Subsystem in the Simulink documentation) have these key similarities and differences:

Capability	Atomic Subchart	Atomic Subsystem
Supports generation of reusable code	Yes	Yes
Supports usage as a library link	Yes	Yes
Requires parameterizing of data when used as a library link	Yes	No
Supports explicit specification of sample time	No	Yes

The following demos show how to use atomic subcharts for modeling typical applications:

This demo...	Shows how you can model...
`sf_atomic_sensor_pair`	A redundant sensor pair
`sf_elevator`	An elevator system with two identical lifts

When to Use Atomic Subcharts

Consider using atomic subcharts when one or more of these scenarios apply:

Scenario	Reason for Using Atomic Subcharts	Reference	Tutorial
You want to reuse the same state or subchart many times across different charts or models to facilitate large-scale modeling.	You can store an atomic subchart in a library to enable reuse across different charts and models. When you change an atomic subchart in a library, the change propagates to all links.	"Comparison of Modeling Methods" on page 11-5	"Tutorial: Reusing a State Multiple Times in a Chart" on page 11-53
You want to use simulation to test your changes, one by one, without recompiling the entire chart.	When you modify an atomic subchart, recompilation occurs for only that object and not the entire chart.	"Comparison of Simulation Methods" on page 11-6	"Tutorial: Reducing the Compilation Time of a Chart" on page 11-63
You want to break a chart into standalone parts because multiple people are working on different parts of the chart.	Because atomic subcharts behave as standalone objects, people can work on different parts of a chart without affecting any work that someone else is doing.	"Comparison of Editing Methods" on page 11-7	"Tutorial: Dividing a Chart into Separate Units for Editing" on page 11-65
You want to inspect Simulink Coder generated code manually for a specific part of a chart.	You can specify that code for an atomic subchart appears in a separate file for unit testing.	"Comparison of Code Generation Methods" on page 11-8	"Tutorial: Generating Reusable Code for Unit Testing" on page 11-68

Benefits of Using Atomic Subcharts in a Stateflow Chart

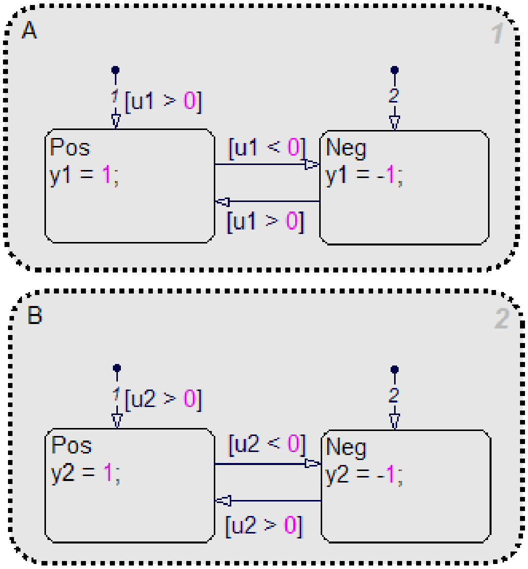
In this section...

Comparison of Modeling Methods

The following sections compare two ways of modeling similar states in charts.

Modeling Without Atomic Subcharts

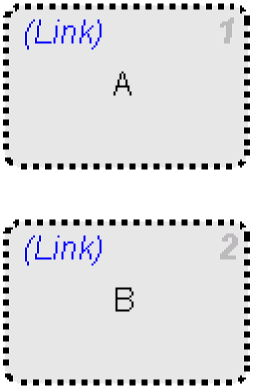
You create a separate instance of each state in your chart.



In this chart, the only difference between the two states are the names of variables.

Modeling With Atomic Subcharts

You create a single state and convert it to an atomic subchart, which you store in a new library. From that library, you can copy and paste the atomic subchart for use in any chart and update the mapping of inputs, outputs, local data, or parameters as needed.



This modeling method minimizes maintenance of similar states. When you modify the atomic subchart in the library, your changes propagate automatically to the links in all charts and models.

For more information, see “Tutorial: Reusing a State Multiple Times in a Chart” on page 11-53.

Comparison of Simulation Methods

The following sections compare two ways of simulating a chart.

Simulation Without Atomic Subcharts

You make a small change to one part of a chart that contains many states or several levels of hierarchy. When you start simulation to test that change, recompilation occurs for the entire chart.

Because recompiling the entire chart takes a long time, you make several changes before testing. However, if you find an error, you must step through all your changes to identify what causes the error.

Simulation With Atomic Subcharts

You make a small change to an atomic subchart in a chart that contains many states or several levels of hierarchy. When you start simulation to test that change, recompilation occurs only for the atomic subchart.

Incremental builds for simulation decrease the time required to recompile the chart. This reduction enables you to test each change, one by one, instead of waiting to test multiple changes. By testing each change individually, you can quickly identify a change that causes an error.

For more information, see "Tutorial: Reducing the Compilation Time of a Chart" on page 11-63.

Comparison of Editing Methods

The following sections compare two ways of editing a chart.

Editing Without Atomic Subcharts

You edit one part of a chart, while someone else edits another part of the same chart. At submission time, you merge your changes with someone else's edits.

Editing With Atomic Subcharts

You store one part of a chart as an atomic subchart in a library. You edit that subchart separately, while someone else edits the main chart. At submission time, no merge is necessary because the changes exist in separate models.

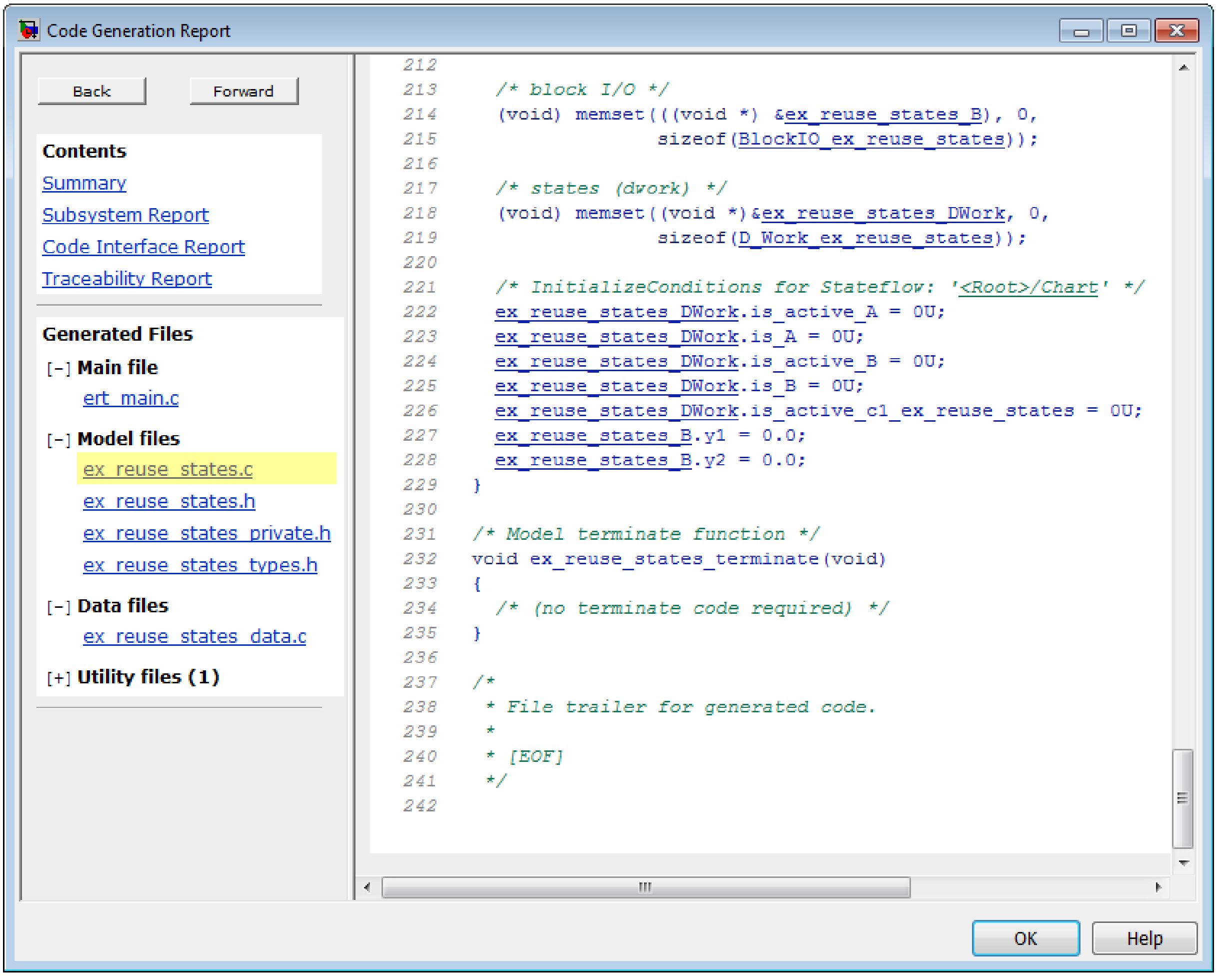
For more information, see "Tutorial: Dividing a Chart into Separate Units for Editing" on page 11-65.

Comparison of Code Generation Methods

The following sections compare two ways of generating code.

Code Generation Without Atomic Subcharts

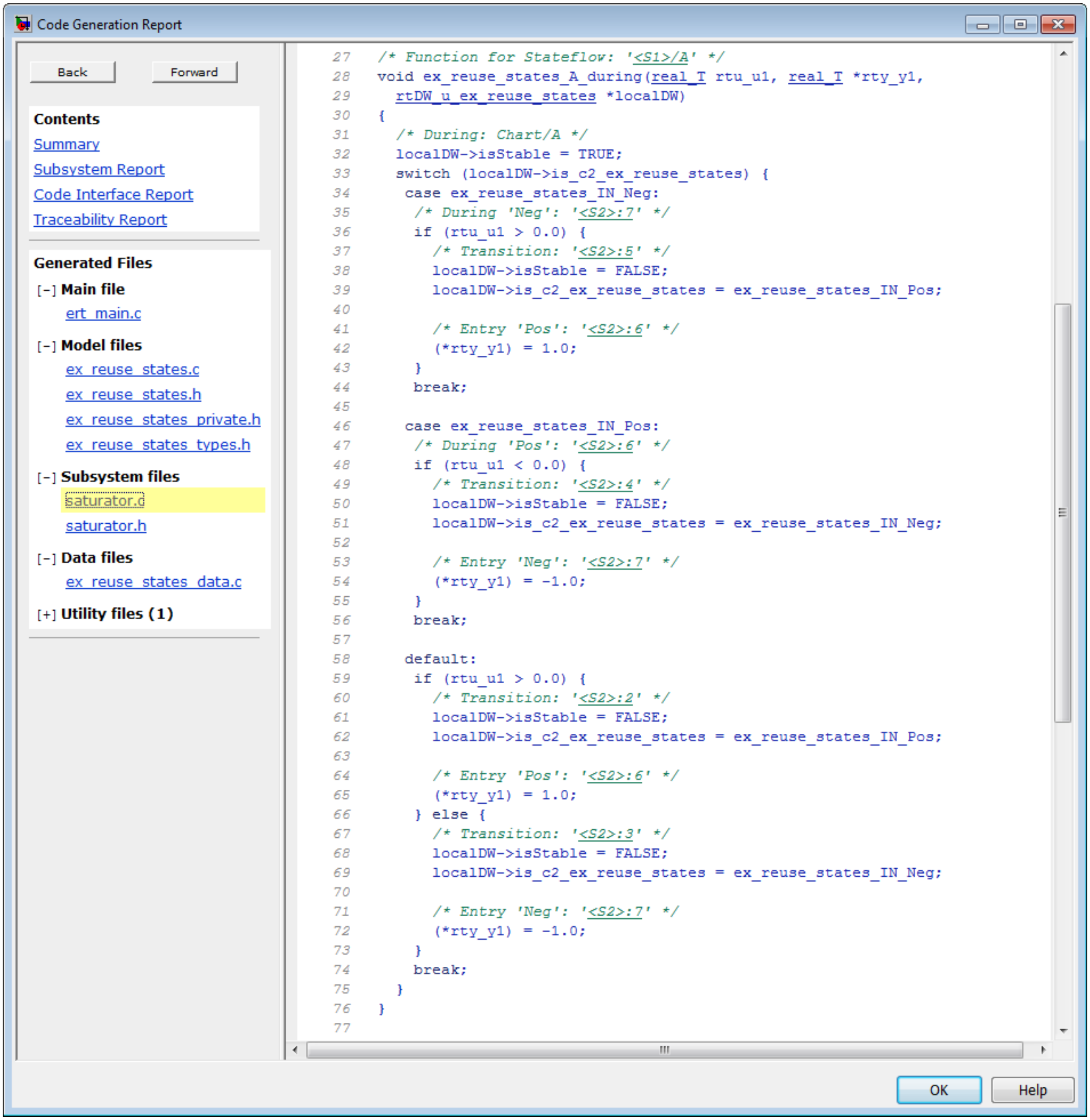
You generate code for the entire model in one file and look through that entire file to find code for a specific part of the chart.



```

  /* block I/O */
  (void) memset(((void *) &ex_reuse_states_B), 0,
                sizeof(BlockIO_ex_reuse_states));

  /* states (dwork) */
  (void) memset((void *)&ex_reuse_states_DWork, 0,
                sizeof(D_Work_ex_reuse_states));

  /* InitializeConditions for Stateflow: '<Root>/Chart' */
  ex_reuse_states_DWork.is_active_A = 0U;
  ex_reuse_states_DWork.is_A = 0U;
  ex_reuse_states_DWork.is_active_B = 0U;
  ex_reuse_states_DWork.is_B = 0U;
  ex_reuse_states_DWork.is_active_c1_ex_reuse_states = 0U;
  ex_reuse_states_B.y1 = 0.0;
  ex_reuse_states_B.y2 = 0.0;
}

/* Model terminate function */
void ex_reuse_states_terminate(void)
{
  /* (no terminate code required) */
}

/*
 * File trailer for generated code.
 *
 * [EOF]
 */

```

Code Generation With Atomic Subcharts

You specify code generation parameters so that code for an atomic subchart appears in a separate file.



```
/* Function for Stateflow: '<S1>/A' */
void ex_reuse_states_A_during(real_T rtu_u1, real_T *rty_y1,
  rtDW_u_ex_reuse_states *localDW)
{
  /* During: Chart/A */
  localDW->isStable = TRUE;
  switch (localDW->is_c2_ex_reuse_states) {
   case ex_reuse_states_IN_Neg:
    /* During 'Neg': '<S2>:7' */
    if (rtu_u1 > 0.0) {
      /* Transition: '<S2>:5' */
      localDW->isStable = FALSE;
      localDW->is_c2_ex_reuse_states = ex_reuse_states_IN_Pos;

      /* Entry 'Pos': '<S2>:6' */
      (*rty_y1) = 1.0;
    }
    break;

   case ex_reuse_states_IN_Pos:
    /* During 'Pos': '<S2>:6' */
    if (rtu_u1 < 0.0) {
      /* Transition: '<S2>:4' */
      localDW->isStable = FALSE;
      localDW->is_c2_ex_reuse_states = ex_reuse_states_IN_Neg;

      /* Entry 'Neg': '<S2>:7' */
      (*rty_y1) = -1.0;
    }
    break;

   default:
    if (rtu_u1 > 0.0) {
      /* Transition: '<S2>:2' */
      localDW->isStable = FALSE;
      localDW->is_c2_ex_reuse_states = ex_reuse_states_IN_Pos;

      /* Entry 'Pos': '<S2>:6' */
      (*rty_y1) = 1.0;
    } else {
      /* Transition: '<S2>:3' */
      localDW->isStable = FALSE;
      localDW->is_c2_ex_reuse_states = ex_reuse_states_IN_Neg;

      /* Entry 'Neg': '<S2>:7' */
      (*rty_y1) = -1.0;
    }
    break;
  }
}
```

This method of code generation enables unit testing for a specific part of a chart. You can avoid searching through unrelated code and focus only on the part that interests you.

For more information, see "Tutorial: Generating Reusable Code for Unit Testing" on page 11-68.

Restrictions for Converting to Atomic Subcharts

In this section...

Rationale for Restrictions

Atomic subcharts facilitate the reuse of states and subcharts as standalone objects. The restrictions in the following sections help you get the benefits described in “Benefits of Using Atomic Subcharts in a Stateflow Chart” on page 11-5.

Access to Data, Graphical Functions, and Events

To convert a state or subchart to an atomic subchart, access to objects not parented by the state or subchart must be one of the following:

- Chart-level data
- Chart-level graphical functions
- Input events

The following restrictions also apply:

If a state or subchart accesses...	Then...
Chart-level data	That data must have: • Static, deterministic size • A built-in data type
Chart-level graphical functions	The chart must export those functions. For more information, see "Exporting Chart-Level Graphical Functions" on page 7-39.

Use of Event Broadcasts

The state or subchart that you want to convert to an atomic subchart cannot refer to:

- Local events that are outside the scope of that state or subchart
- Output events

However, the state or subchart you want to convert can refer to *input* events.

Access to Local Data with a Nonzero First Index

The state or subchart that you want to convert to an atomic subchart cannot access local data where the **First index** property is nonzero. For the conversion process to work, the **First index** property of the local data must be zero, which is the default value.

Use of Machine-Parented Data

The state or subchart that you want to convert to an atomic subchart cannot reside in a chart that uses machine-parented data with the following properties:

- Imported or exported
- Is 2-D or higher, or uses a fixed-point type

Machine-parented data with these properties prevent reuse of generated code and other code optimizations.

Use of Strong Data Typing with Simulink Inputs and Outputs

To convert a state or subchart to an atomic subchart, your chart must use strong data typing with Simulink inputs and outputs. To specify strong data typing:

1. Open the Chart properties dialog box.
2. Select **Use Strong Data Typing with Simulink I/O**.
3. Click **OK** to close the dialog box.

Use of Output State Activity

The state or subchart that you want to convert to an atomic subchart cannot output state activity. To disable this setting:

1. Open the State properties dialog box.
2. Clear **Output State Activity**.
3. Click **OK** to close the dialog box.

Use of Supertransitions

The state or subchart that you want to convert to an atomic subchart cannot have any supertransitions crossing the boundary.

Converting to and from Atomic Subcharts

Converting a State or Subchart to an Atomic Subchart

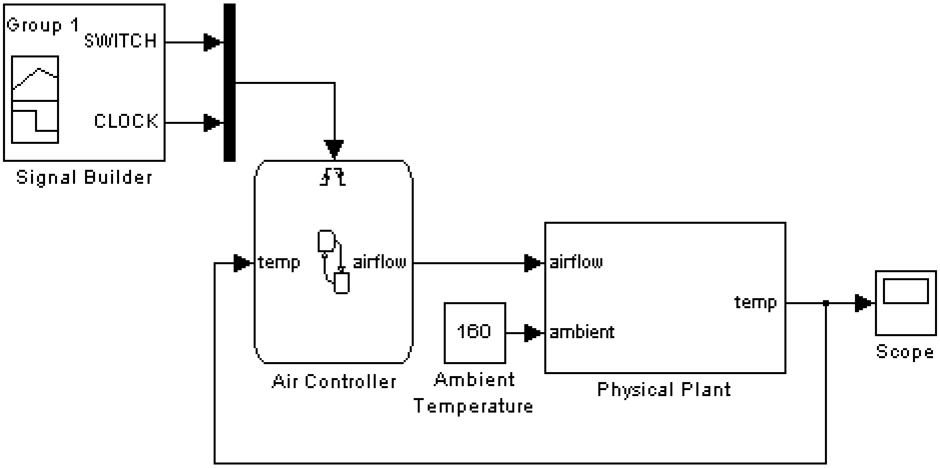
To convert a state or subchart to an atomic subchart, right-click the object in your chart and select **Make Contents > Atomic Subcharted**.

After you convert a state or subchart to an atomic subchart, local data appears as data store memory in the atomic subchart.

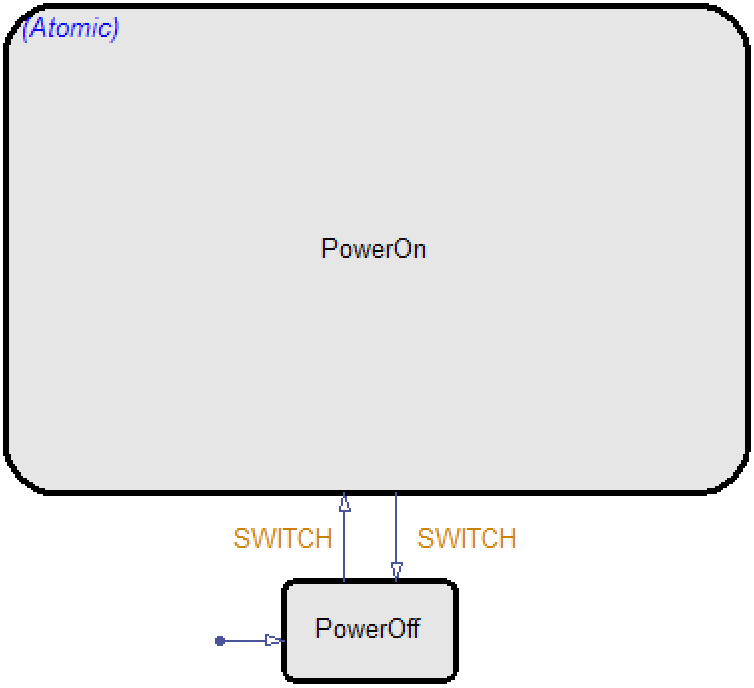
Scope of Data Before Conversion	Scope of Data After Conversion
Input	Input
Output	Output
Local	Data store memory
Parameter	Parameter
Constant	Constant

An atomic subchart looks opaque like a regular subchart but includes the label **(Atomic)** in the upper-left corner. If you use a linked atomic subchart from a library, the label **(Link)** appears in the upper-left corner.

For example, the following model contains a chart, Air Controller, that uses an atomic subchart:



In the Air Controller chart, `PowerOn` is an atomic subchart, but `PowerOff` is a regular subchart:



Converting an Atomic Subchart to a State or Subchart

When an Atomic Subchart Is a Library Link

To convert a linked atomic subchart back to a state or subchart:

1. Right-click the atomic subchart and select **Link Options > Disable Link**.
2. Follow the steps in "When an Atomic Subchart Is Not a Library Link" on page 11-18.

When an Atomic Subchart Is Not a Library Link

To convert an atomic subchart back to...	Follow these steps...
A state	1 Right-click the atomic subchart in your chart and clear the **Make Contents > Atomic Subcharted** check box. 2 Right-click the object again and clear the **Make Contents > Subcharted** check box. You might need to rearrange graphical objects in your chart after performing this step.
A regular subchart	1 Right-click the atomic subchart in your chart and clear the **Make Contents > Atomic Subcharted** check box.

Restrictions for Converting an Atomic Subchart to a State or Subchart

In the following cases, converting an atomic subchart to a state or subchart does not work:

- Your atomic subchart uses a MATLAB function and contains a nontrivial mapping of variables. A mapping is nontrivial when the variable in the subchart does not map to another variable of the same name in the container chart.
- A parameter in the atomic subchart maps to something other than a single variable. For example, the following mappings for a parameter named `data1` prevent conversion of an atomic subchart to a state or subchart:
  - `data2 + 3`
  - `data2.3`
  - `data2(3)`
  - `3`

For more information, see "Mapping Variables for Atomic Subcharts" on page 11-20.

Mapping Variables for Atomic Subcharts

In this section...

Why Map Variables for Atomic Subcharts?

Variables in an atomic subchart do not always map directly to variables in the main chart. To ensure that each variable in your atomic subchart maps to the correct variable in the main chart, you edit the mapping (or parameterize the link). For details, see:

- "Mapping Input and Output Data for an Atomic Subchart" on page 11-21
- "Mapping Data Store Memory for an Atomic Subchart" on page 11-26
- "Mapping Parameter Data for an Atomic Subchart" on page 11-30
- "Mapping Input Events for an Atomic Subchart" on page 11-34

How to Map Variables in an Atomic Subchart

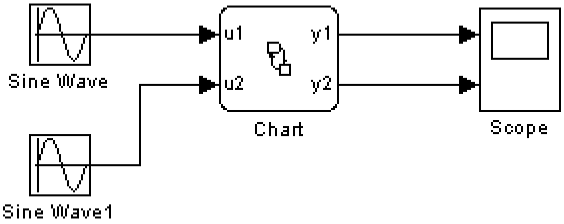
Depending on the scope of data or events in an atomic subchart, you update different sections on the **Mappings** tab of the State properties dialog box.

For...	Go to...	And...
Input data	**Input Mapping**	Specify the chart input or output data that corresponds to each atomic subchart symbol.
Output data	**Output Mapping**	

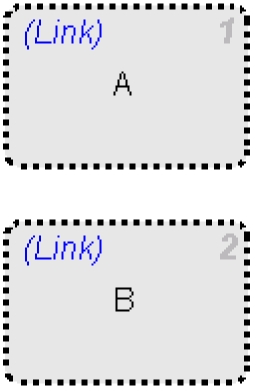
For...	Go to...	And...
Data store memory	**Data Store Memory Mapping**	Specify the data store memory or chart-level local data that corresponds to each atomic subchart symbol.
Parameter data	**Parameter Mapping**	Enter an expression for evaluation in the mask workspace of the main chart.
Input event	**Input Event Mapping**	Specify the chart input event that corresponds to each atomic subchart symbol.

Mapping Input and Output Data for an Atomic Subchart

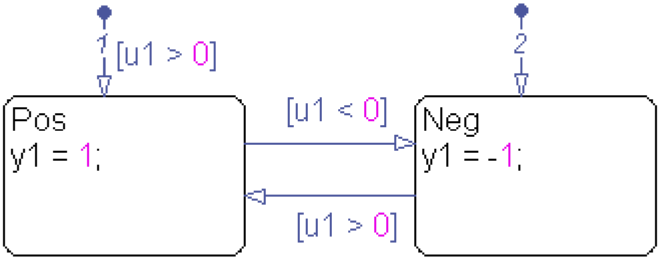
Suppose that you have a model with two Sine Wave blocks that supply input signals to a chart:



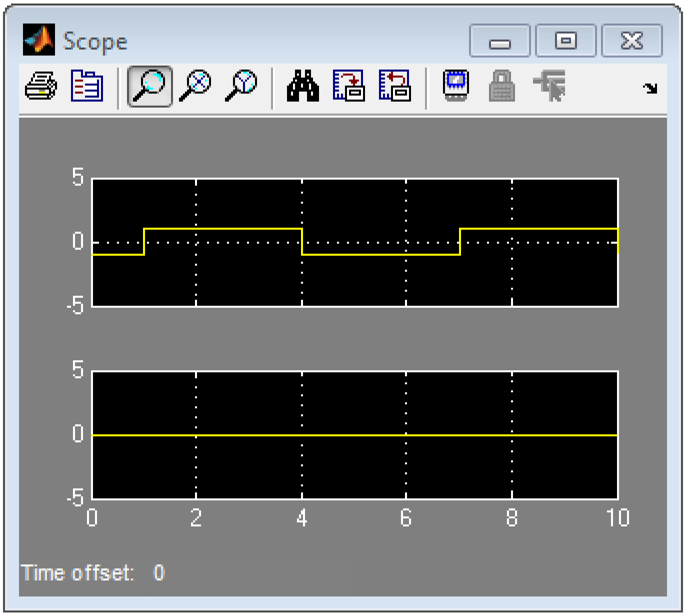
Your chart contains two linked atomic subcharts from the same library:



Both atomic subcharts contain the following objects:



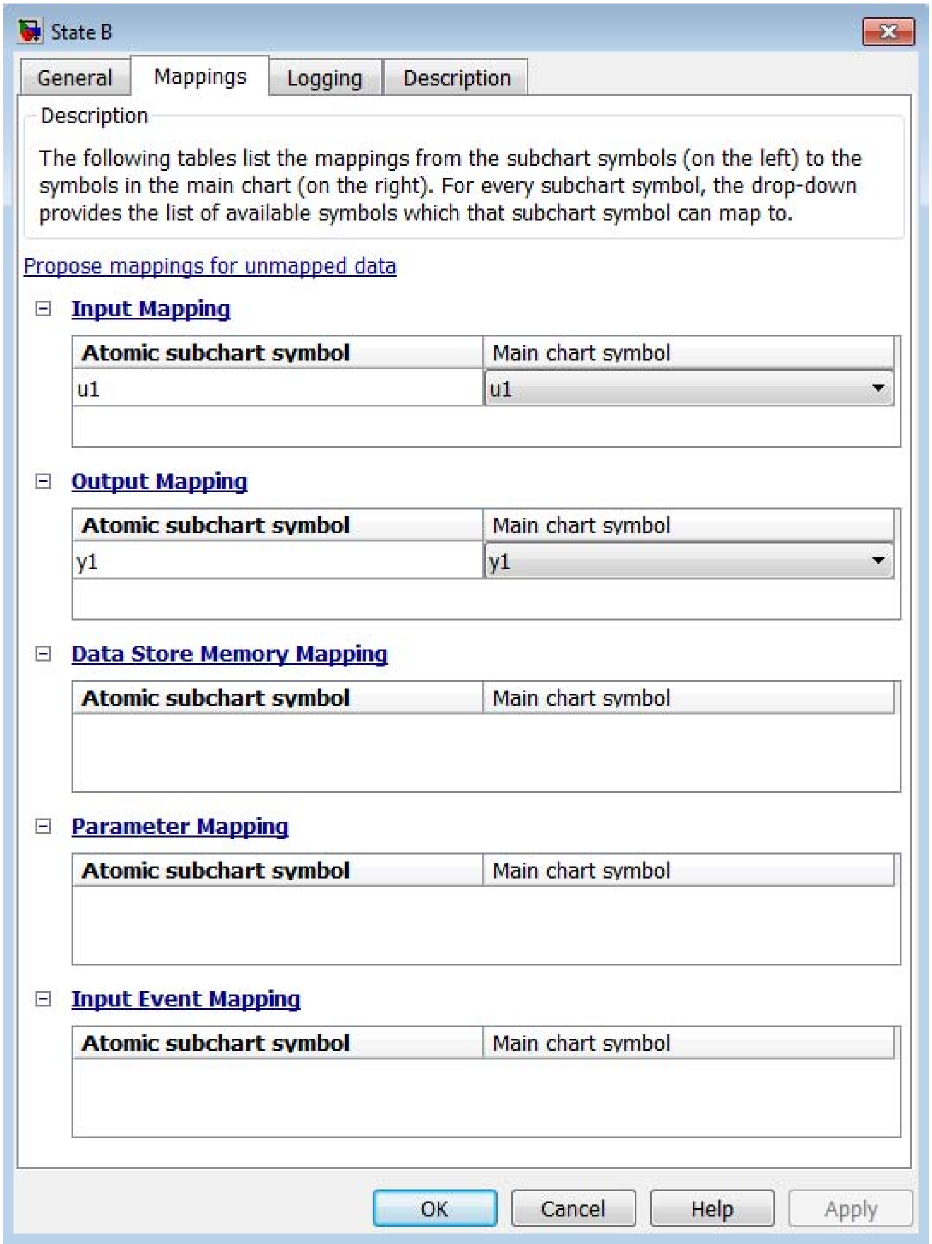
If you simulate the model, the output for `y2` is zero:



Because atomic subchart B uses `u1` and `y1` instead of `u2` and `y2`, you must edit the mapping:

1 Right-click subchart B and select **Properties**.

2 Click the **Mappings** tab in the dialog box that appears.



Description

The following tables list the mappings from the subchart symbols (on the left) to the symbols in the main chart (on the right). For every subchart symbol, the drop-down provides the list of available symbols which that subchart symbol can map to.

Input Mapping

Atomic subchart symbol	Main chart symbol
u1	u1

Output Mapping

Atomic subchart symbol	Main chart symbol
y1	y1

Data Store Memory Mapping

Atomic subchart symbol	Main chart symbol

Parameter Mapping

Atomic subchart symbol	Main chart symbol

Input Event Mapping

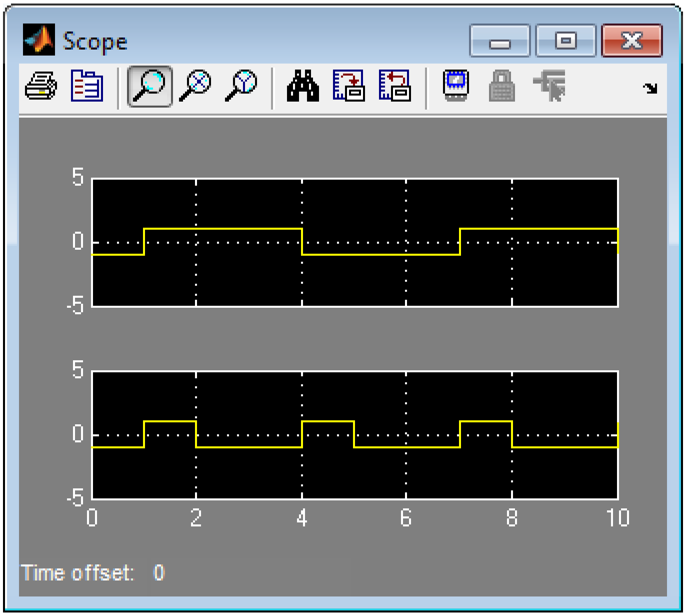
Atomic subchart symbol	Main chart symbol

3 Under **Input Mapping**, specify the main chart symbol for `u1` to be `u2`.

4 Under **Output Mapping**, specify the main chart symbol for `y1` to be `y2`.

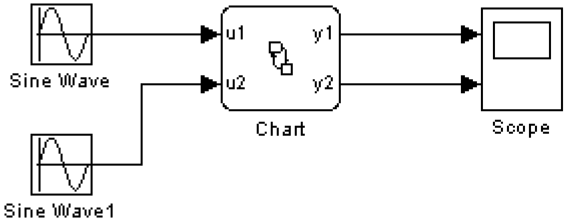
5 Click **OK**.

When you run the model again, you get the following results:

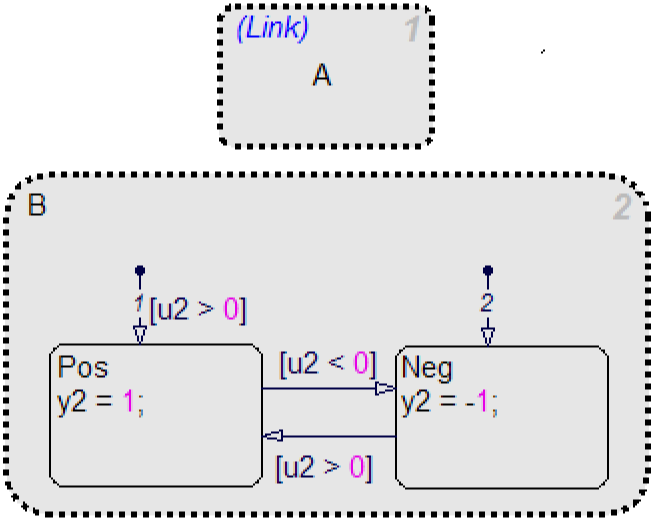


Mapping Data Store Memory for an Atomic Subchart

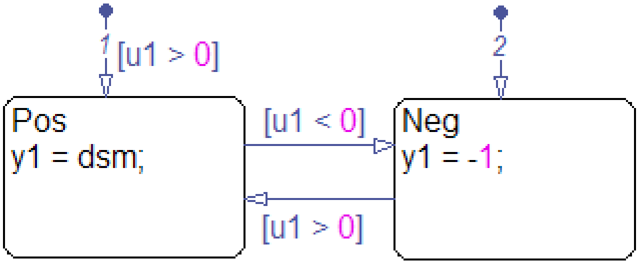
Suppose that you have a model with two Sine Wave blocks that supply input signals to a chart:



Your chart contains a linked atomic subchart from a library:



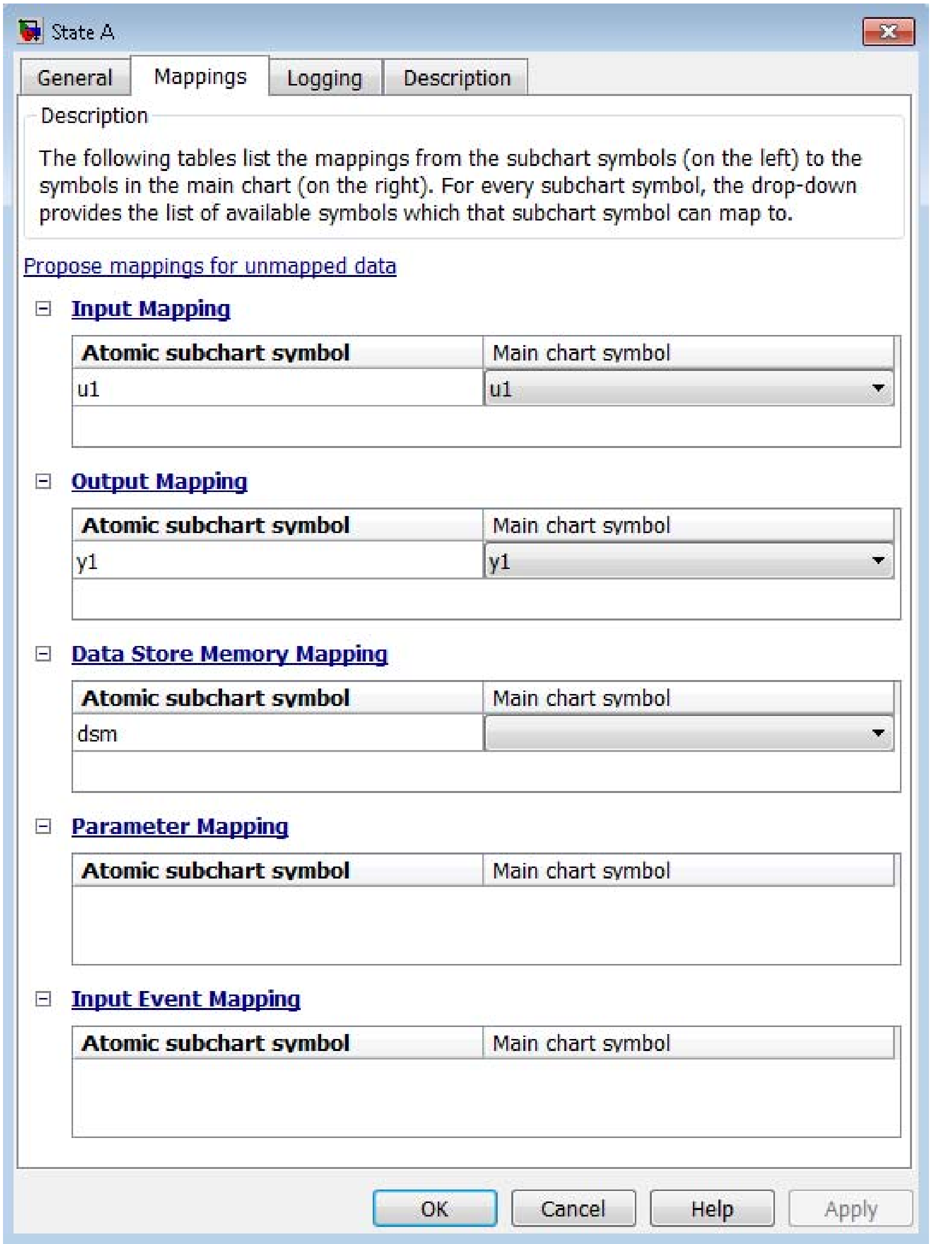
The linked atomic subchart contains the following objects:



If you simulate the model, you get an error because the data store memory, `dsm`, does not map to any variable in the main chart. To fix the mapping for `dsm`:

1 Right-click subchart A and select **Properties**.

2 Click the **Mappings** tab in the dialog box that appears.



Description

The following tables list the mappings from the subchart symbols (on the left) to the symbols in the main chart (on the right). For every subchart symbol, the drop-down provides the list of available symbols which that subchart symbol can map to.

Propose mappings for unmapped data

Input Mapping

Atomic subchart symbol	Main chart symbol
u1	u1

Output Mapping

Atomic subchart symbol	Main chart symbol
y1	y1

Data Store Memory Mapping

Atomic subchart symbol	Main chart symbol
dsm	

Parameter Mapping

Atomic subchart symbol	Main chart symbol

Input Event Mapping

Atomic subchart symbol	Main chart symbol

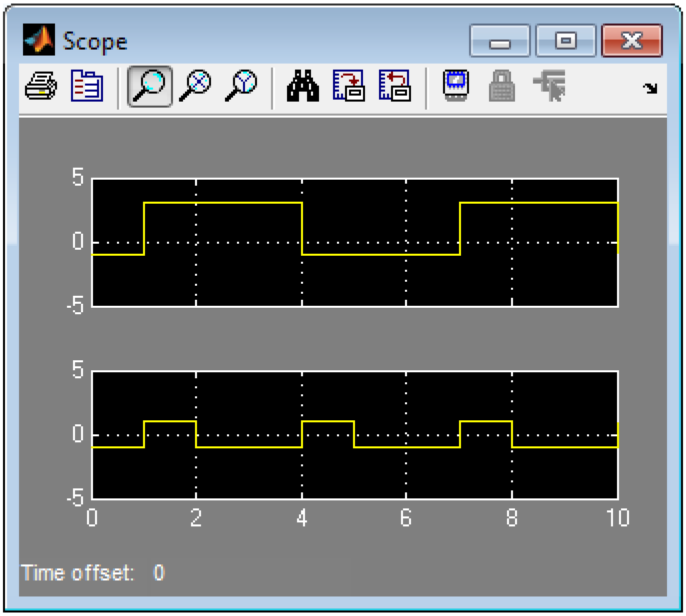
3 Under **Data Store Memory Mapping**, specify the main chart symbol for `dsm` to be `local_for_atomic_subchart`.

Tip You can specify either data store memory or chart-level local data from the main chart. For chart-level local data, the **First index** property must be zero.

---

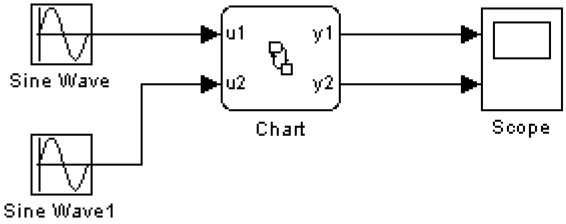
4 Click **OK**.

When you run the model now, you get the following results:

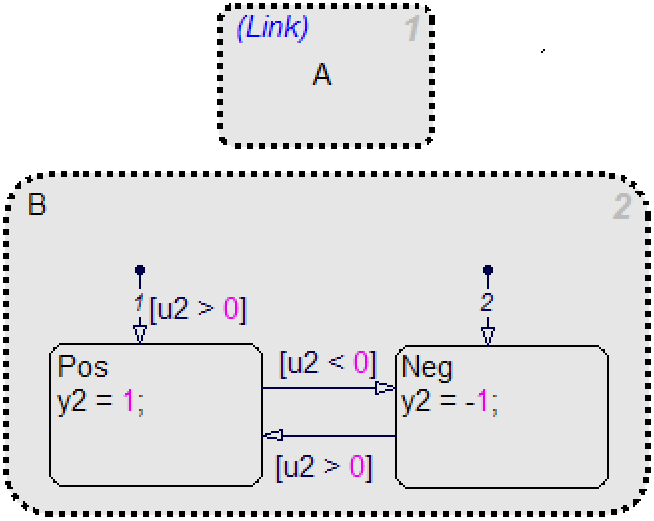


Mapping Parameter Data for an Atomic Subchart

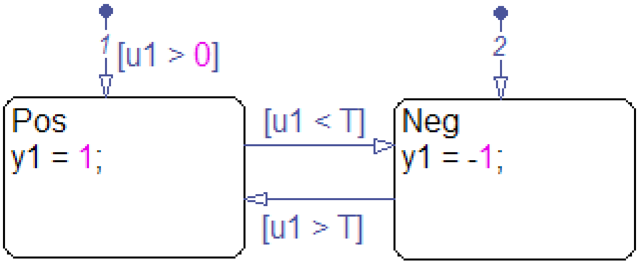
Suppose that you have a model with two Sine Wave blocks that supply input signals to a chart:



Your chart contains a linked atomic subchart from a library:



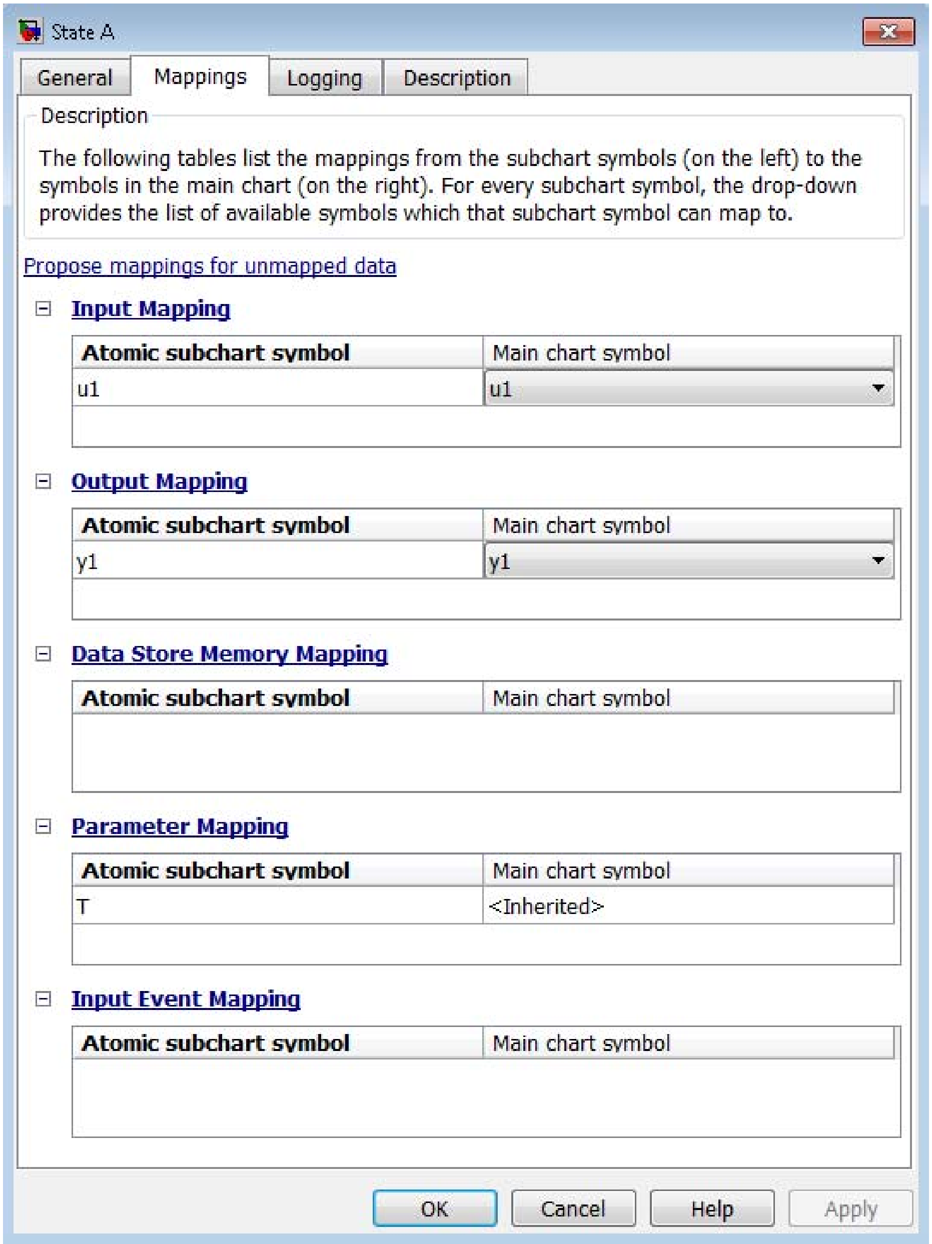
The linked atomic subchart contains the following objects:



If you simulate the model, you get an error because the parameter `T` is undefined. To fix this error, specify an expression for `T` to evaluate in the mask workspace of the main chart:

1 Right-click subchart A and select **Properties**.

2 Click the **Mappings** tab in the dialog box that appears.



Description

The following tables list the mappings from the subchart symbols (on the left) to the symbols in the main chart (on the right). For every subchart symbol, the drop-down provides the list of available symbols which that subchart symbol can map to.

Input Mapping

Atomic subchart symbol	Main chart symbol
u1	u1

Output Mapping

Atomic subchart symbol	Main chart symbol
y1	y1

Data Store Memory Mapping

Atomic subchart symbol	Main chart symbol

Parameter Mapping

Atomic subchart symbol	Main chart symbol
T	<Inherited>

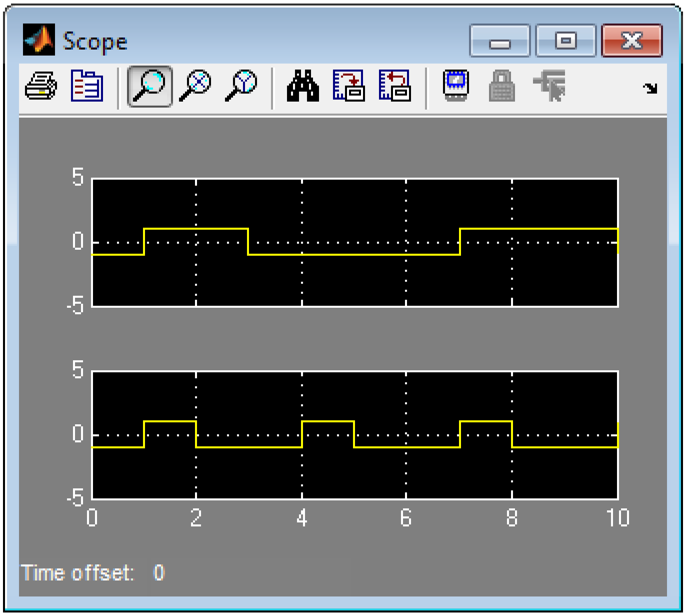
Input Event Mapping

Atomic subchart symbol	Main chart symbol

3 Under **Parameter Mapping**, enter `0.2`.

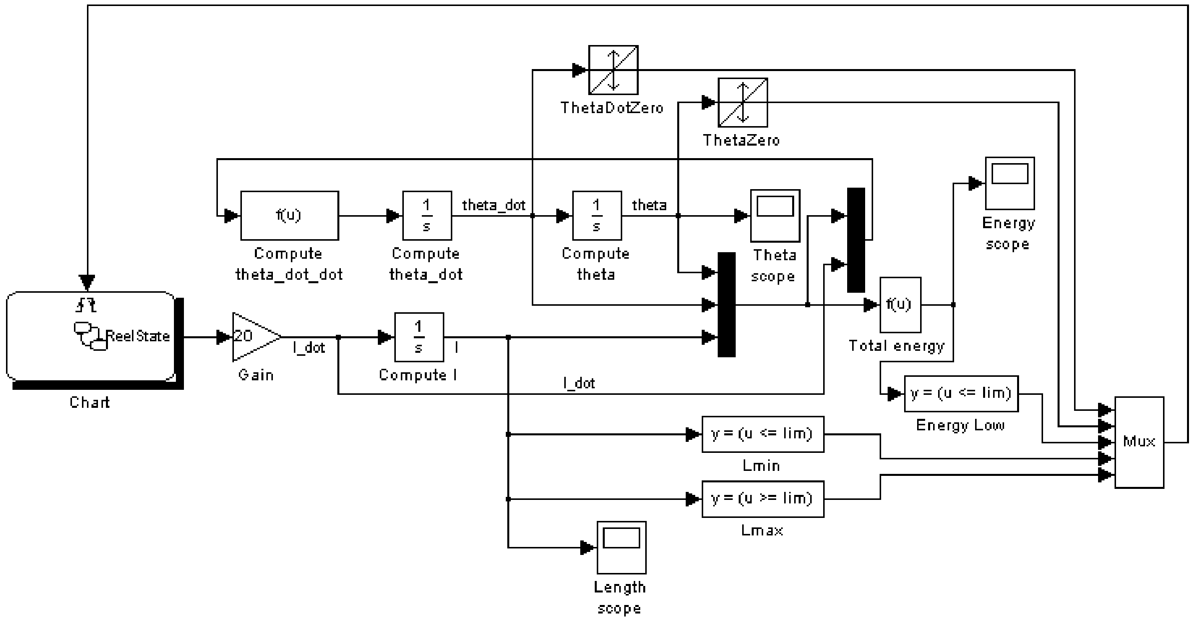
4 Click **OK**.

When you run the model now, you get the following results:

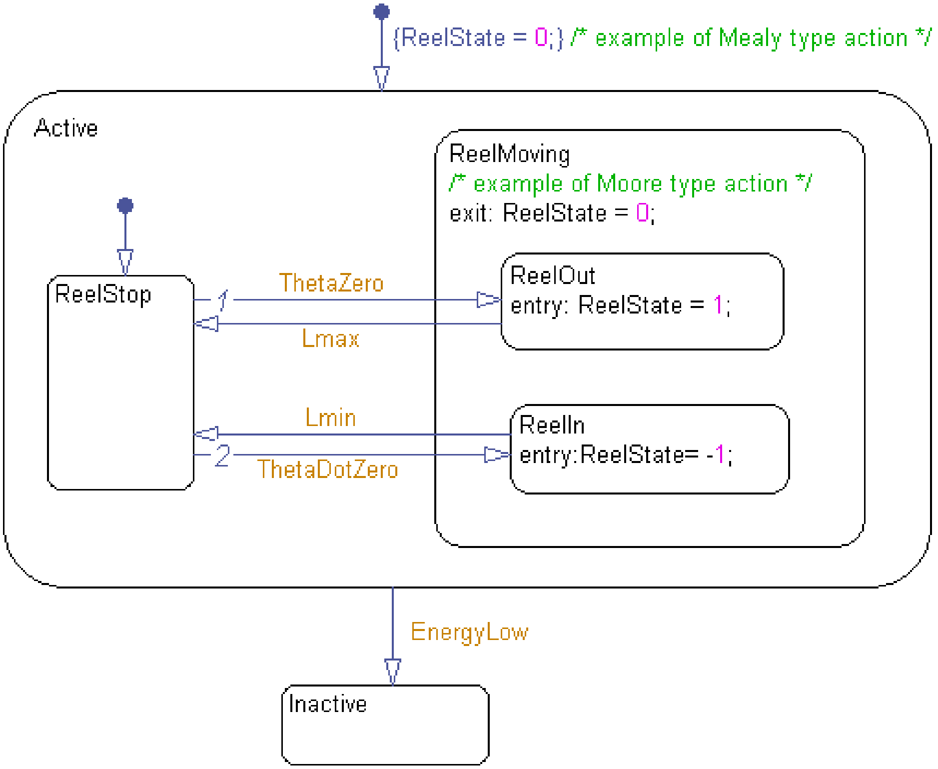


Mapping Input Events for an Atomic Subchart

The `sf_yoyo` model contains a Mux block that supplies input events to a chart:



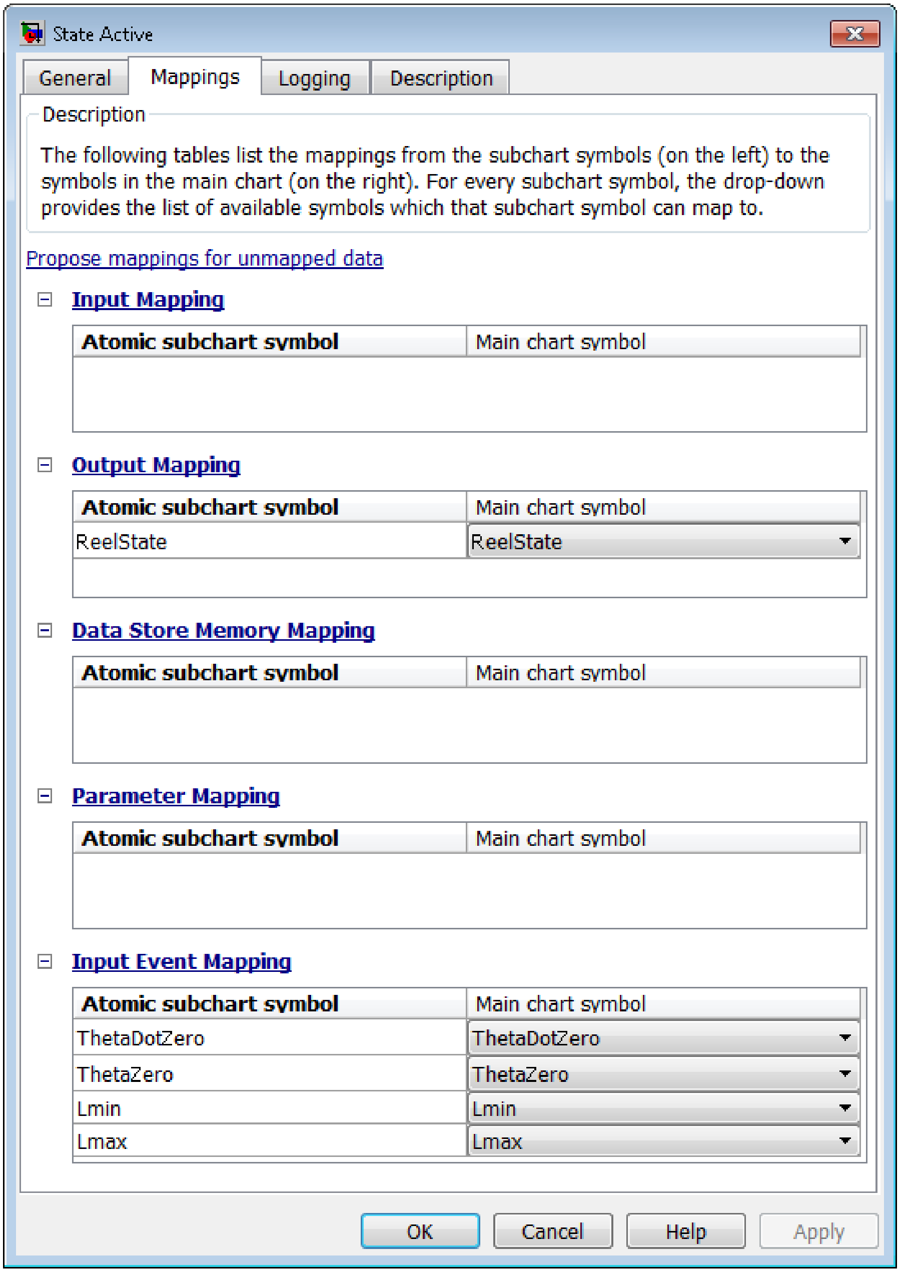
The chart contains two superstates: `Active` and `Inactive`. The `Active` state uses input events to guard transitions between different substates.



To convert the `Active` state to an atomic subchart, follow these steps:

1. Right-click the `Active` state and select **Make Contents > Atomic Subcharted**.
2. Specify the mapping of input events for the atomic subchart.
   a. Right-click the atomic subchart and select **Properties**.

b Click the **Mappings** tab in the dialog box that appears.



Description

The following tables list the mappings from the subchart symbols (on the left) to the symbols in the main chart (on the right). For every subchart symbol, the drop-down provides the list of available symbols which that subchart symbol can map to.

Propose mappings for unmapped data

Input Mapping

Atomic subchart symbol	Main chart symbol

Output Mapping

Atomic subchart symbol	Main chart symbol
ReelState	ReelState

Data Store Memory Mapping

Atomic subchart symbol	Main chart symbol

Parameter Mapping

Atomic subchart symbol	Main chart symbol

Input Event Mapping

Atomic subchart symbol	Main chart symbol
ThetaDotZero	ThetaDotZero
ThetaZero	ThetaZero
Lmin	Lmin
Lmax	Lmax

c Under **Input Event Mapping**, note that each atomic subchart symbol maps to the correct input event in the main chart.

The default mappings also follow the rules of using input events in atomic subcharts. For more information, see “Rules for Using Atomic Subcharts in Stateflow Charts” on page 11-49

d Click **OK**.

Note In this example, the mappings are trivial because each input event in the atomic subchart maps to an input event of the same name in the main chart. For an example of how to use *nontrivial* mapping of input events, see the `sf_elevator` model. A mapping is nontrivial when the variable in the atomic subchart maps to a variable with a different name in the main chart.

At the MATLAB command prompt, enter:

```
showdemo('sf_elevator')
```

In the Elevator System chart, the two linked atomic subcharts use nontrivial mapping of input events.

Generating Reusable Code for Unit Testing

In this section...

How to Generate Reusable Code for Linked Atomic Subcharts

To specify code generation parameters for linked atomic subcharts from the same library:

1. Open the library model that contains your atomic subchart.
2. Unlock the library.
3. Right-click the library chart and select **Subsystem Parameters**.
4. In the dialog box, specify the following parameters:
   a. On the **Main** tab, select **Treat as atomic unit**.
   b. On the **Code Generation** tab, set **Function packaging** to `Reusable function`.
   c. Set **File name options** to `User specified`.
   d. For **File name**, enter the name of the file with no extension.
   e. Click **OK** to apply the changes.
5. (OPTIONAL) Customize the generated function names for atomic subcharts:
   a. Open the Configuration Parameters dialog box.
   b. On the **Code Generation** pane, set **System target file** to `ert.tlc`.
   c. Navigate to the **Code Generation > Symbols** pane.

- **d** For **Subsystem methods**, specify the format of the function names using a combination of the following tokens:
  - `$R` — root model name
  - `$F` — type of interface function for the atomic subchart
  - `$N` — block name
  - `$H` — subsystem index
  - `$M` — mangle string
- **e** Click **OK** to apply the changes.

When you generate code for your model, a separate file stores the code for linked atomic subcharts from the same library.

How to Generate Reusable Code for Unlinked Atomic Subcharts

To specify code generation parameters for an unlinked atomic subchart:

1. In your chart, right-click the atomic subchart and select **Properties**.
2. In the dialog box, specify the following parameters:
   - **a** Set **Code generation function packaging** to `Reusable function`.
   - **b** Set **Code generation file name options** to `User specified`.
   - **c** For **Code generation file name**, enter the name of the file with no extension.
   - **d** Click **OK** to apply the changes.
3. (OPTIONAL) Customize the generated function names for atomic subcharts:
   - **a** Open the Configuration Parameters dialog box.
   - **b** On the **Code Generation** pane, set **System target file** to `ert.tlc`.
   - **c** Navigate to the **Code Generation > Symbols** pane.
   - **d** For **Subsystem methods**, specify the format of the function names using a combination of the following tokens:

- `$R` — root model name
- `$F` — type of interface function for the atomic subchart
- `$N` — block name
- `$H` — subsystem index
- `$M` — mangle string

e Click **OK** to apply the changes.

When you generate code for your model, a separate file stores the code for the atomic subchart. For more information, see “Tutorial: Generating Reusable Code for Unit Testing” on page 11-68.

Reusing Utility Functions Across Multiple Models

In this section...

Rationale for Using Atomic Subcharts

Suppose that you have a library model that contains a set of utility functions for use in multiple charts in a model. The utility functions reside in the library model to enable easier configuration management.

Models that use these utility functions can appear as referenced blocks in a top model. However, when the utility functions are *exported graphical functions* of a Stateflow chart, you can use only one instance of that referenced block per top model. For a complete list of model referencing limitations, see "Limitations on All Model Referencing" in the Simulink documentation.

With atomic subcharts, you can avoid the limitation due to exported graphical functions. You can reuse models with utility functions multiple times as referenced blocks in a top model.

How to Enable Reuse of Utility Functions

To reuse utility functions across multiple models:

1. Create a library model with a chart that contains the utility function you want to reuse.

2. Create a separate model with multiple charts.
   a. In each chart that calls the utility function, add a linked atomic subchart.
   b. Write each call to the utility function using the full path:

      ```
      linked_subchart_name.utility_function_name
      ```

      Using the full path for the function call has the following advantages:

   - Makes clear the dependency on the utility function in the linked atomic subchart
   - Avoids pollution of the global namespace
   - Does not affect efficiency of the generated code

3. Reuse that model multiple times as referenced blocks in a top model.

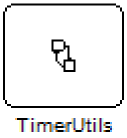
   Because there are no exported graphical functions in the charts, you can use more than one instance of that referenced block in the top model.

Example of Reusing a Timer Function Multiple Times

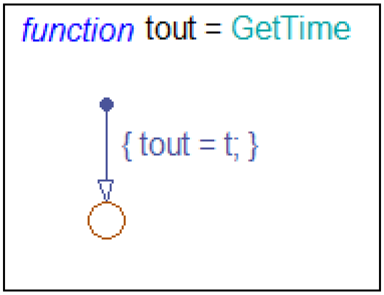
Suppose that you want to reuse a timer function that returns the simulation time. The following procedure shows how you can:

- Call the timer function from multiple locations in a model.
- Reuse that model multiple times in another model.

1. Store the timer function you want to reuse in a library model.
   a. Create a new library named `libTimerUtils`.
   b. Add a chart named `TimerUtils` to the library:



c In your chart, add the following graphical function:



```
function tout = GetTime
{ tout = t; }
```

The utility function `GetTime` returns one output `tout` that corresponds to simulation time `t`. For more information about literal symbols you can use in your chart, see "Supported Symbols in Actions" on page 10-28.

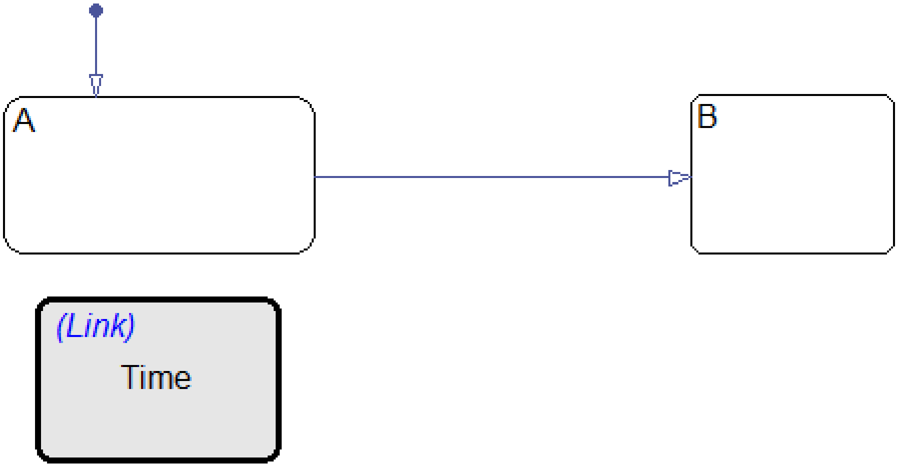
d Save `libTimerUtils`.

2 Develop a separate model with multiple charts that use the timer function.

a Create a new model named `ex_timer_function_calls`.

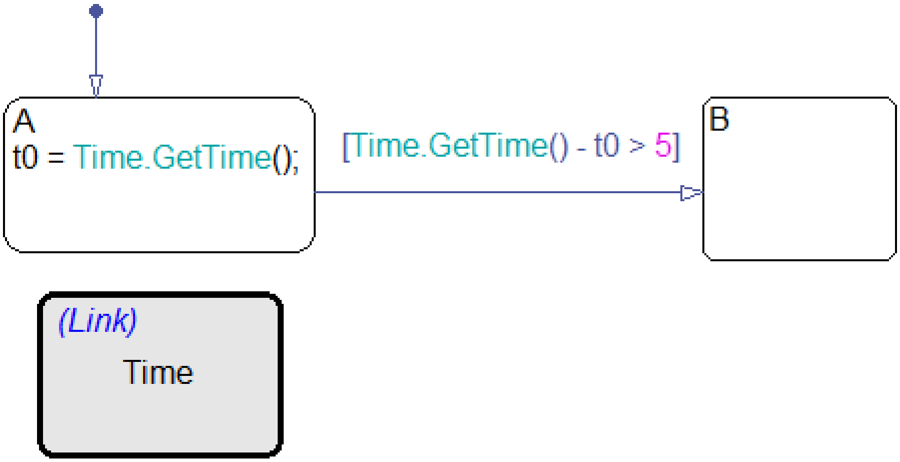
b Add two charts, `Chart1` and `Chart2`, to the model.

c In each chart, add two states, two transitions, and a linked atomic subchart:



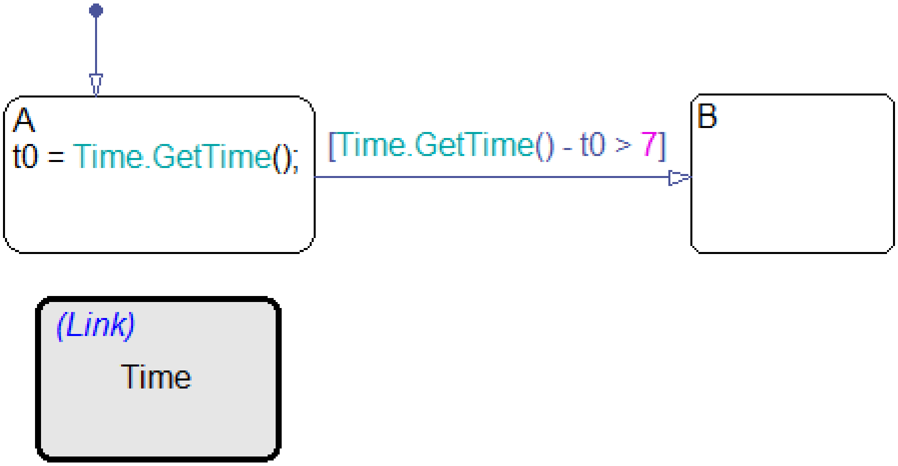
To add the linked atomic subchart, copy the `TimerUtils` library chart and paste it below state A. Name the linked atomic subchart as `Time`.

d In `Chart1`, add the following state action and transition condition:



Upon entry to state A, the call to `GetTime` returns the simulation time. The transition from state A to B occurs when more than 5 seconds of simulation time passes.

e In `Chart2`, add the following state action and transition condition:



Upon entry to state A, the call to `GetTime` returns the simulation time. The transition from state A to B occurs when more than 7 seconds of simulation time passes.

f In each chart, add local data with the following properties:

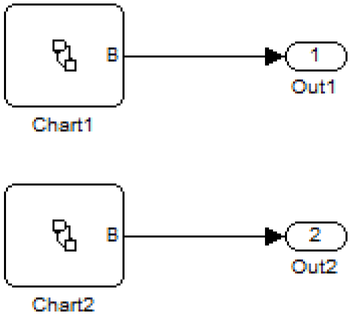
Property	Value
Name	`t0`
Scope	`Local`
Type	`double`

g In each chart, open the State properties dialog box for B and select **Output State Activity**.

This step adds an output data named `B` that is Boolean. The value is 1 when state B is active and 0 otherwise. For more information, see "Outputting State Activity to a Simulink Model" on page 4-23.

h In your model, add two Outport blocks, `Out1` and `Out2`. Then connect each block to the corresponding output of each chart.

Your model should look something like this:



i Configure your model to meet referencing requirements:

  i Open the Configuration Parameters dialog box and navigate to the **Optimization > Signals and Parameters** pane.

  ii Select **Inline parameters**.

  For more information about model referencing requirements, see “Configuration Parameter Requirements” in the Simulink documentation.

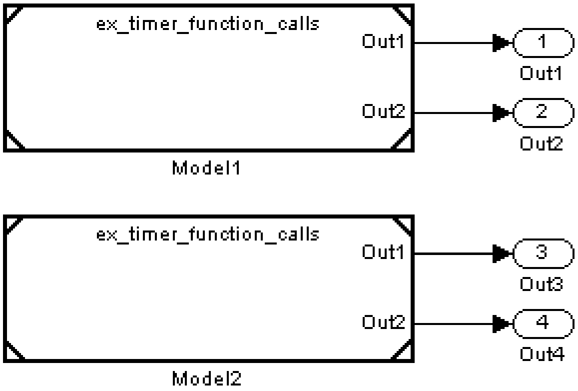
j Save `ex_timer_function_calls`.

3 Reuse the timer function in multiple referenced blocks of a top model.

  a Create a new model named `ex_modelref_utility_functions`.

  b Add two Model blocks that reference `ex_timer_function_calls`.

c Add four Outport blocks and connect them as follows:



d Save `ex_modelref_utility_functions`.

Because the charts in each referenced block do not contain any exported graphical functions, you can reuse the timer function from the `libTimerUtils` library as many times as you want. For more information about model referencing, see “Referencing a Model” in the Simulink documentation.

Rules for Using Atomic Subcharts in Stateflow Charts

Define data in an atomic subchart explicitly

Be sure to define data that appears in an atomic subchart explicitly in the main chart. For instructions on how to define data in a chart, see “Adding Data Using the Model Explorer” on page 8-3.

Map variables of linked atomic subcharts

When you use linked atomic subcharts, map the variables so that data in the subchart correspond to the correct data in the main chart. For more information, see “Mapping Variables for Atomic Subcharts” on page 11-20.

Match size, type, and complexity of variables in linked atomic subcharts

Verify that the size, type, and complexity of variables in a subchart match the settings of the corresponding variables in the main chart. For more information, see “Mapping Variables for Atomic Subcharts” on page 11-20.

Export chart-level graphical functions if called from an atomic subchart

If your atomic subchart contains a function call to a chart-level graphical function, export that function. In the Chart properties dialog box, select **Export Chart Level Graphical Functions (Make Global)**. For more information, see “Exporting Chart-Level Graphical Functions” on page 7-39.

Do not mix edge-triggered and function-call input events in the same atomic subchart

Input events in an atomic subchart must all use edge-triggered type, or they must all use function-call type. This restriction is consistent with the behavior for the container chart. For more information, see “Best Practices for Using Events in Stateflow Charts” on page 9-47.

Do not map multiple input events in an atomic subchart to the same input event in the container chart

Each input event in an atomic subchart must map to a unique input event in the container chart. You can verify unique mappings of input events by opening the properties dialog box for the atomic subchart and checking the **Input Event Mapping** section of the **Mappings** tab.

Match the trigger type when mapping input events

Each input event in an atomic subchart must map to an input event of the same trigger type in the container chart.

Do not use atomic subcharts in continuous-time Stateflow charts

Continuous-time charts do not support atomic subcharts.

Do not use Moore charts as atomic subcharts

Moore charts do not have the same simulation behavior as Classic Stateflow charts with the same constructs.

Do not use outgoing transitions when an atomic subchart uses top-level local events

You cannot use outgoing transitions from an atomic subchart that uses local events at the top level of the subchart. Using this configuration causes a simulation error.

Avoid using execute-at-initialization with atomic subcharts

You get a warning when the following conditions are true:

- The chart property **Execute (enter) Chart At Initialization** is enabled.
- The default transition path of the chart reaches an atomic subchart.

If an `entry` action inside the atomic subchart requires access to a chart input or data store memory, you might get inaccurate results. To avoid this warning, you can disable **Execute (enter) Chart At Initialization** or redirect the default transition path away from the atomic subchart.

For more information about execute-at-initialization behavior, see "Execution of a Chart at Initialization" on page 3-49.

Avoid using the names of subsystem parameters in atomic subcharts

If a parameter in an atomic subchart matches the name of a Simulink built-in subsystem parameter, the only mapping allowed for that parameter is `Inherited`. Specifying any other parameter mapping in the **Mappings** tab of the properties dialog box causes an error. You can, however, change the parameter value at the MATLAB prompt so that all instances of that parameter have the same value.

To get a list of Simulink subsystem parameters, enter:

```
param_list = sort(fieldnames(get_param('built-in/subsystem', 'ObjectParameters')));
```

Restrict use of machine-parented data

If your chart contains atomic subcharts, do not use machine-parented data with the following properties:

- Imported or exported
- Is 2-D or higher, or uses fixed-point type

Machine-parented data with these properties prevent reuse of generated code and other code optimizations.

Use Dataset format for signal logging in atomic subcharts

If you use `ModelDataLogs` format to log signal data of an atomic subchart, an error occurs. To avoid this error, you can use one of the following workarounds:

- Disable signal logging for the atomic subchart.

  On the **Logging** tab of the properties dialog box for the atomic subchart, clear the **Log signal data** check box. This change disables logging of atomic subchart activity, but does not affect logging of any signals *inside* the atomic subchart.

- Change the signal logging format for your model from `ModelDataLogs` to `Dataset`.

  On the **Data Import/Export** pane of the Configuration Parameters dialog box, set **Signal logging format** to `Dataset`. This change affects the logging format for your entire model.

For more information about signal logging in Stateflow charts, see “Logging Data Values and State Activity” on page 26-55.

Do not change the first index of local data to a nonzero value

When a data store memory in an atomic subchart maps to chart-level local data, the **First index** property of the local data must remain zero. If you change **First index** to a nonzero value, an error occurs when you try to update the diagram.

Use consistent settings for super-step semantics

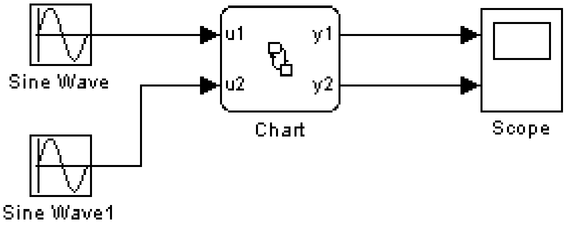
When you use linked atomic subcharts, verify that your settings for super-step semantics match the settings in the main chart. For more information, see “Execution of a Chart with Super Step Semantics” on page 3-41.

Tutorial: Reusing a State Multiple Times in a Chart

In this section...

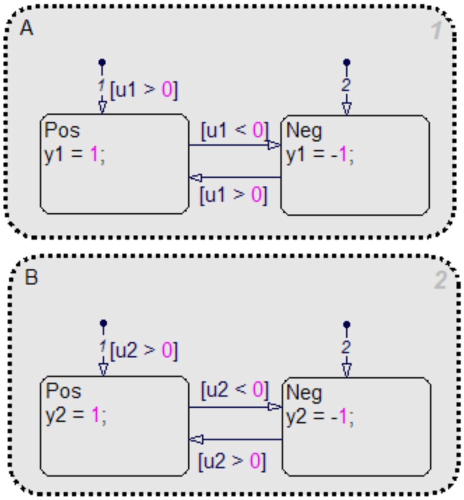
Goal of the Tutorial

Assume that you have the following model:

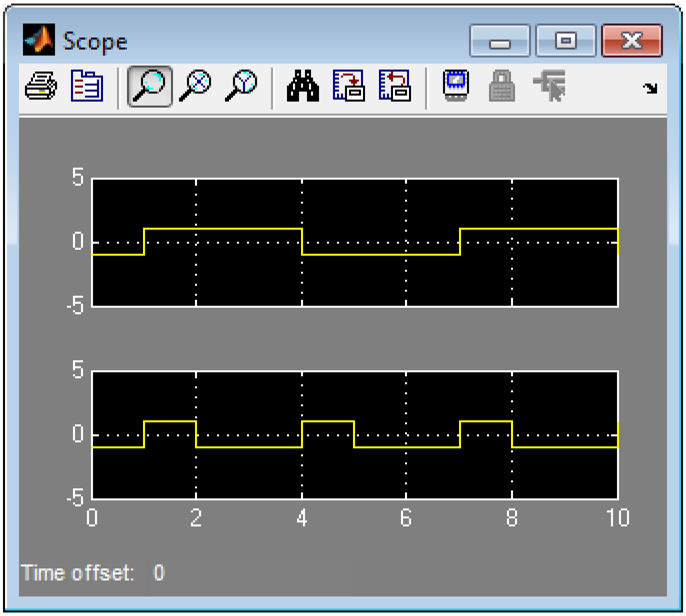


The top Sine Wave block uses a frequency of 1 radian per second, and the bottom Sine Wave block uses a frequency of 2 radians per second. The blocks use the same amplitude (1) and phase shift (0).

In the chart, each state uses saturator logic to convert the input sine wave to an output square wave of the same frequency. The states perform the same actions and differ only in the names of input and output data:



When you run the model, you get the following results:



Suppose that you want to reuse the contents of state A in the chart. You can convert that state to an atomic subchart and then use multiple linked instances of that subchart in your chart.

Editing a Model to Use Atomic Subcharts

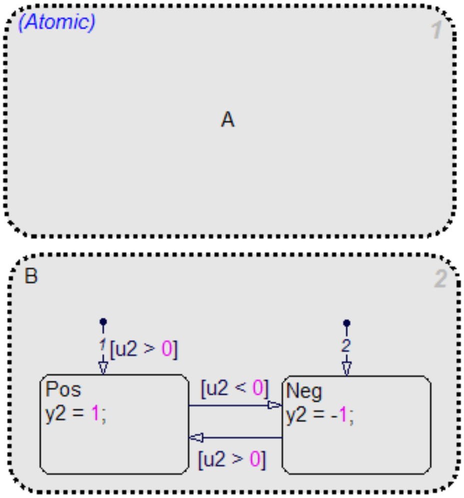
The sections that follow describe how to replace states in your chart with atomic subcharts. This procedure enables reuse of the same object in your model while retaining the same simulation results.

Step	Task	Reference
1	Change one of the states into an atomic subchart.	"Converting a State to an Atomic Subchart" on page 11-56
2	Create a library that contains this atomic subchart.	"Creating a Library for the Atomic Subchart" on page 11-56

Step	Task	Reference
3	Replace the states in your chart with linked atomic subcharts.	"Replacing States with Linked Atomic Subcharts" on page 11-57
4	Edit the mapping of input and output variables where necessary.	"Editing the Mapping of Input and Output Variables" on page 11-59

Converting a State to an Atomic Subchart

To convert state A to an atomic subchart, right-click the state and select **Make Contents > Atomic Subcharted**. State A changes to an atomic subchart:



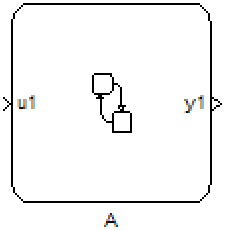
Creating a Library for the Atomic Subchart

To enable reuse of the atomic subchart you created in "Converting a State to an Atomic Subchart" on page 11-56, store the atomic subchart in a library:

1 Create a new library model.

2 Copy the atomic subchart and paste in your library.

The atomic subchart appears as a standalone chart with an input and an output. This standalone property enables you to reuse the contents of the atomic subchart.



Note You cannot drag and drop the atomic subchart into your library model. Only a copy-and-paste operation works.

---

3 Save your library model.

Replacing States with Linked Atomic Subcharts

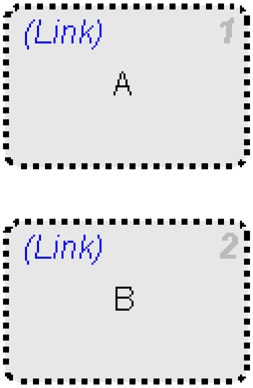
To replace the states in your chart with linked atomic subcharts:

1 Delete both states from the chart.

2 Copy the atomic subchart in your library and paste in your chart twice.

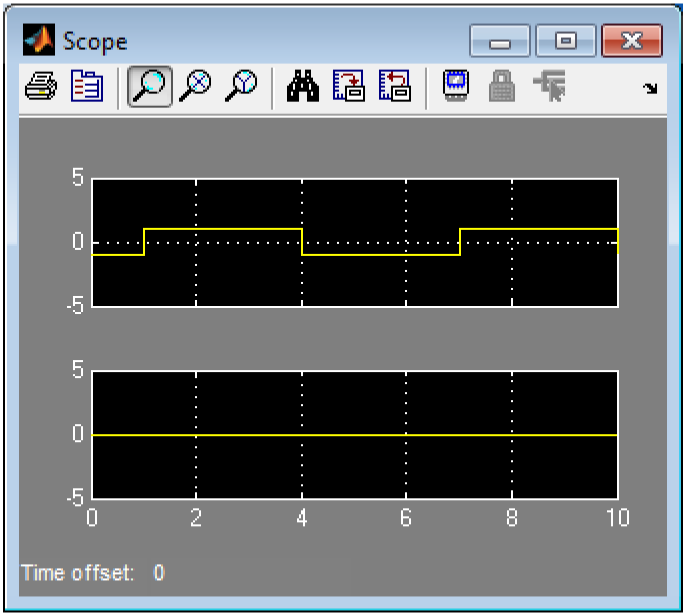
3 Rename the second instance as B.

Each linked atomic subchart appears opaque and contains the label **(Link)** in the upper-left corner.



Editing the Mapping of Input and Output Variables

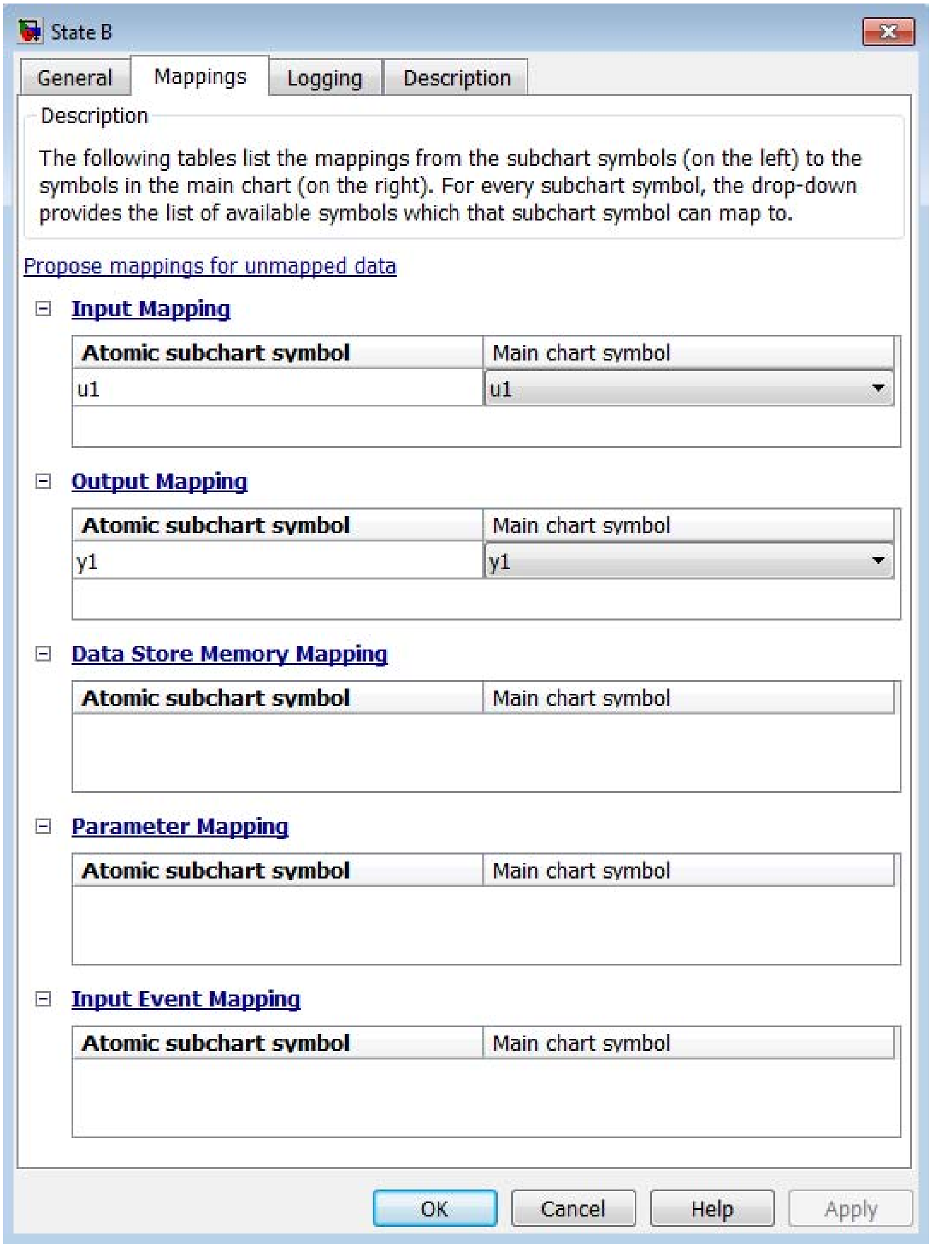
If you simulate the model now, the output for `y2` is zero:



You also see warnings about unused data in the Simulation Diagnostics Viewer. These warnings appear because atomic subchart B uses `u1` and `y1` instead of `u2` and `y2`. To fix these warnings, you must edit the mapping of input and output variables:

1. Open the properties dialog box for B.

2 Click the **Mappings** tab.



Description

The following tables list the mappings from the subchart symbols (on the left) to the symbols in the main chart (on the right). For every subchart symbol, the drop-down provides the list of available symbols which that subchart symbol can map to.

Input Mapping

Atomic subchart symbol	Main chart symbol
u1	u1

Output Mapping

Atomic subchart symbol	Main chart symbol
y1	y1

Data Store Memory Mapping

Atomic subchart symbol	Main chart symbol

Parameter Mapping

Atomic subchart symbol	Main chart symbol

Input Event Mapping

Atomic subchart symbol	Main chart symbol

3. Under **Input Mapping**, select `u2` from the drop-down list.

   The input variable in your atomic subchart now maps to the correct input variable in the main chart.

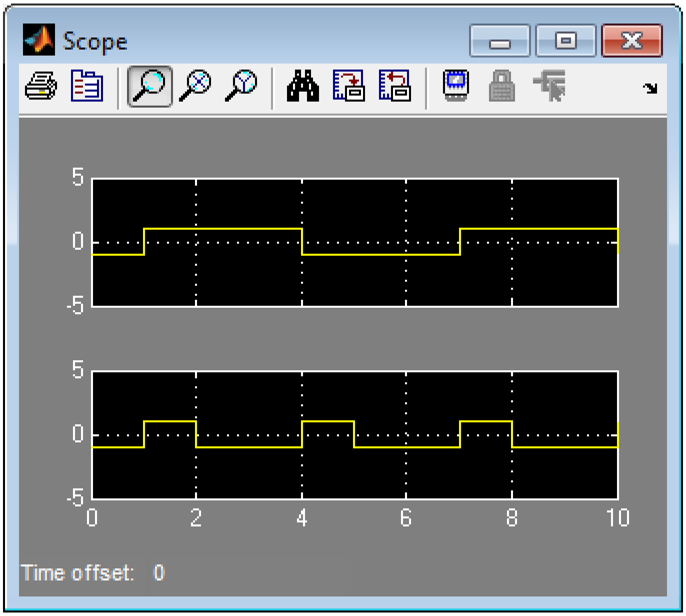
4. Under **Output Mapping**, select `y2` from the drop-down list.

   The output variable in your atomic subchart now maps to the correct output variable in the main chart.

5. Click **OK**.

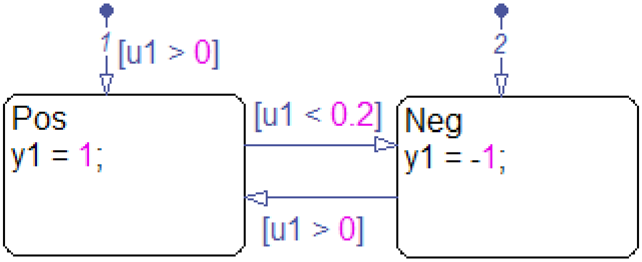
Running the New Model

When you simulate the new model, the results match those of the original design.



Propagating a Change in the Library Chart

Suppose that you edit the transition from `Pos` to `Neg` in the library chart:



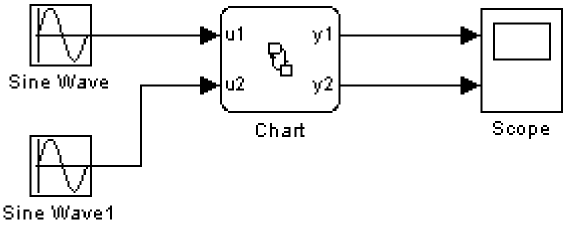
This change propagates to all linked atomic subcharts in your main chart. You do not have to update each state individually.

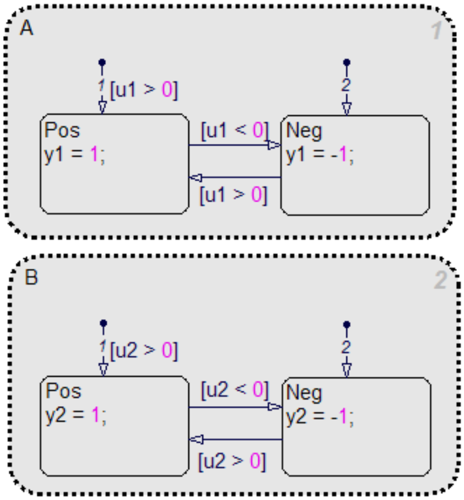
Tutorial: Reducing the Compilation Time of a Chart

In this section...
"Goal of the Tutorial" on page 11-63
"Editing a Model to Use Atomic Subcharts" on page 11-64

Goal of the Tutorial

Assume that you have the following model, and the chart has two states:





Suppose that you want to reduce the compilation time of the chart for simulation. You can convert state A to an atomic subchart. Then you can make changes, one by one, to state A and see how each change affects simulation results. Making one change requires recompilation of only the atomic subchart and not the entire chart.

Editing a Model to Use Atomic Subcharts

1. Right-click state A and select **Make Contents > Atomic Subcharted**.
2. Double-click the atomic subchart.

   The contents of the subchart appear in a separate window.
3. Start simulation.

   Side-by-side animation for the main chart and the atomic subchart occurs.
4. In the atomic subchart, change the state action for `Pos` to `y1 = 2`.
5. Restart simulation.

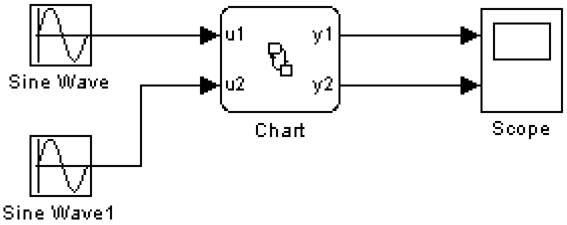
   Recompilation occurs only for the atomic subchart and not the entire chart.

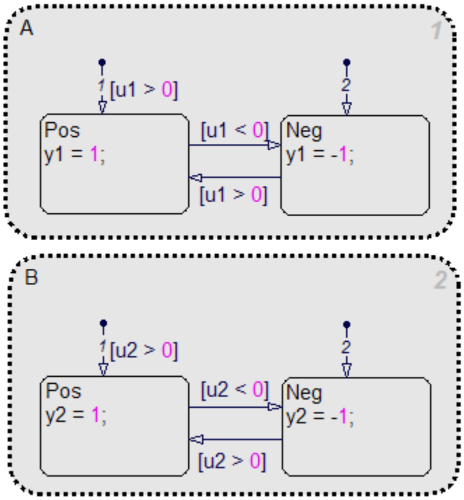
Tutorial: Dividing a Chart into Separate Units for Editing

In this section...
"Goal of the Tutorial" on page 11-65
"Editing a Model to Use Atomic Subcharts" on page 11-66

Goal of the Tutorial

Assume that you have the following model, and the chart has two states:

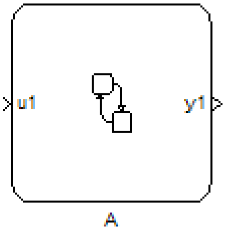




Suppose that you want to edit state A separately, while someone else is editing state B. You can convert state A to an atomic subchart for storage in a library model. After replacing state A with a linked atomic subchart, you can make changes separately in the library. These changes propagate automatically to the chart that contains the linked atomic subchart.

Editing a Model to Use Atomic Subcharts

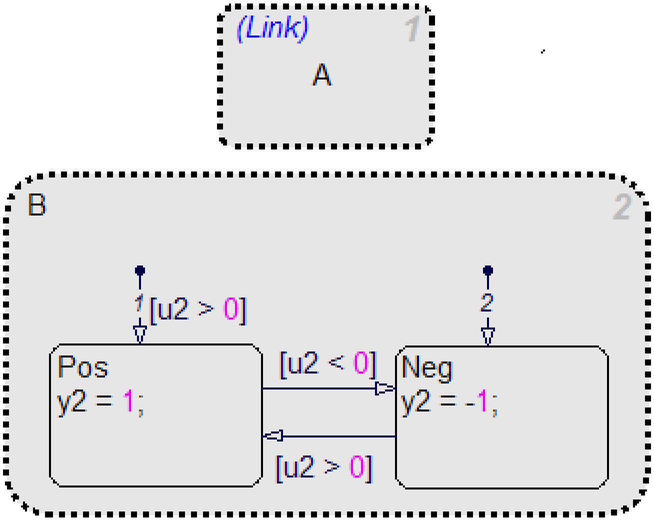
1. Right-click state A and select **Make Contents > Atomic Subcharted**.
2. Create a new library model.
3. Copy the atomic subchart and paste in your library.



Note You cannot drag and drop the atomic subchart into your library model. Only a copy-and-paste operation works.

4. Save your library model.
5. In your main chart, delete state A.

6 Copy the atomic subchart in your library and paste in your main chart.

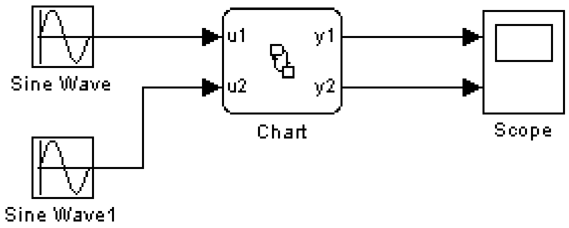


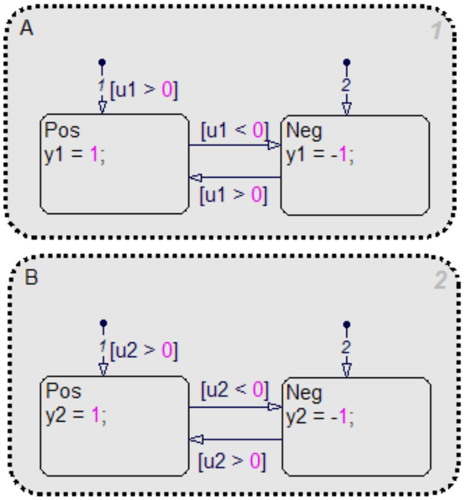
You can now edit state A separately from state B without any merge issues.

Tutorial: Generating Reusable Code for Unit Testing

Goal of the Tutorial

Assume that you have the following model, and the chart has two states:

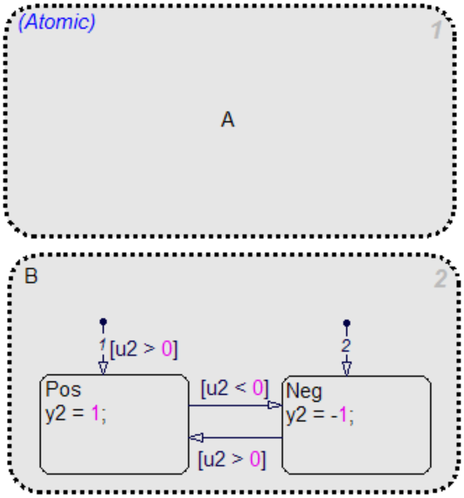




Suppose that you want to generate reusable code so that you can perform unit testing on state A. You can convert that part of the chart to an atomic subchart and then specify a separate file to store the Simulink Coder generated code.

Converting a State to an Atomic Subchart

To convert state A to an atomic subchart, right-click the state and select **Make Contents > Atomic Subcharted**. State A changes to an atomic subchart:



Specifying Code Generation Parameters

Setting Up a Standalone C File for the Atomic Subchart

1. Open the properties dialog box for A.
2. Set **Code generation function packaging** to `Reusable function`.
3. Set **Code generation file name options** to `User specified`.
4. For **Code generation file name**, enter `saturator` as the name of the file.
5. Click **OK**.

Setting Up the Code Generation Report

1. Open the Configuration Parameters dialog box.

2. In the **Code Generation** pane, set **System target file** to `ert.tlc`.

3. In the **Code Generation > Report** pane, select **Create code generation report**.

   This step automatically selects **Open report automatically** and **Code-to-model**.

4. Select **Model-to-code**.

5. Click **Apply**.

Customizing the Generated Function Names

1. In the Configuration Parameters dialog box, go to the **Code Generation > Symbols** pane.

2. Set **Subsystem methods** to the format string `$R$N$M$F`, where:

   - `$R` is the root model name.
   - `$N` is the block name.
   - `$M` is the mangle string.
   - `$F` is the type of interface function for the atomic subchart.

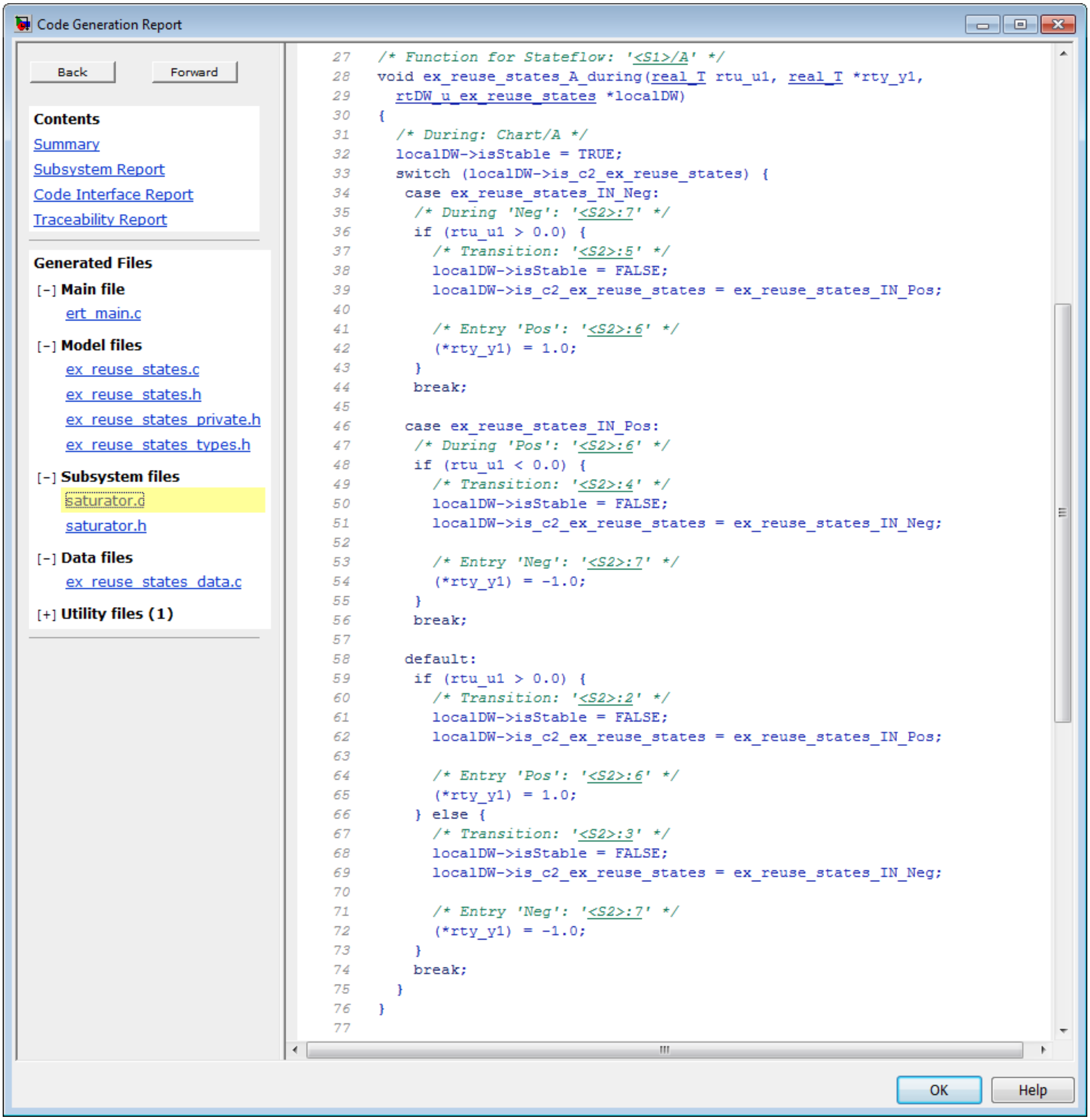
   For more information, see "Subsystem methods" in the Simulink Coder documentation.

3. Click **Apply**.

Generating Code for Only the Atomic Subchart

To generate code for your model, click **Build** on the **Code Generation** pane of the Configuration Parameters dialog box. In the code generation report that appears, you see a separate file that contains the generated code for the atomic subchart.

To inspect the code for `saturator.c`, click the hyperlink in the report to see the following code:



```
/* Function for Stateflow: '<S1>/A' */
void ex_reuse_states_A_during(real_T rtu_u1, real_T *rty_y1,
  rtDW_u_ex_reuse_states *localDW)
{
  /* During: Chart/A */
  localDW->isStable = TRUE;
  switch (localDW->is_c2_ex_reuse_states) {
   case ex_reuse_states_IN_Neg:
    /* During 'Neg': '<S2>:7' */
    if (rtu_u1 > 0.0) {
      /* Transition: '<S2>:5' */
      localDW->isStable = FALSE;
      localDW->is_c2_ex_reuse_states = ex_reuse_states_IN_Pos;

      /* Entry 'Pos': '<S2>:6' */
      (*rty_y1) = 1.0;
    }
    break;

   case ex_reuse_states_IN_Pos:
    /* During 'Pos': '<S2>:6' */
    if (rtu_u1 < 0.0) {
      /* Transition: '<S2>:4' */
      localDW->isStable = FALSE;
      localDW->is_c2_ex_reuse_states = ex_reuse_states_IN_Neg;

      /* Entry 'Neg': '<S2>:7' */
      (*rty_y1) = -1.0;
    }
    break;

   default:
    if (rtu_u1 > 0.0) {
      /* Transition: '<S2>:2' */
      localDW->isStable = FALSE;
      localDW->is_c2_ex_reuse_states = ex_reuse_states_IN_Pos;

      /* Entry 'Pos': '<S2>:6' */
      (*rty_y1) = 1.0;
    } else {
      /* Transition: '<S2>:3' */
      localDW->isStable = FALSE;
      localDW->is_c2_ex_reuse_states = ex_reuse_states_IN_Neg;

      /* Entry 'Neg': '<S2>:7' */
      (*rty_y1) = -1.0;
    }
    break;
  }
}
```

Line 28 shows that the `during` function generated for the atomic subchart has the name `ex_reuse_states_A_during`. This name follows the format string `$R$N$M$F` specified for **Subsystem methods**:

- `$R` is the root model name, `ex_reuse_states`.
- `$N` is the block name, `A`.
- `$M` is the mangle string, which is empty.
- `$F` is the type of interface function for the atomic subchart, `during`.

Note The line numbers shown can differ from the numbers that appear in your code generation report.

12

Saving and Restoring Simulations with SimState

What Is a SimState?

A SimState is the snapshot of the state of a model at a specific time during simulation. For a Stateflow chart, a SimState includes the following information:

- Activity of chart states
- Values of chart local data
- Values of chart output data
- Values of persistent data in MATLAB functions and Truth Table blocks

A SimState lists chart objects in hierarchical order:

- Graphical objects grouped by type (box, function, or state) and in alphabetical order within each group
- Chart data grouped by scope (block output or local) and in alphabetical order within each group

For example, the following SimState illustrates the hierarchical structure of chart objects.

```
c =

  Block:    "shift_logic"    (handle)    (active)
  Path:     sf_car/shift_logic

  Contains:

    + gear_state         "State (AND)"         (active)
    + selection_state    "State (AND)"         (active)
      gear               "Block output data"   double [1, 1]
      down_th            "Local scope data"    double [1, 1]
      up_th              "Local scope data"    double [1, 1]
```

The tree structure maps graphical and nongraphical objects to their respective locations in the chart hierarchy. If name conflicts exist, one or more underscores appear at the end of a name so that all objects have unique identifiers in the SimState hierarchy.

Note Stateless flow charts have an empty SimState, because they do not contain states or persistent data.

For information about using a SimState for other blocks in a Simulink model, see "Saving and Restoring the Simulation State as the SimState" in the *Simulink User's Guide*.

Benefits of Using a Snapshot of the Simulation State

In this section...

Division of a Long Simulation into Segments

You can save the complete simulation state of a model at any time during a long simulation. Then you can load that simulation state and run specific segments of that simulation without starting from time t = 0, which saves time.

For directions, see "Dividing a Long Simulation into Segments" on page 12-5.

Test of a Chart Response to Different Settings

You can load and modify the simulation state of a chart to test the response to different settings. You can change the value of chart local or output data midway through a simulation or change state activity and then test how a chart responds.

Loading and modifying the simulation state provides these benefits:

- Enables testing of a hard-to-reach chart configuration by loading a specific simulation state, which promotes thorough testing
- Enables testing of the same chart configuration with different settings, which promotes reuse of a simulation state

For directions, see:

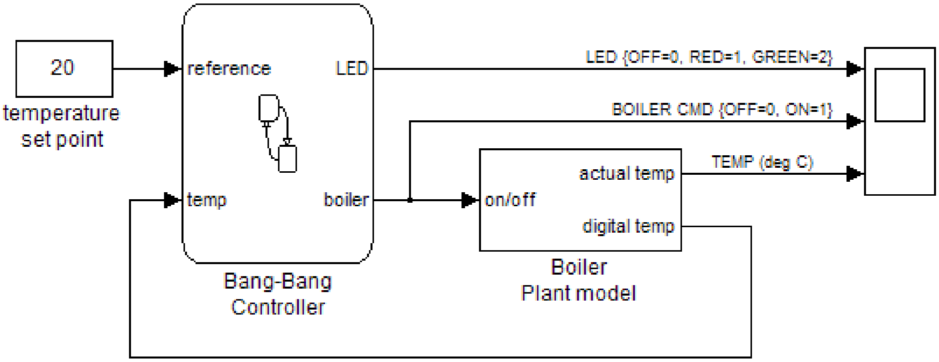
- "Testing a Unique Chart Configuration" on page 12-10
- "Testing a Chart with Fault Detection and Redundant Logic" on page 12-21

Dividing a Long Simulation into Segments

In this section...

Goal of the Tutorial

Suppose that you want to simulate the `sf_boiler` model without starting from t = 0.



This model simulates for 1400 seconds, but the output that interests you occurs sometime between t = 400 and 600. You can simulate the model, save the SimState at time t = 400, and then load that SimState for simulation between t = 400 and 600.

Step	Task	Reference
1	Define the SimState for your chart.	“Defining the SimState” on page 12-6
2	Load the SimState for your chart.	“Loading the SimState” on page 12-7
3	Simulate the specific segment.	“Simulating the Specific Segment” on page 12-9

Defining the SimState

1. Open the model.

   Type `sf_boiler` at the command prompt.

2. Enable saving of a SimState.

   a. Open the Configuration Parameters dialog box and go to the **Data Import/Export** pane.
   b. Select the **Final states** check box.
   c. Enter a name, such as `sf_boiler_ctx01`.

      You can choose any alphanumeric string for the name.
   d. Select the **Save complete SimState in final state** check box.
   e. Click **Apply**.

   Programmatic equivalent

   You can programmatically enable saving of a SimState:

```
set_param('sf_boiler','SaveFinalState','on', ...
'FinalStateName', ['sf_boiler_ctx01'], ...
'SaveCompleteFinalSimState','on');
```

   For details about setting model parameters, see `set_param` in the *Simulink Reference*.

3. Define the start and stop times for this simulation segment.

- **a** In the Configuration Parameters dialog box, go to the **Solver** pane.
- **b** For **Start time**, enter `0`.
- **c** For **Stop time**, enter `400`.
- **d** Click **OK**.

Programmatic equivalent

You can programmatically set the start and stop times:

```
set_param('sf_boiler','StartTime','0', ...
'StopTime','400');
```

4 Start simulation.

When you simulate the model, you save the complete simulation state at t = 400 in the variable `sf_boiler_ctx01` in the MATLAB base workspace.

5 Disable saving of a SimState.

This step prevents you from overwriting the SimState you saved in the previous step.

- **a** Open the Configuration Parameters dialog box and go to the **Data Import/Export** pane.
- **b** Clear the **Save complete SimState in final state** check box.
- **c** Clear the **Final states** check box.
- **d** Click **OK**.

Programmatic equivalent

You can programmatically disable saving of a SimState:

```
set_param('sf_boiler','SaveCompleteFinalSimState','off', ...
'SaveFinalState','off');
```

Loading the SimState

1 Enable loading of a SimState.

- **a** Open the Configuration Parameters dialog box and go to the **Data Import/Export** pane.
- **b** Select the **Initial state** check box.
- **c** Enter the variable that contains the SimState of your chart: `sf_boiler_ctx01`.
- **d** Click **Apply**.

Programmatic equivalent

You can programmatically enable loading of a SimState:

```
set_param('sf_boiler','LoadInitialState','on', ...
'InitialState', ['sf_boiler_ctx01']);
```

2 Define the new stop time for this simulation segment.

- **a** In the Configuration Parameters dialog box, go to the **Solver** pane.
- **b** For **Stop time**, enter `600`.
- **c** Click **OK**.

You do not need to enter a new start time because the simulation continues from where it left off.

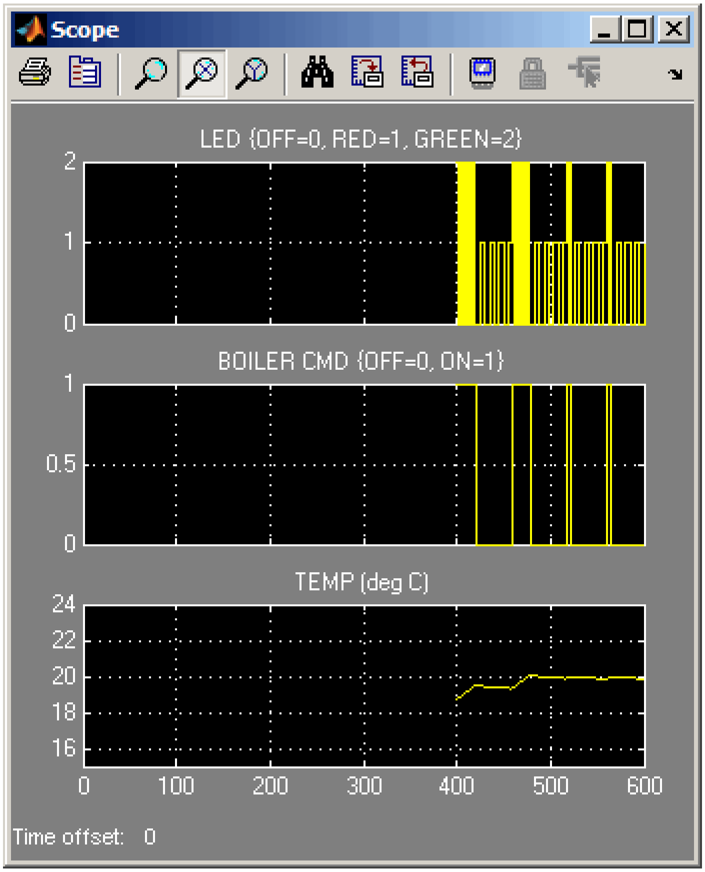
Programmatic equivalent

You can programmatically set the new stop time:

```
set_param('sf_boiler','StopTime','600');
```

Simulating the Specific Segment

When you simulate the model, the following output appears in the Scope block.

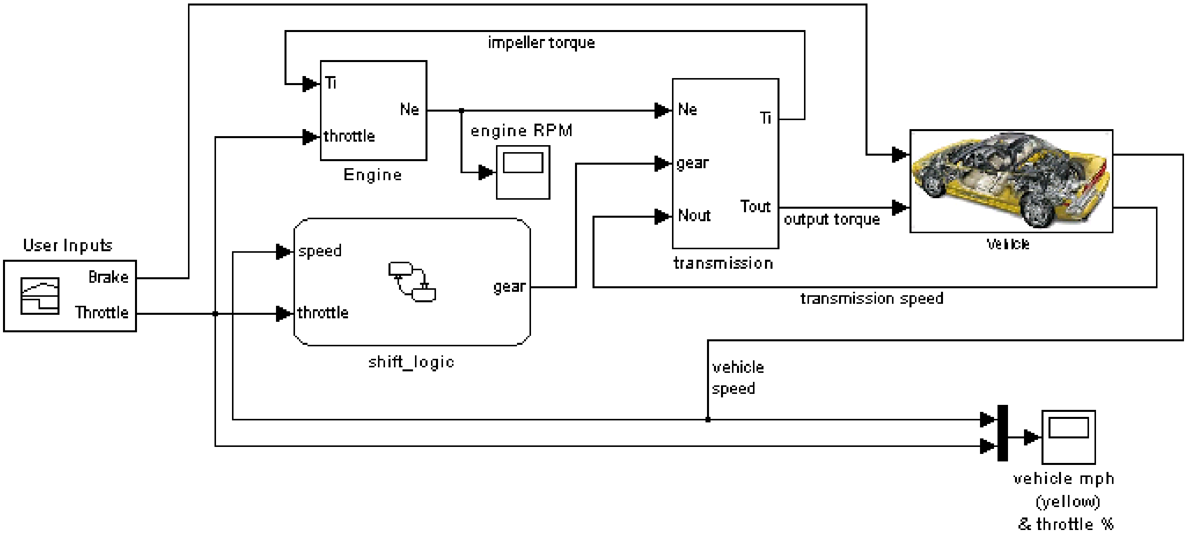


Testing a Unique Chart Configuration

In this section...

Goal of the Tutorial

Suppose that you want to test the response of the `sf_car` model to a sudden change in value for `gear`.



This model simulates for 30 seconds, but you want to see what happens when the value of `gear` changes at t = 10. You can simulate the model, save the SimState at t = 10, load and modify the SimState, and then simulate again between t = 10 and 20.

Step	Task	Reference
1	Define the SimState for your chart.	“Defining the SimState” on page 12-11
2	Load the SimState and modify values.	“Loading the SimState and Modifying Values” on page 12-14
3	Test the modified SimState by running the model.	“Testing the Modified SimState” on page 12-19

Defining the SimState

1 Open the model.

Type `sf_car` at the command prompt.

2 Enable saving of a SimState.

- **a** Open the Configuration Parameters dialog box and go to the **Data Import/Export** pane.
- **b** Select the **Final states** check box.
- **c** Enter a name, such as `sf_car_ctx01`.

  You can choose any alphanumeric string for the name.
- **d** Select the **Save complete SimState in final state** check box.
- **e** Click **Apply**.

Programmatic equivalent

You can programmatically enable saving of a SimState:

```
set_param('sf_car','SaveFinalState','on', ...
'FinalStateName', ['sf_car_ctx01'], ...
'SaveCompleteFinalSimState','on');
```

For details about setting model parameters, see `set_param` in the *Simulink Reference*.

3 Define the start and stop times for this simulation segment.

a. In the Configuration Parameters dialog box, go to the **Solver** pane.
b. For **Start time**, enter `0`.
c. For **Stop time**, enter `10`.
d. Click **OK**.

Programmatic equivalent

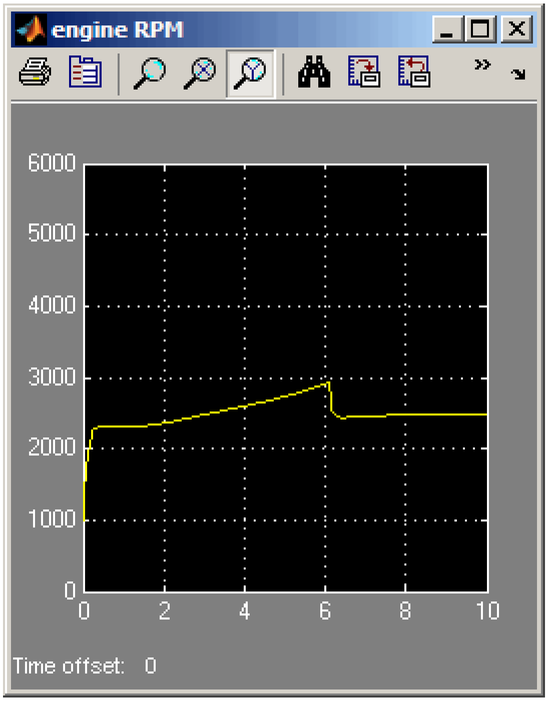
You can programmatically set the start and stop times:

```
set_param('sf_car','StartTime','0', ...
'StopTime','10');
```

4. Start simulation.

   When you simulate the model, you save the complete simulation state at t = 10 in the variable `sf_car_ctx01` in the MATLAB base workspace.

At t = 10, the engine is operating at a steady-state value of 2500 RPM.



5 Disable saving of a SimState.

This step prevents you from overwriting the SimState you saved in the previous step.

a Open the Configuration Parameters dialog box and go to the **Data Import/Export** pane.

b Clear the **Save complete SimState in final state** check box.

c Clear the **Final states** check box.

d Click **OK**.

Programmatic equivalent

You can programmatically disable saving of a SimState:

```
set_param('sf_car','SaveCompleteFinalSimState','off', ...
'SaveFinalState','off');
```

Loading the SimState and Modifying Values

1. Enable loading of a SimState.
   a. Open the Configuration Parameters dialog box and go to the **Data Import/Export** pane.
   b. Select the **Initial state** check box.
   c. Enter the variable that contains the SimState of your chart: `sf_car_ctx01`.
   d. Click **OK**.

   ### Programmatic equivalent

   You can programmatically enable loading of a SimState:

   ```
   set_param('sf_car','LoadInitialState','on', ...
   'InitialState', ['sf_car_ctx01']);
   ```

2. Define an object handle for the SimState values of the `shift_logic` chart.

   At the command prompt, type:

   ```
   blockpath = 'sf_car/shift_logic';
   c = sf_car_ctx01.getBlockSimState(blockpath);
   ```

Tip If the chart appears highlighted in the model window, you can specify the block path using `gcb`:

```
c = sf_car_ctx01.getBlockSimState(gcb);
```

What does the getBlockSimState method do?

The `getBlockSimState` method:

- Makes a copy of the SimState of your chart, which is stored in the final state data of the model.
- Provides a root-level handle or *reference* to the copy of the SimState, which is a hierarchical tree of graphical and nongraphical chart objects.

  Each node in this tree is also a handle to a state, data, or other chart object.

Note Because the entire tree consists of object handles, the following assignment statements do not work:

- `stateCopy = c.state`
- `dataCopy = c.data`
- `simstateCopy = c`

These assignments create copies of the object handles, not SimState values. The only way to copy SimState values is to use the `clone` method. For details, see "Methods for Interacting with the SimState of a Chart" on page 12-35 and "Rules for Using the SimState of a Chart" on page 12-38.

