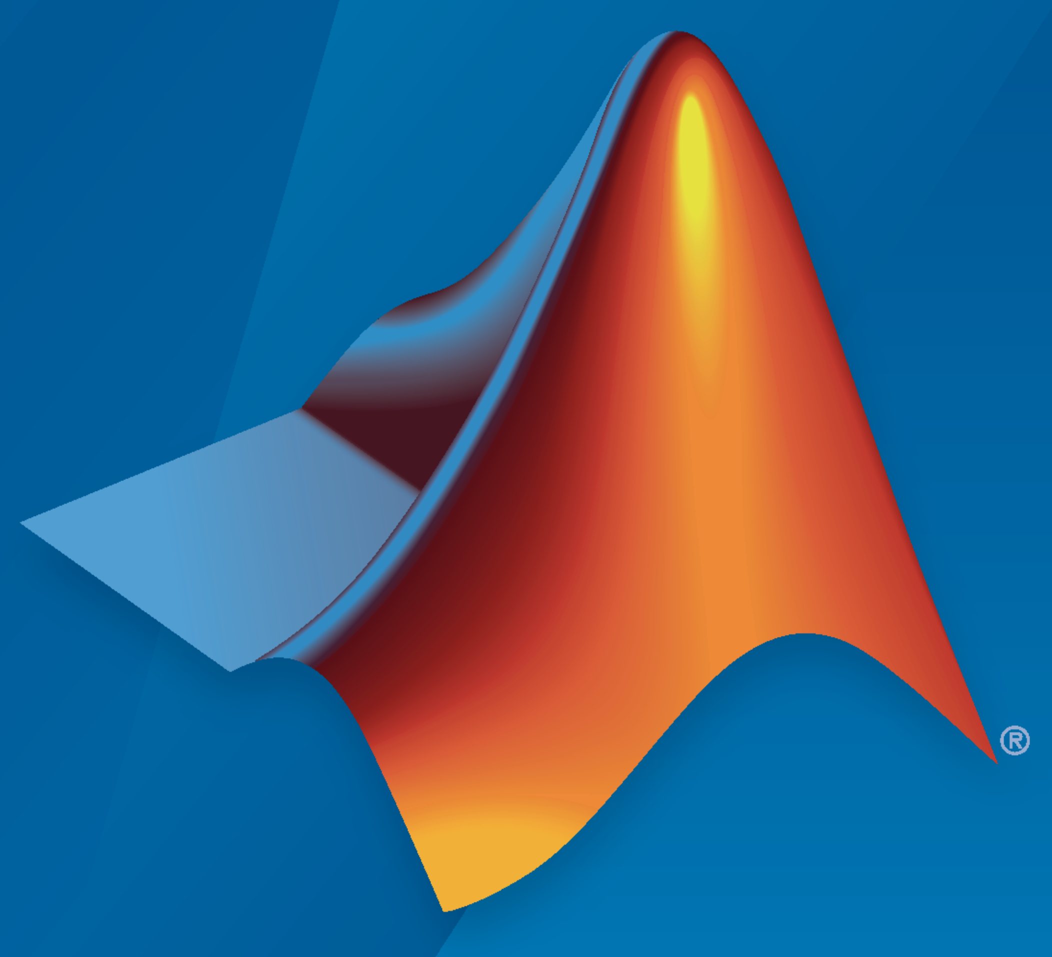
Stateflow®

User's Guide



MATLAB® & SIMULINK®

R2023a

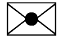
How to Contact MathWorks

Latest news: www.mathworks.com

Sales and services: www.mathworks.com/sales_and_services

User community: www.mathworks.com/matlabcentral

Technical support: www.mathworks.com/support/contact_us



Phone: 508-647-7000

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

Stateflow® User's Guide

© COPYRIGHT 1997–2023 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)
September 2012	Online only	Revised for Version 8.0 (Release 2012b)
March 2013	Online only	Revised for Version 8.1 (Release 2013a)
September 2013	Online only	Revised for Version 8.2 (Release 2013b)
March 2014	Online only	Revised for Version 8.3 (Release 2014a)
October 2014	Online only	Revised for Version 8.4 (Release 2014b)
March 2015	Online only	Revised for Version 8.5 (Release 2015a)
September 2015	Online only	Revised for Version 8.6 (Release 2015b)
October 2015	Online only	Rereleased for Version 8.5.1 (Release 2015aSP1)
March 2016	Online only	Revised for Version 8.7 (Release 2016a)
September 2016	Online only	Revised for Version 8.8 (Release 2016b)
March 2017	Online only	Revised for Version 8.9 (Release 2017a)
September 2017	Online only	Revised for Version 9.0 (Release 2017b)
March 2018	Online only	Revised for Version 9.1 (Release 2018a)
September 2018	Online only	Revised for Version 9.2 (Release 2018b)
March 2019	Online only	Revised for Version 10.0 (Release 2019a)
September 2019	Online only	Revised for Version 10.1 (Release 2019b)
March 2020	Online only	Revised for Version 10.2 (Release 2020a)
September 2020	Online only	Revised for Version 10.3 (Release 2020b)
March 2021	Online only	Revised for Version 10.4 (Release 2021a)
September 2021	Online only	Revised for Version 10.5 (Release 2021b)
March 2022	Online only	Revised for Version 10.6 (Release 2022a)
September 2022	Online only	Revised for Version 10.7 (Release 2022b)
March 2023	Online only	Revised for Version 10.8 (Release 2023a)

Contents

1 Stateflow Chart Programming

Model Finite State Machines by Using Stateflow Charts **1-2**
Types of Stateflow Blocks **1-2**
Program a Stateflow Chart **1-2**

Overview of Stateflow Objects **1-5**
Graphical Objects **1-5**
Nongraphical Objects **1-6**

How Stateflow Objects Interact During Execution **1-8**

Specify Properties for Stateflow Charts **1-19**
Stateflow Chart Properties **1-19**
Fixed-Point Properties **1-23**
Additional Properties **1-24**
Machine Properties **1-24**

Represent Operating Modes by Using States **1-26**
Create a State **1-27**
Define Actions in a State **1-27**
Group States **1-30**
Specify Properties for States **1-30**

Use State Hierarchy to Design Multilevel State Complexity **1-33**
State Hierarchy Example **1-33**
Create Substates and Superstates **1-33**
Objects That a State Can Contain **1-34**

Define Exclusive and Parallel Modes by Using State Decomposition . . . **1-35**
Exclusive (OR) State Decomposition **1-35**
Parallel (AND) State Decomposition **1-35**
Specify Substate Decomposition **1-36**
Specify Activation Order for Parallel States **1-36**

Transition Between Operating Modes **1-37**
Create a Transition **1-38**
Define Actions in a Transition **1-39**
Change Transition Arrowhead Size **1-41**
Specify Properties for Transitions **1-42**

Use Default Transitions to Specify Initial Substate Activity **1-44**
Drawing Default Transitions **1-44**
Label Default Transitions **1-44**
Default Transition Examples **1-44**

Move Between Levels of Hierarchy by Using Supertransitions 1-48
Create a Supertransition That Enters a Subchart 1-49
Create a Supertransition That Exits a Subchart 1-51
Decide Between Supertransitions and Entry and Exit Ports 1-52

Combine Transitions and Junctions to Create Branching Paths 1-54
Add a Connective Junction 1-54
Modify Connective Junction Properties 1-54
Examples of Transition Paths with Connective Junctions 1-54
Specify Properties for Connective Junctions 1-56

Resume Prior Substate Activity by Using History Junctions 1-58
Add a History Junction 1-59
Specify Properties for History Junctions 1-59

Create Entry and Exit Connections Across State Boundaries 1-61
Add Entry and Exit Ports 1-62
Guidelines for Using Entry and Exit Ports 1-62
Decide Between Supertransitions and Entry and Exit Ports 1-63
Specify Properties for Entry and Exit Ports 1-63

Guidelines for Naming Stateflow Objects 1-66
Reserved Keywords 1-66

Speed Up Simulation 1-70
Improve Model Update Performance 1-70
Disable Simulation Target Parameters That Impact Execution Speed . . . 1-70
Speed Up Simulation 1-70

2 Stateflow Semantics

Stateflow Semantics 2-2
Stateflow Objects 2-2
Graphical Objects 2-3
Nongraphical Objects 2-4
Enter a Chart 2-5
Execute an Active Chart 2-5
Enter a State 2-5
Execute an Active State 2-6
Exit an Active State 2-6
Execute a Set of Flow Charts 2-6
Execute an Event Broadcast 2-7

Modeling Guidelines for Stateflow Charts 2-8
Use signals of the same data type for input events 2-8
Use a default transition to mark the first state to become active among exclusive (OR) states 2-8
Use condition actions instead of transition actions whenever possible 2-8
Use explicit ordering to control the testing order of a group of transitions 2-8
Verify intended backtracking behavior in flow charts 2-8

Use a superstate to enclose substates that share the same state actions 2-8
Use MATLAB functions for performing numerical computations in a chart 2-9
Use descriptive names in function signatures 2-9
Use history junctions to record state history 2-9
Do not use history junctions in states with parallel (AND) decomposition 2-9
Use explicit ordering to control the execution order of parallel (AND) states 2-9

Types of Chart Execution 2-10
Life Cycle of a Stateflow Chart 2-10
Execution of an Inactive Chart 2-10
Execution of an Active Chart 2-10
Execution of a Chart at Initialization 2-10

Execution of a Stateflow Chart 2-12
Workflow for Stateflow Chart Execution 2-12
Default Transitions 2-14
Outer Transition 2-14
During Actions 2-14
Inner Transitions 2-14
Chart Execution with a Valid Transition 2-14
Chart Execution Without a Valid Transition 2-15

Enter a Chart or State 2-17
Workflow for Entering a Chart or State 2-17
Chart Entry 2-19
State Entry 2-19
Entry Actions 2-19
Enter a Stateflow Chart 2-19
Entering a State by Using History Junctions 2-20
Entering a State by Using Supertransitions 2-21

Exit a State 2-23
Workflow for Exiting a State 2-23
Exit Actions 2-23
Exit a State Example 2-24
Exit a State by Using Supertransitions 2-24

Evaluate Transitions 2-26
Workflow for Evaluating Transitions 2-27
Transition Evaluation Order 2-28
Transition to the Inner Edge of a Parent State 2-29
Evaluate Outer Transition 2-29
Evaluate Outer Transition with Backtracking 2-30
Evaluate Outer Transitions with Condition and Transition Actions 2-32

Super Step Semantics 2-35
Maximum Number of Iterations 2-35
Enable Super Step Semantics 2-35
Example of Chart with Super Step Semantics 2-36
How Super Step Semantics Works with Multiple Input Events 2-37
Detection of Infinite Loops in Transition Cycles 2-39

Use Events to Execute Charts **2-40**
How Stateflow Charts Respond to Events **2-40**
Events in Simulink Models **2-40**
Events in Standalone Charts **2-42**

Group and Execute Transitions **2-44**
Transition Flow Chart Types **2-44**
Order of Execution for a Set of Flow Charts **2-44**

Execution Order for Parallel States **2-46**
Ordering for Parallel States **2-46**
Explicit Ordering of Parallel States **2-46**
Implicit Ordering of Parallel States **2-47**
Order Maintenance for Parallel States **2-48**
Execution Priorities in Restored States **2-49**
Switching Between Explicit and Implicit Ordering **2-50**
Execution Order of Parallel States in Boxes and Subcharts **2-50**

3 Model Logic Patterns and Iterative Loops Using Flow Charts

Create Flow Charts in Stateflow **3-2**
Draw a Flow Chart **3-2**
Best Practices for Creating Flow Charts **3-3**

Create Flow Charts by Using Pattern Wizard **3-5**
Create Reusable Flow Charts **3-5**
Insert Logic Patterns in Existing Flow Charts **3-6**
Save Custom Flow Chart Patterns **3-9**
Reuse Custom Flow Chart Patterns **3-10**
MAB-Compliant Patterns from the Pattern Wizard **3-11**

Convert MATLAB Code into Stateflow Flow Charts **3-17**
Create Flow Charts from MATLAB Scripts **3-17**

4 Simulink Subsystems as Stateflow States

Simulink Subsystems as States **4-2**
When to Use Simulink Based States **4-2**
Model a Pole Vaulter by Using Simulink Based States **4-2**
Limitations **4-7**

Create and Edit Simulink Based States **4-8**
Create a Simulink Based State **4-8**
Create Inports and Outports **4-11**

Access Block State Data **4-14**
Textual Access **4-16**

Graphical Access 4-18

Map Variables for Simulink Based States 4-20
Map Variables in a Simulink Based State 4-20

Set Simulink Based State Properties 4-22
Simulink Based State Properties 4-22

Hybrid Clutch System 4-24

5 Build Mealy and Moore Charts

Overview of Mealy and Moore Machines 5-2
Semantics of Mealy and Moore Machines 5-2
Create Mealy and Moore Charts 5-3
Advantages of Mealy and Moore Charts 5-3

Design Considerations for Mealy Charts 5-5
Mealy Semantics 5-5
Design Guidelines for Mealy Charts 5-5

Model a Vending Machine by Using Mealy Semantics 5-7

Design Considerations for Moore Charts 5-9
Moore Semantics 5-9
Design Guidelines for Moore Charts 5-9

Model a Traffic Light by Using Moore Semantics 5-12

Convert Charts Between Mealy and Moore Semantics 5-14
Transform Chart from Mealy to Moore Semantics 5-14
Transform Chart from Moore to Mealy Semantics 5-15

Sequence Recognition by Using Mealy and Moore Charts 5-18

Karplus-Strong Algorithm by Using Moore Charts 5-22

Initialize Persistent Variables in MATLAB Functions 5-24
MATLAB Function Block with No Direct Feedthrough 5-25
State Control Block in Synchronous Mode 5-26
Stateflow Chart Implementing Moore Semantics 5-27

6 Techniques for Streamlining Chart Design

Group Chart Objects by Using Boxes 6-2
Semantics of Stateflow Boxes 6-2
Draw and Edit a Box 6-3

Examples of Using Boxes . . . 6-3

Encapsulate Modal Logic by Using Subcharts . . . **6-6**
Create a Subchart . . . 6-6
Rules of Subchart Conversion . . . 6-6
Convert a State to a Subchart . . . 6-6
Manipulate Subcharts as Objects . . . 6-7
Open a Subchart . . . 6-7
Edit a Subchart . . . 6-7
Navigate Subcharts . . . 6-8

Reuse Logic Patterns by Defining Graphical Functions . . . **6-9**
Define a Graphical Function . . . 6-9
Call Graphical Functions in States and Transitions . . . 6-11
Manage Large Graphical Functions . . . 6-11
Specify Properties of Graphical Functions . . . 6-11

Export Stateflow Functions for Reuse . . . **6-14**
Share Functions Across Stateflow Charts . . . 6-14
Guidelines for Exporting Chart-Level Functions . . . 6-16

Reuse Functions by Using Atomic Boxes . . . **6-18**
Example of an Atomic Box . . . 6-18
Benefits of Using Atomic Boxes . . . 6-19
Create an Atomic Box . . . 6-19
When to Use Atomic Boxes . . . 6-21

Add Descriptive Comments in a Chart . . . **6-22**
Change Annotation Properties . . . 6-22
Include TeX Formatting Instructions . . . 6-23

7 MATLAB Functions in Stateflow Charts

Reuse MATLAB Code by Defining MATLAB Functions . . . **7-2**
Define a MATLAB Function in a Chart . . . 7-2
Call MATLAB Functions in States and Transitions . . . 7-3
Specify Properties of MATLAB Functions . . . 7-4

Program a MATLAB Function in a Chart . . . **7-7**
Build Model . . . 7-7
Program MATLAB Functions . . . 7-10

Access Simulink Bus Signals in MATLAB Functions . . . **7-13**

Debug a MATLAB Function in a Chart . . . **7-16**
Check MATLAB Functions for Syntax Errors . . . 7-16
Run-Time Debugging for MATLAB Functions in Charts . . . 7-16
Check for Data Range Violations . . . 7-18

8 Truth Table Functions for Decision-Making Logic

Use Truth Tables to Model Combinatorial Logic **8-2**
Layout of a Truth Table **8-2**
Define a Truth Table Function **8-3**
Call Truth Table Functions in States and Transitions **8-4**
Specify Properties of Truth Table Functions **8-5**
Specify Properties for Truth Table Blocks **8-6**

Program a Truth Table **8-8**
Open a Truth Table for Editing **8-8**
Select an Action Language **8-8**
Enter Truth Table Conditions **8-9**
Enter Truth Table Decisions **8-10**
Enter Truth Table Actions **8-11**
Assign Truth Table Actions to Decisions **8-15**
Add Initial and Final Actions **8-18**

Debug Errors in a Truth Table **8-21**
Find Syntax Errors by Running Diagnostics **8-21**
Debug Logic by Using Breakpoints **8-21**
Edit Breakpoints **8-26**

Correct Overspecified and Underspecified Truth Tables **8-28**
Example of an Overspecified Truth Table **8-28**
Example of an Underspecified Truth Table **8-29**

Home Climate Control Using the Truth Table Block **8-32**

9 Simulink Functions in Stateflow Charts

Reuse Simulink Functions in Stateflow Charts **9-2**
Define a Simulink Function **9-3**
Call Simulink Functions in States and Transitions **9-3**
Specify Properties of Simulink Functions **9-4**
Use a Simulink Function to Access Simulink Blocks **9-4**
Use a Simulink Function to Schedule Execution of Multiple Controllers . . **9-6**
Guidelines for Using Simulink Functions **9-8**

Bind a Simulink Function to a State **9-10**
Control Subsystem Variables When the Simulink Function Is Disabled . . **9-10**
Binding a Simulink Function to a State **9-10**

Design Charts with Simulink Functions **9-15**
Open the Model **9-15**
Add a Simulink Function to the Chart **9-16**
Change the Scope of Chart Data **9-17**
Update State Action in the Chart **9-17**
Add Data to the Chart **9-18**

Remove Unused Items in the Model . . . 9-18
Run the New Model . . . 9-18

Schedule Execution of Multiple Controllers . . . 9-20
Open the Model . . . 9-20
Add Simulink Functions to the Chart . . . 9-22
Change the Scope of Chart Data . . . 9-23
Update State Actions in the Chart . . . 9-23
Add Data to the Chart . . . 9-24
Remove Unused Items in the Model . . . 9-24
Run the New Model . . . 9-24

Schedule Simulink Functions by Using Stateflow . . . 9-26

Design Switching Controllers by Using Simulink Functions . . . 9-28

Manage Queue for Shared Printer Server . . . 9-32

10 Define Data

Add Stateflow Data . . . 10-2
Add Data Through the Symbols Pane . . . 10-2
Add Data by Using the Stateflow Editor Menu . . . 10-2
Add Data Through the Model Explorer . . . 10-3
Best Practices for Using Data in Charts . . . 10-3

Set Data Properties . . . 10-5
Stateflow Data Properties . . . 10-5
Fixed-Point Data Properties . . . 10-10
Logging Properties . . . 10-16
Additional Properties . . . 10-16
Default Data Property Values . . . 10-17
Specify Data Properties by Using MATLAB Expressions . . . 10-18

Specify Type of Stateflow Data . . . 10-20
Specify Data Type by Using the Data Type Assistant . . . 10-20
Inherit Data Types from Simulink Objects . . . 10-23
Derive Data Types from Other Data Objects . . . 10-23
Specify Data Types by Using a Simulink Alias . . . 10-24

Specify Size of Stateflow Data . . . 10-26
Inherit Data Size . . . 10-26
Specify Data Size by Using Numeric Values . . . 10-26
Specify Data Size by Using Expressions . . . 10-27

Specify Units for Stateflow Data . . . 10-29
Units for Input and Output Data . . . 10-29
Consistency Checking . . . 10-29
Units for Stateflow Limitations . . . 10-29

Share Data with Simulink and the MATLAB Workspace **10-30**
Share Input and Output Data with Simulink **10-30**
Initialize Data from the MATLAB Base Workspace **10-30**
Save Data to the MATLAB Base Workspace **10-31**

Share Parameters with Simulink and the MATLAB Workspace **10-32**
Initialize Parameters from the MATLAB Base Workspace **10-32**
Share Simulink Parameters with Charts **10-32**

Access Data Store Memory from a Chart **10-34**
Local and Global Data Store Memory **10-34**
Bind Stateflow Data to Data Stores **10-34**
Store and Retrieve Global Data **10-35**
Best Practices for Using Data Stores **10-35**

Handle Integer Overflow for Chart Data **10-37**
When Integer Overflow Can Occur **10-37**
Support for Handling Integer Overflow in Charts **10-37**
Effect of Integer Promotion Rules on Saturation **10-37**
Impact of Saturation on Error Checks **10-38**

Identify Data by Using Dot Notation **10-39**
Resolution of Qualified Data Names **10-39**
Best Practices for Using Dot Notation **10-40**
Examples of Qualified Data Name Resolution **10-40**

Resolve Data Properties from Simulink Signal Objects **10-43**

11 Active State Data

Monitor State Activity Through Active State Data **11-2**
Types of Active State Data **11-2**
Enable Active State Data **11-2**
Set Scope for Active State Data **11-3**
Define State Activity Enumeration Type **11-4**
State Activity and Parallel States **11-5**
Limitations for Active State Data **11-6**

View State Activity by Using the Simulation Data Inspector **11-7**
Add Signals and States for Logging **11-7**
View Logged Output in Simulation Data Inspector **11-8**

View Stateflow States in the Logic Analyzer **11-10**
Add Signals and States for Logging **11-10**
View Logged Output in Logic Analyzer **11-11**

Log Simulation Output for States and Data **11-13**
Enable Signal Logging **11-13**
Configure States and Data for Logging **11-13**
Access Signal Logging Data **11-15**
Log Multidimensional Data **11-18**

Limitations on Logging Data 11-18

Log Data in Library Charts 11-20
How Library Log Settings Influence Linked Instances 11-20
Override Logging Properties in Chart Instances 11-20
Override Logging Properties in Atomic Subcharts 11-20

Check State Activity by Using the in Operator 11-24
The in Operator 11-24
Resolution of State Activity 11-24
Best Practices for Checking State Activity 11-25
Examples of State Activity Resolution 11-26

Simplify Stateflow Charts by Incorporating Active State Output 11-30

Model An Intersection Of One-Way Streets 11-35

Monitor Test Points in Stateflow Charts 11-38

12 Define Events

Synchronize Model Components by Broadcasting Events 12-2
Types of Events 12-2
Define Events in a Chart 12-2
Access Event Information from a Stateflow Chart 12-3
Best Practices for Using Events in Stateflow Charts 12-4

Set Properties for an Event 12-5
Stateflow Event Properties 12-5

Activate a Stateflow Chart by Sending Input Events 12-8
Activate a Stateflow Chart by Using Edge Triggers 12-8
Activate a Stateflow Chart by Using Function Calls 12-9
Association of Input Events with Control Signals 12-10
Data Types Allowed for Input Events 12-10

Control States in Charts Enabled by Function-Call Input Events 12-12

Activate a Simulink Block by Sending Output Events 12-15
Broadcast Output Events 12-15
Activate a Simulink Block by Using Edge Triggers 12-15
Activate a Simulink Block by Using Function Calls 12-18
Approximate a Function Call by Using Edge-Triggered Events 12-21
Association of Output Events with Output Ports 12-24

Broadcast Local Events to Synchronize Parallel States 12-25
Broadcast Local Events 12-25
Use Qualified Event Names in Event Broadcasts 12-26
Undirected Event Broadcasts 12-26

Control Chart Behavior by Using Implicit Events **12-28**
Implicit Events Based on Chart Execution **12-28**
Implicit Events Based on Data and States **12-28**

Yo-Yo Control of Satellites **12-31**

13 Messages

Communicate with Stateflow Charts by Sending Messages **13-2**
Define Messages in a Chart **13-2**
Lifetime of a Stateflow Message **13-3**
Limitations for Messages **13-4**

Set Properties for a Message **13-5**
Stateflow Message Properties **13-5**
Message Queue Properties **13-7**

Control Message Activity in Stateflow Charts **13-9**
Access Message Data **13-9**
Send a Message **13-9**
Guard Transitions and Actions **13-10**
Receive a Message **13-11**
Discard a Message **13-11**
Forward a Message **13-12**
Determine if a Message Is Valid **13-12**
Determine the Length of the Queue **13-13**

View Differences Between Stateflow Messages, Events, and Data **13-14**

Model Distributed Traffic Control System by Using Messages **13-20**

Use the Sequence Viewer to Visualize Messages, Events, and Entities **13-24**
Components of the Sequence Viewer Window **13-25**
Navigate the Lifeline Hierarchy **13-27**
View State Activity and Transitions **13-29**
View Function Calls **13-30**
Simulation Time in the Sequence Viewer Window **13-31**
Redisplay of Information in the Sequence Viewer Window **13-32**

Build a Shared Communication Channel with Multiple Senders and Receivers **13-33**

Model Wireless Message Communication with Packet Loss and Channel Failure **13-39**

Model an Ethernet Communication Network with CSMA/CD Protocol **13-49**

14 Use Actions in Charts

Eliminate Redundant Code by Combining State Actions **14-2**
How to Combine State Actions **14-2**
Order of Execution of Combined Actions **14-3**
Rules for Combining State Actions **14-3**

Operations for Stateflow Data **14-4**
Binary Operations **14-4**
Unary Operations and Actions **14-6**
Assignment Operations **14-6**
Type Cast Operations **14-7**
Bitwise Operations **14-8**
Pointer and Address Operations **14-9**
Replace Operations with Application Implementations **14-9**

Supported Symbols in Actions **14-11**
Boolean Symbols, true and false **14-11**
Comment Symbols, %, //, /* **14-11**
Hexadecimal Notation Symbols, 0xFF **14-12**
Infinity Symbol, inf **14-12**
Line Continuation Symbol, ... **14-12**
MATLAB Display Symbol, ; **14-12**
Single-Precision Floating-Point Number Symbol, F **14-12**
Time Symbol, t **14-12**

Call Extrinsic MATLAB Functions in Stateflow Charts **14-13**
Use the coder.extrinsic Function **14-13**

Call C Library Functions in C Charts **14-16**
Call C Library Functions **14-16**
Call the abs Function **14-16**
Call min and max Functions **14-17**
Replacement of Math Library Functions with Application Implementations **14-17**
Call Custom C Code Functions **14-17**

Access MATLAB Functions and Workspace Data in C Charts **14-20**
ml Namespace Operator **14-20**
ml Function **14-21**
ml Expressions **14-22**
Which ml Should I Use? **14-22**
ml Data Type **14-23**
How Charts Infer the Return Size for ml Expressions **14-25**

Control Function-Call Subsystems by Using bind Actions **14-29**
Bind a Function-Call Subsystem to a State **14-29**
Bind a Function-Call Subsystem to a State **14-31**
Avoid Muxed Trigger Events with Binding **14-33**

Control Chart Execution by Using Temporal Logic **14-35**
Temporal Logic Operators **14-35**
Examples of Temporal Logic **14-41**

Notation for Event-Based Temporal Logic in Transitions 14-45
Best Practices for Temporal Logic 14-46

Model Bang-Bang Temperature Control System 14-51

Control Oscillations by Using the duration Operator 14-55
Control Oscillation with Parallel State Logic 14-55
Control Oscillation with the duration Operator 14-56

Implement an Automatic Transmission Gear System by Using the duration Operator 14-58

Count Events by Using the temporalCount Operator 14-61

Detect Changes in Data and Expression Values 14-63
Change Detection Operators 14-63
Edge Detection Operators 14-67
Implementation of Change and Edge Detection 14-72

Design a Game by Using Stateflow 14-75

Model an Automatic Transmission Controller 14-79

Vehicle Electrical and Climate Control Systems 14-90

Developing the Apollo Lunar Module Digital Autopilot 14-96

Design a Guidance System in MATLAB and Simulink 14-105

15 MATLAB Syntax Support for States and Transitions

Modify the Action Language for a Chart 15-2
Change the Default Action Language 15-2
Auto Correction When Using MATLAB as the Action Language 15-2
C to MATLAB Syntax Conversion 15-2

Differences Between MATLAB and C as Action Language Syntax 15-4
Compare Functionality of Action Languages 15-4
Guidelines for Using MATLAB as the Action Language 15-7

16 Tabular Expression of Modal Logic

Use State Transition Tables to Express Sequential Logic in Tabular Form 16-2
Program a State Transition Table 16-4
Detect Errors in State Transition Tables 16-7
Specify Properties for State Transition Tables 16-7

Guidelines for Using State Transition Tables 16-8

Inspect the Design of State Transition Tables 16-9

Debug Run-Time Errors in a State Transition Table 16-16
Create the Model and the State Transition Table 16-16
Debug the State Transition Table 16-17
Correct the Run-Time Error 16-17

Model Bang-Bang Controller by Using a State Transition Table 16-19
Design Requirements 16-19
Identify System Attributes 16-19
Add a New State Transition Table 16-20
Add States and Hierarchy 16-21
Specify State Actions 16-22
Specify Transition Conditions and Actions 16-24
Define Data 16-26
Connect the Transition Table and Run the Model 16-28

Modeling a CD Player/Radio Using State Transition Tables 16-30

17 Make States Reusable with Atomic Subcharts

Create Reusable Subcomponents by Using Atomic Subcharts 17-2
Example of an Atomic Subchart 17-2
Benefits of Using Atomic Subcharts 17-3
Create an Atomic Subchart 17-3
When to Use Atomic Subcharts 17-4

Guidelines for Using Atomic Subcharts 17-7
Chart Properties and Atomic Subcharts 17-7
Data in Atomic Subcharts 17-7
Events in Atomic Subcharts 17-8
Functions and Atomic Subcharts 17-9
Restrictions for Converting to Atomic Subcharts 17-9

Map Variables for Atomic Subcharts and Boxes 17-11
Map Input and Output Data for an Atomic Subchart 17-11
Map Atomic Subchart Variables to Bus Elements 17-15
Map Atomic Subchart Variables to the Elements of a Matrix 17-16
Map Atomic Subchart Parameters to Expressions 17-18
Map Input Events for an Atomic Subchart 17-20

Robot Trajectory Planning with Reusable Components 17-24

Reuse a State Multiple Times in a Chart 17-32

Reduce the Compilation Time of a Chart 17-38

Divide a Chart into Separate Units 17-41

Generate Separate Code for an Atomic Subchart **17-44**

Model a Redundant Sensor Pair by Using Atomic Subcharts **17-48**

Model an Elevator System by Using Atomic Subcharts **17-52**

18 Save and Restore Simulations with Operating Points

Save and Restore Operating Points for Stateflow Charts **18-2**
Save Operating Points **18-2**
Modify Operating Point Values **18-4**
Restore Operating Points **18-5**
Limitations on Operating Points **18-6**

Use Operating Points to Specify Initial State of Simulation **18-7**
Save Operating Point for Initial Simulation Segment **18-8**
Start Simulation from Operation Point **18-9**

Test Difficult-to-Reproduce Chart Configurations **18-12**
Define Operating Point for Initial Segment **18-14**
Modify Operating Point for Change in Data Value **18-14**
Test Model Behavior after Change in Data Value **18-16**

Test Chart with Fault Detection and Redundant Logic **18-18**
Define Operating Point for Steady State **18-20**
Modify Operating Point Values for Actuator Failure **18-21**
Test Model Behavior after Actuator Failure **18-23**

19 Vectors and Matrices in Stateflow Charts

Vectors and Matrices in Stateflow Charts **19-2**
Define Vector and Matrix Data **19-2**
Where You Can Use Vectors and Matrices **19-2**
Rules for Vectors and Matrices in Stateflow Charts **19-3**

Operations for Vectors and Matrices in Stateflow **19-4**
Indexing Notation **19-4**
Binary Operations **19-4**
Unary Operations and Actions **19-5**
Assignment Operations **19-6**
Perform Matrix Arithmetic by Using MATLAB Functions **19-7**

Declare Variable-Size Data in Stateflow Charts **19-9**
Enable Support for Variable-Size Data **19-9**
Variable-Size Data in Charts That Use MATLAB as the Action Language **19-9**
Variable-Size Data in Charts That Use C as the Action Language **19-10**

Compute Output Based on Size of Input Signal 19-12

20 Enumerated Data in Charts

Reference Values by Name by Using Enumerated Data **20-2**
Example of Enumerated Data **20-2**
Computation with Enumerated Data **20-2**
Notation for Enumerated Values **20-3**
Where to Use Enumerated Data **20-3**

Define Enumerated Data Types **20-5**
Elements of an Enumerated Data Type Definition **20-5**
Define an Enumerated Data Type **20-5**
Specify Data Type in the Property Inspector **20-7**

Best Practices for Using Enumerated Data **20-8**
Guidelines for Defining Enumerated Data Types **20-8**
Guidelines for Referencing Enumerated Data **20-8**
Guidelines and Limitations for Enumerated Data **20-10**

Assign Enumerated Values in a Chart **20-11**
Chart Behavior **20-11**
Build the Chart **20-11**
View Logged Output **20-12**

Model Media Player by Using Enumerated Data **20-15**

21 String Data in Charts

Manage Textual Information by Using Strings **21-2**
Creating Strings in Stateflow **21-2**
Computation with Strings **21-2**
String Truncation **21-3**
Differences Between Charts That Use MATLAB and C as the Action Language **21-3**
Limitations **21-3**

Log String Data to the Simulation Data Inspector **21-5**

Send Messages With String Data **21-8**

Share String Data with Custom C Code **21-10**

Simulate a Media Player **21-14**

22 Continuous-Time Systems in Stateflow Charts

Continuous-Time Modeling in Stateflow **22-2**
Configure a Stateflow Chart for Continuous-Time Simulation **22-2**
Interaction with Simulink Solver **22-2**
Disable Zero-Crossing Detection **22-3**
Guidelines for Continuous-Time Simulation **22-3**

Store Continuous State Information in Local Variables **22-6**
Define Continuous-Time Variables **22-6**
Compute Implicit Time Derivatives **22-6**
Expose Continuous State to a Simulink Model **22-6**
Guidelines for Continuous-Time Variables **22-6**

Model a Bouncing Ball in Continuous Time **22-8**

Model a DC Motor in Stateflow **22-12**

Model the Dynamics of Moving Billiard Balls **22-14**

Model Newton's Cradle **22-19**

Modeling Newton's Cradle with Virtual Reality **22-23**

23 Fixed-Point Data in Stateflow Charts

Fixed-Point Data in Stateflow Charts **23-2**
Fixed-Point Numbers **23-2**
Specify Fixed-Point Data **23-2**
Conversion Operations **23-3**
Fixed-Point Context-Sensitive Constants **23-4**
Tips for Using Fixed-Point Data **23-5**
Automatic Scaling of Fixed-Point Data **23-6**
Share Fixed-Point Data with Simulink Models **23-6**
Implementation of Fixed-Point Data in Stateflow **23-6**

Operations for Fixed-Point Data in Stateflow **23-8**
Binary Operations **23-8**
Unary Operations and Actions **23-10**
Assignment Operations **23-10**
Compare Results of Fixed-Point Arithmetic **23-12**

Build a Low-Pass Filter by Using Fixed-Point Data **23-15**

Fixed-Point Operations in Stateflow Charts **23-19**
Arithmetic Operations for Fixed-Point Data **23-19**
Relational Operations for Fixed-Point Data **23-19**
Logical Operations for Fixed-Point Data **23-20**
Promotion Rules for Fixed-Point Operations **23-20**

Compare Fixed-Point and Floating-Point Computation in Mandelbrot Set **23-25**

24 Complex Data

Complex Data in Stateflow Charts **24-2**
Define Complex Data **24-2**
When to Use Complex Data **24-2**
Where You Can Use Complex Data **24-2**
How You Can Use Complex Data **24-2**

Operations for Complex Data in Stateflow **24-4**
Notation for Complex Data **24-4**
Binary Operations **24-4**
Unary Operations and Actions **24-5**
Assignment Operations **24-5**
Access Real and Imaginary Parts of a Complex Number **24-5**

Best Practices for Using Complex Data in C Charts **24-7**
Perform Math Function Operations with a MATLAB Function **24-7**
Perform Complex Division with a MATLAB Function **24-8**
Rules for Using Complex Data in C Charts **24-9**

Measure Frequency Response by Using Complex Data in Stateflow . . . **24-11**

Detect Valid Transmission Data by Using Frame Synchronization **24-17**

25 Define Interfaces to Simulink Models and the MATLAB Workspace

Stateflow Editor Operations **25-2**
Stateflow Editor **25-2**
Undo and Redo Editor Operations **25-3**
Specify Colors and Fonts in a Chart **25-3**
Content Preview for Stateflow Objects **25-6**
Intelligent Tab Completion for Stateflow Charts **25-7**
Differentiate Elements of Action Language Syntax **25-7**
Zoom and Navigate with the Miniature Map **25-9**
Format Chart Objects **25-10**

Manage Symbols in the Stateflow Editor **25-14**
Add and Modify Data, Events, and Messages **25-14**
Detect Unused Data in the Symbols Pane **25-15**
Resolve Symbols Through the Symbols Pane **25-15**
Resolve Symbols Through the Symbol Wizard **25-16**
Detect Symbol Definitions in Custom Code **25-16**
Trace Data, Events, and Messages **25-17**

Symbols Pane Limitations 25-20

Use the Model Explorer with Stateflow Objects **25-22**
View Stateflow Objects in the Model Explorer 25-22
Edit Chart Objects in the Model Explorer 25-23
Add Data and Events in the Model Explorer 25-23
Rename Objects in the Model Explorer 25-23
Set Properties for Chart Objects in the Model Explorer 25-23
Move and Copy Data and Events in the Model Explorer 25-24
Change the Port Order of Input and Output Data and Events 25-25
Delete Data and Events in the Model Explorer 25-25

Visualize Chart Execution with the Activity Profiler **25-26**
Debug with the Activity Profiler 25-26
Enable the Activity Profiler 25-26
Activity Profiler Preferences 25-28
Explore 25-29

Connect Dashboard Blocks to Stateflow **25-31**
Monitor a Boiler with Dashboard Blocks 25-31

Reuse Charts in Models with Chart Libraries **25-34**
Create Specialized Chart Libraries for Large-Scale Modeling 25-34
Customize Properties of Library Blocks 25-34
Limitations of Library Charts 25-35

Create a Mask to Share Parameters with Simulink **25-36**
Create a Mask for a Stateflow Chart 25-36
Add an Icon to the Mask 25-37
Add Parameters to the Mask 25-37
View the New Mask 25-38
Look Under the Mask 25-38
Edit the Mask 25-38

26 Structures and Bus Signals in Stateflow Charts

Access Bus Signals Through Stateflow Structures **26-2**
Example of Stateflow Structures 26-2
Define Stateflow Structures 26-2
Specify Structure Types by Calling the type Operator 26-4
Virtual and Nonvirtual Buses 26-5
Debug Structures 26-5
Guidelines for Structure Data Types 26-5

Index and Assign Values to Stateflow Structures **26-7**

Integrate Custom Structures in Stateflow Charts **26-11**

27 Stateflow Design Patterns

Schedule Multiple Subsystems in a Single Step **27-2**

Schedule a Subsystem Multiple Times in a Single Step **27-6**

Schedule Subsystems to Execute at Specific Times **27-9**

Reduce Transient Signals by Using Debouncing Logic **27-12**
How to Debounce a Signal **27-12**
Debounce Signals with the duration Operator **27-12**
Debounce Signals with Fault Detection **27-14**
Use Event-Based Temporal Logic **27-17**

Detect Faults in Aircraft Elevator Control System **27-19**

Map Fault Conditions to Actions by Using Truth Tables **27-24**

Design for Isolation and Recovery in a Chart **27-27**
Mode Logic for the Elevator Actuators **27-27**
States for Failure and Isolation **27-28**
Transitions for Recovery **27-29**

Launch Abort System **27-31**

Model a Fault-Tolerant Fuel Control System **27-36**

Model a Power Window Controller **27-51**

Model a Fitness Tracker **27-59**

28 Custom Code

Reuse Custom Code in Stateflow Charts **28-2**
Integrate Custom C Code in Stateflow Charts **28-2**
Configure Custom Code for Your Model **28-4**
Call Custom Code Functions in States and Transitions **28-6**
Specify Relative Paths to Your Custom Code **28-6**

Configure Custom Code in Library Models **28-9**
Configure Custom Code Settings for Simulation **28-9**
Configure Custom Code Settings for Code Generation **28-10**

Access Custom Code Variables and Functions in Stateflow Charts **28-12**
Custom Code Variables in Charts That Use MATLAB as the Action Language **28-12**
Custom Code Functions in Charts That Use MATLAB as the Action Language **28-12**
Accessing Enumerations in Custom Code **28-13**

Model Battery Management with Custom Code **28-14**

Access Custom C++ Code in Stateflow Charts **28-21**

29 Code Generation

Generate C or C++ Code from Stateflow Blocks **29-2**
Generate Code by Using Simulink Coder **29-2**
Generate Code by Using Embedded Coder **29-2**
Design Tips for Optimizing Generated Code for Stateflow Objects **29-3**
Generate Code for Rapid Prototyping and Production Deployment **29-3**
Traceability of Stateflow Objects in Generated Code **29-4**

Select Array Layout for Matrices in Generated Code **29-5**

Control Indicator Lamp Dimmer Using Variant Conditions **29-9**

Generate Code from Atomic Subcharts **29-16**
Generate Reusable Code for Unlinked Atomic Subcharts **29-16**
Generate Reusable Code for Linked Atomic Subcharts **29-16**

Set Configuration Parameters Programmatically **29-18**

Using Absolute Time Temporal Logic in Stateflow Charts **29-19**

30 Debug and Test Stateflow Charts

Set Breakpoints to Debug Charts **30-2**
Set a Breakpoint for a Stateflow Object **30-2**
Change Breakpoint Types **30-4**
Add Breakpoint Conditions **30-4**
Manage Breakpoints Through the Breakpoints and Watch Window **30-6**

Inspect and Modify Data and Messages While Debugging **30-8**
View Data in the Stateflow Editor **30-8**
View and Modify Data in the Symbols Pane **30-9**
View Data in the Breakpoints and Watch Window **30-10**
View and Modify Data in the MATLAB Command Window **30-11**

Control Chart Execution After a Breakpoint **30-15**
Examine the State of the Chart **30-15**
Step Through the Simulation **30-16**

Debug Run-Time Errors in a Chart **30-19**
Create the Model and the Stateflow Chart **30-19**
Debug the Stateflow Chart **30-20**
Correct the Run-Time Error **30-20**

Detect Modeling Errors During Edit Time **30-21**
Manage Edit-Time Checks **30-21**
Edit-Time Checks on States and Subcharts **30-21**
Edit-Time Checks on Transitions **30-24**
Edit-Time Checks on Junctions **30-28**
Edit-Time Checks on Functions **30-30**
Edit-Time Checks on Entry and Exit Ports **30-30**

Detect Common Modeling Errors During Simulation **30-35**
Detect State Inconsistencies **30-35**
Detect Data Range Violations **30-36**
Detect Cyclic Behavior **30-36**
Fix Cyclic Behavior in Flow Charts **30-37**

Animate Stateflow Charts **30-40**
Set Animation Speeds **30-40**
Maintain Highlighting **30-40**
Disable Animation **30-40**
Animate Charts as Generated Code Executes on a Target System **30-40**

Comment Out Objects in a Stateflow Chart **30-41**
Comment Out a Stateflow Object **30-41**
How Commenting Affects the Chart and Model **30-41**
Add Text to a Commented Object **30-42**
Limitations on Commenting Objects **30-42**

Avoid Unwanted Recursion in a Chart **30-43**
Recursive Function Calls **30-43**
Undirected Local Event Broadcasts **30-43**

31 Standalone Stateflow Charts for Execution in MATLAB

Create Stateflow Charts for Execution as MATLAB Objects **31-2**
Construct a Standalone Chart **31-2**
Create a Stateflow Chart Object **31-2**
Execute a Standalone Chart **31-3**
Stop Chart Execution **31-3**
Share Standalone Charts **31-4**
Properties and Functions of Stateflow Chart Objects **31-4**
Capabilities and Limitations **31-6**

Execute and Unit Test Stateflow Chart Objects **31-8**
Example of a Standalone Stateflow Chart **31-8**
Execute a Standalone Chart from the Stateflow Editor **31-8**
Execute a Standalone Chart in MATLAB **31-9**
Execute Multiple Chart Objects **31-12**

Debug a Standalone Stateflow Chart **31-13**
Set and Clear Breakpoints **31-13**
Manage Breakpoint Types and Conditions **31-14**
Control Chart Execution After a Breakpoint **31-15**

Examine and Change Values of Chart Data 31-17

Execute Stateflow Chart Objects Through Scripts and Models 31-18
Count Ways to Make Change for Currency 31-18
Execute Standalone Chart in a MATLAB Script 31-19
Execute Standalone Chart in a Simulink Model 31-20

Design Human-Machine Interface Logic by Using Stateflow Charts . . 31-24

Model a Communications Protocol by Using Chart Objects 31-28

Implement a Financial Strategy by Using Stateflow 31-32

Model a Fitness App by Using Standalone Charts 31-35

Automate Control of Intelligent Vehicles by Using Stateflow Charts . . 31-40

Model Bluetooth Low Energy Link Layer Using Stateflow 31-44

Analog Triggered Data Acquisition Using Stateflow Charts 31-47

A Semantic Examples

Categories of Semantic Examples A-2

Transition Between Exclusive States A-4
Label Format for a State-to-State Transition A-4
Transition from State to State with Events A-4
Transition from a Substate to a Substate with Events A-6

Control Chart Execution by Using Condition Actions A-8
Condition Action Behavior A-8
Condition and Transition Action Behavior A-8
Create Condition Actions Using a For-Loop A-9
Broadcast Events to Parallel (AND) States Using Condition Actions A-9
Avoid Cyclic Behavior A-10

Control Chart Execution by Using Default Transitions A-12
Default Transition in Exclusive (OR) Decomposition A-12
Default Transition to a Junction A-12
Default Transition and a History Junction A-13
Labeled Default Transitions A-14

Control Chart Execution by Using Inner Transitions A-15
Before Using an Inner Transition A-15
After Using an Inner Transition to a Connective Junction A-15
Using an Inner Transition to a History Junction A-16

Process Events in States Containing Inner Transitions A-17
Process Events with an Inner Transition in an Exclusive (OR) State A-17
Process Events with an Inner Transition to a Connective Junction A-18

Inner Transition to a History Junction **A-20**

Represent Multiple Paths by Using Connective Junctions **A-22**
Label Format for Transition Segments **A-22**
If-Then-Else Decision Construct **A-22**
Self-Loop Transition **A-23**
For-Loop Construct **A-24**
Flow Chart Notation **A-24**
Transition from a Common Source to Multiple Destinations **A-25**
Resolve Equally Valid Transition Paths **A-26**
Transition from Multiple Sources to a Common Destination **A-27**
Transition from a Source to a Destination Based on a Common Event . . . **A-27**
Backtrack in Flow Charts **A-28**

Control Chart Execution by Using Event Actions in a Superstate **A-29**

Undirected Broadcast Events in Parallel States **A-30**
Broadcast Events in State Actions **A-30**
Broadcast Events in Transition Actions **A-31**
Broadcast Events in Condition Actions **A-33**

Broadcast Local Events in Parallel States **A-35**
Directed Event Broadcast Using Send **A-35**
Directed Event Broadcast Using Qualified Event Name **A-35**

32 Simulation Data Inspector

View Data in the Simulation Data Inspector **32-2**
View Logged Data **32-2**
Import Data from the Workspace or a File **32-3**
View Complex Data **32-5**
View String Data **32-6**
View Frame-Based Data **32-9**
View Event-Based Data **32-9**

Import Data from a CSV File into the Simulation Data Inspector **32-11**
Basic File Format **32-11**
Multiple Time Vectors **32-11**
Signal Metadata **32-12**
Import Data from a CSV File **32-13**

Microsoft Excel Import, Export, and Logging Format **32-15**
Basic File Format **32-15**
Multiple Time Vectors **32-15**
Signal Metadata **32-16**
User-Defined Data Types **32-18**
Complex, Multidimensional, and Bus Signals **32-20**
Function-Call Signals **32-21**
Simulation Parameters **32-21**
Multiple Runs **32-21**

Configure the Simulation Data Inspector **32-23**
Logged Data Size and Location **32-23**
Archive Behavior and Run Limit **32-24**
Incoming Run Names and Location **32-25**
Signal Metadata to Display **32-26**
Signal Selection on the Inspect Pane **32-27**
How Signals Are Aligned for Comparison **32-27**
Colors Used to Display Comparison Results **32-28**
Signal Grouping **32-28**
Data to Stream from Parallel Simulations **32-29**
Options for Saving and Loading Session Files **32-29**
Signal Display Units **32-29**

How the Simulation Data Inspector Compares Data **32-31**
Signal Alignment **32-31**
Synchronization **32-32**
Interpolation **32-33**
Tolerance Specification **32-33**
Limitations **32-35**

Save and Share Simulation Data Inspector Data and Views **32-36**
Save and Load Simulation Data Inspector Sessions **32-36**
Share Simulation Data Inspector Views **32-37**
Share Simulation Data Inspector Plots **32-37**
Create Simulation Data Inspector Report **32-38**
Export Data to the Workspace or a File **32-39**
Export Video Signal to an MP4 File **32-40**

Inspect and Compare Data Programmatically **32-42**
Create a Run and View the Data **32-42**
Compare Two Signals in the Same Run **32-43**
Compare Runs with Global Tolerance **32-44**
Analyze Simulation Data Using Signal Tolerances **32-45**

Limit the Size of Logged Data **32-48**
Limit the Number of Runs Retained in the Simulation Data Inspector Archive **32-48**
Specify a Minimum Disk Space Requirement or Maximum Size for Logged Data **32-48**
View Data Only During Simulation **32-49**
Reduce the Number of Data Points Logged from Simulation **32-49**

Stateflow Chart Programming

- “Model Finite State Machines by Using Stateflow Charts” on page 1-2
- “Overview of Stateflow Objects” on page 1-5
- “How Stateflow Objects Interact During Execution” on page 1-8
- “Specify Properties for Stateflow Charts” on page 1-19
- “Represent Operating Modes by Using States” on page 1-26
- “Use State Hierarchy to Design Multilevel State Complexity” on page 1-33
- “Define Exclusive and Parallel Modes by Using State Decomposition” on page 1-35
- “Transition Between Operating Modes” on page 1-37
- “Use Default Transitions to Specify Initial Substate Activity” on page 1-44
- “Move Between Levels of Hierarchy by Using Supertransitions” on page 1-48
- “Combine Transitions and Junctions to Create Branching Paths” on page 1-54
- “Resume Prior Substate Activity by Using History Junctions” on page 1-58
- “Create Entry and Exit Connections Across State Boundaries” on page 1-61
- “Guidelines for Naming Stateflow Objects” on page 1-66
- “Speed Up Simulation” on page 1-70

Model Finite State Machines by Using Stateflow Charts

A finite state machine is a representation of an event-driven, reactive system that transitions from one operating mode to another when 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 operating modes, such as park, reverse, neutral, drive, and low. As the driver moves the gearshift, the system transitions from one operating mode to another.

Types of Stateflow Blocks

To represent the relationships between the inputs, outputs, and operating modes of a finite state machine, you can add Stateflow blocks to a Simulink® model to create state transition diagrams, state transition tables, and truth tables:

- A Chart is a graphical representation of a finite state machine based on a state transition diagram. In a Stateflow chart, states and transitions form the basic building blocks of a sequential logic system. States correspond to operating modes and transitions represent pathways between states. For more information, see “Represent Operating Modes by Using States” on page 1-26 and “Transition Between Operating Modes” on page 1-37.
- A State Transition Table represents a finite state machine for sequential modal logic in tabular format. Instead of drawing states and transitions in a Stateflow chart, you can use a state transition table to model a state machine in a concise, compact format that requires minimal maintenance of graphical objects. For more information, see “Use State Transition Tables to Express Sequential Logic in Tabular Form” on page 16-2.
- A Truth Table implements combinatorial logic design in a tabular format. You can use truth table blocks to model decision making for fault detection and management and mode switching. For more information, see “Use Truth Tables to Model Combinatorial Logic” on page 8-2.

To implement control logic, Stateflow charts and State Transition Table blocks can use MATLAB® or C as the action language. Truth Table blocks use only MATLAB as the action language. For more information, see “Differences Between MATLAB and C as Action Language Syntax” on page 15-4.

Tip To combine the advantages of state machine programming with the full functionality of MATLAB, you can create a standalone Stateflow chart. You execute standalone charts as MATLAB objects directly through the Command Window or by using a script. You can also program a MATLAB app that controls the state of the chart through a graphical user interface. For more information, see “Create Stateflow Charts for Execution as MATLAB Objects” on page 31-2.

Program a Stateflow Chart

To create a Stateflow chart that models a finite state machine:

1. Create a Simulink model that contains an empty Stateflow chart by calling the function `sfnew`.

   ```
   sfnew
   ```
2. To open the Stateflow Editor, double-click the chart block. For more information on using the Stateflow Editor, see “Stateflow Editor Operations” on page 25-2.
3. For each operating mode in your system, draw a state and implement the state actions by adding state labels, as described in “Represent Operating Modes by Using States” on page 1-26.

- To organize complex systems, define a hierarchy of states by drawing child states inside a parent state. For example, you can use a superstate to enclose substates that share the same state actions. For more information, see “Use State Hierarchy to Design Multilevel State Complexity” on page 1-33.
- To model operating modes that are active at the same time, enable parallel (AND) decomposition in a parent state. For more information, see “Define Exclusive and Parallel Modes by Using State Decomposition” on page 1-35.

4. To represent the direction of flow logic between states, draw transitions and implement the transition conditions by adding transition labels, as described in “Transition Between Operating Modes” on page 1-37.

   - To mark the first state to become active, use a default transition. For more information, see “Use Default Transitions to Specify Initial Substate Activity” on page 1-44.
   - To create paths from a single source to multiple destinations or from multiple sources to a single destination, combine transitions and connective junctions. For more information, see “Combine Transitions and Junctions to Create Branching Paths” on page 1-54.

5. If your system has inputs or outputs, or depends on any state variables, add input, output, and local data, as described in “Add Stateflow Data” on page 10-2.
6. If your system reacts to event triggers or must trigger actions in your chart or other blocks in your model, add input, output, or local events, as described in “Synchronize Model Components by Broadcasting Events” on page 12-2.
7. If your chart has complex state actions or transition conditions, add reusable functions to your chart. Use the function format that is most natural for the type of calculation in the state action or transition condition by selecting from these functions:

   - Graphical functions — Encapsulate flow charts that contain logic and iterative loop patterns. See “Reuse Logic Patterns by Defining Graphical Functions” on page 6-9.
   - MATLAB functions — Write matrix-oriented algorithms for data analysis and visualization. See “Reuse MATLAB Code by Defining MATLAB Functions” on page 7-2.
   - Simulink functions — Streamline your design by calling Simulink function-call subsystems. See “Reuse Simulink Functions in Stateflow Charts” on page 9-2.
   - Truth tables — Represent combinational logic for decision-making applications. See “Use Truth Tables to Model Combinatorial Logic” on page 8-2.

   Alternatively, you can write your own C or C++ code for integration with your chart. For more information, see “Reuse Custom Code in Stateflow Charts” on page 28-2.
8. Connect the chart to other blocks in the Simulink model by using input and output ports.
9. To simulate the model, click **Run** . During the simulation, the Stateflow Editor highlights active states and transitions through chart animation.

For a tutorial that illustrates this workflow, see “Construct and Run a Stateflow Chart”.

References

[1] Harel, David. "Statecharts: A Visual Formalism for Complex Systems." *Science of Computer Programming* 8, no.3 (June 1987): 231-74.

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

See Also

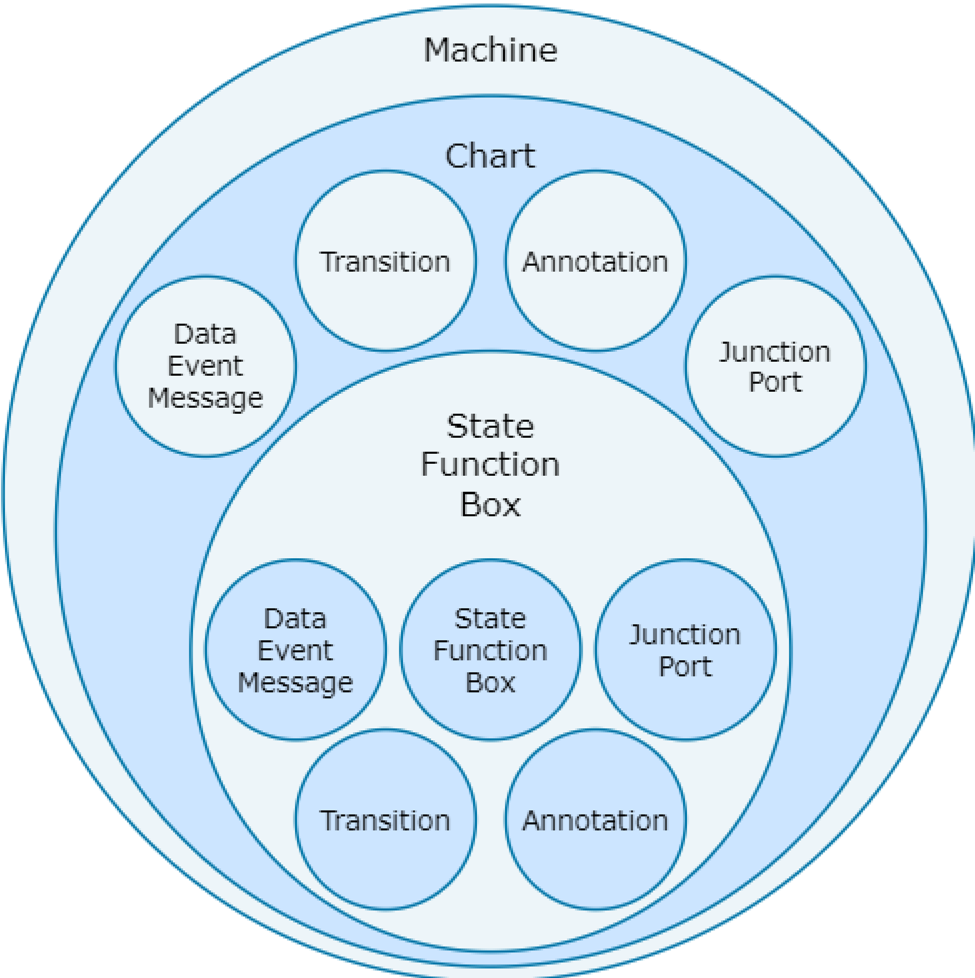
`sfnew` | Chart | State Transition Table | Truth Table

More About

- "Overview of Stateflow Objects" on page 1-5
- "How Stateflow Objects Interact During Execution" on page 1-8
- "Specify Properties for Stateflow Charts" on page 1-19
- "Construct and Run a Stateflow Chart"

Overview of Stateflow Objects

Stateflow objects are arranged 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. The Stateflow machine contains all of the Stateflow charts in a Simulink model.

Stateflow charts can contain states, functions, boxes, data, events, messages, transitions, junctions, entry and exit ports, and annotations. States, functions, and boxes can contain other states, functions, boxes, data, events, messages, transitions, junctions, entry and exit ports, and annotations. Levels of nesting can continue indefinitely.

Graphical Objects

To manage graphical objects, use the Stateflow Editor. This table lists each type of graphical object and the palette icon to use for adding the object. For more information, see “Stateflow Editor Operations” on page 25-2.

Type of Graphical Object	Palette Icon	Reference
State		“Represent Operating Modes by Using States” on page 1-26
Transition		“Transition Between Operating Modes” on page 1-37
Connective junction		“Combine Transitions and Junctions to Create Branching Paths” on page 1-54
Box		“Group Chart Objects by Using Boxes” on page 6-2
Simulink based state		“Create and Edit Simulink Based States” on page 4-8
Simulink function		“Reuse Simulink Functions in Stateflow Charts” on page 9-2
Graphical function		“Reuse Logic Patterns by Defining Graphical Functions” on page 6-9
MATLAB function		“Reuse MATLAB Code by Defining MATLAB Functions” on page 7-2
Truth table function		“Use Truth Tables to Model Combinatorial Logic” on page 8-2
History junction		“Resume Prior Substate Activity by Using History Junctions” on page 1-58
Exit junction		“Create Entry and Exit Connections Across State Boundaries” on page 1-61
Entry junction		“Create Entry and Exit Connections Across State Boundaries” on page 1-61
Annotation		“Add Descriptive Comments in a Chart” on page 6-22
Image		“Add Descriptive Comments in a Chart” on page 6-22

Nongraphical Objects

You can define data, event, and message objects that do not appear graphically in the Stateflow Editor. To manage nongraphical objects, use the **Symbols** pane or Model Explorer. For more information, see:

- “Manage Symbols in the Stateflow Editor” on page 25-14
- “Use the Model Explorer with Stateflow Objects” on page 25-22

Data Objects

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

Event Objects

An event is a Stateflow object that can trigger a whole Stateflow chart or individual actions in a chart. Because Stateflow charts execute by reacting to events, you specify and program events into your charts to control their execution. You can broadcast events to every object in the scope of the object sending the event, or you can send an event to a specific object. You can define explicit events that you specify directly, or you can define implicit events to take place when certain actions are performed, such as entering a state. For more information, see “Synchronize Model Components by Broadcasting Events” on page 12-2.

Message Objects

Stateflow message objects are queued objects that can carry data. You can send a message from one Stateflow chart to another to communicate between charts. You can also send local messages within a chart. You define the type of message data. You can view the lifeline of a message in the Sequence Viewer block. For more information, see “Communicate with Stateflow Charts by Sending Messages” on page 13-2.

How Stateflow Objects Interact During Execution

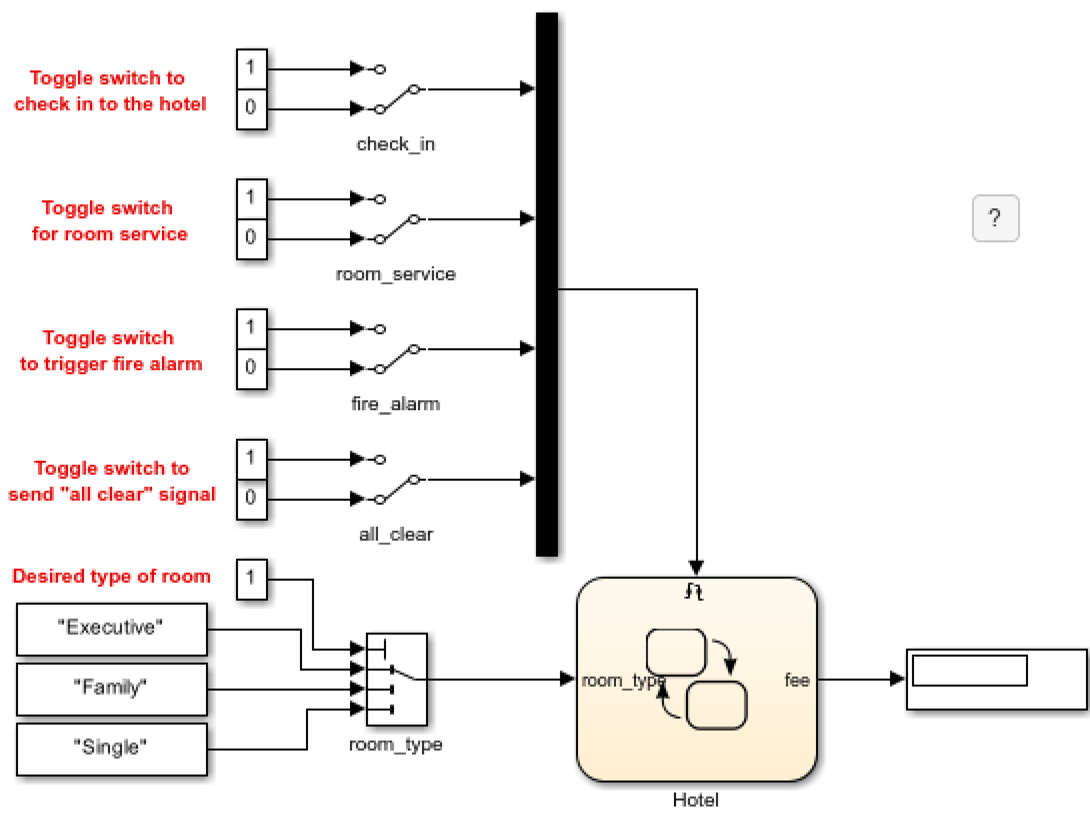
During execution, Stateflow® objects interact with each other to simulate real-world behavior. This example uses the hotel check-in process to explain how common graphical and nongraphical objects in a Stateflow chart interact during execution.

Model of the Check-In Process for a Hotel

This model contains a Stateflow chart called `Hotel`. The chart receives input events from four Manual Switch (Simulink) blocks that you toggle to:

- Check in to the hotel
- Call room service
- Set off a fire alarm
- Send an all-clear signal after a fire alarm

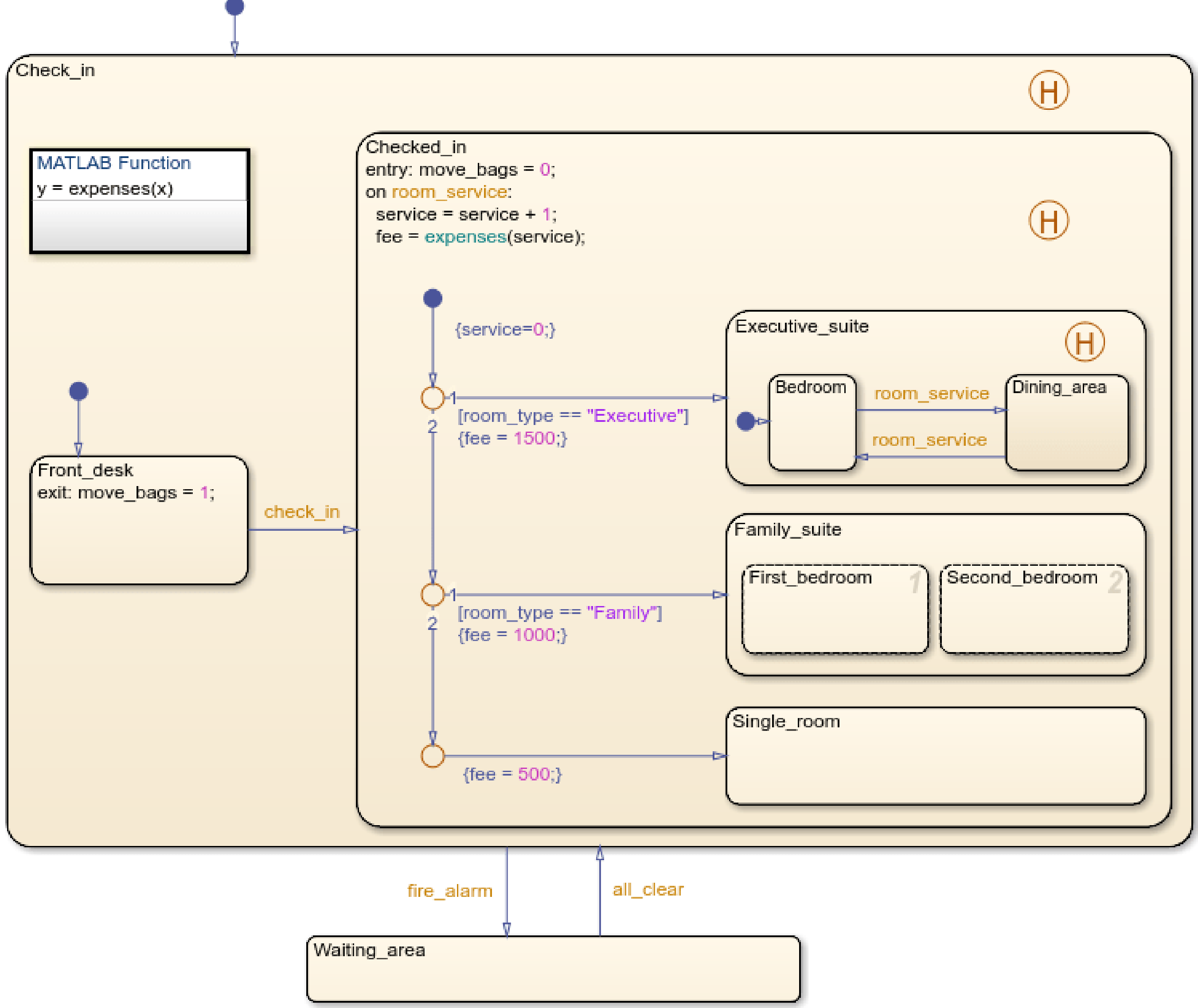
A Mux (Simulink) block combines these input events into a vector of inputs that connects to the trigger port on the top side of the chart.



The chart also receives an input signal called `room_type` from a Multiport Switch (Simulink) block. The value for this signal corresponds to the type of room you want to stay in. The possible options are `"Executive"` for an executive suite, `"Family"` for a family suite, and `"Single"` for a single room.

During simulation, the total amount due, including charges for room service, appears in the Display (Simulink) block.

The `Hotel` chart contains graphical objects, such as states and history junctions, and nongraphical objects, such as data and events. To see an image that labels the objects in this chart, see “Stateflow Objects” on page 2-2.



When you start the simulation, the chart does not wake up until it detects a rising or falling edge in one of its input events.

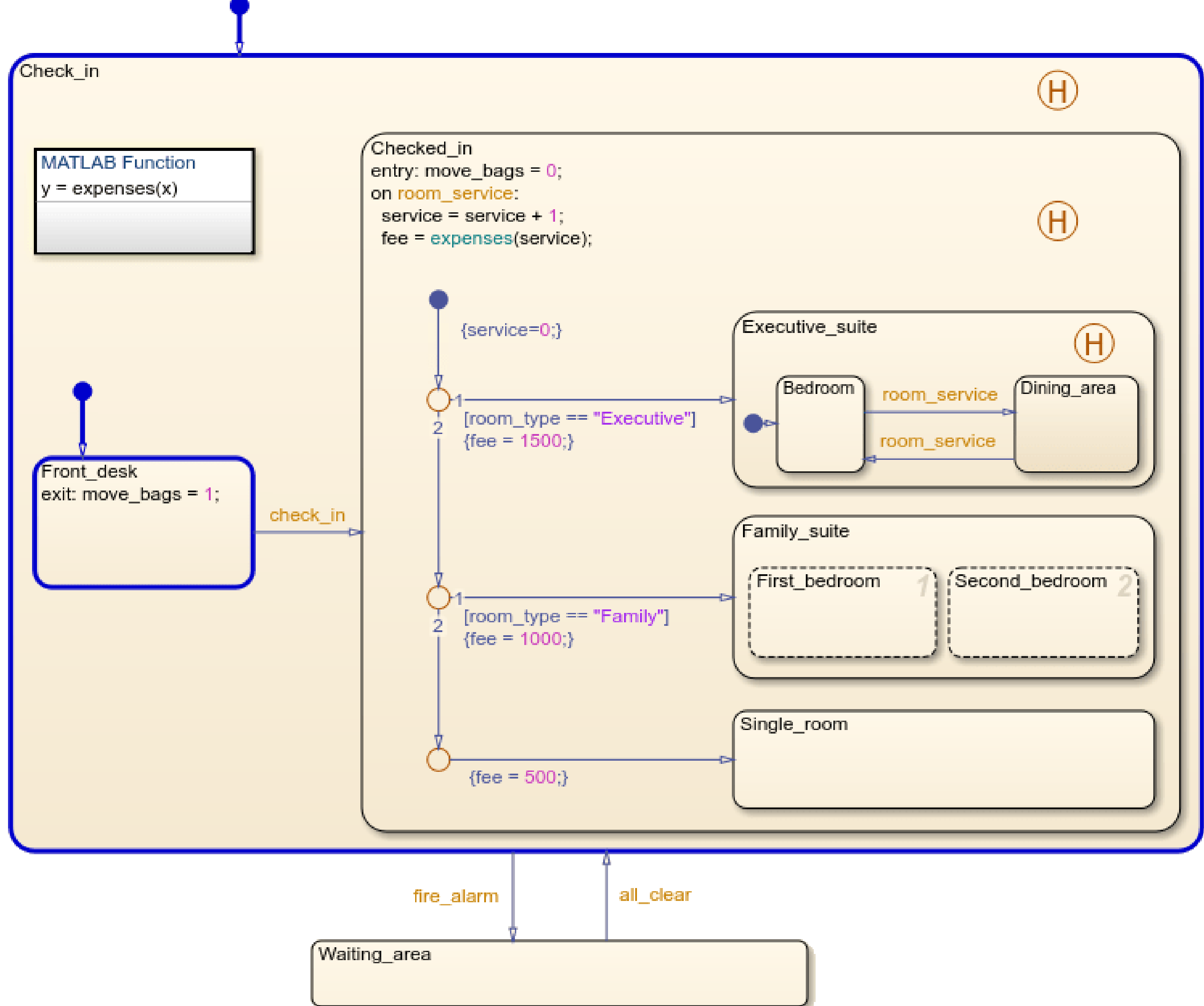
When you toggle a Manual Switch block, you trigger an input event that wakes up the chart. While the chart is awake, it reads a value for the chart input `room_type` from the Multiport Switch block, performs any valid state or transition actions, and outputs the new value of `fee` to the Display block.

After completing all possible phases of execution, the chart goes back to sleep and waits for the next input event.

Chart Initialization

Start the simulation and trigger one of the input events. This action corresponds to entering the hotel and stopping at the front desk.

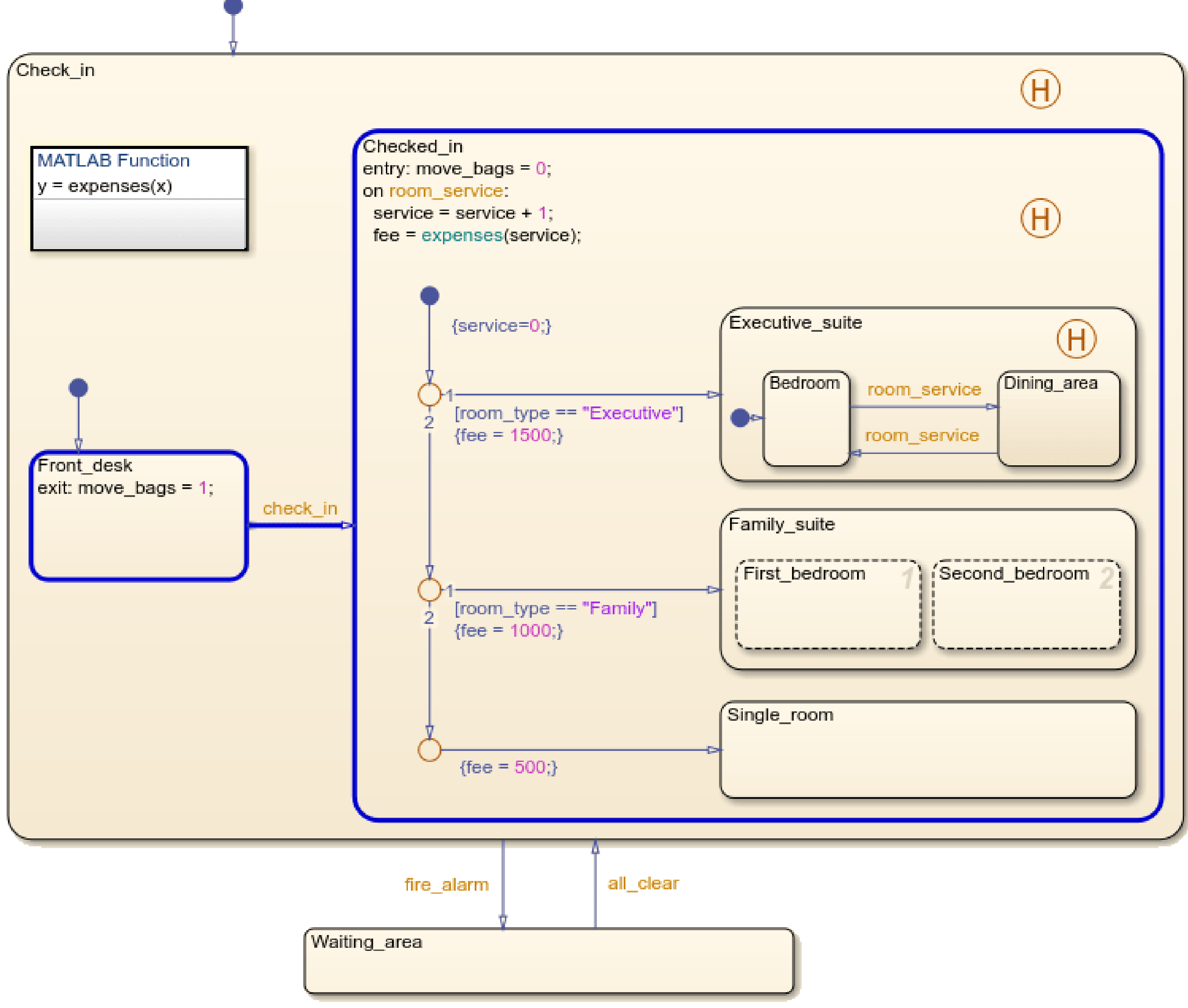
Because the chart property **Execute (enter) chart at initialization** is disabled, the chart remains asleep until it detects a rising or falling edge in one of its input events. Then the chart wakes up and executes its default transitions. The default transition to the state `Check_in` occurs, making that state active. Then, the default transition to the substate `Front_desk` occurs, making that state active. Then the chart goes to sleep. For more information, see "Execution of a Chart at Initialization" on page 2-10 and "Enter a Chart or State" on page 2-17.



Transition Between States

While the substate `Front_desk` is active, trigger the input event `check_in`. This action corresponds to checking in to the hotel. You pick up your bags, move from the front desk to your room, and put your bags down.

In the chart, the `check_in` event guards the outgoing transition from the substate `Front_desk` to the substate `Checked_in`. When you trigger the event, the transition becomes valid. The exit action of `Front_desk` sets the value of the local data object `move_bags` to 1 and the substate becomes inactive. Then, `Checked_in` becomes active and the entry action sets `move_bags` to 0. For more information, see “How Stateflow Charts Respond to Events” on page 2-40, “Exit a State” on page 2-23, and “Enter a Chart or State” on page 2-17.



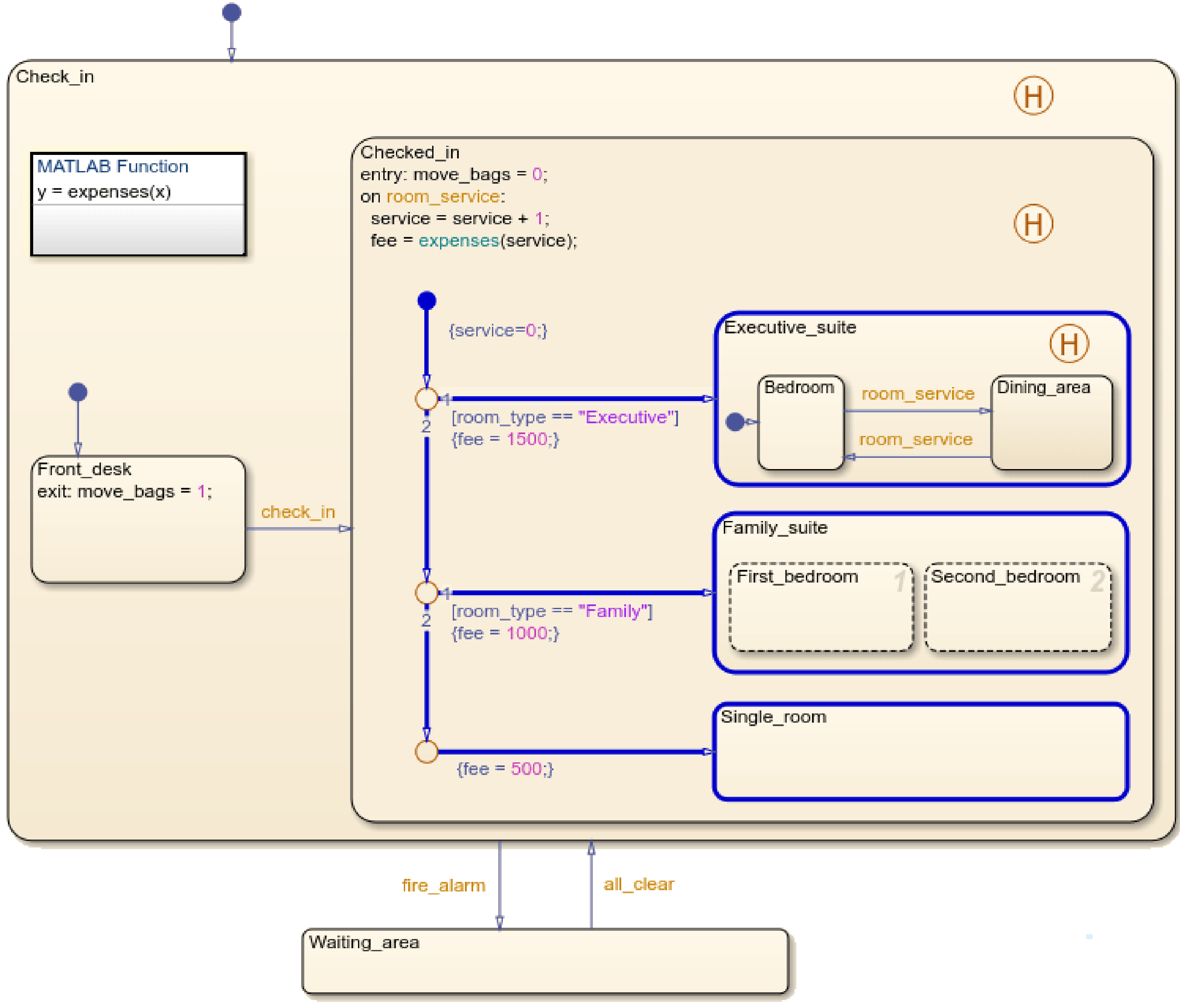
Evaluation of Default Transition Paths

After the chart executes the `entry` actions in the `Checked_in` state, it evaluates the default transition path to one of the substates. The substate that becomes active corresponds to the room type. If you choose an executive suite, the base fee is $1500. If you choose a family suite, the base fee is $1000. If you choose a single room, the base fee is $500.

The chart tests the branches of the default transition path in this order:

- If the chart input `room_type` equals `"Executive"`, the top transition is valid. The condition action sets the chart output `fee` to 1500 and the substate `Executive_suite` becomes active.
- If the chart input `room_type` equals `"Family"`, the middle transition is valid. The condition action sets the fee to 1000 and the substate `Family_suite` becomes active.
- Otherwise, the chart input `room_type` equals `"Single"` and the bottom transition is valid. The condition action sets the fee to 500 and the substate `Single_room` becomes active.

For more information, see “Order of Execution for a Set of Flow Charts” on page 2-44.

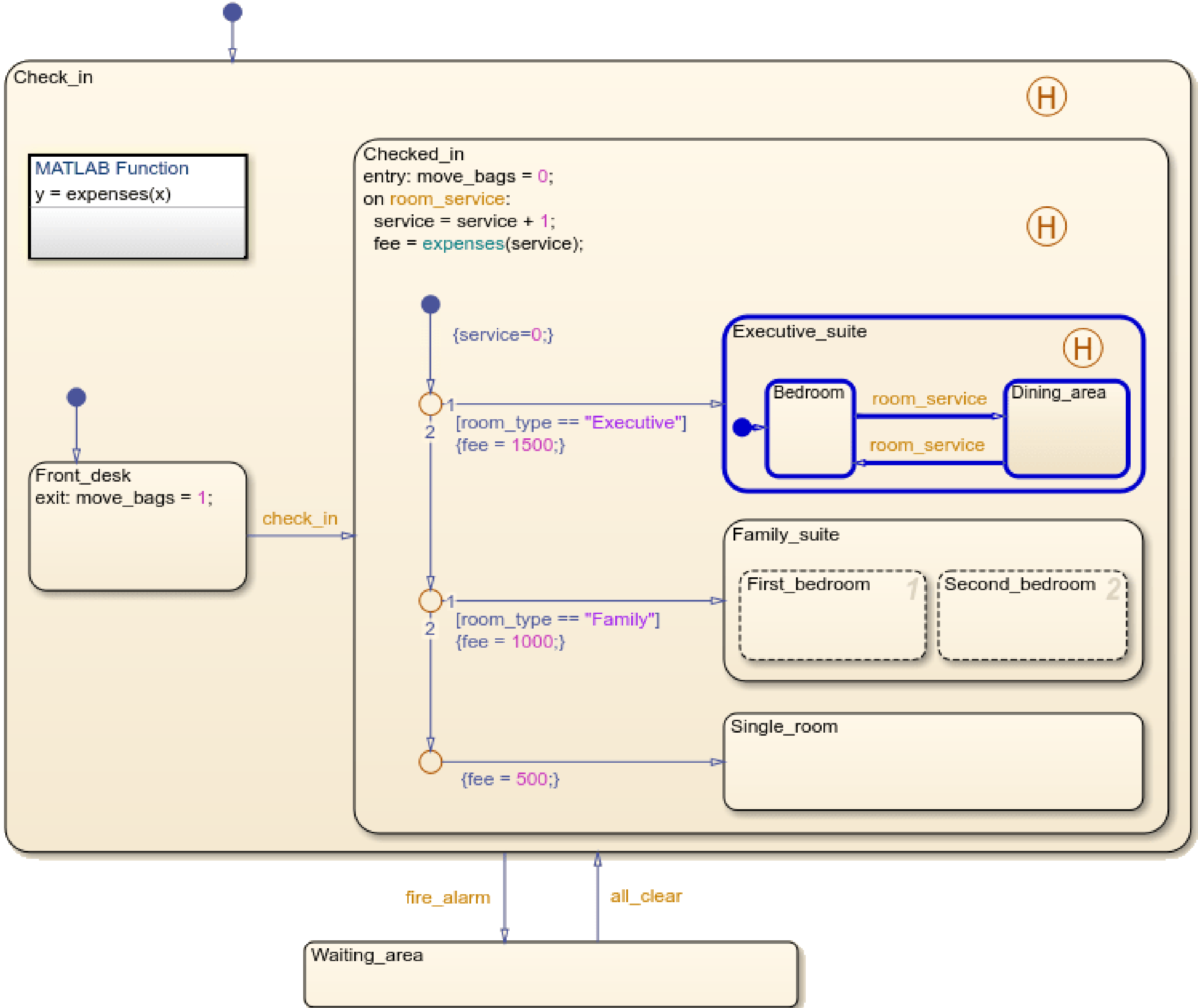


Execution of States with Exclusive Substates

If you trigger the input event `check_in` while the value of the chart input `room_type` is `"Executive"`, the substate `Executive_suite` becomes active. This substate corresponds to staying in the executive suite. This suite has separate bedroom and dining areas, so you can be in only one area of the suite at any time. When you reach the executive suite, you enter the bedroom first. When you order room service, you enter the dining area to eat. When you want the food removed from the dining area, you order room service again and then return to the bedroom.