---

3 Look at the contents of the SimState.

```
c =

  Block:    "shift_logic"    (handle)    (active)
  Path:     sf_car/shift_logic

  Contains:

    + gear_state         "State (AND)"          (active)
    + selection_state    "State (AND)"          (active)
      gear               "Block output data"    double [1, 1]
      down_th            "Local scope data"     double [1, 1]
      up_th              "Local scope data"     double [1, 1]
```

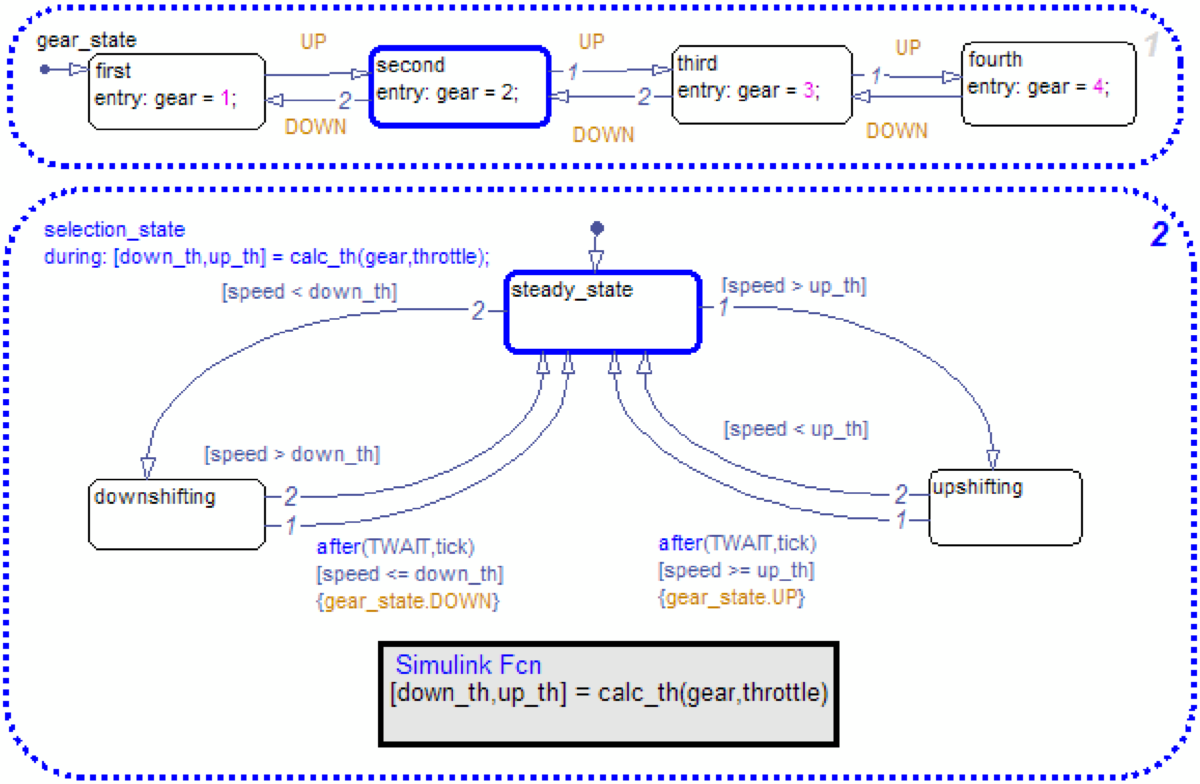
The SimState of your chart contains a list of states and data in hierarchical order.

4 Highlight the states that are active in your chart at t = 10.

At the command prompt, type:

```
c.highlightActiveStates;
```

In the chart, all active states appear highlighted.



To highlight active states *automatically* at the end of a simulation, enable chart animation and select **Maintain Highlighting** in the debugger. For details, see “Animating Stateflow Charts” on page 26-3.

Tip To check if a single state is active, you can use the `isActive` method. For example, type:

```
c.gear_state.second.isActive
```

This command returns true (1) when a state is active and false (0) otherwise. For information on other methods, see "Methods for Interacting with the SimState of a Chart" on page 12-35.

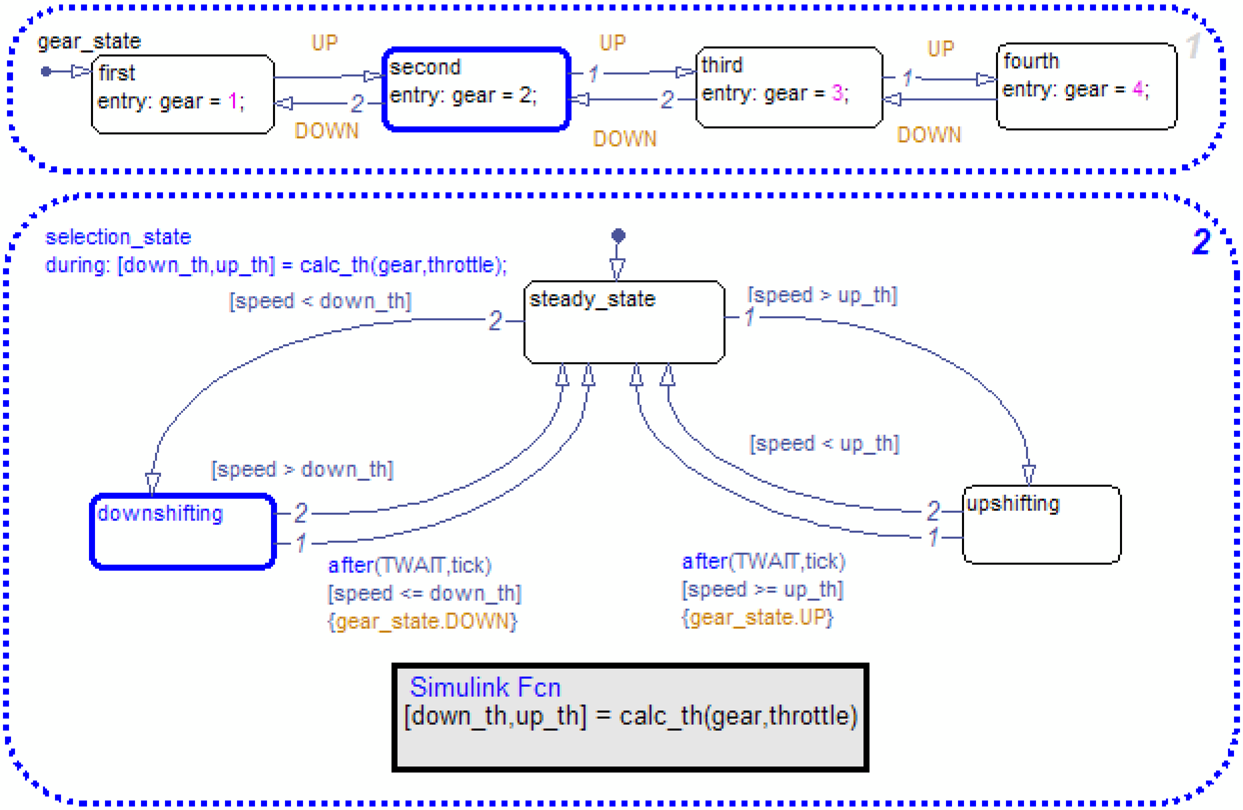
---

5 Change the active substate of `selection_state` to `downshifting`.

Use this command:

```
c.selection_state.downshifting.setActive;
```

The newly active substate appears highlighted in the chart.



6 Change the value of output data `gear`.

When you type `c.gear` at the command prompt, you see a list of data properties similar to this:

```
>> c.gear

ans =

      Description: 'Block output data'
         DataType: 'double'
             Size: '[1, 1]'
            Range: [1x1 struct]
     InitialValue: [1x0 double]
            Value: 2
```

You can change the value of `gear` from 2 to 1 by typing:

```
c.gear.Value = 1;
```

However, you cannot change the data type or size of `gear`. Also, you cannot specify a new value that falls outside the range set by the **Minimum** and **Maximum** parameters. For details, see "Rules for Modifying Data Values" on page 12-38.

7. Save the modified SimState.

   Use this command:

   ```
   sf_car_ctx01 = sf_car_ctx01.setBlockSimState(blockpath, c);
   ```

Testing the Modified SimState

1. Define the new stop time for the simulation segment to test.
   - a In the Configuration Parameters dialog box, go to the **Solver** pane.
   - b For **Stop time**, enter `20`.
   - c Click **OK**.

   You do not need to enter a new start time because the simulation continues from where it left off.

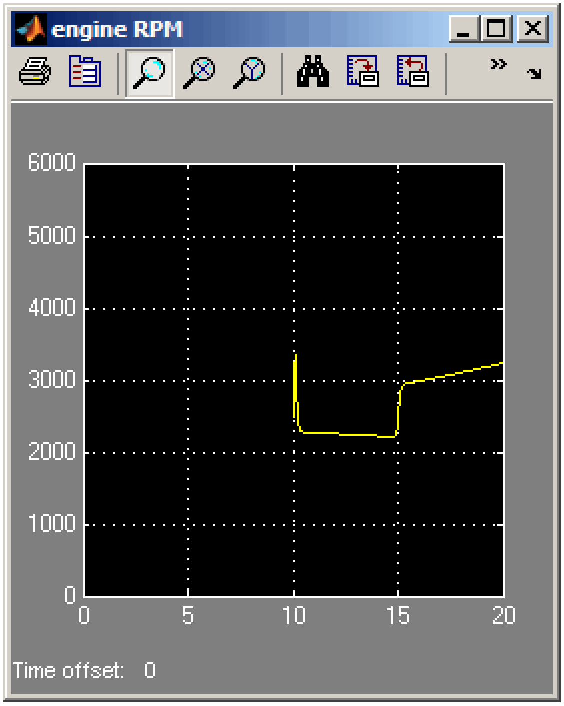
Programmatic equivalent

You can programmatically set the stop time:

```
set_param('sf_car','StopTime','20');
```

2 Start simulation.

The engine reacts as follows:

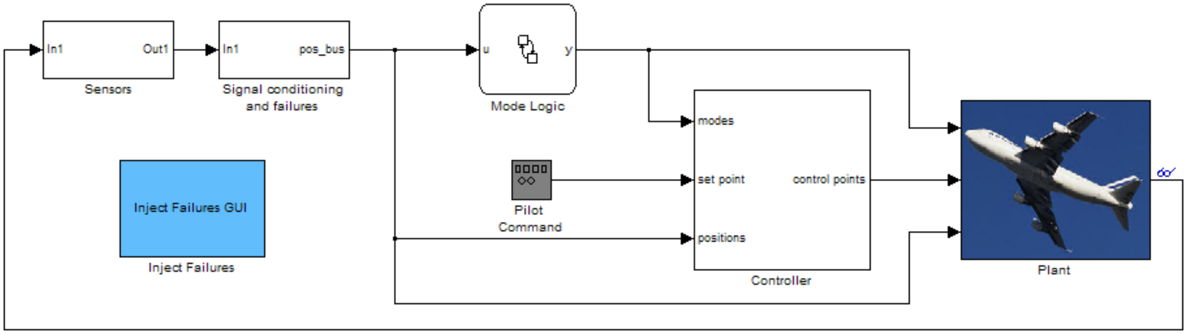


Testing a Chart with Fault Detection and Redundant Logic

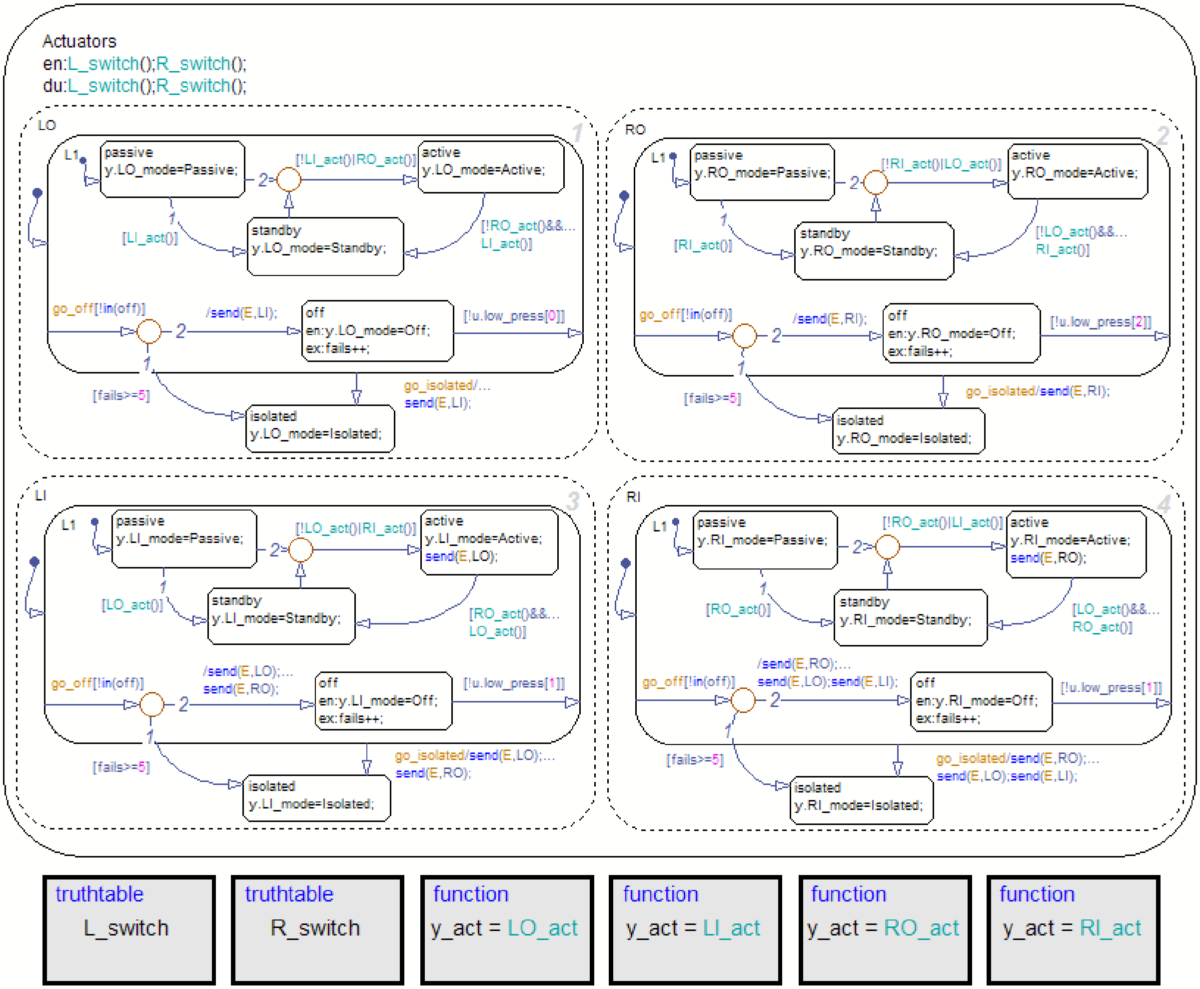
In this section...

Goal of the Tutorial

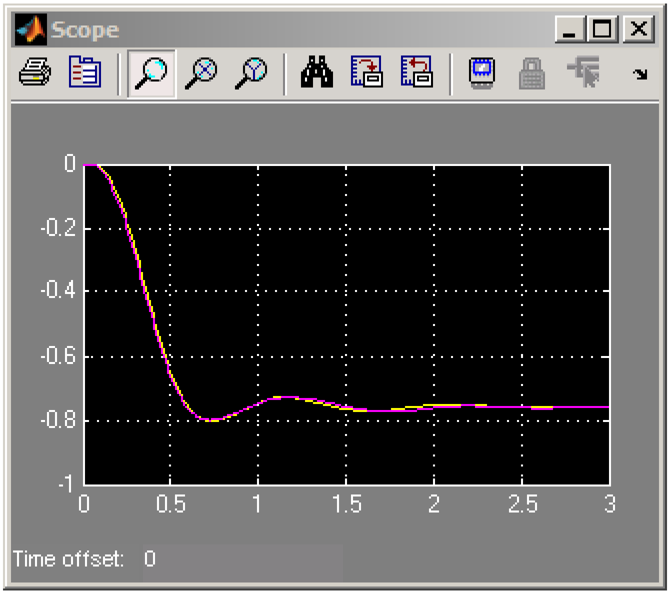
Suppose that you want to test the response of the `sf_aircraft` model to one or more actuator failures in an elevator system. (For details of how this model works, see the demo description for Modeling Fault Management Control Logic in an Aircraft Elevator Control System.)



The Mode Logic chart monitors the status of actuators for two elevators. Each elevator has an outer (primary) actuator and an inner (secondary) actuator. In normal operation, the outer actuators are active and the inner actuators are on standby.



When the four actuators are working correctly, the left and right elevators reach steady-state positions in 3 seconds.



Suppose that you want to see what happens at t = 3 when at least one actuator fails. You can simulate the model, save the SimState at t = 3, load and modify the SimState, and then simulate again between t = 3 and 10.

Step	Task	Reference
1	Define the SimState for your chart.	"Defining the SimState" on page 12-24
2	Load the SimState and modify values for one actuator failure.	"Modifying SimState Values for One Actuator Failure" on page 12-25
3	Test the modified SimState by running the model.	"Testing the SimState for One Failure" on page 12-31

Step	Task	Reference
4	Modify SimState values for two actuator failures.	"Modifying SimState Values for Two Actuator Failures" on page 12-33
5	Test the modified SimState by running the model again.	"Testing the SimState for Two Failures" on page 12-34

Defining the SimState

1. Open the model.

   Type `sf_aircraft` at the command prompt.

2. Enable saving of a SimState.

   a. Open the Configuration Parameters dialog box and go to the **Data Import/Export** pane.
   b. Select the **Final states** check box.
   c. Enter a name, such as `xFinal`.

      You can choose any alphanumeric string for the name.

   d. Select the **Save complete SimState in final state** check box.
   e. Click **Apply**.

   ### Programmatic equivalent

   You can programmatically enable saving of a SimState:

   ```
   set_param('sf_aircraft','SaveFinalState','on', ...
   'FinalStateName', ['xFinal'], ...
   'SaveCompleteFinalSimState','on');
   ```

   For details about setting model parameters, see `set_param` in the *Simulink Reference*.

3. Define the stop time for this simulation segment.

   a. In the Configuration Parameters dialog box, go to the **Solver** pane.

- **b** For **Stop time**, enter `3`.
- **c** Click **OK**.

Programmatic equivalent

You can programmatically set the stop time:

```
set_param('sf_aircraft','StopTime','3');
```

4 Start simulation.

When you simulate the model, you save the complete simulation state at t = 3 in the variable `xFinal` in the MATLAB base workspace.

5 Disable saving of a SimState.

This step prevents you from overwriting the SimState you saved in the previous step.

- **a** Open the Configuration Parameters dialog box and go to the **Data Import/Export** pane.
- **b** Clear the **Save complete SimState in final state** check box.
- **c** Clear the **Final states** check box.
- **d** Click **OK**.

Programmatic equivalent

You can programmatically disable saving of a SimState:

```
set_param('sf_aircraft','SaveCompleteFinalSimState','off', ...
'SaveFinalState','off');
```

Modifying SimState Values for One Actuator Failure

1 Enable loading of a SimState.

- **a** Open the Configuration Parameters dialog box and go to the **Data Import/Export** pane.
- **b** Select the **Initial state** check box.

c Enter the variable that contains the SimState of your chart: `xFinal`.

d Click **OK**.

Programmatic equivalent

You can programmatically enable loading of a SimState:

```
set_param('sf_aircraft','LoadInitialState','on', ...
'InitialState', ['xFinal']);
```

2 Define an object handle for the SimState values of the Mode Logic chart.

At the command prompt, type:

```
blockpath = 'sf_aircraft/Mode Logic';
c = xFinal.getBlockSimState(blockpath);
```

Tip If the chart appears highlighted in the model window, you can specify the block path using `gcb`:

```
c = xFinal.getBlockSimState(gcb);
```

What does the getBlockSimState method do?

The `getBlockSimState` method:

- Makes a copy of the SimState of your chart, which is stored in the final state data of the model.
- Provides a root-level handle or *reference* to the copy of the SimState, which is a hierarchical tree of graphical and nongraphical chart objects.

  Each node in this tree is also a handle to a state, data, or other chart object.

Note Because the entire tree consists of object handles, the following assignment statements do not work:

- `stateCopy = c.state`
- `dataCopy = c.data`
- `simstateCopy = c`

These assignments create copies of the object handles, not SimState values. The only way to copy SimState values is to use the `clone` method. For details, see "Methods for Interacting with the SimState of a Chart" on page 12-35 and "Rules for Using the SimState of a Chart" on page 12-38.

---

3 Look at the contents of the SimState.

```
c =

  Block:    "Mode Logic"    (handle)    (active)
  Path:     sf_aircraft/Mode Logic

  Contains:

    + Actuators          "State (OR)"          (active)
    + LI_act             "Function"
    + LO_act             "Function"
    + L_switch           "Function"
    + RI_act             "Function"
    + RO_act             "Function"
    + R_switch           "Function"
      y                  "Block output data"    ModeBus [1, 1]
```

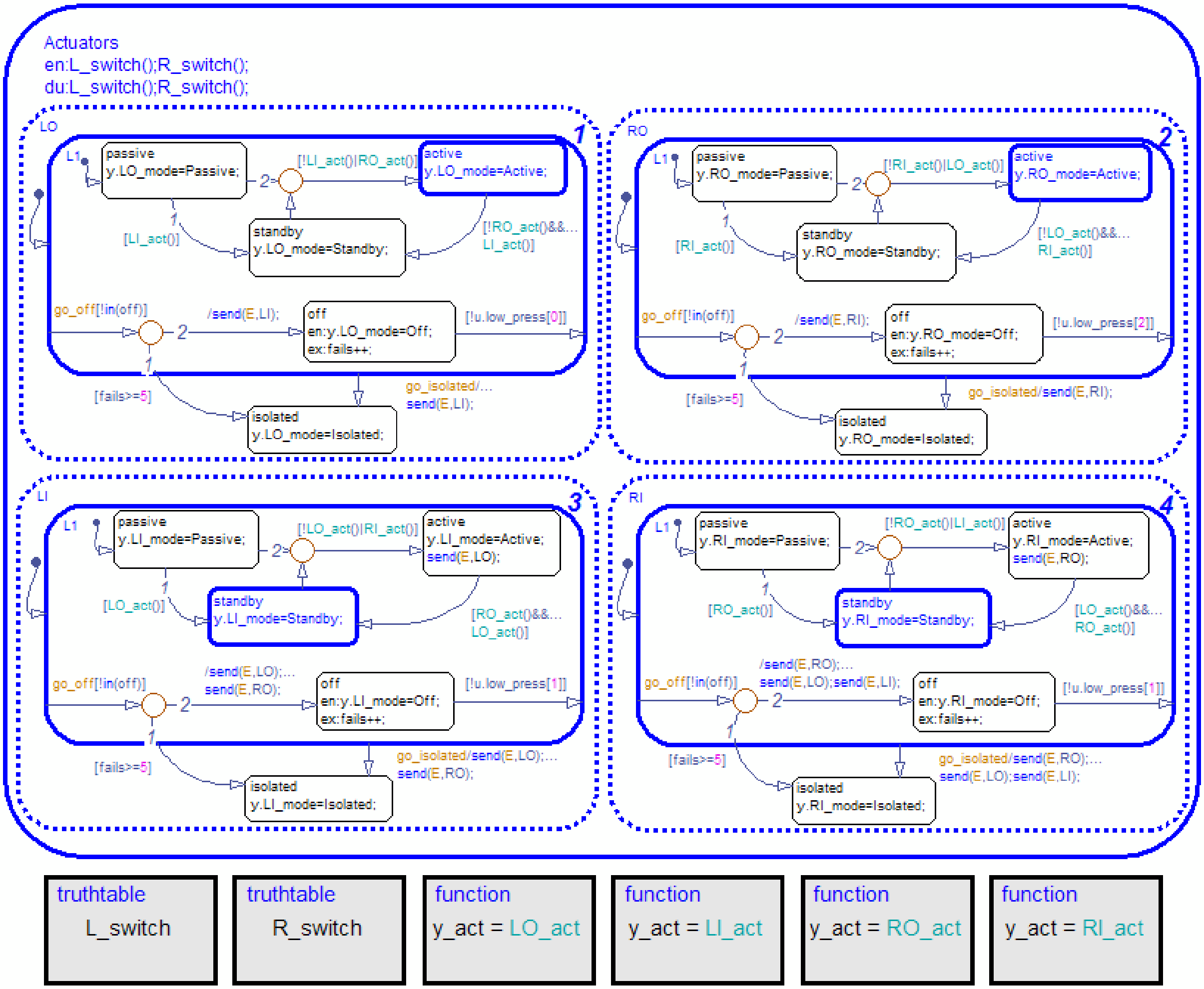
The SimState of your chart contains a list of states, functions, and data in hierarchical order.

4 Highlight the states that are active in your chart at t = 3.

At the command prompt, type:

```
c.highlightActiveStates;
```

Active states appear highlighted. By default, the two outer actuators are active and the two inner actuators are on standby.



To highlight active states *automatically* at the end of a simulation, enable chart animation and select **Maintain Highlighting** in the debugger. For details, see “Animating Stateflow Charts” on page 26-3.

Tip To check if a single state is active, you can use the `isActive` method. For example, type:

```
c.Actuators.LI.L1.standby.isActive
```

This command returns true (1) when a state is active and false (0) otherwise. For information on other methods, see "Methods for Interacting with the SimState of a Chart" on page 12-35.

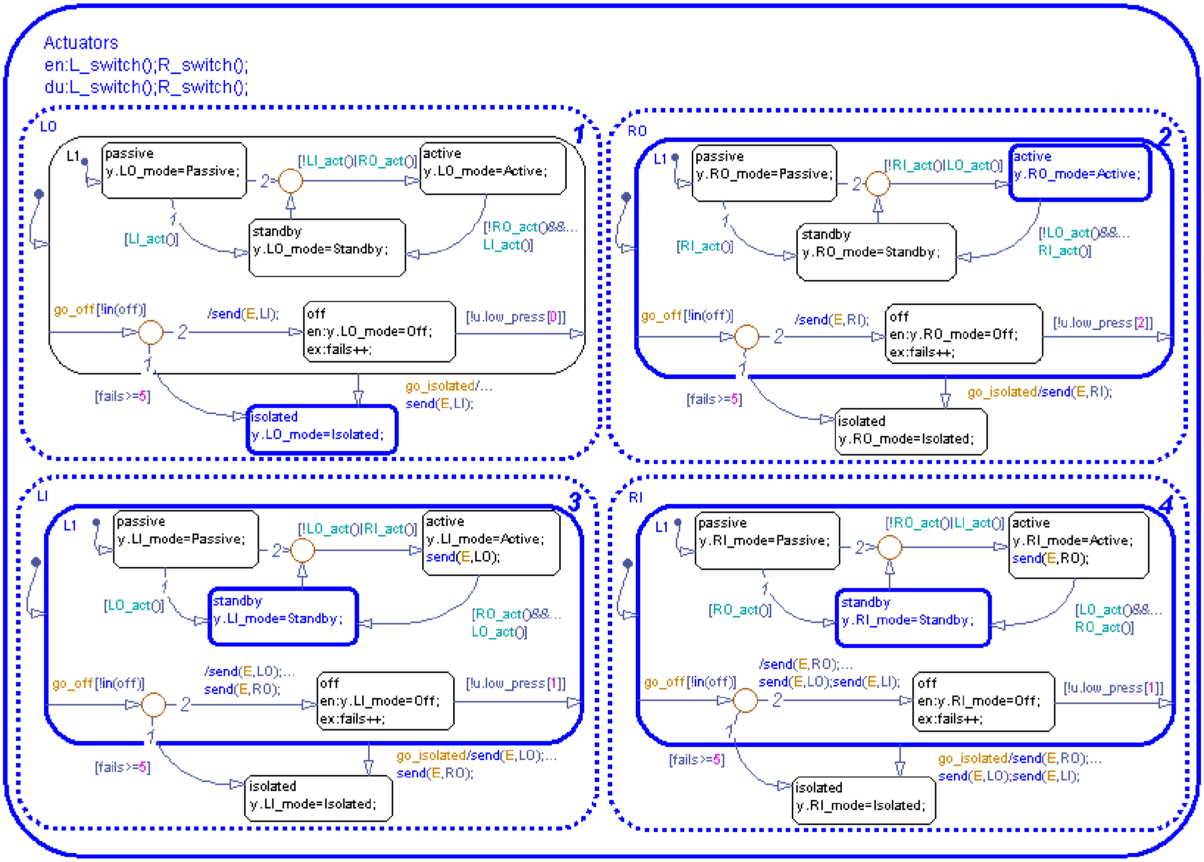
---

5 Change the state activity in the chart to reflect one actuator failure.

Assume that the left outer (LO) actuator fails. To change the state, use this command:

```
c.Actuators.LO.isolated.setActive;
```

The newly active substate appears highlighted in the chart.



The `setActive` method ensures that the chart exits and enters the appropriate states to maintain state consistency. However, the method does not perform `entry` actions for the newly active substate. Similarly, the method does not perform `exit` actions for the previously active substate.

6 Change the value of output bus element `y.LO_mode`.

You can change the value of `y.LO_mode` to `Isolated` by typing:

```
c.y.Value.LO_mode = sf_aircraft_ModeType.Isolated;
```

This value belongs to the list of enumerated values for the `sf_aircraft_ModeType` definition. For more information, see "Rules for Modifying Data Values" on page 12-38.

7. Save the modified SimState by using this command:

```
xFinal = xFinal.setBlockSimState(blockpath, c);
```

Testing the SimState for One Failure

1. Define the new stop time for the simulation segment to test.
   a. Go to the **Solver** pane of the Configuration Parameters dialog box.
   b. For **Stop time**, enter `10`.
   c. Click **OK**.

   You do not need to enter a new start time because the simulation continues from where it left off.

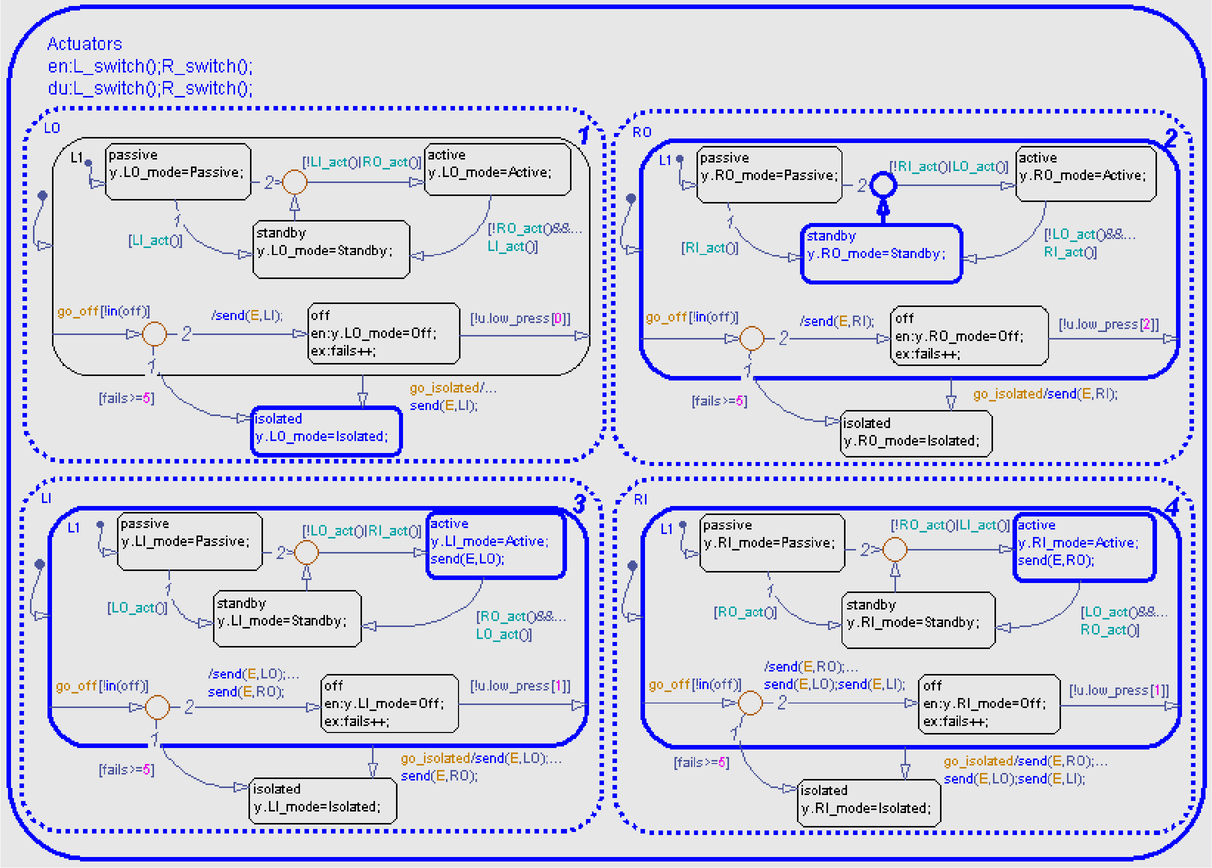
   Programmatic equivalent

   You can programmatically set the stop time:

```
set_param('sf_aircraft','StopTime','10');
```

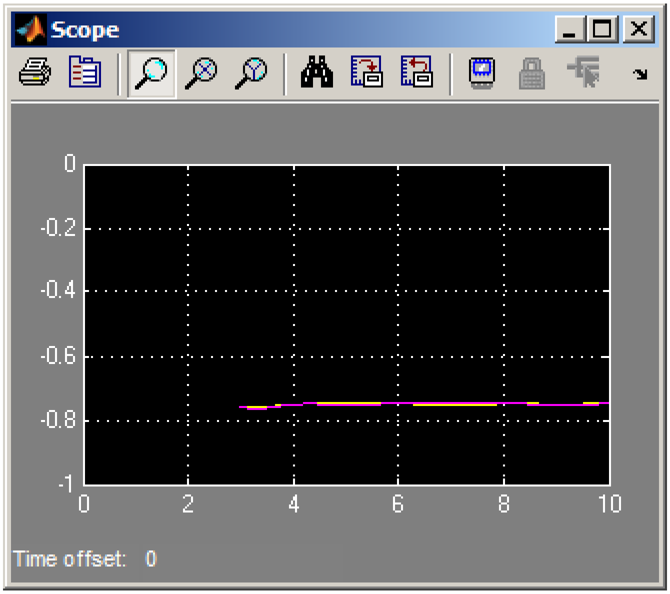
2. Start simulation.

   Chart animation shows that the other three actuators react appropriately to the failure of the left outer (LO) actuator.



This actuator...	Switches from...	Because...
Left inner (LI)	Standby to active	The left elevator must compensate for the left outer (LO) actuator failure.
Right inner (RI)	Standby to active	The same hydraulic line connects to both inner actuators.
Right outer (RO)	Active to standby	Only one actuator per elevator can be active.

Both elevators continue to maintain steady-state positions.



Modifying SimState Values for Two Actuator Failures

1 Change the state activity in the chart to reflect two actuator failures.

Assume that the left inner (LI) actuator also fails. To change the state, use this command:

```
c.Actuators.LI.isolated.setActive;
```

2 Change the value of output bus element `y.LI_mode`.

You can change the value of `y.LI_mode` to `Isolated` by typing:

```
c.y.Value.LI_mode = sf_aircraft_ModeType.Isolated;
```

3 Save the modified SimState by using this command:

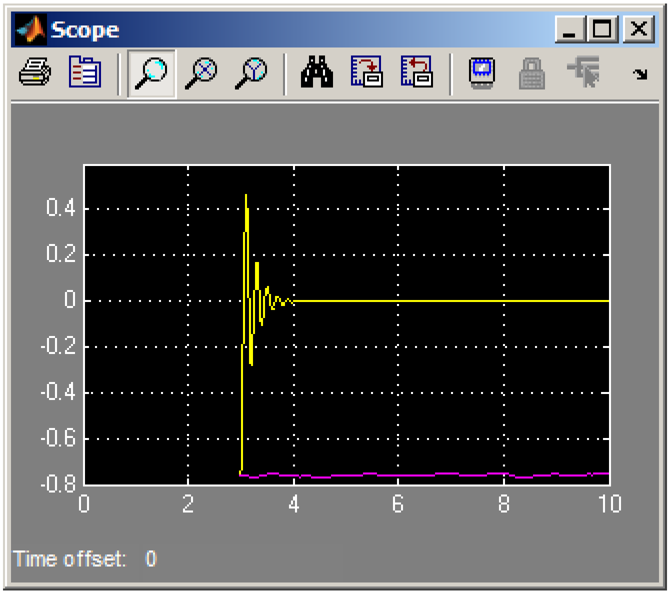
```
xFinal = xFinal.setBlockSimState(blockpath, c);
```

Testing the SimState for Two Failures

1 In the Configuration Parameters dialog box, verify that the stop time is 10.

2 Restart simulation.

Because of failures in both actuators, the left elevator stops working. The right elevator maintains a steady-state position.



If you modify the SimState of your chart to test the response of the right elevator to actuator failures, you get similar results.

Methods for Interacting with the SimState of a Chart

You can use the following methods to interact with the SimState of a chart. Assume that *ch* is a handle to the SimState of your chart, which you obtain using the `getBlockSimState` method.

Type of Object	Method	Description	Example
All chart objects	`open`	For graphical objects, highlights the object in the Stateflow Editor. For nongraphical objects, highlights the object in the Model Explorer. **Note** For persistent data in MATLAB functions, this method opens the function editor and highlights the persistent data at the exact line in the script.	`ch.data.open`
Chart	`checkStateConsistency`	Verifies that all states in a chart are consistent.	`ch.checkStateConsistency`

Type of Object	Method	Description	Example
		• If a state is inactive, no substates are active. • If a state with parallel decomposition is active, all substates are active. • If a state with exclusive decomposition is active, only one substate is active.	
Chart	`clone`	Copies the entire chart simulation state to a new variable.	`*newSimState* = *ch*.clone`
Chart	`highlightActiveStates`	Highlights all active states in the Stateflow Editor.	`*ch*.highlightActiveStates`
Chart	`isStateConsistent`	Returns true (1) if all states pass a consistency check and false (0) otherwise.	`*ch*.isStateConsistent`
Chart	`removeHighlighting`	Removes all highlighting in the Stateflow Editor.	`*ch*.removeHighlighting`

Type of Object	**Method**	**Description**	**Example**
State	`isActive`	Returns true (1) if a state is active and false (0) otherwise.	`ch.state.isActive`
State Must be an exclusive leaf state	`setActive`	Sets a state to be active. This method ensures that no other exclusive states at that level are active.	`ch.state.substate.setActive`
State Must have a history junction and exclusive substates	`getPrevActiveChild`	Returns the previously active substate.	`ch.state.getPrevActiveChild`
State Must be inactive; must have a history junction and exclusive substates	`setPrevActiveChild`	Sets the previously active substate.	`ch.state.setPrevActiveChild('B')` **Note** The argument must be the name of a substate (in quotes), or the full SimState path to a substate (without quotes).

Rules for Using the SimState of a Chart

In this section...

Limitations on Values You Can Modify

A SimState does not include information about these elements:

- Machine-parented data
- Persistent data in custom C code
- Persistent data in external MATLAB code

Therefore, you cannot modify the values of those elements.

Rules for Modifying Data Values

These rules apply when you modify data values:

- You cannot change the data type or size. Scalar data must remain scalar. Vector and matrix data must keep the same dimensions. The only exception to this rule is Stateflow data of `ml` type (see "ml Data Type" on page 10-47 for details).
- For enumerated data types, you can choose only enumerated values from the type definition. For other data types, new values must fall within the range that you specify in the **Minimum** and **Maximum** parameters.
- Use one-based indexing to define rows and columns of a matrix.

Suppose that you want to change the value of an element in a 21-by-12 matrix. To modify the element in the first row and second column, type:

```
c.state_name.data_name.Value(1,2) = newValue;
```

Rules for Modifying State Activity

These rules apply when you use the `setActive` method on an exclusive (OR) leaf state:

- State-parented local data does not reinitialize.
- The newly active state does not execute any entry actions. Similarly, the previously active state does not execute any exit actions.

  If you want these state actions to occur, you must execute them separately. For example, if your state actions assign values to data, you must assign the values explicitly.
- The `setActive` method tries to maintain state consistency by:
  - Updating state activity for parent, grandparent, and sibling states
  - Resetting temporal counters for newly active states
  - Updating values of state output data (read-only)
  - Enabling or disabling function-call subsystems and Simulink functions that bind to states
- The `highlightActiveStates` method also executes when these conditions are true:
  - The model is open.
  - The chart is visible.
  - The `highlightActiveStates` method has executed at least once, but not the `removeHighlighting` method.

Restriction on Continuous-Time Charts

After you load a SimState for a continuous-time chart, you can restart simulation from a nonzero time. However, you cannot modify the state activity or any data values, because the SimState for a continuous-time chart is read-only. For more information, see “Summary of Rules for Continuous-Time Modeling” on page 16-26.

No Partial Loading of a SimState

When you load a SimState, the complete simulation state is available as a variable in the MATLAB base workspace. You cannot perform partial loading of a SimState for a subset of chart objects.

Restriction on Copying SimState Values

Use the `clone` method to copy an entire SimState to a new variable (see "Methods for Interacting with the SimState of a Chart" on page 12-35). You cannot copy a subset of SimState values, because the `clone` method works only at the chart level.

Suppose that you obtain a handle to the SimState of your chart using these commands:

```
blockpath = 'model/chart';
c = xFinal.getBlockSimState(blockpath);
```

Assignment statements such as `stateCopy = c.state`, `dataCopy = c.data`, and `simstateCopy = c` do not work. These assignments create copies of object handles, not SimState values.

SimState Limitations That Apply to All Blocks in a Model

For a list of SimState limitations that apply to all blocks in a Simulink model, see "Limitations of the SimState" in the *Simulink User's Guide.*

Best Practices for Using the SimState of a Chart

In this section...

Use MAT-Files to Save a SimState for Future Use

To save a SimState from the MATLAB base workspace, save the variable with final state data in a MAT-file.

For example, type at the command prompt:

```
save('sf_car_ctx01.mat', 'sf_car_ctx01')
```

For more information, see `save` in the MATLAB documentation.

Use Scripts to Save SimState Commands for Future Use

To save a list of SimState commands for future use, copy them from a procedure and paste them in a MATLAB script.

For example, to reuse the commands in "Dividing a Long Simulation into Segments" on page 12-5, you can store them in a script named `sf_boiler_simstate_commands.m`:

```
% Load the model.
sf_boiler;

% Set parameters to save the SimState at the desired time.
set_param('sf_boiler','SaveFinalState','on','FinalStateName',...
['sf_boiler_ctx01'],'SaveCompleteFinalSimState','on');

% Specify the start and stop times for the simulation segment.
set_param('sf_boiler','StartTime','0','StopTime','400');

% Simulate the model.
sim('sf_boiler');

% Disable saving of the SimState to avoid overwriting.
set_param('sf_boiler','SaveCompleteFinalSimState','off', ...
'SaveFinalState','off');

% Load the SimState.
set_param('sf_boiler', 'LoadInitialState', 'on', ...
'InitialState', ['sf_boiler_ctx01']);

% Specify the new stop time for the simulation segment.
set_param('sf_boiler','StopTime','600');

% Simulate the model.
sim('sf_boiler');
```

13

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 13-17.

Where You Can Use Vectors and Matrices

You can define vectors and matrices at these levels of the Stateflow hierarchy:

- Charts
- Subcharts
- States
- Functions

You can use vectors and matrices to define:

- Input data
- Output data
- Local data
- Function inputs
- Function outputs

You can also use vectors and matrices as arguments for:

- State actions
- Transition actions
- MATLAB functions (see Chapter 23, “Using MATLAB Functions in Stateflow Charts”)
- Truth table functions (see Chapter 22, “Truth Table Functions for Decision-Making Logic”)
- Graphical functions (see “Graphical Functions for Reusing Logic Patterns and Iterative Loops” on page 7-30)
- Simulink functions (see Chapter 24, “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 13-11 and “Rules for Using Vectors and Matrices in Stateflow Charts” on page 13-13.

How to Define Vectors and Matrices

In this section...

Defining a Vector

Define a vector in a Stateflow chart as follows:

1. Add data to your chart as described in "Adding Data" on page 8-2.
2. In the **General** pane of the Data properties dialog box, enter the dimensions of the vector in the **Size** field.

   For example, enter `[4 1]` to specify a 4-by-1 vector.
3. Specify the name, base type, and other properties for the new data.

Note Vectors cannot have the base type `ml`. See "Rules for Using Vectors and Matrices in Stateflow Charts" on page 13-13.

---

4. Set initial values for the vector.
   - If initial values of all elements are the same, enter a real number in the **Initial value** field. This value applies to all elements of a vector of any size.
   - If initial values differ, enter real numbers in the **Initial value** field. For example, you can enter:

     ```
     [1; 3; 5; 7]
     ```

Tip If you want to initialize all elements of a vector to 0, do nothing. When no values are explicitly defined, all elements initialize to 0.

---

5. Click **Apply**.

Defining a Matrix

Define a matrix in a Stateflow chart as follows:

1. Add data to your chart as described in “Adding Data” on page 8-2.

2. In the **General** pane of the Data properties dialog box, enter the dimensions of the matrix in the **Size** field.

   For example, enter `[3 3]` to specify a 3-by-3 matrix.

3. Specify the name, base type, and other properties for the new data.

Note Matrices cannot have the base type `ml`. See “Rules for Using Vectors and Matrices in Stateflow Charts” on page 13-13.

---

4. Set initial values for the matrix.

   - If initial values of all elements are the same, enter a real number in the **Initial value** field. This value applies to all elements of a matrix of any size.

   - If initial values differ, enter real numbers in the **Initial value** field. For example, you can enter:

     ```
     [1 2 3; 4 5 6; 7 8 9]
     ```

Tip If you want to initialize all elements of a matrix to 0, do nothing. When no values are explicitly defined, all elements initialize to 0.

---

5. Click **Apply**.

Scalar Expansion for Converting Scalars to Nonscalars

In this section...
"What Is Scalar Expansion?" on page 13-6
"How Scalar Expansion Works for Functions" on page 13-6

What Is Scalar Expansion?

Scalar expansion is a method of converting scalar data to match the dimensions of vector or matrix data. For example, scalar expansion can convert a value of 1 to a vector or matrix where all the elements are 1.

How Scalar Expansion Works for Functions

Suppose that you have a function signature `yy = example(uu)`, where the formal arguments `yy` and `uu` are scalars. Assume that you have a function call `y = example(u)`. The rules of scalar expansion for function calls with a single output follow.

If the output y is a...	And the input u is a...	Then...
Scalar	Scalar	No scalar expansion occurs.
Vector or matrix	Scalar	Scalar expansion occurs for `example(u)` to match the dimensions of `y`.
Vector or matrix	Vector or matrix	Scalar expansion occurs so that `y[i] = example(u[i])`.
Scalar	Vector or matrix	An error message alerts you to a size mismatch.

For functions with multiple outputs, the same rules apply except for the case where the outputs and inputs of the function call are all vectors or matrices. In this case, scalar expansion does not occur, and an error message alerts you to a size mismatch.

The rules of scalar expansion apply to all functions that you use in Stateflow charts:

- MATLAB functions (see Chapter 23, “Using MATLAB Functions in Stateflow Charts”)
- Graphical functions (see “Graphical Functions for Reusing Logic Patterns and Iterative Loops” on page 7-30)
- Simulink functions (see Chapter 24, “Using Simulink Functions in Stateflow Charts”)
- Truth table functions (see Chapter 22, “Truth Table Functions for Decision-Making Logic”)

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" on page 13-11
"Unary Operations and Actions" on page 13-11
"Assignment Operations" on page 13-12

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 MATLAB Functions to Perform Matrix Multiplication and Division" on page 13-14.

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 13-11.

Do not define vectors and matrices with ml base type

If you define a vector or matrix with `ml` base type, an error message appears when you try to simulate your model. This base type supports only scalar data.

For more information about this type, see "ml Data Type" on page 10-47.

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 MATLAB Functions to Perform Matrix Multiplication and Division

In Stateflow action language, the multiplication and division operators perform element-wise multiplication and division. Use a MATLAB function to perform standard matrix multiplication and division.

For example, suppose that you want to perform standard matrix operations on two square matrices during simulation. Follow these steps:

1. In your chart, add a MATLAB function with the following signature:

   ```
   [y1, y2, y3] = my_matrix_ops(u1, u2)
   ```

2. Double-click the function box to open the editor.

3. In the editor, enter the code below.

   ```
   function [y1, y2, y3] = my_matrix_ops(u1, u2)
   %#codegen

   y1 = u1 * u2;  % matrix multiplication
   y2 = u1 \ u2;  % matrix division from the right
   y3 = u1 / u2;  % matrix division from the left
   ```

   This function computes three values:

   - `y1` is the product of two input matrices `u1` and `u2`.
   - `y2` is the matrix that solves the equation `u1 * y2 = u2`.
   - `y3` is the matrix that solves the equation `y3 * u1 = u2`.

4. Set properties for the input and output data.

  a Open the Model Explorer.

  b In the **Model Hierarchy** pane, navigate to the level of the MATLAB function.

  c In the **Contents** pane, set properties for each data object.

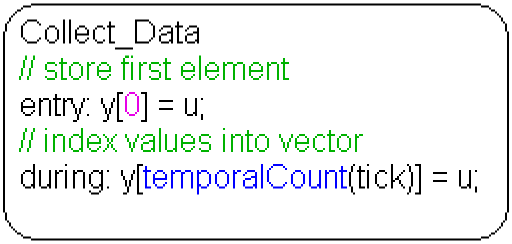
Note To initialize a matrix, see "Defining a Matrix" on page 13-5.

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.



```
Collect_Data
// store first element
entry: y[0] = u;
// index values into vector
during: y[temporalCount(tick)] = u;
```

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.

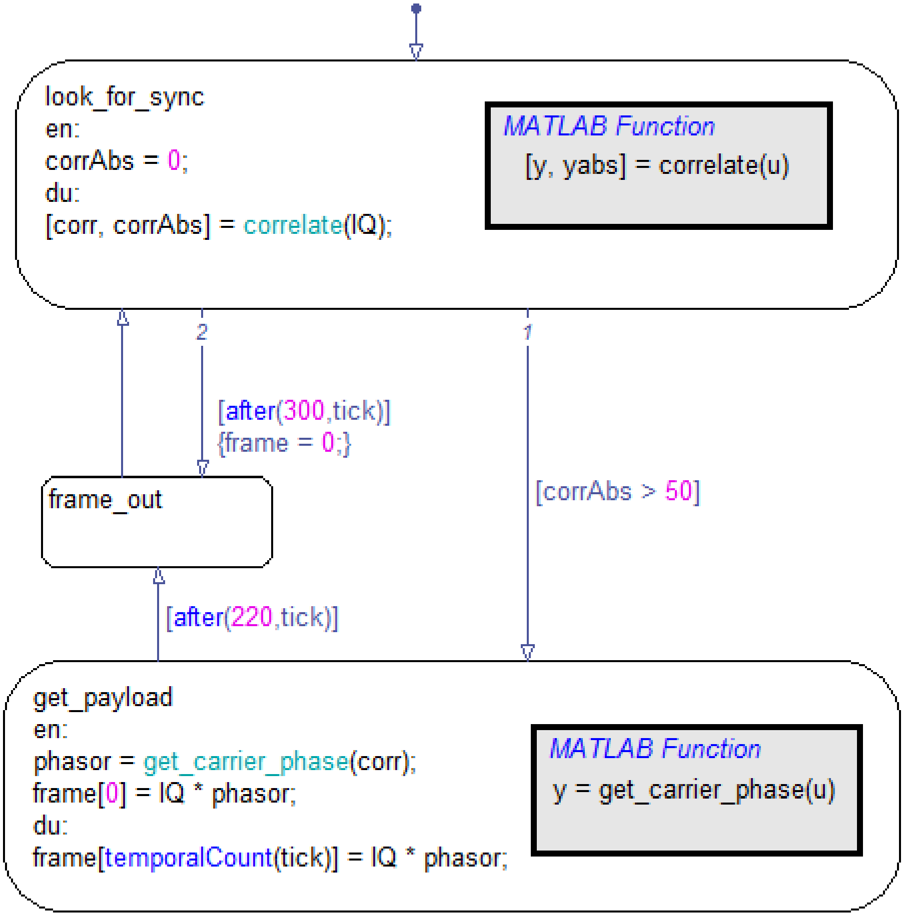
For information about the `temporalCount` operator, see “Using Temporal Logic in State Actions and Transitions” on page 10-63.

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 "Detection of Valid Transmission Data with Frame Synchronization" on page 18-19.

Storage of Complex Data in a Vector

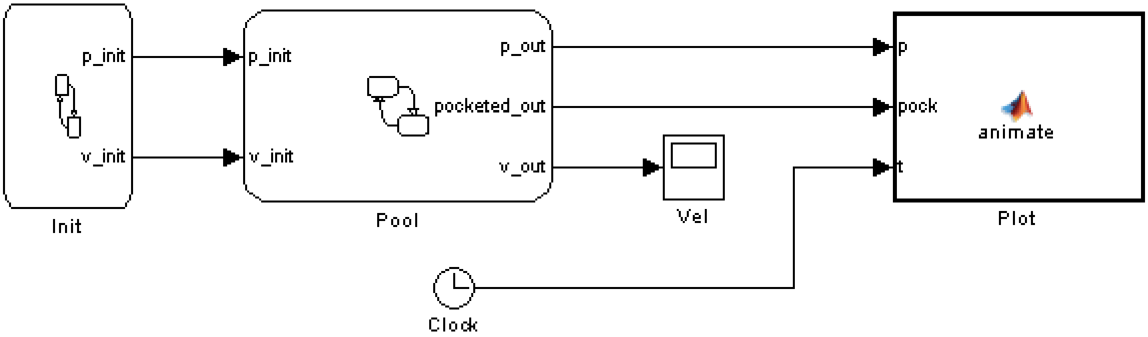
The state `get_payload` stores complex data in the vector `frame`, which is of size 221. The `entry` action assigns the value of `(IQ * phasor)` to the first element of `frame`. The `during` action assigns the next 220 values of `(IQ * phasor)` to successive elements of `frame` until you store 221 elements. (For more information, see "Using the temporalCount Operator to Index a Vector" on page 13-15.)

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 13-10.)

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.

Model Component	Description
Plot block	Animates the motion of each ball during the opening shot.
Vel scope	Displays the velocity of each ball during the opening shot.
Clock	Provides the instantaneous simulation time to the Plot block.

Storage of Two-Dimensional Data in Matrices

To simulate the opening shot, the Pool chart stores two-dimensional data in matrices.

To store values for...	The Pool chart uses...
The instantaneous position of each ball	The 15-by-2 matrix `p`
The instantaneous velocity of each ball	The 15-by-2 matrix `v`
Friction and interaction forces acting on each ball	The 15-by-2 matrix `v_dot`
Boolean data on whether any two balls are in contact	The 15-by-15 matrix `ball_interaction`

Calculation of Two-Dimensional Dynamics of Each Ball

The Pool chart calculates the motion of each ball on the pool table using MATLAB functions that perform matrix calculations.

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

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

```
/initBalls();
```

```
1 [isAnyBallGoingToStop()]
/updateStopFlags();

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

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

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

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.

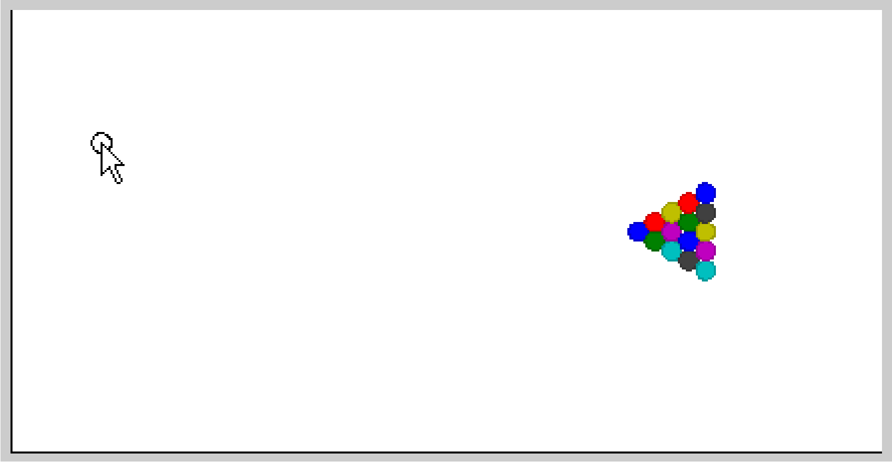
Running the Demo Model

To run the demo model, follow these steps:

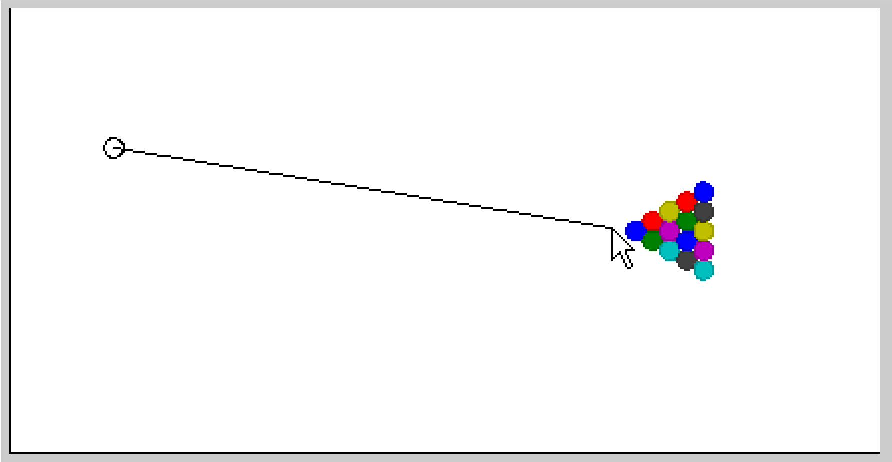
1. Type `sf_pool` at the MATLAB command prompt.

2. Start simulation.

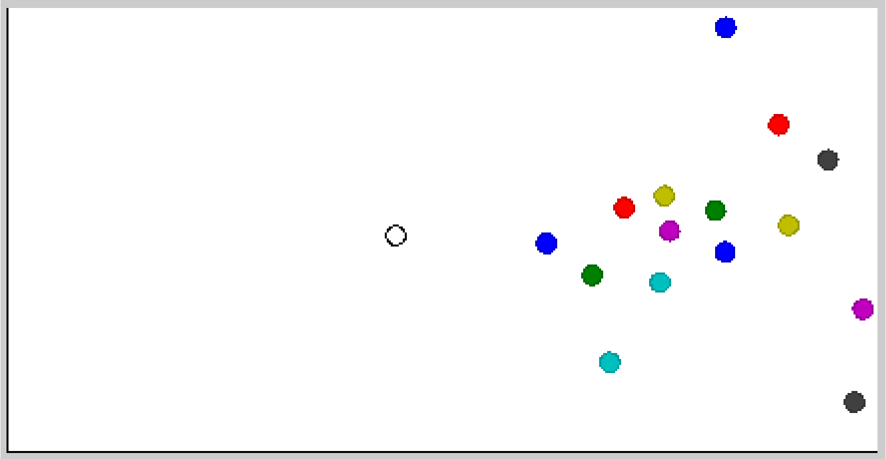
3 Click anywhere in the animated pool table to specify the initial position of the cue ball.



4 Click a different spot to specify the initial velocity of the cue ball.



5 Watch the balls move across the pool table.



14

Using Variable-Size Data in Stateflow Charts

What Is Variable-Size Data?

Variable-size data is data whose size can change at run time. By contrast, fixed-size data is data whose size is known and locked at compile time and, therefore, cannot change at run time.

How Charts Implement Variable-Size Data

Stateflow charts exchange variable-size data with other charts and blocks in their models through MATLAB functions, Simulink functions, and MATLAB truth tables.

You pass variable-size data to these functions as chart-level inputs and outputs from state actions and transition logic. However, you must perform all computations with variable-size data inside the functions, not directly in states or transitions.

For more information about the functions that interact with variable-size, chart-level inputs and outputs, see:

- Chapter 23, “Using MATLAB Functions in Stateflow Charts”
- Chapter 24, “Using Simulink Functions in Stateflow Charts”
- “MATLAB Truth Tables” on page 22-6

Enabling Support for Variable-Size Data

Support for variable-size data is enabled by default. To modify this option for individual charts:

1. Right-click an open area of the chart and select **Properties**.

   The Chart properties dialog box opens.

2. Select or clear the **Support variable-size arrays** check box.

After enabling support at the chart level, declare your variable-size inputs and outputs.

Declaring Variable-Size Inputs and Outputs

1. In the Stateflow Editor, select **Add > Data > Input from Simulink** or **Add > Data > Output to Simulink**.

   The Data properties dialog box opens.

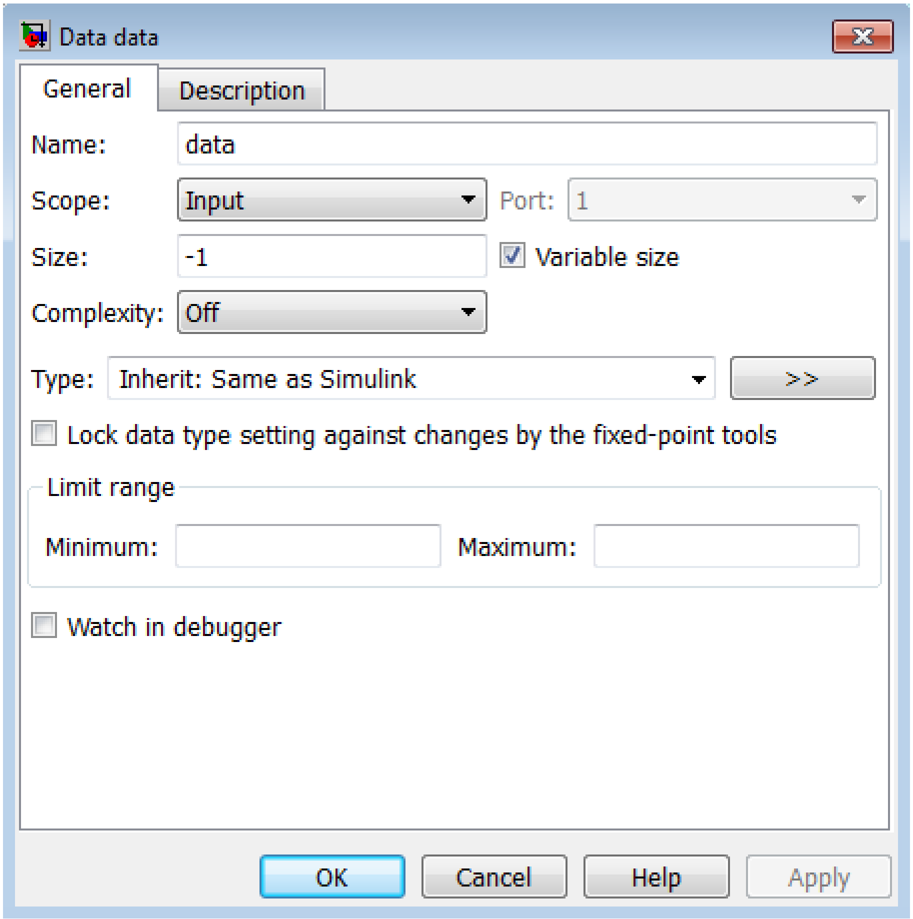
2. Select the **Variable size** check box.

3. Set **Scope** as `Input` or `Output`.

4. Enter size:

Data	What to Specify
Input	Enter `-1` to inherit size from Simulink or specify the explicit size and upper bound. For example, enter `[2 4]` to specify a 2-D matrix where the upper bounds are 2 for the first dimension and 4 for the second.
Output	Specify the explicit size and upper bound.

   For example:



Name: data

Scope: Input

Port: 1

Size: -1

☑ Variable size

Complexity: Off

Type: Inherit: Same as Simulink

☐ Lock data type setting against changes by the fixed-point tools

Limit range

Minimum:

Maximum:

☐ Watch in debugger

Example: Computing Output Based on Size of Input Signal

In this section...

About the Model

The model `sf_varsize_example` shows how MATLAB functions in Stateflow charts exchange variable-size data with other charts and blocks in the model. To open the model, type `sf_varsize_example` at the MATLAB command prompt.

In this model, one Stateflow chart, `VarSizeSignalSource`, uses temporal logic to generate a variable-size signal. A second chart, `size_based_processing`, computes the output based on the size of the signal generated by the first chart:

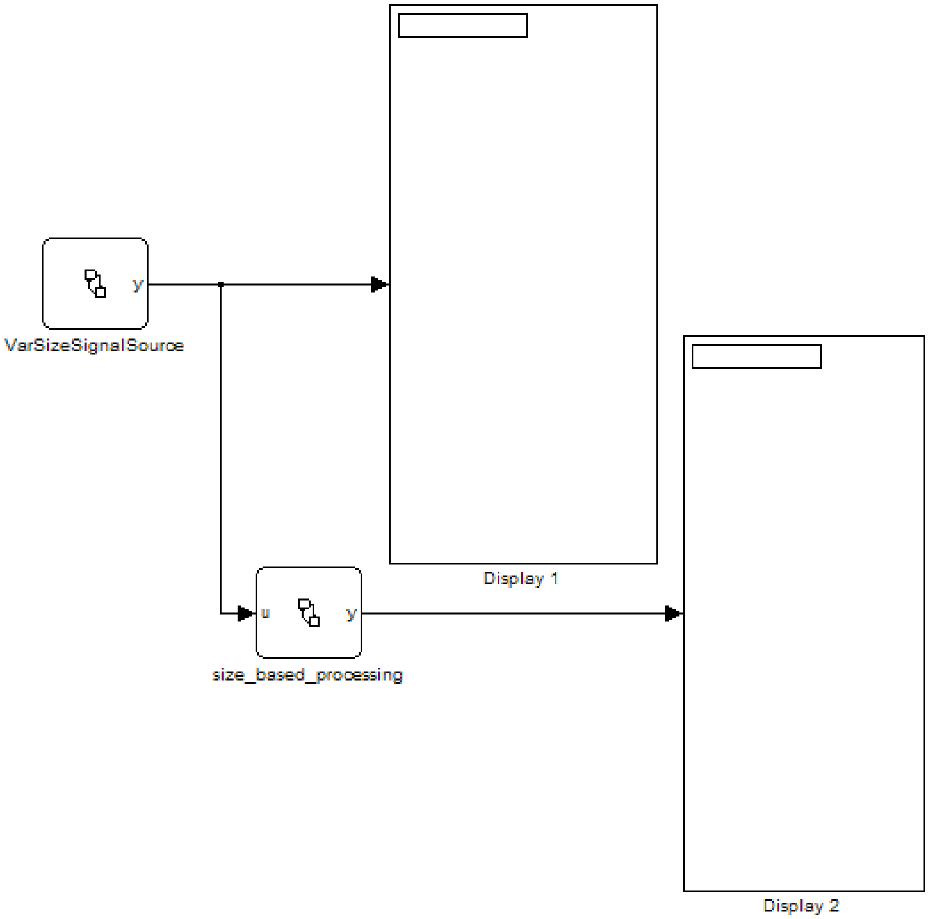
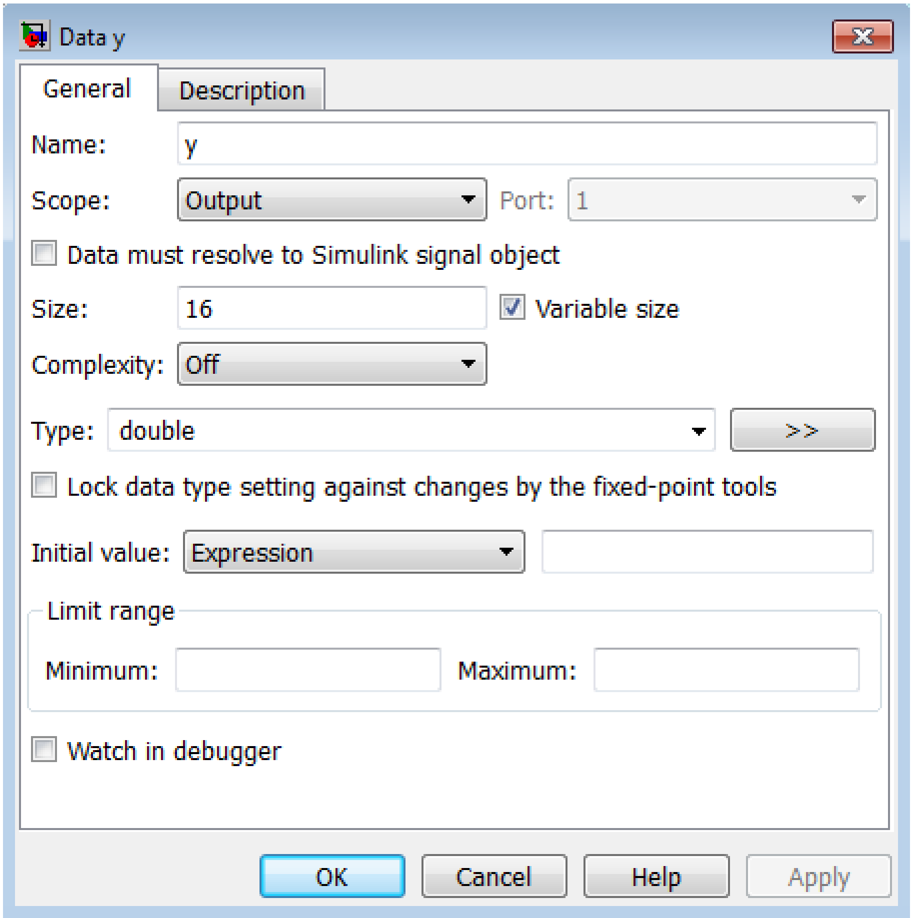


Chart: VarSizeSignalSource

The `VarSizeSignalSource` chart works like a source block. It has no input and one variable-size output `y`:



Name: y

Scope: Output

Port: 1

☐ Data must resolve to Simulink signal object

Size: 16

☑ Variable size

Complexity: Off

Type: double

☐ Lock data type setting against changes by the fixed-point tools

Initial value: Expression

Limit range

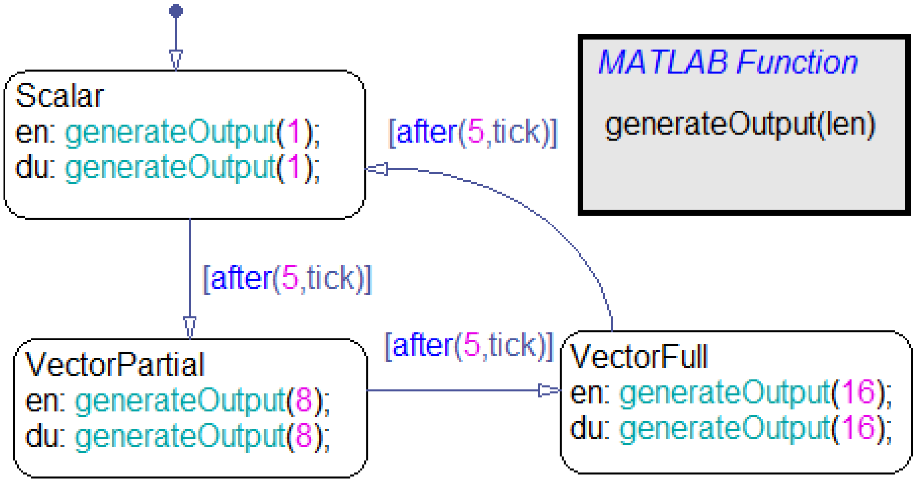
Minimum:

Maximum:

☐ Watch in debugger

For variable-size outputs, you must explicitly specify the size and upper bound for each dimension. In this case, `y` is a vector where the first dimension is assumed to be fixed at size 1 and the second dimension is variable with an upper bound of 16.

This chart uses temporal logic to transition between three states, each generating a different size output:



How the Chart Works with the Variable-Size Output

No states or transitions can read from or write to variable-size data. Therefore, `y` does not appear in any state actions or transition logic. All computations involving variable-size data must occur in MATLAB functions in the chart.

How the MATLAB Function Works with the Variable-Size Output

MATLAB functions access variable-size, chart-level data directly. You do not pass the data as inputs or outputs to the functions. In this chart, the `generateOutput` function adds a different number of elements to the variable-size output `y`, based on how the active state calls it. The function constructs the variable-size vector from a number sequence, then outputs the transpose of the result:

```
function generateOutput(len)
%#codegen
assert(len<=16);
y = (1:len)';
```

MATLAB functions must be able to determine the upper bounds of variable-size data at compile time. In this case, however, the upper bound is `len`, an input for which the model computes the value at run time. To provide this information, the `assert` function specifies an explicit upper bound for `len`, one that matches the upper bound for chart output `y`.

If you do not include the assert statement, you get a compilation error:

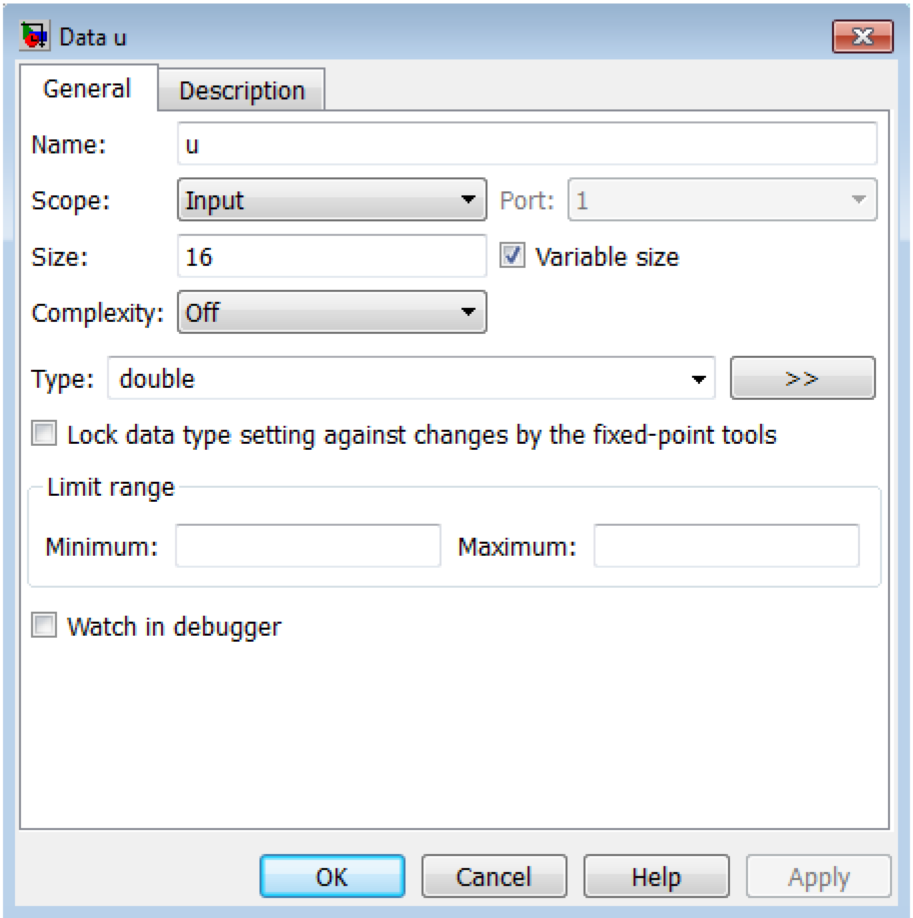
```
Computed maximum size is not bounded.
Static memory allocation requires all sizes to be bounded.
The computed size is [1 x :?].
```

To learn more about how to declare and use variable-size data for MATLAB functions in Stateflow charts, see "How Working with Variable-Size Data Is Different for Code Generation" in the Code Generation from MATLAB documentation.

Chart: size_based_processing

The `size_based_processing` chart computes a variable-size output based on the value of a variable-size input:

- Input `u` is the variable-size signal generated by the `VarSizeSignalSource` chart:



Data u

Name: u

Scope: Input

Port: 1

Size: 16

☑ Variable size

Complexity: Off

Type: double

☐ Lock data type setting against changes by the fixed-point tools

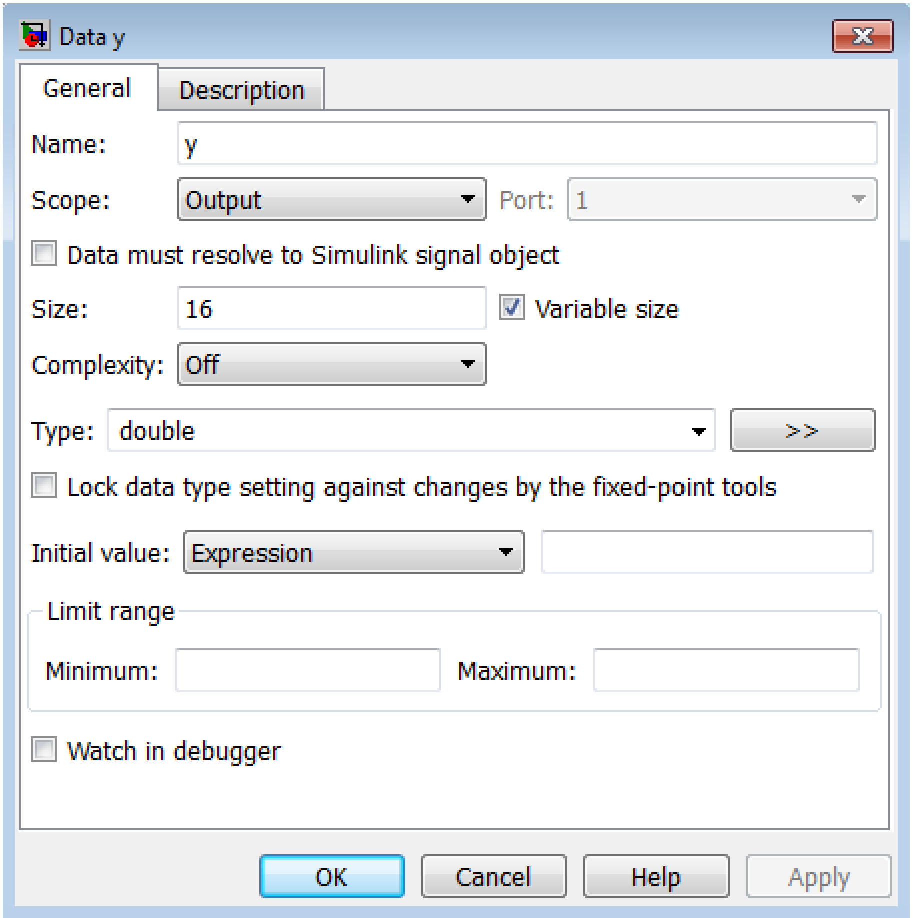
Limit range

Minimum:

Maximum:

☐ Watch in debugger

- Output `y` is a variable-size signal whose size depends on whether `u` is a scalar or vector:



Name: y

Scope: Output Port: 1

☐ Data must resolve to Simulink signal object

Size: 16 ☑ Variable size

Complexity: Off

Type: double >>

☐ Lock data type setting against changes by the fixed-point tools

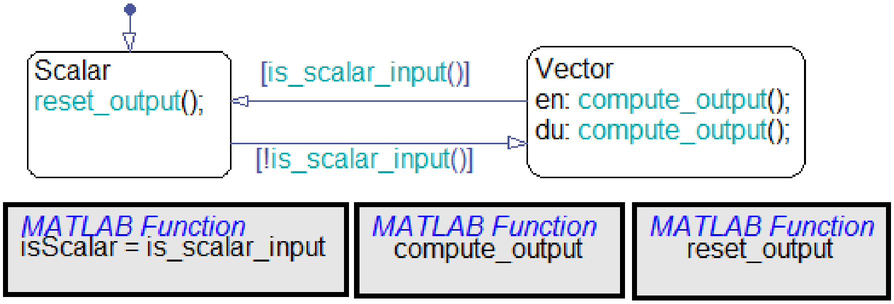
Initial value: Expression

Limit range

Minimum: Maximum:

☐ Watch in debugger

The chart uses three MATLAB functions to evaluate the size of input `u` and generate an associated output `y`:



As in the chart `VarSizeSignalSource`, variable-size data does not appear in state actions or transition logic. Instead, states call MATLAB functions to compute the variable-size output. Transitions call a MATLAB function in a conditional statement to evaluate the variable-size input.

MATLAB Function: is_scalar_input

This function tests whether chart input `u`, the signal generated by chart `VarSizeSignalSource`, is a scalar or vector value:

```
function isScalar = is_scalar_input
%#codegen
isScalar = length(u)==1;
```

MATLAB Function: compute_input

If input `u` is a vector, this function outputs the sine of each of its values:

```
function compute_output
%#codegen
y = sin(u);
```

MATLAB Function: reset_output

If input `u` is a scalar, this function outputs a value of zero:

```
function reset_output
%#codegen
y = 0;
```

Simulating the Model

1. Open the model:

   ```
   sf_varsize_example
   ```

2. Open the chart `VarSizeSignalSource`, but keep the Simulink display blocks in view.

3. Start simulation from the chart.

   The display blocks periodically show 1, 8, and 16 values from the variable-size vector.

Rules for Using Variable-Size Data in Stateflow Charts

- Declare variable-size data as chart inputs and outputs only, not as local data.

  See "Declaring Variable-Size Inputs and Outputs" on page 14-5.

- Perform all computations with variable-size data in MATLAB functions, Simulink functions, and truth tables that use MATLAB action language.

- Do not perform computations with variable-size data directly in states or transitions.

  You can pass the data as inputs and outputs to MATLAB and Simulink functions in your chart from state actions and transition logic. MATLAB functions can also access the chart-level, variable-size data directly (see "Example: Computing Output Based on Size of Input Signal" on page 14-7).

15

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 file 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 15-8.

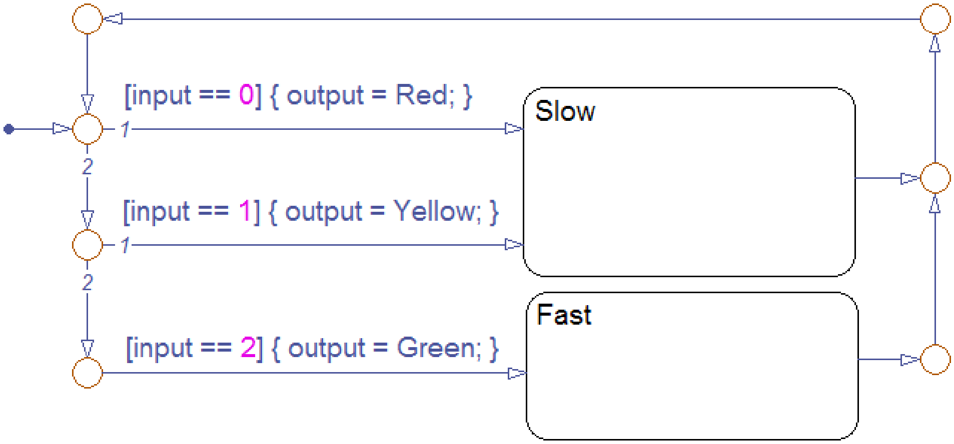
For information on using enumerated data in other blocks of a Simulink model, see "Enumerations and Modeling" in the *Simulink User's Guide*.

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

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
- MATLAB functions (see Chapter 23, “Using MATLAB Functions in Stateflow Charts”)
- Graphical functions (see “Graphical Functions for Reusing Logic Patterns and Iterative Loops” on page 7-30)
- Simulink functions (see Chapter 24, “Using Simulink Functions in Stateflow Charts”)
- Truth Table blocks and truth table functions (see Chapter 22, “Truth Table Functions for Decision-Making Logic”)

You can use enumerated data for simulation and Simulink Coder 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 15-17.

Elements of an Enumerated Data Type Definition

The elements of an enumerated data type definition appear as follows:

```
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 a File” on page 15-8
`enumeration`	Yes	Lists the enumerated values that the data type allows	“Defining an Enumerated Data Type in a File” on page 15-8
`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 Simulink Coder generated code • `getHeaderFile` Enables importing of custom header files that contain enumerated type definitions for Simulink Coder generated code • `addClassNameToEnumNames` Prevents name conflicts with identifiers in Simulink Coder generated code and enhances readability	“Enumerations and Modeling” in the Simulink User’s Guide “Enumerations” in the Simulink Coder 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 Simulink Coder generated code
- Uses a custom header file to prevent the data type from being written out in Simulink Coder generated code
- Adds the name of the data type as a prefix to each enumerated name in Simulink Coder generated code

How to Define Enumerated Data in a Stateflow Chart

In this section...

Tasks for Defining Enumerated Data in a Chart

1. Define an enumerated data type in a file on the MATLAB path.

   This data type is a MATLAB class definition. For details, see *Object-Oriented Programming* in the MATLAB documentation.

   Note For each enumerated type, you must create a new file.

2. Add data of the enumerated type to a chart.

Defining an Enumerated Data Type in a File

1. Create a new file to store the data type definition.

   In the MATLAB Command Window, select **File > New > Enumeration**.

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 15-5 or "Customizing a Simulink Enumeration" in the *Simulink User's Guide*.

4. Save this file on the MATLAB path.

   The name of your file must match exactly the name of your data type. For example, the data type `BasicColors` must reside in a file named `BasicColors.m`.

Tip To add a folder to the MATLAB search path, type `addpath` *`pathname`* at the command prompt.

Adding Enumerated Data to a Chart

1. Add data to your chart and select a scope other than `Constant`.

2 In the **General** pane of the Data properties dialog box, 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 a file on the MATLAB path.

   For example, you can enter `Enum:  BasicColors` in the **Type** field. (See "Defining an Enumerated Data Type in a File" on page 15-8.)

   d Click **Apply**.

3 (Optional) 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 15-17.)

   For example, `BasicColors.Red` is an identifier that uses prefixed notation. (See "Prefixed Notation for Enumerated Values" on page 15-13.)

   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 15-5 or "Customizing a Simulink Enumeration" 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:

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 Simulink Coder code.

Notation for Referring to Enumerated Values in a Chart

In this section...
"Nonprefixed Notation for Enumerated Values" on page 15-12
"Prefixed Notation for Enumerated Values" on page 15-13

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 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 15-13.

Requirements for Using Nonprefixed Notation

The requirements for using nonprefixed notation are:

- The enumerated data type definition is in a 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 a 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 a file on the MATLAB search path.

Example of Prefixed Notation

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 a 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:

1. Open the Stateflow Debugger.
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 26-37.)

Viewing Values of Enumerated Data After Simulation

To view the values of enumerated data after simulation:

1. Open the Model Explorer.
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 **Description** 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 MATLAB 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

Because enumerated values are constants, assigning these values to constant data is redundant and unnecessary. If you try to assign enumerated values to constant data, an error appears.

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 15-12.

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 or the Model Explorer. 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 15-13.

Do not use the ml namespace operator to access enumerated data from the MATLAB base workspace

This operator does not support enumerated data.

Do not enter minimum or maximum values for enumerated data

How the **Minimum** and **Maximum** fields appear in the Data properties dialog box depends on which option you use to define enumerated data.

Type Field Option	Appearance of the Minimum and Maximum Fields
`Enum:  <class name>`	Not available
`<data type expression>` or `Inherit from Simulink`	Available

Leave the **Minimum** and **Maximum** fields empty for enumerated data. The chart ignores any values that you enter in these fields.

Include custom header information for enumerated types in the Configuration Parameters dialog box

If data in your chart uses an enumerated type with a custom header file, include the header information in the **Simulation Target > Custom Code** pane of the Configuration Parameters dialog box. In the **Header file** section, add the following statement:

```
#include "<custom_header_file_for_enum>.h"
```

Suppose that you have three enumerated types in your model that use custom header files: `imported_enum_type1.h`, `imported_enum_type2.h`, and `imported_enum_type3.h`. If you use the three enumerated types for different data in your chart, you can include the header information by using one of these methods:

- Add the following statements in the **Header file** section of the **Simulation Target > Custom Code** pane in the Configuration Parameters dialog box:

  ```
  #include "imported_enum_type1.h"
  #include "imported_enum_type2.h"
  #include "imported_enum_type3.h"
  ```

- Create a separate header file, such as `required_types.h`, that consolidates the list of custom header files:

  ```
  #include "imported_enum_type1.h"
  #include "imported_enum_type2.h"
  #include "imported_enum_type3.h"
  ```

  Then, add the following statement in the **Header file** section of the **Simulation Target > Custom Code** pane in the Configuration Parameters dialog box:

  ```
  #include "required_types.h"
  ```

Best Practices for Using Enumerated Data in a Chart

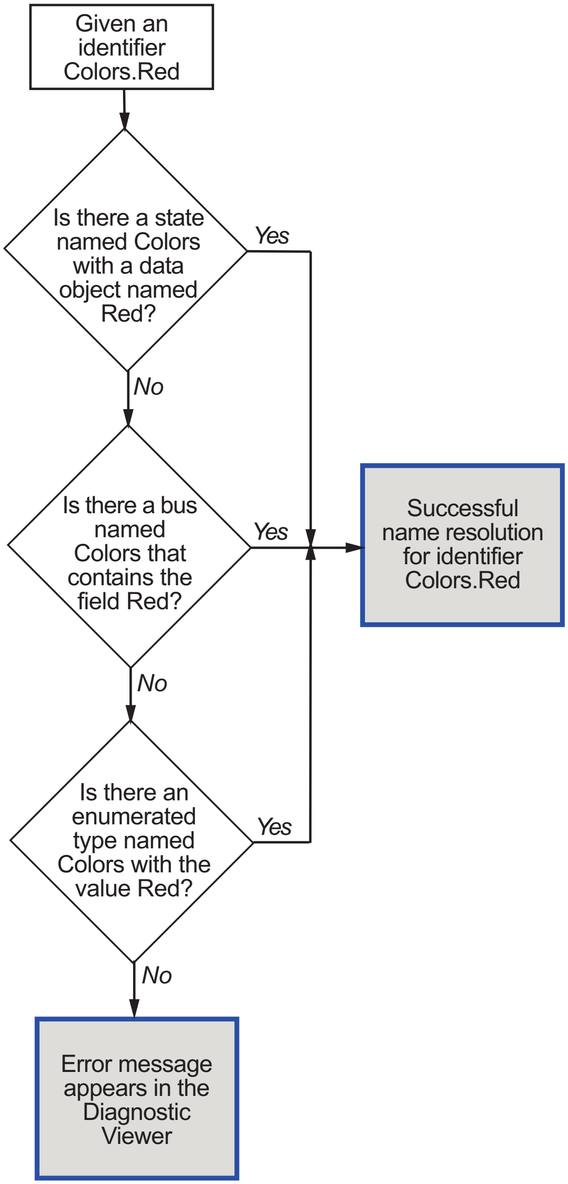
Add prefixes to enumerated names to enhance readability of generated code

If you add prefixes to enumerated names in the generated code, you enhance readability and avoid name conflicts with global symbols. For details, see “Enumerations” in the Simulink Coder documentation.

Use unique identifiers to refer to enumerated values

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`.

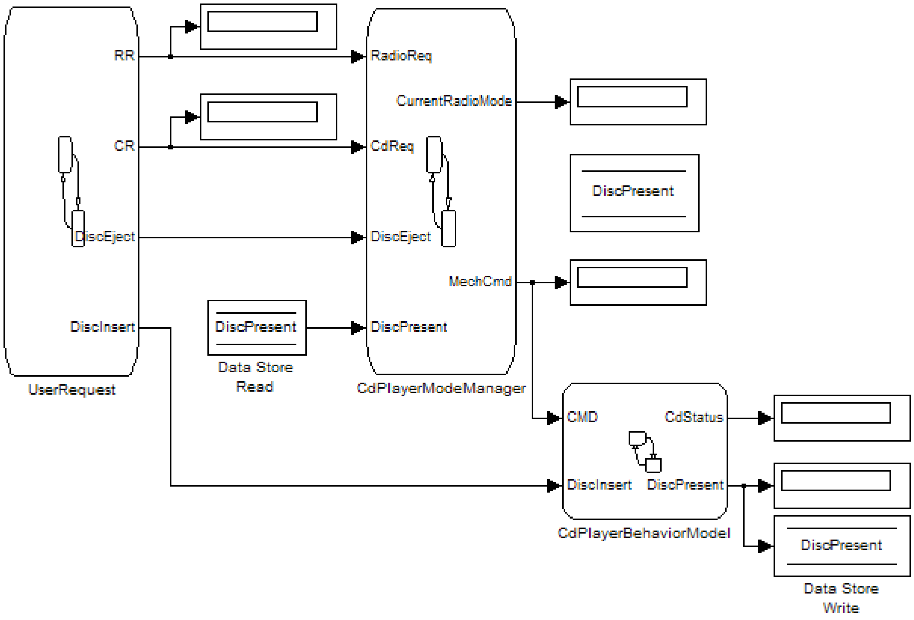

Given an identifier Colors.Red
Is there a state named Colors with a data object named Red?
Yes
No
Is there a bus named Colors that contains the field Red?
Yes
No
Successful name resolution for identifier Colors.Red
Is there an enumerated type named Colors with the value Red?
Yes
No
Error message appears 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 15-27
CdPlayerModeManager chart	Determines whether the CD player operates in CD or radio mode	"How the CdPlayerModeManager Chart Works" on page 15-27
CdPlayerBehaviorModel chart	Describes behavior of the CD player mechanism	"How the CdPlayerBehaviorModel Chart Works" on page 15-31

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:

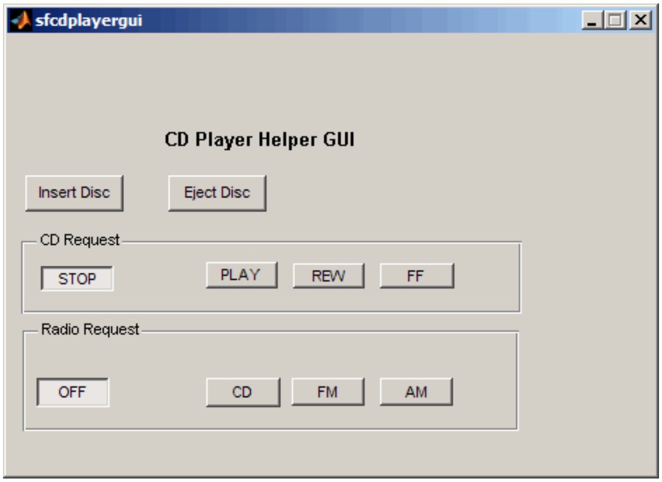
- Enhance readability of data values in each chart.
- Avoid defining a long list of constants, which reduces the amount of data in your model.

Running the CD Player Model

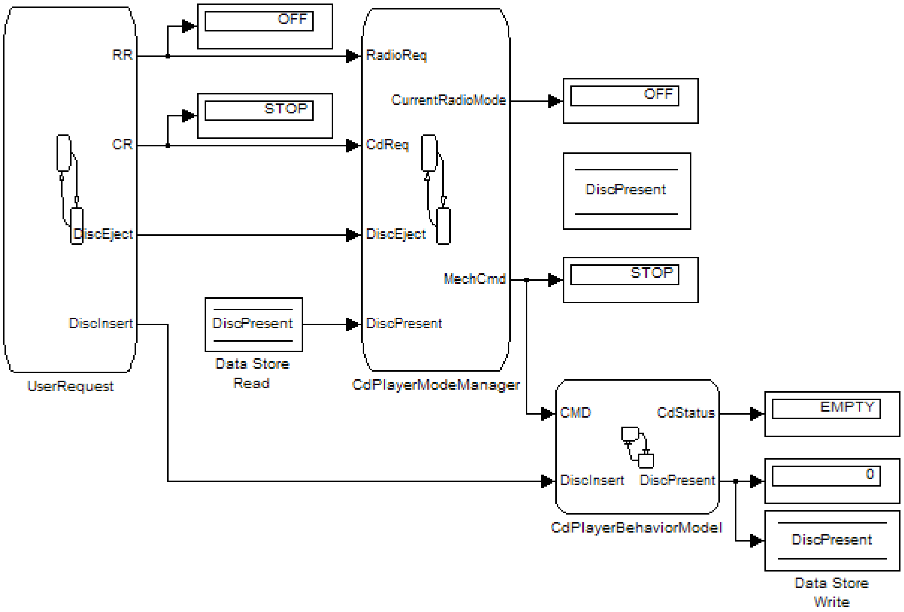
Follow these steps to run the model:

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.

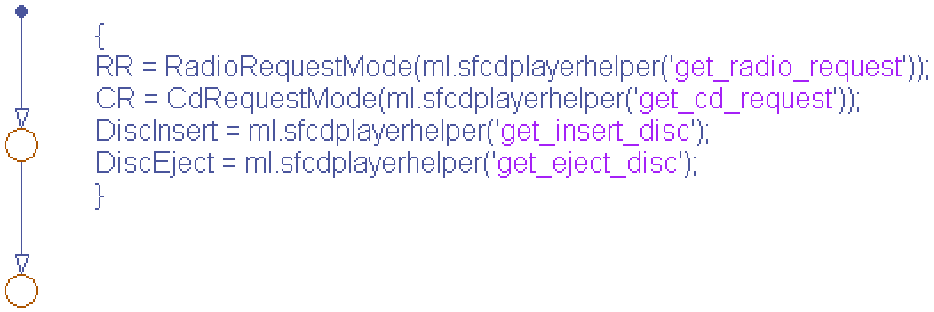
The Display blocks for enumerated data `CR`, `MechCmd`, and `CdStatus` change from `STOP` to `PLAY`.

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 10-42)



```
{
RR = RadioRequestMode(ml.sfcdplayerhelper('get_radio_request'));
CR = CdRequestMode(ml.sfcdplayerhelper('get_cd_request'));
DiscInsert = ml.sfcdplayerhelper('get_insert_disc');
DiscEject = ml.sfcdplayerhelper('get_eject_disc');
}
```

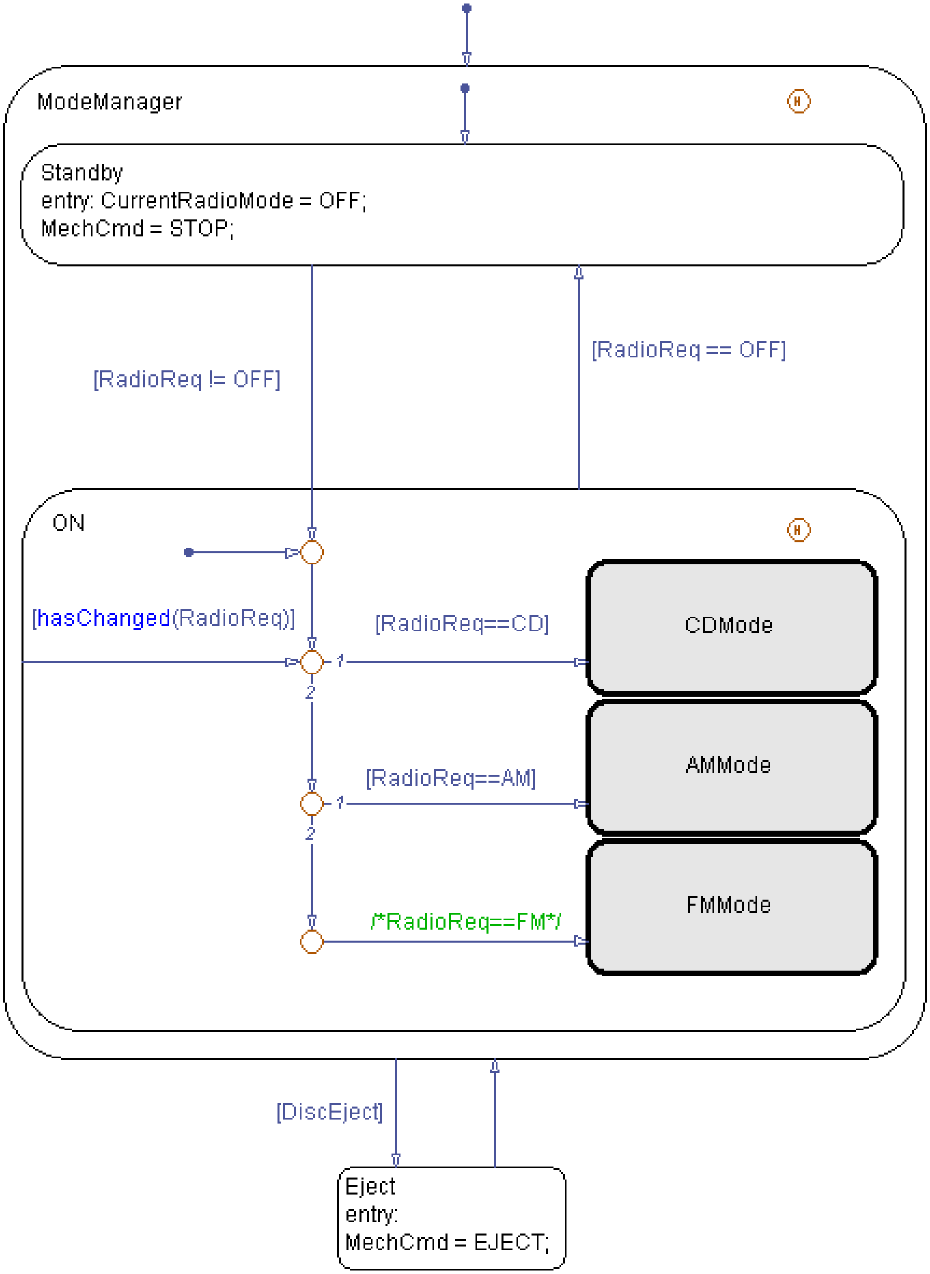
This chart reads user inputs from the CD Player Helper GUI and stores the information as output data.

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 Encapsulate Modal Logic” on page 7-6)
- Change detection (see “Detecting Changes in Data Values” on page 10-83)


ModeManager
Standby
entry: CurrentRadioMode = OFF;
MechCmd = STOP;
[RadioReq != OFF]
[RadioReq == OFF]
ON
[hasChanged(RadioReq)]
[RadioReq==CD]
1
2
CDMode
[RadioReq==AM]
1
2
AMMode
/*RadioReq==FM*/
FMMode
[DiscEject]
Eject
entry:
MechCmd = EJECT;

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 15-30.)

---

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`.

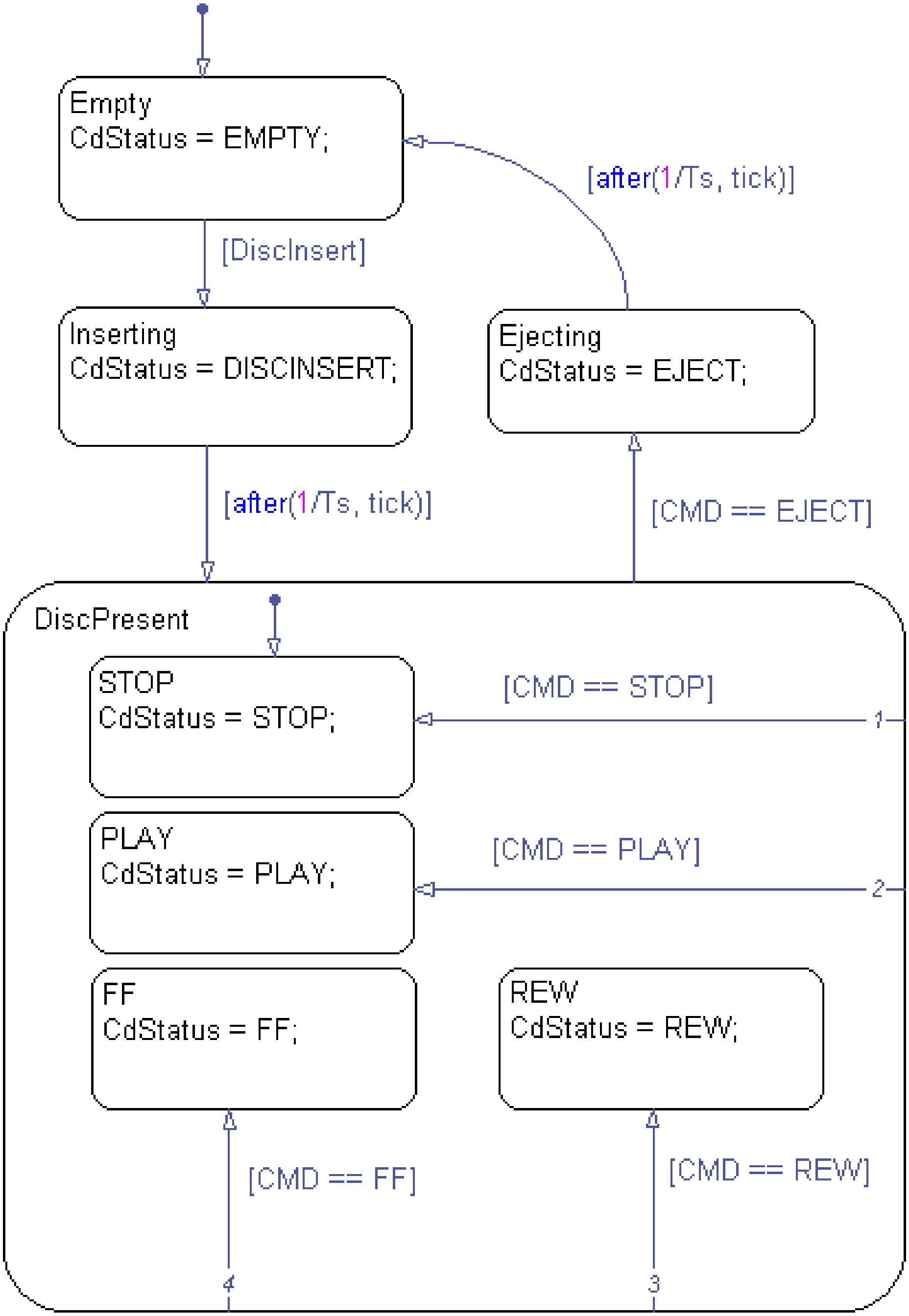
Value of Enumerated Data `RadioReq`	Active Subchart	Purpose of Subchart
`CD`	`CDMode`	Outputs play, rewind, fast forward, and stop commands to the CdPlayerBehaviorModel chart
`AM`	`AMMode`	Sets the CD player to AM radio mode
`FM`	`FMMode`	Sets the CD player to FM radio mode

The `hasChanged` operator detects changes in the value of `RadioReq` with 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 10-63)


Empty
CdStatus = EMPTY;
[DiscInsert]
Inserting
CdStatus = DISCINSERT;
[after(1/Ts, tick)]
Ejecting
CdStatus = EJECT;
[after(1/Ts, tick)]
[CMD == EJECT]
DiscPresent
STOP
CdStatus = STOP;
[CMD == STOP]
1
PLAY
CdStatus = PLAY;
[CMD == PLAY]
2
FF
CdStatus = FF;
REW
CdStatus = REW;
[CMD == FF]
[CMD == REW]
4
3

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.

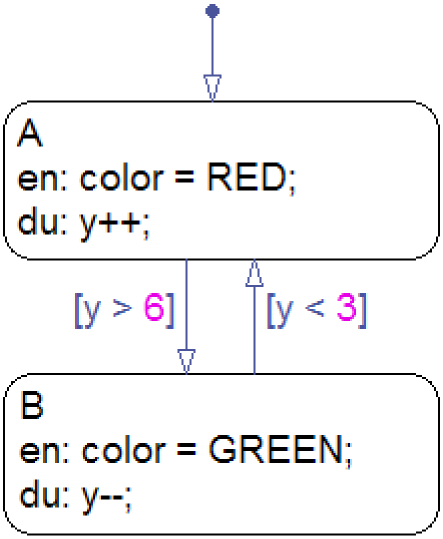
Active State	Value of Enumerated Data `CdStatus`	Behavior of CD Player
`Empty`	`EMPTY`	CD player is empty.
`Inserting`	`DISCINSERT`	CD is being inserted into the player.
`DiscPresent.STOP`	`STOP`	CD is present and stopped.
`DiscPresent.PLAY`	`PLAY`	CD is present and playing.
`DiscPresent.REW`	`REW`	CD is present and rewinding.
`DiscPresent.FF`	`FF`	CD is present and fast forwarding.
`Ejecting`	`EJECT`	CD is being ejected from the player.

Tutorial: Using Enumerated Values for Assignment

In this section...

Goal of the Tutorial

The goal of this tutorial is to build a chart that uses enumerated values in assignment statements.



Building the Chart

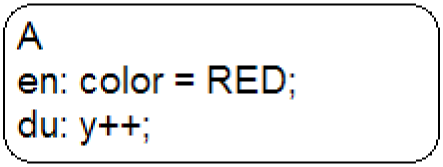
To build the chart, follow these steps.

Adding States and Transitions to the Chart

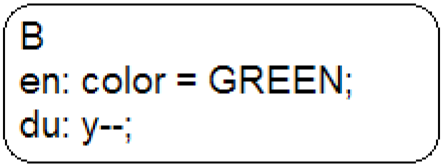
You can add states and transitions to the chart as follows.

1. Type `sfnew` at the command prompt to create a new model with a chart inside.

2. In the chart, add states `A` and `B` to the chart.



```
A
en: color = RED;
du: y++;
```

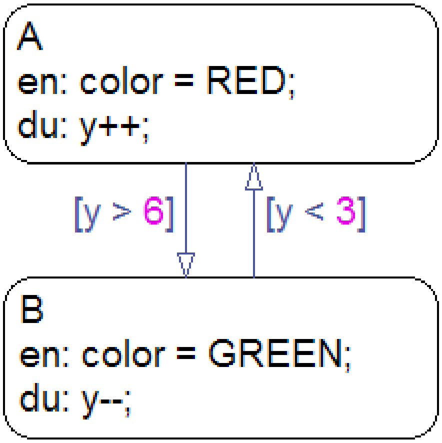


```
B
en: color = GREEN;
du: y--;
```

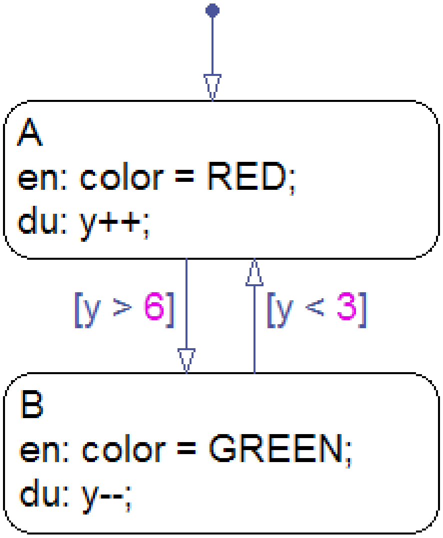
Note You will define the data `color` and `y` in the sections that follow.

---

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 define an enumerated data type for the chart:

1. Create a new file to store the data type definition.

   In the MATLAB Command Window, select **File > New > Enumeration**.

2. Enter these lines in the MATLAB Editor:

   ```
   classdef(Enumeration) TrafficColors < Simulink.IntEnumType
     enumeration
       RED(0)
       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 file as `TrafficColors.m` in a folder on the MATLAB search path.

   The name of your file must match exactly the name of your data type. Therefore, you must use `TrafficColors.m` as the name of your file.

   Tip To add a folder to the MATLAB search path, type `addpath` *`pathname`* at the command prompt.

Adding Enumerated Data to the Chart

To add the enumerated data `color` to the chart:

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 a file in "Defining an Enumerated Data Type for the Chart" on page 15-37.

5. Click **OK**.

Adding Integer Data to the Chart

To add the integer data `y` to the chart:

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.

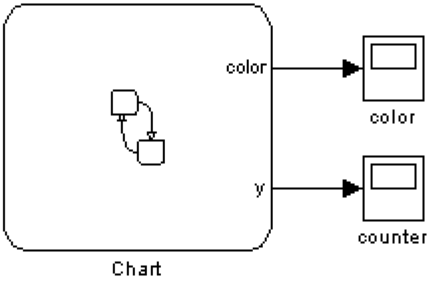
Adding Scopes to View Output

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.



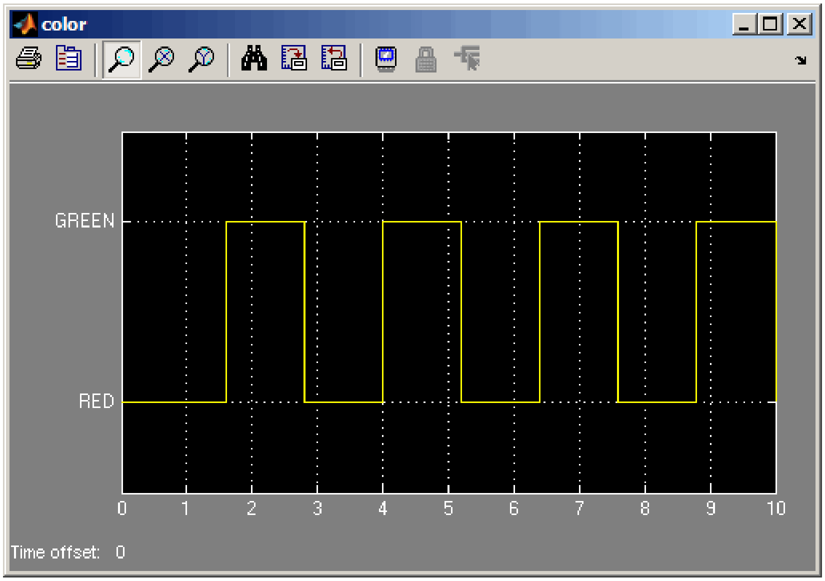
Setting the Sample Time for Simulation

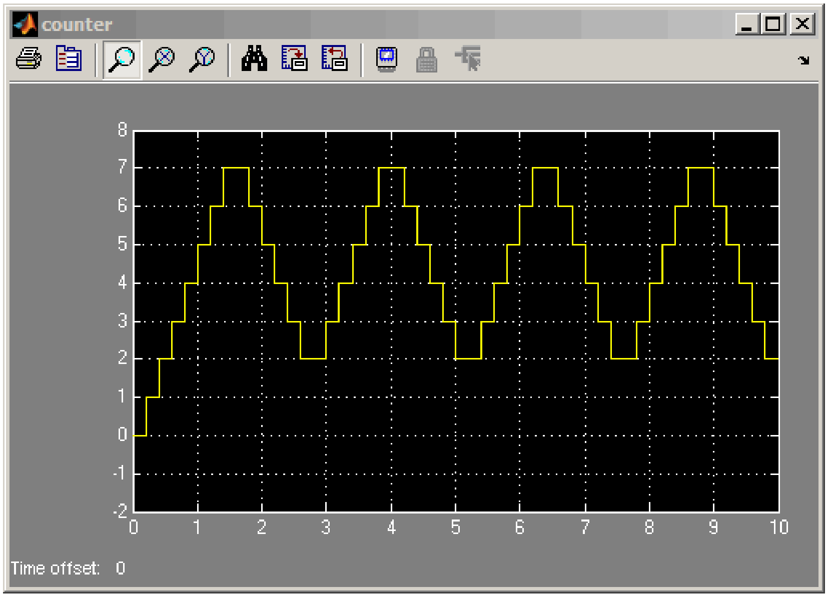
You can set a discrete sample time for simulation using a fixed-step solver. (For details, see “Solvers” in the *Simulink User’s Guide*.)

1. Open the Configuration Parameters dialog box.
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**.

Simulating the Model

Open the Scope blocks. When you simulate the model, you get the following results:


color
GREEN
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.

16

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 16-13).

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. The 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.

- 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.

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 “Purpose of Continuous-Time Variables” on page 16-11.

If your continuous or hybrid system does not contain mode logic, consider using a Simulink model (see “Modeling a Continuous System” in the Simulink User’s Guide).

Models That Demonstrate Continuous-Time Modeling

You can run the following continuous-time models with zero-crossing detection.

Model	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 16-13.
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 models:

1. At the MATLAB prompt, type:

   ```
   demo simulink stateflow
   ```

2. In the Help browser, go to the section titled **Continuous Time Modeling**.
3. Select the model 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 16-14
2	Decide whether to detect zero crossings.	"Task 2: Decide Whether to Enable Zero-Crossing Detection for the Bouncing Ball" on page 16-14
3	Define continuous-time variables.	"Task 3: Define Continuous-Time Variables for Position and Velocity" on page 16-14
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 16-16
5	Add system dynamics.	"Task 5: Add Dynamics for a Free-Falling Ball" on page 16-16
6	Expose continuous states to a Simulink model.	"Task 6: Expose Ball Position and Velocity to the Simulink Model" on page 16-18
7	Validate semantics, based on design considerations for continuous-time simulation.	"Task 7: Validate Semantics of Bouncing Ball Chart" on page 16-19
8	Simulate the chart.	"Task 8: Simulate Bouncing Ball Chart" on page 16-19
9	Debug and revise.	"Task 9: Check for the Bounce" on page 16-21

Configuring a Stateflow Chart to Update in Continuous Time

Continuous updating is a Stateflow chart property. To set this property:

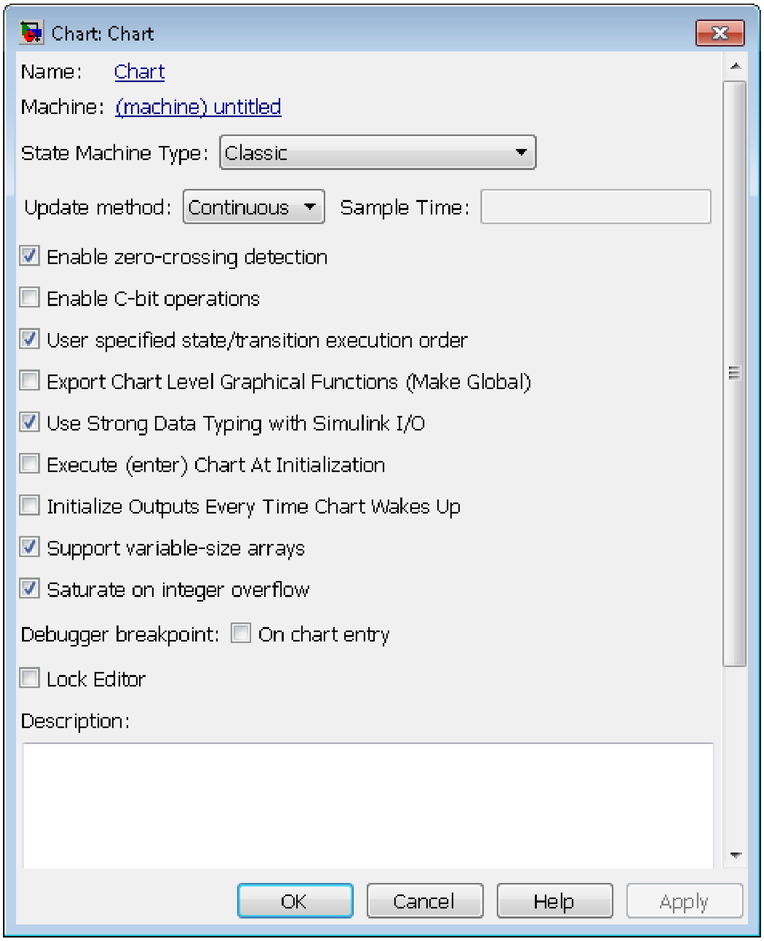
1 Right-click inside a chart and select **Properties** from the context menu.

   The Chart properties dialog box appears.

2 In this dialog box, set the **Update method** to `Continuous`.

   When you set the **Update method** to `Continuous`, the chart:

   - Enables zero-crossing detection



Name: Chart

Machine: (machine) untitled

State Machine Type: Classic

Update method: Continuous Sample Time:

☑ Enable zero-crossing detection

☐ Enable C-bit operations

☑ 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

☑ Support variable-size arrays

☑ Saturate on integer overflow

Debugger breakpoint: ☐ On chart entry

☐ Lock Editor

Description:

- Disables super step semantics

3. Decide whether or not to enable zero-crossing detection, based on considerations described in “When to Enable Zero-Crossing Detection” on page 16-10.

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 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.

Defining Continuous-Time Variables

In this section...

Purpose of 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 chart.

Rules for Using Continuous-Time Variables

Follow these rules when defining and using continuous-time variables:

- Scope can be `Local` or `Output`.
- Define continuous-time variables at the chart level or below in the Stateflow hierarchy.

- Expose continuous state by assigning the local variable to a Stateflow output (see "Exposing Continuous States to a Simulink Model" on page 16-12).

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 16-7.
2. Add local data to your 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 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 "Purpose of Continuous-Time Variables" on page 16-11). 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 16-13.

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 16-6.

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 an Empty State Chart" on page 4-6.

2. 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 might not be able to simulate the physics accurately. For example, the ball might 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**.

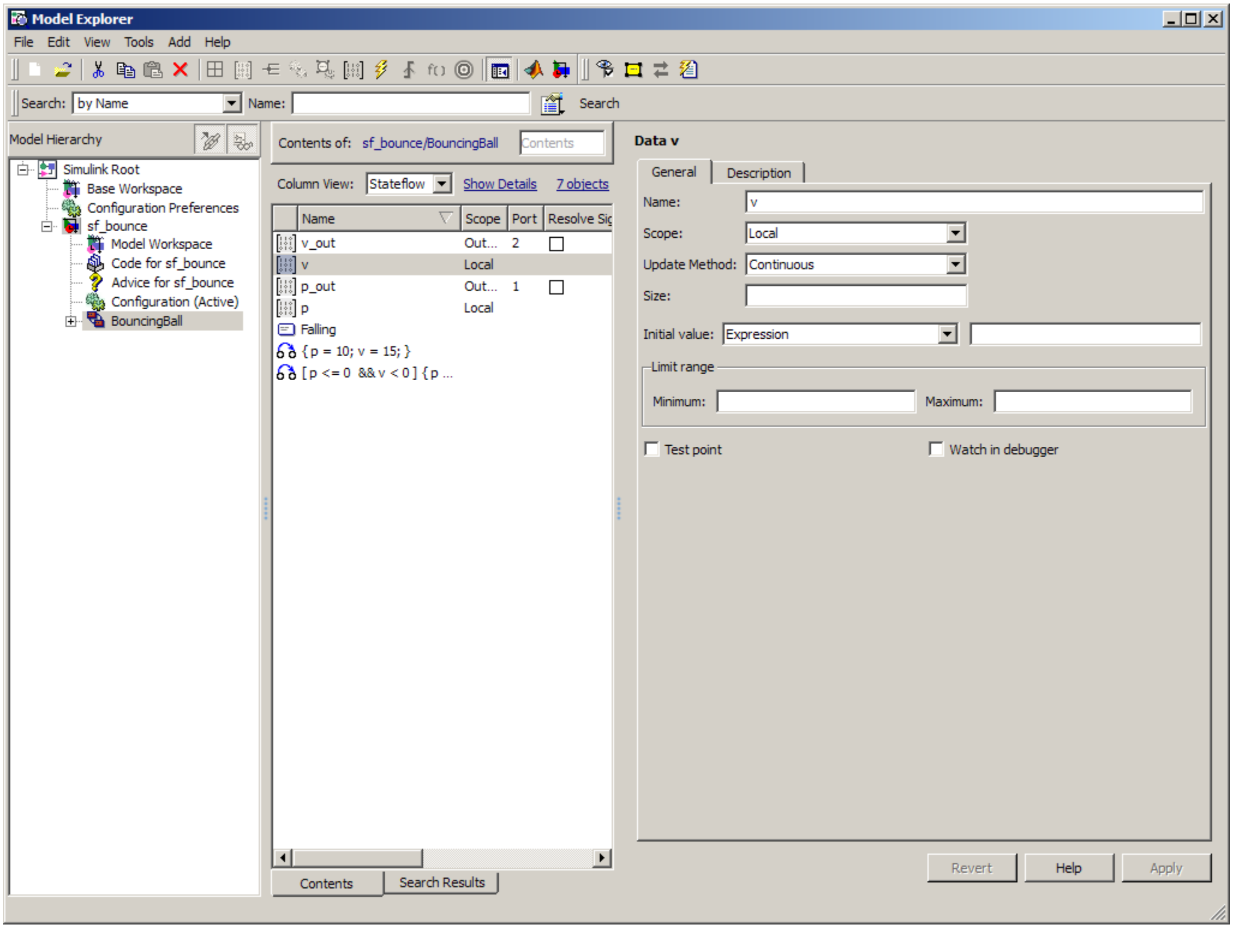
Note For each continuous local variable that you define, the chart implicitly creates its time derivative as a variable of the same name with the suffix `_dot`. In this example, the chart defines `p_dot` as the derivative of position `p` and `v_dot` as the derivative of velocity `v`.

---

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 chart should contain the following data, as viewed in the Model Explorer:



Name	Scope	Port	Resolve Sig
v_out	Out...	2	☐
v	Local		
p_out	Out...	1	☐
p	Local		
Falling			
{p = 10; v = 15; }			
[p <= 0 && v < 0] {p ...			

Name: v

Scope: Local

Update Method: Continuous

Size:

Initial value: Expression

☐ Test point

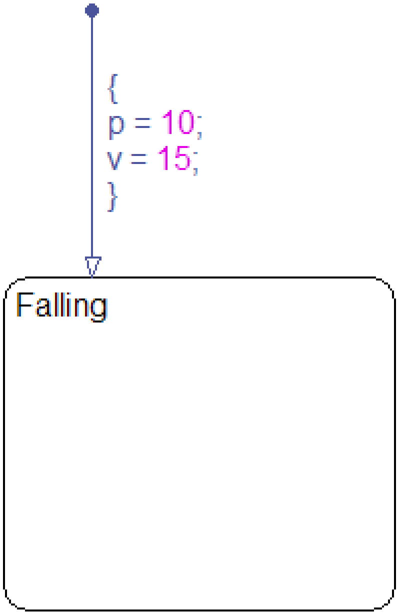
☐ Watch in debugger

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.

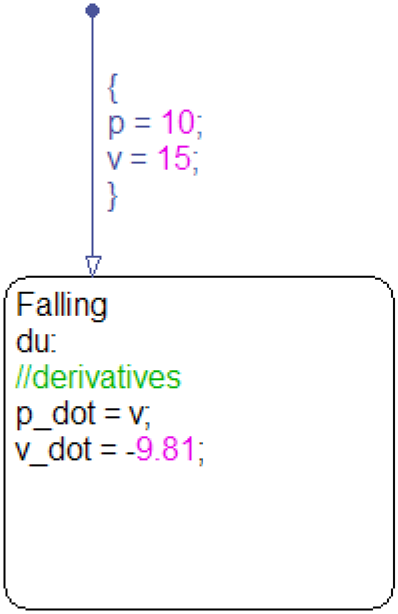
Task 5: Add Dynamics for a Free-Falling Ball

1 Add a state named `Falling` with a default transition. In the default transition, set initial position to 10 meters and initial velocity to 15 meters/second.