The state `Executive_suite` has exclusive (OR) decomposition. The state has two substates, `Bedroom` and `Dining_area`. When `Executive_suite` first becomes active, the default transition to `Bedroom` occurs, making that substate active. A broadcast of the input event `room_service` triggers the transition from `Bedroom` to `Dining_area`, making `Bedroom` inactive and `Dining_area` active. A subsequent broadcast of `room_service` triggers the transition back from `Dining_area` to

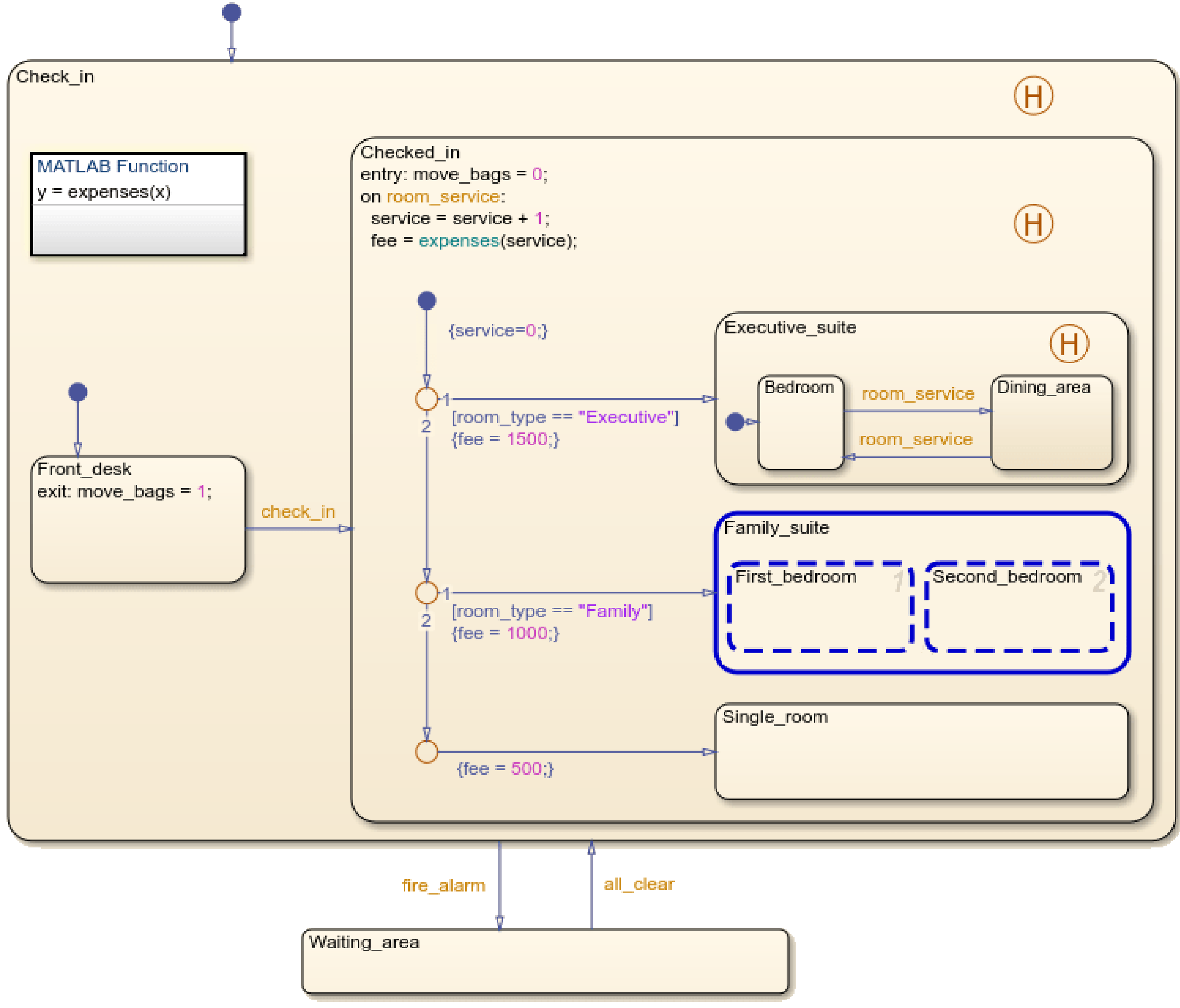
Bedroom, making Bedroom active and Dining_area inactive. For more information, see “Enter a Chart or State” on page 2-17.



Execution of State with Parallel Substates

If you trigger the input event check_in while the value of the chart input room_type is "Family", the substate Family_suite becomes active. This substate corresponds to staying in the family suite. When your family reaches the suite, family members can spend time in two bedrooms. For example, the parents can watch a movie in the first bedroom while the children sleep in the second bedroom.

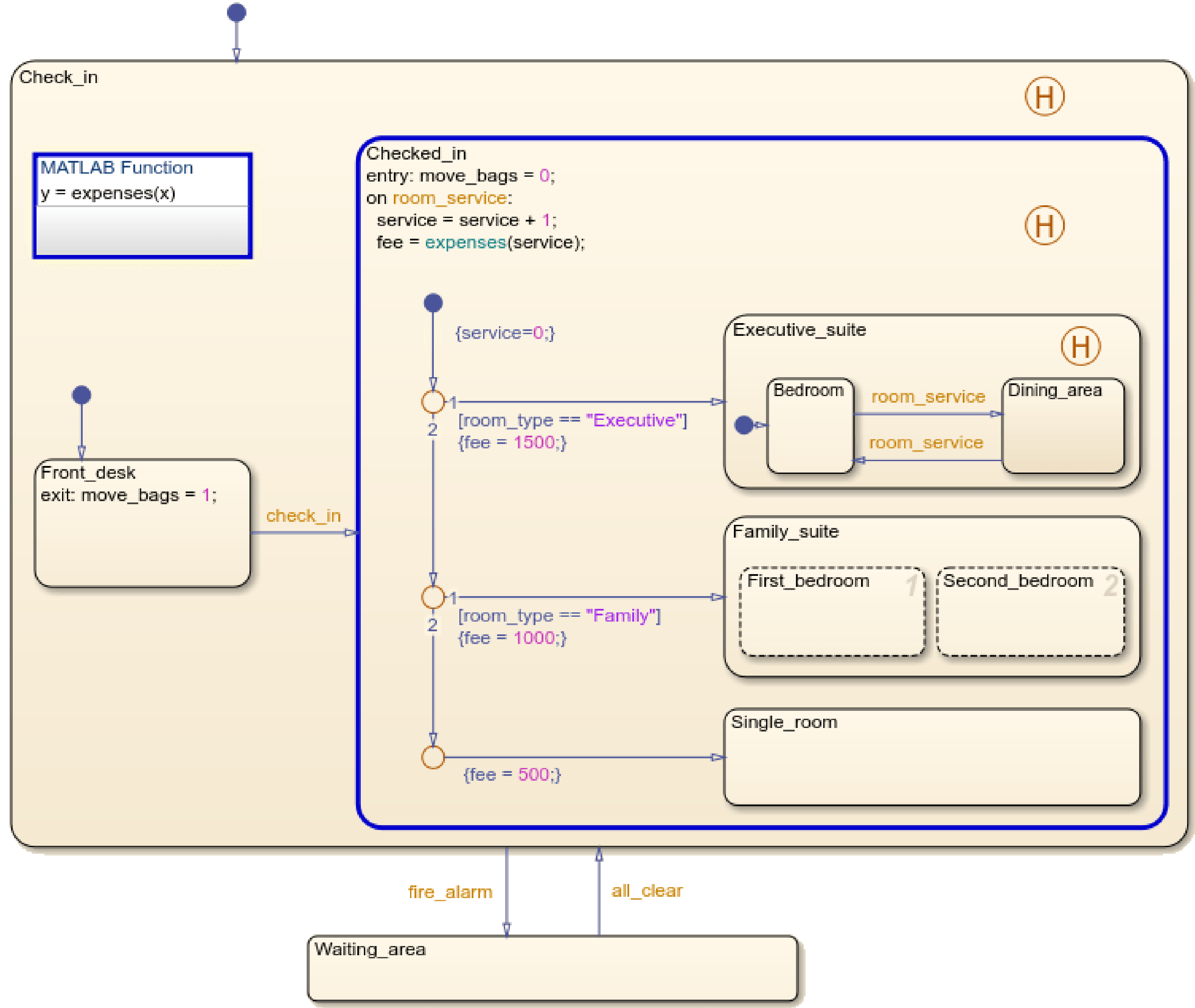
The state Family_suite has parallel (AND) decomposition. The state has two substates, First_bedroom and Second_bedroom. When Family_suite becomes active, the parallel states wake up according to their execution order, as indicated by the number in the upper right corner of each state. The substates remain active at the same time. For more information, see “Execution Order for Parallel States” on page 2-46 and “Enter a Chart or State” on page 2-17.



Function Call from a State Action

While the substate `Checked_in` is active, trigger the input event `room_service`. This action corresponds to calling for room service. Your hotel bill depends on your room type and the number of room service requests you make.

When the chart detects a rising or falling edge in the input event `room_service`, the `Checked_In` state executes the `on` action for this event. The state increments the local data object `service` and calls the MATLAB® function `expenses`. This function takes the total number of room service requests as an input and returns the current hotel bill as an output. For more information, see “Control Chart Execution by Using Event Actions in a Superstate” on page A-29.



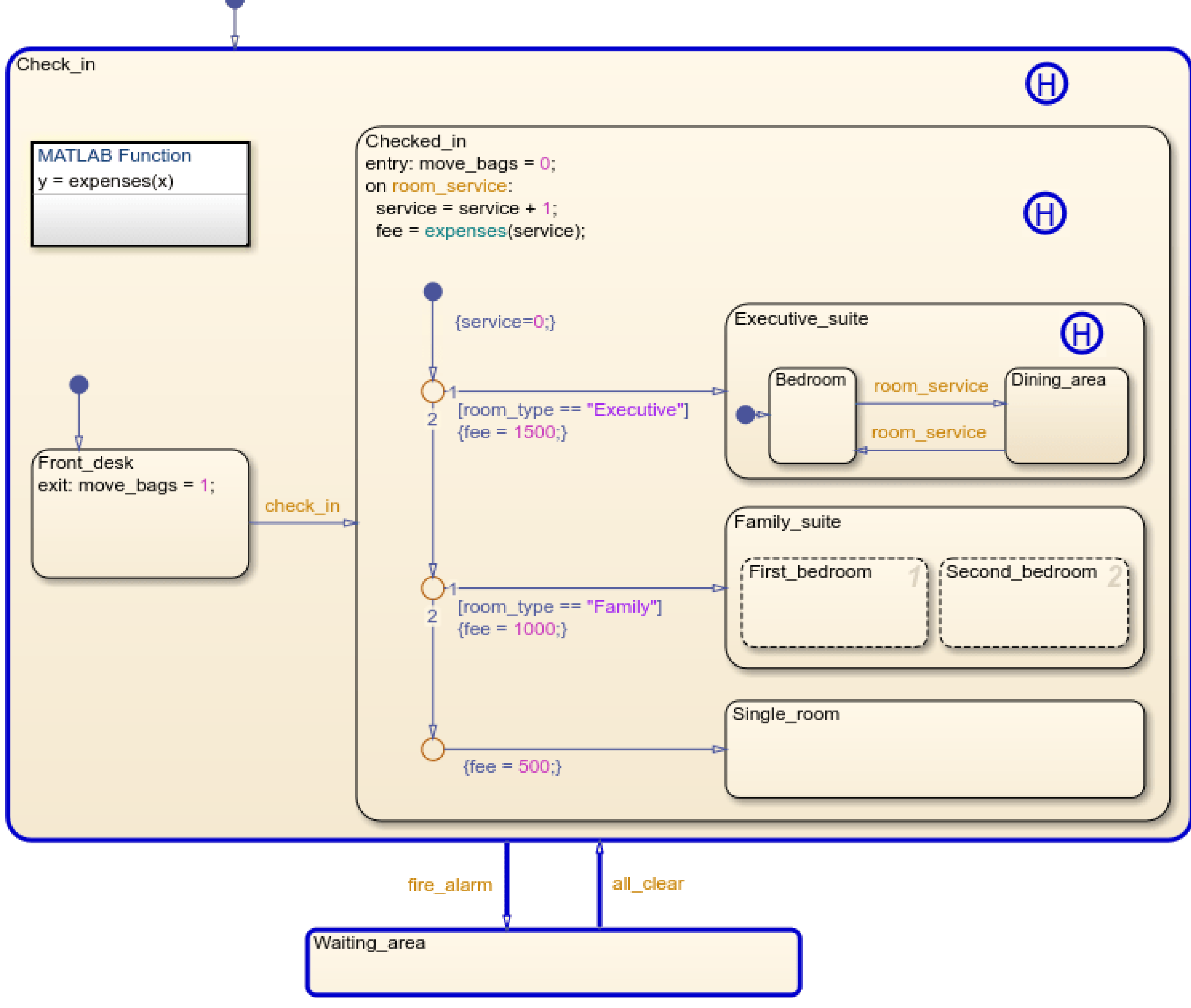
Execution of States with History Junctions

While the substate `Checked_in` is active, trigger the input event `fire_alarm`, which corresponds to setting off a fire alarm. You leave the building and wait outside in the designated waiting area. Then trigger the input event `all_clear`, which corresponds to sending an all-clear signal that allows you to return to your previous location inside the hotel.

When the chart receives an event broadcast for `fire_alarm`, the transition from `Check_in` to `Waiting_area` occurs. The history junctions in `Check_in`, `Checked_in`, and `Executive_suite` record the last active substates in each of these states. The active states become inactive in ascending order of hierarchy, starting with the innermost substates. After `Check_in` becomes inactive, the `Waiting_area` becomes active.

When the chart receives an event broadcast for `all_clear`, the transition from `Waiting_area` to the previously active substate of `Check_in` occurs. `Waiting_area` becomes inactive before the substates of `Check_in` become active in descending order of hierarchy, starting with `Check_in`.

For more information, see "How Stateflow Charts Respond to Events" on page 2-40, "Exit a State" on page 2-23, and "Enter a Chart or State" on page 2-17.



See Also

Manual Switch (Simulink) | Mux (Simulink) | Multiport Switch (Simulink) | Display (Simulink)

Related Examples

- "Stateflow Semantics" on page 2-2

- “How Stateflow Charts Respond to Events” on page 2-40
- “Execution of a Chart at Initialization” on page 2-10
- “Enter a Chart or State” on page 2-17
- “Exit a State” on page 2-23
- “Order of Execution for a Set of Flow Charts” on page 2-44
- “Execution Order for Parallel States” on page 2-46
- “Control Chart Execution by Using Event Actions in a Superstate” on page A-29

Specify Properties for Stateflow Charts

Chart properties specify how your Stateflow chart interfaces with the Simulink model. You can modify these properties in the **Property Inspector**, the Model Explorer, or the Chart properties dialog box.

To use the **Property Inspector**:

1. In the **Modeling** tab, under **Design Data**, select **Property Inspector**.
2. In the Stateflow Editor, click the chart.
3. In the **Property Inspector**, edit the chart properties.

To use the Model Explorer:

1. In the **Modeling** tab, under **Design Data**, select **Model Explorer**.
2. In the **Model Hierarchy** pane, select the chart.
3. In the **Dialog** pane, edit the chart properties.

To use the Chart properties dialog box:

1. In the Stateflow Editor, right-click the chart.
2. Select **Properties**.
3. In the properties dialog box, edit the chart properties.

You can also modify chart properties programmatically by using `Stateflow.Chart` objects. For more information about the Stateflow programmatic interface, see “Overview of the Stateflow API”.

Stateflow Chart Properties

You can set the following chart properties in:

- The **Properties** tab of the **Property Inspector**
- The **General** tab of the Model Explorer or the Chart properties dialog box

Name

Name of the chart. This property is read-only. When you click the chart name hyperlink in the Model Explorer and the Chart properties dialog box, the chart opens in the Stateflow Editor.

Machine

Name of the Simulink subsystem. This property is read-only and is not available in the **Property Inspector**. When you click the machine name hyperlink, the Machine properties dialog box opens.

Action Language

Action language that defines the syntax for state and transition actions in the chart. Options include:

- `MATLAB`
- `C`

The default value is MATLAB. For more information, see “Differences Between MATLAB and C as Action Language Syntax” on page 15-4.

State Machine Type

Type of state machine semantics to implement. Options include:

- Classic
- Mealy
- Moore

Classic charts provide the full set of Stateflow semantics. Mealy and Moore charts use a subset of these semantics. The default value is Classic. For more information, see “Overview of Mealy and Moore Machines” on page 5-2.

Update Method

Method by which a simulation updates or wakes up a chart in a Simulink model.

Setting	Description
Inherited	Input from the Simulink model determines when the chart wakes up during a simulation (default). If you define input events for the chart, the Stateflow chart is explicitly triggered by a signal on its trigger port originating from a connected Simulink block. You can set this trigger input event to occur in response to a Simulink signal. The Simulink signal can be Rising, Falling, or Either (rising and falling), or in response to a Function Call. For more information, see “Activate a Stateflow Chart by Sending Input Events” on page 12-8. If you do not define input events, the Stateflow chart implicitly inherits triggers from the Stateflow model. These implicit events are the discrete or continuous sample times of the Stateflow signals providing inputs to the chart. If you define data inputs, the chart awakens at the rate of the fastest data input. If you do not define any data input for the chart, the chart wakes up as defined by the execution behavior of its parent subsystem.
Discrete	The Simulink model generates an implicit event at regular time intervals to awaken the Stateflow chart at the rate that you specify in the **Sample Time** chart property. Other blocks in the Simulink model can have different sample times.
Continuous	The Stateflow chart updates its state during major time steps only, although it computes outputs and local continuous variables during major and minor time steps. The chart can register zero crossings, which allows Simulink models to sample Stateflow charts whenever state changes occur. The Stateflow chart computes derivatives for local continuous variables. For more information, see “Continuous-Time Modeling in Stateflow” on page 22-2.

Sample Time

The time interval at which the Stateflow chart wakes up during simulation. The sample time can be any nonzero number. The sample time is in the same units as the Simulink simulation time. Other

blocks in the Simulink model can have different sample times. This option is available only when you set the chart property **Update method** to `Discrete`.

Enable zero-crossing detection

Specifies that zero-crossing detection is enabled (default). This option is available only when you set the chart property **Update method** to `Continuous`. See “Disable Zero-Crossing Detection” on page 22-3.

Enable C-bit operations

Specifies that the operators &, `^`, `|`, and ~ perform bitwise operations in action statements (default). If you clear this check box:

- `&`, `|`, and ~ perform logical operations.
- `^` performs the power operation.

This option is available only in charts that use C as the action language. For more information, see “Operations for Stateflow Data” on page 14-4.

User-specified state/transition execution order

Specifies that the chart uses explicit ordering of parallel states and transitions (default). You determine the order in which the chart executes parallel states and tests transitions originating from a source. This option is available only in charts that use C as the action language. For more information, see “Execution Order for Parallel States” on page 2-46 and “Evaluate Transitions” on page 2-26.

Export chart level functions

Extends the scope of functions defined at the root level of the chart to other parts of the model. This option enables Simulink Caller blocks to call Stateflow functions in the local hierarchy by using qualified notation *chartName.functionName*. For more information, see “Export Stateflow Functions for Reuse” on page 6-14.

Treat exported functions as globally visible

Enables Stateflow and Simulink Caller blocks throughout the model to call functions exported from Stateflow without using qualified notation. This option is available only when you select the chart property **Export chart level functions**. For more information, see “Export Stateflow Functions for Reuse” on page 6-14.

Execute (enter) chart at initialization

Specifies that the chart initializes its state configuration at time 0 instead of at the first occurrence of an input event. For more information, see “Execution of a Chart at Initialization” on page 2-10.

Initialize outputs every time chart wakes up

Specifies that the chart resets its output values every time that the chart wakes up, not only at time 0. Output values are reset whenever a chart is triggered by function call, edge trigger, or clock tick. If you set an initial value for an output data object, the output resets to that value. Otherwise, the output resets to zero. Select this option to:

- Ensure that all outputs are defined in every chart execution.
- Prevent latching of outputs (carrying over values of outputs computed in previous executions).
- Provide all chart outputs with a meaningful initial value.

For more information, see “Initial value” on page 10-8.

Enable super step semantics

Specifies that the chart can take multiple transitions in each time step until it reaches a stable state. This option is not available when you set the chart property **Update method** to `Continuous`. For more information, see “Super Step Semantics” on page 2-35.

Maximum iterations in each super step

Specifies the maximum number of transitions that the chart can take in each time step. The chart always takes one transition during a super step, so the value *N* that you specify represents the maximum number of *additional* transitions (for a total of *N+1*). This option is available only when you select the chart property **Enable super step semantics**. For more information, see “Maximum Number of Iterations” on page 2-35.

Behavior after too many iterations

Specifies how the chart behaves after it reaches the maximum number of transitions in a time step.

Behavior	Description
`Proceed`	Chart execution continues to the next time step.
`Throw Error`	Simulation stops and an error message appears. This setting is valid only for simulation. In generated code, chart execution always proceeds to the next time step rather than generating an error.

This option is available only when you select the chart property **Enable super step semantics**.

Support variable-size arrays

Specifies that chart supports data that vary in size during simulation. See “Declare Variable-Size Data in Stateflow Charts” on page 19-9.

Treat dimensions of length 1 as fixed size

Specifies if output data with at least one dimension of length 1 are fixed size. When this property is enabled, the chart sets data that are variable size in the chart with a dimension of 1 to fixed size. When this property is disabled, data in the chart that has the **Variable size** property enabled are always variable size. Prior to R2023a, the chart treats data with at least one dimension of length 1 as fixed size.

This property only affects output data that have the **Variable size** property enabled. See “Declare Variable-Size Data in Stateflow Charts” on page 19-9.

Saturate on integer overflow

Specifies that integer overflows saturate in the generated code. See “Handle Integer Overflow for Chart Data” on page 10-37.

Variant activation time

Specifies active choice of the variant blocks or variant parameters at different stages of the simulation and code generation workflow. Based on the stage you specify using this property, Stateflow determines if the generated code must contain only the active choice or both active and inactive choices.

- `update diagram analyze all choices`– with this option enabled, Stateflow analyzes both active and inactive choices for incompatibilities in signal attributes, however it generates code only for the active choice.
- `code compile`– with this option enabled, Stateflow analyzes both active and inactive choices of variant transitions and generates code for both active and inactive choices. The choices are enclosed in C preprocessor conditional statements #if and #endif that are conditionally compiled when you compile the generated code.

See “Control Indicator Lamp Dimmer Using Variant Conditions” on page 29-9.

States when enabling

Specifies how states behave when function-call input events reenable the chart. Options include:

- `Held`
- `Reset`

See “Control States in Charts Enabled by Function-Call Input Events” on page 12-12.

Create output for monitoring

Specifies that the chart produces active state output. When you enable this option, you can select one of these activity types to output:

- `Child activity`
- `Leaf state activity`

See “Monitor State Activity Through Active State Data” on page 11-2.

Fixed-Point Properties

You can set fixed-point properties for the chart in:

- The **Properties** tab of the **Property Inspector**
- The **Fixed Point Properties** tab of the Model Explorer or the Chart properties dialog box

Fixed-point properties are available only in charts that use MATLAB as the action language.

Treat These Inherited Simulink Signal Types as fi Objects

Specifies whether the chart treats inherited fixed-point and integer signals as Fixed-Point Designer™ `fi` objects.

Setting	Description
`Fixed-point`	The chart treats all fixed-point inputs as `fi` objects (default).

Setting	Description
`Fixed-point & Integer`	The chart treats all fixed-point and integer inputs as `fi` objects.

MATLAB Chart fimath

Specifies default properties for the chart.

Setting	Description
`Same as MATLAB`	Use the same `fimath` properties as the current default `fimath` object in MATLAB.
`Specify Other`	Use your own default `fimath` object. You can: • Construct a `fimath` object inside the edit box. • Create a `fimath` object in the MATLAB or model workspace and enter its variable name in the edit box.

For more information, see “fimath Properties Usage for Fixed-Point Arithmetic” (Fixed-Point Designer).

Additional Properties

You can set additional properties for the chart in:

- The **Info** tab of the **Property Inspector**
- The **Documentation** tab of the Model Explorer or the Chart properties dialog box

Description

Description of the chart.

Document Link

Link to online documentation for the chart. You can enter a web URL address or a MATLAB command that displays documentation as an HTML file or as text in the MATLAB Command Window. When you click the **Document link** hyperlink, Stateflow evaluates the link and displays the documentation.

Machine Properties

The Stateflow machine represents all of the Stateflow blocks in a model (including all charts, state transition tables, and truth tables). You can modify the properties listed below in the Machine properties dialog box.

1. Open the Model Explorer or the Chart properties dialog box for any chart in the model.
2. In the **Machine** chart property field, click the machine name link.
3. In the Machine properties dialog box, edit the properties for the Stateflow machine.

You can also modify machine properties programmatically by using `Stateflow.Machine` objects. For more information about the Stateflow programmatic interface, see “Overview of the Stateflow API”.

Simulink Model

Name of the Simulink model that defines this Stateflow machine. This property is read-only. You change the model name when you save the model.

Creation Date

Date on which this Stateflow machine was created. This property is read-only.

Creator

Name of the person who created this Stateflow machine.

Modified

Comment text for recording modifications to the Simulink model that defines this Stateflow machine.

Version

Comment text for recording the version of the Simulink model that defines this Stateflow machine.

Description

Description of the Stateflow machine.

Document Link

Link to online documentation for the Stateflow machine. You can enter a web URL address or a MATLAB command that displays documentation as an HTML file or as text in the MATLAB Command Window. When you click the **Document link** hyperlink, Stateflow evaluates the link and displays the documentation.

See Also

Blocks
Chart

Objects
`Stateflow.Chart` | `Stateflow.Machine`

Tools
Model Explorer

More About

- “Differences Between MATLAB and C as Action Language Syntax” on page 15-4
- “Overview of Mealy and Moore Machines” on page 5-2
- “Continuous-Time Modeling in Stateflow” on page 22-2
- “Operations for Stateflow Data” on page 14-4

Represent Operating Modes by Using States

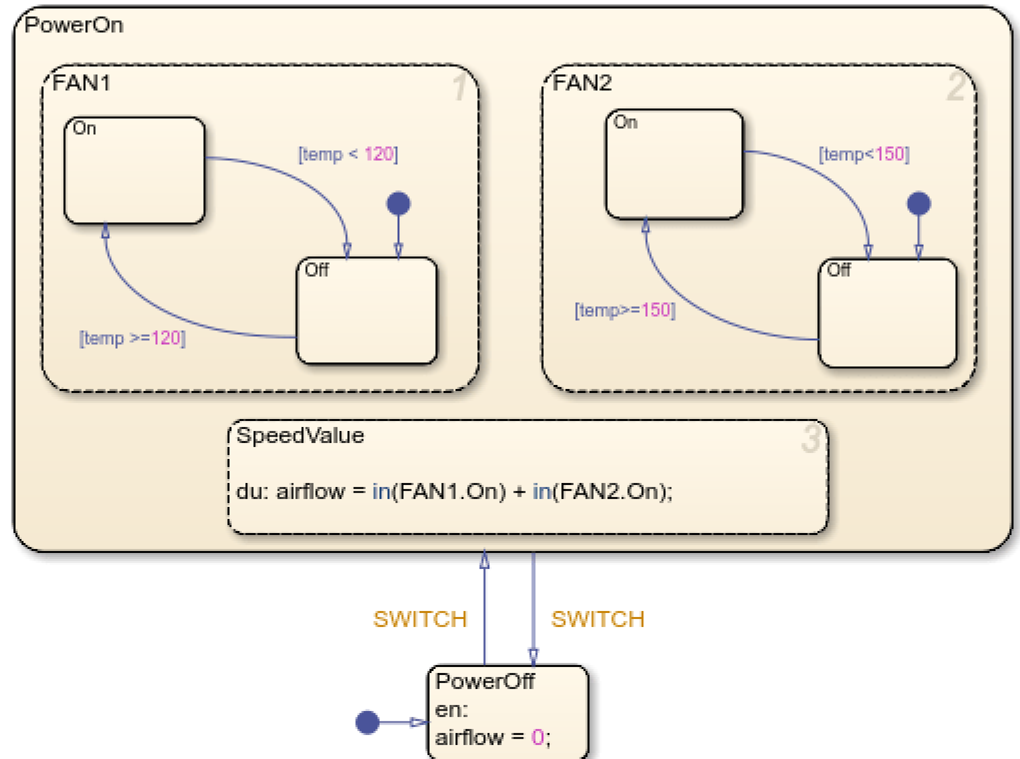
A state describes an operating mode of a reactive system. In a Stateflow chart, states are used for sequential design to create state transition diagrams.

During simulation, states can be active or inactive. The activity of a state changes depending on events and conditions. Events drive the execution of the state transition diagram by making states become active or inactive. For more information, see “Execution of a Stateflow Chart” on page 2-12.

To represent multiple levels of components in a system, create a hierarchy of states by nesting substates inside superstates. For more information, see “Use State Hierarchy to Design Multilevel State Complexity” on page 1-33.

To model mutually exclusive operating modes, enable exclusive (OR) decomposition in a state so at most one of its substates is active at the same time. To implement operating modes that run concurrently, enable parallel (AND) decomposition in a state so all of its substates are active at the same time. For more information, see “Define Exclusive and Parallel Modes by Using State Decomposition” on page 1-35.

For example, in this chart, the states `PowerOn` and `PowerOff` represent the on and off modes of an air controller system. In the state `PowerOn`, the parallel substates `FAN1` and `FAN2` represent the operating modes of a pair of fans. Each of these states contains exclusive substates called `On` and `Off`. For more information on this example, see “Model Synchronous Subsystems by Using Parallel Decomposition”.



Create a State

To add a state to a Stateflow chart:

1. Open the chart.
2. In the object palette, click the State icon .
3. On the chart canvas, click the location for the new state.
4. Enter a label for the state and click outside of the state. The label specifies the name of the state and any optional actions that the state executes during simulation. For more information, see “Define Actions in a State” on page 1-27.

After you create a state, you can use the Stateflow Editor to change the size, position, and contents of the state:

- To resize the state, click and drag the corner of the state.
- To move the state, click and drag the interior of the state.
- To edit the state label, click the label text near the character position you want to edit.

Tip A parent state must be graphically large enough to accommodate all its substates, so you may need to resize a parent state before dragging a new substate into it. Alternatively, you can convert a superstate into a subchart. For more information, see “Encapsulate Modal Logic by Using Subcharts” on page 6-6.

Define Actions in a State

The label for a state specifies the name of the state and any optional actions that the state executes during simulation. A state label appears on the top left corner of the state and consists of this general format:

```
name
entry: entry_actions
during: during_actions
exit: exit_actions
on event_name: on_event_actions
on message_name: on_message_actions
bind: event_name, data_name
```

State actions can appear in any order. For each type of action, you can specify more than one statement by entering each statement on a separate line. Alternatively, to separate multiple statements on the same line, use a comma or a semicolon. You can also combine `entry`, `during`, and `exit` actions that execute the same statements. For more information, see “Eliminate Redundant Code by Combining State Actions” on page 14-2.

Tip If you add statements directly after the state name, the chart interprets these statements as combined `entry` and `during` actions.

State Name

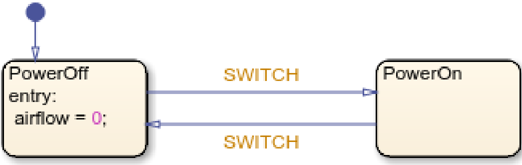
A state label starts with the name of the state, followed by an optional slash (/). State names are case sensitive and consist of a combination of alphanumeric and underscore characters. For more information, see “Guidelines for Naming Stateflow Objects” on page 1-66.

To avoid naming conflicts, do not assign the same name to sibling states. However, you can use the same state name for multiple states if the full name of each state is unique. The full name of a state consists of the sequence of ancestor names in the state hierarchy, separated by periods. For instance, in the previous example, the states in `FAN1` and `FAN2` are identified by these unique full names:

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

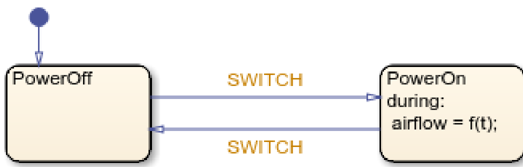
Entry Actions

To add an `entry` action, enter `entry` or `en`, followed by a colon (`:`) and one or more statements. The chart executes these statements when the state becomes active. For example, in this chart, the `entry` action in state `PowerOff` sets the value of `airflow` to zero when the air controller system turns off. For more information, see “Enter a Chart or State” on page 2-17.



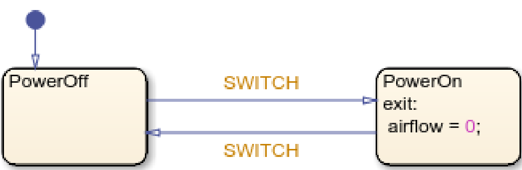
During Actions

To add a `during` action, enter `during` or `du`, followed by a colon (`:`) and one or more statements. The chart executes these statements when the state is active and there are no valid transitions to another state. For example, in this chart, the `during` action in state `PowerOn` calculates the value of `airflow` when the air controller system is on. For more information, see “Execution of a Stateflow Chart” on page 2-12.



Exit Actions

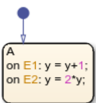
To add an `exit` action, enter `exit` or `ex`, followed by a colon (`:`) and one or more statements. The chart executes these statements when the state is active and a transition out of the state occurs. For example, in this chart, the `exit` action in state `PowerOn` sets the value of `airflow` to zero when the air controller system turns off. For more information, see “Exit a State” on page 2-23.



On Actions

To add an on action, enter on, followed by the name of an event or message, a colon (:), and one or more statements. The chart executes these statements when the state is active and it receives the specified event or message. For more information, see "Synchronize Model Components by Broadcasting Events" on page 12-2 and "Communicate with Stateflow Charts by Sending Messages" on page 13-2.

You can specify on actions for more than one event or message. For example, this state contains different on actions for the events E1 and E2.



If multiple events occur at the same time, the corresponding on actions execute in the order that they appear in the state label. For more information, see "Execution of a Stateflow Chart" on page 2-12.

Tip You can use implicit events (such as change, enter, or exit) and temporal logic operators (such as after, at, before, or every) to trigger on actions in states. For more information, see "Control Chart Behavior by Using Implicit Events" on page 12-28 and "Control Chart Execution by Using Temporal Logic" on page 14-35.

Bind Actions

To add a bind action, enter bind, followed by a colon (:) and the name of one or more events or data objects. To separate multiple events and data, use semicolons or commas or enter the events and data on separate lines. For example, in this chart, state A contains a bind action that binds the event E and the data object x to the state.

A bind action applies regardless of whether the state is active or inactive. Other states can read the bound data or listen for the bound events, but only the state and its children can change the value of the bound data or broadcast the bound events. Otherwise, a compile-time error occurs.

Binding a function-call event to a state also binds the function-call subsystem that the event calls. The function-call subsystem is enabled when the binding state is active and disabled when the binding

state is inactive. For more information, see “Control Function-Call Subsystems by Using bind Actions” on page 14-29.

If a chart includes actions that bind the same data or event to multiple states, a compile-time error occurs.

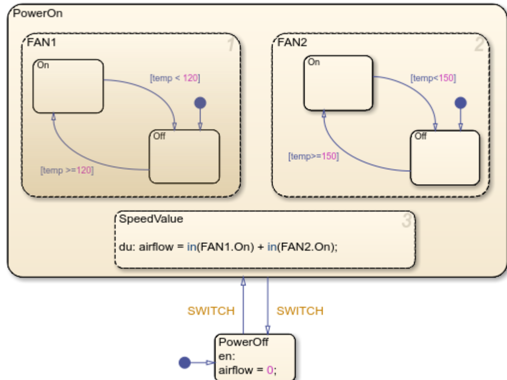
Standalone Stateflow charts in MATLAB do not support `bind` actions.

Group States

You can simplify editing a chart by grouping the contents of a state so they act as a single graphical unit. For example, moving a grouped state also moves the substates, transitions, and other graphical objects inside that state. To group a state, right-click the state and select **Group & Subchart** > **Group** in the context menu.

You must ungroup a state before selecting objects inside the state or moving other graphical objects into the state. For example, trying to move a state or graphical function into a grouped state results in an invalid intersection error on page 30-22. To ungroup a state, right-click the state and clear **Group & Subchart** > **Group** in the context menu.

In the Stateflow Editor, grouped states appear darker than ungrouped states. For example, in this chart, the state `FAN1` is grouped while the state `FAN2` is not grouped.



Specify Properties for States

The properties listed below enable you to specify how a state interacts with the other components in your Stateflow chart. You can modify these properties in the **Property Inspector**, the Model Explorer, or the State properties dialog box.

To use the **Property Inspector**:

1. In the **Modeling** tab, under **Design Data**, select **Property Inspector**.
2. In the Stateflow Editor, select the state.
3. In the **Property Inspector**, edit the state properties.

To use the Model Explorer:

1. In the **Modeling** tab, under **Design Data**, select **Model Explorer**.

2 In the **Model Hierarchy** pane, select the state.
3 In the **Dialog** pane, edit the state properties.

To use the State properties dialog box:

1 In the Stateflow Editor, right-click the state.
2 Select **Properties**.
3 In the properties dialog box, edit the state properties.

You can also modify state properties programmatically by using `Stateflow.State` objects. For more information about the Stateflow programmatic interface, see “Overview of the Stateflow API”.

Name

Name of the state. This property is read-only. When you click the state name hyperlink in the Model Explorer and the State properties dialog box, the Stateflow Editor brings the state to the foreground.

Execution order

Execution order for a parallel (AND) state. This property does not appear for exclusive (OR) states. For more information, see “Execution Order for Parallel States” on page 2-46.

Create output for monitoring

Whether to create an active state data output port for the state. See “Monitor State Activity Through Active State Data” on page 11-2.

Function inline option

Appearance of the state functions in generated code. Options include:

- `Auto` — An internal calculation determines the appearance of state functions in generated code.
- `Inline` — Calls to state functions are replaced by code as long as the function is not part of a recursion.
- `Function` — State functions are implemented as separate static functions.

For more information, see “Inline State Functions in Generated Code” (Simulink Coder). This property is not available in the **Property Inspector**.

Label

Label for the state. For more information, see “Define Actions in a State” on page 1-27. This property is not available in the **Property Inspector**.

Log self activity

Whether to enable signal logging. Signal logging saves the self activity of the state to the MATLAB workspace during simulation. For more information, see “Log Simulation Output for States and Data” on page 11-13.

Logging name

Signal name used to log the state activity.

- To use the name of the state, select `Use state name`.
- To specify a different name, select `Custom` and enter the custom logging name.

Limit data points to last

Whether to limit the number of logged data points to the specified maximum. For example, if you set the maximum number of data points to 5000, the chart logs only the last 5000 data points generated by the simulation.

Decimation

Whether to limit the amount of logged data by skipping samples using the specified decimation interval. For example, if you set a decimation interval of 2, the chart logs every other sample.

Test point

Whether to set the state as a test point that you can monitor with a floating scope during simulation. You can also log test point values to the MATLAB workspace. For more information, see “Monitor Test Points in Stateflow Charts” on page 11-38.

Description

Description of the state.

Document link

Link to online documentation for the state. You can enter a web URL address or a MATLAB command that displays documentation as an HTML file or as text in the MATLAB Command Window. When you click the **Document link** hyperlink, Stateflow evaluates the link and displays the documentation.

See Also

Objects
`Stateflow.State`

Tools
Model Explorer

More About

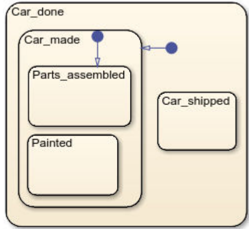
- “Use State Hierarchy to Design Multilevel State Complexity” on page 1-33
- “Define Exclusive and Parallel Modes by Using State Decomposition” on page 1-35
- “Execution of a Stateflow Chart” on page 2-12
- “Eliminate Redundant Code by Combining State Actions” on page 14-2
- “Control Function-Call Subsystems by Using bind Actions” on page 14-29

Use State Hierarchy to Design Multilevel State Complexity

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

State Hierarchy Example

For example, this chart has three levels of hierarchy. 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`

Create Substates and Superstates

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

Note A parent state must be graphically large enough to accommodate all its substates. You might need to resize a parent state before dragging a new substate into it. You can bypass the need for a state of large graphical size by declaring a superstate to be a subchart. See “Encapsulate Modal Logic by Using Subcharts” on page 6-6 for details.

Objects That a State Can Contain

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

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

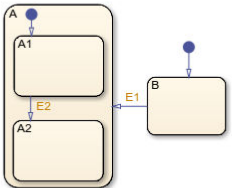
Define Exclusive and Parallel Modes by Using State Decomposition

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.

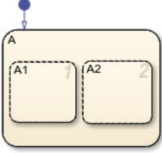
For example, in this chart, either state A or state B can be active. If state A is active, either state A1 or state A2 can be active at a given time.



Parallel (AND) State Decomposition

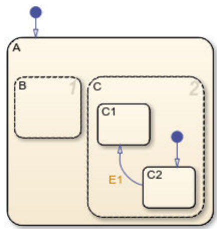
Substates with dashed borders indicate parallel (AND) decomposition. Use this decomposition to describe concurrent operating modes. When a state has parallel (AND) decomposition, all substates are active at the same time.

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.



Specify Substate Decomposition

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

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

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

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

Specify Activation Order for Parallel States

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

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

For more information, see “Explicit Ordering of Parallel States” on page 2-46 and “Implicit Ordering of Parallel States” on page 2-47.

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

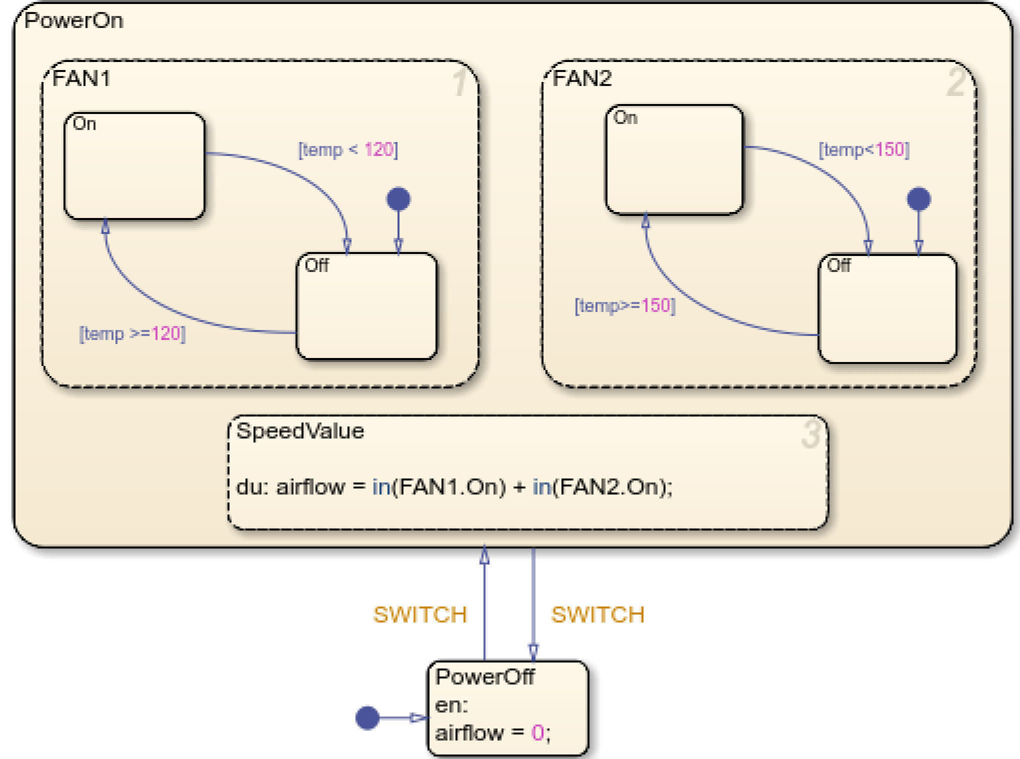
Transition Between Operating Modes

A transition represents the passage of a reactive system from one operating mode to another. In a Stateflow chart, a transition is a line with an arrowhead that typically connects two states. The source of the transition is the state where the transition begins and the destination of the transition is the state where the transition ends.

You can also create a transition path with more than one transition segment by using connective junctions. For example, you can create a transition path from a single source to multiple destinations or from multiple sources to a single destination. In this case, any intermediate transitions have a connective junction as a source or destination. For more information, see “Combine Transitions and Junctions to Create Branching Paths” on page 1-54.

A default transition is a special type of transition that has no source. In charts or states with exclusive (OR) decomposition and at least two substates, there must be a default transition path that is not guarded by a condition or triggered by an event. In the absence of history junctions, default transitions indicate the first substate to become active when the chart or superstate becomes active. For more information, see “Use Default Transitions to Specify Initial Substate Activity” on page 1-44 for details.

For example, the states in this chart represent the operating modes of an air controller system with two fans. The transitions between the states `PowerOn` and `PowerOff` represent the change of mode as the air controller system turns on and off. The default transition to the state `PowerOff` indicates that the system is off when the chart wakes up at the start of the simulation. Similarly, in the states `FAN1` and `FAN2`, the transitions between the substates represent the change of mode as each fan turns on and off and the default transition indicates that the fans are off when the superstate `PowerOn` becomes active. For more information on this example, see “Model Synchronous Subsystems by Using Parallel Decomposition”.

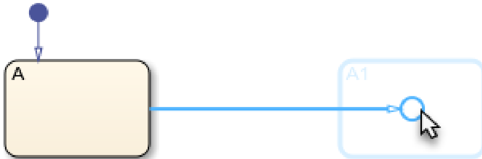


The parent of a transition is the lowest level state or chart that contains the source and destination of the transition. For instance, in the previous example, `FAN1` is the parent state of the transitions between the substates `FAN1.On` and `FAN1.Off`. Similarly, `FAN2` is the parent state of the transitions between the substates `FAN2.On` and `FAN2.Off`. In contrast, the parent of the transitions between the states `PowerOn` and `PowerOff` is the chart itself.

Create a Transition

You can add a transition to a new or existing destination.

1. Point to the border of the source state or junction. The pointer changes to a crosshair.
2. Click and drag away from the source state or junction. The Stateflow Editor provides graphical cues that allow you to add a junction or a state.



3. Select a destination for the new transition:

   - To add a new state at the end of the transition, click the rectangular cue.
   - To add a new junction at the end of the transition, click the circular cue.

- To connect the transition to an existing state or junction, drag the pointer to the desired destination.

4 Enter a label for the transition and click away from the transition. The label specifies the conditions and triggers that make the transition valid, as well as any actions that the transition executes during simulation. For more information, see “Define Actions in a Transition” on page 1-39.

After you create a transition, you can use the Stateflow Editor to change the shape, source, destination, or label of the transition:

- To reshape the transition, click and drag the midpoint of the transition.
- To change the source or destination, click and drag an endpoint of the transition.
- To edit the transition label, click the label text near the character position you want to edit. If your transition has an empty label, first select the transition.
- To move the transition label, click and drag the label.

Create a Default Transition

1. In the object palette, click the Default transition icon .
2. On the chart canvas, click a side of the destination state or junction.
3. Optionally, add a label for the default transition.

Tip The size of the endpoint of the default transition is proportional to the arrowhead size. See “Change Transition Arrowhead Size” on page 1-41.

Define Actions in a Transition

The label for a transition specifies an event or message trigger and a condition that makes the transition valid, as well as a condition action and a transition action that the transition executes during simulation. Transition labels have this general format:

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

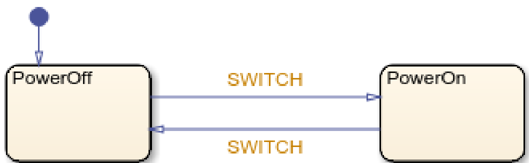
Each part of the label is optional and can appear on a separate line. For more information on how the chart uses labels to determine the validity of transitions, see “Evaluate Transitions” on page 2-26.

Event and Message Triggers

A transition label starts with the name of an event or message that triggers the transition. To specify multiple event or message triggers, use the logical OR (`|`) operator.

A transition with an event trigger is valid only when the chart receives a broadcast of the specified event. A transition with a message trigger is valid only when the specified message is present in the message queue. For more information, see “Synchronize Model Components by Broadcasting Events” on page 12-2 and “Communicate with Stateflow Charts by Sending Messages” on page 13-2.

For example, in this chart, the transitions between states `PowerOff` and `PowerOn` have event triggers. These transitions are valid when the source state is active and the chart receives a broadcast of the input event `SWITCH`.



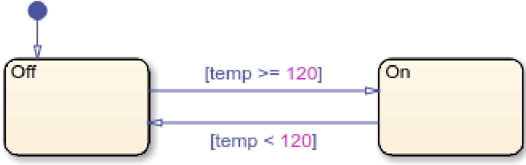
Tip You can use implicit events (such as `change`, `enter`, or `exit`) and temporal logic operators (such as `after`, `at`, `before`, or `every`) to trigger transitions. For more information, see "Control Chart Behavior by Using Implicit Events" on page 12-28 and "Control Chart Execution by Using Temporal Logic" on page 14-35.

Conditions

To add a condition to a transition label, enter a Boolean expression enclosed in square brackets (`[]`). Follow these guidelines for defining conditions:

- The condition expression must be a Boolean expression that evaluates to `true` (1) or `false` (0).
- To combine multiple Boolean expressions, use the logical AND (&&) and OR (`||`) operators.
- To enter the condition expression on more than one line, use an ellipsis (`...`).
- The condition expression can call graphical functions, truth table functions, MATLAB functions, or Simulink functions that return a numeric value. However, the function must not modify any data values or cause the chart to change state.
- Do not use assignment statements in condition expressions.

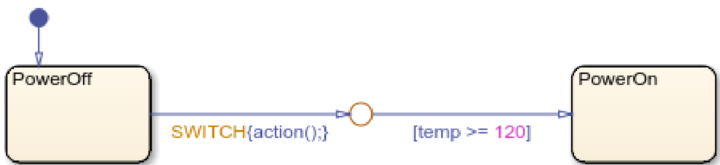
A transition with a condition is valid only when the specified expression is true. For example, in this chart, the transitions between substates `Off` and `On` have conditions that compare the value of the chart input `temp` to a threshold. These transitions are valid when the source state is active and the condition is true.



Condition Actions

The chart executes a condition action as soon as it determines that the transition is valid. To add a condition action, after the condition, enter one or more statements enclosed in braces (`{}`). To separate multiple statements, use a comma or a semicolon. Alternatively, you can enter each statement on a separate line. To enter a single statement on more than one line, use an ellipsis (`...`).

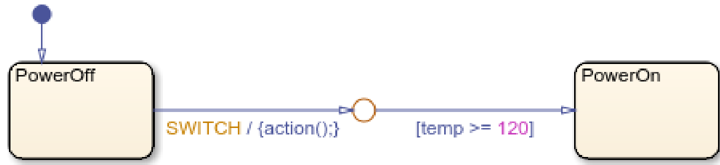
If the transition is part of a transition path that consists of multiple segments, that chart executes the condition action before it evaluates any subsequent segments of the transition path. For example, this chart contains a transition path with two transition segments. When the chart receives a broadcast of the input event `SWITCH`, the chart executes the condition action on the first transition segment regardless of whether the second transition segment is valid. For more information, see "Control Chart Execution by Using Condition Actions" on page A-8.



Transition Actions

The chart executes a transition action when it determines that the entire transition path is valid. To add a transition action, after the condition action, enter a forward slash (`/`) followed by one or more statements enclosed in braces (`{}`). To separate multiple statements, use a comma or a semicolon. Alternatively, you can enter each statement on a separate line. To enter a single statement on more than one line, use an ellipsis (`...`).

If the transition is part of a transition path that consists of multiple segments, that chart executes the transition action after it determines that there is a sequence of valid segments to a destination state or a terminal junction. For example, this chart contains a transition path with two transition segments. When the chart receives a broadcast of the input event `SWITCH`, the chart executes the transition action on the first transition segment only if the second transition segment is valid. For more information, see “Evaluate Outer Transitions with Condition and Transition Actions” on page 2-32.



Transition actions are not supported in standalone Stateflow charts in MATLAB.

Tip In charts that use C as the action language, you do not have to enclose transition actions in braces. In charts that use MATLAB as the action language, the syntax is automatically corrected if the braces are missing from the transition action. See “Auto Correction When Using MATLAB as the Action Language” on page 15-2.

Change Transition Arrowhead Size

To adjust the size of the arrowhead on a transition:

1. Right-click the transition.
2. Select **Arrowhead Size**.
3. Choose an arrowhead size from the drop-down list.

Alternatively, you can adjust the arrowhead size of more than one transitions at the same time.

1. Select multiple transitions.
2. Right-click one of the selected transitions.
3. Select **Format** > **Arrowhead Size**.

4 Choose an arrowhead size from the drop-down list.

Changing the arrowhead size on a transition also changes the arrowhead size of the other transitions with the same destination.

Specify Properties for Transitions

The properties listed below enable you to specify how a transition interacts with the other components in your Stateflow chart. You can modify these properties in the **Property Inspector**, the Model Explorer, or the Transition properties dialog box.

To use the **Property Inspector**:

1. In the **Modeling** tab, under **Design Data**, select **Property Inspector**.
2. In the Stateflow Editor, select the transition.
3. In the **Property Inspector**, edit the transition properties.

To use the Model Explorer:

1. In the **Modeling** tab, under **Design Data**, select **Model Explorer**.
2. In the **Model Hierarchy** pane, select the parent state or chart for the transition.
3. In the **Contents** pane, select the transition.
4. In the **Dialog** pane, edit the transition properties.

To use the Transition properties dialog box:

1. In the Stateflow Editor, right-click the transition.
2. Select **Properties**.
3. In the properties dialog box, edit the transition properties.

You can also modify transition properties programmatically by using `Stateflow.Transition` objects. For more information about the Stateflow programmatic interface, see “Overview of the Stateflow API”.

Source

Source of the transition. This property is read-only and is not available in the **Property Inspector**. When you click the source hyperlink, the Stateflow Editor brings the transition source to the foreground.

Destination

Destination of the transition. This property is read-only and is not available in the **Property Inspector**. When you click the destination hyperlink, the Stateflow Editor brings the transition destination to the foreground.

Parent

Parent of the transition. This property is read-only and is not available in the **Property Inspector**. When you click the parent hyperlink, the Stateflow Editor brings the parent to the foreground.

Execution order

Execution order for the transition. For more information, see “Transition Evaluation Order” on page 2-28.

Treat as variant transition

Whether the transition is a variant transition. For more information, see “Control Indicator Lamp Dimmer Using Variant Conditions” on page 29-9.

Label

The label for the transition. This property is not available in the **Property Inspector**. For more information, see “Define Actions in a Transition” on page 1-39.

Description

Description of the transition.

Document link

Link to online documentation for the transition. You can enter a web URL address or a MATLAB command that displays documentation as an HTML file or as text in the MATLAB Command Window. When you click the **Document link** hyperlink, Stateflow evaluates the link and displays the documentation.

See Also

Objects
`Stateflow.Transition`

Tools
Model Explorer

More About

- “Represent Operating Modes by Using States” on page 1-26
- “Combine Transitions and Junctions to Create Branching Paths” on page 1-54
- “Control Chart Execution by Using Condition Actions” on page A-8
- “Synchronize Model Components by Broadcasting Events” on page 12-2
- “Communicate with Stateflow Charts by Sending Messages” on page 13-2
- “Use Default Transitions to Specify Initial Substate Activity” on page 1-44
- “Control Chart Execution by Using Inner Transitions” on page A-15

Use Default Transitions to Specify Initial Substate Activity

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

Drawing Default Transitions

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

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

Label Default Transitions

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

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

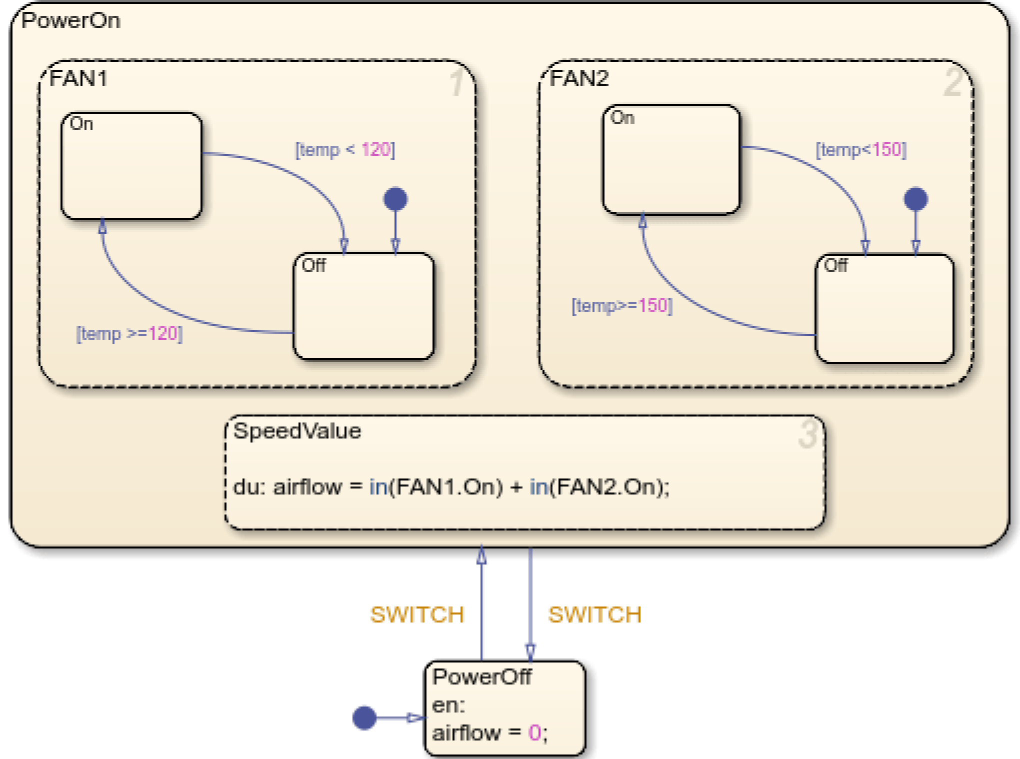
Default Transition Examples

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

- “Default Transition to a State Example” on page 1-44
- “Default Transition to a Junction Example” on page 1-45
- “Default Transition with a Label Example” on page 1-46

Default Transition to a State Example

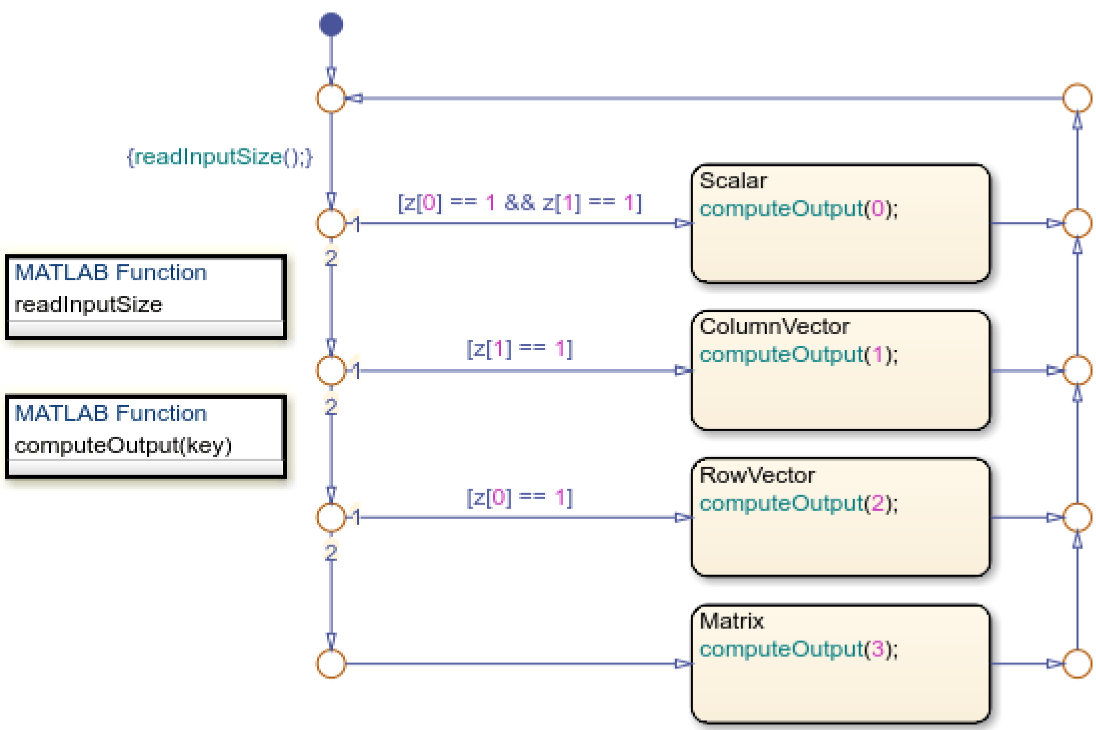
This example shows a default transition to a state.



The default transition to state `PowerOff` ensures that, when the chart wakes up, the state becomes active. For more information, see “Control Chart Execution by Using Default Transitions” on page A-12.

Default Transition to a Junction Example

This example shows a default transition to a connective junction.

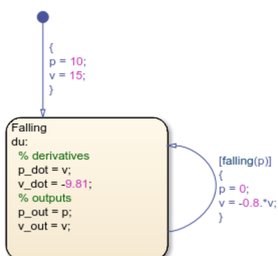


The default transition to the connective junction defines that upon entering the chart, the destination depends on the condition of each transition segment.

See "Default Transition to a Junction" on page A-12 for information on the semantics of this notation.

Default Transition with a Label Example

This example shows a default transition with a label.



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

See "Labeled Default Transitions" on page A-14 for more information on the semantics of this notation.

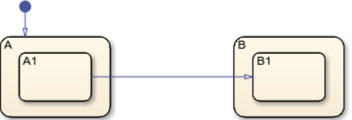
See Also

Related Examples

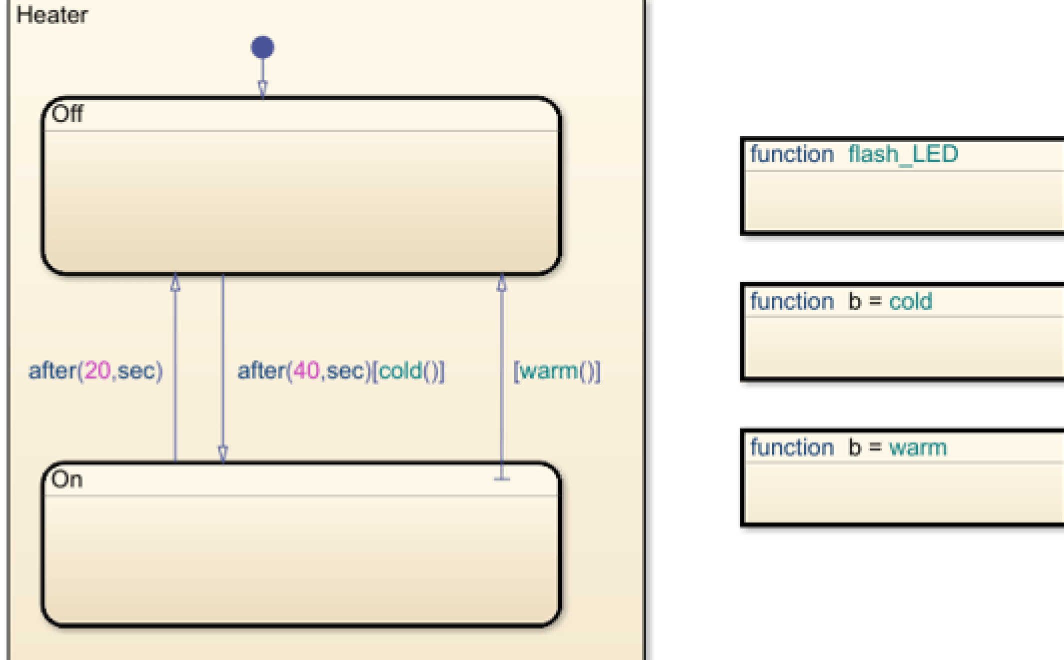
- “Transition Between Operating Modes” on page 1-37

Move Between Levels of Hierarchy by Using Supertransitions

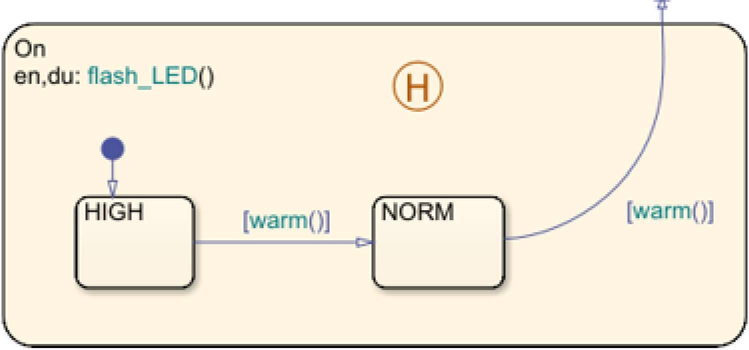
A supertransition is a transition between different levels in a chart. A supertransition can go between a state in a top-level chart and a substate in one of its sibling states, or between substates in different states of the chart. For example, this chart contains a supertransition between the substates of two sibling states.



You can create supertransitions that span any number of levels in your chart. When a supertransition crosses into or out of a subchart, it consists of multiple transition segments, each one at a different containment level. For example, this chart model shows a supertransition leaving the `On` subchart.



The same supertransition appears inside the subchart.



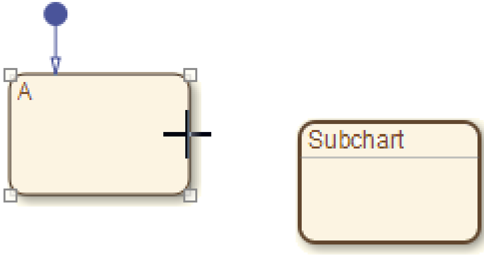
You can label any transition segment by using the procedure described in “Define Actions in a Transition” on page 1-39. The resulting label appears on each segment of the supertransition. For instance, in the previous example, both segments of the supertransition have the same label, `[warm()]`. If you change the label on any segment, the change also appears on the other segments.

The points where each segment enters or exits the subchart affect one another. For example, moving the point where the supertransition exits the boundary of the subchart `On` also moves the point where the supertransition emerges in the top-level chart.

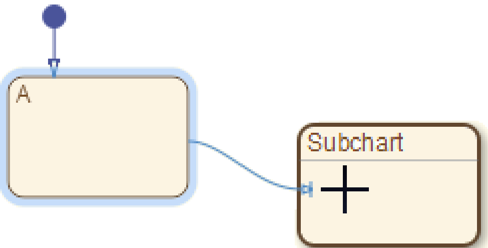
Tip Entry and exit ports provide an alternative method to transition across boundaries in the Stateflow hierarchy. For more information, see “Decide Between Supertransitions and Entry and Exit Ports” on page 1-52.

Create a Supertransition That Enters a Subchart

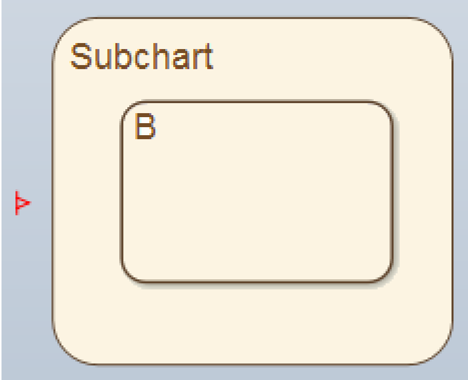
1. Point to the border of the source state. The pointer changes to a crosshair.



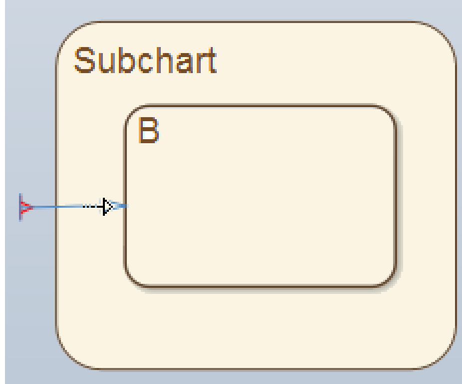
2. Click and drag inside the border of the subchart. A supertransition connects the source state to the subchart. To change where the transition enters the subchart, you can drag the endpoint of the transition segment around the inside boundary of the subchart.



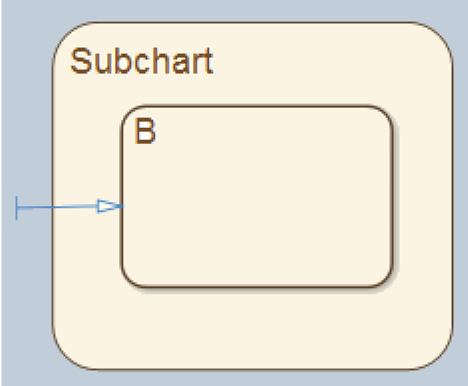
3 Open the subchart by double-clicking it. The arrowhead of the supertransition appears highlighted in red.



4 Click the arrowhead and drag the pointer to the desired destination in the subchart.



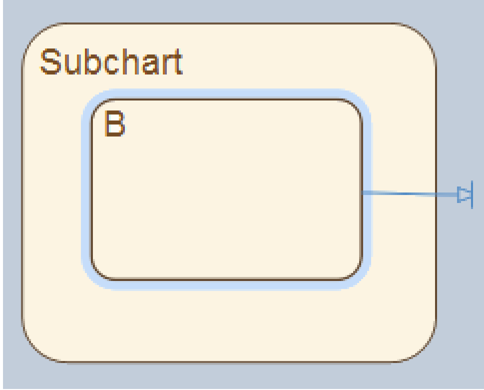
5 Release the pointer.



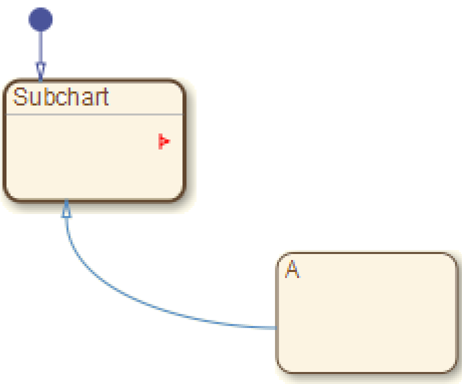
Note When you draw a supertransition across subchart boundaries, the Undo and Redo buttons are disabled. You cannot undo or redo any prior operations.

Create a Supertransition That Exits a Subchart

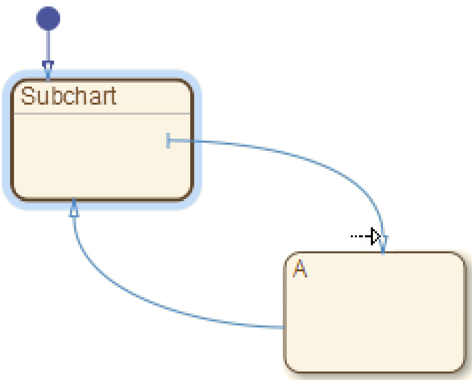
1. Draw a transition to a location outside the border of the subchart.



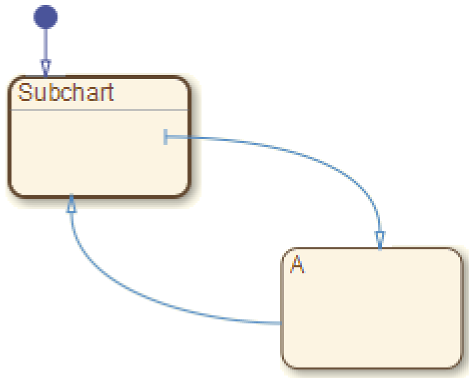
2. Navigate to the parent of the subchart. The arrowhead of the supertransition appears highlighted in red.



3. Click the arrowhead and drag the pointer to the desired destination in the chart.



4 Release the pointer.



Note When you draw a supertransition across subchart boundaries, the Undo and Redo buttons are disabled. You cannot undo or redo any prior operations.

Decide Between Supertransitions and Entry and Exit Ports

Both supertransitions and entry and exit ports enable you to move across different levels in the chart hierarchy. Which approach you select depends on your design requirements.

Scenario	Recommendation
Transition between the substates of two sibling states, neither of which is a subchart	Use a supertransition. You can create a supertransition that does not cross any subchart boundaries by simply clicking the boundary of the source state and dragging your pointer to the destination state.

Scenario	Recommendation
Transition to or from a substate of a normal subchart	Use either a supertransition or an entry or exit port. • If you use a supertransition, the points where each segment of the supertransition enters or exits different levels of the hierarchy affect one another. For example, moving the point where the supertransition enters the boundary of the subchart also moves the point where the supertransition exits the boundary of the subchart. • If you use an entry or exit port, the positions of the port and the matching junction are graphically independent of one another. For example, you can move the port without moving the junction.
Transition to or from a substate of an atomic subchart	Use an entry or exit port. Supertransitions cannot cross the boundary of atomic subcharts.

See Also

Related Examples

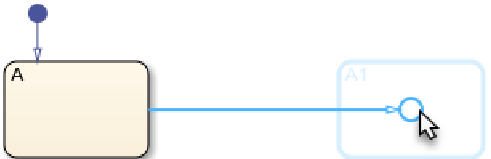
- “Transition Between Operating Modes” on page 1-37
- “Create Entry and Exit Connections Across State Boundaries” on page 1-61

Combine Transitions and Junctions to Create Branching Paths

A connective junction represents a decision point in a transition path. You can combine transitions and connective junctions to create paths from a single source to multiple destinations or from multiple sources to a single destination. For more information on the semantics of branching paths, see "Represent Multiple Paths by Using Connective Junctions" on page A-22.

Add a Connective Junction

When you add a transition to a chart, the Stateflow Editor provides graphical cues that allow you to add a junction or a state. To place a junction at the end of the transition, click the circular cue.



Alternatively, to add an isolated junction to a chart:

1. In the object palette, click the Junction icon.
2. On the chart canvas, click the location for the new connective junction.

Modify Connective Junction Properties

To change the size of one or more connective junctions:

1. Select the connective junctions.
2. Right-click one of the selected junctions and select **Junction Size**.
3. From the drop-down list, select a junction size.

To change other properties of a connective junction, right-click a connective junction and select **Properties**. The Connective Junction dialog box displays these properties:

- **Parent** — Parent state of the connective junction. To bring the parent to the foreground, click the hypertext link. This property is read-only.
- **Description** — Description of the junction.
- **Document link** — Link to online documentation for the junction. You can enter a web URL address or a MATLAB command that displays documentation in a suitable online format, such as an HTML file or text in the MATLAB Command Window. When you click the **Document link** hyperlink, Stateflow evaluates the link and displays the documentation.

Examples of Transition Paths with Connective Junctions

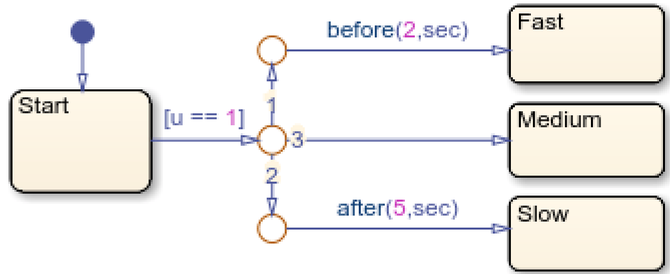
By combining transitions and connective junctions, you can construct common transition patterns such as:

- `if-then-else` decision patterns
- `for` loop patterns
- `while` loop patterns

To reduce the creation time of these patterns, use the Pattern Wizard. For more information, see “Create Flow Charts by Using Pattern Wizard” on page 3-5.

If-then-else Pattern with an Unconditional Transition

In this example, the transition from state `Start` has three connective junctions. The first two branches of the path are guarded by a condition. The last branch of the path is unconditional.



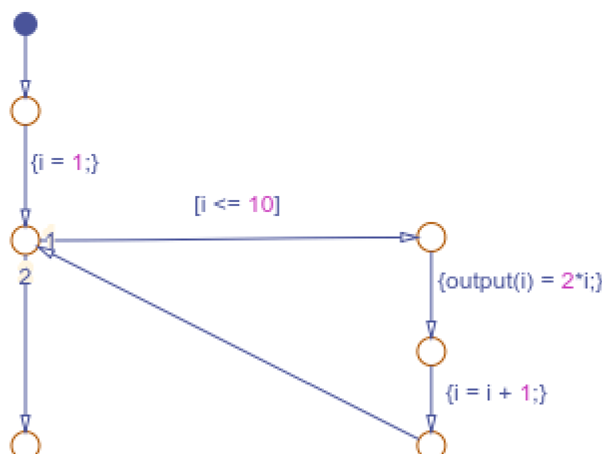
The chart uses temporal logic to determine when the input `u` equals 1:

- If `u` equals 1 before time $t = 2$, the state `Fast` becomes active.
- If `u` equals 1 between $t = 2$ and $t = 5$, the state `Good` becomes active.
- If `u` equals 1 after $t = 5$, the state `Slow` becomes active.

For more information about this chart, see “Detect Elapsed Time” on page 14-41.

For Loop Pattern

In this example, a flow chart uses a combination of transitions and connective junctions to construct a `for` loop.

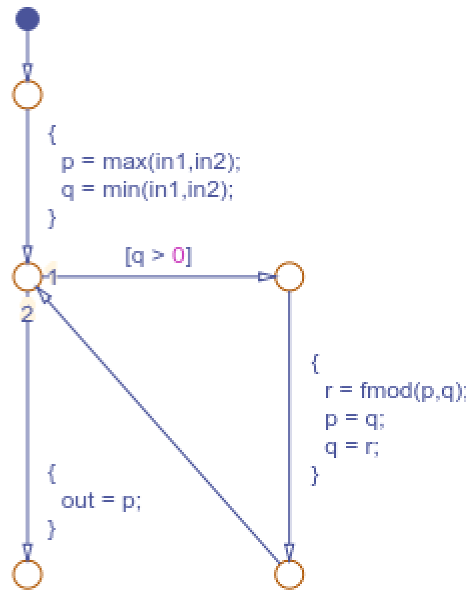


As the value of the counter `i` increases from 1 to 10, the flow chart defines the elements of an array `output`. The loop in this flow chart is equivalent to this snippet of MATLAB code:

```
for i = 1:10
    output(i) = 2*i;
end
```

While Loop Pattern

In this example, a flow chart combines transitions and connective junctions to construct a `while` loop that computes the greatest common divisor of the inputs.



The loop in this flow chart is equivalent to this snippet of C code:

```
while(q > 0) {
    r = fmod(p,q);
    p = q;
    q = r;
}
```

Specify Properties for Connective Junctions

You can modify the properties listed below in the **Property Inspector**, the Model Explorer, or the Connective Junction properties dialog box.

To use the **Property Inspector**:

1 In the **Modeling** tab, under **Design Data**, select **Property Inspector**.
2 In the Stateflow Editor, select the connective junction.
3 In the **Property Inspector**, edit the connective junction properties.

To use the Model Explorer:

1. In the **Modeling** tab, under **Design Data**, select **Model Explorer**.
2. In the **Model Hierarchy** pane, select the parent state or chart for the connective junction.
3. In the **Contents** pane, select the connective junction.
4. In the **Dialog** pane, edit the connective junction properties.

To use the Connective Junction properties dialog box:

1. In the Stateflow Editor, right-click the connective junction.
2. Select **Properties**.
3. In the properties dialog box, edit the connective junction properties.

You can also modify junction properties programmatically by using `Stateflow.Junction` objects. For more information about the Stateflow programmatic interface, see “Overview of the Stateflow API”.

Parent

Parent of the connective junction. This property is read-only and is not available in the **Property Inspector**. When you click the parent hyperlink, the Stateflow Editor brings the parent to the foreground.

Description

Description of the connective junction.

Document link

Link to online documentation for the connective junction. You can enter a web URL address or a MATLAB command that displays documentation as an HTML file or as text in the MATLAB Command Window. When you click the **Document link** hyperlink, Stateflow evaluates the link and displays the documentation.

See Also

Objects
`Stateflow.Junction`

Tools
Model Explorer

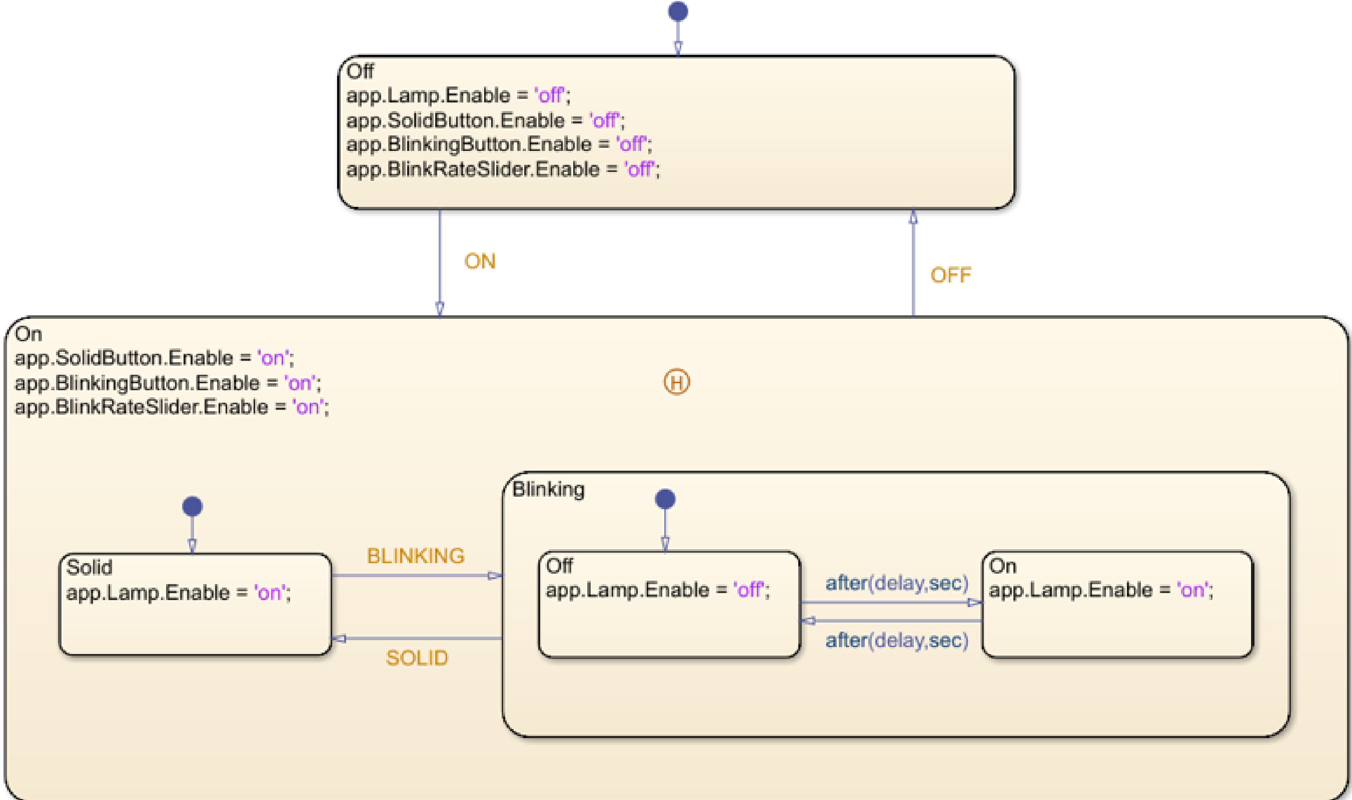
More About

- “Create Flow Charts by Using Pattern Wizard” on page 3-5
- “Represent Multiple Paths by Using Connective Junctions” on page A-22
- “How Stateflow Objects Interact During Execution” on page 1-8
- “Control Chart Execution by Using Temporal Logic” on page 14-35

Resume Prior Substate Activity by Using History Junctions

A history junction in a state records the activity of the substates. The first time that a state that contains a history junction becomes active, the state determines which substate is active by executing the default transition. If, after a period of inactivity, the state becomes active again, the state does not execute the default transition. Instead, the substates return to the prior state of activity.

For example, in this chart, the top-level states represent the on and off modes of a lamp. The state `On` contains two substates, `Solid` and `Blinking`, which correspond to the two modes of operation for the lamp. When state `On` becomes active for the first time, the state executes the default transition and substate `Solid` becomes active. This substate represents the default mode of operation for the lamp. The history junction Ⓗ in state `On` records which substate is active, so that, as the chart transitions from `On` to `Off` and back to `On`, the last active substate becomes active once again. In other words, when you turn on the lamp, it always returns to the previous mode of operation. For more information on this example, see “Design Human-Machine Interface Logic by Using Stateflow Charts” on page 31-24.



Tip A history junction can be the destination of a transition. For example, connecting an inner transition to a history junction is equivalent to drawing a self-loop transition on every substate. When the inner transition is valid, the chart exits and then immediately reenters the active substate. For more information, see “Inner Transition to a History Junction” on page A-20.

Add a History Junction

To add a history junction to a Stateflow chart:

1. Open the chart.
2. In the object palette, click the History Junction icon (H).
3. On the chart canvas, click the location for the new history junction.

After you add a history junction, you can use the Stateflow Editor to change the size and position of the junction:

- To move the junction, click and drag the junction.
- To resize the junction, right-click the junction, select **Junction Size**, and choose a junction size from the drop-down list.

Specify Properties for History Junctions

You can modify the properties listed below in the **Property Inspector**, the Model Explorer, or the History Junction properties dialog box.

To use the **Property Inspector**:

1. In the **Modeling** tab, under **Design Data**, select **Property Inspector**.
2. In the Stateflow Editor, select the history junction.
3. In the **Property Inspector**, edit the history junction properties.

To use the Model Explorer:

1. In the **Modeling** tab, under **Design Data**, select **Model Explorer**.
2. In the **Model Hierarchy** pane, select the parent state or chart for the history junction.
3. In the **Contents** pane, select the history junction.
4. In the **Dialog** pane, edit the history junction properties.

To use the History Junction properties dialog box:

1. In the Stateflow Editor, right-click the history junction.
2. Select **Properties**.
3. In the properties dialog box, edit the history junction properties.

You can also modify junction properties programmatically by using `Stateflow.Junction` objects. For more information about the Stateflow programmatic interface, see “Overview of the Stateflow API”.

Parent

Parent of the history junction. This property is read-only and is not available in the **Property Inspector**. When you click the parent hyperlink, the Stateflow Editor brings the parent to the foreground.

Description

Description of the history junction.

Document link

Link to online documentation for the history junction. You can enter a web URL address or a MATLAB command that displays documentation as an HTML file or as text in the MATLAB Command Window. When you click the **Document link** hyperlink, Stateflow evaluates the link and displays the documentation.

See Also

Objects
`Stateflow.Junction`

Tools
Model Explorer

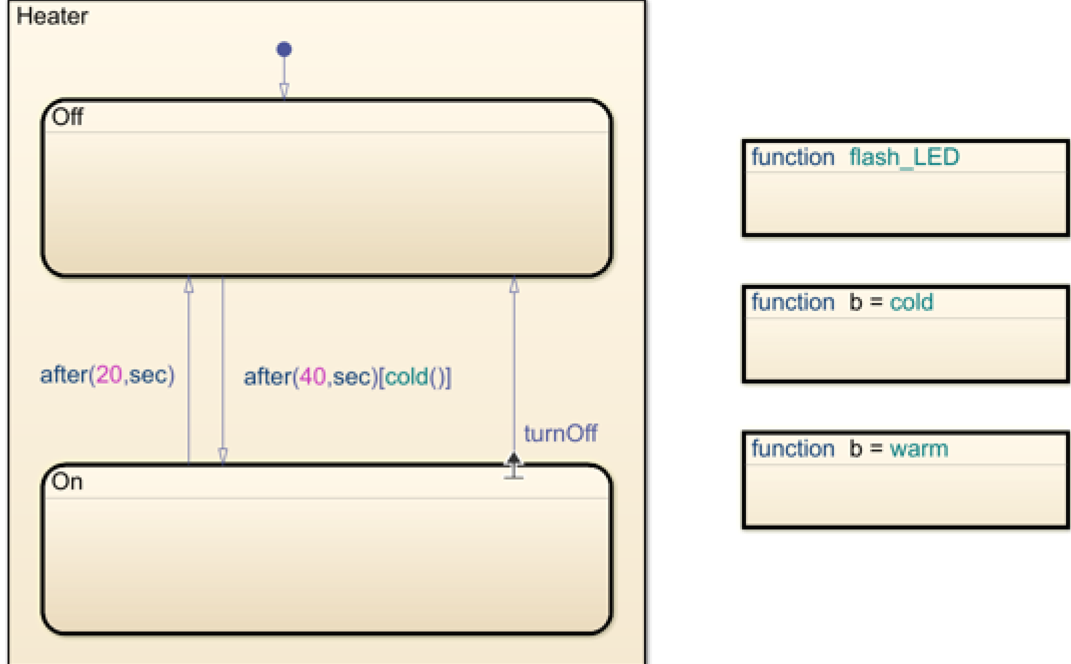
Related Examples

- “Use State Hierarchy to Design Multilevel State Complexity” on page 1-33
- “Monitor State Activity Through Active State Data” on page 11-2

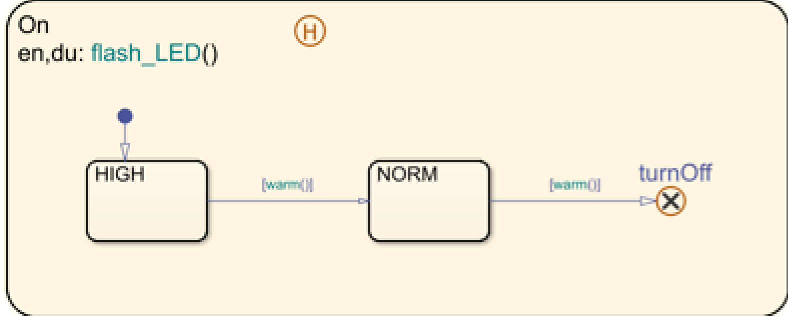
Create Entry and Exit Connections Across State Boundaries

Entry and exit ports represent connections into and out of states and subcharts. Like supertransitions, entry and exit ports create transition paths across boundaries in the Stateflow hierarchy. However, because entry and exit ports isolate the transition logic for entering and exiting states, they can be used in atomic subcharts. Entry and exit ports are not supported in standalone Stateflow charts in MATLAB.

In the Stateflow Editor, entry and exit ports appear as arrows on the boundary of a state or subchart. Each port has a matching junction that marks the entry or exit point inside the state or subchart. The entry junction icon ● and the exit junction icon ⊗ indicate the junction. A transition path that leads to an entry port continues along the transition connected to the matching entry junction. Similarly, a transition path that leads to an exit junction continues along the transition connected to the matching exit port. For example, in this chart, the exit port labeled `turnOff` represents the exit connection out of the subchart `On`.



In the subchart, the transition path leading to the exit junction defines the logic for exiting the subchart. In this example, the function `warm` must evaluate to `true` on two consecutive time steps before the chart makes the transition out of the `On` state.



For more information about this example, see “Model Bang-Bang Temperature Control System” on page 14-51. For other examples that use entry and exit ports, see “Robot Trajectory Planning with Reusable Components” on page 17-24 and “Launch Abort System” on page 27-31.

Add Entry and Exit Ports

To create an entry or exit port, add an entry or exit junction inside a state or subchart.

1. In the object palette, click the Entry icon ⊙ or the Exit icon ⊗.
2. On the chart canvas, click the location for the new entry or exit junction. A matching entry or exit port appears on the boundary of the state or subchart that contains the new entry or exit junction.
3. Enter a label for the junction and matching port. See “Add Labels to Identify Matching Junctions and Ports” on page 1-62.

Guidelines for Using Entry and Exit Ports

Add Entry and Exit Junctions Only to Exclusive (OR) States and Atomic Subcharts

Entry and exit junctions are supported only in exclusive (OR) states and atomic subcharts. Do not add entry or exit junctions to top level charts, parallel (AND) states, or boxes.

Add Labels to Identify Matching Junctions and Ports

Labels on entry and exit ports indicate which junction connects to which port. Adding a label is optional when a state contains only one entry or exit junction. Unique labels are required when a state contains more than one entry junction or more than one exit junction.

Prevent Backtracking Through Entry and Exit Ports

To ensure that the chart successfully enters or exits a state without backtracking, each entry junction and exit port must have a path that is not guarded by a condition or triggered by an event. Transition paths from entry junctions and exit ports must lead to states and must not contain terminal junctions.

Isolate the Transition Logic for Entry and Exit Junctions

Transition paths that start at entry junctions or end at exit junctions must be contained in the parent state.

Do Not Enter and Exit States in the Same Time Step

Default and inner transition paths must not connect to an exit junction.

Decide Between Supertransitions and Entry and Exit Ports

Both supertransitions and entry and exit ports enable you to move across different levels in the chart hierarchy. Which approach you select depends on your design requirements.

Scenario	Recommendation
Transition between the substates of two sibling states, neither of which is a subchart	Use a supertransition. You can create a supertransition that does not cross any subchart boundaries by simply clicking the boundary of the source state and dragging your pointer to the destination state.
Transition to or from a substate of a normal subchart	Use either a supertransition or an entry or exit port. • If you use a supertransition, the points where each segment of the supertransition enters or exits different levels of the hierarchy affect one another. For example, moving the point where the supertransition enters the boundary of the subchart also moves the point where the supertransition exits the boundary of the subchart. • If you use an entry or exit port, the positions of the port and the matching junction are graphically independent of one another. For example, you can move the port without moving the junction.
Transition to or from a substate of an atomic subchart	Use an entry or exit port. Supertransitions cannot cross the boundary of atomic subcharts.

Specify Properties for Entry and Exit Ports

You can modify the properties listed below in the **Property Inspector**, the Model Explorer, or the Entry Port, Exit Port, Entry Junction, or Exit Junction properties dialog box.

To use the **Property Inspector**:

1. In the **Modeling** tab, under **Design Data**, select **Property Inspector**.
2. In the Stateflow Editor, select the port or junction.
3. In the **Property Inspector**, edit the properties for the port or junction.

To use the Model Explorer:

1. In the **Modeling** tab, under **Design Data**, select **Model Explorer**.
2. In the **Model Hierarchy** pane, select the parent state or chart for the port or junction.

3. In the **Contents** pane, select the port or junction.
4. In the **Dialog** pane, edit the properties for the port or junction.

To use the Entry Port, Exit Port, Entry Junction, or Exit Junction properties dialog box:

1. In the Stateflow Editor, right-click the port or junction.
2. Select **Properties**.
3. In the properties dialog box, edit the properties for the port or junction.

You can also modify port or junction properties programmatically by using `Stateflow.Port` objects. For more information about the Stateflow programmatic interface, see “Overview of the Stateflow API”.

Parent

Parent of the port or junction. This property is read-only and is not available in the **Property Inspector**. When you click the parent hyperlink, the Stateflow Editor brings the parent to the foreground.

Home

Home state or subchart of the entry or exit port. The home of an entry or exit port is the state or subchart whose boundary contains the port. This property is read-only and is not available in the **Property Inspector**. When you click the home hyperlink, the Stateflow Editor brings the home state or subchart to the foreground.

Label

The label for the port or junction. This property is read-only for entry or exit ports. For more information, see “Define Actions in a Transition” on page 1-39.

Description

Description of the port or junction.

Document link

Link to online documentation for the port or junction. You can enter a web URL address or a MATLAB command that displays documentation as an HTML file or as text in the MATLAB Command Window. When you click the **Document link** hyperlink, Stateflow evaluates the link and displays the documentation.

See Also

Objects
`Stateflow.Port`

Tools
Model Explorer

Related Examples

- “Robot Trajectory Planning with Reusable Components” on page 17-24

- “Move Between Levels of Hierarchy by Using Supertransitions” on page 1-48
- “Model Bang-Bang Temperature Control System” on page 14-51
- “Launch Abort System” on page 27-31

Guidelines for Naming Stateflow Objects

You can name Stateflow objects with a combination of alphanumeric and underscore characters.

- Names cannot begin with a numeric character or contain embedded spaces.
- Name length should comply with the maximum identifier length enforced by Simulink Coder™ software. You can set the **Maximum identifier length** parameter. The default is 31 characters and the maximum length you can specify is 256 characters.
- Avoid using reserved keywords to name Stateflow objects. These keywords are part of the action language syntax.

Note Do not use the file names `sf.slx` for Simulink models or `sf.sfx` for standalone Stateflow charts in MATLAB. Using these file names can shadow Stateflow program files and result in unpredictable behavior.

Reserved Keywords

Usage in Action Language Syntax	Keywords	Syntax References
Change detection	• `hasChanged` • `hasChangedFrom` • `hasChangedTo` • `change` • `chg`	"Detect Changes in Data and Expression Values" on page 14-63
Complex data	• `complex` • `imag` • `real`	"Operations for Complex Data in Stateflow" on page 24-4
Data type operations	• `cast` • `type`	"Type Cast Operations" on page 14-7 "Specify Type of Stateflow Data" on page 10-20

Usage in Action Language Syntax	Keywords	Syntax References
Data types	• `boolean` • `double` • `int8` • `int16` • `int32` • `int64` • `single` • `uint8` • `uint16` • `uint32` • `uint64`	"Specify Type of Stateflow Data" on page 10-20
Edge detection	• `crossing` • `falling` • `rising`	"Detect Changes in Data and Expression Values" on page 14-63
Events	• `send` • `tick`	"Broadcast Local Events to Synchronize Parallel States" on page 12-25 "Control Chart Behavior by Using Implicit Events" on page 12-28
Interface with MATLAB code	• `matlab` • `ml` • `this`	"Access MATLAB Functions and Workspace Data in C Charts" on page 14-20 "Model a Power Window Controller" on page 27-51 "Simulate a Media Player" on page 21-14
Literal symbols	• `false` • `inf` • `t` • `true`	"Supported Symbols in Actions" on page 14-11
Messages	• `discard` • `forward` • `isvalid` • `length` • `receive` • `send`	"Control Message Activity in Stateflow Charts" on page 13-9

Usage in Action Language Syntax	Keywords	Syntax References
State actions	• bind • du • during • en • entry • ex • exit • on	“Represent Operating Modes by Using States” on page 1-26
State activity	• enter • en • exit • ex • in	“Check State Activity by Using the in Operator” on page 11-24
String manipulation	• ascii2str • str2ascii • str2double • strcat • strcmp • strcpy • strlen • substr • tostring	“Manage Textual Information by Using Strings” on page 21-2 **Note** These operator names are reserved keywords only in charts that use C as the action language.
Temporal logic	• after • at • before • count • duration • elapsed • et • every • msec • sec • temporalCount • usec	“Control Chart Execution by Using Temporal Logic” on page 14-35

See Also

More About

- “Supported Symbols in Actions” on page 14-11
- “Operations for Stateflow Data” on page 14-4
- “Operations for Vectors and Matrices in Stateflow” on page 19-4
- “Operations for Fixed-Point Data in Stateflow” on page 23-8
- “Operations for Complex Data in Stateflow” on page 24-4

Speed Up Simulation

Improve Model Update Performance

Stateflow uses Just-In-Time (JIT) compilation mode to improve model update performance for most charts in Simulink models. Stateflow applies JIT mode to charts that qualify. For a chart in JIT mode, Stateflow generates an execution engine in memory for simulation. For these charts, Stateflow does not generate C code or a MEX file to simulate the chart. JIT mode provides the best performance during the compilation of a model.

Some charts, such as charts with signal logging, do not qualify for JIT mode.

Stateflow models include debugging support for simulation. To gain optimal performance, turn off debugging by using this command:

```
sfc("coder_options",forceDebugOff=true);
```

When you run this command, your Stateflow charts do not have debugging support or run-time error checking.

Note When you turn off debugging, animation is also turned off.

Disable Simulation Target Parameters That Impact Execution Speed

To simulate your model more quickly, open the Configuration Parameters dialog box and, in the **Simulation Target** pane, under **Advanced parameters**, clear the check boxes for these parameters:

- **Echo expressions without semicolons** — To disable run-time output in the MATLAB Command Window, such as actions that do not terminate with a semicolon, clear this check box. See "Echo expressions without semicolons" (Simulink).
- **Break on Ctrl+C** — To disable ability to break out of long-running execution using Ctrl+C, clear this check box. See "Break on Ctrl+C" (Simulink).

Speed Up Simulation

Use these tips to further speed up simulation:

Keep Charts Closed

During model simulation, any open charts with animation enabled take longer to simulate. If you keep all charts closed, the simulation runs faster.

Disable Content Preview

During model simulation, any open charts with content preview enabled take longer to simulate. If you disable content preview, the simulation runs faster. To disable content preview, select the chart that has content preview enabled. On the **State Chart** tab, click **Content Preview**.

Keep Scope Blocks Closed

During model simulation, any open Scope blocks continuously update their display. If you keep all Scope blocks closed, you can speed up the simulation. After the simulation ends, you can open the Scope blocks to view the results.

Use Library Charts in Your Model

If your model contains multiple charts that do not use JIT mode and contain the same elements, you might generate multiple copies of identical simulation code. By using library charts, you can minimize the number of copies of identical simulation code. For example, using five library charts reduces the number of identical copies from five to one.

For more information, see “Create Specialized Chart Libraries for Large-Scale Modeling” on page 25-34.

See Also

More About

- “Reduce the Compilation Time of a Chart” on page 17-38

Stateflow Semantics

- “Stateflow Semantics” on page 2-2
- “Modeling Guidelines for Stateflow Charts” on page 2-8
- “Types of Chart Execution” on page 2-10
- “Execution of a Stateflow Chart” on page 2-12
- “Enter a Chart or State” on page 2-17
- “Exit a State” on page 2-23
- “Evaluate Transitions” on page 2-26
- “Super Step Semantics” on page 2-35
- “Use Events to Execute Charts” on page 2-40
- “Group and Execute Transitions” on page 2-44
- “Execution Order for Parallel States” on page 2-46

Stateflow Semantics

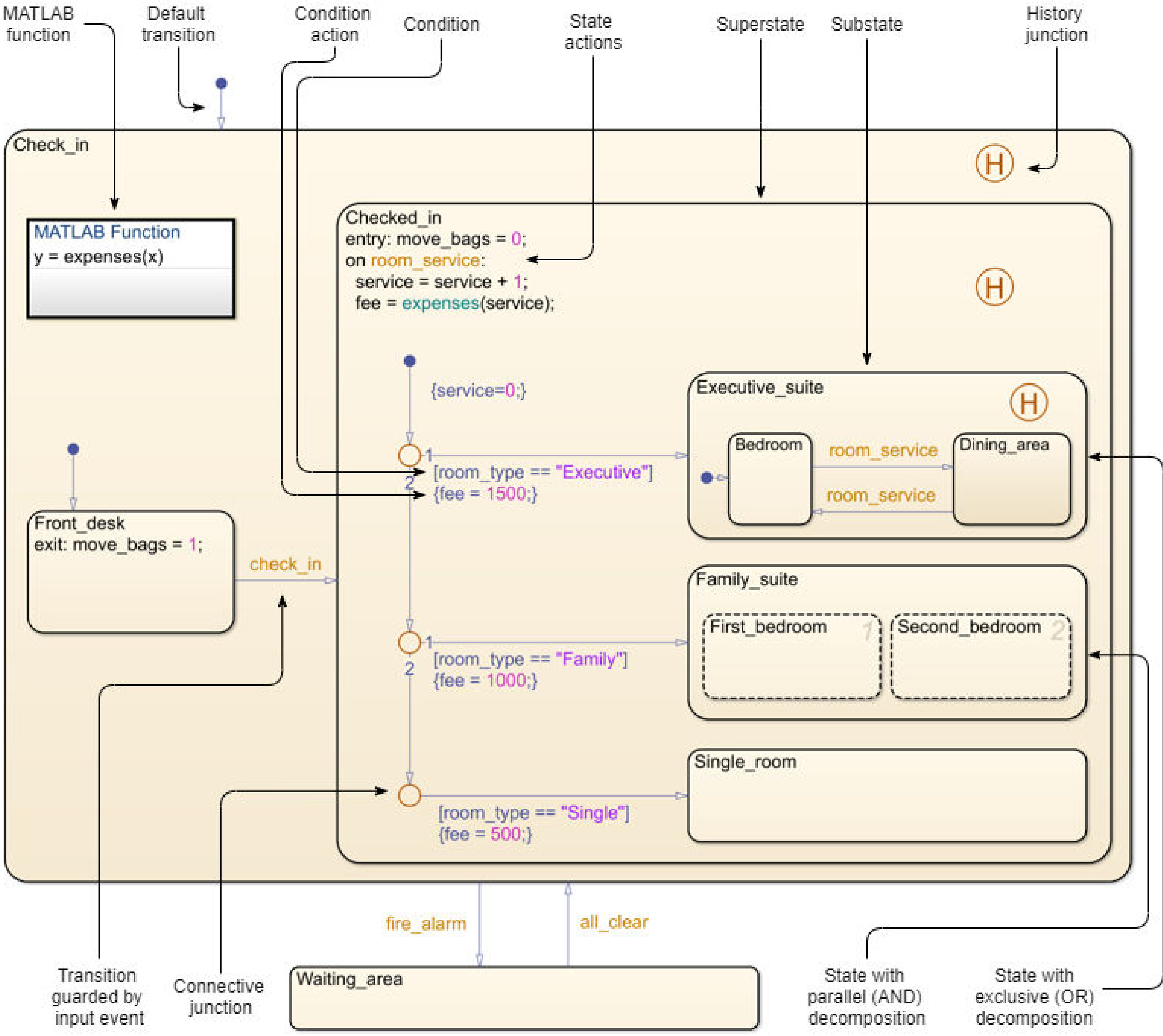
In Stateflow, semantics describe the execution behavior of your Stateflow chart. Various factors can affect how your chart executes, including:

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

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

Stateflow Objects

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



For more information on this example, see "How Stateflow Objects Interact During Execution" on page 1-8.

Graphical Objects

To build graphical objects, use the object palette in the Stateflow Editor (see "Stateflow Editor Operations" on page 25-2).

Graphical Objects	Types	References
Flow charts	Decision logic patterns	"Create Flow Charts in Stateflow" on page 3-2
	Loop logic patterns	

Graphical Objects	Types	References
Functions	Graphical functions	“Reuse Logic Patterns by Defining Graphical Functions” on page 6-9
	MATLAB functions	“Reuse MATLAB Code by Defining MATLAB Functions” on page 7-2
	Truth table functions	“Use Truth Tables to Model Combinatorial Logic” on page 8-2
	Simulink functions	“Reuse Simulink Functions in Stateflow Charts” on page 9-2
Junctions	Connective junctions	“Combine Transitions and Junctions to Create Branching Paths” on page 1-54
	History junctions	“Resume Prior Substate Activity by Using History Junctions” on page 1-58
States	States with exclusive (OR) decomposition	“Exclusive (OR) State Decomposition” on page 1-35
	States with parallel (AND) decomposition	“Parallel (AND) State Decomposition” on page 1-35
	Substates and superstates	“Create Substates and Superstates” on page 1-33
Transitions	Object-to-object transitions	“Transition Between Operating Modes” on page 1-37
	Default transitions	“Use Default Transitions to Specify Initial Substate Activity” on page 1-44
	Inner transitions	“Control Chart Execution by Using Inner Transitions” on page A-15
	Self-loop transitions	“Self-Loop Transition” on page A-23

Nongraphical Objects

You create nongraphical objects textually in your chart. See “Add Stateflow Data” on page 10-2, “Define Events in a Chart” on page 12-2, and “Define Messages in a Chart” on page 13-2 for details. Examples of nongraphical objects include:

Nongraphical Object	Description	Reference
Condition	Boolean expression that specifies that a transition path is valid if the expression is true; part of a transition label	“Define Actions in a Transition” on page 1-39 and “Conditions” on page 1-40
Condition action	Action that executes as soon as the condition evaluates to true; part of a transition label	“Define Actions in a Transition” on page 1-39 and “Condition Actions” on page 1-40
State actions	Expressions that specify actions to take when a state is active, such as initializing or updating data; part of a state label	“Define Actions in a State” on page 1-27

Nongraphical Object	Description	Reference
Function calls	Expression used to activate a specific function within a chart.	"Reuse MATLAB Code by Defining MATLAB Functions" on page 7-2 and "Reuse Simulink Functions in Stateflow Charts" on page 9-2
Temporal logic statements	Operators that are used to control chart actions.	"Control Chart Execution by Using Temporal Logic" on page 14-35

Enter a Chart

The set of default flow paths execute. If this action does not cause a state entry and the chart has parallel decomposition, then each parallel state becomes active.

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

Execute an Active Chart

If the chart has no states, each execution is equivalent to initializing a chart. Otherwise, the active children execute. Parallel states execute in the same order that they become active.

Enter a State

1. If the parent of the state is not active, perform steps 1 through 4 for the parent.
2. If this state is a parallel state, check that all siblings with a higher (that is, earlier) entry order are active. If not, perform steps 1 through 5 for these states first.

   Parallel (AND) states are ordered for entry based on whether you use explicit ordering (default) or implicit ordering.
3. Mark the state active.
4. Perform any entry actions.
5. Enter children, if needed:

   a. If the state contains a history junction and there was an active child of this state at some point after the most recent chart initialization, perform the entry actions for that child. Otherwise, execute the default flow paths for the state.
   b. If this state has children that are parallel states (parallel decomposition), perform entry steps 1 through 5 for each state according to its entry order.
   c. If this state has only one child substate, the substate becomes active when the parent becomes active, regardless of whether a default transition is present. Entering the parent state automatically makes the substate active. The presence of any inner transition has no effect on determining the active substate.
6. If this state is a parallel state, perform all entry steps for the sibling state next in entry order if one exists.
7. If the transition path parent is not the same as the parent of the current state, perform entry steps 6 and 7 for the immediate parent of this state.

Execute an Active State

1. The set of outer flow charts execute. If this action causes a state transition, execution stops. This step never occurs for parallel states.
2. During actions and valid on-event actions are performed.
3. The set of inner flow charts execute. If this action does not cause a state transition, the active children execute, starting at step 1. Parallel states execute in the same order that they become active.

Exit an Active State

1. If this is a parallel state, make sure that all sibling states that became active after this state have already become inactive. Otherwise, perform all exiting steps on those sibling states.
2. If there are any active children, perform the exit steps on these states in the reverse order that they became active.
3. Perform any exit actions.
4. Mark the state as inactive.

Execute a Set of Flow Charts

Flow charts execute by starting at step 1 below with a set of starting transitions. The starting transitions for inner flow charts are all transition segments that originate on the respective state and reside entirely within that state. The starting transitions for outer flow charts are all transition segments that originate on the respective state but reside at least partially outside that state. The starting transitions for default flow charts are all default transition segments that have starting points with the same parent:

1. Ordering of a set of transition segments occurs.
2. While there are remaining segments to test, testing a segment for validity occurs. If the segment is invalid, testing of the next segment occurs. If the segment is valid, execution depends on the destination:

   States

   a. Testing of transition segments stops and a transition path forms by backing up and including the transition segment from each preceding junction until the respective starting transition.
   b. The states that are the immediate children of the parent of the transition path exit.
   c. The transition action from the final transition path executes.
   d. The destination state becomes active.

   Junctions with no outgoing transition segments

   Testing stops without any state exits or entries.

   Junctions with outgoing transition segments

   Step 1 is repeated with the set of outgoing segments from the junction.
3. After testing all outgoing transition segments at a junction, backtrack the incoming transition segment that brought you to the junction and continue at step 2, starting with the next transition

segment after the backtrack segment. The set of flow charts finishes execution when testing of all starting transitions is complete.

Execute an Event Broadcast

Output edge-trigger event execution is equivalent to changing the value of an output data value. All other events have the following execution:

1. If the *receiver* of the event is active, then it executes. The event *receiver* is the parent of the event unless a direct event broadcast occurs using the `send()` function.

   If the receiver of the event is not active, nothing happens.
2. After broadcasting the event, the broadcaster performs early return logic based on the type of action statement that caused the event.

Action Type	Early Return Logic
State Entry	If the state is no longer active at the end of the event broadcast, any remaining steps in entering a state do not occur.
State Exit	If the state is no longer active at the end of the event broadcast, any remaining exit actions and steps in state transitioning do not occur.
State During	If the state is no longer active at the end of the event broadcast, any remaining steps in executing an active state do not occur.
Condition	If the origin state of the inner or outer flow chart or parent state of the default flow chart is no longer active at the end of the event broadcast, the remaining steps in the execution of the set of flow charts do not occur.
Transition	If the parent of the transition path is not active or if that parent has an active child, the remaining transition actions and state entry do not occur.

See Also

More About

- “Graphical Objects” on page 1-5
- “Nongraphical Objects” on page 1-6
- “Types of Chart Execution” on page 2-10

Modeling Guidelines for Stateflow Charts

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

Use signals of the same data type for input events

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

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 transitions

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

Verify intended backtracking behavior in flow charts

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

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

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

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

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

Use MATLAB functions for performing numerical computations in a chart

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

Use descriptive names in function signatures

Descriptive function names enhance readability of chart objects.

Use history junctions to record state history

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

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

This guideline prevents compile-time 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.

Types of Chart Execution

Life Cycle of a Stateflow Chart

Stateflow charts go through several stages of execution:

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

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

See “Use Events to Execute Charts” on page 2-40.

Execution of an Inactive Chart

When a chart is inactive and first triggered by an event from a Simulink model, it first executes its set of default flow charts (see “Order of Execution for a Set of Flow Charts” on page 2-44). If this action does not cause an entry into a state and the chart has parallel decomposition, then each parallel state becomes active (see “Enter a Chart or State” on page 2-17).

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

Execution of an Active Chart

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

Execution of a Chart at Initialization

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

If you want your chart to begin executing from a known configuration, you can enable the **Execute (enter) Chart At Initialization** chart property. When you turn on this option, the state configuration of a chart initializes at time 0 instead of the first occurrence of an input event. The default transition paths of the chart execute during the model initialization phase at time 0, corresponding to the mdlInitializeConditions() (Simulink) phase for S-functions. For more information, see “Execute (enter) chart at initialization” on page 1-21.

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.

To control the level of diagnostic action for invalid access to chart input data, open the Configuration Parameters dialog box and, in the **Diagnostics > Stateflow** pane, set the **Invalid input data access in chart initialization** parameter to `none`, `warning`, or `error`. The default setting is `warning`. For more information, see "Invalid input data access in chart initialization" (Simulink).

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

See Also

More About

- "Execution of a Stateflow Chart" on page 2-12
- "Exit a State" on page 2-23
- "Evaluate Transitions" on page 2-26

Execution of a Stateflow Chart

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

- At each time step
- When the Stateflow chart receives an event

When a chart wakes up for the first time, the chart is initialized and becomes active, as described in "Workflow for Entering a Chart or State" on page 2-17. When there are no more actions to take, the chart goes to sleep until a new time step or event wakes up the chart.

Workflow for Stateflow Chart Execution

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

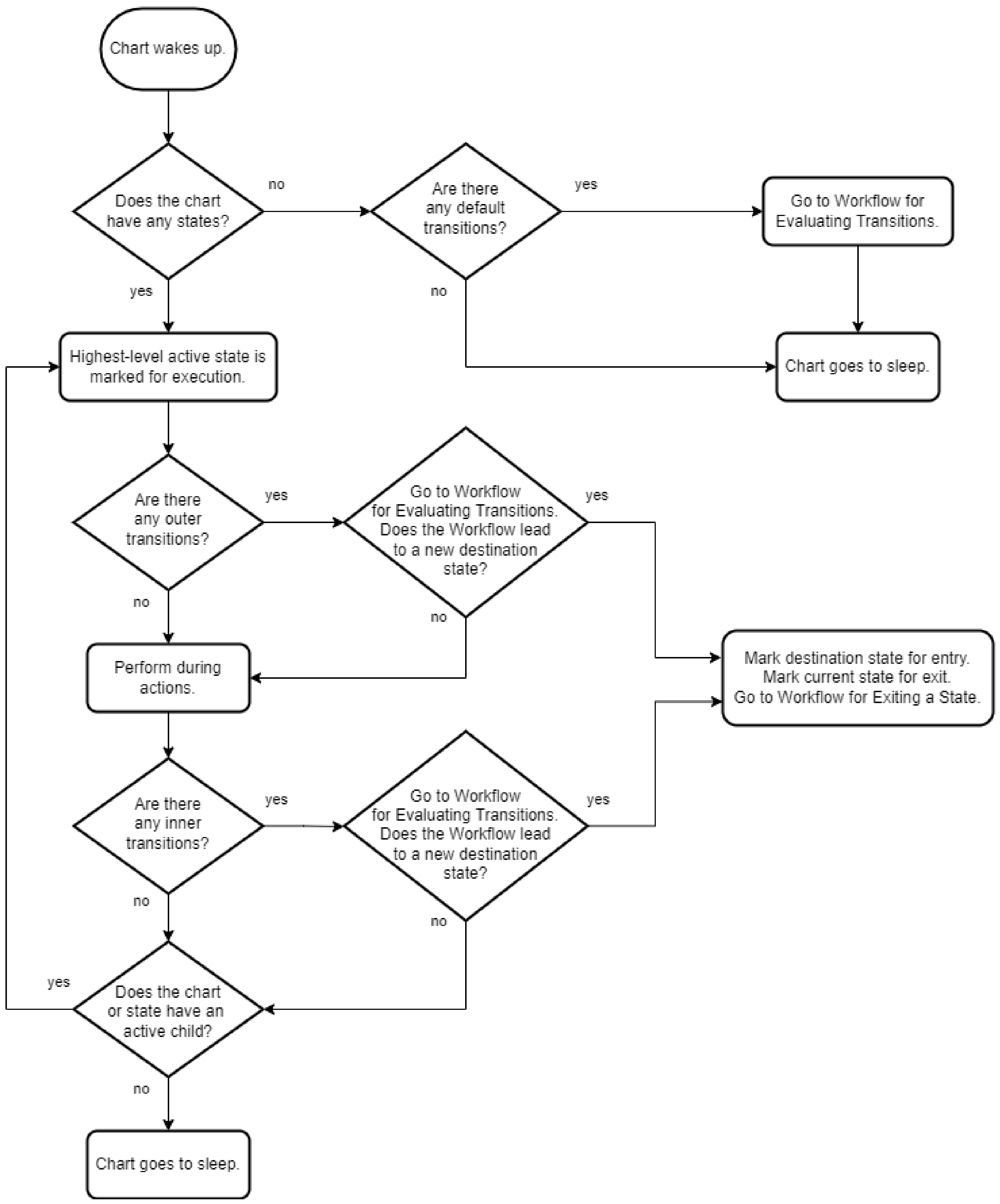


Chart wakes up.
Does the chart have any states?
no
yes
Are there any default transitions?
yes
no
Go to Workflow for Evaluating Transitions.
Chart goes to sleep.
Highest-level active state is marked for execution.
Are there any outer transitions?
yes
no
Go to Workflow for Evaluating Transitions. Does the Workflow lead to a new destination state?
yes
no
Mark destination state for entry. Mark current state for exit. Go to Workflow for Exiting a State.
Perform during actions.
Are there any inner transitions?
yes
no
Go to Workflow for Evaluating Transitions. Does the Workflow lead to a new destination state?
yes
no
yes
Does the chart or state have an active child?
no
Chart goes to sleep.

Default Transitions

A default transition is a transition that has no source. In a Stateflow chart that does not contain any states, a default transition marks the start of a flow chart. For more information, see “Create Flow Charts in Stateflow” on page 3-2.

If a Stateflow chart does not contain any states, the chart evaluates the default transition paths each time the chart wakes up. After marking a default transition for evaluation, the chart follows the steps shown in “Workflow for Evaluating Transitions” on page 2-27.

Note Charts evaluate default transition paths inside a state only during state entry, and not each time the chart wakes up.

Outer Transition

An outer transition is a transition that exits the source state. A Stateflow chart marks outer transitions for evaluation as the first step in executing a state. After marking an outer transition for evaluation, the chart follows the steps shown in “Workflow for Evaluating Transitions” on page 2-27.

During Actions

A `during` action is an action defined in a state label actions by using the prefix `during` or `du`. For more information, see “Define Actions in a State” on page 1-27.

A state performs `during` actions when the chart wakes up, the state is active, and there are no valid outer transitions.

Inner Transitions

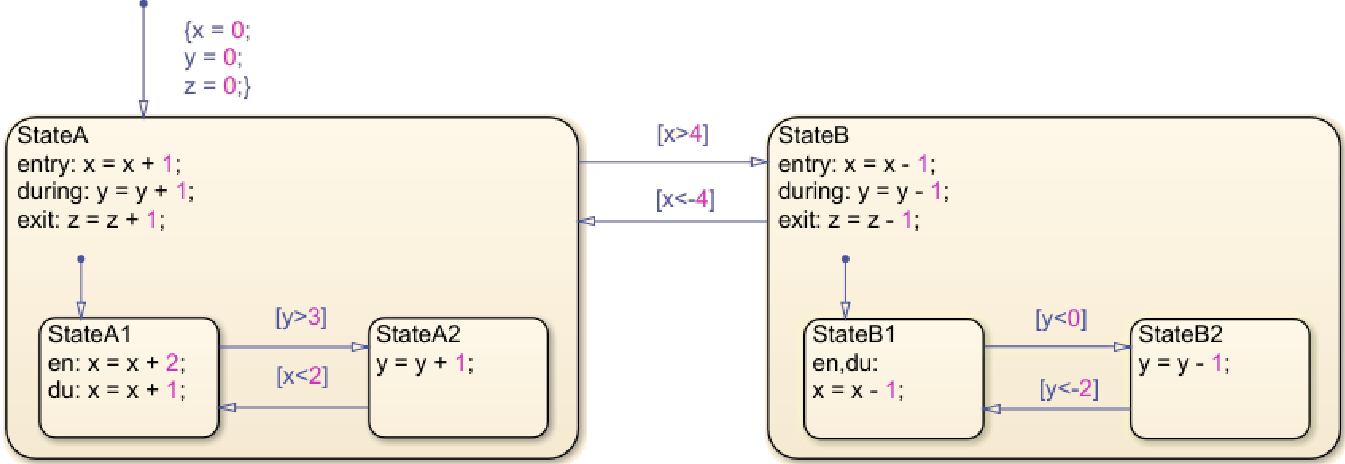
An inner transition is a transition that does not exit the source state. For more information, see “Control Chart Execution by Using Inner Transitions” on page A-15.

A Stateflow chart marks inner transitions for evaluation after a state performs `during` actions. After marking an inner transition for evaluation, the chart follows the steps shown in “Workflow for Evaluating Transitions” on page 2-27.

Chart Execution with a Valid Transition

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

At this time step, `x = 5`, `y = 2`, and `z = 0`.



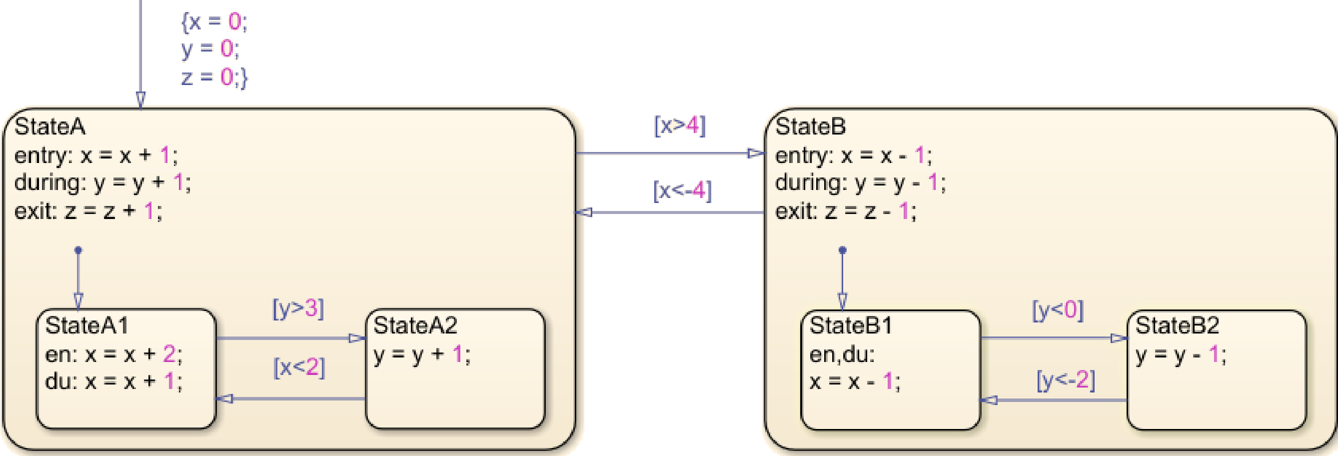
The chart executes these steps:

1. The chart has an active substate, `StateA`.
2. `StateA` has an outer transition to `StateB`. The chart determines that the transition is valid.
3. The chart marks `StateB` for entry and `StateA` is marked for exit.
4. To exit `StateA`, the chart follows the steps shown in “Workflow for Exiting a State” on page 2-23.
5. To enter `StateB`, the chart follows the steps shown in “Workflow for Entering a Chart or State” on page 2-17

Chart Execution Without a Valid Transition

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

At this time step, `x = 3`, `y = 0`, and `z = 0`.



The chart executes these steps:

1. The chart has an active substate, `StateA`.
2. `StateA` has an outer transition to `StateB`. The chart determines that the transition is invalid.
3. The chart performs the `during` actions for `StateA`. Now `y = 1`.
4. `StateA` does not have any inner transitions.
5. The active substate of `StateA` is `StateA1`.
6. `StateA1` has an outer transition to `StateA2`. The chart determines that the transition is invalid.
7. The chart performs the `during` actions for `StateA1`. Now `x = 4`.
8. `StateA1` does not have any active substates.
9. The chart goes to sleep.

See Also

More About

- “Enter a Chart or State” on page 2-17
- “Exit a State” on page 2-23
- “Evaluate Transitions” on page 2-26

Enter a Chart or State

Chart and state entry occurs when:

- A chart is activated for the first time. This is called chart initialization.
- A valid transition into a state exists. See "Evaluate Transitions" on page 2-26.

Workflow for Entering a Chart or State

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

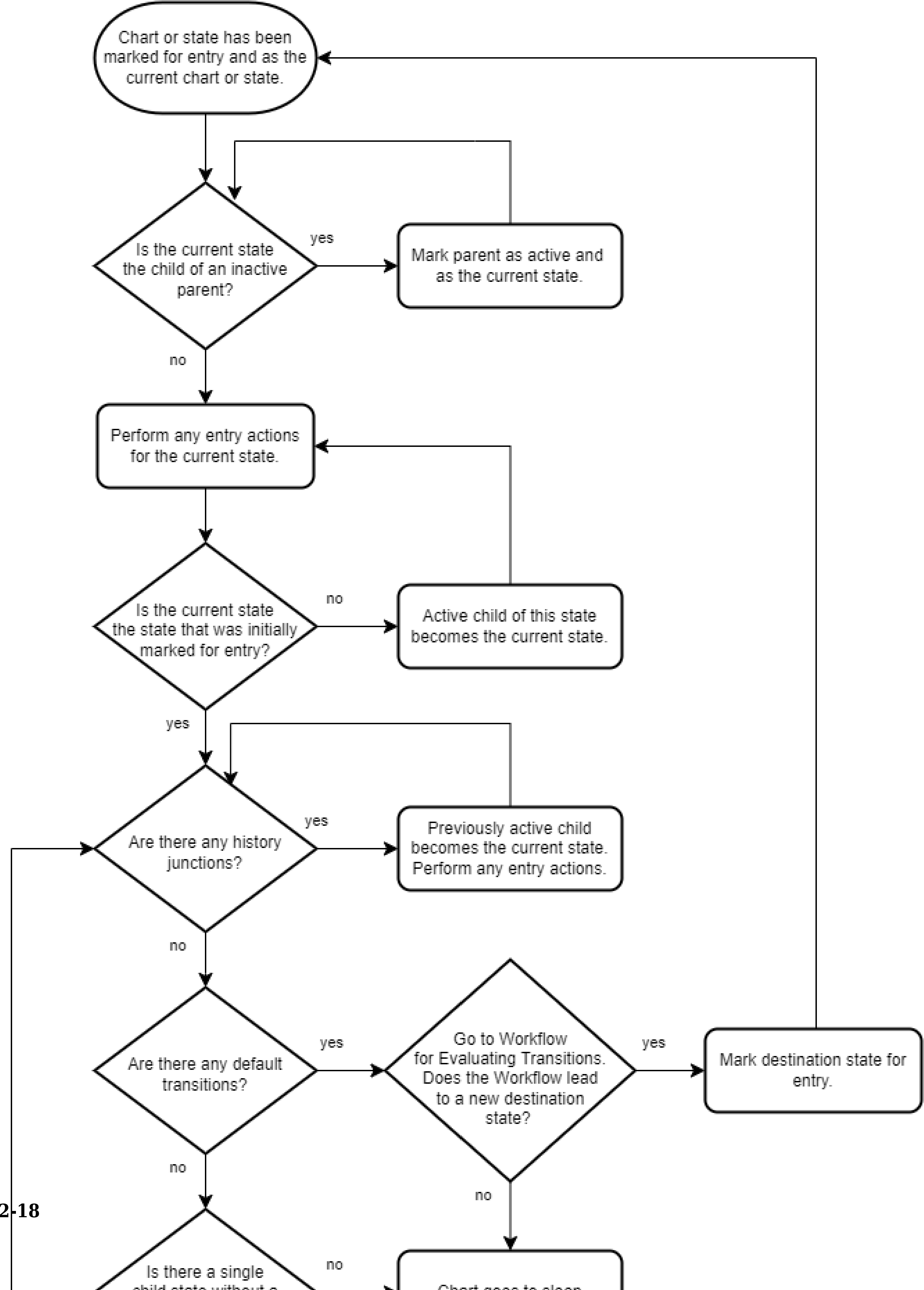


Chart or state has been marked for entry and as the current chart or state.
Is the current state the child of an inactive parent?
yes
Mark parent as active and as the current state.
no
Perform any entry actions for the current state.
Is the current state the state that was initially marked for entry?
no
Active child of this state becomes the current state.
yes
Are there any history junctions?
yes
Previously active child becomes the current state. Perform any entry actions.
no
Are there any default transitions?
yes
Go to Workflow for Evaluating Transitions. Does the Workflow lead to a new destination state?
yes
Mark destination state for entry.
no
no
Is there a single
no

Chart Entry

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

If you want your chart to take any default transitions before time `t = 0`, in the Chart Properties dialog box, select the **Execute (enter) chart at initialization** check box. This option causes the Stateflow chart to initialize at the same time as Simulink initialization. The default transition paths of the chart then execute during the model initialization phase.

State Entry

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

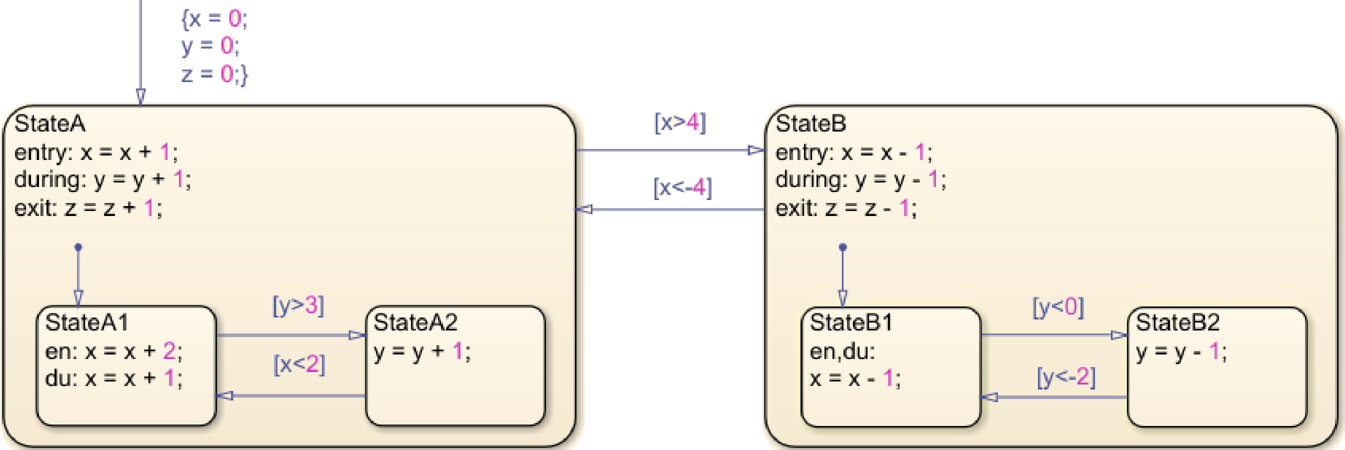
- An incoming transition crosses state boundaries.
- An incoming transition ends at the state boundary.
- The state is a parallel state child of an active state.

Entry Actions

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

Enter a Stateflow Chart

In this example, the first time the chart becomes active, chart initialization occurs.



By following the “Workflow for Entering a Chart or State” on page 2-17 until the chart goes to sleep, the steps for chart initialization are in this order:

1. The default transition actions are executed, and x = 0, y = 0, and z = 0.
2. StateA is marked for entry.
3. StateA is not a substate of an inactive parent. Perform the entry actions for StateA. Now x = 1.
4. StateA is the state that was initially marked for entry.
5. StateA does not contain any history junctions.
6. There is a default transition to the substate, StateA1. Go to the Evaluate Transitions flow chart.
7. By following the Evaluate Transitions flow chart, mark StateA1 for entry. Go to the Exit Actions flow chart.
8. The current state, StateA, is a superstate of the destination state, StateA1. Return to the Entry Actions flow chart.
9. StateA1 is not a substate of an inactive parent. Perform entry actions for StateA1. Now x = 3.
10. StateA1 is the state that was initially marked for entry.
11. StateA1 does not contain any history junctions.
12. StateA1 does not contain any default transitions.
13. StateA1 does not contain any single substates.
14. The chart goes to sleep.

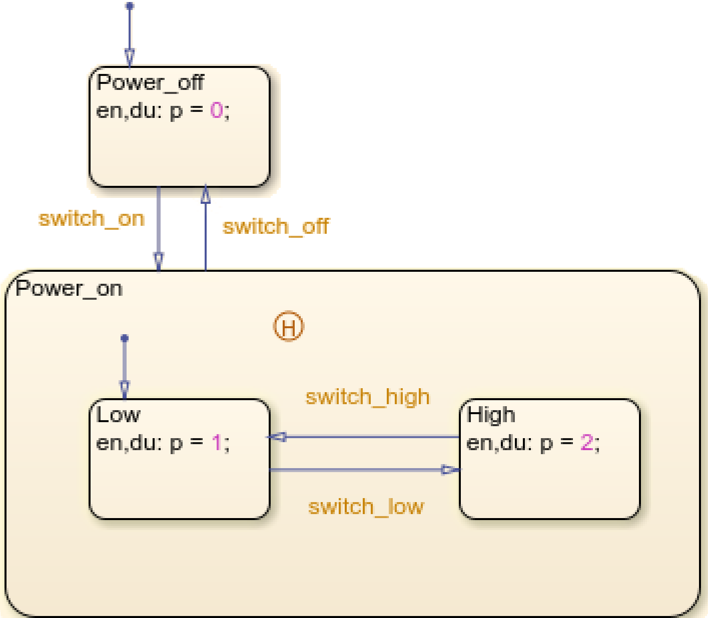
Steps 1 through 14 take place in the initial time step. This completes the chart initialization process.

Entering a State by Using History Junctions

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

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

Initially, the chart is asleep. The state Power_off is active. When the state Power_on was last active, High was the previously active substate. The event switch_on occurs and the state Power_on is marked for entry. At this time p = 0.



By following the “Workflow for Entering a Chart or State” on page 2-17 until the chart goes to sleep, the execution steps for entering the state `Power_on` are in this order:

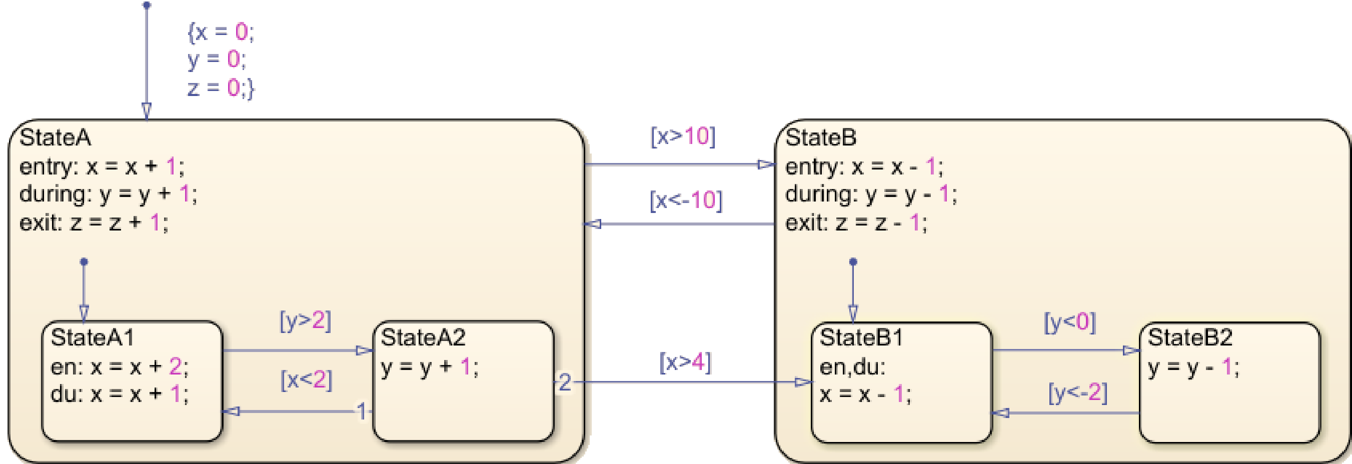
1. `Power_on` is not the child of an inactive parent.
2. There are no entry actions for `Power_on`.
3. `Power_on` is the state that was initially marked for entry.
4. There are history junctions in `Power_on`.
5. `High` was the previously active substate. Now `p = 2`.
6. `High` does not contain any history junctions.
7. `High` does not contain any default transitions.
8. `High` does not contain any single substates.
9. The chart goes to sleep.

This completes the `entry` actions for `Power_on` and `High`.

Entering a State by Using Supertransitions

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

When a state is entered through a supertransition, before the entry actions for the final destination are executed, its superstates must be marked active and their entry actions must be executed. In this example, `StateB1` has been marked for entry from `StateA2`. At this point, `x = 5`, `y = 5`, and `z = 1`.



By following the "Workflow for Entering a Chart or State" on page 2-17 until the chart goes to sleep, the execution steps for entering the state `StateB1` are in this order:

1. `StateB1` is the substate of an inactive parent (`StateB`).
2. `StateB` is marked as active.
3. `StateB` is not the substate of an inactive parent.
4. Perform the entry actions for `StateB`. Now `x = 4`.
5. `StateB` is not the state that was initially marked for entry.
6. Perform the entry actions for `StateB1`. Now `x = 3`.
7. `StateB1` is the state that was initially marked for entry.
8. `StateB1` has no history junctions.
9. `StateB1` does not contain any default transitions.
10. `StateB1` does not contain any single substates.
11. The chart goes to sleep.

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

See Also

More About

- "Execution of a Stateflow Chart" on page 2-12
- "Exit a State" on page 2-23
- "Evaluate Transitions" on page 2-26

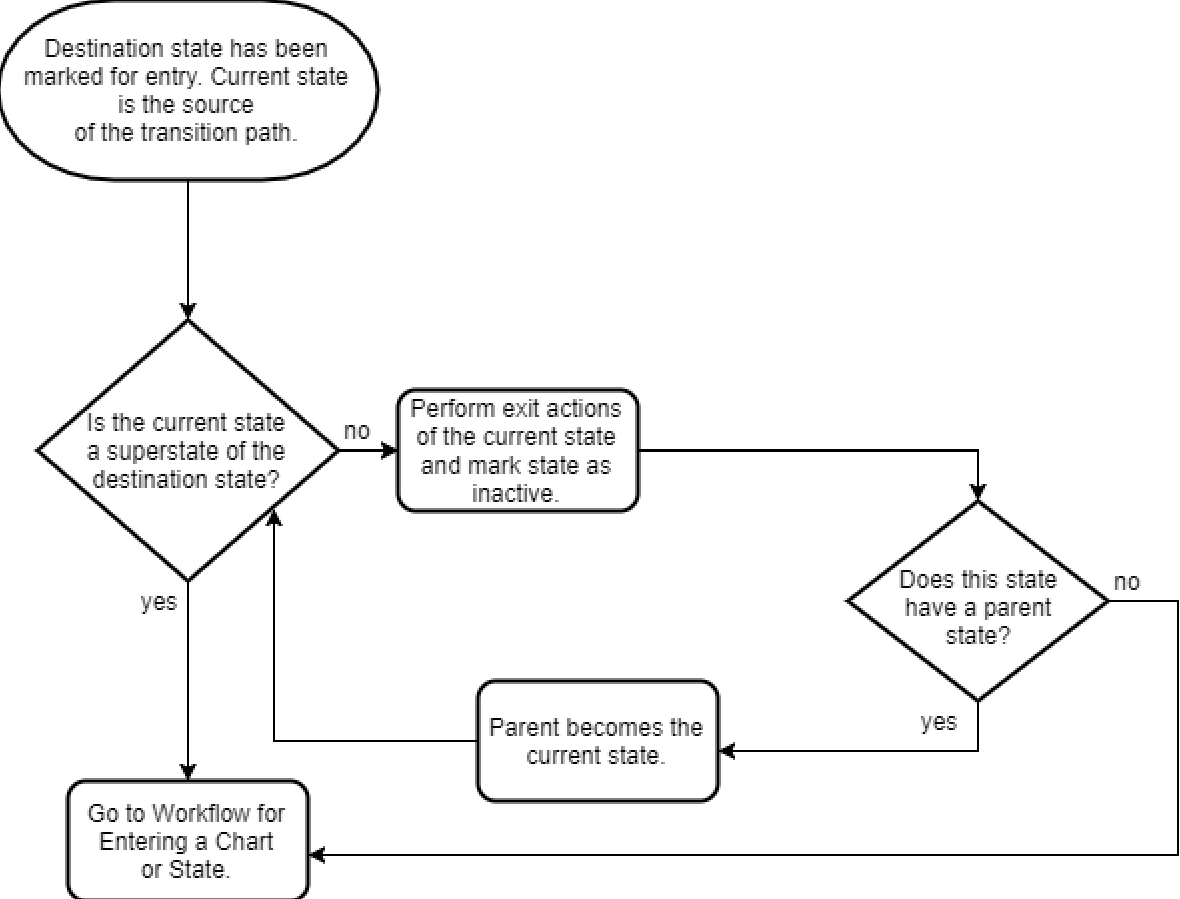
Exit a State

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

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

Workflow for Exiting a State

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



Exit Actions

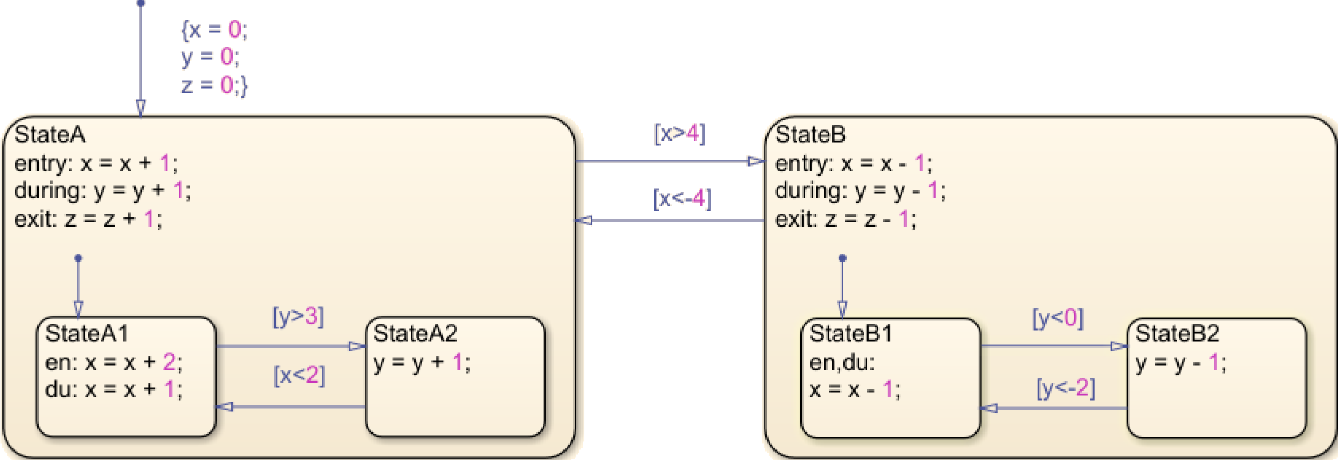
`Exit` actions for a state execute when the state is active and a valid transition from the state exists. A state performs its `exit` actions before becoming inactive.

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

Exit a State Example

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

By following the "Workflow for Stateflow Chart Execution" on page 2-12 and the "Workflow for Evaluating Transitions" on page 2-27, `StateB` has been marked for entry. `StateA` is the source of the transition. At this time step `x = 5`, `y = 2`, and `z = 0`.



By following the flow chart for state exit actions until the chart goes to sleep, the execution steps for this chart are in this order:

1. `StateA` is not a superstate of `StateB`.
2. Perform the exit actions of `StateA` and mark `StateA` as inactive. Now `z = 1`.
3. `StateA` does not have a parent state.
4. Go to "Entry Actions" on page 2-19.

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

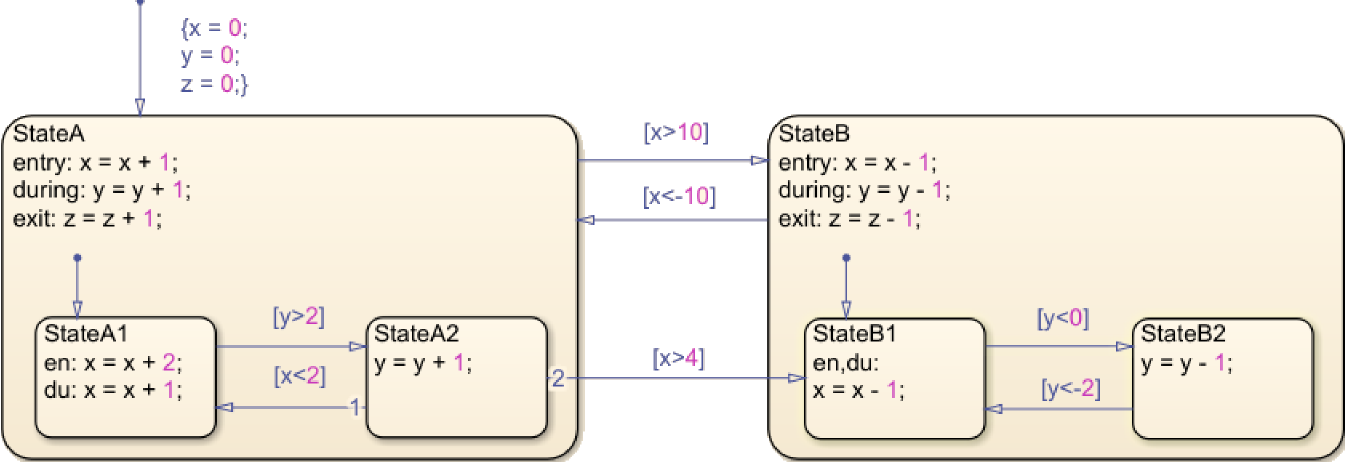
Perform the "Workflow for Entering a Chart or State" on page 2-17 for `StateB` to complete the time step.

Exit a State by Using Supertransitions

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

When a state is exited through a supertransition, after the exit actions for the source of the transition are executed, its superstates are marked inactive and exit actions of the superstates are executed. In

this example, StateA2 is marked for exit and StateB1 is marked for entry. At this point, x = 5, y = 5, and z = 0.



By following the “Workflow for Entering a Chart or State” on page 2-17 until the chart goes to sleep, the execution steps for exiting the state StateA2 are in this order:

1. StateA2 is not a superstate of the destination state (StateB1).
2. Perform the exit actions for StateA2 and mark StateA2 as inactive.
3. StateA2 does have a parent state, StateA.
4. StateA is not a superstate of the destination state (StateB1).
5. Perform the exit actions for StateA, and mark StateA as inactive.
6. StateA does not have a parent state.

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

Perform the “Workflow for Entering a Chart or State” on page 2-17 for StateB and StateB1 to complete the time step.

See Also

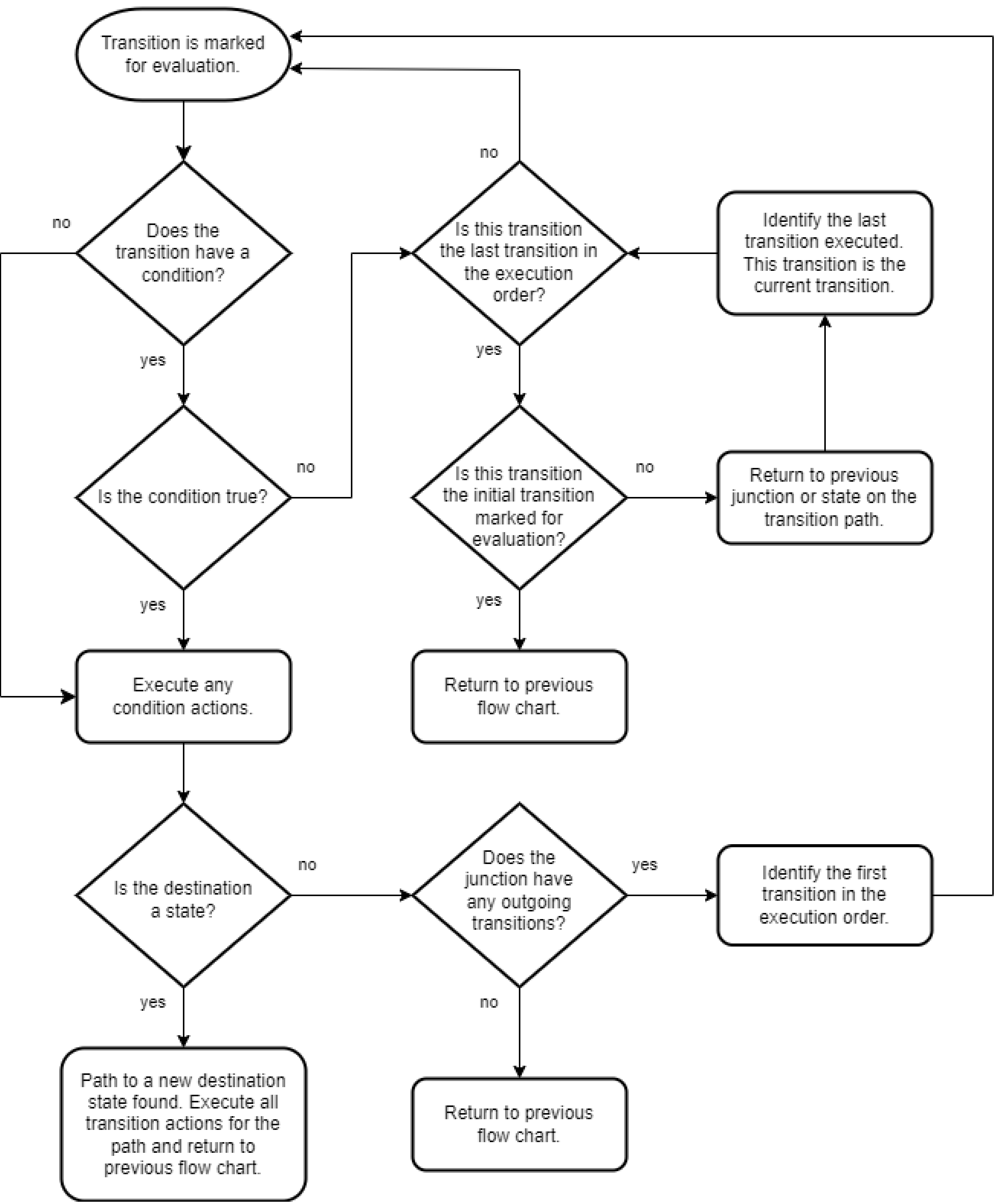
More About

- “Execution of a Stateflow Chart” on page 2-12
- “Enter a Chart or State” on page 2-17
- “Evaluate Transitions” on page 2-26

Evaluate Transitions

Stateflow uses transitions in charts to move from one exclusive (OR) state to another exclusive (OR) state. For the `entry` and `execution` workflows of chart execution, Stateflow evaluates transitions to determine if they are valid. A valid transition is a transition whose condition labels are true and whose path ends at a state. If a transition is valid, Stateflow exits from the source state and enters the destination state. To learn about when evaluation occurs during the `execution` and `entry` workflows, see “Execution of a Stateflow Chart” on page 2-12 and “Enter a Chart or State” on page 2-17.

Workflow for Evaluating Transitions



Transition Evaluation Order

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

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

Explicit Ordering

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

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

Implicit Ordering

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

- Hierarchy.

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

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

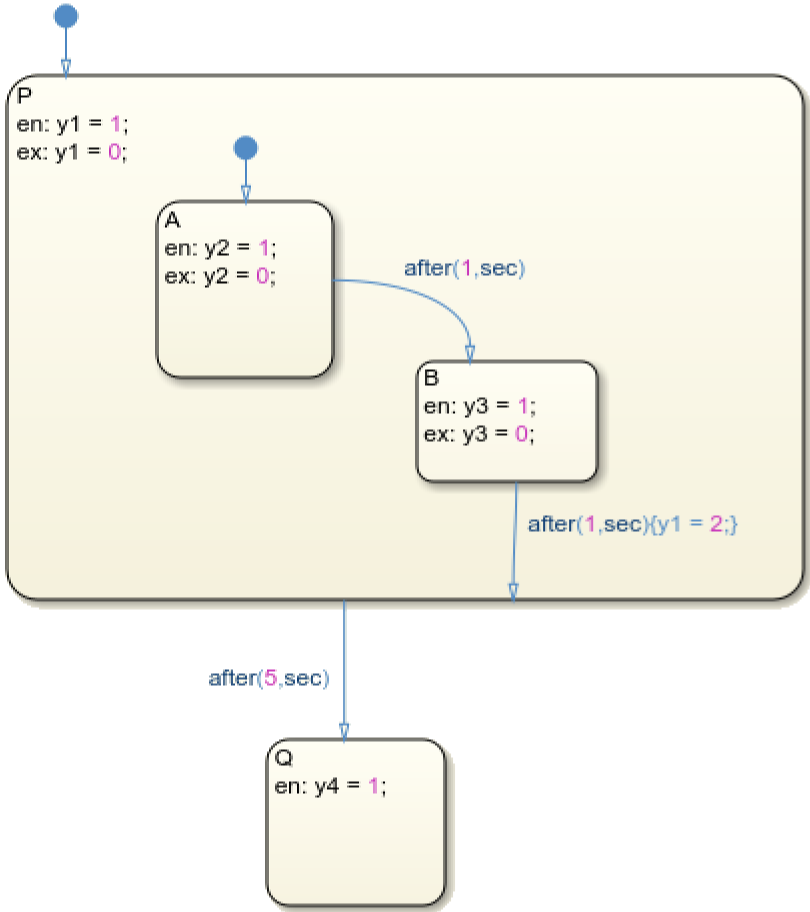
  1. Labels with events and conditions
  2. Labels with events
  3. Labels with conditions
  4. No label

  For more information about chart behavior when using events, see “Control Chart Behavior by Using Implicit Events” on page 12-28.
- Angular surface position of transition source.

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

Transition to the Inner Edge of a Parent State

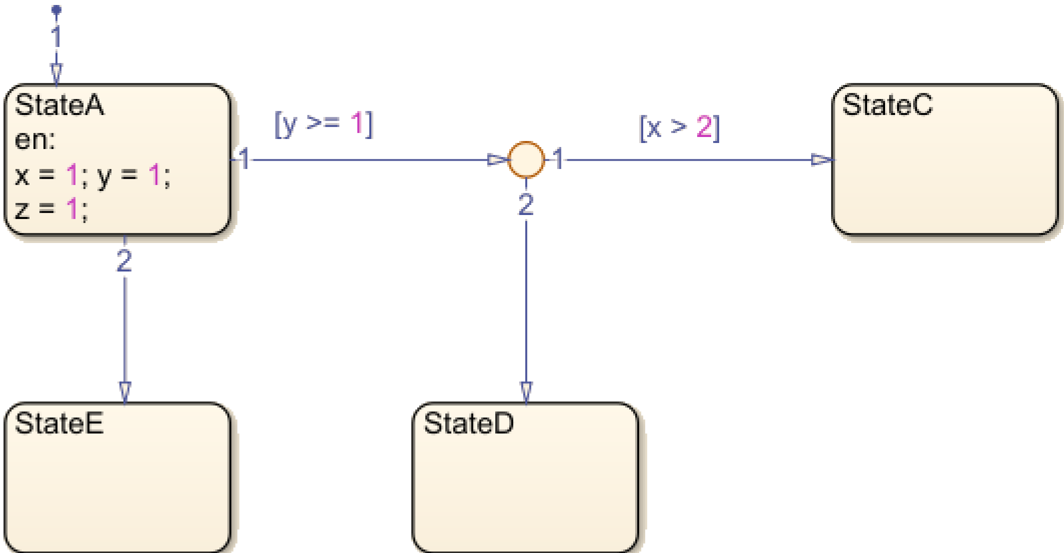
Transitions that end on the inside edge of a parent state are a shortcut back to the default transition path, and the default path is evaluated during the current time step. In this example, the transition from state B leads immediately to the default transition to state A.



If there are default transitions, then Stateflow immediately executes those paths. If not, and there are no children, then that is the end of the time step. In both cases, the parent remains active, and exit and entry actions of the parent are not executed.

Evaluate Outer Transition

In this example, the Stateflow chart is initialized and the `entry` actions are performed for `StateA`. A new time step occurs and the chart wakes up. By following the "Workflow for Stateflow Chart Execution" on page 2-12, Stateflow finds multiple outer transition paths from `StateA`. At this time step `x = 1`, `y = 1`, and `z = 1`.



By following the “Workflow for Evaluating Transitions” on page 2-27, the steps for evaluating the transitions of this chart are in this order:

1. Transition 1 from `StateA` is marked for evaluation.
2. Transition 1 from `StateA` has a condition.
3. The condition is true.
4. The destination of transition 1 from `StateA` is not a state.
5. The junction does have outgoing transitions.
6. Transition 1 from the junction is marked for evaluation.
7. Transition 1 from the junction has a condition.
8. The condition is false.
9. Transition 2 from the junction is marked for evaluation.
10. Transition 2 from the junction does not have a condition.
11. The destination of transition 2 from the junction is a state (`StateD`).
12. `StateD` is marked for `entry`, and `StateA` is marked for `exit`.

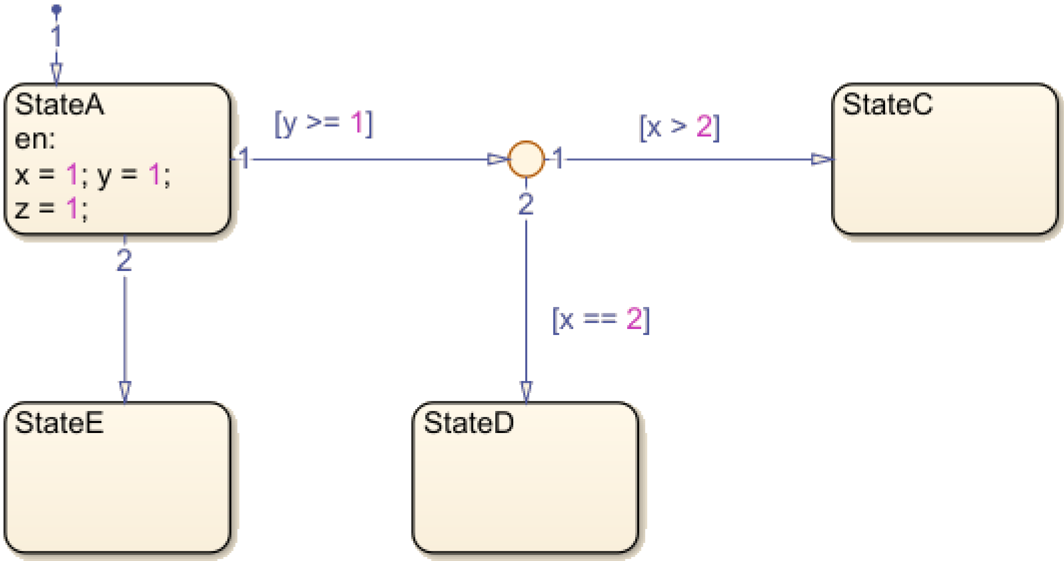
To complete the time step, follow the “Workflow for Exiting a State” on page 2-23 for `StateA` and the “Workflow for Entering a Chart or State” on page 2-17 for `StateE`.

Evaluate Outer Transition with Backtracking

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

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

Execution” on page 2-12, Stateflow finds multiple outer transition paths from `StateA`. At this time step `x = 1`, `y = 1`, and `z = 1`.



By following the “Workflow for Evaluating Transitions” on page 2-27, the steps for evaluating the transitions of this chart are in this order:

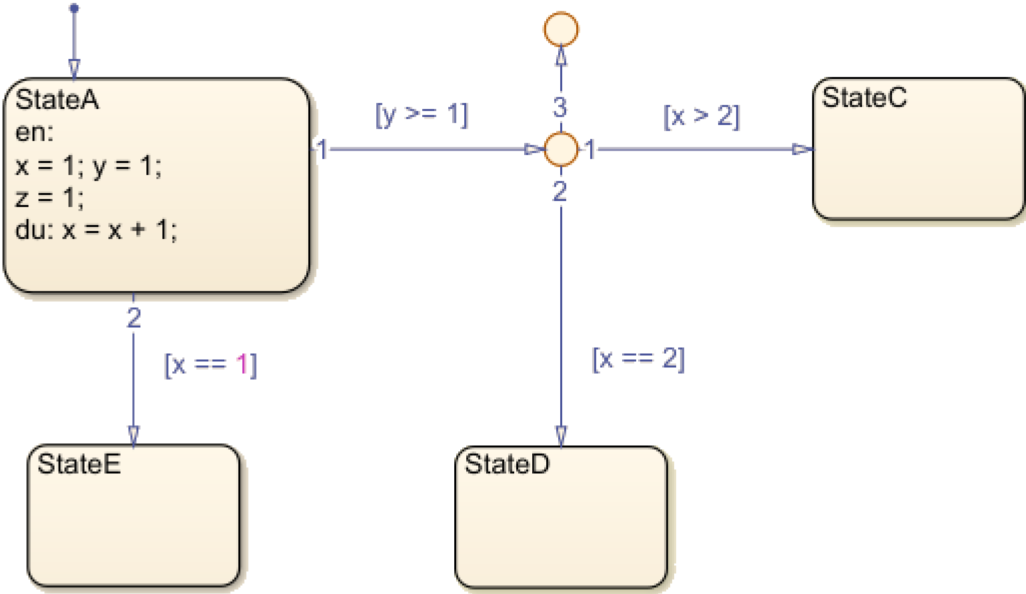
1. Transition 1 from `StateA` is marked for evaluation.
2. Transition 1 from `StateA` has a condition.
3. The condition is true.
4. The destination of transition 1 from `StateA` is not a state.
5. The junction does have outgoing transitions.
6. Transition 1 from the junction is marked for evaluation.
7. Transition 1 from the junction has a condition.
8. The condition is false.
9. Transition 2 from the junction is marked for evaluation.
10. Transition 2 from the junction has a condition.
11. The condition is false.
12. Transition 2 from `StateA` is marked for evaluation.
13. Transition 2 from `StateA` does not have a condition.
14. The destination of transition 2 from `StateA` is a state (`StateE`).
15. `StateE` is marked for `entry`, and `StateA` is marked for `exit`.

To complete the time step, follow the “Workflow for Exiting a State” on page 2-23 for `StateA` and the “Workflow for Entering a Chart or State” on page 2-17 for `StateE`.

Prevent Backtracking

In this example, a terminating junction prevents backtracking. The Stateflow chart is initialized and the `entry` actions are performed for `StateA`. A new time step occurs and the chart wakes up. By

following the “Workflow for Stateflow Chart Execution” on page 2-12, Stateflow finds multiple outer transition paths from `StateA`. At this time step `x = 1`, `y = 1`, and `z = 1`.



By following the “Workflow for Evaluating Transitions” on page 2-27, the steps for evaluating the transitions of this chart are in this order:

1. Transition 1 from `StateA` is marked for evaluation.
2. Transition 1 from `StateA` has a condition.
3. The condition is true.
4. The destination of transition 1 from `StateA` is not a state.
5. The junction does have outgoing transitions.
6. Transition 1 from the junction is marked for evaluation.
7. Transition 1 from the junction has a condition.
8. The condition is false.
9. Transition 2 from the junction is marked for evaluation.
10. Transition 2 from the junction has a condition.
11. The condition is false.
12. Transition 3 from the junction is marked for evaluation.
13. Transition 3 from the junction does not have a condition.
14. The destination is not a state and does not have any outgoing transitions.
15. Return to “Workflow for Stateflow Chart Execution” on page 2-12.

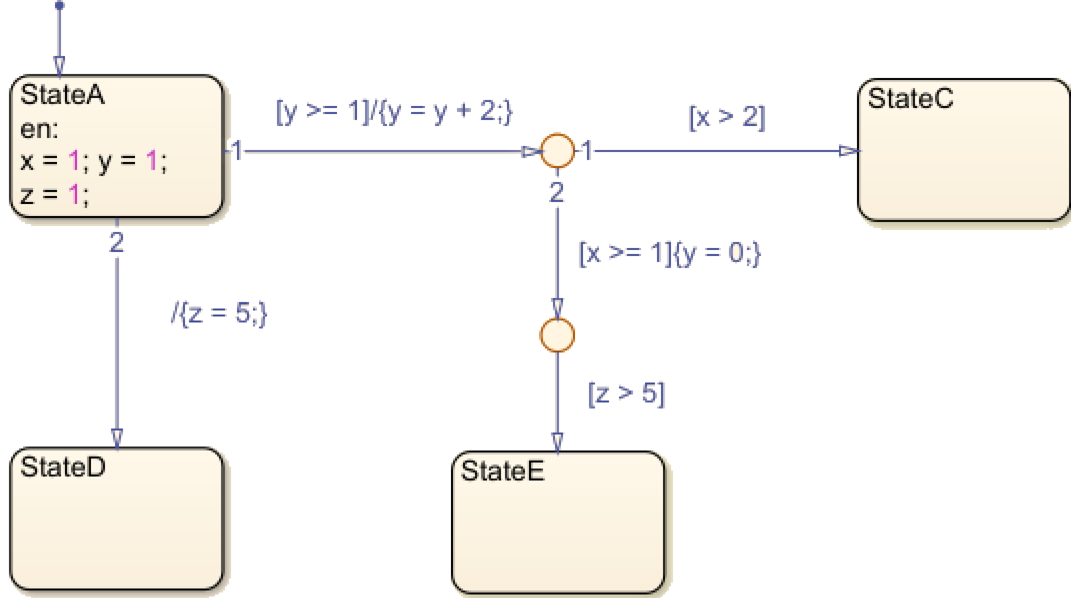
To complete the time step, follow the “Workflow for Stateflow Chart Execution” on page 2-12 for `StateA`, starting where you left off.

Evaluate Outer Transitions with Condition and Transition Actions

This example contains both condition actions and transition actions:

- In transition label syntax, condition actions follow the transition condition and are enclosed in curly braces ({}). Condition actions are executed when the condition is evaluated as true but before the transition path has been determined to be valid.
- In transition label syntax, transition actions are preceded with a forward slash (/) and are enclosed in curly braces ({}). Transition actions execute only after the transition path is determined to be valid.

The Stateflow chart is initialized and the `entry` actions are performed for `StateA`. A new time step occurs and the chart wakes up. There are multiple outer transition paths from `StateA`. At this time step `x = 1`, `y = 1`, and `z = 1`.



By following the “Workflow for Evaluating Transitions” on page 2-27, the steps for evaluating the transitions of this chart are in this order:

1. Transition 1 from `StateA` is marked for evaluation.
2. Transition 1 from `StateA` has a condition (`[y >= 1]`).
3. The condition is true.
4. There are no condition actions.
5. The destination of transition 1 from `StateA` is not a state.
6. The junction does have outgoing transitions.
7. Transition 1 from the junction is marked for evaluation.
8. Transition 1 from the junction has a condition (`[x > 2]`).
9. The condition is false.
10. Transition 2 from the junction is marked for evaluation.
11. Transition 2 from the junction has a condition (`[x >= 1]`).
12. The condition is true.
13. There is a condition action (`{y = 0;}`). Now `y = 0`.

14. The junction does have outgoing transitions.
15. The transition from the junction is marked for evaluation.
16. Transition 1 from the junction has a condition (`[z >= 5]`).
17. The condition is false.
18. Transition 2 from `StateA` is marked for evaluation.
19. Transition 2 from `StateA` does not have a condition.
20. The destination of transition 2 from `StateA` is a state (`StateD`).
21. `StateD` is marked for `entry`, and `StateA` is marked for `exit`. Execute the transition action for this valid path (`/{z = 5}`). Now `z = 5`.

To complete the time step, follow the "Workflow for Exiting a State" on page 2-23 for `StateA` and the "Workflow for Entering a Chart or State" on page 2-17 for `StateE`.

See Also

More About

- "Execution of a Stateflow Chart" on page 2-12
- "Enter a Chart or State" on page 2-17
- "Exit a State" on page 2-23

Super Step Semantics

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

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, so the chart is in a stable active state configuration.
- The number of transitions taken exceeds a user-specified maximum number of iterations.

For simulation targets, you can specify whether the chart goes to the next time step or generates an error if it reaches the maximum number of iterations prematurely. In generated code for embedded targets, the chart always goes to the next time step after reaching the maximum number of iterations.

Super step semantics are not supported in standalone Stateflow charts in MATLAB.

Maximum Number of Iterations

In a super step, your chart always takes at least one transition. Therefore, when you set a maximum number of iterations in each super step, the chart takes that number of transitions plus 1. For example, if you specify 10 as the maximum number of iterations, your chart takes 11 transitions in each super step.

Tip 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 your chart by setting the maximum number of iterations that the chart takes per time step.

Enable Super Step Semantics

To enable super step semantics:

1. Select the **Enable super step semantics** chart property, as described in “Specify Properties for Stateflow Charts” on page 1-19.
2. Enter a value for **Maximum iterations in each super step**.

   The chart always takes one transition during a super step, so the value `N` that you specify represents the maximum number of *additional* transitions (for a total of `N+1`). Try to choose a number that allows the chart to reach a stable state within the time step, based on the mode logic of your chart.
3. Select an action for **Behavior after too many iterations**.

   Your selection determines how the chart behaves during simulation after it reaches the maximum number of iterations in a time step.

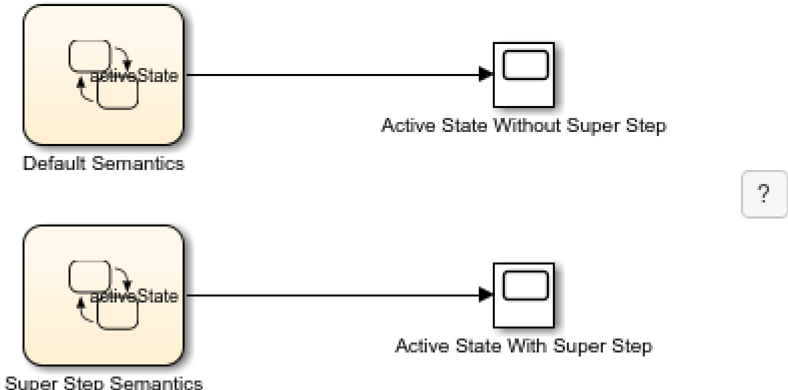
Behavior	Description
`Proceed`	Chart execution continues to the next time step.

Behavior	Description
`Throw Error`	Simulation stops and an error message appears. This setting is valid only for simulation. In generated code, chart execution always proceeds to the next time step rather than generating an error.

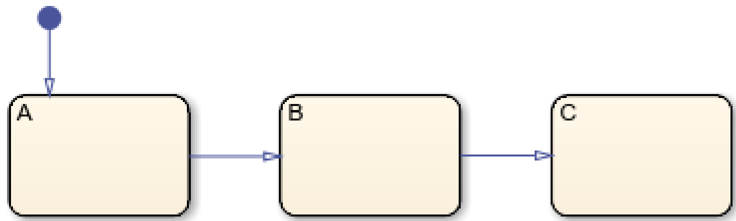
Note Selecting **Throw Error** can help detect infinite loops in transition cycles. For more information, see "Detection of Infinite Loops in Transition Cycles" on page 2-39.

Example of Chart with Super Step Semantics

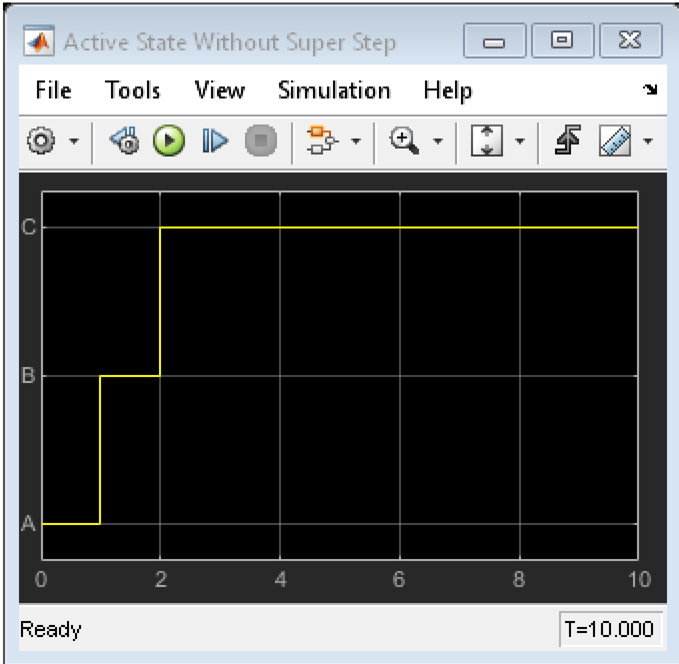
This example shows how super step semantics differs from default semantics. The model contains two Stateflow charts. One chart uses super step semantics. In the other, super step semantics are disabled.