```
{
p = 10;
v = 15;
}
```

2 Add a `during` action to the `Falling` state that defines the derivatives of position and velocity, as follows.



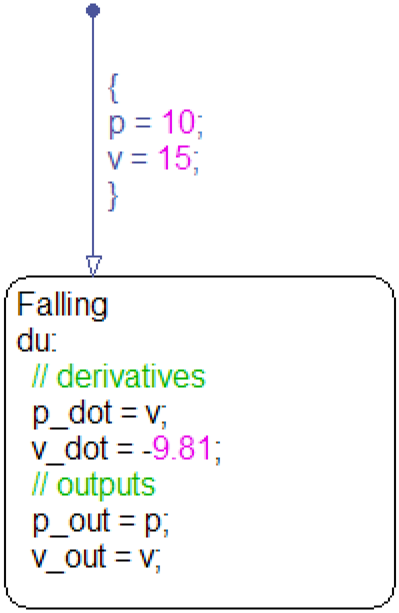
```
{
p = 10;
v = 15;
}
```

```
Falling
du:
//derivatives
p_dot = v;
v_dot = -9.81;
```

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.



```
{
p = 10;
v = 15;
}
```

```
Falling
du:
 // derivatives
 p_dot = v;
 v_dot = -9.81;
 // outputs
 p_out = p;
 v_out = v;
```

Task 7: Validate Semantics of Bouncing Ball Chart

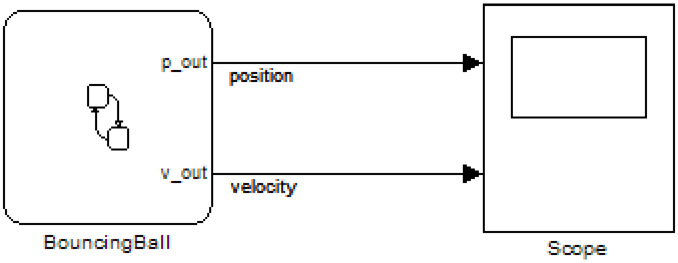
Check semantics against the requirements defined in "Design Considerations for Continuous-Time Modeling in Stateflow Charts" on page 16-26.

This chart meets design requirements:

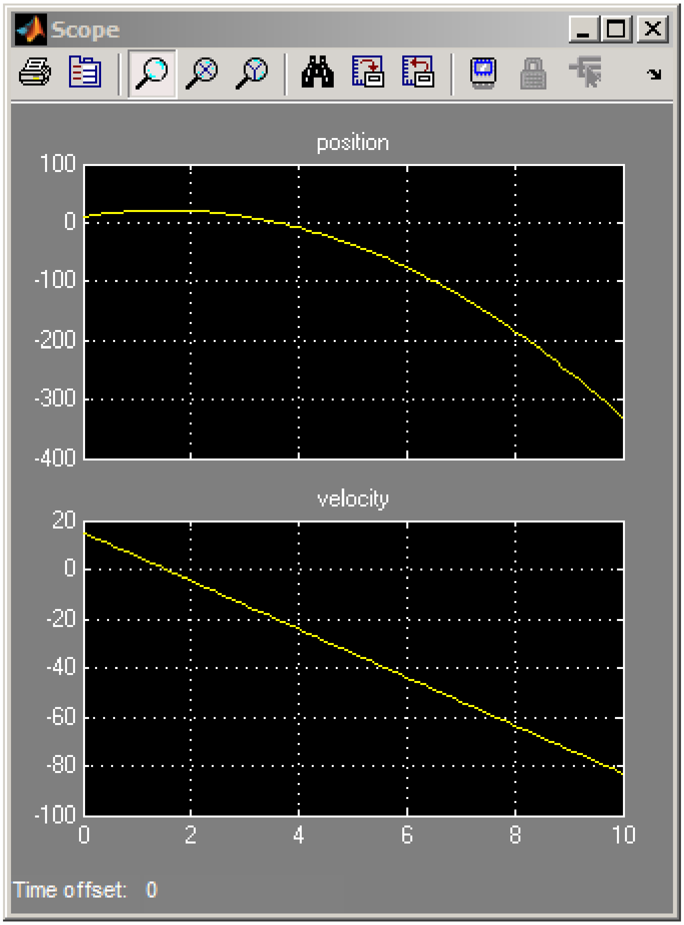
- Assigns values to derivatives `p_dot` and `v_dot` in a `during` action
- Writes to local variables `p` and `v` in a transition action
- Initializes local variables on the default transition
- Does not contain events, inner transitions, event-based temporal logic, or change detection operators

Task 8: Simulate Bouncing Ball Chart

1. Attach each output to a scope.



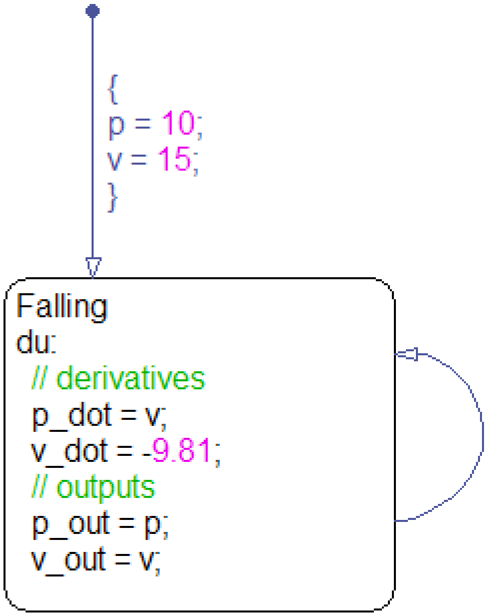
2 Simulate the chart and view the outputs in the scope.



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`.



```
{
p = 10;
v = 15;
}
```

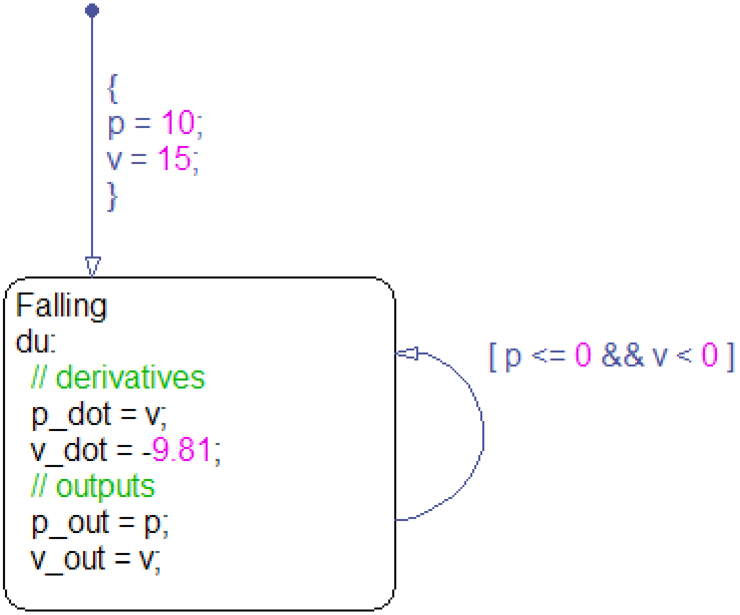
```
Falling
du:
  // derivatives
  p_dot = v;
  v_dot = -9.81;
  // outputs
  p_out = p;
  v_out = v;
```

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.



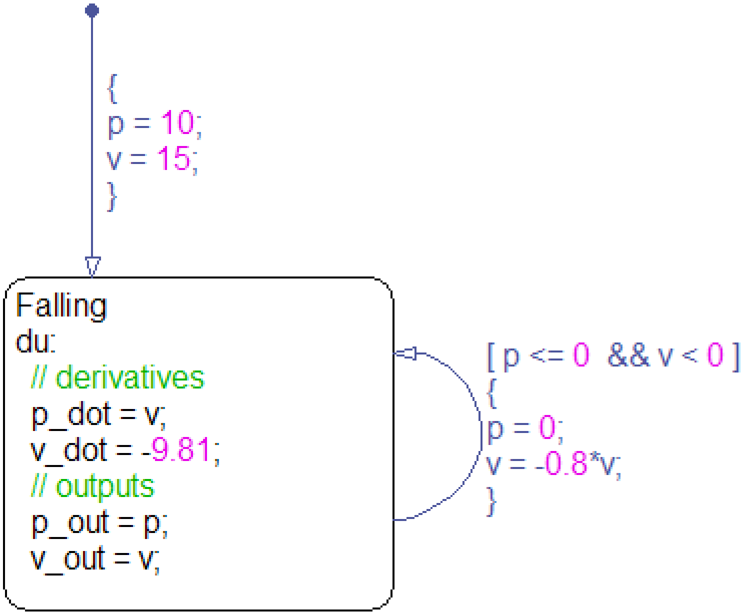
Why not just check for p == 0?

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.

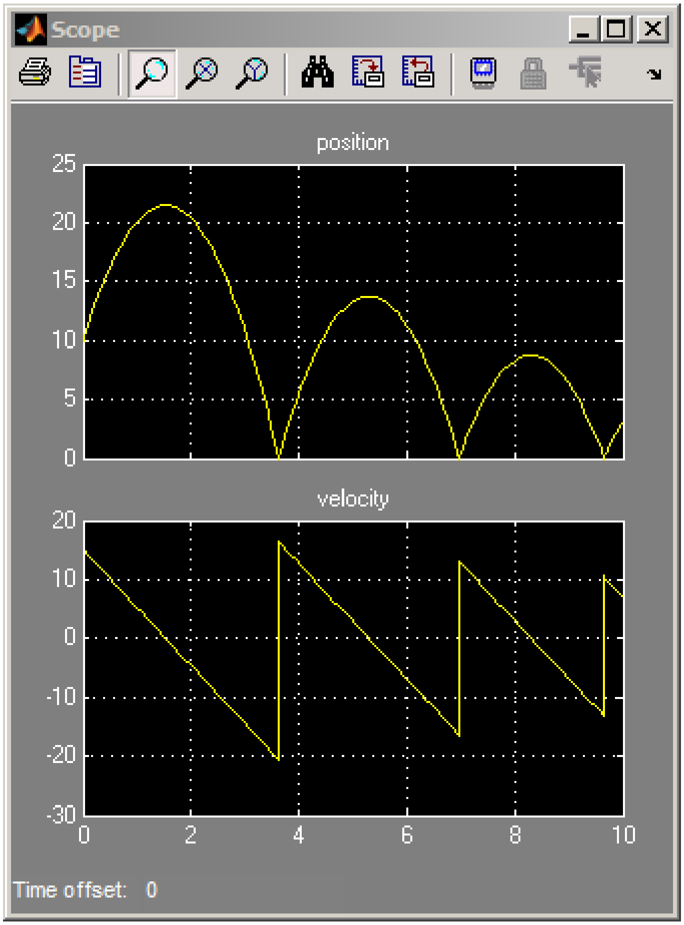


```
{
p = 10;
v = 15;
}
```

```
Falling
du:
  // derivatives
  p_dot = v;
  v_dot = -9.81;
  // outputs
  p_out = p;
  v_out = v;
```

```
[ p <= 0  && v < 0 ]
{
p = 0;
v = -0.8*v;
}
```

4 Simulate the chart again. This time, the scope shows the expected bounce pattern.


Scope
position
25
20
15
10
5
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 maintain the integrity — or *smoothness* — of the results in continuous-time modeling, you must constrain your charts to a restricted subset of Stateflow chart semantics. By restricting the semantics, the 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:

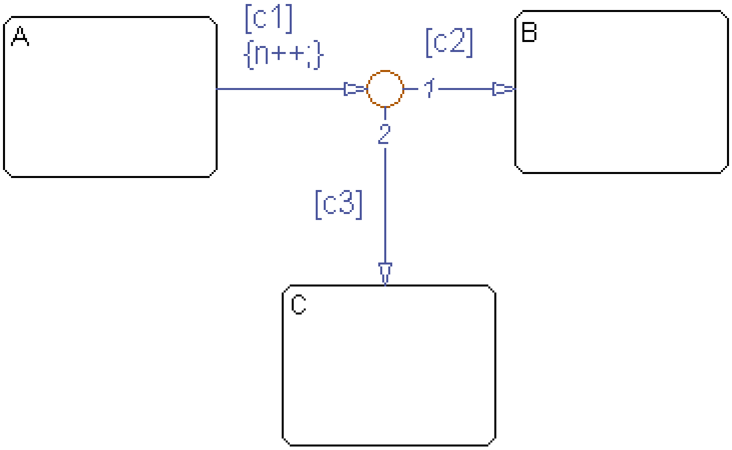
Update local data *only* in transition, `entry`, and `exit` actions

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 the following 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 24, “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 state `during` actions or transition conditions

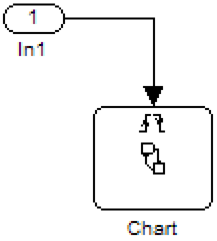
This restriction provides smooth outputs in a major time step by preventing a chart from using values that might no longer be valid in the current minor time step. Instead, a chart computes outputs from local discrete data, local continuous data, and chart inputs.

Use discrete variables to govern conditions in `during` actions

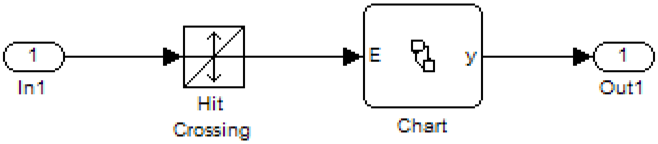
This restriction prevents mode changes from occurring between major time steps. When placed in `during` actions, conditions that affect control flow should be governed by discrete variables because they do not change between major time steps.

Do not use input events in continuous-time charts

The presence of input events makes a chart behave like a triggered subsystem and therefore unable to simulate in continuous time. For example, the following model generates an error if the 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 10-70 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.

Do not modify any SimState values for continuous-time charts

If you load the SimState for a continuous-time chart, you cannot modify the activity of states or any values of chart local or output data. Modifying the SimState of a continuous-time chart is not supported. For more information, see “Rules for Using the SimState of a Chart” on page 12-38.

17

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 17-10.

Fixed-Point Numbers

Fixed-point numbers use integers and integer arithmetic to represent real numbers and arithmetic with the following encoding scheme:

$$V = \tilde{V} = SQ + B$$

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 $\tilde{V}$. This value is the *quantized integer*.

  Q is the actual stored integer value used in representing the fixed-point number. If a fixed-point number changes, its quantized integer, Q, changes but S and B remain unchanged.

- S is a coefficient of Q, or the *slope*.
- B is an additive correction, or 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 equation

$$\tilde{V} = SQ + B$$

for Q and rounding the result to an integer value as follows:

$$Q = \text{round}((V - B)/S)$$

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

$$Q = \text{round}((15.375 - 0.1)/0.5) = 30$$

However, because Q is rounded to an integer, you lose some precision in representing the number 15.375. If you calculate the number that Q actually represents, you now get a slightly different answer.

$$V = \tilde{V} = SQ + B = 0.5 \times 30 + 0.1 = 15.1$$

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 a binary operation: addition, subtraction, multiplication, or division.

The general form for a fixed-point number `x` is $S_x Q_x + B_x$ (see “Fixed-Point Numbers” on page 17-2). Substituting this form for the result and operands in the preceding equation yields this expression:

$$(S_c Q_c + B_c) = (S_a Q_a + B_a) \texttt{<op>} (S_b Q_b + B_b)$$

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 17-28) 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 17-7).

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 17-34.

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

  $$Q_c = ((S_a S_b/S_c)Q_a Q_b + (B_a S_b/S_c)Q_a + (B_b S_a/S_c)Q_b + (B_a B_b - B_c)/S_c)$$

- The operation `c=a/b` implies that

  $$Q_c = ((S_a Q_a + B_a)/(S_c(S_b Q_b + B_b)) - (B_c/S_c))$$

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 17-2 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 17-7.

  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 $S = F \times 2^E$ and $1 \leq F < 2$.

  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 17-3.

  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 17-32 and "Multiplication (*) and Division (/)" on page 17-32.

  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 17-34 and "Fixed-Point Context-Sensitive Constants" on page 17-9.

- 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 "Detecting Overflow for Fixed-Point Types" on page 17-11.

Specifying Fixed-Point Data

You can specify fixed-point data in a chart as follows:

1. From the Stateflow Editor, select **Add > Data**, and then select the scope for the new data object. (See "Scope" on page 8-9 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 8-14.

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 8-5.

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 17-9.

Rules for Specifying Fixed-Point Word Length

- For chart-level data of the following scopes, word length can be any integer between 0 and 128.
  - Input
  - Output
  - Parameter
  - Data Store Memory
- For other Stateflow data, word length can be any integer between 0 and 32.
- You can explicitly pass chart-level data with word lengths up to 128 bits as inputs and outputs of the following functions:
  - MATLAB functions
  - Simulink functions
  - Truth table functions that use MATLAB action language

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 these factors:

- The operator
- The data types in the context
- The value of the constant

The algorithm computes a type that provides 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 17-29.

   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 17-9 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 "Detecting Overflow for Fixed-Point Types" on page 17-11 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 17-35.

Detecting Overflow 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. Open the Configuration Parameters dialog box and go to the **Simulation Target** pane.
2. Select **Enable debugging/animation** and **Enable overflow detection (with debugging)**.

   For descriptions of these check boxes, see "Speeding Up Simulation" on page 25-16.
3. Click **Execute** to build the simulation target.
4. Open the Stateflow debugger.

   For more information, see "Using the Stateflow Debugger" on page 26-2.
5. In the Debugging window, select **Data Range**.

   See "Options for Error Checking in the Debugger" on page 26-19 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 17-7) must match similar fields that you enter for fixed-point data in a Simulink model. See "Tutorial: Using Fixed-Point Chart Inputs" on page 17-14 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 dialog box of the Constant block by using the **Output data type** parameter.

- 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.

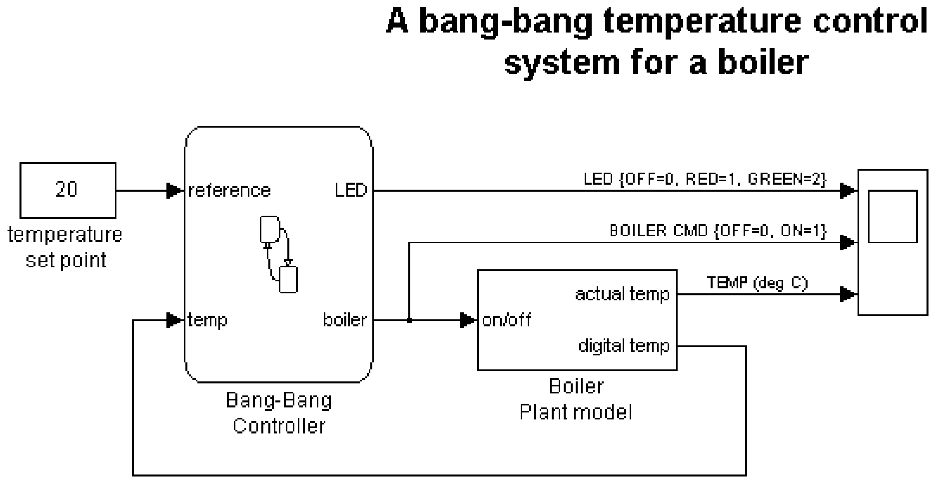
  For example, you can set the Constant block to inherit its type from the Stateflow **Input to Simulink** port that it supplies. To do so, select `Inherit via back propagation` for the **Output data type** parameter in the block dialog box.

Tutorial: Using Fixed-Point Chart Inputs

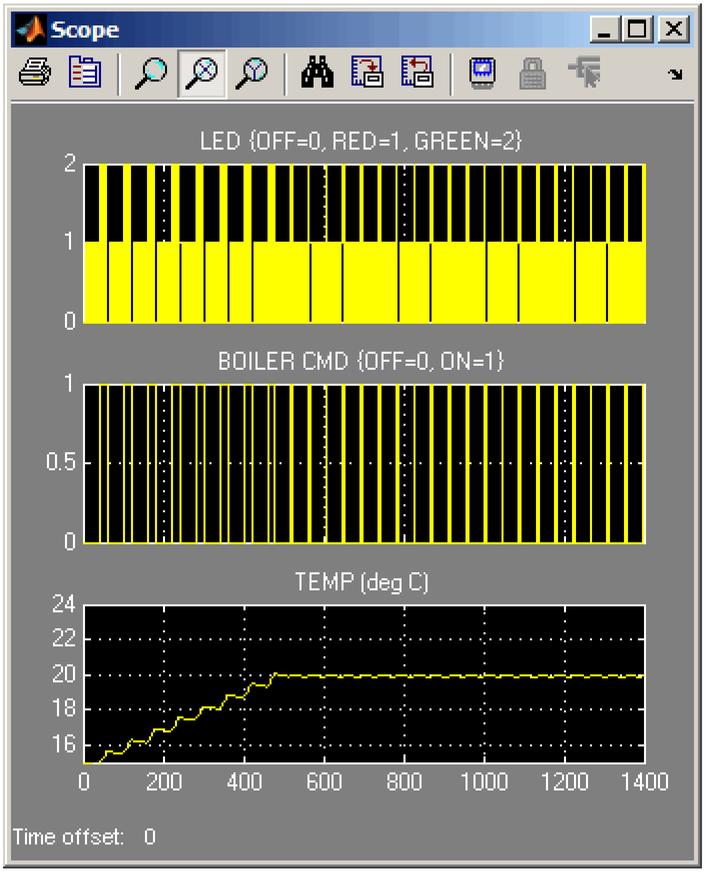
In this section...

Running the Fixed-Point "Bang-Bang Control" Model

Stateflow software includes demo models with applications of fixed-point data. For this example, load the model by typing `sf_boiler` at the MATLAB command prompt.



When you simulate the model, you get these results:

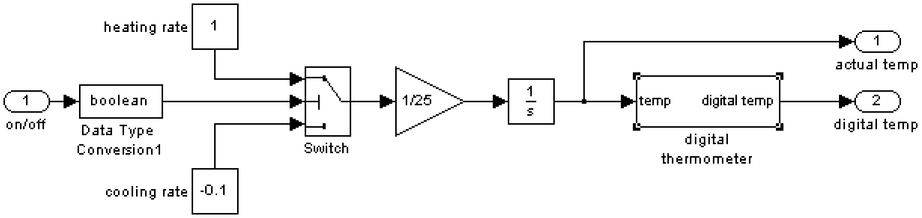


Exploring the Fixed-Point "Bang-Bang Control" Model

To explore the model, follow these steps:

1 Double-click the Boiler Plant model subsystem block.

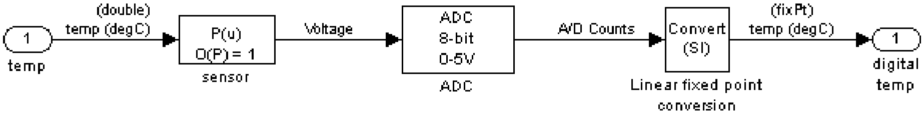
The subsystem appears.



The Boiler Plant model subsystem simulates the temperature reaction of the boiler to periods of heating or cooling dictated by the Stateflow block. Depending on the Boolean value coming from the Controller, a temperature increment (+1 for heating, –0.1 for cooling) is added to the previous boiler temperature. The resulting boiler temperature is sent to the digital thermometer subsystem block.

2 In the Boiler Plant model subsystem, double-click the digital thermometer subsystem block.

The subsystem appears.



The digital thermometer subsystem produces an 8-bit fixed-point representation of the input temperature with the blocks described in the sections that follow.

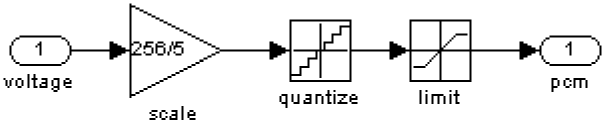
sensor Block

The sensor block converts input boiler temperature *(T)* to an intermediate analog voltage output *V* with a first-order polynomial that gives this output:

$$V = 0.05 \times T + 0.75$$

ADC Block

Double-click the ADC block to reveal these contents:



The ADC subsystem digitizes the analog voltage from the sensor block by multiplying the analog voltage by 256/5, rounding it to its integer floor, and limiting it to a maximum of 255 (the largest unsigned 8-bit integer value). Using the value for the output V from the sensor block, the new digital coded temperature output by the ADC block, $T_{digital}$, is given by this equation:

$$T_{digital} = (256/5) \times V = (256 \times 0.05/5) \times T + (256/5) \times 0.75$$

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:

$$V = SQ + B$$

Therefore,

$$Q = (V - B)/S = (1/S) \times V + (-1/S) \times B$$

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:

$$(1/S) \times V + (-1/S) \times B = (256 \times 0.05/5) \times T + (256/5) \times 0.75$$

Since T is the real-world value for the environment temperature, the above equation implies these relations:

$$V = T$$

and

$$1/S = (256 \times 0.05)/5$$

$$S = 5/(256 \times 0.05) = 0.390625$$

and

$$(-1/S) \times B = (256/5) \times 0.75$$

$$B = -(256/5) \times 0.75 \times 5/(256 \times 0.05) = -0.75/0.05 = 15$$

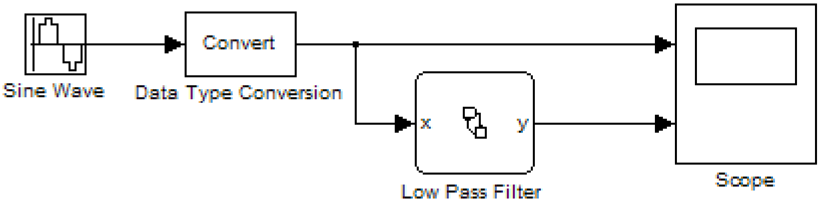
By setting T_{digital} to be a fixed-point data as the output of the Linear fixed point conversion block and the input of the Stateflow block Bang-Bang Controller, the Stateflow chart interprets and processes this data automatically in an 8-bit environment with no need for any explicit conversions.

Tutorial: Using Fixed-Point Parameters and Local Data

In this section…

Goal of the Tutorial

In the sections that follow, you build a model that uses fixed-point parameters and local data in a Stateflow chart. In this model, the chart acts as a low-pass Butterworth filter:



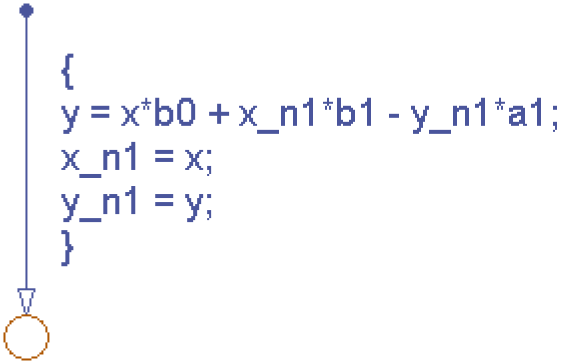
Building this model requires a Signal Processing Toolbox™ license.

Building the Fixed-Point Butterworth Filter

In this section, you create a stateless flow chart that accepts one input and provides one output.

1 At the MATLAB prompt, type `sfnew` to create a new model with an empty chart.

2 In your chart, add a flow graph with a single branch:



```
{
y = x*b0 + x_n1*b1 - y_n1*a1;
x_n1 = x;
y_n1 = y;
}
```

The values `b0`, `b1`, and `a1` are the coefficients of the low-pass Butterworth filter. For more information about the filter coefficients, see "Defining the Model Callback Function" on page 17-20.

3 Add the following data to your chart:

Data Name	Scope	Type
`x`	`Input`	`Inherit:Same as Simulink`
`y`	`Output`	`fixdt(1,16,10)`
`x_n1`	`Local`	`fixdt(1,16,12)`
`y_n1`	`Local`	`fixdt(1,16,10)`
`b0`	`Parameter`	`fixdt(1,16,15)`
`b1`	`Parameter`	`fixdt(1,16,15)`
`a1`	`Parameter`	`fixdt(1,16,15)`

4 Save your model.

Defining the Model Callback Function

In this section, you define a preload callback for the model. This callback function computes the values for `b0`, `b1`, and `a1` in the chart.

1. Open the Model Properties dialog box by selecting **File > Model Properties** in the model window.

2. In the **Callbacks** tab, select **PreLoadFcn**.

3. Enter the following MATLAB code for the preload function:

   ```
   Fs = 1000;
   Fc = 50;
   [B,A] = butter(1,2*pi*Fc/(Fs/2));
   b0 = B(1);
   b1 = B(2);
   a1 = A(2);
   ```

   In the code:

   - The sampling frequency `Fs` is 1000 Hz.
   - The cutoff frequency `Fc` is 50 Hz.
   - The `butter` function constructs a first-order low-pass Butterworth filter with a normalized cutoff frequency of `(2*pi*Fc/(Fs/2))` radians per second. The function output `B` contains the numerator coefficients of the filter in descending powers of z. The function output `A` contains the denominator coefficients of the filter in descending powers of z.

4. Click **OK** to close the dialog box.

5. Save your model.

Adding Other Blocks to the Model

In this section, you add the remaining blocks to the model.

1. Open the Simulink Library Browser.

2. From the Simulink/Sources library, add a Sine Wave block with the following parameter settings to the model:

Parameter	Setting
Sine type	`Time based`
Time	`Use simulation time`

Parameter	Setting
Amplitude	`1`
Bias	`0`
Frequency	`2*pi*Fc`
Phase	`0`
Sample time	`1/Fs`
Interpret vector parameters as 1-D	`On`

The Sine Wave block provides the signal that you want to filter using the Stateflow chart. This block outputs a floating-point signal.

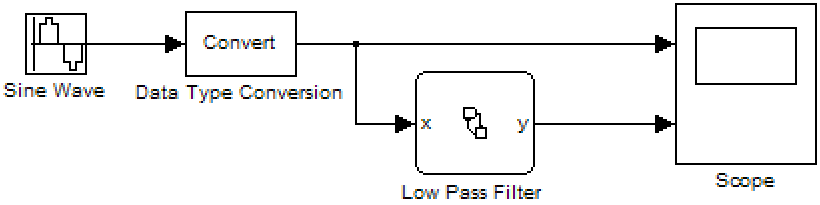
3 From the Simulink/Signal Attributes library, add a Data Type Conversion block with the following parameter settings to the model:

Parameter	Setting
Output minimum	`[]`
Output maximum	`[]`
Output data type	`fixdt(1,16,14)`
Lock output data type setting against changes by the fixed-point tools	`Off`
Input and output to have equal	`Real World Value (RWV)`
Integer rounding mode	`Floor`
Saturate on integer overflow	`Off`
Sample time	`-1`

The Data Type Conversion block converts the floating-point signal from the Sine Wave block to a fixed-point signal. By converting the signal to a fixed-point type, the model can simulate using less memory.

4 From the Simulink/Sinks library, add a Scope block to the model.

5 Connect and label the blocks as follows:



6 Close the Library Browser and save your model.

Setting Configuration Parameters for the Model

In this section, you specify solver and diagnostic options for simulation.

1 Open the Configuration Parameters dialog box.

2 In the **Solver** pane, set the following parameters:

Parameter	Setting
Stop time	`0.1`
Type	`Fixed-step`
Solver	`discrete (no continuous states)`
Fixed-step size (fundamental sample time)	`1/Fs`

Because none of the blocks in your model have a continuous sample time, a discrete solver is appropriate. For more information, see "Solver Pane" in the *Simulink Graphical User Interface* documentation.

3 In the **Diagnostics > Data Validity** pane, set the following parameters:

Parameter	Setting
Signals > Signal resolution	`Explicit and warn implicit`
Parameters > Detect precision loss	`none`

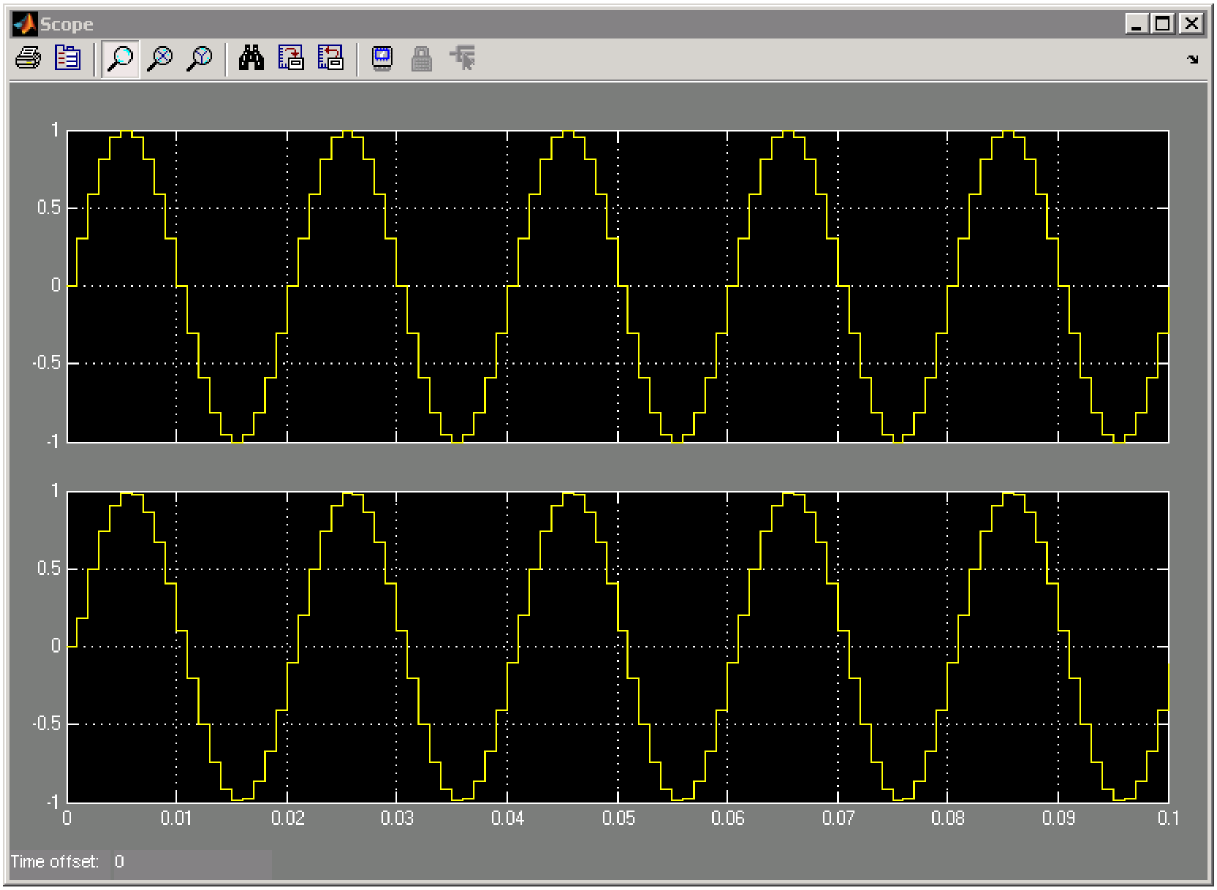
By setting the diagnostic settings for data validity, you control what types of warnings or errors appear during simulation. For more information, see "Diagnostics Pane: Data Validity" in the *Simulink Graphical User Interface* documentation.

4 Click **OK** to close the dialog box.

5 Save and close your model.

Running the Model

When you reopen and simulate the model, you see these results in the scope:



The top signal shows the fixed-point version of the sine wave input to the chart. The bottom signal corresponds to the filtered output from the chart. The filter removes high-frequency values from the signal but allows low-frequency values to pass through the chart unchanged.

Operations with Fixed-Point Data

In this section...
"Supported Operations with Fixed-Point Operands" on page 17-26
"Promotion Rules for Fixed-Point Operations" on page 17-28
"Assignment (=, :=) Operations" on page 17-34
"Fixed-Point Conversion Operations" on page 17-42
"Automatic Scaling of Stateflow Fixed-Point Data" on page 17-44

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 19-4. 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 19-4. 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
`!a`	Logical NOT

Example	Description
`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 17-35.
`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 17-26.
`a &= expression`	Equivalent to `a = a & expression` (bit operation). See operation `a & b` in "Binary Operations" on page 17-26.

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 17-35.

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 Set all available integer sizes because they affect code generation. The integer sizes 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 17-29. 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 17-31.

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 (-)" on page 17-32
- "Multiplication (*) and Division (/)" on page 17-32
- "Relational Operations (>, <, >=, <=, ==, -=, !=, <>)" on page 17-32
- "Logical Operations (&, |, &&, ||)" on page 17-33

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 17-29. 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 17-29.

Note Because nonzero biases are computationally very expensive, those biases 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 17-26 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:

$$S_b\ (newQ_a) = S_a Q_a$$

$$newQ_a = (S_a S_b)\ Q_a = (2^{-4}/2^{-2})\ 701 = 701/4 = 175$$

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 17-9). 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 =" on page 17-34.
- "Assignment Operator :=" on page 17-35
- "When to Use the := Operator Instead of the = Operator" on page 17-35
- "Example of Using the := Operator for Addition and Subtraction" on page 17-35
- "Example of Using the := Operator for Multiplication" on page 17-39
- "Example of Using the := Operator for Division" on page 17-40
- ":= Assignment and Context-Sensitive Constants" on page 17-42

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 17-42). 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 17-42). 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:

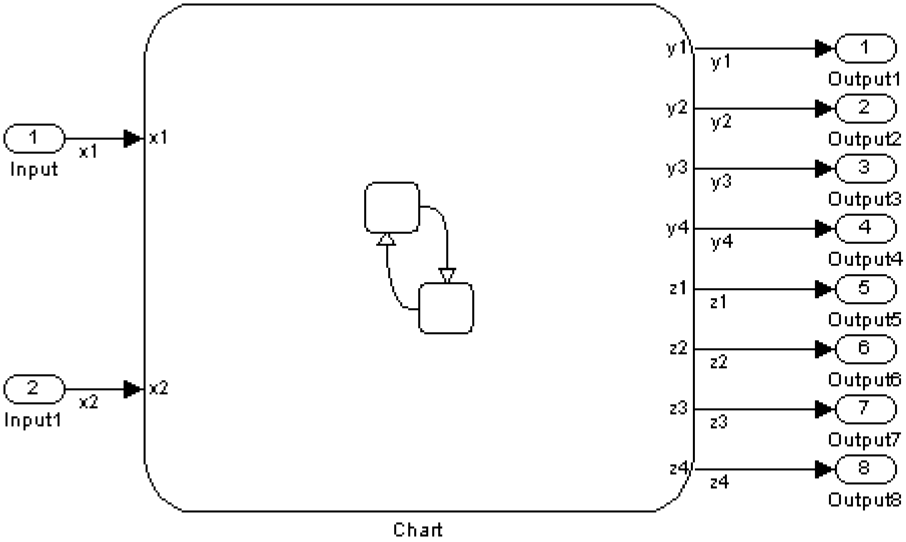
- Arithmetic operations where you want to avoid overflow
- Multiplication and division operations where you want to retain precision

---

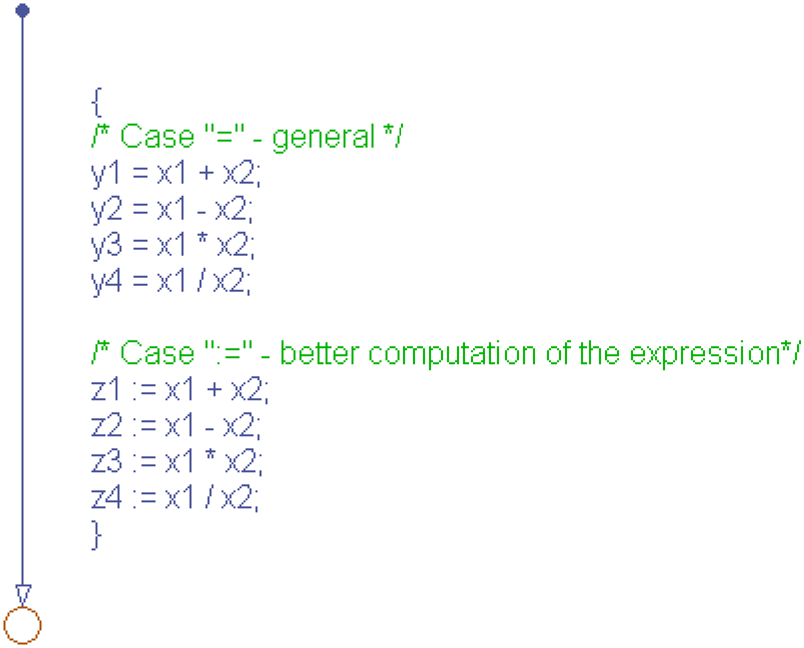
Caution Using the := assignment operator to produce a more accurate result can generate code that is less efficient than the code you generate using the normal fixed-point promotion rules.

Example of Using the := Operator for Addition and Subtraction

This 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.



```
{
/* Case "=" - general */
y1 = x1 + x2;
y2 = x1 - x2;
y3 = x1 * x2;
y4 = x1 / x2;

/* Case ":=" - better computation of the expression*/
z1 := x1 + x2;
z2 := x1 - x2;
z3 := x1 * x2;
z4 := x1 / x2;
}
```

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 doc_sf_colon_equal_step(void)
```

```
{
  /*  Case "=" - general  */
  y1 = x1 + x2;
  y2 = 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 << 3) / (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.

Assume that the integer word size for production targets is 16 bits. To learn how to change the integer word size for a target, see “Setting the Integer Word Size for a Target” on page 17-29.

Because the target `int` size is 16 bits, you can avoid overflow by using the := operator instead of the = operator. For example, assume that the inputs have these values:

- `x1` = $2^{15} - 1$
- `x2` = 1

Operator	Addition Operation	Result	Overflow
=	Adds the inputs in 16 bits before casting the sum to 32 bits	`y1` = -2^{15}	Yes
:=	Casts the inputs to 32 bits before computing the sum	`z1` = $+2^{15}$	No

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 17-3. Then cast that result to the type for `c`.

The calculation of intermediate value occurs as follows:

$$Q_{iv} = Q_a Q_b = 323 \times 245 = 79135$$

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 "Detecting Overflow for Fixed-Point Types" on page 17-11.

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 17-3 with the requirement of zero bias, which occurs as follows:

$$Q_c = ((S_a S_b / S_c) Q_a Q_b) = (2^{-4} \times 2^{-4} / 2^{-5})(323 \times 245) = 79135/8 = 9892$$

No overflow occurs in this case, and the approximate real-world value is as follows:

$$\tilde{V}_c = S_c Q_c = 2^{-5} \times 9892 = 9892/32 = 309.125$$

This value is very close to the actual 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 17-3. Then cast that result to the type for `c`.

The calculation of intermediate value occurs 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:

$$S_c Q_c = S_{iv} Q_{iv}$$

$$Q_c = (S_{iv} / S_c)\ Q_{iv}$$

The calculation for slope of the intermediate value for a division operation occurs as follows:

$$S_{iv} = S_a / S_b = 2^{-4} / 2^{-3} = 2^{-1}$$

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 of 2/3.

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 17-3 with the simplification of zero bias, which is as follows:

$$Q_c = (S_a Q_a)/(S_c(S_b Q_b)) = (S_a/(S_b S_c)) \times (Q_a/Q_b) = (2^{-4}/(2^{-3} \times 2^{-6})) \times (32/24) = 42$$

In this case, the approximate real-world value is as follows:

$$\tilde{V}_c = 42/64 = 0.6563$$

This value is a much better approximation to the precise result of 2/3.

:= 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.

Automatic Scaling of Stateflow Fixed-Point Data

Automatic scaling tools can change the settings of Stateflow fixed-point data. You can prevent automatic scaling by selecting the **Lock data type setting against changes by the fixed-point tools** check box in the Data properties dialog box for fixed-point data (see “Setting Data Properties in the Data Dialog Box” on page 8-5 for details). Selecting this check box prevents replacement of the current fixed-point type with a type that the Fixed-Point Tool or Fixed-Point Advisor chooses. See “Automatic Data Typing Tools” in the Simulink Fixed Point documentation for instructions on autoscaling fixed-point data.

18

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 MATLAB functions can drive the mode logic.

For an example of modeling a frame synchronization algorithm, see “Detection of Valid Transmission Data with Frame Synchronization” on page 18-19.

Note Continuous-time variables of complex type are *not* supported. For more information, see “Defining Continuous-Time Variables” on page 16-11.

Where You Can Use Complex Data

You can define complex data at these levels of the Stateflow hierarchy:

- Charts
- Subcharts
- States
- Functions

How You Can Use Complex Data

You can use complex data to define:

- Complex vectors
- Complex matrices

You can also use complex data as arguments for:

- State actions
- Transition actions
- MATLAB functions (see Chapter 23, “Using MATLAB Functions in Stateflow Charts”)
- Truth table functions (see Chapter 22, “Truth Table Functions for Decision-Making Logic”)
- Graphical functions (see “Graphical Functions for Reusing Logic Patterns and Iterative Loops” on page 7-30)
- Change detection operators (see “Detecting Changes in Data Values” on page 10-83)

Note Exported functions do not support complex data as arguments.

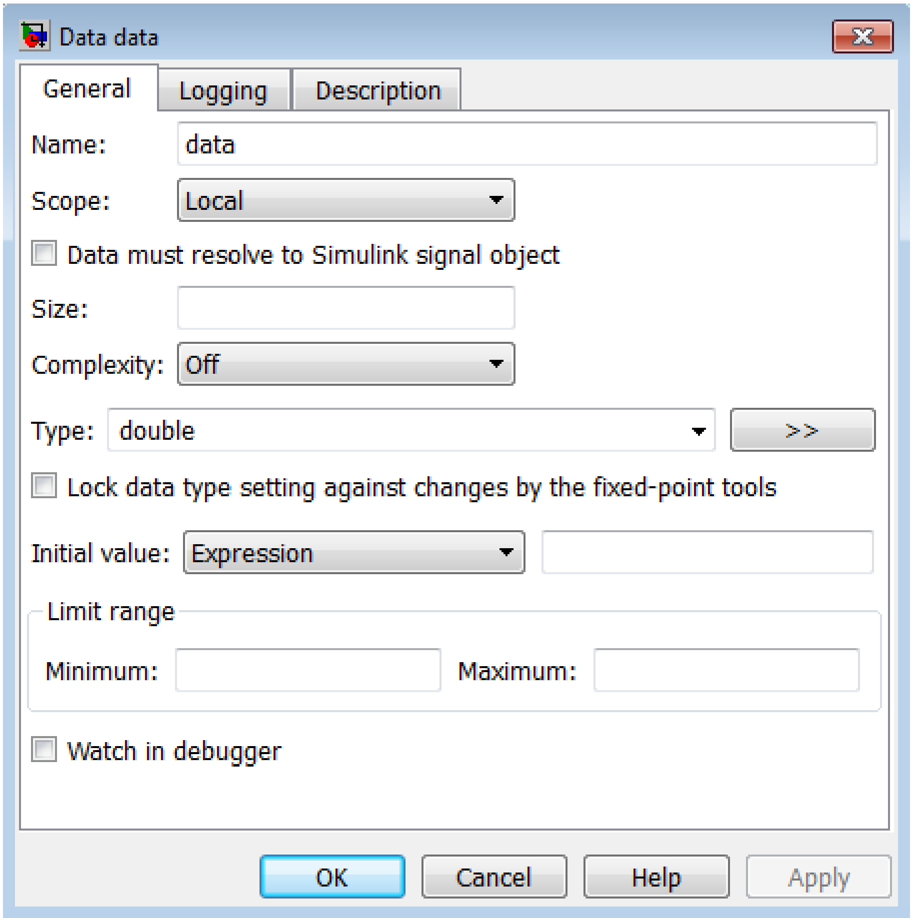
For more information, see “Operations on Complex Data in Stateflow Action Language” on page 18-7 and “Rules for Using Complex Data in Stateflow Charts” on page 18-12.

How to Define Complex Data

Define complex data in a 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.



Name: data

Scope: Local

☐ Data must resolve to Simulink signal object

Size:

Complexity: Off

Type: double

☐ Lock data type setting against changes by the fixed-point tools

Initial value: Expression

Minimum:

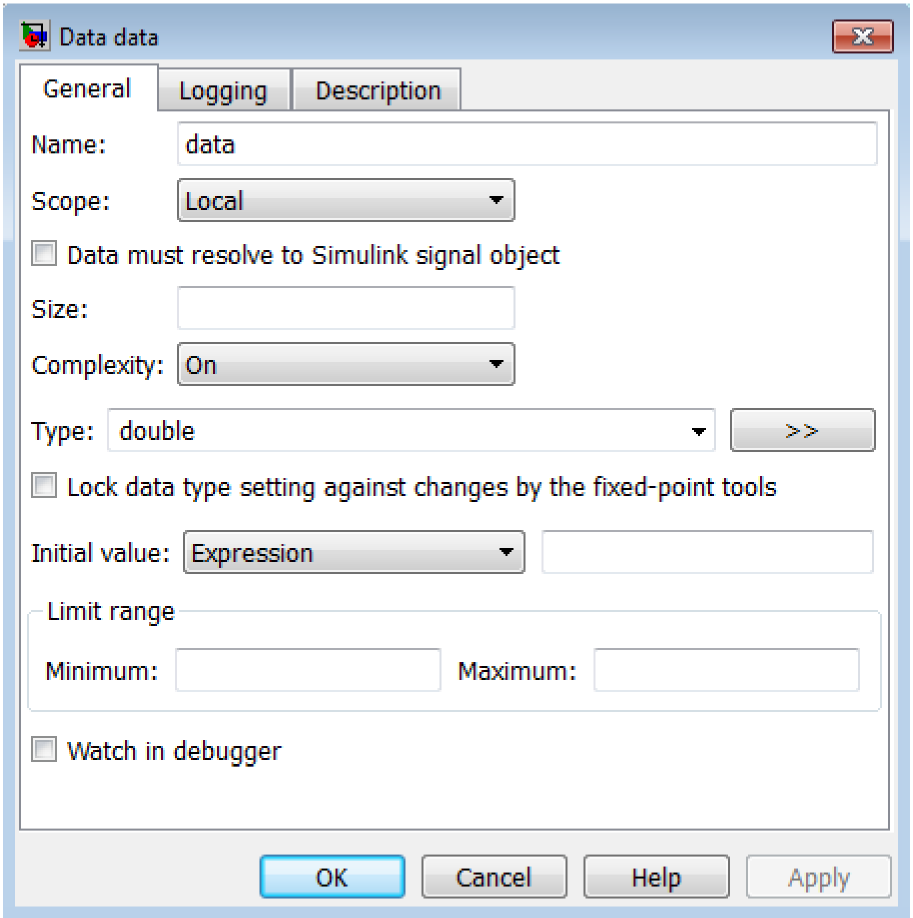
Maximum:

☐ Watch in debugger

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`.



Name: data

Scope: Local

☐ Data must resolve to Simulink signal object

Size:

Complexity: On

Type: double

☐ Lock data type setting against changes by the fixed-point tools

Initial value: Expression

Limit range

Minimum:

Maximum:

☐ Watch in debugger

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 8-5.

Note Complex data does not support the base types `ml`, `struct`, and `boolean`. See "Built-In Data Types" on page 8-49 for more information.

---

4 Click **OK**.

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 a MATLAB function, which provides a numerically accurate and stable result. For details, see "Performing Complex Division with a MATLAB Function" on page 18-17.

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 18-9.

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 MATLAB functions for these operations.

For more information, see “Performing Math Function Operations with a MATLAB Function” on page 18-15.

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 appears when you try to simulate your model.

To mix complex and real operands for division, you can use a MATLAB function as described in “Performing Complex Division with a MATLAB Function” on page 18-17.

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 18-9.

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 appears when you try to simulate your model.

Do not define complex data with ml, struct, or boolean base type

If you define complex data with `ml`, `struct`, or `boolean` base type, an error appears when you try to simulate your model.

Use only real values to set initial values of complex data

When you define the initial value for data that is complex, use only a real value. See "Properties You Can Set in the Description Pane" on page 8-26 for instructions on setting an initial value in the Data properties dialog box.

Do not enter minimum or maximum values for complex data

In the Data properties dialog box, do not enter any values in the **Minimum** or **Maximum** field when you define complex data. If you enter a value in either field, an error message appears when you try to simulate your model.

Assign complex values only to data of complex type

If you assign complex values to real data types, an error appears when you try to simulate your model.

Note You can assign both real and complex values to complex data types.

Do not pass real values to function inputs of complex type

This restriction applies to the following types of chart functions:

- Graphical functions
- Truth table functions
- MATLAB functions
- Simulink functions

If your chart passes real values to function inputs of complex type, an error appears when you try to simulate your model.

Do not use complex data with temporal logic operators

You cannot use complex data as an argument for temporal logic operators, because you cannot define time as a complex number.

Best Practices for Using Complex Data in Stateflow Charts

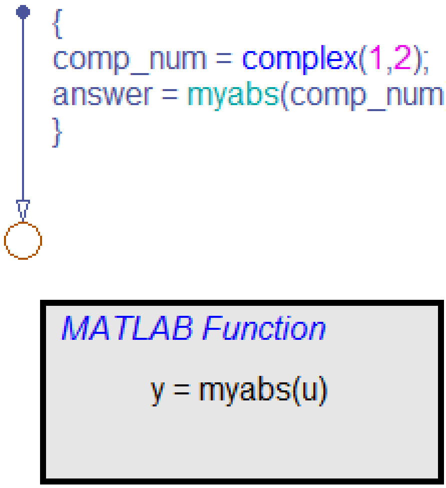
In this section...

Performing Math Function Operations with a 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 a MATLAB function in your chart to perform math function operations on complex data.

A Simple Example

In the following chart, a MATLAB function calculates the absolute value of a complex number:



```
{
comp_num = complex(1,2);
answer = myabs(comp_num)
}
```

```
y = myabs(u)
```

The value of `comp_num` is `1+2i`. Calculating the absolute value gives an answer of `2.2361`.

How to Calculate Absolute Value

Suppose that you want to find the absolute value of a complex number. Follow these steps:

1. Add a MATLAB function to your chart with this signature:

   ```
   y = myabs(u)
   ```

2. Double-click the function box to open the editor.

3. In the editor, enter the code below:

   ```
   function y = myabs(u)
   %#codegen
   y = abs(u);
   ```

   The function `myabs` takes a complex input `u` and returns the absolute value as an output `y`.

4. Configure the input argument `u` to accept complex values.
   1. Open the Model Explorer.
   2. In the **Model Hierarchy** pane of the Model Explorer, navigate to the MATLAB function `myabs`.
   3. In the **Contents** pane of the Model Explorer, right-click the input argument `u` and select **Properties** from the context menu.
   4. In the Data properties dialog box, select `On` in the **Complexity** field and click **OK**.

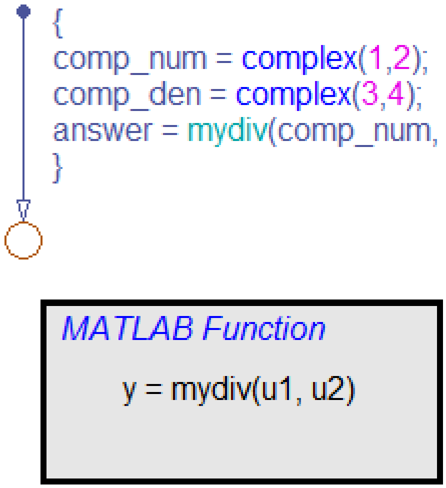
You cannot pass real values to function inputs of complex type. For details, see “Rules for Using Complex Data in Stateflow Charts” on page 18-12.

Performing Complex Division with a MATLAB Function

Division with complex operands is not available as a binary or assignment operation in Stateflow action language. However, you can use a MATLAB function in your chart to perform division on complex data.

A Simple Example

In the following chart, a MATLAB function performs division on two complex operands:



```
{
comp_num = complex(1,2);
comp_den = complex(3,4);
answer = mydiv(comp_num, comp_den)
}
```

The values of `comp_num` and `comp_den` are `1+2i` and `3+4i`, respectively. Dividing these values gives an answer of `0.44+0.08i`.

How to Perform Complex Division

To divide two complex numbers:

1 Add a MATLAB function to your chart with this function signature:

```
y = mydiv(u1, u2)
```

2 Double-click the function box to open the editor.

3. In the editor, enter the code below:

   ```
   function y = mydiv(u1, u2)
   %#codegen
   y = u1 / u2;
   ```

   The function `mydiv` takes two complex inputs, `u1` and `u2`, and returns the complex quotient of the two numbers as an output `y`.

4. Configure the input and output arguments to accept complex values.

   a. Open the Model Explorer.

   b. In the **Model Hierarchy** pane of the Model Explorer, navigate to the MATLAB function `mydiv`.

   c. For each input and output argument, follow these steps:

      i. In the **Contents** pane of the Model Explorer, right-click the argument and select **Properties** from the context menu.

      ii. In the Data properties dialog box, select `On` in the **Complexity** field and click **OK**.

You cannot pass real values to function inputs of complex type. For details, see "Rules for Using Complex Data in Stateflow Charts" on page 18-12.

Detection of Valid Transmission Data with Frame Synchronization

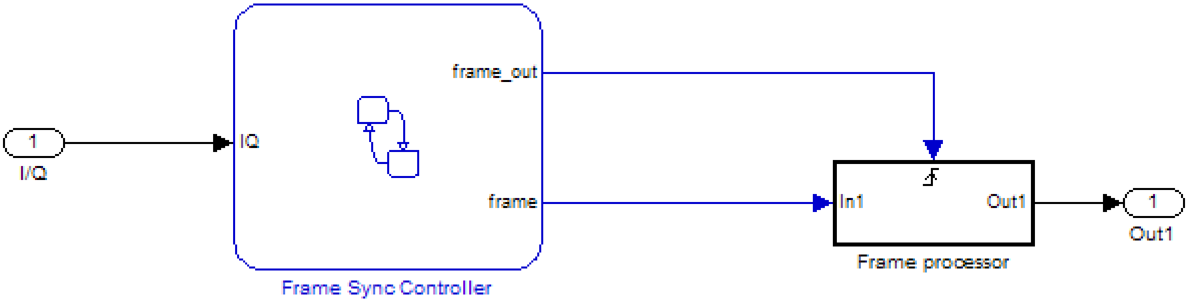
This Simulink model shows how to process complex data in transmission signals of a communication system.

What Is Frame Synchronization?

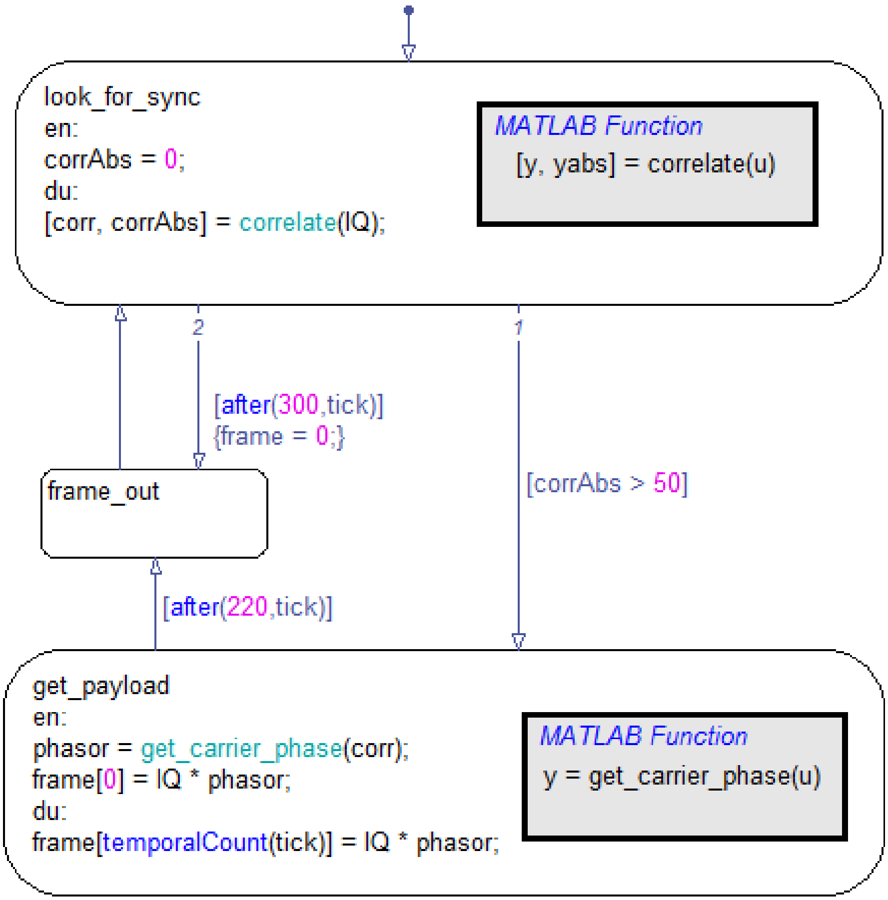
In communication systems, frame synchronization is a method of finding valid data in a transmission that consists of *data frames*. To aid frame synchronization, the transmitter inserts a fixed data pattern at the start of each data frame to mark the start of valid data. The receiver searches for the fixed pattern in each data frame and achieves frame synchronization when the correlation between the input data and the fixed pattern is high.

Model Structure

The model contains the following components.



The chart contains the following states, transitions, and MATLAB functions.



Key characteristics 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 10-63 for information about the `temporalCount` operator.)

- MATLAB functions with complex arguments

  The MATLAB functions `correlate` and `get_carrier_phase` have complex input and output arguments.

Simulation Results

The `sf_frame_sync_controller` model does not produce simulation results. The purpose of this example is to explain how to process complex data in a chart.

How the Chart Works

The chart calculates the correlation between the input signal `I/Q` and the fixed data pattern `trainSig`. You define `trainSig` by writing and running a MATLAB script before you simulate the model.

- If the correlation exceeds 50 percent, frame synchronization occurs. The chart stores 220 valid data points in the complex vector `frame`.
- If the correlation stays below 50 percent after the chart has evaluated 300 data points, the frame synchronization algorithm resets.

Stage	Summary	Details
1	Activation of the frame synchronization algorithm	When the chart wakes up, the state `look_for_sync` activates to start the frame synchronization algorithm.
2	Calculation of correlation between the input signal and the fixed pattern	The MATLAB function `correlate` finds the correlation between the input signal `I/Q` and the fixed data pattern `trainSig`. Then, the function stores the complex correlation as `corr`.

Stage	Summary	Details
3	Calculation of absolute value of the complex correlation	The MATLAB function `correlate` also finds the absolute value of `corr` and stores the output as `corrAbs`. The value of `corrAbs` is the correlation percentage, which can range from 0 to 100 percent. At 0 percent, there is no correlation; at 100 percent, there is perfect correlation.
4	Identification of valid data in a frame	If `corrAbs` exceeds 50 percent, the correlation is high and the chart has identified the start of valid data in a data frame. The transition from the state `look_for_sync` to `get_payload` occurs. If `corrAbs` stays below 50 percent after the chart has evaluated 300 data points, the frame synchronization algorithm restarts.
5	Storage of valid data in a complex vector	When the correlation is high, the state `get_payload` activates. The MATLAB function `get_carrier_phase` finds the phase angle of the carrier wave and stores the value as `phasor`. Then, the state multiplies the input signal `I/Q` with the phase angle `phasor` and stores each complex product in successive elements of the vector `frame`.
6	Output of valid frame data	After collecting 220 data points, the chart outputs the vector `frame` to the next block in the model.
7	Restart of the frame synchronization algorithm	The state `look_for_sync` reactivates, and the frame synchronization algorithm restarts for the next data frame.

Frequency Response Measurement with a Spectrum Analyzer

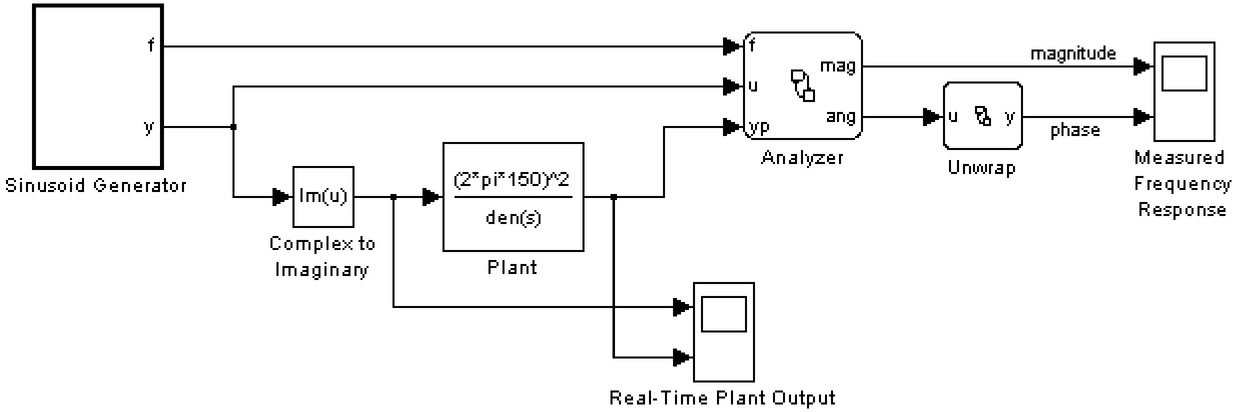
This Simulink model measures the frequency response of a second-order system driven by a complex sinusoidal signal. A scope displays the measured frequency response as discrete Bode plots.

What Is a Spectrum Analyzer?

A spectrum analyzer is a tool that measures the frequency response (magnitude and phase angle) of a physical system over a range of frequencies.

Model Structure

The model contains the following components.

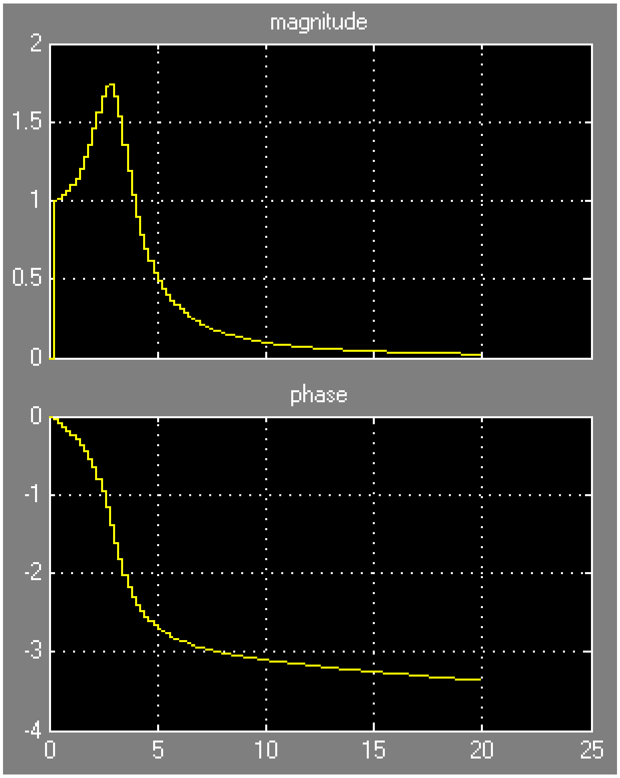


Model Component	Description
Sinusoid Generator block	Generates a complex sinusoidal signal of increasing frequency and supplies this signal to other blocks.
Complex to Imaginary block	Extracts the imaginary part of the complex signal from the Sinusoid Generator block so that a sine wave of increasing frequency can drive the Plant block.

Model Component	Description
Plant block	Uses a transfer function to describe a second-order system with a natural frequency of 150 Hz (300π radians per second) and a damping ratio of 0.3. Since the ratio is less than 1, this system is underdamped and contains two complex conjugate poles in the denominator of the transfer function. **Note** Typical applications implement the Plant block using a D/A (digital-to-analog) converter on the input signal and an A/D (analog-to-digital) converter on the output signal.
Analyzer chart	Calculates the frequency response of the second-order system defined by the Plant block.
Unwrap chart	Processes the phase angle output of the Analyzer chart.

Simulation Results

Simulation of the `sf_spectrum_analyzer` model produces discrete Bode plots in the Measured Frequency Response scope.



To adjust the scope display, right-click inside the grid and select **Autoscale** from the context menu.

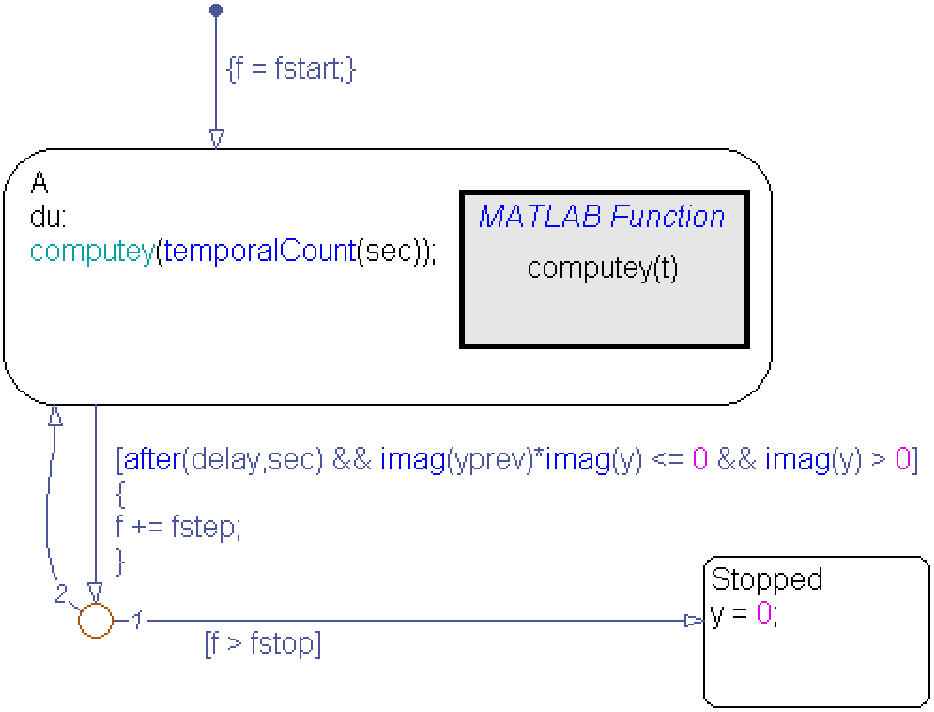
- In the magnitude plot, the sharp peak is the response of the Plant block to a resonant frequency.
- In the phase plot, the angle changes from 0 to –π radians (–180 degrees). Each complex pole in the Plant block adds –π/2 radians to the phase angle.

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**.

Key characteristics of the signal generator chart include:

- Absolute-time temporal logic for controlling changes in frequency (see “Operators for Absolute-Time Temporal Logic” on page 10-70)
- MATLAB function that generates a complex signal (see Chapter 23, “Using MATLAB Functions in Stateflow Charts”)
- Transition condition that contains complex operands (see “Transition Action Types” on page 10-7)

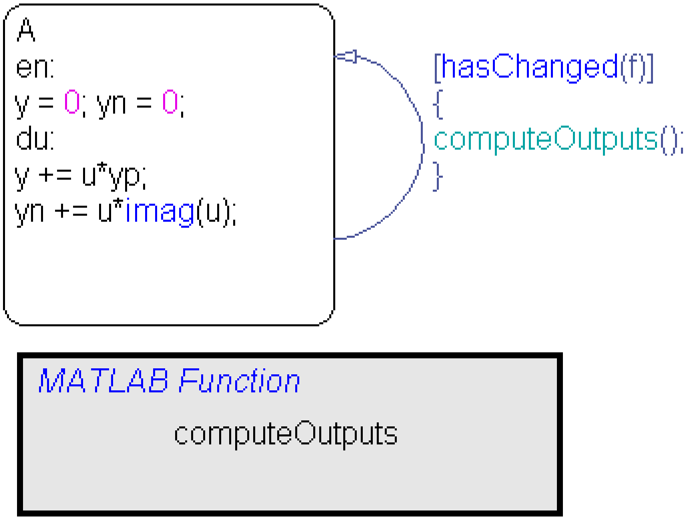


Stage	Summary	Details
1	Signal frequency specification	When the chart awakens, the default transition sets the signal frequency `f` to `fstart` and activates state `A`. **Note** To set `fstart`, double-click the Sinusoid Generator block and enter a value (in Hz) in the **Initial frequency** field.
2	Complex signal generation	While state `A` is active, the MATLAB function `computey` generates the complex signal `y` based on frequency `f` and simulation time `t`.
3	Frequency and complex signal updates	If `delay` seconds have elapsed since activation of state `A`, the frequency `f` increases by an amount `fstep` and the 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.
4	Complex signal termination	When the frequency `f` reaches the value `fstop`, the state `Stopped` becomes active. The complex signal terminates and the simulation ends. **Note** To set `fstop`, double-click the Sinusoid Generator block and enter a value (in Hz) in the **Stop frequency** field.

How the Analyzer Chart Works

Key characteristics of the Analyzer chart include:

- Change detection of input frequency (see “Detecting Changes in Data Values” on page 10-83)
- MATLAB function that processes complex data (see Chapter 23, “Using MATLAB Functions in Stateflow Charts”)
- State `during` action that contains complex operands (see “State Action Types” on page 10-2)



```
A
en:
y = 0; yn = 0;
du:
y += u*yp;
yn += u*imag(u);
```

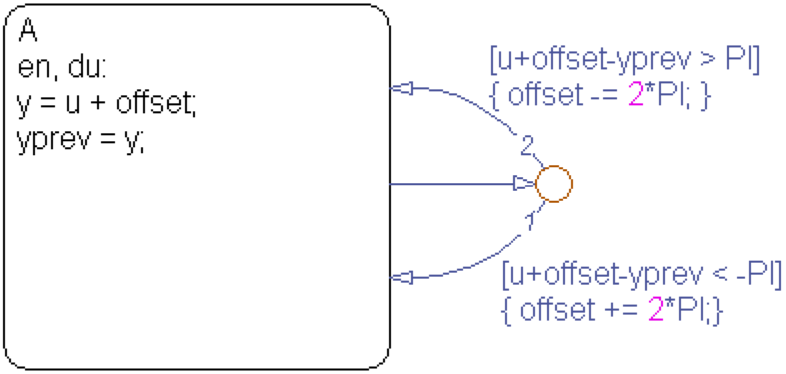
```
[hasChanged(f)]
{
computeOutputs();
}
```

Stage	Summary	Details
1	State A activation	When the chart wakes up, the values of `y` and `yn` initialize to zero. • The data `y` stores the second-order system response to a signal from the Sinusoid Generator block.

Stage	Summary	Details
		• The data `yn` stores an input signal of a given frequency.
2	Frequency response calculation	For a given frequency, the MATLAB function `computeOutputs` finds the magnitude and phase angle of the system response.
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 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.

19

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 7-39 for more details.
- The MATLAB workspace

  See "Calling Built-In MATLAB Functions and Accessing Workspace Data" on page 10-42 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 19-13.
- 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 9-24
  - "Sharing Output Data with Simulink" on page 8-31
- 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 19-23 and "Interface to External Sources" on page 19-25.

The preceding task list is a typical sequence. You might find that another sequence better complements your model development.

See "Implementing Update Interfaces to Simulink Models" on page 19-15 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 9-24
- "Sharing Output Data with Simulink" on page 8-31
- "Sharing Chart Data with External Modules" on page 8-42

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 single charts or all charts in a model.

Setting Properties for a Single Chart

To specify properties for a single chart:

1. Double-click a chart to open it.
2. Right-click an open area of the chart and select **Properties**.

   The Chart properties dialog box appears.

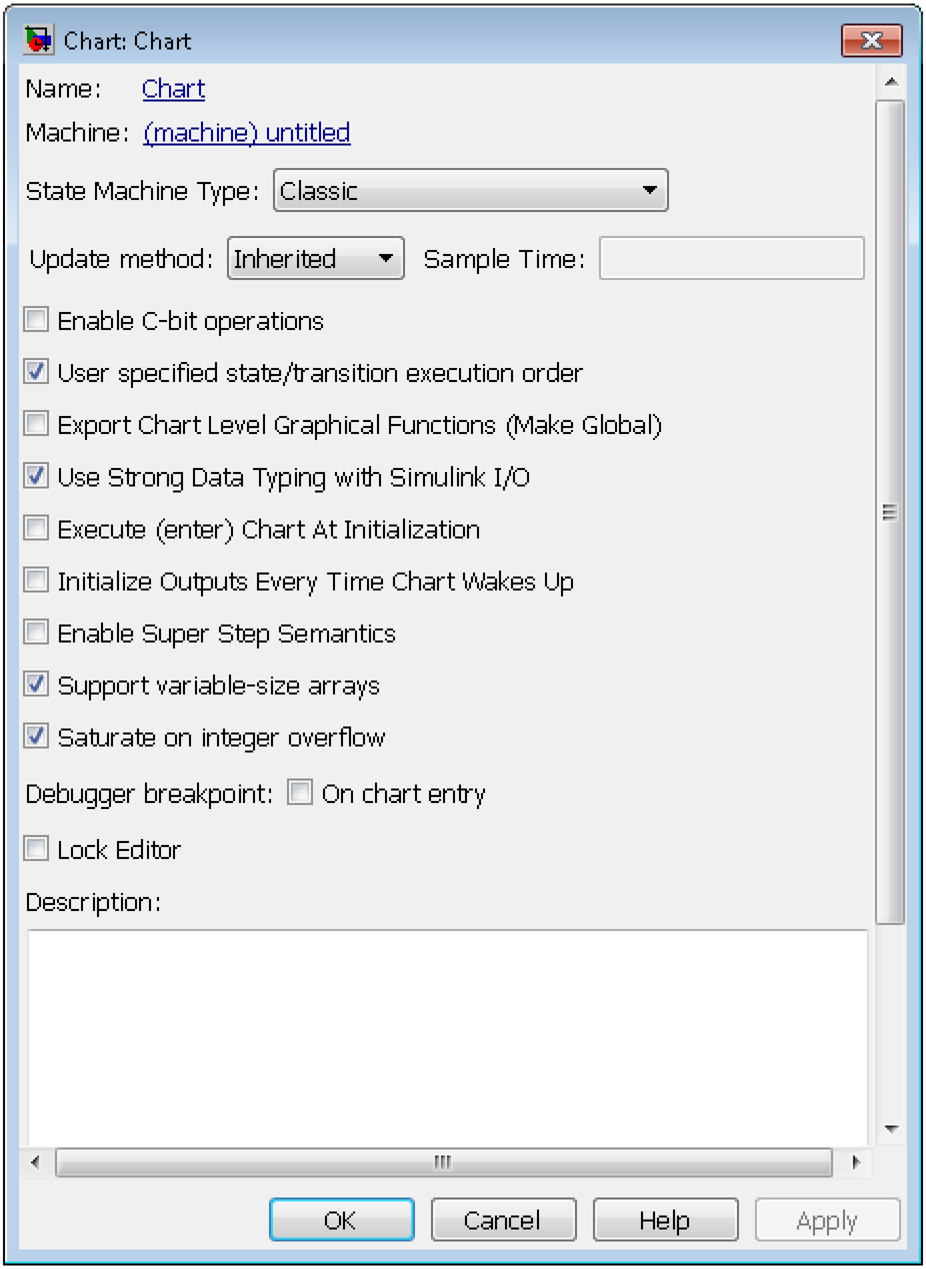


Chart: Chart

Name: Chart

Machine: (machine) untitled

State Machine Type: Classic

Update method: Inherited Sample Time:

☐ Enable C-bit operations

☑ 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

☑ Support variable-size arrays

☑ Saturate on integer overflow

Debugger breakpoint: ☐ On chart entry

☐ Lock Editor

Description:

3 Specify properties for the chart.

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 Machine properties dialog box to the foreground.
State Machine Type	Type of state machine to create. Choices include: • **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 19-13). You can select `Inherited`, `Discrete`, or `Continuous`. For more information about continuous updating, see Chapter 16, “Modeling Continuous-Time Systems in Stateflow Charts”.
Sample Time	If **Update method** is `Discrete`, enter a sample time.

Field	Description
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 16-10 in Chapter 16, "Modeling Continuous-Time Systems in Stateflow Charts".
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.
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 3-75 and "Evaluation Order for Outgoing Transitions" on page 3-55.
Export Chart Level Graphical Functions	Select this check box to export graphical functions defined at the chart's root level. See "Exporting Chart-Level Graphical Functions" on page 7-39 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 Output Data with Simulink" on page 8-31). 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 Output Data with Simulink" on page 8-31) 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 "Execution of a Chart at Initialization" on page 3-49).
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

Field	Description
	• 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 “Execution of a Chart with Super Step Semantics” on page 3-41.
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.
Behavior after too many iterations	If you enable super step semantics, specify how the chart behaves after reaching the maximum number of transitions before taking all valid transitions. Options include: • `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.
Support variable-size arrays	Select to support chart input and output data that vary in dimension during simulation. For more information, see Chapter 14, “Using Variable-Size Data in Stateflow Charts”.

Field	Description
Saturate on integer overflow	Select to specify that integer overflows saturate in the generated code. For more information, see "Handling Integer Overflow for Chart Data" on page 8-60.
States When Enabling	If your chart uses function-call input events, specify how states behave when the event reenables the chart. Options include: • `Held` — Maintain most recent values of the states. • `Reset` — Revert to the initial conditions of the states. • `Inherit` — Inherit this setting from the parent subsystem. For more information, see "Controlling States When Function-Call Inputs Reenable Charts" on page 9-16.
Debugger breakpoint: On chart entry	Select to set a debugging breakpoint on entry to this chart.
Lock Editor	Select to mark the chart as read-only and prevent 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. If an input or output fixed-point data in a chart does not match its counterpart data in a model, a mismatch error results.

4 Click one of these 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 represents all the Stateflow blocks in a model.

To set properties for the Stateflow machine:

1 In the Chart properties dialog box for a particular chart, select the **Machine** link at the top of the dialog box.

   The Machine properties dialog box appears.

2 Enter information in the fields that appear.

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, which is read-only.
Creator	Name of the person who created this Stateflow machine.
Modified	Time of the most recent modification of this Stateflow machine.
Version	Version number of this Stateflow machine.

Field	Description
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 a Single Chart" on page 19-4 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 these buttons:

- **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, set **Update method** in the Chart properties dialog box (see "Specifying Chart Properties" on page 19-4). Select one of 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 Output Data with Simulink" on page 8-31), 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. 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 16, “Modeling Continuous-Time Systems in Stateflow Charts”.

Implementing Update Interfaces to Simulink Models

In this section...

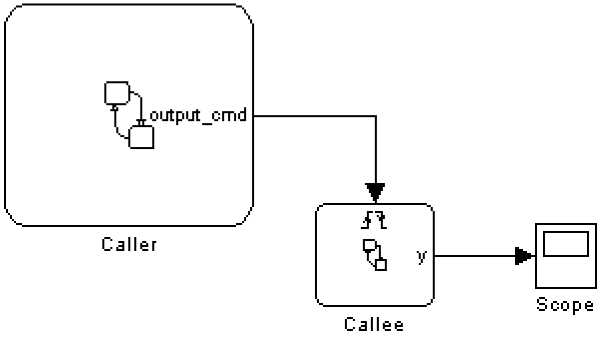
Defining a Triggered Stateflow Block

Essential conditions that define an edge-triggered Stateflow block are:

- The chart **Update method** (set in the Chart properties dialog box) is **Discrete** or **Inherited**. (See "Specifying Chart Properties" on page 19-4.)
- 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

The following model shows an edge-triggered Stateflow block named Callee:



The **Input from Simulink** event has an **Either** edge-trigger type. If you define more than one **Input from Simulink** event, the Simulink model determines the sample times to be consistent with various rates of all the incoming signals. The outputs of a triggered Stateflow block are held after the execution of the block.

Defining a Sampled Stateflow Block

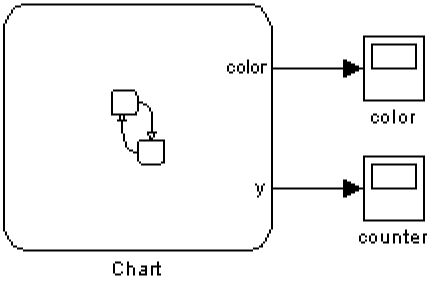
There are two ways you can define a sampled Stateflow block. Setting the chart **Update method** (set in the Chart properties dialog box) to **Discrete** and entering a **Sample Time** value define a sampled Stateflow block. (See "Specifying Chart Properties" on page 19-4.)

Alternatively, you can add and define an **Input from Simulink** data object. You add and define data using either the Stateflow Editor **Add** menu or the Model Explorer. (See "Sharing Output Data with Simulink" on page 8-31.) Simulink determines the chart sample time to be consistent with the rate of the incoming data signal.

The **Sample Time** you set in the Chart properties dialog box takes precedence over the sample time of any **Input from Simulink** data.

Sampled Stateflow Block Example

You specify a discrete sample rate to have Simulink trigger a Stateflow block that does use an explicit trigger port. You can specify a sample time for the chart in the Chart properties dialog box. Simulink then calls the Stateflow block at a defined, regular sample time.



The outputs of a sampled Stateflow block are held after the execution of the block.

Defining an Inherited Stateflow Block

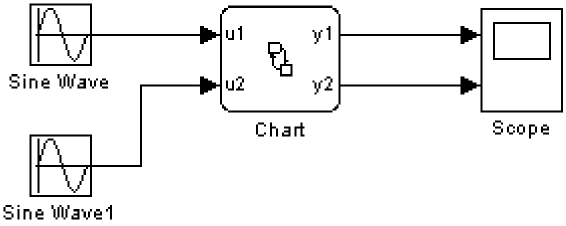
Essential conditions that define an inherited trigger Stateflow block are:

- The chart **Update method** (set in the Chart properties dialog box) is **Discrete** or **Inherited**. (See "Specifying Chart Properties" on page 19-4)
- The chart has an **Input from Simulink** data object defined using the Stateflow Editor **Add** menu or the Model Explorer. (See "Sharing Output Data with Simulink" on page 8-31.) Simulink determines the chart sample time to be consistent with the rate of the incoming data signal.

Inherited Stateflow Block Example

Simulink can trigger a Stateflow block that does not use an explicit trigger port or a specified discrete sample time. In this case, the Simulink calls the Stateflow block at a sample time determined by the model.

In this example, the chart contains two **Input from Simulink** data objects. Simulink determines the sample times to be consistent with the rates of both incoming signals.



The outputs of an inherited trigger Stateflow block are held after the execution of the block.

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 16, “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. The procedure assumes that you have a programmed function-call subsystem and a Stateflow block in the model. Use the following steps to connect the Stateflow block to the function-call subsystem and trigger it during simulation.

1. In your chart, select **Add > Event > Output to Simulink**.

   The Event properties dialog box appears with a default name of `event` and a **Scope** of `Output to Simulink`.

2. Set **Trigger** to `Function Call`.

3. Name the event appropriately and click **OK** to close the dialog box.

   An output port with the name of the event you add appears on the right side of the Stateflow block.

4. Connect the output port on the Stateflow block for the function-call output event to the input trigger port of the subsystem.

   Avoid placing any other blocks in the connection lines between the Stateflow block and the function-call subsystem.

   Note You cannot connect a function-call output event from a chart to a Demux block to trigger multiple subsystems.

   ---

5. To execute the function-call subsystem, include an event broadcast of the function-call output event in the actions of the chart.

For examples of using function-call output events, see “Using Function Calls to Activate a Simulink Block” on page 9-33.

Defining Edge-Triggered Output Events

Simulink controls the execution of edge-triggered subsystems with output events. Essential conditions that define this use of triggered output events are:

- The chart has an **Output to Simulink** event with the trigger type set to **Either**. See "Using Output Events to Activate a Simulink Block" on page 9-24.
- The Simulink block connected to the edge-triggered **Output to Simulink** event has its own trigger type set to the equivalent edge trigger.

For examples of using edge-triggered output events, see "Using Edge Triggers to Activate a Simulink Block" on page 9-24.

Creating Specialized Chart Libraries for Large-Scale Modeling

In this section...

When to Use Chart Libraries

In Simulink, you can create your own block libraries as a way to reuse the functionality of blocks or subsystems in one or more models. Similarly, you can reuse a set of Stateflow algorithms by encapsulating the functionality in a chart library.

As with other Simulink block libraries, you can specialize each instance of chart library blocks in your model to use different data types, sample times, and other properties. Library instances that inherit the same properties can reuse generated code.

For more information about Simulink block libraries, see “Working with Block Libraries” in the Simulink documentation.

How to Create Chart Libraries

1. Add Stateflow charts with polymorphic logic to a Simulink model.

   Polymorphic logic is logic that can process data with different properties, such as type, size, and complexity.

2. Configure the charts to inherit the properties you want to specialize.

   For a list, see “Properties You Can Specialize Across Instances of Library Blocks” on page 19-21.

3. Optionally, customize your charts using masking.

   For more information, see "Masks on Blocks in User Libraries" in the Simulink documentation.

4. Simulate and debug your charts.

5. In Simulink, create a library model by selecting **File > New > Library**.

6. Copy or drag the charts into a library model.

For an example using MATLAB Function blocks, see "Creating Custom Block Libraries with MATLAB Function Blocks" in the Simulink documentation.

Properties You Can Specialize Across Instances of Library Blocks

You can specialize instances of Stateflow library blocks by allowing them to inherit any of the following properties from Simulink.

Property	Inherits by Default?	How to Specify Inheritance
Type	Yes	Set the data type property to **Inherit: Same as Simulink**.
Size	Yes	Set the data size property to **-1**.
Complexity	Yes	Set the data complexity property to **Inherited**.
Limit range	No	Specify minimum and maximum values as Simulink parameters. For example, if minimum value = `aParam` and maximum value = `aParam` + 3, different instances of a Stateflow library block can resolve to different `aParam` parameters defined in their parent mask subsystems.

Property	Inherits by Default?	How to Specify Inheritance
Initial value	Depends on scope	For local data, temporary data, and outputs, specify initial values as Simulink parameters. Other data always inherits the initial value: • Parameters inherit the initial value from the associated parameter in the parent mask subsystem. • Inputs inherit the initial value from the Simulink input signal. • Data store memory inherits the initial value from the Simulink data store to which it is bound.
Sampling mode (input)	Yes	Stateflow chart input ports always inherit sampling mode.
Data type override mode for fixed-point data	Yes	Different library instances inherit different data type override modes from their ancestors in the model hierarchy.
Sample time (block)	Yes	Set the block sample time property to **-1**.

Limitations of Library Charts

Events parented by a library Stateflow machine are invalid. The parser flags such events as errors.

MATLAB Workspace Interfaces

In this section...

About the MATLAB Workspace

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 all` at the MATLAB command line. See the MATLAB documentation for more information.

Interfacing the MATLAB Workspace with Charts

A chart has 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 "Calling Built-In MATLAB Functions and Accessing Workspace Data" on page 10-42 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 8-27.

- You can save chart data to the workspace at the end of a simulation.

See “Saving Data to the MATLAB Workspace” on page 8-34 for more information.

Interface to External Sources

In this section...

Supported External Sources

Any source of data 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 from external sources to your Stateflow chart. For information on defining data, see Chapter 8, "Defining Data".
- You can include external source code in the **Simulation Target > Custom Code** pane of the Configuration Parameters dialog box. For details, see Chapter 25, "Building Targets".

Exported Data

You might want an external source (outside the chart and the model) to be able to access a data object. By defining the scope of a data object as `Exported`, you make it accessible to external sources. Exported data must be parented by the Stateflow machine, because the machine is 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

Suppose that you want to export a Stateflow data object named `ext_data`. Follow these steps:

1. In your Stateflow machine, define data named `ext_data` of `Exported` scope.
2. Define `ext_data` as imported in the external code source (custom code) using the following format:

   ```
   extern int ext_data;

   void func_example(void)
   {
   ...
   ext_data = 123;
   ...
   }
   ```

Stateflow software generates the following code for the exported data:

```
int ext_data;
```

Imported Data

Similarly, you might want to access a data object that is defined outside the chart and the model. If you define the scope of the data as `Imported`, you can access the data anywhere in the hierarchy of the Stateflow machine. The parent of an imported data object is external. However, the data object needs an adoptive parent to resolve symbols for code generation. The adoptive parent of an imported data object must be the Stateflow machine, because the machine is the highest level in the 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

Suppose that you want to import a Stateflow data object named `ext_data`. Follow these steps:

1. In your Stateflow machine, define data named `ext_data` of `Imported` scope.
2. Define `ext_data` as exported in the external code source (custom code) using the following format:

   ```
   int ext_data;

   void func_example(void)
   {
   ...
   }
   ```

Stateflow software generates the following code for the imported data:

```
extern int ext_data;
```

20

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.

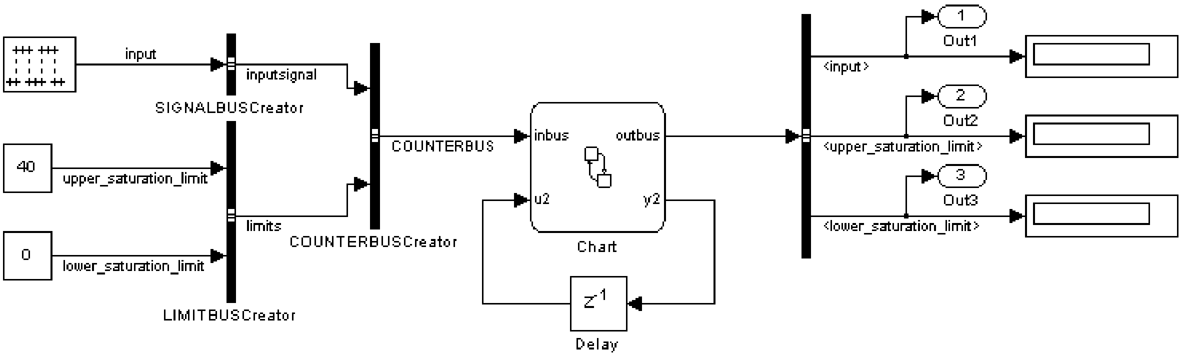
What You Can Do with Structures

With the Stateflow structure data type, you can create:

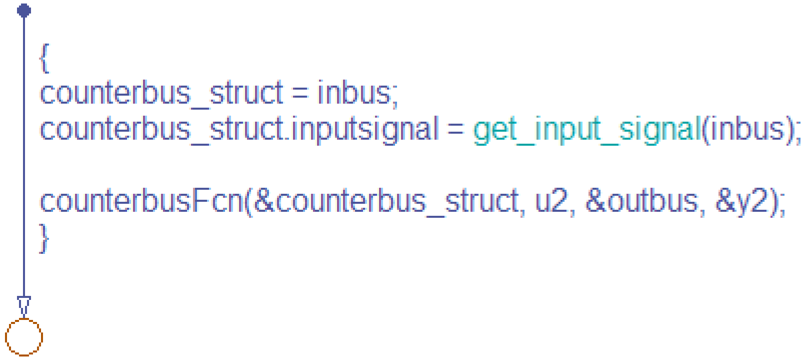
- Inputs and outputs for accessing Simulink bus signals from Stateflow charts, Truth Table blocks, and MATLAB Function blocks (see "Defining Structure Inputs and Outputs" on page 20-8)
- Local structure data in Stateflow charts, truth tables, graphical functions, MATLAB functions, and boxes (see "Defining Local Structures" on page 20-11)
- Temporary structure data in Stateflow graphical functions, truth tables, and MATLAB functions (see "Defining Temporary Structures" on page 20-14)

Example of Stateflow Structures

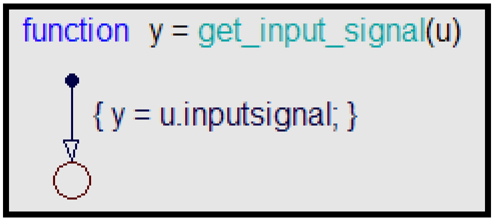
The model sfbus_demo provides examples of structures in a Stateflow chart.



The chart contains a graphical function.



```
{
counterbus_struct = inbus;
counterbus_struct.inputsignal = get_input_signal(inbus);

counterbusFcn(&counterbus_struct, u2, &outbus, &y2);
}
```



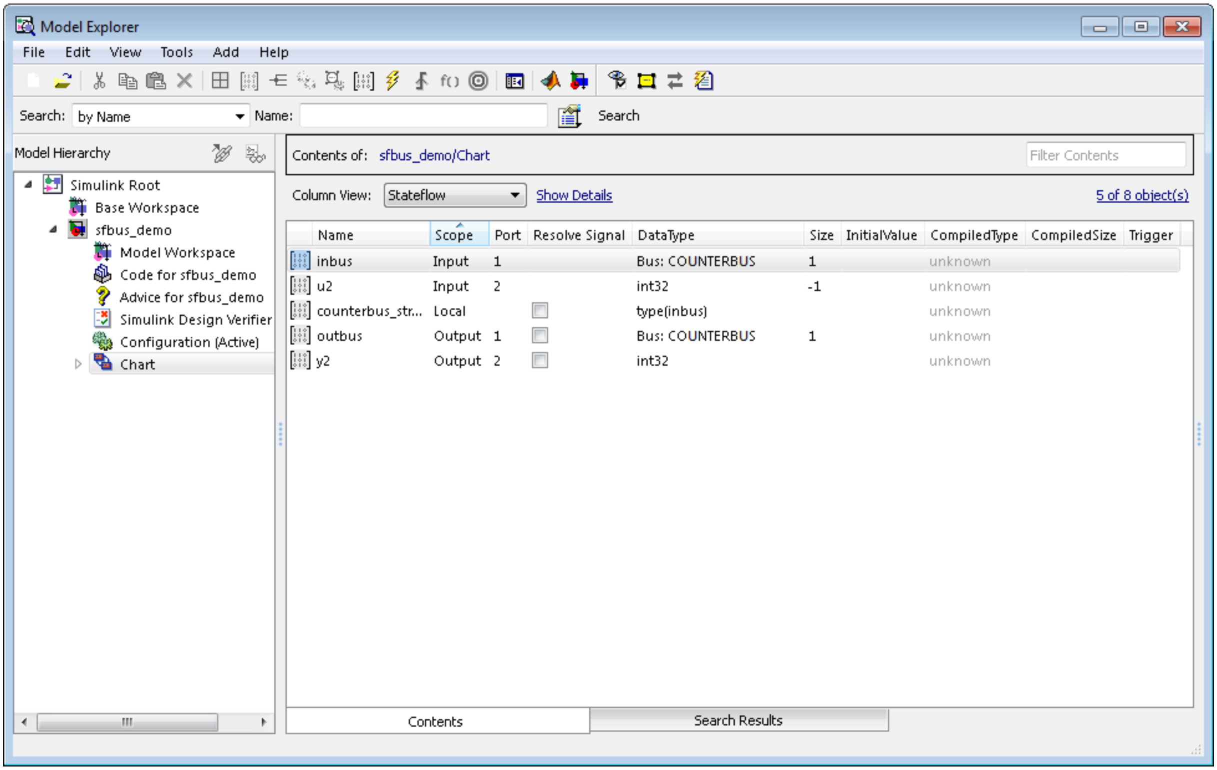
```
function  y = get_input_signal(u)
{ y = u.inputsignal; }
```

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

Definitions of structures in the chart of the `sfbus_demo` model appear in the Model Explorer as follows:



Search: by Name Name: Search

Contents of: sfbus_demo/Chart

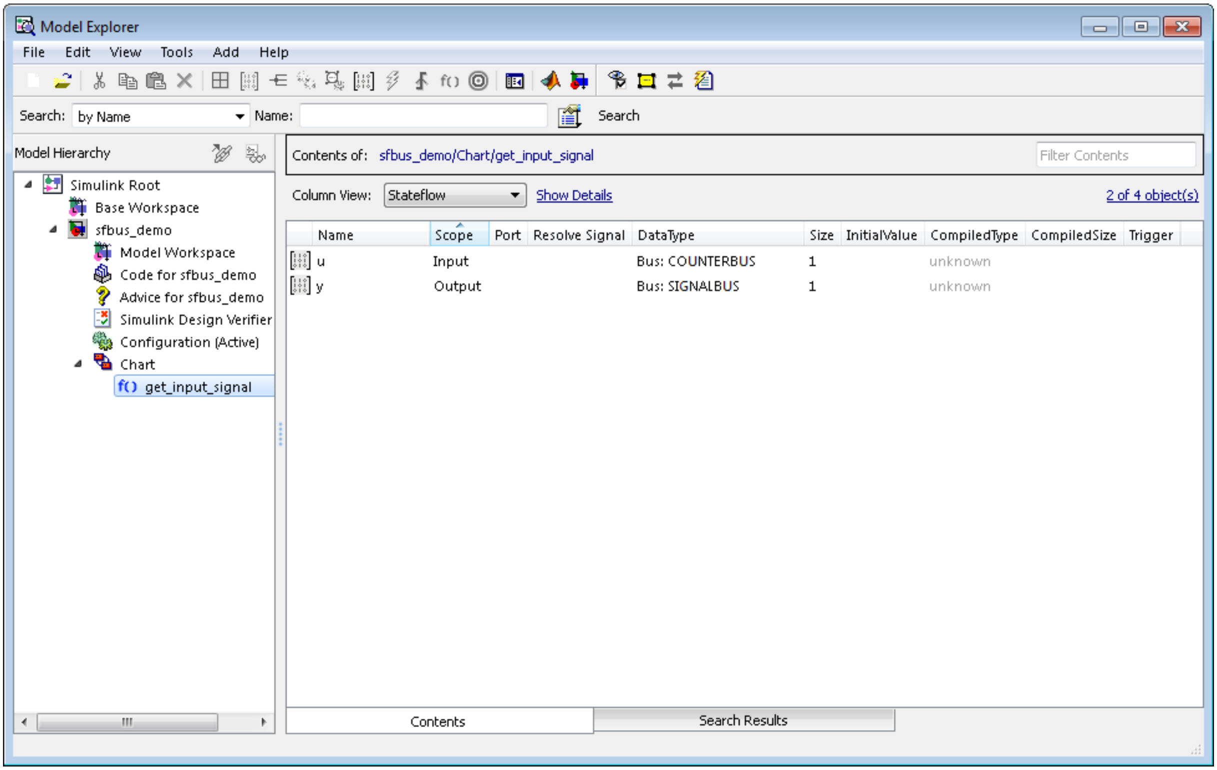
Column View: Stateflow Show Details 5 of 8 object(s)

Name	Scope	Port	Resolve Signal	DataType	Size	InitialValue	CompiledType	CompiledSize	Trigger
inbus	Input	1		Bus: COUNTERBUS	1		unknown		
u2	Input	2		int32	-1		unknown		
counterbus_str...	Local		☐	type(inbus)			unknown		
outbus	Output	1	☐	Bus: COUNTERBUS	1		unknown		
y2	Output	2	☐	int32			unknown		

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 20-15.

Structure Definitions in sfbus_demo Stateflow Graphical Function

Definitions of structures in the graphical function `get_input_signal` appear in the Model Explorer as follows:



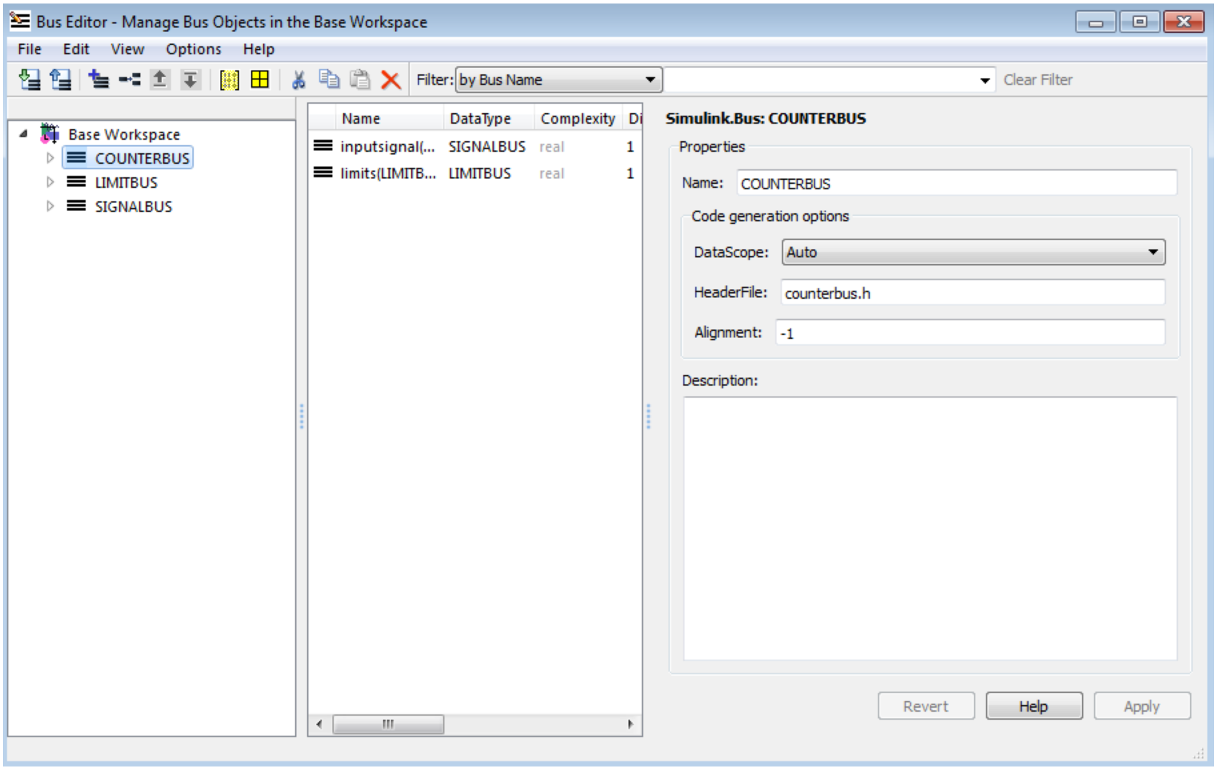
Contents of: sfbus_demo/Chart/get_input_signal

Column View: Stateflow

Name	Scope	Port	Resolve Signal	DataType	Size	InitialValue	CompiledType	CompiledSize	Trigger
u	Input			Bus: COUNTERBUS	1		unknown		
y	Output			Bus: SIGNALBUS	1		unknown		

Simulink Bus Objects Define Stateflow Structures

Each Stateflow structure must be defined by a `Simulink.Bus` object in the base workspace. 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:



Filter: by Bus Name

Name	DataType	Complexity	Di
inputsignal(...	SIGNALBUS	real	1
limits(LIMITB...	LIMITBUS	real	1

Name: COUNTERBUS

DataScope: Auto

HeaderFile: counterbus.h

Alignment: -1

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`.

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 20-8.

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 20-22.

Defining Stateflow Structures

In this section...

Rules for Defining Structure Data Types in 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 The Stateflow machine is the object that contains all other Stateflow objects in a Simulink model (see "Stateflow Hierarchy of Objects" on page 1-8).

- Structures cannot have these scopes: Constant or Data Store Memory.
- Structures of parameter scope must be tunable.
- Data array objects cannot contain structures.
- Structures cannot contain arrays of buses.

Defining Structure Inputs and Outputs

- "Interfacing Stateflow Structures with Simulink Bus Signals" on page 20-9
- "Working with Virtual and Nonvirtual Buses" on page 20-11

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 documentation.

2. 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:  <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 20-11).

Type	Works with Scope	Requirements
		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 specify an appropriate bus object for **Output data type** in the Bus Creator dialog box. When you specify the bus object, Simulink verifies that the properties of the `Simulink.Bus` object in the base workspace match the properties of the Simulink bus signal.
Bus: <object name>	**Input** or **Output**	Replace "*`<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 20-15.

Type	Works with Scope	Requirements
		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 20-9.

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. 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:  <object name>`, or `<data type expression>`, and then specify the expression as follows:

Type	What to Specify
Bus: <object name>	Replace "*`<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 20-15 • Enter the name of the `Simulink.Bus` object in the base workspace that defines the Stateflow structure.

7. Click **Apply**.

Defining Structures of Parameter Scope

To define structures of parameter scope:

1. Create a Simulink bus object in the base workspace to define the structure type for your chart.

   For information about how to create Simulink bus objects, see `Simulink.Bus` in the Simulink Reference documentation.

2. 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 `Parameter`.

6. In the **Type** field, select either `Bus:  <object name>`, or `<data type expression>`, and then specify the expression as follows:

Type	What to Specify
Bus: <object name>	Replace "*`<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 20-15 • Enter the name of the `Simulink.Bus` object in the base workspace that defines the Stateflow structure.

7. Click **Apply**.

Tip Stateflow structures with parameter scope must be tunable. To ensure tunability, open the Configuration Parameters dialog box and clear the **Inline parameters** check box on the **Optimization > Signals and Parameters** pane. In this case, each element in the structure is tunable.

For more information, see "Tunable Parameters" in the Simulink documentation.

Defining Temporary Structures

You can define temporary structures in truth tables, graphical functions, and MATLAB functions of a Stateflow chart.

To define a temporary structure:

1. Create a Simulink bus object in the base workspace to define the structure type for your chart.

   For information about how to create Simulink bus objects, see `Simulink.Bus` in the Simulink Reference documentation.

2. 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`.

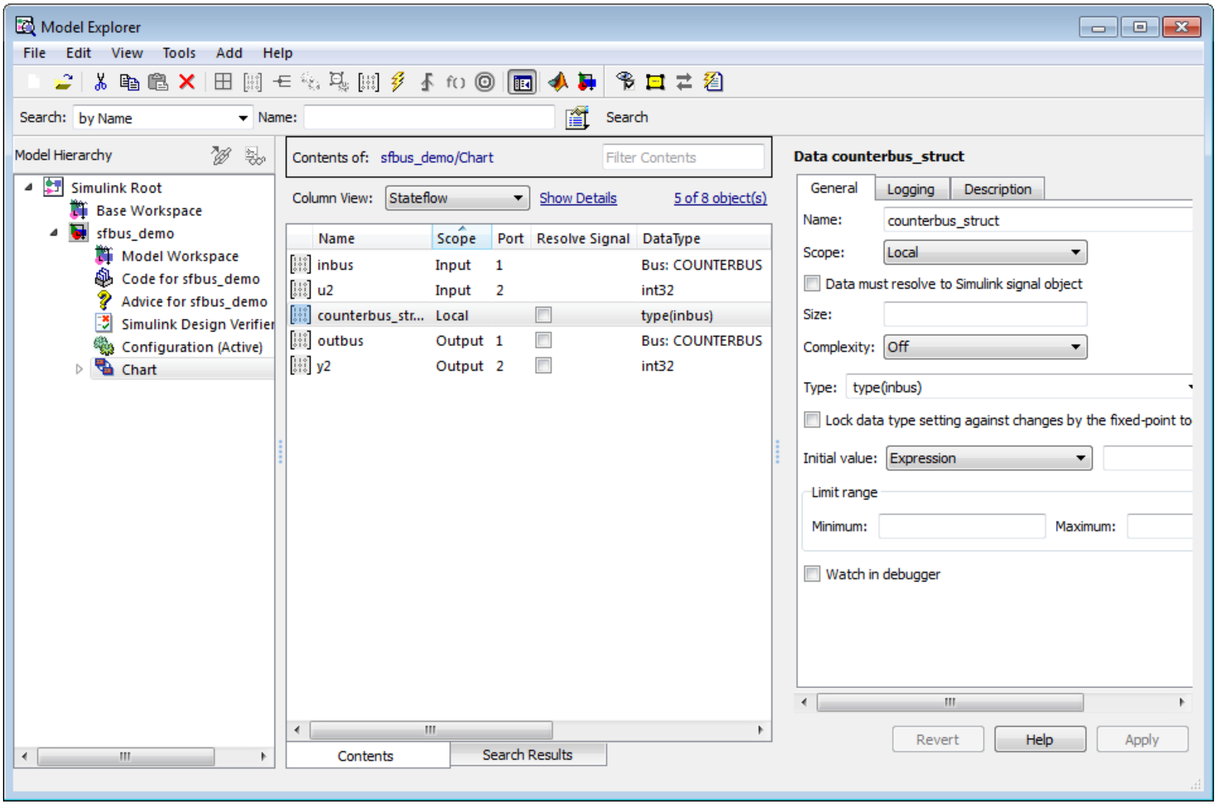
6. In the **Type** field, select either `Bus:  <object name>`, or `<data type expression>`, and then specify the expression as follows:

Type	What to Specify
Bus: <object name>	Replace "*`<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 20-15 • 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:



Contents of: sfbus_demo/Chart

Column View: Stateflow

5 of 8 object(s)

Name	Scope	Port	Resolve Signal	DataType
inbus	Input	1		Bus: COUNTERBUS
u2	Input	2		int32
counterbus_str...	Local		☐	type(inbus)
outbus	Output	1	☐	Bus: COUNTERBUS
y2	Output	2	☐	int32

Data counterbus_struct

Name: counterbus_struct

Scope: Local

☐ Data must resolve to Simulink signal object

Size:

Complexity: Off

Type: type(inbus)

☐ Lock data type setting against changes by the fixed-point to

Initial value: Expression

Limit range

Minimum:

Maximum:

☐ Watch in debugger

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 8-50.

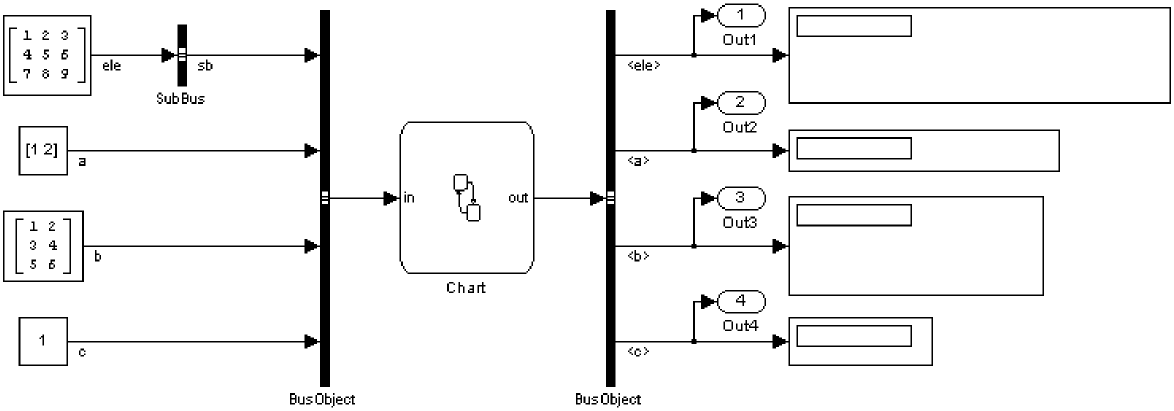
Structure Operations

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.

Suppose that you have the following model:

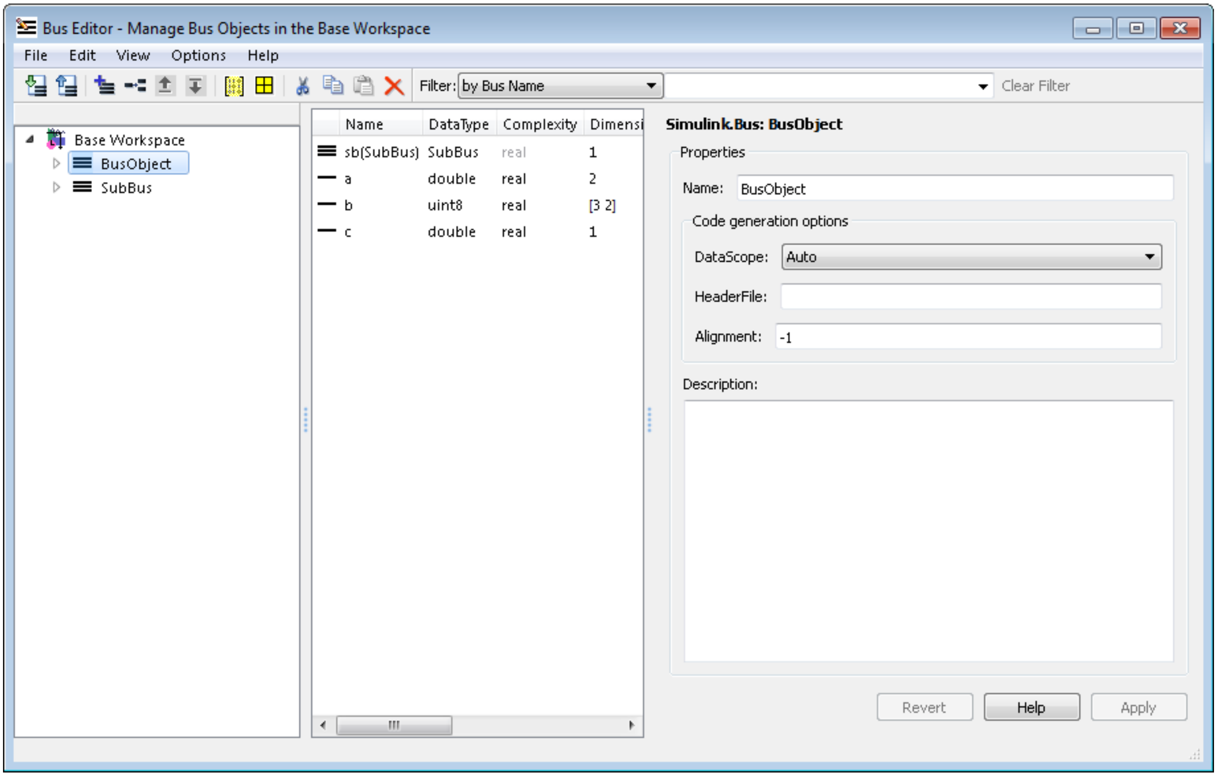


In this example, the `SubBus` and `BusObject` blocks to the left of the chart are Bus Creator blocks. The `BusObject` block to the right of the chart is a Bus Selector block.

The following structures are defined in the 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:



Filter: by Bus Name

Name	DataType	Complexity	Dimensi
sb(SubBus)	SubBus	real	1
a	double	real	2
b	uint8	real	[3 2]
c	double	real	1

Simulink.Bus: BusObject

Properties

Name: BusObject

Code generation options

DataScope: Auto

HeaderFile:

Alignment: -1

Description:

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`
`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`

Guidelines for Assignment of Values

You can assign values to any Stateflow structure *except* input structures — that is, a structure with scope equal to `Input`. Here are the guidelines for assigning values to output, local, parameter, 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 specifications for the sfbus_demo model, as described in "Example of Stateflow Structures" on page 20-2:

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 20-2 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 chart for simulation and real-time code generation. Follow these steps:

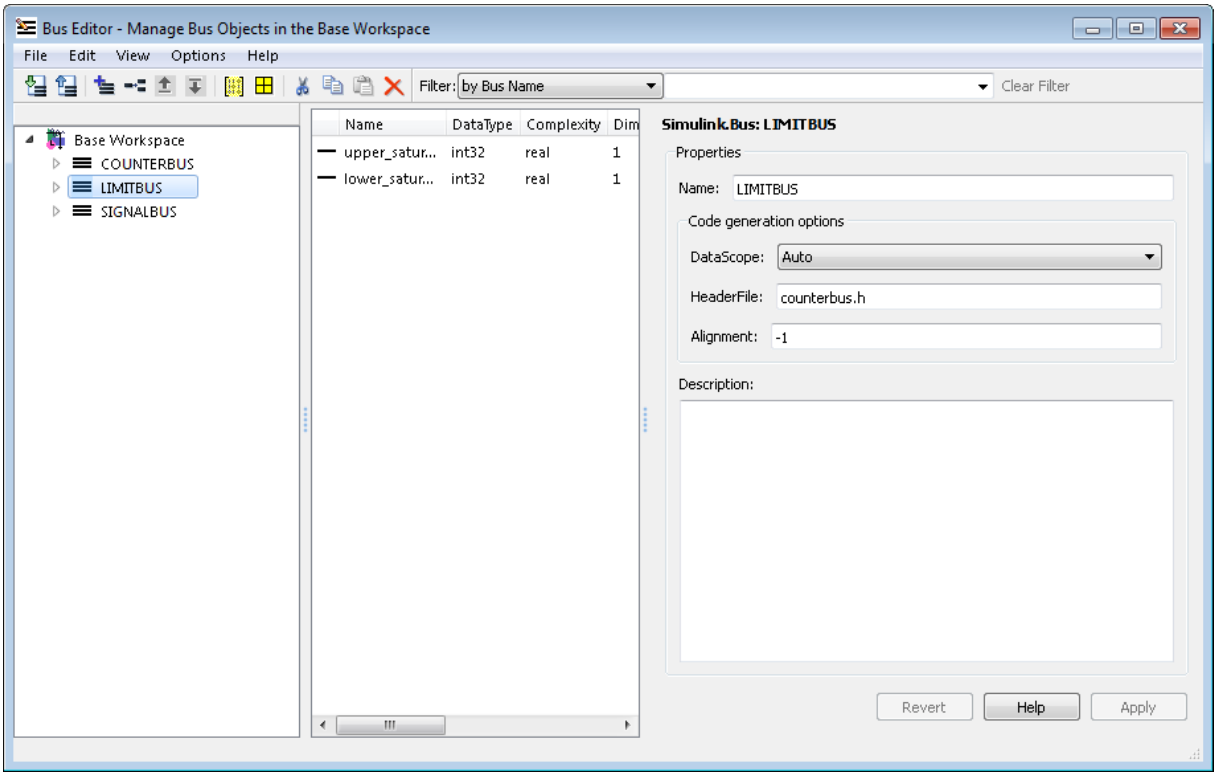
1. Define your structure in C, creating custom source and header files.

   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:



Filter: by Bus Name

Name	DataType	Complexity	Dim
upper_satur...	int32	real	1
lower_satur...	int32	real	1

Simulink.Bus: LIMITBUS

Properties

Name: LIMITBUS

Code generation options

DataScope: Auto

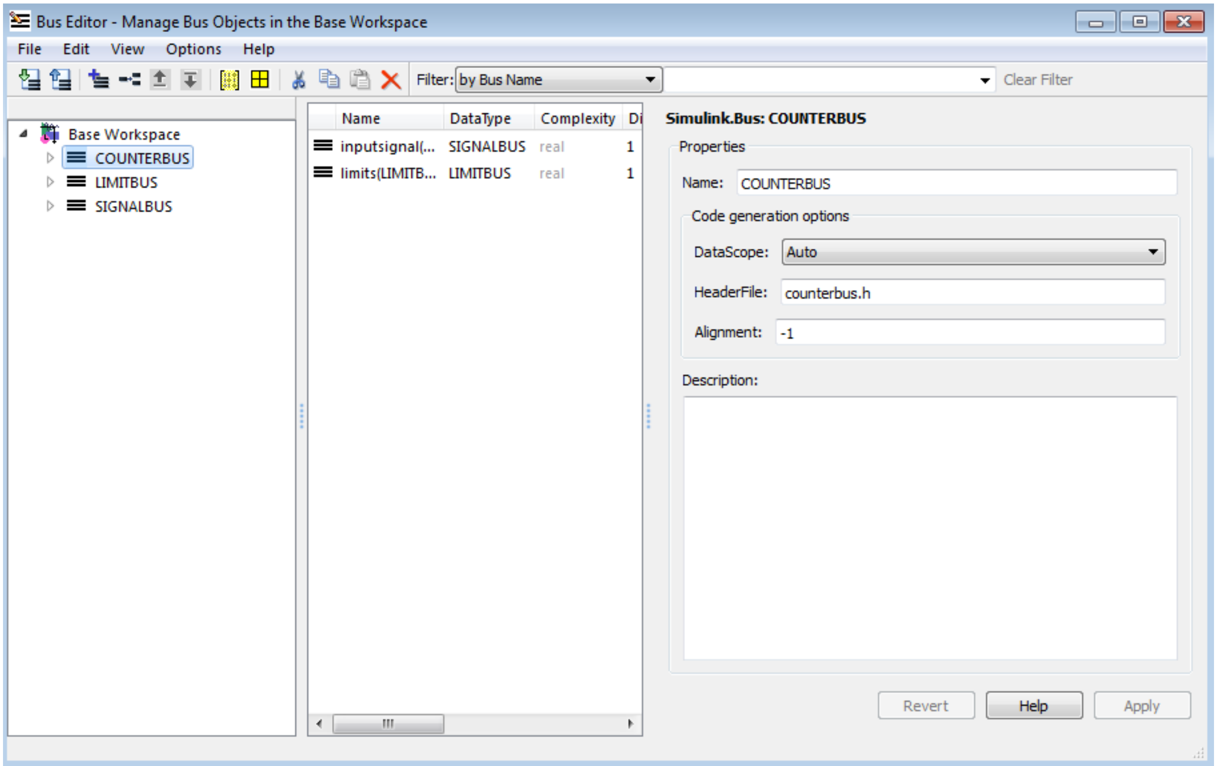
HeaderFile: counterbus.h

Alignment: -1

Description:

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`:



Name	DataType	Complexity	Di
inputsignal(...	SIGNALBUS	real	1
limits(LIMITB...	LIMITBUS	real	1

Simulink.Bus: COUNTERBUS

Properties

Name: COUNTERBUS

Code generation options

DataScope: Auto

HeaderFile: counterbus.h

Alignment: -1

Description:

4 Configure your chart to include custom C code, as follows.

To Include Custom C Code:	**Do This:**
In code generated for simulation	Follow these steps: **1** Open the chart that uses your custom C structures. **2** Open the Configuration Parameters dialog box. **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 25-8. For more information, see Chapter 25, "Building Targets".
In code generated for real-time applications	Follow these steps: **1** Open the chart that uses your custom C structures. **2** Open the Configuration Parameters dialog box. **3** In the Configuration Parameters dialog box, select **Code Generation > Custom Code** in the Select tree. Custom code options appear in the right pane. **4** Follow instructions in "Configure Model for External Code Integration" in the Simulink Coder documentation.

5. Build your model and fix errors (see "Debugging Structures" on page 20-26).

6. Run your model.

Debugging Structures

You debug structures as you would other Stateflow chart data, as described in Chapter 26, “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 During Simulation” on page 26-37. 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 20-17.

21

Stateflow Design Patterns

Debouncing Signals

In this section...

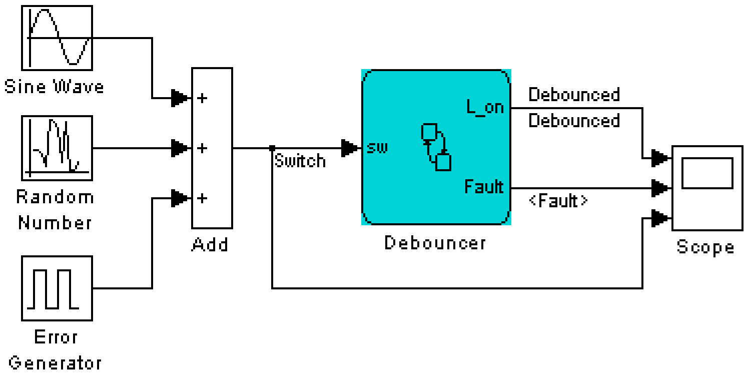
Why Debounce Signals

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*.

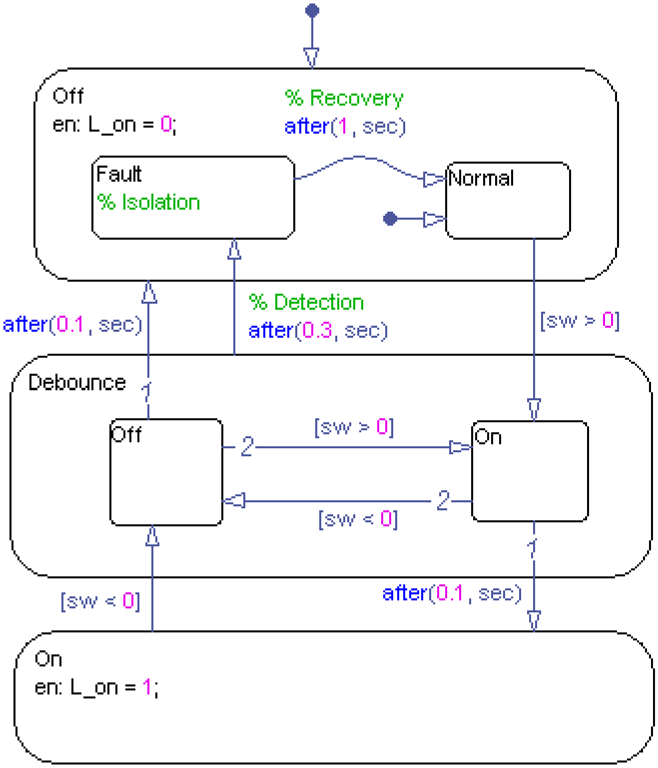
For example, if you model a controller in a Stateflow chart, you do not want your switch logic to overwork the controller by turning it on and off in response to every transient signal it receives. Instead, you can design a Stateflow debouncer that uses temporal logic to determine whether the switch is really on or off.

The Debouncer Model

The model `sf_debouncer` illustrates a design pattern that uses temporal logic to isolate transient signals.



The Debouncer chart contains the following logic:



Key Behaviors of Debouncer Chart

The key behaviors of the Debouncer chart are:

- "Intermediate Debounce State Isolates Transients" on page 21-4
- "Temporal Logic Determines True State" on page 21-5

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 10-70). 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 10-64). The keyword `tick` specifies and implicitly generates a local event when the chart awakens (see "Using Implicit Events" on page 9-40).

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.

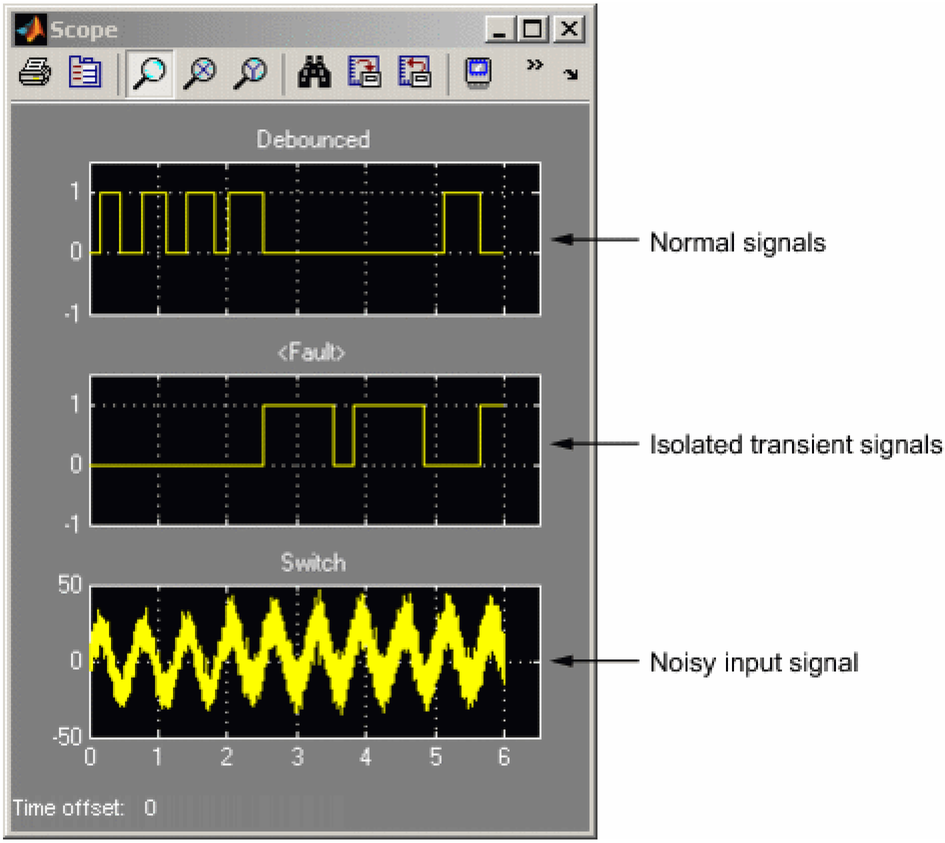
Absolute Time-Based Logic	Event-Based Logic
`after ( 0.1, sec )`	`after ( 100, tick )`
`after ( 0.3, sec )`	`after ( 300, tick )`
`after ( 1, sec )`	`after ( 1000, tick )`

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 21-6 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.

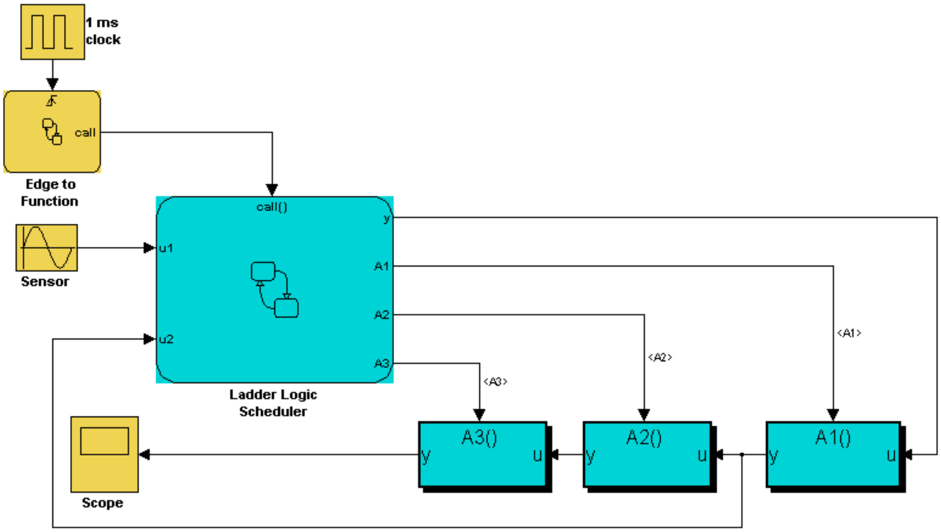
Types of Schedulers

You can implement the following types of schedulers using Stateflow charts.

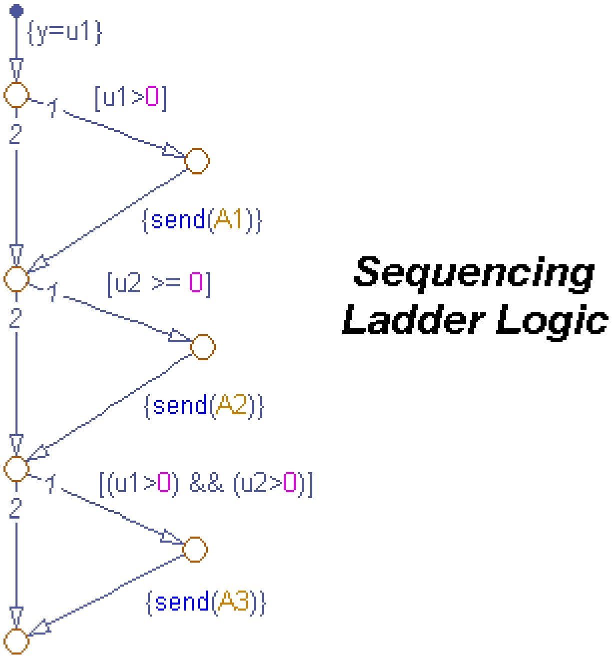
Scheduler Design Pattern	Description
Ladder logic scheduler	Schedules multiple Simulink subsystems to execute in a single time step
Loop scheduler	Schedules one Simulink subsystem to execute multiple times in a single time step
Temporal logic scheduler	Schedules Simulink subsystems to execute at specific times

Scheduling Multiple Subsystems in a Single Time Step

The **ladder logic scheduler** design pattern allows you to specify the order in which multiple Simulink subsystems execute in a single time step. The model `sf_ladder_logic_scheduler` illustrates this design pattern.



The Ladder Logic Scheduler chart contains the following logic:



Key Behaviors of Ladder Logic Scheduler

The key behaviors of the ladder logic scheduler are:

- "Function-Call Output Events Trigger Multiple Subsystems" on page 21-10
- "Flow Graph Determines Order of Execution" on page 21-11

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 21-11).

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 "Example of Directed Event Broadcasting Using send" on page 10-59).

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

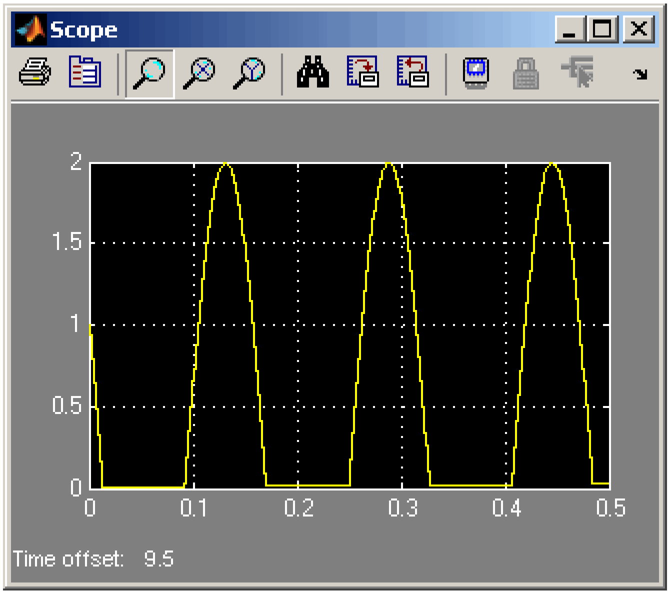
To run the `sf_ladder_logic_scheduler` model, follow these steps:

1. Open the model by typing `sf_ladder_logic_scheduler` at the MATLAB command prompt.

2. Open the Scope block.

3. Start simulation.

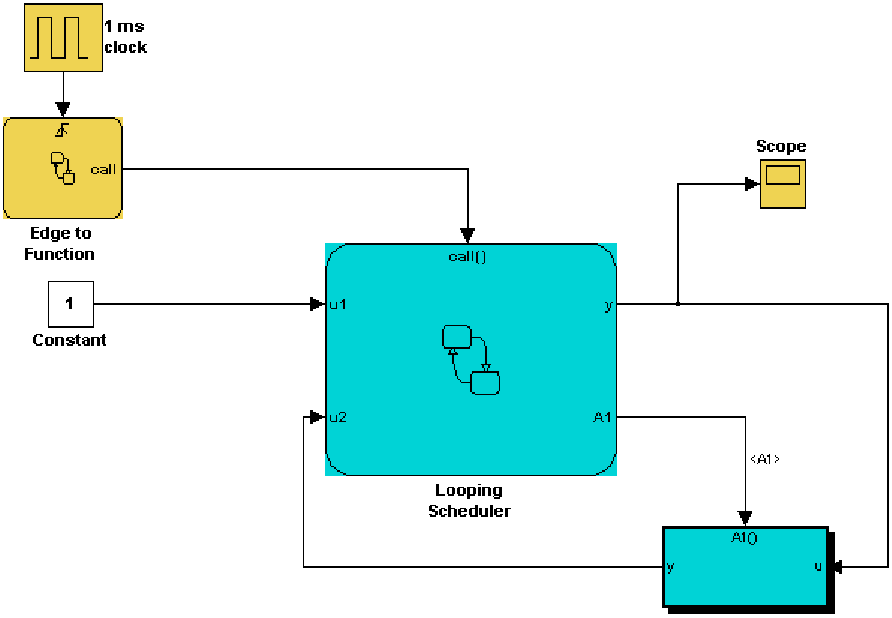
   The scope shows how output `y` changes, depending on which subsystems the Ladder Logic Scheduler chart calls during each time step.



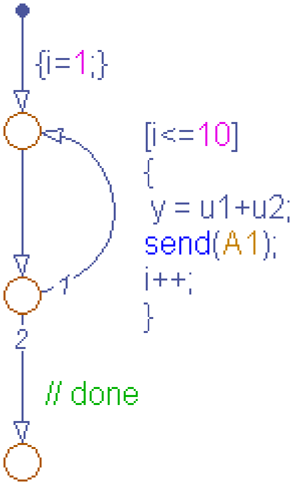
Tip If you keep the chart closed, the simulation runs much faster. For other tips, see "Speeding Up Simulation" on page 25-16.

Scheduling One Subsystem in a Single Time Step

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.



The Looping Scheduler chart contains the following logic:



Key Behaviors of Loop Scheduler

The key behaviors of the loop scheduler are:

- "Function-Call Output Event Triggers Subsystem Multiple Times" on page 21-15
- "Flow Graph Implements For Loop" on page 21-16

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 21-16).

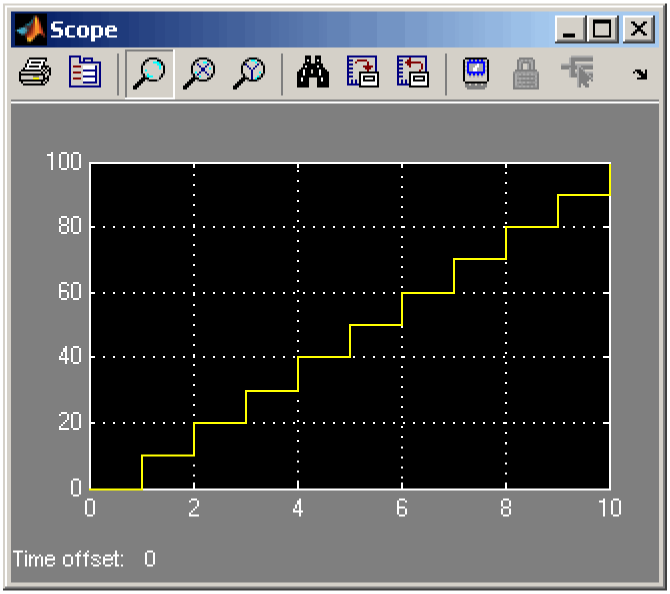
Flow Graph Implements For Loop. The Looping Scheduler chart uses Stateflow flow charting capabilities to implement a `for` loop for broadcasting an event multiple times in a single time step. The chart contains a Stateflow flow graph that uses a local data variable `i` to control the loop. At each iteration, 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 Scope block.
3. Start simulation.

The scope displays the value of `y` at each time step.

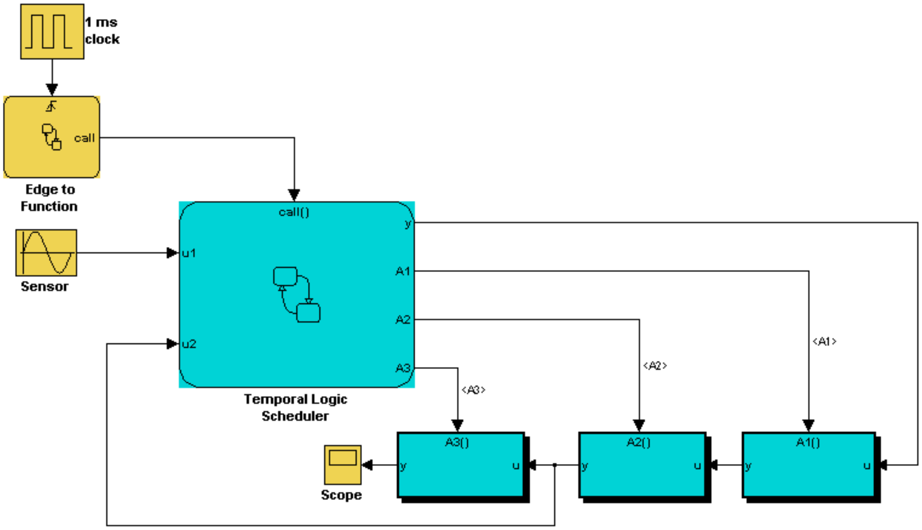


In this example, the Looping Scheduler chart executes the `for` loop 10 times in each time step. During each iteration:

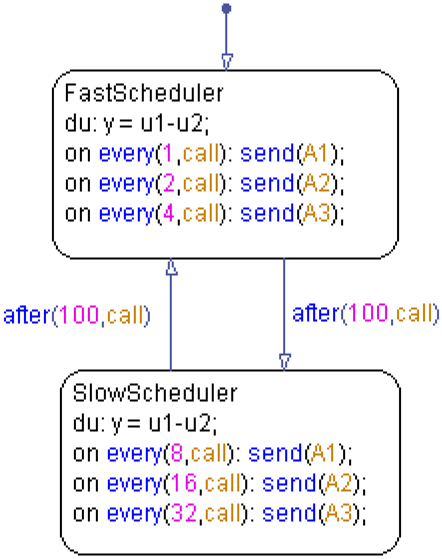
1. The chart increments `y` by 1 (the constant value of input `u1`).
2. The chart broadcasts a function-call output event that executes subsystem `A1`.
3. Subsystem `A1` multiplies `y` by a gain of 1.
4. Control returns to the chart.

Scheduling Subsystems to Execute at Specific Times

The **temporal logic scheduler** design pattern allows you to schedule Simulink subsystems to execute at specified times. The model `sf_temporal_logic_scheduler` illustrates this design pattern.



The Temporal Logic Scheduler chart contains the following logic:



```
FastScheduler
du: y = u1-u2;
on every(1,call): send(A1);
on every(2,call): send(A2);
on every(4,call): send(A3);
```

```
SlowScheduler
du: y = u1-u2;
on every(8,call): send(A1);
on every(16,call): send(A2);
on every(32,call): send(A3);
```

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 10-64).

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

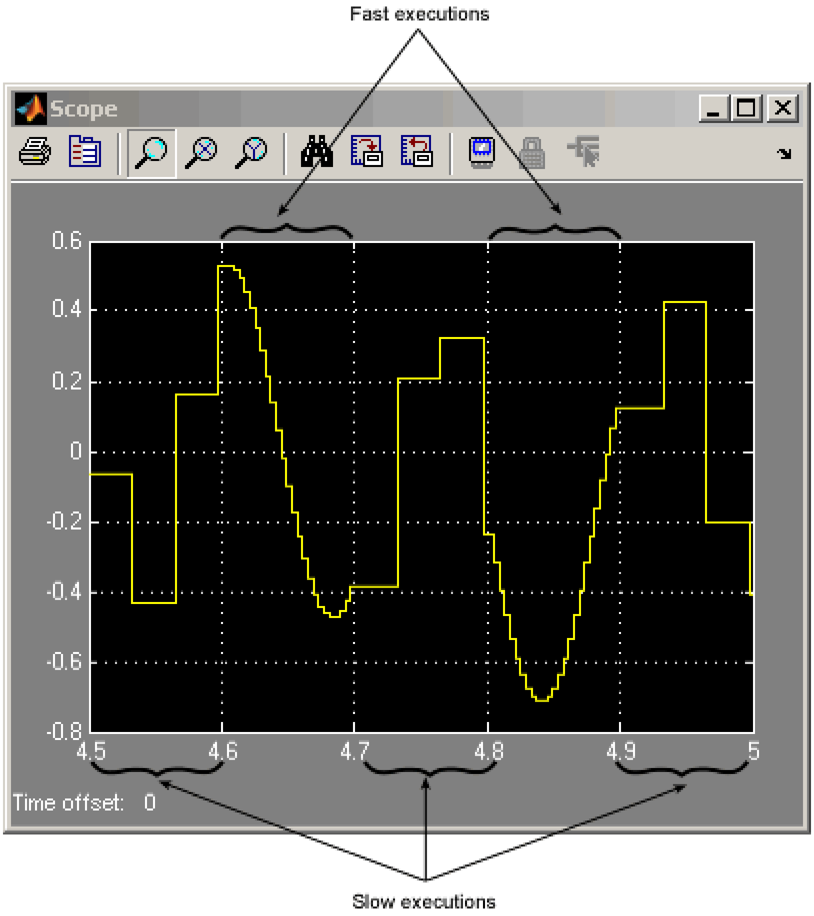
The SlowScheduler state schedules function calls less frequently — at 8, 16, and 32 times slower than the base rate. The chart switches between fast and slow executions after every 100 invocations of the `call` event.

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 Scope block.
3. Start simulation.
4. After the simulation ends, click the **Autoscale** button in the Scope block.

The scope illustrates the different rates of execution.



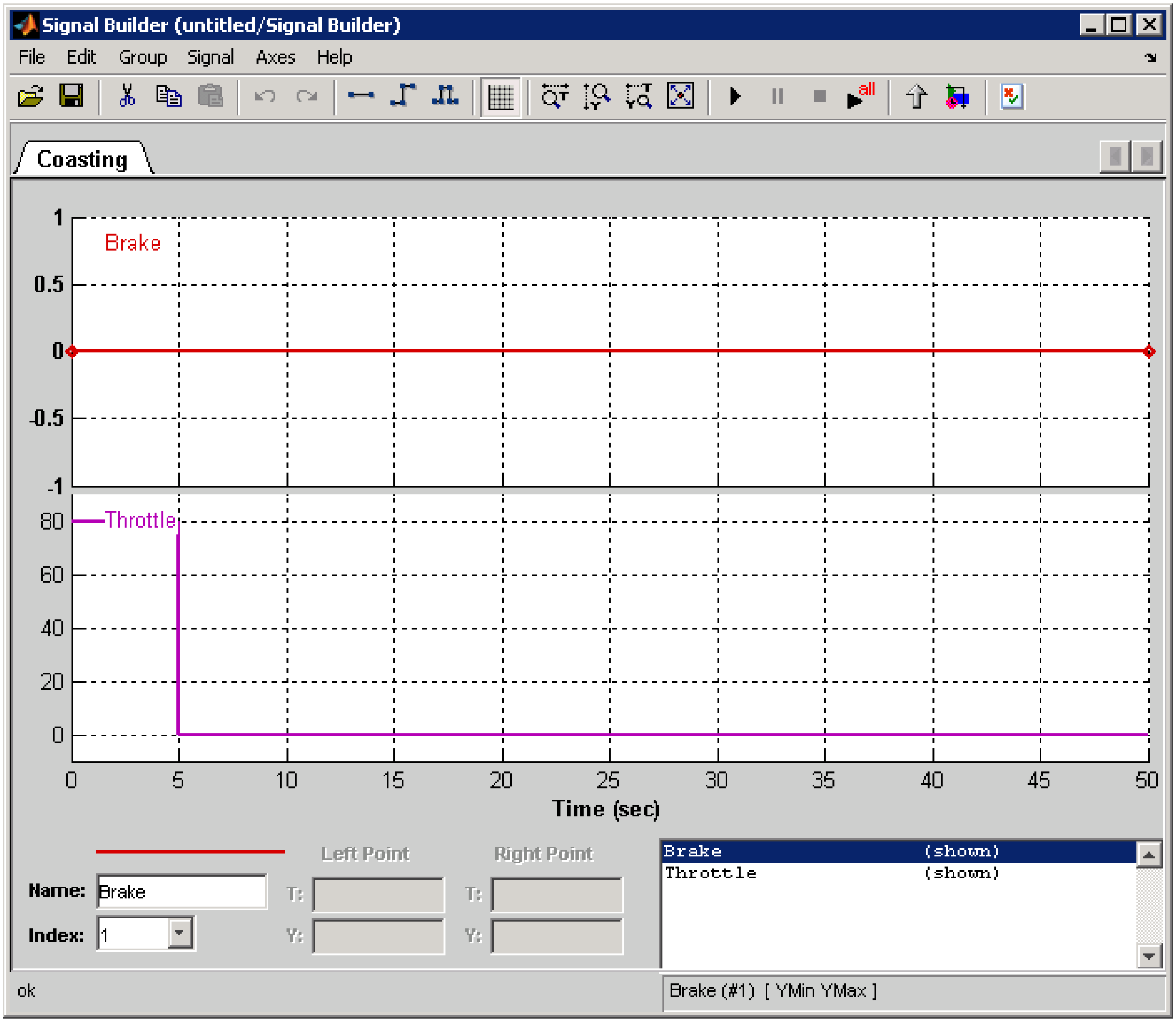
Implementing Dynamic Test Vectors with Hierarchy and Parallelism

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.



Name: Brake

Index: 1

Brake	(shown)
Throttle	(shown)

ok

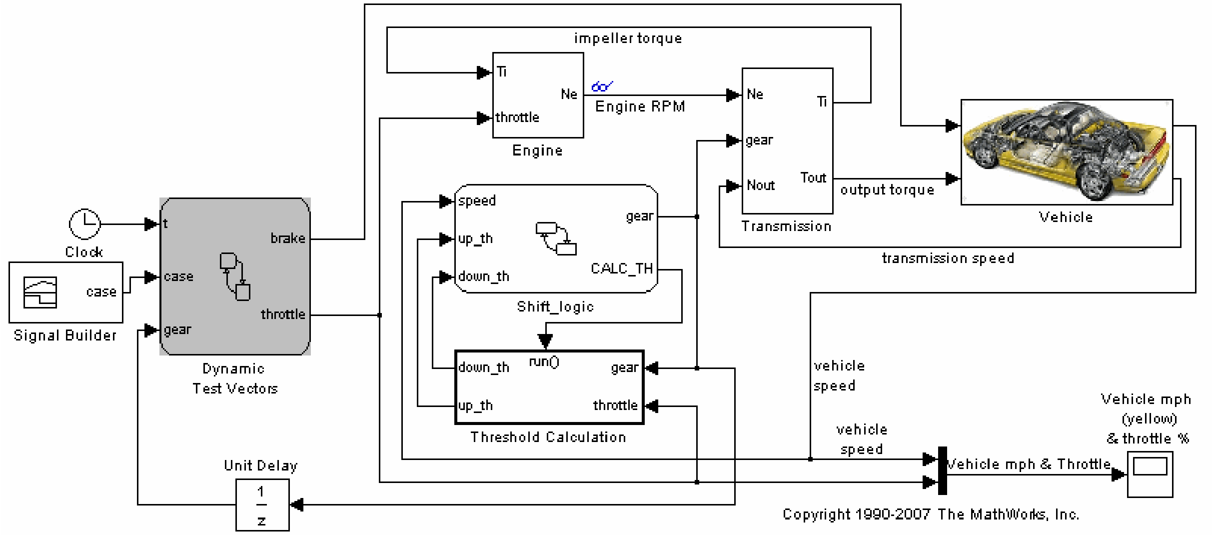
Brake (#1) [YMin YMax]

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.

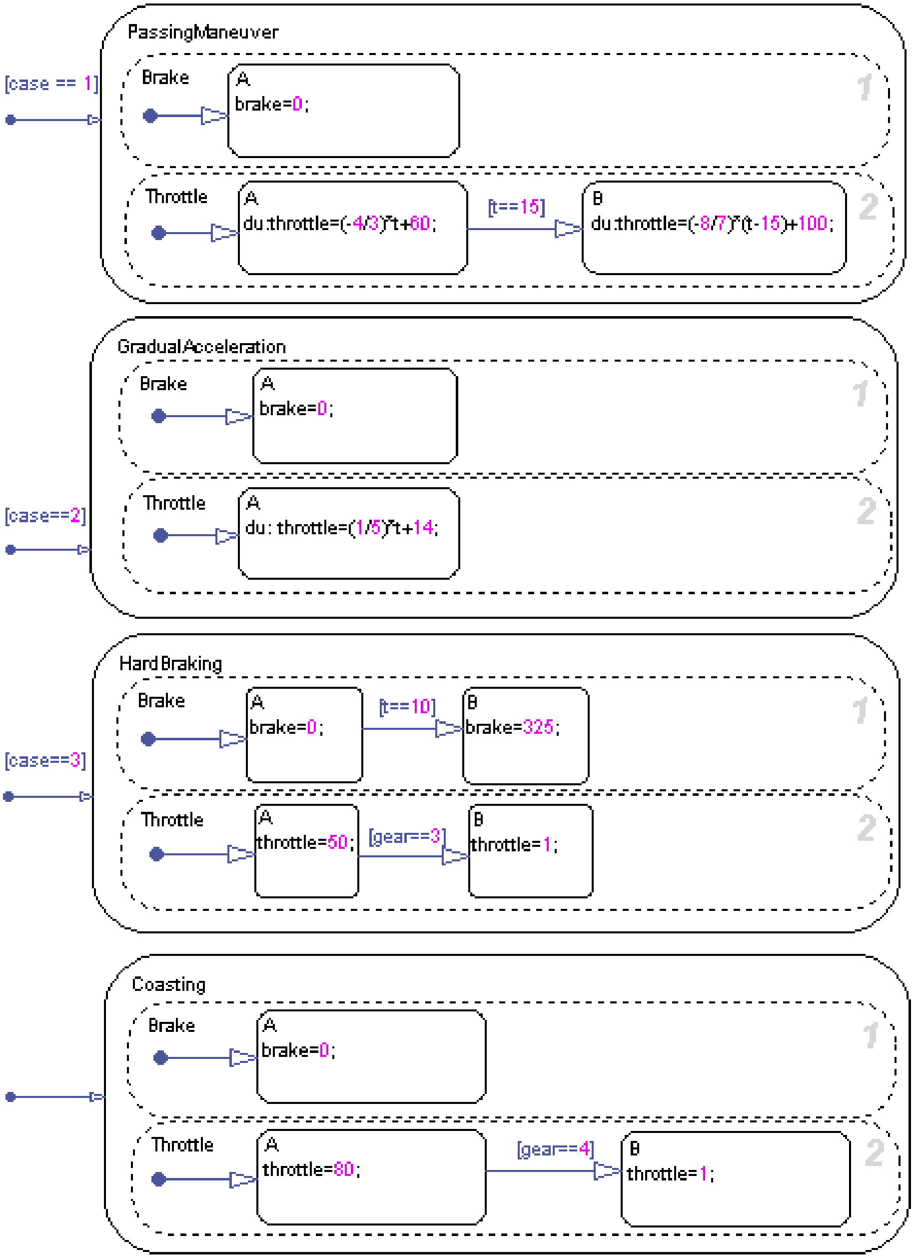
By contrast, you can use Stateflow charts to develop test vectors that use conditional logic to evaluate and respond to changes in system state as they occur. For example, to test the coasting scenario, the chart can evaluate an output that represents the gear range and reduce speed only after the transmission 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 21-24.

A Dynamic Test Vector Chart

The following model of an automatic transmission controller uses a Stateflow chart to implement test vectors that represent brake, throttle, and gear shift dynamics. The chart, called Dynamic Test Vectors, interfaces with the rest of the model as shown.



The chart models the dynamic relationship between the brake and throttle to test four driving scenarios. Each scenario is represented by a state.



In some of these scenarios, the throttle changes in response to time; in other cases, it responds to gear selection, an output of the Stateflow chart Shift_logic. The Shift_logic chart determines the gear value based on the speed of the vehicle.

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" on page 21-26
- "Chart Uses Conditional Logic to Respond to Dynamic Changes" on page 21-26
- "Model Provides an Interface for Selecting Test Cases" on page 21-27

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:

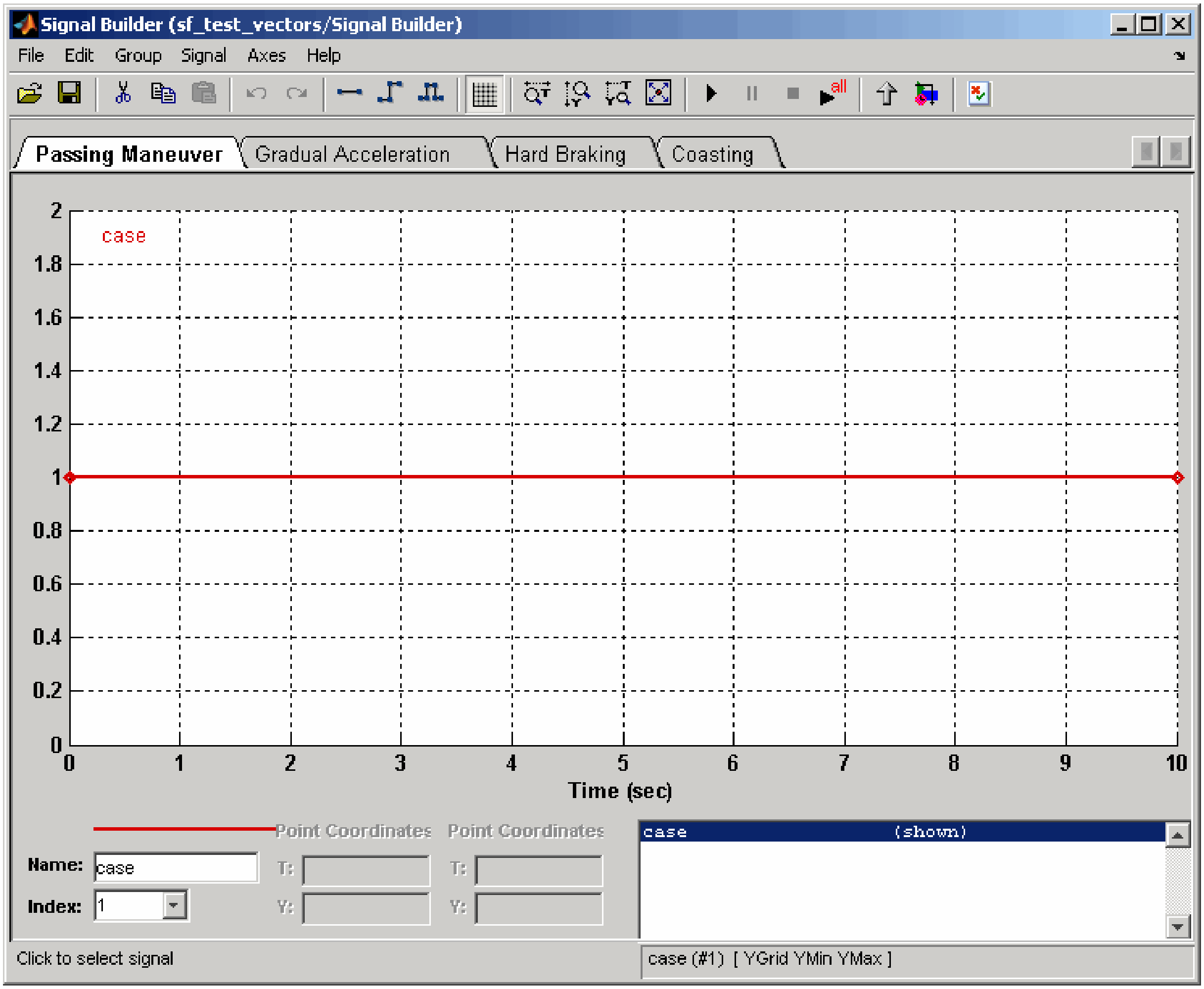
- Conditional logic (see "State Action Types" on page 10-2 and "Transition Action Types" on page 10-7)
- Temporal logic (see "Using Temporal Logic in State Actions and Transitions" on page 10-63)

- Change detection operators (see “Detecting Changes in Data Values” on page 10-83)
- MATLAB functions (see “Calling Built-In MATLAB Functions and Accessing Workspace Data” on page 10-42)

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.



Name: case

Index: 1

case (shown)

Click to select signal

case (#1) [YGrid YMin YMax]

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: ▶all

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

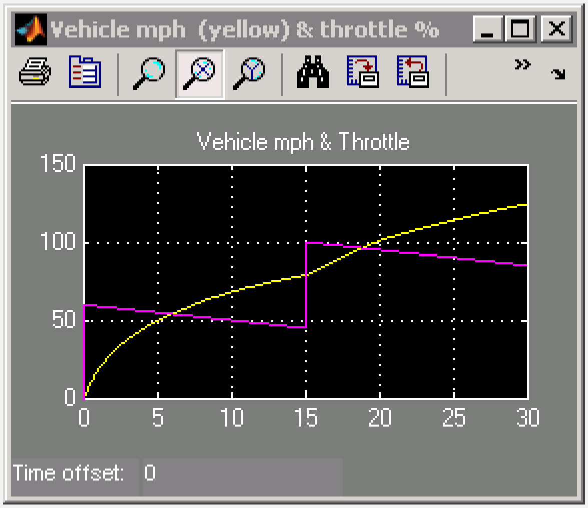
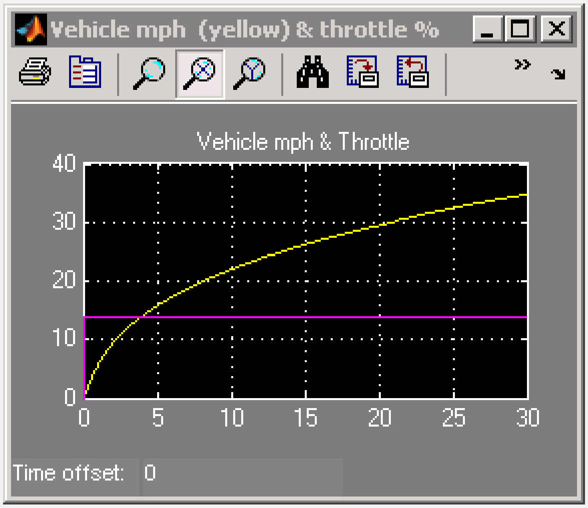
To run the `sf_test_vectors` model, follow these steps:

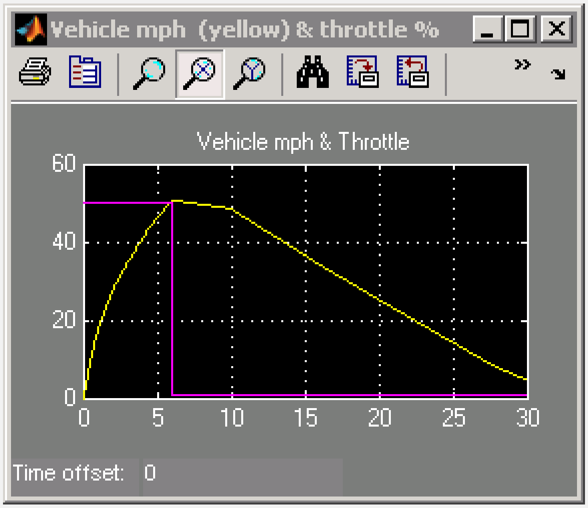
1. Open the model by typing `sf_test_vectors` at the MATLAB command prompt.

2. Open the Dynamic Test Vectors chart, the Signal Builder block, and the Scope block.

3. Select and simulate a driving scenario from the Signal Builder block, as described in “Selecting and Running Test Cases” on page 21-28.

   The scope shows the interaction between speed and throttle for the selected scenario.

Driving Scenario	Scope Display	Description
Passing Maneuver		Driver accelerates rapidly. At t = 15 seconds, steps the throttle to 100. With continued heavy throttle, the vehicle accelerates to about 100 MPH and then shifts into overdrive at about t = 21 seconds. The vehicle cruises along in fourth gear for the remainder of the simulation.
Gradual Acceleration		Driver maintains a slow but steady rate of acceleration.

Driving Scenario	**Scope Display**	**Description**
Hard Braking		Driver accelerates until the transmission shifts to third gear, then removes foot from the gas pedal. After a short delay, moves foot to the brake pedal and pushes hard.
Coasting		Driver accelerates until transmission shifts to highest gear, then eases up on the gas.

22

Truth Table Functions for Decision-Making Logic

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;
```

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
```

Why Use a Truth Table in a Stateflow Chart?

A truth table implements combinatorial logic in a concise, tabular format. Typical applications for truth tables include decision making for:

- Fault detection and management
- Mode switching

Where to Use a Truth Table

A truth table function can reside anywhere in a chart, state, or subchart. The location of a function determines its scope, that is, the set of states and transitions that can call the function. Follow these guidelines:

- If you want to call the function only within one state or subchart and its substates, put your truth table function in that state or subchart. That function overrides any other functions of the same name in the parents and ancestors of that state or subchart.
- If you want to call the function anywhere in that chart, put your truth table function at the chart level.

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.

MATLAB Truth Tables

You can specify conditions and actions for MATLAB truth tables by using MATLAB action language, which provides optimizations for code generation.

MATLAB truth tables offer several advantages over Stateflow Classic truth tables:

- The MATLAB action language provides a richer syntax for specifying control flow logic in truth table actions. It provides `for` loops, `while` loops, nested `if` statements, and `switch` statements.
- You can call MATLAB functions directly in truth table actions. Also, you can call library functions (for example, MATLAB `sin` and `fft` functions) and generate code for these functions using Simulink Coder 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.

- Better debugging tools are available. You can set breakpoints on lines of code, step through code, and watch data values using 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.

Selecting a Language for Stateflow Truth Tables

To specify an action language for your Stateflow truth table:

1. Double-click the truth table to open the Truth Table Editor.
2. Select **Language** from the **Settings** menu.
3. Select a language from the drop-down menu.

Migration from Stateflow Classic to MATLAB Truth Tables

When you migrate from a Stateflow Classic truth table to a MATLAB truth table, you must verify that the code used to program the actions conforms to MATLAB syntax. Between the two action languages, these differences exist.

For this type of action language...	Indices are...	And the expression for ***not equal to*** is...
MATLAB	One-based	`~=`
Stateflow	Zero-based	`!=`

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 22-50.

Workflow for Using Truth Tables

Here is the recommended workflow for using truth tables in Simulink models.

Step	Task	Reference
1	Add a truth table to your Simulink model.	"Building a Model with a Stateflow Truth Table" on page 22-9
2	Specify properties of the truth table function.	"Specifying Properties of Truth Table Functions in Stateflow Charts" on page 22-13
3	Select an action language and program the conditions and actions in the truth table.	"Programming a Truth Table" on page 22-24
4	Debug the truth table for syntax errors and for error during simulation.	"Debugging a Truth Table" on page 22-50
5	Simulate the model and check the generated content for the truth tables.	"How Stateflow Software Generates Content for Truth Tables" on page 22-73

Building a Model with a Stateflow Truth Table

In this section...
"Methods for Adding Truth Tables to Simulink Models" on page 22-9
"Adding a Stateflow Block that Calls a Truth Table Function" on page 22-9

Methods for Adding Truth Tables to Simulink Models

Methods for adding a Stateflow truth table to a Simulink model are:

Procedure	Action Languages Supported	How To Do It
Add a Truth Table block directly to the model.	MATLAB only	See the Truth Table block reference page.
Add a Stateflow block that calls a truth table function.	Stateflow Classic and MATLAB	See "Adding a Stateflow Block that Calls a Truth Table Function" on page 22-9.

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 include:

- "Creating a Simulink Model" on page 22-10
- "Creating a Stateflow Truth Table" on page 22-12
- "Specifying Properties of Truth Table Functions in Stateflow Charts" on page 22-13
- "Calling a Truth Table in a Stateflow Action" on page 22-16
- "Creating Truth Table Data in Stateflow Charts and Simulink Models" on page 22-19

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 22-24.

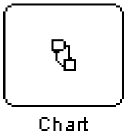
Creating a Simulink Model

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 model with a Stateflow block appears.

2. Click and drag the Stateflow block to the center of the model window.

   This step makes room for the blocks you add in the steps that follow.

3. In the model 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 blocks of the **Sources** library.

5. From the right pane of the Simulink Library Browser, click and drag the Constant block to the left of the Chart block in the 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 model, double-click the middle Constant block.

8 In the Block Parameters dialog box that appears, change **Constant value** to 0.

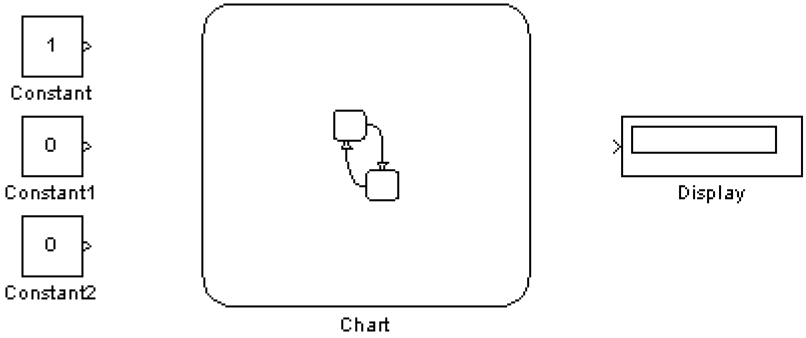
9 Click **OK** to close the dialog box.

10 In the model, double-click the bottom Constant block.

11 In the Block Parameters dialog box that appears, change **Constant value** to 0.

12 Click **OK** to close the dialog box.

Your model should now look something like this:



13 Open the Configuration Parameters dialog box.

14 On the **Solver** pane, set:

- **Type** to `Fixed-step`
- **Solver** to `discrete (no continuous states)`
- **Fixed-step size** to 1

15 Click **OK** to accept these values and close the Configuration Parameters dialog box.

16 Save the model as `ex_first_truth_table.mdl`.

Creating a Stateflow Truth Table

You created a Simulink model in “Creating a Simulink Model” on page 22-10 that contains a Stateflow block. Now you need to open the chart for the block and specify a truth table for it:

1. In your model, double-click the Chart block to open an empty chart.
2. Click the Truth Table drawing tool:

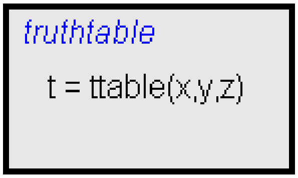
3. Move your pointer into the empty chart and notice that it appears in the shape of a box.
4. Click to place a new truth table.

   A shaded box appears with the title `truthtable`.

5. Enter the signature label

   ```
   t = ttable(x,y,z)
   ```

   and click outside the truth table box.

   truthtable

   t = ttable(x,y,z)

   The signature label of the truth table function follows this syntax:

   ```
   [return_val1, return_val2,...] = function_name(arg1, arg2,...)
   ```

   You can specify multiple return values and multiple input arguments. Each return value and input argument can be a scalar, vector, or matrix of values.

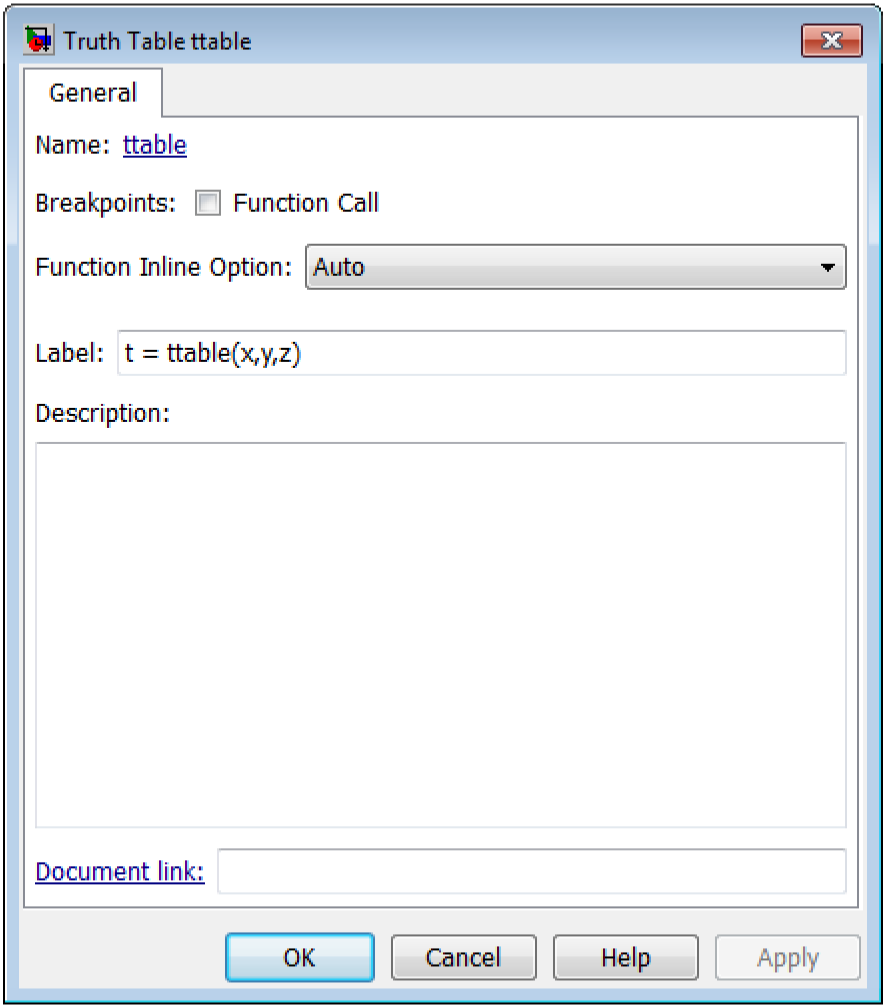
Note For functions with only one return value, you can omit the brackets in the signature label.

Specifying Properties of Truth Table Functions in Stateflow Charts

After you add a truth table function to a 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.



Truth Table ttable

Name: ttable

Breakpoints: ☐ Function Call

Function Inline Option: Auto

Label: t = ttable(x,y,z)

Description:

Document link:

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.
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 22-12 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`.

3 Click **OK** to close the dialog box.

Calling a Truth Table in a Stateflow Action

In “Creating a Stateflow Truth Table” on page 22-12, you created the truth table function `ttable` with the signature:

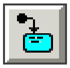
```
t = ttable(x,y,z)
```

Now you need to specify a call to the truth table function in the 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 chart in the model.

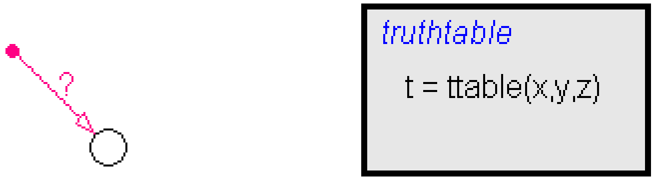
To call the `ttable` function from the default transition of its own chart, follow these steps:

1. From the toolbar, select the Default Transition tool:

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 default transition.

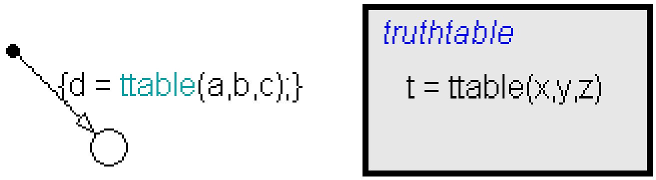
A blinking cursor in a text field appears for entering the label of the default transition.

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 chart should look something like this:



```
{d = ttable(a,b,c);}
```

```
t = ttable(x,y,z)
```

The label on the default transition 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 executes and a call to the truth table `ttable` occurs.

The call to the truth table in Stateflow action language must match the truth table signature. 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 chart.

Tip If the formal arguments of a function signature are scalars, verify that inputs and outputs of function calls follow the rules of scalar expansion. For more information, see "How Scalar Expansion Works for Functions" on page 13-6.

---

6 Save the model.

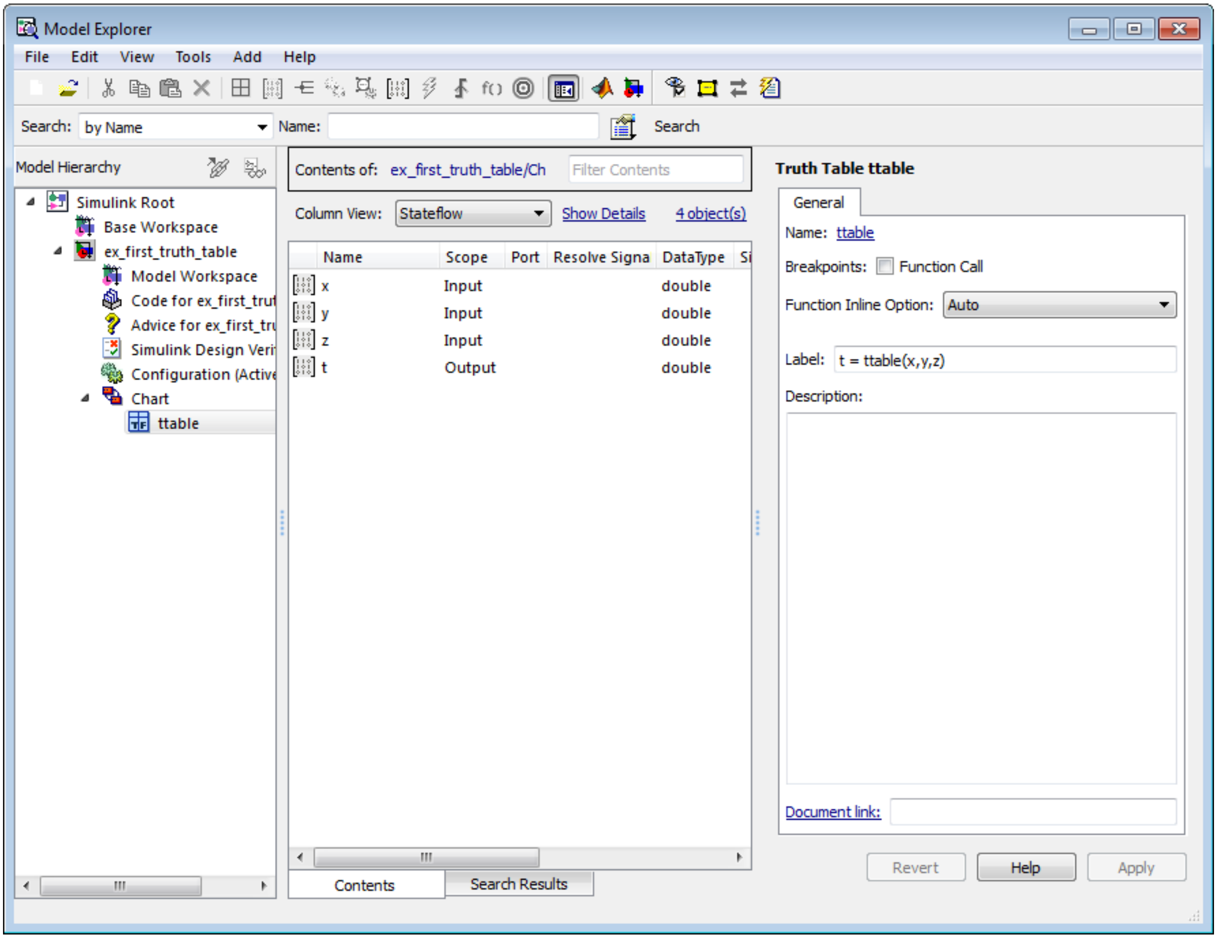
Creating Truth Table Data in Stateflow Charts and Simulink Models

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 22-16, 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 chart as follows:

1. Double-click the truth table function to open the Truth Table Editor.

2. In the Truth Table Editor, select **Add > Edit Data/Ports**.

   The Model Explorer appears.



Search: by Name

Name:

Contents of: ex_first_truth_table/Ch

Filter Contents

Column View: Stateflow

Name	Scope	Port	Resolve Signa	DataType	Si
x	Input			double	
y	Input			double	
z	Input			double	
t	Output			double	

Name: ttable

Breakpoints: ☐ Function Call

Function Inline Option: Auto

Label: t = ttable(x,y,z)

In the **Model Hierarchy** pane, the node for the function `ttable` appears highlighted, and the **Contents** pane displays the output (t) and inputs (`x`, `y`, `z`) for `ttable`. By default, these data are scalars of type `double`. If you want to redefine these data with a different 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 chart that contains the truth table `ttable`.

How do I enable the third pane in the Model Explorer?

To enable or disable the third pane in the Model Explorer, select **View > Show Dialog Pane**.

3 In the **Model Hierarchy** pane, select **Chart**.

Notice that `Chart` contains no input or output data in the **Contents** pane. You must add the return and argument data for 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.

How do I verify type and size?

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`.

7 Under **Scope**, click the entry `Local` .

A drop-down menu of selectable scopes appears with `Local` selected.

8 Select `Input`.

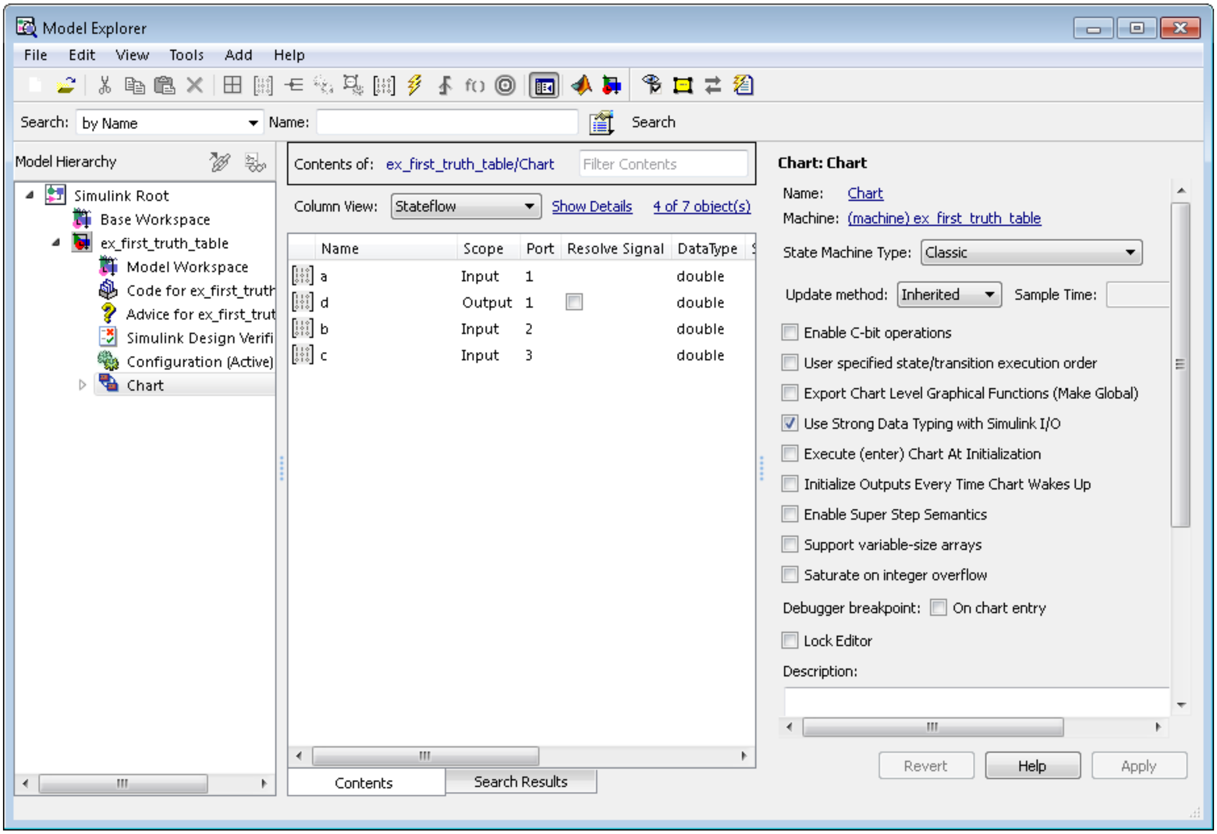
The scope `Input` means that the Simulink model provides the value for this data, which passes to the chart through an input port on the Stateflow block.

You should now see the new data input `a` in the **Contents** pane.

9 Repeat steps 4 through 8 to add the data `b` and `c` with the scope `Input`, and data `d` with the scope `Output`.

The scope `Output` means that the chart provides this data, which passes to the Simulink model through an output port on the Stateflow block.

You should now see the input and output data in the Model Explorer.



Contents of: ex_first_truth_table/Chart

Column View: Stateflow

Name	Scope	Port	Resolve Signal	DataType
a	Input	1		double
d	Output	1	☐	double
b	Input	2		double
c	Input	3		double

Chart: Chart

Name: Chart

Machine: (machine) ex_first_truth_table

State Machine Type: Classic

Update method: Inherited Sample Time:

☐ Enable C-bit operations

☐ 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

☐ Support variable-size arrays

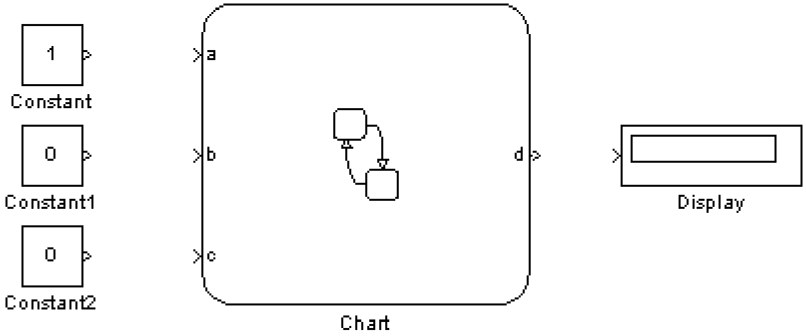
☐ Saturate on integer overflow

Debugger breakpoint: ☐ On chart entry

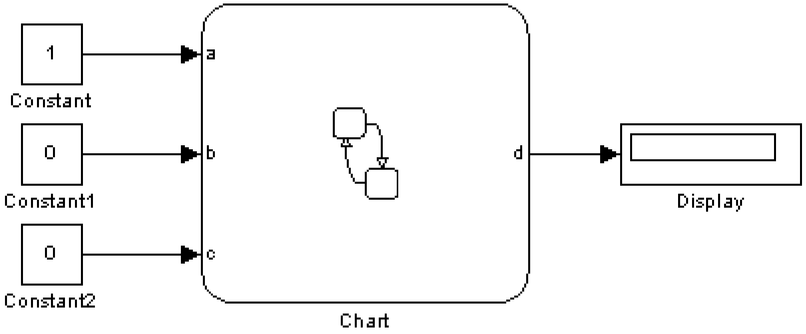
☐ Lock Editor

Description:

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 model.



10 Complete connections to the Simulink blocks:



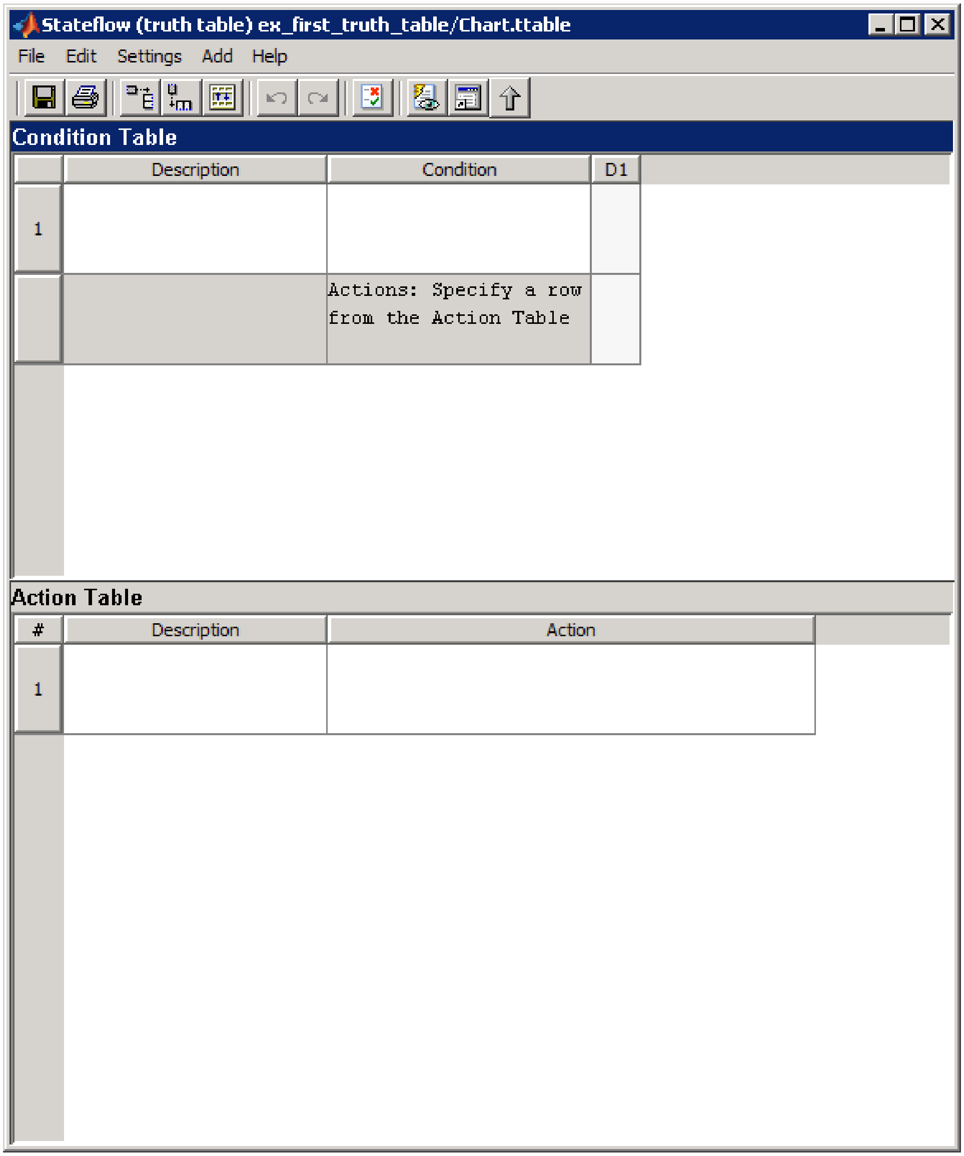
11 Save the model.

Programming a Truth Table

In this section...

Opening a Truth Table for Editing

After you create and label a truth table in a chart, you specify its logical behavior. Double-click the truth table function to open the Truth Table Editor.



	Description	Condition	D1
1			
		Actions: Specify a row from the Action Table	

#	Description	Action
1		

By default, a 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:

1. In the Truth Table Editor, select **Settings > Language**.
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 22-24. In this topic, you start programming the behavior of `ttable` by specifying conditions.

You enter conditions in the **Condition** column of the **Condition Table**. For each condition that you enter, you can also enter an optional description in the **Description** column. Use the following procedure to enter conditions for the truth table `ttable`:

1. Click anywhere in the **Condition Table** to select it.
2. Select **Edit > Append Row** twice.

   The editor appends two rows to the bottom of the **Condition Table**.
3. Click and drag the bar that separates the **Condition Table** and the **Action Table** panes down to enlarge the **Condition Table** pane.
4. In the **Condition Table**, click the top cell of the **Description** column.

   A flashing text cursor appears in the cell, which appears highlighted.
5. Enter 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 cell of the **Condition** column, enter the following text:

```
XEQ1:
```

This text is an optional label you can include with the condition. Each label must begin with an alphabetic character (`[a-z][A-Z]`) followed by any number of alphanumeric characters (`[a-z][A-Z][0-9]`) or an underscore (_).

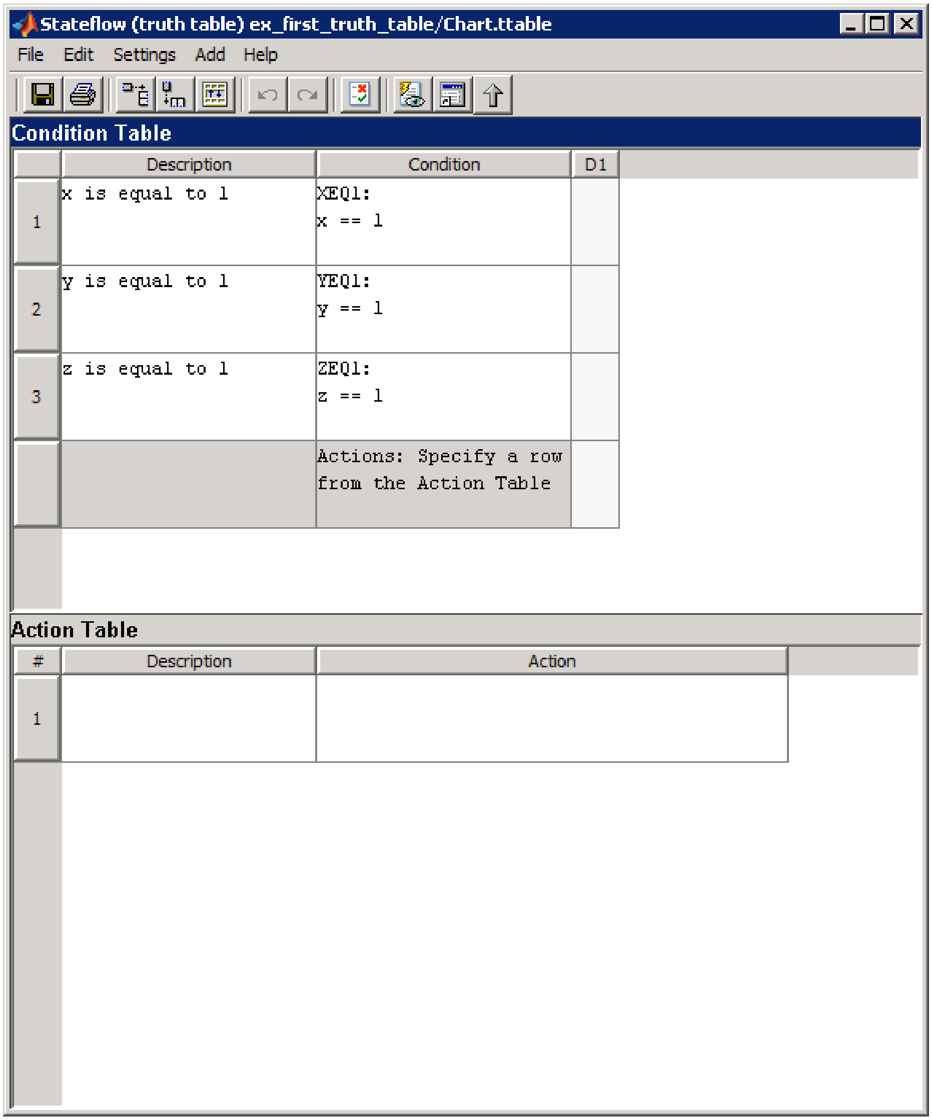
8 Press **Enter** and enter the following text:

```
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 do in Stateflow action language.

In truth table conditions, you can use data that passes to the truth table function through its arguments. The preceding condition tests whether the argument `x` is equal to 1. You can also use data defined for parent objects of the truth table, including the chart.

9 Repeat the preceding steps to enter the other two conditions.



Condition Table

	Description	Condition	D1
1	x is equal to 1	XEQ1: x == 1	
2	y is equal to 1	YEQ1: y == 1	
3	z is equal to 1	ZEQ1: z == 1	
		Actions: Specify a row from the Action Table	

Action Table

#	Description	Action
1		

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 possible values for condition outcomes in a decision are `T` (true), `F` (false), and `-` (true or false). In "Entering Truth Table Conditions" on page 22-26 you entered conditions for the truth table `ttable`. Continue by entering values in the decision columns:

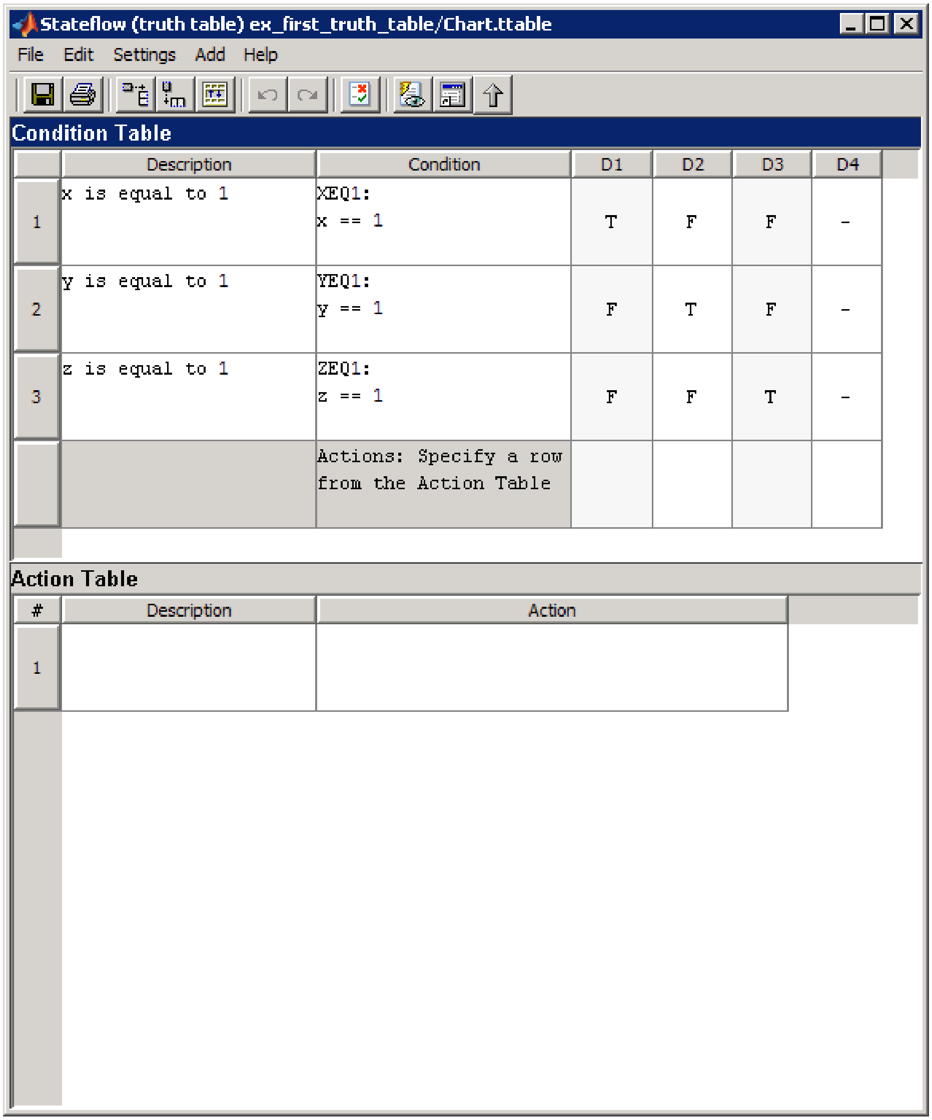
1. Click anywhere in the **Condition Table** to select it.
2. Select **Edit > Append Column** three times to add three columns to the right end of the **Condition Table**.
3. Click the top cell in decision column **D1**.

   A flashing text cursor appears in the cell, which appears highlighted.
4. Press the space bar until a value of `T` appears.

   Pressing the space bar toggles through the possible values of `F`, `-`, and `T`. You can also enter these characters directly. The editor rejects all other entries.
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:



Condition Table

	Description	Condition	D1	D2	D3	D4
1	x is equal to 1	XEQ1: x == 1	T	F	F	-
2	y is equal to 1	YEQ1: y == 1	F	T	F	-
3	z is equal to 1	ZEQ1: z == 1	F	F	T	-
		Actions: Specify a row from the Action Table				

Action Table

#	Description	Action
1		

During execution of the truth table, decision testing occurs in left-to-right order. The order of testing for individual condition outcomes within a decision is undefined. Truth tables evaluate the conditions for each decision in top-down order (first condition 1, then condition 2, and so on). Because this implementation is subject to change in the future, do not rely on a specific evaluation order.

The Default Decision Column

The last decision column in `ttable`, **D4**, is the default decision for this truth table. The default decision covers any decisions not tested for in preceding decision columns to the left. You enter a default decision as the last decision column on the right with an entry of `-` for all conditions in the decision. This entry represents any outcome for the condition, `T` or `F`.

In the preceding example, the default decision column, **D4**, specifies these decisions:

Condition	**Decision 4**	**Decision 5**	**Decision 6**	**Decision 7**	**Decision 8**
x == 1	F	T	F	T	T
y == 1	F	F	T	T	T
z == 1	F	T	T	F	T

Tip The default decision column must be the last column on the right in the **Condition Table**.

Entering Truth Table Actions

During execution of the truth table, decision testing occurs in left-to-right order. When a decision match occurs, the action in the **Action Table** specified in the **Actions** row for that decision column executes. Then the truth table exits.

In “Entering Truth Table Decisions” on page 22-29, you entered decisions in the Truth Table Editor. The next step is to enter actions you want to occur for each decision in the **Action Table**. Later, you assign these actions to their decisions in the **Actions** row of the **Condition Table**.

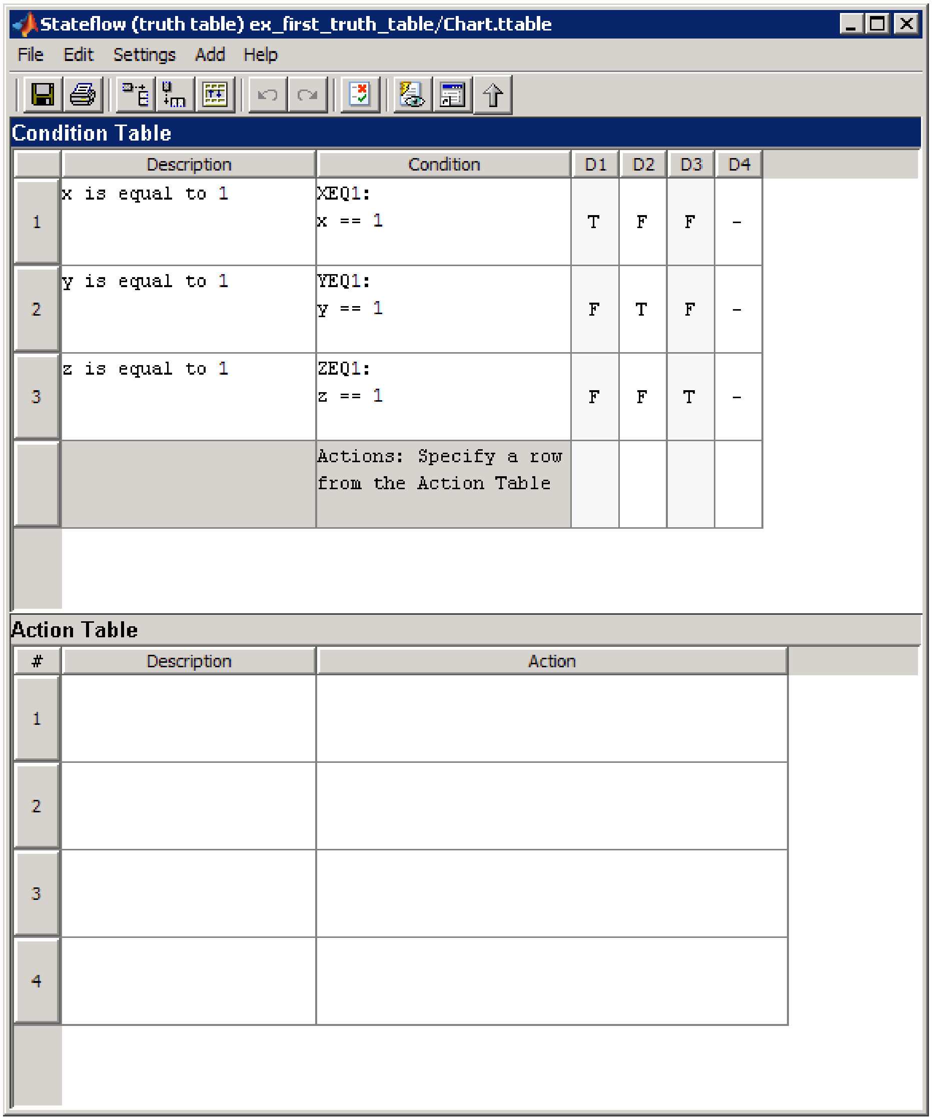
This section describes how to program truth table actions with these topics:

- "Setting Up the Action Table" on page 22-32 — Shows you how to set up the Action Table in truth table `ttable`.
- "Programming Actions in Stateflow Classic Action Language" on page 22-34 — Provides sample code in Stateflow action language to program actions in `ttable`. Follow this section if you selected **Stateflow Classic** as the language for this truth table.
- "Programming Actions in MATLAB Action Language" on page 22-37 — Provides sample MATLAB code to program actions in `ttable`. Follow this section if you selected **MATLAB** as the language for this truth table.

Setting Up the Action Table

1. Click anywhere in the **Action Table** to select it.

2 Select **Edit > Append Row** three times to add three rows to the bottom of the **Action Table**:



Condition Table

	Description	Condition	D1	D2	D3	D4
1	x is equal to 1	XEQ1: x == 1	T	F	F	-
2	y is equal to 1	YEQ1: y == 1	F	T	F	-
3	z is equal to 1	ZEQ1: z == 1	F	F	T	-
		Actions: Specify a row from the Action Table				

Action Table

#	Description	Action
1		
2		
3		
4		

3 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 22-34
MATLAB	"Programming Actions in MATLAB Action Language" on page 22-37

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**.

   A flashing text cursor appears in the cell, which appears highlighted.

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:

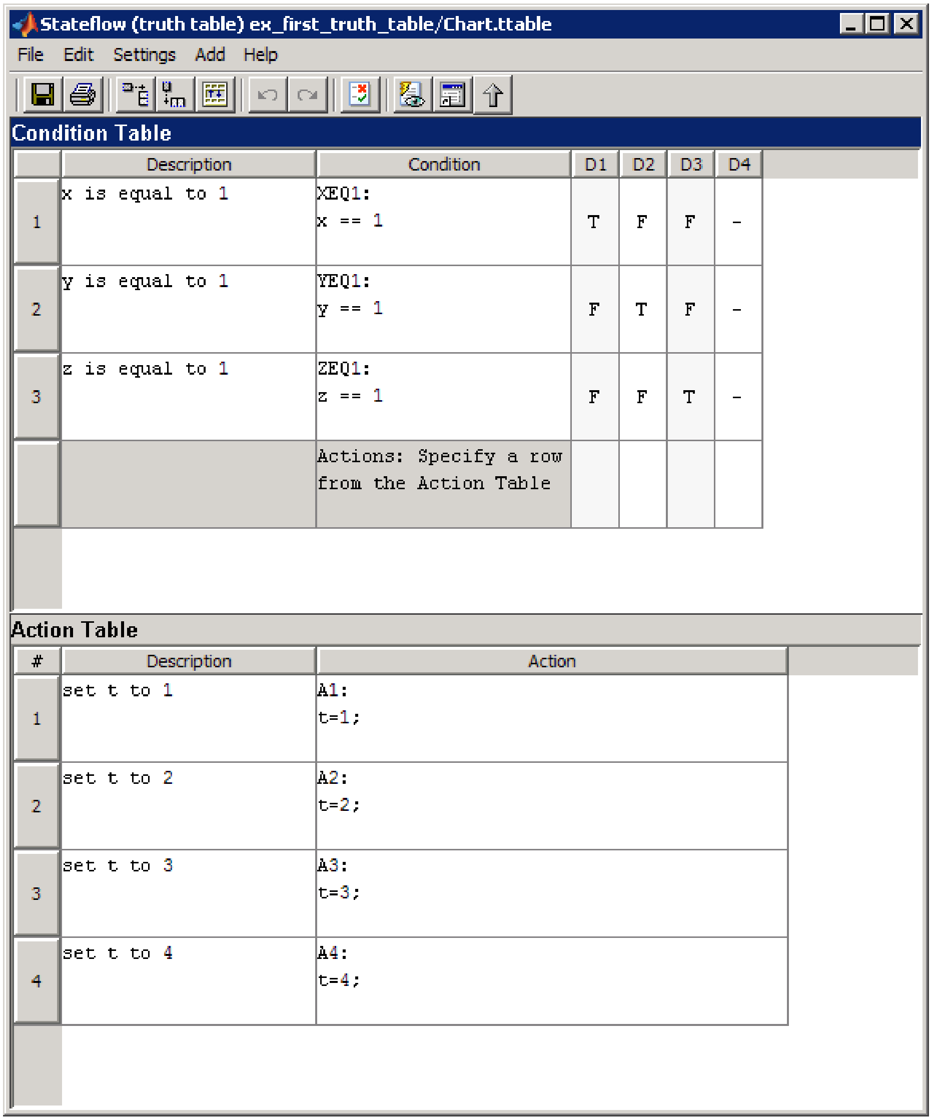
   ```
   t=1;
   ```

   In truth table actions, you can use data that passes to the truth table function through its arguments and return value. 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 chart. Truth table actions can also broadcast or send events that are defined for the truth table, or for a parent, such as the chart itself.

Tip If you omit the semicolon at the end of an action, the result of the action echoes to the MATLAB Command Window when the action executes during simulation. Use this echoing option as a debugging tool.

---

6 Enter the remaining actions in the **Action Table**, as shown:



Condition Table

	Description	Condition	D1	D2	D3	D4
1	x is equal to 1	XEQ1: x == 1	T	F	F	-
2	y is equal to 1	YEQ1: y == 1	F	T	F	-
3	z is equal to 1	ZEQ1: z == 1	F	F	T	-
		Actions: Specify a row from the Action Table				

Action Table

#	Description	Action
1	set t to 1	A1: t=1;
2	set t to 2	A2: t=2;
3	set t to 3	A3: t=3;
4	set t to 4	A4: t=4;

Now you are ready to assign actions to decisions, as described in “Assigning Truth Table Actions to Decisions” on page 22-41.

Programming Actions in MATLAB Action Language

If you selected MATLAB action language, you can write MATLAB code to program your actions. Using this code, you can add control flow logic and call MATLAB functions directly. In the following procedure, you program an action in the truth table `ttable`, using the following features of MATLAB syntax:

- Persistent variables
- `if ... else ... end` control flows
- `for` loop

Follow these steps:

1. Click the top cell in the **Description** column of the **Action Table**.

   A flashing text cursor appears in the cell, which appears highlighted.

2. Enter this description:

   ```
   Maintain a counter and a circular
   vector of values of length 6.
   Every time this action is called,
   output 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 (_).

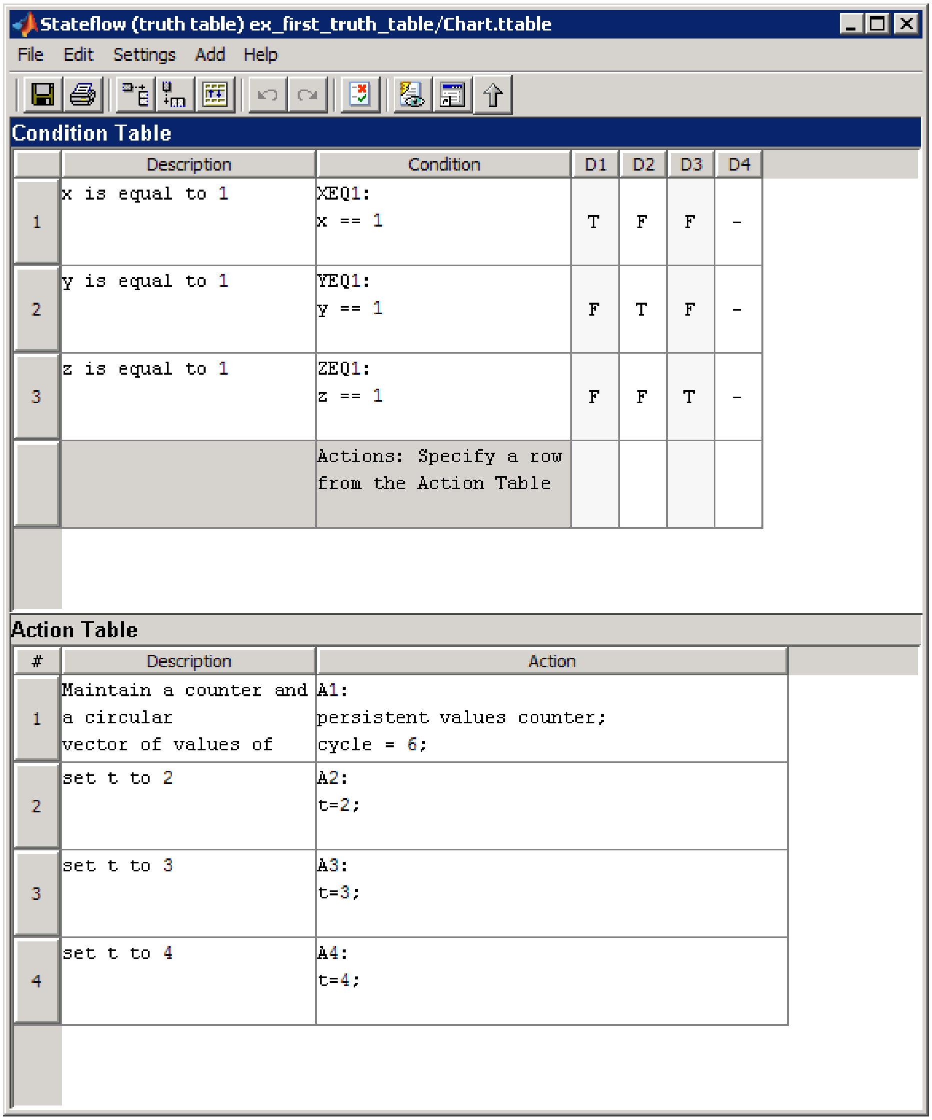
5 Press **Enter** and enter the following text:

```
persistent values counter;
cycle = 6;

coder.extrinsic('plot');

if isempty(counter)
   % Initialize counter to be zero
   counter = 0;
else
   % Otherwise, increment counter
   counter = counter + 1;
end

if isempty(values)
   % Values is a vector of 1 to cycle
   values = zeros(1, cycle);
   for i = 1:cycle
      values(i) = i;
   end

   % For debugging purposes, call the MATLAB
   % function "plot" to show values
   plot(values);
end

% Output t takes the next value in values vector
t = values( mod(counter, cycle) + 1);
```

In truth table actions, you can use data that passes to the truth table function through its arguments and return value. The preceding action sets the return value `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 chart. Truth table actions can also broadcast or send events that are defined for the truth table, or for a parent, such as the chart itself.

Note If you omit the semicolon at the end of an action, the result of the action echoes to the MATLAB Command Window when the action executes during simulation. Use this echoing option as a debugging tool.

6 Enter the remaining actions in the **Action Table**, as shown:



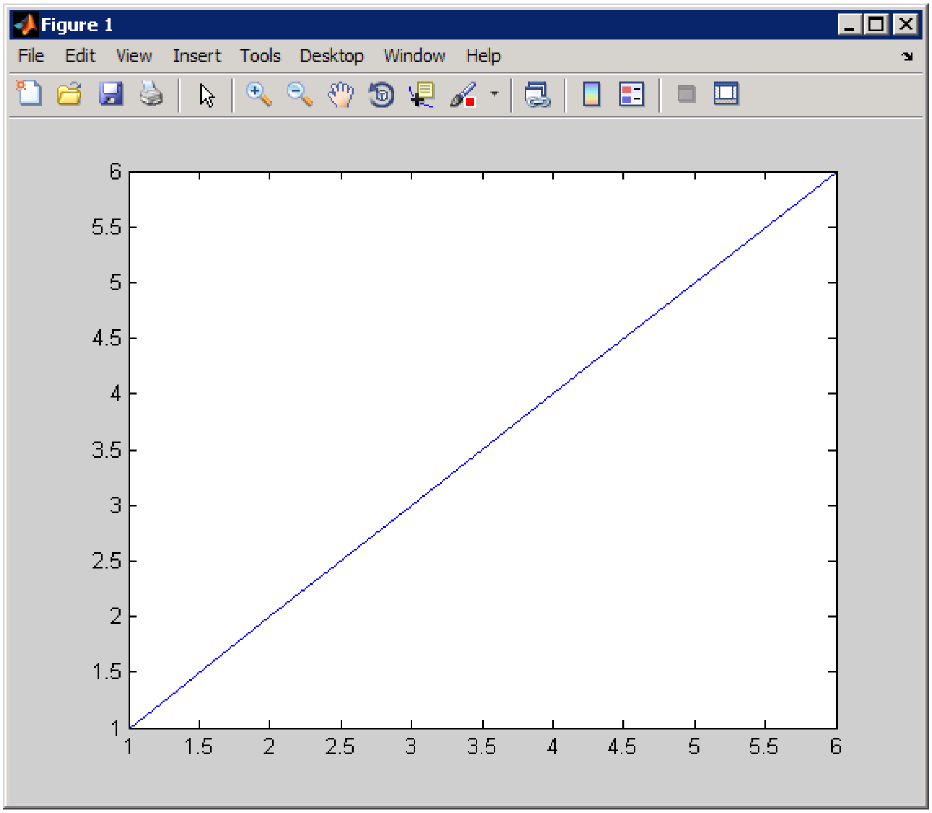
Condition Table

	Description	Condition	D1	D2	D3	D4
1	x is equal to 1	XEQ1: x == 1	T	F	F	-
2	y is equal to 1	YEQ1: y == 1	F	T	F	-
3	z is equal to 1	ZEQ1: z == 1	F	F	T	-
		Actions: Specify a row from the Action Table				

Action Table

#	Description	Action
1	Maintain a counter and a circular vector of values of	A1: persistent values counter; cycle = 6;
2	set t to 2	A2: t=2;
3	set t to 3	A3: t=3;
4	set t to 4	A4: t=4;

If action `A1` executes during simulation, a plot of the `values` vector appears:



Now you are ready to assign actions to decisions, as described in "Assigning Truth Table Actions to Decisions" on page 22-41.

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 uses this association to determine what action to execute when a decision tests as true.

Rules for Assigning Actions to Decisions

The following rules apply when you assign actions to decisions in a truth table:

- You specify actions for decisions by entering a row number or a label in the **Actions** row cell of a decision column.

  If you use a label specifier, the label must appear with the action in the **Action Table**.

- You must specify at least one action for each decision.

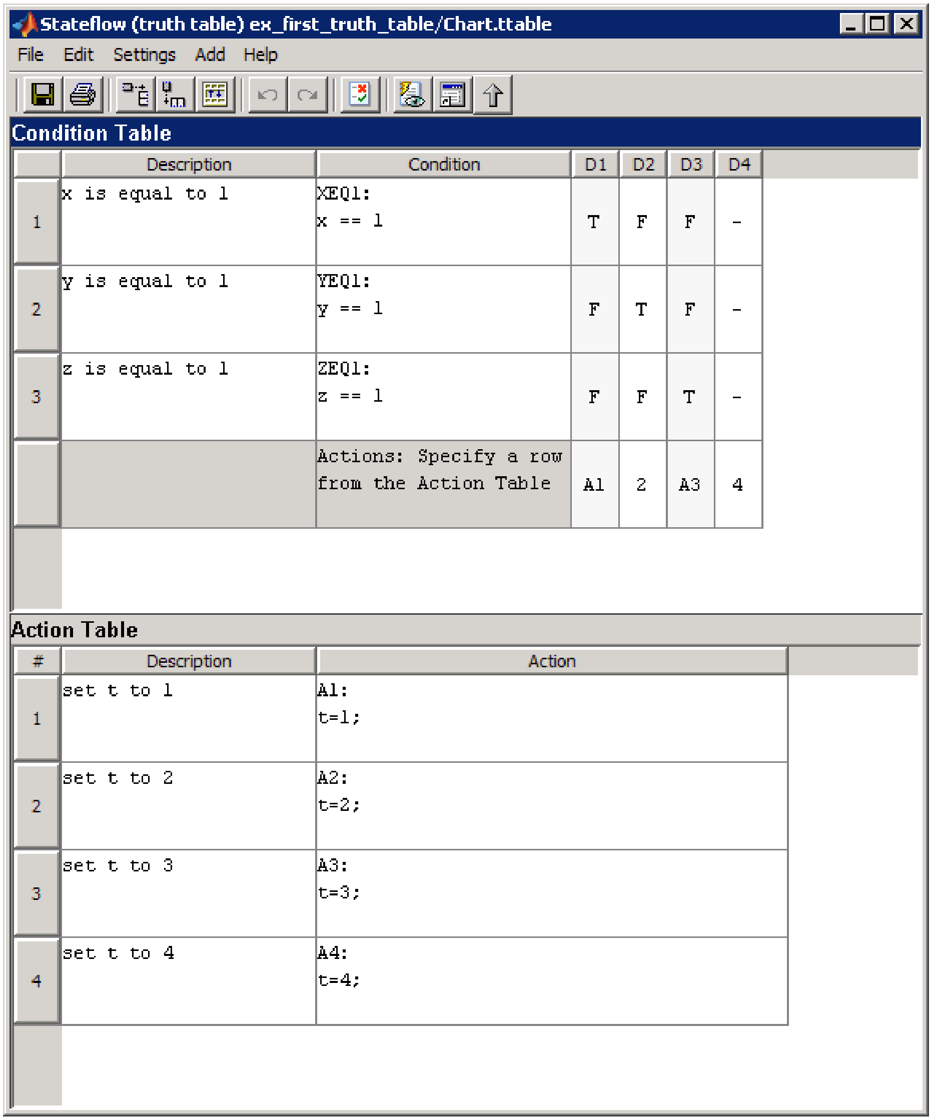
  Actions for decisions are not optional. Each decision must have at least one action specifier that points to an action in the **Action Table**. If you want to specify no action for a decision, specify a row that contains no action statements.

- You can specify multiple actions for a decision with multiple specifiers separated by a comma, semicolon, or space.

  For example, for the decision column **D1**, you can specify `A1,A2,A3` or `1;2;3` to execute the first three actions when decision **D1** is true.

- You can mix row number and label action specifiers interchangeably in any order.

The following example uses both row and label action specifiers.



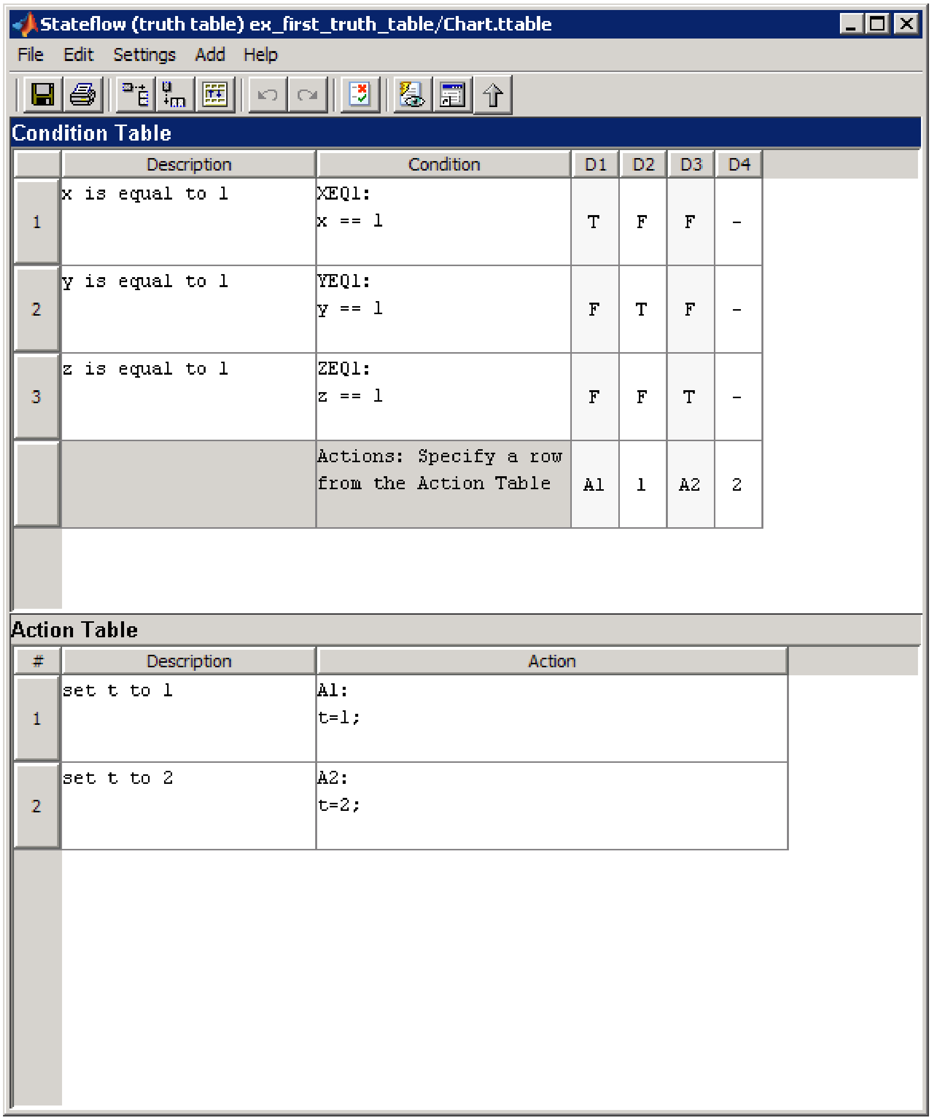
Condition Table

	Description	Condition	D1	D2	D3	D4
1	x is equal to 1	XEQ1: x == 1	T	F	F	-
2	y is equal to 1	YEQ1: y == 1	F	T	F	-
3	z is equal to 1	ZEQ1: z == 1	F	F	T	-
		Actions: Specify a row from the Action Table	A1	2	A3	4

Action Table

#	Description	Action
1	set t to 1	A1: t=1;
2	set t to 2	A2: t=2;
3	set t to 3	A3: t=3;
4	set t to 4	A4: t=4;

- You can specify the same action for more than one decision, as shown:



Condition Table

	Description	Condition	D1	D2	D3	D4
1	x is equal to 1	XEQ1: x == 1	T	F	F	-
2	y is equal to 1	YEQ1: y == 1	F	T	F	-
3	z is equal to 1	ZEQ1: z == 1	F	F	T	-
		Actions: Specify a row from the Action Table	A1	1	A2	2

Action Table

#	Description	Action
1	set t to 1	A1: t=1;
2	set t to 2	A2: t=2;

- Row number action specifiers in the **Actions** row of the **Condition Table** automatically adjust to changes in the row order of the **Action Table**.

How to Assign Actions to Decisions

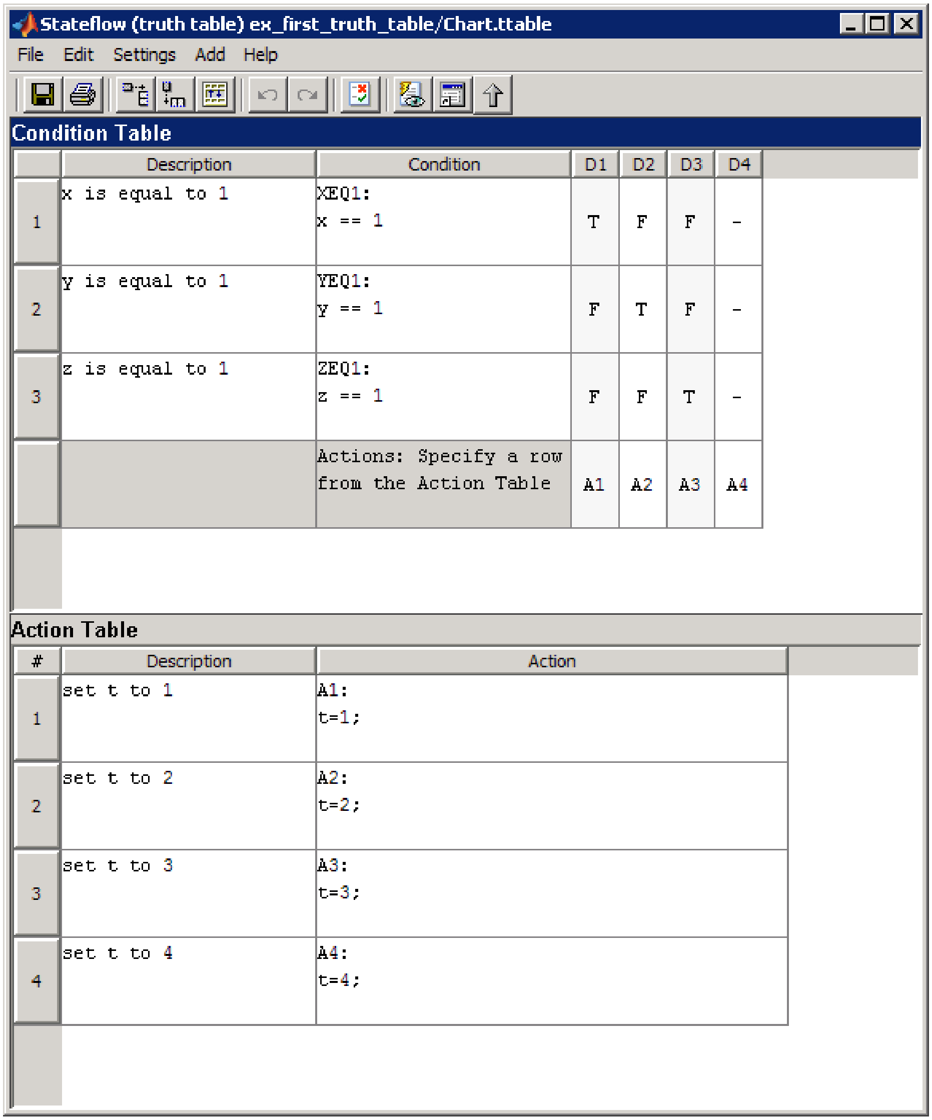
This section describes how to assign actions to decisions in the truth table `ttable`. In this example, the **Actions** row cell for each decision column contains a label specified for each action in the **Action Table**. Follow these steps:

1. Click the bottom cell in decision column **D1**, the first cell of the **Actions** row of the **Condition Table**.

2. Enter the action specifier `A1` for decision column **D1**.

   When **D1** is true, action `A1` in the **Action Table** executes.

3. Enter the action specifiers for the remaining decision columns:



Condition Table

	Description	Condition	D1	D2	D3	D4
1	x is equal to 1	XEQ1: x == 1	T	F	F	-
2	y is equal to 1	YEQ1: y == 1	F	T	F	-
3	z is equal to 1	ZEQ1: z == 1	F	F	T	-
		Actions: Specify a row from the Action Table	A1	A2	A3	A4

Action Table

#	Description	Action
1	set t to 1	A1: t=1;
2	set t to 2	A2: t=2;
3	set t to 3	A3: t=3;
4	set t to 4	A4: t=4;

Now you are ready to perform the final step in programming a truth table, "Adding Initial and Final Actions" on page 22-47.

Adding Initial and Final Actions

In addition to actions for decisions, you can add initial and final actions to the truth table function. Initial actions specify an action that executes before any decision testing occurs. Final actions specify an action that executes as the last action before the truth table exits. To specify initial and final actions for a truth table, use the action labels `INIT` and `FINAL` in the **Action Table**.

Use this procedure to add initial and final actions that display diagnostic messages in the MATLAB Command Window before and after execution of the truth table `ttable`:

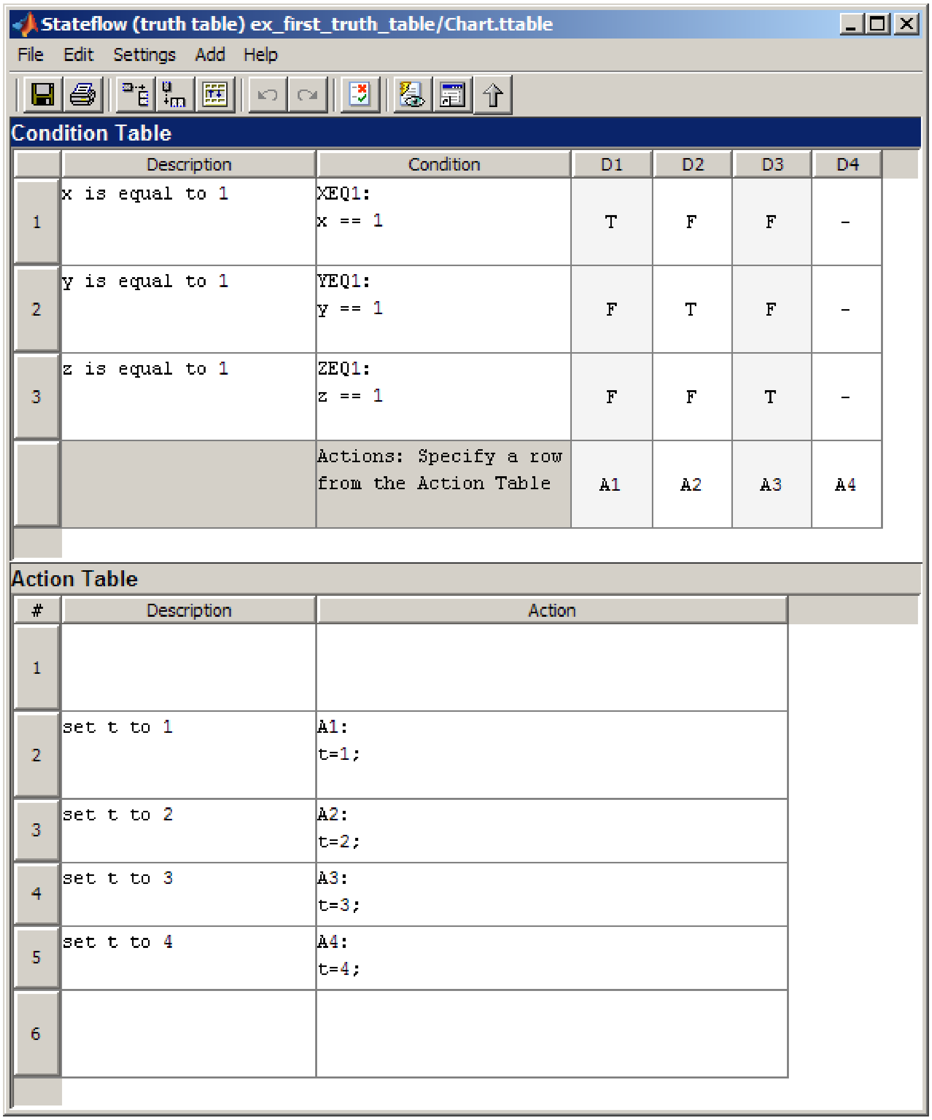
1. In the Truth Table Editor, right-click row 1 of the **Action Table** and select **Insert Row**.

   A blank row appears at the top of the **Action Table**.

2. Select **Edit > Append Row**.

   A blank row appears at the bottom of the **Action Table**.

3 Click and drag the bottom border of the Truth Table Editor to show all six rows of the **Action Table**:



Condition Table

	Description	Condition	D1	D2	D3	D4
1	x is equal to 1	XEQ1: x == 1	T	F	F	-
2	y is equal to 1	YEQ1: y == 1	F	T	F	-
3	z is equal to 1	ZEQ1: z == 1	F	F	T	-
		Actions: Specify a row from the Action Table	A1	A2	A3	A4

Action Table

#	Description	Action
1		
2	set 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		

4 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');`

MATLAB	Initial action:
	Display message
		`INIT:`
		`coder.extrinsic('disp');`
		`disp('truth table ttable entered');`

5 Add the final action in row 6 as follows:

Truth Table Type	Description	Action
Stateflow Classic	Final action: Display message	`FINAL:` `ml.disp('truth table ttable exited');`

MATLAB	Final action:
	Display message
		`FINAL:`
		`coder.extrinsic('disp');`
		`disp('truth table ttable exited');`

Although the initial and final actions for the preceding truth table example appear in the first and last rows of the **Action Table**, you can enter these actions in any row. You can also assign initial and final actions to decisions by using the action specifier `INIT` or `FINAL` in the **Actions** row of the **Condition Table**.

Debugging a Truth Table

In this section...
"Checking Truth Tables for Errors" on page 22-50
"Debugging a Truth Table During Simulation" on page 22-51

Checking Truth Tables for Errors

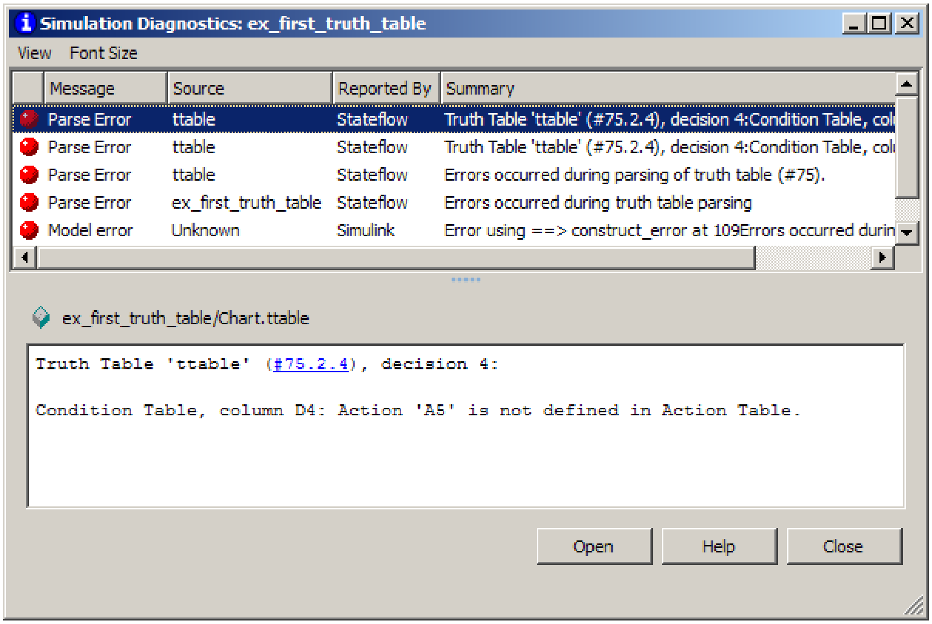
Once you completely specify your truth tables, you begin the process of debugging them. The first step is to run diagnostics to check truth tables for syntax errors including overspecification and underspecification, as described in "Correcting Overspecified and Underspecified Truth Tables" on page 22-64.

To check for syntax errors:

1 Double-click the truth table to open its editor.

2 In the Truth Table Editor, select **Settings > Run Diagnostics**.

   If there are no errors or warnings, a message of success appears. If errors exist, you see a window with diagnostic messages. For example, if you change the action for decision column **D4** to an action that does not exist, you get the following messages:



	Message	Source	Reported By	Summary
●	Parse Error	ttable	Stateflow	Truth Table 'ttable' (#75.2.4), decision 4:Condition Table, col
●	Parse Error	ttable	Stateflow	Truth Table 'ttable' (#75.2.4), decision 4:Condition Table, col
●	Parse Error	ttable	Stateflow	Errors occurred during parsing of truth table (#75).
●	Parse Error	ex_first_truth_table	Stateflow	Errors occurred during truth table parsing
●	Model error	Unknown	Simulink	Error using ==> construct_error at 109Errors occurred durin

ex_first_truth_table/Chart.ttable

```
Truth Table 'ttable' (#75.2.4), decision 4:

Condition Table, column D4: Action 'A5' is not defined in Action Table.
```

Each 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 exist, the diagnostic window does not appear and simulation starts immediately.

Debugging a Truth Table During Simulation

Ways to debug truth tables during simulation include:

Method	Type of Truth Tables	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 MATLAB truth table	See "Using Stateflow Debugging Tools" on page 22-52.
Use MATLAB debugging tools to step through generated code for the truth table.	MATLAB truth table only	See "Using MATLAB Debugging Tools" on page 22-63.

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. This breakpoint pauses execution during simulation so that you can debug each execution step of a truth table using the Stateflow Debugger.

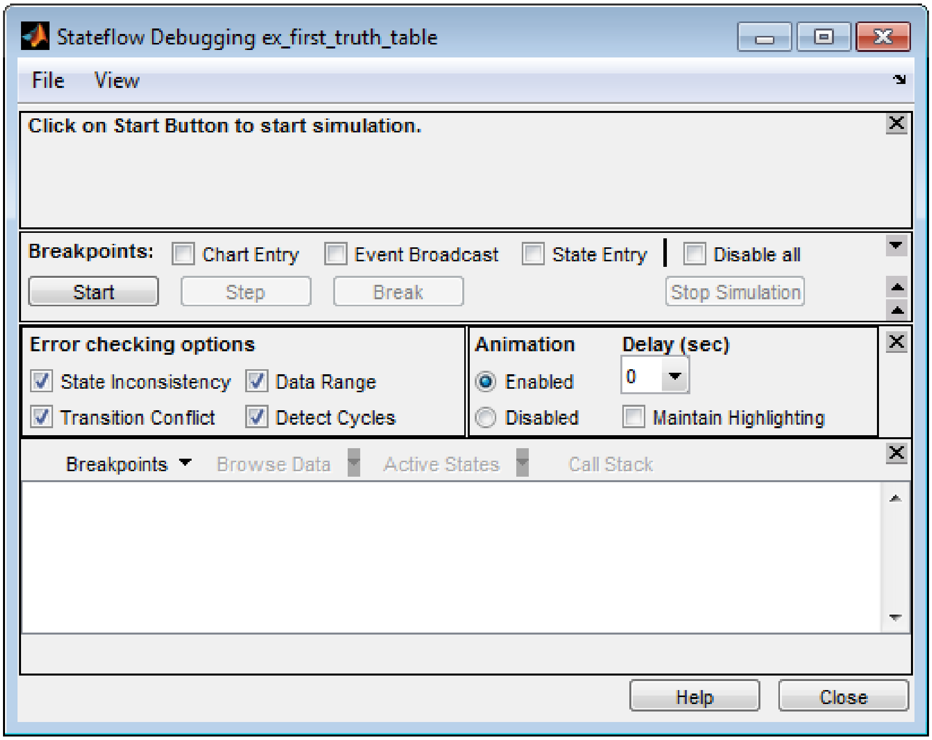
1. In the chart, right-click the function box for the truth table and select **Breakpoints**.
2. Select **Function Call**.

   A breakpoint occurs when the chart calls this truth table function during simulation.

Note You can also set breakpoints using the Truth Table properties dialog box. However, using the right-click context menu is faster. For more information, see "Setting Breakpoints to Debug Charts" on page 26-7.

Stepping Through Conditions and Actions of a Truth Table. After setting a breakpoint for the truth table function call, you can step through conditions and actions:

1. Enter `sfdebugger` at the command prompt to open the Stateflow Debugging window.



Stateflow Debugging ex_first_truth_table

Click on Start Button to start simulation.

Breakpoints: ☐ Chart Entry ☐ Event Broadcast ☐ State Entry ☐ Disable all

Error checking options

☑ State Inconsistency ☑ Data Range

☑ Transition Conflict ☑ Detect Cycles

Animation: ● Enabled ○ Disabled

Delay (sec): 0

☐ Maintain Highlighting

2. In the Stateflow Debugging window, click **Start** to begin simulation of your model.

   If you made any changes to the truth tables since the last simulation, the debugger checks automatically for syntax errors. If you receive errors or warnings, make corrections before you try to simulate again.

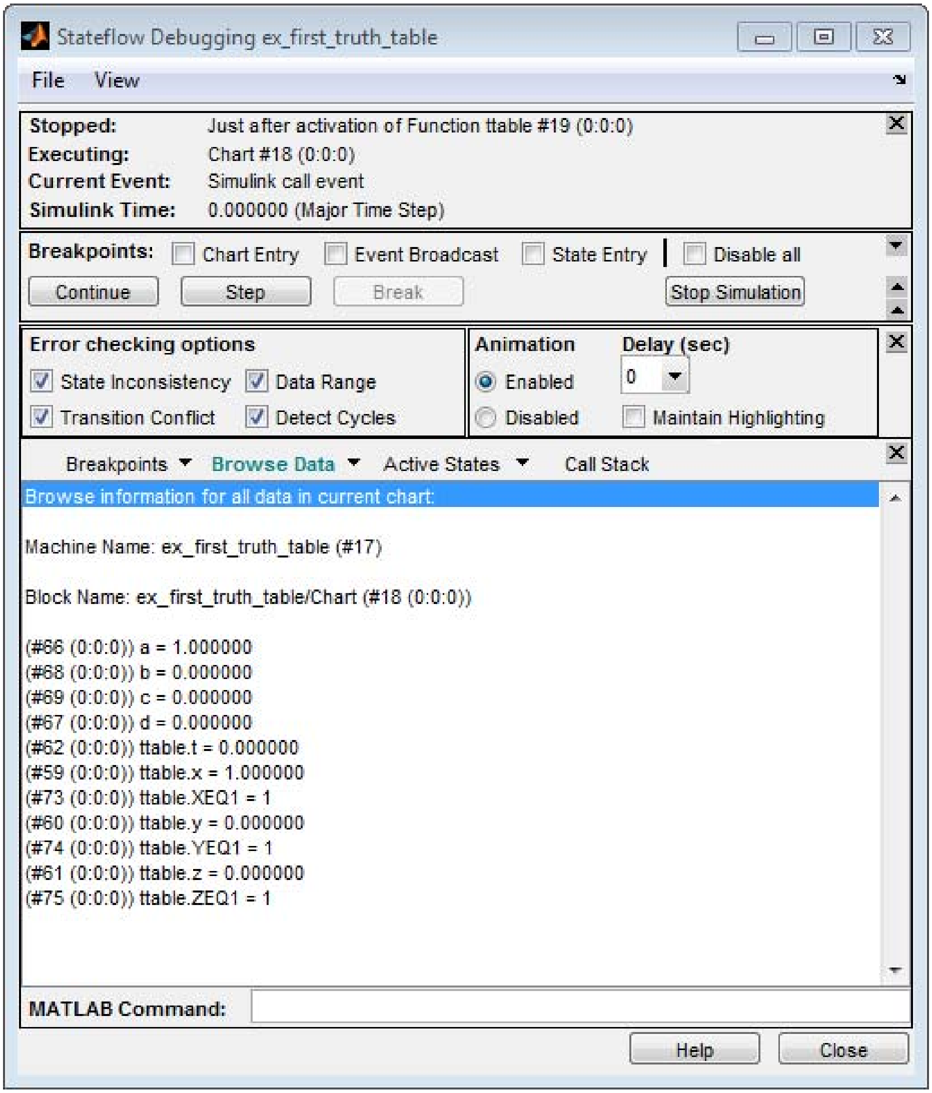
   If no syntax errors exist in the truth table, simulation of your model begins.

3. Wait until the breakpoint for the call to the truth table occurs.

When this breakpoint occurs, the Truth Table Editor appears and the **Start** button in the Stateflow Debugging window changes to **Continue**.

4. In the Stateflow Debugging window, from the **Browse Data** pull-down, select `All Data (Current Chart)`.

An updated display appears in the bottom pane.



Stopped: Just after activation of Function ttable #19 (0:0:0)
Executing: Chart #18 (0:0:0)
Current Event: Simulink call event
Simulink Time: 0.000000 (Major Time Step)

Breakpoints: ☐ Chart Entry ☐ Event Broadcast ☐ State Entry ☐ Disable all

Error checking options
☑ State Inconsistency ☑ Data Range
☑ Transition Conflict ☑ Detect Cycles

Animation
● Enabled
○ Disabled

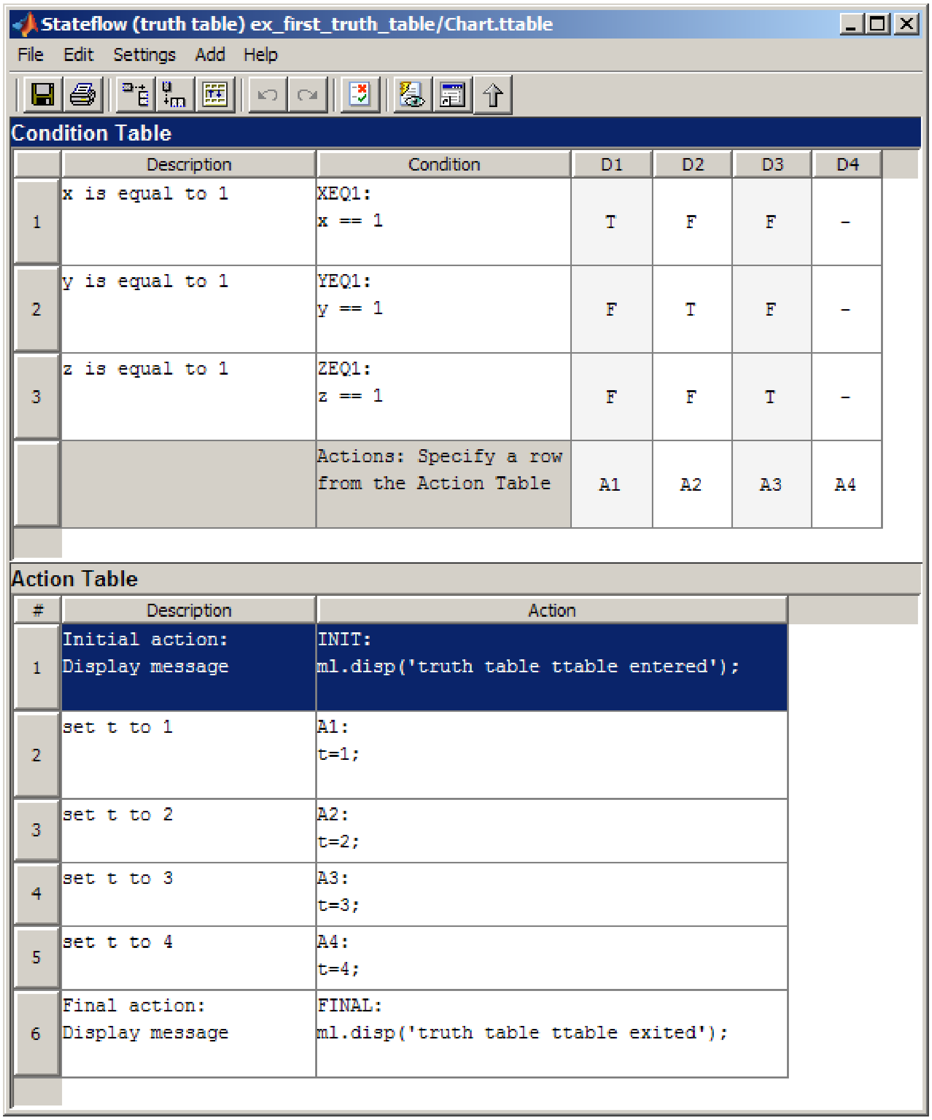
Delay (sec) 0
☐ Maintain Highlighting

```
Browse information for all data in current chart:

Machine Name: ex_first_truth_table (#17)

Block Name: ex_first_truth_table/Chart (#18 (0:0:0))

(#66 (0:0:0)) a = 1.000000
(#68 (0:0:0)) b = 0.000000
(#69 (0:0:0)) c = 0.000000
(#67 (0:0:0)) d = 0.000000
(#62 (0:0:0)) ttable.t = 0.000000
(#59 (0:0:0)) ttable.x = 1.000000
(#73 (0:0:0)) ttable.XEQ1 = 1
(#60 (0:0:0)) ttable.y = 0.000000
(#74 (0:0:0)) ttable.YEQ1 = 1
(#61 (0:0:0)) ttable.z = 0.000000
(#75 (0:0:0)) ttable.ZEQ1 = 1
```

You can use this display to monitor Stateflow data during simulation.

5 In the Stateflow Debugging window, click **Step** four times to advance simulation through the call to the truth table.

The `INIT` action of the truth table highlights prior to execution.



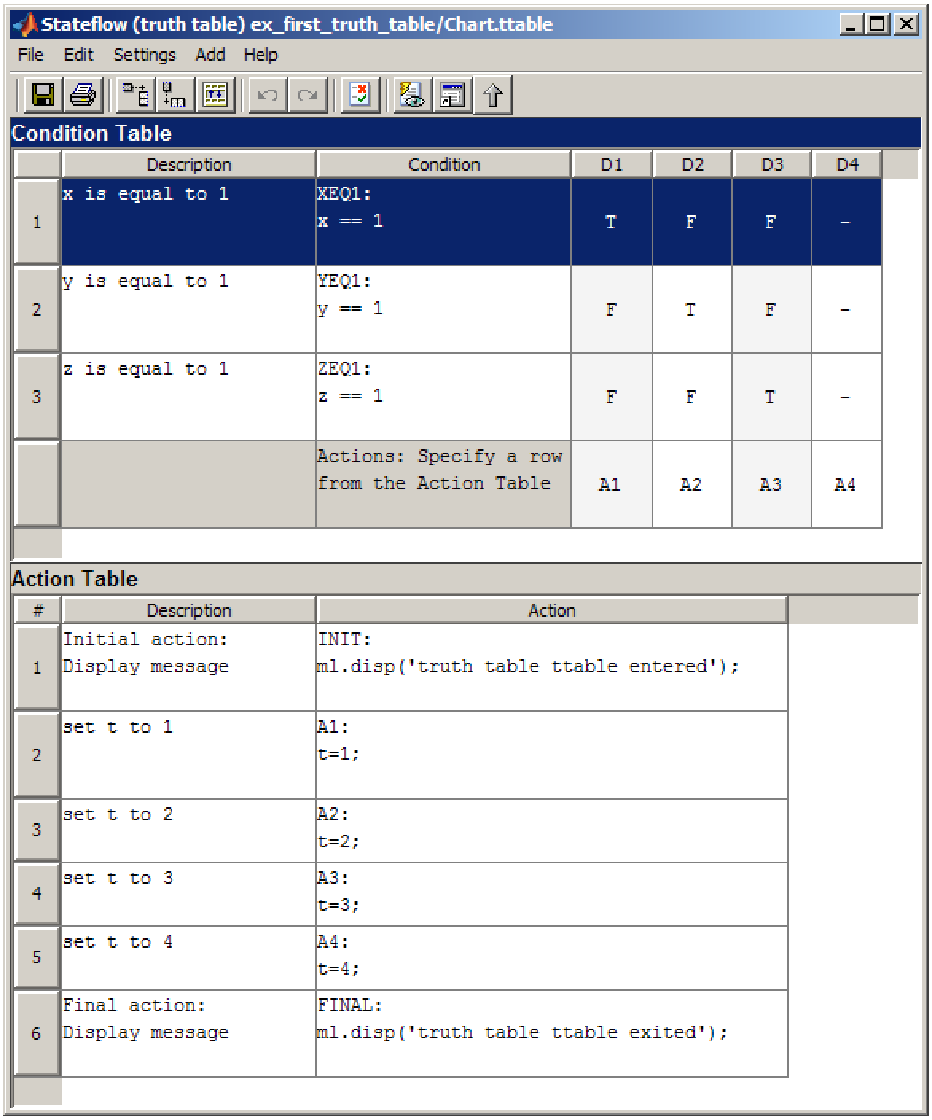
Condition Table

	Description	Condition	D1	D2	D3	D4
1	x is equal to 1	XEQ1: x == 1	T	F	F	-
2	y is equal to 1	YEQ1: y == 1	F	T	F	-
3	z is equal to 1	ZEQ1: z == 1	F	F	T	-
		Actions: Specify a row from the Action Table	A1	A2	A3	A4

Action Table

#	Description	Action
1	Initial action: Display message	INIT: ml.disp('truth table ttable entered');
2	set t to 1	A1: t=1;
3	set t to 2	A2: t=2;
4	set t to 3	A3: t=3;
5	set t to 4	A4: t=4;
6	Final action: Display message	FINAL: ml.disp('truth table ttable exited');

6. Click **Step** twice to execute the `INIT` action and advance truth table execution to the first condition.



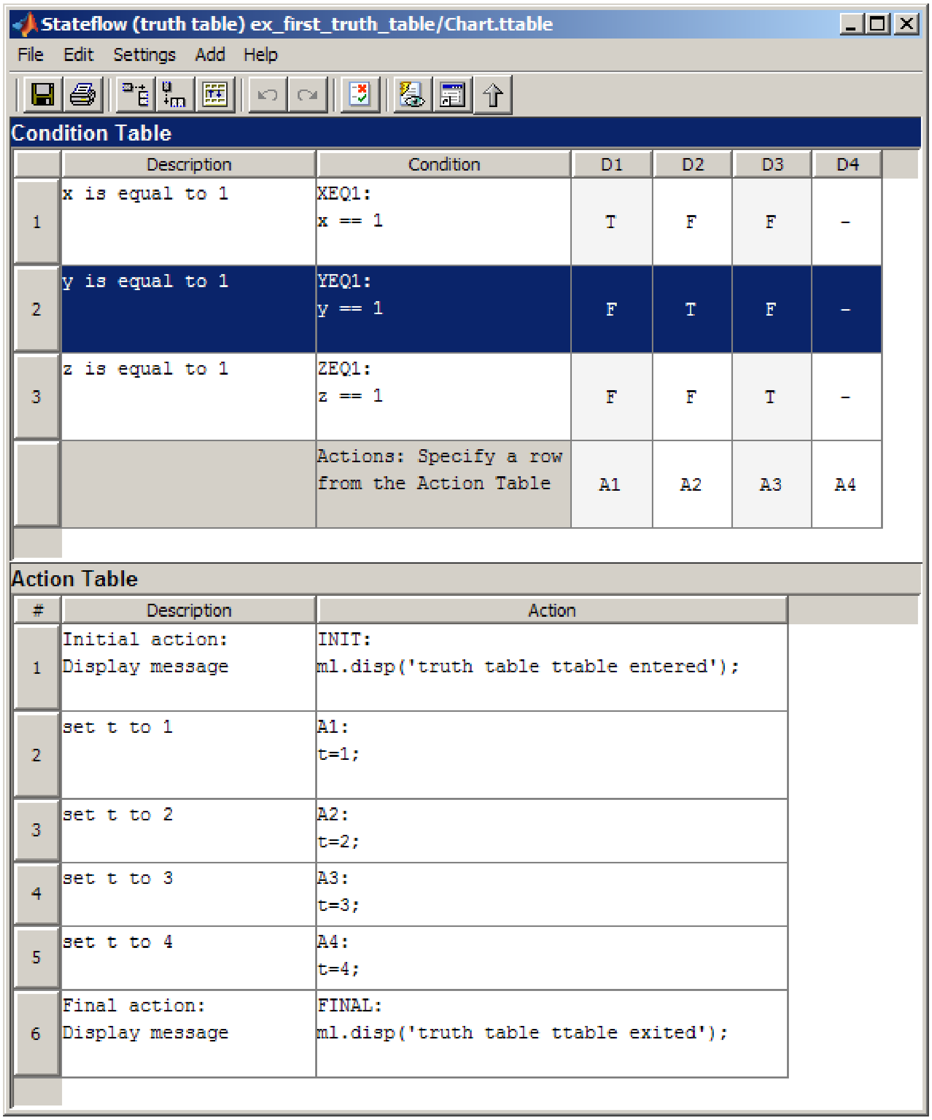
Condition Table

	Description	Condition	D1	D2	D3	D4
1	x is equal to 1	XEQ1: x == 1	T	F	F	-
2	y is equal to 1	YEQ1: y == 1	F	T	F	-
3	z is equal to 1	ZEQ1: z == 1	F	F	T	-
		Actions: Specify a row from the Action Table	A1	A2	A3	A4

Action Table

#	Description	Action
1	Initial action: Display message	INIT: ml.disp('truth table ttable entered');
2	set t to 1	A1: t=1;
3	set t to 2	A2: t=2;
4	set t to 3	A3: t=3;
5	set t to 4	A4: t=4;
6	Final action: Display message	FINAL: ml.disp('truth table ttable exited');

7 Click **Step** twice to evaluate the first condition and advance truth table execution to the second condition.



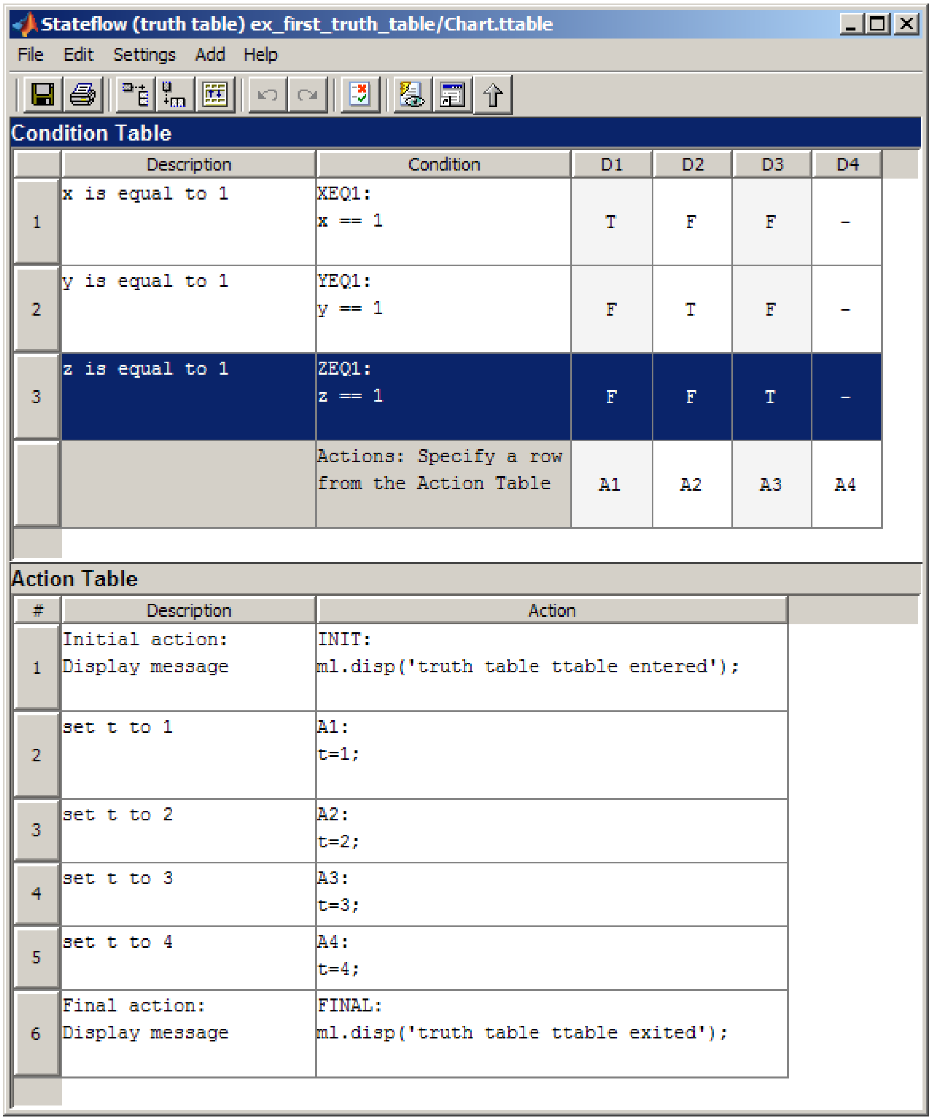
Condition Table

	Description	Condition	D1	D2	D3	D4
1	x is equal to 1	XEQ1: x == 1	T	F	F	-
2	y is equal to 1	YEQ1: y == 1	F	T	F	-
3	z is equal to 1	ZEQ1: z == 1	F	F	T	-
		Actions: Specify a row from the Action Table	A1	A2	A3	A4

Action Table

#	Description	Action
1	Initial action: Display message	INIT: ml.disp('truth table ttable entered');
2	set t to 1	A1: t=1;
3	set t to 2	A2: t=2;
4	set t to 3	A3: t=3;
5	set t to 4	A4: t=4;
6	Final action: Display message	FINAL: ml.disp('truth table ttable exited');

8 Click **Step** twice to evaluate the second condition and advance truth table execution to the third condition.



Stateflow (truth table) ex_first_truth_table/Chart.ttable

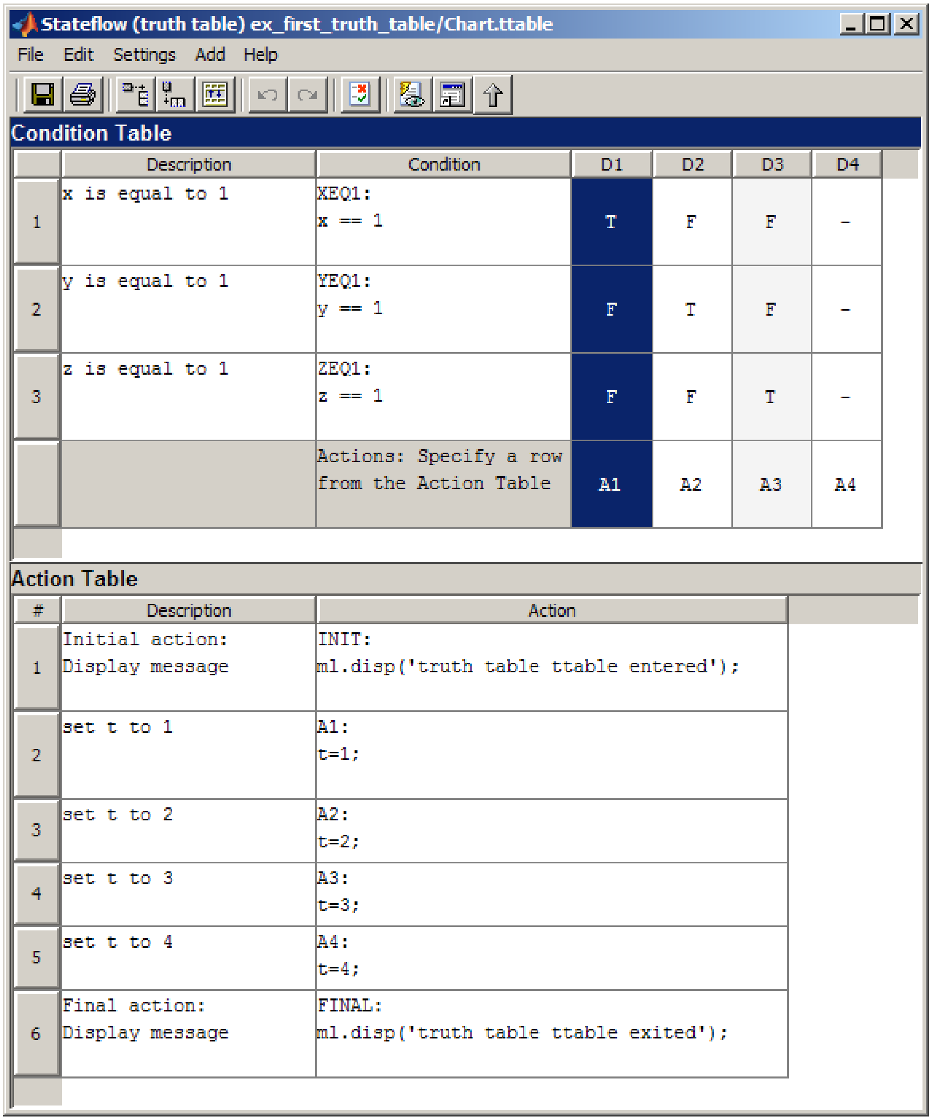
Condition Table

	Description	Condition	D1	D2	D3	D4
1	x is equal to 1	XEQ1: x == 1	T	F	F	-
2	y is equal to 1	YEQ1: y == 1	F	T	F	-
3	z is equal to 1	ZEQ1: z == 1	F	F	T	-
		Actions: Specify a row from the Action Table	A1	A2	A3	A4

Action Table

#	Description	Action
1	Initial action: Display message	INIT: ml.disp('truth table ttable entered');
2	set t to 1	A1: t=1;
3	set t to 2	A2: t=2;
4	set t to 3	A3: t=3;
5	set t to 4	A4: t=4;
6	Final action: Display message	FINAL: ml.disp('truth table ttable exited');

9 Click **Step** twice to evaluate the third condition and advance truth table execution to the first decision.



Condition Table

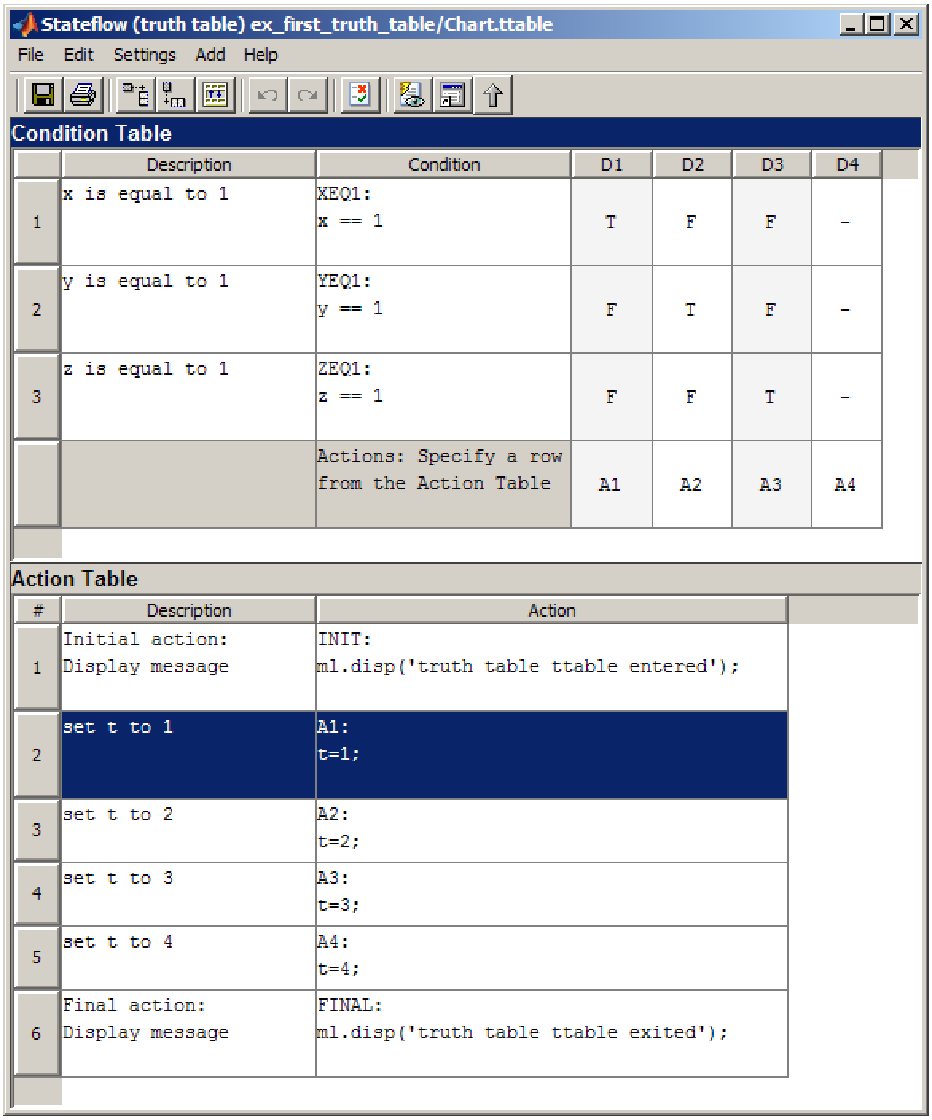
	Description	Condition	D1	D2	D3	D4
1	x is equal to 1	XEQ1: x == 1	T	F	F	-
2	y is equal to 1	YEQ1: y == 1	F	T	F	-
3	z is equal to 1	ZEQ1: z == 1	F	F	T	-
		Actions: Specify a row from the Action Table	A1	A2	A3	A4

Action Table

#	Description	Action
1	Initial action: Display message	INIT: ml.disp('truth table ttable entered');
2	set t to 1	A1: t=1;
3	set t to 2	A2: t=2;
4	set t to 3	A3: t=3;
5	set t to 4	A4: t=4;
6	Final action: Display message	FINAL: ml.disp('truth table ttable exited');

10 Click **Step** twice.

Because the first decision is true, truth table execution advances to its action A1.



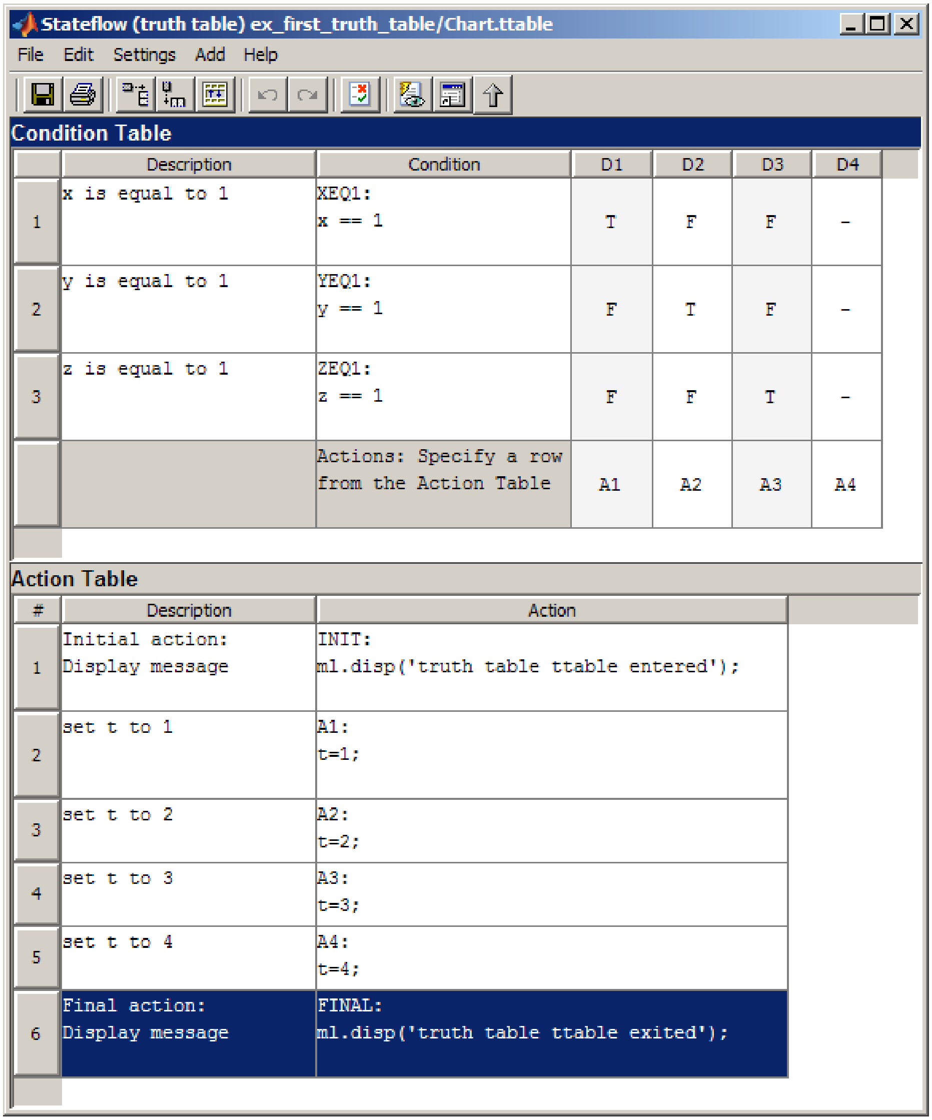
Condition Table

	Description	Condition	D1	D2	D3	D4
1	x is equal to 1	XEQ1: x == 1	T	F	F	-
2	y is equal to 1	YEQ1: y == 1	F	T	F	-
3	z is equal to 1	ZEQ1: z == 1	F	F	T	-
		Actions: Specify a row from the Action Table	A1	A2	A3	A4

Action Table

#	Description	Action
1	Initial action: Display message	INIT: ml.disp('truth table ttable entered');
2	set t to 1	A1: t=1;
3	set t to 2	A2: t=2;
4	set t to 3	A3: t=3;
5	set t to 4	A4: t=4;
6	Final action: Display message	FINAL: ml.disp('truth table ttable exited');

11 Click **Step** eight times to execute action `A1` and advance to the `FINAL` action.



Condition Table

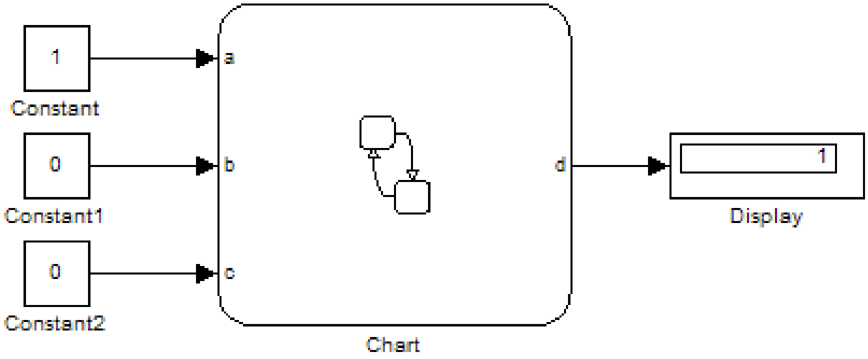
	Description	Condition	D1	D2	D3	D4
1	x is equal to 1	XEQ1: x == 1	T	F	F	-
2	y is equal to 1	YEQ1: y == 1	F	T	F	-
3	z is equal to 1	ZEQ1: z == 1	F	F	T	-
		Actions: Specify a row from the Action Table	A1	A2	A3	A4

Action Table

#	Description	Action
1	Initial action: Display message	INIT: ml.disp('truth table ttable entered');
2	set t to 1	A1: t=1;
3	set t to 2	A2: t=2;
4	set t to 3	A3: t=3;
5	set t to 4	A4: t=4;
6	Final action: Display message	FINAL: ml.disp('truth table ttable exited');

12 In the Stateflow Debugging window, click **Stop Simulation**.

This step executes the final action and exits the truth table. The Display block in the model displays the number 1.



Using MATLAB Debugging Tools

MATLAB truth tables generate content as 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 that MATLAB truth tables generate the same way you debug a MATLAB function.

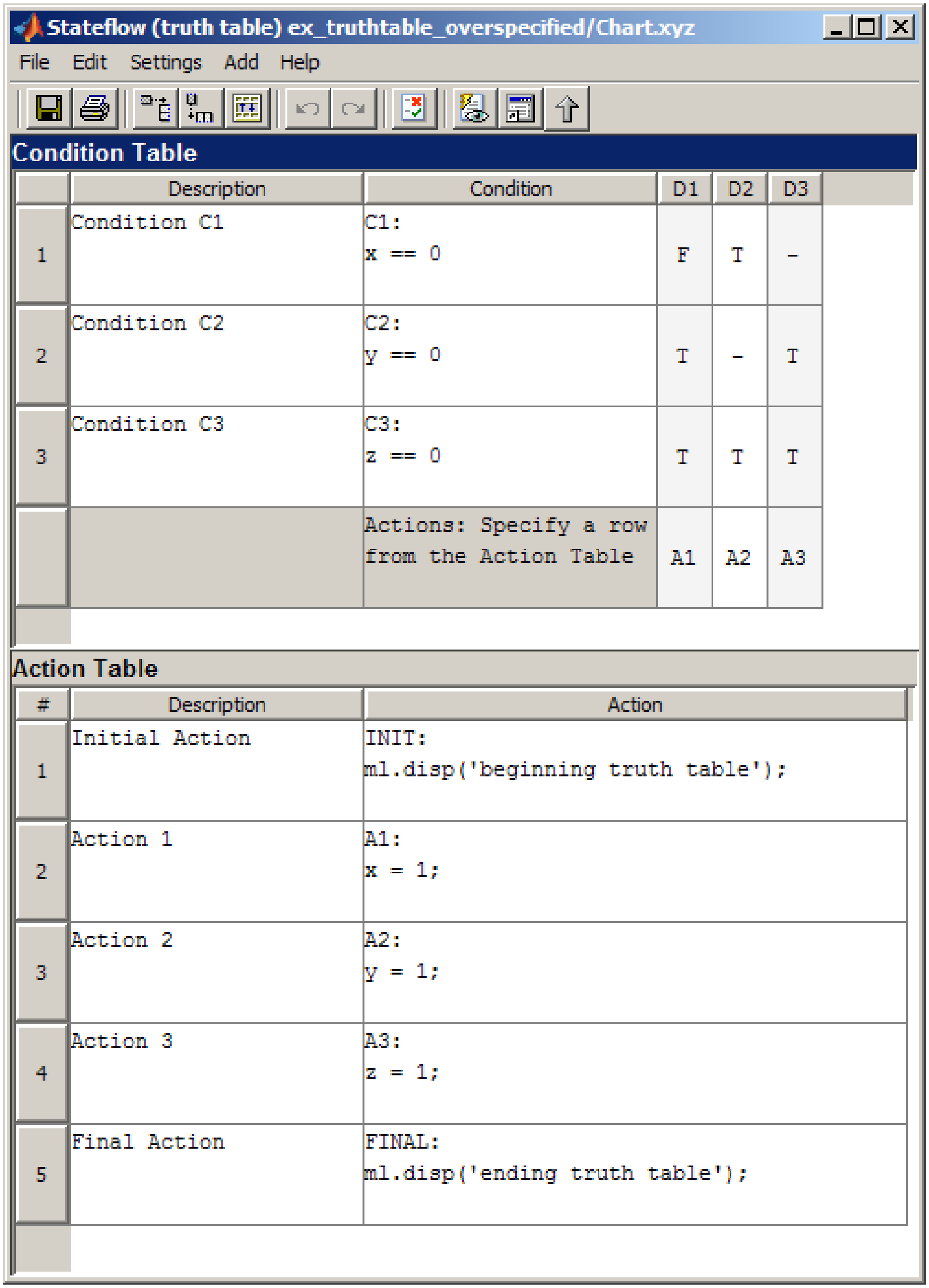
For more information about how to generate content for truth tables, see "How Stateflow Software Generates Content for Truth Tables" on page 22-73.

Correcting Overspecified and Underspecified Truth Tables

In this section...

Example of an Overspecified Truth Table

An overspecified truth table contains at least one decision that never executes because a previous decision specifies it in the **Condition Table**. The following example shows the **Condition Table** of an overspecified truth table.



Condition Table

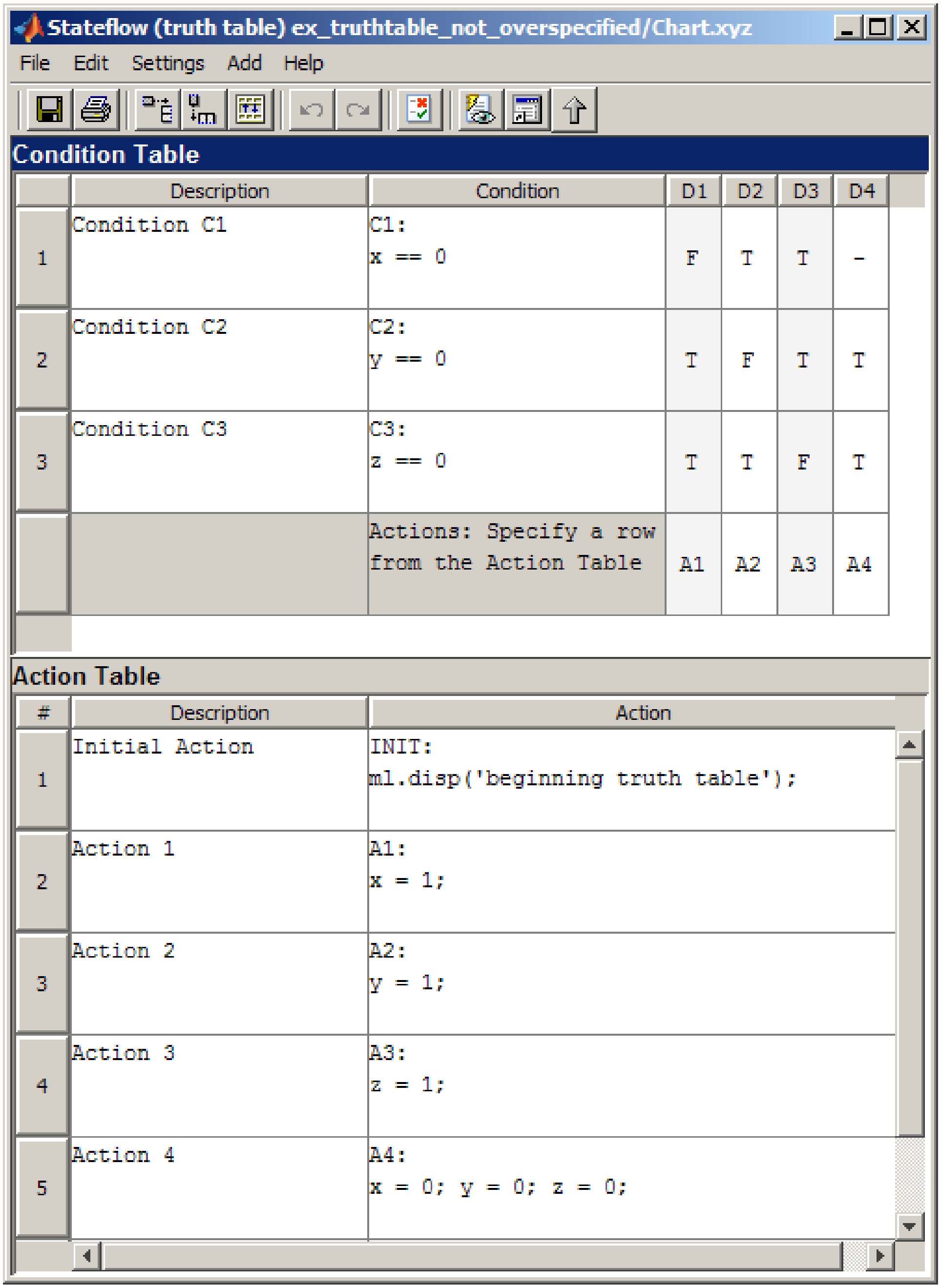
	Description	Condition	D1	D2	D3
1	Condition C1	C1: x == 0	F	T	-
2	Condition C2	C2: y == 0	T	-	T
3	Condition C3	C3: z == 0	T	T	T
		Actions: Specify a row from the Action Table	A1	A2	A3

Action Table

#	Description	Action
1	Initial Action	INIT: ml.disp('beginning truth table');
2	Action 1	A1: x = 1;
3	Action 2	A2: y = 1;
4	Action 3	A3: z = 1;
5	Final Action	FINAL: ml.disp('ending truth table');

The decision in column **D3** (`-TT`) specifies the decisions `FTT` and `TTT`. These decisions are duplicates of decisions **D1** (`FTT`) and **D2** (`TTT` and `TFT`). Therefore, column **D3** is an overspecification.

The following example shows the **Condition Table** of a truth table that appears to be overspecified, but is not.



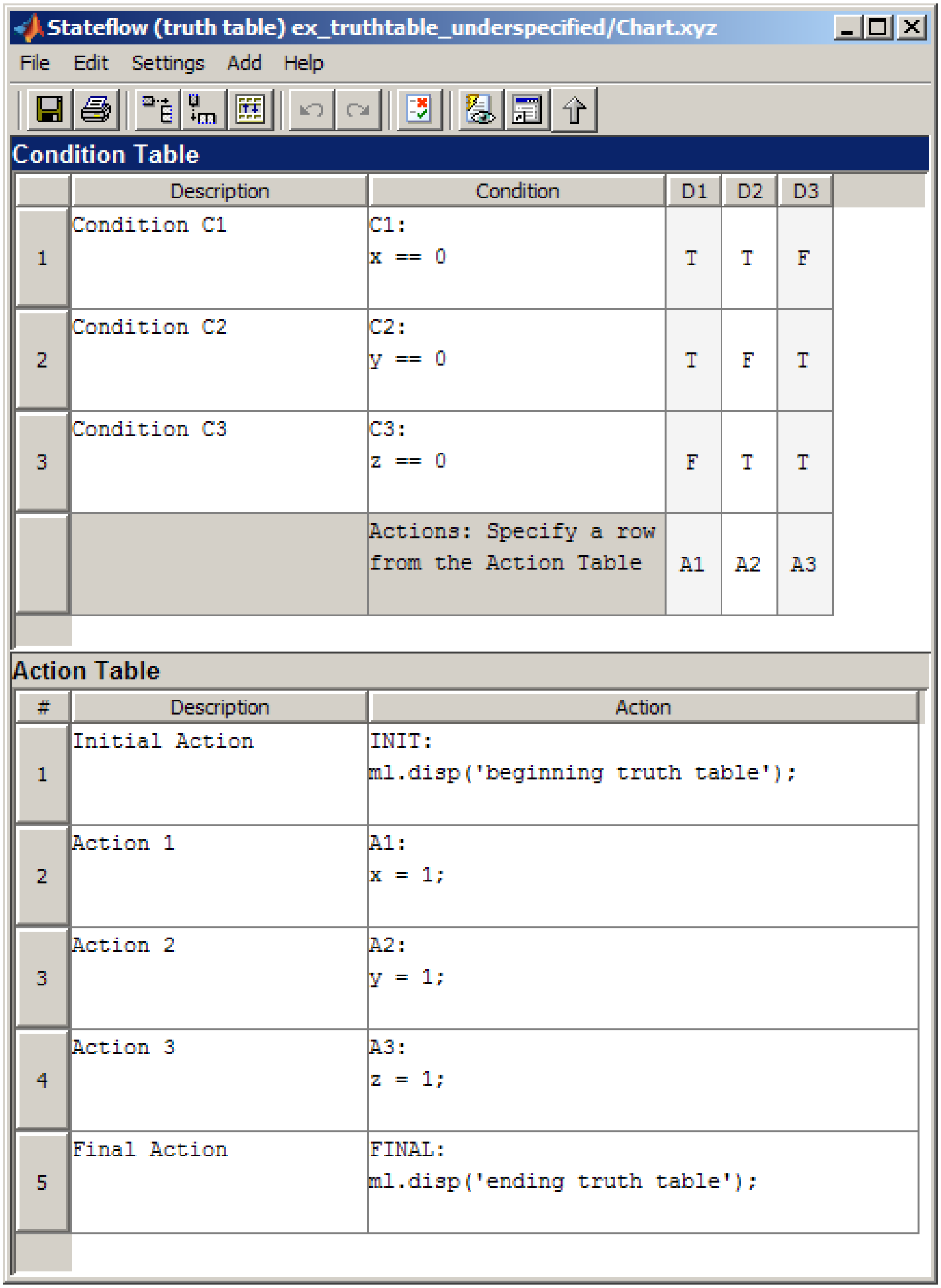
	Description	Condition	D1	D2	D3	D4
1	Condition C1	C1: x == 0	F	T	T	-
2	Condition C2	C2: y == 0	T	F	T	T
3	Condition C3	C3: z == 0	T	T	F	T
		Actions: Specify a row from the Action Table	A1	A2	A3	A4

#	Description	Action
1	Initial Action	INIT: ml.disp('beginning truth table');
2	Action 1	A1: x = 1;
3	Action 2	A2: y = 1;
4	Action 3	A3: z = 1;
5	Action 4	A4: x = 0; y = 0; z = 0;

In this case, the decision **D4** specifies two decisions (`TTT` and `FTT`). `FTT` also appears in decision **D1**, but `TTT` is not a duplicate. Therefore, this **Condition Table** is not overspecified.

Example of an Underspecified Truth Table

An underspecified truth table lacks one or more possible decisions that require an action to avoid undefined behavior. The following example shows the **Condition Table** of an underspecified truth table.



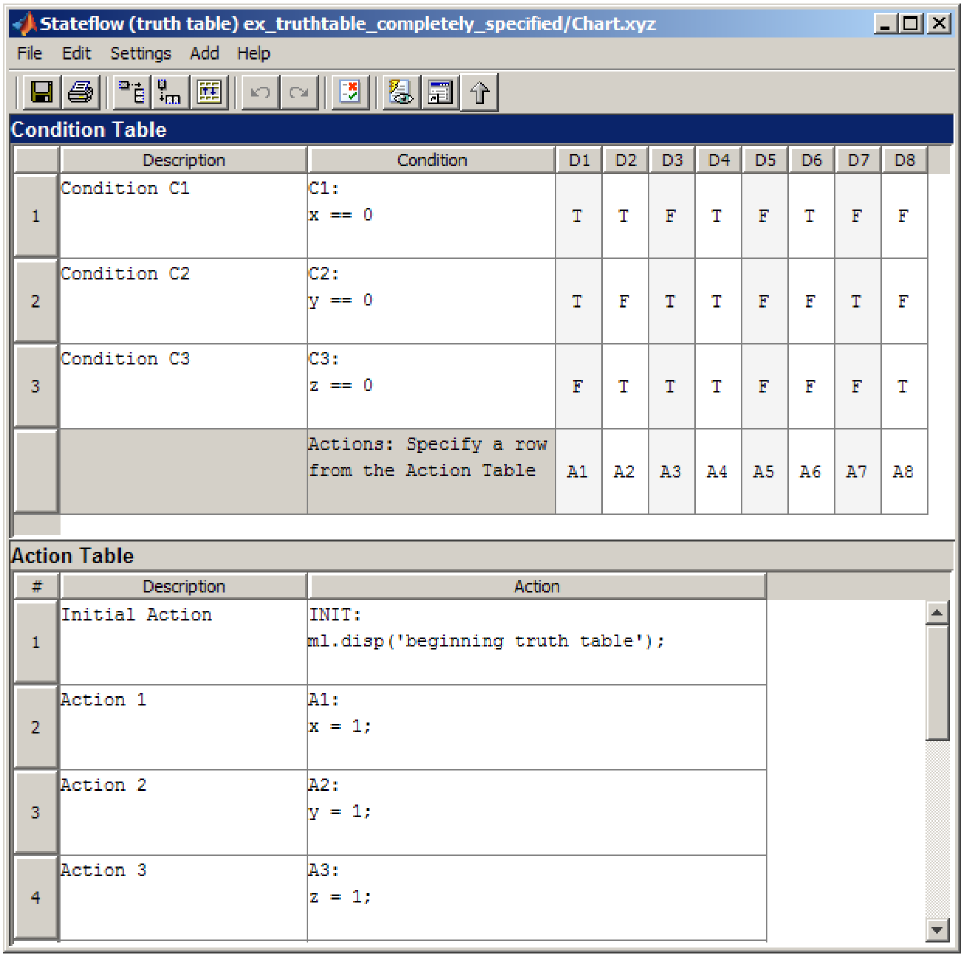
Condition Table

	Description	Condition	D1	D2	D3
1	Condition C1	C1: x == 0	T	T	F
2	Condition C2	C2: y == 0	T	F	T
3	Condition C3	C3: z == 0	F	T	T
		Actions: Specify a row from the Action Table	A1	A2	A3

Action Table

#	Description	Action
1	Initial Action	INIT: ml.disp('beginning truth table');
2	Action 1	A1: x = 1;
3	Action 2	A2: y = 1;
4	Action 3	A3: z = 1;
5	Final Action	FINAL: ml.disp('ending truth table');

Complete coverage of the conditions in the preceding truth table requires a **Condition Table** with every possible decision:



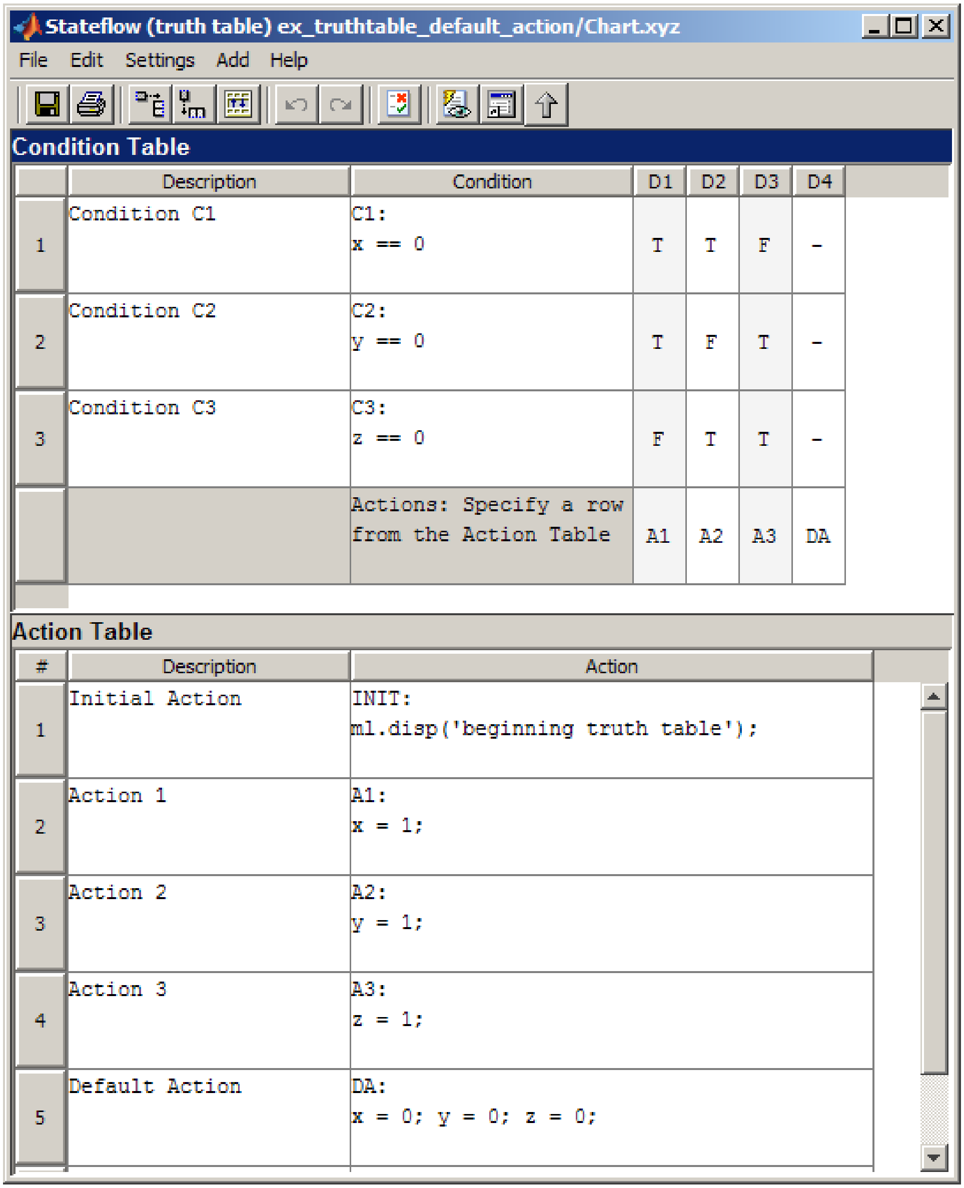
Condition Table

	Description	Condition	D1	D2	D3	D4	D5	D6	D7	D8
1	Condition C1	C1: x == 0	T	T	F	T	F	T	F	F
2	Condition C2	C2: y == 0	T	F	T	T	F	F	T	F
3	Condition C3	C3: z == 0	F	T	T	T	F	F	F	T
		Actions: Specify a row from the Action Table	A1	A2	A3	A4	A5	A6	A7	A8

Action Table

#	Description	Action
1	Initial Action	INIT: ml.disp('beginning truth table');
2	Action 1	A1: x = 1;
3	Action 2	A2: y = 1;
4	Action 3	A3: z = 1;

A possible workaround is to specify an action for all other possible decisions through a default decision, named `DA`:



Condition Table

	Description	Condition	D1	D2	D3	D4
1	Condition C1	C1: x == 0	T	T	F	-
2	Condition C2	C2: y == 0	T	F	T	-
3	Condition C3	C3: z == 0	F	T	T	-
		Actions: Specify a row from the Action Table	A1	A2	A3	DA

Action Table

#	Description	Action
1	Initial Action	INIT: ml.disp('beginning truth table');
2	Action 1	A1: x = 1;
3	Action 2	A2: y = 1;
4	Action 3	A3: z = 1;
5	Default Action	DA: x = 0; y = 0; z = 0;

The last decision column is the default decision for the truth table. The default decision covers any remaining decisions not tested in the preceding decision columns. See "The Default Decision Column" on page 22-31 for an example and more complete description of the default decision column for a **Condition Table**.

How Stateflow Software Generates Content for 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
MATLAB	MATLAB code

Viewing Generated Content

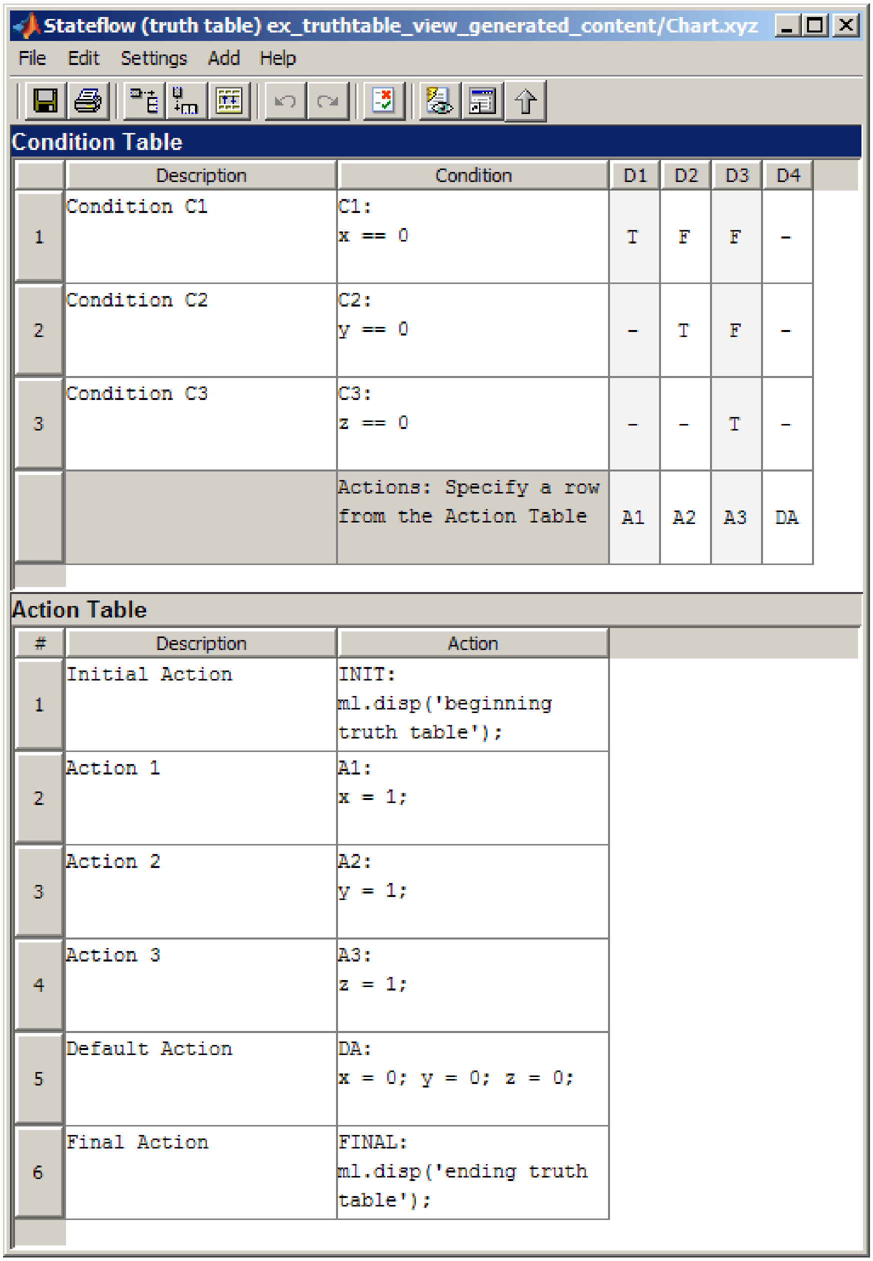
You generate content for a truth table when you simulate your model. Content regenerates whenever a truth table changes. To view the generated content of a truth table, follow these steps:

1. Simulate the model that contains the truth table.
2. Double-click the truth table to open its editor.
3. Click the **View Generated Content** button:

How Stateflow Software Generates Graphical Functions for Truth Tables

This section describes how Stateflow software translates the logic of a Stateflow Classic truth table into a graphical function.

In the following example, a Stateflow Classic truth table has three conditions, four decisions and actions, and initial and final actions.



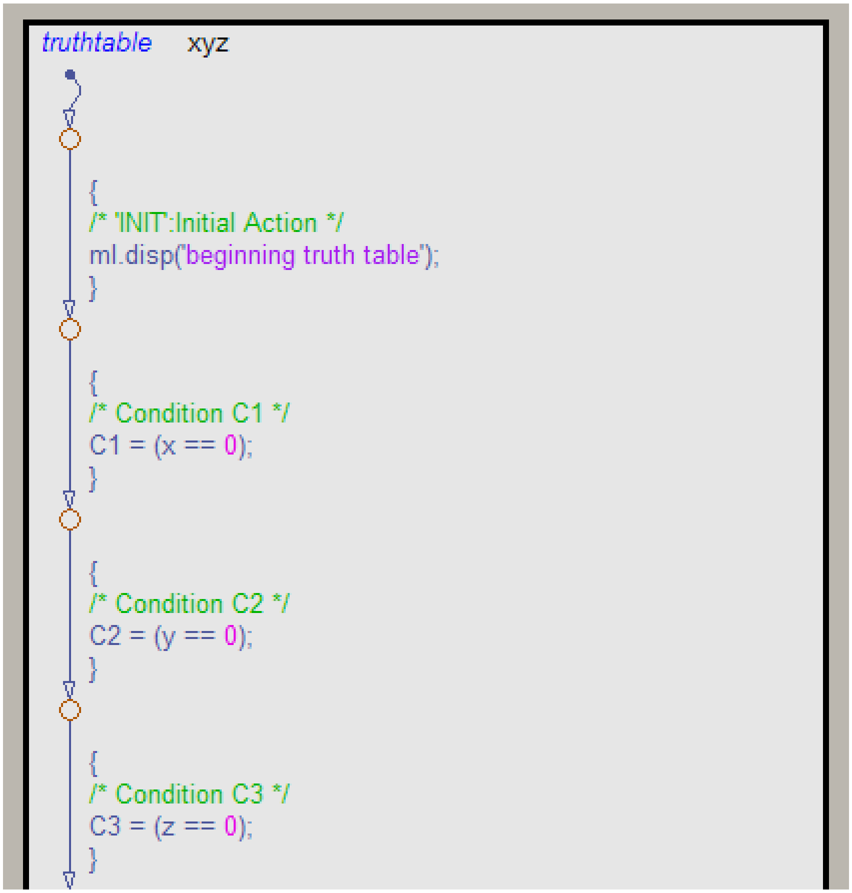
Condition Table

	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 looks like this:



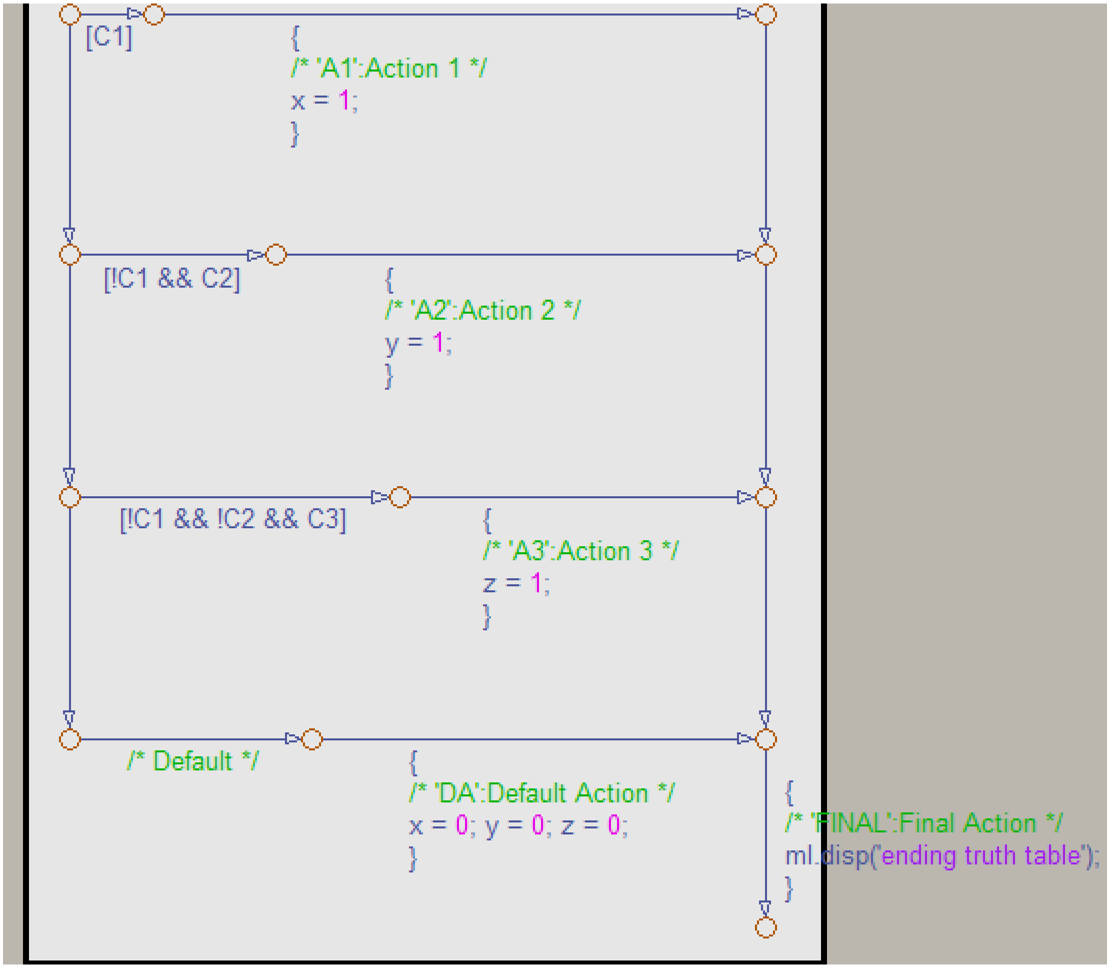
```
{
/* 'INIT':Initial Action */
ml.disp('beginning truth table');
}

{
/* Condition C1 */
C1 = (x == 0);
}

{
/* Condition C2 */
C2 = (y == 0);
}

{
/* Condition C3 */
C3 = (z == 0);
}
```

The top half of the flow graph executes as follows:

- Performs initial actions
- Evaluates the conditions and stores the results in temporary data variables

  The temporary data for storing conditions is based on the labels that you enter for the conditions. If you do not specify the labels, temporary data variables appear.

The bottom half of the flow graph looks like this:



```
{
/* 'A1':Action 1 */
x = 1;
}
```

```
{
/* 'A2':Action 2 */
y = 1;
}
```

```
{
/* 'A3':Action 3 */
z = 1;
}
```

```
{
/* 'DA':Default Action */
x = 0; y = 0; z = 0;
}
```

```
{
/* 'FINAL':Final Action */
ml.disp('ending truth table');
}
```

In the bottom half of the flow graph, the stored values for conditions determine which decision is true and what action to perform. Each decision appears as a fork from a connective junction with one of two possible paths:

- A transition segment with a decision followed by a segment with the consequent action

The action appears as a condition action that leads to the `FINAL` action and termination of the flow 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 decision match occurs, the action for that decision executes as a condition action of its transition segment. After the action executes, 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 means that no data dependencies are possible between different decisions.

How Stateflow Software Generates MATLAB Code for Truth Tables

Stateflow software generates the content of MATLAB truth tables as MATLAB code that represents each action as a *nested* function inside the main truth table function.

Nested functions offer these advantages over subfunctions:

- Nested functions are independent of one another. Variables are local to each function and not subject to naming conflicts.
- Nested functions can access all data from the main truth table function.

The generated content appears in the function editor, which provides tools for simulation and debugging.

Here is the generated content for the MATLAB truth table described in “Programming Actions in MATLAB Action Language” on page 22-37:

- Main truth table function

  ```
  function t = ttable(x,y,z)

  % Initialize condition vars to logical scalar
  ```

```
XEQ1 = false;
YEQ1 = false;
ZEQ1 = false;

% Condition #1, "XEQ1"
% x is equal to 1
XEQ1 = logical(x == 1);

% Condition #2, "YEQ1"
% y is equal to 1
YEQ1 = logical(y == 1);

% Condition #3, "ZEQ1"
% z is equal to 1
ZEQ1 = logical(z == 1);

if (XEQ1 && ~YEQ1 && ~ZEQ1) % D1
    A1();
elseif (~XEQ1 && YEQ1 && ~ZEQ1) % D2
    A2();
elseif (~XEQ1 && ~YEQ1 && ZEQ1) % D3
    A3();
else % Default
    A4();
end
```

- Action `A1`

```
function A1()
% Action #1, "A1"
% Maintain a counter and a circular vector of length 6.
% Every time this action is called,
% output t takes the next value of the vector.

persistent values counter;
cycle = 6;

if isempty(counter)
  % Initialize counter to be zero
  counter = 0;
```

```
else
  % Otherwise, increment counter
  counter = counter + 1;
end

if isempty(values)
  % Values is a vector of 1 to cycle
  values = zeros(1, cycle);
  for i = 1:cycle
    values(i) = i;
  end

  % For debugging purposes, call the MATLAB
  %  function "plot" to show values
  plot(values);
end

% Output t takes the next value in values vector
t = values( mod(counter, cycle) + 1);
```

- Actions `A2`, `A3`, and `A4`

```
function A2()
% Action #2, "A2"
% set t to 2

t=2;

%==================================
function A3()
% Action #3, "A3"
% set t to 3

t=3;

%==================================
function A4()
% Action #4, "A4"
% set t to 4
```

```
t=4;
```

Truth Table Editor Operations

In this section…

Adding or Modifying Stateflow Data

	Edit Data/Ports opens the Model Explorer so that you can add or modify Stateflow data.

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 22-50.

Viewing Generated Content

	View Generated Content displays the code generated for the truth table. Stateflow Classic truth tables generate graphical functions. MATLAB truth tables generate MATLAB code. For details, see "How Stateflow Software Generates Content for Truth Tables" on page 22-73.

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 22-82.

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 `-`. After 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 left or right:

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...	Perform this action...
Select a cell for editing	Click the cell
Select text in a cell	Click and drag your pointer over the text
Select a row	Click the header for the row
Select a decision column in the **Condition Table**	Click the header for the column
Deselect a selected cell, row, or column	Press **Esc** or click another table, cell, row, or column

Undoing and Redoing Edit Operations

↶	Click the **Undo** button or press **Ctrl+Z (Command+Z)** to undo the effects of the preceding operation.
↷	Click the **Redo** button or press **Ctrl+Y (Command+Y)** to redo the most recently undone operation.

Viewing the Stateflow Chart for the Truth Table

⇧	**Go to Editor** displays the current truth table function in its native chart.

23

Using MATLAB Functions in Stateflow Charts

What Is a MATLAB Function in a Stateflow Chart?

A MATLAB function in a Stateflow chart is a graphical element you use to write algorithms that are easier to perform by calling built-in MATLAB functions. Typical applications include:

- Matrix-oriented calculations
- Data analysis and visualization

Why Use a MATLAB Function in a Stateflow Chart?

This type of function is useful for coding algorithms that are more easily expressed in the textual MATLAB language than in the graphical Stateflow action language. This function also provides optimizations for generating efficient, production-quality C code for embedded applications. For more information, see the Code Generation from MATLAB documentation.

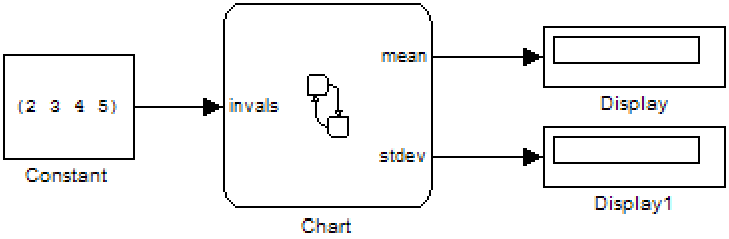
Where to Use a MATLAB Function

A MATLAB function can reside anywhere in a chart, state, or subchart. The location of a function determines its scope, that is, the set of states and transitions that can call the function. Follow these guidelines:

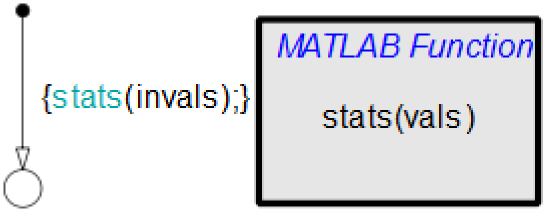
- If you want to call the function only within one state or subchart and its substates, put your MATLAB function in that state or subchart. That function overrides any other functions of the same name in the parents and ancestors of that state or subchart.
- If you want to call the function anywhere in that chart, put your MATLAB function at the chart level.

Example of a MATLAB Function in a Stateflow Chart

The following model contains a Stateflow chart with a MATLAB function.



The chart contains the following logic:



The function contains the following code:

```
function stats(vals)
%#codegen

% calculates a statistical mean and standard deviation
% for the values in vals.

len = length(vals);
mean = avg(vals, len);
stdev = sqrt(sum(((vals-avg(vals,len)).^2))/len);
coder.extrinsic('plot');
plot(vals,'-+');

function mean = avg(array,size)
mean = sum(array)/size;
```

You will build a similar model in “Building a Model with a MATLAB Function in a Chart” on page 23-8.

Note in this example that the MATLAB function can call any of these types of functions:

- Subfunctions

  Subfunctions are defined in the body of the MATLAB function. In this example, `avg` is a subfunction. See “Calling Subfunctions” in the Code Generation from MATLAB documentation.

- Stateflow functions

  Graphical, truth table, and other MATLAB functions can be called from a MATLAB function in a chart.

- MATLAB toolbox functions that support code generation

  Toolbox functions for code generation are a subset of the functions that you can call in the MATLAB workspace. These functions generate C code for building targets that conform to the memory and data type requirements of embedded environments. In this example, `length`, `sqrt`, and `sum` are examples of toolbox functions for code generation. See “Calling Supported Toolbox Functions” in the Code Generation from MATLAB documentation.

- MATLAB toolbox functions that do not support code generation

  You can also call *extrinsic* functions on the MATLAB path that do not generate code. These functions execute only in the MATLAB workspace during simulation of the model. See “Calling MATLAB Functions” in the Code Generation from MATLAB documentation.

- Simulink Design Verifier functions

  Simulink Design Verifier software provides MATLAB functions for property proving and test generation.

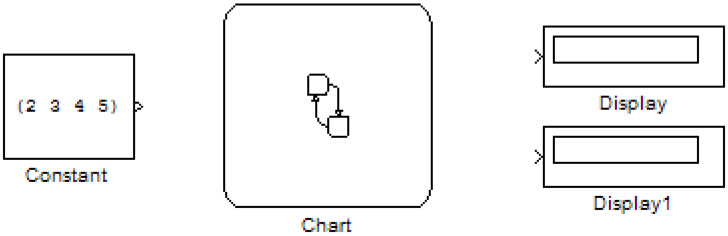
  - `sldv.prove`
  - `sldv.assume`
  - `sldv.test`
  - `sldv.condition`

Building a Model with a MATLAB Function in a Chart

This section explains how to create a model with a Stateflow chart that calls two 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 model with the following blocks:



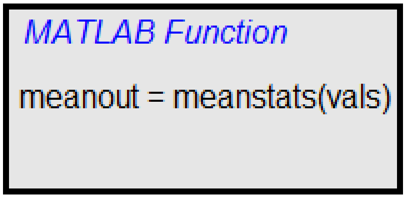
2 Save the model as `call_stats_function_stateflow`.

3 In the model, double-click the Chart block.

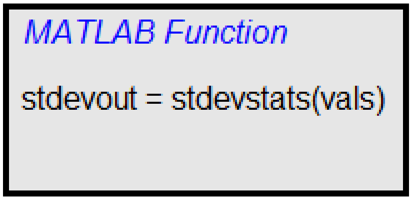
4 Drag two MATLAB functions into the empty chart using this icon from the toolbar:

A text field with a flashing cursor appears in the middle of each MATLAB function.

5 Label each function as shown:



meanout = meanstats(vals)



stdevout = stdevstats(vals)

You must label a MATLAB function with its signature. Use the following syntax:

`[`*`return_val1`*`, `*`return_val2`*`,...] = `*`function_name`*`(`*`arg1`*`, `*`arg2`*`,...)`

You can specify multiple return values and multiple input arguments, as shown in the syntax. Each return value and input argument can be a scalar, vector, or matrix of values.

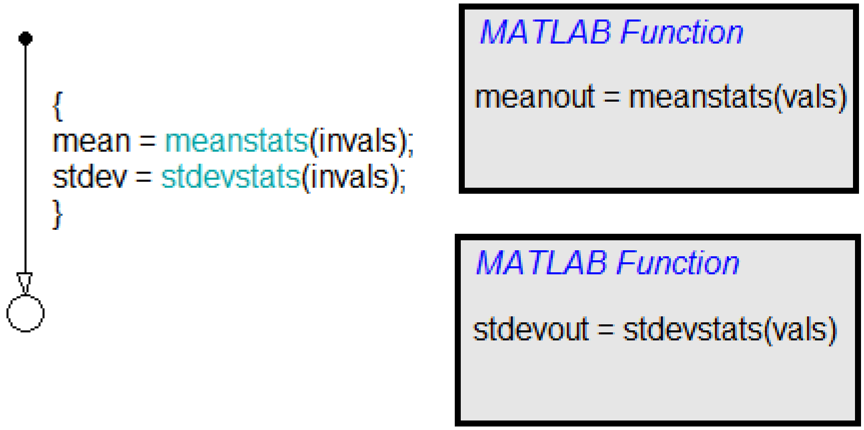
Note For MATLAB functions with only one return value, you can omit the brackets in the signature label.

---

6 In the chart, draw a default transition into a terminating junction with this condition action:

```
{
mean = meanstats(invals);
stdev = stdevstats(invals);
}
```

The chart should look something like this:



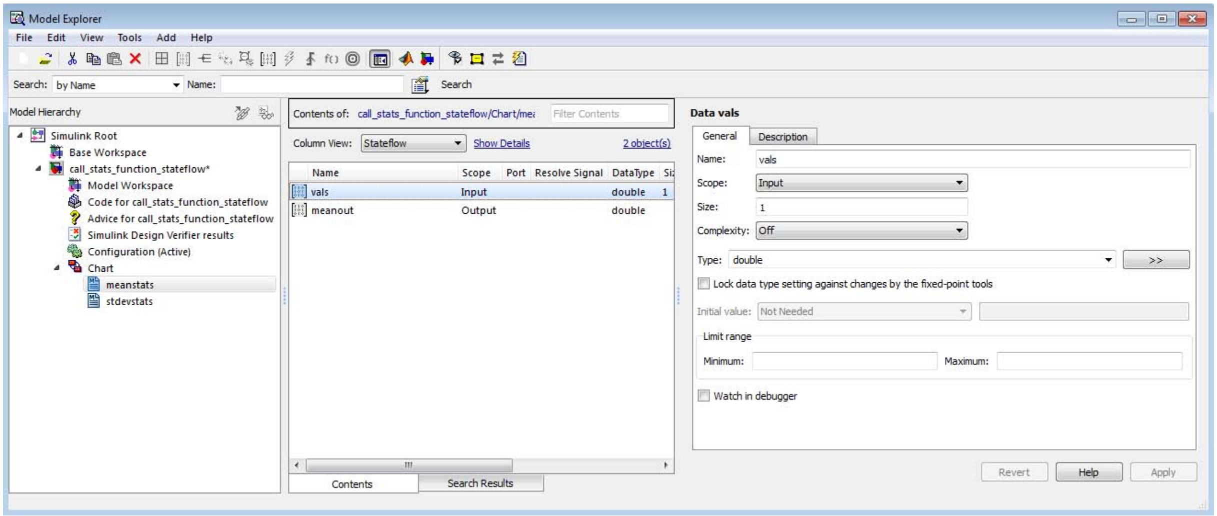
```
{
mean = meanstats(invals);
stdev = stdevstats(invals);
}
```

Tip If the formal arguments of a function signature are scalars, verify that inputs and outputs of function calls follow the rules of scalar expansion. For more information, see "How Scalar Expansion Works for Functions" on page 13-6.

---

7 In the chart, double-click the function `meanstats` to edit its function body in the editor.

8 In the function editor, select **Tools > Model Explorer** to open the Model Explorer.



Name	Scope	Port	Resolve Signal	DataType	Si
vals	Input			double	1
meanout	Output			double	

Name: vals
Scope: Input
Size: 1
Complexity: Off
Type: double
☐ Lock data type setting against changes by the fixed-point tools
Initial value: Not Needed
☐ Watch in debugger

The function `meanstats` is highlighted in the **Model Hierarchy** pane. The **Contents** pane displays the input argument `vals` and output argument `meanout`. Both are scalars of type `double` by default.

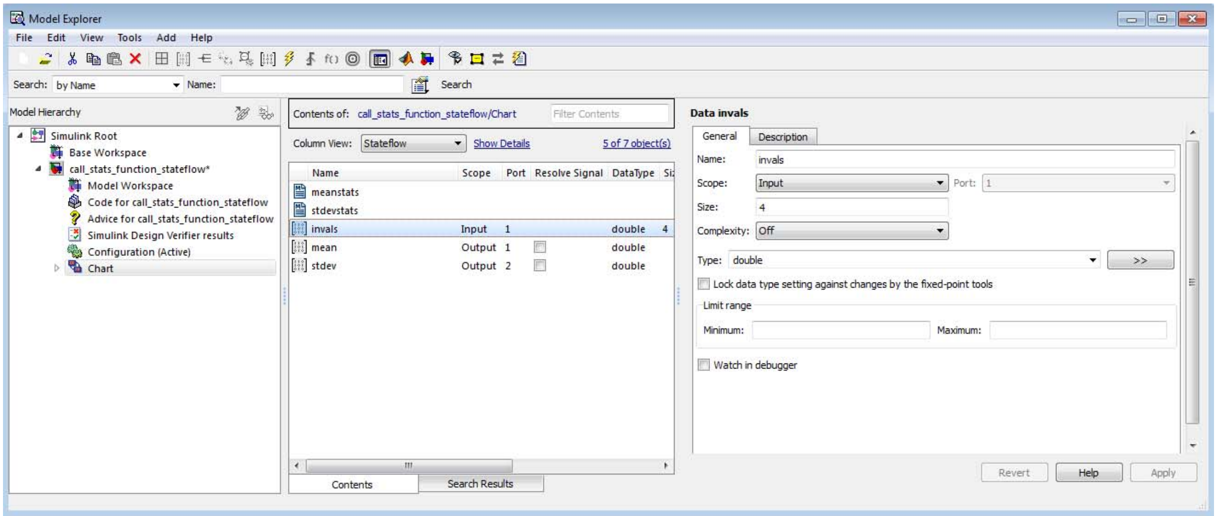
9 Double-click the `vals` row under the **Size** column to set the size of `vals` to `4`.

10 Back in the chart, double-click the function `stdevstats` and repeat the previous two steps.

11 Back in the **Model Hierarchy** pane of the Model Explorer, select **Chart** and add the following data:

Name	Scope	Size
`invals`	`Input`	4
`mean`	`Output`	Scalar (no change)
`stdev`	`Output`	Scalar (no change)

You should now see the following data in the Model Explorer.



Contents of: call_stats_function_stateflow/Chart

Column View: Stateflow

5 of 7 object(s)

Name	Scope	Port	Resolve Signal	DataType	Si
meanstats					
stdevstats					
invals	Input	1		double	4
mean	Output	1	☐	double	
stdev	Output	2	☐	double	

Data invals

General | Description

Name: invals

Scope: Input Port: 1

Size: 4

Complexity: Off

Type: double

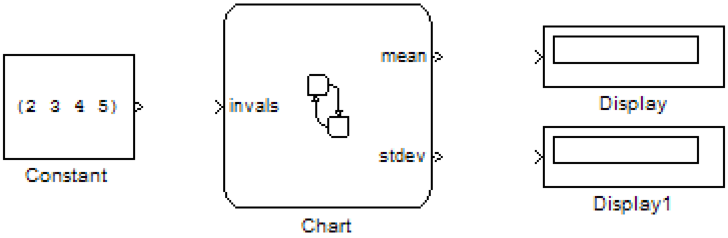
☐ Lock data type setting against changes by the fixed-point tools

Limit range

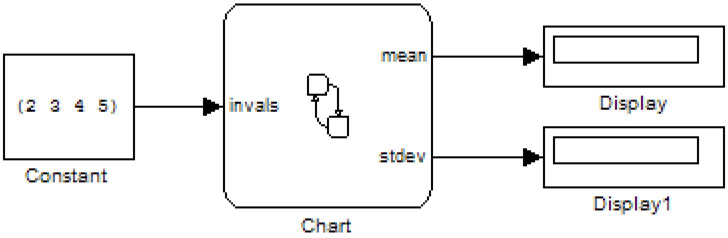
Minimum: Maximum:

☐ Watch in debugger

After you add the data `invals`, `mean`, and `stdev` to the chart, the corresponding input and output ports appear on the Stateflow block in the model.



12 Connect the Constant and Display blocks to the ports of the Chart block and save the model.



The section “Debugging a MATLAB Function in a Chart” on page 23-18 shows you how to program the functions `meanstats` and `stdevstats`.

Programming a MATLAB Function in a Chart

To program the functions `meanstats` and `stdevstats` that you created in "Building a Model with a MATLAB Function in a Chart" on page 23-8, follow these steps:

1. Open the chart in the model `call_stats_function_stateflow`.

2. In the chart, open the function `meanstats`.

   The function editor appears with the header:

   ```
   function meanout = meanstats(vals)
   ```

   This header is taken from the function label in the chart. You can edit the header directly in the editor, and your changes appear in the chart after you close the editor.

3. On the line after the function header, enter:

   ```
   %#codegen
   ```

   The `%#codegen` compilation directive helps detect compile-time violations of syntax and semantics in MATLAB functions supported for code generation. To learn more about detecting compile-time errors, see "Adding the Compilation Directive %#codegen" in the Code Generation from MATLAB documentation.

4. Enter a line space and this comment:

   ```
   % Calculates the statistical mean for vals
   ```

5. Add the line:

   ```
   len = length(vals);
   ```

   The function `length` is an example of a built-in MATLAB function that is supported for code generation. You can call this function directly to return the vector length of its argument `vals`. When you build a simulation target, the function length is implemented with generated C code. Functions supported for code generation appear in "Functions Supported for Code Generation" in the Code Generation from MATLAB documentation.

The variable `len` is an example of implicitly declared local data. It has the same size and type as the value assigned to it — the value returned by the function `length`, a scalar `double`. To learn more about declaring variables, see "Defining MATLAB Variables for C/C++ Code Generation" in the Code Generation from MATLAB documentation.

The MATLAB function treats implicitly declared local data as temporary data, which exists only when the function is called and disappears when the function exits. You can declare local data for a MATLAB function in a chart to be persistent by using the `persistent` construct (see "Defining and Initializing Persistent Variables" in the Code Generation from MATLAB documentation).

6 Enter this line to calculate the value of `meanout`:

```
meanout = avg(vals,len);
```

The function `meanstats` stores the mean of `vals` in the Stateflow data `meanout`. Because these data are defined for the parent Stateflow chart, you can use them directly in the MATLAB function.

Two-dimensional arrays with a single row or column of elements are treated as vectors or matrices in MATLAB functions. For example, in `meanstats`, the argument `vals` is a four-element vector. You can access the fourth element of this vector with the matrix notation `vals(4,1)` or the vector notation `vals(4)`.

The MATLAB function uses the functions `avg` and `sum` to compute the value of `mean`. `sum` is a function supported for code generation. `avg` is a subfunction that you define later. When resolving function names, MATLAB functions in your chart look for subfunctions first, followed by functions supported for code generation.

Note If you call a function that the MATLAB function cannot resolve as a subfunction or a function for code generation, you must declare the function to be extrinsic. For more information, see "Calling MATLAB Functions" in the Code Generation from MATLAB documentation.

7 Now enter this statement:

```
coder.extrinsic('plot');
```

8 Enter this line to plot the input values in `vals` against their vector index.

```
plot(vals,'-+');
```

Recall that you declared `plot` to be an extrinsic function because it is not supported for code generation. When the MATLAB function encounters an extrinsic function, it sends the call to the MATLAB workspace for execution during simulation.

9 Now, define the 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 the MATLAB documentation.

The complete code for the function `meanstats` looks like this:

```
function meanout = meanstats(vals)
%#codegen

% Calculates the statistical mean for vals

len = length(vals);
meanout = avg(vals,len);

coder.extrinsic('plot');
plot(vals,'-+');

function mean = avg(array,size)
mean = sum(array)/size;
```

10 Save the model.

11 Back in the chart, open the function `stdevstats` and add code to compute the standard deviation of the values in `vals`. The complete code should look like this:

```
function stdevout = stdevstats(vals)
%#codegen

% Calculates the standard deviation for vals

len = length(vals);
stdevout = sqrt(sum(((vals-avg(vals,len)).^2))/len);

function mean = avg(array,size)
mean = sum(array)/size;
```

12 Save the model again.

Debugging a MATLAB Function in a Chart

In this section...

Checking MATLAB Functions for Syntax Errors

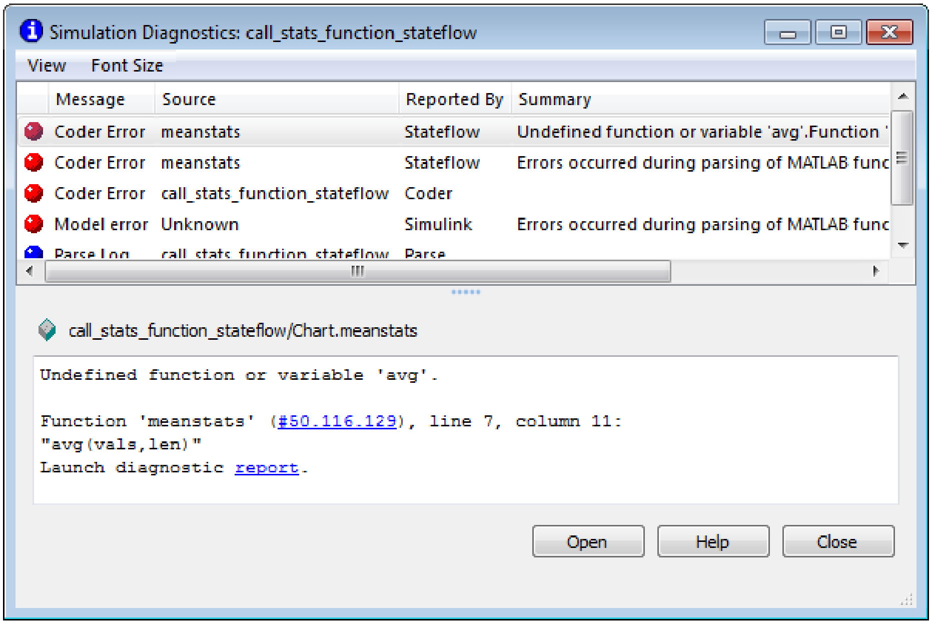
Before you can build a simulation application for a model, you must fix syntax errors. Follow these steps to check the MATLAB function `meanstats` for syntax violations:

1. Open the function `meanstats` inside the chart in the `call_stats_function_stateflow` model that you updated in "Programming a MATLAB Function in a Chart" on page 23-14.

   The editor automatically checks your function code for errors and recommends corrections.

2. In the editor, select **Tools > Build**.

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 following errors appear:



Message	Source	Reported By	Summary
Coder Error	meanstats	Stateflow	Undefined function or variable 'avg'.Function '
Coder Error	meanstats	Stateflow	Errors occurred during parsing of MATLAB func
Coder Error	call_stats_function_stateflow	Coder	
Model error	Unknown	Simulink	Errors occurred during parsing of MATLAB func
Parse Log	call_stats_function_stateflow	Parse	

call_stats_function_stateflow/Chart.meanstats

```
Undefined function or variable 'avg'.

Function 'meanstats' (#50.116.129), line 7, column 11:
"avg(vals,len)"
Launch diagnostic report.
```

Each error message appears with a red button. The selected error message displays diagnostic information in the bottom pane.

3 Click the first error line to display its diagnostic message in the bottom error window.

The diagnostic message provides details of the type of error and a link to the code where the error occurred. The diagnostic message also contains a link to a diagnostic report that provides links to your MATLAB functions and compile-time type information for the variables and expressions in these functions. If your model fails to build, this information simplifies finding sources of error messages and aids understanding of type propagation rules. For more information about this report, see “Working with MATLAB Function Reports” in the Simulink documentation.

4 In the diagnostic message, click the link after the function name `meanstats` to display the offending line of code.

   The offending line appears highlighted in the editor.

5 Correct the error by changing `aug` back to `avg` and recompile. No errors are found and the compile completes successfully.

Run-Time Debugging for MATLAB Functions in Charts

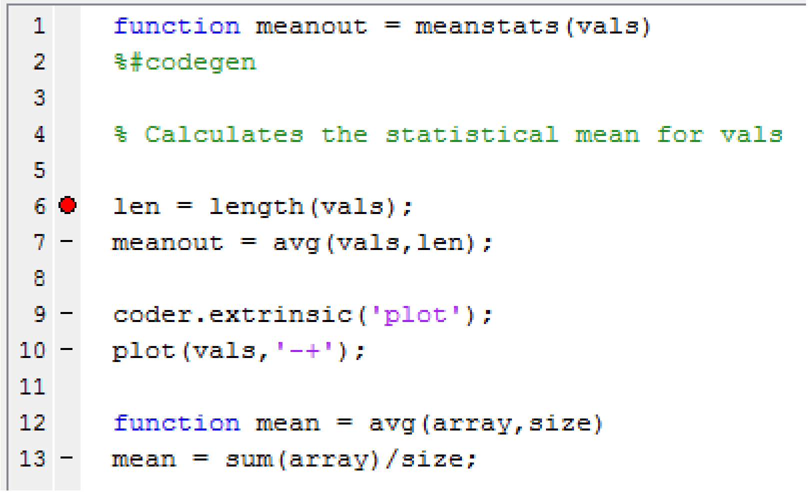
You use simulation to test your MATLAB functions for run-time errors that are not detectable by the Stateflow Debugger. When you simulate your model, your MATLAB functions undergo diagnostic tests for missing or undefined information and possible logical conflicts as described in “Checking MATLAB Functions for Syntax Errors” on page 23-18. If no errors are found, the simulation of your model begins.

Follow these steps to simulate and debug the `meanstats` function during run-time conditions:

1 In the function editor, click the dash (-) character in the left margin of this line:

```
len = length(vals);
```

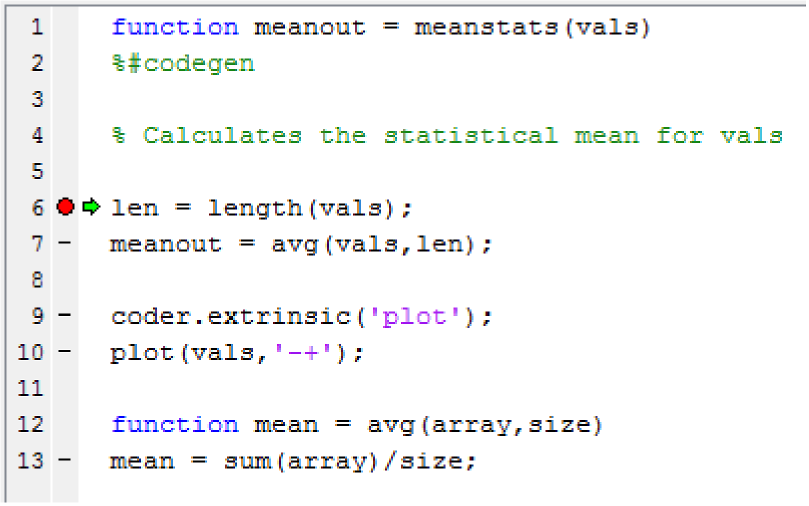
A small red ball appears in the margin of this line, indicating that you have set a breakpoint.



```
function meanout = meanstats(vals)
%#codegen

% Calculates the statistical mean for vals

len = length(vals);
meanout = avg(vals,len);

coder.extrinsic('plot');
plot(vals,'-+');

function mean = avg(array,size)
mean = sum(array)/size;
```

2 Start simulation for the model.

If you get any errors or warnings, make corrections before you try to simulate again. Otherwise, simulation pauses when execution reaches the breakpoint you set. A small green arrow in the left margin indicates this pause.



```
function meanout = meanstats(vals)
%#codegen

% Calculates the statistical mean for vals

len = length(vals);
meanout = avg(vals,len);

coder.extrinsic('plot');
plot(vals,'-+');

function mean = avg(array,size)
mean = sum(array)/size;
```

3 To advance execution to the next line, select **Debug > Step**.

   Notice that this line calls the subfunction `avg`. If you select **Step** here, execution advances past the execution of the subfunction `avg`. To track execution of the lines in the subfunction `avg`, select **Debug > Step In** instead.

4 To advance execution to the first line of the called subfunction `avg`, select **Debug > Step In**.

   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. To continue through the remaining lines of the subfunction and go back to the line after the subfunction call, select **Debug > Step Out**.

5 Select **Step** 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 Select **Step** to return to the function `meanstats`.

Execution advances to the line after the call to the subfunction `avg`.

7 To display the value of the variable `len`, place your pointer over the text `len` in the function editor for at least a second.

The value of `len` appears adjacent to your pointer.

You can display the value for any data in the MATLAB function in this way, no matter where it appears in the function. For example, you can display the values for the vector `vals` by placing your pointer over it as an argument to the function `length`, or as an argument in the function header.

You can also report the values for MATLAB function data in the MATLAB Command Window during simulation. When you reach a breakpoint, the `debug>>` command prompt appears in the MATLAB Command Window (you might have to press **Enter** to see it). At this prompt, you can inspect data defined for the function by entering the name of the data, as shown in this example:

```
debug>> len
len =
     4
debug>>
```

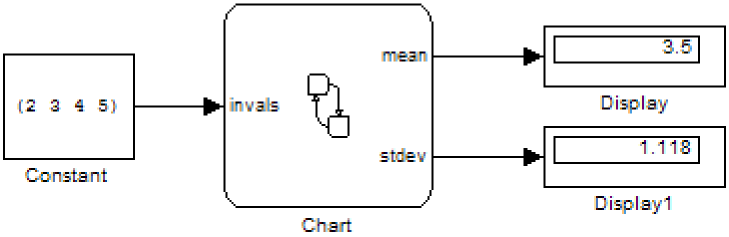
As another debugging alternative, you can display the execution result of a function line by omitting the terminating semicolon. If you do, execution results appear in the MATLAB Command Window during simulation.

8 To leave the function until it is called again and the breakpoint is reached, select **Debug > Continue**.

At any point in a function, you can advance through the execution of the remaining lines of the function with the **Continue** tool. If you are at the end of the function, selecting **Step** does the same thing.

9 Click the breakpoint to remove it and select **Debug > Exit Debug Mode** to complete the simulation.

In the model, the computed values of `mean` and `stdev` now appear in the Display blocks.



Checking for Data Range Violations

During debugging, 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 MATLAB function input or output of interest.

   The Data properties dialog box opens in the **Dialog** pane of the Model Explorer.

2. In the Data properties dialog box, click the **General** tab and enter a limit range, as described in "Limit range properties" on page 8-14.

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 26-2.

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

Working with Structures and Bus Signals in MATLAB Functions

In this section...

About Structures in MATLAB Functions

MATLAB functions support MATLAB structures. You can create structures in top-level MATLAB functions in Stateflow charts to interface with Simulink bus signals at input and output ports. Simulink buses appear inside the MATLAB function as structures; structure outputs from the MATLAB function appear as buses.

You can also create structures as local and persistent variables in top-level functions and subfunctions of MATLAB functions.

Defining Structures in MATLAB Functions

This section describes how to define structures in MATLAB functions.

- "Rules for Defining Structures in MATLAB Functions" on page 23-26
- "Defining Structure Inputs and Outputs to Interface with Bus Signals" on page 23-27
- "Defining Local and Persistent Structure Variables" on page 23-28

Rules for Defining Structures in MATLAB Functions

Follow these rules when defining structures for MATLAB functions in Stateflow charts:

- For each structure input or output in a MATLAB function, you must define a `Simulink.Bus` object in the base workspace to specify its type to the Simulink signal.
- MATLAB structures cannot inherit their type from Simulink signals.

- MATLAB functions support nonvirtual buses only (see “Virtual and Nonvirtual Buses” in the Simulink documentation).
- Structures cannot have scopes defined as **Constant**.

Defining Structure Inputs and Outputs to Interface with Bus Signals

When you create structure inputs in MATLAB functions, the function determines the type, size, and complexity of the structure from the Simulink input signal. When you create structure outputs, you must define their type, size, and complexity in the MATLAB function.

You can connect MATLAB structure inputs and outputs to any Simulink bus signal, including:

- Simulink blocks that output bus signals — such as Bus Creator blocks
- Simulink blocks that accept bus signals as input — such as Bus Selector and Gain blocks
- S-Function blocks
- Other MATLAB functions

To define structure inputs and outputs for MATLAB functions in Stateflow charts, follow these steps:

1. Create a Simulink bus object in the base workspace to specify the properties of the structure you will create in the MATLAB function.

   For information about how to create Simulink bus objects, see `Simulink.Bus` in the Simulink Reference.

2. Open the Model Explorer and follow these steps:
   - a In the **Model Hierarchy** pane, select the MATLAB function in your chart.
   - b Add a data object, as described in “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 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, open the Data Type Assistant. Then, click the **Edit** button to open the Simulink Bus Editor (see "Using the Bus Editor" in the Simulink documentation).

e Click **Apply**.

3 If your structure is an output (has scope of `Output`), define the output implicitly in the MATLAB function to have the same type, size, and complexity as its `Simulink.Bus` object. For details, see "Code Generation for MATLAB Structures" in the Code Generation from MATLAB documentation.

Defining Local and Persistent Structure Variables

You can define structures as local or persistent variables inside MATLAB functions (see "Structure Operations Allowed for Code Generation" in the Code Generation from MATLAB documentation).

Working with Enumerated Data in MATLAB Functions

Define and use enumerated data with MATLAB functions in Stateflow charts the same way as in MATLAB Function blocks in a model. For more information, see "Using Enumerated Data in MATLAB Function Blocks" in the Simulink documentation.

To learn how to define and use enumerated data in Stateflow charts, see Chapter 15, "Using Enumerated Data in Stateflow Charts".

Working with Variable-Size Data in MATLAB Functions

Declare and use variable-size matrices and arrays with MATLAB functions in Stateflow charts the same way as in MATLAB Function blocks in a model. For more information, see "Using Variable-Size Data in MATLAB Function Blocks" in the Simulink documentation.

To learn how to declare variable-size data at the chart level, see Chapter 14, "Using Variable-Size Data in Stateflow Charts".

Enhancing Readability of Generated Code for MATLAB Functions

You can enhance the readability of generated code for MATLAB functions in Stateflow charts the same way as in MATLAB Function blocks in a model. For more information, see "Enhancing Readability of Generated Code for MATLAB Function Blocks" in the Simulink documentation.

To learn how to enhance readability of generated code for flow graphs in Stateflow charts, see "Enhancing Readability of Generated Code for Flow Graphs" on page 5-32.

24

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 the Simulink documentation)
- Scheduling execution of multiple controllers

Differences Between Simulink Functions and Function-Call Subsystems

In a Stateflow chart, a Simulink function behaves like a function-call subsystem block of a Simulink model. (See "Function-Call Subsystems" in the Simulink documentation.) 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

Why 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

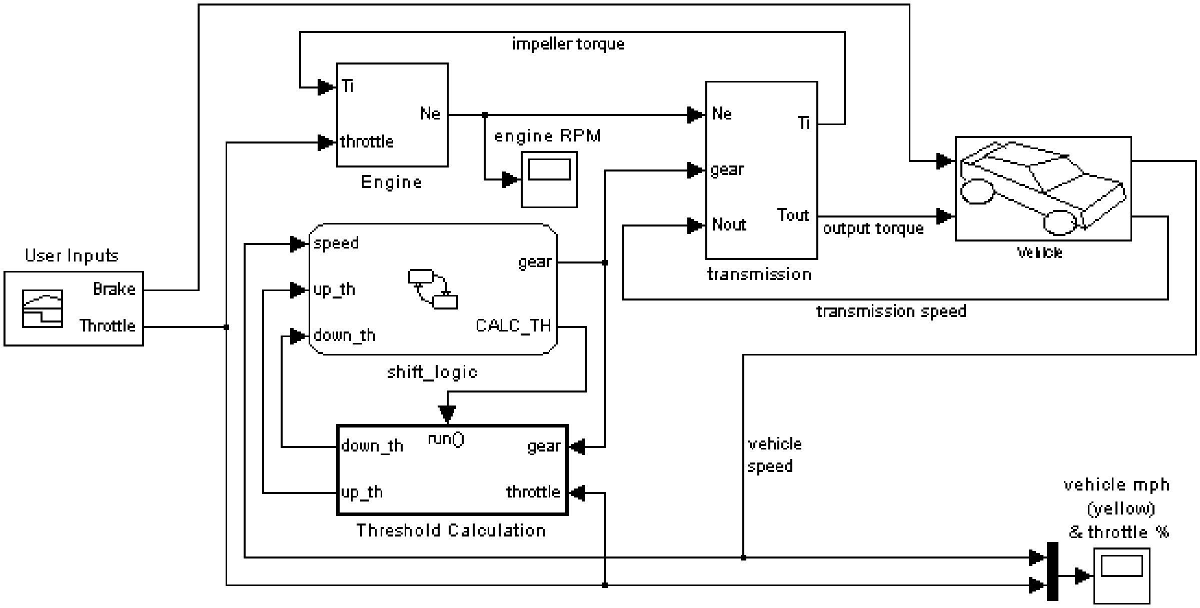
For details about each modeling method, see "Benefits of Using a Simulink Function to Access Simulink Blocks" on page 24-5 and "Benefits of Using a Simulink Function to Schedule Execution of Multiple Controllers" on page 24-7.

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.

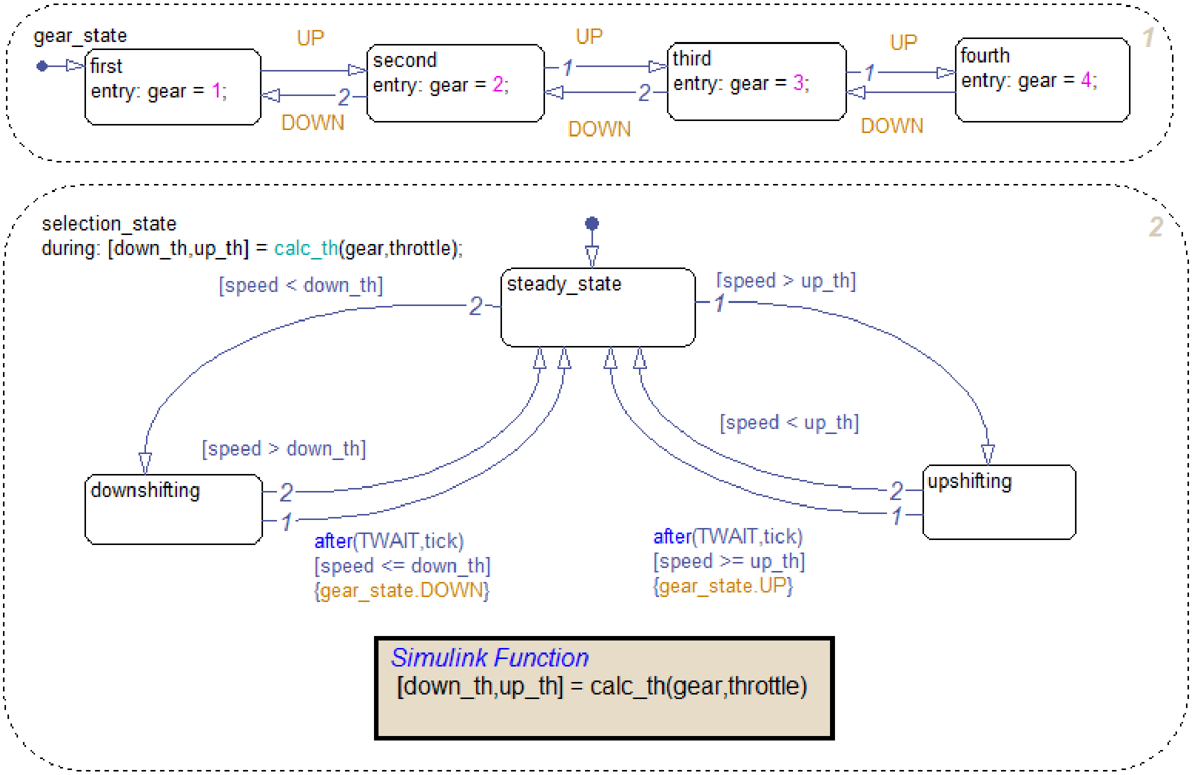
Modeling Method Without a Simulink Function

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.



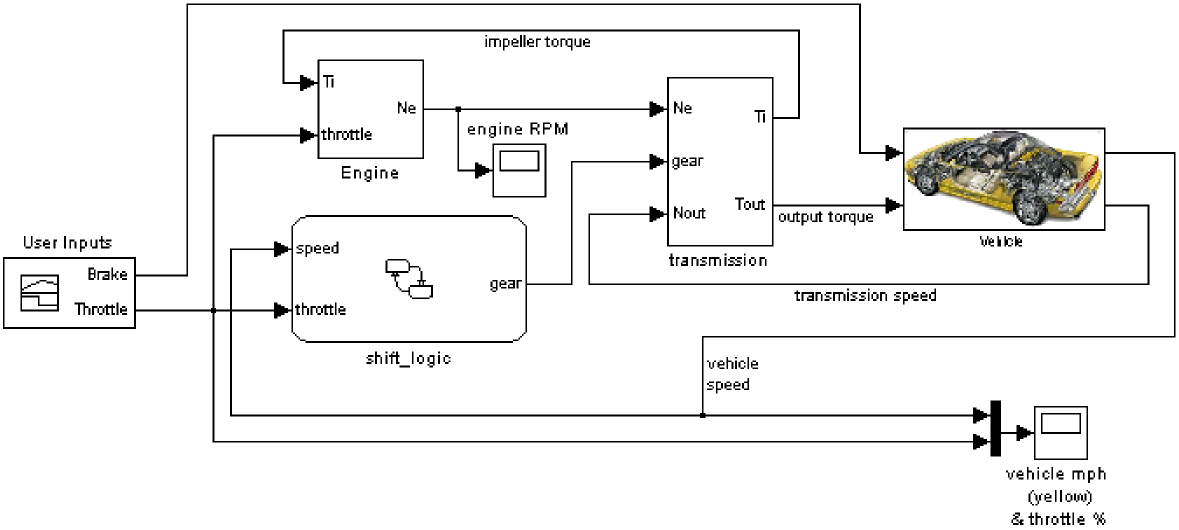
Modeling Method With a Simulink Function

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.



In the chart, the `during` action in `selection_state` contains a function call to `calc_th`, which is a function that contains Simulink blocks.

This modeling method minimizes the objects in your model.



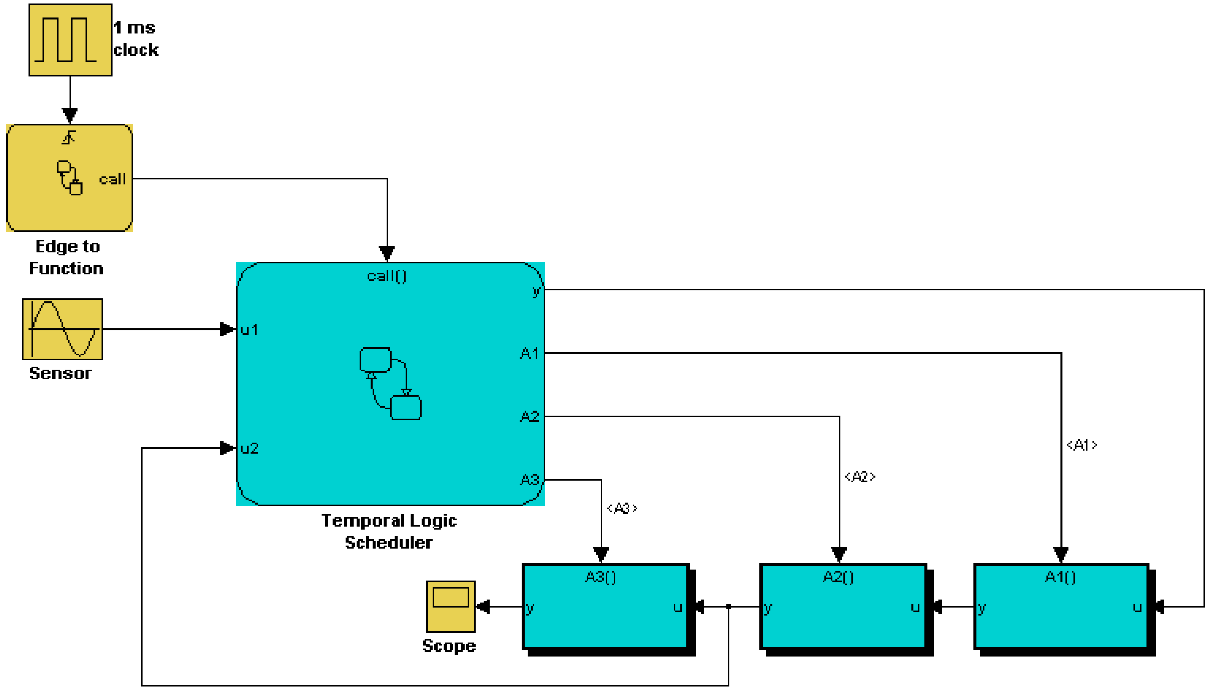
For more information, see "Defining a Function That Uses Simulink Blocks" on page 24-27.

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.

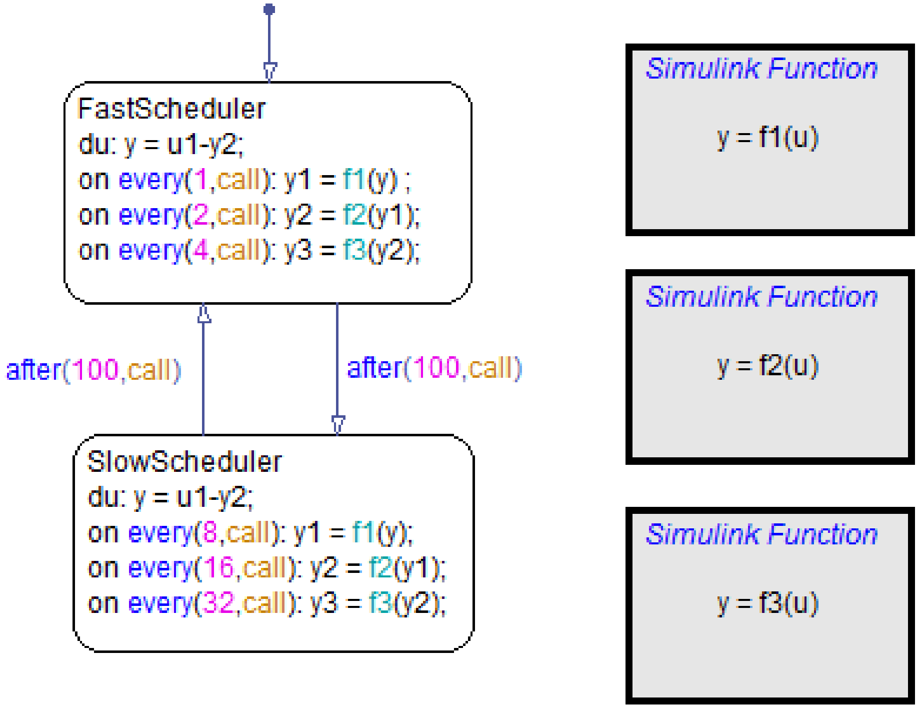
Modeling Method Without Simulink Functions

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.



Modeling Method With Simulink Functions

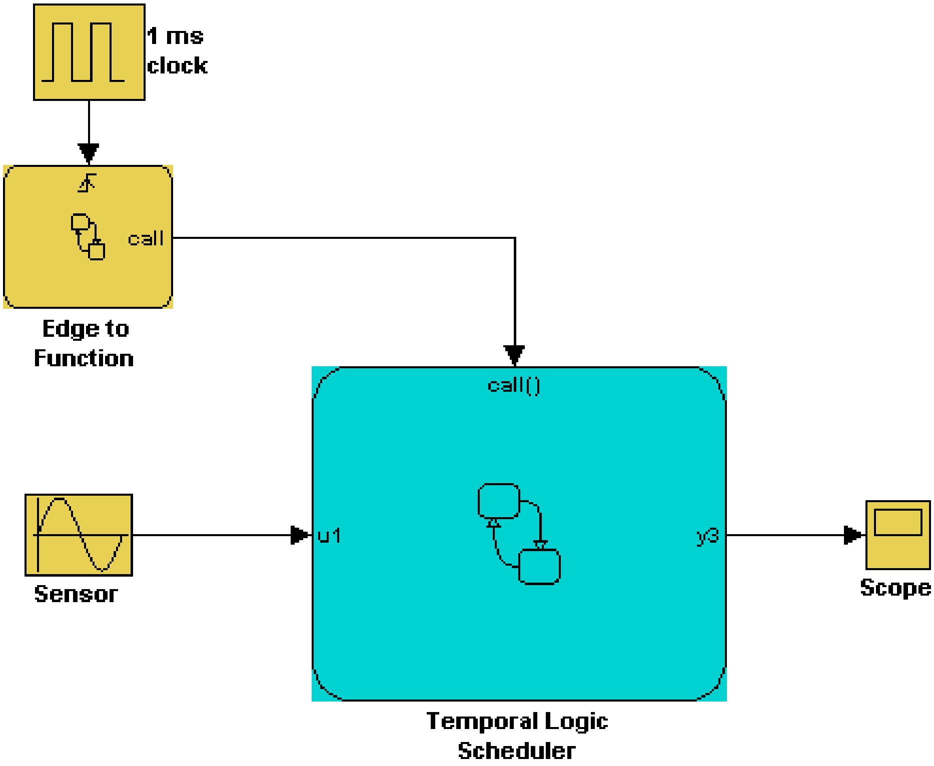
You define each controller as a Simulink function in a Stateflow chart and use function calls to schedule execution of the subsystems, as shown.



```
FastScheduler
du: y = u1-y2;
on every(1,call): y1 = f1(y) ;
on every(2,call): y2 = f2(y1);
on every(4,call): y3 = f3(y2);
```

```
SlowScheduler
du: y = u1-y2;
on every(8,call): y1 = f1(y);
on every(16,call): y2 = f2(y1);
on every(32,call): y3 = f3(y2);
```

This modeling method minimizes the objects in your model.



For more information, see "Scheduling Execution of Multiple Controllers" on page 24-36.

Where to Use a Simulink Function

A Simulink function can reside anywhere in a chart, state, or subchart. The location of a function determines its scope, that is, the set of states and transitions that can call the function. Follow these guidelines:

- If you want to call the function only within one state or subchart and its substates, put your Simulink function in that state or subchart. That function overrides any other functions of the same name in the parents and ancestors of that state or subchart.
- If you want to call the function anywhere in that chart, put your Simulink function at the chart level.

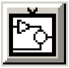
How to Define a Simulink Function in a Stateflow Chart

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)
```

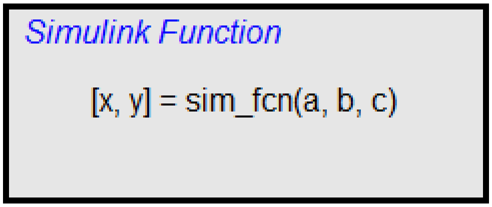
where `simfcn` is the name of your function, *`a_1`*, *`a_2`*, ..., *`a_n`* are formal names for the arguments, and *`r_1`*, *`r_2`*, ..., *`r_n`* are formal names for the return values.

Note This syntax is the same as what you use for graphical functions, truth tables, and MATLAB functions. You can define arguments and return values as scalars, vectors, or matrices of any data type.

---

4. Click outside the function box.

The following example shows a Simulink function that has the name `sim_fcn`, which takes three arguments (`a`, `b`, and `c`) and returns two values (`x` and `y`).



Simulink Function

[x, y] = sim_fcn(a, b, c)

Note You can also create and edit a Simulink function by using API methods. 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.

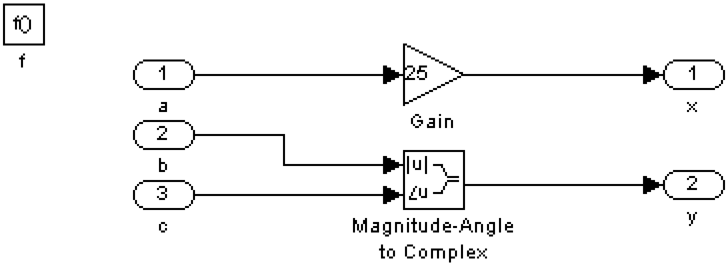
   The contents of the subsystem appear: input and output ports that match the function signature and a single function-call trigger port.

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 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 24-26.