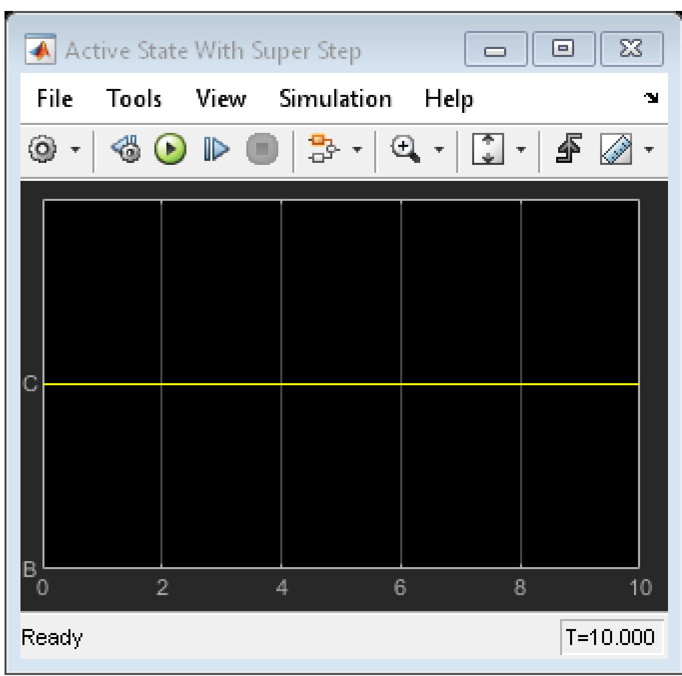
Each chart contains an identical sequence of states connected by transitions.



By default, the chart takes only one transition in each simulation step, progressing through states A, B, and C.



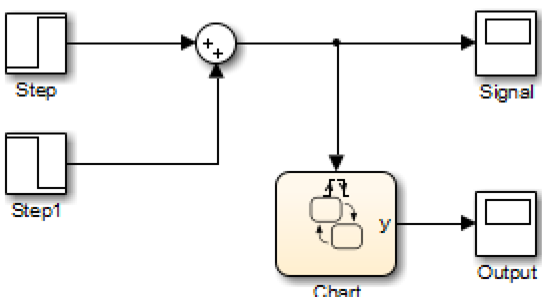
When you enable super step semantics, the chart takes all valid transitions in one step, stopping at state C.



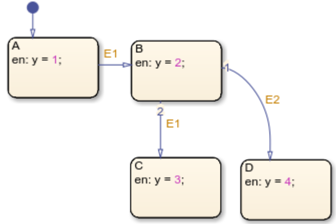
How Super Step Semantics Works with Multiple Input Events

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

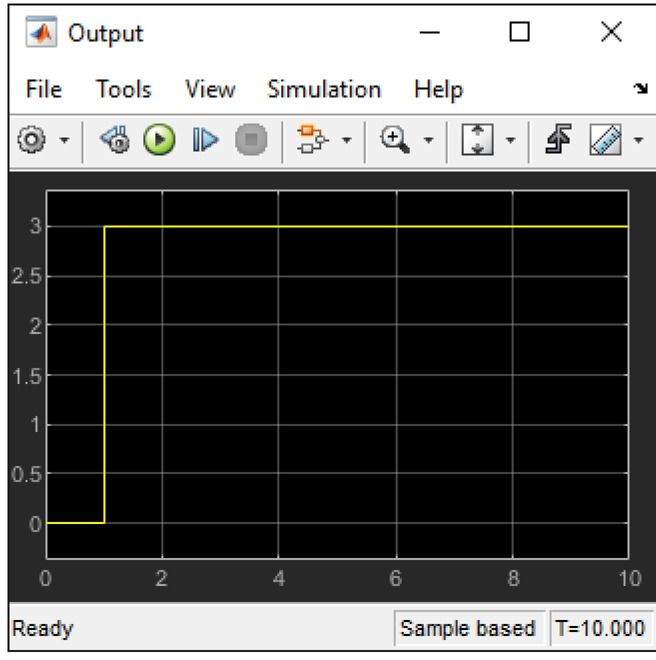
example, in this model, the Add block produces a 2-by-1 vector signal that goes from [0,0] to [1,1] at time `t` = 1.



As a result, when the model wakes up the chart, events `E1` and `E2` are both active:



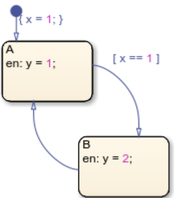
If you enable super step semantics, the chart takes all valid transitions for event `E1`. The chart takes transitions from state `A` to `B` and then from state `B` to `C` in a single super step. The Scope block shows that `y` = 3 at the end of the super step.



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

Detection of Infinite Loops in Transition Cycles

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



In this example, the transitions between states A and B cycle and produce an infinite loop because the value of x remains constant at 1. One way to detect infinite loops is to configure your chart to generate an error if it reaches a maximum number of iterations in a super step. See "Enable Super Step Semantics" on page 2-35.

See Also

Related Examples

- "Specify Properties for Stateflow Charts" on page 1-19
- "Execution of a Stateflow Chart" on page 2-12

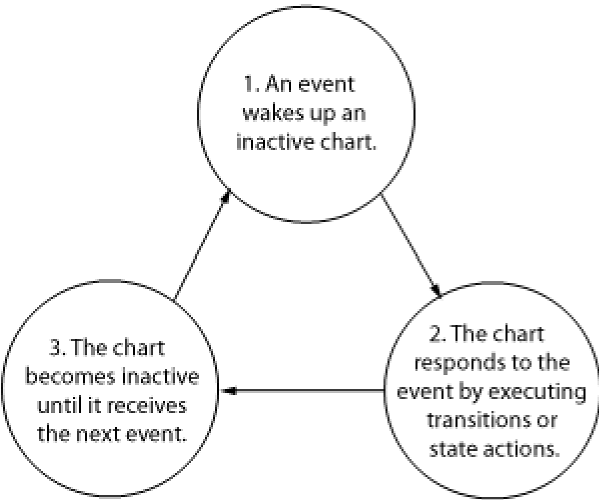
Use Events to Execute Charts

An event is a nongraphical object that can wake up and trigger actions in Stateflow chart. For more information, see “Synchronize Model Components by Broadcasting Events” on page 12-2.

How Stateflow Charts Respond to Events

Stateflow charts respond to events in a cyclical manner.

1. An event wakes up an inactive chart.
2. The chart responds to the event by executing transitions and state actions from the top down through the chart hierarchy. Starting at the chart level:
   a. The chart checks for valid transitions between states.
   b. The chart executes `during` and `on` actions for the active state.
   c. The chart proceeds to the next level down the hierarchy.
3. The chart becomes inactive until it receives the next event.



For more information, see “Execution of a Stateflow Chart” on page 2-12.

Events in Simulink Models

In Simulink models, Stateflow charts receive input events from other blocks in the model.

While processing an event, a state or transition action can generate explicit or implicit events that trigger additional steps. For example:

- The operator `send` can broadcast local or output events.
- The operators `enter` and `exit` can generate implicit local events when the chart execution enters or exits a state.
- The operator `change` can generate an implicit local event when the chart sets the value of a variable.

In each case, the chart interrupts its current activity to process the new event. When the activity caused by the new event finishes executing, the chart returns to the activity that was taking place before the interruption.

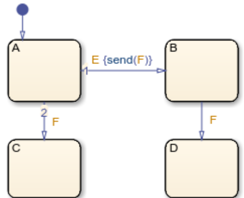
Note In a Simulink model, the execution of output edge-trigger events is equivalent to toggling the value of an output data value between 0 and 1. This type of event does not interrupt the current activity of a chart. Instead, the receiving block processes the event the next time that the model executes the block. For more information, see “Activate a Simulink Block by Sending Output Events” on page 12-15.

Early Return Logic

The results of processing a local event can conflict with the action that was taking place before the event was generated. Depending on the type of action, charts resolve these conflicts by using early return logic.

Action Type	Early Return Logic
State `entry` action	If the state is no longer active after the local event is processed, the chart stops the process of entering the state. The chart does not perform the remaining statements in the `entry` action.
State `during` action	If the state is no longer active after the local event is processed, the chart stops executing the state. The chart does not perform the remaining statements in the `during` action.
State `exit` action	If the state is no longer active after the local event is processed, the chart stops the process of exiting the state. The chart does not perform the remaining statements in the `exit` action nor any transition actions and state `entry` actions that result from exiting the state.
Condition action	If the source state of the inner or outer transition path, or the parent state of the default transition path, is no longer active after the local event is processed, the chart stops the transition process. The chart does not perform the remaining actions on the transition path or any state `exit` and `entry` actions that result from taking the transition.
Transition action	If the parent of the transition path is not active, or if the parent has an active substate, the chart stops the transition process. The chart does not perform the remaining actions on the transition path or any state `entry` actions that result from taking the transition.

For example, in this chart, the input event `E` and the local event `F` trigger the transitions between states.



Suppose that state `A` is active when the chart receives event `E`. The chart responds to the event with these steps:

1. The chart determines that the transition from the active state A to state B is valid as a result of event E.
2. The chart executes the condition action of the valid transition and broadcasts event F.
3. The chart interrupts the transition from state A to state B and begins to process event F.
4. The chart determines that the transition from the active state A to state C is valid as a result of event F.
5. State A executes its `exit` action.
6. State A becomes inactive.
7. State C becomes active.
8. State C executes its `entry` action.

After the chart processes event F, state C is the active state of the chart. Because state A is no longer active, the chart uses early return logic and stops the transition from state A to state B.

Tip Avoid using undirected local event broadcasts. Undirected local event broadcasts can cause unwanted recursive behavior in your chart. Instead, send local events by using directed broadcasts. For more information, see “Broadcast Local Events to Synchronize Parallel States” on page 12-25.

During simulation, Stateflow charts can detect undirected local event broadcasts. To control the level of diagnostic action, open the Configuration Parameters dialog box and, in the **Diagnostics > Stateflow** pane, set the **Undirected event broadcasts** parameter to `none`, `warning`, or `error`. The default setting is `warning`. For more information, see “Undirected event broadcasts” (Simulink).

Events in Standalone Charts

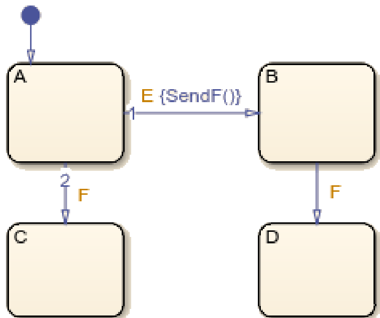
Standalone Stateflow charts receive an input event when you call the `step` function or an input event function in MATLAB.

Standalone charts also receive implicit events from MATLAB `timer` objects associated with the absolute-time temporal logic operators `after`, `at`, and `every`. These operators define temporal logic in terms of wall-clock time. If the state associated with the temporal logic operator becomes inactive before the chart processes the implicit event, the event does not wake up the chart.

Queuing of Events

If a chart is processing another operation when it receives an event, the chart queues the event for execution when the current step is completed. You can specify the size of the event queue by setting the configuration option `-eventQueueSize` when you create the chart object. For more information, see “Chart Object Configuration Options” on page 31-5.

For example, in this chart, the input events E and F trigger the transitions between states. Assume that `SendF` is a function in the MATLAB path that calls the input event function F.



Suppose that state A is active when the chart receives event E. The chart responds to the event with these steps:

1. The chart determines that the transition from the active state A to state B is valid as a result of event E.
2. The chart executes the condition action of the valid transition and calls the function SendF.
3. SendF calls the input event function F. Because the chart is busy processing a condition action, it queues event F.
4. The chart completes executing the condition action.
5. State A executes its exit action.
6. State A becomes inactive.
7. State B becomes active.
8. State B executes its entry action.
9. The chart begins to process the queued event F.
10. The chart determines that the transition from the active state B to state D is valid as a result of event F.
11. State B executes its exit action.
12. State B becomes inactive.
13. State D becomes active.
14. State D executes its entry action.

After the chart processes event F, state D is the active state of the chart.

See Also

after | at | change | enter | every | exit | send | timer

More About

- “Execution of a Stateflow Chart” on page 2-12
- “Synchronize Model Components by Broadcasting Events” on page 12-2
- “Control Chart Behavior by Using Implicit Events” on page 12-28
- “Control Chart Execution by Using Temporal Logic” on page 14-35

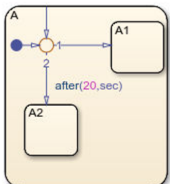
Group and Execute Transitions

Transition Flow Chart Types

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

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

Each set of flow charts includes other transition segments connected to a qualifying transition segment through junctions and transitions. Consider the following example:



In this example, state A has both an inner and a default transition that connect to a junction with outgoing transitions to states A.A1 and A.A2. If state A is active, its set of inner flow charts includes:

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

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

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

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

Order of Execution for a Set of Flow Charts

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

1. Order the transition segments for the active state.

   An active state can have several possible outgoing transitions. The chart orders these transitions before checking them for validity. See “Transition Evaluation Order” on page 2-28.
2. Select the next transition segment in the set of ordered transitions.
3. Test the transition segment for validity.

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

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

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

   b The states that are the immediate substates of the parent of the transition path exit (see “Exit a State” on page 2-23).

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

   d The destination state becomes active (see “Enter a Chart or State” on page 2-17).

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

   a Testing stops without any state exits or entries.

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

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

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

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

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

Execution Order for Parallel States

Ordering for Parallel States

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

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

- Explicit ordering

  Specify explicitly the execution order of parallel states on a state-by-state basis (see “Explicit Ordering of Parallel States” on page 2-46).
- 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 2-47).

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.

In code that is generated from Stateflow charts that contain parallel states, each state executes based on its order.

Explicit Ordering of Parallel States

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

How Explicit Ordering Works

When you open a new Stateflow chart — or one that does not yet contain any parallel states — the chart automatically assigns priority numbers to parallel states in the order you create them. Numbering starts with the next available number within the parent container.

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

You can reset execution order assignments at any time on a state-by-state basis, as described in “Set Execution Order for Parallel States Individually” on page 2-47. When you change execution order for a parallel state, the Stateflow chart automatically renumbers the other parallel states to preserve their relative execution order. For details, see “Order Maintenance for Parallel States” on page 2-48.

Order Parallel States Explicitly

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

1 “Enable Explicit Ordering at the Chart Level” on page 2-47
2 “Set Execution Order for Parallel States Individually” on page 2-47

Enable Explicit Ordering at the Chart Level

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

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

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

   If your chart already contains parallel states that have been ordered implicitly, the existing priorities are preserved until you explicitly change them. When you add new parallel states in explicit mode, your chart automatically assigns priorities based on order of creation (see “How Explicit Ordering Works” on page 2-46). However you can now explicitly change execution order on a state-by-state basis, as described in “Set Execution Order for Parallel States Individually” on page 2-47.

Set Execution Order for Parallel States Individually

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

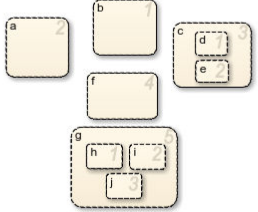
Implicit Ordering of Parallel States

Rules of Implicit Ordering for Parallel States

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

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

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



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

Order Parallel States Implicitly

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

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

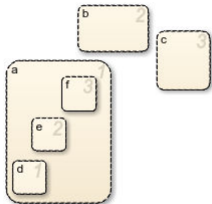
Order Maintenance for Parallel States

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

- Fill in gaps in the sequence so that ordering is contiguous
- Ensure that no two states have the same priority
- Preserve the intended relative priority of execution

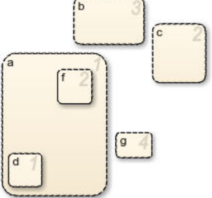
How a Chart Preserves Relative Priorities in Explicit Mode

For explicit ordering, a chart preserves the user-specified priorities. Consider this example of explicit ordering:



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

1. Change the priority of top-level state b to *3*.
2. Add a top-level state g.
3. Remove substate e.

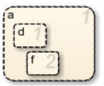


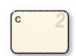
The chart preserves the priority set explicitly for top-level state b, but renumbers all other parallel states to preserve their prior relative order.

How a Chart Preserves Relative Priorities in Implicit Mode

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

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





Execution Priorities in Restored States

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

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

If you remove a state by...	And restore the state by...	What is the priority?
Deleting, cutting, dragging outside the boundaries of the parent state, or dragging so its boundaries overlap the parent state	Using the undo command	The original priority is restored.
Dragging outside the boundaries of the parent state or so its boundaries overlap the parent state *and* releasing the mouse button	Dragging it back into the parent state	The original priority is lost. The Stateflow chart treats the restored state as the last created and assigns it the lowest execution priority.
Dragging outside the boundaries of the parent state or so its boundaries overlap the parent state *without* releasing the mouse button	Dragging it back into the parent state	The original priority is restored.
Dragging so its boundaries overlap one or more sibling states	Dragging it to a location with no overlapping boundaries inside the same parent state	The original priority is restored.
Cutting	Pasting	The original priority is lost. The Stateflow chart treats the restored state as the last created and assigns it the lowest execution priority.

Switching Between Explicit and Implicit Ordering

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

Execution Order of Parallel States in Boxes and Subcharts

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

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

Model Logic Patterns and Iterative Loops Using Flow Charts

- "Create Flow Charts in Stateflow" on page 3-2
- "Create Flow Charts by Using Pattern Wizard" on page 3-5
- "Convert MATLAB Code into Stateflow Flow Charts" on page 3-17

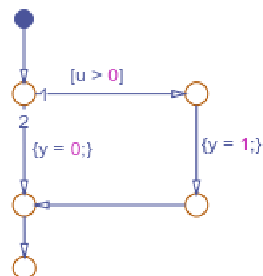
Create Flow Charts in Stateflow

A Stateflow flow chart is a graphical construct that models logic patterns such as decision trees and iterative loops. Flow charts represent combinatorial logic in which one result does not depend on prior results. You build flow charts by combining only connective junctions and transitions. The junctions provide decision branches between different transition paths. Executing a flow chart begins at a default transition and ends at a terminating junction, which is a junction that has no outgoing transitions. For more information, see “Combine Transitions and Junctions to Create Branching Paths” on page 1-54.

Note If a Stateflow chart contains only a flow chart, the chart evaluates the flow chart each time the chart wakes up. In contrast, the chart evaluates any flow chart inside a state only during state entry.

A best practice is to encapsulate flow charts in graphical functions to create modular and reusable logic that you can call anywhere in a chart. For more information about graphical functions, see “Reuse Logic Patterns by Defining Graphical Functions” on page 6-9.

For example, this flow chart models simple `if-else` logic:



The flow chart models this code:

```
if u > 0
   y = 1;
else
   y = 0;
end
```

Draw a Flow Chart

You can draw flow charts by using connective junctions as branch points between alternate transition paths. To draw a flow chart:

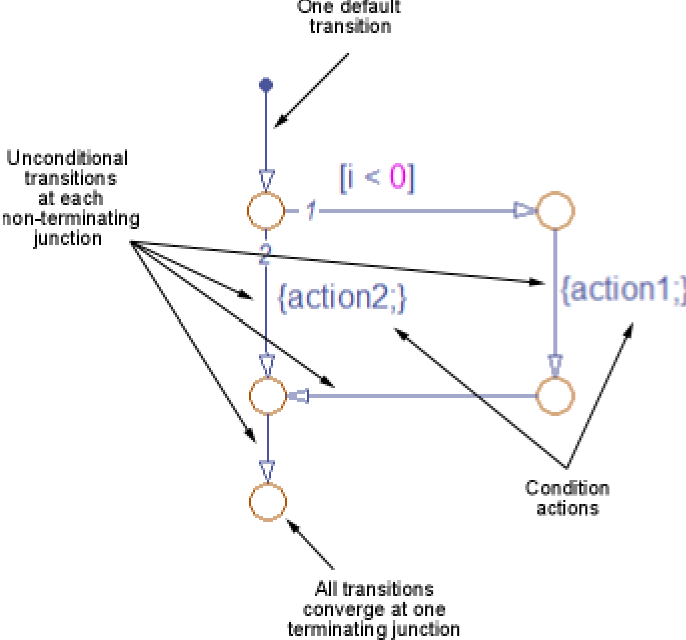
1. Open a new chart.
2. Add a default transition to the junction where the flow chart execution starts. In the object palette, click the Default transition icon. Then, on the chart canvas, click the location for the transition. The Stateflow Editor adds a new connective junction as the destination of the transition.
3. To add a new transition, point to the border of a junction. Then, click and drag away from the junction. The Stateflow Editor provides graphical cues that allow you to add a junction or a state.

- To place a junction at the end of the transition, click the circular cue.
- To connect the transition to an existing junction, drag the pointer to the desired destination.

4 To add a connective junction to the chart, in the object palette, click the Junction icon. Then, on the chart canvas, click the location for the new junction.
5 Repeat the previous steps as required.
6 Label the transitions as described in “Define Actions in a Transition” on page 1-39.

Best Practices for Creating Flow Charts

Follow these best practices when creating flow charts.



Use only one default transition

Flows charts have a single entry point.

Provide only one terminating junction

Multiple terminating junctions reduce readability of a flow chart.

Converge all transition paths to the terminating junction

Execution of a flow chart always reaches the termination point.

Provide an unconditional transition from every junction except the terminating junction

If unintended backtracking occurs during simulation, a warning message appears.

To control the level of diagnostic action for unintended backtracking, open the Configuration Parameters dialog box and, in the **Diagnostics** > **Stateflow** pane, set the **Unexpected backtracking** parameter to `none`, `warning`, or `error`. The default setting is `warning`. For more information, see “Unexpected backtracking” (Simulink).

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.

To process updates, use condition actions instead of transition actions

Flow charts test transitions, but do not execute them. As a result, flow charts never execute transition actions. Furthermore, in charts that use MATLAB as the action language, using a transition action in a graphical function results in a compile-time error.

See Also

More About

- “Create Flow Charts by Using Pattern Wizard” on page 3-5
- “Reuse Logic Patterns by Defining Graphical Functions” on page 6-9
- “Convert MATLAB Code into Stateflow Flow Charts” on page 3-17
- “How Stateflow Objects Interact During Execution” on page 1-8

Create Flow Charts by Using Pattern Wizard

The Pattern Wizard is a utility that generates common flow chart patterns for use in graphical functions and charts. The Pattern Wizard offers several advantages over manually creating flow charts. The Pattern Wizard:

- Generates common logic and iterative loop patterns.
- Promotes consistency in geometry and layout across patterns.
- Facilitates storing and reusing patterns from a central location.
- Allows inserting patterns in an existing flow chart.

The Pattern Wizard generates flow charts whose geometry and layout comply with the guidelines from the MathWorks Advisory Board (MAB). You can customize your flow chart by modifying the conditions and actions or by inserting additional logic patterns. You can also save your flow chart as a custom pattern in the Pattern Wizard for later reuse.

For example, suppose that you want to use the Pattern Wizard to create a graphical function for iterating over the upper triangle of a two-dimensional matrix. The function consists of two nested `for` loops in which the row index `i` is always less than or equal to the column index `j`. By using the Pattern Wizard, you can:

1. Create a flow chart for the outer loop that iterates over the row index `i`. See "Create Reusable Flow Charts" on page 3-5.
2. Extend the flow chart by inserting an inner loop that iterates over the column index `j`. See "Insert Logic Patterns in Existing Flow Charts" on page 3-6.
3. Save the flow chart as a custom pattern in the Pattern Wizard. See "Save Custom Flow Chart Patterns" on page 3-9.
4. Reuse the custom pattern in a graphical function. See "Reuse Custom Flow Chart Patterns" on page 3-10.

Create Reusable Flow Charts

To create a flow chart, on the **Modeling** tab, select a pattern from the **Pattern** gallery. Pattern selections include:

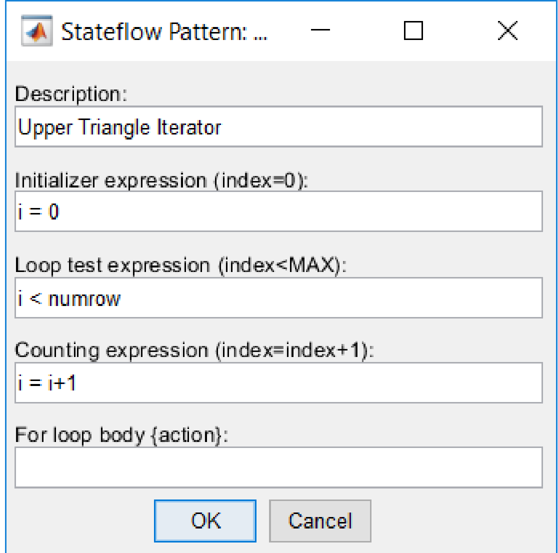
- **If**, **If-Else**, **If-Elseif**, and other nested decision patterns.
- **For**, **While**, and **DoWhile** loop patterns.
- Switch patterns with up to four cases.
- Custom patterns that you saved for later reuse.
- Patterns that you define in a MATLAB `.m` file.

The Pattern dialog box prompts you for conditions and actions specific to the pattern that you select. For more information on flow chart patterns, see "MAB-Compliant Patterns from the Pattern Wizard" on page 3-11.

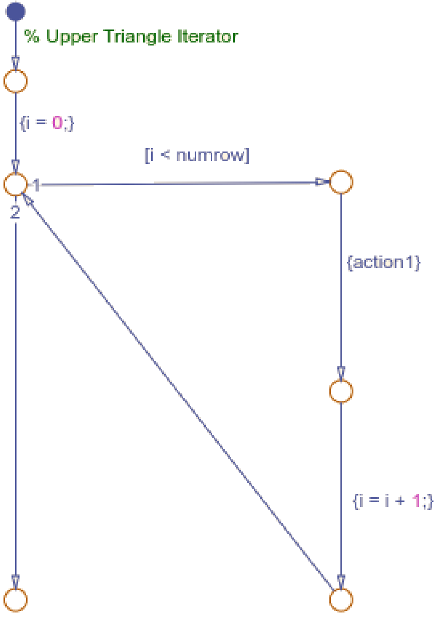
For example, to create the outer `for` loop in the upper triangle iterator pattern:

1. On the **Modeling** tab, select **Pattern > For Loop**.

2 In the Pattern dialog box, specify the initializer, loop test, and counting expressions for iterating through the first dimension of the matrix:



3 Click **OK**. The Pattern Wizard generates this flow chart.



To complete the upper triangle iterator pattern, insert a second `for` loop along the vertical transition in this flow chart.

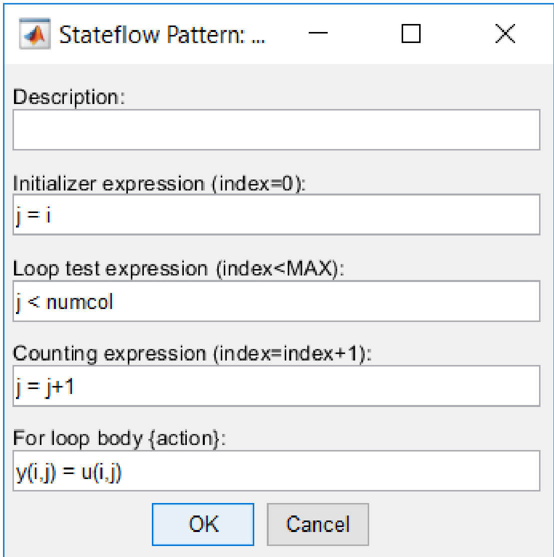
Insert Logic Patterns in Existing Flow Charts

Use the Pattern Wizard to add loop or decision logic extensions to an existing flow chart. Select an eligible vertical transition and choose a pattern from the **Pattern** gallery. Options include decision,

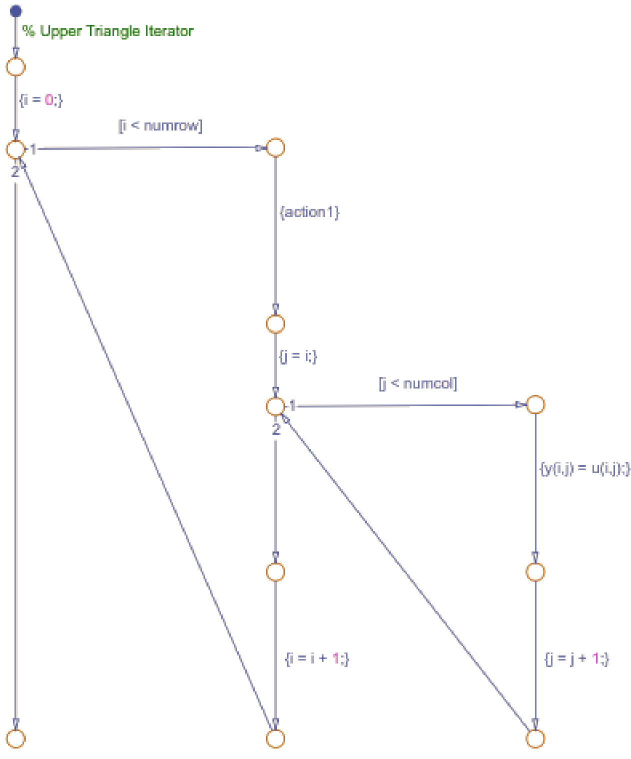
loop, and switch patterns. The Pattern dialog box prompts you for conditions and actions specific to the pattern that you select.

For example, to add the second loop in the upper triangle iterator pattern:

1. In the Stateflow Editor, from the outer `for` loop pattern, select the vertical transition labeled `{action1}`.
2. On the **Modeling** tab, select **Pattern** > **For Loop**.
3. In the Pattern dialog box, specify the initializer, loop test, and counting expressions for iterating through the second dimension of the matrix. The initializer expression ensures that *i* never exceeds *j*. Also enter an action that retrieves each element in the upper triangle of the matrix.



4. Click **OK**. The Pattern Wizard adds the second loop to the flow chart.



5 Save the model containing the pattern.

Guidelines for Inserting Logic Patterns

When you create logic extensions:

- You can select only one transition to extend at a time. The selected transition must be exactly vertical and have a destination junction.
- You can extend only flow charts created by the Pattern Wizard.
- The Stateflow chart containing the flow chart can contain only junctions and transitions. The chart cannot contain other objects, such as states, functions, or truth tables.
- You cannot extend a pattern that has been custom-created or modified.
- You cannot choose a custom pattern as the extension.

If your selection is not eligible for insertion, when you choose a pattern from the **Pattern** gallery, you see a message instead of pattern options.

Message	Issue
Select a vertical transition	You have not selected a vertical transition.
Selected transition must be exactly vertical	You selected a transition, but it is not vertical.
Select only one vertical transition	You have selected more than one transition.

Message	Issue
Editor must contain only transitions and junctions	There are other objects, such as states, functions, or truth tables, in the chart.

Save Custom Flow Chart Patterns

Use the Pattern Wizard to save flow chart patterns in a central location, and then easily retrieve them for reuse in graphical functions and charts. Select the flow chart with the pattern that you want to save and select **Pattern** > **Save As Pattern**.

For example, suppose that you want to save the upper triangle iterator pattern for later reuse:

1. Create a folder for storing your custom patterns. See “Guidelines for Creating a Custom Pattern Folder” on page 3-9.
2. In the Stateflow Editor, select the upper triangle iterator flow chart.
3. On the **Modeling** tab, select **Pattern** > **Save As Pattern**.
4. If you have not designated the custom pattern folder, the Pattern Wizard prompts you to select a folder. Choose the folder that you created and click **Select Folder**.
5. At the prompt, name your pattern `UpperTriangleIterator` and click **Save**. The Pattern Wizard saves your pattern as a model file `UpperTriangleIterator.slx` in the custom pattern folder.

Note You can use the Pattern Wizard to reuse only flow charts. To reuse states and subcharts, create an atomic subchart. For more information, see “Create Reusable Subcomponents by Using Atomic Subcharts” on page 17-2.

Guidelines for Creating a Custom Pattern Folder

The Pattern Wizard uses a single, flat folder for saving and retrieving flow chart patterns.

- Store all flow charts at the top level of the custom pattern folder. Do not create subfolders.
- Make sure that all flow chart files have an `.mdl` or `.slx` extension.

Change Your Custom Pattern Folder

The Pattern Wizard remembers your choice of custom pattern folder for future sessions. To choose a different folder, use the `sfpref` function. For example, to set the custom pattern folder to `C:\patterns`, enter:

```
sfpref(PatternWizardCustomDir=fullfile("C:","patterns"));
```

Alternatively, rename your existing custom pattern folder and do one of the following:

- Save a new custom pattern to the Pattern Wizard.
- Reuse an existing custom pattern from the Pattern Wizard.

The Pattern Wizard prompts you to choose a new folder.

Reuse Custom Flow Chart Patterns

The Pattern Wizard stores your flow charts as model files in the custom pattern folder. The patterns that you save in this folder appear in a drop-down list when you select **Pattern** > **Custom**. You can add a custom pattern directly to a chart or to a subcharted graphical function in your chart.

For example, to add the upper triangle iterator custom pattern to a graphical function:

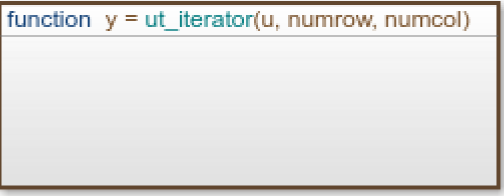
1. From the object palette, add a graphical function to your chart as described in “Define a Graphical Function” on page 6-9.
2. Enter this function signature:

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

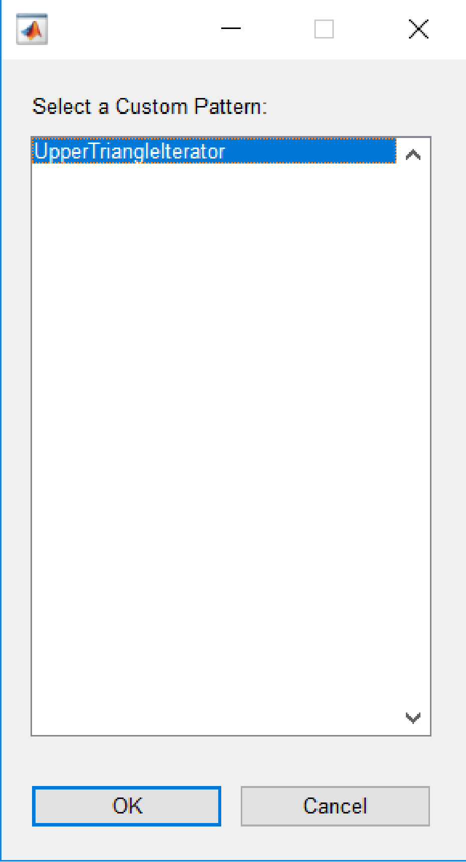
   The function takes three inputs.

Input	Description
u	2-D matrix
numrow	Number of rows in the matrix
numcol	Number of columns in the matrix

3. Right-click inside the function and select **Group & Subchart** > **Subchart**. The function appears as an opaque box.



4. Double-click the subcharted function to open it.
5. Remove the default flow chart from the graphical function.
6. On the **Modeling** tab, select **Pattern** > **Custom**. A dialog box opens, listing all the patterns that you have saved in your custom pattern folder.



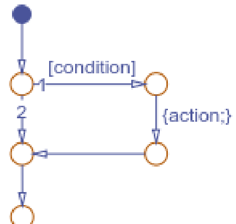
7 Select the upper triangle iterator pattern and click **OK**. The Pattern Wizard adds your custom pattern to the graphical function.
8 In the graphical function, in place of `action1`, substitute an appropriate action. This action repeats once for every row of the matrix.

MAB-Compliant Patterns from the Pattern Wizard

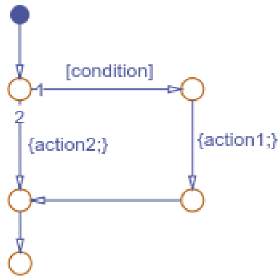
The Pattern Wizard generates flow charts whose geometry and layout comply with the guidelines from the MathWorks Advisory Board (MAB).

Decision Patterns

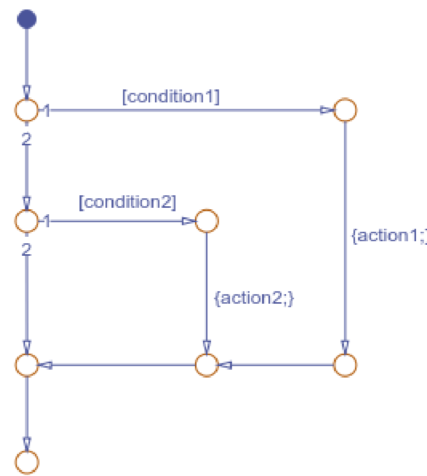
If



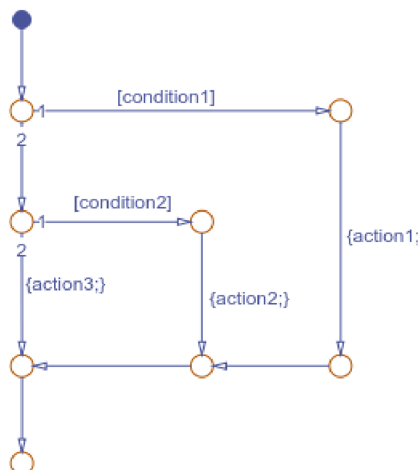
If-Else



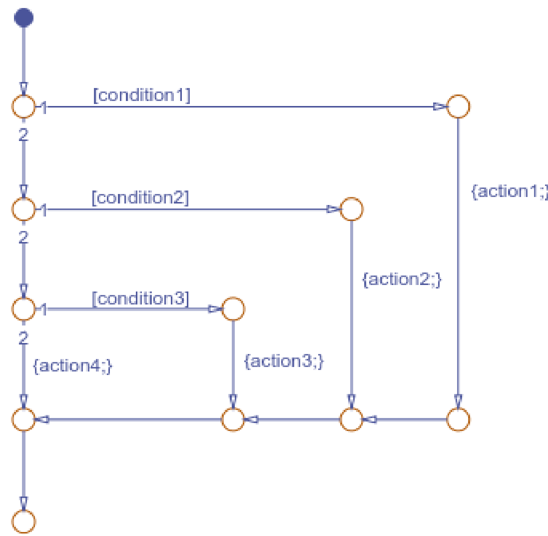
If-Elseif



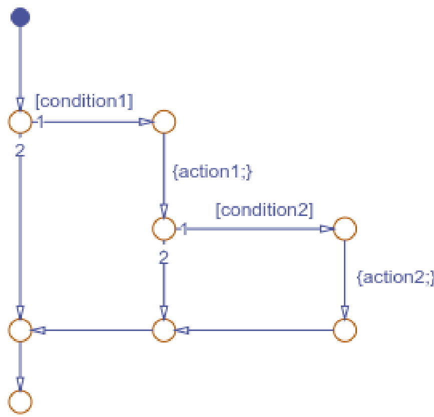
If-Elseif-Else



If-Elseif-Elseif-Else

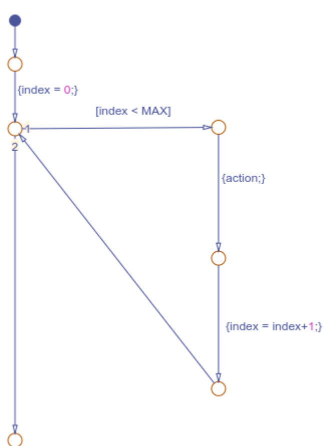


Nested If

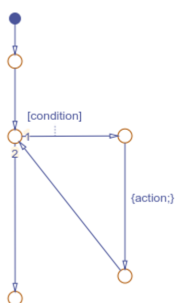


Loop Patterns

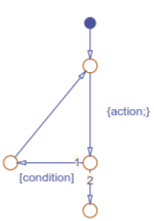
For Loop



While Loop

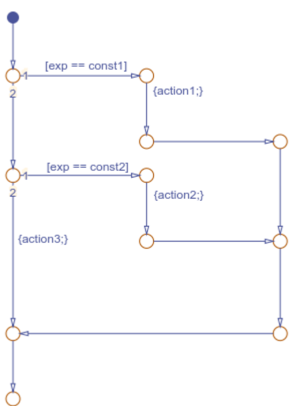


DoWhile Loop

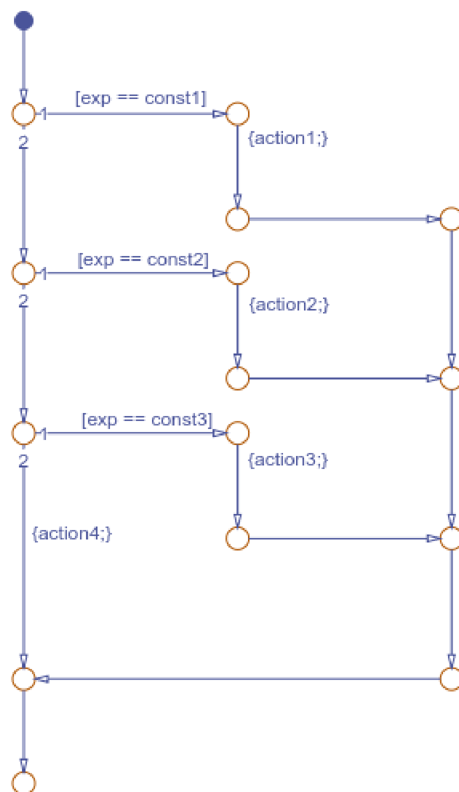


Switch Patterns

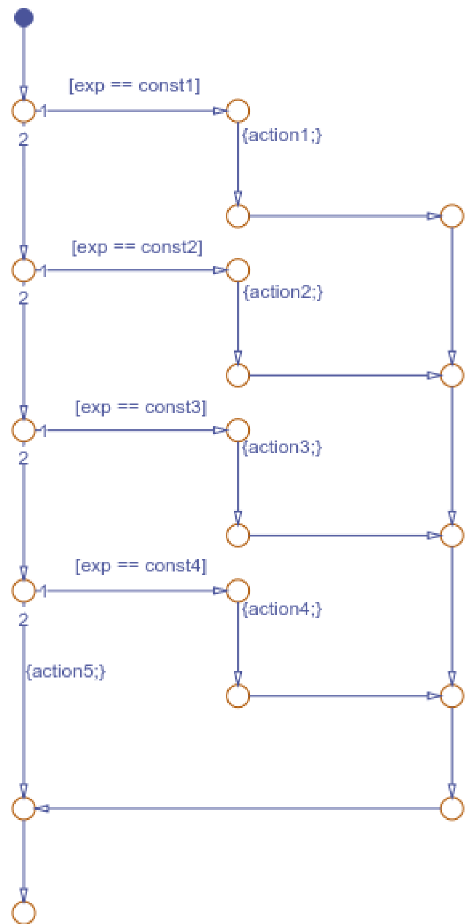
Two Cases



Three Cases



Four Cases



See Also

`sfpref`

More About

- "Create Flow Charts in Stateflow" on page 3-2
- "Reuse Logic Patterns by Defining Graphical Functions" on page 6-9
- "Convert MATLAB Code into Stateflow Flow Charts" on page 3-17

Convert MATLAB Code into Stateflow Flow Charts

To transform your MATLAB code into Stateflow flow charts and graphical functions, use the Pattern Wizard. Supported patterns for conversion include:

- `if`, `if-else`, and other nested decision statements.
- `for` and `while` loops.
- `switch` statements.

The Pattern Wizard can convert MATLAB functions and scripts.

- MATLAB functions are converted to Stateflow graphical functions.
- MATLAB scripts are converted to Stateflow flow charts.

Converting MATLAB code is supported only in standalone Stateflow charts. For more information, see “Create Stateflow Charts for Execution as MATLAB Objects” on page 31-2.

Create Flow Charts from MATLAB Scripts

This MATLAB script empirically verifies one instance of the Collatz conjecture. When given the numeric input `u`, the script computes the hailstone sequence n_0 = `u`, n_1, n_2, n_3, ⋯ by iterating this rule:

- If n_i is even, then $n_{i+1} = n_i/2$.
- If n_i is odd, then $n_{i+1} = 3n_i + 1$.

The Collatz conjecture states that every positive integer has a hailstone sequence that eventually reaches one.

```
% Hailstone sequence u, c(u), c(c(u)),...
y = u;
while y(end) ~= 1
    y(end+1) = c(y(end));
end
disp(y);

function n = c(n)
% Compute next number in hailstone sequence.
% If n is even, then c(n) = n/2.
% If n is odd, then c(n) = 3*n+1.
    if rem(n,2) == 0
        n = n/2;
    else
        n = 3*n+1;
    end
end
```

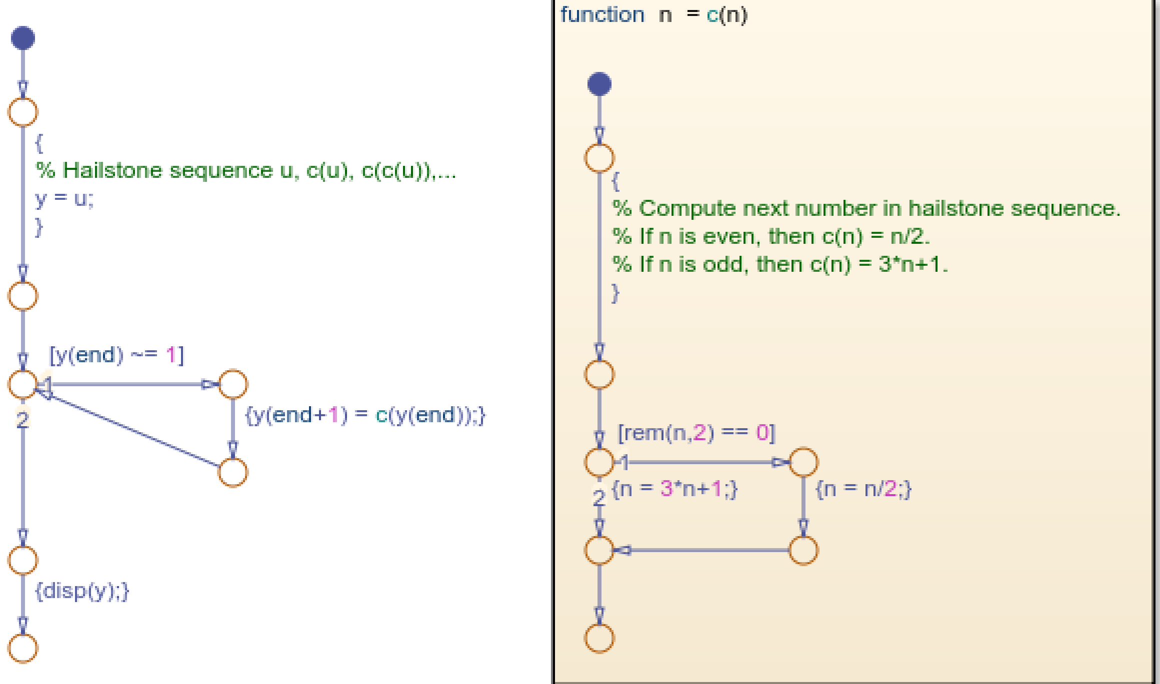
The script executes a `while` loop that repeatedly calls the auxiliary function `c` until it produces an output value of one. The function `c` consists of a conditional `if-else` statement whose output depends on the parity of the input.

To convert this script into a flow chart and a graphical function:

1. Open a new standalone chart.

```
edit hailstone.sfx
```

2 On the **State Chart** tab, select **Pattern** > **Select File**.
3 In the dialog box, choose the MATLAB script and click **Open**. The Pattern Wizard adds a flow chart and a graphical function to your Stateflow chart. Double-click the graphical function to see its contents.



4 In the **Symbols** pane, click **Resolve Undefined Symbols** . The Stateflow Editor resolves u and y as local data.
5 Save your chart.
6 To execute the chart from the Stateflow Editor, in the **Symbols** pane, enter a value of u = 9 and click **Run** . While the flow chart is executing, the Stateflow Editor highlights the active transitions through chart animation. When the execution stops, the MATLAB Command Window displays the hailstone sequence, starting with a value of nine:

```
     9    28    14     7    22    11    34    17    52    26    13    40    20    10     5
```

7 Click **Stop** .

You can copy generated flow charts and graphical functions and paste them in other charts, including Stateflow charts in Simulink models. If your MATLAB code uses functionality that is restricted for code generation in Simulink, you must modify the flow chart actions before simulating the chart. For more information, see “Call Extrinsic MATLAB Functions in Stateflow Charts” on page 14-13.

Note Suppose that you use `nargin` in a MATLAB function that you convert to a graphical function in a chart. Because `nargin` counts the chart object as one of the input arguments of the graphical

function, the value of `nargin` in the graphical function is equal to one plus the value of `nargin` in the original MATLAB function. For more information, see “Execute a Standalone Chart” on page 31-3.

See Also

More About

- “Create Flow Charts in Stateflow” on page 3-2
- “Create Flow Charts by Using Pattern Wizard” on page 3-5
- “Reuse Logic Patterns by Defining Graphical Functions” on page 6-9
- “Create Stateflow Charts for Execution as MATLAB Objects” on page 31-2

4

Simulink Subsystems as Stateflow States

- “Simulink Subsystems as States” on page 4-2
- “Create and Edit Simulink Based States” on page 4-8
- “Access Block State Data” on page 4-14
- “Map Variables for Simulink Based States” on page 4-20
- “Set Simulink Based State Properties” on page 4-22
- “Hybrid Clutch System” on page 4-24

Simulink Subsystems as States

By using a Simulink subsystem within a Stateflow state, you can model hybrid dynamic systems or systems that switch between periodic and continuous time dynamics. In your Stateflow chart, you can use Simulink based states to model a periodic or continuous dynamic system combined with switching logic that uses transitions. You can access inputs and outputs from your chart within each Simulink based state. Simulink based states are not supported in standalone Stateflow charts in MATLAB.

To initialize Simulink blocks when switching between Simulink based states, use Stateflow textual notation or Simulink State Reader and State Writer blocks.

To create linked Simulink based states, use libraries to save action subsystems. When you copy an action subsystem from a library model into a Stateflow chart, it appears as a linked Simulink based state. When you update the library block, the changes are reflected in all Stateflow charts containing the block.

Using Simulink based states means that you do not have to use complex textual syntax in Stateflow to model hybrid systems.

When to Use Simulink Based States

Use Simulink based states when:

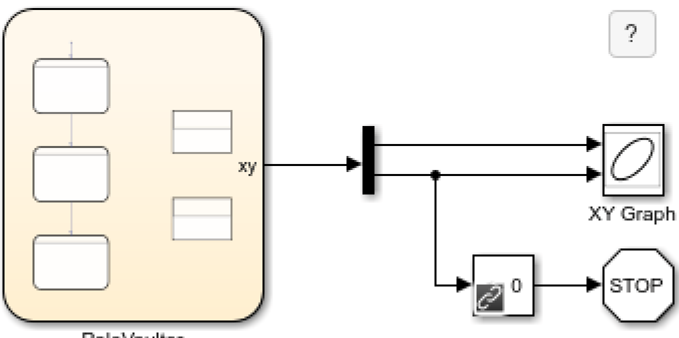
- You want to model hybrid dynamic systems that include continuous or periodic dynamics.
- The structure of the system dynamics change substantially between the various modes of operation, for example, modeling PID controllers.

For systems where you call logic intermittently, use Simulink functions.

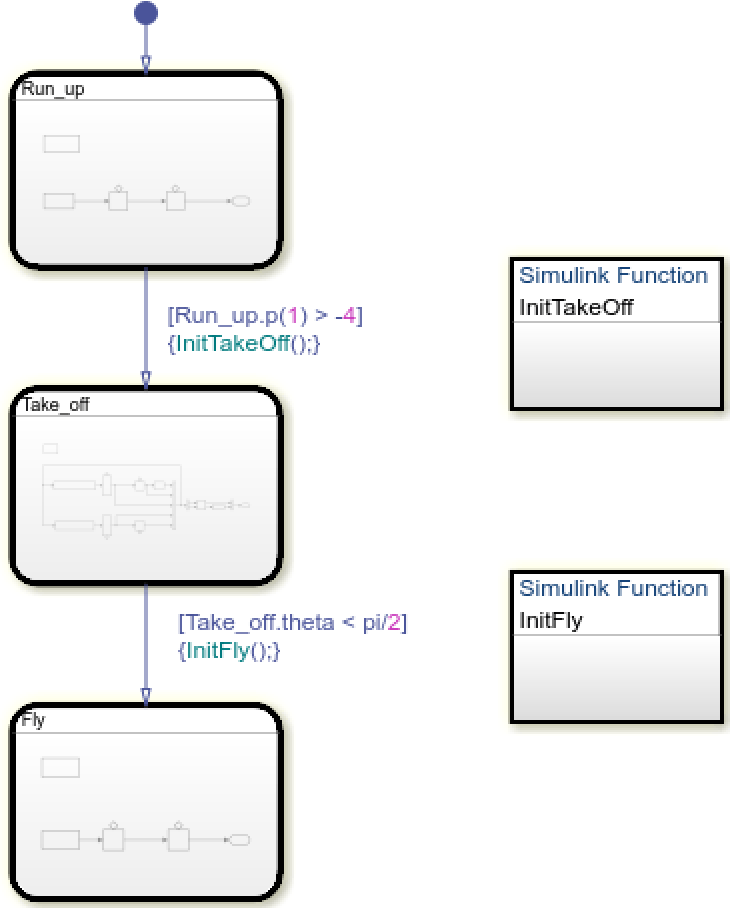
When the structure of the Simulink algorithm remains substantially unchanged, but certain gains or parameters switch between various models, use Simulink logic outside of Stateflow. An example of this type of algorithm is gain scheduling. See "Model Gain-Scheduled Control Systems in Simulink" (Simulink Control Design).

Model a Pole Vaulter by Using Simulink Based States

This Stateflow chart models a person moving through the stages of pole vaulting by using Simulink based states.



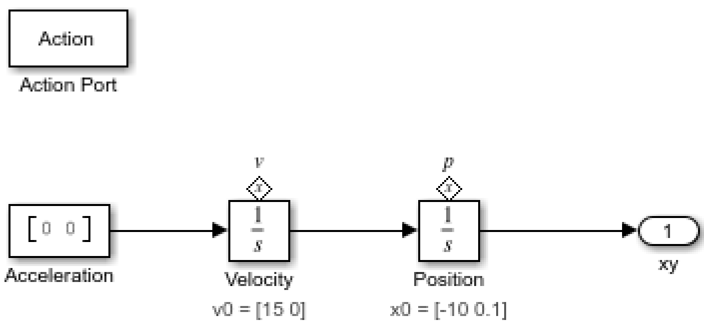
The first stage is the approach run of the vaulter, which is modeled in the Simulink based state Run_up. In the second stage, the vaulter plants the pole and takes off, which is modeled by the Simulink based state Take_off. The final stage happens when the vaulter clears the bar and releases the pole, which is modeled by the Simulink based state Fly.



The states Run_up and Fly are easier to model by using Cartesian coordinates. The state Take_off is easier to model by using polar coordinates. To switch from one coordinate system to another, use Simulink functions InitTakeOff and InitFly.

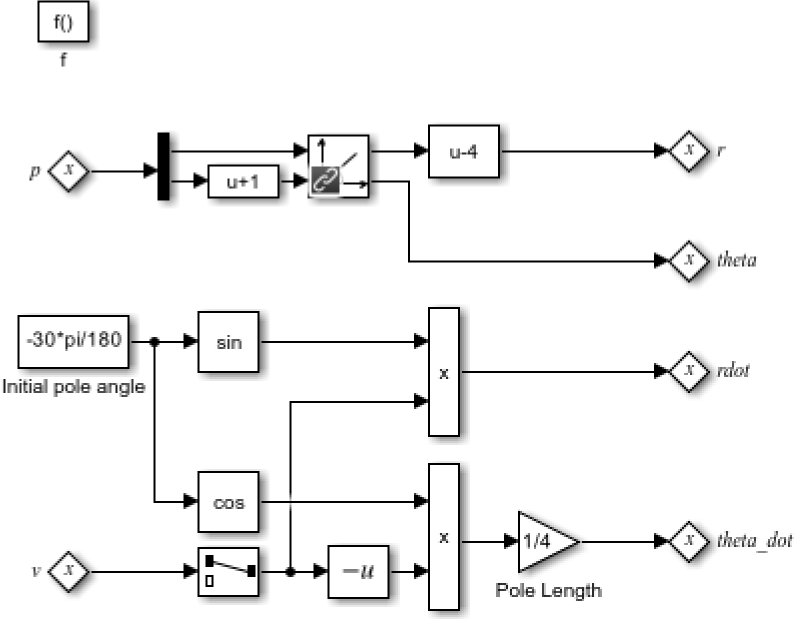
Model the Approach of the Pole Vaulter

The default state in the chart `PoleVaulter` is `Run_up`. This state models the pole vaulter traveling along the ground toward the jump. The pole vaulter starts at -10 on the x-axis and runs toward zero. As the pole vaulter moves along the ground, the position of the pole vaulter in the xy-plane is continuously changing, but the state of running remains the same. In this model, the integrator blocks `Position` and `Velocity` are state owner blocks for State Reader blocks in the Simulink function `InitTakeOff`. This subsystem outputs the Cartesian coordinates of the pole vaulter.



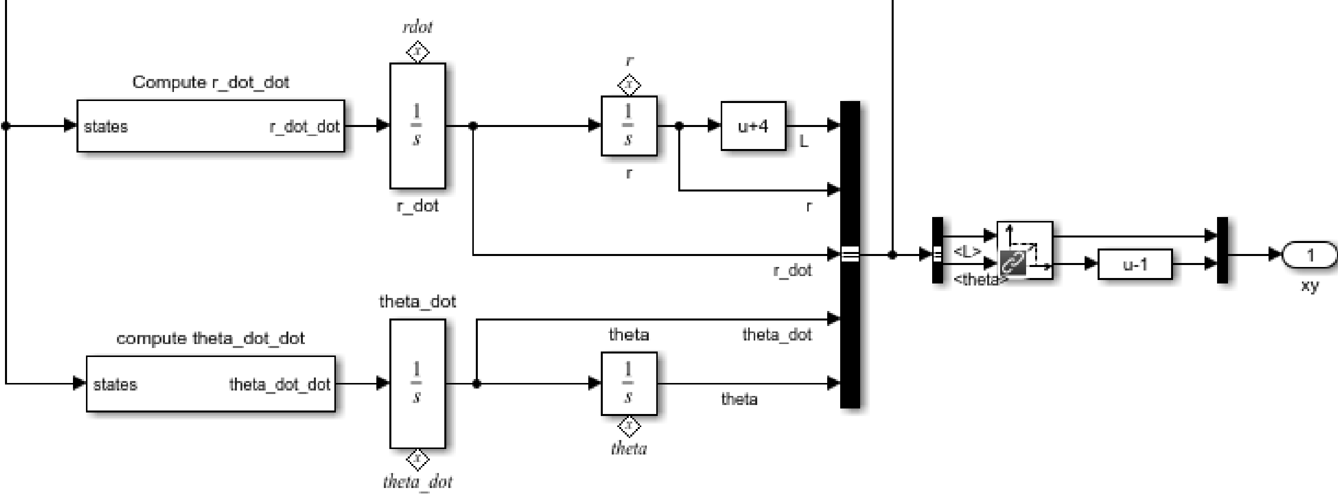
Convert Cartesian Coordinates to Polar Coordinates

The transition from `Run_up` to `Take_off` occurs when the position of the pole vaulter along the x-axis, `Run_up.p(1)`, becomes greater than -4. During the transition `InitTakeOff` is initialized, the State Reader block connects to its owner block, and the function is executed. This function converts the Cartesian coordinates from `Position` and `Velocity` to polar coordinates, `r`, `theta`, `rdot`, and `theta_dot`. These coordinates are output as State Writer blocks, which are connected to owner blocks in the state `Take_off`. The Simulink function `InitTakeOff` contains this logic:



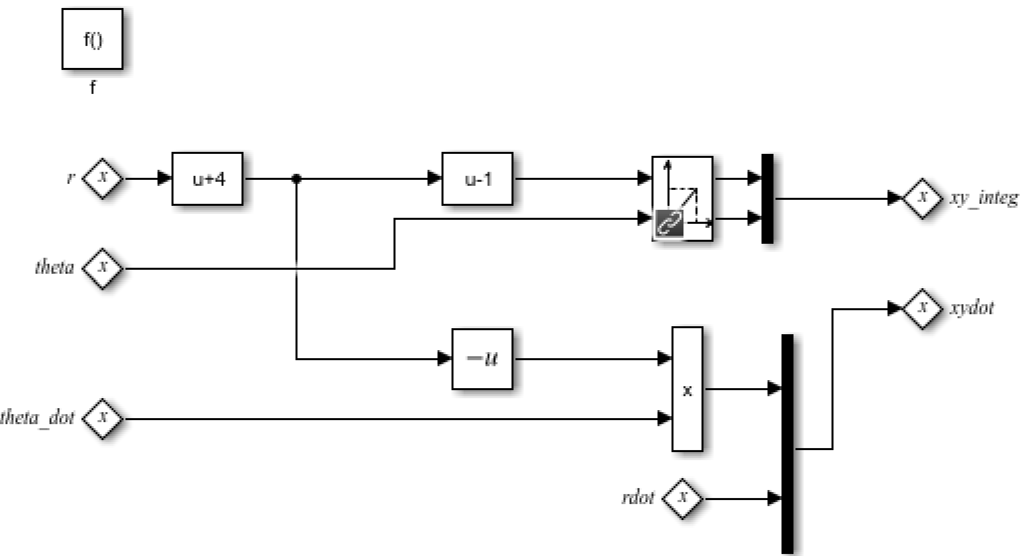
Model the Take Off of the Pole Vaulter

When the position of the pole vaulter along the x-axis, `Run_up.p(1)`, becomes greater than -4, the Simulink based state `Take_off` becomes the active state. This state models the pole vaulter during the take off phase of the jump. This subsystem outputs the Cartesian coordinates of the pole vaulter.



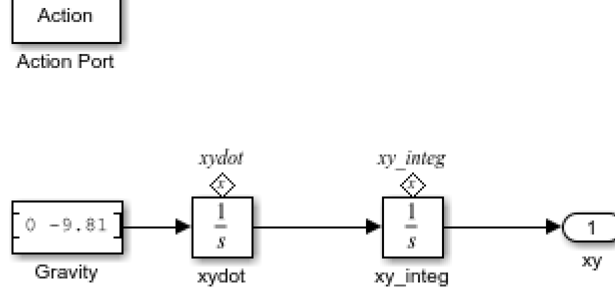
Convert Polar Coordinates to Cartesian Coordinates

The transition from `Take_off` to `Fly` occurs when the angle of the pole vaulter, `theta`, becomes less than $\pi/2$. During the transition, `InitFly` is initialized, the State Reader block connects to its owner block, and the function is executed. This function converts the polar coordinates `r`, `theta`, `rdot`, and `theta_dot` to Cartesian coordinates, `xy_integ` and `xydot`. These coordinates are output as State Writer blocks, which are connected to owner blocks in the state `Fly`. The Simulink function `InitFly` contains this logic:

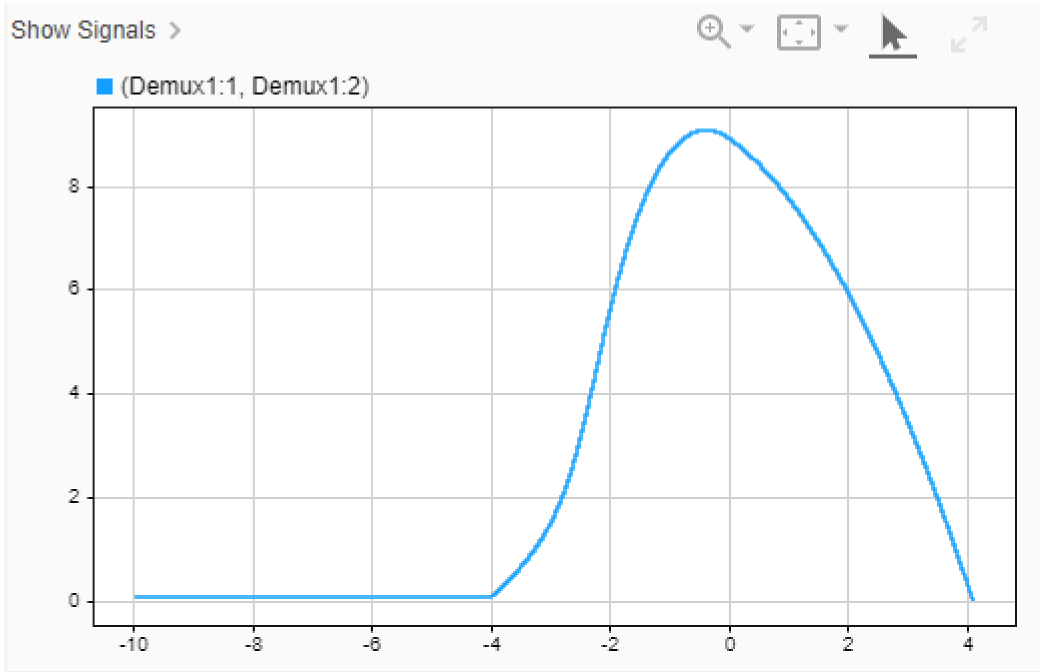


Model the Free Fall of the Pole Vaulter

When the angle of the pole vaulter, `theta`, is less than $\pi/2$, the Simulink based state `Fly` becomes the active state. This state models the pole vaulter after the jump has cleared and the pole vaulter is falling to the ground. As the pole vaulter falls, the position of the pole vaulter in the x-y plane is continuously changing, but the state of falling remains the same. In this model, the integrator blocks xydot and xy_integ are state owner blocks for State Writer blocks in the Simulink function InitFly. This subsystem outputs the Cartesian coordinates of the pole vaulter.



The Record block shows the results of this simulation.



Limitations

You cannot use Simulink based states with:

- Moore charts
- Discrete Event charts
- HDL Coder
- PLC Coder
- Simulink Code Inspector
- Super step transitions

Simulink based states do not support debugging.

See Also

More About

- “Create and Edit Simulink Based States” on page 4-8
- “Reuse Charts in Models with Chart Libraries” on page 25-34
- “Create Custom Library” (Simulink)

Create and Edit Simulink Based States

To model systems that switch between periodic or continuous time dynamics, use Simulink based states. Simulink based states are not supported in standalone Stateflow charts in MATLAB. For more information, see "Simulink Subsystems as States" on page 4-2.

You can create Simulink based state by using the object palette. To reuse systems from separate Simulink models, copy and paste enabled subsystems. To reuse subsystems in multiple Stateflow charts, copy and paste action subsystems that are saved in a library.

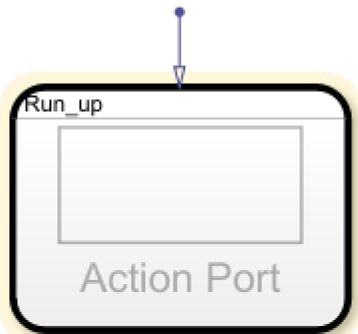
Create a Simulink Based State

To create a Simulink based state, do one of the following:

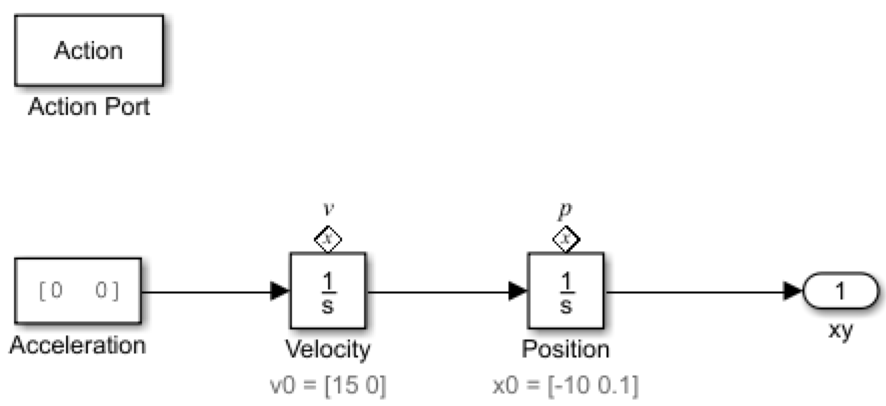
- Create an empty Simulink based state by using the Simulink based state palette icon.
- Create a Simulink based state from another model by copying an enabled subsystem or an action subsystem to your Stateflow chart.
- Create a linked Simulink based state by copying an action subsystem from a library to your Stateflow chart.

Create an Empty Simulink Based State

1. In the object palette, click the Simulink state icon .
2. On the chart canvas, click the location for the new Simulink based state.
3. Enter a name for the state. In this example, the state models a pole vaulter running along a flat surface, so the state label is Run_up. Simulink based states are action subsystems, so an Action Port appears with your new state.



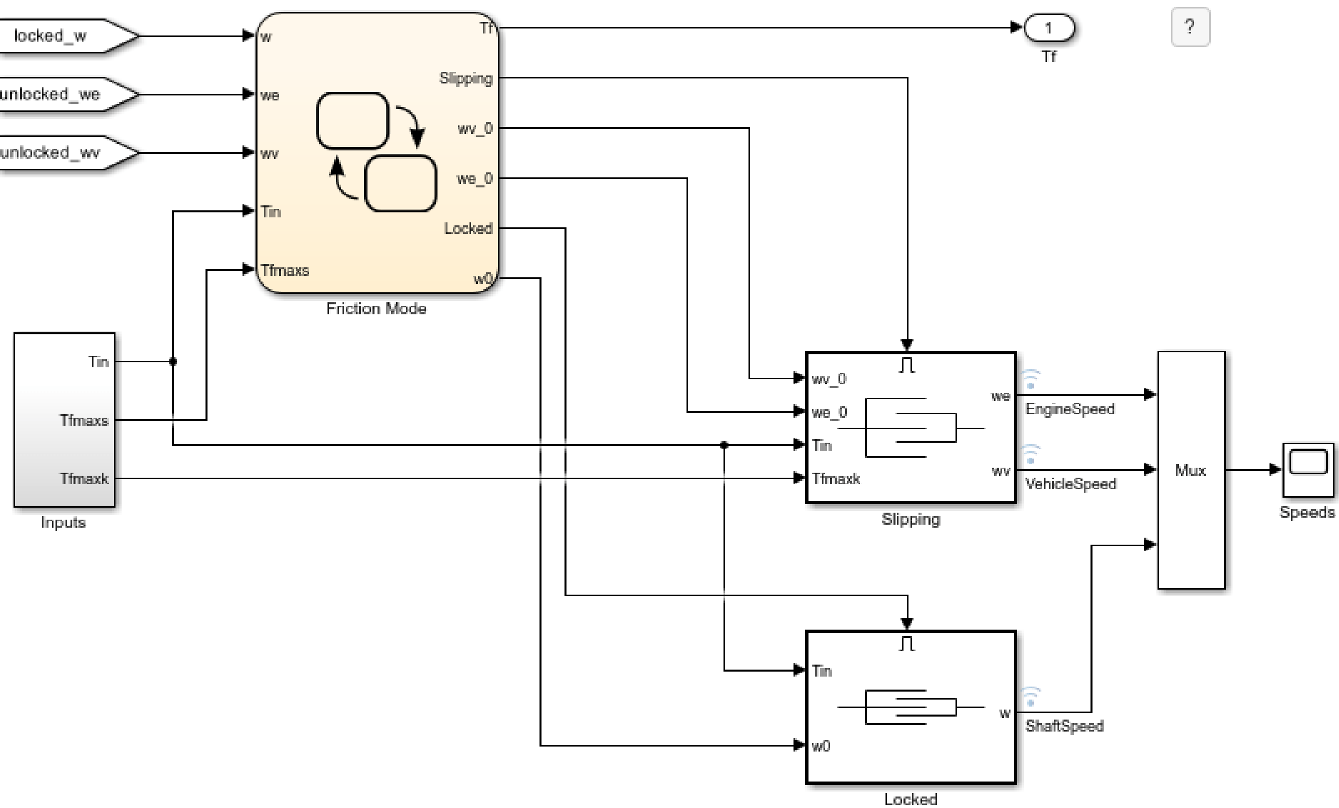
4. Build your Simulink subsystem. This subsystem outputs the Cartesian coordinates of the pole vaulter. For more information about this model, see "Access Block State Data" on page 4-14.



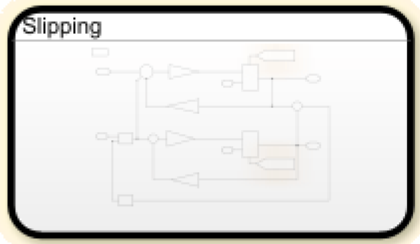
Create a Simulink Based State from an Enabled Subsystem

To create a Simulink based state in your Stateflow chart, copy enabled subsystems from separate Simulink models. You can reuse components from Simulink models in a Stateflow chart without creating a brand new Simulink based state.

1. Open the model `sf_clutch_enabled_subsystems`.



2. From the model, copy the block `Slipping` to your Stateflow chart.



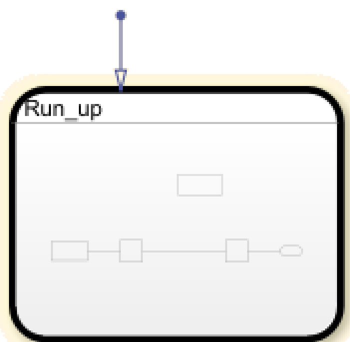
3. The inports and outports of your Simulink subsystem appear as undefined symbols in your Stateflow chart. To add corresponding input and output data to your Stateflow chart, open the **Symbols** pane and click the **Resolve undefined symbols** button. for more information, see "Resolve Symbols Through the Symbols Pane" on page 25-15.

Create a Linked Simulink Based State

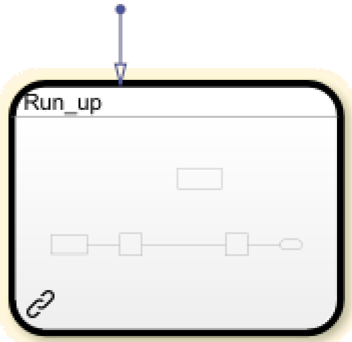
To create a linked Simulink based state in your Stateflow chart, copy an action subsystem from a library to Stateflow. When the library block is updated, the changes are reflected in all Stateflow charts containing the block.

1. Open the library model `sf_pole_vault_lib`.

   ```
   openExample("stateflow/PoleVaulterExample", ...
       supportingFile="sf_pole_vault_lib")
   ```
2. Copy and paste the library block `Run_up` to your Stateflow chart.



3. To display a link in the bottom leftmost corner on a linked subsystem, in the **Debug** tab, select **Information Overlays > Show All Links**.



4 The outports of this Simulink subsystem, `xy`, appears as an undefined symbol in your Stateflow chart. To add a corresponding output data to your Stateflow chart, click the **Resolve undefined symbols** button .

Create Inports and Outports

When using Simulink based states, inports and outports for your Simulink subsystem connect to input and output data at the Stateflow chart level. This connection allows the top-level Simulink model to read data from the subsystem contained within your Simulink based state.

When you create an empty Simulink based state, Stateflow creates inputs and outputs in your Simulink subsystem that correspond to inputs and outputs that exist in the parent Stateflow chart. However, if you add inports and outports to your Simulink based state after it is created, you must create corresponding input and output data for your Stateflow chart.

To create additional inports or outports for a Simulink based state:

1 Open your Simulink based state.
2 Click the Simulink canvas, type `in1`, and press **Enter**. An undefined inport is created.
3 The undefined symbol `in1` appears in the **Symbols** pane of your Stateflow chart. To resolve the undefined symbol, click the **Resolve undefined symbols** button .
4 A chart inport named `In1` is created.

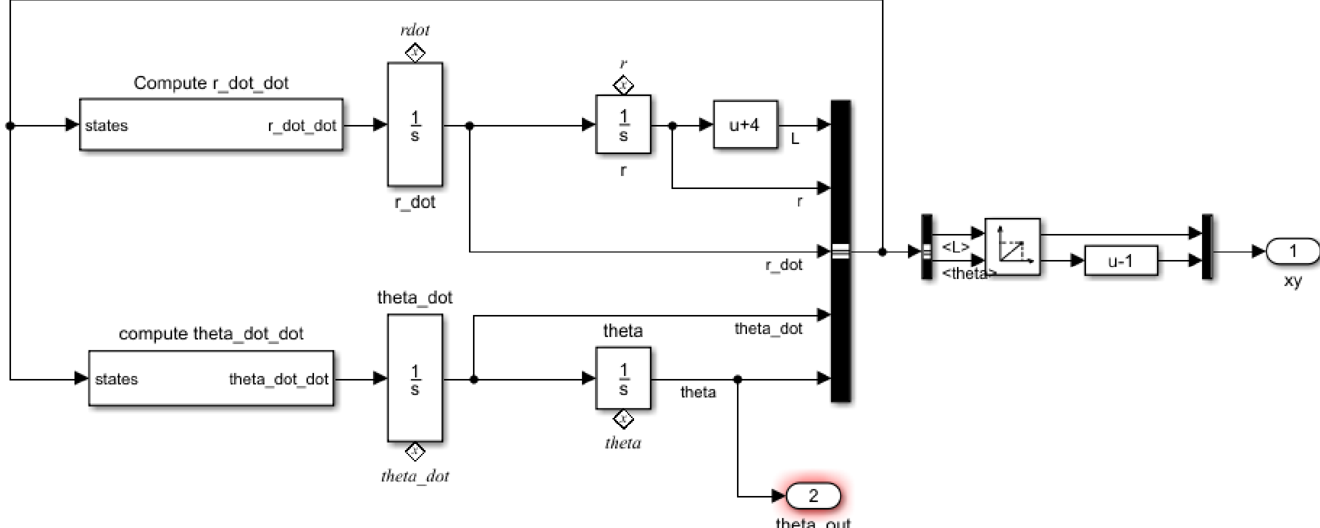
Create an Additional Outport

In this example, you create an additional outport for the model `sf_pole_vault`:

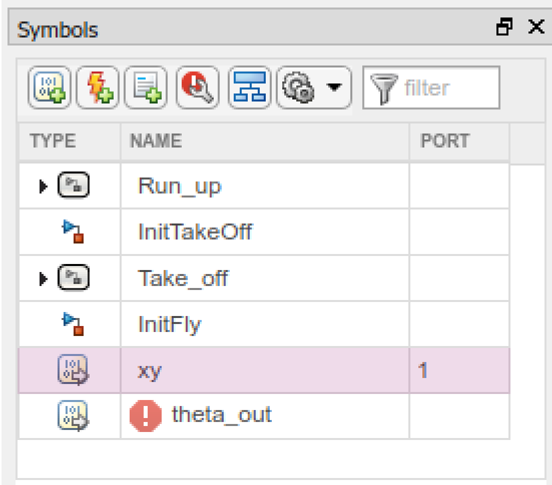
1 Open the model `sf_pole_vault`.

```
openExample("stateflow/PoleVaulterExample")
```

2 Open the chart `PoleVaulter` and double-click Simulink based state `Take_off`.
3 Click the Simulink based state canvas and type `out1` and press **Enter**. An undefined outport is created. Rename the outport `theta_out` and connect it to the signal for theta.



4 In the **Symbols** pane of `PoleVaulter`, an undefined symbol for `theta_out` appears. To resolve the undefined symbol, click the **Resolve undefined symbols** button .



5 Stateflow creates an output in the chart called `theta_out` that corresponds to the outport `theta_out`.

For more information about editing data, see “Add and Modify Data, Events, and Messages” on page 25-14.

See Also

More About

- “Simulink Subsystems as States” on page 4-2
- “Access Block State Data” on page 4-14
- “Map Variables in a Simulink Based State” on page 4-20
- “Reuse Charts in Models with Chart Libraries” on page 25-34
- “Create Custom Library” (Simulink)

Access Block State Data

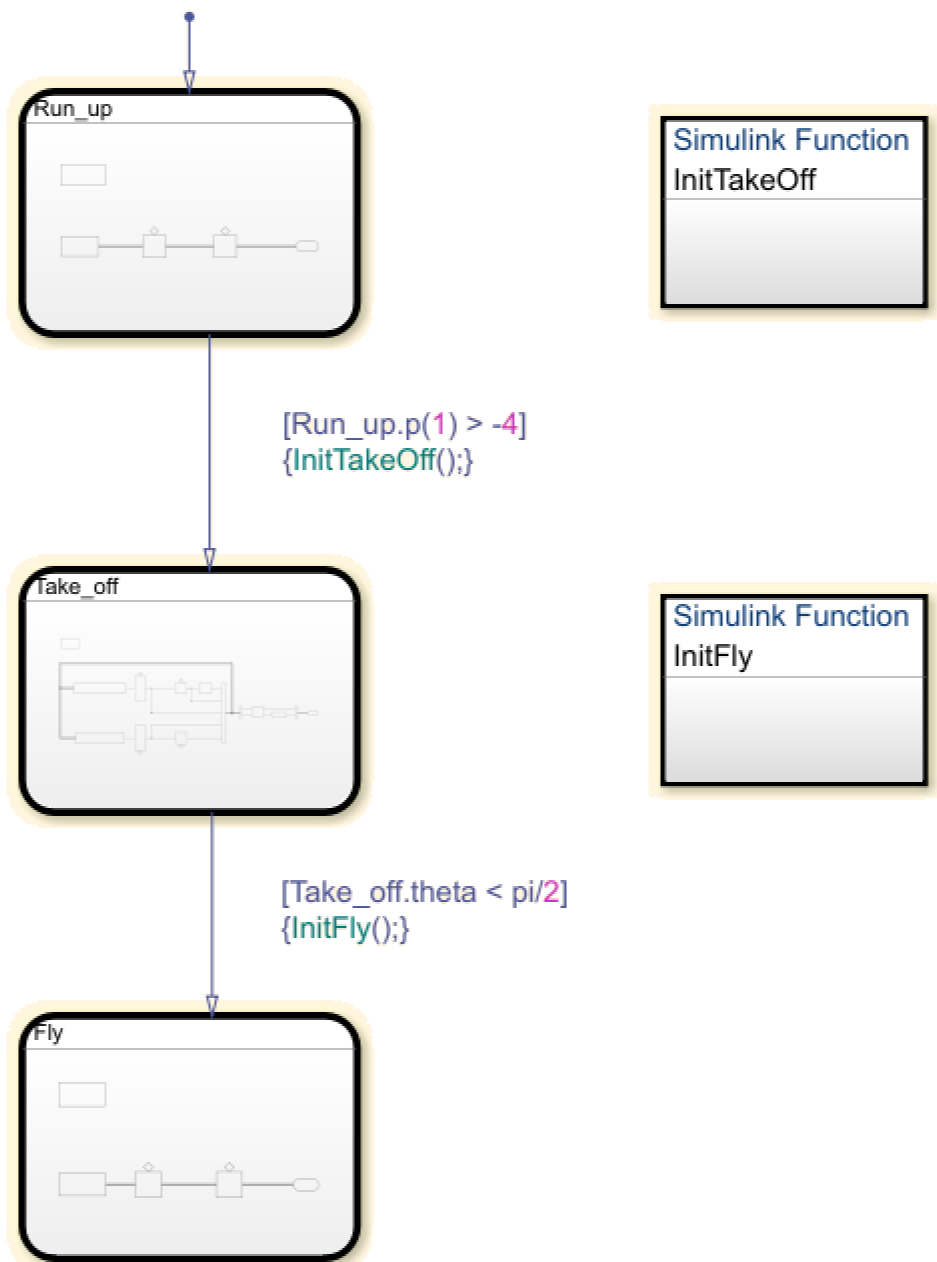
To model systems that switch between periodic or continuous time dynamics, use Simulink based states. Simulink based states are not supported in standalone Stateflow charts in MATLAB. For more information, see “Simulink Subsystems as States” on page 4-2.

You can read and write the state of blocks within your Simulink based states in transition actions of your Stateflow chart. You can read and write the state of blocks textually on the chart transitions or by using Simulink State Reader and State Writer blocks.

For example, the Stateflow chart in this example models a person moving through the stages of pole vaulting.

```
openExample("stateflow/PoleVaulterExample")
```

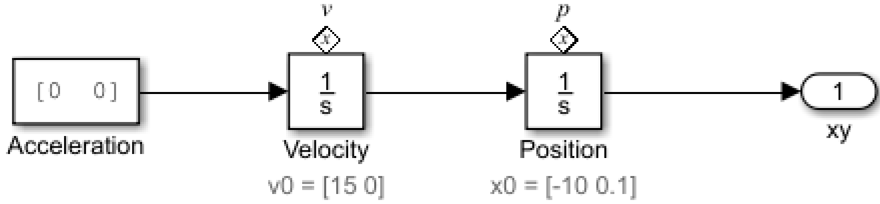
The first stage is the approach run of the vaulter, which is modeled by the Simulink based state `Run_up`. In the second stage, the vaulter plants the pole and takes off, which is modeled by the Simulink based state `Take_off`. The final stage happens when the vaulter clears the bar and releases the pole, which is modeled by the Simulink based state `Fly`.



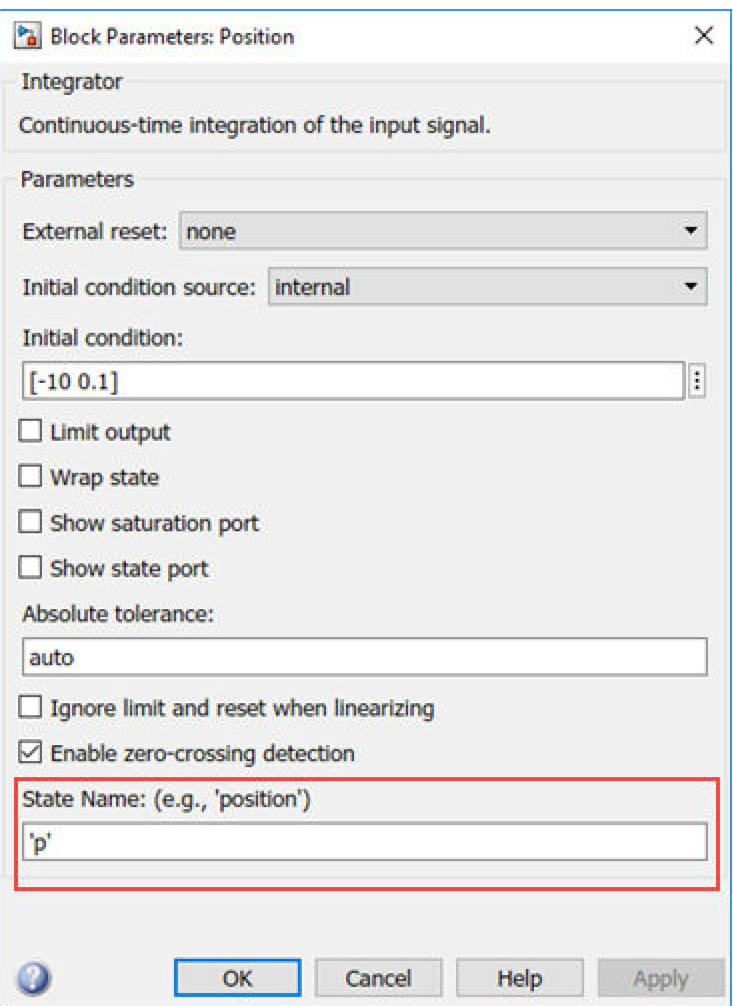
The states Run_up and Fly are easier to model by using Cartesian coordinates. The state Take_off is easer to model by using polar coordinates. The Simulink functions InitTakeOff and InitFly are used to switch from one coordinate system to another. For more information on this chart, see “Model a Pole Vaulter by Using Simulink Based States” on page 4-2.