How a Simulink Function Binds to a State

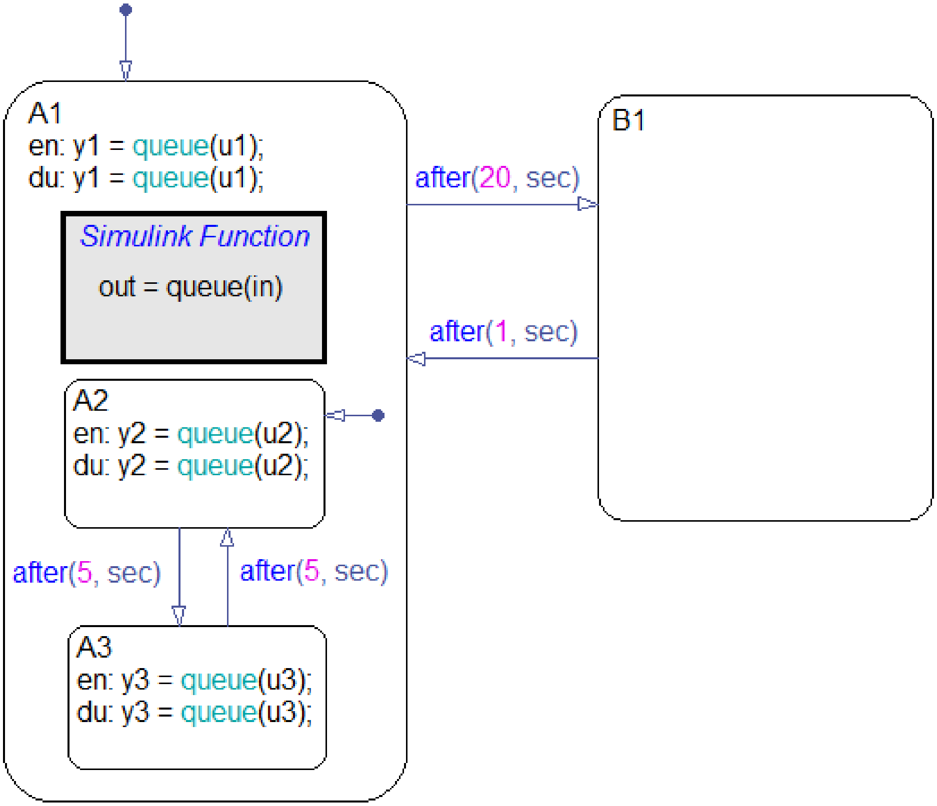
In this section...

Binding Behavior of a Simulink Function

When a Simulink function resides inside a state, the function binds to that state. Binding results in the following behavior:

- Function calls can occur only in state actions and on transitions within the state and its substates.
- When the state is entered, the function is enabled.
- When the state is exited, the function is disabled.

For example, the following Stateflow chart shows a Simulink function that binds to a state.



Because the function `queue` resides in state `A1`, the function binds to that state.

- State `A1` and its substates `A2` and `A3` can call `queue`, but state `B1` cannot.
- When state `A1` is entered, `queue` is enabled.
- When state `A1` is exited, `queue` is disabled.

Controlling Subsystem Variables When the Simulink Function Is Disabled

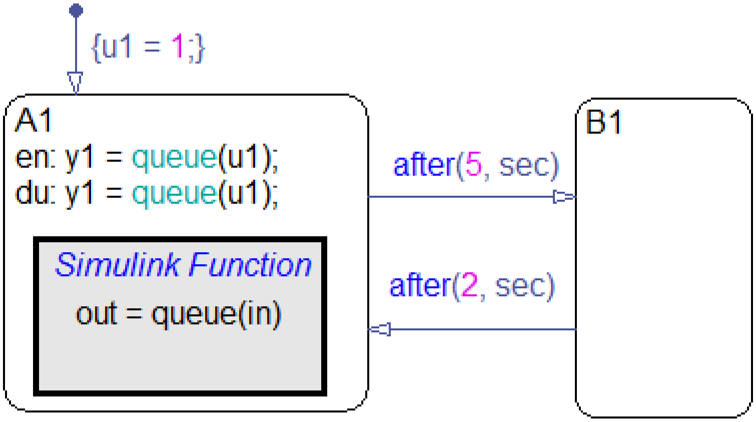
If a Simulink function binds to a state, you can hold the subsystem variables at their values from the previous execution or reset the variables to their initial values. Follow these steps:

1. In the Simulink function, double-click the trigger port to open the Block Parameters dialog box.
2. Select an option for **States when enabling**.

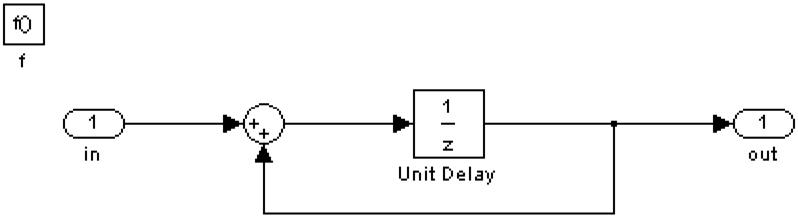
Option	Description	Reference Section
`held`	Holds the values of the subsystem variables from the previous execution	"How the Function Behaves When Variables Are Held" on page 24-21
`reset`	Resets the subsystem variables to their initial values	"How the Function Behaves When Variables Are Reset" on page 24-22

Example of Binding a Simulink Function to a State

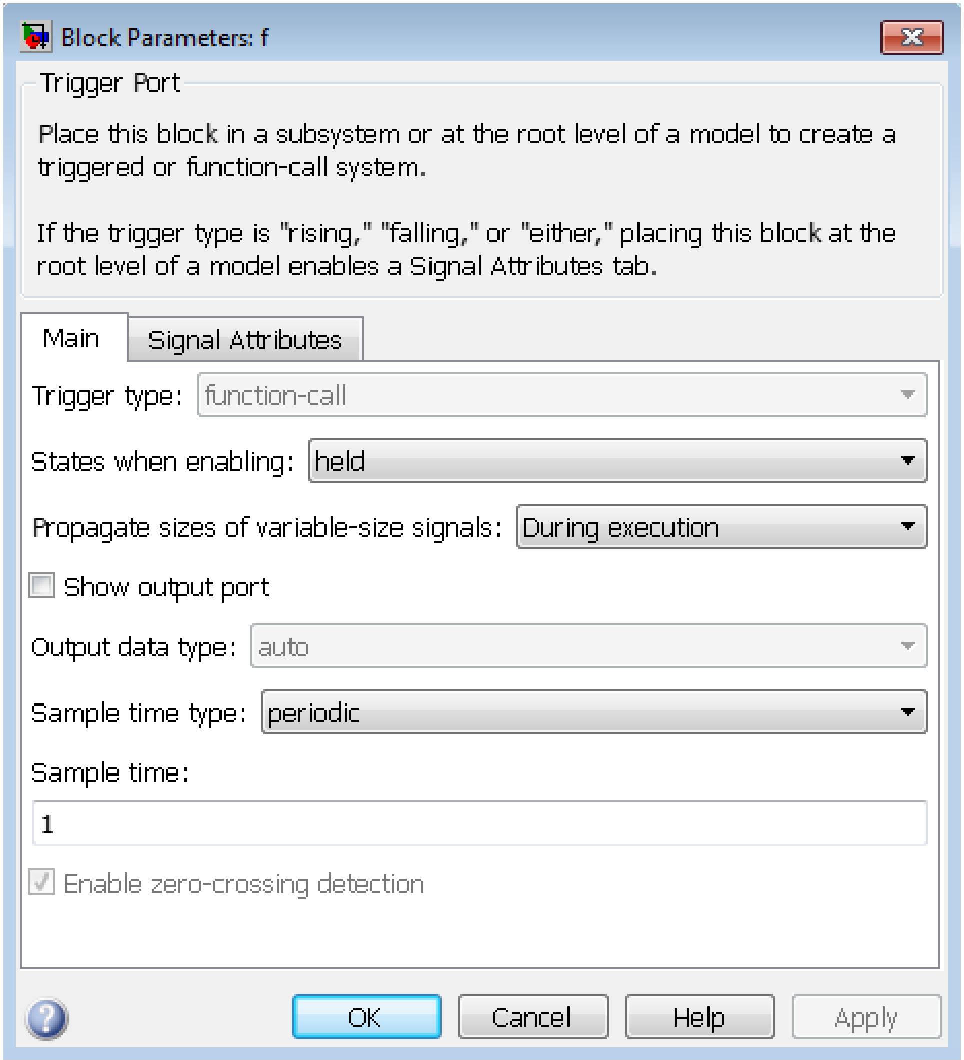
This example shows how a Simulink function behaves when bound to a state.



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.



Trigger Port

Place this block in a subsystem or at the root level of a model to create a triggered or function-call system.

If the trigger type is "rising," "falling," or "either," placing this block at the root level of a model enables a Signal Attributes tab.

Trigger type: function-call

States when enabling: held

Propagate sizes of variable-size signals: During execution

☐ Show output port

Output data type: auto

Sample time type: periodic

Sample time:

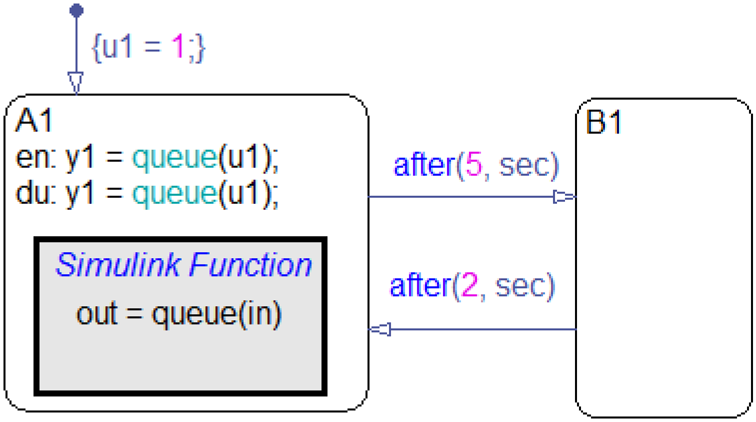
1

☑ Enable zero-crossing detection

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. (For more information, see "Solvers" in the Simulink documentation.)

Simulation Behavior of the Chart

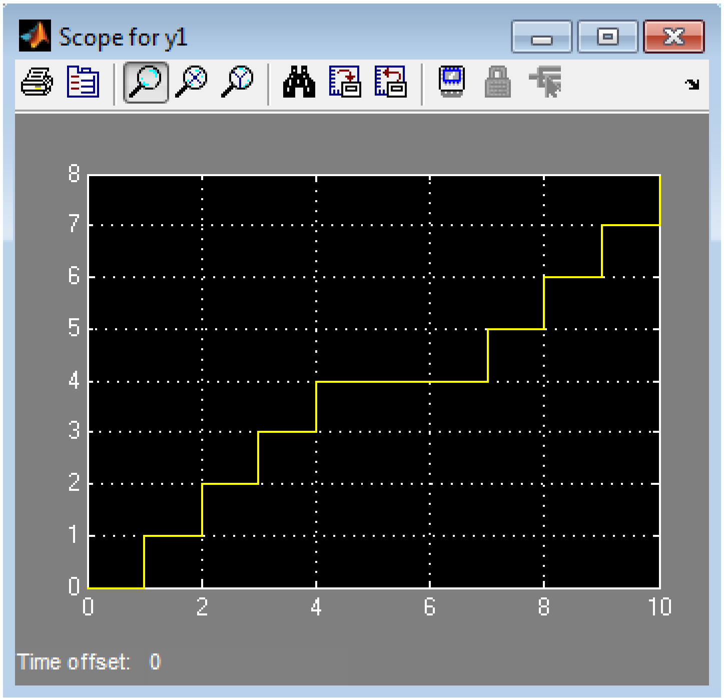


When you simulate the chart, the following actions occur.

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

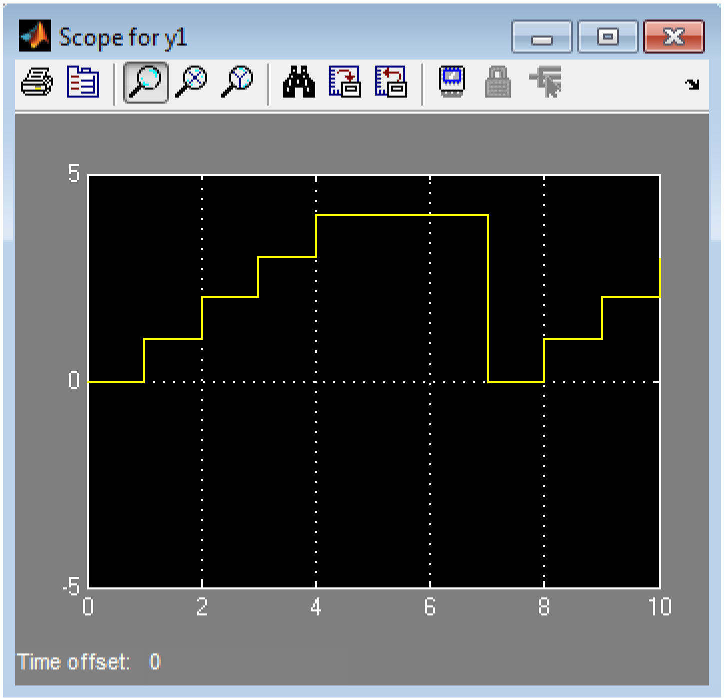
If you set **States when enabling** to `held`, the output `y1` is as follows.



When state A1 becomes inactive at t = 5, the Simulink function holds the counter value. When A1 is active again at t = 7, the counter has the same value as it did at t = 5. Therefore, the output `y1` continues to increment over time.

How the Function Behaves When Variables Are Reset

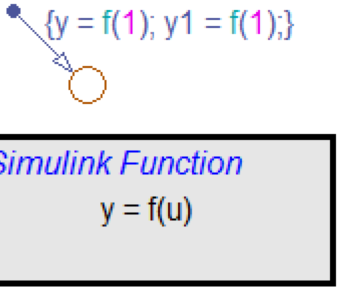
If you set **States when enabling** to `reset`, the output `y1` is as follows.



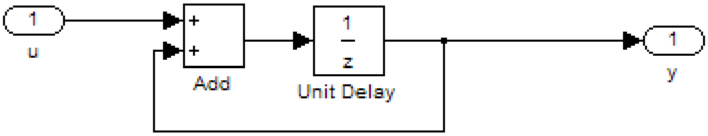
When state A1 becomes inactive at t = 5, the Simulink function does *not* hold the counter value. When A1 is active again at t = 7, the counter resets to zero. Therefore, the output `y1` resets too.

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 16, “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 6-13 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. HDL code generation does not support Simulink functions.

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.

Verify that function-call expressions have inputs and outputs of correct size

If the formal arguments of a function signature are scalars, verify that inputs and outputs of function calls follow the rules of scalar expansion. For more information, see "How Scalar Expansion Works for Functions" on page 13-6.

Avoid using machine-parented data with Simulink functions

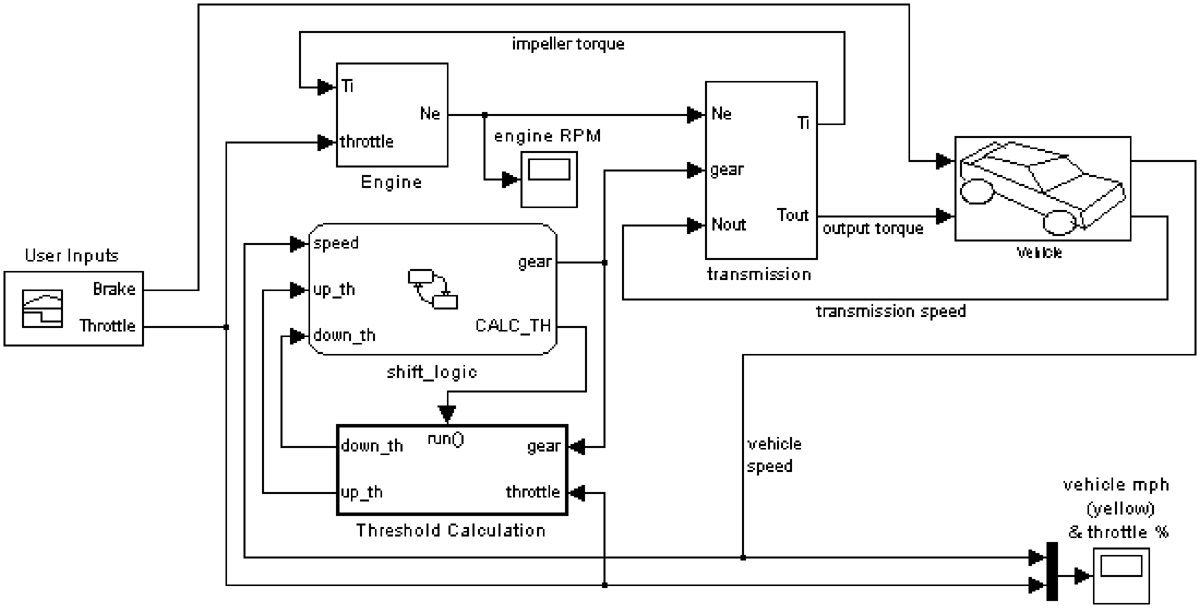
Use data store memory instead of machine-parented data. For more information, see "Sharing Global Data with Multiple Charts" on page 8-35.

Defining a Function That Uses Simulink Blocks

In this section...

Goal of the Tutorial

The goal of this tutorial is to use a Simulink function in a Stateflow chart to improve the design of a model named `old_sf_car`.



Rationale for Improving the Model Design

The `old_sf_car` model contains a function-call subsystem named Threshold Calculation and a Stateflow chart named shift_logic. The two blocks interact as follows:

- The chart broadcasts the output event `CALC_TH` to trigger the function-call subsystem.
- The subsystem uses lookup tables to interpolate two values for the shift_logic chart.
- The subsystem outputs (`up_th` and `down_th`) feed directly into the chart as inputs.

  No other blocks in the model access the subsystem outputs.

You can replace a function-call subsystem with a Simulink function in a chart when:

- The subsystem performs calculations required by the chart.
- Other blocks in the model do not need access to the subsystem outputs.

Editing a Model to Use a Simulink Function

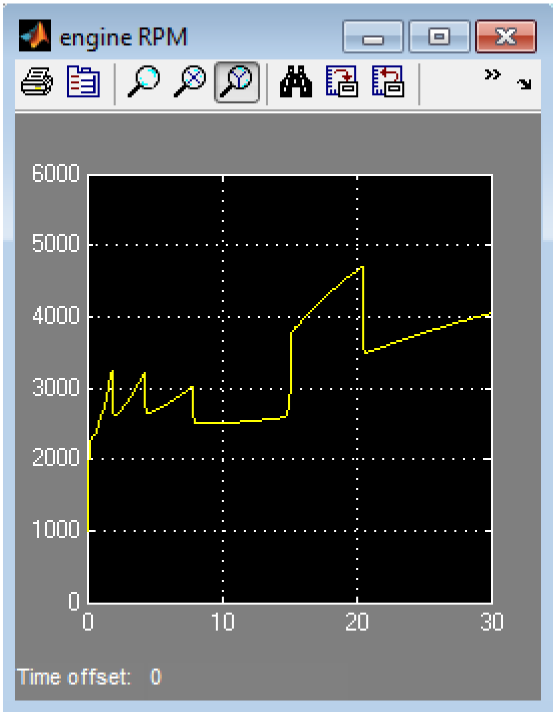
The sections that follow describe how to replace a function-call subsystem in a Simulink model with a Simulink function in a Stateflow chart. This procedure reduces the number of objects in the model while retaining the same simulation results.

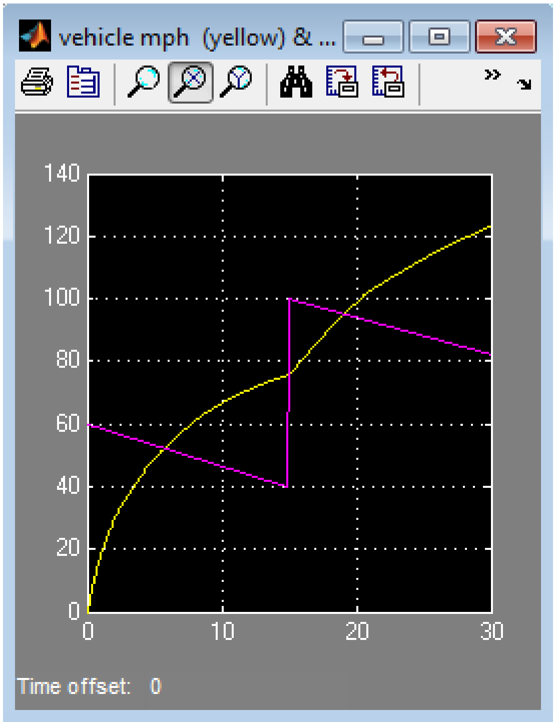
Step	Task	Reference
1	Open the model.	"Open the Model" on page 24-29
2	Move the contents of the function-call subsystem into a Simulink function in the chart.	"Add a Simulink Function to the Chart" on page 24-30
3	Change the scope of specific chart-level data to `Local`.	"Change the Scope of Chart Data" on page 24-33
4	Replace the event broadcast with a function call.	"Update State Action in the Chart" on page 24-34
5	Verify that function inputs and outputs are defined.	"Add Data to the Chart" on page 24-34
6	Remove unused items in the model.	"Remove Unused Items in the Model" on page 24-35

Note To skip the conversion steps and access the new model directly, type `sf_car` at the MATLAB command prompt.

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.

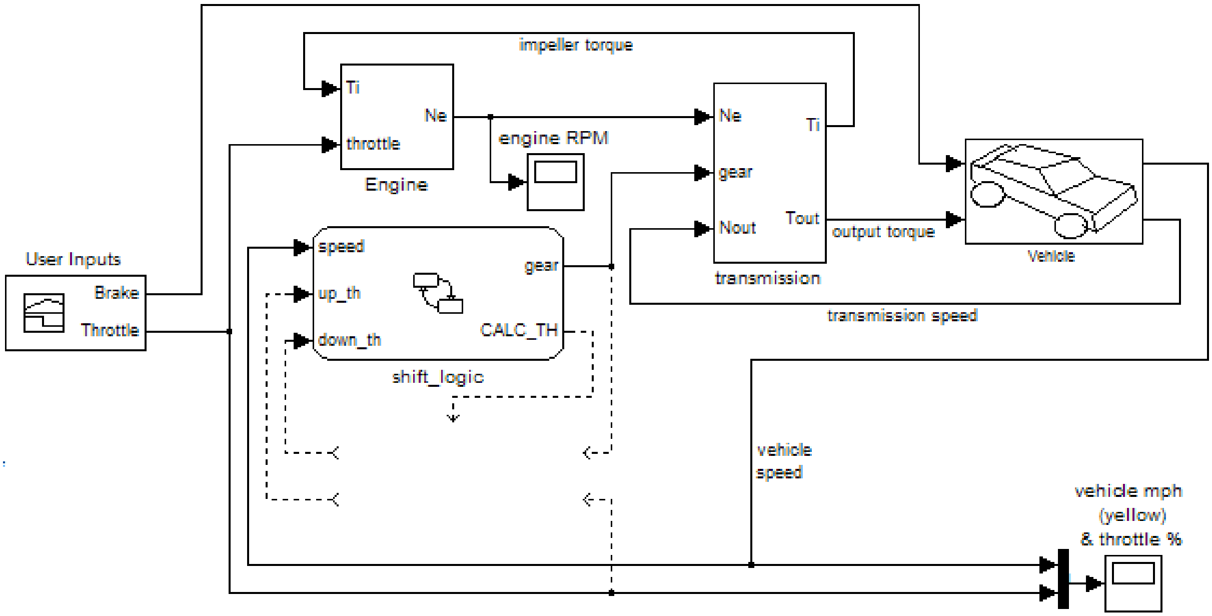




Add a Simulink Function to the Chart

Follow these steps to add a Simulink function to the shift_logic chart.

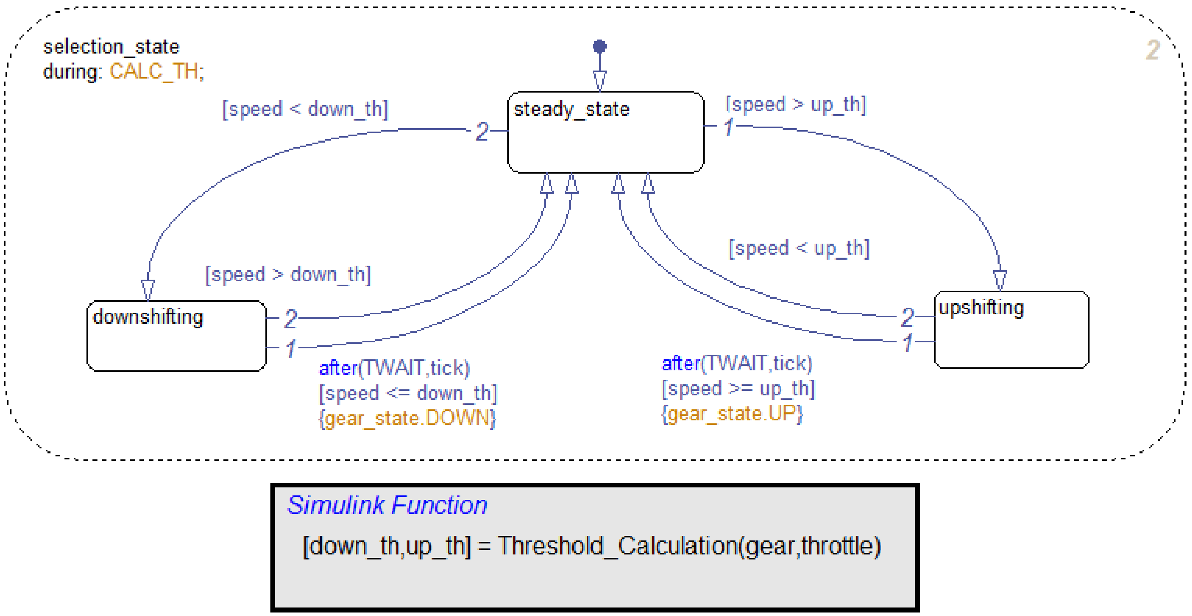
1 In the Simulink model, right-click the Threshold Calculation block in the lower left corner and select **Cut** from the context menu.



2 Open the shift_logic chart.

3 In the chart, right-click below `selection_state` and select **Paste** from the context menu.

4 Expand the new Simulink function so that the signature fits inside the function box.

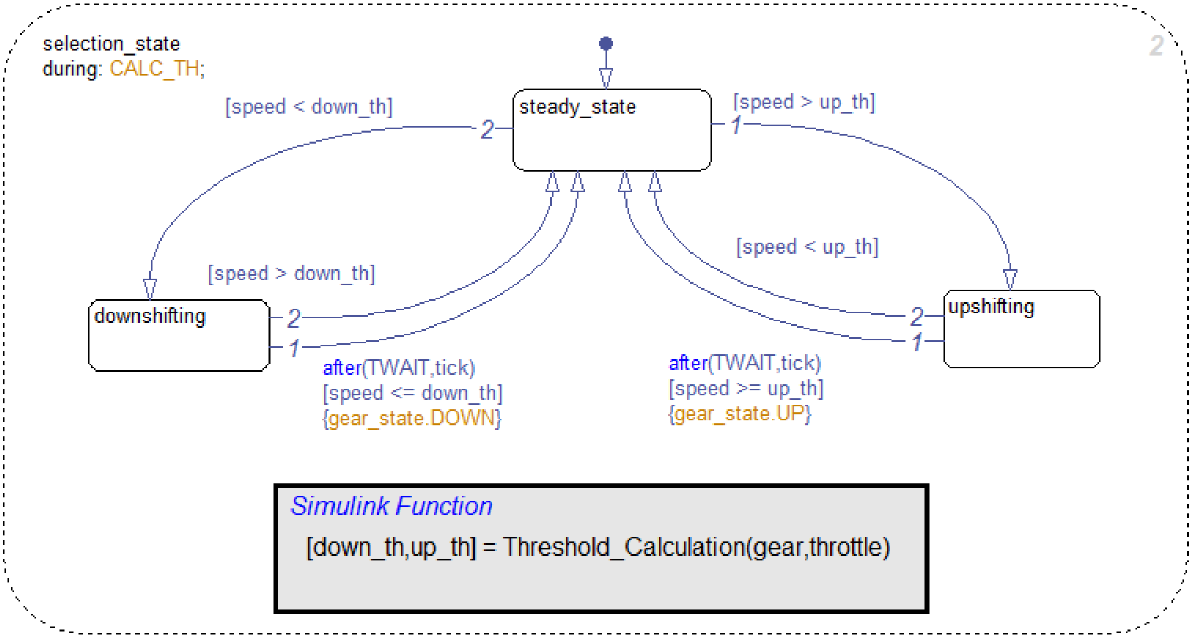


```
[down_th,up_th] = Threshold_Calculation(gear,throttle)
```

Tip To change the font size of a function, right-click the function box and select a new size from the **Font Size** menu.

---

5 Expand the border of `selection_state` to include the new function.



Note The function resides in this state instead of the chart level because no other state in the chart requires the function outputs `up_th` and `down_th`. See "How a Simulink Function Binds to a State" on page 24-15.

---

6 Rename the Simulink function from `Threshold_Calculation` to `calc_threshold` by entering `[down_th, up_th] = calc_threshold(gear, throttle)` in the function box.

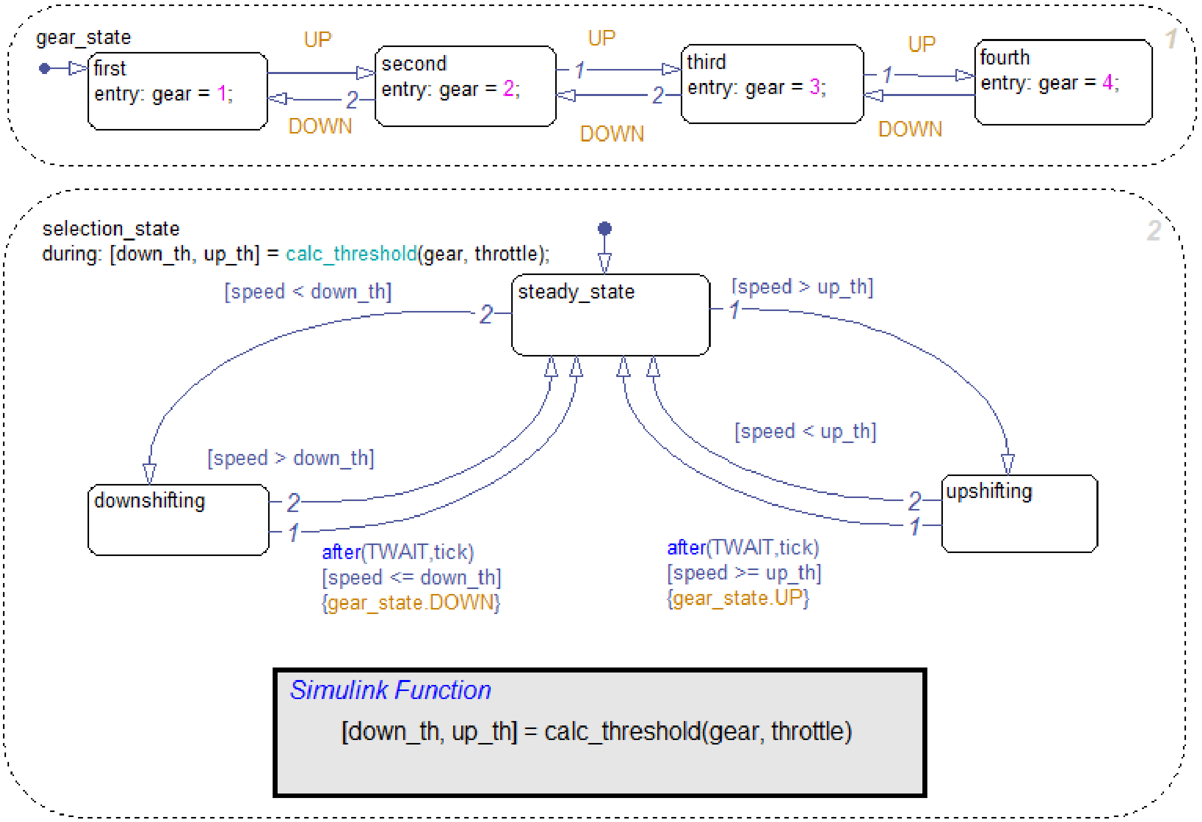
Change the Scope of Chart Data

In the Model Explorer, change the scope of chart-level data `up_th` and `down_th` to `Local` because calculations for those data now occur inside the chart.

Update State Action in the Chart

In the Stateflow Editor, change the `during` action in `selection_state` to call the Simulink function `calc_threshold`.

```
during:  [down_th, up_th] = calc_threshold(gear, throttle);
```



Add Data to the Chart

Because the function `calc_threshold` takes `throttle` as an input, you must define that data as a chart input. (For details, see “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.

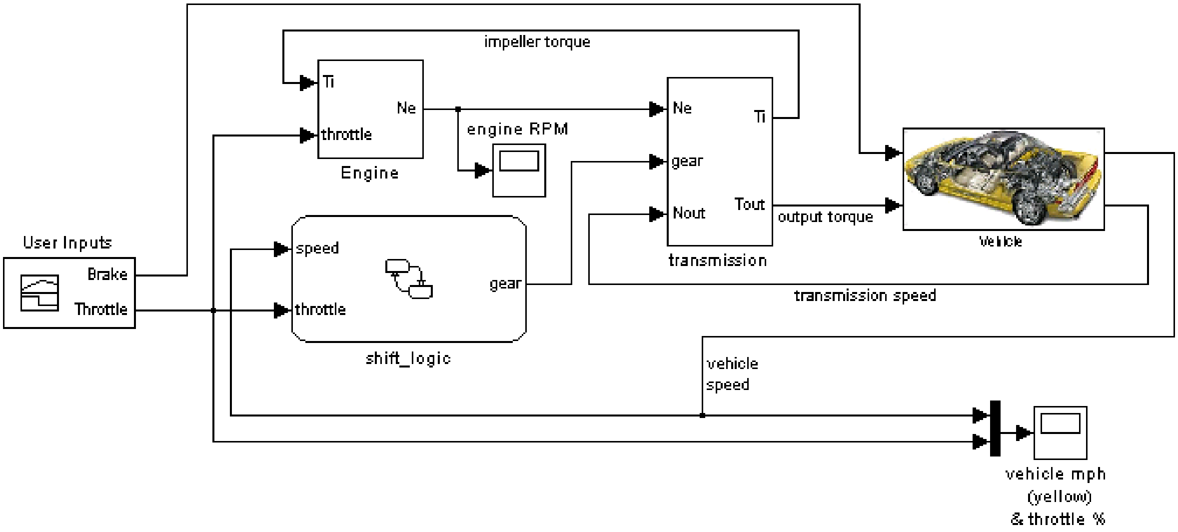
Remove Unused Items in the Model

1 In the Model Explorer, delete the function-call output event `CALC_TH` because the Threshold Calculation block no longer exists.

2 Delete any dashed signal lines from your model.

Running the New Model

Your new model looks something like this:



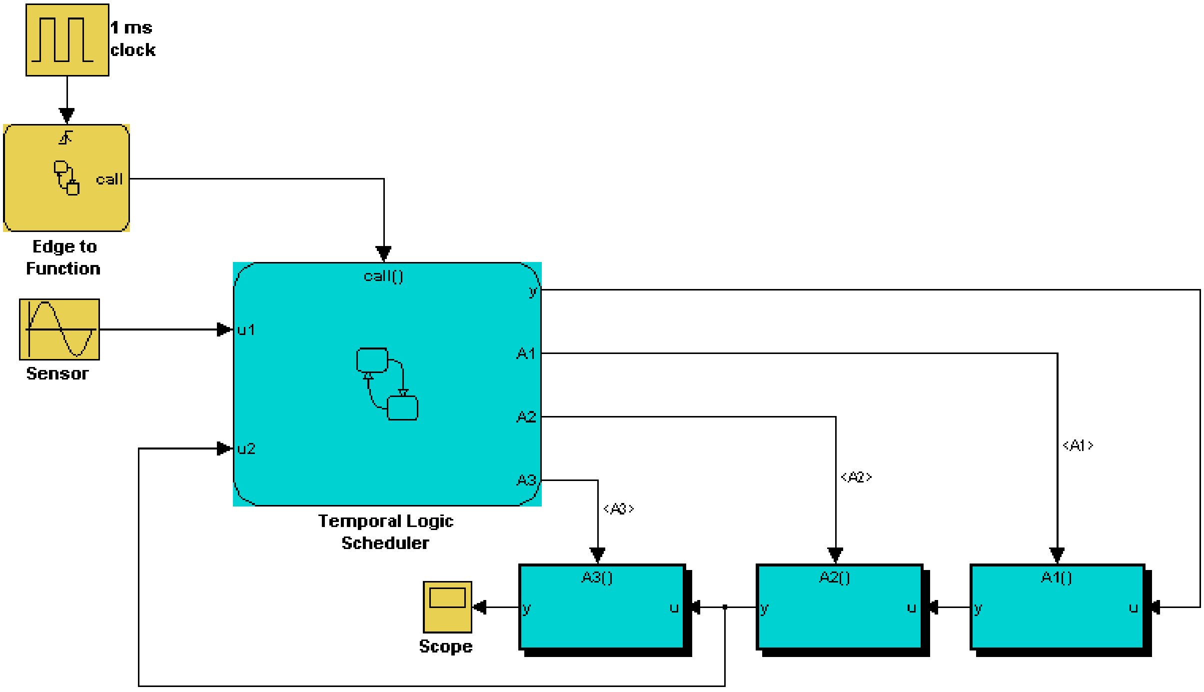
If you simulate the new model, the results match those of the original design.

Scheduling Execution of Multiple Controllers

In this section...
"Goal of the Tutorial" on page 24-36
"Editing a Model to Use Simulink Functions" on page 24-37
"Running the New Model" on page 24-44

Goal of the Tutorial

The goal of this tutorial is to use Simulink functions in a Stateflow chart to improve the design of a model named `sf_temporal_logic_scheduler`.



Rationale for Improving the Model Design

The `sf_temporal_logic_scheduler` model contains a Stateflow chart and three function-call subsystems. These blocks interact as follows:

- The chart broadcasts the output events `A1`, `A2`, and `A3` to trigger the function-call subsystems.
- The subsystems A1, A2, and A3 execute at different rates defined by the chart.
- The subsystem outputs feed directly into the chart.

  No other blocks in the model access the subsystem outputs.

You can replace function-call subsystems with Simulink functions inside a chart when:

- The subsystems perform calculations required by the chart.
- Other blocks in the model do not need access to the subsystem outputs.

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.

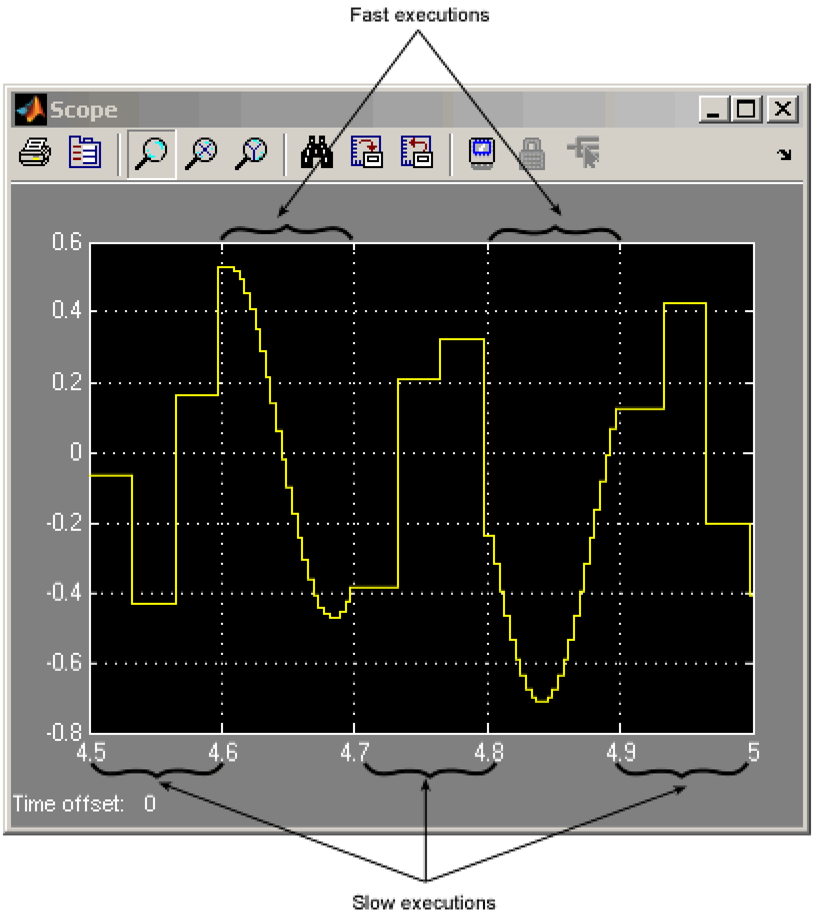
Step	Task	Reference
1	Open the model.	"Open the Model" on page 24-39
2	Move the contents of the function-call subsystems into Simulink functions in the chart.	"Add Simulink Functions to the Chart" on page 24-39
3	Change the scope of specific chart-level data to `Local`.	"Change the Scope of Chart Data" on page 24-42
4	Replace event broadcasts with function calls.	"Update State Actions in the Chart" on page 24-43

Step	Task	Reference
5	Verify that function inputs and outputs are defined.	"Add Data to the Chart" on page 24-43
6	Remove unused items in the model.	"Remove Unused Items in the Model" on page 24-44

Note To skip the conversion steps and access the new model directly, type `sf_temporal_logic_scheduler_with_sl_fcns` at the MATLAB command prompt.

Open the Model

Type `sf_temporal_logic_scheduler` at the MATLAB command prompt. If you simulate the model, you see this result in the scope.

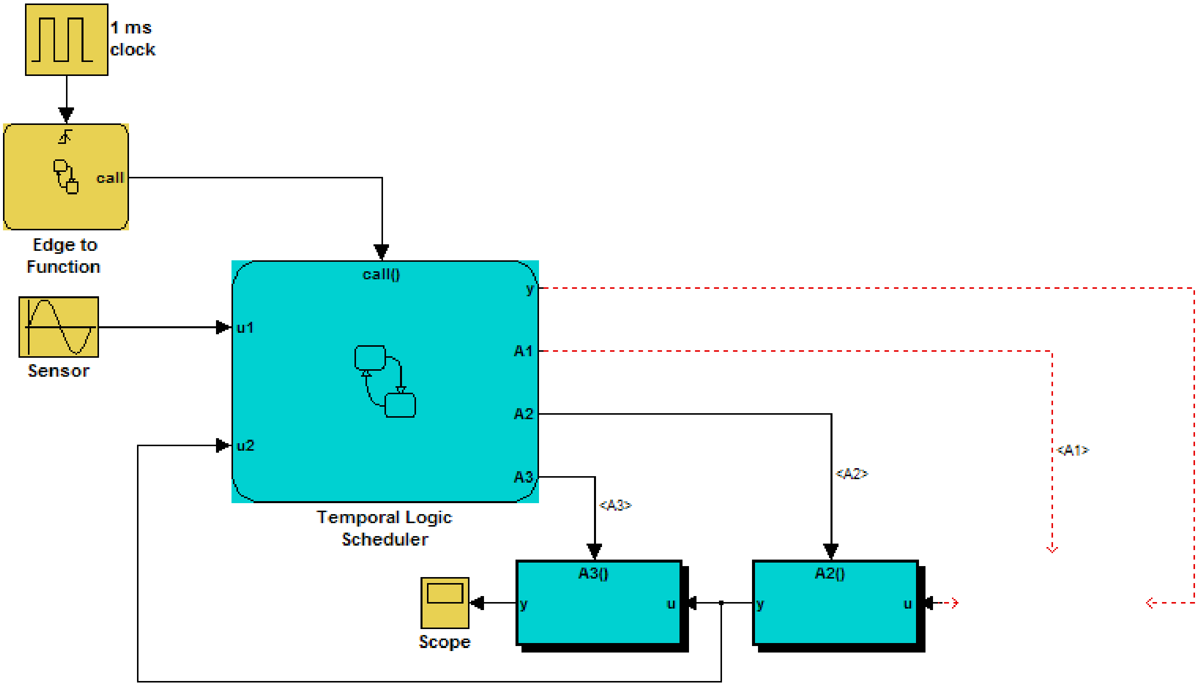


For more information, see “Scheduling Subsystems to Execute at Specific Times” on page 21-18.

Add Simulink Functions to the Chart

Follow these steps to add Simulink functions to the Temporal Logic Scheduler chart.

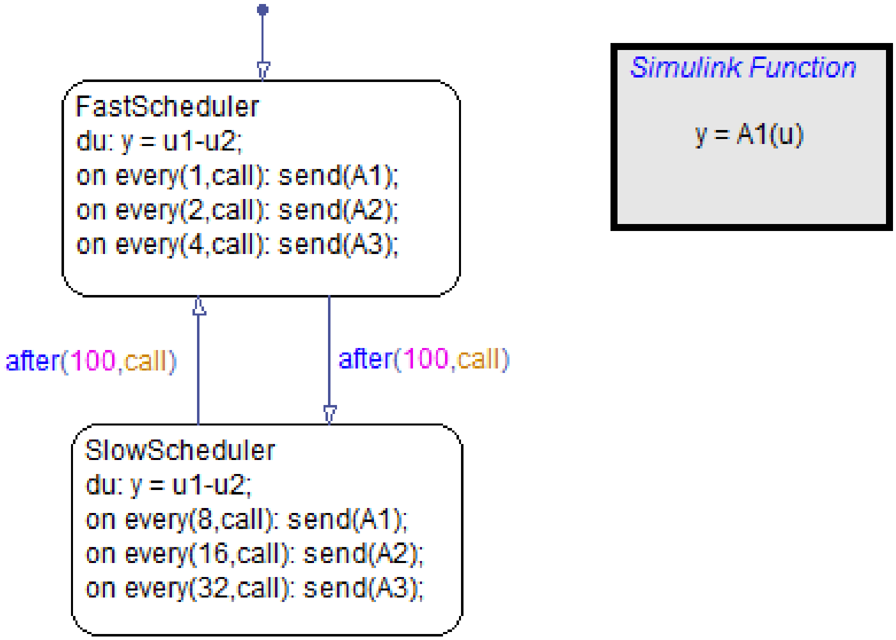
1 In the Simulink model, right-click the A1 block in the lower right corner and select **Cut** from the context menu.



2 Open the Temporal Logic Scheduler chart.

3 In the chart, right-click outside any states and select **Paste** from the context menu.

4 Expand the new Simulink function so that the signature fits inside the function box.



```
FastScheduler
du: y = u1-u2;
on every(1,call): send(A1);
on every(2,call): send(A2);
on every(4,call): send(A3);
```

```
SlowScheduler
du: y = u1-u2;
on every(8,call): send(A1);
on every(16,call): send(A2);
on every(32,call): send(A3);
```

```
Simulink Function

y = A1(u)
```

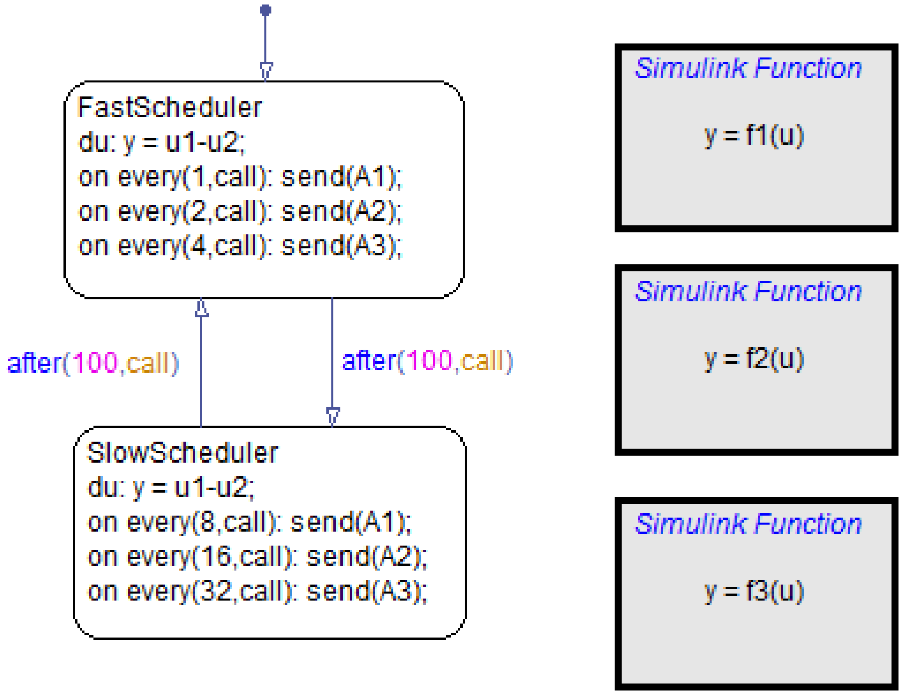
Tip To change the font size of a function, right-click the function box and select a new size from the **Font Size** menu.

---

5 Rename the Simulink function from `A1` to `f1` by entering `y = f1(u)` in the function box.

6 Repeat steps 1 through 5 for these cases:

- Copying the contents of A2 into a Simulink function named `f2`.
- Copying the contents of A3 into a Simulink function named `f3`.



Note The new functions reside at the chart level because both states `FastScheduler` and `SlowScheduler` require access to the function outputs.

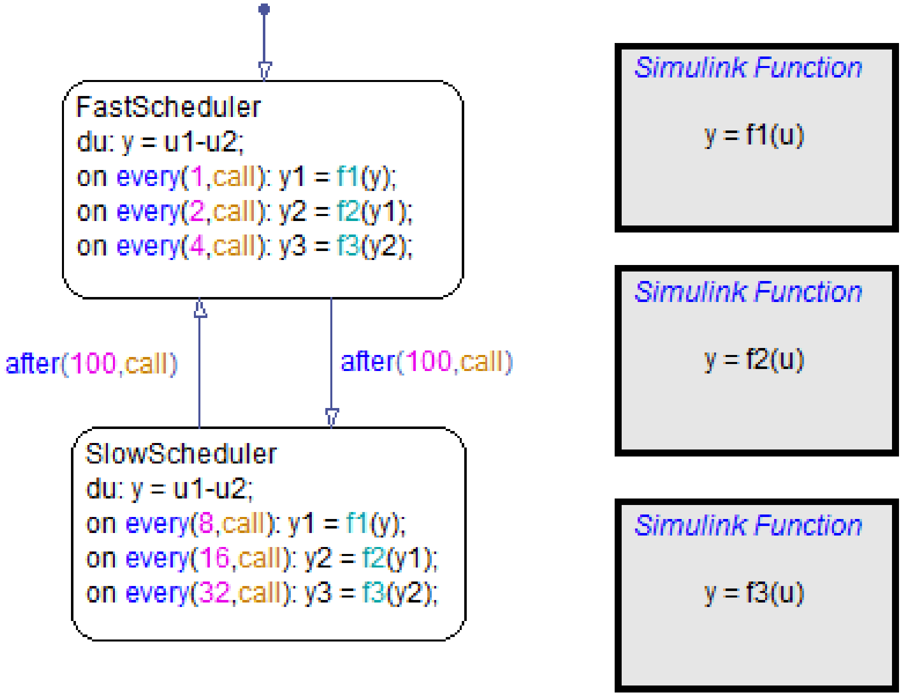
Change the Scope of Chart Data

In the Model Explorer, change the scope of chart-level data `y` to `Local` because the calculation for that data now occurs inside the chart.

Update State Actions in the Chart

In the Stateflow Editor, you can replace event broadcasts in state actions with function calls.

1 Edit the state actions in `FastScheduler` and `SlowScheduler` to call the Simulink functions `f1`, `f2`, and `f3`.



```
FastScheduler
du: y = u1-u2;
on every(1,call): y1 = f1(y);
on every(2,call): y2 = f2(y1);
on every(4,call): y3 = f3(y2);
```

```
after(100,call)
```

```
after(100,call)
```

```
SlowScheduler
du: y = u1-u2;
on every(8,call): y1 = f1(y);
on every(16,call): y2 = f2(y1);
on every(32,call): y3 = f3(y2);
```

```
y = f1(u)
```

```
y = f2(u)
```

```
y = f3(u)
```

2 In both states, update each `during` action as follows.

```
du: y = u1-y2;
```

Add Data to the Chart

For the `on every` state actions of `FastScheduler` and `SlowScheduler`, define three data. (For details, see “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.

Tip To flip the Scope block, right-click and select **Format > Flip Block** from the context menu.

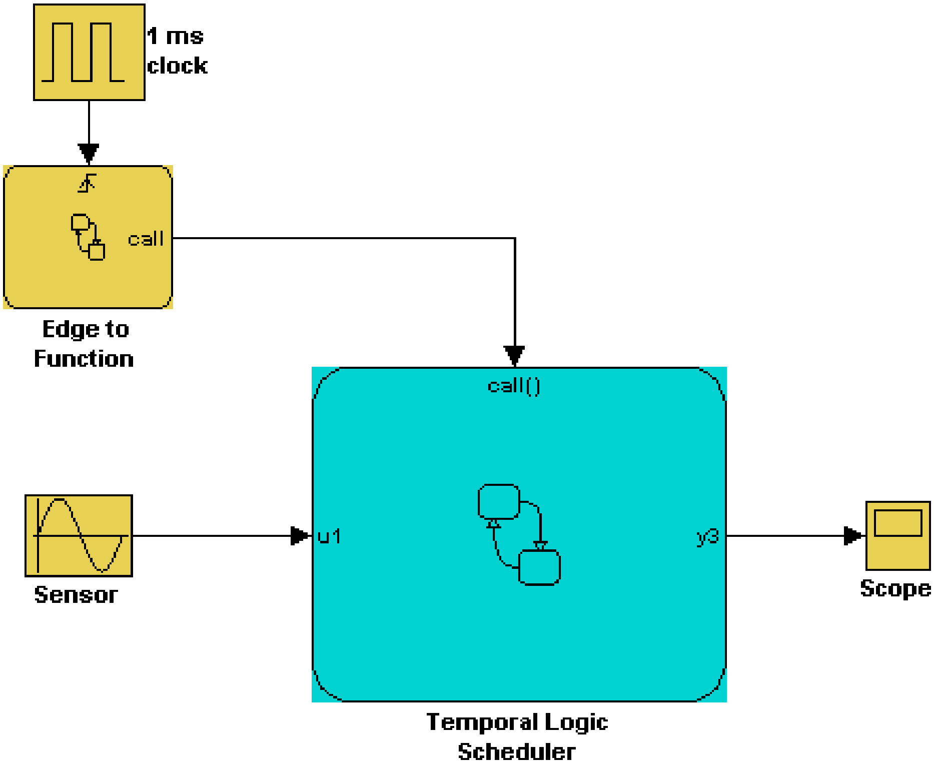
Remove Unused Items in the Model

1. In the Model Explorer, delete output events `A1`, `A2`, and `A3` and input data `u2` because the function-call subsystems no longer exist.

2. Delete any dashed signal lines from your model.

Running the New Model

Your new model looks something like this:



If you simulate the new model, the results match those of the original design.

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 25-74). 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.

Simulink Coder software can 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	Simulink Coder

The default target type of Simulink Coder code generation is generic real-time (`grt`). To build other targets, you must have the appropriate license. See "Available Targets" in the Simulink Coder documentation for more information.

Choosing a Procedure to Simulate a Model

In this section...

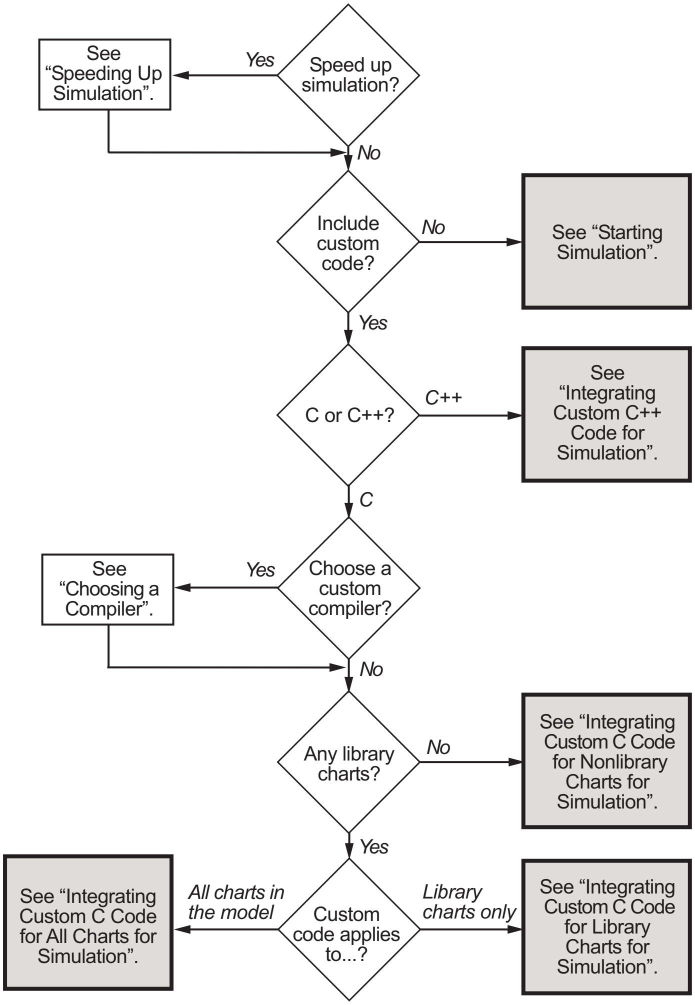
Guidelines for Simulation

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 25-16.
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 25-43.
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 25-6. These procedures apply to models that contain Chart or Truth Table blocks.


See "Speeding Up Simulation".
Yes
Speed up simulation?
No
Include custom code?
No
See "Starting Simulation".
Yes
C or C++?
C++
See "Integrating Custom C++ Code for Simulation".
C
See "Choosing a Compiler".
Yes
Choose a custom compiler?
No
Any library charts?
No
See "Integrating Custom C Code for Nonlibrary Charts for Simulation".
Yes
See "Integrating Custom C Code for All Charts for Simulation".
All charts in the model
Custom code applies to...?
Library charts only
See "Integrating Custom C Code for Library Charts for Simulation".

Procedures for Simulation

In this section...

Starting Simulation

Simulate your model by clicking the play button in the toolbar of the editor. See "Generated Code Files for Targets You Build" on page 25-74 for details about the simulation code you generate for your model and the folder structure.

For information on setting simulation options using the command-line API, see "Using Command-Line API to Set Simulation and Code Generation Parameters" on page 25-31.

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 Open the Configuration Parameters dialog box.

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 **Code Generation** 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 25-43.

Task 4: Simulate the Model

For instructions, see “Starting Simulation” on page 25-6.

Integrating Custom C Code for Nonlibrary Charts for Simulation

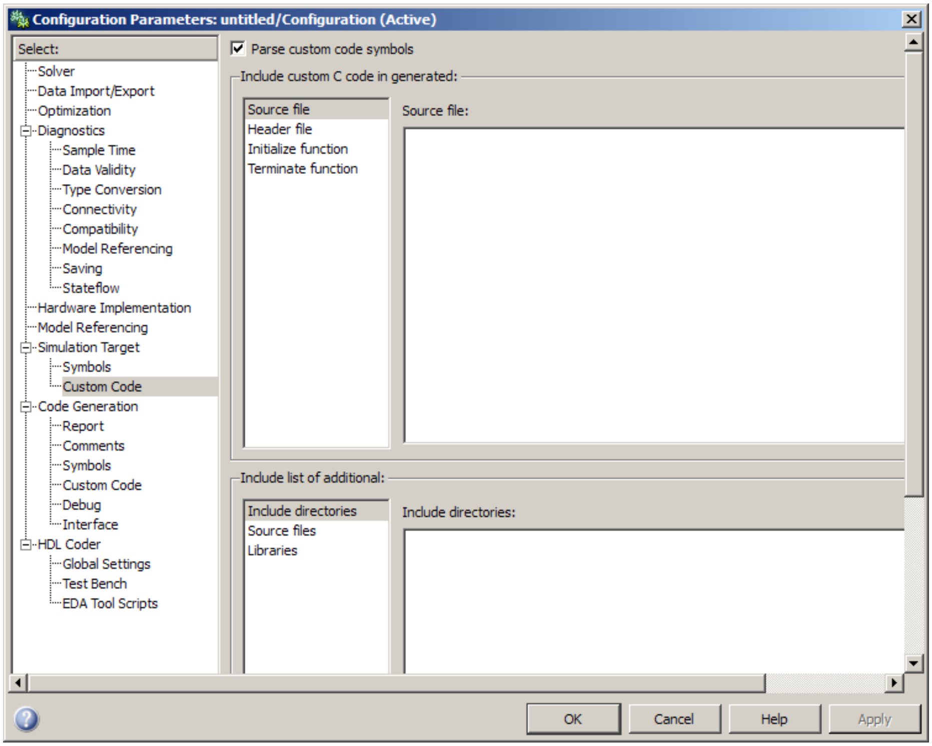
To integrate custom C code that applies to nonlibrary charts for simulation, perform the tasks that follow.

Task 1: Include Custom C Code in the Simulation Target

Specify custom code options in the simulation target for your model:

1. Open the Configuration Parameters dialog box.
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 25-41.

- **Source file** — Enter code lines to include at the top of a generated source code file. These code lines appear at the top of the generated *`model`*`.c` source file, outside of any function.

  For example, you can include `extern int` declarations for global variables.

- **Header file** — Enter code lines to include at the top of the generated *`model`*`.h` header file that declares custom functions and data in the generated code. These code lines appear at the top of all generated source code files and are visible to all generated code.

Note When you include a custom header file, you must enclose the file name in double quotes. For example, `#include ''sample_header.h''` is a valid declaration for a custom header file.

Since the code you specify in this option appears in multiple source files that link into a single binary, limitations exist on what you can include. For example, do not include a global variable definition such as `int x;` or a function body such as

```
void myfun(void)
{
...
}
```

These code lines cause linking errors because their symbol definitions appear multiple times in the source files of the generated code. You can, however, include `extern` declarations of variables or functions such as `extern int x;` or `extern void myfun(void);`.

- **Initialize function** — Enter code statements that execute once at the start of simulation. Use this code to invoke functions that allocate memory or perform other initializations of your custom code.
- **Terminate function** — Enter code statements that execute at the end of simulation. Use this code to invoke functions that free memory allocated by the custom code or perform other cleanup tasks.
- **Include directories** — Enter a space-separated list of the folder paths that contain custom header files that you include either directly (see **Header file** option) or indirectly in the compiled target.
- **Source files** — Enter a list of source files to compile and link into the target. You can separate source files with a comma, a space, or a new line.
- **Libraries** — Enter a space-separated list of static libraries that contain custom object code to link into the target.

4 Click **OK**.

Tip If you want to rebuild the target to include custom code changes, select **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 25-6.

Integrating Custom C Code for Library Charts for Simulation

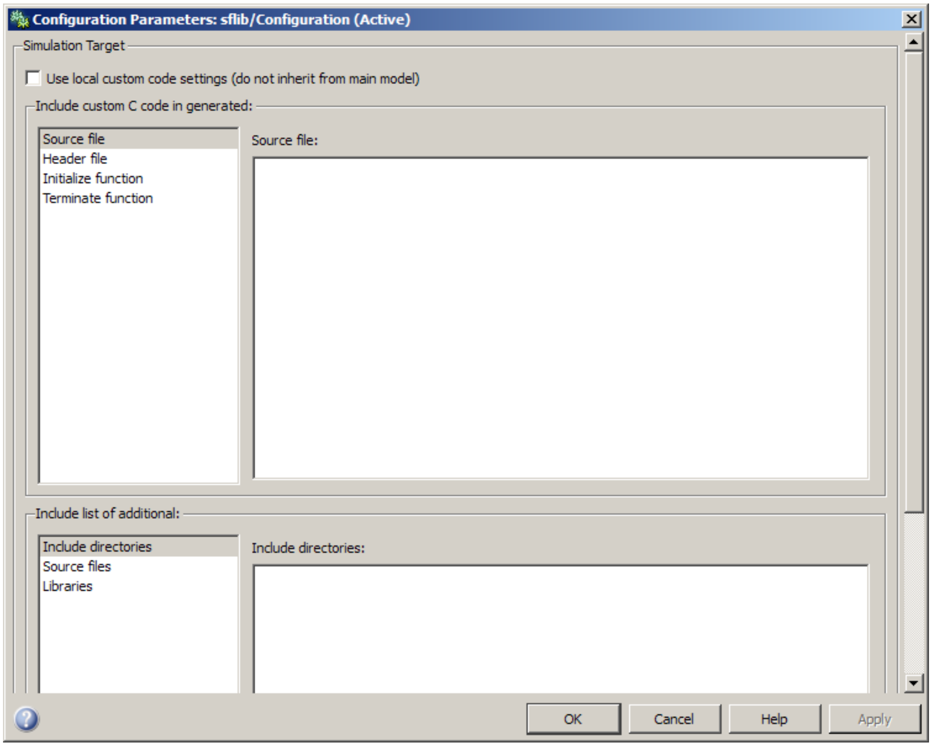
To integrate custom C code that applies only to library charts for simulation, perform the tasks that follow.

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 step ensures that each library model retains its own custom code settings during simulation.

3 Specify your custom code in the subpanes.

Follow the guidelines in “Specifying Relative Paths for Custom Code” on page 25-41.

Note See “Task 1: Include Custom C Code in the Simulation Target” on page 25-8 for descriptions of the custom code options.

---

4. Click **OK**.

Task 2: Simulate the Model

For instructions, see “Starting Simulation” on page 25-6.

Integrating Custom C Code for All Charts for Simulation

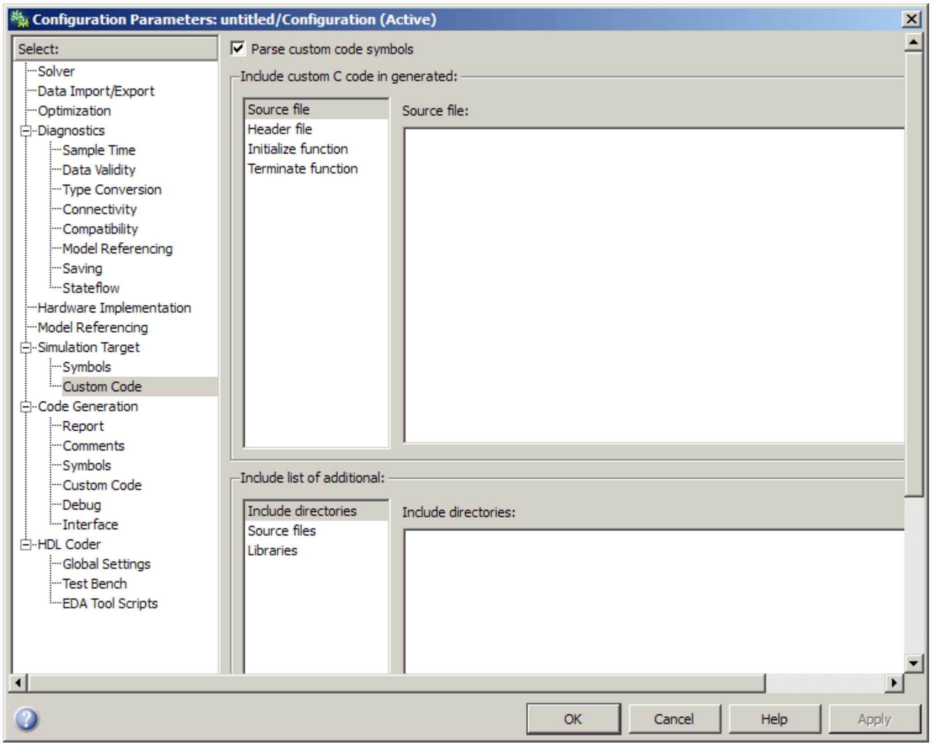
To integrate custom C code that applies to all charts for simulation, perform the tasks that follow.

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. Open the Configuration Parameters dialog box.
2. In the Configuration Parameters dialog box, select the **Simulation Target > Custom Code** pane.

The custom code options appear.



☑ Parse custom code symbols

3 Specify your custom code in the subpanes.

Follow the guidelines in “Specifying Relative Paths for Custom Code” on page 25-41.

Note See “Task 1: Include Custom C Code in the Simulation Target” on page 25-8 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**.
2. In the **Simulation Target** pane, clear the **Use local custom code settings (do not inherit from main model)** check box.

   This step ensures that library charts inherit the custom code settings of your main model.
3. Click **OK**.

Task 3: Simulate the Model

For instructions, see “Starting Simulation” on page 25-6.

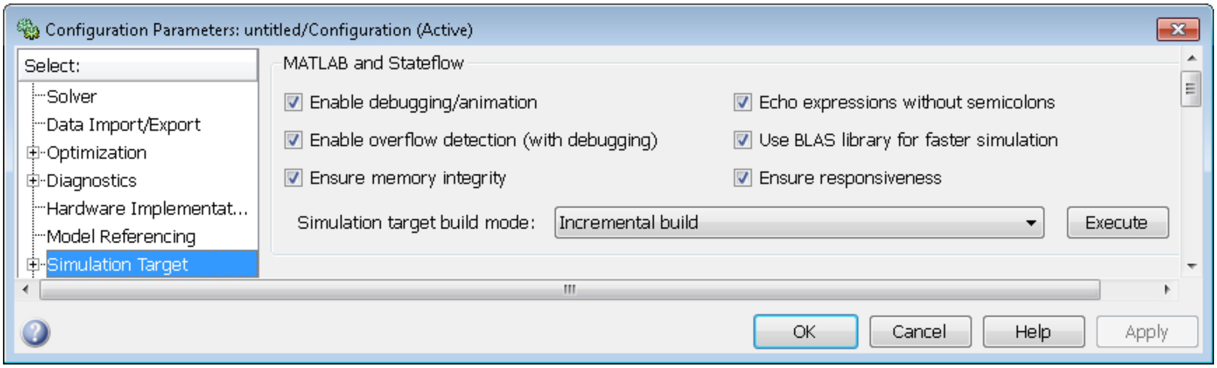
Speeding Up Simulation

In this section...

Disable Simulation Target Options That Impact Execution Speed

To simulate your model more quickly, disable options as described in the steps that follow:

1 Open the Configuration Parameters dialog box and select the **Simulation Target** pane.



Configuration Parameters: untitled/Configuration (Active)

Select:
- Solver
- Data Import/Export
- Optimization
- Diagnostics
- Hardware Implementat...
- Model Referencing
- Simulation Target

MATLAB and Stateflow

☑ Enable debugging/animation

☑ Enable overflow detection (with debugging)

☑ Ensure memory integrity

☑ Echo expressions without semicolons

☑ Use BLAS library for faster simulation

☑ Ensure responsiveness

Simulation target build mode: Incremental build

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 "Detecting Overflow for Fixed-Point Types" on page 17-11.

To detect overflow in data during simulation, you must also select the **Data Range** check box in the Debugger window. See "Data Range Violations in a Chart" on page 26-31 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**.

Keep Charts Closed During Simulation

During model simulation, any open charts with animation enabled take longer to simulate. If you keep all charts closed, you can speed up the simulation.

Keep Scope Blocks Closed During Simulation

During model simulation, any open Scope blocks continuously update their display. If you keep all Scope blocks closed, you can speed up the simulation. After the simulation ends, you can open the Scope blocks to view the results.

Use Library Charts in Your Model

When your model has multiple charts that contain the same elements, you might generate multiple copies of identical simulation code. By using library charts, you can minimize the number of copies of identical simulation code.

For example, using five library charts reduces the number of identical copies from five down to one.

For more information about using library charts, see “Creating Specialized Chart Libraries for Large-Scale Modeling” on page 19-20.

Choosing a Procedure to Generate Embeddable Code for a Model

In this section…
"Guidelines for Embeddable Code Generation" on page 25-19
"Choosing the Right Procedure for Embeddable Code Generation" on page 25-19

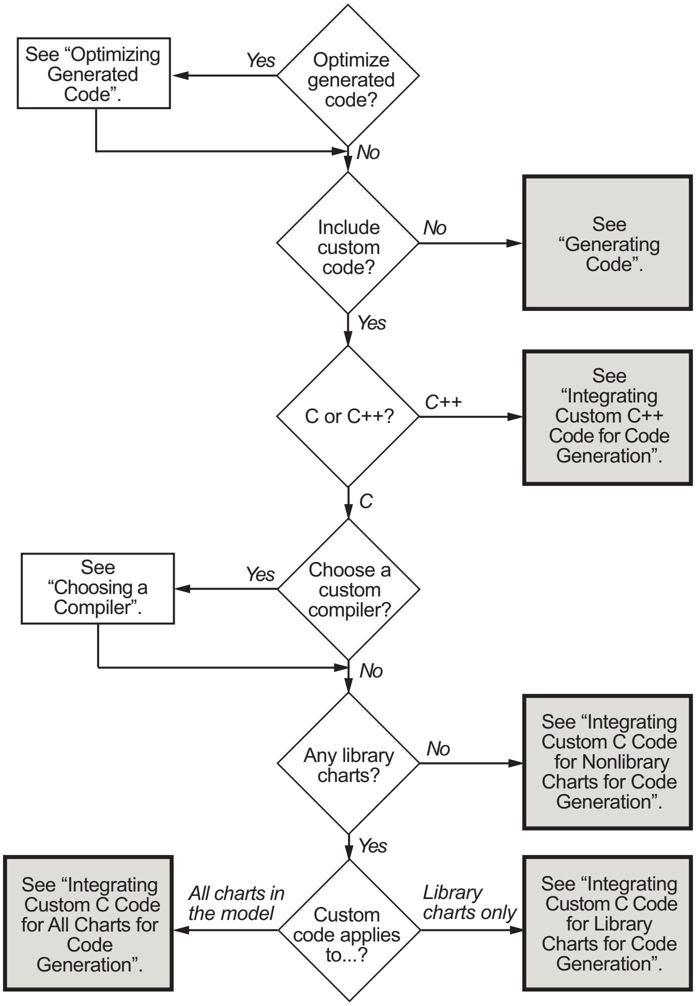
Guidelines for Embeddable Code Generation

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 25-29.
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 25-43.
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 25-21. 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 **Code Generation** pane of the Configuration Parameters dialog box.

See "Generated Code Files for Targets You Build" on page 25-74 for details about the embeddable code you generate for your model and the folder structure.

For information on setting code generation options using the command-line API, see "Using Command-Line API to Set Simulation and Code Generation Parameters" on page 25-31.

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 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 Code Generation

To include custom C++ code for Simulink Coder code generation, perform these steps:

1. Open the Configuration Parameters dialog box.
2. In the Configuration Parameters dialog box, select the **Code Generation** pane.
3. Select `C++` from the **Language** menu. Click **Apply**.
4. Select the **Code Generation > 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 25-43.

Task 4: Generate Code

For instructions, see "Generating Code" on page 25-21.

Integrating Custom C Code for Nonlibrary Charts for Code Generation

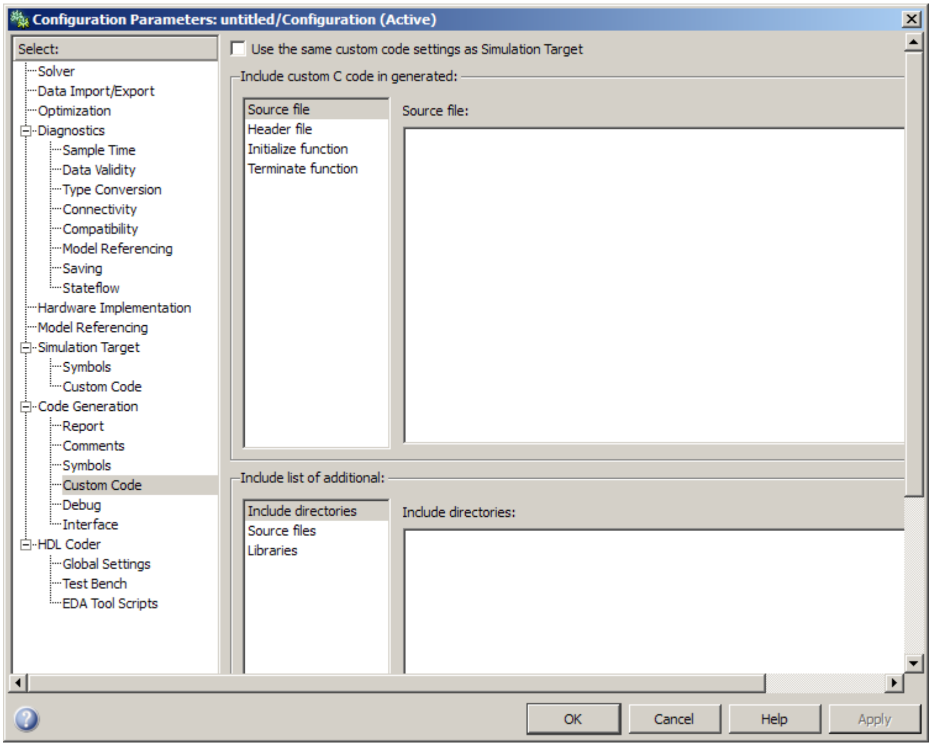
To integrate custom C code that applies to nonlibrary charts for embeddable code generation, perform the tasks that follow.

Task 1: Include Custom C Code for Embeddable Code Generation

Specify custom code options for Simulink Coder code generation of your model:

1. Open the Configuration Parameters dialog box.
2. In the Configuration Parameters dialog box, select **Code Generation > Custom Code**.

The custom code options appear.



☐ Use the same custom code settings as Simulation Target

3 Specify your custom code in the subpanes.

Follow the guidelines in “Specifying Relative Paths for Custom Code” on page 25-41.

Note If you specified custom code settings for simulation, you can apply these settings to 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 25-21.

Integrating Custom C Code for Library Charts for Code Generation

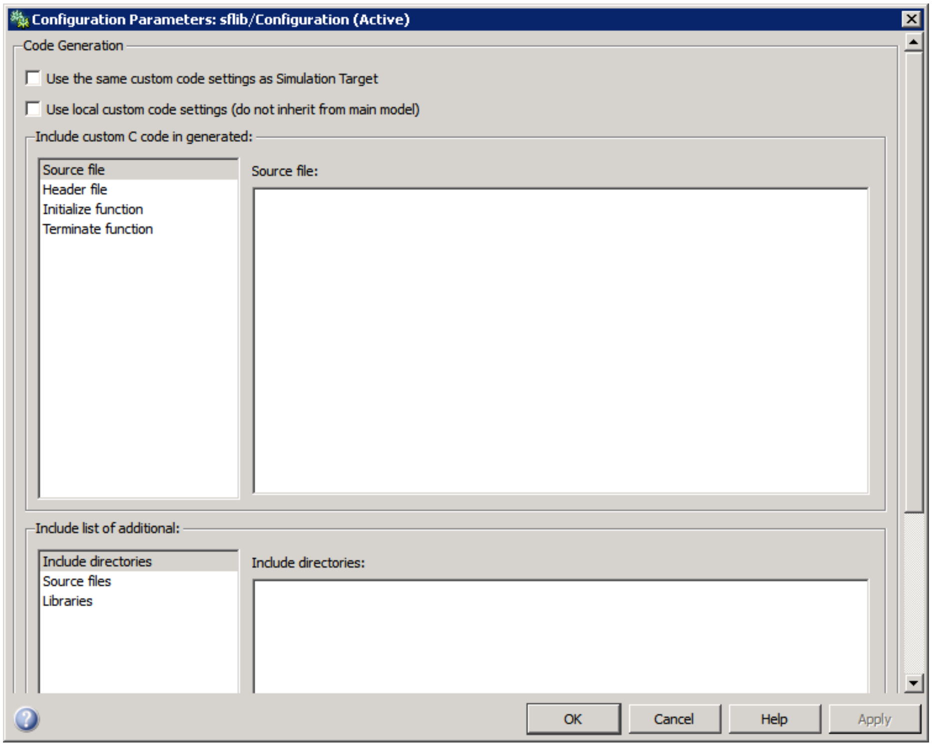
To integrate custom C code that applies only to library charts for embeddable code generation, perform the tasks that follow.

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 Code Generation Target**.

The Configuration Parameters dialog box appears.



Configuration Parameters: sflib/Configuration (Active)

Code Generation

☐ Use the same custom code settings as Simulation Target

☐ Use local custom code settings (do not inherit from main model)

Include custom C code in generated:

Source file:

Include list of additional:

Include directories:

2. In the **Code Generation** pane, select **Use local custom code settings (do not inherit from main model)**.

   This step ensures that each library model retains its own custom code settings during code generation.

3. Specify your custom code in the subpanes.

   Follow the guidelines in "Specifying Relative Paths for Custom Code" on page 25-41.

   Note If you specified custom code settings for simulation, you can apply these settings to 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 25-21.

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.

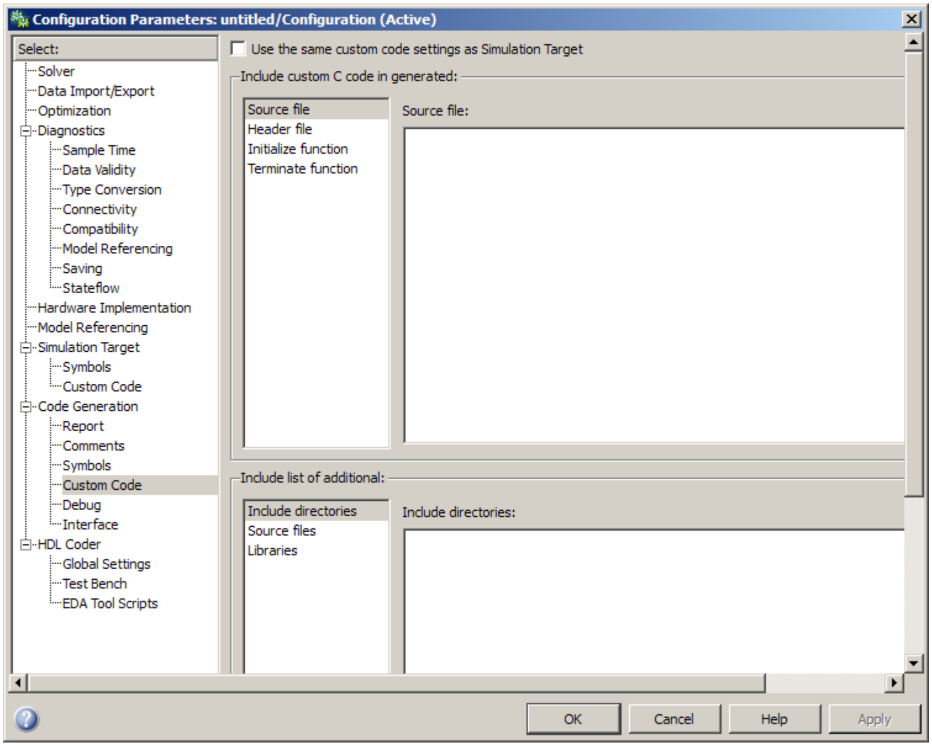
Task 1: Include Custom C Code for Embeddable Code Generation of the Main Model

Specify custom code options for Simulink Coder code generation of your main model:

1. Open the Configuration Parameters dialog box.

2. In the Configuration Parameters dialog box, select **Code Generation > Custom Code**.

The custom code options appear.



☐ Use the same custom code settings as Simulation Target

Source file:

Include directories:

3 Specify your custom code in the subpanes.

Follow the guidelines in "Specifying Relative Paths for Custom Code" on page 25-41.

Note If you specified custom code settings for simulation, you can apply these settings to 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 Code Generation Target**.

2. In the **Code Generation** pane, clear the **Use local custom code settings (do not inherit from main model)** check box.

   This step ensures that library charts inherit the custom code settings of your main model.

3. Click **OK**.

Task 3: Generate Code

For instructions, see "Generating Code" on page 25-21.

Optimizing Generated Code

In this section...

How to Optimize Generated Code for Embeddable Targets

To optimize code generation for your model:

1. Open the Configuration Parameters dialog box.

2. In the Configuration Parameters dialog box, select the **Optimization > Stateflow** pane.

3. 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 "Graphical Functions for Reusing Logic Patterns and Iterative Loops" on page 7-30.

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 7-47.

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 20, "Working with Structures and Bus Signals in Stateflow Charts".

Using Command-Line API to Set Simulation and Code Generation Parameters

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 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**	Enable 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)**	Enable 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
`SimIntegrity` string – `'off'`, **`'on'`**	**Simulation Target > Ensure memory integrity**	Detect violations of memory integrity in code generated for MATLAB Function blocks and stop execution with a diagnostic.
`SFSimEcho` string – `'off'`, **`'on'`**	**Simulation Target > Echo expressions without semicolons**	Enable run-time output to appear in the MATLAB Command Window during simulation.
`SimBlas` string – `'off'`, **`'on'`**	**Simulation Target > Use BLAS library for faster simulation**	Enable MATLAB Function blocks in Simulink models and MATLAB functions in Stateflow charts to speed up low-level matrix operations during simulation.
`SimCtrlC` string – `'off'`, **`'on'`**	**Simulation Target > Ensure responsiveness**	Enable responsiveness checks in code generated for MATLAB Function blocks.
`SimBuildMode` string – **`'sf_incremental_build'`**, `'sf_nonincremental_build'`, `'sf_make'`, `'sf_make_clean'`, `'sf_make_clean_objects'`	**Simulation Target > Simulation target build mode**	Specify 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.
`SimParseCustomCode` string – `'off'`, **`'on'`**	**Simulation Target > Custom Code > Parse custom code symbols**	Specify whether or not to parse the custom code and report unresolved symbols in a model.

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`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 folder paths that contain files you include in the compiled target. **Note** If your list includes any Windows path strings that contain spaces, each instance must be enclosed in double quotes within the argument string, for example, `'C:\Project "C:\Custom Files"'`

Parameter and Values	**Configuration Parameters Dialog Box Equivalent**	**Description**
`SimUserSources` *string* – `''`	**Simulation Target > Custom Code > Source files**	Enter a space-separated 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 folder paths that contain files you include in the compiled target. **Note** If your list includes any Windows path strings that contain spaces, each instance must be enclosed in double quotes within the argument string, for example, `'C:\Project "C:\Custom Files"'`
`SimUserSources` *string* – `''`	**Simulation Target > Source files**	Enter a space-separated 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'`	**Code Generation > 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 MATLAB Function blocks, Stateflow charts, or Truth Table blocks.)
`ReservedNameArray` *string array* – **`{}`**	**Code Generation > 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.
`RTWUseSimCustomCode` string – **`'off'`**, `'on'`	**Code Generation > 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 MATLAB Function blocks, Stateflow charts, or Truth Table blocks.)
`CustomSourceCode` *string* – `''`	**Code Generation > Custom Code > Source file**	Enter code lines to appear near the top of a generated source code file.
`CustomHeaderCode` *string* – `''`	**Code Generation > Custom Code > Header file**	Enter code lines to appear near the top of a generated header file.
`CustomInitializer` *string* – `''`	**Code Generation > Custom Code > Initialize function**	Enter code statements that execute once at the start of simulation.
`CustomTerminator` *string* – `''`	**Code Generation > Custom Code > Terminate function**	Enter code statements that execute at the end of simulation.

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`CustomInclude` *string* – ''	**Code Generation > Custom Code > Include directories**	Enter a space-separated list of folder paths that contain files you include in the compiled target. **Note** If your list includes any Windows path strings that contain spaces, each instance must be enclosed in double quotes within the argument string, for example, `'C:\Project "C:\Custom Files"'`
`CustomSource` *string* – ''	**Code Generation > Custom Code > Source files**	Enter a space-separated list of source files to compile and link into the target.
`CustomLibrary` *string* – ''	**Code Generation > 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'`	**Code Generation > 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 MATLAB Function blocks, Stateflow charts, or Truth Table blocks.)
`RTWUseLocalCustomCode` string – **`'off'`**, `'on'`	**Code Generation > 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* – `''`	**Code Generation > Source file**	Enter code lines to appear near the top of a generated source code file.
`CustomHeaderCode` *string* – `''`	**Code Generation > Header file**	Enter code lines to appear near the top of a generated header file.
`CustomInitializer` *string* – `''`	**Code Generation > Initialize function**	Enter code statements that execute once at the start of simulation.
`CustomTerminator` *string* – `''`	**Code Generation > Terminate function**	Enter code statements that execute at the end of simulation.

Parameter and Values	Configuration Parameters Dialog Box Equivalent	Description
`CustomInclude` *string* – `''`	**Code Generation > Include directories**	Enter a space-separated list of folder paths that contain files you include in the compiled target. **Note** If your list includes any Windows path strings that contain spaces, each instance must be enclosed in double quotes within the argument string, for example, `'C:\Project "C:\Custom Files"'`
`CustomSource` *string* – `''`	**Code Generation > Source files**	Enter a space-separated list of source files to compile and link into the target.
`CustomLibrary` *string* – `''`	**Code Generation > 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 Simulink Coder 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 folders:

- The current folder
- The model folder (if different from the current folder)
- The custom list of folders that you specify
- All the folders on the MATLAB search path, excluding the toolbox folders

Path Syntax Rules

When you construct relative paths for custom code, follow these syntax rules:

- You can use the forward slash (`/`) or backward slash (`\`) as a file separator, regardless of whether you are on a UNIX or PC platform. The makefile generator 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 folder `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 a list of supported compilers, see:

`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.
```

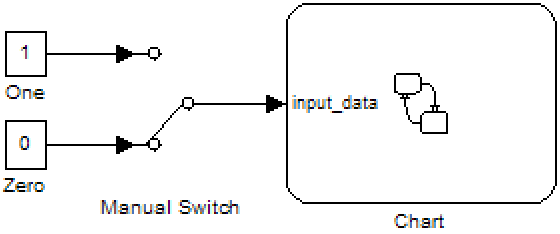
Examples of Integrating Custom C Code in Nonlibrary Models

In this section...
"Example of Using Custom C Code to Define Global Constants" on page 25-44
"Example of Using Custom C Code to Define Global Constants, Variables, and Functions" on page 25-47

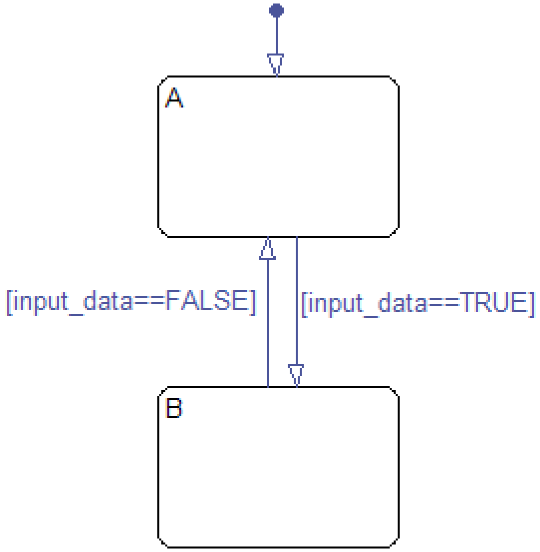
Example of Using Custom C Code to Define Global Constants

This example describes how to use custom C code to define constants that apply to all charts in your model.

1 Suppose that you have the following model:



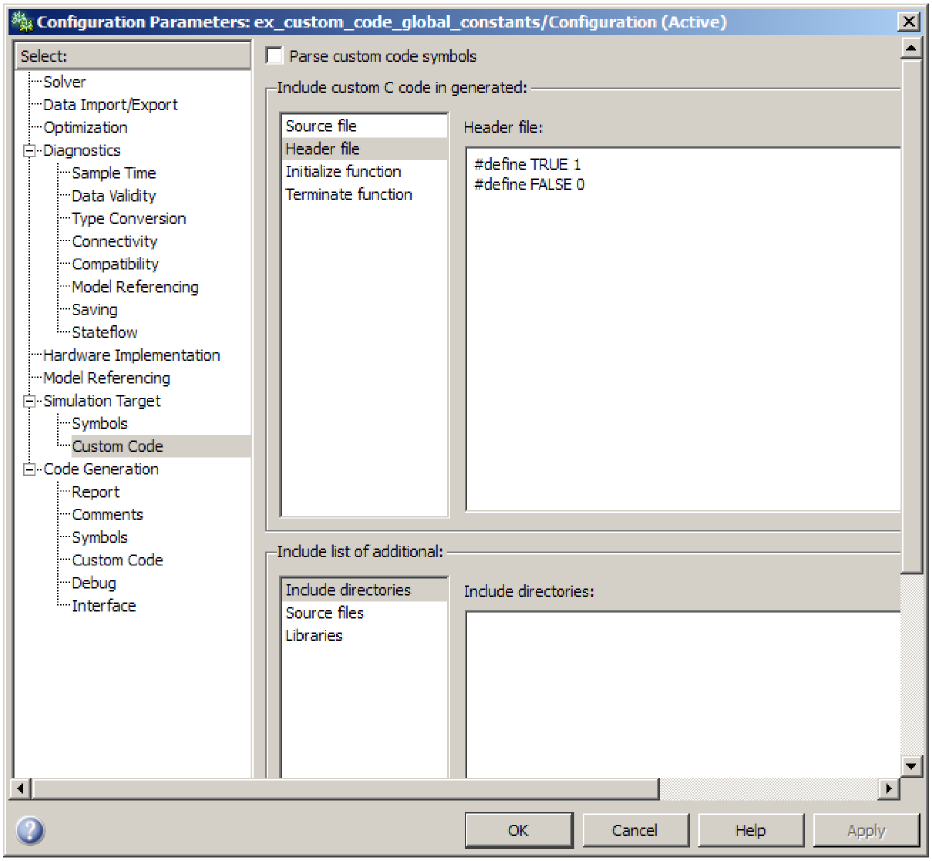
The chart contains the following logic:



The chart contains two states `A` and `B`, along with a Simulink input named `input_data`, which you can set to 0 or 1 by toggling the Manual Switch in the model during simulation.

2. Open the Configuration Parameters dialog box.
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.



Configuration Parameters: ex_custom_code_global_constants/Configuration (Active)

☐ Parse custom code symbols

Include custom C code in generated:

- Source file
- Header file
- Initialize function
- Terminate function

Header file:

```
#define TRUE 1
#define FALSE 0
```

Include list of additional:

- Include directories
- Source files
- Libraries

Include directories:

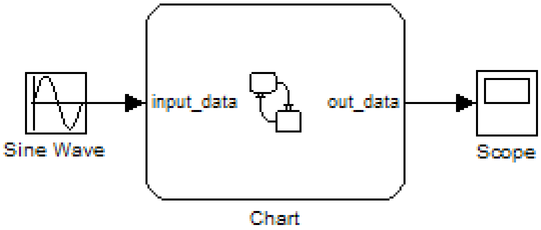
In this example, you define two constants named `TRUE` and `FALSE` to move between states in your chart, instead of using the values 1 and 0. These custom definitions improve the readability of your chart actions. Note that `TRUE` and `FALSE` are not Stateflow data objects.

Because the two custom definitions appear at the top of your generated machine header file `ex_custom_code_global_constants_sfun.h`, you can use `TRUE` and `FALSE` in all charts that belong to this model. For more information about generated files, see "Code Files for a Simulation Target" on page 25-76.

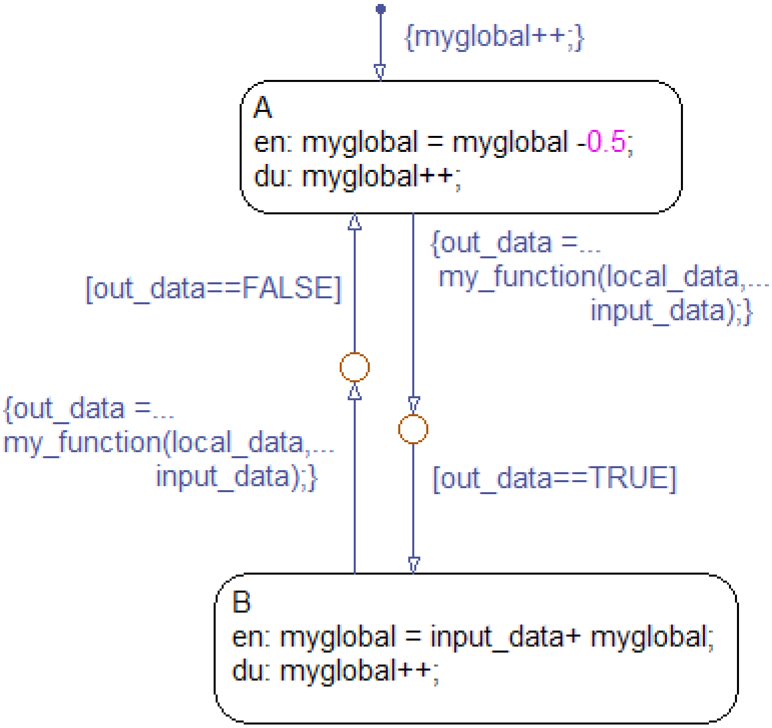
Example of Using Custom C Code to Define Global Constants, Variables, and Functions

This example describes how to use custom C code to define constants, variables, and functions that apply to all charts in your model.

1 Suppose that you have the following model:



The chart contains the following logic:



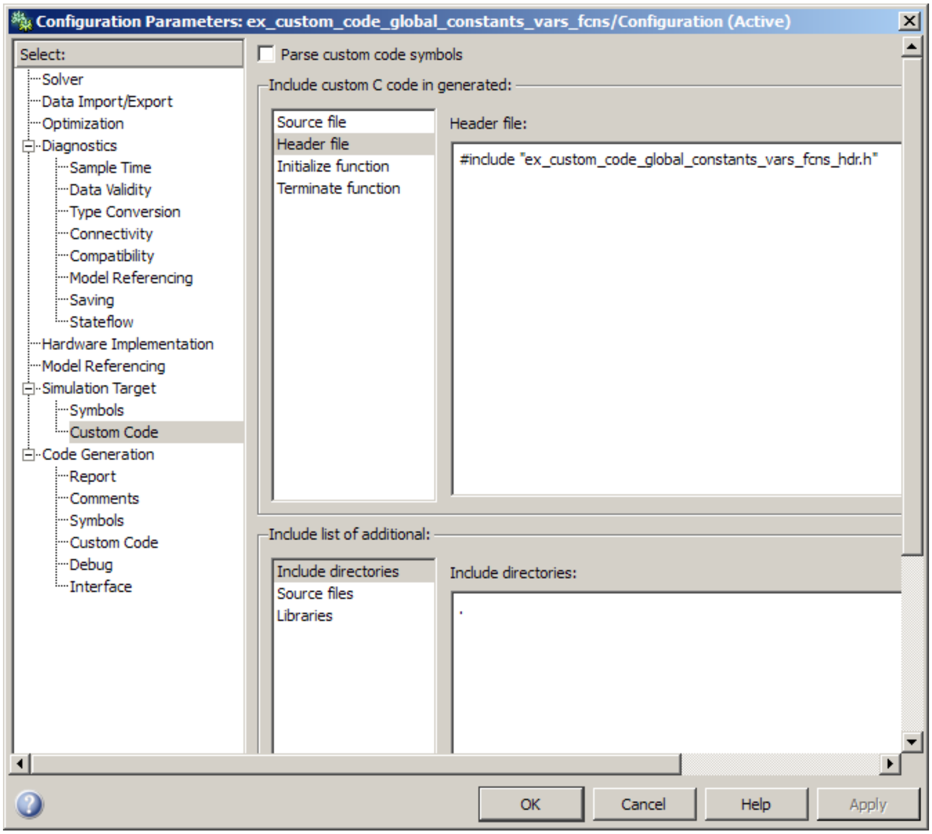
The chart contains two states `A` and `B`, along with three data objects: `input_data`, `local_data`, and `out_data`. The chart accesses a custom variable named `myglobal` and calls a custom function named `my_function`.

2 Open the Configuration Parameters dialog box.

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.



Configuration Parameters: ex_custom_code_global_constants_vars_fcns/Configuration (Active)

☐ Parse custom code symbols

Include custom C code in generated:

Header file:

```
#include "ex_custom_code_global_constants_vars_fcns_hdr.h"
```

Include list of additional:

Include directories:

```
.
```

Note When you include a custom header file, you must enclose the file name in double quotes.

The custom header file `ex_custom_code_global_constants_vars_fcns_hdr.h` contains the definitions of three constants:

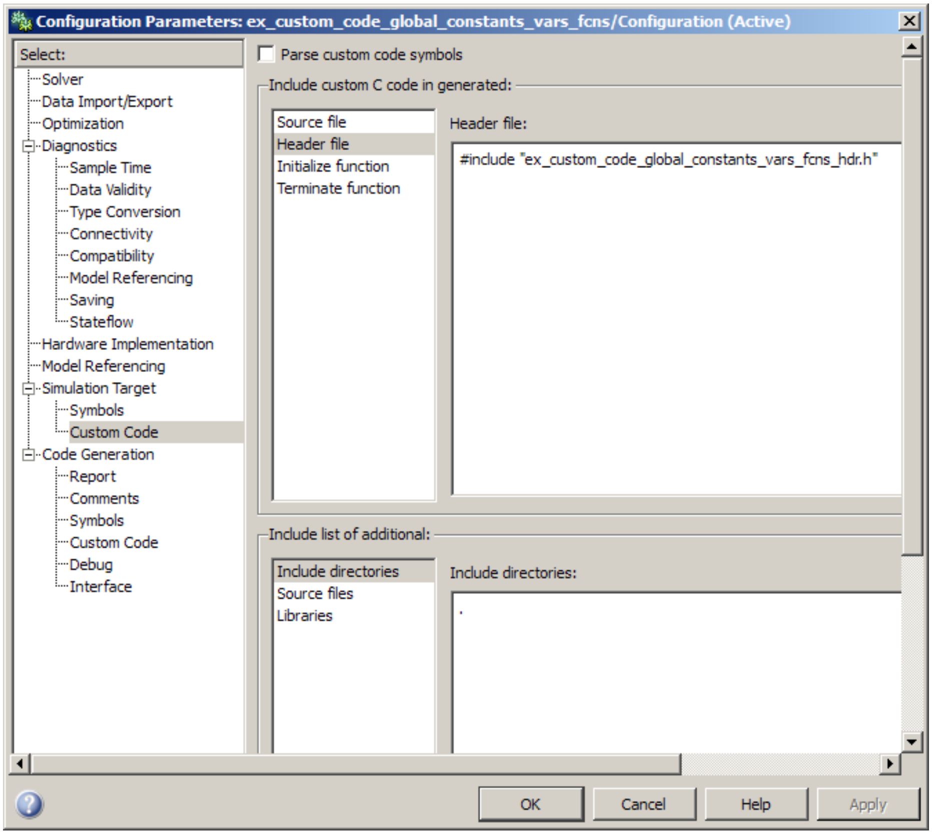
```
#define TRUE 1
#define FALSE 0
```

```
#define MAYBE 2
```

This header file also contains declarations for the variable `myglobal` and the function `my_function`:

```
extern int myglobal;
extern int my_function(int var1, double var2);
```

5 Select the **Include directories** subpane.



☐ Parse custom code symbols

Header file:

```
#include "ex_custom_code_global_constants_vars_fcns_hdr.h"
```

Include directories:

```
.
```

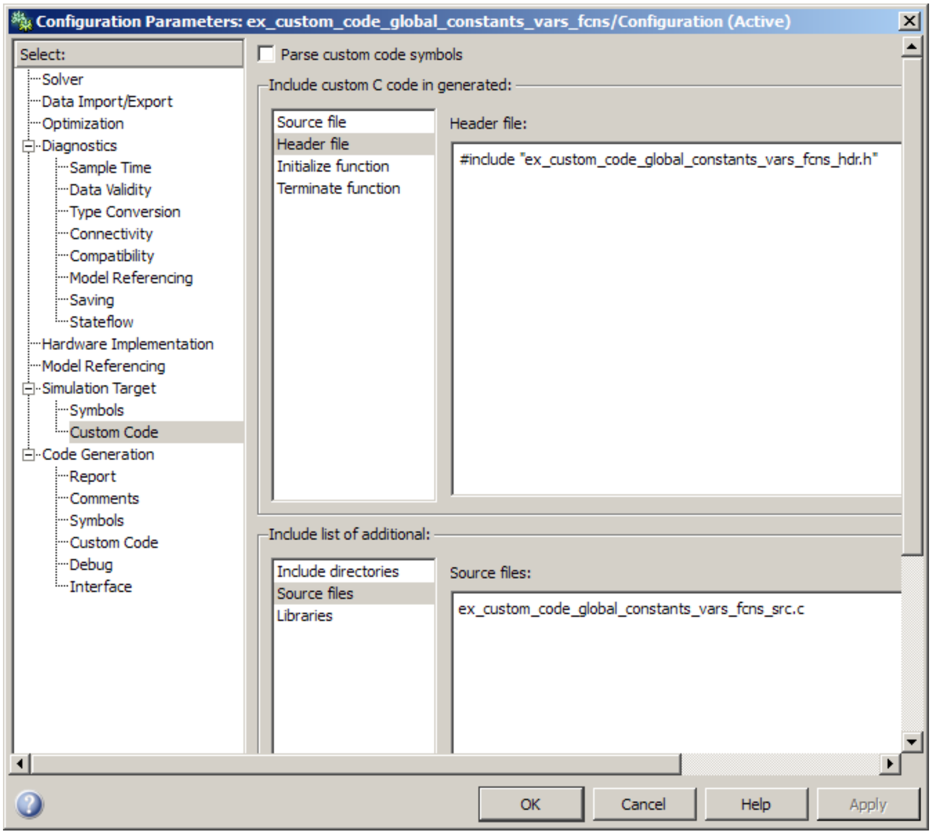
The single period `(.)` indicates that all your custom code files reside in the same folder as `ex_custom_code_global_constants_vars_fcns.mdl`.

Tip To direct your makefile to look for header or source files in a subfolder relative to the model folder, use this relative path name:

```
.\subfolder_name
```

---

6 Select the **Source files** subpane.



Configuration Parameters: ex_custom_code_global_constants_vars_fcns/Configuration (Active)

☐ Parse custom code symbols

Include custom C code in generated:

Header file:

```
#include "ex_custom_code_global_constants_vars_fcns_hdr.h"
```

Include list of additional:

Source files:

```
ex_custom_code_global_constants_vars_fcns_src.c
```

The custom source file `ex_custom_code_global_constants_vars_fcns_src.c` compiles along with the Stateflow generated code into a single S-function MEX file. See "S-Function MEX-Files" on page 25-74 for details.

Tip To include a source file that resides in a subfolder relative to the model folder, use this relative path name:

```
.\subfolder_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 `ex_custom_code_global_constants_vars_fcns_sfun.h`, you can access them in all charts that belong to this model. For more information about generated files, see "Code Files for a Simulation Target" on page 25-76.

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 Simulink Coder code generation software to build a custom target.

For information on setting custom target options programmatically, see "Target Properties" and "Target Methods" in the *Stateflow API*.

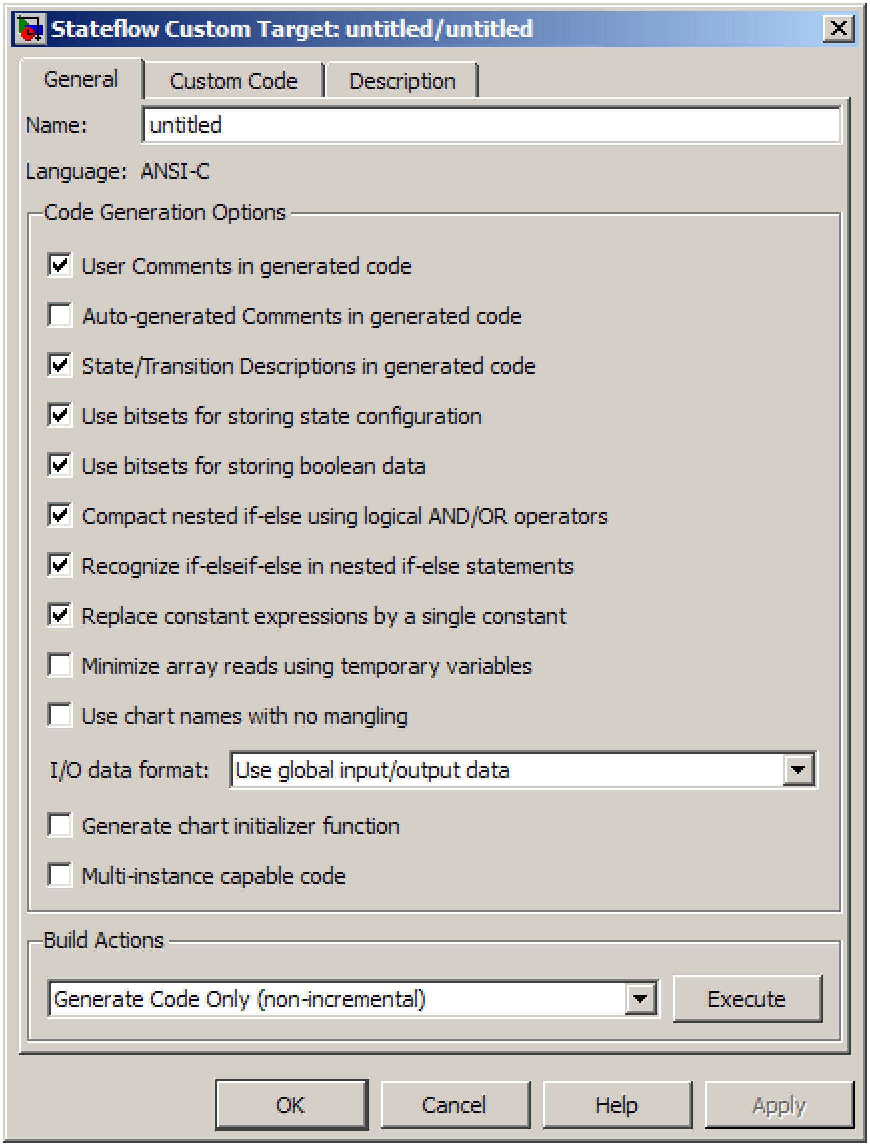
Adding a Stateflow Custom Target to Your Model

To add a custom target to your model:

1. In the Model Explorer, select **Add > Stateflow Target**.

2. In the **Contents** pane of the Model Explorer, right-click the row of the custom target and select **Properties**.

The Stateflow Custom Target dialog box appears.



Stateflow Custom Target: untitled/untitled

Name: untitled

Language: ANSI-C

Code Generation Options

☑ User Comments in generated code

☐ Auto-generated Comments in generated code

☑ State/Transition Descriptions in generated code

☑ Use bitsets for storing state configuration

☑ Use bitsets for storing boolean data

☑ Compact nested if-else using logical AND/OR operators

☑ Recognize if-elseif-else in nested if-else statements

☑ Replace constant expressions by a single constant

☐ Minimize array reads using temporary variables

☐ Use chart names with no mangling

I/O data format: Use global input/output data

☐ Generate chart initializer function

☐ Multi-instance capable code

Build Actions

Generate Code Only (non-incremental)

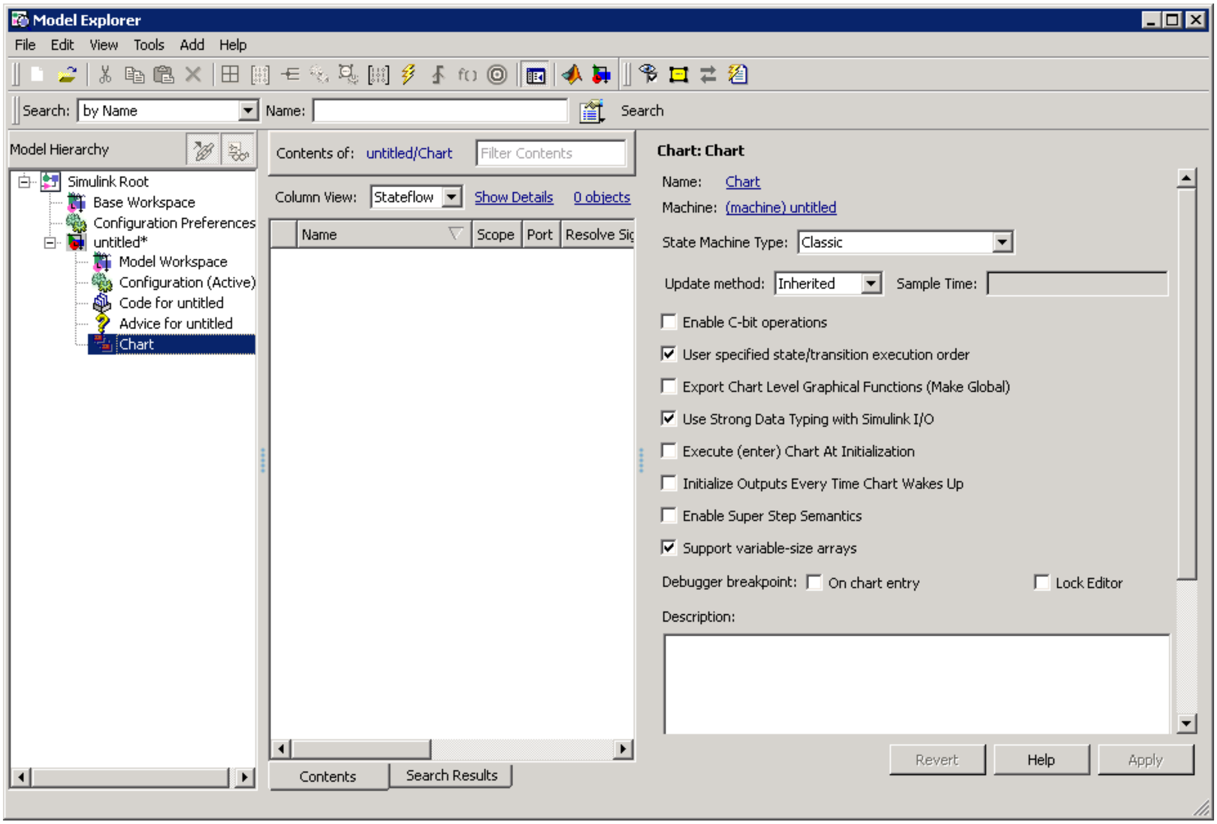
3 In the **Name** field, enter any name except the reserved names `sfun` and `rtw`. Then click **OK**.

Configuring a Custom Target

To configure a custom target:

1 Open the Model Explorer.

The chart appears highlighted in the **Model Hierarchy** pane.



Search: by Name
Name:

Model Hierarchy
- Simulink Root
  - Base Workspace
  - Configuration Preferences
  - untitled*
    - Model Workspace
    - Configuration (Active)
    - Code for untitled
    - Advice for untitled
    - Chart

Contents of: untitled/Chart
Filter Contents
Column View: Stateflow
Show Details
0 objects

Name	Scope	Port	Resolve Sig

Chart: Chart

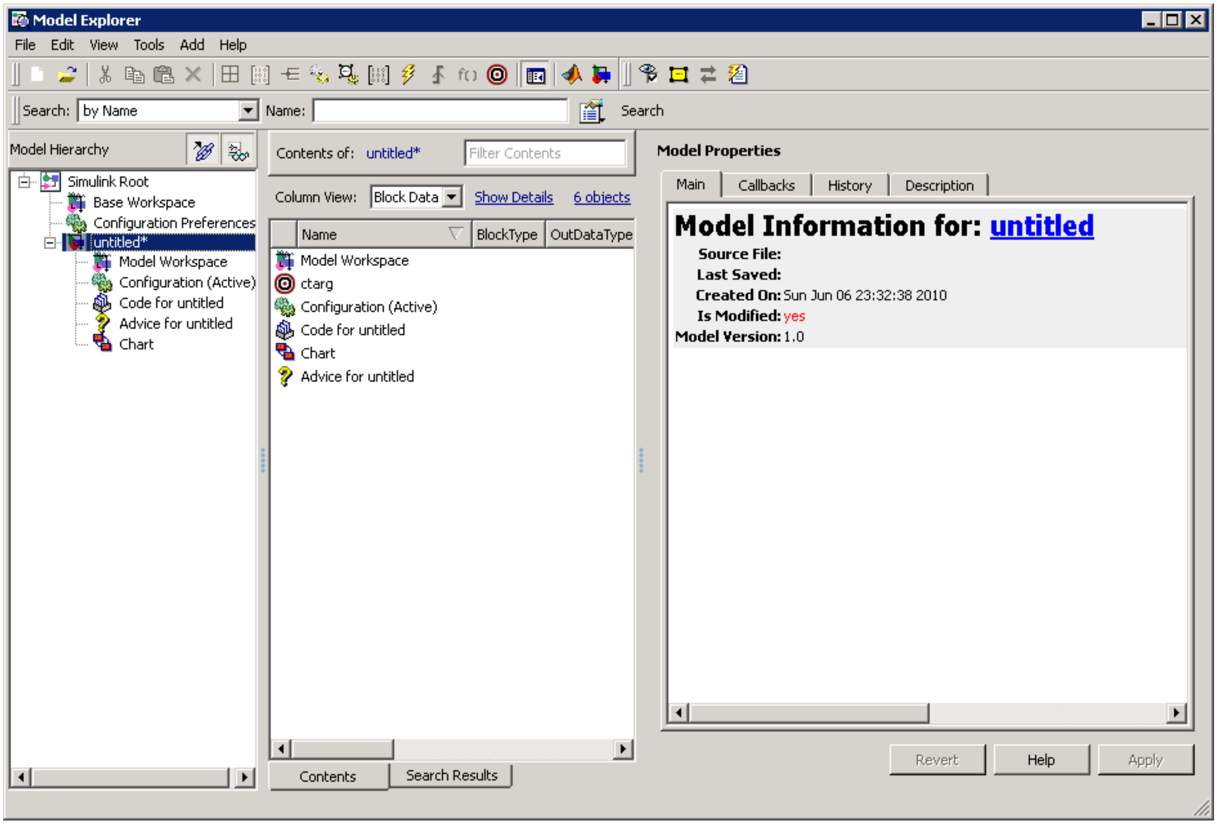
Name: Chart
Machine: (machine) untitled
State Machine Type: Classic
Update method: Inherited
Sample Time:

☐ Enable C-bit operations
☑ 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
☑ Support variable-size arrays

Debugger breakpoint: ☐ On chart entry
☐ Lock Editor

Description:

2 In the **Model Hierarchy** pane, select the main model with the custom target.



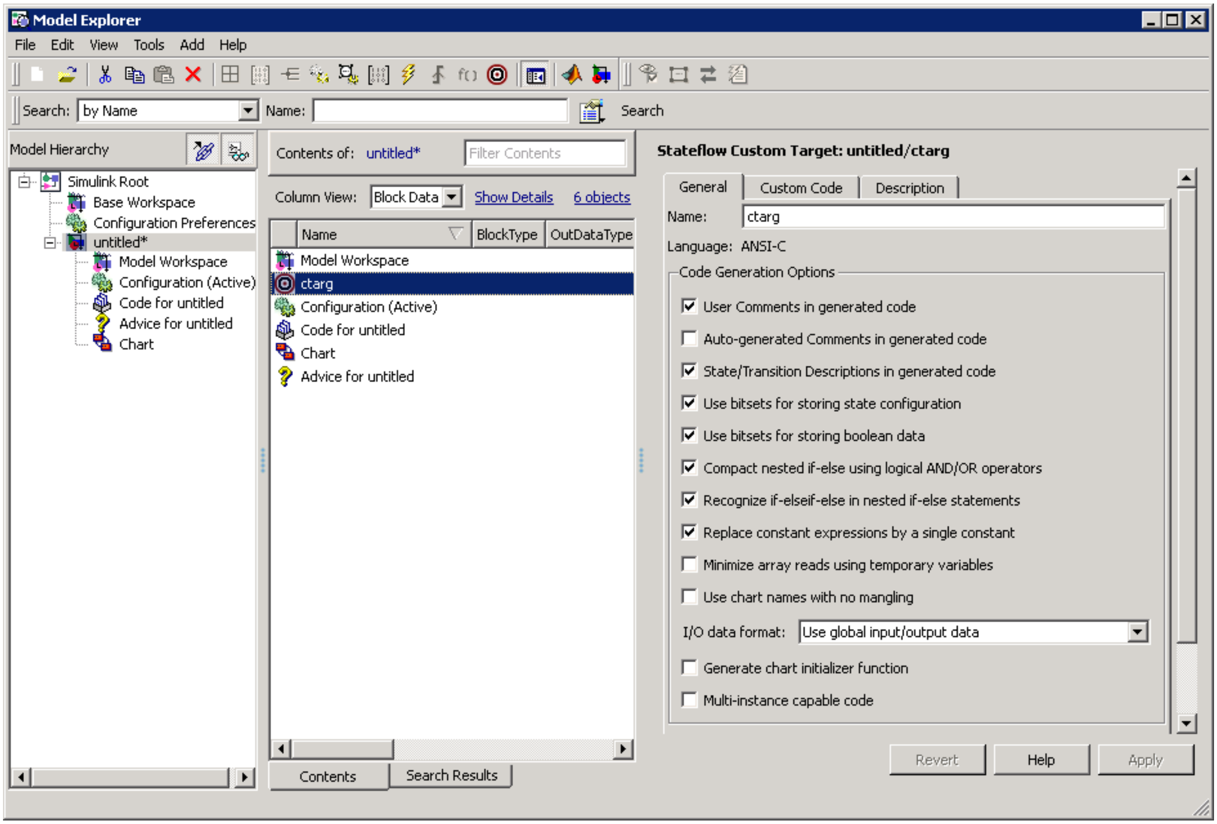
Model Information for: untitled

Source File:
Last Saved:
Created On: Sun Jun 06 23:32:38 2010
Is Modified: yes
Model Version: 1.0

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 pane on the right.



Stateflow Custom Target: untitled/ctarg

General | Custom Code | Description

Name: ctarg

Language: ANSI-C

Code Generation Options

☑ User Comments in generated code

☐ Auto-generated Comments in generated code

☑ State/Transition Descriptions in generated code

☑ Use bitsets for storing state configuration

☑ Use bitsets for storing boolean data

☑ Compact nested if-else using logical AND/OR operators

☑ Recognize if-elseif-else in nested if-else statements

☑ Replace constant expressions by a single constant

☐ Minimize array reads using temporary variables

☐ Use chart names with no mangling

I/O data format: Use global input/output data

☐ Generate chart initializer function

☐ Multi-instance capable code

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 the following 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. See “Generated Code Files for Targets You Build” on page 25-74 for details about the code you generate for this target and the folder structure.

Restrictions on Building a Custom Target

You cannot build a custom target if your chart contains any of the following items:

- Enumerated data (see Chapter 15, “Using Enumerated Data in Stateflow Charts”)
- Atomic subcharts (see Chapter 11, “Making States Reusable with Atomic Subcharts”)

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 25-64 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 chart notation and the action language syntax.

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 chart 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 chart to the front with the source object highlighted.

Parser Error Checking

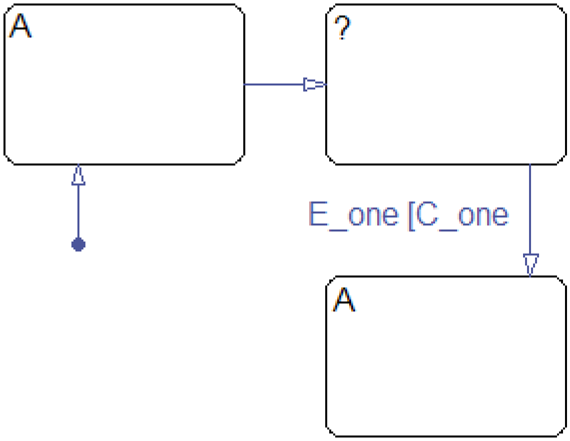
Using the Debugger, you can detect the following errors during simulation:

- State Inconsistency — Most commonly caused by the omission of a default transition to a substate in superstates with exclusive (OR) decomposition. See "State Inconsistencies in a Chart" on page 26-27.
- Transition Conflict — Occurs when there are two equally valid transition paths from the same source. See "Conflicting Transitions in a Chart" on page 26-29.
- 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 "Data Range Violations in a Chart" on page 26-31.
- Cyclic Behavior — Occurs when a step or sequence of steps repeats itself indefinitely. See "Cyclic Behavior in a Chart" on page 26-32.