Textual Access

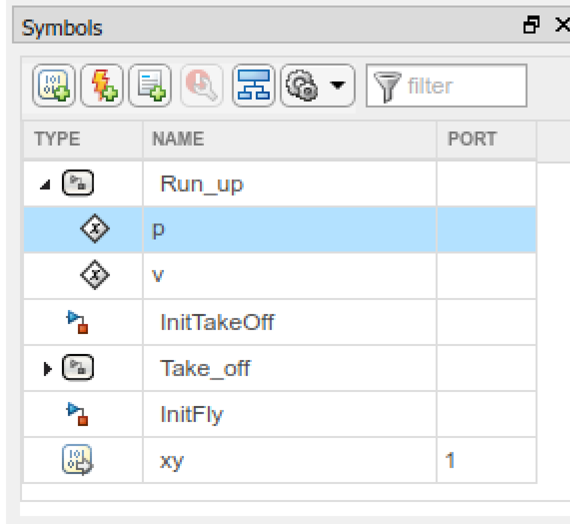
This subsystem is contained within the Simulink based state `Run_up`. For the transition from `Run_up` to `Take_off` to occur, the position of the pole vaulter along the x-axis, `p(1)`, must be greater than `-4`.



By setting the State Name of the integrator block `Position` to `'p'`, you can textually access the state of this block from your Stateflow chart. To access the state of the integrator block in the transition condition, type `[Run_up.p(1)> -4]`. When this condition becomes true, the transition is taken and the active state becomes `Take_off`.



In the **Symbols** pane, you can see that the state p appears under the state Run_up.



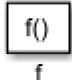
Graphical Access

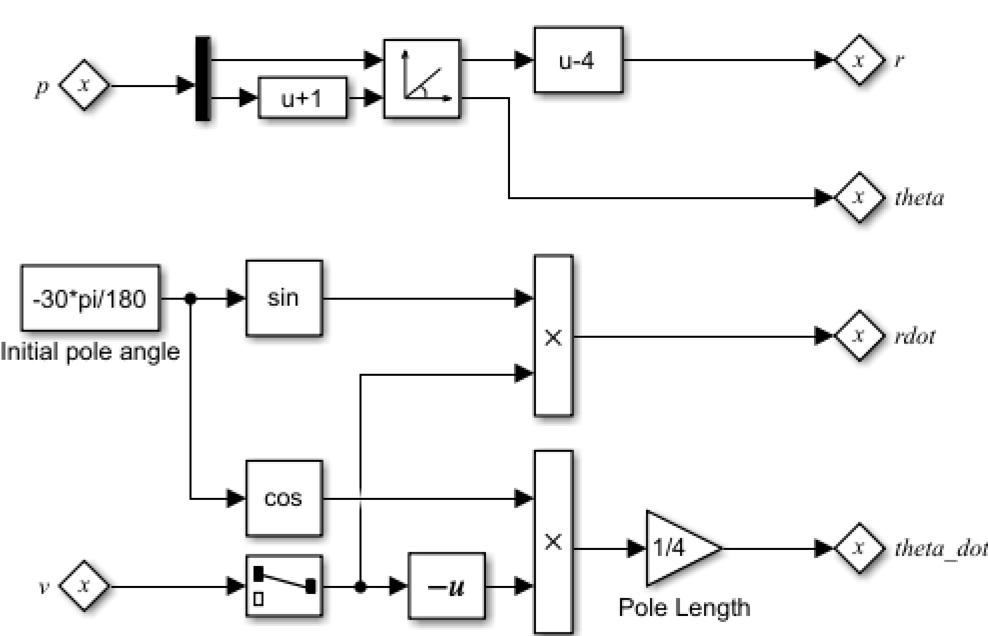
Stateflow uses State Reader and State Writer blocks to connect the subsystems within a Simulink based state to other Simulink subsystems in your model. State Reader and State Writer blocks display the name of the state owner block that they are connected to. Conversely, the state owner block displays a tag indicating a link to a State Reader or a State Writer block. If you click the label above the tag, a list opens with a link for navigating to the State Writer block.

Connect a State Reader Block to an Owner Block

The following subsystem is contained within the Simulink function `InitTakeOff`. The function uses State Reader blocks to connect to the state `Run_up` and reads `p` and `v`. The function then converts the Cartesian values for the position of the pole vaulter and velocity into polar coordinates, `r` and `theta` and `rdot` and `theta_dot`, respectively. These polar coordinates are then accessed by using state owner blocks in the state `Take_off`.

When the transition action occurs, the State Reader blocks in `InitTakeOff` read the state of their state owner blocks. Once the Simulink function finishes executing, the State Writer blocks write to the state owner blocks in the Simulink based state `Take_off`.

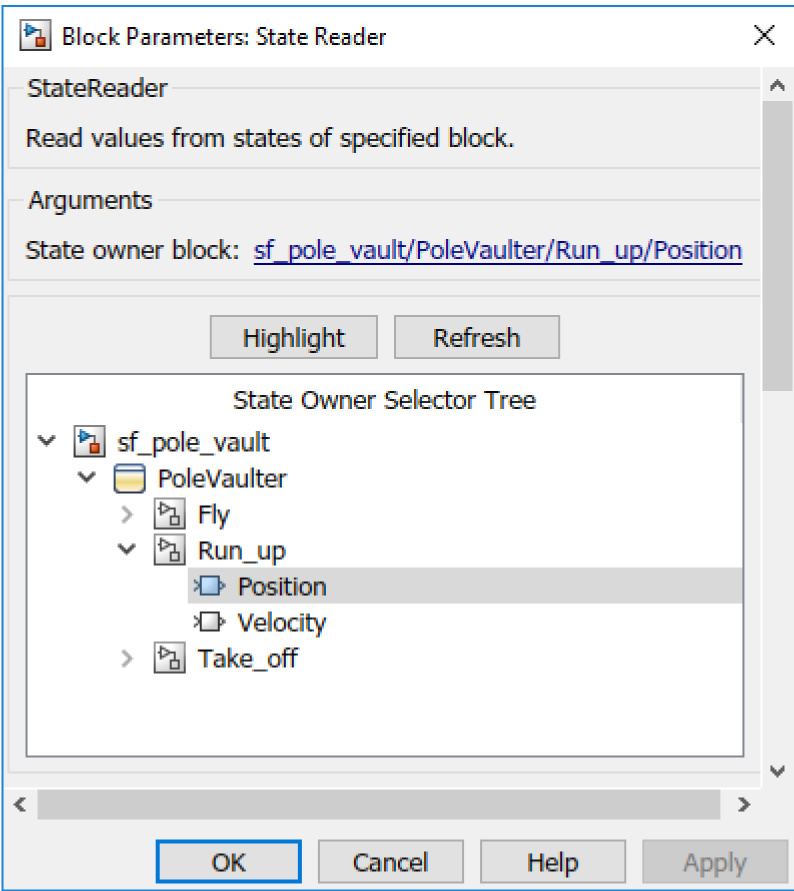




To connect a State Reader or a State Writer block to an owner block within a Simulink subsystem:

1 To open the properties, double-click the State Reader.

2 In the **State Owner Selector Tree**, navigate to the block that you want to be the state owner block. In this example, by choosing `Position`, you connect the State Reader block to the integrator `Position` in the state `Run_up`.



3 By connecting the State Reader block to the `Position` integrator block, this Simulink function can use the state of the integrator `Position` to execute.

See Also

More About

- "Simulink Subsystems as States" on page 4-2
- State Reader (Simulink)
- State Writer (Simulink)

Map Variables for Simulink Based States

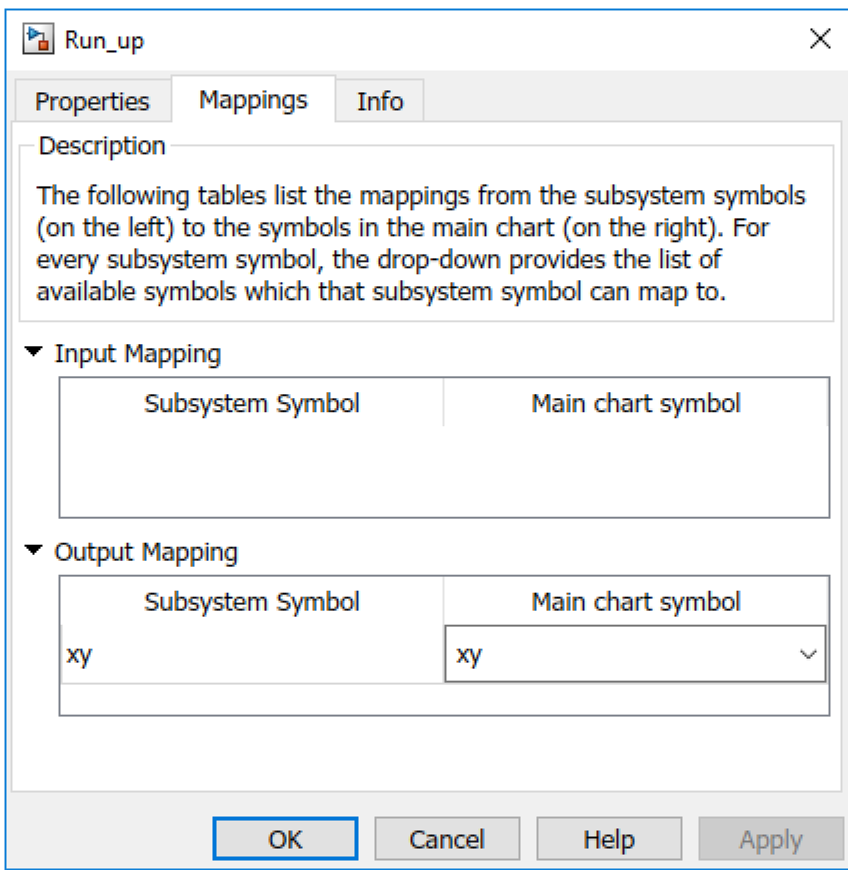
To model systems that switch between periodic or continuous time dynamics, use Simulink based states. Simulink based states are not supported in standalone Stateflow charts in MATLAB. For more information, see “Simulink Subsystems as States” on page 4-2.

You can access inports or outports of a subsystem within a Simulink based state by using inputs and outputs in Stateflow that have the same name as your inports and outports. For Simulink based states that are created by copying and pasting enabled subsystems and action subsystems from a library, click the **Resolve undefined symbols** button to map your Simulink inports and outports to Stateflow inputs and outputs automatically. See “Create Inports and Outports” on page 4-11.

If you are using a linked Simulink based state where the name of the inport or outport differs from the Stateflow chart input or output, you must ensure that your variables are mapped correctly. You can change your mappings from the **Property Inspector** or in the Mappings dialog box.

Map Variables in a Simulink Based State

To open the mappings dialog box, select the Simulink Based State. In the **Simulink State** tab, click **Mappings**.



Under **Input Mapping**, you can specify which parent chart input maps to an inport in your Simulink subsystem.

Under **Output Mapping**, you can specify which parent chart output maps to an outport in your Simulink subsystem.

See Also

More About

- "Simulink Subsystems as States" on page 4-2
- "Create and Edit Simulink Based States" on page 4-8
- "Access Block State Data" on page 4-14
- "Manage Symbols in the Stateflow Editor" on page 25-14

Set Simulink Based State Properties

Simulink based states are Simulink subsystems within a Stateflow state that enable you to model systems that switch between periodic or continuous time dynamics. For more information, see "Simulink Subsystems as States" on page 4-2.

You can specify properties for a Simulink based state in the **Property Inspector**, the Model Explorer, or the Simulink based state properties dialog box.

To use the **Property Inspector**:

1 In the **Modeling** tab, under **Design Data**, select **Property Inspector**.
2 In the Stateflow Editor, select the Simulink based state.
3 In the **Property Inspector**, edit the properties of the Simulink based state.

To use the Model Explorer:

1 In the **Modeling** tab, under **Design Data**, select **Model Explorer**.
2 In the **Model Hierarchy** pane, select the Simulink based state.
3 In the **Dialog** pane, edit the properties of the Simulink based state.

To use the Simulink Based State properties dialog box:

1 In the Stateflow Editor, right-click the Simulink based state.
2 Select **Properties**.
3 In the properties dialog box, edit the Simulink based state properties.

You can also specify Simulink based state properties programmatically by using `Stateflow.SimulinkBasedState` objects. For more information about the Stateflow programmatic interface, see "Overview of the Stateflow API".

Simulink Based State Properties

Create data for monitoring self activity

Creates a data output port on the Stateflow block for this self-activity of the state.

Log self activity

Logs the state self-activity. View the activity of the state in the Simulation Data Inspector.

Logging name

Specify the signal logging name. To create a signal logging name that is different from the state name, choose `Custom`, and add the name.

Test point

Sets the Simulink based state as a Stateflow test point. For more information, see "Monitor Test Points in Stateflow Charts" on page 11-38.

Limit data points to last

Maximum number of data points to log. Default value is 5000, which means the chart logs the last 5000 data points generated by the simulation.

Decimation

Decimation interval limits the amount of data logged by skipping samples. Default value is 2, which means the chart logs every other sample.

Function packaging

Specify the code format generated for a Simulink based state. You can set the format to one of these options:

- **Auto** — Simulink Coder software chooses the optimal format for you based on the type and number of instances of the subsystem that exist in the model.
- **Inline** — Simulink Coder software inlines the subsystem unconditionally.
- **Nonreusable function** — Simulink Coder software explicitly generates a separate function in a separate file.
- **Reusable function** — Simulink Coder software generates a function with arguments that allows reuse of Simulink based state code when a model includes multiple instances of the Simulink based state.

For more information, see “Function packaging” (Simulink).

Description

Description of the Simulink based state.

Document Link

Link to online documentation for the Simulink based state. You can enter a web URL address or a MATLAB command that displays documentation as an HTML file or as text in the MATLAB Command Window. When you click the **Document link** hyperlink, Stateflow evaluates the link and displays the documentation.

See Also

Objects
`Stateflow.SimulinkBasedState`

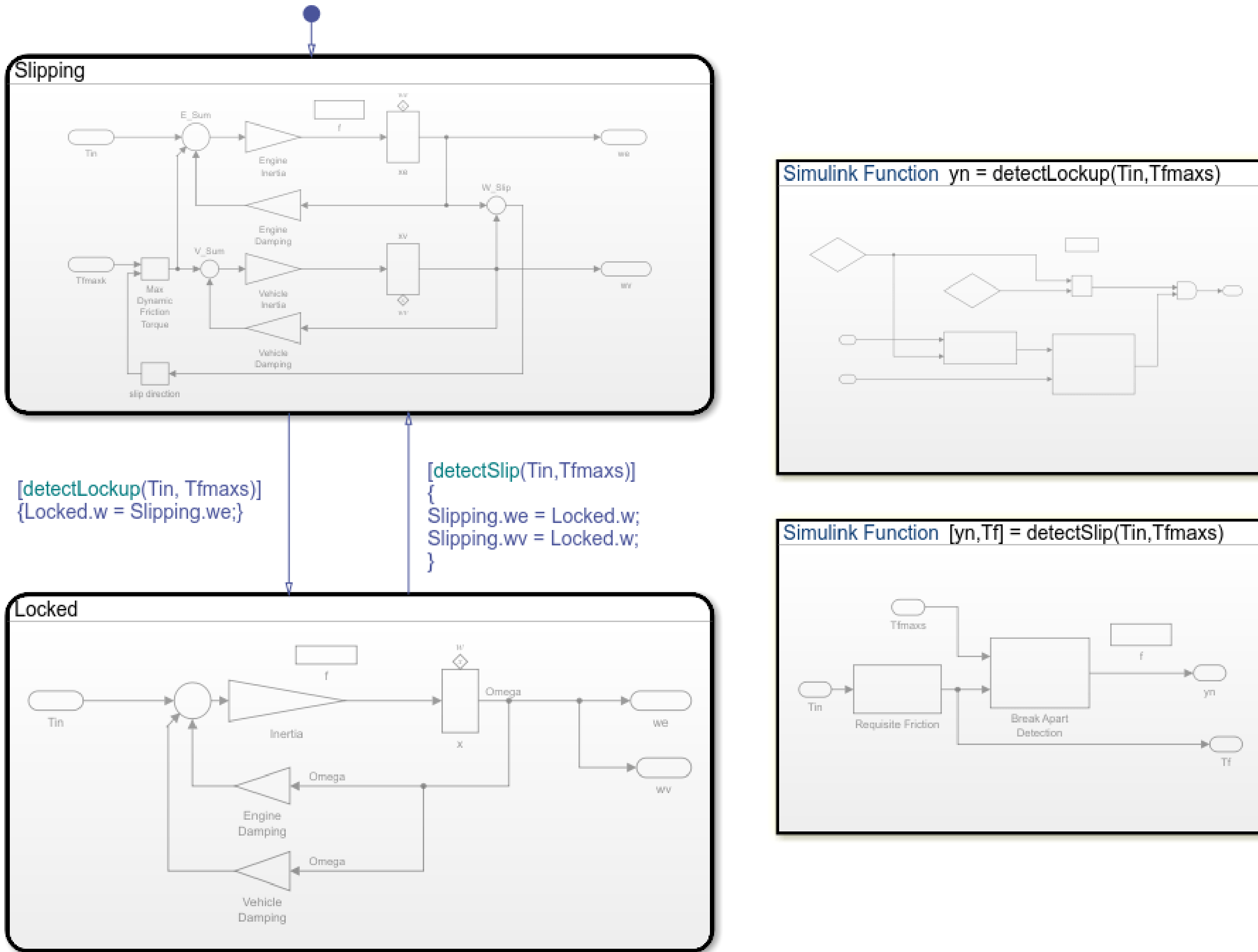
Tools
Model Explorer

More About

- “Guidelines for Naming Stateflow Objects” on page 1-66
- “Inspect and Modify Data and Messages While Debugging” on page 30-8
- “Specify Size of Stateflow Data” on page 10-26
- “Specify Type of Stateflow Data” on page 10-20

Hybrid Clutch System

This example shows how to use Simulink based states inside a Stateflow® chart to model a clutch system. For a detailed explanation of the physical system, see “Building a Clutch Lock-Up Model” (Simulink).



Recommended Workflow

This model shows the recommended way of modeling hybrid systems by using Simulink and Stateflow. This model also covers when to use Simulink or physical modeling tools if the continuous dynamics are complex coupled with mode changes.

Modeling a hybrid system involves addressing the following concerns:

- Modeling the continuous dynamics
- Modeling the mode logic
- Initializing states when switching between modes

Continuous Dynamics

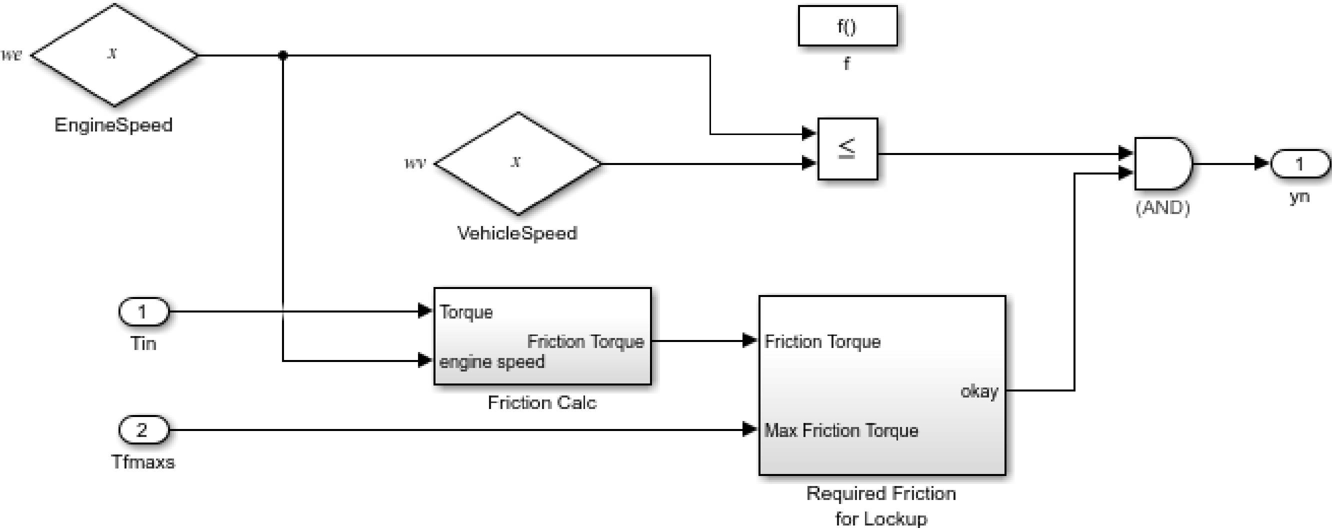
Hybrid systems have multiple modes of operation where each mode is defined by continuous dynamics. When the continuous dynamics are complex, model them by using Simulink based states. In this model, the `Locked` and `Slipping` states represent the two modes of operation of a clutch. Simulink blocks within a Simulink based state represent the logic of the state. These blocks include continuous time blocks, such as integrators. Within each Simulink based state, you can access chart inputs and outputs by creating inports and outports with the same name. Each Simulink based state reads from a subset of chart inputs and writes to all the chart outputs.

Mode Logic

Mode logic refers to the conditions under which the model switches from one mode of operation to another. In this example, the mode logic is described by the transition logic between the two Simulink based states. The conditions needed to switch from one Simulink based state to another depend on the internal state of the integrators within in the current active mode. For example, when switching from `Slipping` to `Locked` Stateflow must read the internal state of the integrator in the `Slipping` mode.

This is possible using two different mechanisms:

1. **Using State Reader and State Writer blocks inside Simulink functions**: Stateflow can call Simulink functions on the transition logic between the two modes. Inside the Simulink function, use State Reader blocks to refer to the internal state of the integrator. For example, the Simulink function `detectLockup` uses the State Reader block `EngineSpeed` to read the state of the integrator block `sf_clutch/Clutch/Slipping/xe`.



2. **Using qualified dot notation on the transition conditions**: If the transition logic is simple and can be expressed textually, it is possible to use a syntax like `Slipping.we == ...` to refer to the state of the integrator `sf_clutch/Clutch/Slipping/xe`. For this syntax to work, the `State Name` parameter of the integrator has to be set to "we".

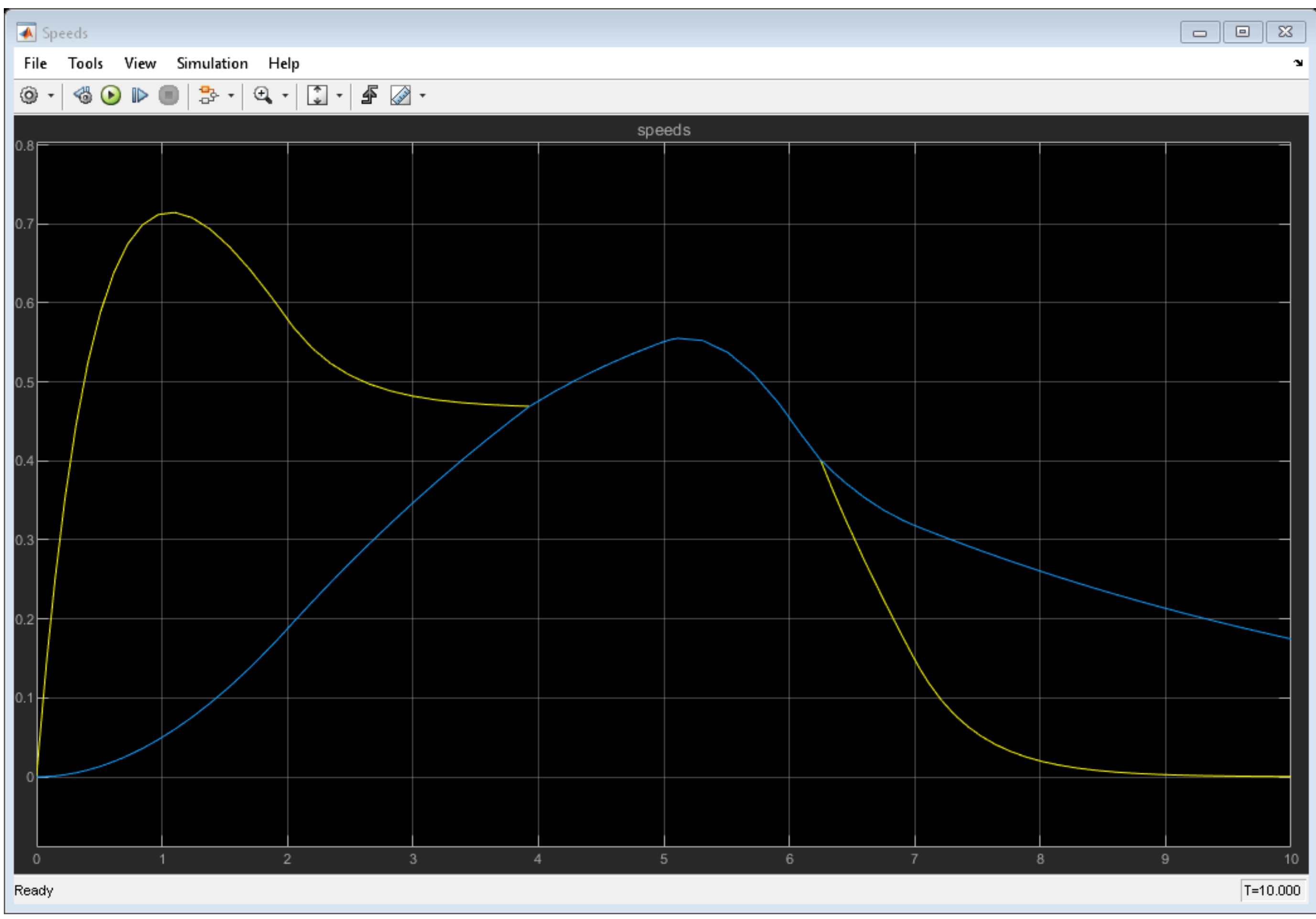
State Handoff

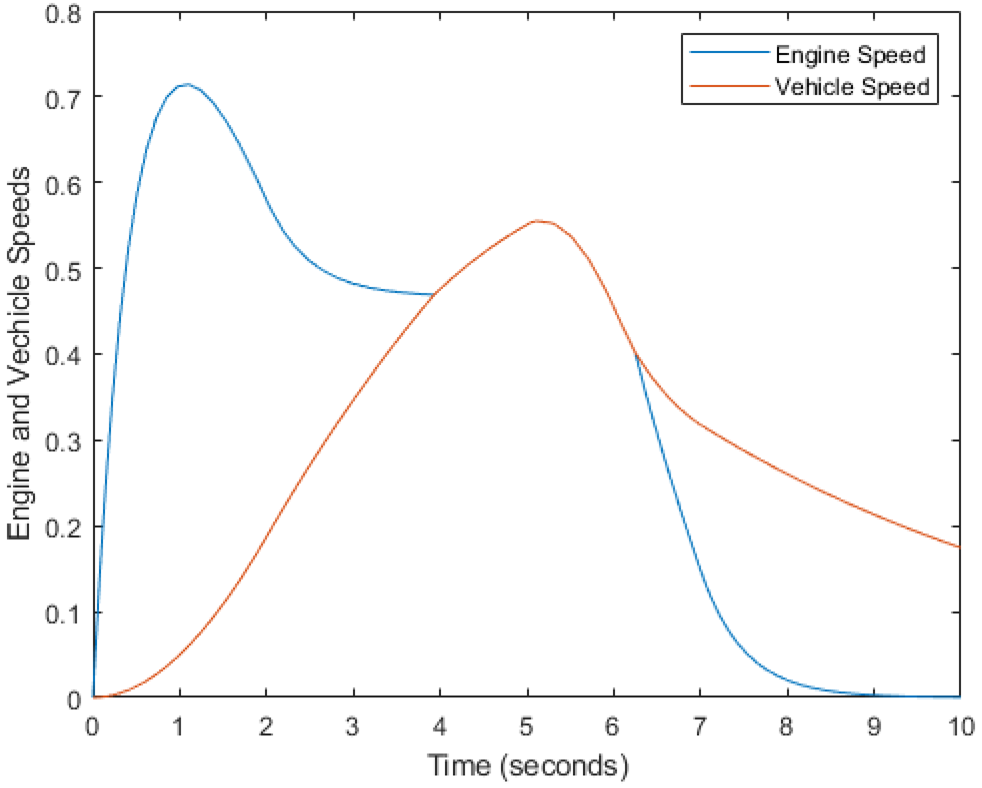
When switching from one mode of operation to another, the integrators in the newly activated subsystem need to be initialized properly in order to get smooth output. This can be done using either Simulink State Reader and State Writer blocks in Simulink functions or textually using the qualified dot notation. For example, on the transition from `Slipping` to `Locked`, initialize the state of the single integrator in `Locked` by using the state of one of the integrators in `Slipping`. Initialize the state by using the syntax:

```
Locked.w = Slipping.we;
```

Simulation Results

When the system is simulated, the engine and vehicle velocities are as shown in the following graph. The plates lock at about 4 seconds and begin slipping again at about 6.25 seconds.





See Also

More About

- "Continuous-Time Modeling in Stateflow" on page 22-2
- "Building a Clutch Lock-Up Model" (Simulink)

Build Mealy and Moore Charts

- “Overview of Mealy and Moore Machines” on page 5-2
- “Design Considerations for Mealy Charts” on page 5-5
- “Model a Vending Machine by Using Mealy Semantics” on page 5-7
- “Design Considerations for Moore Charts” on page 5-9
- “Model a Traffic Light by Using Moore Semantics” on page 5-12
- “Convert Charts Between Mealy and Moore Semantics” on page 5-14
- “Sequence Recognition by Using Mealy and Moore Charts” on page 5-18
- “Karplus-Strong Algorithm by Using Moore Charts” on page 5-22
- “Initialize Persistent Variables in MATLAB Functions” on page 5-24

Overview of Mealy and Moore Machines

In a finite state machine, *state* is a combination of local data and chart activity. "Computing state" means updating local data and making transitions from a currently active state to a new state. In state machine models, the next state is a function of the current state and its inputs:

$$X(n + 1) = f(X(n), u)$$

In this equation:

- *X(n)* represents the state at time step *n*.
- *X(n+1)* represents the state at the next time step *n+1*.
- *u* represents inputs.

State persists from one time step to the next time step.

Semantics of Mealy and Moore Machines

Mealy and Moore machines are often considered the basic, industry-standard paradigms for modeling finite-state machines. You can create charts that implement pure Mealy or Moore semantics as a subset of Stateflow chart semantics. You can use Mealy and Moore charts in simulation and code generation with Embedded Coder®, Simulink Coder, and HDL Coder™ software. Mealy and Moore semantics are not supported in standalone Stateflow charts in MATLAB.

Semantics of Mealy Charts

Mealy machines are finite state machines in which transitions occur on clock edges. The output of a Mealy chart is a function of inputs and state:

$$y = g(X, u)$$

At every time step, a Mealy chart wakes up, evaluates its input, and then transitions to a new configuration of active states, also called its *next state*. The chart computes its output as it transitions to the next state.

To ensure that output is a function of input and state, Mealy state machines enforce these semantics:

- Outputs do not depend on the next state.
- The chart computes outputs only in transitions, not in states.
- The chart wakes up periodically based on a system clock.

Mealy machines compute their output on transitions. Therefore, Mealy charts can compute their first output at the time that the default path for the chart executes. If you enable the chart property **Execute (enter) Chart At Initialization** for a Mealy chart, this computation occurs at $t = 0$ (first time step). Otherwise, it occurs at $t = 1$ (next time step). For more information, see “Execute (enter) chart at initialization” on page 1-21.

Semantics of Moore Charts

Moore machines are finite state machines in which output is modified at clock edges. The output of a Moore chart is a function only of state:

$$y = g(X)$$

At every time step, a Moore chart wakes up, computes its output, and then evaluates its input to reconfigure itself for the next time step. For example, after evaluating its input, the chart can transition to a new configuration of active states. The chart computes its output before evaluating its input and updating its state.

To ensure that output is a function only of the current state, Moore state machines enforce these semantics:

- Outputs do not depend on inputs.
- Outputs do not depend on previous outputs.
- Outputs do not depend on temporal logic.

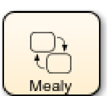
Moore machines compute their output in states. Therefore, Moore machines can compute outputs only *after* the default path executes. Until then, outputs take the default values.

Create Mealy and Moore Charts

When you create a Stateflow chart, the default type is a hybrid state machine model called a *Classic* chart. Classic charts combine the semantics of Mealy and Moore charts with the extended Stateflow chart semantics.

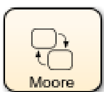
To create a Mealy chart, at the MATLAB command prompt, enter:

```
sfnew -Mealy
```



To create a Moore chart, at the MATLAB command prompt, enter:

```
sfnew -Moore
```



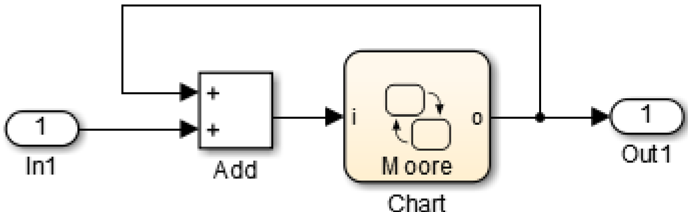
Alternatively, after adding a Stateflow chart block to a Simulink model, you can choose the type of semantics for the chart by setting the **State Machine Type** chart property. For more information, see “State Machine Type” on page 1-20.

Advantages of Mealy and Moore Charts

Mealy and Moore charts offer these advantages over Classic Stateflow charts:

- You can verify that the Mealy and Moore charts you create conform to their formal definitions and semantic rules. Error messages appear at compile time (not at design time).
- Moore charts provide a more efficient implementation than Classic charts for C/C++ and HDL targets.
- You can use a Moore chart to model a feedback loop. In Moore charts, inputs do not have direct feedthrough. You can design a loop with feedback from the output port to the input port without

introducing an algebraic loop. Mealy and Classic charts have direct feedthrough and produce an error in the presence of an algebraic loop.



See Also

`sfnew`

More About

- “Design Considerations for Mealy Charts” on page 5-5
- “Design Considerations for Moore Charts” on page 5-9
- “Sequence Recognition by Using Mealy and Moore Charts” on page 5-18
- “Model a Vending Machine by Using Mealy Semantics” on page 5-7
- “Model a Traffic Light by Using Moore Semantics” on page 5-12
- “Convert Charts Between Mealy and Moore Semantics” on page 5-14

Design Considerations for Mealy Charts

Mealy machines are finite state machines in which transitions occur on clock edges. In Mealy charts, output is a function of input and state. At every time step, a Mealy chart wakes up, evaluates its input, and then transitions to a new configuration of active states, also called its *next state*. The chart computes its output as it transitions to the next state. Mealy semantics are not supported in standalone Stateflow charts in MATLAB.

Mealy Semantics

To ensure that output is a function of input and state, Mealy state machines enforce these semantics:

- Outputs do not depend on the next state.
- The chart computes outputs only in transitions, not in states.
- The chart wakes up periodically based on a system clock.

Note A chart provides one time base for input and clock (see "Calculate Output and State by Using One Time Base" on page 5-6).

Design Guidelines for Mealy Charts

To conform to the Mealy definition of a state machine, ensure that every time there is a change on the input port, the chart computes outputs.

Compute Outputs in Condition Actions Only

You can compute outputs only in the condition actions of outer and inner transitions. A common modeling style for Mealy machines is to test inputs in conditions and compute outputs in the associated action.

Do Not Use State Actions or Transition Actions

You cannot use state actions or transition actions in Mealy charts. This restriction enforces Mealy semantics by:

- Preventing the chart from computing output without considering changes on the input port.
- Ensuring that output depends on the current state and not the next state.

Do Not Use Data Store Memory

You cannot use data store memory (DSM) in Mealy charts because objects external to the chart can modify DSM. A Stateflow chart uses data store memory to share data with a Simulink model. Data store memory acts as global data. In the Simulink hierarchy that contains the chart, other blocks and models can modify DSM. Mealy charts must not access data that can change unpredictably.

Restrict Use of Events

Limit the use of events in Mealy charts:

- **Valid Uses:**

- Use input events to trigger the chart.
- Use event-based temporal logic to guard transitions.

  The change in value of a temporal logic condition behaves like an event that the Mealy chart schedules internally. At each time step, the number of ticks before the temporal event executes depends only on the state of the chart. For more information, see “Temporal Logic Operators” on page 14-35.

  Note In Mealy charts, the base event for temporal logic operators must be a predefined event such as `tick` (see “Implicit Events Based on Data and States” on page 12-28).

- **Invalid Uses:**

  - You cannot broadcast an event of any type.
  - You cannot use local events to guard transitions. Local events violate Mealy semantics because they are not deterministic and can occur while the chart computes its outputs.
  - You cannot use implicit events such as `chg(data_name)`, `en(state_name)`, or `ex(state_name)`.

Calculate Output and State by Using One Time Base

You can use one time base for clock and input, as determined by the Simulink solver. The Simulink solver sets the clock rate to be fast enough to capture input changes. As a result, a Mealy chart commonly computes outputs and changes states in the same time step. For more information, see “Compare Solvers” (Simulink).

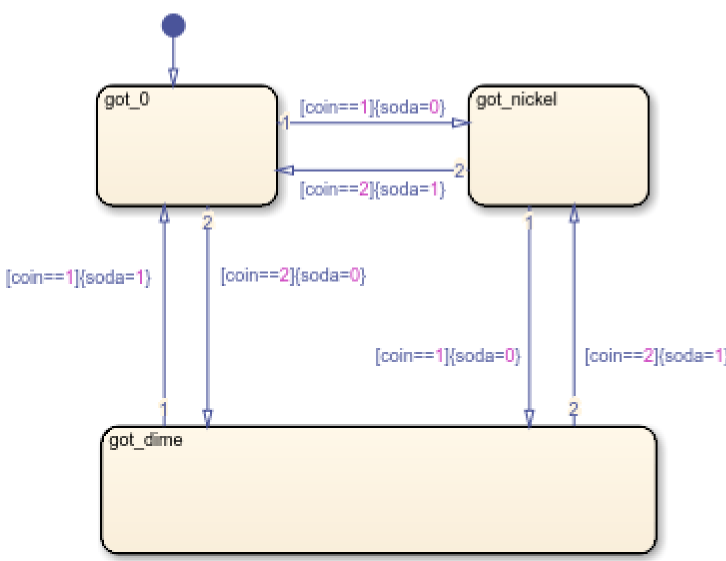
See Also

More About

- “Overview of Mealy and Moore Machines” on page 5-2
- “Model a Vending Machine by Using Mealy Semantics” on page 5-7
- “Convert Charts Between Mealy and Moore Semantics” on page 5-14

Model a Vending Machine by Using Mealy Semantics

This example shows how to use Mealy semantics to model a vending machine. Mealy charts compute outputs only in transitions, not in states. For more information, see "Design Considerations for Mealy Charts" on page 5-5.



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:

`got_0` — 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`).

`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`).

`got_dime` — A dime was received.

- If a nickel is received (`coin == 1`), a can is released (`soda = 1`), the coins are banked, and the active state becomes `got_0` because no coins are left.
- If a dime is received (`coin == 2`), a can is released (`soda = 1`), 15 cents are banked, and the active state becomes `got_nickel` because a nickel (change) is left.
- If input coin is not a dime or a nickel, state `got_dime` stays active and no can is released (output `soda = 0`).

Design Rules in Mealy Vending Machine

This example of a Mealy vending machine illustrates these Mealy design rules:

- The chart computes outputs in condition actions.
- There are no state actions or transition actions.
- The value of the input `coin` determines the value of the output `soda`.

See Also

More About

- "Overview of Mealy and Moore Machines" on page 5-2
- "Design Considerations for Mealy Charts" on page 5-5
- "Convert Charts Between Mealy and Moore Semantics" on page 5-14

Design Considerations for Moore Charts

Moore machines are finite state machines in which output is modified at clock edges. In Moore charts, output is a function of the current state only. At every time step, a Moore chart wakes up, computes its output, and then evaluates its input to reconfigure itself for the next time step. For example, after evaluating its input, the chart can transition to a new configuration of active states. The chart computes its output before evaluating its input and updating its state. Moore semantics are not supported in standalone Stateflow charts in MATLAB.

Moore Semantics

To ensure that output is a function *only* of the current state, Moore state machines enforce these semantics:

- Outputs do not depend on inputs.
- Outputs do not depend on previous outputs.
- Outputs do not depend on temporal logic.

Design Guidelines for Moore Charts

To conform to the Moore definition of a state machine, ensure that every time that a Moore chart wakes up, it computes outputs from the current set of active states without regard to inputs.

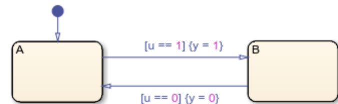
Restrictions on State Actions

To ensure that outputs depend solely on the current state, you can compute outputs in state actions, subject to these restrictions:

- **Combine Actions.** In Moore charts, you can include only one action per state. The action can consist of multiple command statements. Stateflow evaluates states in Moore charts from the top level down. Active states in Moore charts execute the state action before evaluating the transitions. Therefore, outputs are computed at each time step whether an outer transition is valid or not.
- **Do Not Label State Actions.** Do not label state actions in Moore charts with any keywords, such as `entry`,`during`, or `exit`. By default, Moore charts execute the actions in the active states before evaluating inputs and updating state.

Restrictions on Transition Actions

Transitions in Moore charts can contain condition and transition actions if these actions do not introduce a dependency between output values and input values. For example, in this chart, each transition tests the input `u` in a condition and modifies the output `y` in a condition action. Because the output value depends on the value of the input, this construct violates Moore semantics and triggers an error.



Do Not Use Inputs to Compute Outputs

In Moore charts, outputs cannot depend on inputs. Using an input to contribute directly or indirectly to the computation of an output triggers an error.

Do Not Use Data Store Memory

You cannot use data store memory (DSM) in Moore charts because objects external to the chart can modify DSM objects. A Stateflow chart uses data store memory to share data with a Simulink model. Data store memory acts as global data. In the Simulink hierarchy that contains the chart, other blocks and models can modify DSM. Moore charts must not access data that can change unpredictably.

Do Not Use coder.extrinsic to Call Extrinsic Functions

You cannot call extrinsic functions with `coder.extrinsic` in Moore charts because it is not possible to enforce that the outputs of extrinsic functions depend only on the current state. Calling an extrinsic function with `coder.extrinsic` in a Moore chart triggers an error.

Do Not Call Custom Code Functions

You cannot call custom code functions in Moore charts because it is not possible to enforce that the outputs of custom code functions depend only on the current state. Calling a custom code function in a Moore chart triggers an error.

Do Not Use Simulink Functions

You cannot use Simulink functions in Moore charts. This restriction prevents violations of Moore semantics during chart execution.

Do Not Export Functions

You cannot export functions in a Moore chart.

Do Not Disable Inlining

Moore chart semantics require inlining.

Do Not Enable Super Step Semantics

You cannot use super step semantics in a Moore chart.

Do Not Use Messages

You cannot use messages in a Moore chart.

Restrict Use of Events

Limit the use of events in Moore charts:

- **Valid Uses:**
  - Use only one input event to trigger the chart.
  - Use event-based temporal logic to guard transitions.

    The change in value of a temporal logic condition behaves like an event that the Moore chart schedules internally. At each time step, the number of ticks before the temporal event executes

depends only on the state of the chart. For more information, see “Temporal Logic Operators” on page 14-35.

Note In Moore charts, the base event for temporal logic operators must be a predefined event such as `tick` (see “Implicit Events Based on Data and States” on page 12-28).

- **Invalid Uses:**
  - You cannot broadcast an event of any type.
  - You cannot use local events to guard transitions. Local events violate Moore semantics because they are not deterministic and can occur while the chart computes its outputs.
  - You cannot use implicit events such as `chg(data_name)`, `en(state_name)`, or `ex(state_name)`.

Do Not Use Moore Charts for Modeling Continuous-Time Systems

In Moore charts, you cannot set the update method to `Continuous`. For modeling systems with continuous time in Stateflow, use Classic or Mealy charts.

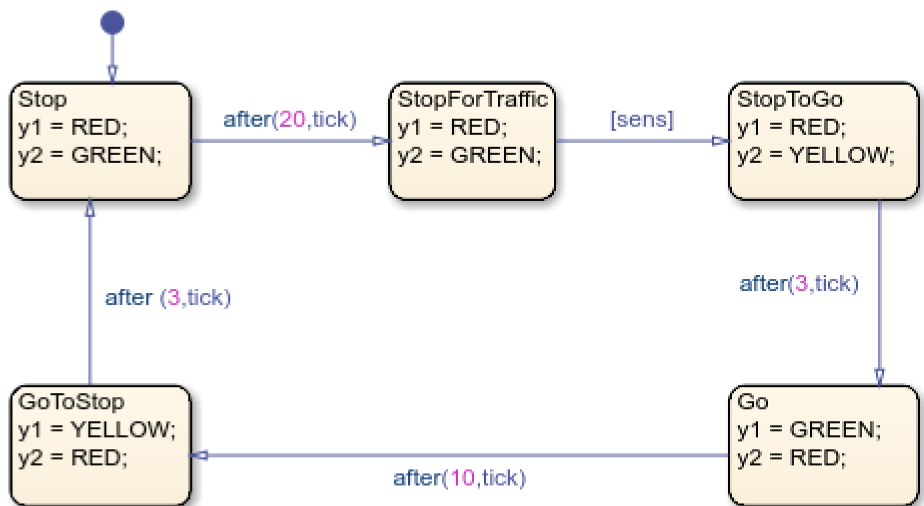
See Also

More About

- “Overview of Mealy and Moore Machines” on page 5-2
- “Model a Traffic Light by Using Moore Semantics” on page 5-12
- “Convert Charts Between Mealy and Moore Semantics” on page 5-14

Model a Traffic Light by Using Moore Semantics

This example shows how to use Moore semantics to model a traffic light. Moore charts compute outputs only in states, not in transitions. For more information, see “Design Considerations for Moore Charts” on page 5-9.



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:

`Stop` — 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`.

`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`.

`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`.

`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`.

`GoToStop` — Traffic light has been green for North-South, red for East-West for 10 clock ticks.

- Sets output `y1 = YELLOW` (North-South) based on current state.
- Sets output `y2 = RED` (East-West) based on current state.
- After 3 clock ticks, active state becomes `Stop`.

Design Rules in Moore Traffic Light

This example of a Moore traffic light illustrates these Moore design rules:

- The chart computes outputs `y1` and `y2` in state actions.
- The chart tests the input `sens` in conditions on transitions.
- The chart uses temporal logic, but no asynchronous events.

See Also

More About

- “Overview of Mealy and Moore Machines” on page 5-2
- “Design Considerations for Moore Charts” on page 5-9
- “Convert Charts Between Mealy and Moore Semantics” on page 5-14
- “Model An Intersection Of One-Way Streets” on page 11-35
- “Model Distributed Traffic Control System by Using Messages” on page 13-20

Convert Charts Between Mealy and Moore Semantics

Mealy and Moore machines are often considered the basic, industry-standard paradigms for modeling finite-state machines. You can create charts that implement pure Mealy or Moore semantics as a subset of Stateflow chart semantics. Mealy and Moore semantics are not supported in standalone Stateflow charts in MATLAB. For more information, see “Overview of Mealy and Moore Machines” on page 5-2.

A best practice is to not change from one Stateflow chart type to another in the middle of development. You cannot automatically convert the semantics of the original chart to conform to the design rules of the new chart type. Changing chart type usually requires you to redesign your chart to achieve the equivalent behavior of the original chart, so that both charts produce the same sequence of outputs given the same sequence of inputs. Sometimes, however, there is no way to translate specific behaviors without violating chart definitions.

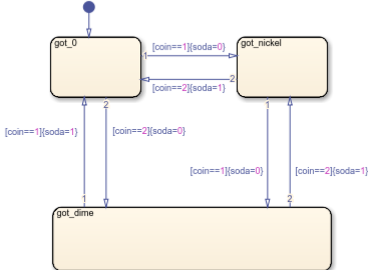
This table lists a summary of what happens when you change chart types mid-design.

From	To	Result
Mealy	Classic	Mealy charts retain their semantics when changed to Classic type.
Classic	Mealy	If the Classic chart defines its output at every time step and conforms to Mealy semantic rules, the Mealy chart exhibits behavior equivalent to the original Classic chart.
Moore	Classic	State actions in the Moore chart behave as `entry` and `during` actions because they are not labeled. The Classic chart exhibits behavior that is not equivalent to the original Moore chart. Requires redesign.
Classic	Moore	Actions that are unlabeled in the Classic chart (`entry` and `during` actions by default) behave as `during` and `exit` actions. The Moore chart exhibits behavior that is not equivalent to the original Classic chart. Requires redesign.
Mealy	Moore	Mealy and Moore rules about placement of actions are mutually exclusive. Converting chart types between Mealy and Moore semantics does not produce equivalent behavior. Requires redesign.
Moore	Mealy	

Transform Chart from Mealy to Moore Semantics

This example shows how to use a Mealy chart to model a vending machine, as described in “Model a Vending Machine by Using Mealy Semantics” on page 5-7.

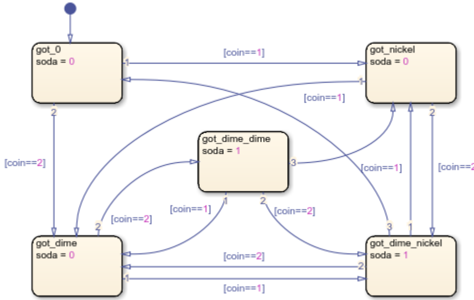
```
openExample("stateflow/MealyVendingMachineExample")
```



In the Mealy chart, each state represents one of the three possible coin inputs: nickel, dime, or no coin. Each condition action computes output (whether soda is released) based on input (the coin received) as the chart transitions to the next state. If you change the chart type to `Moore`, you get a compile-time diagnostic message indicating that the chart violates Moore chart semantics. According to Moore semantics, the chart output `soda` cannot depend on the value of the input `coin`.

To convert the chart to valid Moore semantics, redesign your chart by moving the logic that computes the output out of the transitions and into the states. In the Moore chart, each state must represent both coins received and the soda release condition (`soda = 0` or `soda = 1`). As a result, the redesigned chart requires more states.

```
openExample("stateflow/MealyVendingMachineExample", ...
    supportingFile="sf_moore_vending_machine")
```



Before considering the value of `coin`, the Moore chart must compute the value of `soda` according to the active state. As a result, there is a delay in releasing soda, even if the machine receives enough money to cover the cost.

Compare Semantics of Vending Machines

This table compares the semantics of the charts before and after the transformation.

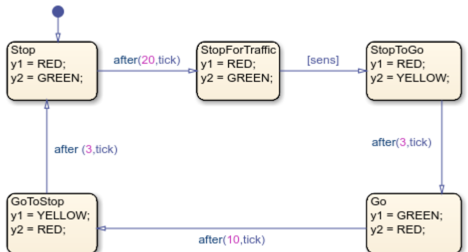
Mealy Vending Machine	Moore Vending Machine
Uses three states	Uses five states
Computes outputs in condition actions	Computes outputs in state actions
Updates output based on input	Updates output before evaluating input, requiring an extra time step to release the soda

For this vending machine, Mealy is a better modeling paradigm because there is no delay in releasing the soda once sufficient coins are received. By contrast, the Moore vending machine requires an extra time step before producing the soda. Therefore, it is possible for the Moore vending machine to produce a can of soda at the same time step that it accepts a coin toward the next purchase. In this situation, the delivery of a soda can appear to be in response to this coin, but actually occurs because the vending machine received the purchase price in previous time steps.

Transform Chart from Moore to Mealy Semantics

This example shows how to use a Moore chart to model a traffic light, as described in “Model a Traffic Light by Using Moore Semantics” on page 5-12.

```
openExample("stateflow/MooreTrafficLightExample")
```

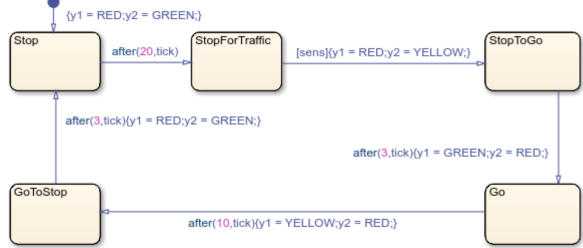


In the Moore chart, each state represents the colors of the traffic light in two opposite directions and the duration of the current color. Each state action computes output (the colors of the traffic light) regardless of input (if there are cars waiting at a sensor). If you change the chart type to `Mealy`, you get a compile-time diagnostic message indicating that the chart violates Mealy chart semantics. According to Mealy semantics, the chart cannot compute its outputs in state actions.

To convert the chart to valid Mealy semantics, redesign your chart by moving the logic that computes the output out of the states and into the transitions. The redesigned Mealy chart consists of five states, just like the Moore chart. In most transitions, a temporal logic expression or the Boolean input `sens` guards a condition action computing the outputs `y1` and `y2`. The only exceptions are:

- The default transition, which computes the initial outputs without a guarding condition.
- The transition from the `Stop` state to the `StopForTraffic` state, which does not compute new outputs.

```
openExample("stateflow/MooreTrafficLightExample", ...
    supportingFile="sf_mealy_traffic_light")
```



In the same time step, the Mealy chart evaluates the temporal logic expressions and the input signal `sens`, and computes the value of the outputs `y1` and `y2`. As a result, in the Mealy chart, the output changes one time step before the corresponding change occurs in the original Moore chart. In the Simulink model, you can compensate for the anticipated changes in output by adding a Delay block to each output signal.

Compare Semantics of Traffic Light Controllers

This table compares the semantics of the charts before and after the transformation.

Moore Traffic Light Controller	Mealy Traffic Light Controller
Uses five states	Uses five states
Computes outputs in state actions	Computes outputs in condition actions

Moore Traffic Light Controller	Mealy Traffic Light Controller
Updates output before evaluating input	Updates output based on input, requiring a Delay block in each output signal

See Also

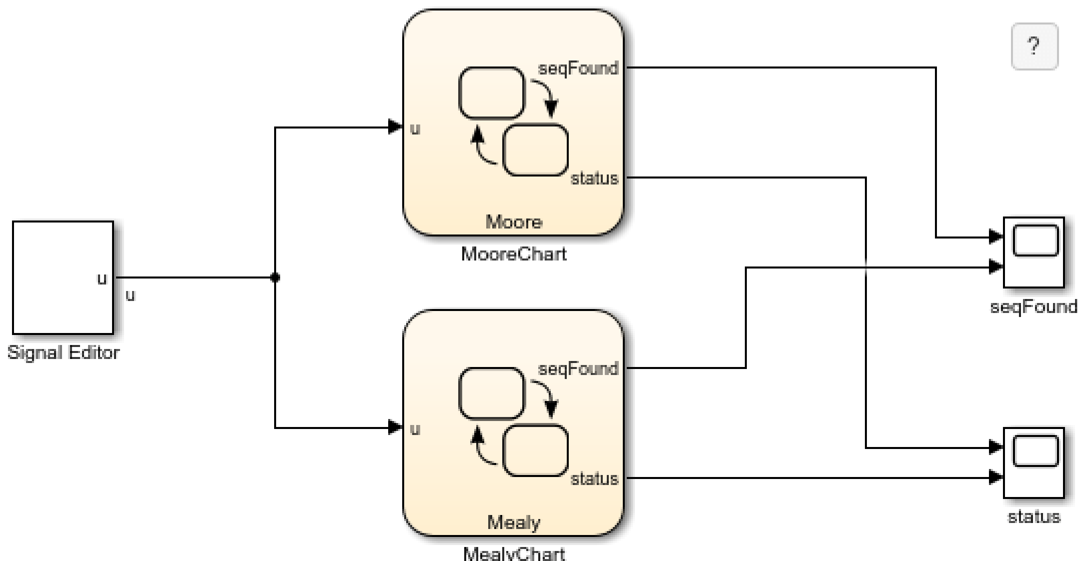
More About

- "Overview of Mealy and Moore Machines" on page 5-2
- "Design Considerations for Mealy Charts" on page 5-5
- "Design Considerations for Moore Charts" on page 5-9
- "Model a Vending Machine by Using Mealy Semantics" on page 5-7
- "Model a Traffic Light by Using Moore Semantics" on page 5-12

Sequence Recognition by Using Mealy and Moore Charts

This example shows how to use Mealy and Moore machines for a sequence recognition application in signal processing. For more information, see “Overview of Mealy and Moore Machines” on page 5-2.

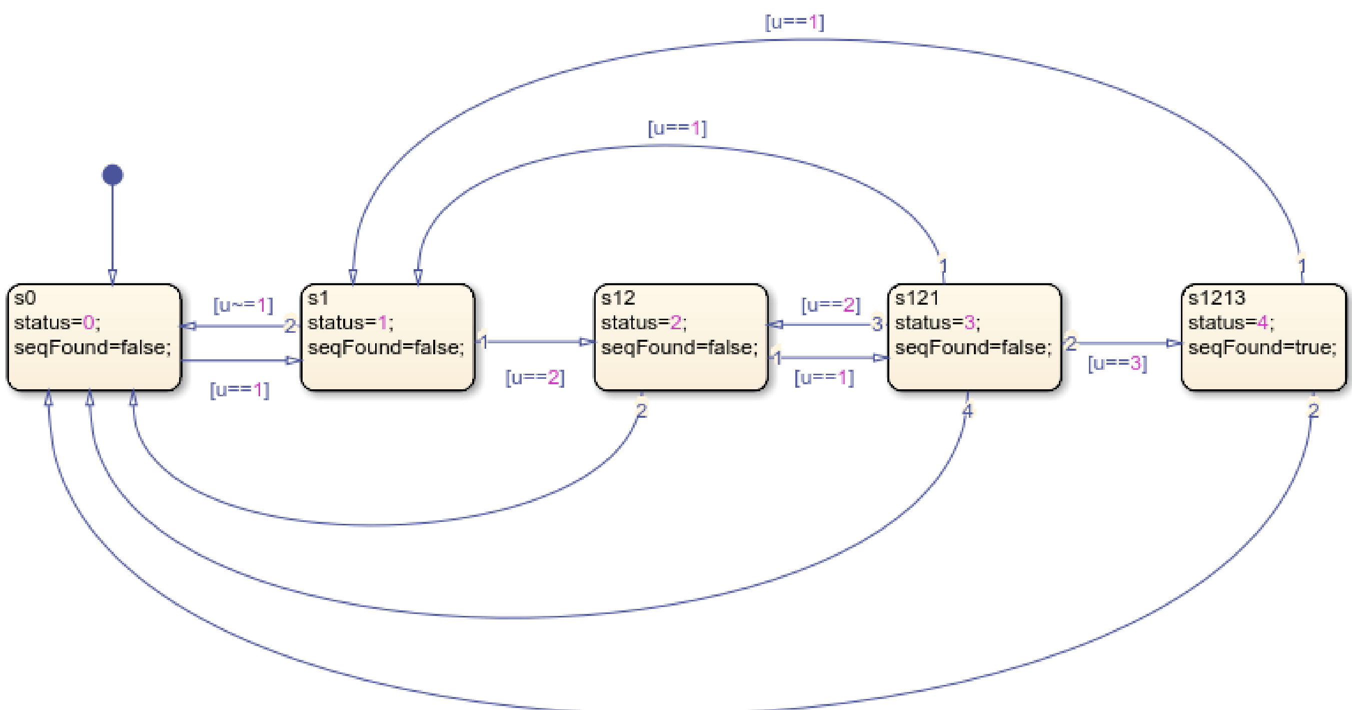
In this model, two Stateflow® charts use a different set of semantics to find the sequence `1`, `2`, `1`, `3` in the input signal from a Signal Editor (Simulink) block.



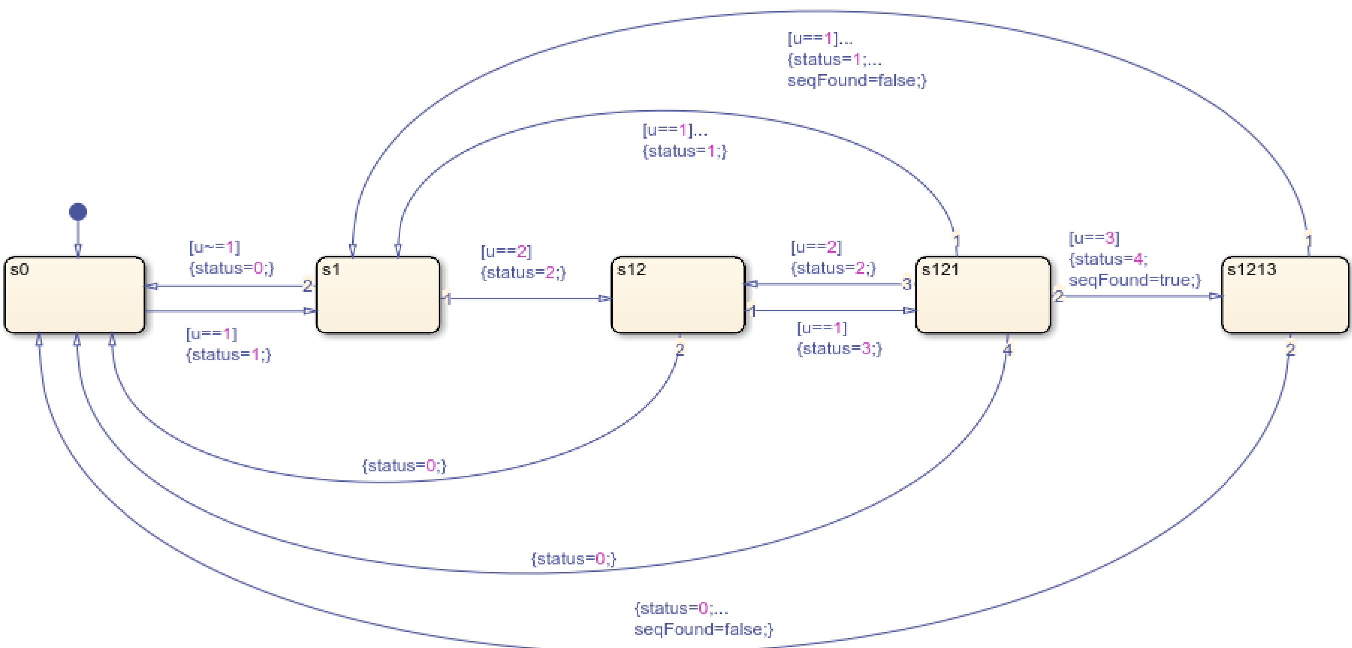
Each chart contains an input data `u` and two output data:

- `seqFound` indicates when the chart finds the sequence. A value of `false` means that the chart is still searching for the sequence. A value of `true` means that the chart has found the sequence.
- `status` records the status of the sequence recognition. This value ranges from 0 to 4 and indicates the number of symbols detected by the chart.

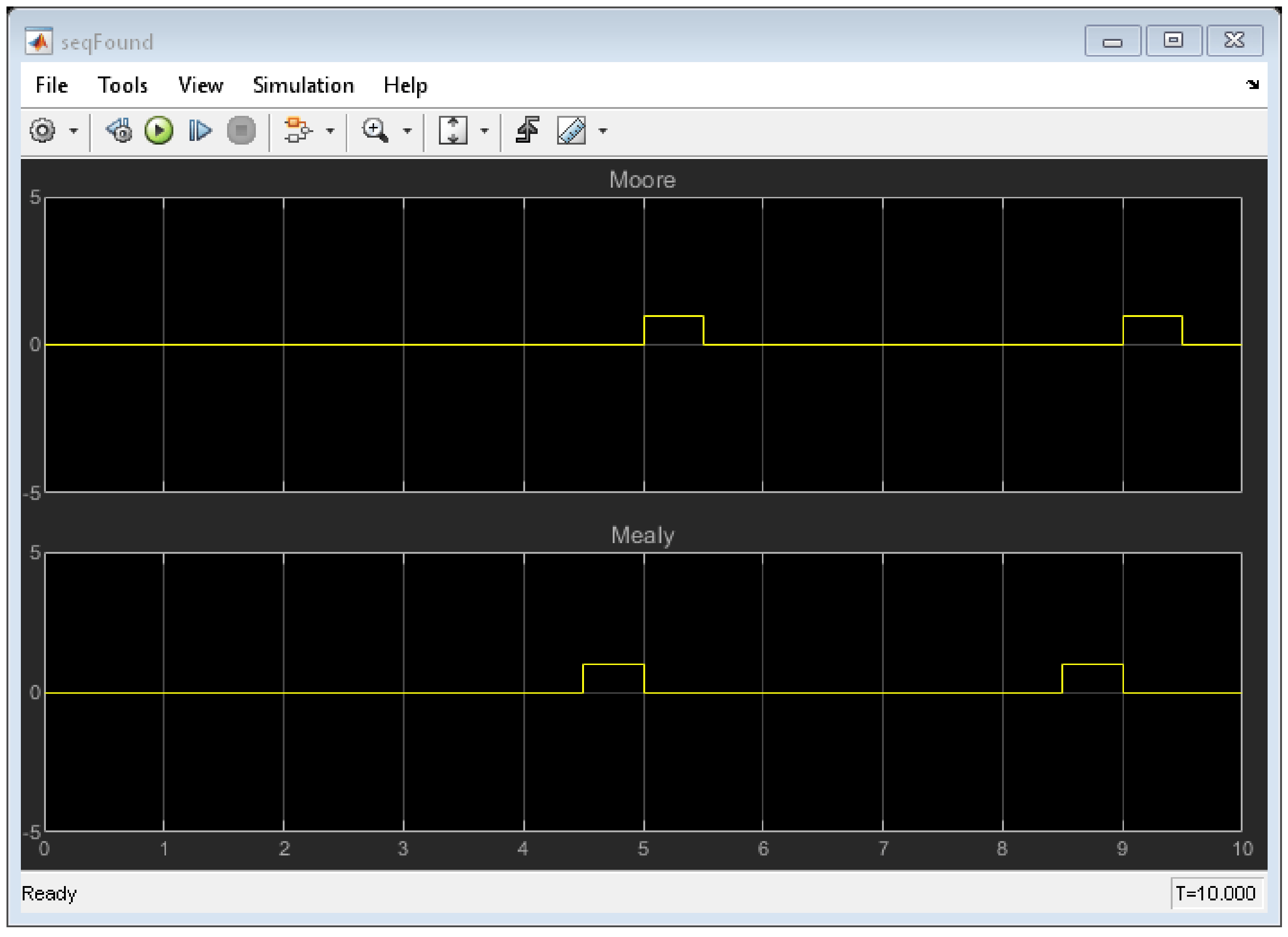
The Moore chart outputs `seqFound` and `status` based on the current state of the chart. At each time step, the chart executes the actions for the current state, evaluates the input `u`, and transitions to a new state. For example, when the chart receives the sequence of input values `1`, `2`, `1`, `3` from the Signal Editor block, it transitions from state `s0` to state `s1` to state `s12` to state `s121` to state `s1213` in four time steps. The chart sets the value of `seqFound` to `true` in a state action after state `s1213` becomes active.



The Mealy chart outputs `seqFound` and `status` based on the current state of the chart and the value of the input. At each time step, the chart evaluates the input `u`, makes the transition to a new state, and executes the corresponding condition actions. Because this chart computes its output values in the condition actions of its transitions, these actions are taken before the state becomes active. For example, when the chart receives the sequence of input values `1`, `2`, `1`, `3` from the Signal Editor block, it transitions from state `s0` to state `s1` to state `s12` to state `s121` to state `s1213` in four time steps. The chart sets the value of `seqFound` to `true` in a condition action in the same time step that state `s1213` becomes active.



When you simulate the model, the `seqFound` scope shows that the output of the Moore chart lags one time step behind the output of the Mealy chart. The delay is a result of the Moore semantics, in which the output is based on the state of the chart at the start of each time step and not on the current input.



Reference

Katz, Bruce F. *Digital Design: From Gates to Intelligent Machines*, 2006.

See Also

Signal Editor

More About

- “Overview of Mealy and Moore Machines” on page 5-2
- “Design Considerations for Mealy Charts” on page 5-5
- “Design Considerations for Moore Charts” on page 5-9

Karplus-Strong Algorithm by Using Moore Charts

This example shows a simple implementation of the Karplus-Strong algorithm for string synthesis by using Stateflow® charts with Moore semantics. For more information, see “Design Considerations for Moore Charts” on page 5-9.

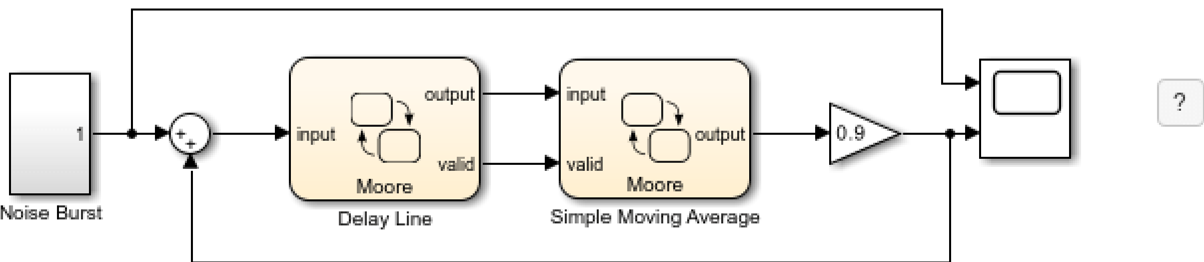
Principle

The initial burst of white noise is produced by a Uniform Random Number block. It is fed back into a delay line of the same length. The moving average smooth the signal at each cycle, while the gain less than 1 maintains the stability of the feedback loop. They both model the string losses over each cycle.

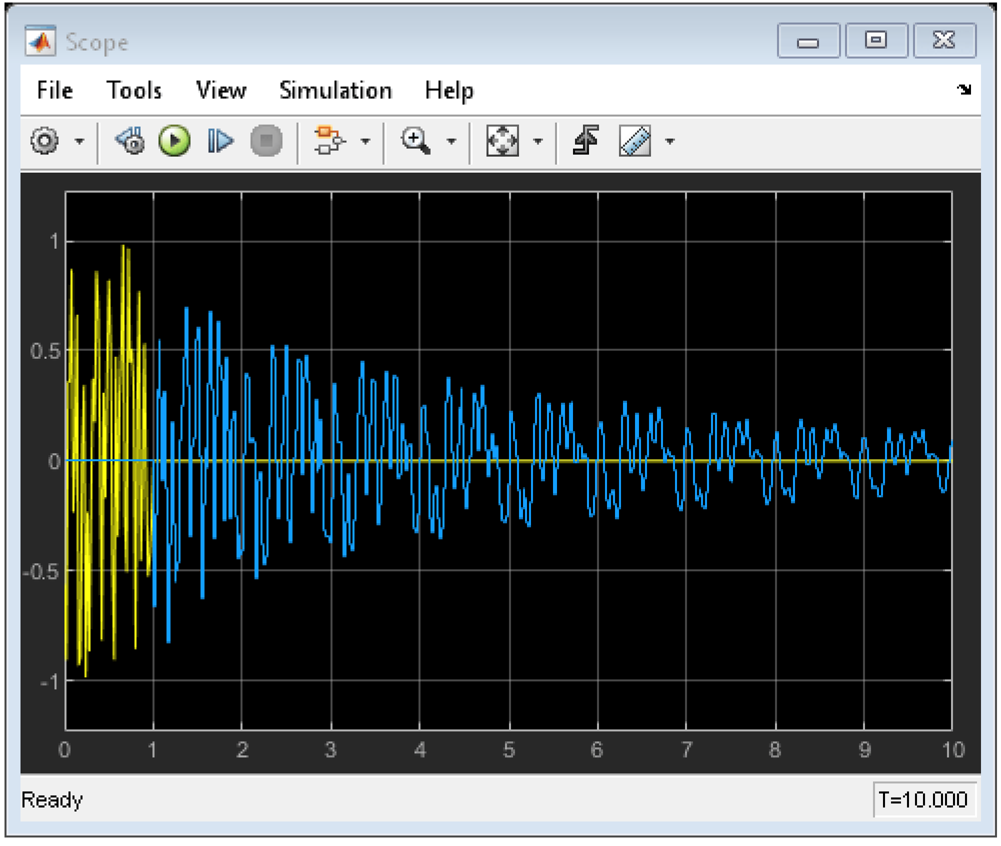
Nondirect Feedthrough

This example illustrates the benefits of Moore Charts in loops. Moore semantics guarantees that outputs only depend on the current state, but neither on inputs nor the next state. Hence a Moore Chart has nondirect feedthrough and can safely be used in feedback loops. For more information, see “Algebraic Loop Concepts” (Simulink).

On the contrary, Classic or Mealy Charts provide direct feedthrough. These charts do not prevent algebraic loops, unless an external Delay block breaks cyclic dependencies. Simulation would issue an error if both charts were Classic or Mealy charts.



Copyright 2015-2018 The MathWorks, Inc.



See Also

More About

- “Overview of Mealy and Moore Machines” on page 5-2
- “Design Considerations for Moore Charts” on page 5-9
- “Algebraic Loop Concepts” (Simulink)

Initialize Persistent Variables in MATLAB Functions

A persistent variable is a local variable in a MATLAB function that retains its value in memory between calls to the function. If you generate code from your model, you must initialize a persistent variable for your MATLAB functions. For more information, see `persistent`.

When using MATLAB functions that contain persistent variables in Simulink models, you should follow these guidelines:

- Initialize the persistent variables in functions only by accessing constants.
- Ensure the control flow of the function does not depend on whether the initialization occurs.

If you do not follow these guidelines, several conditions produce an initialization error:

- MATLAB Function blocks with persistent variables where the **Allow direct feedthrough** property is cleared
- MATLAB Function blocks with persistent variables in models with State Control blocks where **State control** is set to `Synchronous`
- Stateflow charts that implement Moore machine semantics and that use MATLAB functions with persistent variables

For example, the function `fcn` below uses a persistent variable, `n`. `fcn` violates both guidelines. The initial value of `n` depends on the input `u` and the `return` statement interrupts the normal control flow of the function. Consequently, this code produces an error when used in a model that has one of the conditions described above.

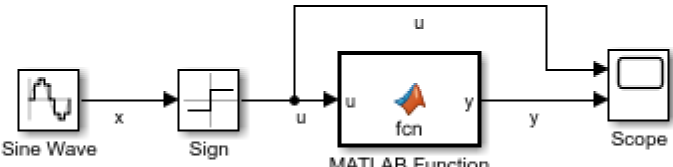
```
function y = fcn(u)
    persistent n

    if isempty(n)
        n = u;
        y = 1;
        return
    end

    y = n;
    n = n + u;
end
```

To prevent the error, initialize the persistent variable by setting it to a constant value and removing the `return` statement. This modified version of `fcn` initializes the persistent variable without producing an error:

```
function y = fcn(u)
    persistent n

    if isempty(n)
        n = 1;
    end

    y = n;
    n = n + u;
end
```

MATLAB Function Block with No Direct Feedthrough

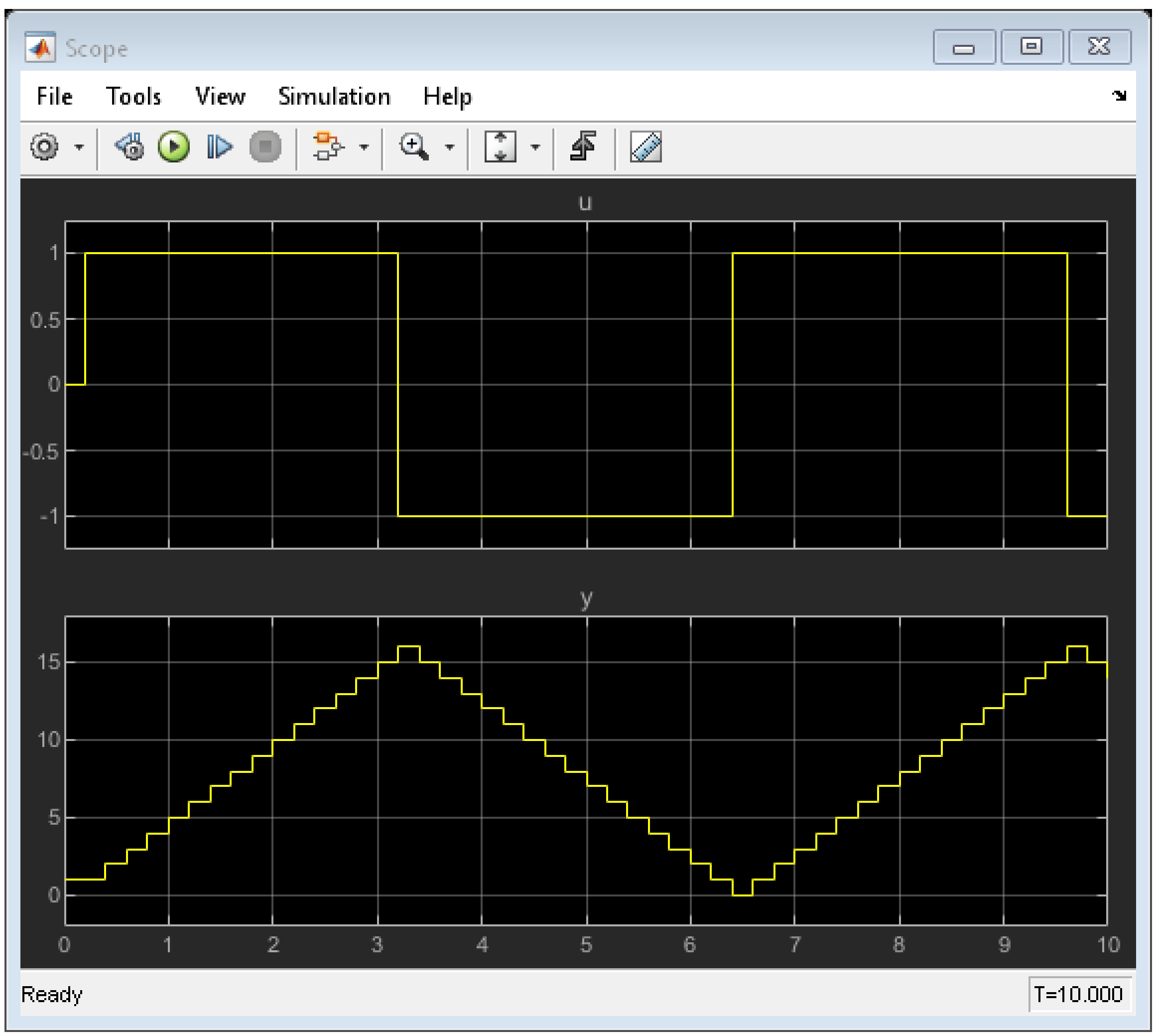
This model contains a MATLAB Function block that uses the first version of `fcn`, described previously. The MATLAB Function block input is a square wave, which is provided by a Sign and Sine Wave block. The MATLAB Function block adds the value of `u` to the persistent variable `n` at each time step.



Simulate the model. The simulation returns an error because:

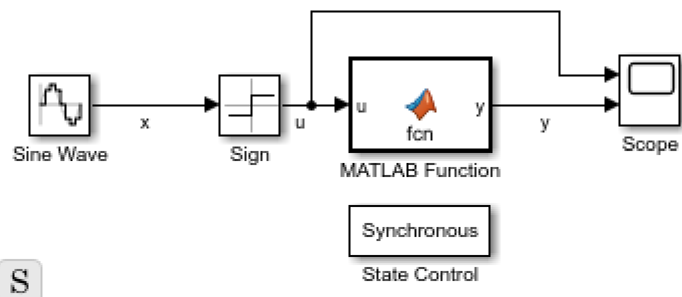
- The initial value of the persistent variable `n` depends on the input `u`.
- The `return` statement interrupts the normal control flow of the function.
- The **Allow direct feedthrough** property of the MATLAB Function block is cleared.

Modify the MATLAB Function block code, as shown in the corrected version of `fcn`. Simulate the model again.



State Control Block in Synchronous Mode

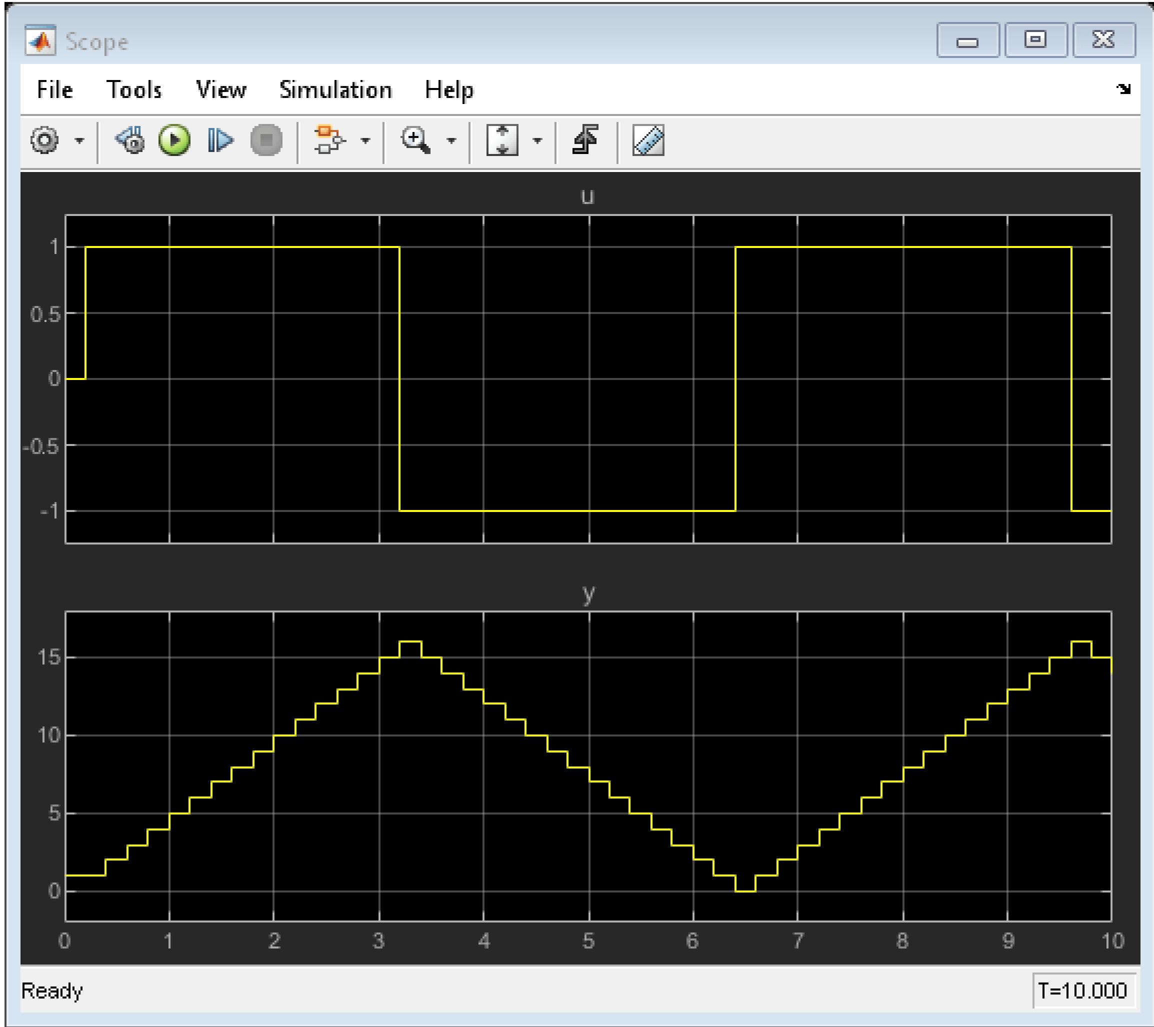
This model contains a MATLAB Function block that uses the first version of `fcn`, described previously. The MATLAB Function block input is a square wave, which is provided by a Sign and Sine Wave block. The MATLAB Function block adds the value of `u` to the persistent variable `n` at each time step. The model contains a State Control block where **State control** is set to `Synchronous`.



Simulate the model. The simulation returns an error because:

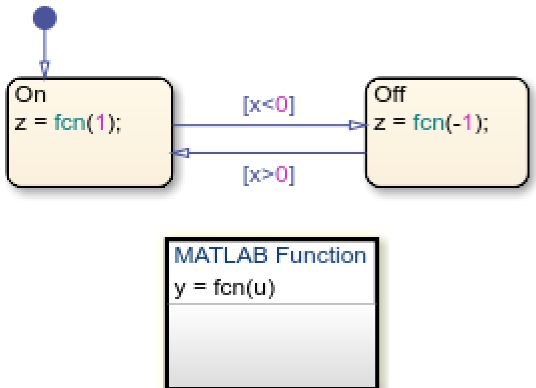
- The initial value of the persistent variable `n` depends on the input `u`.
- The `return` statement interrupts the normal control flow of the function.
- The model contains a State Control block where **State control** is set to `Synchronous`.

Modify the MATLAB Function block code, as shown in the corrected version of `fcn`. Simulate the model again.



Stateflow Chart Implementing Moore Semantics

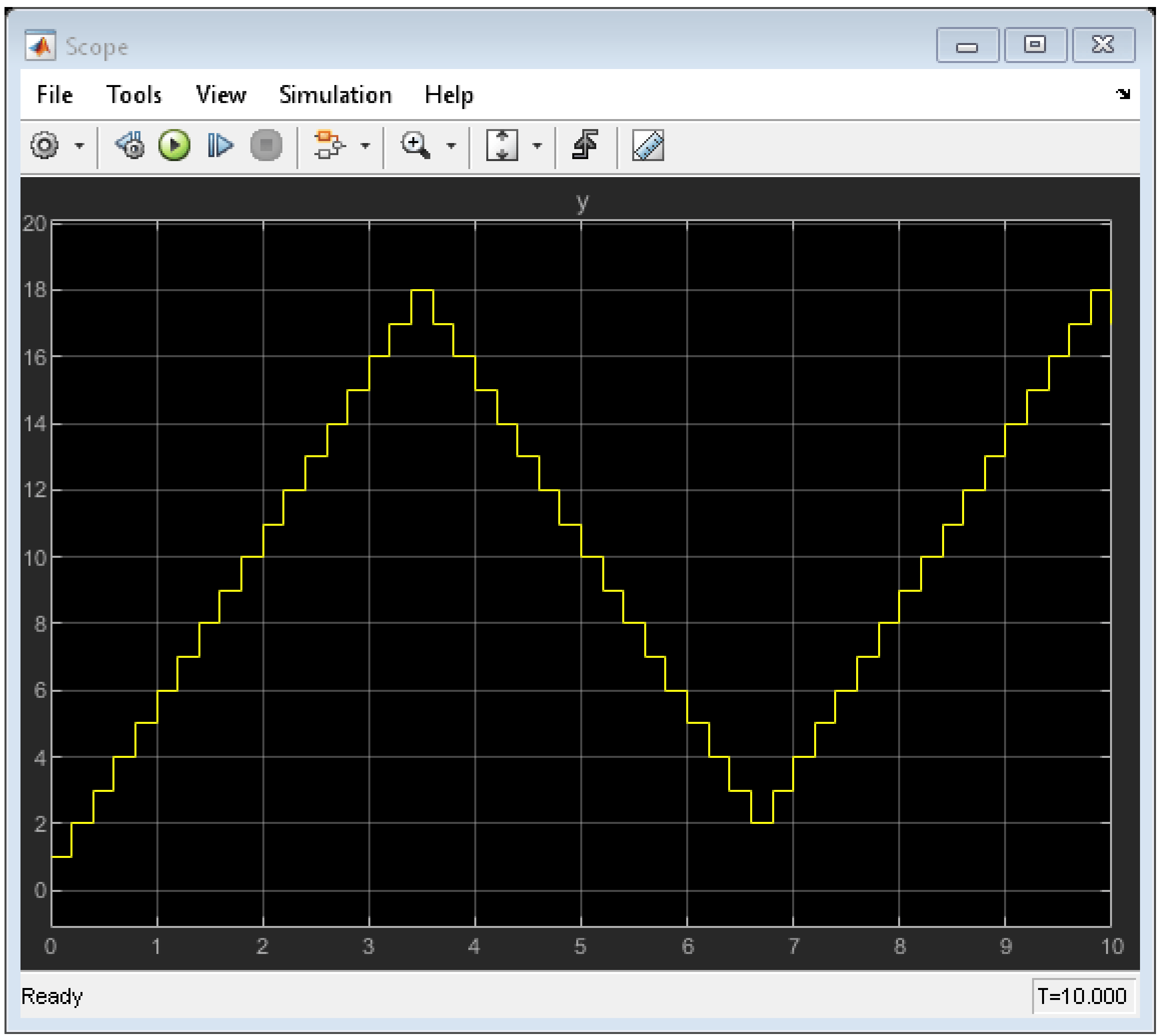
This model contains a Stateflow Chart with a MATLAB function that uses the first version of `fcn`, described previously. The MATLAB function adds the value (`1` or `-1`) determined by the active state to the persistent variable `n` at each time step.



Simulate the model. The simulation returns an error because:

- The initial value of the persistent variable `n` depends on the input `u`.
- The `return` statement interrupts the normal control flow of the function.
- The chart implements Moore semantics.

Modify the MATLAB function code, as shown in the corrected version of `fcn`. Simulate the model again.



See Also

Blocks
MATLAB Function | State Control | Chart

Functions
`persistent`

More About

- “Use Nondirect Feedthrough in a MATLAB Function Block” (Simulink)
- “Synchronous Subsystem Behavior with the State Control Block” (HDL Coder)
- “Design Considerations for Moore Charts” on page 5-9

Techniques for Streamlining Chart Design

- “Group Chart Objects by Using Boxes” on page 6-2
- “Encapsulate Modal Logic by Using Subcharts” on page 6-6
- “Reuse Logic Patterns by Defining Graphical Functions” on page 6-9
- “Export Stateflow Functions for Reuse” on page 6-14
- “Reuse Functions by Using Atomic Boxes” on page 6-18
- “Add Descriptive Comments in a Chart” on page 6-22

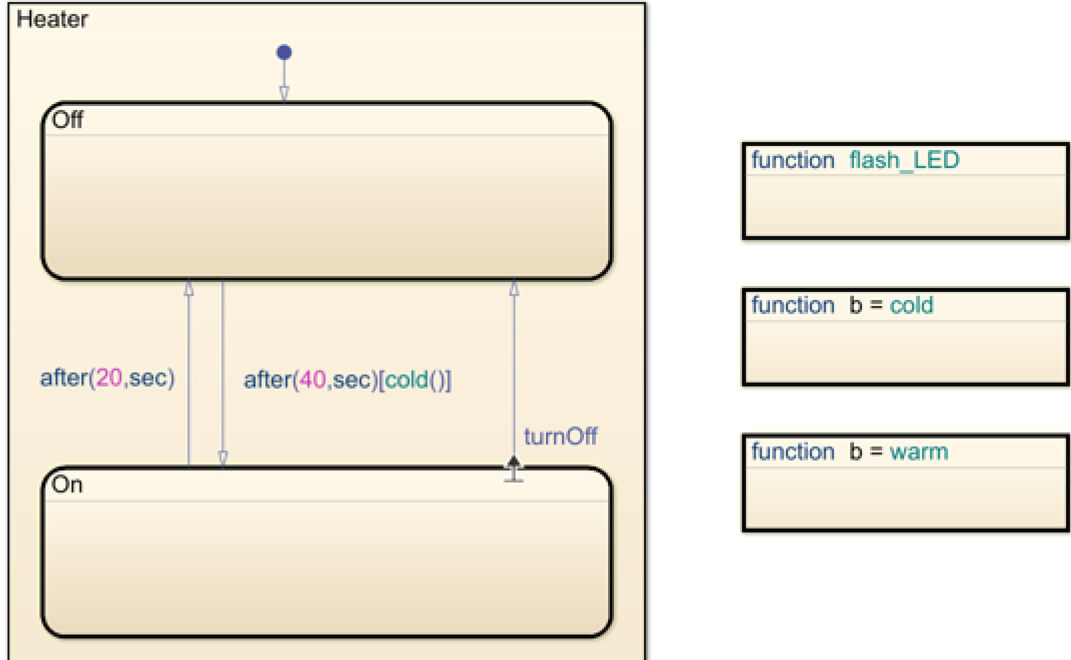
Group Chart Objects by Using Boxes

A box is a graphical object that defines a namespace that you can use to organize objects in your chart, such as states, functions, and data. Boxes allow you to quickly glance at your chart and recognize which states or functions work together to perform certain tasks.

Boxes have square corners to distinguish them from states, which have rounded corners. Boxes are not supported in standalone Stateflow charts in MATLAB.

Note To add notes to your Stateflow chart, use annotations instead of boxes. For more information, see “Add Descriptive Comments in a Chart” on page 6-22.

In this chart, the box `Heater` groups together the related states `Off` and `On`.



For more information about this example, see “Model Bang-Bang Temperature Control System” on page 14-51.

Semantics of Stateflow Boxes

Hierarchy of Graphical Objects in Boxes

Boxes add a level of hierarchy to Stateflow charts. If you refer to a box-parented function or state from outside of the box, you must include the box name in the path. See “Group Functions Using a Box” on page 6-3.

Activation Order of Parallel States

Boxes affect the implicit activation order of parallel states in a chart. If your chart uses implicit ordering, parallel states within a box wake up before other parallel states that are lower or to the right in that chart. Within a box, parallel states wake up in top-down, left-right order. See "Group States Using a Box" on page 6-4.

To specify activation order explicitly on a state-by-state basis, select **User-specified state/transition execution order** in the Chart properties dialog box. This option is selected by default when you create a new chart. For details, see "Explicit Ordering of Parallel States" on page 2-46.

Guidelines for Using Boxes

When you use a box:

- 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 add data to a box so that all the elements in the box can share the same data.
- You can group a box and its contents into a single graphical element. See "Group States" on page 1-30.
- You can subchart a box to hide its elements. See "Encapsulate Modal Logic by Using Subcharts" on page 6-6.
- 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.

Draw and Edit a Box

Create a Box

You create boxes in your chart by using the Box icon in the object palette.

1. In the object palette, click the Box tool.
2. On the chart canvas, click the location for the new box. The new box appears with the cursor in place to add a name.
3. Enter a name for the box and then click outside of the box.

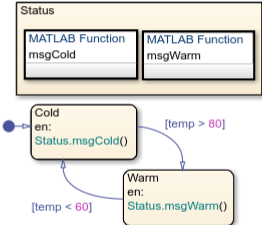
Delete a Box

To delete a box, click the box and press the **Delete** key.

Examples of Using Boxes

Group Functions Using a Box

This chart shows a box named `Status` that groups two MATLAB functions.



The chart executes in this order:

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.

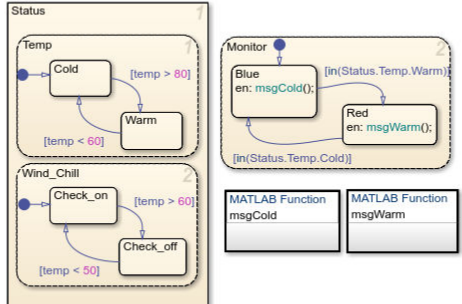
   Note Because the MATLAB function resides inside a box, the path of the function call must include the box name `Status`. If you omit this prefix, an error message appears.

   ---
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.
5. If the value of the input data `temp` drops below 60, a transition to the state `Cold` occurs.
6. Steps 2 through 5 repeat until the simulation ends.

Group States Using a Box

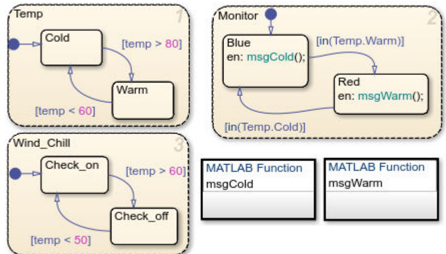
This chart shows a box named `Status` that groups together related states. The chart uses implicit ordering for parallel states. For more information, see "Implicit Ordering of Parallel States" on page 2-47.



In this chart:

- The state `Temp` wakes up first, followed by the state `Wind_Chill`. Then, the state `Monitor` wakes up.

  This implicit activation order occurs because `Temp` and `Wind_Chill` reside in a box. If you remove the box, the implicit activation order changes to `Temp`, `Monitor`, then `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.

  Because the substates `Status.Temp.Cold` and `Status.Temp.Warm` reside inside a box, the argument of the `in` operator must include the box name `Status`. If you omit this prefix, an error message appears. For information about the `in` operator, see “Check State Activity by Using the in Operator” on page 11-24.

See Also

`Stateflow.Annotation`

More About

- “Add Descriptive Comments in a Chart” on page 6-22
- “Encapsulate Modal Logic by Using Subcharts” on page 6-6

Encapsulate Modal Logic by Using Subcharts

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

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

The subchart appears as a block with its name in the block center. However, you can define actions and default transitions for subcharts just as you can for superstates. You can also create transitions to and from subcharts just as you can create transitions to and from superstates. You can create transitions between states residing outside a subchart and any state within a subchart. The term *supertransition* refers to a transition that crosses subchart boundaries in this way. See "Move Between Levels of Hierarchy by Using Supertransitions" on page 1-48 for more information.

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

Some subcharts can become *atomic* units if they meet certain modeling requirements. For more information, see "Restrictions for Converting to Atomic Subcharts" on page 17-9.

Create a Subchart

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

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

1. Right-click the object and select **Group & Subchart** > **Subchart**.
2. Confirm that the object now appears as a subchart.

To convert the subchart back to its original form, right-click the subchart. In the context menu, select **Group & Subchart** > **Subchart**.

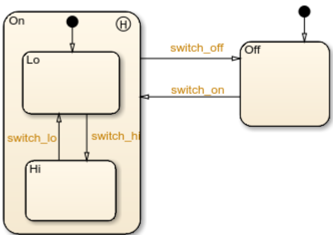
Rules of Subchart Conversion

When you convert a box to a subchart, the subchart retains the attributes of a box. For example, the position of the resulting subchart determines its activation order in the chart if implicit ordering is enabled (see "Group Chart Objects by Using Boxes" on page 6-2 for more information).

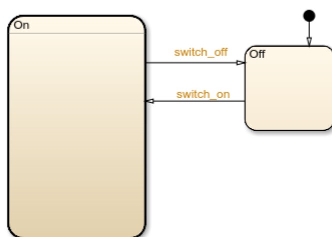
You cannot undo the operation of converting a subchart back to its original form. When you perform this operation, the undo and redo buttons are disabled from undoing and redoing any prior operations.

Convert a State to a Subchart

Suppose that you have the following chart:



1. To convert the On state to a subchart, right-click the state and select **Group & Subchart** > **Subchart**.
2. Confirm that the On state now appears as a subchart.



Manipulate Subcharts as Objects

Subcharts also act as individual objects. You can move, copy, cut, paste, relabel, and resize subcharts as you would states and boxes. You can also draw transitions to and from a subchart and any other state or subchart at the same or different levels in the chart hierarchy (see “Move Between Levels of Hierarchy by Using Supertransitions” on page 1-48).

Open a Subchart

Opening a subchart allows you to view and change its contents. To open a subchart, do one of the following:

- Double-click anywhere in the box that represents the subchart.
- Select the box representing the subchart and press the **Enter** key.

Edit a Subchart

After you open a subchart (see “Open a Subchart” on page 6-7), you can perform any editing operation on its contents that you can perform on a top-level chart. This means that you can create, copy, paste, cut, relabel, and resize the states, transitions, and subcharts in a subchart. You can also group states, boxes, and graphical functions inside subcharts.

You can also cut and paste objects between different levels in your chart. For example, to copy objects from a top-level chart to one of its subcharts, first open the top-level chart and copy the objects. Then open the subchart and paste the objects into the subchart.

Transitions from outside subcharts to states or junctions inside subcharts are called *supertransitions*. You create supertransitions differently than you do ordinary transitions. See “Move Between Levels of Hierarchy by Using Supertransitions” on page 1-48 for information on creating supertransitions.

Navigate Subcharts

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

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

Note You can also use the **Escape** key to navigate up to the parent object for a subcharted state, box, or function.

Reuse Logic Patterns by Defining Graphical Functions

A *graphical function* in a Stateflow chart is a graphical element that helps you reuse control-flow logic and iterative loops. You create graphical functions with flow charts that use connective junctions and transitions. You can call a graphical function in the actions of states and transitions. With graphical functions, you can:

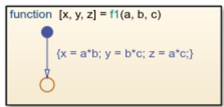
- Create modular, reusable logic that you can call anywhere in your chart.
- Track simulation behavior visually during chart animation.

A graphical function can reside anywhere in a chart, state, or subchart. The location of the function determines the set of states and transitions that can call the function.

- If you want to call the function 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 a chart, put your graphical function at the chart level.
- If you want to call the function from any chart in your model, put your graphical function at the chart level and enable exporting of chart-level functions. For more information, see “Export Stateflow Functions for Reuse” on page 6-14.

Note A graphical function can access chart and state data above it in the Stateflow hierarchy.

For example, this graphical function has the name `f1`. It takes three arguments (`a`, `b`, and `c`) and returns three output values (`x`, `y`, and `z`). The function contains a flow chart that computes three different products of the arguments.



Define a Graphical Function

1. In the object palette, click the graphical function icon.
2. On the chart canvas, click the location for the new graphical function.
3. Enter the signature label for the function.

   The signature label of the function specifies a name for your function and the formal names for its arguments and return values. A signature label has this syntax:

   ```
   [return_val1,return_val2,...] = function_name(arg1,arg2,...)
   ```

   You can specify multiple return values and multiple input arguments. Each return value and input argument can be a scalar, vector, or matrix of values. For functions with only one return value, omit the brackets in the signature label.

You can use the same variable name for both arguments and return values. When you use the same variable for an input and output, you create in-place data. For example, a function with this signature label uses the variables `y1` and `y2` as both inputs and outputs:

```
[y1,y2,y3] = f(y1,u,y2)
```

If you export this function to C code, the generated code treats `y1` and `y2` as in-place arguments passed by reference. Using in-place data reduces the number of times that the generated code copies intermediate data, which results in more efficient code.

In the **Symbols** pane and the Model Explorer, the arguments and return values of the function signature appear as data items that belong to your function. Arguments have the scope `Input`. Return values have the scope `Output`.

4. Specify the data properties for each argument and return value, as described in “Set Data Properties” on page 10-5. When an argument and a return value have the same name, you can edit properties only for the argument. The properties for the return value are read-only.
5. To program the function, construct a flow chart inside the function box, as described in “Create Flow Charts in Stateflow” on page 3-2.

   Because a graphical function must execute completely when you call it, you cannot use states. Connective junctions and transitions are the only graphical elements that you can use in a graphical function.

   Note In a graphical function, do not broadcast events that can cause the active state to change. In a graphical function, the behavior of an event broadcast that causes an exit from the active state is unpredictable.

   ---
6. Create any additional data items required by your function. For more information, see “Add Data Through the Model Explorer” on page 10-3.

   Your function can access its own data or data belonging to parent states or the chart. The data items in the function can have one of these scopes:

   - `Constant` — Constant data retains its initial value through all function calls.
   - `Parameter` — Parameter data retains its initial value through all function calls.
   - `Local` — Local data persists across function calls throughout the simulation. Valid only in charts that use C as the action language.
   - `Temporary` — Temporary data initializes at the start of every function call. Valid only in charts that use C as the action language.

   In charts that use C as the action language, define local data when you want your data values to persist across function calls throughout the simulation. Define temporary data when you want to initialize data values at the start of every function call. For example, you can define a counter with `Local` scope if you want to track the number of times that you call the function. In contrast, you can designate a loop counter to have `Temporary` scope if you do not need the counter value to persist after the function completes.

   In charts that use MATLAB as the action language, you do not need to define local or temporary data in graphical functions. Instead, you can use undefined variables to store values that are accessible only during the rest of the function call. To store values that persist across function calls, use local data at the chart level.

Tip You can initialize local and temporary data in your function from the MATLAB workspace. For more information, see “Initialize Data from the MATLAB Base Workspace” on page 10-30.

Call Graphical Functions in States and Transitions

You can call graphical functions from the actions of any state or transition or from other functions. If you export a graphical function, you can call it from any chart in the model. For more information about exporting functions, see “Export Stateflow Functions for Reuse” on page 6-14.

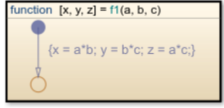
To call a graphical function, use the function signature and include an actual argument value for each formal argument in the function signature.

```
[return_val1,return_val2,...] = function_name(arg1,arg2,...)
```

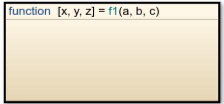
If the data types of the actual and formal arguments differ, the function casts the actual argument to the type of the formal argument.

Manage Large Graphical Functions

You can choose to make your graphical function as large as you want. If your function grows too large, you can hide its contents by right-clicking inside the function box and selecting **Group & Subchart** > **Subchart** from the context menu.



To make the graphical function box opaque, right-click the function and clear the **Content Preview** property from the context menu.



To dedicate the entire chart window to programming your function, access the flow chart in your subcharted graphical function by double-clicking the function box. For more information, see “Encapsulate Modal Logic by Using Subcharts” on page 6-6.

Specify Properties of Graphical Functions

The properties listed below specify how a graphical function interacts with the other components in your Stateflow chart. You can modify these properties in the **Property Inspector**, the Model Explorer, or the Function properties dialog box.

To use the **Property Inspector**:

1. In the **Modeling** tab, under **Design Data**, select **Property Inspector**.
2. In the Stateflow Editor, select the graphical function.

3. In the **Property Inspector**, edit the transition properties.

To use the Model Explorer:

1. In the **Modeling** tab, under **Design Data**, select **Model Explorer**.
2. In the **Model Hierarchy** pane, select the graphical function.
3. In the **Dialog** pane, edit the graphical function properties.

To use the Function properties dialog box:

1. In the Stateflow Editor, right-click the graphical function.
2. Select **Properties**.
3. In the properties dialog box, edit the graphical function properties.

You can also modify these properties programmatically by using `Stateflow.Function` objects. For more information about the Stateflow programmatic interface, see “Overview of the Stateflow API”.

Name

Function name. Click the function name link to bring your function to the foreground in its native chart.

Function Inline Option

Controls the inlining of your function in generated code:

- `Auto` — Determines whether to inline your function based on an internal calculation.
- `Inline` — Inlines your function if you do not export it to other charts and it is not part of a recursion. (A recursion exists if your function calls itself directly or indirectly through another function call.)
- `Function` — Does not inline your function.

This property is not available in the **Property Inspector**.

Label

Signature label for your function. The function signature label specifies a name for your function and the formal names for its arguments and return values. This property is not available in the **Property Inspector**.

Description

Description of the graphical function.

Document Link

Link to online documentation for the graphical function. You can enter a web URL address or a MATLAB command that displays documentation as an HTML file or as text in the MATLAB Command Window. When you click the **Document link** hyperlink, Stateflow evaluates the link and displays the documentation.

See Also

Objects
`Stateflow.Function`

Tools
Model Explorer

More About

- "Create Flow Charts in Stateflow" on page 3-2
- "Create Flow Charts by Using Pattern Wizard" on page 3-5
- "Export Stateflow Functions for Reuse" on page 6-14
- "Reuse Functions by Using Atomic Boxes" on page 6-18

Export Stateflow Functions for Reuse

You can extend the scope of the chart-level functions in your Stateflow chart to other blocks in the Simulink model by exporting the functions. You can export:

- Graphical functions
- MATLAB functions
- Simulink functions
- Truth tables

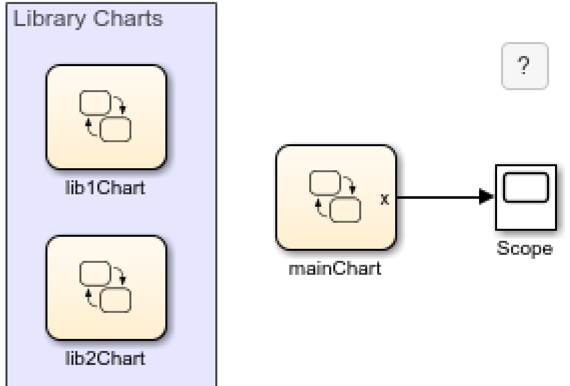
When you export chart-level functions, you can call them in other Stateflow charts and Simulink Caller blocks.

- To export chart-level functions that you can call by using qualified notation such as *`chartName.functionName`*, select the **Export chart level functions** chart property, as described in “Specify Properties for Stateflow Charts” on page 1-19.
- To export chart-level functions that you can call without using qualified notation, select **Export chart level functions**, and then select **Treat exported functions as globally visible**. You cannot export functions with the same name.

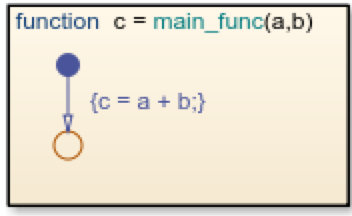
For more information, see “Call a Simulink Function from a Model” (Simulink).

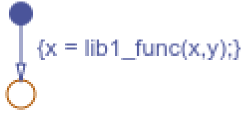
Share Functions Across Stateflow Charts

This example shows how to call exported functions from other charts in your Simulink model. This model contains a main Stateflow chart, `mainChart`, and two auxiliary library charts, `lib1Chart` and `lib2Chart`. Each chart contains a chart-level graphical function and has both the **Export chart level functions** and **Treat exported functions as globally visible** chart properties enabled.

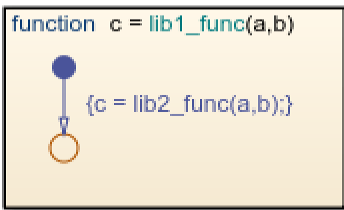


The main chart contains two data objects, `x` and `y`, with initial values of `0` and `1`, respectively. When you simulate the model, the default transition in this chart calls the function `lib1_func` using these values as arguments.

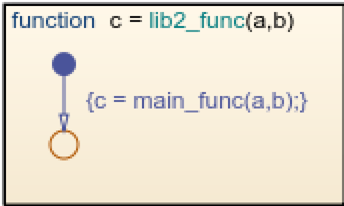




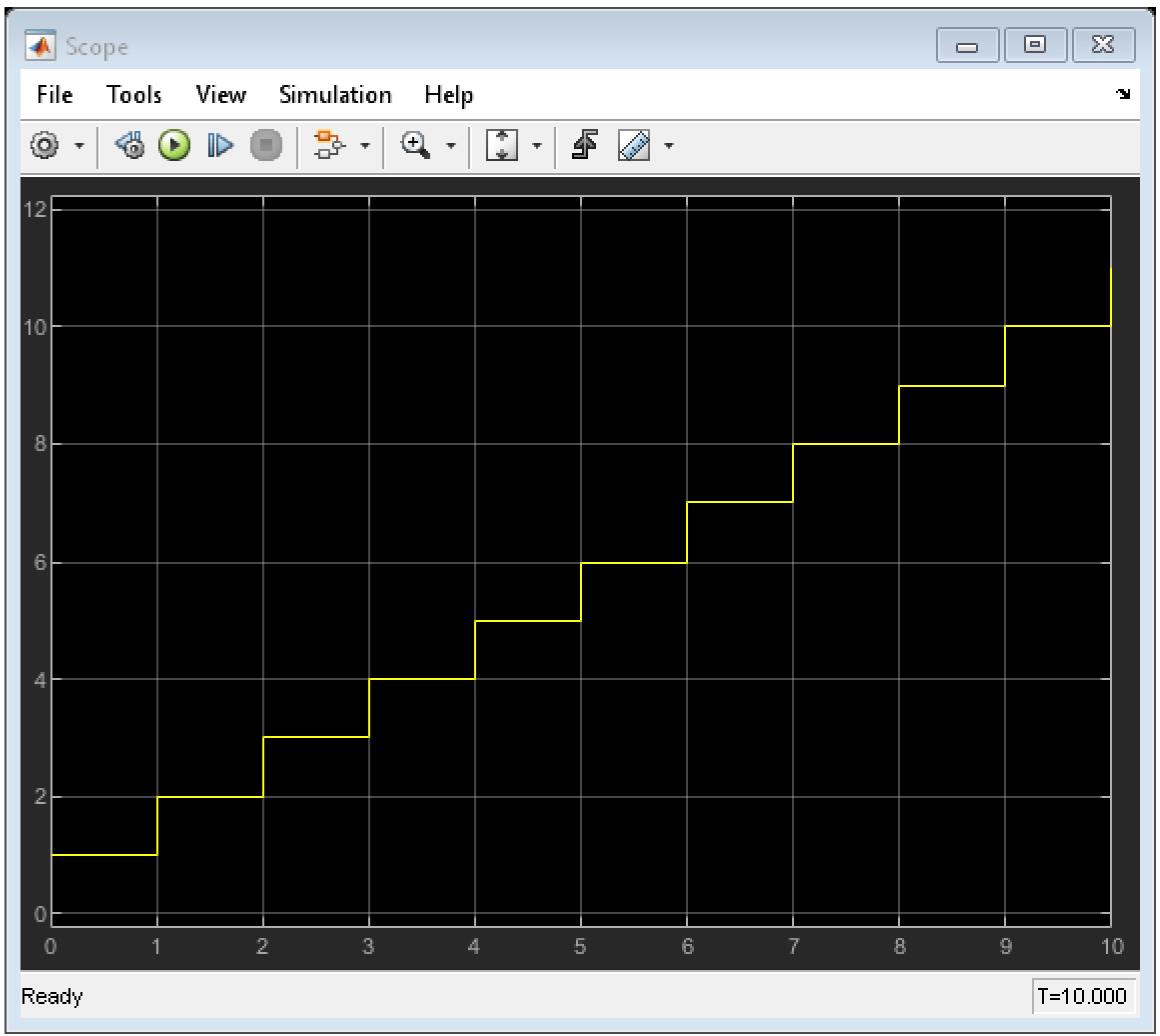
The function `lib1_func` is defined in the library chart `lib1Chart`. The function reads its input arguments and passes them to the function `lib2_func`.



The function `lib2_func` is defined in the library chart `lib2Chart`. The function reads its input arguments and passes them to the function `main_func`.



The function `main_func` is defined in the main chart. The function adds its input arguments and returns the result. The main chart stores this result as the output data `x`. The sequence of function calls repeats in each time step of the simulation. The Scope block shows the value of `x` increasing during the simulation.



Guidelines for Exporting Chart-Level Functions

Do Not Export Chart-Level Functions That Contain Unsupported Inputs or Outputs

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

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

Do Not Export Functions Across Model Reference Boundaries

You cannot export functions from a referenced model and call the functions from a parent model.

Combine Output and Update Functions When Generating Code

If you generate code for a model that uses exported chart-level functions, enable the model configuration parameter Single output/update function (Simulink Coder) to ensure consistent behavior between simulation and code generation.

See Also

Related Examples

- “Manage Queue for Shared Printer Server” on page 9-32
- “Reuse Logic Patterns by Defining Graphical Functions” on page 6-9
- “Reuse MATLAB Code by Defining MATLAB Functions” on page 7-2
- “Use Truth Tables to Model Combinatorial Logic” on page 8-2

Reuse Functions by Using Atomic Boxes

An atomic box is a graphical object that helps you encapsulate graphical, truth table, MATLAB, and Simulink functions in a separate namespace. Atomic boxes are not supported in standalone Stateflow charts in MATLAB. Atomic boxes allow for:

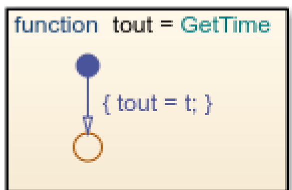
- Faster simulation after making small changes to a function in a chart with many states or levels of hierarchy
- Reuse of the same functions across multiple charts and models
- Ease of team development for people working on different parts of the same chart
- Manual inspection of generated code for a specific function in a chart

An atomic box looks opaque and includes the label **Atomic** in the upper left corner. If you use a linked atomic box from a library, the label **Link** appears in the upper left corner.

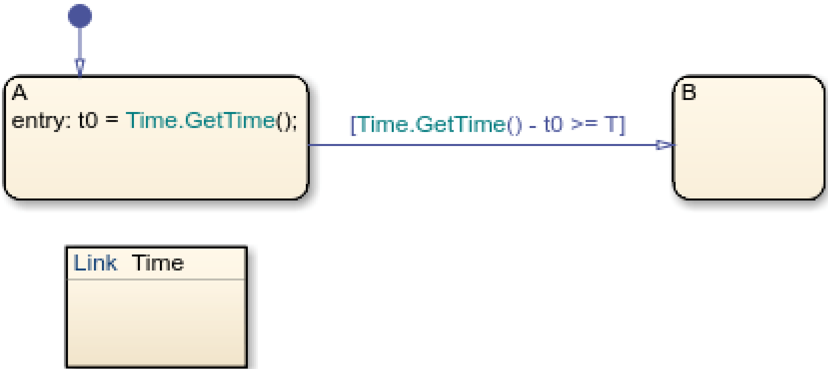
Example of an Atomic Box

This example shows how to use a linked atomic box to reuse a graphical function across multiple charts and models.

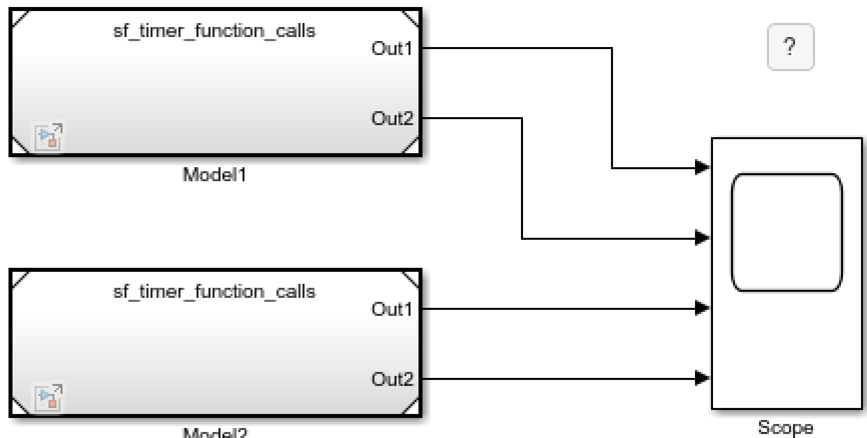
The function `GetTime` is defined in a chart in the library model `sf_timer_utils_lib`. The graphical function returns the simulation time in C charts where the equivalent MATLAB® function `getSimulationTime` is not available.



The model `sf_timer_function_calls` consists of two charts with a similar structure. Each chart contains a pair of states (`A` and `B`) and an atomic box (`Time`) linked from the library chart. The entry action in state `A` calls the function `GetTime` and stores its value as `t0`. The condition guarding the transition from `A` to `B` calls the function again and compares its output with the parameter `T`.



The top model `sf_timer_modelref` reuses the timer function in multiple referenced blocks. Because there are no exported functions, you can use more than one instance of the referenced block in the top model.



Benefits of Using Atomic Boxes

Atomic boxes combine the functionality of normal boxes and atomic subcharts. Atomic boxes:

- Improve the organization and clarity of complex charts.
- Support usage as library links.
- Support the generation of reusable code.
- Allow mapping of inputs, outputs, parameters, data store memory, and input events.

Atomic boxes contain only functions. They cannot contain states. Adding a state to an atomic box results in a compilation-time error.

To call a function that resides in an atomic box from a location outside the atomic box, use dot notation to specify its full path:

`atomic_box_name.function_name`

Using the full path for the function call:

- Makes clear the dependency on the function in the linked atomic box.
- Avoids pollution of the global namespace.
- Does not affect the efficiency of generated code.

Create an Atomic Box

You can create an atomic box by converting an existing box or by linking a chart from a library model. After creating the atomic box, update the mapping of variables by right-clicking the atomic box and selecting **Subchart Mappings**. For more information, see "Map Variables for Atomic Subcharts and Boxes" on page 17-11.

Convert a Normal Box to an Atomic Box

To create a container for your functions that allows for faster debugging and code generation workflows, convert an existing box into an atomic box. In your chart, right-click a normal box and select **Group & Subchart > Atomic Subchart**. The label **Atomic** appears in the upper left corner of the box.

The conversion process gives the atomic box its own copy of every data object that the box accesses in the chart. Local data is copied as data store memory. The scope of other data, including input and output data, does not change.

Note If a box contains any states or messages, you cannot convert it to an atomic box.

Link an Atomic Box from a Library

To create a collection of functions for reuse across multiple charts and models, create a link from a library model. Copy a chart in a library model and paste it to a chart in another model. If the library chart contains only functions and no states, it appears as a linked atomic box with the label **Link** in the upper left corner.

This modeling method minimizes maintenance of reusable functions. When you modify the atomic box in the library, your changes propagate to the links in all charts and models.

If the library chart contains any states, then it appears as a linked atomic subchart in the chart. For more information, see “Create Reusable Subcomponents by Using Atomic Subcharts” on page 17-2.

Convert an Atomic Box to a Normal Box

Converting an atomic box back to a normal box removes all of its variable mappings by merging subchart-parented data objects with the chart-parented data to which they map.

1. If the atomic box is a library link, right-click the atomic box and select **Library Link > Disable Link**.
2. To convert an atomic box to a subcharted box, right-click the atomic box and clear the **Group & Subchart > Atomic Subchart** check box.
3. To convert the subcharted box back to a normal box, right-click the subchart and clear the **Group & Subchart > Subchart** check box.
4. If necessary, rearrange graphical objects in your chart.

You cannot convert an atomic box to a normal box if:

- The atomic box maps a parameter to an expression other than a single variable name. For example, mapping a parameter `data1` to one of these expressions prevents the conversion of an atomic box to a normal box:
  - `3`
  - `data2(3)`
  - `data2 + 3`
- Both of these conditions are true:
  - The atomic box contains MATLAB functions or truth table functions that use MATLAB as the action language.

- The atomic box does not map each variable to a variable of the same name in the main chart.

When to Use Atomic Boxes

Debug Functions Incrementally

Suppose that you want to test a sequence of changes to a library of functions. The functions are part of a chart that contains many states or several levels of hierarchy, so recompiling the entire chart can take a long time. If you define the functions in an atomic box, recompilation occurs for only the box and not for the entire chart. For more information, see “Reduce the Compilation Time of a Chart” on page 17-38.

Reuse Functions

Suppose that you have a set of functions for use in multiple charts and models. The functions reside in the library model to enable easier configuration management. To use the functions in another model, you can either:

- Configure the library chart to export functions and create a link to the library chart in the model.
- Link the library chart as an atomic box in each chart of the model.

Models that use these functions can appear as referenced blocks in a top model. When the functions are exported, you can use only one instance of that referenced block for each top model. For more information, see “Model Reference Requirements and Limitations” (Simulink).

With atomic boxes, you can avoid this limitation. Because there are no exported functions in the charts, you can use more than one instance of the referenced block in the top model.

Develop Charts Used by Multiple People

Suppose that multiple people are working on different parts of a chart. If you store each library of functions in a linked atomic box, different people can work on different libraries without affecting the other parts of the chart. For more information, see “Divide a Chart into Separate Units” on page 17-41.

Inspect Generated Code

Suppose that you want to inspect code generated by Simulink Coder or Embedded Coder manually for a specific function. You can specify that the code for an atomic box appears in a separate file to avoid searching through unrelated code. For more information, see “Generate Code from Atomic Subcharts” on page 29-16.

See Also

More About

- “Group Chart Objects by Using Boxes” on page 6-2
- “Create Reusable Subcomponents by Using Atomic Subcharts” on page 17-2
- “Map Variables for Atomic Subcharts and Boxes” on page 17-11

Add Descriptive Comments in a Chart

You can enter comments or annotations in any location on a Stateflow chart. Annotations can contain any combination of:

- Text.
- Images.
- Equations using TeX commands.
- Hyperlinks that open a website or perform MATLAB functions. See “Annotate Models” (Simulink).

To create an annotation:

1. Double-click in the desired location of the chart. An annotation box opens.
2. In the annotation box, type your comments. To start a new line, press the **Enter** key.
3. After you finish typing, click outside the annotation box.

Alternatively, in the object palette, click the Annotation icon or the Image icon . Then, on the chart canvas, click the location for the new annotation.

Change Annotation Properties

You can change the style of an existing annotation by using the annotation formatting toolbar, the annotation context menu, or the Annotation properties dialog box.

- To access the annotation formatting toolbar, double-click the annotation text. Buttons on the formatting toolbox enable you to change font styles, text alignment, colors, and other options. For more information, see “Annotate Models” (Simulink).

- To open the annotation context menu, right-click the annotation text and select one of these options:
  - **Format** — choose font size, style (bold or italics), and whether to display a drop shadow around the annotation text.
  - **Text Alignment** — choose between left, center, and right justified text.
  - **Enable TeX Commands** — include TeX formatting commands in the annotation text. See “Include TeX Formatting Instructions” on page 6-23.
- To open the Annotation properties dialog box, right-click the annotation text and select **Properties**. You can specify the layout of the annotation, including:
  - Fixed height and width options.
  - Text and background color.
  - Text alignment.

- Margins between the text and the border of the annotation.

Include TeX Formatting Instructions

In your annotations, you can embed a subset of TeX commands to produce special characters such as Greek letters and mathematical symbols. For example, suppose that you enter this annotation text:

```
\it{\omega_N = e^{(-2\pii)/N}}
```

When you select the **Enable TeX Commands** annotation property, the chart renders this annotation like this:

$$\omega_N = e^{(-2\pi i)/N}$$

For a list of more information on using TeX commands in annotations, see “Annotate Models” (Simulink).

See Also

More About

- “Annotate Models” (Simulink)

MATLAB Functions in Stateflow Charts

- “Reuse MATLAB Code by Defining MATLAB Functions” on page 7-2
- “Program a MATLAB Function in a Chart” on page 7-7
- “Access Simulink Bus Signals in MATLAB Functions” on page 7-13
- “Debug a MATLAB Function in a Chart” on page 7-16

Reuse MATLAB Code by Defining MATLAB Functions

A MATLAB function in a Stateflow chart is a graphical element that you use to write algorithms that are easier to implement by calling built-in MATLAB functions. Typical applications include:

- Matrix-oriented calculations
- Data analysis and visualization

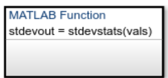
This type of function is useful for coding algorithms that are more easily expressed by using MATLAB instead of the graphical Stateflow constructs. MATLAB functions also provide optimizations for generating efficient, production-quality C code for embedded applications.

A MATLAB function can reside anywhere in a chart, state, or subchart. The location of the function determines the set of states and transitions that can call the function.

- If you want to call the function 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 a chart, put your MATLAB function at the chart level.
- If you want to call the function from any chart in your model, put your MATLAB function at the chart level and enable exporting of chart-level functions. For more information, see “Export Stateflow Functions for Reuse” on page 6-14.

Note A MATLAB function can access chart and state data above it in the Stateflow hierarchy.

For example, this MATLAB function has the name `stdevstats`. It takes an argument `vals` and returns an output value `stdevout`.



To compute the standard deviation of the values in `vals`, the function uses this code.

```
function stdevout = stdevstats(vals)
%#codegen

% Calculates the standard deviation for vals

len = length(vals);
stdevout = sqrt(sum(((vals-avg(vals,len)).^2))/len);

function mean = avg(array,size)
mean = sum(array)/size;
```

Define a MATLAB Function in a Chart

1. In the object palette, click the MATLAB function icon .
2. On the chart canvas, click the location for the new MATLAB function.

3 Enter the signature label for the function.

   The signature label of the function specifies a name for your function and the formal names for its arguments and return values. A signature label has this syntax:

```
[return_val1,return_val2,...] = function_name(arg1,arg2,...)
```

   You can specify multiple return values and multiple input arguments. Each return value and input argument can be a scalar, vector, or matrix of values. For functions with only one return value, omit the brackets in the signature label.

   You can use the same variable name for both arguments and return values. When you use the same variable for an input and output, you create in-place data. For example, a function with this signature label uses the variables `y1` and `y2` as both inputs and outputs:

```
[y1,y2,y3] = f(y1,u,y2)
```

   If you export this function to C code, the generated code treats `y1` and `y2` as in-place arguments passed by reference. Using in-place data reduces the number of times that the generated code copies intermediate data, which results in more efficient code.

   In the **Symbols** pane and the Model Explorer, the arguments and return values of the function signature appear as data items that belong to your function. Arguments have the scope `Input`. Return values have the scope `Output`.
4 Specify the data properties for each argument and return value, as described in “Set Data Properties” on page 10-5. When an argument and a return value have the same name, you can edit properties only for the argument. The properties for the return value are read-only.
5 To program the function, open the MATLAB Function Block Editor by double-clicking the function box.
6 In the MATLAB Function Block Editor, enter the MATLAB code implementing your function. For more information, see “Program a MATLAB Function in a Chart” on page 7-7.
7 Create any additional data items required by your function. For more information, see “Add Data Through the Model Explorer” on page 10-3.

   Your function can access its own data or data belonging to parent states or the chart. The data items in the function can have one of these scopes:

   - `Constant` — Constant data retains its initial value through all function calls.
   - `Parameter` — Parameter data retains its initial value through all function calls.

   In MATLAB functions, you do not need to create local or temporary function data explicitly. Instead, you can use undefined variables to store values that are accessible only during the rest of the function call. To store values that persist across function calls, use local data at the chart level or use the keyword `persistent`.

Call MATLAB Functions in States and Transitions

You can call MATLAB functions from the actions of any state or transition or from other functions. If you export a MATLAB function, you can call it from any chart in the model. For more information about exporting functions, see “Export Stateflow Functions for Reuse” on page 6-14.

To call a MATLAB function, use the function signature and include an actual argument value for each formal argument in the function signature.

```
[return_val1,return_val2,...] = function_name(arg1,arg2,...)
```

If the data types of the actual and formal arguments differ, the function casts the actual argument to the type of the formal argument.

Specify Properties of MATLAB Functions

The properties listed below specify how a MATLAB function interacts with the other components in your Stateflow chart. You can modify these properties in the **Property Inspector**, the Model Explorer, or the MATLAB Function properties dialog box.

To use the **Property Inspector**:

1. In the **Modeling** tab, under **Design Data**, select **Property Inspector**.
2. In the Stateflow Editor, select the MATLAB function.
3. In the **Property Inspector**, edit the transition properties.

To use the Model Explorer:

1. In the **Modeling** tab, under **Design Data**, select **Model Explorer**.
2. In the **Model Hierarchy** pane, select the MATLAB function.
3. In the **Dialog** pane, edit the MATLAB function properties.

To use the MATLAB Function properties dialog box:

1. In the Stateflow Editor, right-click the MATLAB function.
2. Select **Properties**.
3. In the properties dialog box, edit the MATLAB function properties.

You can also modify these properties programmatically by using `Stateflow.EMFunction` objects. For more information about the Stateflow programmatic interface, see “Overview of the Stateflow API”.

Name

Function name. Click the function name link to open your function in the MATLAB Function Block Editor.

Function Inline Option

Controls the inlining of your function in generated code:

- `Auto` — Determines whether to inline your function based on an internal calculation.
- `Inline` — Inlines your function if you do not export it to other charts and it is not part of a recursion. (A recursion exists if your function calls itself directly or indirectly through another function call.)
- `Function` — Does not inline your function.

This property is not available in the **Property Inspector**.

Label

Signature label for your function. The function signature label specifies a name for your function and the formal names for its arguments and return values. This property is not available in the **Property Inspector**.

Saturate on Integer Overflow

Specifies whether integer overflows saturate in the generated code. For more information, see "Handle Integer Overflow for Chart Data" on page 10-37.

This property applies only to MATLAB functions in charts that use C as the action language. In charts that use MATLAB as the action language, the behavior of data depends on the value of the "Saturate on integer overflow" on page 1-22 property for the chart.

This property is not available in the **Property Inspector**.

MATLAB Function fimath

Defines the `fimath` properties for the MATLAB function. The `fimath` properties specified are associated with all `fi` and `fimath` objects constructed in the MATLAB function. Choose one of these options:

- **Same as MATLAB** — The function uses the same `fimath` properties as the current global `fimath`. The edit box appears dimmed and displays the current global `fimath` in read-only form. For more information on the global `fimath` and `fimath` objects, see the Fixed-Point Designer documentation.
- **Specify Other** — Specify your own `fimath` object by one of these methods:
  - Construct the `fimath` object inside the edit box.
  - Construct the `fimath` object in the MATLAB or model workspace and enter its variable name in the edit box.

This property applies only to MATLAB functions in charts that use C as the action language. In charts that use MATLAB as the action language, the behavior of data depends on the value of the "MATLAB Chart fimath" on page 1-24 property for the chart.

This property is not available in the **Property Inspector**.

Description

Description of the MATLAB function.

Document Link

Link to online documentation for the MATLAB function. You can enter a web URL address or a MATLAB command that displays documentation as an HTML file or as text in the MATLAB Command Window. When you click the **Document link** hyperlink, Stateflow evaluates the link and displays the documentation.

See Also

Objects
`Stateflow.EMFunction`

Tools
Model Explorer

More About

- "Program a MATLAB Function in a Chart" on page 7-7
- "Export Stateflow Functions for Reuse" on page 6-14
- "Reuse Functions by Using Atomic Boxes" on page 6-18
- "Enhance Code Readability for MATLAB Function Blocks" (Embedded Coder)

Program a MATLAB Function in a Chart

A MATLAB function in a Stateflow chart is a graphical element that you use to write algorithms that are easier to implement by calling built-in MATLAB functions. This type of function is useful for coding algorithms that are more easily expressed by using MATLAB instead of the graphical Stateflow constructs. For more information, see “Reuse MATLAB Code by Defining MATLAB Functions” on page 7-2.

Inside a MATLAB function, you can call these types of functions:

- Local functions defined in the body of the MATLAB function.
- Graphical, Simulink, truth table, and other MATLAB functions in the chart.
- Built-in MATLAB functions that support code generation. These functions generate C code for building targets that conform to the memory and data type requirements of embedded environments.
- Extrinsic MATLAB functions that do not support code generation. These functions execute only in the MATLAB workspace during simulation of the model. For more information, see “Call Extrinsic MATLAB Functions in Stateflow Charts” on page 14-13.
- Simulink Design Verifier™ functions for property proving and test generation. These functions include:
  - `sldv.prove`
  - `sldv.assume`
  - `sldv.test`
  - `sldv.condition`

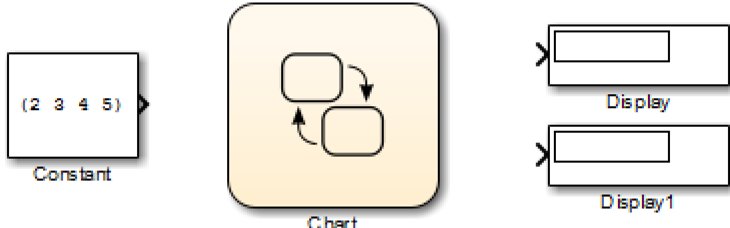
This example shows how to create a model with a Stateflow chart that calls two MATLAB functions, `meanstats` and `stdevstats`:

- `meanstats` calculates the mean of the values in `vals`.
- `stdevstats` calculates a standard deviation for the values in `vals`.

Build Model

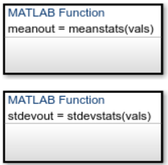
Follow these steps:

1. Create a new model with the following blocks:



2. Save the model as `call_stats_function_stateflow`.
3. In the model, double-click the Chart block.

4 In the object palette, use the MATLAB function icon to add two functions in the empty chart.
5 Label each function as shown:



You must label a MATLAB function with its signature. Use the following syntax:

```
[return_val1,return_val2,...] = function_name(arg1,arg2,...)
```

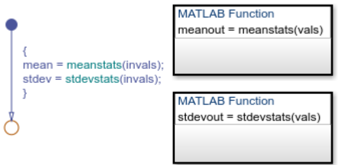
You can specify multiple return values and multiple input arguments, as shown in the syntax. Each return value and input argument can be a scalar, vector, or matrix of values.

Tip For MATLAB functions with only one return value, you can omit the brackets in the signature label.

6 In the chart, draw a default transition into a terminating junction with this condition action:

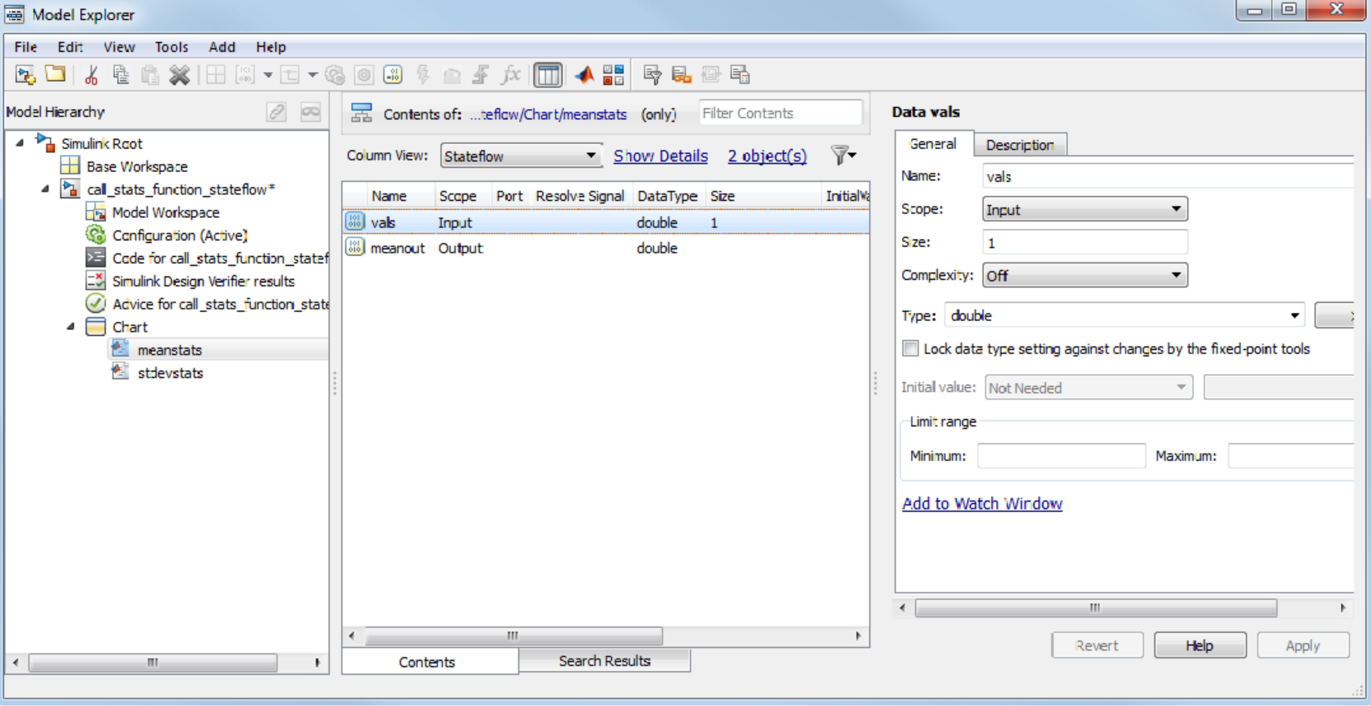
```
{
mean = meanstats(invals);
stdev = stdevstats(invals);
}
```

The chart should look something like this:



Tip If the formal arguments of a function signature are scalars, verify that inputs and outputs of function calls follow the rules of scalar expansion. For more information, see “Assign Values to All Elements of a Matrix” on page 19-6.

7 In the **Modeling** tab, under **Design Data**, select **Model Explorer**.
8 In the **Model Hierarchy** pane of the Model Explorer, select the function `meanstats`.



The **Contents** pane displays the input argument `vals` and output argument `meanout`. Both are scalars of type `double` by default.

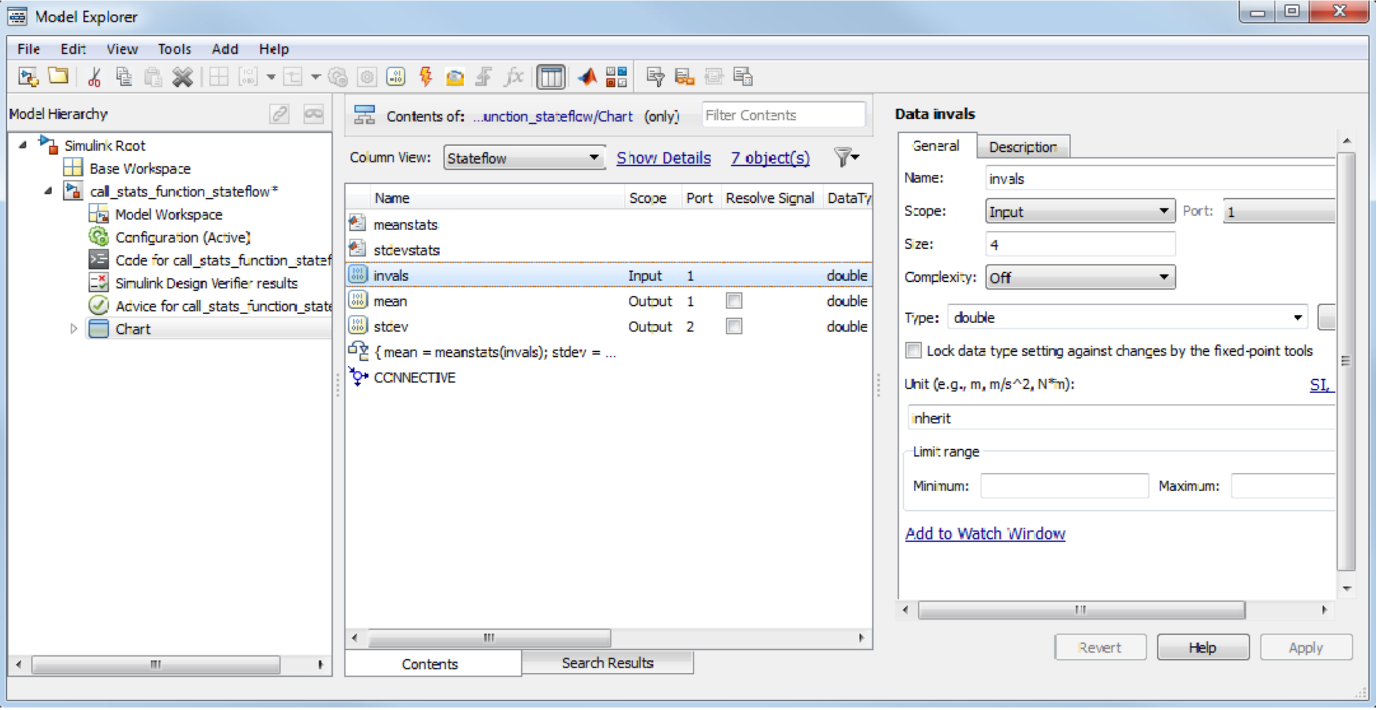
9 Double-click the `vals` row under the **Size** column to set the size of `vals` to `4`.

10 In the **Model Hierarchy** pane of the Model Explorer, select the function `stdevstats` and repeat the previous step.

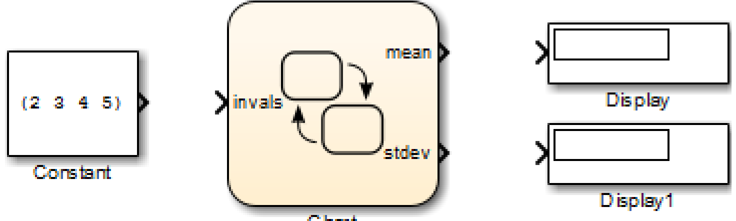
11 In the **Model Hierarchy** pane of the Model Explorer, select **`Chart`** and add the following data:

Name	Scope	Size
`invals`	`Input`	4
`mean`	`Output`	Scalar (no change)
`stdev`	`Output`	Scalar (no change)

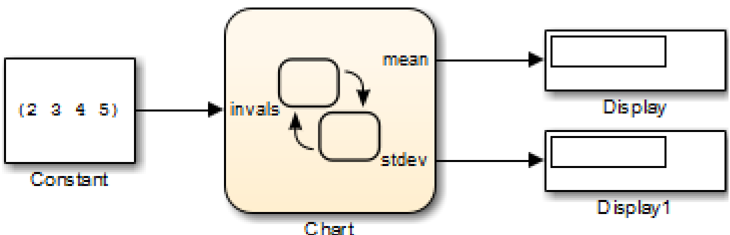
You should now see the following data in the Model Explorer.



After you add the data `invals`, `mean`, and `stdev` to the chart, the corresponding input and output ports appear on the Stateflow block in the model.



12 Connect the Constant and Display blocks to the ports of the Chart block and save the model.



Program MATLAB Functions

To program the functions `meanstats` and `stdevstats`, follow these steps:

1 Open the chart in the model `call_stats_function_stateflow`.

2 In the chart, open the function `meanstats`.

   The function editor appears with the header:

```
function meanout = meanstats(vals)
```

   This header is taken from the function label in the chart. You can edit the header directly in the editor, and your changes appear in the chart after you close the editor.

3 On the line after the function header, enter:

```
%#codegen
```

   The `%#codegen` compilation directive helps detect compile-time violations of syntax and semantics in MATLAB functions supported for code generation.

4 Enter a line space and this comment:

```
% Calculates the statistical mean for vals
```

5 Add the line:

```
len = length(vals);
```

   The function `length` is an example of a built-in MATLAB function that is supported for code generation. You can call this function directly to return the vector length of its argument `vals`. When you build a simulation target, the function length is implemented with generated C code. Functions supported for code generation appear in “Functions and Objects Supported for C/C++ Code Generation” (MATLAB Coder).

   The variable `len` is an example of implicitly declared local data. It has the same size and type as the value assigned to it — the value returned by the function `length`, a scalar `double`. To learn more about declaring variables, see “Numeric Types” (MATLAB Coder).

   The MATLAB function treats implicitly declared local data as temporary data, which exists only when the function is called and disappears when the function exits. You can declare local data for a MATLAB function in a chart to be persistent by using the `persistent` construct.

6 Enter this line to calculate the value of `meanout`:

```
meanout = avg(vals,len);
```

   The function `meanstats` stores the mean of `vals` in the Stateflow data `meanout`. Because these data are defined for the parent Stateflow chart, you can use them directly in the MATLAB function.

   Two-dimensional arrays with a single row or column of elements are treated as vectors or matrices in MATLAB functions. For example, in `meanstats`, the argument `vals` is a four-element vector. You can access the fourth element of this vector with the matrix notation `vals(4,1)` or the vector notation `vals(4)`.

   The MATLAB function uses the functions `avg` and `sum` to compute the value of `mean`. `sum` is a function supported for code generation. `avg` is a local function that you define later. When resolving function names, MATLAB functions in your chart look for local functions first, followed by functions supported for code generation.

Note If you call a function that the MATLAB function cannot resolve as a local function or a function for code generation, you must declare the function to be extrinsic.

---

7 Now enter this statement:

```
coder.extrinsic("plot");
```

8 Enter this line to plot the input values in `vals` against their vector index.

```
plot(vals,"-+");
```

Recall that you declared `plot` to be an extrinsic function because it is not supported for code generation. When the MATLAB function encounters an extrinsic function, it sends the call to the MATLAB workspace for execution during simulation.

9 Now, define the local function `avg`, as follows:

```
function mean = avg(array,size)
mean = sum(array)/size;
```

The header for `avg` defines two arguments, `array` and `size`, and a single return value, `mean`. The local function `avg` calculates the average of the elements in `array` by dividing their sum by the value of argument `size`.

The complete code for the function `meanstats` looks like this:

```
function meanout = meanstats(vals)
%#codegen

% Calculates the statistical mean for vals

len = length(vals);
meanout = avg(vals,len);

coder.extrinsic("plot");
plot(vals,"-+");

function mean = avg(array,size)
mean = sum(array)/size;
```

10 Save the model.

11 Back in the chart, open the function `stdevstats` and add code to compute the standard deviation of the values in `vals`. The complete code should look like this:

```
function stdevout = stdevstats(vals)
%#codegen

% 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.

Access Simulink Bus Signals in MATLAB Functions

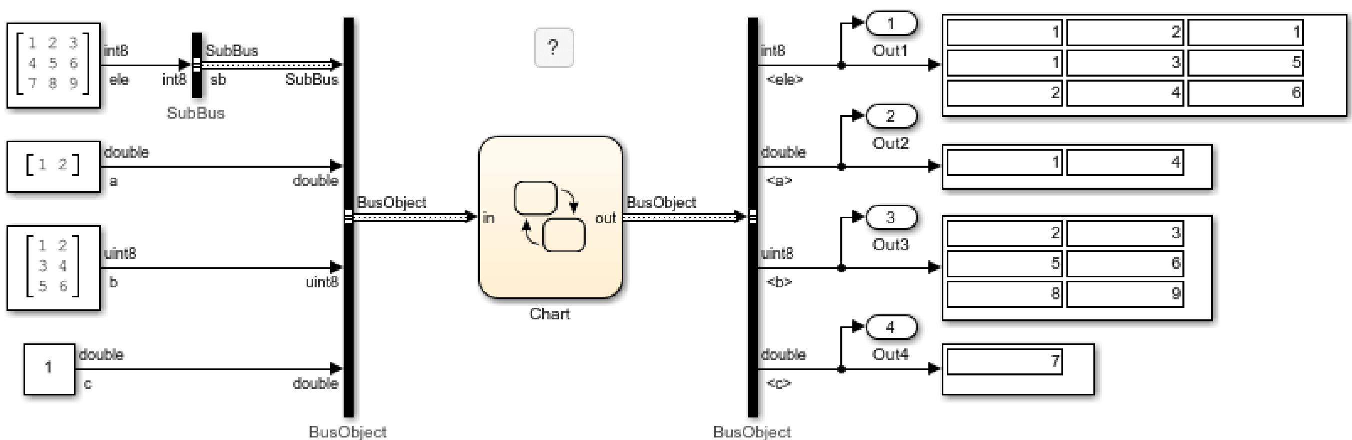
This example shows how to read from and write to Simulink® bus signals in a MATLAB® function by using MATLAB and Stateflow® structures. MATLAB structures enable you to bundle data of different sizes and types into a single variable. You can create a MATLAB structure to:

- Store related data in a local or persistent variable of a MATLAB function
- Read from or write to a local Stateflow structure
- Interface with a Simulink bus signal at an input or output port

MATLAB functions support nonvirtual buses only. For more information, see "Composite Interface Guidelines" (Simulink).

Define Structures in MATLAB Functions

In this example, a Stateflow chart processes data from one Simulink bus signal and outputs the result to another Simulink bus signal. Both the input and output bus signals are defined by the `Simulink.Bus` (Simulink) object `BusObject`. These buses have four fields: `sb`, `a`, `b`, and `c`. The field `sb` is also a bus signal defined by the `Simulink.Bus` object `SubBus`. It has one field called `ele`.



In the chart, the Simulink bus signals interface with the Stateflow structures `in` and `out`. The function `sb2abc` extracts information from the input structure and stores it in the local Stateflow structure `localbus`. Then the chart writes to the output structure by combining the values of the local structure and one of the elements of the array of structures `subBusArray`. For more information on accessing and modifying the contents of a Stateflow structure or an array of Stateflow structures, see "Index and Assign Values to Stateflow Structures" on page 26-7.

```
A
en:
localbus = sb2abc(in.sb);
subBusArray(1) = in.sb;
subBusArray(2) = abc2sb(in);
subBusArray(3).ele = transpose(in.sb.ele);
subBusArray(4).ele = int8(magic(3));
out = localbus;
out.sb = subBusArray(idx);
```

MATLAB Function y = sb2abc(u)

```
function y = abc2sb(u)
{
y.ele = int8([u.a(1) u.a(2) u.c; transpose(u.b)]);
}
```

The MATLAB® function `sb2abc` takes a Stateflow structure of type `SubBus` and returns a Stateflow structure of type `BusObject`. The function decomposes the value of the field `ele` from its input into three components: a vector, a 3-by-2 matrix, and a scalar. The function stores these components in a local MATLAB `struct` that has the same fields as the `Simulink.Bus` object `BusObject`. Then the function assigns the value of the MATLAB `struct` to the output structure `y`.

```
function y = sb2abc(u)

% extract data from input structure

A = double(u.ele(1:2,1));
B = uint8(u.ele(:,2:3));
C = double(u.ele(3,1));

% create local structure

X = struct(ele=int8(zeros(3)));
Y = struct(sb=X,a=A,b=B,c=C);

% assign value to output structure

y = Y;

end
```

Define Input and Output Structures

In a MATLAB function, you can access a local Stateflow structure or interface with a Simulink bus signal by defining the input and output structures for the function:

1. In the base workspace, create a `Simulink.Bus` object that defines the structure data type.
2. In the **Symbols** pane, select the function input or output.
3. In the **Property Inspector**, set the **Type** property to `Bus: <object name>`. Replace <object name> with the name of the `Simulink.Bus` object that defines the Stateflow structure.

For example, in the function `sb2abc`:

- The **Type** property of the input structure `u` is specified as `Bus: SubBus`.
- The **Type** property of the output structure `y` is specified as `Bus: BusObject`.

For more information, see “Define Stateflow Structures” on page 26-2.

Define Local and Persistent Structure Variables

To store related data in a single variable inside a MATLAB function, you can create a MATLAB struct as a local or persistent variable. For example, the function sb2abc defines two local MATLAB structures to temporarily store the data extracted from the input structure u before writing to the output structure y:

- X is a scalar struct with a single field called ele. This field contains a 3-by-3 matrix of type int8, which matches the structure of the Simulink.Bus object SubBus.
- Y is a scalar struct with four fields: sb is a substructure of type SubBus, a is a two-dimensional vector of type double, b is a 3-by-2 matrix of type uint8, and c is a scalar of type double. These fields match the structure of the Simulink.Bus object BusObject.

For more information, see “Define Scalar Structures for Code Generation” (Simulink).

See Also

struct | Simulink.Bus

More About

- “Access Bus Signals Through Stateflow Structures” on page 26-2
- “Index and Assign Values to Stateflow Structures” on page 26-7
- “Identify Data by Using Dot Notation” on page 10-39
- “Define Scalar Structures for Code Generation” (Simulink)

Debug a MATLAB Function in a Chart

Check MATLAB Functions for Syntax Errors

Before you can build a simulation application for a model, you must fix syntax errors. Follow these steps to check the MATLAB function `meanstats` for syntax violations:

1. Open the function `meanstats` inside the chart in the `call_stats_function_stateflow` model that you constructed in "Program a MATLAB Function in a Chart" on page 7-7.

   The editor automatically checks your function code for errors and recommends corrections.
2. In the **Apps** tab, click **Simulink Coder**. In the **C Code** tab, click **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 local function `avg` to a nonexistent local function `aug` in `meanstats`, errors appear in the Diagnostic Viewer.
3. The diagnostic message provides details of the type of error and a link to the code where the error occurred. The diagnostic message also contains a link to a diagnostic report that provides links to your MATLAB functions and compile-time type information for the variables and expressions in these functions. If your model fails to build, this information simplifies finding sources of error messages and aids understanding of type propagation rules. For more information about this report, see "MATLAB Function Reports" (Simulink).
4. In the diagnostic message, click the link after the function name `meanstats` to display the offending line of code.

   The offending line appears highlighted in the editor.
5. Correct the error by changing `aug` back to `avg` and recompile. No errors are found and the compile completes successfully.

Run-Time Debugging for MATLAB Functions in Charts

You use simulation to test your MATLAB functions for run-time errors that are not detectable by Stateflow. When you simulate your model, your MATLAB functions undergo diagnostic tests for missing or undefined information and possible logical conflicts as described in "Check MATLAB Functions for Syntax Errors" on page 7-16. 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 line number of this line:

   ```
   len = length(vals);
   ```

   The line number is highlighted in red, indicating that you have set a breakpoint.
2. Start simulation for the model.

   If you get any errors or warnings, make corrections before you try to simulate again. Otherwise, simulation pauses when execution reaches the breakpoint you set. The line of code is highlighted, indicating this pause.
3. To advance execution to the next line, select **Step Over**.

Notice that this line calls the local function `avg`. If you select **Step Over** here, execution advances past the execution of the local function `avg`. To track execution of the lines in the local function `avg`, select **Step In** instead.

4. To advance execution to the first line of the called local function `avg`, select **Step In**.

   Once you are in the local function, you can advance through one line at a time with the **Step Over** tool. If the local function calls another local function, use the **Step In** tool to step into it. To continue through the remaining lines of the local function and go back to the line after the local function call, select **Step Out**.
5. Select **Step Over** to execute the only line in `avg`.
6. Select **Step Over** to return to the function `meanstats`.

   Execution advances to the line after the call to `avg`.
7. To display the value of the variable `len`, point to the text `len` in the function editor for at least a second.

   The value of `len` appears adjacent to your pointer.

   You can display the value for any data in the MATLAB function in this way, no matter where it appears in the function. For example, you can display the values for the vector `vals` by placing your pointer over it as an argument to the function `length`, or as an argument in the function header.

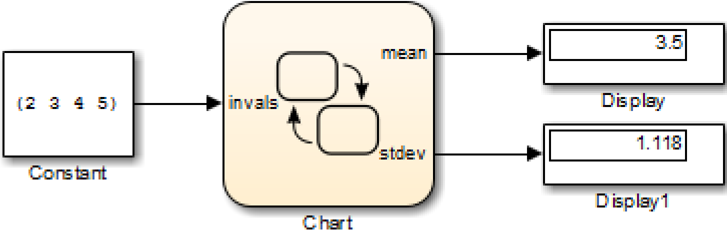
   You can also report the values for MATLAB function data in the MATLAB Command Window during simulation. When you reach a breakpoint, the `debug>>` command prompt appears in the MATLAB Command Window (you may have to press **Enter** to see it). At this prompt, you can inspect data defined for the function by entering the name of the data, as shown in this example:

```
debug>> len

len =
     4
```

   As another debugging alternative, you can display the execution result of a function line by omitting the terminating semicolon. If you do, execution results appear in the MATLAB Command Window during simulation.
8. To leave the function until it is called again and the breakpoint is reached, select **Continue**.

   At any point in a function, you can advance through the execution of the remaining lines of the function with the **Continue** tool. If you are at the end of the function, selecting **Step Over** completes the same action.
9. Click the breakpoint to remove it and click **Stop** to complete the simulation.

   In the model, the computed values of `mean` and `stdev` now appear in the Display blocks.



Check for Data Range Violations

To control the level of diagnostic action for data range violations, open the Configuration Parameters dialog box and, in the **Diagnostics** > **Data Validity** pane, set the **Simulation range checking** parameter to `none`, `warning`, or `error`. The default setting is `none`. For more information, see "Simulation range checking" (Simulink).

Specify a Range

To specify a range for input and output data, follow these steps:

1. In the Model Explorer, select the MATLAB function input or output of interest.

   The Data properties dialog box opens in the **Dialog** pane of the Model Explorer.
2. In the Data properties dialog box, click the **General** tab and enter a limit range, as described in "Limit range" on page 10-10.

Truth Table Functions for Decision-Making Logic

- "Use Truth Tables to Model Combinatorial Logic" on page 8-2
- "Program a Truth Table" on page 8-8
- "Debug Errors in a Truth Table" on page 8-21
- "Correct Overspecified and Underspecified Truth Tables" on page 8-28
- "Home Climate Control Using the Truth Table Block" on page 8-32

Use Truth Tables to Model Combinatorial Logic

Truth tables implement combinatorial logic design in a tabular format. You can use Stateflow truth tables to model decision making for fault detection and management and mode switching.

Truth table functions in a Stateflow chart execute only when you call the truth table function. You can define a truth table function in a Stateflow chart, state, or subchart. The location of the function determines the set of states and transitions that can call the function.

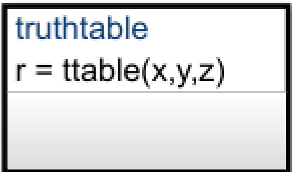
- If you want to call the function from within one state or subchart and its substates, put your truth table function in that state or subchart. That function overrides any other functions of the same name in the parents and ancestors of that state or subchart.
- If you want to call the function anywhere in a chart, put your truth table function at the chart level.
- If you want to call the function from any chart in your model, put your truth table at the chart level and enable exporting of chart-level functions. For more information, see “Export Stateflow Functions for Reuse” on page 6-14.

A truth table function can access chart and state data above it in the Stateflow hierarchy.

Alternatively, for a more direct implementation of decision logic, you can add a Truth Table block directly to your Simulink model. To implement control logic, Truth Table blocks use MATLAB as the action language.

Layout of a Truth Table

This truth table function has the name `ttable`. It takes three arguments (`x`, `y`, and `z`) and returns one output value (`r`).



The function consists of this arrangement of conditions, decisions, and actions.

Condition	Decision 1	Decision 2	Decision 3	Decision 4
`x == 1`	T	F	F	-
`y == 1`	F	T	F	-
`z == 1`	F	F	T	-
Action	`r = 1`	`r = 2`	`r = 3`	`r = 4`

Each of the conditions entered in the **Condition** column must evaluate to true (nonzero value) or false (zero value). Outcomes for each condition are specified as T (true), F (false), or - (true or false). Each of the decision columns combines an outcome for each condition with a logical AND into a compound condition, which is referred to as a decision.

You evaluate a truth table one decision at a time, starting with **Decision 1**. The **Decision 4** covers all possible remaining decisions. If one of the decisions is true, the table perform the associated action, and then the truth table execution is complete.

For example, if conditions `x == 1` and `y == 1` are false and condition `z == 1` is true, then **Decision 3** is true and the variable `r` is set equal to 3. The remaining decisions are not tested and evaluation of the truth table is finished. If the first three decisions are false, then the default decision is automatically true and its action (`r=4`) is executed. This table lists pseudocode corresponding to the evaluation of this truth table example.

Pseudocode	Description
`if ((x == 1) & !(y == 1) & !(z == 1))` `  r = 1;`	If **Decision 1** is true, then set `r=1`.
`elseif (!(x == 1) & (y == 1) & !(z == 1))` `  r = 2;`	If **Decision 2** is true, then set `r=2`.
`elseif (!(x == 1) & !(y == 1) & (z == 1))` `  r = 3;`	If **Decision 3** is true, then set `r=3`.

`else`
`  r = 4;`
`endif`
	If all other decisions are false, then default decision is true. Set `r=4`.

Define a Truth Table Function

To define a truth table function:

1. In the object palette, click the truth table function icon.
2. On the chart canvas, click the location for the new truth table function.
3. Enter the signature label for the function.

   The signature label of the function specifies a name for your function and the formal names for its arguments and return values. A signature label has this syntax:

   ```
   [return_val1,return_val2,...] = function_name(arg1,arg2,...)
   ```

   You can specify multiple return values and multiple input arguments. Each return value and input argument can be a scalar, vector, or matrix of values. For functions with only one return value, omit the brackets in the signature label.

   You can use the same variable name for both arguments and return values. When you use the same variable for an input and output, you create in-place data. For example, a function with this signature label uses the variables `y1` and `y2` as both inputs and outputs:

   ```
   [y1,y2,y3] = f(y1,u,y2)
   ```

   If you export this function to C code, the generated code treats `y1` and `y2` as in-place arguments passed by reference. Using in-place data reduces the number of times that the generated code copies intermediate data, which results in more efficient code.

   In the **Symbols** pane and the Model Explorer, the arguments and return values of the function signature appear as data items that belong to your function. Arguments have the scope `Input`. Return values have the scope `Output`.

4. Specify the data properties for each argument and return value, as described in "Set Data Properties" on page 10-5. When an argument and a return value have the same name, you can edit properties only for the argument. The properties for the return value are read-only.
5. To program the function, open the truth table editor by double-clicking the function box.
6. In the truth table editor, add conditions, decisions, and actions. For more information, see "Program a Truth Table" on page 8-8.
7. Create any additional data items required by your function. For more information, see "Add Data Through the Model Explorer" on page 10-3.

   Your function can access its own data or data belonging to parent states or the chart. The data items in the function can have one of these scopes:

   - `Constant` — Constant data retains its initial value through all function calls.
   - `Parameter` — Parameter data retains its initial value through all function calls.
   - `Local` — Local data persists from one function call to the next function call. Valid only for truth tables that use C as the action language.
   - `Temporary` — Temporary data initializes at the start of every function call. Valid only for truth tables that use C as the action language.

   In truth table functions that use C as the action language, define local data when you want your data values to persist across function calls throughout the simulation. Define temporary data when you want to initialize data values at the start of every function call. For example, you can define a counter with `Local` scope if you want to track the number of times that you call the function. In contrast, you can designate a loop counter to have `Temporary` scope if you do not need the counter value to persist after the function completes.

   In truth table functions that use MATLAB as the action language, you do not need to define local or temporary function data. Instead, in these functions, you can use undefined variables to store values that are accessible only during the rest of the function call. To store values that persist across function calls, use local data at the chart level.

   Tip You can initialize local and temporary data in your function from the MATLAB workspace. For more information, see "Initialize Data from the MATLAB Base Workspace" on page 10-30.

Call Truth Table Functions in States and Transitions

You can call truth table functions from the actions of any state or transition or from other functions. If you export a truth table function, you can call it from any chart in the model. For more information about exporting functions, see "Export Stateflow Functions for Reuse" on page 6-14.

To call a truth table function, use the function signature and include an actual argument value for each formal argument in the function signature.