You can modify the notation to resolve run-time errors. See Chapter 26, "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.

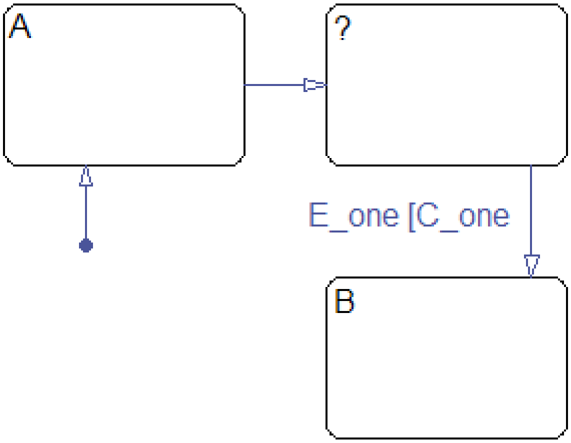


1. In the Stateflow Editor, select **Tools > Parse Diagram** to parse the chart.

   State `A` appears highlighted in the chart and a parsing error message indicates that the name A is not unique.

2. Fix the parse error.

   In this example, two states with the name `A` exist. Edit the chart and label the duplicate state with the text `B`.

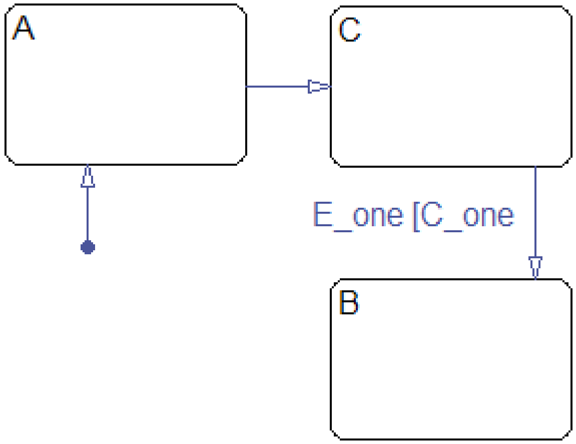


3. In the Stateflow Editor, select **Tools > Parse Diagram** to reparse the chart.

   State `?` appears highlighted in the chart and a parsing error message indicates that the name ? is invalid.

4. Fix the parse error.

   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`.

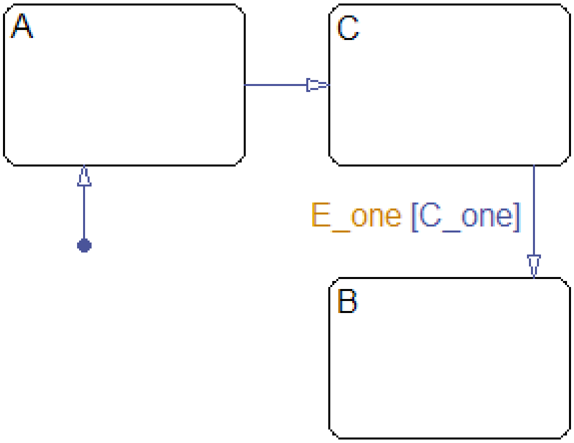


5 In the Stateflow Editor, select **Tools > Parse Diagram** to reparse the chart.

The transition for `E_one [C_one` appears highlighted in the chart and a parsing error message indicates that the transition label contains a syntax error.

6 Fix the parse error.

The closing bracket of the condition is missing on the transition label. 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.

The chart now has no parse errors.

Resolving Undefined Symbols in Your Chart

In this section...

How to Check for Undefined Symbols

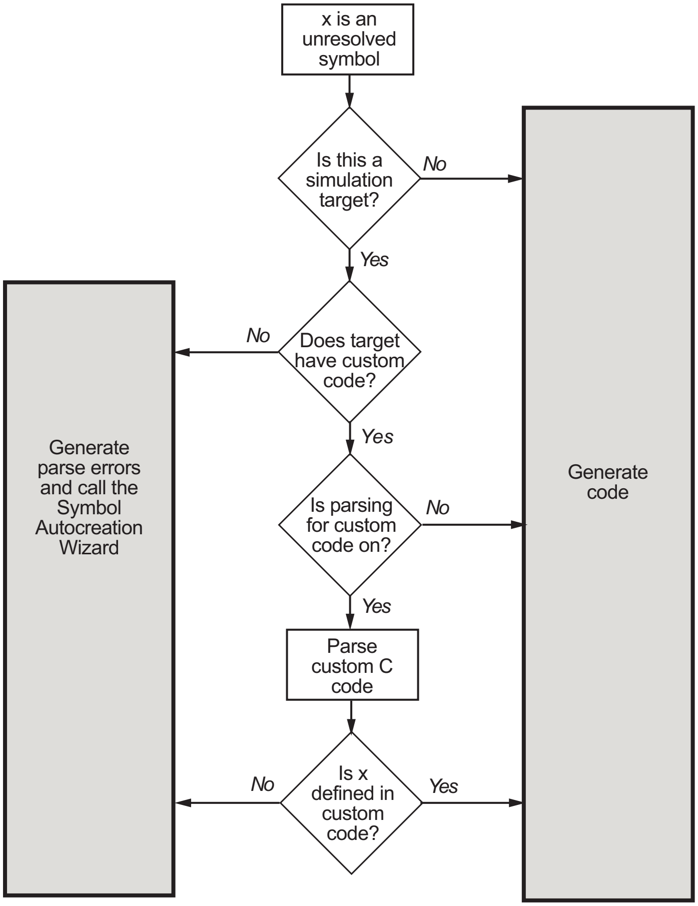
To check for undefined symbol errors, you can use one of these methods:

- 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)

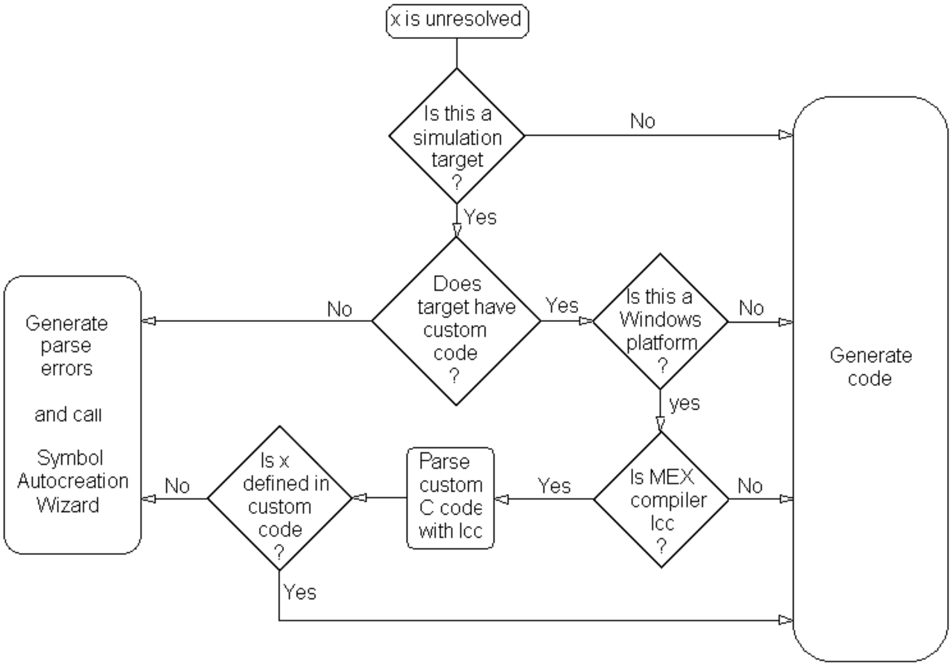
Each method triggers parsing of the Stateflow machine (see "Parsing Stateflow Charts" on page 25-64). The parser behaves differently depending on the setting of **Parse custom code symbols** in the **Simulation Target > Custom Code** pane of the Configuration Parameters dialog box.

- If you select this check box, the parser tries to find unresolved chart symbols in the custom code. If the custom code does not define these symbols, a parse error appears.
- If you do not select this check box, the parser automatically assumes that unresolved chart symbols are defined in the custom code. If the custom code does not define these symbols, an error does not appear until make time.

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.


x is an unresolved symbol
Is this a simulation target?
No
Yes
Does target have custom code?
No
Yes
Is parsing for custom code on?
No
Yes
Parse custom C code
Is x defined in custom code?
No
Yes
Generate parse errors and call the Symbol Autocreation Wizard
Generate code

Note In versions R2010a and earlier, the following process for resolving symbols applies:

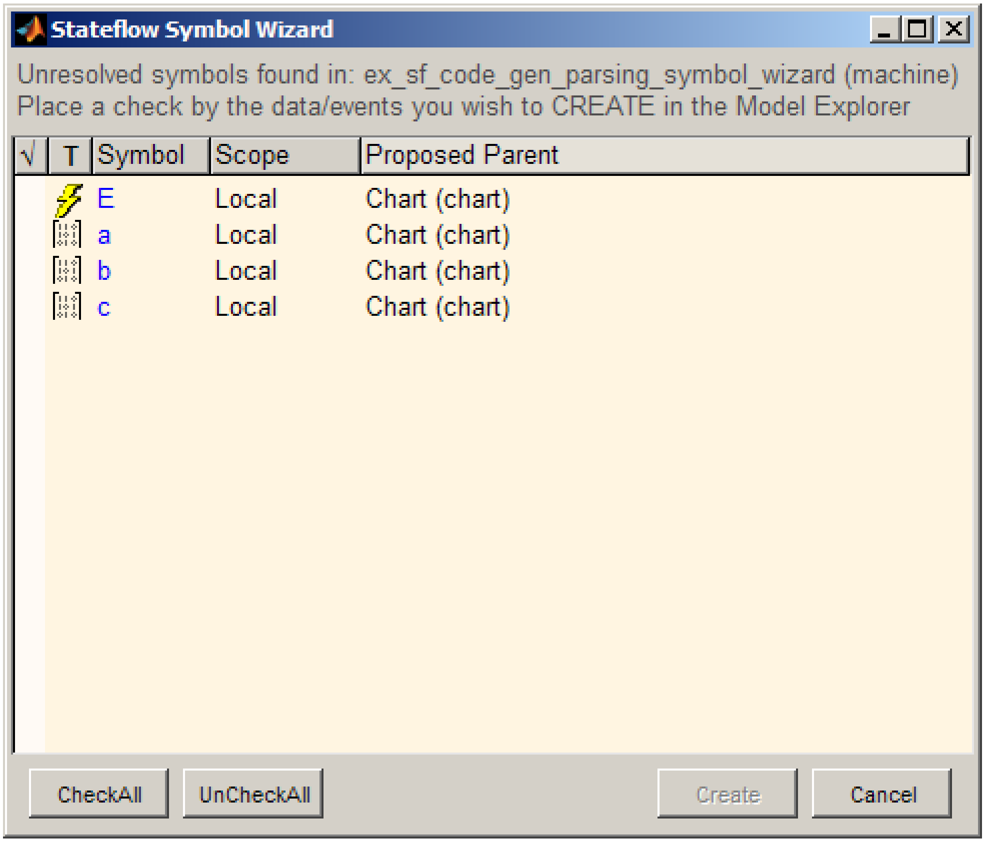


When you parse a chart without simulation or diagram updates, the Stateflow parser does not have access to all the information needed to check for unresolved symbols, such as exported graphical functions from other charts and enumerated data types. However, if you start simulation or update the model 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.

Using the Symbol Wizard to Define Chart Symbols

You can use the Symbol Wizard to add missing data and events to your chart. When you start simulation or update the model diagram, the Wizard detects references to undefined data and events and presents a list of data or events that you must define.



Unresolved symbols found in: ex_sf_code_gen_parsing_symbol_wizard (machine)
Place a check by the data/events you wish to CREATE in the Model Explorer

√	T	Symbol	Scope	Proposed Parent
	⚡	E	Local	Chart (chart)
	▦	a	Local	Chart (chart)
	▦	b	Local	Chart (chart)
	▦	c	Local	Chart (chart)

To accept, reject, or change a recommended item, perform 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.

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 folder. You can change this location at the MATLAB command prompt with a `cd` command.

Folder Structure of Generated Files

Most of the code files that you generate reside in a subfolder of the current MATLAB folder. This table summarizes the default folder structure for different targets.

Target Type	**Model Type**	**Folder Under** `<pwd>/slprj/_sfprj/<mainModel>`
Simulation	Main (nonlibrary)	`/_self/sfun/src`
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:

- `<pwd>` is the current working folder.
- `<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.

For embeddable targets, the integrated C code for the entire model resides in the subfolder `<mainModel>_<sys_targ>_rtw` of the current MATLAB folder. The executable file generated for the entire model resides in the current MATLAB folder.

Tip To use a root folder different from `<pwd>` for storing generated files, open the Simulink Preferences Window and update the **File generation control** section.

- For simulation targets, specify **Simulation cache folder**.
- For embeddable targets, specify **Code generation folder**.

For more information, see "File generation control" in the Simulink documentation.

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 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 folder as the source code. The file extension depends on the platform. On a Windows operating system, the library file is `<libModel>_sfun.lib`, but on a UNIX operating system, the library file 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 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 a list of supported compilers, see:

`http://www.mathworks.com/support/compilers/current_release/`

Traceability of Stateflow Objects in 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 "Code Tracing" in the Embedded Coder documentation for information about how traceability works for Simulink blocks.

Traceability Requirements

To enable traceability comments, you must have a license for 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
- MATLAB functions

Note Traceability is not supported for external code that you call from a 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 MATLAB Function blocks that you insert directly in a Simulink model are also traceable. For more information, see "Using Traceability in MATLAB Function Blocks" in the Simulink 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. Open the Configuration Parameters dialog box.
3. In the **Code Generation** 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 **Code Generation > Report** pane, select **Create code generation report**.

   This step automatically selects **Open report automatically** and **Code-to-model**.

5. Select **Model-to-code** in the **Navigation** section. Then click **Apply**.

   This step automatically selects all check boxes in the **Traceability Report Contents** section.

Tip For large models that contain over 1000 blocks, clear the **Model-to-code** check box to speed up code generation.

---

6. Go to the **Code Generation > 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.

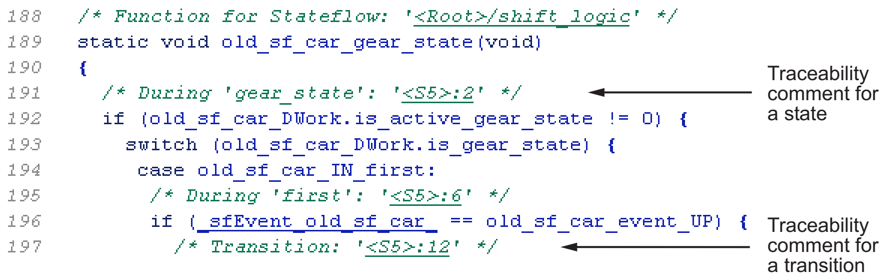
---

7. In the **Code Generation** pane, click **Build** in the lower right corner.

   This step 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.



```
188   /* Function for Stateflow: '<Root>/shift_logic' */
189   static void old_sf_car_gear_state(void)
190   {
191     /* During 'gear_state': '<S5>:2' */
192     if (old_sf_car_DWork.is_active_gear_state != 0) {
193       switch (old_sf_car_DWork.is_gear_state) {
194        case old_sf_car_IN_first:
195         /* During 'first': '<S5>:6' */
196         if (_sfEvent_old_sf_car_ == old_sf_car_event_UP) {
197           /* 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' */
```

The corresponding state appears highlighted in the chart.

11 Click the `<S5>:12` hyperlink in this traceability comment:

```
/* Transition: '<S5>:12' */
```

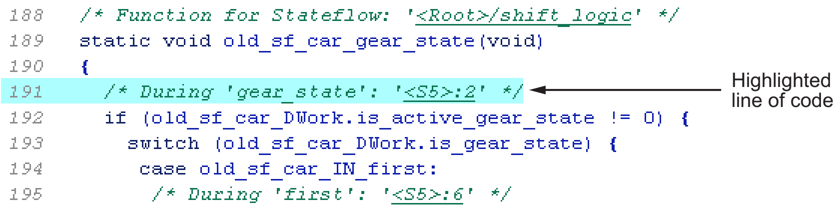
The corresponding transition appears highlighted in the chart.

Tip To remove highlighting from an object in the chart, select **View > Remove Highlighting**.

---

12 You can also trace an object in the model to a line of generated code. In the chart, right-click the object `gear_state` and select **Code Generation > Navigate to Code**.

The code for that state appears highlighted in `old_sf_car.c`.

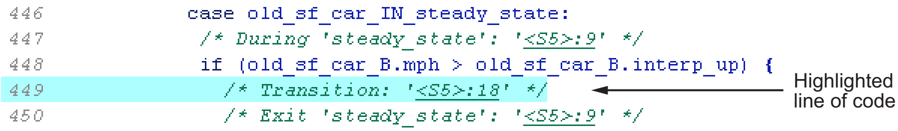


```
/* Function for Stateflow: '<Root>/shift_logic' */
static void old_sf_car_gear_state(void)
{
  /* During 'gear_state': '<S5>:2' */
  if (old_sf_car_DWork.is_active_gear_state != 0) {
    switch (old_sf_car_DWork.is_gear_state) {
     case old_sf_car_IN_first:
      /* During 'first': '<S5>:6' */
```

Highlighted line of code

13 In the chart, right-click the transition with the condition `[speed > up_th]` and select **Code Generation > Navigate to Code**.

The code for that transition appears highlighted in `old_sf_car.c`.



```
         case old_sf_car_IN_steady_state:
          /* During 'steady_state': '<S5>:9' */
          if (old_sf_car_B.mph > old_sf_car_B.interp_up) {
            /* Transition: '<S5>:18' */
            /* Exit 'steady_state': '<S5>:9' */
```

Highlighted line of code

Note For a list of all Stateflow objects in your model that are traceable, click the `Traceability Report` hyperlink in 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.

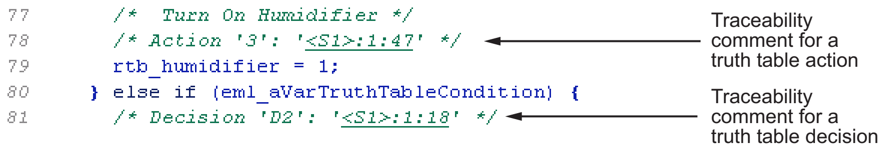
2 Complete steps 2 through 5 in "Bidirectional Traceability for States and Transitions" on page 25-81.

3 In the **Code Generation** 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.



```
77          /*  Turn On Humidifier */
78          /* Action '3': '<S1>:1:47' */
79          rtb_humidifier = 1;
80        } else if (eml_aVarTruthTableCondition) {
81          /* Decision 'D2': '<S1>:1:18' */
```

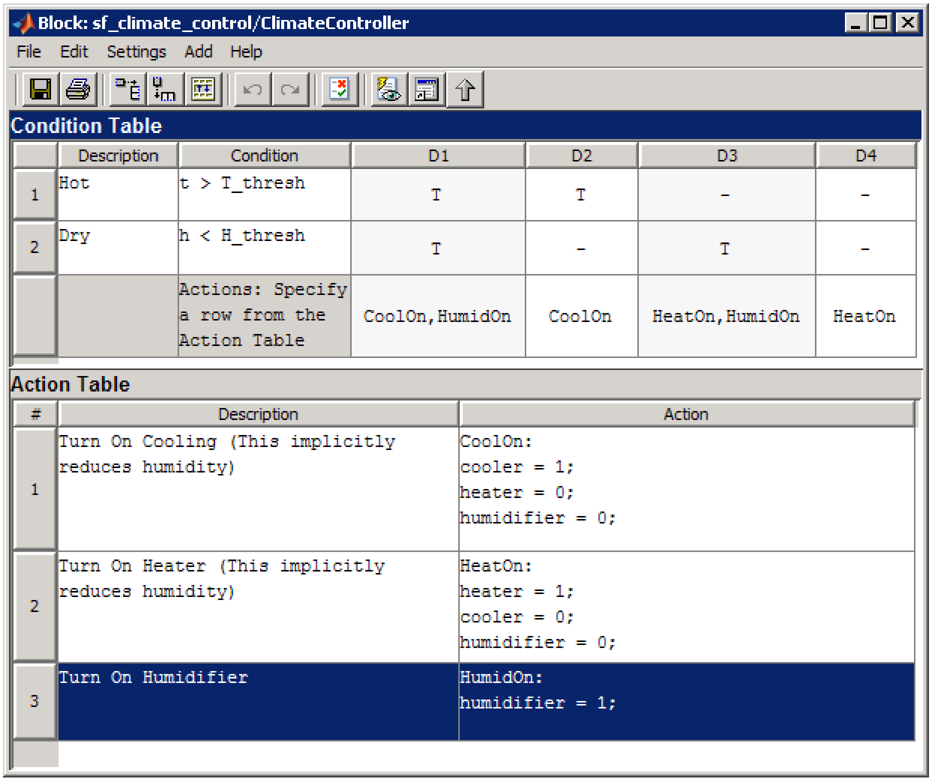
Note The line numbers shown above can differ from the numbers that appear in your code generation report.

---

6 Click the `<S1>:1:47` hyperlink in this traceability comment:

```
/* Action '3': '<S1>:1:47' */
```

In the Truth Table Editor, row 3 of the Action Table appears highlighted.



Condition Table

	Description	Condition	D1	D2	D3	D4
1	Hot	t > T_thresh	T	T	-	-
2	Dry	h < H_thresh	T	-	T	-
		Actions: Specify a row from the Action Table	CoolOn,HumidOn	CoolOn	HeatOn,HumidOn	HeatOn

Action Table

#	Description	Action

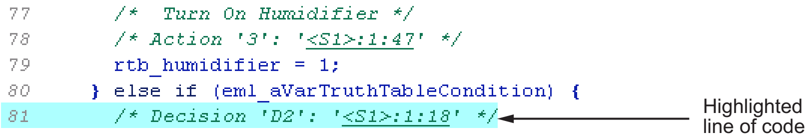
1	Turn On Cooling (This implicitly reduces humidity)	CoolOn:
		cooler = 1;
		heater = 0;
		humidifier = 0;

2	Turn On Heater (This implicitly reduces humidity)	HeatOn:
		heater = 1;
		cooler = 0;
		humidifier = 0;

3	Turn On Humidifier	HumidOn: humidifier = 1;

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 **Code Generation > Navigate to Code**.

The code for that decision appears highlighted in `sf_climate_control.c`.


```
    /*  Turn On Humidifier */
    /* Action '3': '<S1>:1:47' */
    rtb_humidifier = 1;
  } else if (eml_aVarTruthTableCondition) {
    /* Decision 'D2': '<S1>:1:18' */
```

Tip To select **Code Generation > 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 through 6 in "Bidirectional Traceability for States and Transitions" on page 25-81.
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**.

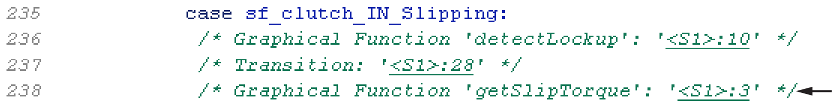
Note Because this demo model does not work with variable-step solvers, you must perform this step before generating code.

---

4. In the **Code Generation** 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.



```
          case sf_clutch_IN_Slipping:
           /* Graphical Function 'detectLockup': '<S1>:10' */
           /* Transition: '<S1>:28' */
           /* 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.

---

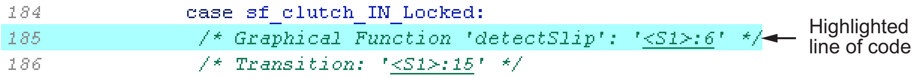
7. Click the `<S1>:3` hyperlink in this traceability comment:

   ```
   /* Graphical Function 'getSlipTorque': '<S1>:3' */
   ```

   In the chart, the graphical function `getSlipTorque` appears highlighted.

8. You can also trace a graphical function in the chart to a line of generated code. For example, right-click the graphical function `detectSlip` and select **Code Generation > Navigate to Code**.

   The code for that graphical function appears highlighted in `sf_clutch.c`.



```
        case sf_clutch_IN_Locked:
         /* Graphical Function 'detectSlip': '<S1>:6' */
         /* Transition: '<S1>:15' */
```

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.

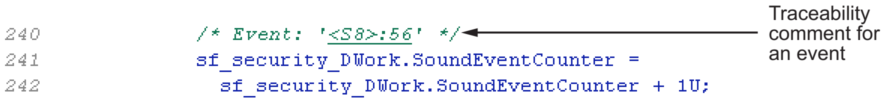
2. Complete steps 2 through 6 in “Bidirectional Traceability for States and Transitions” on page 25-81.

3. In the **Code Generation** 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.



```
240            /* Event: '<S8>:56' */
241            sf_security_DWork.SoundEventCounter =
242              sf_security_DWork.SoundEventCounter + 1U;
```

Traceability comment for an event

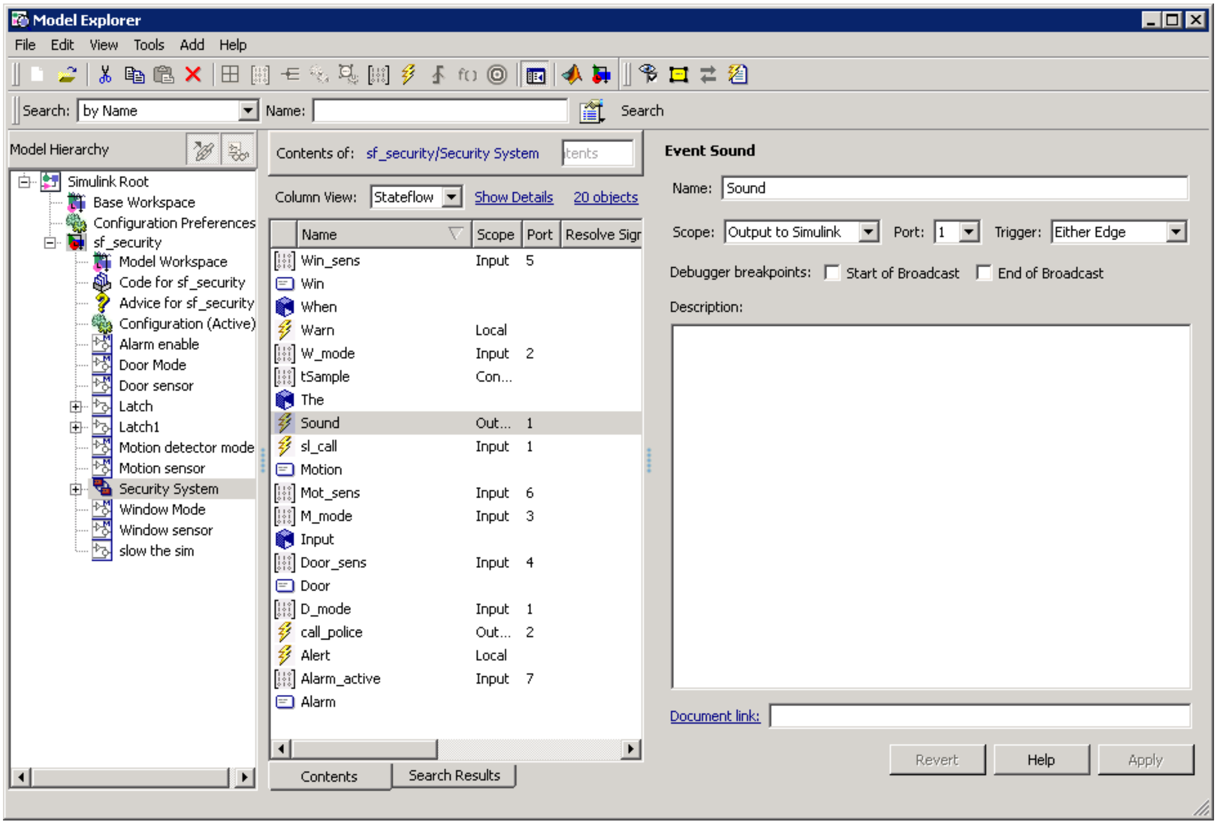
Note The line numbers shown above can differ from the numbers that appear in your code generation report.

---

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.



- Simulink Root
  - Base Workspace
  - Configuration Preferences
  - sf_security
    - Model Workspace
    - Code for sf_security
    - Advice for sf_security
    - Configuration (Active)
    - Alarm enable
    - Door Mode
    - Door sensor
    - Latch
    - Latch1
    - Motion detector mode
    - Motion sensor
    - Security System
    - Window Mode
    - Window sensor
    - slow the sim

Column View: Stateflow

Name	Scope	Port	Resolve Sigr
Win_sens	Input	5	
Win			
When			
Warn	Local		
W_mode	Input	2	
tSample	Con...		
The			
Sound	Out...	1	
sl_call	Input	1	
Motion			
Mot_sens	Input	6	
M_mode	Input	3	
Input			
Door_sens	Input	4	
Door			
D_mode	Input	1	
call_police	Out...	2	
Alert	Local		
Alarm_active	Input	7	
Alarm			

Event Sound

Name: Sound

Scope: Output to Simulink Port: 1 Trigger: Either Edge

Debugger breakpoints: ☐ Start of Broadcast ☐ End of Broadcast

Description:

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 through 6 in "Bidirectional Traceability for States and Transitions" on page 25-81.
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.

---

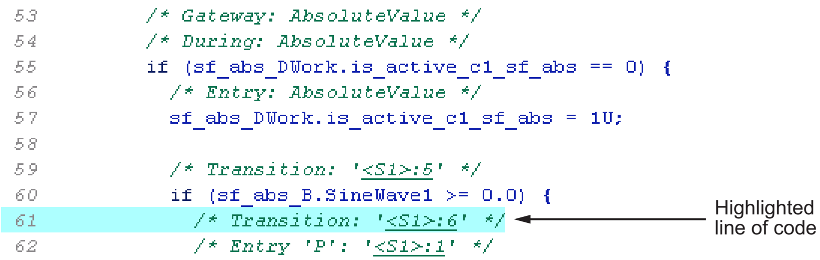
4 In the **Code Generation** pane, click **Build** in the lower right corner.

   The code generation report appears automatically.

5 Open the `AbsoluteValue` chart.

6 Right-click the left junction and select **Code Generation > Navigate to Code**.

   The code for the first outgoing transition of that junction appears highlighted in `sf_abs.c`.



```
/* Gateway: AbsoluteValue */
/* During: AbsoluteValue */
if (sf_abs_DWork.is_active_c1_sf_abs == 0) {
  /* Entry: AbsoluteValue */
  sf_abs_DWork.is_active_c1_sf_abs = 1U;

  /* Transition: '<S1>:5' */
  if (sf_abs_B.SineWave1 >= 0.0) {
    /* Transition: '<S1>:6' */
    /* Entry 'P': '<S1>:1' */
```

Highlighted line of code

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`.

MATLAB Function

Syntax.

```
/* MATLAB Function '<Name>': '<ObjectHyperlink>' */
```

Within the inlined code for a MATLAB function, comments that link to individual lines of the function have the following syntax:

```
/* '<ObjectHyperlink>' */
```

Examples.

```
/* MATLAB Function 'test_function': '<S50>:99' */
```

```
/* '<S50>:99:20' */
```

The first comment refers to the MATLAB function named `test_function`, which has the hyperlink `<S50>:99`.

The second comment refers to line 20 of the 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 25-93 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`.

Controlling Inlining of State Functions in Generated Code

In this section...

How Stateflow Software Inlines Generated Code for State Functions

By default, Stateflow software uses an internal heuristic to determine whether or not to inline state functions in code generated with Simulink Coder software. The heuristic takes into consideration an inlining threshold, so as your code grows and shrinks in size, the generated code for state functions can be unpredictable.

However, if you have rigorous requirements for traceability between generated code and the corresponding state functions, you can override this default behavior. Stateflow software provides a state property `Function Inline Option` that helps you explicitly force or prevent inlining of state functions.

What Happens When You Force Inlining

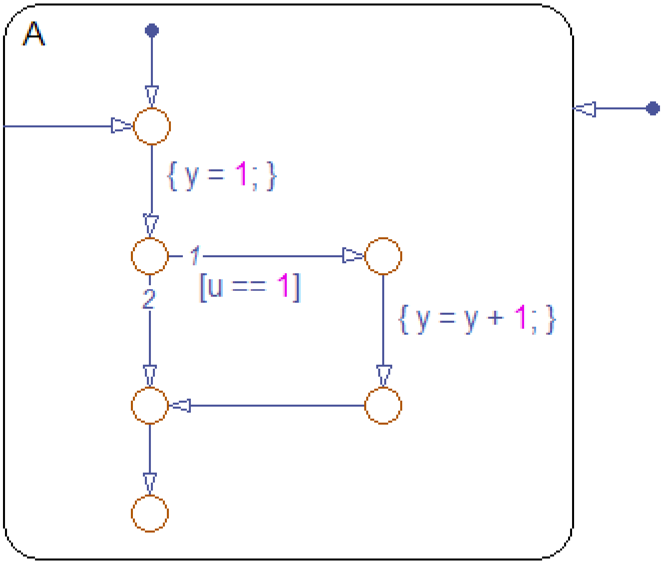
If you force inlining for a state, all code generated for its state actions will be inlined into the parent function. The parent function contains code for executing the state actions, outer transitions, and flow graphs. It does not include code for empty state actions.

What Happens When You Prevent Inlining

If you prevent inlining for a state, Simulink Coder software generates the following static functions, as in this example for state *foo*:

Function	Description
`enter_atomic_`*`foo`*	Marks *`foo`* active and performs entry actions
`enter_internal_`*`foo`*	Calls default paths
`inner_default_`*`foo`*	Executes flow graphs that originate when an inner transition and default transition reach the same junction inside a state. Stateflow software generates this function only when the flow graph is complex enough to exceed the inlining threshold. In generated code, Stateflow software calls this function from both the `enter_internal_`*`foo`* and *`foo`* functions.
`foo`	Checks for valid outer transitions and if none, performs during actions
`exit_atomic_`*`foo`*	Performs exit actions and marks *`foo`* inactive
`exit_internal_`*`foo`*	Performs exit actions of the child substates and then exits *`foo`*

Suppose that you explicitly prevent inlining for the following state `A` in model `M`:



Stateflow software generates the following functions:

```
static void M_inner_default_A(void);
static void M_exit_atomic_A(void);
static void M_A(void);
static void M_enter_atomic_A(void);
static void M_enter_internal_A(void);
```

How to Set the State Function Inline Option

To set the function inlining property for a state:

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

   The State properties dialog box opens.

2. In the **Function Inline Option** field, select one of these options:

Option	Behavior
Inline	Forces inlining of state functions into the parent function, as long as the function is not part of a recursion. See "What Happens When You Force Inlining" on page 25-95.
Function	Prevents inlining of state functions. Generates up to six static functions for the state. See "What Happens When You Prevent Inlining" on page 25-95.
Auto	Uses internal heuristics to determine whether or not to inline the state functions.

3 Click **Apply**.

Best Practices for Controlling State Function Inlining

To...	Set the Function Inline Option property to...
Generate a separate function for each action of a state and a separate function for each action of its substates	**Function** for the state and each substate
Generate a separate function for each action of a state, but include code for the associated action of its substates	**Function** for the state and **Inline** for each substate

26

Debugging and Testing Stateflow Charts

Using the Stateflow Debugger

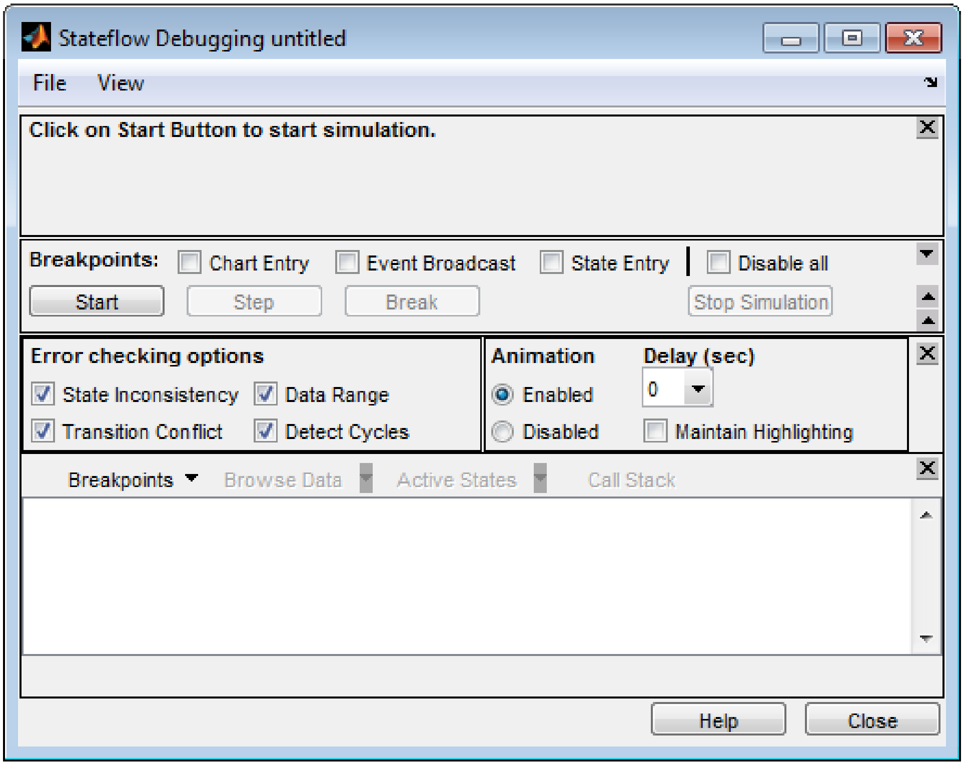
In this section...

Opening the Stateflow Debugger

To open the debugger, use the menu item or enter the command-line function.

How to Open the Debugger Using the Editor

In the editor, select **Debug > Stateflow Debugger**.



Click on Start Button to start simulation.

Breakpoints: ☐ Chart Entry ☐ Event Broadcast ☐ State Entry ☐ Disable all

Error checking options
☑ State Inconsistency ☑ Data Range
☑ Transition Conflict ☑ Detect Cycles

Animation
● Enabled
○ Disabled

Delay (sec) 0
☐ Maintain Highlighting

How to Open the Debugger at the Command Line

At the MATLAB command line, enter `sfdebugger`.

Animating Stateflow Charts

During simulation, you can animate a chart in your model to provide visual verification that your chart behaves as expected. Animation highlights objects in a chart as execution progresses.

You can animate a chart during simulation in one of 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 26-4)
- In *external* mode on a target machine where your generated code runs (see "Animating Stateflow Charts in External Mode" on page 26-4)

Animating Stateflow Charts in Normal Mode

During simulation in normal mode on a host machine, you can animate states and transitions in a chart.

1. Open the chart you want to animate.
2. In the editor, select **Debug > Stateflow Debugger** to open the debugger.
3. In the Animation section of the debugger, select **Enabled**.
4. Control the speed of animation by entering a value in the **Delay** field:
   - For the fastest animation, select a value of 0 seconds.
   - For the slowest animation, select a value of 1 second.
5. To maintain highlighting of active states in the chart after simulation ends, select the **Maintain Highlighting** check box.

   By default, active state highlighting disappears after chart simulation ends.
6. Start simulation.

   The chart highlights states and transitions as they execute.
7. To remove highlighting of active states after simulation ends, select **View > Remove Highlighting** in the chart.

Animating Stateflow Charts in External Mode

You can animate a chart in external mode — the mode in which Simulink Coder code generation software establishes communication between a Simulink model and code executing on a target system (see "Data Exchange" in the Simulink Coder documentation). In external mode, you can animate states in a chart, and view test point signals in a floating scope or signal viewer.

- "Animating States During Simulation in External Mode" on page 26-5
- "Viewing Test Point Data in Floating Scopes and Signal Viewers" on page 26-6

Animating States During Simulation in External Mode. To animate states in a chart in external mode:

1. Load the chart you want to animate to the target machine.
2. In the Stateflow Editor, select **Debug > Stateflow Debugger** to open the debugger.
3. In the Animation section of the debugger, select **Enabled**.
4. In the Stateflow Editor, select **Simulation > Configuration Parameters**.
5. In the left Select pane, select **Code Generation > 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 editor, select **Tools > External Mode Control Panel**.
8. In the External Mode Control Panel dialog box, click **Signal & Triggering**.
9. In the External Signal & Triggering dialog box, set these parameters:

In:	Select:
Signal selection pane	Chart 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 Model Editor, select **Simulation > External**, and then select **Simulation > Connect to Target**.
13. Start simulation.

    The chart highlights states as they execute.

Viewing Test Point Data in Floating Scopes and Signal Viewers. When you simulate a chart in external mode, you can view test point data in floating scopes and signal viewers. You can designate local data and states to be test points.

To view test point data during simulation in external mode:

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 chart.

4. In the Signal Selector **List contents** menu, select **Testpointed/Logged signals only** and then select the signals you want to view.

5. Simulate the model in external mode as described in “Animating States During Simulation in External Mode” on page 26-5.

   The scope or viewer displays the values of the test point signals as the simulation runs.

For more information, see “Behavior of Scopes and Viewers with Rapid Accelerator Mode” in the Simulink documentation.

Setting Breakpoints to Debug Charts

A breakpoint indicates a point at which the Stateflow debugger halts execution of a simulating chart. At this time, you can inspect Stateflow data and the MATLAB workspace to examine the status of a simulating chart.

The debugger 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 debugger to specify global breakpoints. When a global breakpoint occurs during simulation, execution stops and the debugger takes control. Select any or all of these breakpoints:

- **Chart Entry** — Simulation halts on any chart entry.
- **Event Broadcast** — Simulation halts for any event broadcast.
- **State Entry** — Simulation halts for any state entry.

Global breakpoints can be changed during run time and are immediately enforced. When you save the chart, all the Stateflow debugger 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 local breakpoints for:

- Charts
- States
- Transitions
- Graphical functions
- Truth table functions
- Events

For graphical objects, you can set local breakpoints using the right-click context menu in the chart or the properties dialog box for that object. To set local breakpoints for events, you must use the properties dialog box because events are not graphical objects.

To set local breakpoints using the right-click context menu:

1 Right-click the graphical object (chart, state, transition, graphical function, or truth table function) and select **Breakpoints**.

2 Depending on the object that you right-click, you can set different breakpoints:

For:	Select:
Charts	**Chart Entry** — Stop execution before entering the chart.
States	**State Entry** — Stop execution before performing the state entry actions. **State During** — Stop execution before performing the state during actions. **State Exit** — Stop execution before performing the state exit actions.

For:	Select:
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.
Graphical or truth table functions	**Function Call** — Stop execution before calling the function.

To set local breakpoints using the properties dialog box of the chart object:

1 Use one of the following tools to open the dialog box:

Tool	Action
Stateflow Editor	For a chart, select **File > Chart Properties**. For a state, transition, graphical function, or truth table function, right-click the object and select **Properties**. **What if my chart objects are grouped?** Double-click the chart to ungroup objects so you can access them individually.
Model Explorer	**1** Show all Stateflow objects by selecting **View > Row Filter > All Stateflow Objects**. **2** Right-click a chart, state, transition, graphical function, truth table function, or event and select **Properties**.

A dialog box appears for setting the properties of the object.

2 In the properties dialog box, select from the following breakpoint options:

For:	Select:
Charts	**On chart entry** — Stop execution before entering the chart.
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.
Graphical or truth table 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, select the **Disable all** check box.

Clearing All Breakpoints

To find and clear all breakpoints without disabling them, use the **Debug** menu in the chart editor or enter Stateflow API commands.

How to Clear Breakpoints Using the Editor. In the chart editor, select **Debug > Clear All Breakpoints**.

How to Clear Breakpoints at the Command Line. Use the following Stateflow API commands. (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, events, and charts
stateObjects = rootObj.find('-isa','Stateflow.State');
transitionObjects =rootObj.find('-isa','Stateflow.Transition');
dataObjects = rootObj.find('-isa','Stateflow.Data');
eventObjects = rootObj.find('-isa', 'Stateflow.Event');
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 events, clear their breakpoints
for i = 1:size(eventObjects,1)
eventObjects(i).Debug.Breakpoints.StartBroadcast = 0;
eventObjects(i).Debug.Breakpoints.EndBroadcast = 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 five 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, events, and charts as shown in the code.

How to Enable Debugging for Charts

You can enable debugging for all charts in a model or do so on a chart-by-chart basis.

How to Enable Debugging for All Charts in a Model

Follow these steps when you want debugging to apply to all charts in a model:

1. Open the Configuration Parameters dialog box.
2. On the **Simulation Target** pane, select **Enable debugging/animation**.

If you open each chart in its own editor, you see that **Debug > Enable Debugging** is selected by default. To see instructions for library link charts, go to "How to Enable or Disable Debugging for Library Link Charts" on page 26-17.

How to Enable Debugging for a Single Chart

Follow these steps when you want debugging to apply to only one chart:

1. Open the Configuration Parameters dialog box.
2. On the **Simulation Target** pane, select **Enable debugging/animation**.
3. Disable debugging for each chart that you do not want to debug:
   - a. Open the chart.
   - b. In the editor, clear **Debug > Enable Debugging**.

To see instructions for library link charts, go to "How to Enable or Disable Debugging for Library Link Charts" on page 26-17.

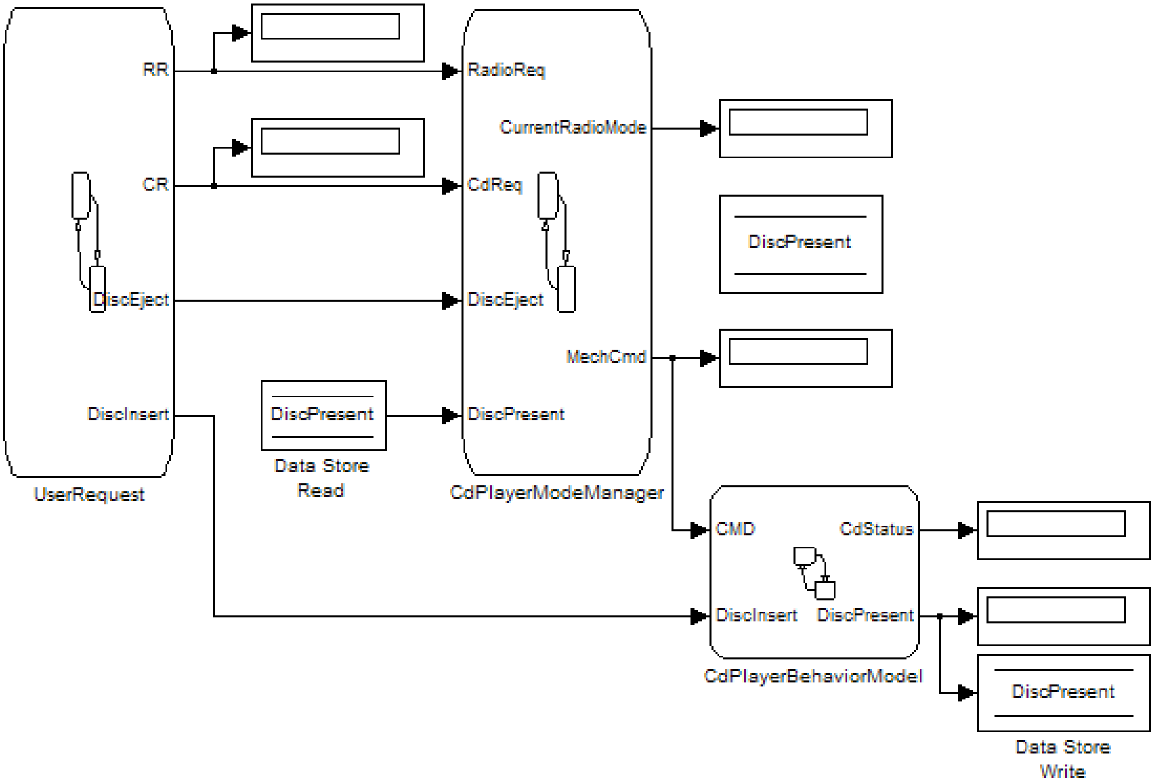
Example of Configuring a Model to Debug a Single Chart

The `sf_cdplayer` model contains three charts:

- `UserRequest`
- `CdPlayerModeManager`
- `CdPlayerBehaviorModel`

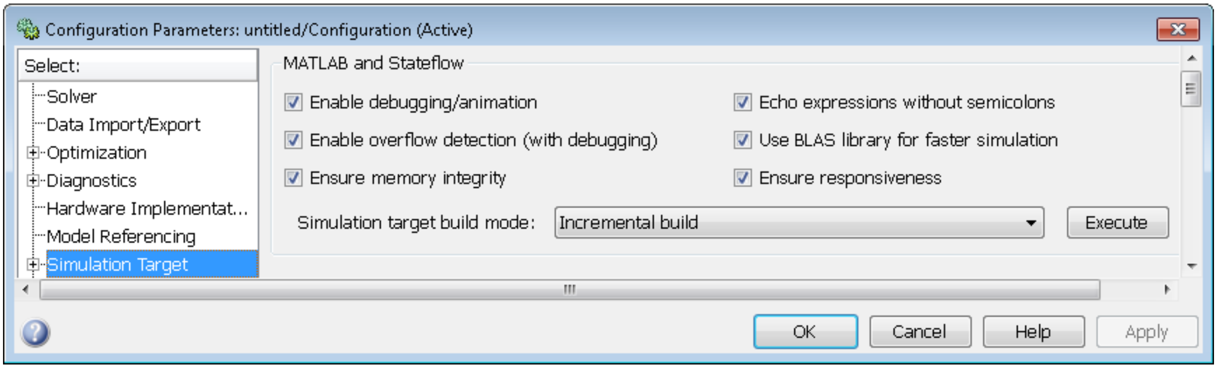
To enable debugging for only the `CdPlayerModeManager` chart:

1 Open the `sf_cdplayer` model.



2 Open the Configuration Parameters dialog box.

3 On the **Simulation Target** pane, select **Enable debugging/animation**.


Configuration Parameters: untitled/Configuration (Active)

Select:
- Solver
- Data Import/Export
- Optimization
- Diagnostics
- Hardware Implementat...
- Model Referencing
- Simulation Target

MATLAB and Stateflow

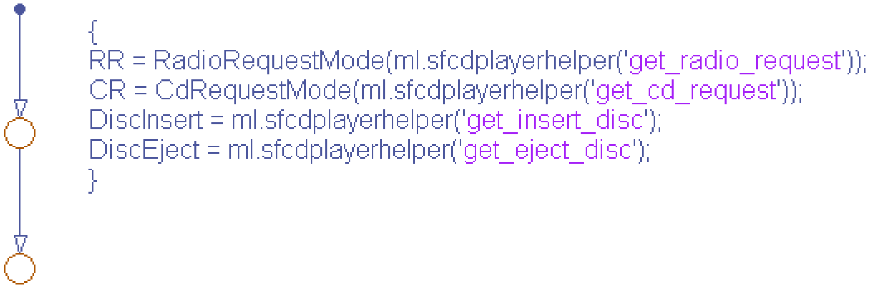
☑ Enable debugging/animation
☑ Echo expressions without semicolons
☑ Enable overflow detection (with debugging)
☑ Use BLAS library for faster simulation
☑ Ensure memory integrity
☑ Ensure responsiveness

Simulation target build mode: Incremental build

This step enables debugging and animation for all charts in your model.

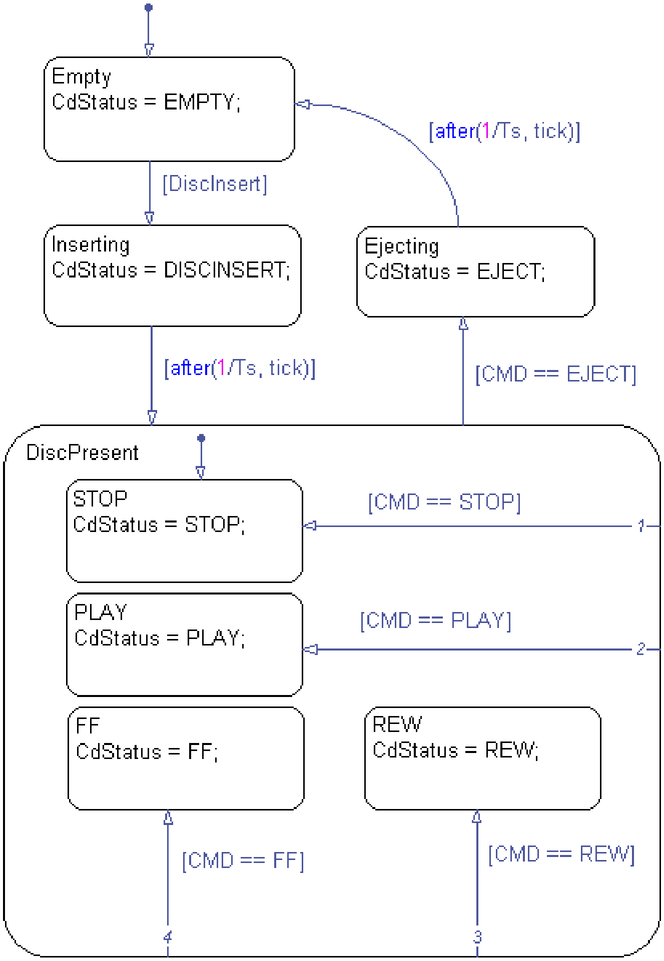
4 Disable debugging for the charts that you do not want to debug.

a Open the `UserRequest` chart.


```
{
RR = RadioRequestMode(ml.sfcdplayerhelper('get_radio_request'));
CR = CdRequestMode(ml.sfcdplayerhelper('get_cd_request'));
DiscInsert = ml.sfcdplayerhelper('get_insert_disc');
DiscEject = ml.sfcdplayerhelper('get_eject_disc');
}
```

b In the editor, clear **Debug > Enable Debugging**.

c Open the `CdPlayerBehaviorModel` chart.



d In the editor, clear **Debug > Enable Debugging**.

If you start simulation of `sf_cdplayer` in the Stateflow debugger, the debugger ignores execution of all charts except for `CdPlayerModeManager`. For more information, see “Starting Simulation in the Debugger” on page 26-17.

How to Enable or Disable Debugging for Library Link Charts

Whether you enable or disable debugging for a library link chart depends on the setting you specify for the chart in the library model. The **Enable debugging/animation** parameter in the Configuration Parameters dialog box does not control debugging preferences for library link charts.

To enable debugging for a library link chart in your model:

1. Open the library model.

2. Unlock the library.

3. Open the chart in that library.

4. Select **Debug > Enable Debugging**.

   This step specifies that all linked instances of that library chart have debugging enabled.

To disable debugging for a library link chart in your model:

1. Open the library model.

2. Unlock the library.

3. Open the chart in that library.

4. Clear **Debug > Enable Debugging**.

   This step specifies that all linked instances of that library chart have debugging disabled.

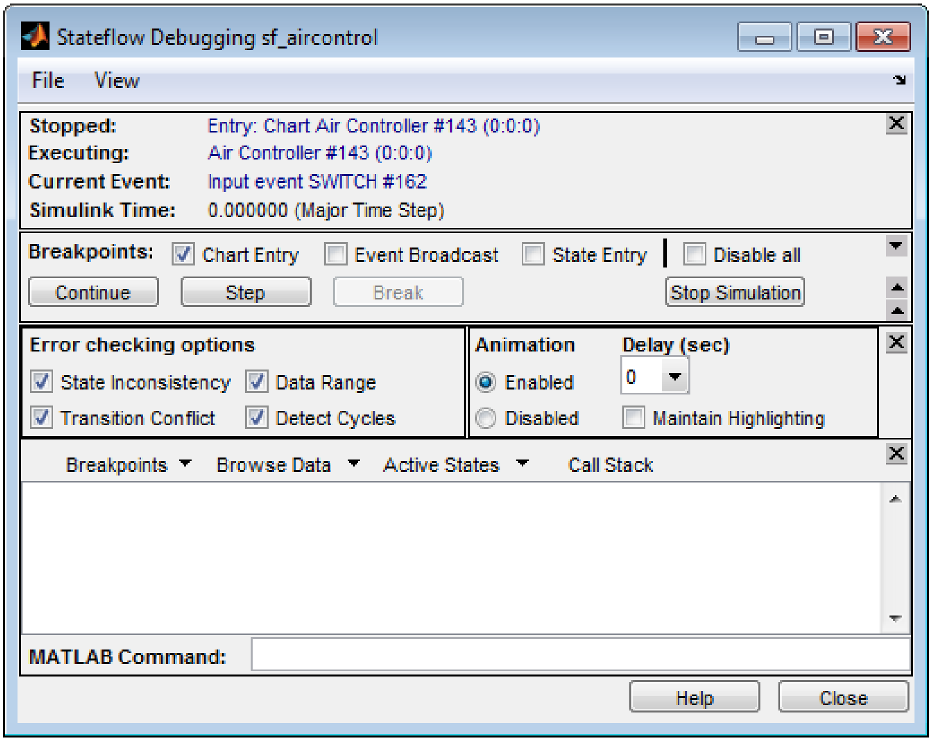
Options for Controlling the Debugger

Starting Simulation in the Debugger

To debug the charts in a model, you start simulation in the debugger:

1. Click the **Start** button.

A debugging simulation session starts. When simulation reaches a breakpoint that you set, the Stateflow debugger appears as follows:



Stopped: Entry: Chart Air Controller #143 (0:0:0)
Executing: Air Controller #143 (0:0:0)
Current Event: Input event SWITCH #162
Simulink Time: 0.000000 (Major Time Step)

Breakpoints: ☑ Chart Entry ☐ Event Broadcast ☐ State Entry ☐ Disable all

Error checking options
☑ State Inconsistency ☑ Data Range
☑ Transition Conflict ☑ Detect Cycles

Animation
● Enabled
○ Disabled

Delay (sec) 0
☐ Maintain Highlighting

MATLAB Command:

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 chart.
- **Current Event** — Displays the event that the chart is processing.
- **Simulink Time** — Displays the current simulation time.

During simulation, the chart is in read-only mode. The toolbar and menus change so that object creation is not possible. In this read-only mode, the chart is *iced*.

Options to Control Execution Rate in the Debugger

When the chart reaches a breakpoint, you can control the execution rate using single-step mode or continuous execution until the chart reaches another breakpoint. Use the following buttons in the Stateflow debugger to control the execution rate:

- **Continue** — After simulation starts and the chart reaches a breakpoint, the **Start** button becomes **Continue**. Click **Continue** to continue simulation.
- **Step** — Execute the next execution step, and suspend the simulation.
- **Break** — Suspend the simulation and transfer control to the debugger.
- **Stop Simulation** — Stop simulation 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.

During single-step mode, the debugger does not zoom automatically to the chart object that is executing. Instead, the debugger opens the subviewer that contains that object. This behavior minimizes visual disruptions as you step through your analysis of a simulation.

Options for Error Checking in the Debugger

The options in the **Error checking options** section of the Stateflow debugger insert generated code in the simulation target to provide breakpoints to catch different types of errors that might occur during simulation. Select any 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 “State Inconsistencies in a Chart” on page 26-27 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 “Conflicting Transitions in a Chart” on page 26-29 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 "Data Range Violations in a Chart" on page 26-31 for a complete description and example.
- **Detect Cycles** — Check whether a step or sequence of steps indefinitely repeats itself. See "Cyclic Behavior in a Chart" on page 26-32 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 25-16.

Note You must rebuild the target for any changes to the settings referenced above to take effect.

Options to Control Chart Animation

You can enable animation of the chart to show which states and transitions execute during a particular time step. Use the following controls:

- **Animation** — Select **Enabled** to turn on animation for the chart.
- **Delay** — Enter the speed of animation for the chart: `0` for fastest animation and `1` for slowest animation.
- **Maintain Highlighting** — Select this check box to maintain highlighting of active states at the end of chart simulation.

The options for **Delay** and **Maintain Highlighting** are available only when you enable animation. For more information, see "Animating Stateflow Charts" on page 26-3.

Options to Control the Output Display Pane

During simulation, the debugger monitors several execution indicators in the output display in the bottom pane of the debugger. You select the contents of this display with the following pull-down menus, which are available only after chart execution reaches a breakpoint.

- **Breakpoints** — Display a list of the set breakpoints. You can set breakpoints in the debugger and in the properties dialog boxes of individual objects such as states, transitions, and functions. See "Setting Breakpoints to Debug Charts" on page 26-7 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 26-37.
- **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 menu 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.

After you make a selection, the pull-down menu for the current display appears highlighted. When you select an output display button, that type of output appears until you choose a different display type. You can clear the display by selecting **File > Clear Display** in the Stateflow debugger.

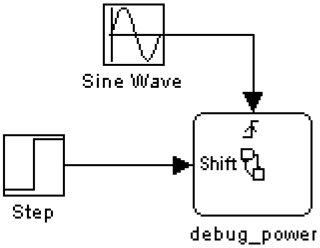
Example of Debugging Run-Time Errors in a Chart

In this section...
"Creating the Model and the Stateflow Chart" on page 26-22
"Debugging the Stateflow Chart" on page 26-24
"Correcting the Run-Time Error" on page 26-25
"Identifying Stateflow Objects in Error Messages" on page 26-26

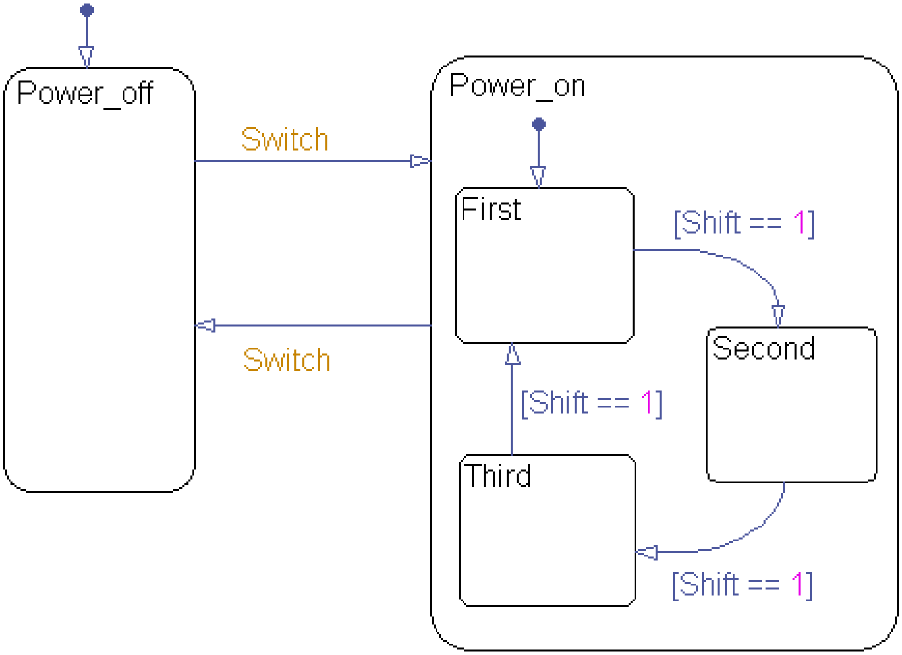
Creating the Model and the Stateflow Chart

In this topic, you create a model with a Stateflow chart to debug. Follow these steps:

1 Create the following Simulink model:



2 Add the following states and transitions to your chart:



3 In your chart, add an event `Switch` with a scope of **Input from Simulink** and a **Rising** edge trigger.

4 Add a data `Shift` with a scope of **Input from Simulink**.

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

In the model, there is an event input and a data input. A Sine Wave block generates a repeating input event that corresponds with the Stateflow event `Switch`. The Step block generates a repeating pattern of 1 and 0 that corresponds with the Stateflow data object `Shift`. Ideally, the `Switch` event occurs at a frequency that allows at least one cycle through `First`, `Second`, and `Third`.

Debugging the Stateflow Chart

To debug the chart in "Creating the Model and the Stateflow Chart" on page 26-22, follow these steps:

1. Open the Configuration Parameters dialog box.
2. In the **Simulation Target** pane, verify that **Enable debugging/animation** is selected.
3. Click **OK** to close the Configuration Parameters dialog box.
4. Open the Stateflow debugger.
5. In the **Breakpoints** section, select the **Chart Entry** check box.
6. Under **Animation**, select **Enabled** to enable animation of the chart during simulation.
7. Click **Start** to start the simulation.

   Because you specified a breakpoint on chart entry, execution stops at that point and the debugger shows you informational messages.
8. Click **Step**.

   The **Step** button executes the next step and stops.
9. Continue clicking the **Step** button and watching the animating chart.

   After each step, watch the chart animation and the debugger status area to see the sequence of execution.

Single-stepping shows that the chart does not exhibit the desired behavior. The transitions from `First` to `Second` to `Third` inside the state `Power_on` are not occurring because the transition from `Power_on` to `Power_off` takes priority. The output display of code coverage also confirms this observation.

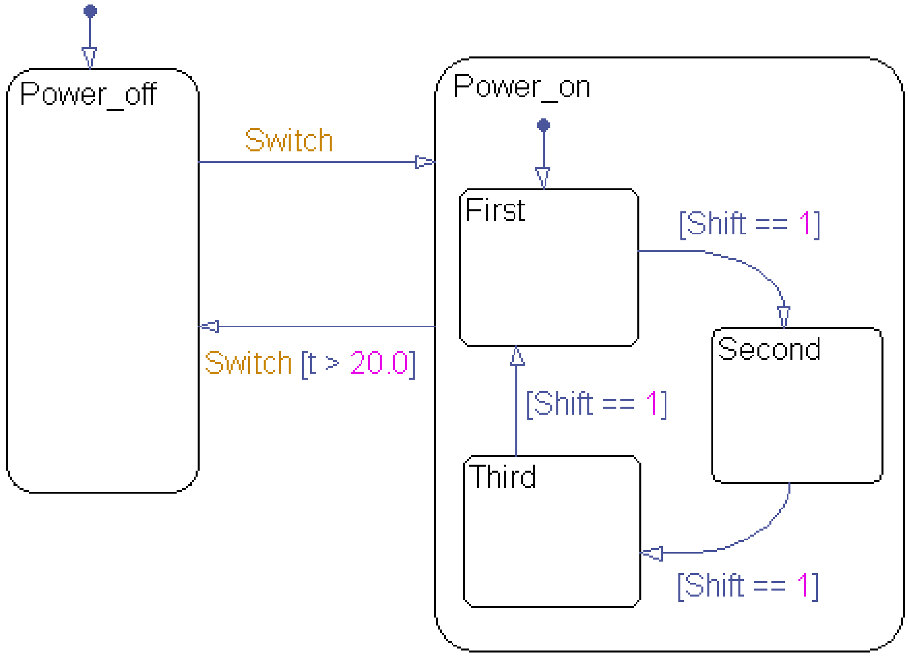
Correcting the Run-Time Error

In "Debugging the Stateflow Chart" on page 26-24, you step through a simulation of a chart and find an error: the event `Switch` drives the simulation but the simulation time passes too quickly for the input data object `Shift` to have an effect.

Correct this error as follows:

1 Stop the simulation so that you can edit the chart.

2 Add the condition `[t > 20.0]` to the transition from `Power_on` to `Power_off`.



Now the transition from `Power_on` to `Power_off` does not occur until simulation time is greater than 20.0.

3 In the Stateflow debugger, 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, the error 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, enter the following Stateflow API commands at the MATLAB prompt:

```
theObject = find(sfroot, 'Id', <id number>);
theObject.view
```

The first command finds the Stateflow object that matches the `<id number>` you specify. The second command highlights the chosen object in your chart. (See the Stateflow API documentation for information about the `find` and `view` methods.)

Common Modeling Errors the Debugger Can Detect

In this section...

State Inconsistencies in a Chart

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

An error occurs at compile time when the following conditions are all true:

- A transition leads to a state that has exclusive (OR) decomposition and multiple substates. There are no default paths that lead to the entry of any substate. This condition results in a state inconsistency error. (However, if all transitions into that state are supertransitions leading directly to the substates, there is no error.)
- The state with multiple substates does not contain a history junction.

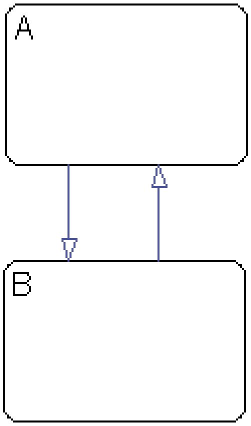
You can control the level of diagnostic action that occurs due to omission of a default transition in the **Diagnostics > Stateflow** pane of the Configuration Parameters dialog box. For more information, see the documentation for the "No unconditional default transitions" diagnostic.

Detecting State Inconsistency with the Debugger

1. Open the chart you want to debug.
2. Open the debugger and select **State Inconsistency**.
3. Start the simulation.

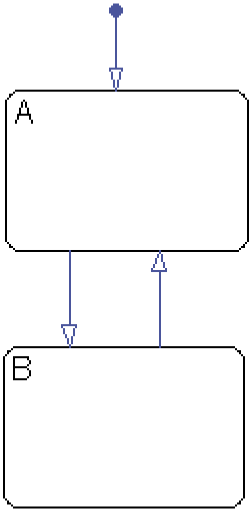
State Inconsistency Example

The following chart has a state inconsistency.



In the absence of a default transition indicating which substate is to become active, the chart has a state inconsistency error.

Adding a default transition to one of the substates resolves the state inconsistency.



Conflicting Transitions in a Chart

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 3-56.
- For implicit ordering, evaluation of conflicting transitions occurs based on internal rules described in "Implicit Ordering of Outgoing Transitions" on page 3-59.

Detecting Conflicting Transitions

To detect conflicting transitions during a simulation:

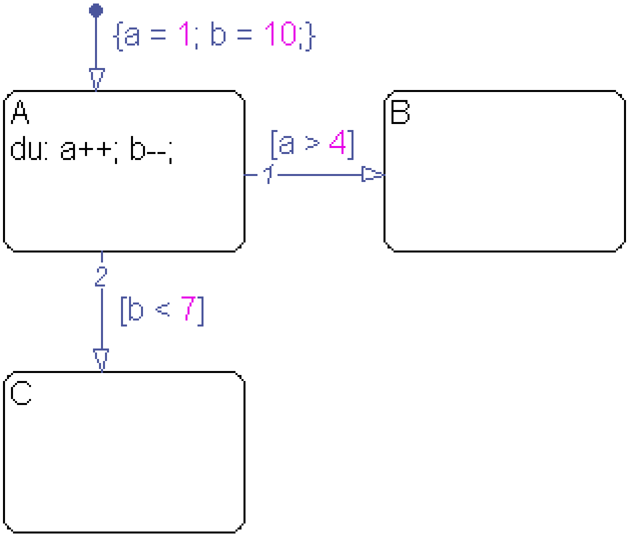
1. Build the target with debugging enabled.

2 Open the debugger and select **Transition Conflict**.

3 Start the simulation.

Example of Conflicting Transitions

The following 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.

If you select the **Transition Conflict** check box in the debugger, you get a run-time error. If you do not select the check box, resolution of the transition conflict depends on the ordering you use for evaluation of outgoing transitions.

Conflict Resolution for Explicit Ordering When Check Box Is Not Selected. 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.

Conflict Resolution for Implicit Ordering When Check Box Is Not Selected. 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.

Data Range Violations in a Chart

Types of Data Range Violations

The Stateflow debugger detects the following data range violations during simulation:

- When a data object equals a value outside the range of the values set in the **Initial value**, **Minimum**, and **Maximum** fields specified in the Data properties dialog box

  See "Setting Data Properties in the Data Dialog Box" on page 8-5 for a description of the **Initial value**, **Minimum**, and **Maximum** fields in the Data properties dialog box.

- When the result of a fixed-point operation overflows its bit size

  See "Detecting Overflow for Fixed-Point Types" on page 17-11 for a description of the overflow condition in fixed-point numbers.

When you select **Saturate on integer overflow** for your chart, the debugger does not flag any cases of integer overflow during simulation. However, the debugger continues to flag out-of-range data violations based on minimum-and-maximum range checks. For more information, see "Impact of Saturation on Debugger Checks" on page 8-64.

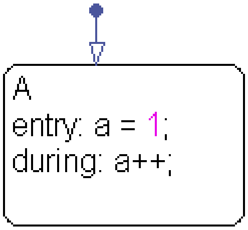
Detecting Data Range Violations

To detect data range violations during a simulation:

1. Build the target with debugging enabled.
2. Open the debugger and select **Data Range**.
3. Start the simulation.

Data Range Violation Example

The following chart has a data range violation.



Assume that the data `a` has an **Initial value** and **Minimum** value of 0 and a **Maximum** value of 2. Each time an event awakens this chart and state `A` is active, `a` increments. The value of `a` quickly becomes a data range violation.

Cyclic Behavior in a Chart

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:

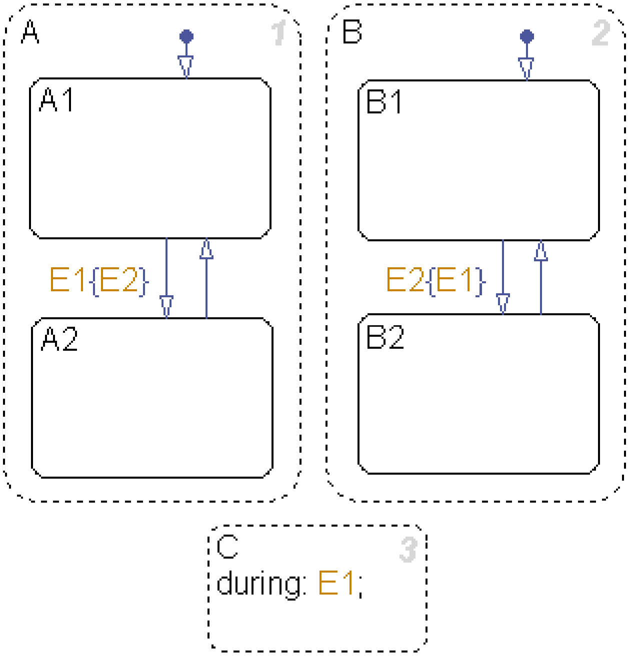
1. Build the target with debugging enabled.

2 Open the debugger and select **Detect Cycles**.

3 Start the simulation.

Cyclic Behavior Example

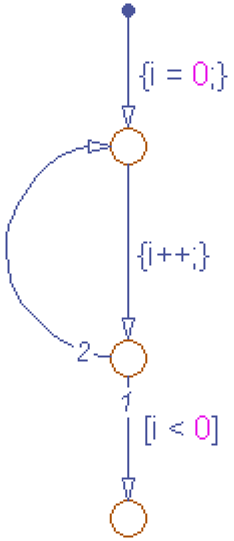
This chart shows how an event broadcast can cause infinite recursive cycles.



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 chart shows an example of cyclic behavior in a flow graph that the debugger cannot detect.



The data object `i` is set to 0 in the condition action of the default transition. `i` increments 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 is never true in this flow graph, resulting in a cycle.

The debugger cannot detect this cycle because it does not involve recursion due to event broadcasts. Although the debugger cannot detect cycles that depend on data values, a separate diagnostic error does appear during simulation, for example:

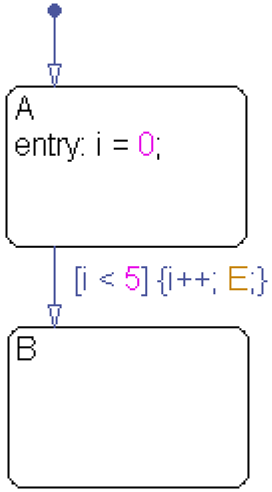
```
Junction is part of a cycle and does not have an
unconditional path leading to termination.
```

For information on fixing cyclic behavior in flow graphs, type the following at the MATLAB command prompt:

```
sfhelp('cycle_error');
```

Noncyclic Behavior Flagged as a Cycle Example

This 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 behavior as a cycle, but the so-called cycle breaks when `i` becomes greater than 5.

Guidelines for Avoiding Unwanted Recursion in a Chart

Recursion can be useful for controlling substate transitions among parallel states at the same level of the chart hierarchy. For example, you can send a directed event broadcast from one parallel state to a sibling parallel state to specify a substate transition. (For details, see "Directed Event Broadcasting" on page 10-59.) This type of recursive behavior is desirable and efficient.

However, unwanted recursion can also occur during chart execution. To avoid unwanted recursion, follow these guidelines:

Do not call functions recursively

Suppose that you have functions named `f`, `g`, and `h` in a chart. These functions can be any combination of graphical functions, truth table functions, MATLAB functions, or Simulink functions.

To avoid recursive behavior, do not:

- Have `f` calling `g` calling `h` calling `f`
- Have `f`, `g`, or `h` calling itself

Do not use undirected event broadcasts

Follow these rules:

- Use directed event broadcasts with the syntax `send(`*`event`*`,`*`state`*`)`. The *`event`* is a chart local event and the *`state`* is the destination state that you want to wake up using the event broadcast.
- If the source of the event broadcast is a state action, ensure that the destination state is *not* an ancestor of the source state in the chart hierarchy.
- If the source of the event broadcast is a transition, ensure that the destination state is *not* an ancestor of the transition in the chart hierarchy.

  Also, ensure that the transition does not connect to the destination state.

Watching Data Values During Simulation

In this section...

Watching Data in the Stateflow Debugger

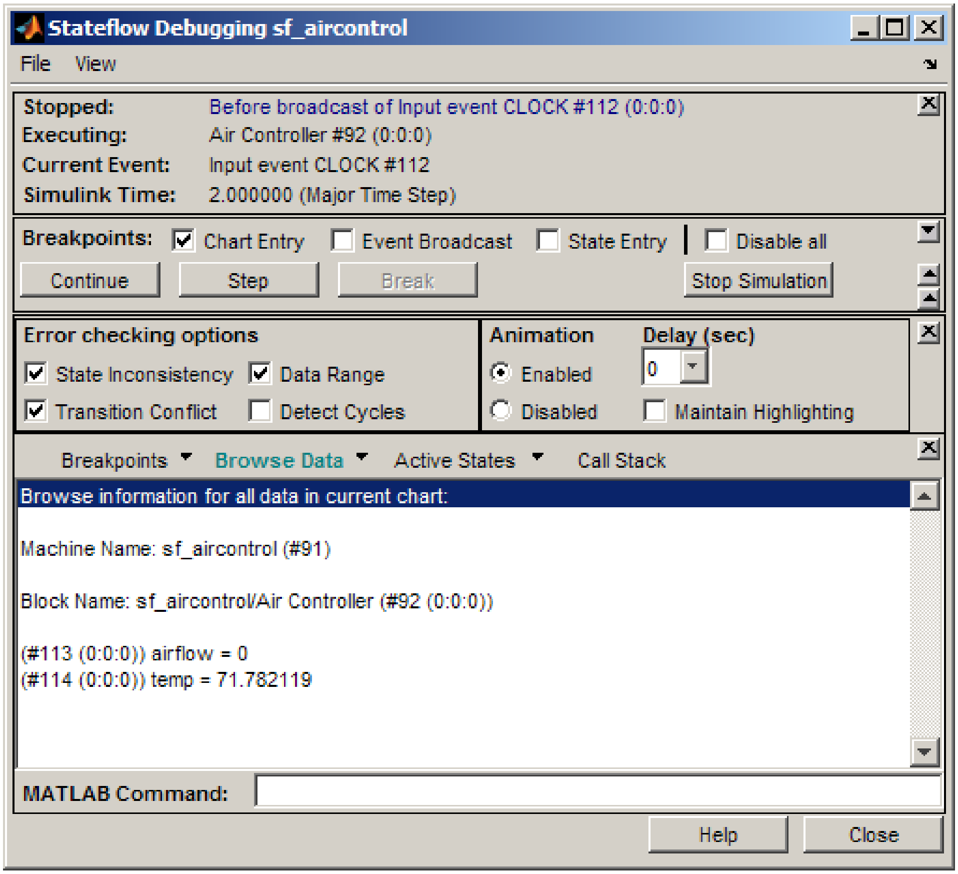
The **Browse Data** pull-down menu in the Stateflow debugger lets you display selected data in the bottom output display pane of the Stateflow debugger during simulation, after a breakpoint is reached. The debugger can filter the display between:

- 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 debugger" on page 8-14, as described in "Properties You Can Set in the General Pane" on page 8-8.

The following example displays **All Data (All Charts)** for a chart named `Air Controller`. This chart has two data values: `airflow` and `temp`.



Stopped: Before broadcast of Input event CLOCK #112 (0:0:0)
Executing: Air Controller #92 (0:0:0)
Current Event: Input event CLOCK #112
Simulink Time: 2.000000 (Major Time Step)

Breakpoints: ☑ Chart Entry ☐ Event Broadcast ☐ State Entry ☐ Disable all

Error checking options
☑ State Inconsistency ☑ Data Range
☑ Transition Conflict ☐ Detect Cycles

Animation ● Enabled ○ Disabled
Delay (sec) 0
☐ Maintain Highlighting

```
Browse information for all data in current chart:

Machine Name: sf_aircontrol (#91)

Block Name: sf_aircontrol/Air Controller (#92 (0:0:0))

(#113 (0:0:0)) airflow = 0
(#114 (0:0:0)) temp = 71.782119
```

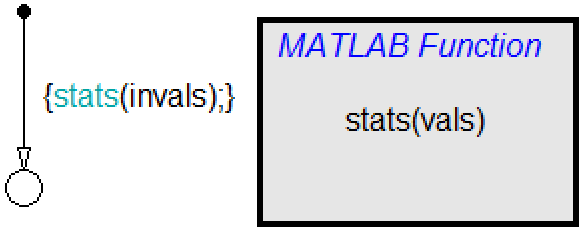
MATLAB Command:

Each displayed object (chart, state, data, and so on) appears with a unique identifier of the form (`#id(xx:yy:zz)`), which links the listed object to its appearance in the chart. In the **Browse Data** section, data appears in alphabetical order, regardless of its scope in a 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 17-6.

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 chart, a default transition calls a MATLAB function:



A breakpoint is set at the last executable line of the function:

```
function stats(vals)
%#codegen

% calculates a statistical mean and standard deviation
% for the values in vals.

len = length(vals);
mean = avg(vals, len);
stdev = sqrt(sum(((vals-avg(vals,len)).^2))/len);
coder.extrinsic('plot');
plot(vals,'-+');  % Breakpoint set at this line
```

When simulation reaches the breakpoint, you can display Stateflow data in the MATLAB Command Window.

1. At the MATLAB prompt, press **Enter**.

   A `debug>>` prompt appears.

2. Type `whos` to view the data that is visible at the current scope.

3. Enter the name of data array `vals` at the prompt to display its value.
4. Enter `vals(2:3)` to view specific values of that array.

The Command Line Debugger provides these commands during simulation:

Command	**Description**
`dbstep`	Advance to next executable line of code.
`dbstep [in/out]`	When debugging MATLAB functions in a chart: - `dbstep [in]` advances to the next executable line of code. If that line contains a call to another function, execution continues to the first executable line of the function. - `dbstep [out]` executes the rest of the function and stops just after leaving the function.
`dbcont`	Continue execution to next breakpoint.
`dbquit (ctrl-c)`	Stop simulation of the model. Press **Enter** after this command to return to the command prompt.
`help`	Display help for command-line debugging.
`print` *`var`* ...or... *`var`*	Display the value of the variable *`var`*.
`var` (*`i`*)	Display the value of the *`i`*th element of the vector or matrix *`var`*.
`var` (*`i:j`*)	Display the value of a submatrix of the vector or matrix *`var`*.
`save`	Saves all variables to the specified file. Follows the syntax of the MATLAB `save` command. To retrieve variables in the MATLAB base workspace, use the `load` command after simulation has ended.
`whos`	Display the size and class (type) of all variables in the scope of the halted MATLAB function in your chart.

You can issue any other MATLAB command at the `debug>>` prompt but the results are executed in the Stateflow workspace. For example, you can issue the MATLAB command `plot(`*`var`*`)` to plot the values of the variable *`var`*.

To issue a command in the MATLAB base workspace at the `debug>>` prompt, use the `evalin` command with the first argument `'base'` followed by the second argument command string, for example, `evalin('base','whos')`.

Note To return to the MATLAB base workspace, use the `dbquit` command.

Changing Data Values During Simulation

In this section...

How to Change Values of Stateflow Data

When your chart is in debug mode, you can test the simulation by changing the values of data in the chart. After the `debug>>` prompt appears, as described in "Watching Stateflow Data in the MATLAB Command Window" on page 26-39, you can assign a different value to your data. To change a data value, enter the new value at the prompt using the following format:

```
data_name = new_value
```

For a list of data that you cannot change, see "Data That Is Read-Only During Simulation" on page 26-45.

Examples of Changing Data Values

Scalar Example

Suppose that, after the `debug>>` prompt appears, you enter `whos` at the prompt and see the following data:

```
Name          Size       Bytes  Class

airflow       1x1            1  uint8 array
temp          1x1            8  double array
```

To change...	To this value...	Enter...
`airflow`	2	`airflow = uint8(2)`
`temp`	68.75	`temp = 68.75`

If you try to enter `airflow = 2`, you get an error message because MATLAB interprets that expression as the assignment of a `double` value to data of `uint8` type. For reference, see "Cases When Casting Is Necessary" on page 26-46.

Multidimensional Example

Suppose that, after the `debug>>` prompt appears, you enter `whos` at the prompt and see the following data:

```
Name                  Size                 Bytes  Class

ball_interaction      16x16                  256  int8 array
last_vel              16x2                   256  double array
stopped               16x1                    16  int16 array
```

To change...	To this value...	Enter...
The element in row 8, column 8 of `ball_interaction`	1	`ball_interaction(8,8) = int8(1)`
The element in row 16, column 1 of `last_vel`	120.52	`last_vel(16,1) = 120.52`
The last element in `stopped`	0	`stopped(16) = int16(0)`

One-based indexing applies when you change values of Stateflow data while the chart is in debug mode.

Variable-Size Example

Suppose that, after the `debug>>` prompt appears, you enter `whos` at the prompt and see the following data:

```
Name   Size   Bytes  Class

y1     1x1        8  double array (variable sized: MAX 16x16)
y2     1x1        8  double array (variable sized: MAX 16x4)
```

To change...	To...	Enter...
`y1`	A 10-by-5 array of ones	`y1 = ones(10,5)`
`y2`	A 6-by-4 array of zeros	`y2 = zeros(6,4)`

Changing the dimensions of variable-size data works only when the new size does not exceed the dimension bounds.

Fixed-Point Example

Suppose that, after the `debug>>` prompt appears, you enter `whos` at the prompt and see the following data:

```
Name       Size      Bytes  Class

y_n1       1x1           2  fixpt (int16 array (2^-10)*SI)
x_n1       1x1           2  fixpt (int16 array (2^-12)*SI)
```

Both `y_n1` and `x_n1` have signed fixed-point types, with a word length of 16. `y_n1` has a fraction length of 10 and `x_n1` has a fraction length of 12.

To change...	To this fixed-point value...	Enter...
`y_n1`	0.5410	`y_n1 = fi(0.5412,1,16,10)`
`x_n1`	0.4143	`x_n1 = fi(0.4142,1,16,12)`

For more information about using `fi` objects, see the Fixed-Point Toolbox™ documentation.

Enumerated Example

Suppose that, after the `debug>>` prompt appears, you enter `whos` at the prompt and see the following data:

```
Name                    Size      Bytes  Class

CurrentRadioMode        1x1            4  int32 array
MechCmd                 1x1            4  int32 array
```

Assume that `CurrentRadioMode` and `MechCmd` use the enumerated types `RadioRequestMode` and `CdRequestMode`, respectively.

To change...	To this enumerated value...	Enter...
`CurrentRadioMode`	CD	`CurrentRadioMode = RadioRequestMode.CD`
`MechCmd`	PLAY	`MechCmd = CdRequestMode.PLAY`

You must include the enumerated type explicitly in the assignment. Otherwise, an error appears at the `debug>>` prompt.

Limitations on Changing Data Values

Data That Is Read-Only During Simulation

You cannot change data of the following scopes while the chart is in debug mode:

- Constant
- Input

Limitations on Changing Type and Size

The following data properties cannot change:

- Data type
- Size

However, for variable-size data, you can change the dimensions of the data as long as the size falls within the dimension bounds. For example, `varsizedData = ones(5,7);` is a valid assignment for a variable-size 10-by-10 array.

Limitations for Fixed-Point Data

- Do not assign a value that falls outside the range of values that the fixed-point type can represent. Avoid selecting a value that causes overflow.
- Sign, word length, fraction length, slope, and bias cannot change.

Limitations for Structures

- You cannot change the data type or size of any fields.
- Addition or deletion of fields does not work because the size of the structure cannot change.

Cases When Casting Is Necessary

When you change a data value, you must explicitly cast values for data of the following built-in types:

- `single`
- `int8`
- `uint8`
- `int16`
- `uint16`
- `int32`
- `uint32`

For example, the following assignments are valid:

- `my_data1 = uint8(2)`
- `my_data2 = single(5.3)`

Casting is not necessary when you change the value of data that is of type `double`.

Monitoring Test Points in Stateflow Charts

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

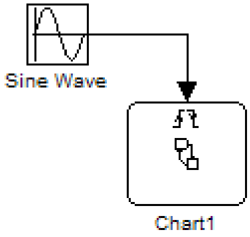
You can specify individual data or states as test points by setting their **TestPoint** property via the Stateflow API or in the Model Explorer (see "Setting Test Points for Stateflow States and Local Data with the Model Explorer" on page 26-48).

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

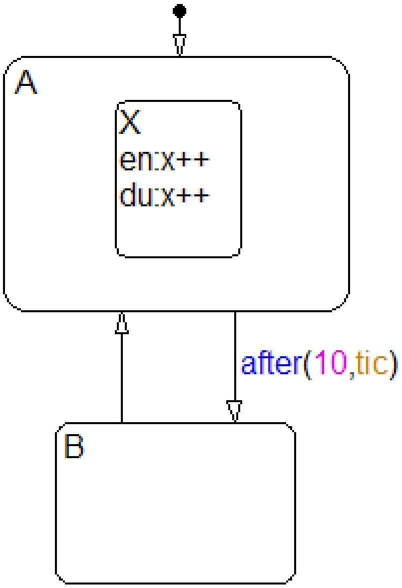
You can explicitly set individual states or local data as test points in the Model Explorer. The following procedure shows how to set individual test points for Stateflow states and data.

1 Create this model:



The model consists of a Sine Wave block that triggers a Stateflow chart using the input trigger event `tic`.

2 Add the following states and transitions to your chart:



The state `A` and its substate `X` are entered on the first `tic` event. State `A` and substate `X` stay active until 10 `tic` events have occurred, and then state `B` is entered. On the next event, state `A` and substate `X` are entered and the cycle continues.

The data `x` belongs to substate `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 then the cycle repeats.

3. Save the model as `myModel.mdl`.
4. Open the Configuration Parameters dialog box.
5. In the **Solver** pane, specify solver options:
   a. Set **Type** to `Fixed-step`.
   b. Set **Solver** to `discrete (no continuous states)`.
   c. Set **Fixed-step size (fundamental sample time)** to `0.1`.
   d. Click **OK**.
6. Open the Model Explorer.
7. In the Model Explorer, expand the `myModel` node and then the `Chart1` node.
8. Right-click `A` and select **Properties**.
9. In the properties dialog box, select the **Test point** check box and then click **OK**.

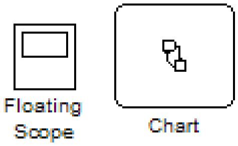
   This step creates a test point for the state `A`.
10. Repeat the previous step for states `A.X` and `B`.
11. In the Model Explorer, select state `X` again.
12. Right-click the local data `x` and select **Properties**.
13. In the properties dialog box, select the **Test point** check box and then click **OK**.
14. Close the Model Explorer and save the model.

You can also log these test points. See “Logging Multiple Signals At Once” on page 26-60 for instructions on using the Signal Logging dialog box. See “Logging Chart Signals Using the Command-Line API” on page 26-61 for instructions on logging signals at the MATLAB command line.

Using a Floating Scope to Monitor Data Values and State Activity

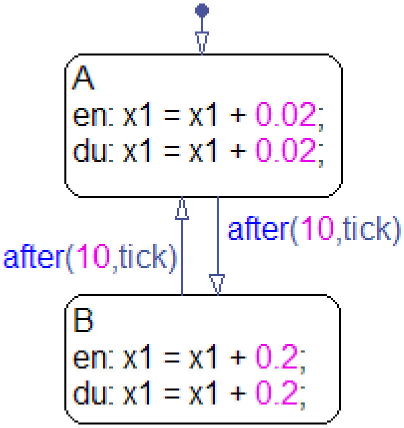
In this section, you configure a Floating Scope block to monitor a data value and the activity of a state.

1 Create this model:



The model consists of a Floating Scope block and a Stateflow chart.

2 Add the following states and transitions to your chart:



The chart 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 then the cycle repeats.

3. Save the model.

4. Open the Configuration Parameters dialog box.

5. In the **Solver** pane, specify solver options:
   1. Set **Type** to `Fixed-step`.
   2. Set **Solver** to `discrete (no continuous states)`.
   3. Set **Fixed-step size (fundamental sample time)** to `0.1`.
   4. Click **OK**.

6. Specify states `A` and `B` as test points:
   1. In the chart, right-click each state and select **Properties**.
   2. In the State properties dialog box, select **Test point**.
   3. Click **OK**.

7. Specify data `x1` as a test point:

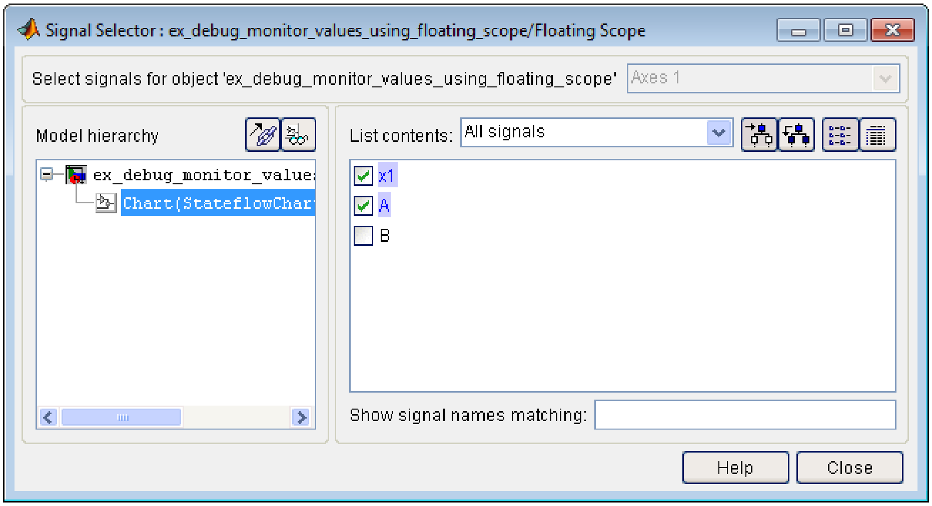
   a. Open the Model Explorer.
   b. In the **Model Hierarchy** pane, navigate to the chart.
   c. In the **Contents** pane, right-click `x1` and select **Properties**.
   d. In the Data properties dialog box, select **Test point**.
   e. Click **OK**.
   f. Close the Model Explorer.

8. Double-click the Floating Scope block to open the window.

9. In the Floating Scope window, click the Signal Selection icon.

   The Signal Selector dialog box appears with a hierarchy of Simulink blocks for the model.

10. In the **Model hierarchy** pane, select the chart whose signals you want to monitor and in the **List contents** pane, select the signals.



Signal Selector : ex_debug_monitor_values_using_floating_scope/Floating Scope

Select signals for object 'ex_debug_monitor_values_using_floating_scope' Axes 1

Model hierarchy

ex_debug_monitor_value

Chart(StateflowChar

List contents: All signals

☑ x1

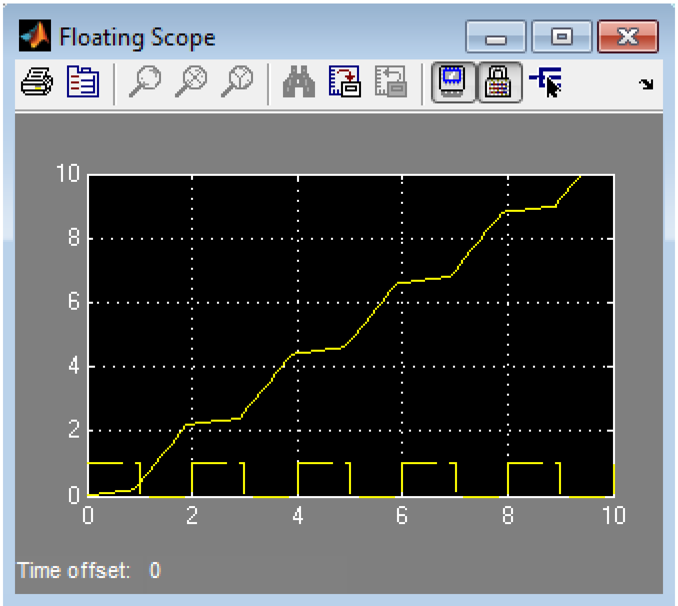
☑ A

☐ B

Show signal names matching:

11. Simulate the model.

    You see a signal trace for `x1` and the activity of state `A`.



When state A is active, the signal value is 1. When that state is inactive, the signal value is 0. Because this value can be very low or high compared to other data, you might want to add a second Floating Scope block to compare the activity signal with other data.

Logging Data Values and State Activity

In this section...

What You Can Log During Chart Simulation

When you simulate a chart, you can log values for local data and state activity into Simulink objects. After simulation, you can access these objects in the MATLAB workspace and use them to report and analyze the values.

When you log a state, its value is 1 when active and 0 when inactive.

Logging Stateflow data and state activity follows the same general guidelines as for logging signals in Simulink models.

See Also

- "Exporting Signal Data Using Signal Logging" in the Simulink documentation.

Workflow for Logging States and Local Data

The workflow for logging chart local data and state activity is similar to the workflow for logging signals in a model:

1. Enable signal logging for the chart and choose a logging format.

   See "Enabling Signal Logging and Choosing a Logging Format" on page 26-56.

2. Configure states and local data for signal logging, which includes controlling how much output the simulation generates.

   See "Configuring States and Local Data for Logging" on page 26-58.

3. Simulate the chart.

4. Access the logged data.

   See "Accessing Logged Data" on page 26-62.

Example for Logging Workflow

The procedures that take you through the logging workflow use `sf_semantics_hotel_checkin`. The model simulates a hotel check-in process. To open the model, type `sf_semantics_hotel_checkin` at the MATLAB command prompt.

In this model, the chart `Hotel` controls activities that trigger transitions to different rooms in the hotel. The chart contains a hierarchy of nested states and uses two local variables:

Local Variable	Description
`move_bags`	Indicates whether bags should move to another room or stay in the current room.
`service`	Accumulates the number of room service calls.

Enabling Signal Logging and Choosing a Logging Format

1. Open the chart and select **Simulation > Configuration Parameters**.

2 Select **Data Import/Export**.

3 In the Signals pane, select the **Signal logging** check box to enable logging for the chart.

Signal logging is enabled by default for models and charts. To disable logging, clear the check box.

4 Optionally, specify a custom name for the signal logging object.

The default name is `logsout`. Using this object, you can access the logging data in a MATLAB workspace variable (see "Signal Logging Object" on page 26-63).

5 Specify a format for logged data from the **Signal logging format** menu.

See "Supported Formats for Logged Data" on page 26-57.

6 Click **OK**.

Supported Formats for Logged Data

Stateflow charts support the same formats for logged data as Simulink models:

Format	Description
`Dataset`	Stores logged data as MATLAB timeseries objects in objects of the value class `Simulink.SimulationData.Dataset`.
`ModelDataLogs`	Stores logged data in objects of the handle class `Simulink.ModelDataLogs`.

Advantages of Dataset Format. The `Dataset` format offers several advantages over the `ModelDataLogs` format for logging chart data and state activity:

- Supports logging multiple changes to data values for a given time step.

- Provides a `getElement` function for easy access to logged data in charts with deep hierarchies.
- Provides consistent logging output when charts appear in referenced models.
- Represents model hierarchy as a flat list for easy access by index to nested elements.
- Supports logging of atomic subchart activity.

  See "Rules for Using Atomic Subcharts in Stateflow Charts" on page 11-49.

See Also.

- "Specifying the Signal Logging Data Format" in the Simulink documentation
- "Benefits of Using the Dataset Format for Signal Logging" in the Simulink documentation

Configuring States and Local Data for Logging

- "Properties to Configure for Logging" on page 26-58
- "Choosing a Configuration Method for Logging" on page 26-59
- "Logging Individual States and Data" on page 26-60
- "Logging Multiple Signals At Once" on page 26-60
- "Logging Chart Signals Using the Command-Line API" on page 26-61

Properties to Configure for Logging

You can configure the same properties for logging states and local data in a chart as you can for logging signals in a model:

Property	Description
Log signal data	Saves the signal's value to the MATLAB workspace during simulation.
Logging name	Name of the logged signal. Defaults to the original name of the state or local data. To rename the logged signal, select `Custom` and enter a new name. For guidance on when to use a different name for a logged signal, see "Specifying a Logging Name" in the Simulink documentation.
Limit data points to last	Limits the amount of data logged to the most recent samples. See "Limit Data Points to Last" in the Simulink documentation.
Decimation value	Limits the amount of data logged by skipping samples. For example, a decimation factor of 2 saves every other sample. See "Decimation" in the Simulink documentation.

See Also.

- "Logging and Accessibility Options" in the Simulink documentation

Choosing a Configuration Method for Logging

There are several ways to configure states and local data for logging:

Method	When to Use
"Logging Individual States and Data" on page 26-60	Configure states or local data for logging one at a time from inside the chart.
"Logging Multiple Signals At Once" on page 26-60	Configure multiple signals for logging from a list of all states and local data.
"Logging Chart Signals Using the Command-Line API" on page 26-61	Configure logging properties programmatically.

Logging Individual States and Data

1 Open the properties dialog box for the state or local data.

For:	Do This:
States	Right-click the state and select **Properties**.
Local data	Right-click the state or transition that uses the local data and select **Explore > (data)** *variable_name*.

2 In the properties dialog box, click the **Logging** tab.

3 Modify properties as needed, as described in "Properties to Configure for Logging" on page 26-58.

For example, from the `Hotel` chart of the `sf_semantics_hotel_checkin` model:

1 Open the properties dialog box for the `service` local data and select the **Log signal data** check box.

2 Open the properties dialog box for the `Dining_area` state, select the **Log signal data** check box, and change the logging name to `Dining_Room`.

See Also.

- "Setting Data Properties in the Data Dialog Box" on page 8-5
- "Changing State Properties" on page 4-15

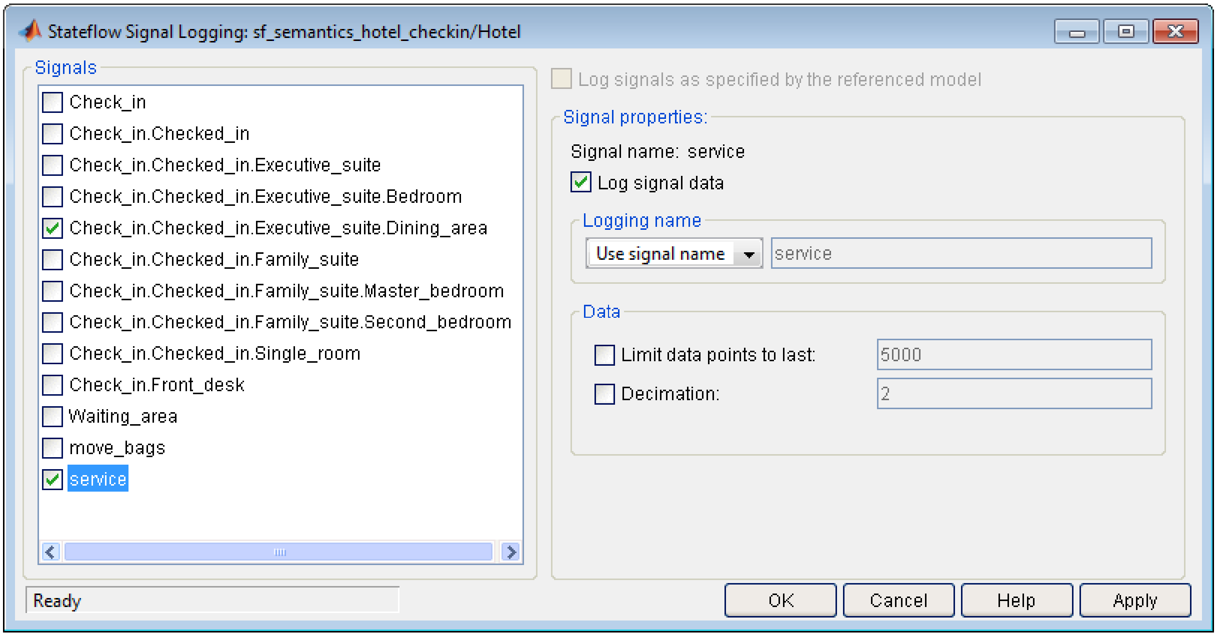
Logging Multiple Signals At Once

1 In the model, right-click the Stateflow chart and select **Log Chart Signals**.

The Stateflow Signal Logging dialog box opens, showing all states and local data. These chart objects are the signals you can log.

2 Select the check box next to each signal you want to log.

The **Log signal data** check box is selected automatically for each signal you log. For example, in the `Hotel` chart of the `sf_semantics_hotel_checkin` model, log the `Check_in.Checked_in.Executive_suite.Dining_area` state and local variable `service`:



Stateflow Signal Logging: sf_semantics_hotel_checkin/Hotel

Signals

☐ Check_in
☐ Check_in.Checked_in
☐ Check_in.Checked_in.Executive_suite
☐ Check_in.Checked_in.Executive_suite.Bedroom
☑ Check_in.Checked_in.Executive_suite.Dining_area
☐ Check_in.Checked_in.Family_suite
☐ Check_in.Checked_in.Family_suite.Master_bedroom
☐ Check_in.Checked_in.Family_suite.Second_bedroom
☐ Check_in.Checked_in.Single_room
☐ Check_in.Front_desk
☐ Waiting_area
☐ move_bags
☑ service

☐ Log signals as specified by the referenced model

Signal properties:

Signal name: service

☑ Log signal data

Logging name

Use signal name: service

Data

☐ Limit data points to last: 5000

☐ Decimation: 2

Ready

3 For each signal you select, modify the properties of what gets captured in the log.

For example, change the logging name of `Check_in.Checked_in.Executive_suite.Dining_area` to `Dining_Room`.

For a description of each property, see “Properties to Configure for Logging” on page 26-58.

Logging Chart Signals Using the Command-Line API

1 Open the model that contains the chart.

For example, open the `sf_semantics_hotel_checkin` model, which has a chart called `Hotel`.

2 Get the states whose activity you want to log.

For example, get the `Dining_area` state in the `Hotel` chart:

```
rt = sfroot;
da_state = rt.find('-isa','Stateflow.State','Name','Dining_area');
```

Get the local data you want to log.

For example, get the `service` local data in the `Hotel` chart:

```
svc_data = rt.find('-isa','Stateflow.Data','Name','service');
```

3 Enable logging for states and data.

For example, enable logging for the `Dining_area` state and the `service` data:

```
da_state.LoggingInfo.DataLogging = 1;
svc_data.LoggingInfo.DataLogging = 1;
```

4 Modify logging properties as needed.

For example, change the logged name of the `Dining_area` state. By default, the logged name is the hierarchical signal name, which is `Check_in.Checked_in.Executive_suite.Dining_area`. To assign the shorter, custom name of `Dining_Room`:

```
% Enable custom naming
da_state.LoggingInfo.NameMode = 'Custom';

% Enter the custom name
da_state.LoggingInfo.UserSpecifiedLogName = 'Dining_Room';
```

See Also. "Logging Multiple Signals At Once" on page 26-60

Accessing Logged Data

- "Signal Logging Object" on page 26-63

- "Accessing Logged Data Saved in Dataset Format" on page 26-63
- "Accessing Logged Data Saved in ModelDataLogs Format" on page 26-66

Signal Logging Object

During simulation, Stateflow saves logged data in a signal logging object, which you can access in the MATLAB workspace. The type of signal logging object depends on the signal logging format that you choose.

Format	Signal Logging Object
`Dataset`	`Simulink.SimulationData.Dataset`
`ModelDataLogs`	`Simulink.ModelDataLogs`

The default name of the signal logging object is `logsout`.

See Also.

- "Enabling Signal Logging and Choosing a Logging Format" on page 26-56 to learn how to change the name of the signal logging object
- "Supported Formats for Logged Data" on page 26-57
- `Simulink.SimulationData.Dataset` reference page in the Simulink documentation
- `Simulink.ModelDataLogs` reference page in the Simulink documentation

Accessing Logged Data Saved in Dataset Format

1. View the signal logging object in the MATLAB environment.

   For example:

   a. Start simulating the `sf_semantics_hotel_checkin` model using the `Dataset` signal logging format.

   b. When the `Front_desk` state becomes active, check in to the hotel by toggling the first switch.

c When the `Bedroom` state in the `Executive_suite` state becomes active, order room service multiple times, for example, by toggling the second switch 10 times.

d Stop simulation.

e Enter:

```
logsout
```

Result:

```
Simulink.SimulationData.Dataset
Package: Simulink.SimulationData

Characteristics:
            Name: 'logsout'
  Total Elements: 2

Elements:
  1: 'Dining_Room'
  2: 'service'
```

The output indicates:

- `logsout` is a Simulink object of type `SimulationData.Dataset`.
- Two elements were logged.

2 Use the `getElement` method to access logged elements by index and by name.

For example:

- To access logged activity for the `Check_in.Checked_in.Executive_suite.Dining_area` state:

By:	**Enter:**
Index	`logsout.getElement(1)`
Name	`logsout.getElement('Dining_Room')`
Block path	**1** `logsout.getElement(1).BlockPath` Returns: • `Block Path:  'sf_semantics_hotel_checkin/Hotel'` • `SubPath:  'Check_in.Checked_in.Executive_suite.Dining_area'` **2** `bp = Simulink.BlockPath('sf_semantics_hotel_checkin/Hotel');` **3** `bp.SubPath = 'Check_in.Checked_in.Executive_suite.Dining_area';` **4** `logsout.getElement(bp)`

The result is a `Stateflow.SimulationData.State` object:

```
Stateflow.SimulationData.State
Package: Stateflow.SimulationData

Properties:
       Name: 'Dining_Room'
  BlockPath: [1x1 Simulink.SimulationData.BlockPath]
     Values: [1x1 timeseries]
```

- To access logged activity for the `service` local data:

By:	**Enter:**
Index	`logsout.getElement(2)`
Name	`logsout.getElement('service')`
Block path	**1** `logsout.getElement(2).BlockPath` Returns: • `Block Path:  'sf_semantics_hotel_checkin/Hotel'` • `SubPath:  'service'` **2** `bp = Simulink.BlockPath('sf_semantics_hotel_checkin/Hotel');` **3** `bp.SubPath = 'service';` **4** `logsout.getElement(bp)`

The result is a `Stateflow.SimulationData.Data` object:

```
  Stateflow.SimulationData.Data
 Package: Stateflow.SimulationData

 Properties:
        Name: 'service'
   BlockPath: [1x1 Simulink.SimulationData.BlockPath]
      Values: [1x1 timeseries]
```

The logged values for `Stateflow.SimulationData.State` and `Stateflow.SimulationData.Data` objects are stored in the `Values` property as Simulink objects of type `Timeseries`.

3 Access logged data and time through the `Values` property.

For example:

For:	Enter:
Data	`logsout.getElement(1).Values.Data;`
Time	`logsout.getElement(1).Values.Time;`

4 View the logged data.

See "Viewing Logged Data" on page 26-68.

Accessing Logged Data Saved in ModelDataLogs Format

1 View the signal logging object in the MATLAB environment.

For example:

a Start simulating the `sf_semantics_hotel_checkin` model using the `ModelDataLogs` signal logging format.

b When the `Front_desk` state becomes active, check in to the hotel by toggling the first switch.

c When the `Bedroom` state in the `Executive_suite` state becomes active, order room service multiple times, for example, by toggling the second switch 10 times.

d Stop simulation.

e Enter:

```
logsout
```

Result:

```
 Simulink.ModelDataLogs (sf_semantics_hotel_checkin):
  Name                   Elements  Simulink Class

  Hotel                     2      StateflowDataLogs
```

The output indicates:

- `logsout` is a Simulink object of type `Simulink.ModelDataLogs`.
- Two elements were logged, all in the `Hotel` chart.

2 List the logged states and local data.

For example:

```
logsout.Hotel
```

Result:

```
Simulink.StateflowDataLogs (Hotel):
  Name                   Elements  Simulink Class

  service                   1      Timeseries
  Dining_Room               1      Timeseries
```

The logged signals are stored as Simulink objects of type `Timeseries`.

3 Use dot notation to access logged values for data or state activity.

Because of the way Simulink stores logged signals in `ModelDataLogs` format, you must use dot notation to access logged data for Stateflow objects below the chart level in the model hierarchy.

For example, to access logged values for `service` data, enter:

```
logsout.Hotel.service
```

Result:

```
       Name: 'service'
  BlockPath: 'sf_semantics_hotel_checkin/Hotel/service'
  PortIndex: 1
 SignalName: 'service'
 ParentName: 'service'
   TimeInfo: [1x1 Simulink.TimeInfo]
       Time: [560498x1 double]
       Data: [560498x1 double]
```

A structure stores all logging information. The logged data values reside in the `Data` field and the logged times reside in the `Time` field. The size of your `Time` and `Data` vectors might differ from those shown here, depending on how you interact with the switches in the model.

4. View the logged data.

   See "Viewing Logged Data" on page 26-68.

Viewing Logged Data

Use one of the following approaches to view logged data.

Approach	**Action**
View logged data in a figure window	Use the `plot` function for the `Timeseries` object. For example, in the `Hotel` chart, plot logged values over time in `ModelDataLogs` format for the local variable `service` by entering: `logsout.Hotel.service.plot`
View logged data in a spreadsheet	Pass a numeric, cell, or logical array of logged values to the `xlswrite` function. For example, in the `Hotel` chart, to view logged activity over time in `Dataset` format for the `Check_in.Checked_in.Executive_suite.Dining_area` state: 1 Assign logged `Dining_Room` time and data values to an array A:

```
A = [logsout.getElement('Dining_Room').Values.Time ...
    logsout.getElement('Dining_Room').Values.Data];
```

Approach	Action
	2 Export the data to an Excel® file named `dining_log.xls`: `xlswrite('dining_log.xls',A);` **3** Open `dining_log.xls` in Excel.

Logging Data in Library Charts

- "How Library Log Settings Influence Linked Instances" on page 26-69
- "Overriding Logging Properties in Chart Instances" on page 26-69
- "Example: Override Logging Properties in Atomic Subcharts" on page 26-70

How Library Log Settings Influence Linked Instances

Chart instances inherit logging properties from the library chart to which they are linked. You can override logging properties in the instance, but only for signals you select in the library. You cannot select additional signals to log from the instance.

Overriding Logging Properties in Chart Instances

To override properties of logged signals in chart instances, use one of the following approaches.

Approach	How To Use
Simulink Signal Logging Selector dialog box	See "Overriding Logging Properties with the Logging Selector" on page 26-70
Command-line interface	See "Overriding Logging Properties with the Command-Line API" on page 26-72

Example: Override Logging Properties in Atomic Subcharts

This example uses `sf_atomic_sensor_pair`. This model simulates a redundant sensor pair as atomic subcharts `Sensor1` and `Sensor2` in the chart `RedundantSensors`. Each atomic subchart contains instances of the states `Fail`, `FailOnce`, and `OK` from the library chart `sf_atomic_sensor_lib`.

Overriding Logging Properties with the Logging Selector.

1. Open the example library by typing `sf_atomic_sensor_lib` at the MATLAB command prompt.

2. In the library, right click `SingleSensor` and select **Log Chart Signals**.

3. In Stateflow Signal Logging dialog box, set the following logging properties, then click **OK**.

For Signal:	What to Specify:
`Fail`	• Select the **Log signal data** check box. • Change **Logging name** to the custom name `LogFail`. • Click **Apply**.
`FailOnce`	• Select the **Log signal data** check box. • Change **Logging name** to the custom name `LogFailOnce`. • Click **Apply**.
`OK`	• Select the **Log signal data** check box. • Change **Logging name** to the custom name `LogOK`. • Click **Apply**.

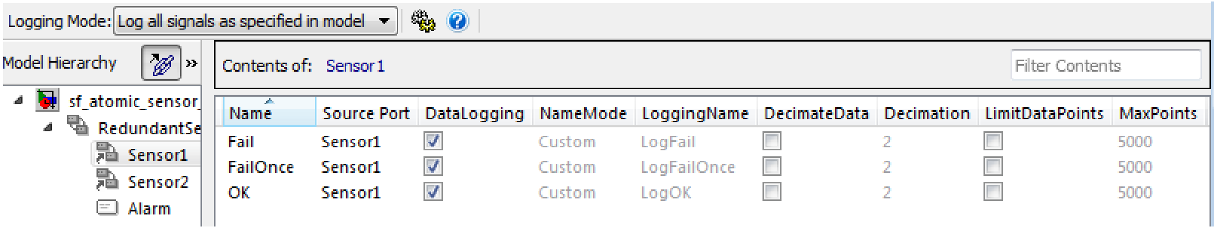
4. Open the model that contains instances of the library chart by typing `sf_atomic_sensor_pair` at the MATLAB command prompt.

5. Open the `RedundantSensors` chart and select **Simulation > Configuration Parameters**.

6 In the **Data Import/Export** pane, click **Configure Signals to Log** to open the Simulink Signal Logging Selector.

7 In the **Model Hierarchy** pane, expand `RedundantSensors`, and click `Sensor1` and `Sensor2`.

Each instance inherits logging properties from the library chart. For example:



Logging Mode: Log all signals as specified in model

Contents of: Sensor1

Name	Source Port	DataLogging	NameMode	LoggingName	DecimateData	Decimation	LimitDataPoints	MaxPoints
Fail	Sensor1	☑	Custom	LogFail	☐	2	☐	5000
FailOnce	Sensor1	☑	Custom	LogFailOnce	☐	2	☐	5000
OK	Sensor1	☑	Custom	LogOK	☐	2	☐	5000

8 Now, override some logging properties for `Sensor1`:

a In the **Model Hierarchy** pane, select `Sensor1`.

b Change **Logging Mode** to `Override signals`.

The selector clears all **DataLogging** check boxes for the model.

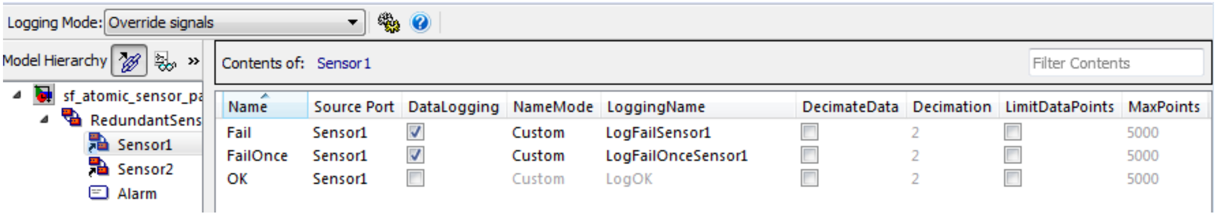
c Enable logging only for the `Fail` and `FailOnce` states in `Sensor1`:

Select **DataLogging** for these two signals. Leave **DataLogging** cleared for the `OK` signal.

d Append the string `Sensor1` to the logging names for `Fail` and `FailOnce`:

Double-click the logging names for signals `Fail` and `FailOnce`, and rename them `LogFailSensor1` and `LogFailOnceSensor1`, respectively.

The settings should look like this:



Logging Mode: Override signals

Contents of: Sensor1

Name	Source Port	DataLogging	NameMode	LoggingName	DecimateData	Decimation	LimitDataPoints	MaxPoints
Fail	Sensor1	☑	Custom	LogFailSensor1	☐	2	☐	5000
FailOnce	Sensor1	☑	Custom	LogFailOnceSensor1	☐	2	☐	5000
OK	Sensor1	☐	Custom	LogOK	☐	2	☐	5000

Overriding Logging Properties with the Command-Line API.

1. Open the example library by typing `sf_atomic_sensor_lib` at the MATLAB command prompt.

2. Log the signals `Fail`, `FailOnce`, and `OK` in the `SingleSensor` chart using these commands:

```
% Get states in the SingleSensor chart
rt=sfroot;
states = rt.find('-isa', 'Stateflow.State');

% Enable logging for each state
for i = 1: length(states)
   states(i).LoggingInfo.DataLogging = 1;
end
```

3. Open the model that contains instances of the library chart by typing `sf_atomic_sensor_pair` at the MATLAB command prompt.

4. Create a `ModelLoggingInfo` object for the model.

   This object contains a vector `Signals` that stores all logged signals.