```
[return_val1,return_val2,...] = function_name(arg1,arg2,...)
```

If the data types of the actual and formal arguments differ, the function casts the actual argument to the type of the formal argument.

Specify Properties of Truth Table Functions

The properties listed below specify how a truth table function interacts with the other components in your Stateflow chart. You can modify these properties in the **Property Inspector**, the Model Explorer, or the Truth Table properties dialog box.

To use the **Property Inspector**:

1. In the **Modeling** tab, under **Design Data**, select **Property Inspector**.
2. In the Stateflow Editor, select the truth table function.
3. In the **Property Inspector**, edit the truth table function properties.

To use the Model Explorer:

1. In the **Modeling** tab, under **Design Data**, select **Model Explorer**.
2. In the **Model Hierarchy** pane, select the truth table function.
3. In the **Dialog** pane, edit the truth table function properties.

To use the Truth Table properties dialog box:

1. In the Stateflow Editor, right-click the truth table function.
2. Select **Properties**.
3. In the properties dialog box, edit the truth table function properties.

You can also modify these properties programmatically by using `Stateflow.TruthTable` objects. For more information about the Stateflow programmatic interface, see "Overview of the Stateflow API".

Name

Function name. Click the function name link to bring your function to the foreground in its native chart.

Function Inline Option

Controls the inlining of your function in generated code:

- `Auto` — Determines whether to inline your function based on an internal calculation.
- `Inline` — Inlines your function if you do not export it to other charts and it is not part of a recursion. (A recursion exists if your function calls itself directly or indirectly through another function call.)
- `Function` — Does not inline your function.

This property is not available in the **Property Inspector**.

Label

Signature label for your function. The function signature label specifies a name for your function and the formal names for its arguments and return values. This property is not available in the **Property Inspector**.

Underspecification

Controls the level of diagnostics for underspecification in your truth table function. For more information, see “Correct Overspecified and Underspecified Truth Tables” on page 8-28.

Overspecification

Controls the level of diagnostics for overspecification in your truth table function. For more information, see “Correct Overspecified and Underspecified Truth Tables” on page 8-28.

Action Language

Controls the action language for your Stateflow truth table function. Choose between MATLAB or C. This property is available only In charts that use C as the action language. For more information, see “Differences Between MATLAB and C as Action Language Syntax” on page 15-4.

Description

Description of the truth table function.

Document Link

Link to online documentation for the truth table function. You can enter a web URL address or a MATLAB command that displays documentation as an HTML file or as text in the MATLAB Command Window. When you click the **Document link** hyperlink, Stateflow evaluates the link and displays the documentation.

Specify Properties for Truth Table Blocks

Truth Table block properties specify how your truth table interfaces with the Simulink model. You can modify these properties in the **Property Inspector**, the Model Explorer, or the Truth Table properties dialog box.

To use the **Property Inspector**:

1. In the **Modeling** tab, under **Design Data**, select **Property Inspector**.
2. In the Stateflow Editor, click the truth table.
3. In the **Property Inspector**, edit the truth table properties.

To use the Model Explorer:

1. In the **Modeling** tab, under **Design Data**, select **Model Explorer**.
2. In the **Model Hierarchy** pane, select the truth table.
3. In the **Dialog** pane, edit the truth table properties.

To use the Truth Table properties dialog box:

1. Open the Stateflow Editor.
2. In the **Modeling** tab, click **Table Properties**.
3. In the properties dialog box, edit the truth table properties.

You can also modify these properties programmatically by using `Stateflow.TruthTableChart` objects. For more information about the Stateflow programmatic interface, see “Overview of the Stateflow API”.

Tip Truth Table block properties are a combination of the properties of truth table functions and charts that use MATLAB as the action language. For a description of each property, see “Specify Properties of Truth Table Functions” on page 8-5 and “Specify Properties for Stateflow Charts” on page 1-19.

See Also

Blocks
Truth Table

Objects
`Stateflow.TruthTable` | `Stateflow.TruthTableChart`

Tools
Model Explorer

More About

- “Program a Truth Table” on page 8-8
- “Differences Between MATLAB and C as Action Language Syntax” on page 15-4
- “Export Stateflow Functions for Reuse” on page 6-14
- “Reuse Functions by Using Atomic Boxes” on page 6-18

Program a Truth Table

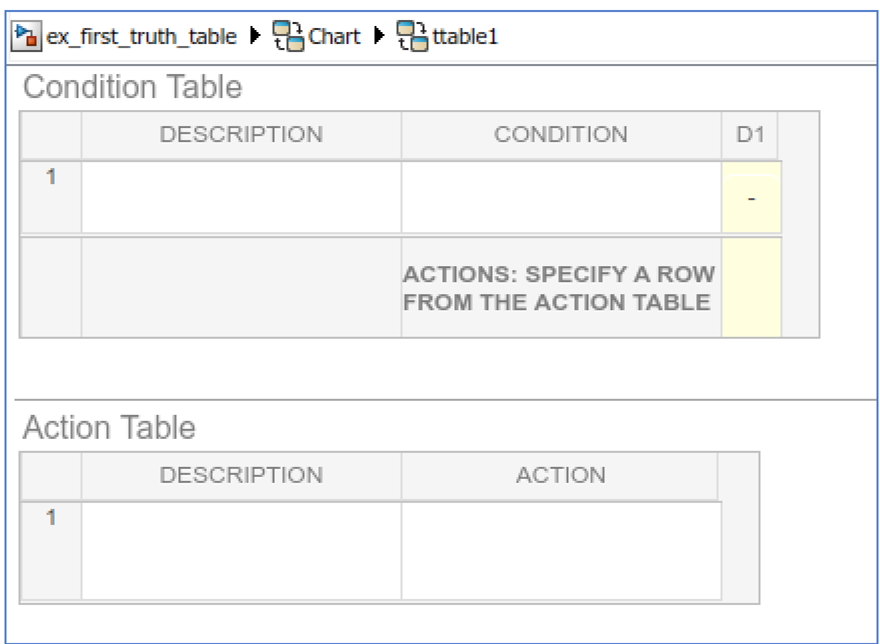
After you create a truth table, you can program it to execute according to your specifications. To program a truth table you add conditions, decisions, and actions. For more information about creating a truth table, see “Use Truth Tables to Model Combinatorial Logic” on page 8-2.

Truth tables are not supported in standalone Stateflow charts in MATLAB. For more information, see “Use Truth Tables to Model Combinatorial Logic” on page 8-2.

Open a Truth Table for Editing

After you create and label a truth table in a chart or Simulink model, you specify its logical behavior. These specifications apply to both the truth table block in a Simulink model and the truth table function in a Stateflow chart. In this example, you specify the behavior of a truth table function.

To open the truth table, double-click the truth table function, `ttable`, that you created in “Use Truth Tables to Model Combinatorial Logic” on page 8-2.



By default, a truth table contains a **Condition Table** and an **Action Table**, each with one row. The **Condition Table** contains a single decision column, **D1**, and a single action row.

Select an Action Language

If the truth table is inside a Stateflow chart that uses C as the action language, you can specify the action language for your Stateflow truth table:

1. Open the **Property Inspector**. In the **Modeling** tab, under **Design Data**, select **Property Inspector**.
2. Under the **Properties** section, select **C** or **MATLAB** as the Action Language.

Stateflow charts that use MATLAB as the action language only support truth tables that use MATLAB as the action language.

Enter Truth Table Conditions

Conditions are the starting point for specifying logical behavior in a truth table. Open your new truth table, `ttable`, for editing. You start programming the behavior of `ttable` by specifying conditions.

You enter conditions in the **Condition** column of the **Condition Table**. For each condition that you enter, you can enter an optional description in the **Description** column. To enter conditions for the truth table `ttable`:

1. Click the row on the **Condition Table** that you want to append.
2. Click the **Append Row** button on the side panel twice.

   The truth table appends two rows to the bottom of the **Condition Table**.
3. Click and drag the bar that separates the **Condition Table** and the **Action Table** panes down to enlarge the **Condition Table** pane.
4. In the **Condition Table**, click the top cell of the **Description** column.

   A flashing text cursor appears in the cell, which appears highlighted.
5. Enter this text:

   ```
   x is equal to 1
   ```
6. Click the next cell on the right, in the **Condition** column..
7. In the first cell of the **Condition** column, enter:

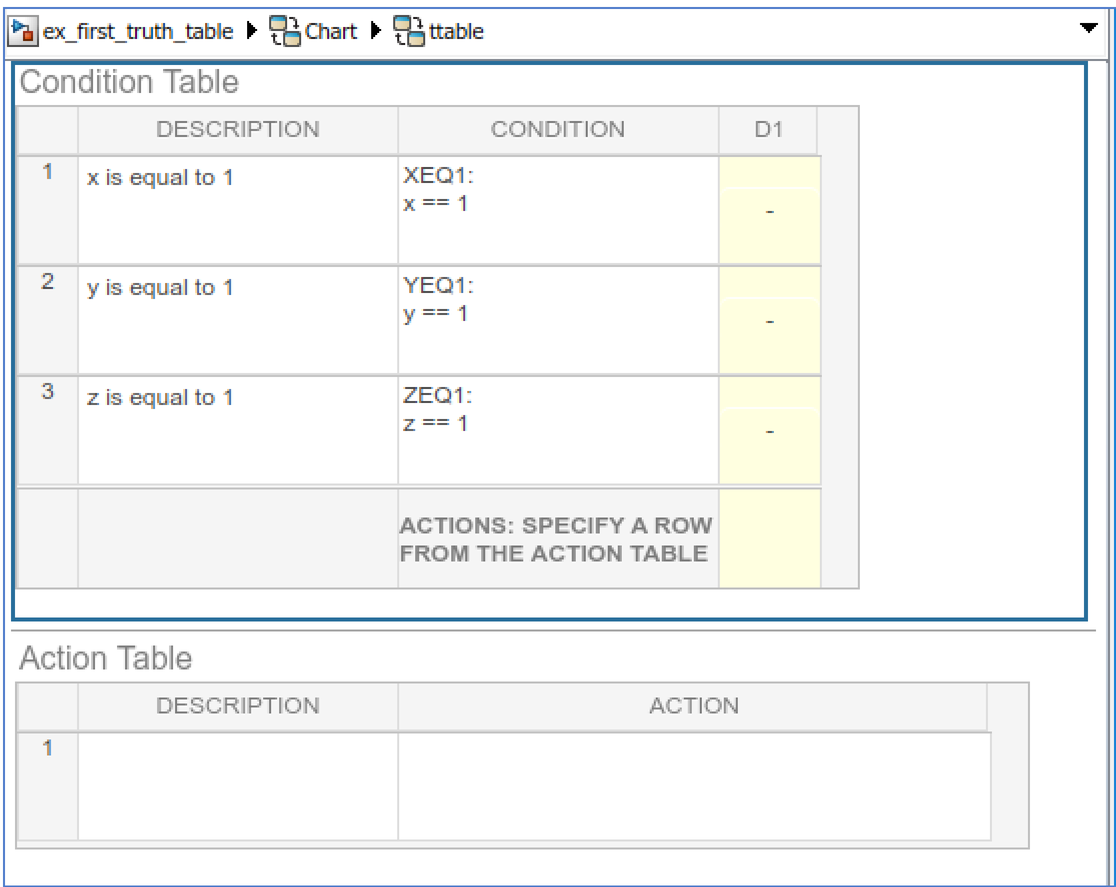
   ```
   XEQ1:
   ```

   This text is an optional label that you can include with the condition. Each label must begin with an alphabetic character (`[a-z][A-Z]`) followed by any number of alphanumeric characters (`[a-z][A-Z][0-9]`) or an underscore (_).
8. Press **Enter** and this text:

   ```
   x == 1
   ```

   This text is the actual condition. Each condition that you enter must evaluate to zero (false) or nonzero (true). You can use optional brackets in the condition (for example, `[x == 1]`).

   In truth table conditions, you can use data that passes to the truth table function through its arguments. The preceding condition tests whether the argument `x` is equal to 1. You can also use data defined for parent objects of the truth table, including the chart.
9. Repeat the preceding steps to enter the other two conditions.

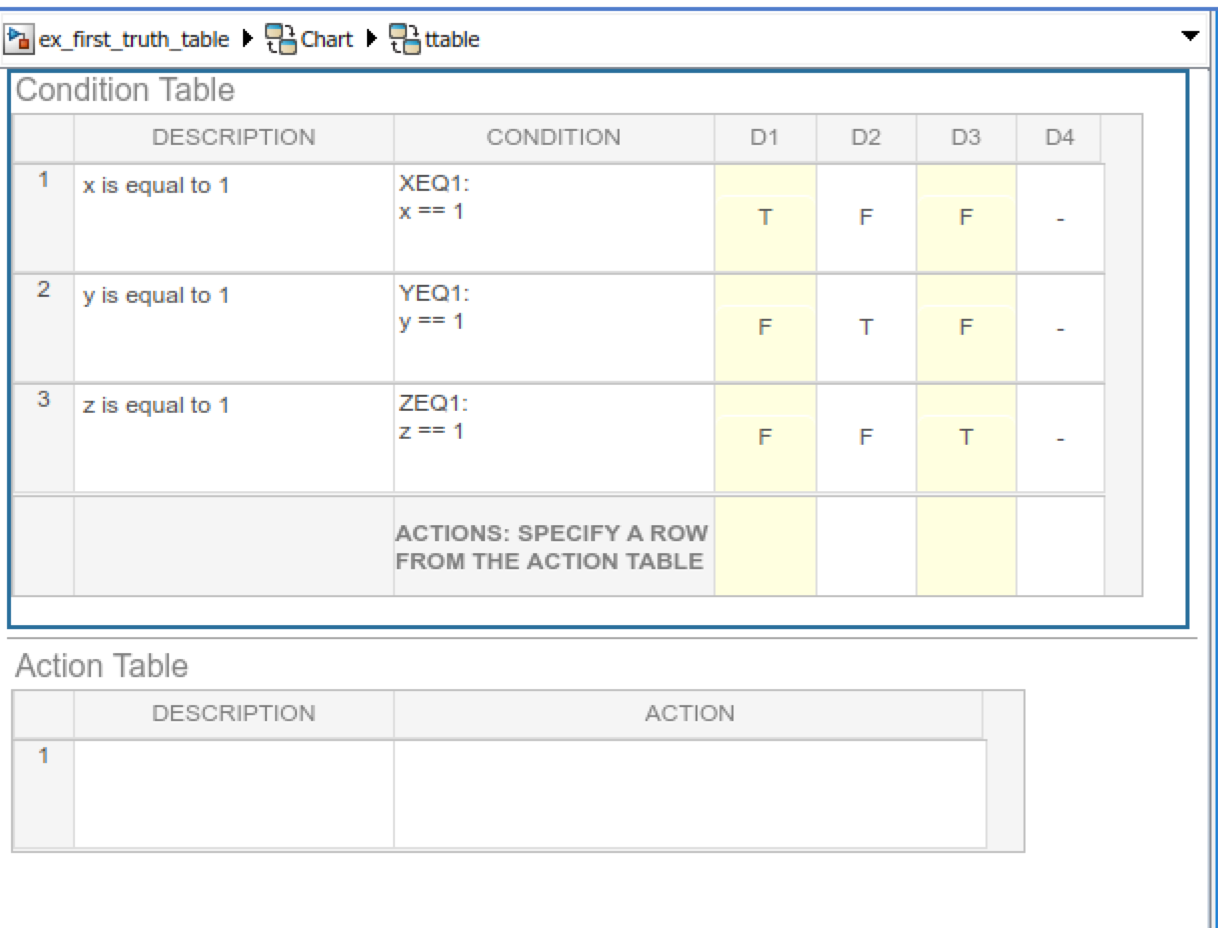


Enter Truth Table Decisions

Each decision column (**D1**, **D2**, and so on) binds a group of condition outcomes together with an AND relationship into a decision. The possible values for condition outcomes in a decision are `T` (true), `F` (false), and `-` (true or false). In "Enter Truth Table Conditions" on page 8-9, you entered conditions for the truth table `ttable`. Continue by entering values in the decision columns:

1. Click the decision column of the **Condition Table** that you want to append.
2. Click the **Append Column** button on the side panel three times.
3. Click the top cell in decision column **D1**.
4. Press the space bar until a value of `T` appears.

   Pressing the space bar toggles through the possible values of `F`, `T`, and `-`. You can also enter these characters directly. Pressing 1 sets the value to `T`, while pressing 0 sets the value to `F`. Pressing `x` sets the value to `-`.
5. Enter the remaining values for the decision columns:



During execution of the truth table, decision testing occurs in left-to-right order. The order of testing for individual condition outcomes within a decision is undefined. Truth tables evaluate the conditions for each decision in top-down order (first condition 1, then condition 2, and so on). Because this implementation is subject to change in the future, do not rely on a specific evaluation order.

The Default Decision Column

The last decision column in `ttable`, **D4**, is the default decision for this truth table. The default decision covers any decisions not tested for in preceding decision columns. Create a default decision by entering - in every cell of the farthest right decision column. This entry represents any outcome for the condition, `T` or `F`. The default decision column must be the last column on the right in the **Condition Table**.

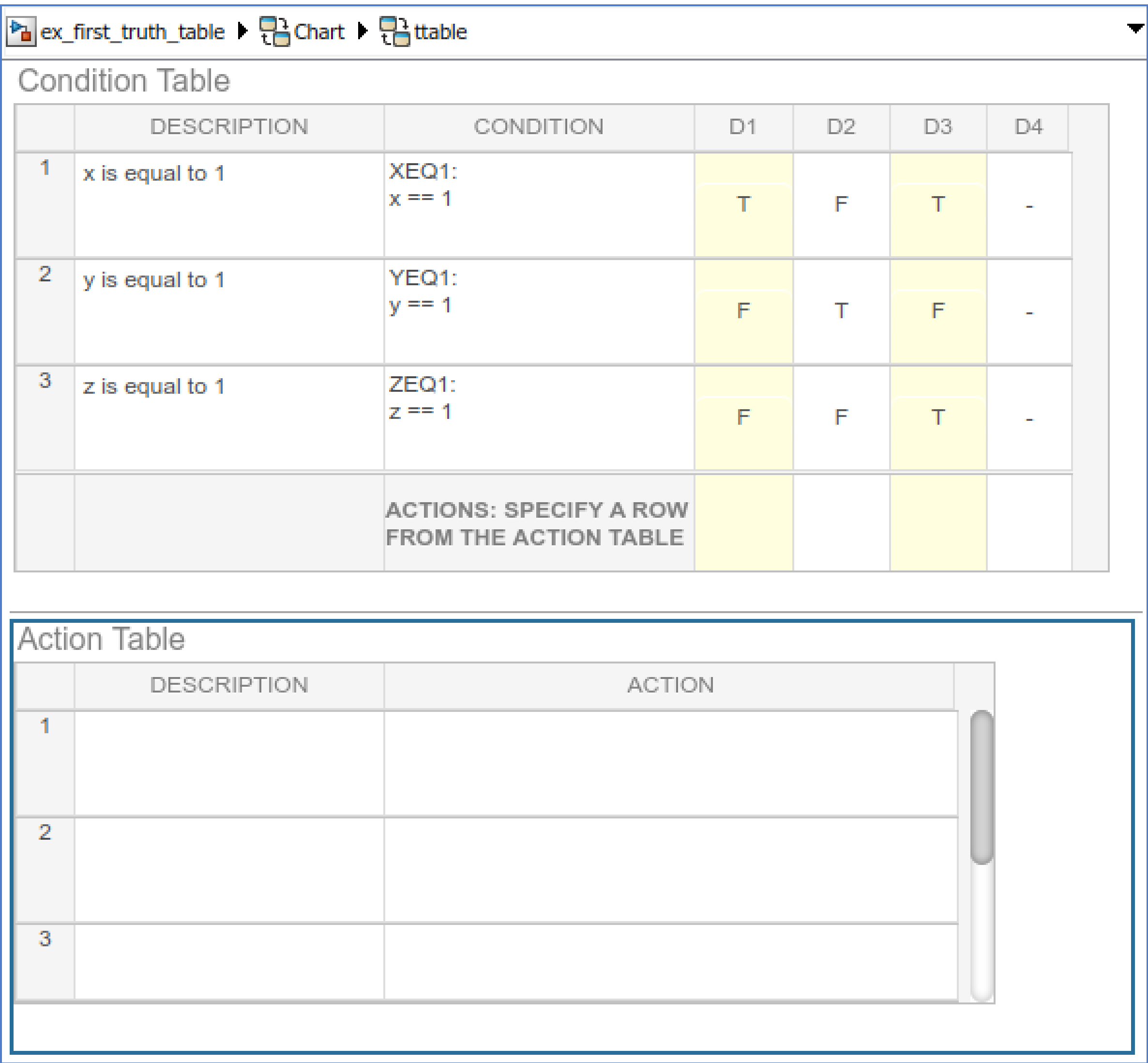
Enter Truth Table Actions

During execution of the truth table, decision testing occurs in left-to-right order. When a decision match occurs, the action in the **Action Table** that is specified in the **Actions** row for that decision column executes. Then the truth table exits.

In “Enter Truth Table Decisions” on page 8-10, you entered decisions in the truth table. The next step is to enter actions you want to occur for each decision in the **Action Table**. Later, you assign these actions to their decisions in the **Actions** row of the **Condition Table**.

Set Up the Action Table

1. Click the row **Action Table** that you want to append.
2. Click the **Append Row** button on the side panel three times.



3. Program the actions for the truth table.

Program Actions of a Truth Table

For truth tables that use MATLAB as the action language, you can write MATLAB code to program your actions. Using this code, you can add control flow logic and call MATLAB functions directly. In the following procedure, you program an action in the truth table `ttable`, using the following features of MATLAB syntax:

- Persistent variables
- `if ... else ... end` control flows

- `for` loop

Follow these steps:

1. Click the top cell in the **Description** column of the **Action Table**.

   A flashing text cursor appears in the cell, which appears highlighted.
2. Enter this description:

   ```
   Maintain a counter and a circular
   vector of values of length 6.
   Every time this action is called,
   output r takes the next value of
   the vector.
   ```
3. Press the right arrow key to select the next cell on the right, in the **Action** column.
4. Enter the following text:

   ```
   A1:
   ```

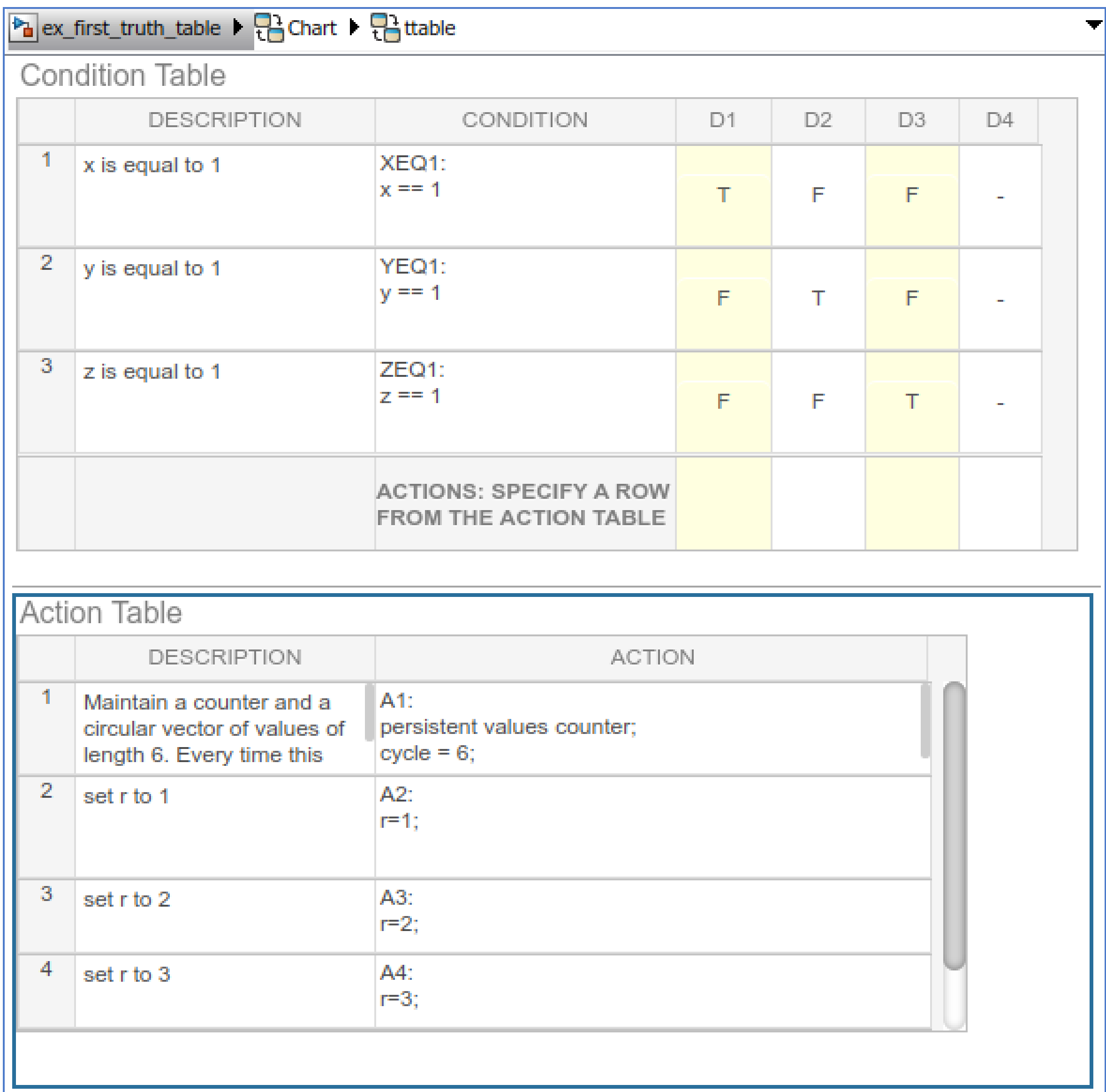
   You begin an action with an optional label followed by a colon (`:`). Later, you enter these labels in the **Actions** row of the **Condition Table** to specify an action for each decision column. Like condition labels, action labels must begin with an alphabetic character (`[a-z][A-Z]`) followed by any number of alphanumeric characters (`[a-z][A-Z][0-9]`) or an underscore (_).
5. Press **Enter** and enter this code:

   ```
   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 r takes the next value in values vector
   r = values( mod(counter, cycle) + 1);
   ```

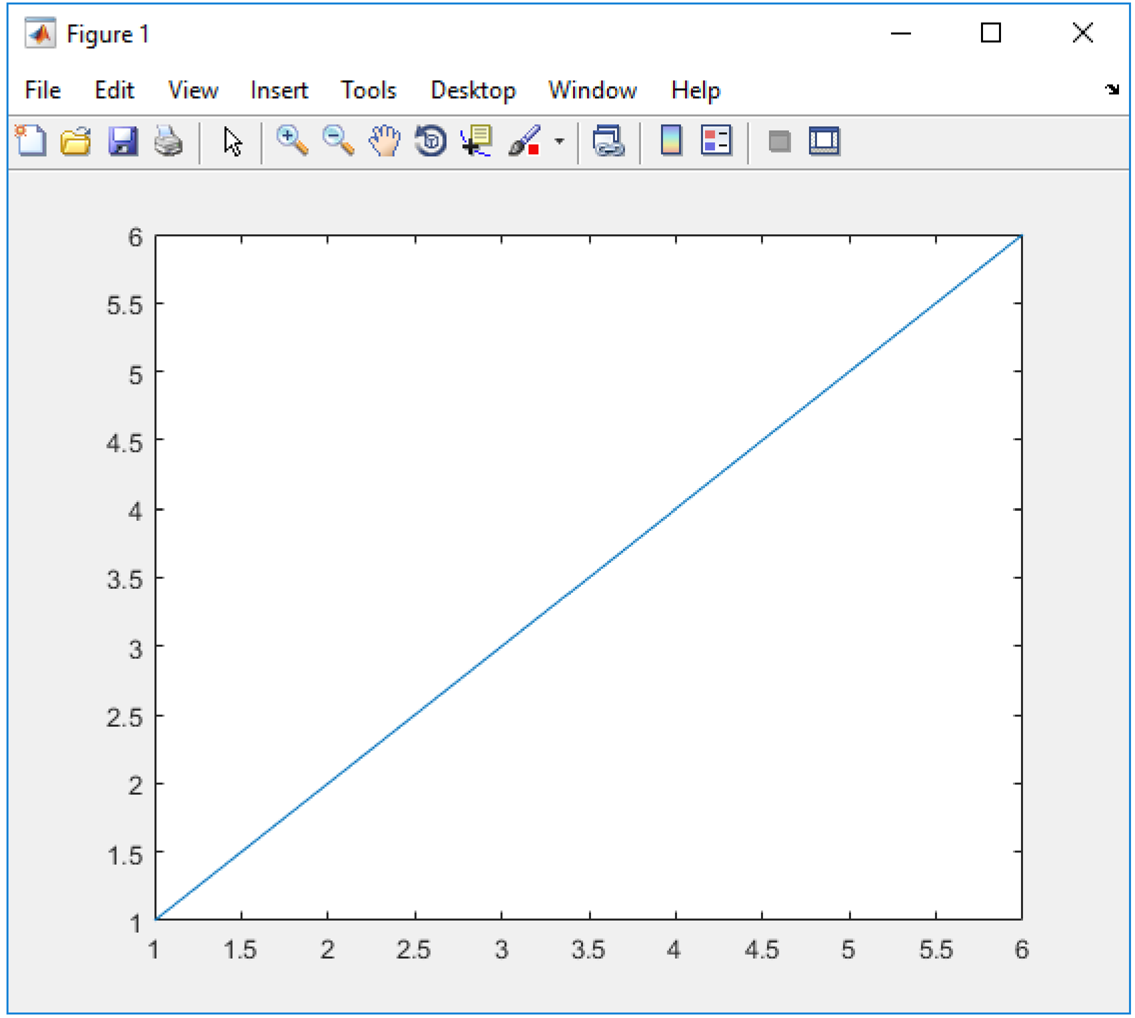
   In truth table actions, you can use data that passes to the truth table function through its arguments and return value. The preceding action sets the return value `r` equal to the next value of the vector `values`. You can also specify actions with data defined for a parent object of the

truth table, including the chart. Truth table actions can also broadcast or send events that are defined for the truth table, or for a parent, such as the chart itself.

6 Enter the remaining actions in the **Action Table**, as shown:



If action `A1` executes during simulation, a plot of the `values` vector appears:



Now you are ready to assign actions to decision.

Assign Truth Table Actions to Decisions

You must assign at least one action from the **Action Table** to each decision in the **Condition Table**. The truth table uses this association to determine what action to execute when a decision tests as true.

Rules for Assigning Actions to Decisions

The following rules apply when you assign actions to decisions in a truth table:

- You specify actions for decisions by entering a row number or a label in the **Actions** row cell of a decision column.

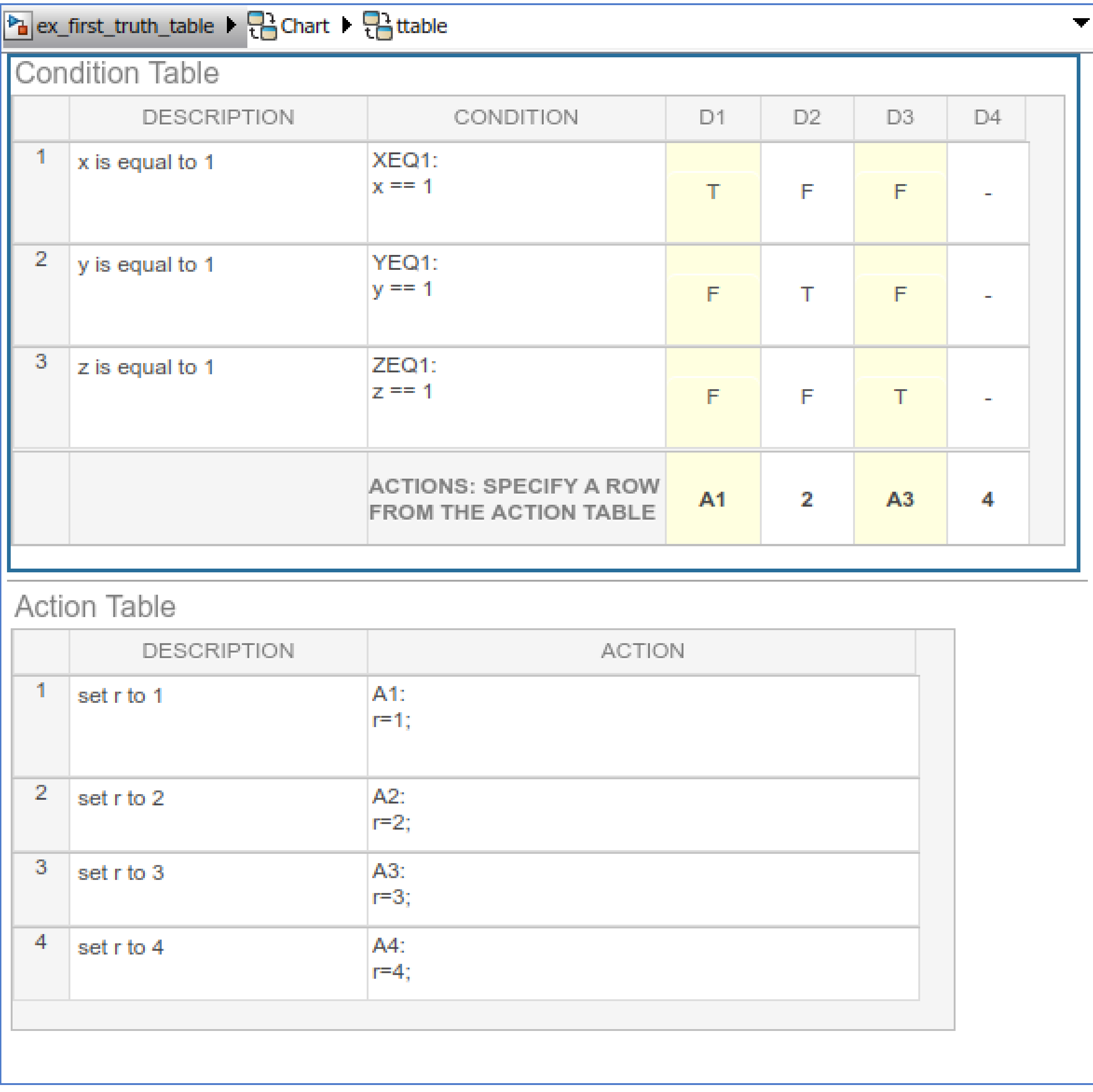
  If you use a label specifier, the label must appear with the action in the **Action Table**.
- You must specify at least one action for each decision.

Actions for decisions are not optional. Each decision must have at least one action specifier that points to an action in the **Action Table**. If you want to specify no action for a decision, specify a row that contains no action statements.

- You can specify multiple actions for a decision with multiple specifiers separated by a comma, semicolon, or space.

  For example, for the decision column **D1**, you can specify `A1,A2,A3` or `1;2;3` to execute the first three actions when decision **D1** is true.
- You can mix row number and label action specifiers interchangeably in any order.

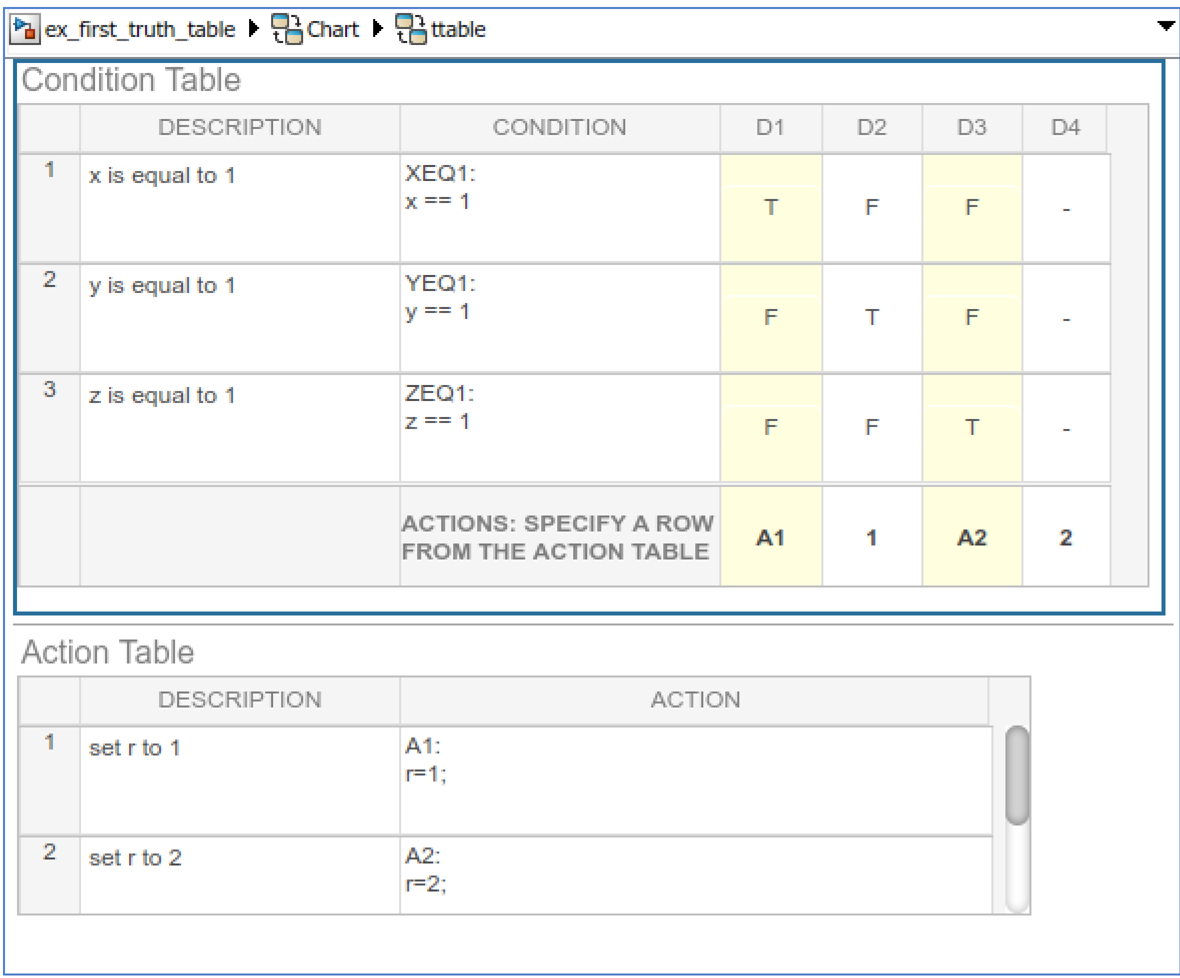
  The following example uses both row and label action specifiers.



	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

	DESCRIPTION	ACTION
1	set r to 1	A1: r=1;
2	set r to 2	A2: r=2;
3	set r to 3	A3: r=3;
4	set r to 4	A4: r=4;

- You can specify the same action for more than one decision, as shown:



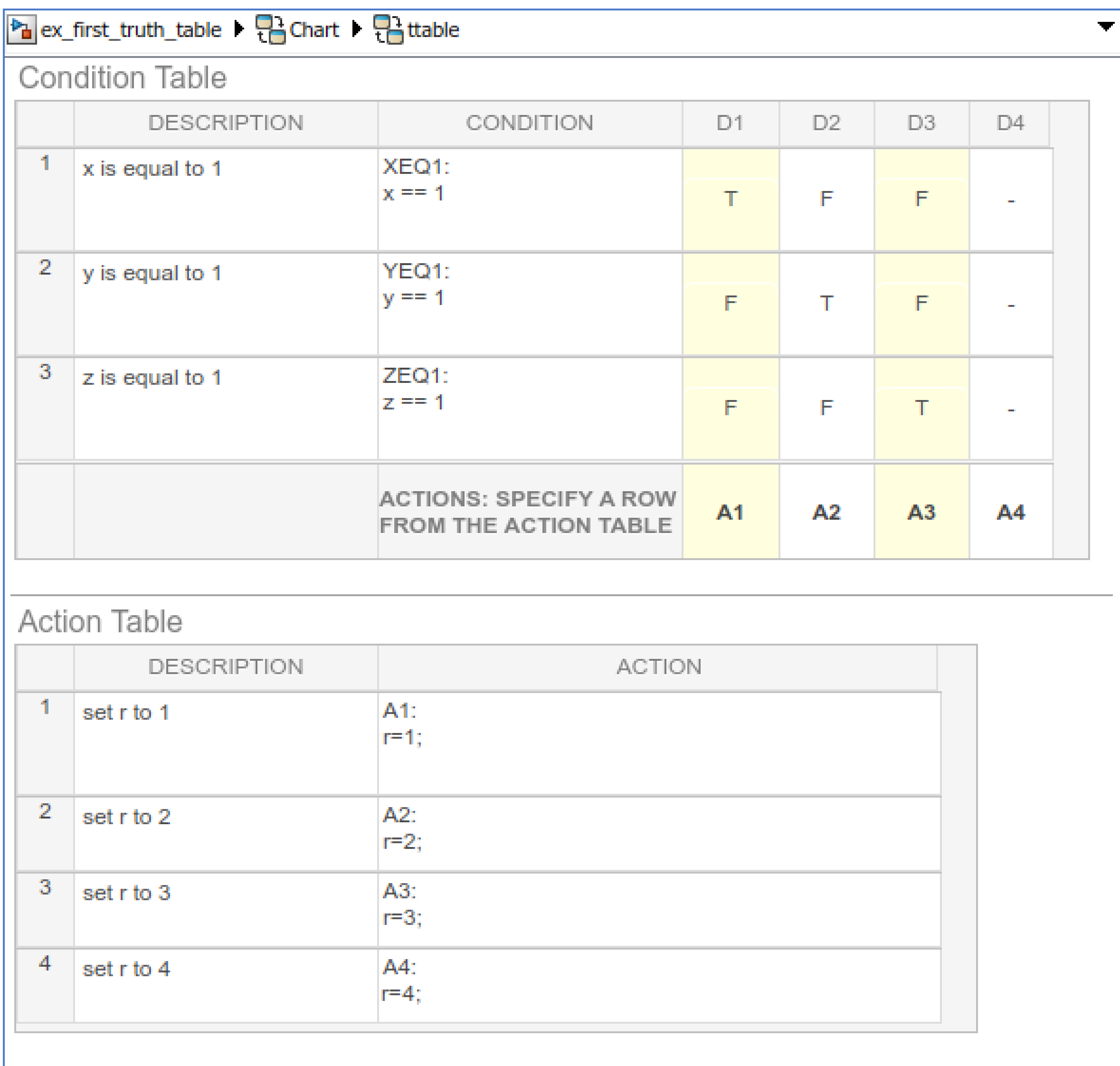
- Row number action specifiers in the **Actions** row of the **Condition Table** automatically adjust to changes in the row order of the **Action Table**.

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:



Now you are ready to perform the final step in programming a truth table.

Add Initial and Final Actions

In addition to actions for decisions, you can add initial and final actions to the truth table function. Initial actions specify an action that executes before any decision testing occurs. Final actions specify an action that executes as the last action before the truth table exits. To specify initial and final actions for a truth table, use the action labels `INIT` and `FINAL` in the **Action Table**.

Use this procedure to add initial and final actions that display diagnostic messages in the MATLAB Command Window before and after execution of the truth table `ttable`:

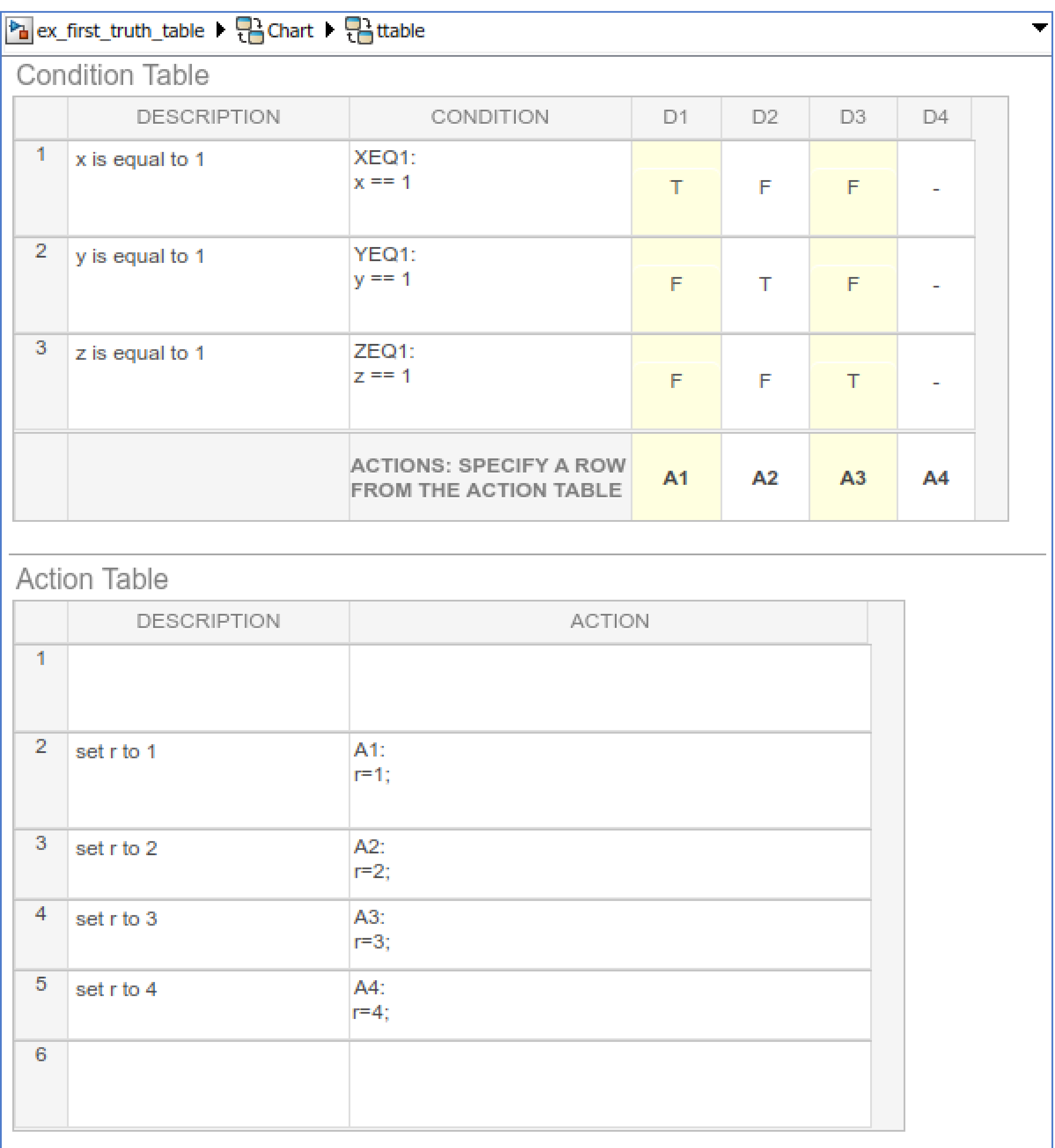
1 In the truth table, right-click row 1 of the **Action Table** and select **Insert Row**.

A blank row appears at the top of the **Action Table**.

2 In the **Modeling** tab, select **Append Row**.

A blank row appears at the bottom of the **Action Table**.

3 Click and drag the bottom border of the truth table to show all six rows of the **Action Table**:



4 Add the initial action in row 1 as follows:

Description	Action

Initial action:
 	
Display message
	`INIT:`
	
	`coder.extrinsic("disp");`
	
	`disp("truth table ttable entered");`

5 Add the final action in row 6 as follows:

Description	Action

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**.

See Also

More About

- "Use Truth Tables to Model Combinatorial Logic" on page 8-2
- "Export Stateflow Functions for Reuse" on page 6-14

Debug Errors in a Truth Table

Once you completely specify your truth tables, you begin the process of debugging them. The first step is to run diagnostics to check truth tables for syntax errors including overspecification and underspecification, as described in “Correct Overspecified and Underspecified Truth Tables” on page 8-28. Additionally, you can add breakpoints directly into your truth table to debug during simulation.

Truth tables are not supported in standalone Stateflow charts in MATLAB. For more information, see “Use Truth Tables to Model Combinatorial Logic” on page 8-2.

Find Syntax Errors by Running Diagnostics

To check for syntax errors:

1 Double-click the truth table.

2
In the truth table, click **Run Diagnostics** .

For example, if you change the action for decision column **D4** to an action that does not exist, you get an error message in the Diagnostic Viewer.

Truth table diagnostics run automatically when you simulate a model with a new or modified truth table. If no errors exist, the diagnostic window does not appear and simulation starts immediately.

Debug Logic by Using Breakpoints

You can use breakpoints in a Stateflow truth table to pause simulation and debug your logic. Once a breakpoint causes the simulation to pause, you can step through the actions and examine the data values at that specific point in the simulation.

With truth tables, you can set these different breakpoint types:

- Condition tested
- Decision tested
- Decision valid
- Action executed

After simulation stops at a breakpoint, you can continue chart execution on the Stateflow Editor toolbar, at the MATLAB command prompt, or by selecting a keyboard shortcut.

Toolbar Icon	Option	Command	Description	Keyboard Shortcut
	Continue	`dbcont`	Continue the simulation to the next breakpoint.	**Ctrl+T**
	Step Over	`dbstep`	Advance to the next step in the truth table execution.	**F10**

Toolbar Icon	Option	Command	Description	Keyboard Shortcut
	Step In	`dbstep in`	From a state or transition action that calls a function, advance to the first executable statement in the function. From a statement in a function containing another function call, advance to the first executable statement in the second function. Otherwise, advance to the next step in the truth table execution. (See Step Over.)	**F11**
	Step Out	`dbstep out`	From a function call, return to the statement calling the function. Otherwise, continue simulation to the next breakpoint. (See Continue.)	**Shift+F11**
	Step Forward		Exit debug mode and pause simulation before next time step.	
	Stop	`dbquit`	Exit debug mode and stop simulation.	**Ctrl+Shift+T**

Condition Breakpoints

To set a breakpoint when a condition is tested, right-click the condition cell and select **Set Breakpoint (Condition Tested)**. A red badge appears on the far left of the table next to the number of the condition. When you run the model, the simulation pauses when the condition is tested. Stateflow highlights the condition row being tested. Place your cursor over the data in the truth table to see its current value.

ex_first_truth_table ▸ Chart ▸ ttable

Condition Table

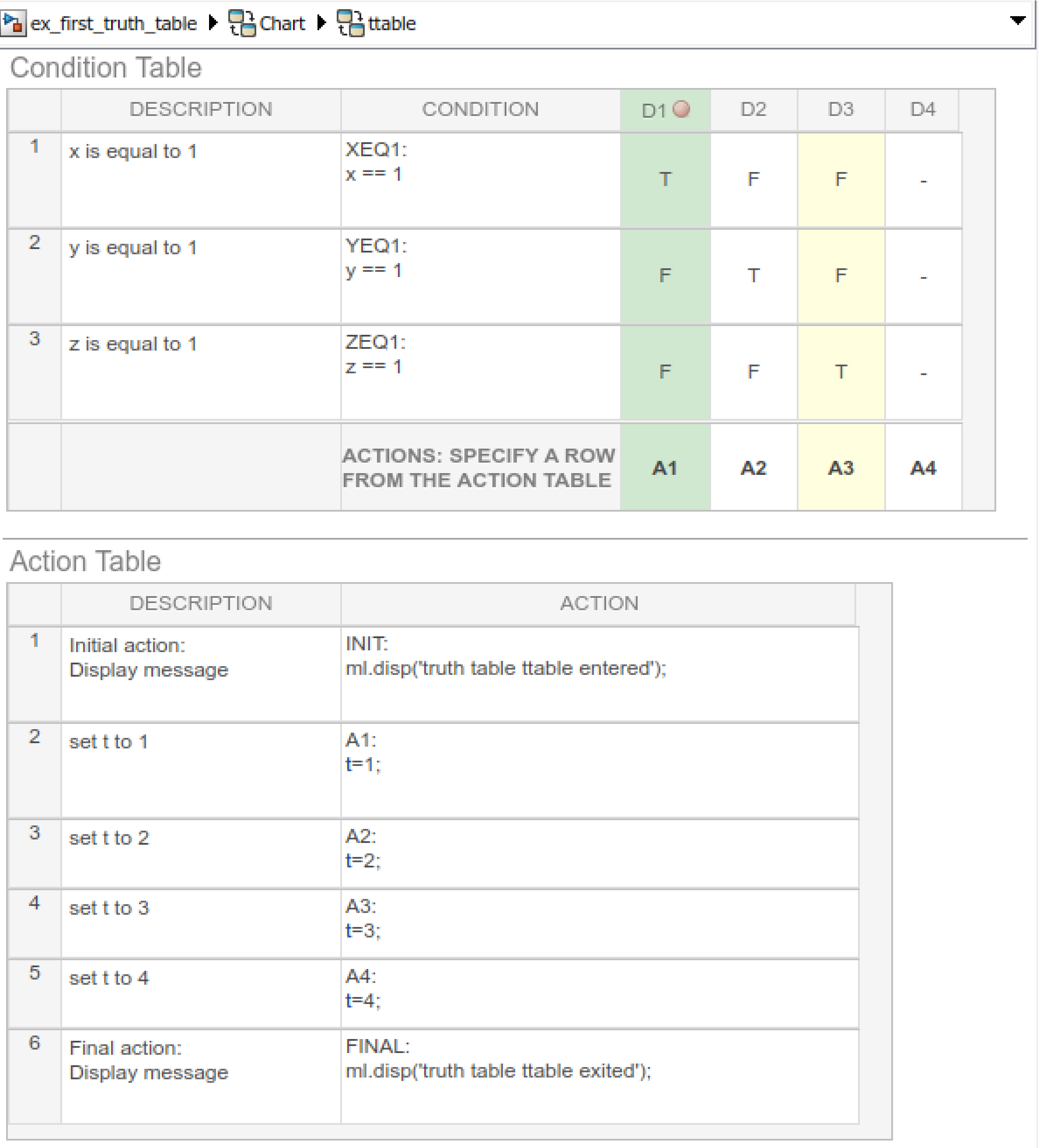
	DESCRIPTION	CONDITION	D1	D2	D3	D4
1	x is equal to 1	XEQ1: x == 1	T	F	F	-
2	y is equal to 1	YEQ1: y == 1	F	T	F	-
3	z is equal to 1	ZEQ1: z == 1	F	F	T	-
		ACTIONS: SPECIFY A ROW FROM THE ACTION TABLE	A1	A2	A3	A4

Action Table

	DESCRIPTION	ACTION
1	Initial action: Display message	INIT: ml.disp('truth table ttable entered');
2	set t to 1	A1: t=1;
3	set t to 2	A2: t=2;
4	set t to 3	A3: t=3;
5	set t to 4	A4: t=4;
6	Final action: Display message	FINAL: ml.disp('truth table ttable exited');

Decision Breakpoints

To set a breakpoint when a decision is tested, right-click the top of the decision column and select **Set Breakpoint (Decision Tested)**. A red badge appears on the top of the decision column next to the number of the decision. When you run the model, the simulation pauses when the decision is tested. Stateflow highlights the decision column being tested. Place your cursor over the data in the truth table to see its current value.



Condition Table

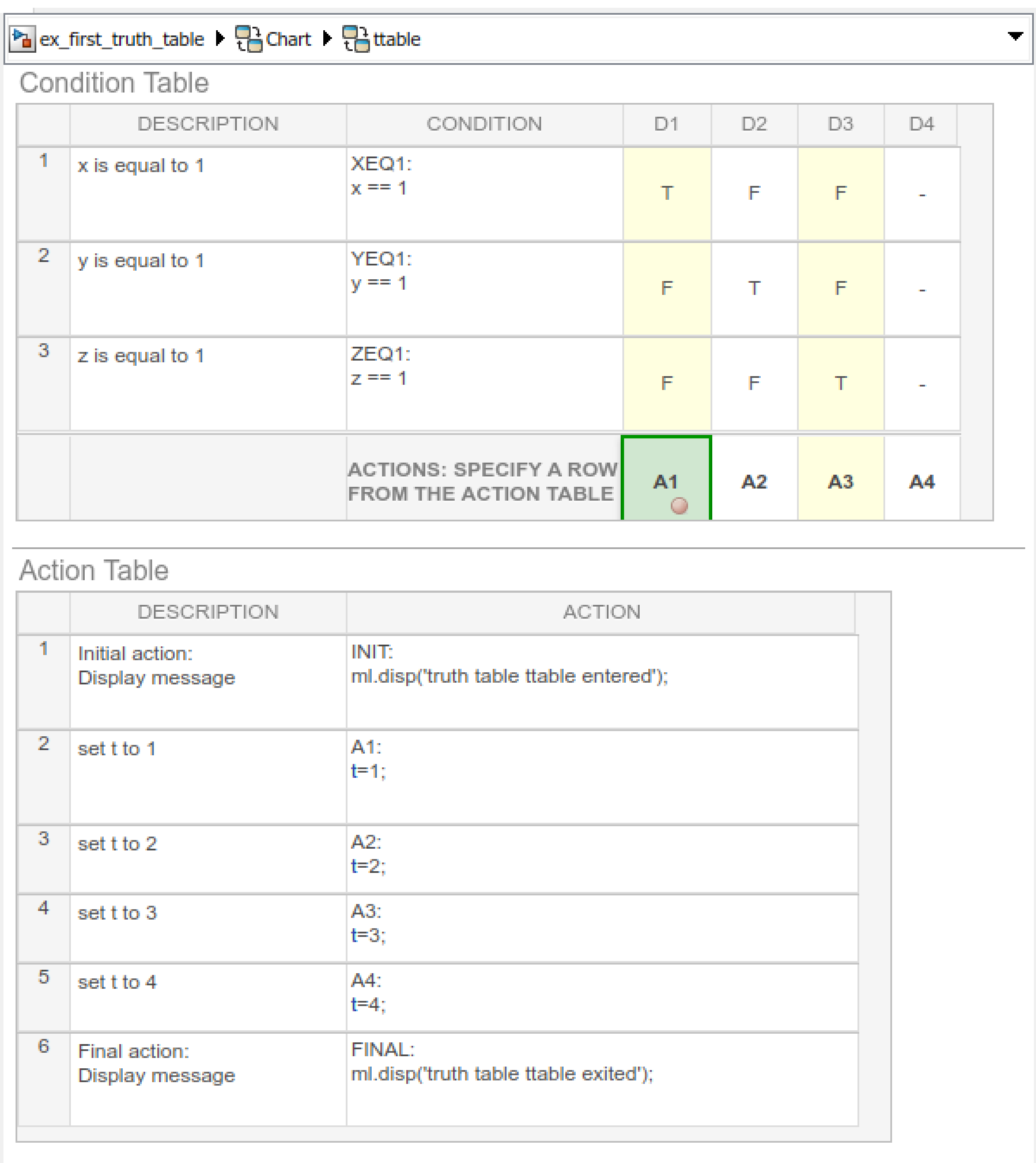
	DESCRIPTION	CONDITION	D1	D2	D3	D4
1	x is equal to 1	XEQ1: x == 1	T	F	F	-
2	y is equal to 1	YEQ1: y == 1	F	T	F	-
3	z is equal to 1	ZEQ1: z == 1	F	F	T	-
		ACTIONS: SPECIFY A ROW FROM THE ACTION TABLE	A1	A2	A3	A4

Action Table

	DESCRIPTION	ACTION
1	Initial action: Display message	INIT: ml.disp('truth table ttable entered');
2	set t to 1	A1: t=1;
3	set t to 2	A2: t=2;
4	set t to 3	A3: t=3;
5	set t to 4	A4: t=4;
6	Final action: Display message	FINAL: ml.disp('truth table ttable exited');

To set a breakpoint when a decision is valid, right-click the action cell at the bottom of the decision column and select **Set Breakpoint (Decision Valid)**. A red badge appears on the top of the cell next to the action number. When you run the model, the simulation pauses when the action is valid. Stateflow highlights the valid decision. Place your cursor over the data in the truth table to see its current value.

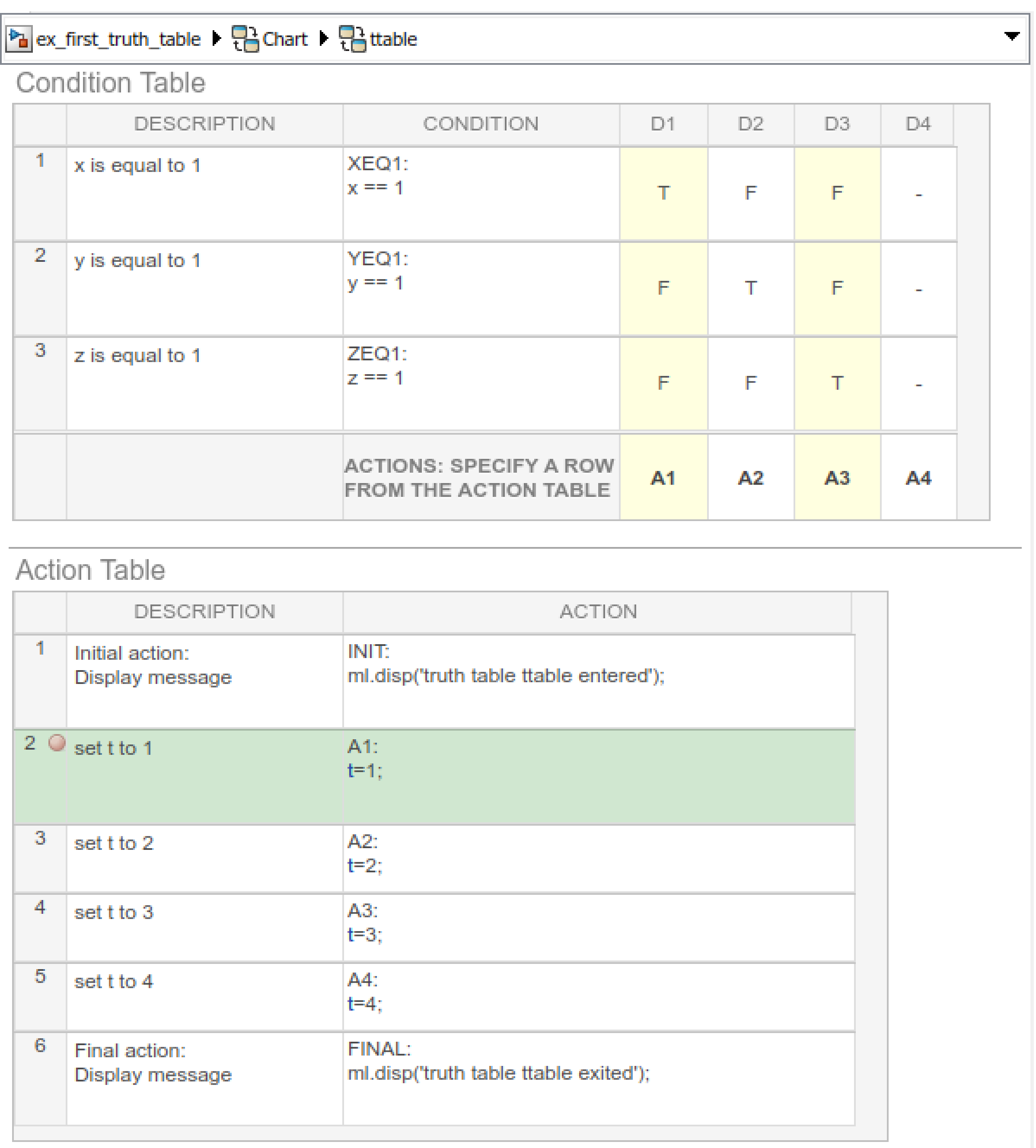
If there is more than one action to take when a decision is valid, the breakpoint is set for the first executable.



Action Breakpoints

To set a breakpoint when an action is executed, right-click the action cell and select **Set Breakpoint (Action Executed)**. A red badge appears on the far left of the table next to the number of the action. When you run the model, the simulation pauses when the action is executed. Stateflow highlights the action row being tested. Place your cursor over the data in the truth table to see its current value.

If there is more than one action within the action cell, the breakpoint is set for the first action.



Edit Breakpoints

Click the breakpoint to open the **Edit Breakpoint** dialog box. From this window you can disable the breakpoint by clearing the **Enable Breakpoint** check box.

When you add a condition to a breakpoint, the breakpoint pauses the simulation only when its associated condition is `true`.

See Also

More About

- “Use Truth Tables to Model Combinatorial Logic” on page 8-2
- “Program a Truth Table” on page 8-8
- “Correct Overspecified and Underspecified Truth Tables” on page 8-28
- “Debug MATLAB Function Blocks” (Simulink)

Correct Overspecified and Underspecified Truth Tables

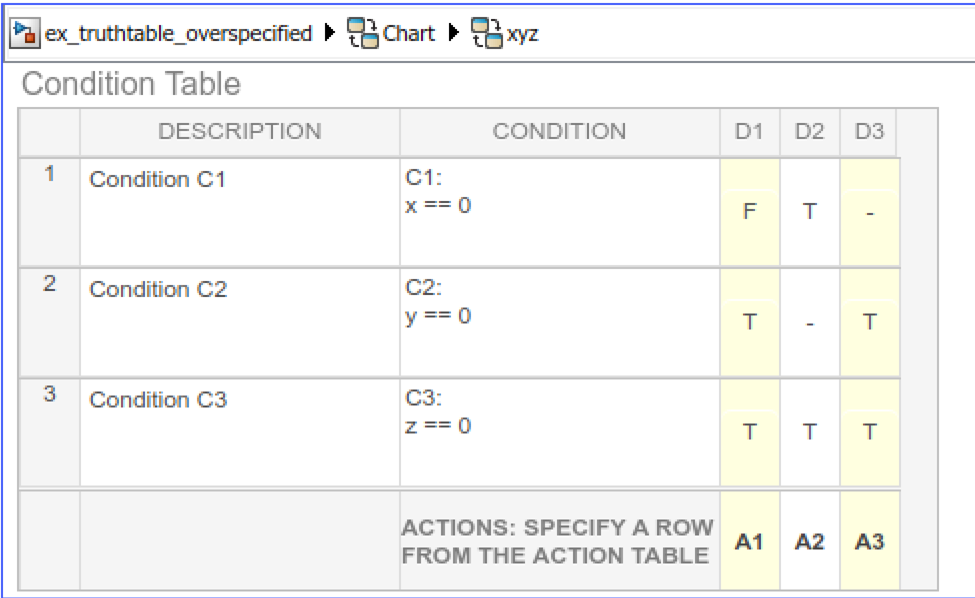
When programming your truth table, you might program an overspecified or underspecified truth table. An overspecified truth table contains at least one true or false combination that is specified by another decision column. When this happens, the action associated with that decision column never executes. An underspecified truth table occurs when your truth table does not have enough decision columns to account for all possible true or false combinations.

By default, Stateflow reports an error for overspecified and underspecified truth tables. To adjust the error settings for truth tables, open your truth table. After opening the truth table, in the **Modeling** tab, click **Table Properties** and change the settings for **Underspecification** or **Overspecification**.

Truth tables are not supported in standalone Stateflow charts in MATLAB. For more information, see “Use Truth Tables to Model Combinatorial Logic” on page 8-2.

Example of an Overspecified Truth Table

An overspecified truth table contains at least one decision that never executes because a previous decision specifies it in the **Condition Table**. The following example shows the **Condition Table** of an overspecified truth table.

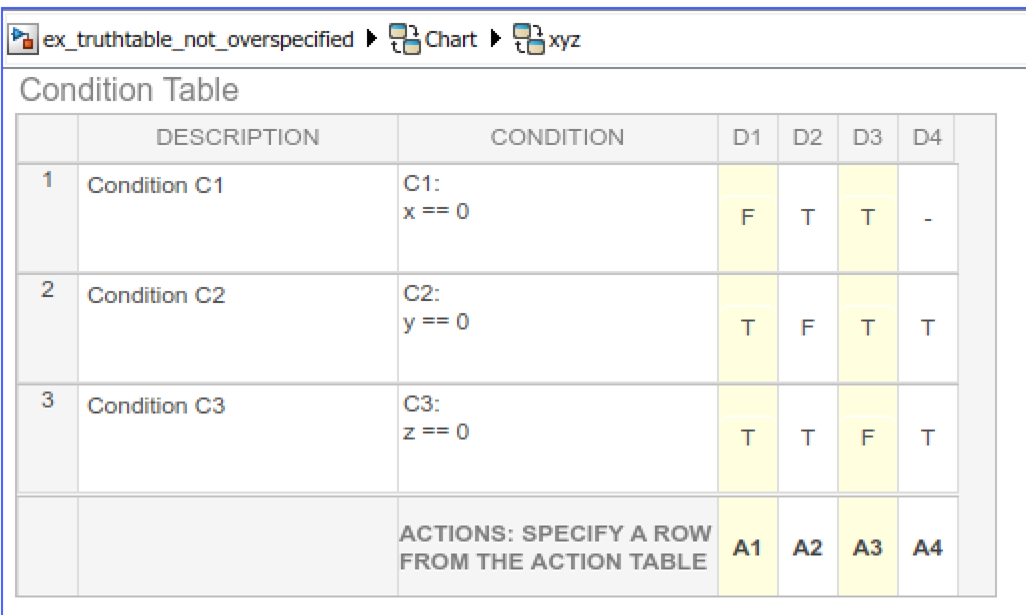

ex_truthtable_overspecified ▶ Chart ▶ xyz

Condition Table

	DESCRIPTION	CONDITION	D1	D2	D3
1	Condition C1	C1: x == 0	F	T	-
2	Condition C2	C2: y == 0	T	-	T
3	Condition C3	C3: z == 0	T	T	T
		ACTIONS: SPECIFY A ROW FROM THE ACTION TABLE	A1	A2	A3

The decision in column **D3** (-TT) specifies the decisions FTT and TTT. These decisions are duplicates of decisions **D1** (FTT) and **D2** (TTT and TFT). Therefore, column **D3** is an overspecification.

The following example shows the **Condition Table** of a truth table that appears to be overspecified, but is not.


ex_truthtable_not_overspecified ▶ Chart ▶ xyz

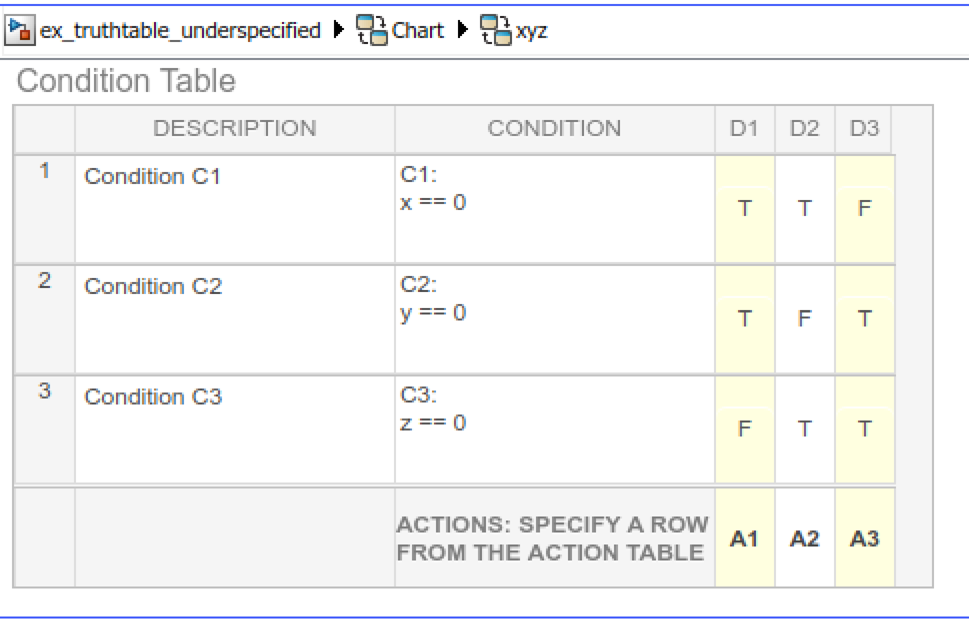
Condition Table

	DESCRIPTION	CONDITION	D1	D2	D3	D4
1	Condition C1	C1: x == 0	F	T	T	-
2	Condition C2	C2: y == 0	T	F	T	T
3	Condition C3	C3: z == 0	T	T	F	T
		ACTIONS: SPECIFY A ROW FROM THE ACTION TABLE	A1	A2	A3	A4

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

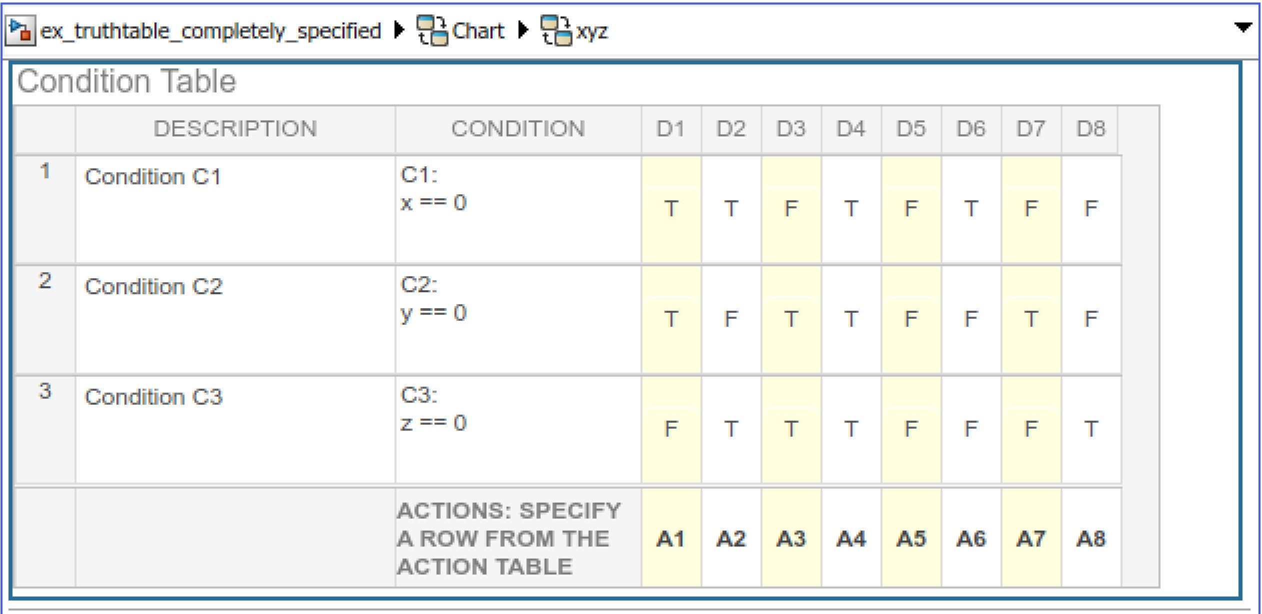
An underspecified truth table includes undefined behavior because it lacks decisions that cover every combination of the specified conditions.. The following example shows the **Condition Table** of an underspecified truth table.


ex_truthtable_underspecified ▶ Chart ▶ xyz

Condition Table

	DESCRIPTION	CONDITION	D1	D2	D3
1	Condition C1	C1: x == 0	T	T	F
2	Condition C2	C2: y == 0	T	F	T
3	Condition C3	C3: z == 0	F	T	T
		ACTIONS: SPECIFY A ROW FROM THE ACTION TABLE	A1	A2	A3

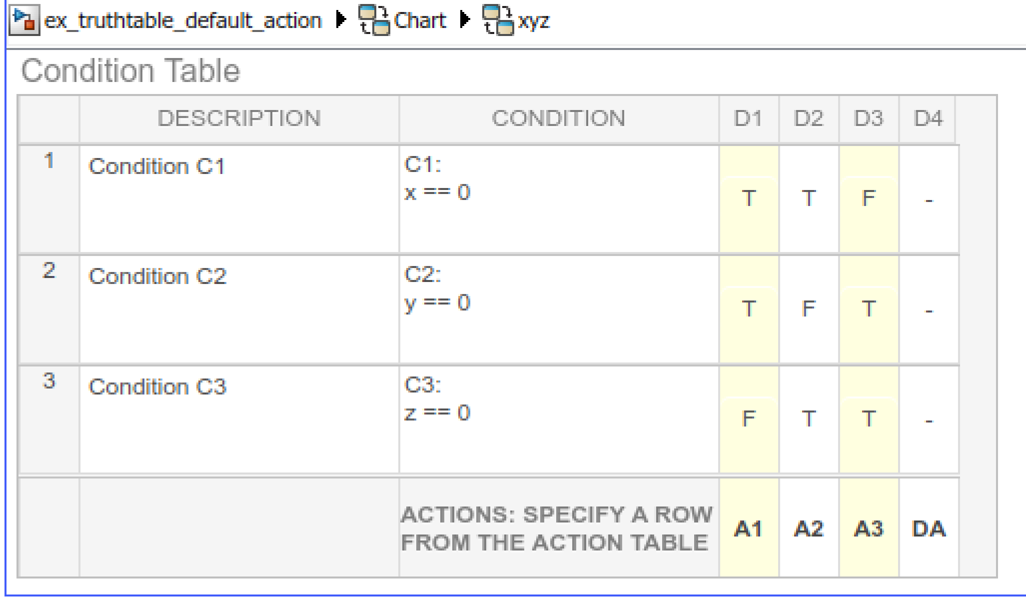
Complete coverage of the conditions in the preceding truth table requires a **Condition Table** with every possible decision:


ex_truthtable_completely_specified ▸ Chart ▸ xyz

Condition Table

	DESCRIPTION	CONDITION	D1	D2	D3	D4	D5	D6	D7	D8
1	Condition C1	C1: x == 0	T	T	F	T	F	T	F	F
2	Condition C2	C2: y == 0	T	F	T	T	F	F	T	F
3	Condition C3	C3: z == 0	F	T	T	T	F	F	F	T
		ACTIONS: SPECIFY A ROW FROM THE ACTION TABLE	A1	A2	A3	A4	A5	A6	A7	A8

To avoid underspecification specify an action for all other possible decisions through a default decision, named DA:


ex_truthtable_default_action ▸ Chart ▸ xyz

Condition Table

	DESCRIPTION	CONDITION	D1	D2	D3	D4
1	Condition C1	C1: x == 0	T	T	F	-
2	Condition C2	C2: y == 0	T	F	T	-
3	Condition C3	C3: z == 0	F	T	T	-
		ACTIONS: SPECIFY A ROW FROM THE ACTION TABLE	A1	A2	A3	DA

The last decision column, **D4**, 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 8-11 for an example and more complete description of the default decision column for a **Condition Table**.

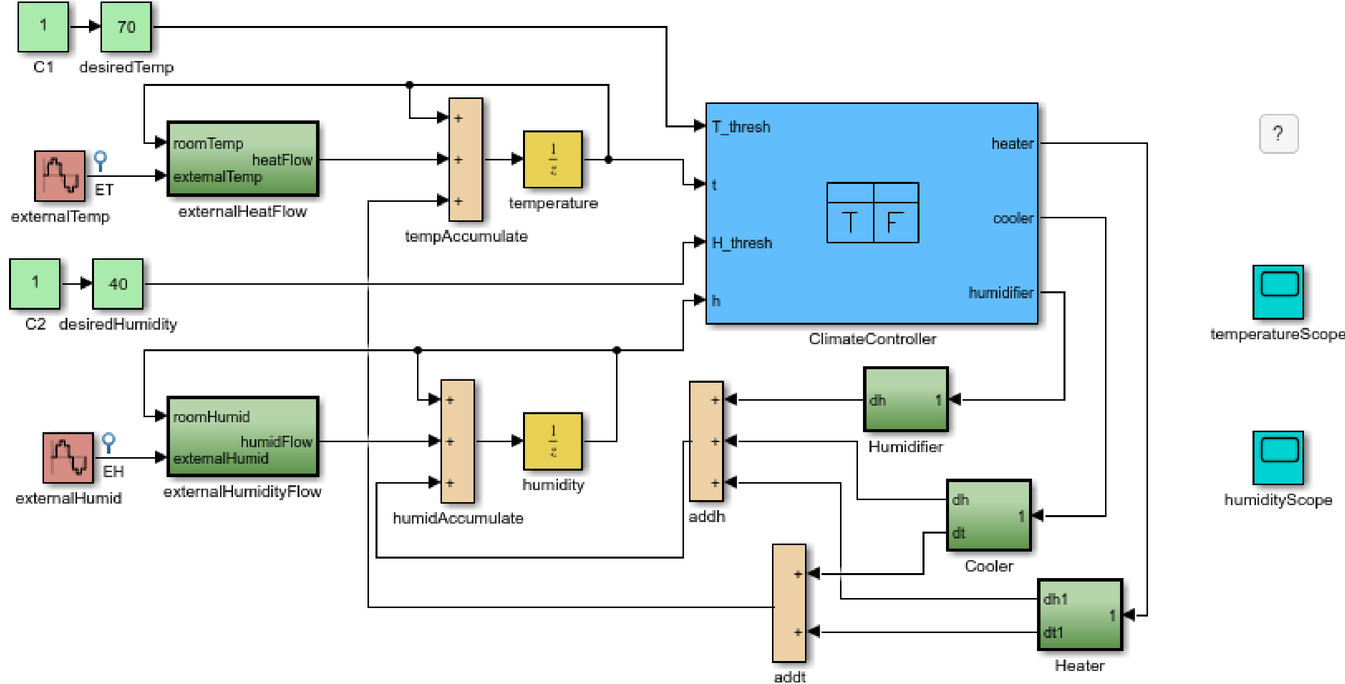
See Also

More About

- "Use Truth Tables to Model Combinatorial Logic" on page 8-2
- "Program a Truth Table" on page 8-8
- "Debug Errors in a Truth Table" on page 8-21

Home Climate Control Using the Truth Table Block

This example models a home climate control system by using a Truth Table block. Homes rarely maintain a constant climate without a climate control system in place, and occupants usually rely on automated systems to maintain a desired climate. Because temperature and humidity are dynamic, maintaining desired conditions requires consistent monitoring and adjusting. To model how a home activates different subsystems that maintain a desired climate, this model uses a Truth Table block to manage logical decision making.



Copyright 2005-2018 The MathWorks, Inc.

Examine the Truth Table

In this example, the Truth Table block labeled `ClimateController` controls all of the physical subsystem outputs. The block uses four inputs: the desired temperature `T_thresh`, the actual home temperature `t`, the desired humidity `H_thresh`, and the actual home humidity `h`. Double-click the block to see how the block uses the inputs to produce the outputs. The `ClimateController` block includes two tables: the Condition Table and the Action Table.

The Condition Table shows how the inputs are logically evaluated, and illustrates the two comparisons made by the block and the four actions that can be taken. To execute the first action, the two conditions must be `True`. If either condition is not `True`, the block tests the conditions outlined in the next decision column, which requires only the first condition to be `True`. This evaluation continues from left to right until a decision is made or the last decision column is reached, which then executes. In this example, the `-` entries function like `False` conditions. As a result, the block would behave the same way if the `-` conditions were explicitly defined as `False`. However, automatically generated code using only `True` and `False` conditions may produce suboptimal code coverage. To avoid that issue, this example uses `-` conditions.

In the first row, the block compares the home temperature to the desired temperature, and the home cooler and heater are controlled using the `CoolOn` and `HeatOn` actions, respectively. When `t > T_thresh`, the block activates the `CoolOn` action. If this condition is not `True`, the block activates the `HeatOn` action. In the second row, the block compares the home humidity to the desired humidity, and the humidifier is controlled using the `HumidOn` action. When `h < H_thresh`, the block activates the `HumidOn` action.

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

The Action Table defines the block outputs associated with each logical action. In the first row, `CoolOn` sets the value of `cooler` to `1` and the value of `heater` to `0`. In the second row, `HeatOn` sets the value of `heater` to `1` and the value of `cooler` to `0`. By default, the humidifier is `0` unless the block enables `HumidOn`.

Action Table

	DESCRIPTION	ACTION

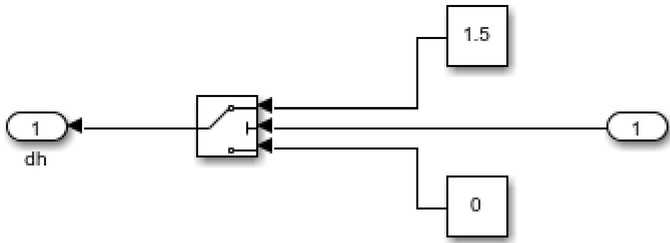
1	Turn On Cooling (This implicitly reduces humidity)	CoolOn:
		cooler = 1;
		heater = 0;
		humidifier = 0;

2	Turn On Heater (This implicitly reduces humidity)	HeatOn:
		heater = 1;
		cooler = 0;
		humidifier = 0;

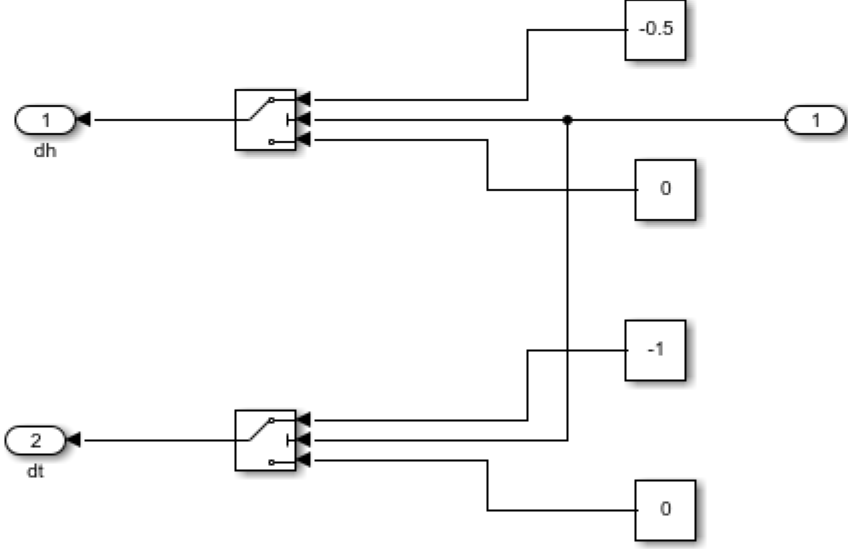
3	Turn On Humidifier	HumidOn: humidifier = 1;

Examine the Home Subsystems

In the model, the green blocks labeled `Humidifier`, `Cooler`, and `Heater` represent the physical subsystems that regulate the climate of the home. The `Humidifier` subsystem includes a Switch block that engages from the output of the `ClimateController` block. If the input to the `Humidifier` subsystem is `1`, the subsystem outputs `1.5`. Otherwise, the subsystem outputs a value of `0`.



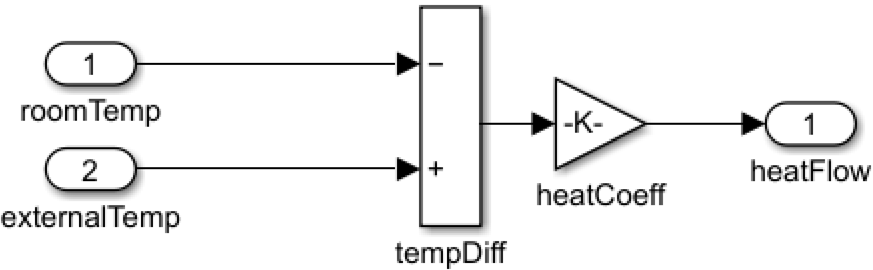
The `Heater` and `Cooler` subsystems work on similar principles. They each include two Switch blocks. One Switch block outputs a value that affects the temperature, which is the output at the **dt** and **dt1** ports for the `Cooler` and `Heater`, respectively. The other Switch block outputs a value that affects the humidity, which is output at the **dh** and **dh1** ports for the `Cooler` and `Heater`, respectively. If `cooler = 1`, the `Cooler` subsystem activates, and if `heater = 1`, the `Heater` subsystem activates. When engaged, the `Heater` subsystem outputs `1` at **dt1**, and the `Cooler` outputs `-1` at **dt**. Both subsystems output `-0.5` at **dh** and **dh1** when engaged.



Due to how the `ClimateController` block is configured, the `Cooler` and `Heater` subsystems will not activate at the same time.

Examine the External Subsystems

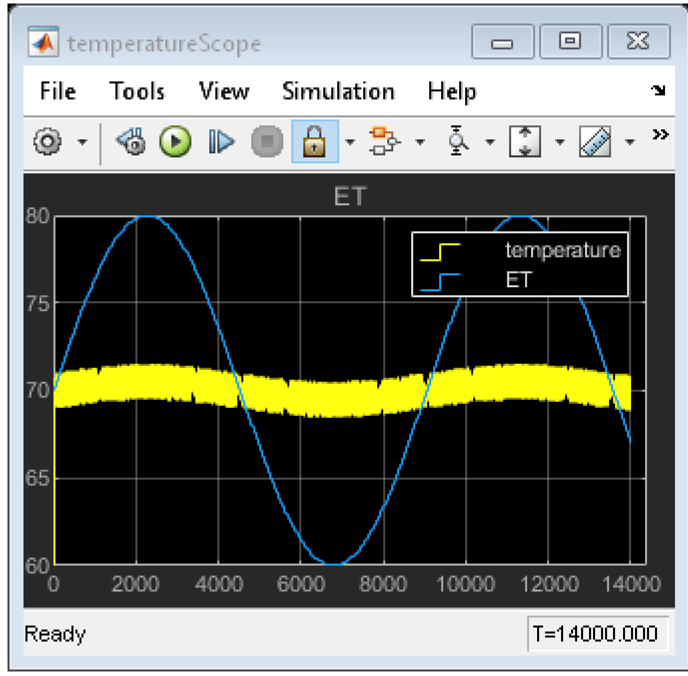
External heat and humidity also affect the climate of the home. The model represents the effect of these conditions as heat and humidity flow. The `externalHeatFlow` subsystem models external heat flow, and the `externalHumidityFlow` subsystem models external humidity flow. The `externalHeatFlow` subsystem takes the difference between the external and internal temperatures and multiplies the difference by a coefficient.



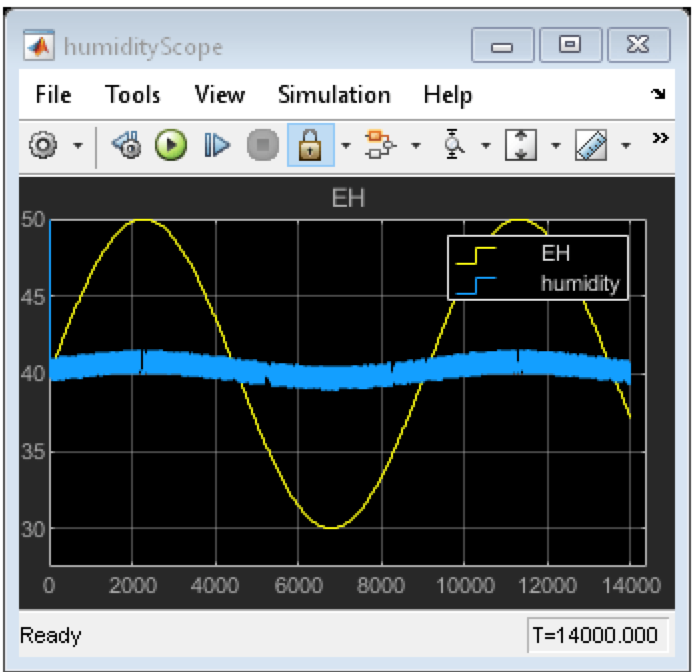
Higher values of the coefficient represent larger heat flows, which occur in less insulated homes. Although the `externalHumidityFlow` subsystem represents a different physical behavior than `externalHeatFlow`, the `externalHumidityFlow` subsystem uses the same arrangement of blocks and connections. The `externalHumidityFlow` subsystem takes the difference between the external and internal humidities and multiplies the difference by a coefficient.

Simulate the Model

Running the model populates the two Floating Scope blocks. The Scope block labeled `temperatureScope` displays the external temperature (`ET`) and the home temperature (`temperature`).



The Scope block labeled `humidityScope` plots the external humidity (`EH`) and the home humidity (`humidity`).



The simulation is configured to run indefinitely. To stop the simulation, you can stop it manually by pressing the **Stop** button or by adjusting the stop time before running the simulation.

Explore and Modify the Model

You can adjust the external temperature by using a different external temperature signal or by modifying the signal amplitude. Try adjusting the amplitude of the Sine Wave blocks, `externalTemp` and `externalHumid`, and observe how the model responds.

Other homes may not be as insulated or might have more effective climate control subsystems. These physical differences affect the outputs of the subsystems. Try adjusting the `Heater` or `Cooler` subsystem outputs by changing the Constant block values.

See Also

Truth Table

More About

- "Use Truth Tables to Model Combinatorial Logic" on page 8-2

Simulink Functions in Stateflow Charts

- “Reuse Simulink Functions in Stateflow Charts” on page 9-2
- “Bind a Simulink Function to a State” on page 9-10
- “Design Charts with Simulink Functions” on page 9-15
- “Schedule Execution of Multiple Controllers” on page 9-20
- “Schedule Simulink Functions by Using Stateflow” on page 9-26
- “Design Switching Controllers by Using Simulink Functions” on page 9-28
- “Manage Queue for Shared Printer Server” on page 9-32

Reuse Simulink Functions in Stateflow Charts

A Simulink function is a graphical object that enables you to call a Simulink subsystem in the actions of states and transitions. Simulink functions are not supported in standalone Stateflow charts in MATLAB.

Simulink functions can improve the efficiency of your design and increase the readability of your model. Typical applications include:

- Defining a function that requires Simulink blocks
- Scheduling execution of multiple controllers

Simulink functions in a Stateflow chart provide these advantages:

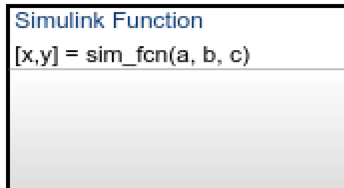
- No function-call subsystem blocks
- No output events
- No signal lines

A Simulink function can reside anywhere in a chart, state, or subchart. The location of a function determines what states and transitions are able to call the function.

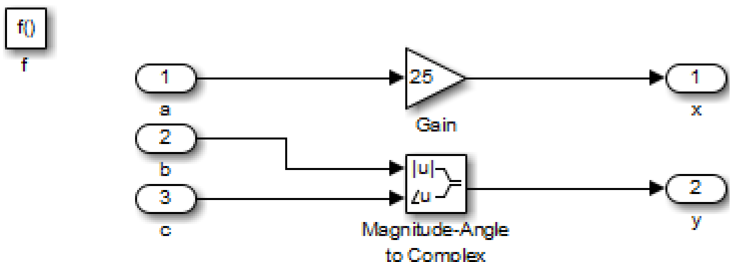
- If you want to call the function within only one state or subchart and its substates, put your Simulink function in that state or subchart. That function overrides any other functions of the same name in the parents of that state or subchart.
- If you want to call the function from anywhere in a chart, put your Simulink function at the chart level.
- If you want to call the function from any chart in your model, use a Simulink Function block to define the function directly in the Simulink canvas. For more information, see "Simulink Functions Overview" (Simulink).

To access Stateflow data from your Simulink function, you must include that data as an input to your Simulink function.

This Simulink function has the name `sim_fcn`. It takes three arguments (`a`, `b`, and `c`) and returns two output values (`x` and `y`).



The function contains a Simulink subsystem that multiplies the first argument times a gain of 25 and combines the other two arguments into a complex output signal.



Once you have your Simulink function defined, you can place it anywhere in your Stateflow chart or Simulink model. Additionally, you can reuse this function as many times as needed within the same or different models.

Define a Simulink Function

1. In the object palette, click the Simulink function icon .
2. On the chart canvas, click the location for the new Simulink function.
3. Enter the signature label for the function.

   The signature label of the function specifies a name for your function and the formal names for its arguments and return values. A signature label has this syntax:

   ```
   [return_val1,return_val2,...] = function_name(arg1,arg2,...)
   ```

   You can specify multiple return values and multiple input arguments. Each return value and input argument can be a scalar, vector, or matrix of values. For functions with only one return value, omit the brackets in the signature label.

   You must use unique variable names for all arguments and return values.
4. To program the function, open the Simulink Editor by double-clicking the function box. Initially, the editor contains a function-call Trigger block and Inport and Outport blocks that match the function signature. You cannot delete the Trigger block.
5. In the Simulink Editor, add blocks to create your Simulink subsystem and connect them to the Inport and Outport blocks.
6. Configure the Inport and Outport blocks.

   a. Double-click each block to open the Block Parameters dialog box.

   b. In the **Signal Attributes** tab, enter the **Data type** and **Port dimensions** of the input parameter or return value.

   c. Click **OK**.

Note An Inport block in a Simulink function cannot inherit its **Data type** and **Port dimensions**. For more information, see “Guidelines for Using Simulink Functions” on page 9-8.

Call Simulink Functions in States and Transitions

You can call Simulink functions from the actions of any state or transition or from other functions.

To call a Simulink function, use the function signature and include an argument value for each formal argument in the function signature.