```
mi = Simulink.SimulationData.ModelLoggingInfo. ...
createFromModel('sf_atomic_sensor_pair')
```

   The result is:

```
mi =

  Simulink.SimulationData.ModelLoggingInfo
  Package: Simulink.SimulationData

  Properties:
                     Model: 'sf_atomic_sensor_pair'
               LoggingMode: 'OverrideSignals'
    LogAsSpecifiedByModels: {}
                   Signals: [1x6 Simulink.SimulationData.SignalLoggingInfo]
```

The `Signals` vector contains the signals marked for logging in the library chart:

- Library instances of `Fail`, `FailOnce`, and `OK` states in atomic subchart `Sensor1`
- Library instances of `Fail`, `FailOnce`, and `OK` states in atomic subchart `Sensor2`

5 Make sure that `LoggingMode` equals `'OverrideSignals'`.

6 Create a block path to each logged signal whose properties you want to override.

To access signals inside Stateflow charts, use `Simulink.SimulationData.BlockPath(paths, subpath)`, where `subpath` represents a signal inside the chart.

To create block paths for the signals `Fail`, `FailOnce`, and `OK` in the atomic subchart `Sensor1` in the `RedundantSensors` chart:

```
failPath = Simulink.SimulationData. ...
BlockPath('sf_atomic_sensor_pair/RedundantSensors/Sensor1','Fail')

failOncePath = Simulink.SimulationData. ...
BlockPath('sf_atomic_sensor_pair/RedundantSensors/Sensor1','FailOnce')

OKPath = Simulink.SimulationData. ...
BlockPath('sf_atomic_sensor_pair/RedundantSensors/Sensor1','OK')
```

7 Get the index of each logged signal in the `Simulink.SimulationData.BlockPath` object.

To get the index for the signals `Fail`, `FailOnce`, and `OK`:

```
failidx = mi.findSignal(failPath);
failOnceidx = mi.findSignal(failOncePath);
OKidx = mi.findSignal(OKPath);
```

8 Override some logging properties for the signals in `Sensor1`:

a Disable logging for signal `OK`:

```
mi.Signals(OKidx).LoggingInfo.DataLogging = 0;
```

b Append the string `Sensor1` to the logging names for `Fail` and `FailOnce`:

```
% Enable custom naming
mi.Signals(failidx).LoggingInfo.NameMode = 1;
mi.Signals(failOnceidx).LoggingInfo.NameMode = 1;

% Enter the custom name
mi.Signals(failidx).LoggingInfo.LoggingName = 'LogFailSensor1';
mi.Signals(failOnceidx).LoggingInfo.LoggingName = 'LogFailOnceSensor1';
```

9 Apply the changes:

```
set_param(bdroot, 'DataLoggingOverride', mi);
```

See Also

- `Simulink.SimulationData.ModelLoggingInfo`
- `Simulink.SimulationData.BlockPath`

How Stateflow Logs Multidimensional Data

Stateflow logs each update to a multidimensional signal as a single change. For example, an update to a 2-by-2 matrix `A` during simulation is logged as a single change, not as four changes (one for each element):

Update	Is Logged As
`A = 1;`	A single change, even though the statement implies all `A[i] = 1`
`A[1][1] = 1;` `A[1][2] = 1;`	Two different changes

Limitations on Logging Data

You cannot log bus data.

Exploring and Modifying Charts

Using the Model Explorer with Stateflow Objects

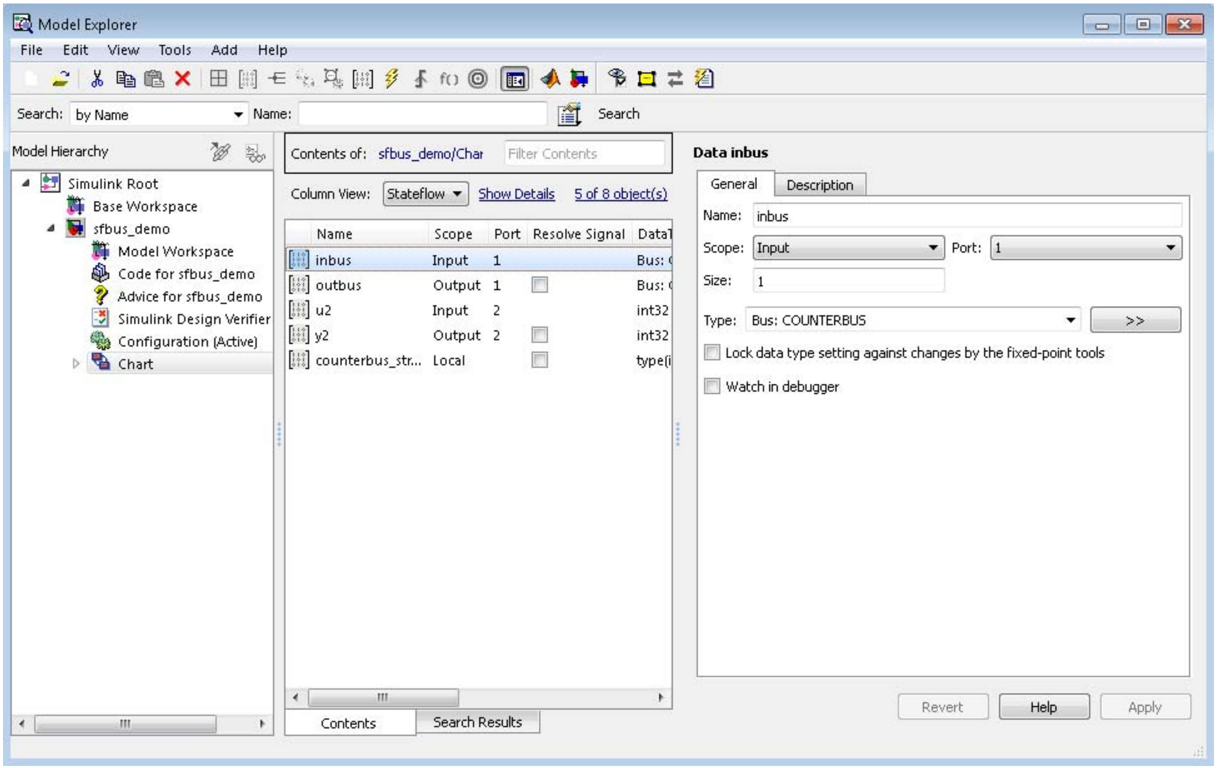
In this section...

Viewing Stateflow Objects in the Model Explorer

You can use one of these methods for opening the Model Explorer:

- In the model window, select **View > Model Explorer**.
- Right-click an empty area in the chart and select **Explore**.
- In the Truth Table Editor, select **Tools > Explore**.

The Model Explorer appears something like this:



Contents of: sfbus_demo/Char

Column View: Stateflow Show Details 5 of 8 object(s)

Name	Scope	Port	Resolve Signal	DataT
inbus	Input	1		Bus:
outbus	Output	1	☐	Bus:
u2	Input	2		int32
y2	Output	2	☐	int32
counterbus_str...	Local		☐	type(i

Name: inbus

Scope: Input Port: 1

Size: 1

Type: Bus: COUNTERBUS

☐ Lock data type setting against changes by the fixed-point tools

☐ Watch in debugger

The main window has two panes: a **Model Hierarchy** pane on the left and a **Contents** pane on the right. When you open the Model Explorer, the Stateflow object you are editing appears highlighted in the **Model Hierarchy** pane and its objects appear in the **Contents** pane. This example shows how the Model Explorer appears when opened from the chart.

The **Model Hierarchy** pane displays the elements of all loaded Simulink models, which includes Stateflow charts. A preceding plus (+) character for an object indicates that you can expand the display of its child objects by double-clicking the entry or by clicking the plus (+). A preceding minus (-) character for an object indicates that it has no child objects.

Clicking an entry in the **Model Hierarchy** pane selects that entry and displays its child objects in the **Contents** pane. A hypertext link to the currently selected object in the **Model Hierarchy** pane appears after the **Contents of:** label at the top of the **Contents** pane. Click this link to display that object in its native editor. In the preceding example, clicking the link `sfbus_demo/Chart` displays the contents of the chart in its editor.

Each type of object, whether in the **Model Hierarchy** or **Contents** pane, appears with an adjacent icon. Subcharted objects (states, boxes, or graphical functions) appear altered with shading.

The display of child objects in the **Contents** pane includes properties for each object, most of which are directly editable. You can also access the properties dialog box for an object from the Model Explorer. See "Setting Properties for Chart Objects in the Model Explorer" on page 27-8 for more details.

Editing Chart Objects in the Model Explorer

To edit a chart object that appears in the **Model Hierarchy** pane of the Model Explorer:

1. Right-click the object.

2. Select **Open** from the context menu.

   The selected object appears highlighted in the chart.

Adding Data and Events in the Model Explorer

Charts, states, subcharts, boxes, and functions can parent data and events. To add data or events to a Stateflow object:

1. In the **Model Hierarchy** pane of the Model Explorer, select a chart, state, subchart, box, or function.

2. From the **Add** menu, select **Data** or **Event**.

   A new data or event appears in the **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 properties for data or events directly in the Model Explorer. You can also access the complete list of properties for data or events from the Model Explorer. See "Setting Properties for Chart Objects in the Model Explorer" on page 27-8.

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.

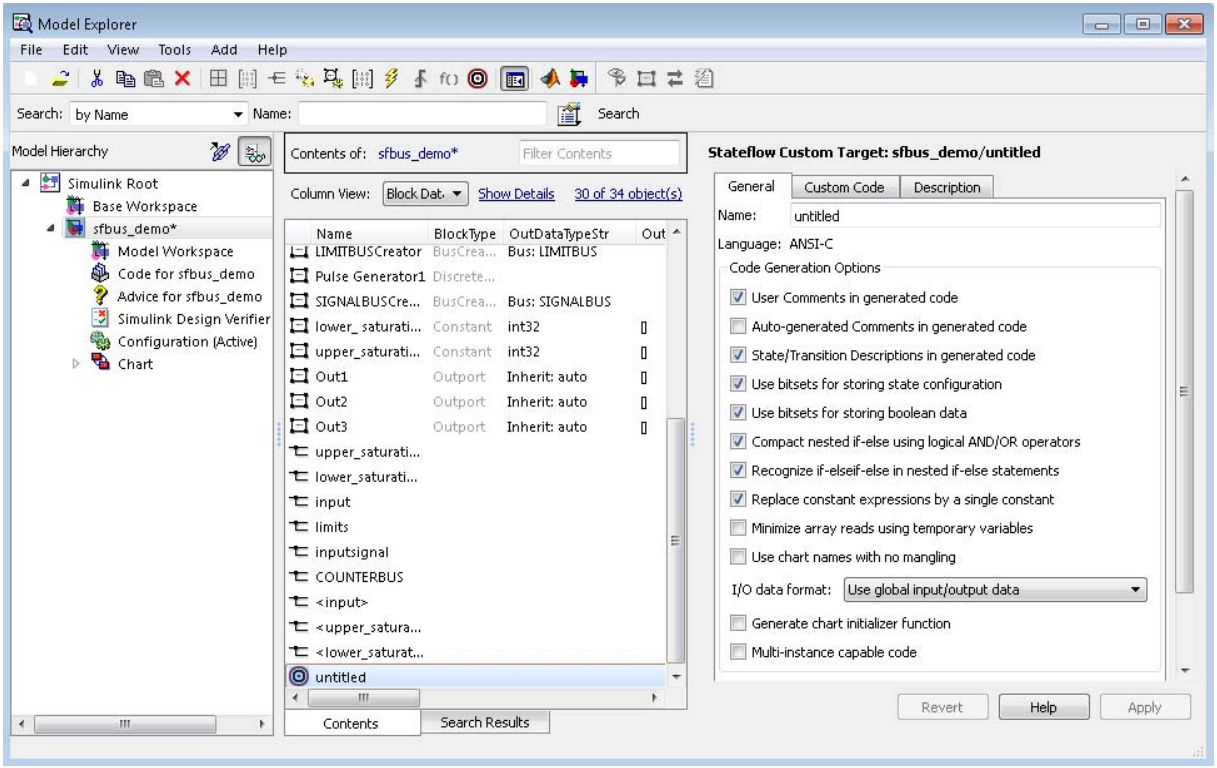
Adding Custom Targets in the Model Explorer

Custom targets belong to a model, not a chart. 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`.



Contents of: sfbus_demo*

Column View: Block Dat

30 of 34 object(s)

Name	BlockType	OutDataTypeStr	Out
LIMITBUSCreator	BusCrea...	Bus: LIMITBUS	
Pulse Generator1	Discrete...		
SIGNALBUSCre...	BusCrea...	Bus: SIGNALBUS	
lower_saturati...	Constant	int32	[]
upper_saturati...	Constant	int32	[]
Out1	Outport	Inherit: auto	[]
Out2	Outport	Inherit: auto	[]
Out3	Outport	Inherit: auto	[]
upper_saturati...			
lower_saturati...			
input			
limits			
inputsignal			
COUNTERBUS			
<input>			
<upper_satura...			
<lower_saturat...			
untitled			

Stateflow Custom Target: sfbus_demo/untitled

Name: untitled

Language: ANSI-C

Code Generation Options

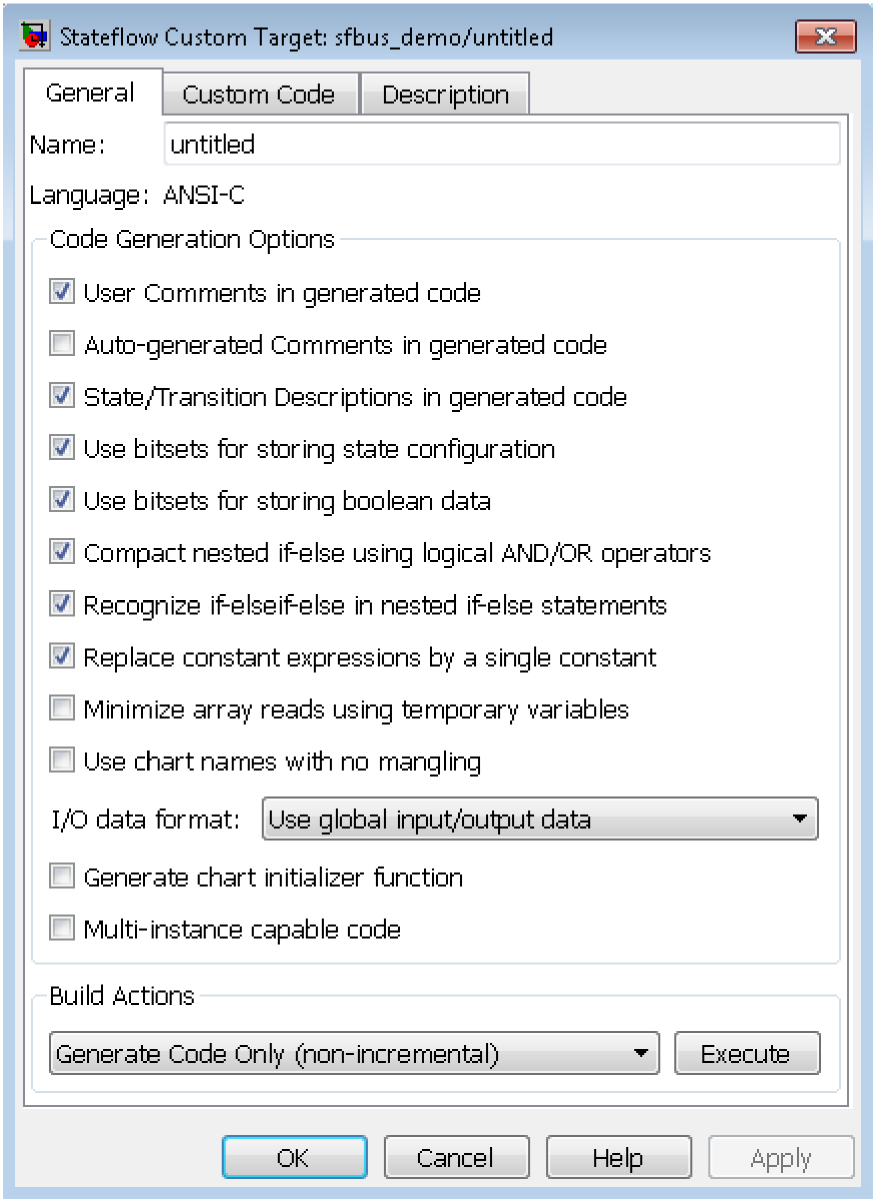
- ☑ User Comments in generated code
- ☐ Auto-generated Comments in generated code
- ☑ State/Transition Descriptions in generated code
- ☑ Use bitsets for storing state configuration
- ☑ Use bitsets for storing boolean data
- ☑ Compact nested if-else using logical AND/OR operators
- ☑ Recognize if-elseif-else in nested if-else statements
- ☑ Replace constant expressions by a single constant
- ☐ Minimize array reads using temporary variables
- ☐ Use chart names with no mangling

I/O data format: Use global input/output data

- ☐ Generate chart initializer function
- ☐ Multi-instance capable code

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.



Stateflow Custom Target: sfbus_demo/untitled

Name: untitled

Language: ANSI-C

Code Generation Options

☑ User Comments in generated code

☐ Auto-generated Comments in generated code

☑ State/Transition Descriptions in generated code

☑ Use bitsets for storing state configuration

☑ Use bitsets for storing boolean data

☑ Compact nested if-else using logical AND/OR operators

☑ Recognize if-elseif-else in nested if-else statements

☑ Replace constant expressions by a single constant

☐ Minimize array reads using temporary variables

☐ Use chart names with no mangling

I/O data format: Use global input/output data

☐ Generate chart initializer function

☐ Multi-instance capable code

Build Actions

Generate Code Only (non-incremental)

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 dialog box.

6 Click **Apply**.

For more information, see “How to Build a Stateflow Custom Target” on page 25-53.

Renaming Objects in the Model Explorer

To rename a chart object in the Model Explorer:

1 Right-click the object row in the **Contents** pane of the Model Explorer and select **Rename**.

   The name of the selected object appears in a text edit box.

2 Change the name of the object and click outside the edit box.

Setting Properties for Chart Objects in the Model Explorer

To change the property of an object in the **Contents** pane of the Model Explorer:

1 In the **Contents** pane, click in the row of the displayed object.

2 Click an individual entry for a property column in the highlighted row.

   - For text properties, such as the Name property, a text editing field with the current text value overlays the displayed value. Edit the field and press the **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 and select **Properties**.

   The properties dialog box for the object appears.

2. Edit the appropriate properties and click **Apply** or **OK**.

To display the properties dialog box dynamically for the selected object in the **Model Hierarchy** or **Contents** pane of the Model Explorer:

1. Select **View > Show Dialog Pane**.

   The properties dialog box for the selected object appears in the far right pane of the Model Explorer.

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** automatically changes to **Local**.

To move data, event, or target objects to another parent:

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.

To cut or copy the selected data, events, and targets:

1 Select the event, data, and targets to cut or copy in the **Contents** pane of the Model Explorer.

2 In the Model Explorer, select **Edit > Cut** or **Edit > Copy**.

  If you select **Cut**, the selected items are deleted and then copied to the clipboard for copying elsewhere. If you select **Copy**, the selected items are left unchanged.

  You can also right-click a single selection and select **Cut** or **Copy** from the context menu. The Model Explorer also uses the keyboard equivalents of **Ctrl+X** (Cut) and **Ctrl+C** (Copy) on a computer running the UNIX or Windows operating system.

3 Select a new parent object in the **Model Hierarchy** pane of the Model Explorer.

4 Select **Edit > Paste**. The cut items appear in the **Contents** pane of the Model Explorer.

  You can also paste the cut items by right-clicking an empty part of the **Contents** pane and selecting **Paste** from the context menu. The Model Explorer also uses the keyboard equivalent of **Ctrl+V** (Paste) on a computer running the UNIX or Windows operating system.

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 data, event, 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 **Edit > Cut** or **Ctrl+X** from the keyboard to delete an object.

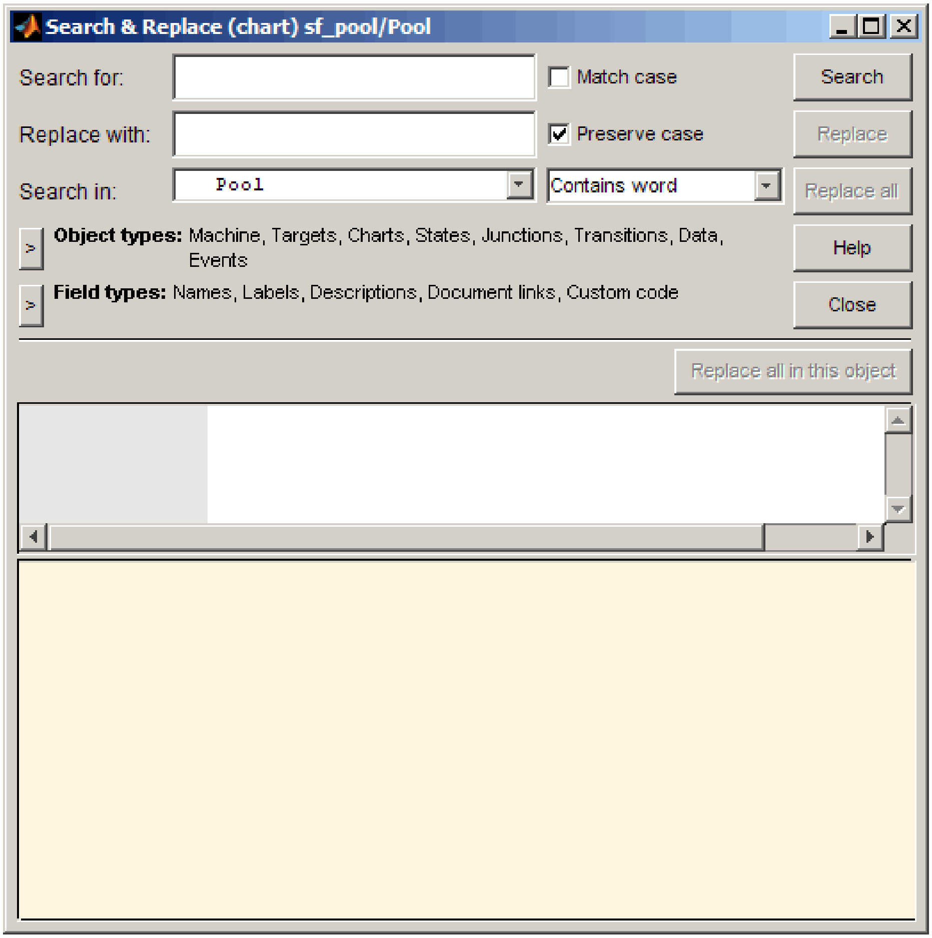
Using the Search & Replace Tool

In this section...

Opening the Search & Replace Tool

To open the Search & Replace dialog box:

1. Open a chart.
2. Select **Tools > Search & Replace**.



Search & Replace (chart) sf_pool/Pool

Search for:

Replace with:

Search in: `Pool`

Contains word

☐ Match case

☑ Preserve case

Object types: Machine, Targets, Charts, States, Junctions, Transitions, Data, Events

Field types: Names, Labels, Descriptions, Document links, Custom code

The Search & Replace dialog box contains the following fields:

- **Search for**

  Enter search pattern text in the **Search for** text box. You can select the interpretation of the search pattern with the **Match case** check box and the **Match options** field (unlabeled and just to the right of the **Search in** field).

- **Match case**

  If you select this check box, the search is case sensitive and the Search & Replace tool finds only text matching the search pattern exactly. See "Match case (Case Sensitive)" on page 27-15.

- **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 27-24.

- **Preserve case**

  This option modifies replacement text. For an understanding of this option, see "Replacing with Case Preservation" on page 27-23.

- **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 27-15.

- **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 appear next to their titles.

- **Search** and **Replace** buttons

  These are described in “Using the Search Button and View Area” on page 27-19 and “Using the Replace Buttons” on page 27-24.

- **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 27-20.

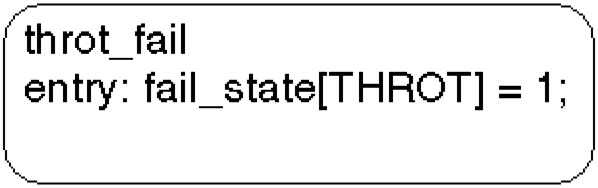
Using Different Search Types

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 that you have a state with this label and entry action:



```
throt_fail
entry: fail_state[THROT] = 1;
```

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 27-24 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 27-23.

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.

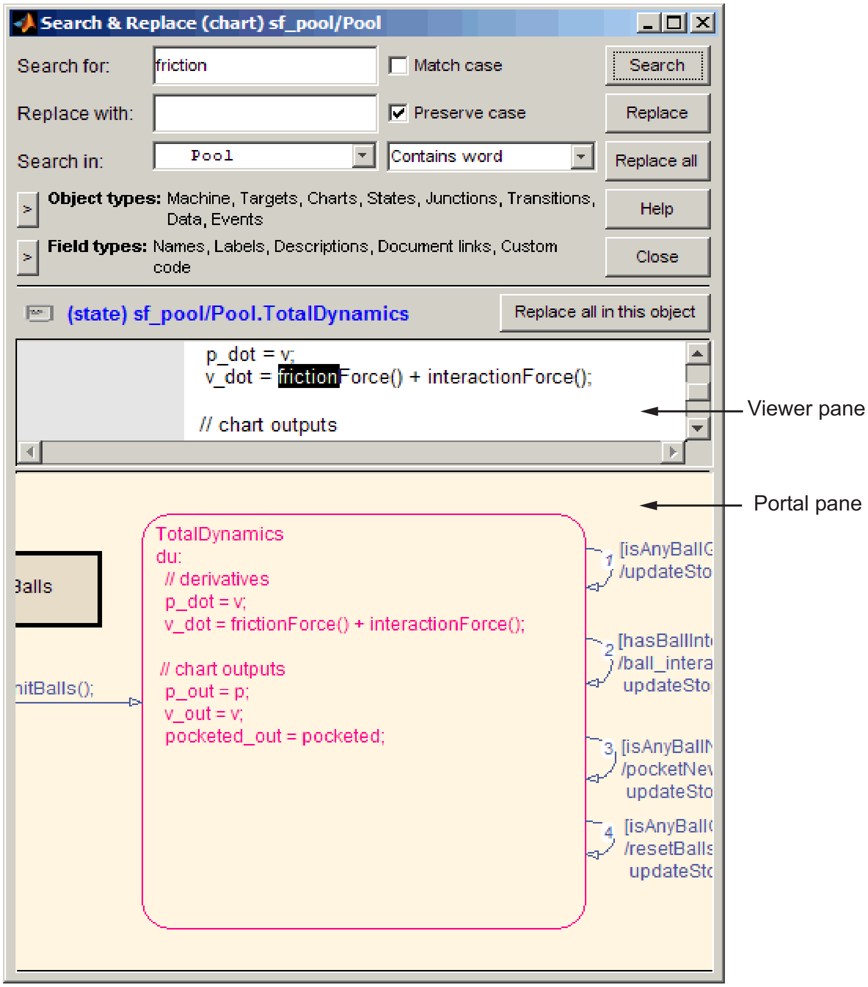
Custom code. Only target objects contain custom code.

Using the Search Button and View Area

This topic contains the following subtopics:

- “A Breakdown of the View Area” on page 27-20
- “The Search Order” on page 27-21
- “Additional Display Options” on page 27-22

Click **Search** to initiate a single-search operation. If an object match is made, its text fields appear in the **Viewer** pane in the middle of the Search & Replace dialog box. If the object is graphical (state, transition, junction, chart), the matching object appears highlighted in a **Portal** pane below the **Viewer** pane.



Search for: friction

Replace with:

☐ Match case

☑ Preserve case

Search in: Pool

Contains word

Object types: Machine, Targets, Charts, States, Junctions, Transitions, Data, Events

Field types: Names, Labels, Descriptions, Document links, Custom code

(state) sf_pool/Pool.TotalDynamics

```
p_dot = v;
v_dot = frictionForce() + interactionForce();

// chart outputs
```

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

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

Viewer pane

Portal pane

A Breakdown of the View Area

The view area of the Search & Replace dialog box displays matching text and its containing object, if viewable. In the previous example, taken from the `sf_pool` model, a search for the word `"friction"` finds the **Description** field for the state `TotalDynamics`. The resulting view area consists of these parts:

Icon. Displays an icon appropriate to the object containing the matching text. These icons are identical to the icons in the Model Explorer that represent Stateflow objects displayed in "Viewing Stateflow Objects in the Model Explorer" on page 27-2.

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.

ℹ – 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.

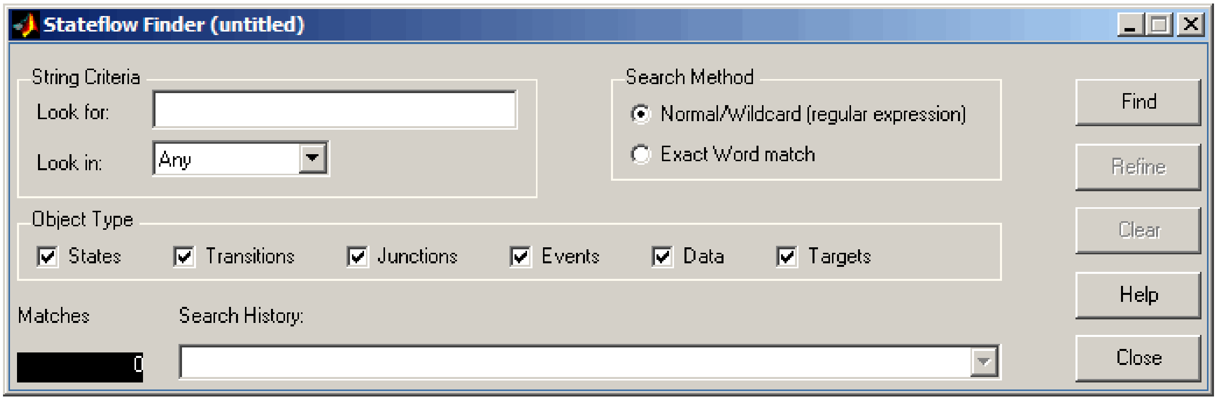
- On most platforms, when you select **Tools > Find** in the Stateflow Editor, the Simulink Find dialog box appears. You can use this tool to search for Simulink and Stateflow objects that meet criteria you specify. Any objects that meet your criteria appear in the search results pane of the dialog box.

  For details, see "The Finder" in the Simulink documentation.

- On platforms that do not support the Simulink Find tool, the original Stateflow Finder appears when you right-click inside a chart and select **Find**.

Note If you launch a MATLAB session by typing `matlab nojvm`, the original Stateflow Finder appears when you right-click inside a chart and select **Find**.

Using the Stateflow Finder



- "String Criteria" on page 27-29
- "Search Method" on page 27-30
- "Object Type" on page 27-31
- "Matches" on page 27-31
- "Search History" on page 27-31
- "Find Button" on page 27-31
- "Refine Button" on page 27-32
- "Clear Button" on page 27-32
- "Help Button" on page 27-32
- "Close Button" on page 27-32

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

Character	Description
+	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 Type** options include:

- States
- Transitions
- Junctions
- Events
- Data
- Targets

Matches

The **Matches** field displays the number of objects that match the 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.

Find Button

Click the **Find** button to initiate the search operation. The results appear in the display area.

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.

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.

Help Button

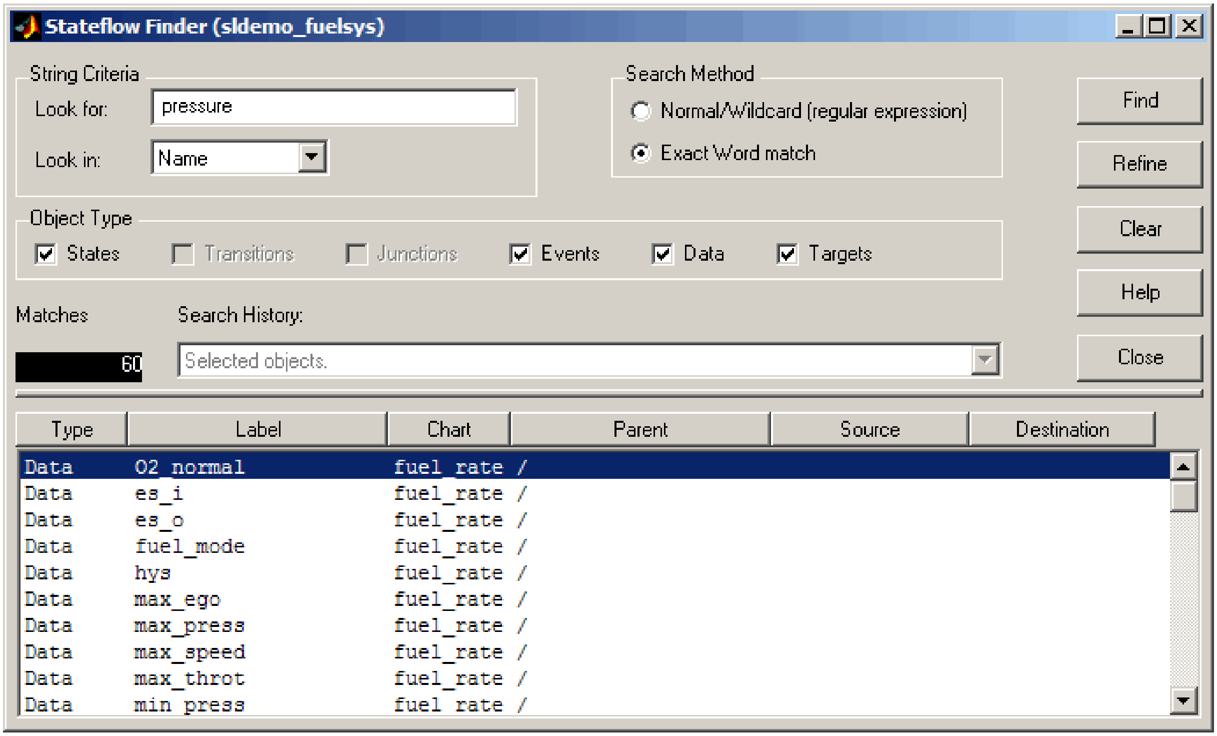
Click **Help** to display the Stateflow software documentation in an HTML browser window.

Close Button

Click **Close** to close the Finder.

Finder Display Area

The Stateflow Finder display area looks something like this.



Stateflow Finder (sldemo_fuelsys)

String Criteria

Look for: pressure

Look in: Name

Search Method

○ Normal/Wildcard (regular expression)

● Exact Word match

Object Type

☑ States ☐ Transitions ☐ Junctions ☑ Events ☑ Data ☑ Targets

Matches: 60

Search History: Selected objects.

Type	Label	Chart	Parent	Source	Destination
Data	O2_normal	fuel_rate /			
Data	es_i	fuel_rate /			
Data	es_o	fuel_rate /			
Data	fuel_mode	fuel_rate /			
Data	hys	fuel_rate /			
Data	max_ego	fuel_rate /			
Data	max_press	fuel_rate /			
Data	max_speed	fuel_rate /			
Data	max_throt	fuel_rate /			
Data	min_press	fuel_rate /			

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.
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.

A

Semantic Rules Summary

Summary of Chart Semantic Rules

In this section...

Entering a Chart

The set of default flow paths execute (see "Executing a Set of Flow Graphs" on page A-4). If this action does not cause a state entry and the chart has parallel decomposition, then each parallel state becomes active (see "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 execute. Parallel states execute in the same order that they become active.

Entering a State

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

2. If this state is a parallel state, check that all siblings with a higher (that is, earlier) entry order are active. If not, perform steps 1 through 5 for these states first.

Parallel (AND) states are ordered for entry based on whether you use explicit ordering (default) or implicit ordering. For details, see "Explicit Ordering of Parallel States" on page 3-76 and "Implicit Ordering of Parallel States" on page 3-77.

3. Mark the state active.

4. Perform any entry actions.

5. Enter children, if needed:

   a. If the state contains a history junction and there was an active child of this state at some point after the most recent chart initialization, perform the entry actions for that child. Otherwise, execute the default flow paths for the state.

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

   c. If this state has only one child substate, the substate becomes active when the parent becomes active, regardless of whether a default transition is present. Entering the parent state automatically makes the substate active. The presence of any inner transition has no effect on determining the active substate.

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

7. If the transition path parent is not the same as the parent of the current state, perform entry steps 6 and 7 for the immediate parent of this state.

Executing an Active State

1. The set of outer flow graphs execute (see "Executing a Set of Flow Graphs" on page A-4). If this action causes a state transition, execution stops. (Note that this step never occurs for parallel states.)

2. During actions and valid on-event actions are performed.

3 The set of inner flow graphs execute. If this action does not cause a state transition, the active children execute, starting at step 1. Parallel states execute in the same order that they become active.

Exiting an Active State

1 If this is a parallel state, make sure that all sibling states that became active after this state have already become inactive. Otherwise, perform all exiting steps on those sibling states.

2 If there are any active children, perform the exit steps on these states in the reverse order that they became active.

3 Perform any exit actions.

4 Mark the state as inactive.

Executing a Set of Flow Graphs

Flow graphs execute 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 Ordering of a set of transition segments occurs.

2 While there are remaining segments to test, testing a segment for validity occurs. If the segment is invalid, testing of the next segment occurs. If the segment is valid, execution depends on the destination:

   States

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

   b The states that are the immediate children of the parent of the transition path exit (see “Exiting an Active State” on page A-4).

c The transition action from the final transition segment executes.

d The destination state becomes active (see “Entering a State” on page A-2).

Junctions with no outgoing transition segments

Testing stops without any state exits or entries.

Junctions with outgoing transition segments

Step 1 is repeated with the set of outgoing segments from the junction.

3 After testing all outgoing transition segments at a junction, backtrack the incoming transition segment that brought you to the junction and continue at step 2, starting with the next transition segment after the backtrack segment. The set of flow graphs finishes execution when testing of all starting transitions is complete.

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 executes (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 a direct event broadcast occurs using the `send()` function.)

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

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

Action Type	Early Return Logic
State Entry	If the state is no longer active at the end of the event broadcast, any remaining steps in entering a state do not occur.
State Exit	If the state is no longer active at the end of the event broadcast, any remaining exit actions and steps in state transitioning do not occur.

Action Type	Early Return Logic
State During	If the state is no longer active at the end of the event broadcast, any remaining steps in executing an active state do not occur.
Condition	If the origin state of the inner or outer flow 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 do not occur.
Transition	If the parent of the transition path is not active or if that parent has an active child, the remaining transition actions and state entry do not occur.

B

Semantic Examples

Categories of Semantic Examples

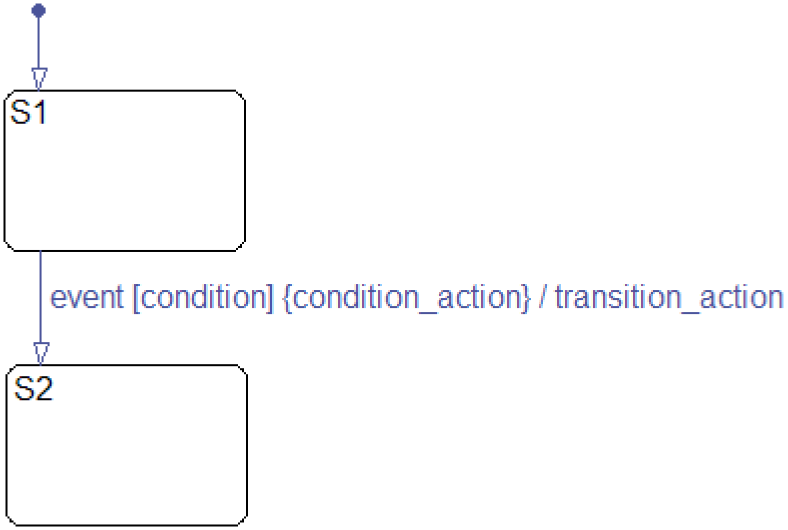
The following examples show the detailed semantics (behavior) of Stateflow charts.

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.



A chart executes this transition as follows:

1. When an event occurs, state `S1` checks for an outgoing transition with a matching event specified.

2 If a transition with a matching event is found, the condition for that transition (`[condition]`) is evaluated.

3 If the `condition` is true, `condition_action` is executed.

4 If there is a valid transition to the destination state, the transition is taken.

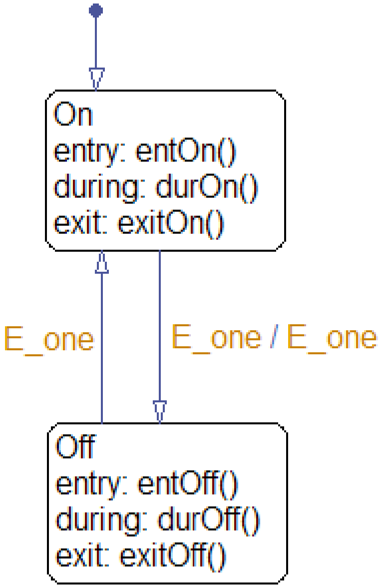
5 State `S1` is exited.

6 The `transition_action` is executed when the transition is taken.

7 State `S2` is entered.

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 chart is asleep. State `On` and state `Off` are OR states. State `On` is active. Event `E_one` occurs and awakens the chart, which processes the event from the root down through the hierarchy:

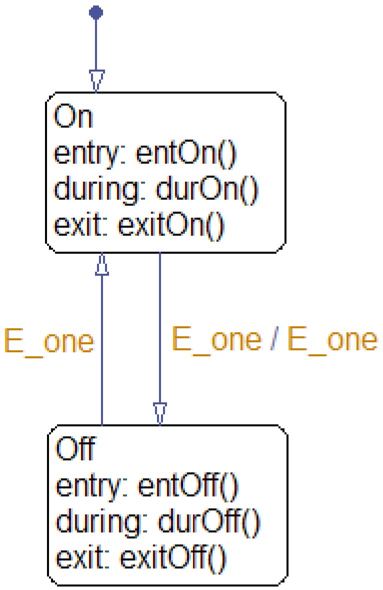
1. The chart root checks to see if there is a valid transition as a result of `E_one`. A valid transition from state `On` to state `Off` is detected.
2. State `On` exit actions (`exitOn()`) execute and complete.
3. State `On` is marked inactive.
4. The event `E_one` is broadcast as the transition action.

   This second event `E_one` is processed, but because neither state is active, it has no effect. If the second broadcast of `E_one` resulted in a valid transition, it would preempt the processing of the first broadcast of `E_one`. See “Early Return Logic for Event Broadcasts” on page 3-85.
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?



Initially, the chart is asleep. State `Off` is active. Event `E_one` occurs and awakens the chart, which processes the event from the root down through the hierarchy:

1. The chart root checks to see if there is a valid transition as a result of `E_one`.

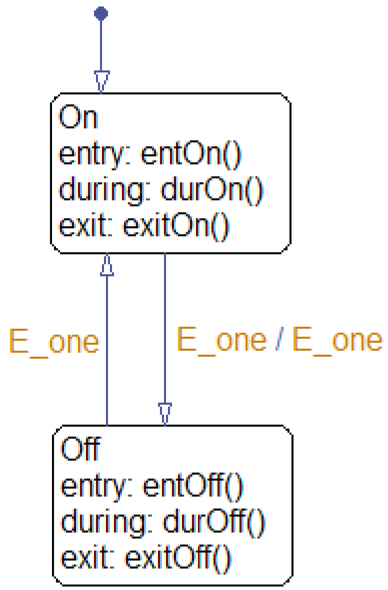
   A valid transition from state `Off` to state `On` is detected.

2. State `Off` exit actions (`exitOff()`) execute and complete.
3. State `Off` is marked inactive.
4. State `On` is marked active.
5. State `On` entry actions (`entOn()`) execute and complete.
6. The chart goes back to sleep.

This sequence completes the execution of the Stateflow chart associated with the second event `E_one` when state `Off` is initially active.

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 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.

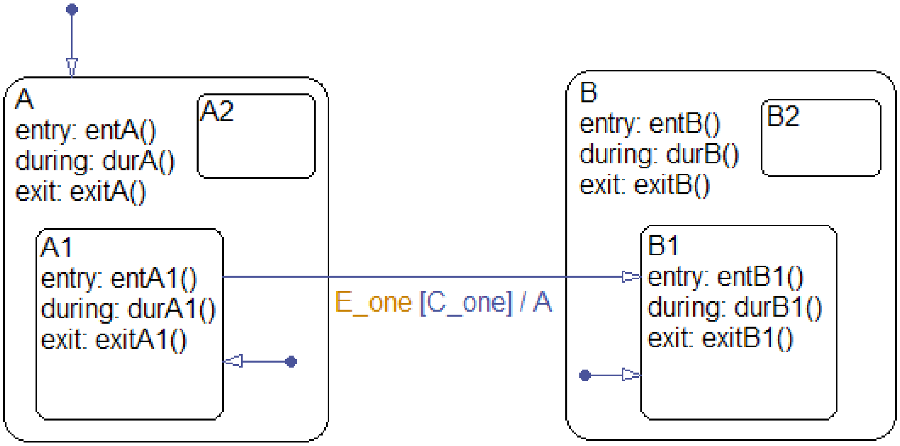
3. State `On` during actions (`durOn()`) execute and complete.

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 chart is asleep. State `A.A1` is active. Condition `C_one` is true. Event `E_one` occurs and awakens the chart, which processes the event from the root down through the hierarchy:

1. The chart root checks to see if there is a valid transition as a result of `E_one`. There is a valid transition from state `A.A1` to state `B.B1`. (Condition `C_one` is true.)
2. State `A` during actions (`durA()`) execute and complete.
3. State `A.A1` exit actions (`exitA1()`) execute and complete.
4. State `A.A1` is marked inactive.
5. State `A` exit actions (`exitA()`) execute and complete.

6. State `A` is marked inactive.
7. The transition action, `A`, is executed and completed.
8. State `B` is marked active.
9. State `B` entry actions (`entB()`) execute and complete.
10. State `B.B1` is marked active.
11. State `B.B1` entry actions (`entB1()`) execute and complete.
12. The chart goes back to sleep.

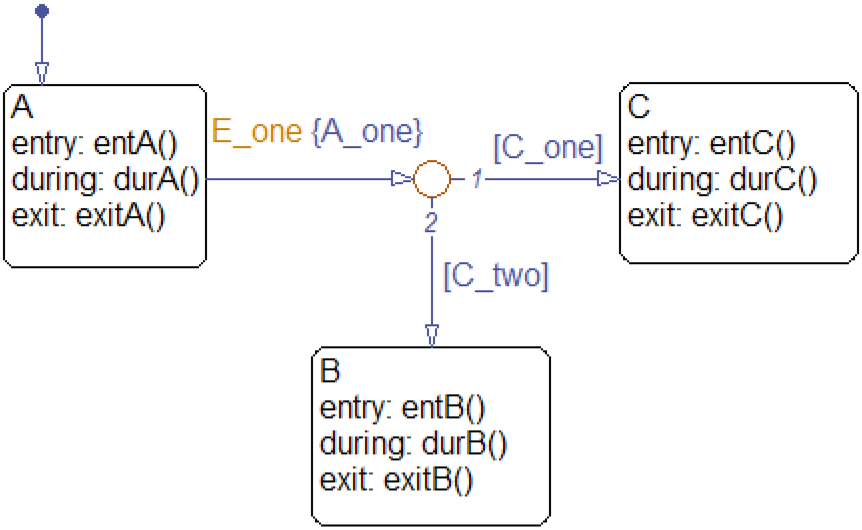
This sequence completes the execution of this Stateflow chart associated with event `E_one`.

Condition Action Examples

In this section...

Condition Action Example

This example shows the behavior of a simple condition action in a transition path with multiple segments. The chart uses implicit ordering of outgoing transitions (see "Implicit Ordering of Outgoing Transitions" on page 3-59).



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

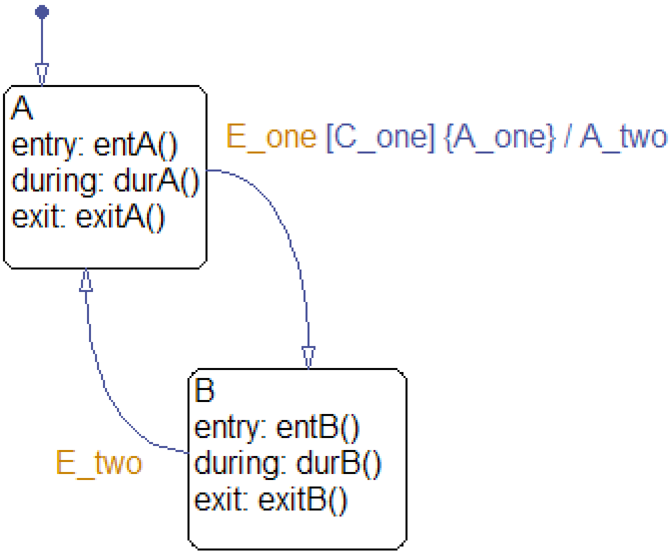
1. The chart root checks to see if there is a valid transition as a result of `E_one`. A valid transition segment from state `A` to a connective junction is detected. The condition action `A_one` is detected on the valid transition segment and is immediately executed and completed. State `A` is still active.
2. Because the conditions on the transition segments to possible destinations are false, none of the complete transitions is valid.
3. State `A` during actions (`durA()`) execute and complete.

   State `A` remains active.
4. The chart goes back to sleep.

This sequence completes the execution of this Stateflow chart associated with event `E_one` when state `A` is initially active.

Condition and Transition 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 chart is asleep. State `A` is active. Condition `C_one` is true. Event `E_one` occurs and awakens the chart, which processes the event from the root down through the hierarchy:

1 The chart root checks to see if there is a valid transition as a result of `E_one`. A valid transition from state `A` to state `B` is detected. The condition `C_one` is true. The condition action `A_one` is detected on the valid transition and is immediately executed and completed. State `A` is still active.

2 State `A` exit actions (`ExitA()`) execute and complete.

3 State `A` is marked inactive.

4 The transition action `A_two` is executed and completed.

5 State `B` is marked active.

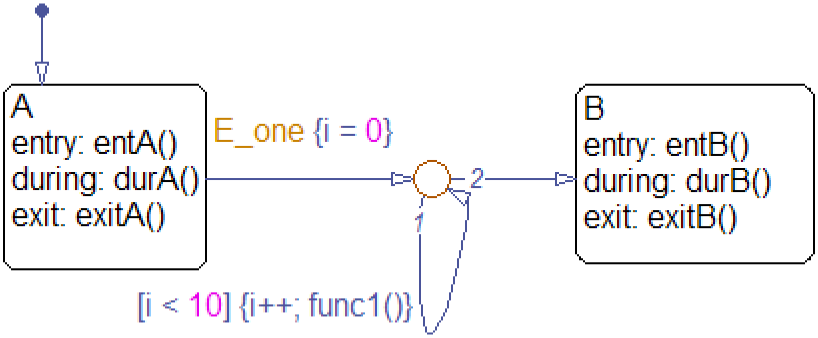
6 State `B` entry actions (`entB()`) execute and complete.

7 The chart goes back to sleep.

This sequence completes the execution of this Stateflow chart associated with event `E_one` when state `A` is initially active.

Condition Actions in For-Loop Construct Example

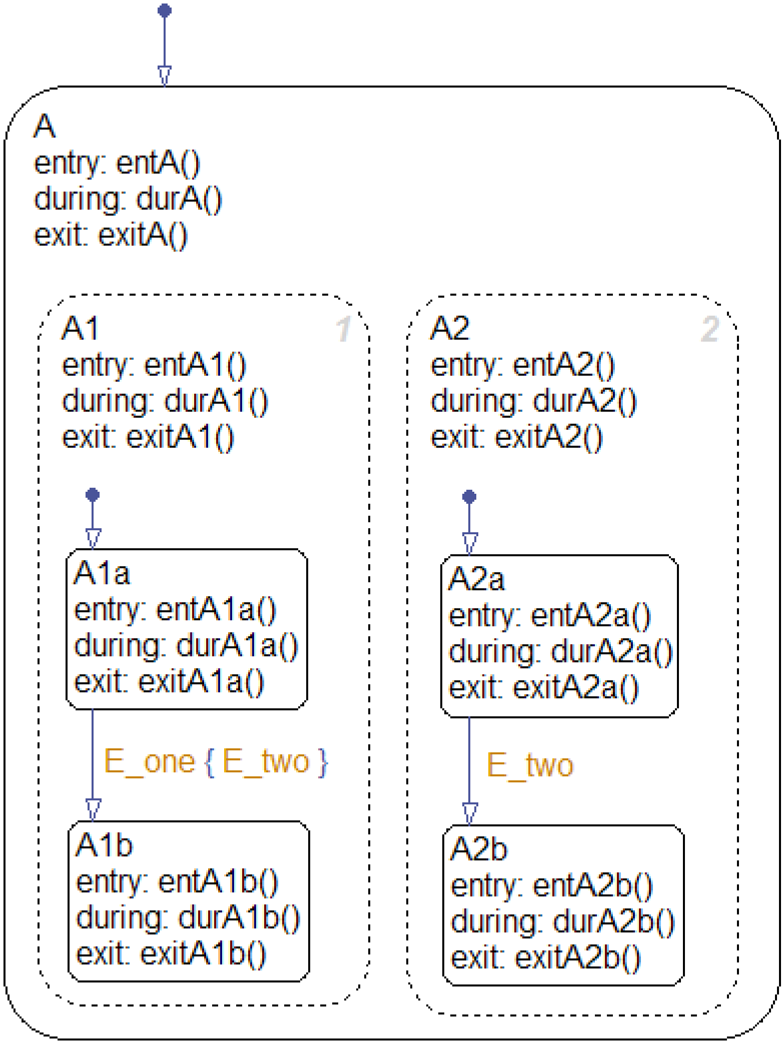
Condition actions and connective junctions are used to design a `for` loop construct. This example shows the use of a condition action and connective junction to create a `for` loop construct. The chart uses implicit ordering of outgoing transitions (see "Implicit Ordering of Outgoing Transitions" on page 3-59).



See "For-Loop Construct Example" on page B-39 to see the behavior of this example.

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

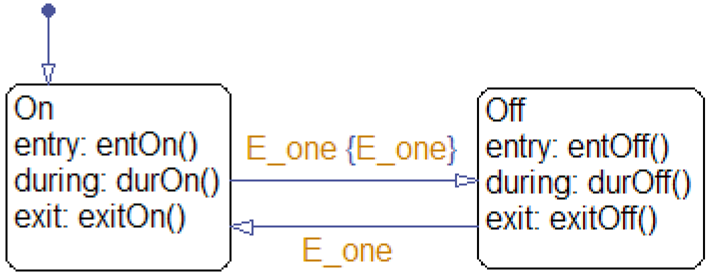
This example shows how to use condition actions to broadcast events immediately to parallel (AND) states. The chart uses implicit ordering of parallel states (see "Implicit Ordering of Parallel States" on page 3-77).



See “Event Broadcast Condition Action Example” on page B-56 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 chart is asleep. State `On` is active. Event `E_one` occurs and awakens the chart, which processes the event from the root down through the hierarchy:

1. The chart root checks to see if there is a valid transition as a result of `E_one`.

   A valid transition from state `On` to state `Off` is detected.

2. The condition action on the transition broadcasts event `E_one`.
3. Event `E_one` is detected on the valid transition, which is immediately executed. State `On` is still active.
4. The broadcast of event `E_one` awakens the chart a second time.
5. Go to step 1.

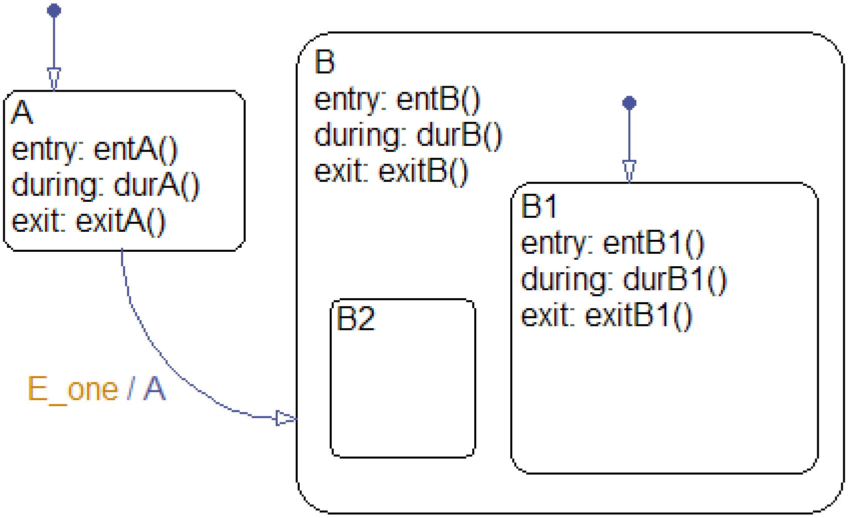
Steps 1 through 5 continue to execute in a cyclical manner. The transition label indicating a trigger on the same event as the condition action broadcast event results in unrecoverable cyclic behavior. This sequence never completes when event `E_one` is broadcast and state `On` is active.

Default Transition Examples

In this section...
"Default Transition in Exclusive (OR) Decomposition Example" on page B-18
"Default Transition to a Junction Example" on page B-19
"Default Transition and a History Junction Example" on page B-20
"Labeled Default Transitions Example" on page B-22

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.



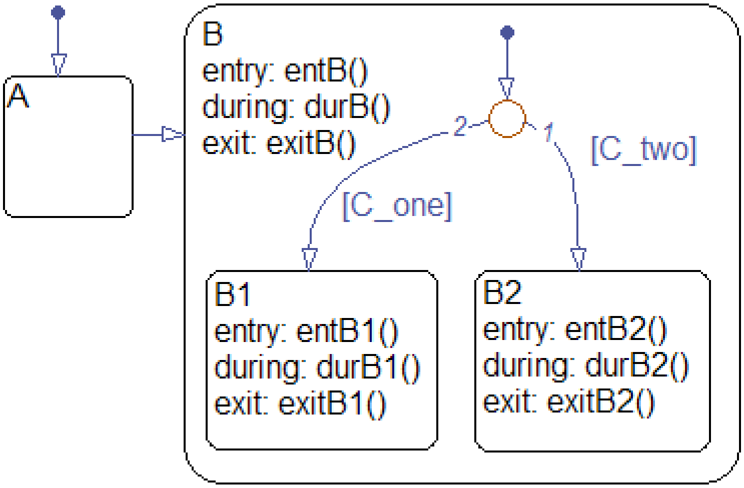
Initially, the chart is asleep. State `A` is active. Event `E_one` occurs and awakens the chart, which processes the event from the root down through the hierarchy:

1. The chart root checks to see if there is a valid transition as a result of `E_one`. There is a valid transition from state `A` to superstate `B`.
2. State `A` exit actions (`exitA()`) execute and complete.
3. State `A` is marked inactive.
4. The transition action, `A`, is executed and completed.
5. State `B` is marked active.
6. State `B` entry actions (`entB()`) execute and complete.
7. State `B` detects a valid default transition to state `B.B1`.
8. State `B.B1` is marked active.
9. State `B.B1` entry actions (`entB1()`) execute and complete.
10. The chart goes back to sleep.

This sequence completes the execution of this Stateflow chart associated with event `E_one` when state `A` is initially active.

Default Transition to a Junction Example

The following example shows the behavior of a default transition to a connective junction. The chart uses implicit ordering of outgoing transitions (see "Implicit Ordering of Outgoing Transitions" on page 3-59).



Initially, the chart is asleep. State `B.B1` is active. Condition `[C_two]` is true. An event occurs and awakens the chart, which processes the event from the root down through the hierarchy:

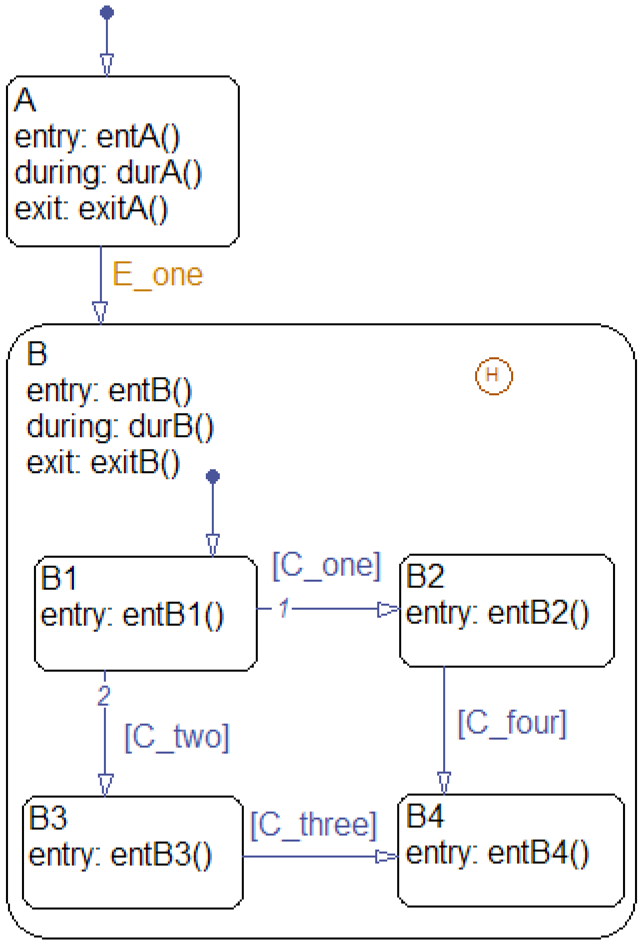
1. State `B` checks to see if there is a valid transition as a result of any event. There is none.

2. State `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. The chart uses implicit ordering of outgoing transitions (see “Implicit Ordering of Outgoing Transitions” on page 3-59).



Initially, the chart is asleep. State `A` is active. A history junction records the fact that state `B4` is the previously active substate of superstate `B`. Event `E_one` occurs and awakens the chart, which processes the event from the root down through the hierarchy:

1. The chart root checks to see if there is a valid transition as a result of `E_one`.

   There is a valid transition from state `A` to superstate `B`.

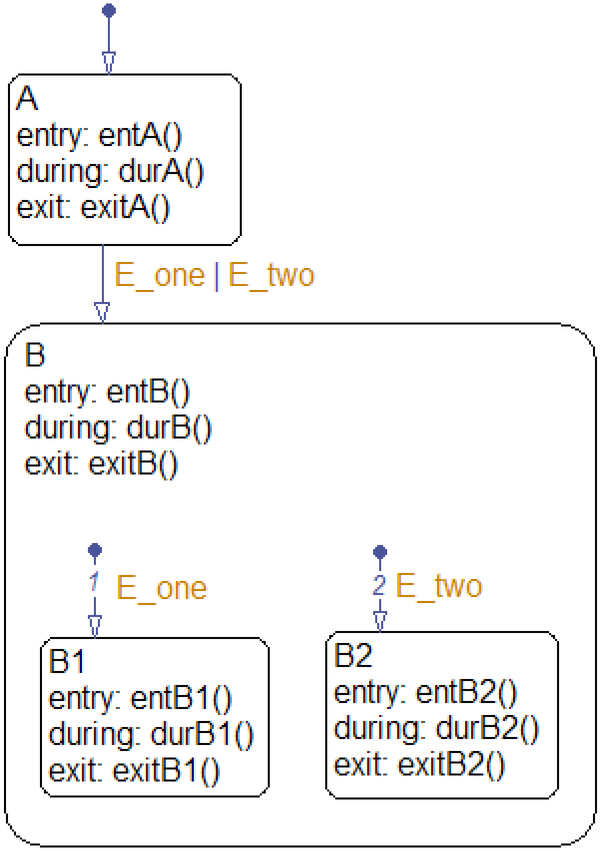
2. State `A` exit actions (`exitA()`) execute and complete.
3. State `A` is marked inactive.
4. State `B` is marked active.
5. State `B` entry actions (`entB()`) execute and complete.
6. State `B` uses the history junction to determine the substate destination of the transition into the superstate.

   The history junction indicates that substate `B.B4` was the last active substate, which becomes the destination of the transition.
7. State `B.B4` is marked active.
8. State `B.B4` entry actions (`entB4()`) execute and complete.
9. The chart goes back to sleep.

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

Labeled Default Transitions Example

This example shows the use of a default transition with a label. The chart uses implicit ordering of outgoing transitions (see “Implicit Ordering of Outgoing Transitions” on page 3-59).



Initially, the chart is asleep. State `A` is active. Event `E_one` occurs and awakens the chart, which processes the event from the root down through the hierarchy:

1. The chart root checks to see if there is a valid transition as a result of `E_one`.

   There is a valid transition from state `A` to superstate `B`. The transition is valid if event `E_one` or `E_two` occurs.

2. State `A` exit actions execute and complete (`exitA()`).

3. State `A` is marked inactive.

4. State `B` is marked active.
5. State `B` entry actions execute and complete (`entB()`).
6. State `B` detects a valid default transition to state `B.B1`. The default transition is valid as a result of `E_one`.
7. State `B.B1` is marked active.
8. State `B.B1` entry actions execute and complete (`entB1()`).
9. The chart goes back to sleep.

This sequence completes the execution of this Stateflow chart associated with event `E_one` when state `A` is initially active.

Inner Transition Examples

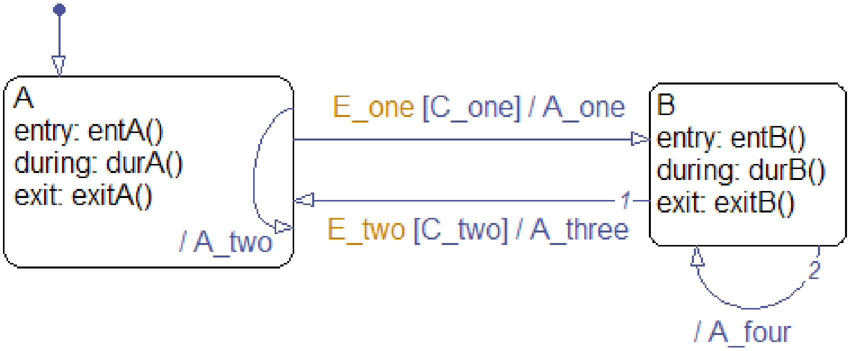
In this section...

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. The chart uses implicit ordering of outgoing transitions (see "Implicit Ordering of Outgoing Transitions" on page 3-59).



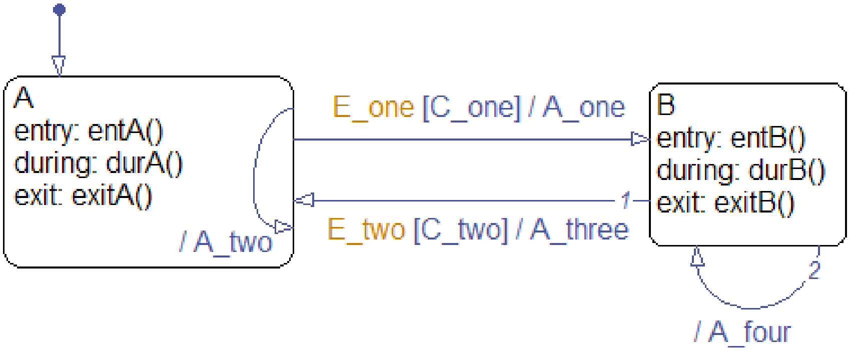
Initially, the chart is asleep. State `A` is active. Condition `[C_one]` is false. Event `E_one` occurs and awakens the chart, which processes the event from the root down through the hierarchy:

1. The chart root checks to see if there is a valid transition as a result of `E_one`. A potentially valid transition from state `A` to state `B` is detected. However, the transition is not valid, because `[C_one]` is false.
2. State `A` during actions (`durA()`) execute and complete.
3. State `A` checks its children for a valid transition and detects a valid inner transition.
4. State `A` remains active. The inner transition action `A_two` is executed and completed. Because it is an inner transition, state `A`'s exit and entry actions are not executed.
5. The chart goes back to sleep.

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

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. The chart uses implicit ordering of outgoing transitions (see "Implicit Ordering of Outgoing Transitions" on page 3-59).



Initially, the chart is asleep. State `A` is still active. Condition `[C_one]` is true. Event `E_one` occurs and awakens the chart, which processes the event from the root down through the hierarchy:

1. The chart root checks to see if there is a valid transition as a result of `E_one`.

   The transition from state `A` to state `B` is now valid because `[C_one]` is true.

2. State `A` exit actions (`exitA()`) execute and complete.

3. State `A` is marked inactive.

4. The transition action `A_one` is executed and completed.

5. State `B` is marked active.

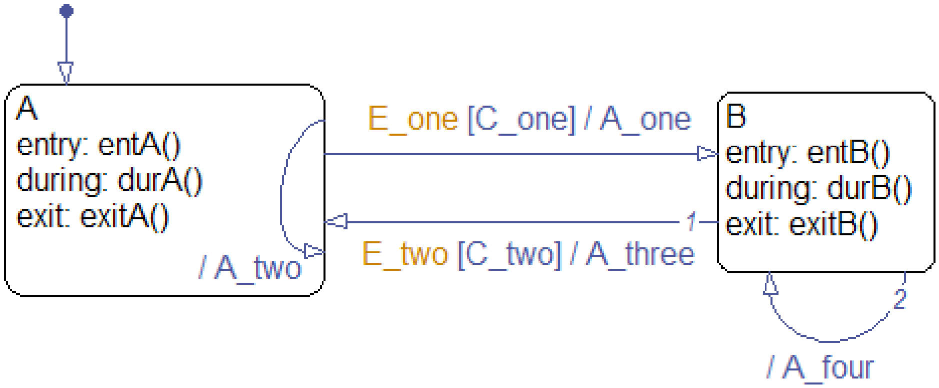
6. State `B` entry actions (`entB()`) execute and complete.

7. The chart goes back to sleep.

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

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. The chart uses implicit ordering of outgoing transitions (see "Implicit Ordering of Outgoing Transitions" on page 3-59).



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

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

   A potentially valid transition from state `B` to state `A` is detected. The transition is not valid because `[C_two]` is false. However, active state `B` has a valid self-loop transition.

2. State `B` exit actions (`exitB()`) execute and complete.

3. State `B` is marked inactive.

4. The self-loop transition action, `A_four`, executes and completes.

5. State `B` is marked active.

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

7. The chart goes back to sleep.

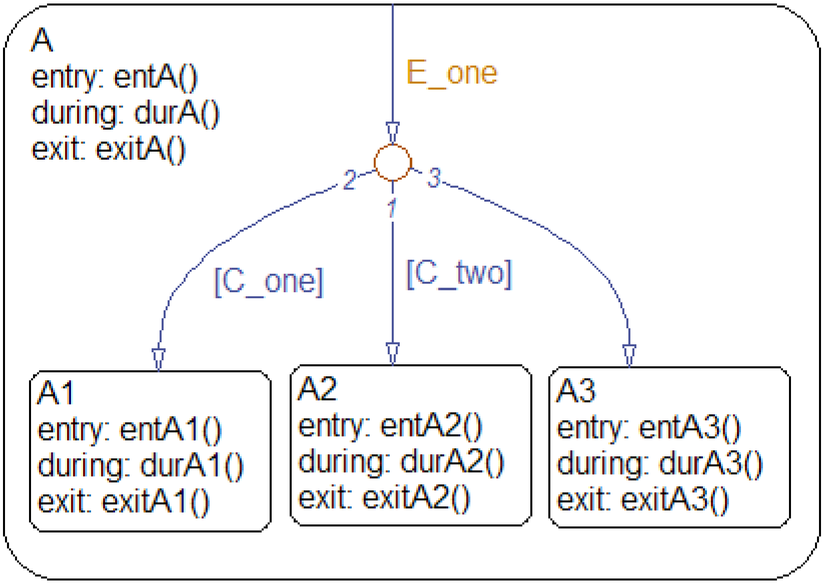
This sequence completes the execution of this Stateflow chart associated with event `E_two`. This example shows the difference in behavior between inner and self-loop transitions.

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 3-59).



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

1. The chart root checks to see if there is a valid transition at the root level as a result of `E_one`. There is no valid transition.

2. State `A` during actions (`durA()`) execute and complete.

3. State `A` checks itself for valid transitions and detects that there is a valid inner transition to a connective junction.

   The conditions are evaluated to determine whether one of the transitions is valid. Because implicit ordering applies, the segments labeled with a condition are evaluated before the unlabeled segment. The evaluation starts from a 12 o'clock position on the junction and progresses in a clockwise manner. Because `[C_two]` is true, the inner transition to the junction and then to state `A.A2` is valid.

4. State `A.A1` exit actions (`exitA1()`) execute and complete.

5 State `A.A1` is marked inactive.

6 State `A.A2` is marked active.

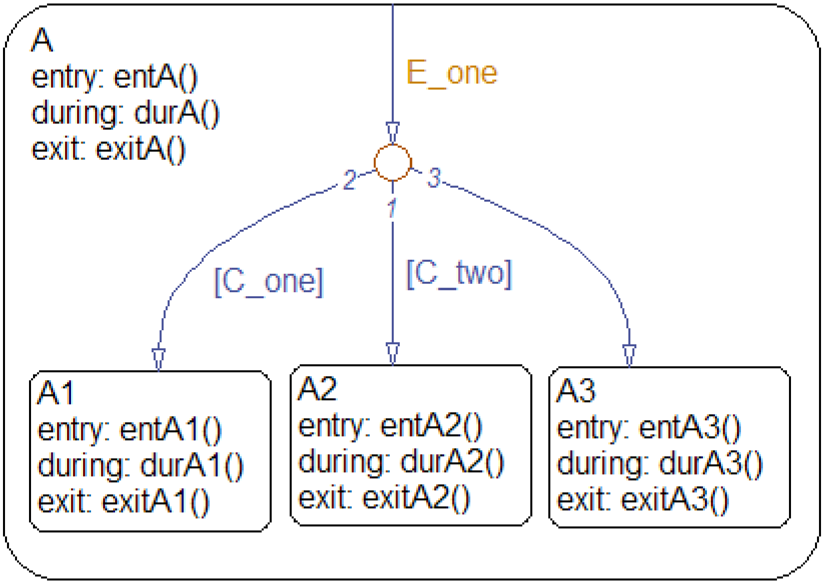
7 State `A.A2` entry actions (`entA2()`) execute and complete.

8 The chart goes back to sleep.

This sequence completes the execution of this Stateflow chart associated with event `E_one` when state `A1` is active and condition `[C_two]` is true.

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 3-59).



Initially, the chart is asleep. State `A2` is active. Condition `[C_two]` is true. Event `E_one` occurs and awakens the chart, which processes the event from the root down through the hierarchy:

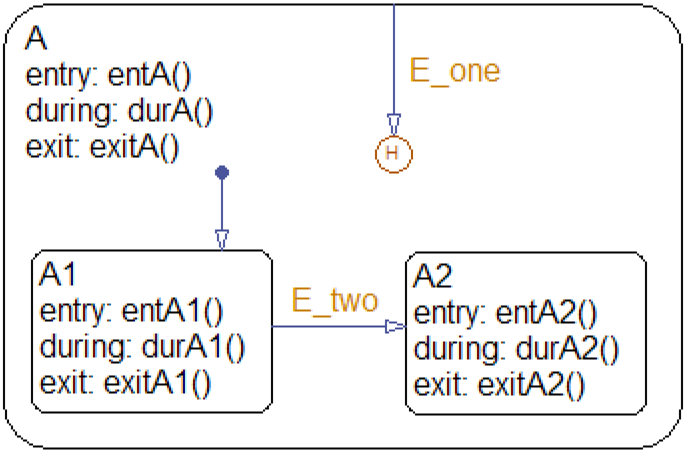
1. The chart root checks to see if there is a valid transition at the root level as a result of `E_one`. There is no valid transition.
2. State `A` during actions (`durA()`) execute and complete.
3. State `A` checks itself for valid transitions and detects a valid inner transition to a connective junction.

   The conditions are evaluated to determine whether one of the transitions is valid. Because implicit ordering applies, the segments labeled with a condition are evaluated before the unlabeled segment. The evaluation starts from a 12 o'clock position on the junction and progresses in a clockwise manner. Because `[C_two]` is true, the inner transition to the junction and then to state `A.A2` is valid.
4. State A.A2 exit actions (`exitA2()`) execute and complete.
5. State A.A2 is marked inactive.
6. State A.A2 is marked active.
7. State A.A2 entry actions (`entA2()`) execute and complete.
8. The chart goes back to sleep.

This sequence completes the execution of this Stateflow chart associated with event `E_one` when state `A2` is active and condition `[C_two]` is true. For a state with a valid inner transition, an active substate can be exited and reentered immediately.

Inner Transition to a History Junction Example

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



Initially, the chart is asleep. State `A.A1` is active. History information exists because superstate `A` is active. Event `E_one` occurs and awakens the chart, which processes the event from the root down through the hierarchy:

1. The chart root checks to see if there is a valid transition as a result of `E_one`. There is no valid transition.
2. State `A` during actions execute and complete.
3. State `A` checks itself for valid transitions and detects that there is a valid inner transition to a history junction. Based on the history information, the last active state, `A.A1`, is the destination state.
4. State `A.A1` exit actions execute and complete.
5. State `A.A1` is marked inactive.
6. State `A.A1` is marked active.
7. State `A.A1` entry actions execute and complete.
8. The chart goes back to sleep.

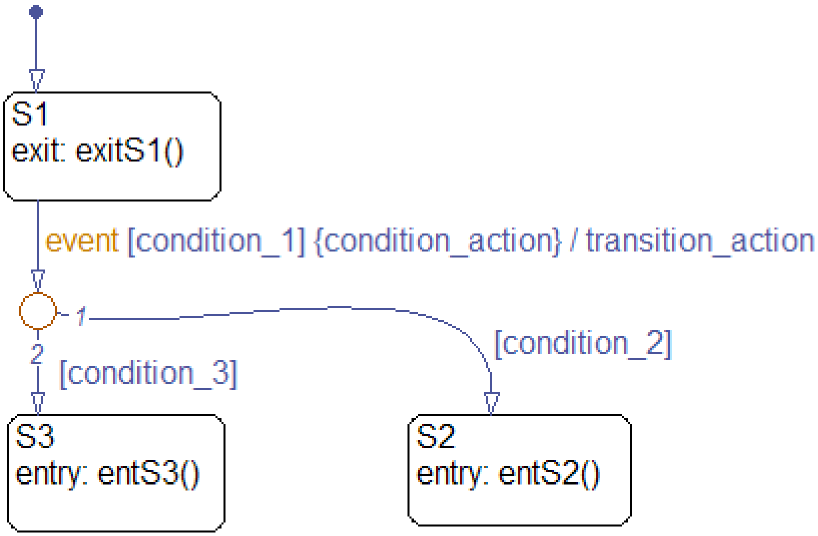
This sequence completes the execution of this Stateflow chart associated with event `E_one` when there is an inner transition to a history junction and state `A.A1` is active. For a state with a valid inner transition, an active substate can be exited and reentered immediately.

Connective Junction Examples

In this section...

Label Format for Transition Segments Example

The label format for a transition segment entering a junction is the same as for transitions entering states, as shown in the following example. The chart uses implicit ordering of outgoing transitions (see "Implicit Ordering of Outgoing Transitions" on page 3-59).



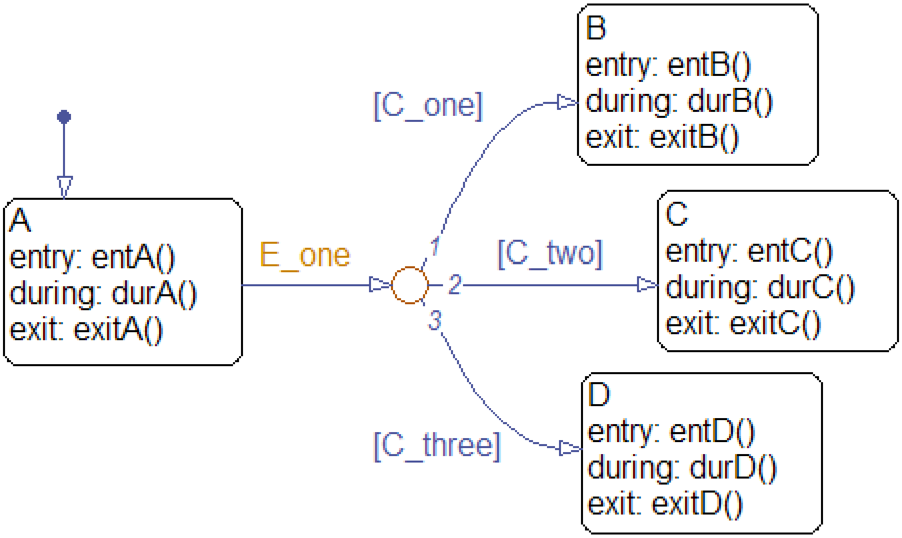
Execution of a transition in this example occurs as follows:

1. When an event occurs, state `S1` is checked for an outgoing transition with a matching event specified.
2. If a transition with a matching event is found, the transition condition for that transition (in brackets) is evaluated.
3. If `condition_1` evaluates to true, the condition action `condition_action` (in braces) is executed.
4. The outgoing transitions from the junction are checked for a valid transition. Since `condition_2` is true, a valid state-to-state transition from `S1` to `S2` exists.
5. State `S1` exit actions execute and complete.
6. State `S1` is marked inactive.
7. The transition action `transition_action` executes and completes.

8. The completed state-to-state transition from `S1` to `S2` occurs.
9. State `S2` is marked active.
10. State `S2` entry actions execute and complete.

If-Then-Else Decision Construct 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 3-59).



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

1. The chart root checks to see if there is a valid transition as a result of `E_one`.

   A valid transition segment from state `A` to the connective junction exists. Because implicit ordering applies, the transition segments beginning from a 12 o'clock position on the connective junction are evaluated for validity.

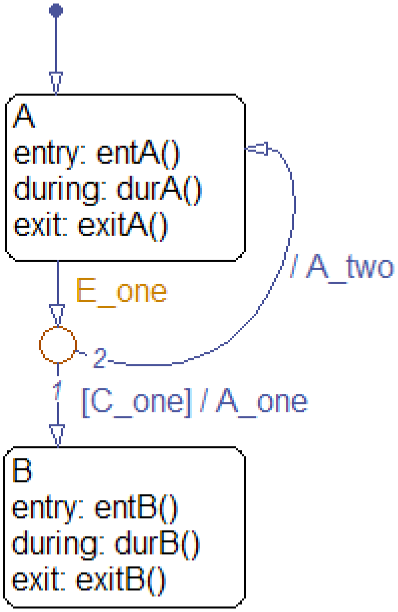
The first transition segment, labeled with condition [`C_one`], is not valid. The next transition segment, labeled with the condition [`C_two`], is valid. The complete transition from state `A` to state `C` is valid.

2. State `A` exit actions (`exitA()`) execute and complete.
3. State `A` is marked inactive.
4. State `C` is marked active.
5. State `C` entry actions (`entC()`) execute and complete.
6. The chart goes back to sleep.

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

Self-Loop Transition Example

This example shows the behavior of a self-loop transition using a connective junction. The chart uses implicit ordering of outgoing transitions (see "Implicit Ordering of Outgoing Transitions" on page 3-59).



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

1. The chart root checks to see if there is a valid transition as a result of `E_one`. A valid transition segment from state `A` to the connective junction exists. Because implicit ordering applies, the transition segment labeled with a condition is evaluated for validity. Because the condition `[C_one]` is not valid, the complete transition from state `A` to state `B` is not valid. The transition segment from the connective junction back to state `A` is valid.
2. State `A` exit actions (`exitA()`) execute and complete.
3. State `A` is marked inactive.
4. The transition action `A_two` is executed and completed.
5. State `A` is marked active.

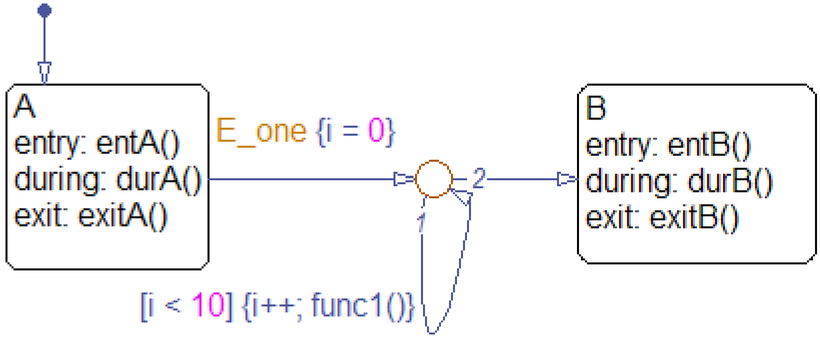
6 State `A` entry actions (`entA()`) execute and complete.

7 The chart goes back to sleep.

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

For-Loop Construct 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 3-59).



Initially, the chart is asleep. State `A` is active. Event `E_one` occurs and awakens the chart, which processes the event from the root down through the hierarchy:

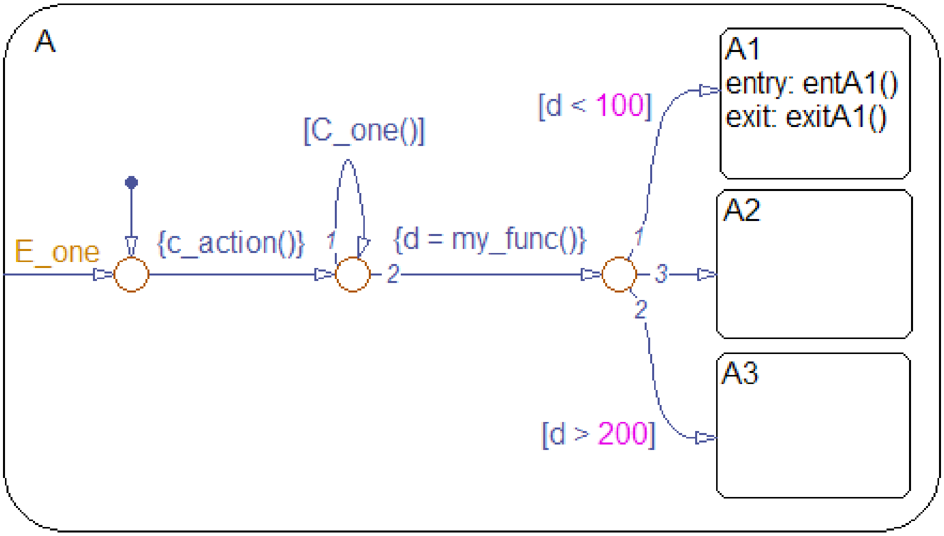
1 The chart root checks to see if there is a valid transition as a result of `E_one`. There is a valid transition segment from state `A` to the connective junction. The transition segment condition action, `i = 0`, executes and completes. Of the two transition segments leaving the connective junction, the transition segment that is a self-loop back to the connective junction evaluates next for validity. That segment takes priority in evaluation because it has a condition, whereas the other segment is unlabeled. This evaluation behavior reflects implicit ordering of outgoing transitions in the chart.

2. The condition `[i < 10]` evaluates as true. The condition actions `i++` and a call to `func1` execute and complete until the condition becomes false. Because a connective junction is not a final destination, the transition destination is still unknown.
3. The unconditional segment to state `B` is now valid. The complete transition from state `A` to state `B` is valid.
4. State `A` exit actions (`exitA()`) execute and complete.
5. State `A` is marked inactive.
6. State `B` is marked active.
7. State `B` entry actions (`entB()`) execute and complete.
8. The chart goes back to sleep.

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

Flow 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 3-59).



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

1. The chart root checks to see if there is a valid transition as a result of `E_one`. There is no valid transition.

2. State `A` checks itself for valid transitions and detects a valid inner transition to a connective junction.

3. The next possible segments of the transition are evaluated. Only one outgoing transition exists, and it has a condition action defined. The condition action executes and completes.

4. The next possible segments are evaluated. Two outgoing transitions exist: a conditional self-loop transition and an unconditional transition segment. Because implicit ordering applies, the conditional transition segment takes precedence. Since the condition `[C_one()]` is true, the self-loop transition is taken. Since a final transition destination has not been reached, this self-loop continues until `[C_one()]` is false.

   Assume that after five iterations, `[C_one()]` is false.

5. The next possible transition segment (to the next connective junction) is evaluated. It is an unconditional transition segment with a condition action. The transition segment is taken and the condition action, `{d=my_func()}`, executes and completes. The returned value of `d` is 84.

6. The next possible transition segment is evaluated. Three outgoing transition segments exist: two conditional and one unconditional. Because implicit ordering applies, the segment labeled with the condition `[d < 100]` evaluates first based on the geometry of the two outgoing conditional transition segments. Because the returned value of `d` is 84, the condition `[d < 100]` is true and this transition to the destination state `A.A1` is valid.

7. State `A.A1` exit actions (`exitA1()`) execute and complete.

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

9. State `A.A1` is marked active.

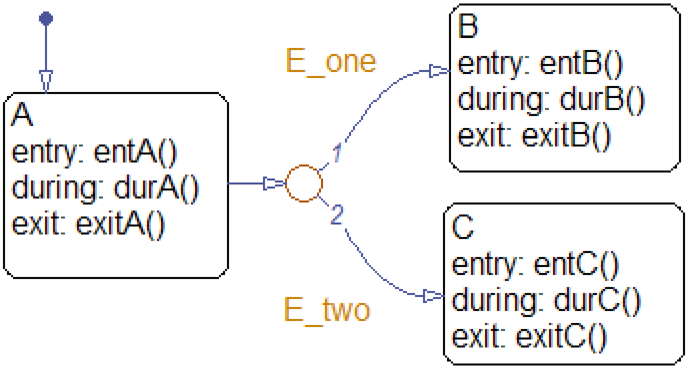
10. State `A.A1` entry actions (`entA1()`) execute and complete.

11. The chart goes back to sleep.

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

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 3-59).



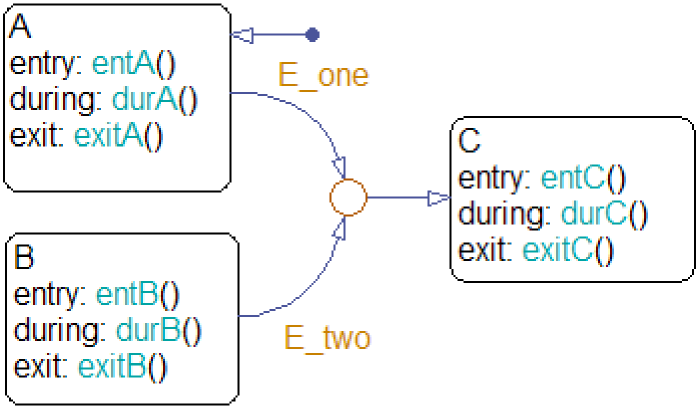
Initially, the chart is asleep. State `A` is active. Event `E_two` occurs and awakens the chart, which processes the event from the root down through the hierarchy:

1. The chart root checks to see if there is a valid transition as a result of `E_two`. A valid transition segment exists from state `A` to the connective junction. Because implicit ordering applies, evaluation of segments with equivalent label priority begins from a 12 o'clock position on the connective junction and progresses clockwise. The first transition segment, labeled with event `E_one`, is not valid. The next transition segment, labeled with event `E_two`, is valid. The complete transition from state `A` to state `C` is valid.
2. State `A` exit actions (`exitA()`) execute and complete.
3. State `A` is marked inactive.
4. State `C` is marked active.
5. State `C` entry actions (`entC()`) execute and complete.
6. The chart goes back to sleep.

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

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.



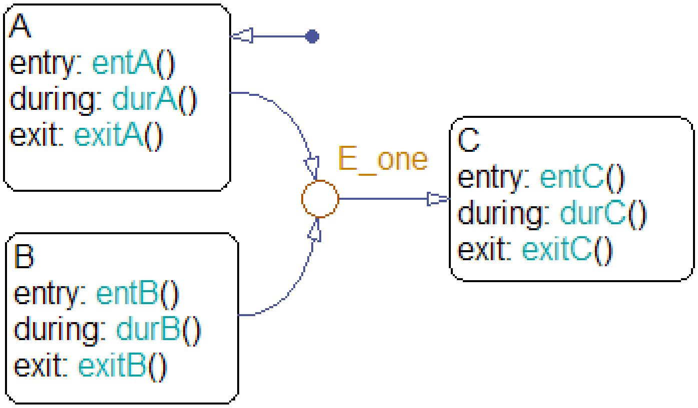
Initially, the chart is asleep. State `A` is active. Event `E_one` occurs and awakens the chart, which processes the event from the root down through the hierarchy:

1. The chart root checks to see if there is a valid transition as a result of `E_one`. A valid transition segment exists from state `A` to the connective junction and from the junction to state `C`.
2. State `A` exit actions (`exitA()`) execute and complete.
3. State `A` is marked inactive.
4. State `C` is marked active.
5. State `C` entry actions (`entC()`) execute and complete.
6. The chart goes back to sleep.

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

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.



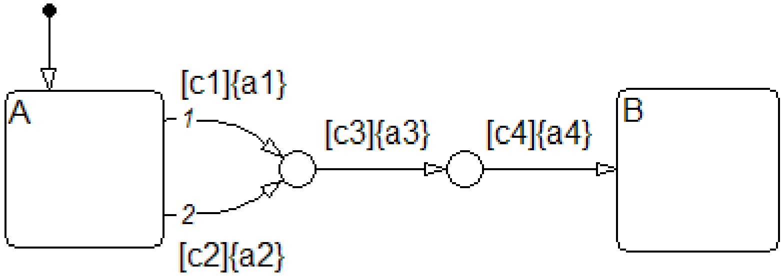
Initially, the chart is asleep. State `B` is active. Event `E_one` occurs and awakens the chart, which processes the event from the root down through the hierarchy:

1. The chart root checks to see if there is a valid transition as a result of `E_one`. A valid transition segment exists from state `B` to the connective junction and from the junction to state C.
2. State `B` exit actions (`exitB()`) execute and complete.
3. State `B` is marked inactive.
4. State `C` is marked active.
5. State `C` entry actions (`entC()`) execute and complete.
6. The chart goes back to sleep.

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

Backtracking Behavior in Flow Graphs Example

This example shows the behavior of transitions with junctions that force backtracking behavior in flow graphs. The chart uses implicit ordering of outgoing transitions (see "Implicit Ordering of Outgoing Transitions" on page 3-59).



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

1. The chart root checks to see if there is a valid transition from state A.

   There is a valid transition segment marked with the condition c1 from state A to a connective junction.

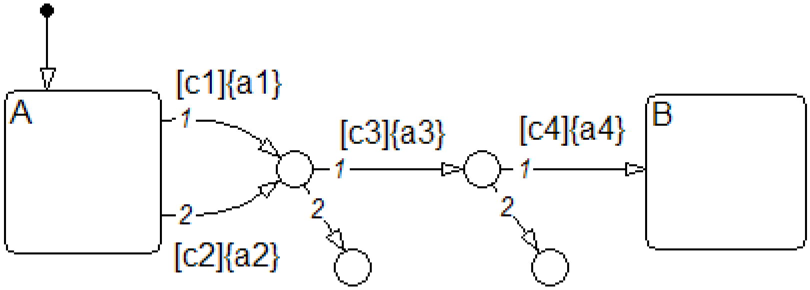
2. Condition c1 is true and action a1 executes.
3. Condition c3 is true and action a3 executes.
4. Condition c4 is not true and control flow backtracks to state A.
5. The chart root checks to see if there is another valid transition from state A.

   There is a valid transition segment marked with the condition c2 from state A to a connective junction.

6. Condition c2 is true and action a2 executes.
7. Condition c3 is true and action a3 executes.
8. Condition c4 is not true and control flow backtracks to state A.

9 The chart goes to sleep.

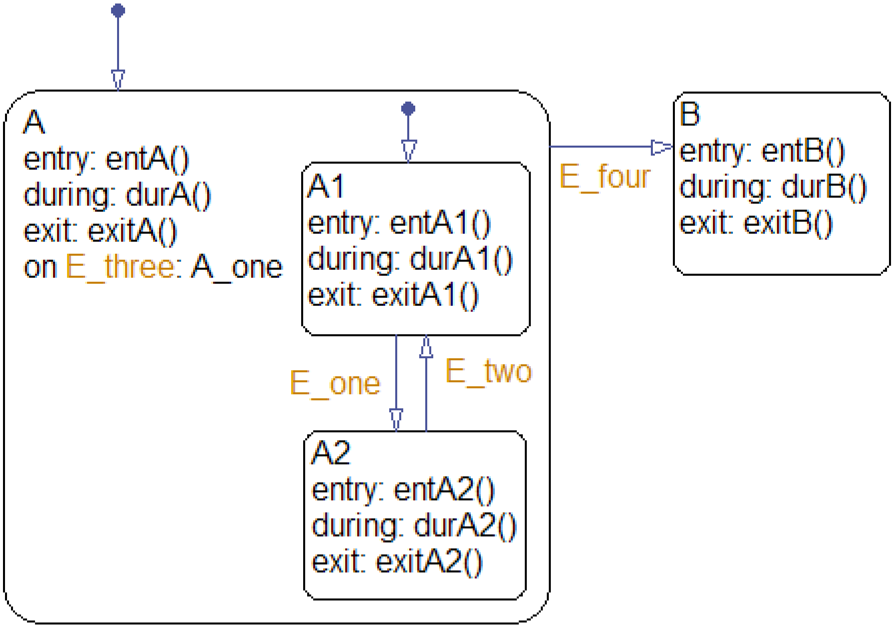
The preceding example shows the unexpected behavior of executing both actions `a1` and `a2`. Another unexpected behavior is the execution of action `a3` twice. To resolve this problem, consider adding unconditional transitions to terminating junctions.



The terminating junctions allow flow to end if either `c3` or `c4` is not true. This design leaves state A active without executing unnecessary actions.

Event Actions in a Superstate Example

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



Initially, the chart is asleep. State `A.A1` is active. Event `E_three` occurs and awakens the chart, which processes the event from the root down through the hierarchy:

1. The chart root checks to see if there is a valid transition as a result of `E_three`. No valid transition exists.
2. State `A` during actions (`durA()`) execute and complete.
3. State `A` executes and completes the `on` event `E_three` action (`A_one`).
4. State `A` checks its children for valid transitions. No valid transitions exist.
5. State `A1` during actions (`durA1()`) execute and complete.

6. The chart goes back to sleep.

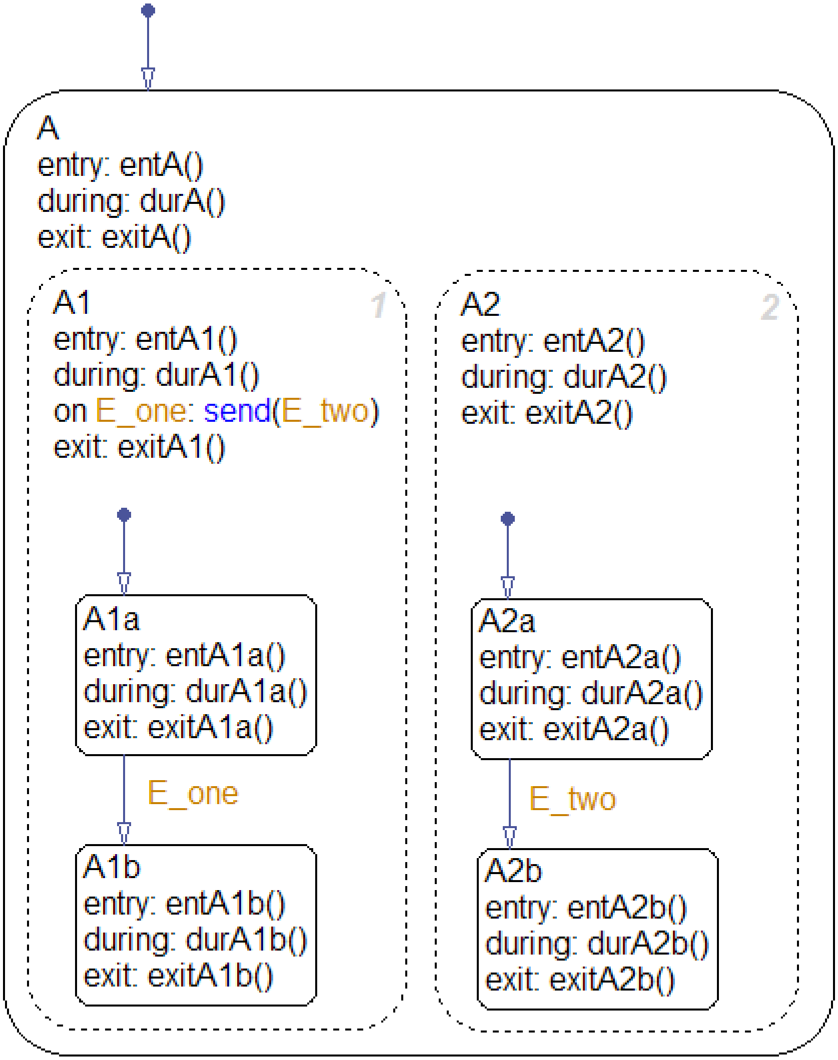
This sequence completes the execution of this Stateflow chart associated with event `E_three`.

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 3-77).



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

1. The chart root checks to see if there is a valid transition at the root level as a result of `E_one`. No valid transition exists.

2. State `A` `during` actions (`durA()`) execute and complete.

3. The children of state `A` are parallel (AND) states. Because implicit ordering applies, the states are evaluated and executed from left to right and top to bottom. State `A.A1` is evaluated first. State `A.A1` `during` actions (`durA1()`) execute and complete. State `A.A1` executes and completes the `on E_one` action and broadcasts event `E_two`. The `during` and `on` *`event_name`* actions are processed based on their order of appearance in the state label:

   a. The broadcast of event `E_two` awakens the chart a second time. The chart root checks to see if there is a valid transition as a result of `E_two`. No valid transition exists.

   b. State A `during` actions (`durA()`) execute and complete.

   c. State A checks its children for valid transitions. No valid transitions exist.

   d. State A's children are evaluated starting with state `A.A1`. State `A.A1` `during` actions (`durA1()`) execute and complete. State `A.A1` is evaluated for valid transitions. There are no valid transitions as a result of `E_two` within state `A1`.

   e. State `A1a`'s `during` actions (`durA1a()`) execute.

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

   g. State `A.A2.A2a` `exit` actions (`exitA2a()`) execute and complete.

   h. State `A.A2.A2a` is marked inactive.

   i. State `A.A2.A2b` is marked active.

   j. State `A.A2.A2b` `entry` actions (`entA2b()`) execute and complete.

4. 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`. A valid transition exists from state `A.A1.A1a` to state `A.A1.A1b`.

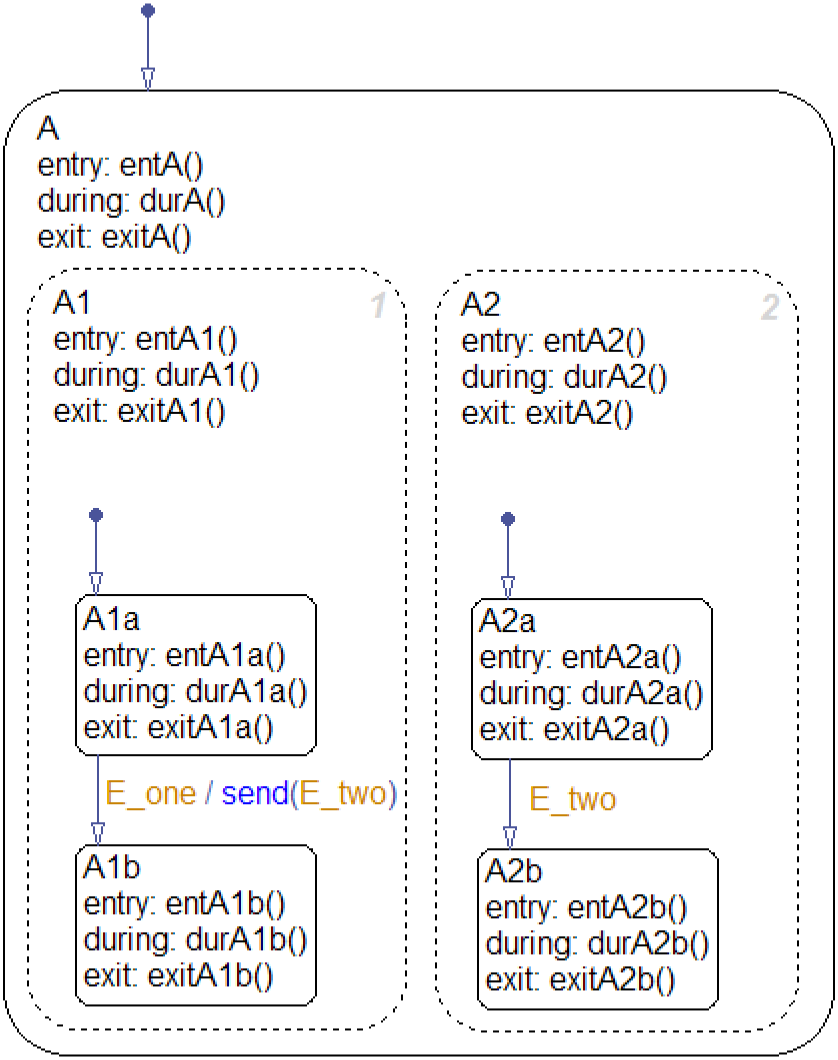
6. State `A.A1.A1a` is marked inactive.
7. State `A.A1.A1b` is marked active.
8. State `A.A1.A1b` `entry` actions (`entA1b()`) execute and complete.
9. Parallel state `A.A2` is evaluated next. State `A.A2` `during` actions (`durA2()`) execute and complete. There are no valid transitions as a result of `E_one`.
10. State `A.A2.A2b` `during` actions (`durA2b()`) execute and complete.

    State `A.A2.A2b` is now active as a result of the processing of the `on` event broadcast of `E_two`.
11. The chart goes back to sleep.

This sequence completes the execution of this Stateflow chart associated with event `E_one` and the `on` event broadcast to a parallel state of event `E_two`. The final chart activity is that parallel substates `A.A1.A1b` and `A.A2.A2b` are active.

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 3-77).



Start of Event E_one Processing

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

1 The chart root checks to see if there is a valid transition as a result of `E_one`. There is no valid transition.

2 State `A` `during` actions (`durA()`) execute and complete.

3 State A's children are parallel (AND) states. Because implicit ordering applies, the states are evaluated and executed from left to right and top to bottom. State `A.A1` is evaluated first. State `A.A1during` actions (`durA1()`) execute and complete.

4 State `A.A1` checks for any valid transitions as a result of event `E_one`. There is a valid transition from state `A.A1.A1a` to state `A.A1.A1b`.

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

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

Event E_two Preempts E_one

7 The transition action that broadcasts event `E_two` executes and completes:

   a The broadcast of event `E_two` now preempts the transition from state `A1a` to state `A1b` that event `E_one` triggers.

   b The broadcast of event `E_two` awakens the chart a second time. The chart root checks to see if there is a valid transition as a result of `E_two`. No valid transition exists.

   c State A `during` actions (`durA()`) execute and complete.

   d State A's children are evaluated starting with state `A.A1`. State `A.A1during` actions (`durA1()`) execute and complete. State `A.A1` is evaluated for valid transitions. There are no valid transitions as a result of `E_two` within state `A1`.

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

   f State `A.A2.A2a` `exit` actions (`exitA2a()`) execute and complete.

   g State `A.A2.A2a` is marked inactive.

   h State `A.A2.A2b` is marked active.

i State `A.A2.A2b` `entry` actions (`entA2b()`) execute and complete.

Event E_one Processing Resumes

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.

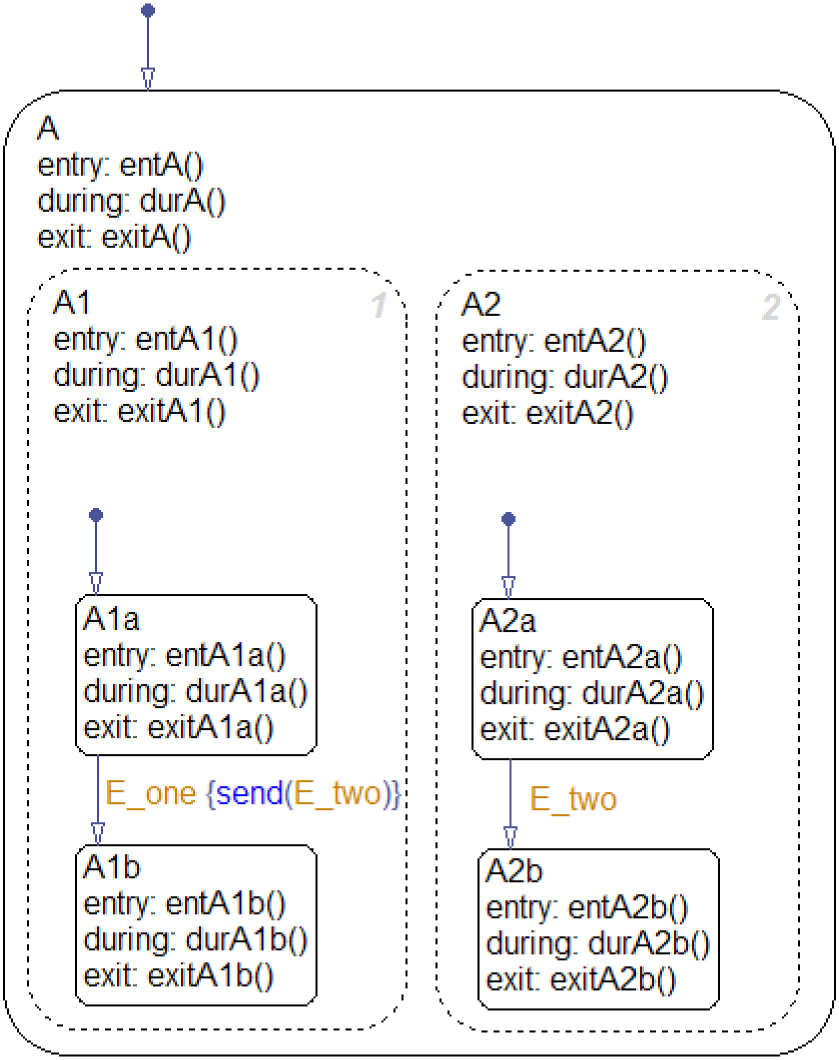
State `A.A2.A2b` is now active as a result of the processing of event broadcast `E_two`.

12 The chart goes back to sleep.

This sequence completes the execution of this Stateflow chart associated with event `E_one` and the event broadcast on a transition action to a parallel state of event `E_two`. The final chart activity is that parallel substates `A.A1.A1b` and `A.A2.A2b` are active.

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 3-77).



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

1. The chart root checks to see if there is a valid transition as a result of `E_one`. No valid transition exists.

2. State `A` `during` actions (`durA()`) execute and complete.

3. State A's children are parallel (AND) states. Because implicit ordering applies, the states are evaluated and executed from top to bottom, and from left to right. State `A.A1` is evaluated first. State `A.A1` `during` actions (`durA1()`) execute and complete.

4. State `A.A1` checks for any valid transitions as a result of event `E_one`. A valid transition from state `A.A1.A1a` to state `A.A1.A1b` exists. A valid condition action also exists. The condition action event broadcast of `E_two` executes and completes. State `A.A1.A1a` is still active:

   a. The broadcast of event `E_two` awakens the Stateflow chart a second time. The chart root checks to see if there is a valid transition as a result of `E_two`. There is no valid transition.

   b. State `A` `during` actions (`durA()`) execute and complete.

   c. State `A`'s children are evaluated starting with state `A.A1`. State `A.A1` `during` actions (`durA1()`) execute and complete. State `A.A1` is evaluated for valid transitions. There are no valid transitions as a result of `E_two` within state `A1`.

   d. State `A1a` `during` actions (`durA1a()`) execute.

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

   f. State `A.A2.A2a` `exit` actions (`exitA2a()`) execute and complete.

   g. State `A.A2.A2a` is marked inactive.

   h. State `A.A2.A2b` is marked active.

   i. State `A.A2.A2b` `entry` actions (`entA2b()`) execute and complete.

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

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

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

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

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

10. State `A.A2.A2b` `during` actions (`durA2b()`) execute and complete.

    State `A.A2.A2b` is now active as a result of the processing of the condition action event broadcast of `E_two`.

11. The chart goes back to sleep.

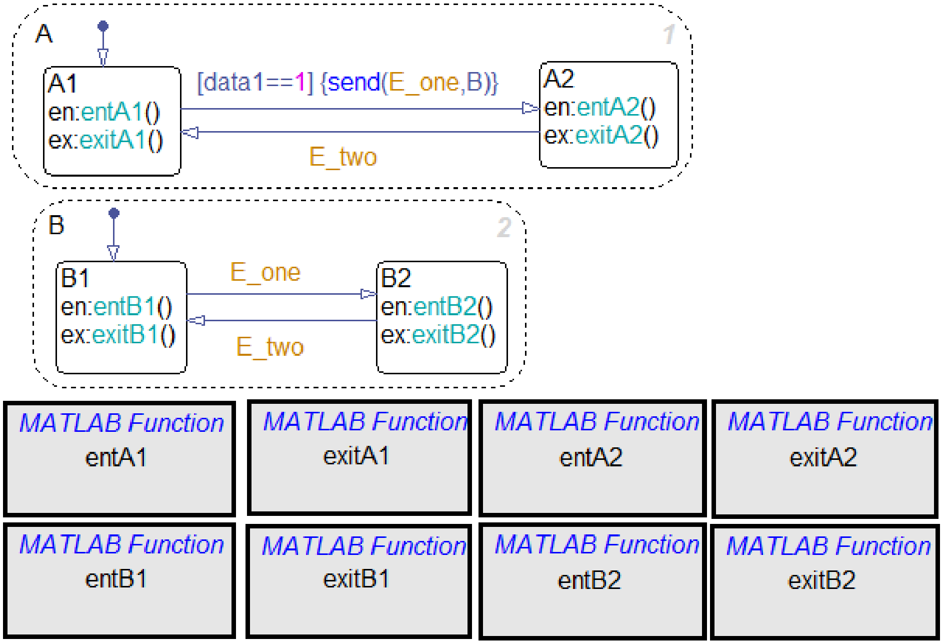
This sequence completes the execution of this Stateflow chart associated with event `E_one` and the event broadcast on a condition action to a parallel state of event `E_two`. The final chart activity is that parallel substates `A.A1.A1b` and `A.A2.A2b` are active.

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`*`)` syntax on a transition. The chart uses implicit ordering of parallel states (see "Implicit Ordering of Parallel States" on page 3-77).



Initially, the chart is asleep. Parallel substates `A.A1` and `B.B1` are active, which implies that parallel (AND) superstates `A` and `B` are also active. The condition `[data1==1]` is true. The event `E_one` belongs to the chart and is visible to both `A` and `B`.

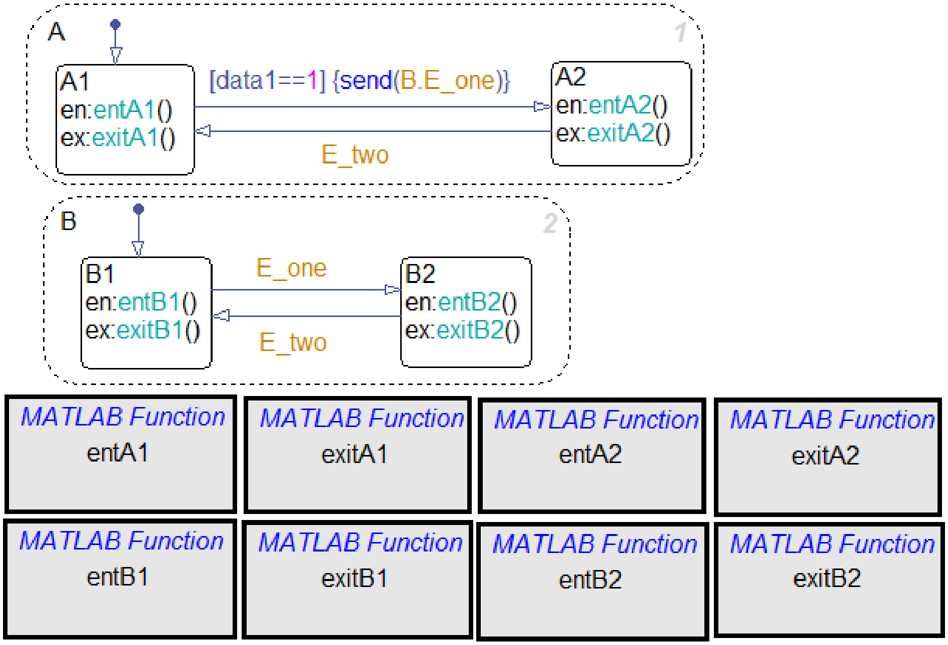
After waking up, the chart checks for valid transitions at every level of the hierarchy:

1. The chart root checks to see if there is a valid transition as a result of the event. There is no valid transition.
2. State `A` checks for any valid transitions as a result of the event. Because the condition `[data1==1]` is true, there is a valid transition from state `A.A1` to state `A.A2`.
3. The action `send(E_one,B)` executes:
   - **a** The broadcast of event `E_one` reaches state `B`. Because state `B` is active, that state receives the event broadcast and checks to see if there is a valid transition. There is a valid transition from `B.B1` to `B.B2`.
   - **b** State `B.B1` `exit` actions (`exitB1()`) execute and complete.
   - **c** State `B.B1` becomes inactive.
   - **d** State `B.B2` becomes active.
   - **e** State `B.B2` `entry` actions (`entB2()`) execute and complete.
4. State `A.A1` `exit` actions (`exitA1()`) execute and complete.
5. State `A.A1` becomes inactive.
6. State `A.A2` becomes active.
7. State `A.A2` `entry` actions (`entA2()`) execute and complete.

This sequence completes execution of a chart with a directed event broadcast to a parallel state.

Directed Event Broadcast Using Qualified Event Name Example

This example shows the behavior of directed event broadcast using a qualified event name on a transition. The chart uses implicit ordering of parallel states (see “Implicit Ordering of Parallel States” on page 3-77).



The only differences from the chart in “Directed Event Broadcast Using Send Example” on page B-60 are:

- The event `E_one` belongs to state `B` and is visible only to that state.
- The action `send(E_one,B)` is now `send(B.E_one)`.

  Using a qualified event name is necessary because `E_one` is not visible to state `A`.

After waking up, the chart checks for valid transitions at every level of the hierarchy:

1. The chart root checks to see if there is a valid transition as a result of the event. There is no valid transition.

2. State `A` checks for any valid transitions as a result of the event. Because the condition `[data1==1]` is true, there is a valid transition from state `A.A1` to state `A.A2`.

3. The action `send(B.E_one)` executes and completes:

   a. The broadcast of event `E_one` reaches state `B`. Because state `B` is active, that state receives the event broadcast and checks to see if there is a valid transition. There is a valid transition from `B.B1` to `B.B2`.

   b. State `B.B1` `exit` actions (`exitB1()`) execute and complete.

   c. State `B.B1` becomes inactive.

   d. State `B.B2` becomes active.

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

4. State `A.A1` `exit` actions (`exitA1()`) execute and complete.

5. State `A.A1` becomes inactive.

6. State `A.A2` becomes active.

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

This sequence completes execution of a chart with a directed event broadcast using a qualified event name to a parallel state.

Glossary

actions

Actions take place as part of Stateflow chart execution. The action can execute as part of a transition from one state to another, or depending on the activity status of a state. Transitions can contain condition actions and transition actions.

Action language defines the categories of actions you can specify and their associated notations. For example, states can have `entry`, `during`, `exit`, and `on` *`event_name`* actions.

An action can be a function call, an event broadcast, 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.

atomic subchart

Graphical object that enables you to reuse states and subcharts multiple times in a chart. For more information, see Chapter 11, “Making States Reusable with Atomic Subcharts”.

box

Graphical object that groups together other graphical objects in your chart. For details about how a box affects chart execution, see “Grouping Chart Objects with Boxes” on page 7-50.

chart instance

Link from a model to a chart stored in a Simulink library. A chart in a library can have many chart instances. Updating the chart in the library automatically updates all instances of that chart.

condition

Boolean expression to specify that a transition occurs when the specified expression is true.

connective junction

Illustrates decision points in the system. A connective junction is a graphical object that simplifies Stateflow chart representations and facilitates generation of efficient code. Connective junctions provide different ways to represent desired system behavior.

See "Connective Junctions" on page 2-36 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-7.

decomposition

A state has a *decomposition* when it consists of one or more substates. A chart that contains at least one state also has decomposition. Rules govern how you can group states in the hierarchy. A superstate has either parallel (AND) or exclusive (OR) decomposition. All substates at a particular level in the hierarchy must have the same decomposition.

- Parallel (AND) State Decomposition

  Parallel (AND) state decomposition applies when states have dashed borders. This decomposition describes states at that same level in the hierarchy that can be active at the same time. The activity within parallel states is essentially independent.

- Exclusive (OR) State Decomposition

  Exclusive (OR) state decomposition applies when states have solid borders. This decomposition describes states that are mutually exclusive. Only one state at the same level in the hierarchy can be active at a time.

default transition

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

See “Default Transitions” on page 2-31 for more information.

events

Events drive chart execution. All events that affect the chart must be defined. The occurrence of an event causes the status of states in a chart to be evaluated. The broadcast of an event can trigger a transition to occur or an action to execute. Events are broadcast in a top-down manner starting from the event’s parent in the hierarchy.

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-8.

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

A chart function whose logic is defined by a flow graph. See “Graphical Functions for Reusing Logic Patterns and Iterative Loops” on page 7-30.

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 1-8 for more information.

history junction

Specifies the destination substate of a transition based on historical information. If a superstate has a history junction, the transition to the destination substate is the substate that was most recently active. The history junction applies only to the level of the hierarchy in which it appears.

See the following sections for more information:

- “History Junctions” on page 2-43
- “Default Transition and a History Junction Example” on page B-20
- “Labeled Default Transitions Example” on page B-22
- “Inner Transition to a History Junction Example” on page B-31

inner transitions

Transition that does not exit the source state. Inner transitions are useful when defined for superstates with exclusive (OR) decomposition. Use of inner transitions can greatly simplify chart layout.

See “Inner Transitions” on page 2-26 and “Inner Transition to a History Junction Example” on page B-31 for more information.

library link

Link to a chart that is stored in a library model.

library model

Stateflow model that is stored in a Simulink library. You can include charts from a library in your model by copying them. When you copy a chart from a library into your model, you create only a link to the library chart. You can create multiple links to a single chart. Each link is called a *chart instance*. When you include a chart from a library in your model, you also include its Stateflow machine. Therefore, a Stateflow model that includes links to library charts has multiple Stateflow machines.

When you simulate a model that includes charts from a library model, you include all charts from the library model even if links exist only for some of its models. 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 when all charts in the library model are free of parse and compile errors.

machine
Collection of all Stateflow blocks defined by a Simulink model. This collection excludes chart instances from library links. If a model includes any library links, it also includes the Stateflow machines defined by the models from which the links originate.

MATLAB function
A chart function that works with a subset of the MATLAB programming language. See Chapter 23, "Using MATLAB Functions in Stateflow Charts" for more information.

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 27-2 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 includes:

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

parallelism

A system with *parallelism* can have two or more states that can be active at the same time. The activity of parallel states is essentially independent. Parallelism is represented with a parallel (AND) state decomposition.

See "State Decomposition" on page 2-10 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 25-74.

semantics

Semantics describe how the notation is interpreted and implemented behind the scenes. A completed Stateflow chart communicates how the system will behave. A chart contains actions associated with transitions and states. The semantics describe in what sequence these actions take place during chart execution.

Simulink function

A chart function that you fill with Simulink blocks and call in the actions of states and transitions. This function provides an efficient model design and improves readability by minimizing the graphical and nongraphical objects required in a model. In a Stateflow chart, this function acts like a function-call subsystem block of a Simulink model.

See Chapter 24, “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 chart represent these modes. The activity or inactivity of the states dynamically changes based on transitions among events and conditions.

Every state has hierarchy. In a chart consisting of a single state, the parent of that state is the Stateflow chart itself. A state also has history that applies to its level of hierarchy in the chart. States can have actions that execute in a sequence based upon action type. The action types are `entry`, `during`, `exit`, or `on event_name` actions.

Stateflow block

Masked Simulink model that is equivalent to an empty, untitled Stateflow chart. Use the Stateflow block to include a chart in a Simulink model.

The control behavior modeled by a Stateflow block complements the algorithmic behavior modeled in Simulink block diagrams. By incorporating Stateflow blocks into Simulink models, you can add complex event-driven behavior to Simulink simulations. You create models that represent both data and decision flow by combining Stateflow blocks with the standard Simulink and toolbox block libraries.

Stateflow chart

A Stateflow chart is a graphical representation of a finite state machine where *states* and *transitions* form the basic building blocks of the system. See “Stateflow Charts and Simulink Models” on page 1-4 for more information.

Stateflow Debugger

Tool for debugging and animating your Stateflow charts. Each state in the 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 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 27-28 for more information.

subchart

Chart contained by another chart. See “Using Subcharts to Encapsulate Modal Logic” on page 7-6.

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 “Moving Between Different Levels of Hierarchy with Supertransitions” on page 7-12 for more information.

target

A container object for the generated code from the Stateflow charts in a model. The collection of all charts for a model appears as a Stateflow machine. Therefore, target objects belong to the Stateflow machine.

The code generation process can produce these target types: simulation, embeddable, and custom. See Chapter 25, “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. Usually, the occurrence of an event causes a transition to take place.

transition path

Flow path that starts and ends on a state.

transition segment

A state-to-junction, junction-to-junction, or junction-to-state part of a complete state-to-state transition.

truth table function

A chart function that specifies logical behavior with conditions, decisions, and actions. Truth tables are easier to program and maintain than graphical functions.

See Chapter 22, “Truth Table Functions for Decision-Making Logic” for instructions on how to use truth tables in Stateflow charts.

Index