```
[return_val1,return_val2,...] = function_name(arg1,arg2,...)
```

If the data types of the two arguments differ, the function casts the argument to the type of the formal argument.

Specify Properties of Simulink Functions

You can modify properties for a Simulink function in the Block Parameters dialog box.

1 In the Stateflow Editor, right-click the Simulink function.
2 Select **Properties**.
3 Edit the Simulink function properties.

You can also modify these properties in the Model Explorer. For more information, see Model Explorer (Simulink).

For a description of the Simulink function properties, see Subsystem.

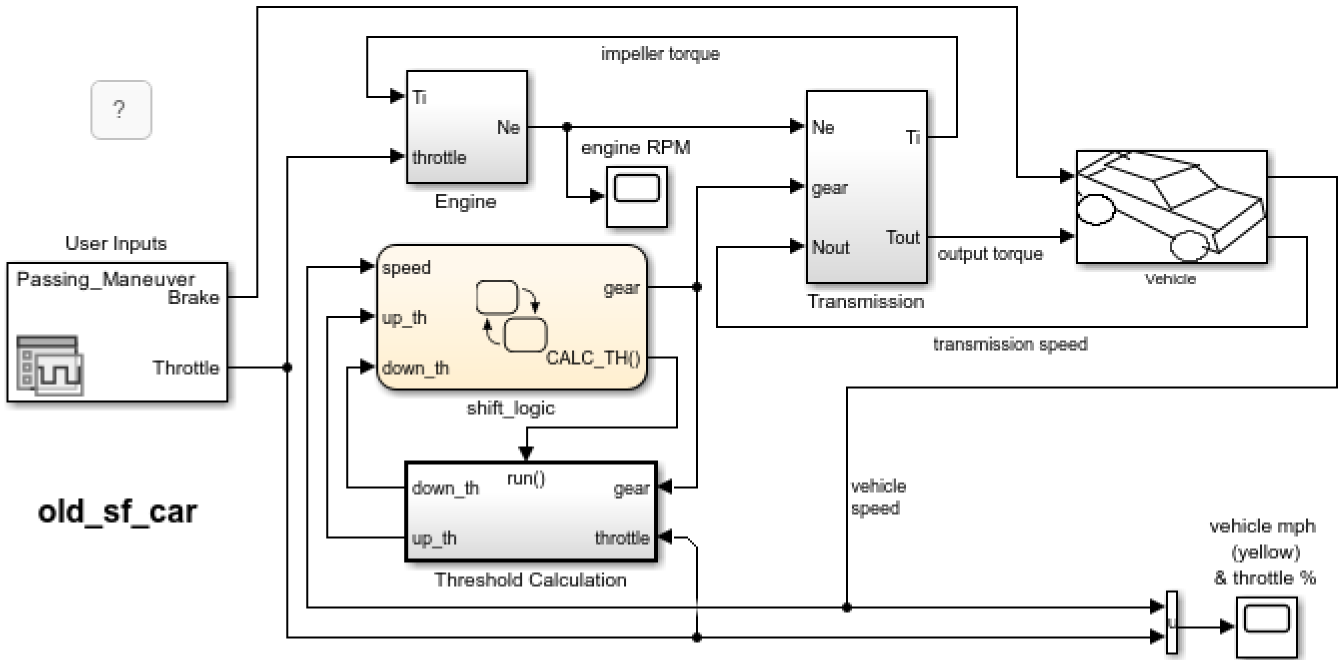
You can specify additional properties of Simulink functions programmatically by using `Stateflow.SLFunction` objects. For more information about the Stateflow programmatic interface, see “Overview of the Stateflow API”.

Use a Simulink Function to Access Simulink Blocks

In this example, you can compare modeling the shift logic of a car system. The first model does not use Simulink functions while the second does.

Model Without a Simulink Function

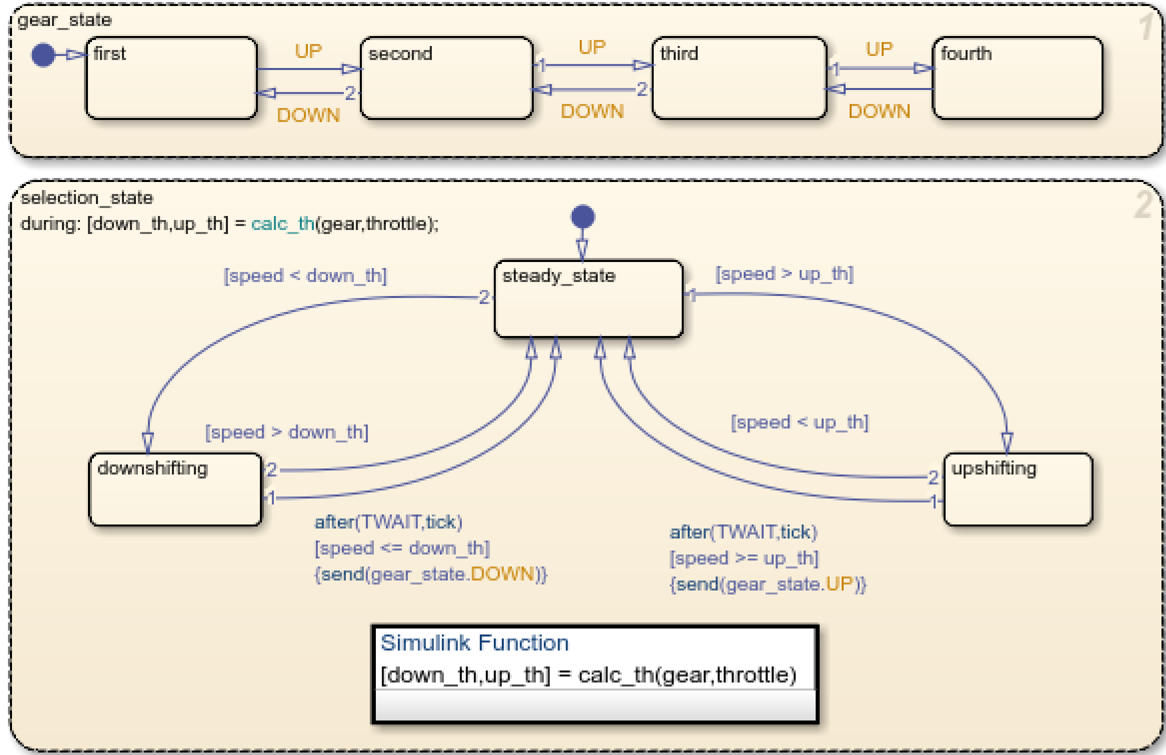
This model uses a function-call subsystem, `run()`, Simulink model to calculate the threshold for shifting gears. The Stateflow chart then uses an output event, `CALC_TH()`, to call the subsystem.



For more information about function-call subsystems, see “Using Function-Call Subsystems” (Simulink).

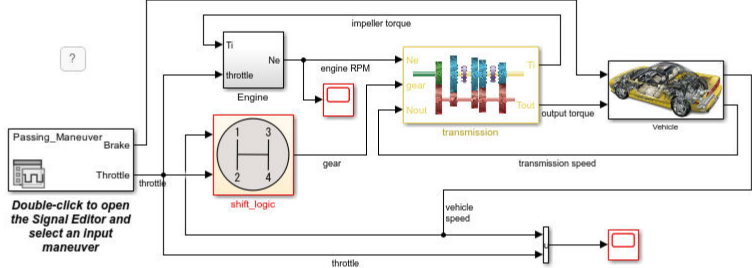
Model With a Simulink Function

This same functionality can be achieved with a Simulink function. In this Stateflow chart, the Simulink function `calc_th` is used to calculate the threshold.



The `during` action in `selection_state` contains a function call to `calc_th`, which contains Simulink blocks.

This modeling method minimizes the objects in your model.

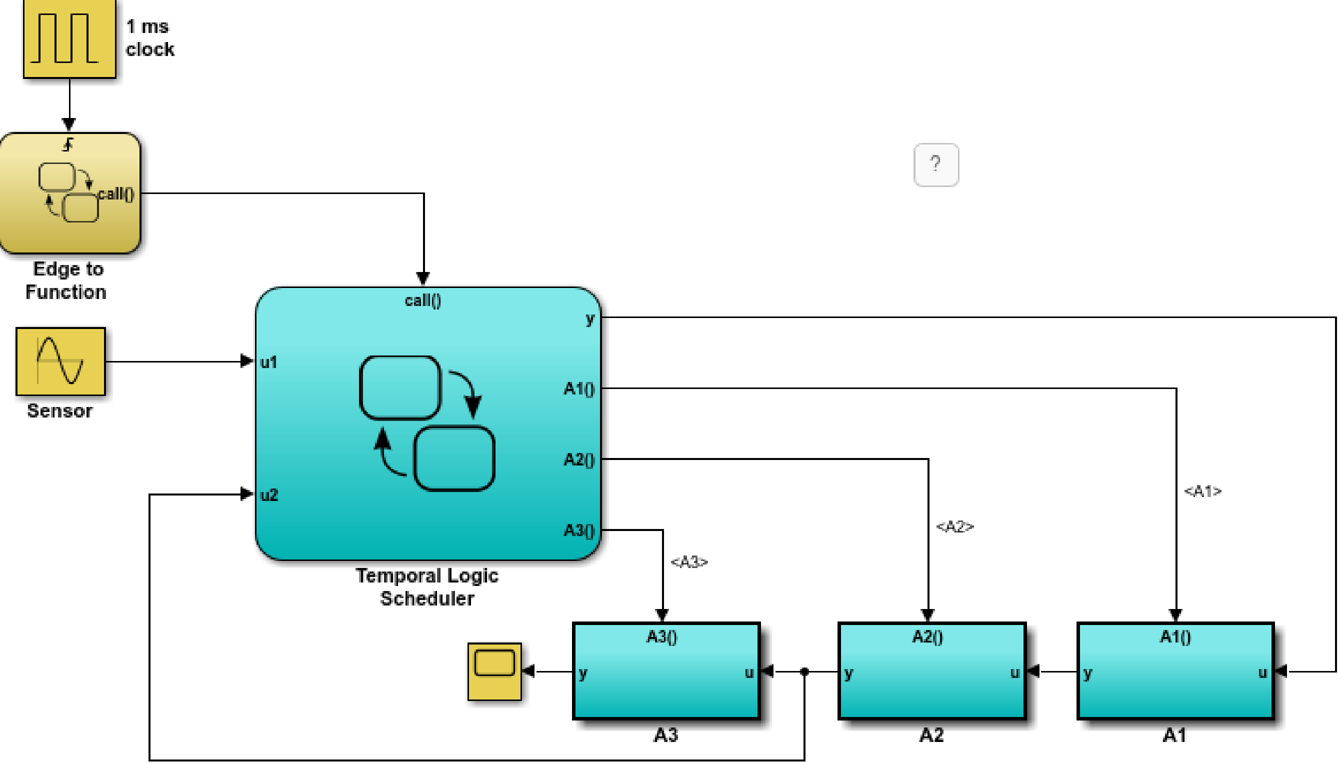


Use a Simulink Function to Schedule Execution of Multiple Controllers

In this example, you can compare two ways of scheduling execution of multiple controllers. The first model does not use Simulink functions while the second does.

Model Without Simulink Functions

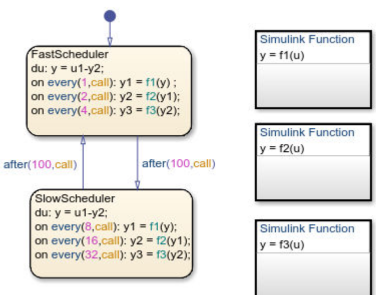
This model uses function-call subsystems to model each controller. The model includes output events in a Stateflow chart to schedule execution of the subsystems.



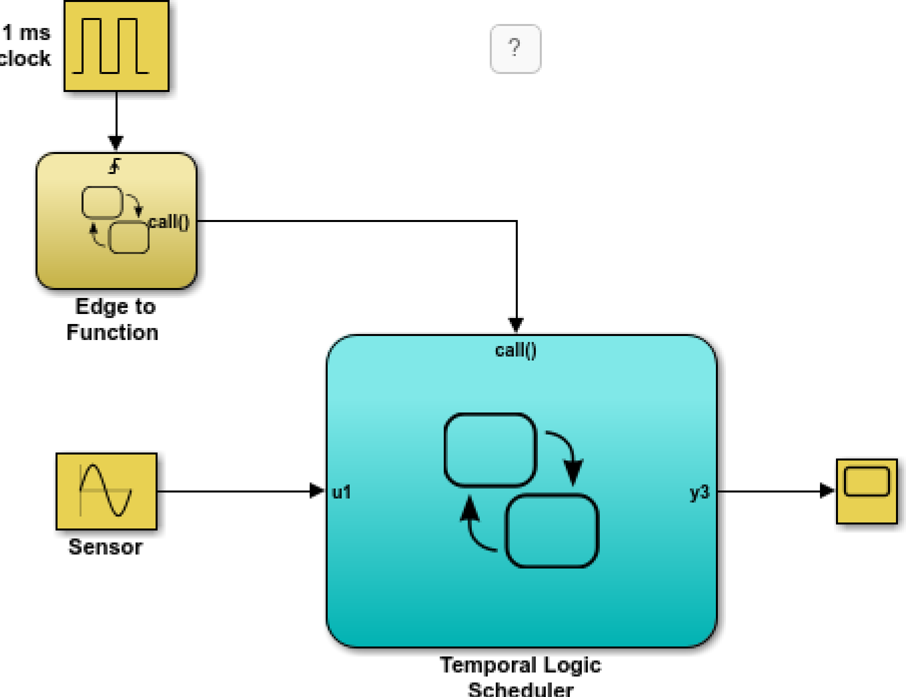
For each output event, a signal line is needed to connect the Stateflow chart with the corresponding function-call subsystem.

Model Method With Simulink Functions

Each controller can also be modeled with a Simulink function in a Stateflow chart. This model uses function calls to schedule execution of the subsystems.



This modeling method minimizes the objects in your model.



Guidelines for Using Simulink Functions

Use Alphanumeric Characters and Underscores in Argument Names

By using alphanumeric characters for argument names, you ensure that the names of Inport and Output blocks are compatible with the identifier naming rules of Stateflow charts.

Explicitly Set the Properties of Inport Blocks

The Inport blocks in a Simulink function cannot inherit their data types and sizes. You must set the **Data type** and **Port dimensions** of each Inport block that is not a scalar of type `double`.

The Outport blocks in a Simulink function can inherit sizes and data types based on the connections inside the subsystem. You can specify the **Data type** and **Port dimensions** of these blocks as inherited.

Tip To make it easier to update the properties of Inport blocks, you can specify data types and sizes as parameters.

Convert Discontiguous Signals to Contiguous Signals

Outport blocks in Simulink functions 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 Outport block. This 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 when the 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 “Create Nonvirtual Buses” (Simulink).

Do Not Use Simulink Functions in Moore Charts

You cannot use Simulink functions in Moore charts. This restriction prevents violations of Moore semantics during chart execution.

Do Not Call Simulink Functions in Default Transitions That Execute During Chart Initialization

If you select the chart property **Execute (enter) Chart At Initialization**, you cannot call Simulink functions in default transitions that execute the first time that the chart awakens. Otherwise, the chart generates a run-time error during simulation.

Do Not Call Simulink Functions in State `During` Actions Or Transition Conditions of Continuous-time Charts

In continuous-time charts, you cannot call Simulink functions during minor time steps. Instead, call Simulink functions in actions that occur during major time steps: state `entry` or `exit` actions and transition actions. Calling Simulink functions in state `during` actions or transition conditions results in a run-time error during simulation.

Do Not Generate HDL Code for Simulink Functions

Simulink functions do not support HDL code generation. Generating HDL code for charts that contain Simulink functions results in a run-time error during simulation.

Pass Arguments by Value

Passing an argument to a Simulink function by reference results in a run-time error during simulation.

See Also

Simulink Function | Trigger | Inport | Outport | Signal Conversion | Bus Creator | Mux

More About

- “Export Stateflow Functions for Reuse” on page 6-14
- “Simulink Functions Overview” (Simulink)
- “Add a Simulink Function to a Model” (Simulink)
- “Manage Queue for Shared Printer Server” on page 9-32

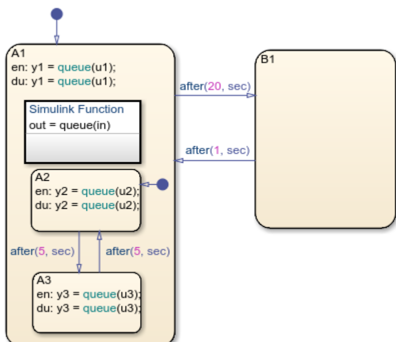
Bind a Simulink Function to a State

Simulink functions are not supported in standalone Stateflow charts in MATLAB.

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.



Since the function `queue` resides in state `A1`, the function binds to state `A1`.

- State `A1` and its substates `A2` and `A3` can call the function `queue`, but state `B1` cannot.
- When state `A1` is entered, `queue` is enabled.
- When state `A1` is exited, `queue` is disabled.

Control Subsystem Variables When the Simulink Function Is Disabled

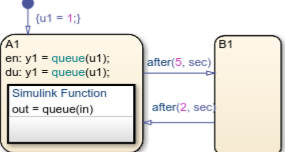
If a Simulink function binds to a state, you can hold the subsystem variables at their values from the previous execution or reset the variables to their initial values. To choose the desired behavior for your subsystem, follow these steps:

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**.

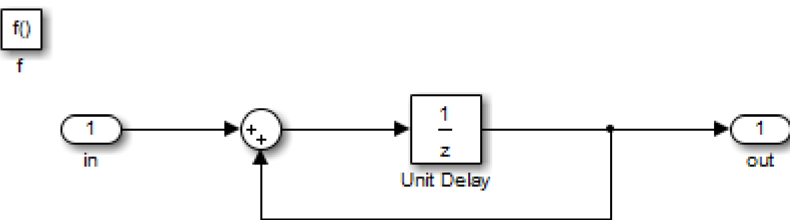
Option	Description
`held`	Holds the values of the subsystem variables from the previous execution
`reset`	Resets the subsystem variables to their initial values

Binding a Simulink Function to a State

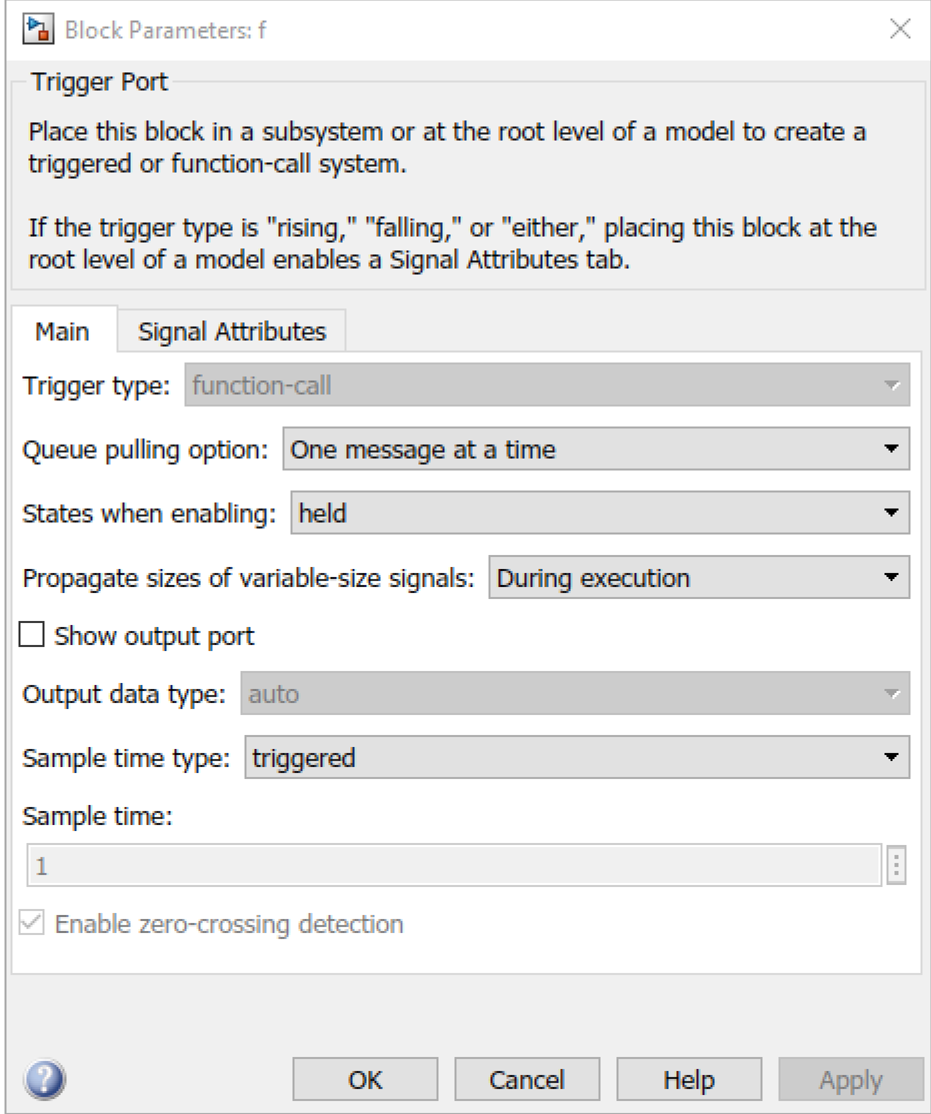
This example shows how a Simulink function behaves when bound to a state.



The function `queue` contains a block diagram that increments a counter by 1 each time the function executes.



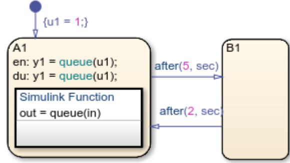
The Block Parameters dialog box for the trigger port appears as follows.



In the dialog box, setting **Sample time type** to `periodic` enables the **Sample time** field, which defaults to 1. These settings tell the function to execute for each time step specified in the **Sample time** field while the function is enabled.

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 “Compare Solvers” (Simulink).

Simulation Behavior of the Chart

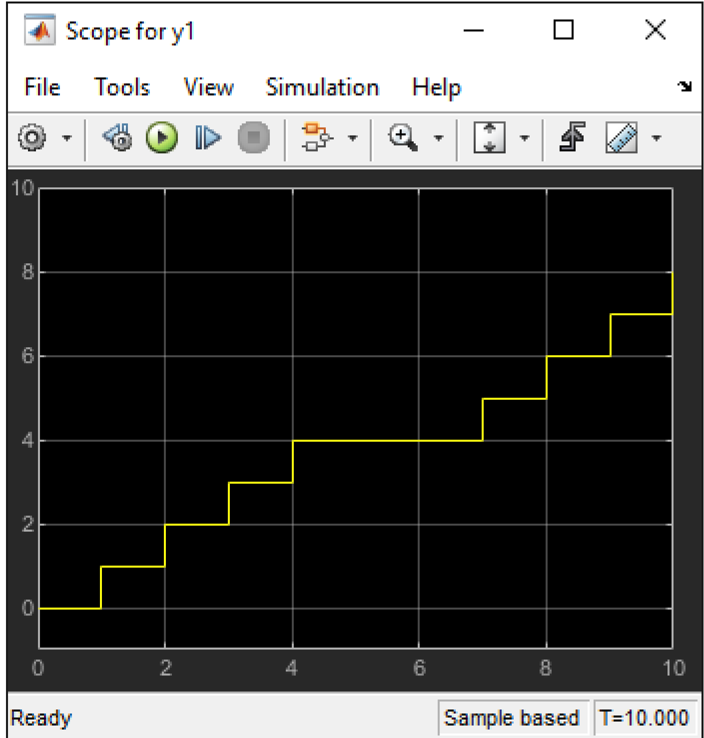


When you simulate the chart, the following actions occur.

1 The chart takes the default transition to state `A1`, and the local data `u1` is set 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 Once the condition is true, 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.

Function Behavior When Variables Are Held

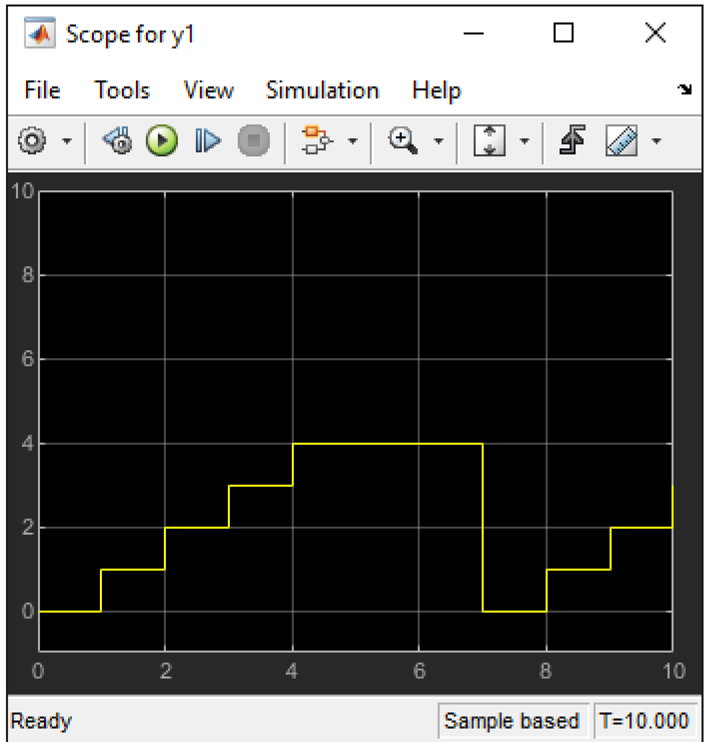
If you set **States when enabling** to `held`, the output `y1` is as follows.



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.

Function Behavior When Variables Are Reset

If you set **States when enabling** to `reset`, the output `y1` is as follows.



When state A1 becomes inactive at t = 5, the Simulink function does *not* hold the counter value. When A1 is active again at t = 7, the counter resets to zero. Therefore, the output `y1` resets too.

See Also

More About

- "Reuse Simulink Functions in Stateflow Charts" on page 9-2

Design Charts with Simulink Functions

In this tutorial, you use a Simulink function in a Stateflow chart to improve the design of a model that contains a function-call subsystem. 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.

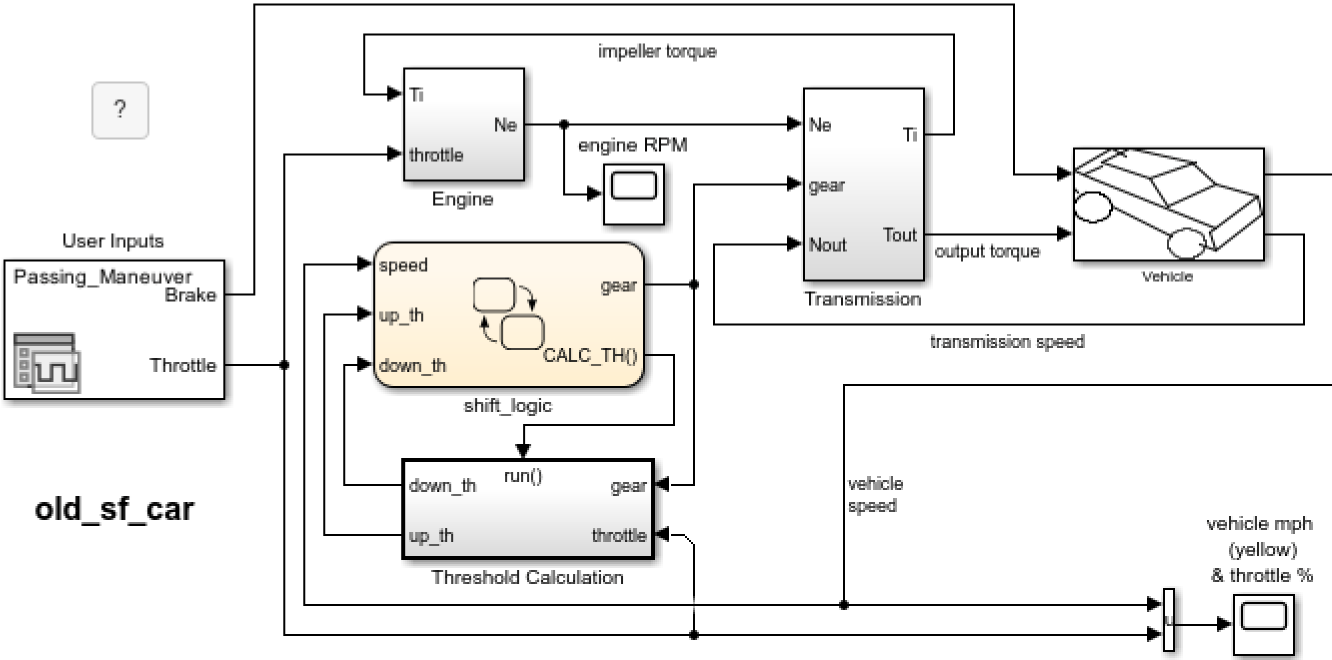
Simulink functions are not supported in standalone Stateflow charts in MATLAB. For more information, see “Reuse Simulink Functions in Stateflow Charts” on page 9-2.

Note To skip the conversion steps, you can open the modified model by entering:

```
openExample("stateflow/AutomaticTransmissionLegacyExample", ...
    supportingFile="old_sf_car_with_sl_function"))
```

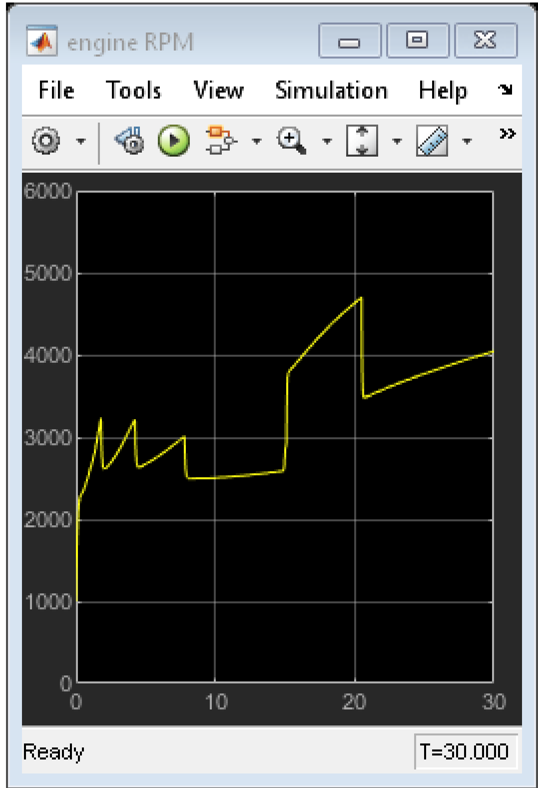
Open the Model

Open the model `old_sf_car`. This model contains a function-call subsystem named `Threshold Calculation` and a Stateflow chart named `shift_logic`.



When you run this model, the chart broadcasts the output event `CALC_TH` to trigger the function-call subsystem. The subsystem interpolates two values for the `shift_logic` chart. The subsystem outputs (`up_th` and `down_th`) return to the chart as inputs.

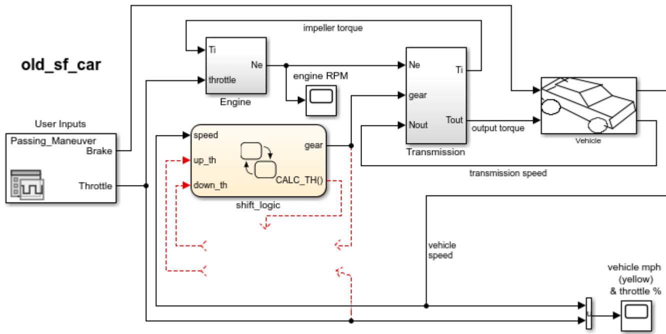
After the simulation, the `engine RPM` Scope block displays these results.



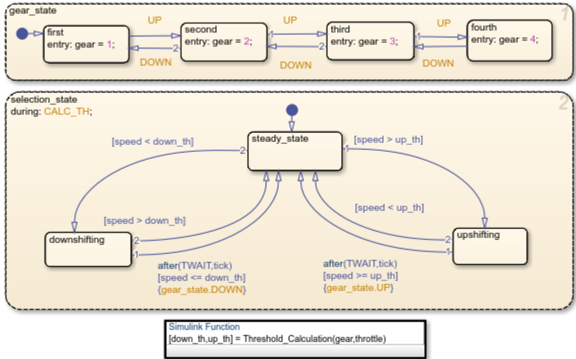
Add a Simulink Function to the Chart

Follow these steps to add a Simulink function to the shift_logic chart.

1. In the Simulink model, right-click the Threshold Calculation block in the lower left corner and select **Cut** from the context menu.

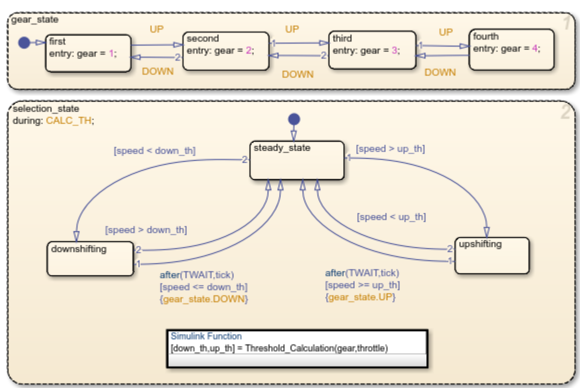


2. Open the shift_logic chart.
3. In the chart, right-click below `selection_state` and select **Paste** from the context menu.
4. Expand the new Simulink function so that the signature fits inside the function box.



Tip To change the font size of a function, right-click the function box and select a new size from the **Font Size** menu.

5 Expand the border of `selection_state` to include the new function.



Note The function resides in this state instead of the chart level because no other state in the chart requires the function outputs `up_th` and `down_th`. See “Bind a Simulink Function to a State” on page 9-10.

6 Rename the Simulink function from `Threshold_Calculation` to `calc_threshold` by entering `[down_th, up_th] = calc_threshold(gear, throttle)` in the function box.

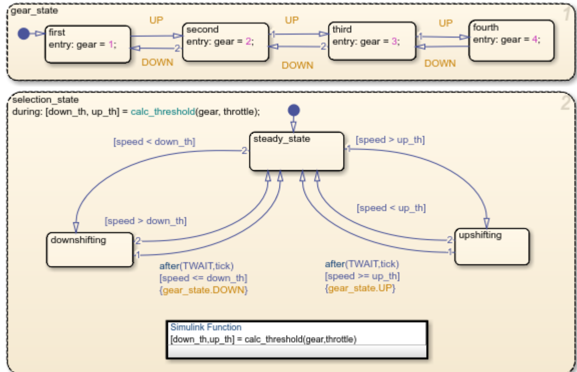
Change the Scope of Chart Data

In the Model Explorer, change the scope of chart-level data `up_th` and `down_th` to `Local` because calculations for those data now occur inside the chart.

Update State Action in the Chart

In the Stateflow Editor, change the `during` action in `selection_state` to call the Simulink function `calc_threshold`.

```
during: [down_th, up_th] = calc_threshold(gear, throttle);
```



Add Data to the Chart

Because the function `calc_threshold` takes `throttle` as an input, you must define that data as a chart input. (For details, see "Add Stateflow Data" on page 10-2.)

1. Add input data `throttle` to the chart with a **Port** property of 1.

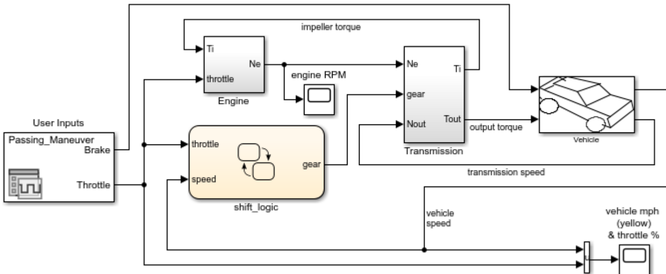
   Using port 1 prevents signal lines from overlapping in the Simulink model.
2. In the Simulink model, add a signal line for `throttle` to an input to the Engine block and to an input to the shift_logic chart.

Remove Unused Items in the Model

1. In the Model Explorer, delete the function-call output event `CALC_TH` because the Threshold Calculation block no longer exists.
2. Delete any dashed signal lines from your model.

Run the New Model

Your new model looks something like this:



If you simulate the new model, the results match those of the original design.

See Also

More About

- “Reuse Simulink Functions in Stateflow Charts” on page 9-2
- “Define a Simulink Function” on page 9-3

Schedule Execution of Multiple Controllers

In this tutorial, you use Simulink functions in a Stateflow chart to improve the design of a model that contains three function-call subsystems. In the model `sf_temporal_logic_scheduler`:

- 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.

Simulink functions are not supported in standalone Stateflow charts in MATLAB. For more information, see “Reuse Simulink Functions in Stateflow Charts” on page 9-2.

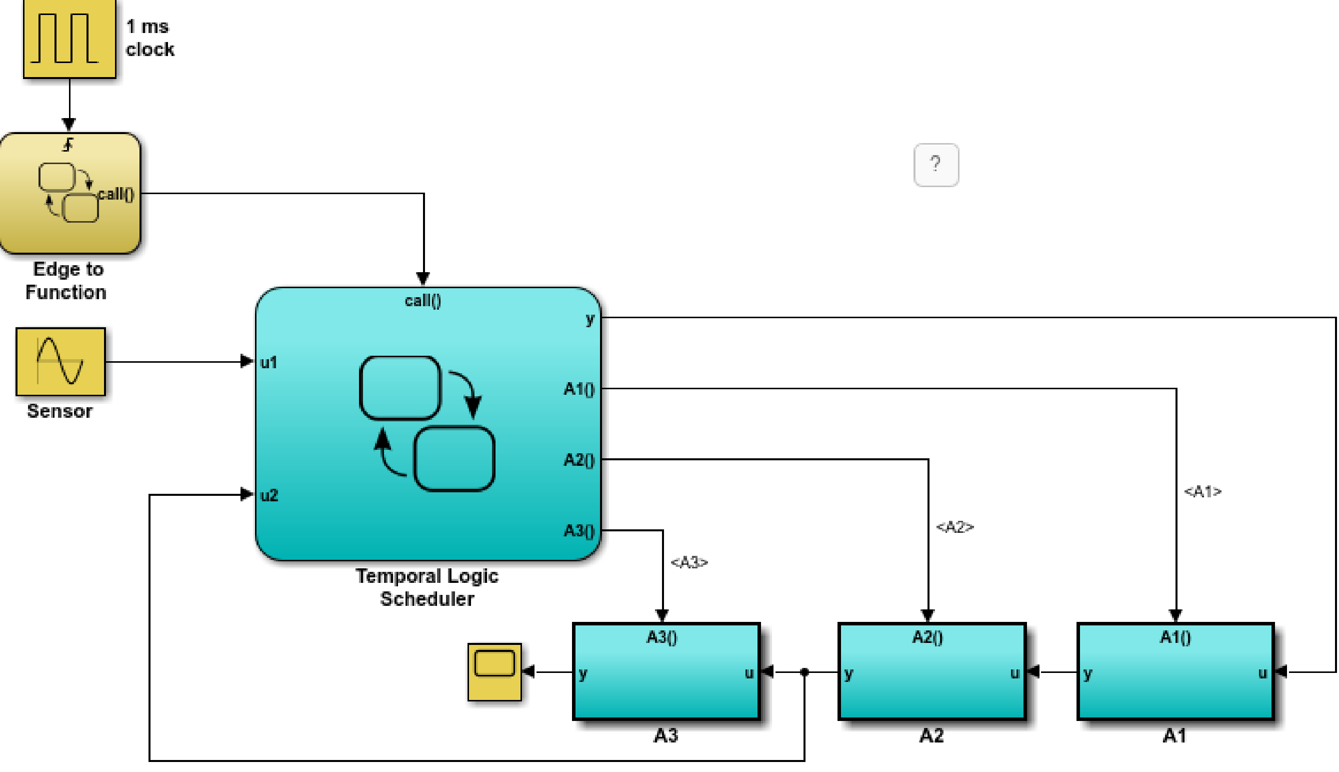
Note To skip the conversion steps, you can open the modified model by entering:

```
openExample("stateflow/TemporalLogicSchedulerExample", ...
    supportingFile="sf_temporal_logic_scheduler_with_sl_fcns")
```

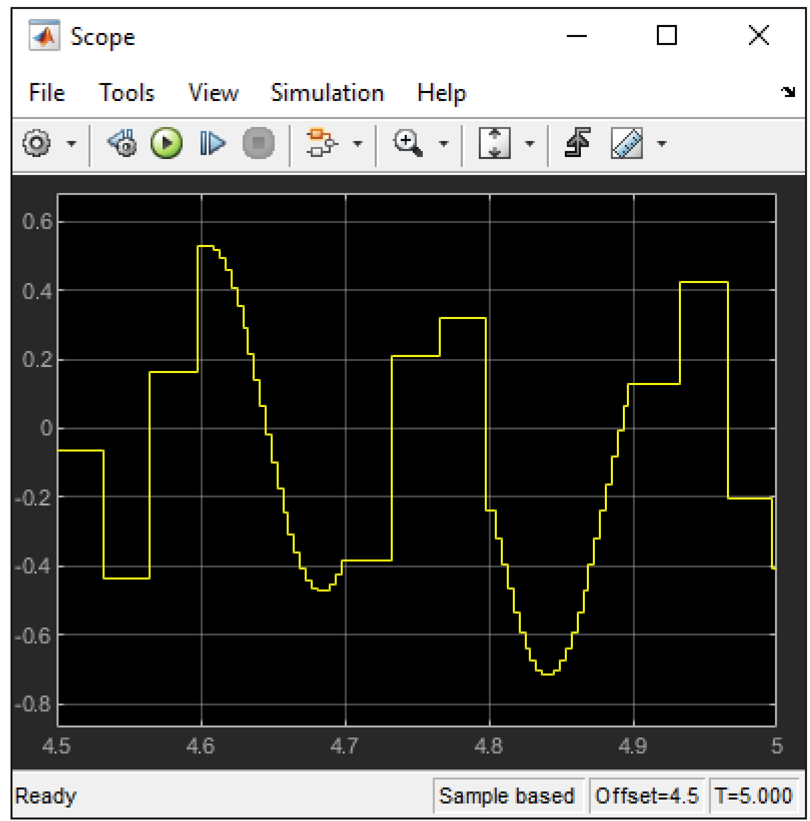
Open the Model

Open the `sf_temporal_logic_scheduler` model.

```
openExample("stateflow/TemporalLogicSchedulerExample")
```



If you simulate the model, you see this result in the scope.

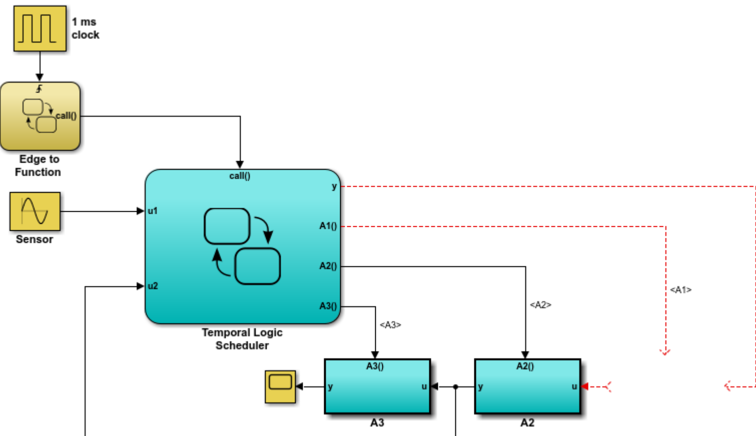


For more information on this example, see “Schedule Subsystems to Execute at Specific Times” on page 27-9.

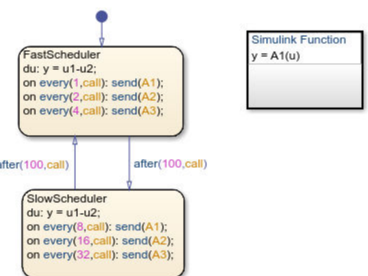
Add Simulink Functions to the Chart

Follow these steps to add Simulink functions to the Temporal Logic Scheduler chart.

1 In the Simulink model, right-click the A1 block in the lower right corner and select **Cut** from the context menu.



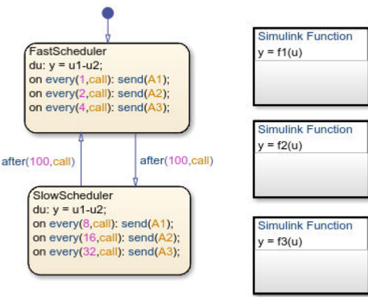
2 Open the Temporal Logic Scheduler chart.
3 In the chart, right-click outside any states and select **Paste** from the context menu.
4 Expand the new Simulink function so that the signature fits inside the function box.



Tip To change the font size of a function, right-click the function box and select a new size from the **Font Size** menu.

5 Rename the Simulink function from `A1` to `f1` by entering `y = f1(u)` in the function box.
6 Repeat steps 1 through 5 for these cases:

- Copying the contents of A2 into a Simulink function named `f2`.
- Copying the contents of A3 into a Simulink function named `f3`.



Note The new functions reside at the chart level because both states `FastScheduler` and `SlowScheduler` require access to the function outputs.

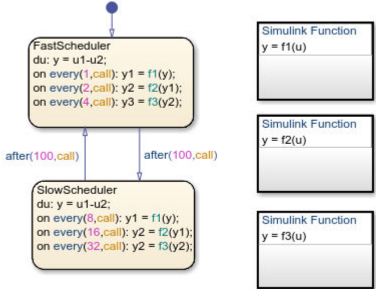
Change the Scope of Chart Data

In the Model Explorer, change the scope of chart-level data `y` to `Local` because the calculation for that data now occurs inside the chart.

Update State Actions in the Chart

In the Stateflow Editor, you can replace event broadcasts in state actions with function calls.

1 Edit the state actions in `FastScheduler` and `SlowScheduler` to call the Simulink functions `f1`, `f2`, and `f3`.



2. In both states, update each `during` action as follows.

```
du: y = u1-y2;
```

Add Data to the Chart

For the `on every` state actions of `FastScheduler` and `SlowScheduler`, define three data. (For details, see “Add Stateflow Data” on page 10-2.)

1. Add local data `y1` and `y2` to the chart.
2. Add output data `y3` to the chart.
3. In the model, connect the output for `y3` to the scope.

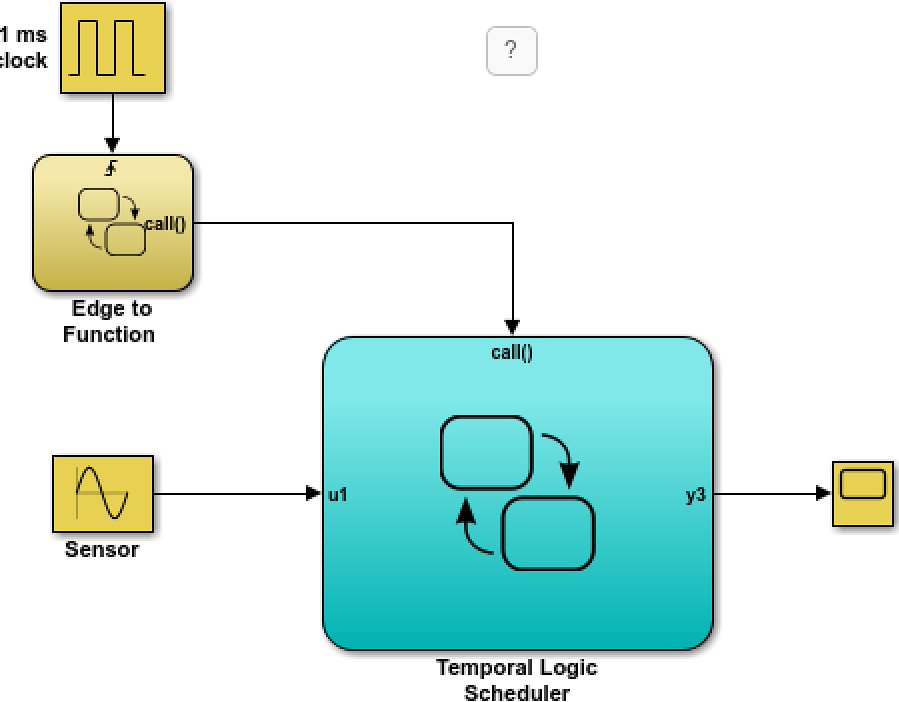
Tip To flip the Scope block, select the block. Then, in the toolstrip, on the **Format** tab, click **Flip left-right** .

Remove Unused Items in the Model

1. In the Model Explorer, delete output events `A1`, `A2`, and `A3` and input data `u2` because the function-call subsystems no longer exist.
2. Delete any dashed signal lines from your model.

Run the New Model

Your new model looks something like this:



If you simulate the new model, the results match those of the original design.

See Also

More About

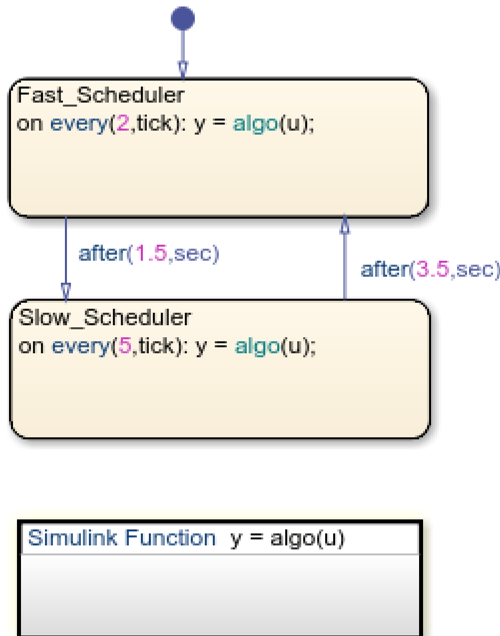
- “Reuse Simulink Functions in Stateflow Charts” on page 9-2
- “Define a Simulink Function” on page 9-3

Schedule Simulink Functions by Using Stateflow

This example shows how to schedule a Simulink® function in a Stateflow® chart by using temporal logic. A Simulink function is a Simulink subsystem that you define inside the Stateflow chart and call in the actions of states and transitions. For more information, see “Reuse Simulink Functions in Stateflow Charts” on page 9-2.

Open and Examine the Model

In this example, the Simulink function `algo` increments the input by one at each time step. The Stateflow chart includes two states that call the function `algo` at different rates.



Temporal logic operators determine the schedules for function calls and transitions between the states. The event-based temporal logic operator `every` sets `FastScheduler` to call the Simulink function every two time steps and `SlowScheduler` to call the same function every five time steps. Consequently, `FastScheduler` executes the function more frequently than `SlowScheduler`.

The transition from `FastScheduler` to `SlowScheduler` occurs after `FastScheduler` is active for 1.5 seconds. The transition back to `FastScheduler` occurs after `SlowScheduler` is active for 3.5 seconds. The absolute-time temporal logic operator `after` controls the timing of transitions between states.

Differences Between Event-Based and Absolute-Time Temporal Logic Operators

The type of temporal logic operator that you use depends on whether you are scheduling a function call or a transition.

Event-based temporal logic operators, such as `every`, depend on the step size used by the Simulink solver. The number of function calls since a state became active also depends on the solver's step

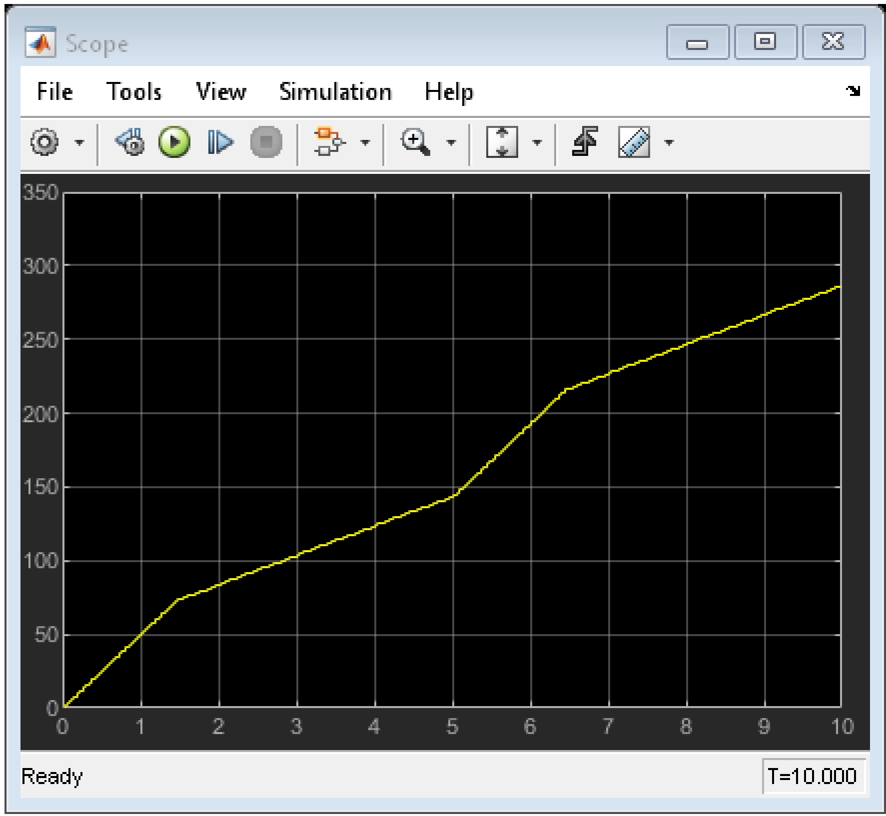
size. Therefore, in charts in a Simulink model, using `every` as an absolute-time temporal logic operator is not supported.

Absolute-time temporal logic operators such as after depend on the elapsed time since a state became active. In charts in a Simulink model, using `at` as an absolute-time temporal logic operator is not supported

For more information, see “Control Chart Execution by Using Temporal Logic” on page 14-35.

View Simulation Result

Run the model. The Scope block illustrates the rates of each function call by the differing slope steepness. Steeper slopes indicate the more frequent Simulink function calls.



See Also

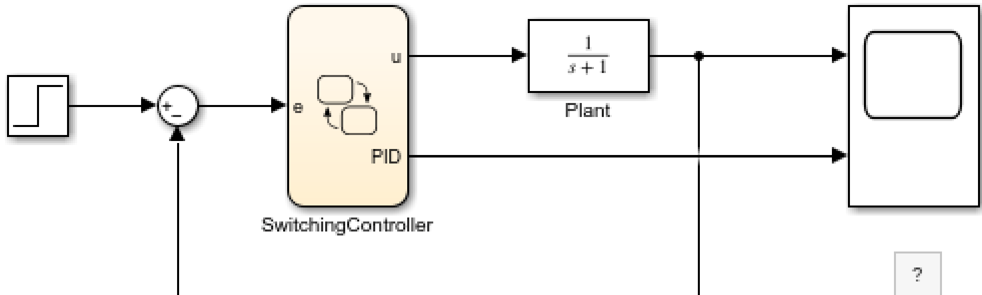
`after` | `every` | `Temporal Logic Operators`

More About

- “Reuse Simulink Functions in Stateflow Charts” on page 9-2
- “Control Chart Execution by Using Temporal Logic” on page 14-35
- “Schedule Execution of Multiple Controllers” on page 9-20

Design Switching Controllers by Using Simulink Functions

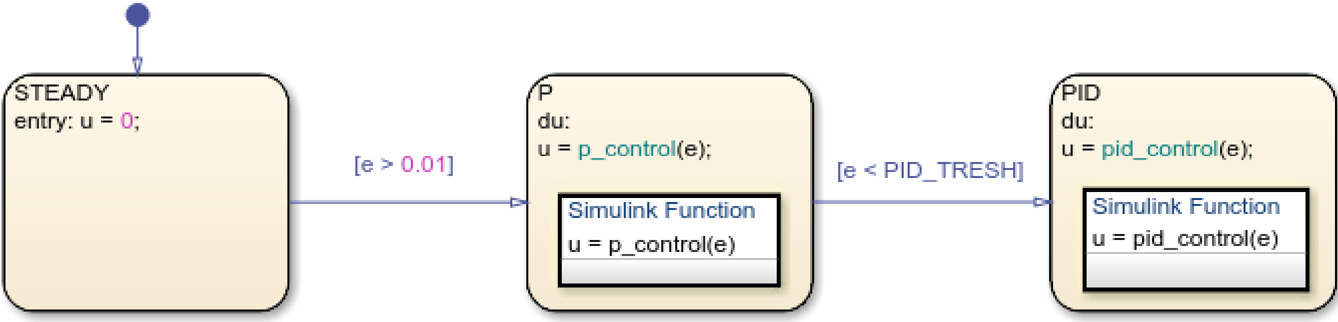
This example shows to use a Stateflow® chart to model the decision logic for switching between a P-only controller and a PID controller.



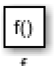
The Stateflow chart `SwitchingController` implements a simple switching controller which switches between three states: `STEADY`, `P` and `PID`. When in `STEADY` state, we produce zero control output. When in `P` or `PID`, we delegate to Simulink® function call subsystems in order to compute the required control effort.

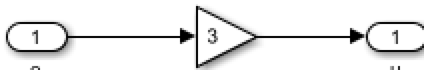
The **Create data for monitoring** option for the state `PID` is checked. Therefore, in addition to the control output `u`, the Stateflow chart also produces a logging output with the same name as the state `PID`.

The condition for switching from `P` to `PID` is based on the error being low enough `[e < PID_TRESH]`. `PID_TRESH` is a variable defined in the Model Workspace with a value of 0.3.

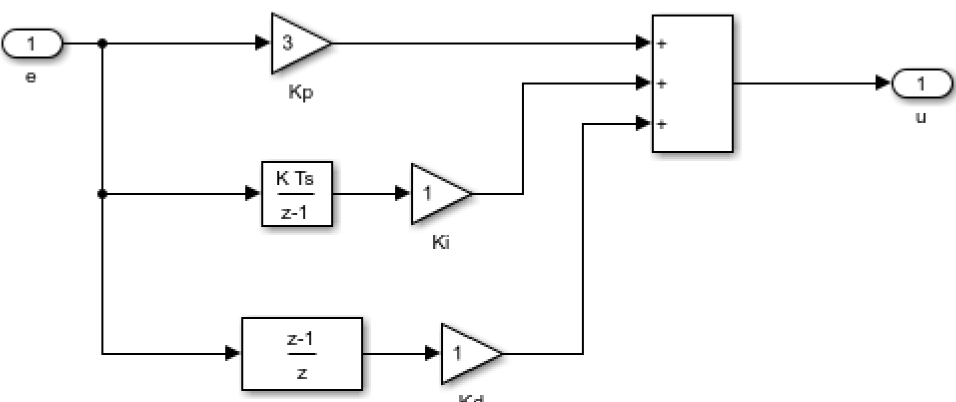


The Simulink subsystem `SwitchingController/P.p_control` implements a very simple proportional control with a gain of 3. If we had continued to stay in the state `P`, the steady state gain of the closed-loop system would be 3/4 = 0.75. Therefore, we would get an error of 0.25.

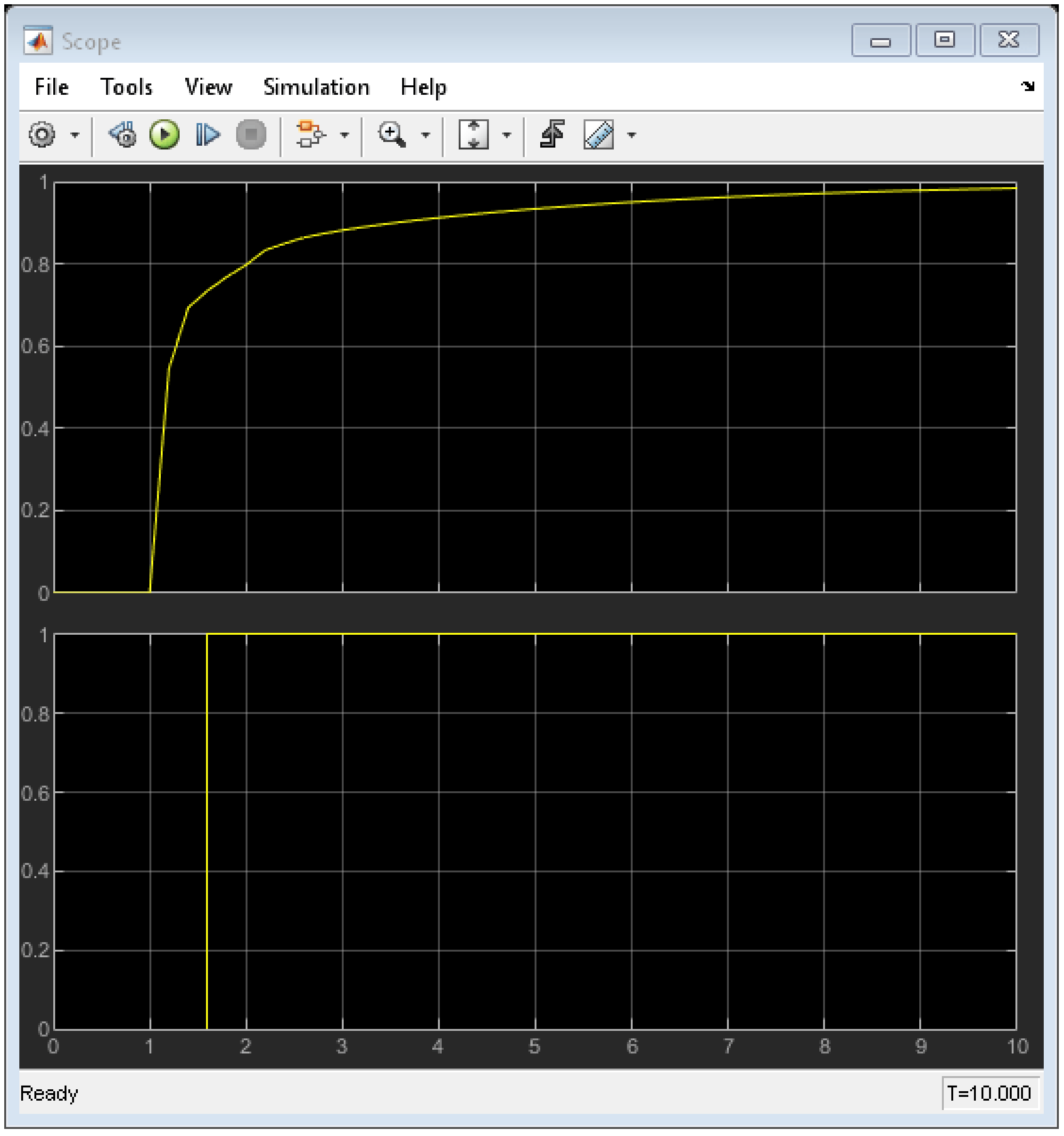




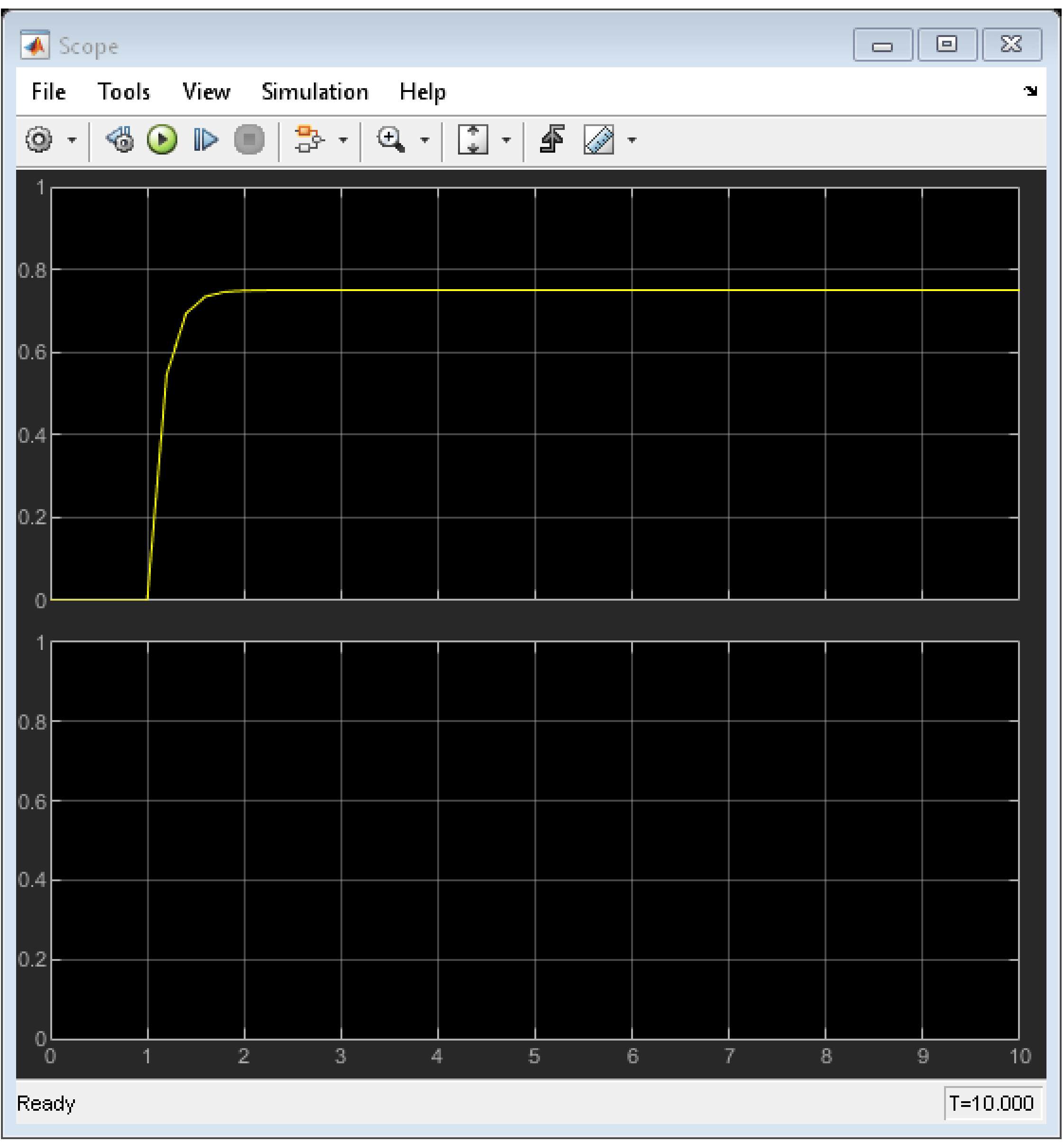
The Simulink subsystem `SwitchingController/PID.pid_control` implements a simple PID control strategy. The proportional gain is the same as in `P` thereby ensuring a smooth transition in the control effort.



When we simulate the model, we notice that the steady state error approaches zero.



In the absence of the PID control, we would have had a steady state error of 0.25. If we change `PID_TRESH` to 0.1, we will *never* get to `PID`, because the error will never get below 0.25 as long as we are in state `P`.



See Also

More About

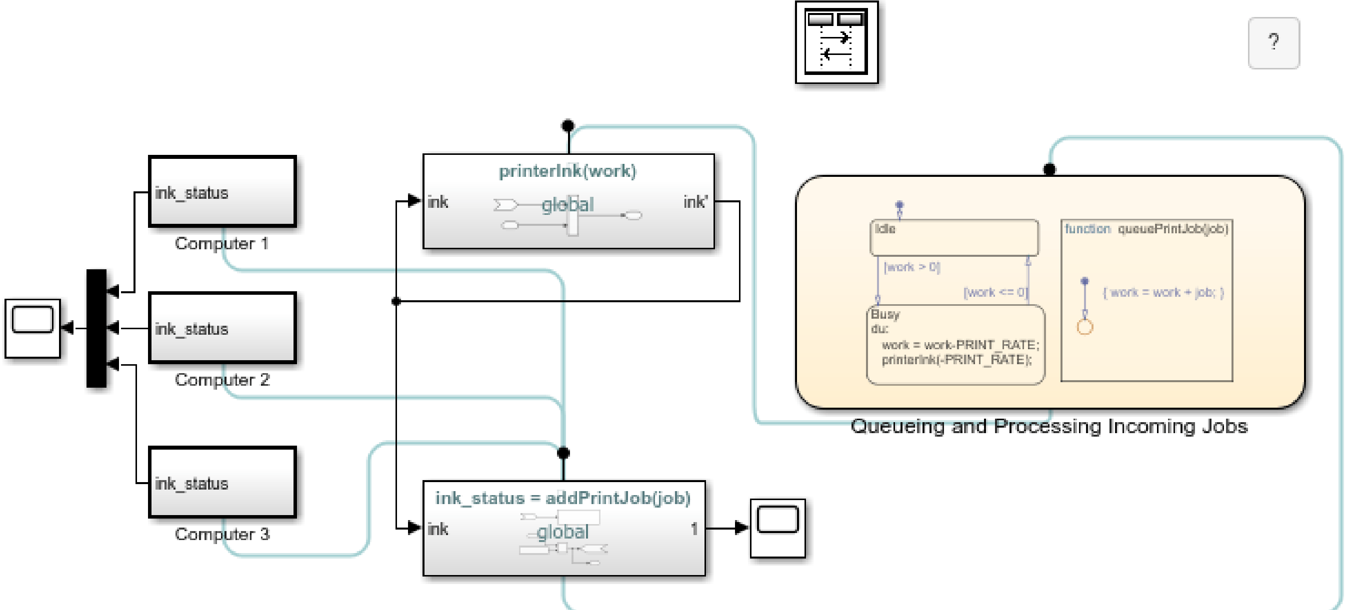
- "Reuse Simulink Functions in Stateflow Charts" on page 9-2

Manage Queue for Shared Printer Server

This example shows how to share functions to communicate between a Simulink® model and a Stateflow® chart. For instance, you can:

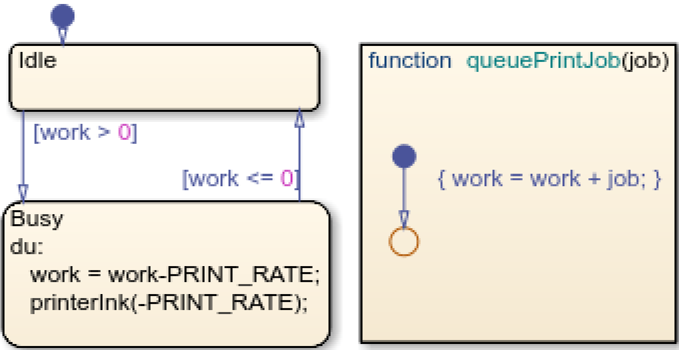
- Use a Stateflow chart to call a Simulink function that you define in your model.
- Use a Function Caller block in your Simulink model to call a function that you define in your Stateflow chart.

This example models three computer clients that share a network printer. Each computer sends print jobs to a common queue for processing. Each computer client invokes the printer server with a call to the Simulink Function block `addPrintJob`. To add the print job to the work load, the `addPrintJob` function calls the graphical function `queuePrintJob` in the Stateflow chart. To model usage of printer ink, the chart processes the work and calls the Simulink Function block `printerInk`.



Call a Simulink Function from Stateflow

The function `printerInk` is defined in a Simulink Function block at the top level of the model. The function interface `printerInk(work)` defines one input argument. The Simulink Function, printerInk, also interacts with the model with signal lines through the inport `ink` and outport `ink'`. The state `Busy` matches the function signature for `printerInk(work)` by passing one input argument.



Export Stateflow Functions to Simulink

In the chart `Queuing` and `Processing Incoming Jobs`, the properties **Export chart level functions** and **Treat exported functions as globally visible** are selected. These properties allow the Simulink function `addPrintJob` to call the chart graphical function, `queuePrintJob`.

See Also

Simulink Function

More About

- “Export Stateflow Functions for Reuse” on page 6-14
- “Simulink Functions Overview” (Simulink)
- “Model Reference Basics” (Simulink)

Define Data

- “Add Stateflow Data” on page 10-2
- “Set Data Properties” on page 10-5
- “Specify Type of Stateflow Data” on page 10-20
- “Specify Size of Stateflow Data” on page 10-26
- “Specify Units for Stateflow Data” on page 10-29
- “Share Data with Simulink and the MATLAB Workspace” on page 10-30
- “Share Parameters with Simulink and the MATLAB Workspace” on page 10-32
- “Access Data Store Memory from a Chart” on page 10-34
- “Handle Integer Overflow for Chart Data” on page 10-37
- “Identify Data by Using Dot Notation” on page 10-39
- “Resolve Data Properties from Simulink Signal Objects” on page 10-43

Add Stateflow Data

When you want to store values that are visible at a specific level of the Stateflow hierarchy, add data to your chart. When you simulate your model, chart data objects retain their values for the duration of the simulation.

Data defined in a Stateflow chart is visible by multiple Stateflow objects in the chart, including states, transitions, MATLAB functions, and truth tables. To determine what data is used in a state or transition, right-click the state or transition and select **Explore**. A context menu lists the names and scopes of all resolved symbols in the state or transition. Selecting a symbol from the context menu displays its properties in the Model Explorer. Selecting an output event from the context menu opens the Simulink subsystem or Stateflow chart associated with the event.

Note Stateflow data is not available to Simulink functions within a Stateflow chart.

You can add data to a Stateflow chart by using the **Symbols** pane, the Stateflow Editor menu, or the Model Explorer.

Add Data Through the Symbols Pane

1. In the **Modeling** tab, under **Design Data**, select **Symbols Pane**.
2. Click the **Create Data** icon .
3. In the row for the new data, under **Type**, click the icon and choose:

   - `Input Data`
   - `Local Data`
   - `Output Data`
   - `Constant`
   - `Data Store Memory`
   - `Parameter`
   - `Temporary`

   For more information about these options, see "Scope" on page 10-5.
4. Edit the name of the data.
5. For input and output data, click the **Port** field and choose a port number.
6. To specify properties for data, open the **Property Inspector**. In the **Symbols** pane, right-click the row for the symbol and select **Explore**. For more information, see "Set Data Properties" on page 10-5.

Add Data by Using the Stateflow Editor Menu

1. In a Stateflow chart in a Simulink model, select the menu option corresponding to the scope of the data that you want to add. For more information about these options, see "Scope" on page 10-5.

Scope	Menu Option
`Input`	In the **Modeling** tab, under **Design Data**, select **Data Input**.
`Output`	In the **Modeling** tab, under **Design Data**, select **Data Output**.
`Local`	In the **Modeling** tab, under **Design Data**, select **Local**.
`Constant`	In the **Modeling** tab, under **Design Data**, select **Constant**.
`Parameter`	In the **Modeling** tab, under **Design Data**, select **Parameter**.
`Data Store Memory`	In the **Modeling** tab, under **Design Data**, select **Data Store**.

2. In the Data dialog box, specify data properties. For more information, see “Set Data Properties” on page 10-5.

Add Data Through the Model Explorer

To add function- or state-parented data to Stateflow charts in Simulink models, use the Model Explorer:

1. In the **Modeling** tab, under **Design Data**, select **Model Explorer**.
2. In the **Model Hierarchy** pane, select the object in the Stateflow hierarchy where you want to make the new data visible. The object that you select becomes the parent of the new data.
3. In the Model Explorer toolstrip, select the Add Data button. Alternatively, in the Model Explorer menu, select **Add** > **Data**. The new data with a default definition appears in the **Contents** pane of the Model Explorer.
4. In the **Data** pane, specify the properties of the data. For more information, see “Set Data Properties” on page 10-5.

Tip You do not need to create local or temporary data explicitly in these types of functions:

- Graphical functions in charts that use MATLAB as the action language
- Truth table functions that use MATLAB as the action language
- MATLAB functions

Instead, in these functions, you can use undefined variables to store values that are accessible only during the rest of the function call. To store values that persist across function calls, use local data at the chart level. Alternatively, in MATLAB functions, you can use the keyword `persistent`.

Best Practices for Using Data in Charts

Avoid Inheriting Output Data Properties from Simulink Blocks

Stateflow output data should not inherit properties from output signals, because the values back propagate from Simulink blocks and can be unpredictable.

Generate More Efficient Code by Using In-Place Data

You can improve the performance and decrease the memory footprint of the generated code for your Stateflow charts, truth tables, and state transition tables by using in-place data. You create in-place

data when you use the same data name for a chart input and a chart output. When you generate code from the chart, the generated code treats the input and output data as a single in-place argument passed by reference. Using in-place data reduces the number of times that the generated code copies intermediate data, which results in more efficient code.

When input and output data have the same name, you can edit properties only for the input data. The properties for the output data are read-only.

See Also

More About

- “Set Data Properties” on page 10-5
- “Specify Type of Stateflow Data” on page 10-20
- “Manage Symbols in the Stateflow Editor” on page 25-14

Set Data Properties

When you create Stateflow charts in Simulink, you can modify data properties in the **Property Inspector** or the Model Explorer.

To use the **Property Inspector**:

1. In the **Modeling** tab, under **Design Data**, select **Symbols Pane** and **Property Inspector**.
2. In the **Symbols** pane, select the data object.
3. In the **Property Inspector**, edit the data properties.

To use the Model Explorer:

1. In the **Modeling** tab, under **Design Data**, select **Model Explorer**.
2. In the **Model Hierarchy** pane, select the parent of the data object.
3. In the **Contents** pane, select the data object.
4. In the **Dialog** pane, edit the data properties.

You can also modify these properties programmatically by using `Stateflow.Data` objects. For more information about the Stateflow programmatic interface, see “Overview of the Stateflow API”.

Properties vary according to the scope and type of the data object. For many data properties, you can enter expressions or parameter values. Using parameters to set properties for many data objects simplifies maintenance of your model because you can update multiple properties by changing a single parameter.

Stateflow Data Properties

You can set these data properties in:

- The **Properties** tab of the **Property Inspector**.
- The **General** tab of the Model Explorer.

Name

Name of the data object. For more information, see “Guidelines for Naming Stateflow Objects” on page 1-66.

Scope

Location where data resides in memory, relative to its parent.

Setting	Description
`Local`	Data defined in the current chart only.
`Constant`	Read-only constant value that is visible to the parent Stateflow object and its children.

Setting	Description
`Parameter`	Constant whose value is defined in the MATLAB base workspace or derived from a Simulink block parameter that you define and initialize in the parent masked subsystem. The Stateflow data object must have the same name as the MATLAB variable or the Simulink parameter. For more information, see "Share Parameters with Simulink and the MATLAB Workspace" on page 10-32.
`Input`	Input argument to a function if the parent is a graphical function, truth table, or MATLAB function. Otherwise, the Simulink model provides the data to the chart through an input port on the Stateflow block. For more information, see "Share Input and Output Data with Simulink" on page 10-30.
`Output`	Return value of a function if the parent is a graphical function, truth table, or MATLAB function. Otherwise, the chart provides the data to the Simulink model through an output port on the Stateflow block. For more information, see "Share Input and Output Data with Simulink" on page 10-30.
`Data Store Memory`	Data object that binds to a Simulink data store, which is a signal that functions like a global variable. All blocks in a model can access that signal. This binding allows the chart to read and write to the Simulink data store, sharing global data with the model. The Stateflow object must have the same name as the Simulink data store. For more information, see "Access Data Store Memory from a Chart" on page 10-34.
`Temporary`	Data that persists during only the execution of a function. You can define temporary data only for graphical functions, truth tables, or MATLAB functions in charts that use C as the action language.

Port

Index of the port associated with the data object. This property applies only to input and output data. See "Share Input and Output Data with Simulink" on page 10-30.

Update method

Specifies whether a variable updates in discrete or continuous time. This property applies only when the chart is configured for continuous-time simulation. See "Continuous-Time Modeling in Stateflow" on page 22-2.

Data must resolve to signal object

Specifies that output or local data explicitly inherits properties from `Simulink.Signal` objects of the same name in the MATLAB base workspace or the Simulink model workspace. The data can inherit these properties:

- Size
- Complexity
- Type
- Unit
- Minimum value

- Maximum value
- Initial value
- Storage class
- Sampling mode (for Truth Table block output data)

This option is available only when you set the model configuration parameter **Signal resolution** to a value other than `None`. For more information, see “Resolve Data Properties from Simulink Signal Objects” on page 10-43.

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 the field blank.
- To specify an *n*-by-1 column vector, set the **Size** property to *n*.
- To specify a 1-by-*n* row vector, set the **Size** property to `[1 n]`.
- To specify an *n*-by-*m* matrix, set the **Size** property to `[n m]`.
- To specify an *n*-dimensional array, set the **Size** property to $[d_1\ d_2\ \cdots\ d_n]$, where d_i is the size of the ith dimension.
- To configure a Stateflow data object to inherit its size from the corresponding Simulink signal or from its definition in the chart, specify a size of `–1`.

The scope of the data object determines what sizes you can specify. Stateflow data store memory inherits all its properties, including its size, from the Simulink data store to which it is bound. For all other scopes, size can be scalar, vector, or a matrix of n-dimensions. For more information, see “Specify Size of Stateflow Data” on page 10-26.

You can specify data size through a MATLAB expression that evaluates to a valid size specification. For more information, see “Specify Data Size by Using Expressions” on page 10-27 and “Specify Data Properties by Using MATLAB Expressions” on page 10-18.

Variable size

Specifies that the data object changes size during simulation. This option is available only when you enable the chart property **Support variable-size arrays**. For more information, see “Declare Variable-Size Data in Stateflow Charts” on page 19-9.

Complexity

Specifies whether the data object accepts complex values.

Setting	Description
`Off`	Data object does not accept complex values.
`On`	Data object accepts complex values.
`Inherited`	Data object inherits the complexity setting from a Simulink block.

The default value is `Off`. For more information, see “Complex Data in Stateflow Charts” on page 24-2.

First index

Index of the first element of the data array. The first index can be any integer. The default value is `0`. This property is available only for C charts.

Type

Type of data object. To specify the data type:

- From the **Type** drop-down list, select a built-in type.
- In the **Type** field, enter an expression that evaluates to a data type. Use one of these expressions:

  - A call to the `fixdt` function to create a `Simulink.NumericType` object that describes a fixed-point or floating-point data type. See “Specify Fixed-Point Data” on page 23-2.
  - A call to the `type` operator to specify the type of previously defined data. See “Derive Data Types from Other Data Objects” on page 10-23.
  - A `Simulink.AliasType` object that defines a data type alias in the MATLAB base workspace. See “Specify Data Types by Using a Simulink Alias” on page 10-24.

  For more information, see “Specify Data Properties by Using MATLAB Expressions” on page 10-18.

Additionally, in the Model Explorer, you can open the Data Type Assistant by clicking the **Show data type assistant** button ››. Specify a data **Mode**, and then specify the data type based on that mode. For more information, see “Specify Type of Stateflow Data” on page 10-20.

Note If you enter an expression for a fixed-point data type, you must specify scaling explicitly. For example, you cannot enter an incomplete specification such as `fixdt(1,16)` in the **Type** field. If you do not specify scaling explicitly, an error appears when you try to simulate your model.

Lock data type against Fixed-Point tools

Prevents replacement of the current fixed-point type with an autoscaled type chosen by the Fixed-Point Tool (Fixed-Point Designer). For more information, see “Iterative Fixed-Point Conversion Using the Fixed-Point Tool” (Fixed-Point Designer).

Unit (e.g., m, m/s^2, N*m)

Specifies physical units for input and output data. For more information, see “Specify Units for Stateflow Data” on page 10-29.

Initial value

Initial value of the data object. For constant data, this property is called **Constant value**. The options for specifying this property depend on the scope of the data object.

<table>
<tr><th>Scope</th><th>Specify for Initial Value</th></tr>
<tr><td><code>Local</code></td><td>Expression or parameter defined in the Stateflow hierarchy, MATLAB base workspace, or Simulink masked subsystem. To specify the initial value when you leave the Initial value field blank, open the Model Explorer or the Data properties dialog box and set the Initial value drop-down list to <code>Expression</code> or <code>Parameter</code>.
• <code>Expression</code> — Numeric data resolves to a default value of 0. For enumerated data, the default value typically is the first one listed in the <code>enumeration</code> section of the definition. You can specify a different default enumerated value in the <code>methods</code> section of the definition. For more information, see “Define Enumerated Data Types” on page 20-5.
• <code>Parameter</code> — The data object resolves to a variable in the base workspace with the same name.
The default setting is <code>Expression</code>.</td></tr>
<tr><td><code>Constant</code></td><td>Constant value or expression. The expression is evaluated when you update the chart. The resulting value is used as a constant for running the chart.
When you leave the Constant value field blank, numeric data resolves to a default value of 0. For enumerated data, the default value typically is the first one listed in the <code>enumeration</code> section of the definition. You can specify a different default enumerated value in the <code>methods</code> section of the definition. For more information, see “Define Enumerated Data Types” on page 20-5.</td></tr>
<tr><td><code>Parameter</code></td><td>You cannot enter a value. The chart inherits the initial value from the parameter.</td></tr>
<tr><td><code>Input</code></td><td>You cannot enter a value. The chart inherits the initial value from the Simulink input signal at the designated port.</td></tr>
<tr><td><code>Output</code></td><td>Expression or parameter defined in the Stateflow hierarchy, MATLAB base workspace, or Simulink masked subsystem. To specify the initial value when you leave the Initial value field blank, open the Model Explorer or the Data properties dialog box and set the Initial value drop-down list to <code>Expression</code> or <code>Parameter</code>.
• <code>Expression</code> — Numeric data resolves to a default value of 0. For enumerated data, the default value typically is the first one listed in the <code>enumeration</code> section of the definition. You can specify a different default enumerated value in the <code>methods</code> section of the definition. For more information, see “Define Enumerated Data Types” on page 20-5.
• <code>Parameter</code> — The data object resolves to a variable in the base workspace with the same name.
The default setting is <code>Expression</code>.</td></tr>
<tr><td><code>Data Store Memory</code></td><td>You cannot enter a value. The chart inherits the initial value from the Simulink data store to which it resolves.</td></tr>
</table>

The time of initialization depends on the data parent and scope of the Stateflow data object.

Data Parent	Scope	Initialization Time
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 entry
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

For more information on using an expression to specify an initial value, see “Specify Data Properties by Using MATLAB Expressions” on page 10-18.

Limit range

Range of acceptable values for this data object. Stateflow charts use this range to validate the data object during simulation.

- **Minimum** — The smallest value allowed for the data item during simulation. You can enter an expression or parameter that evaluates to a numeric scalar value.
- **Maximum** — The largest value allowed for the data item during simulation. You can enter an expression or parameter that evaluates to a numeric scalar value.

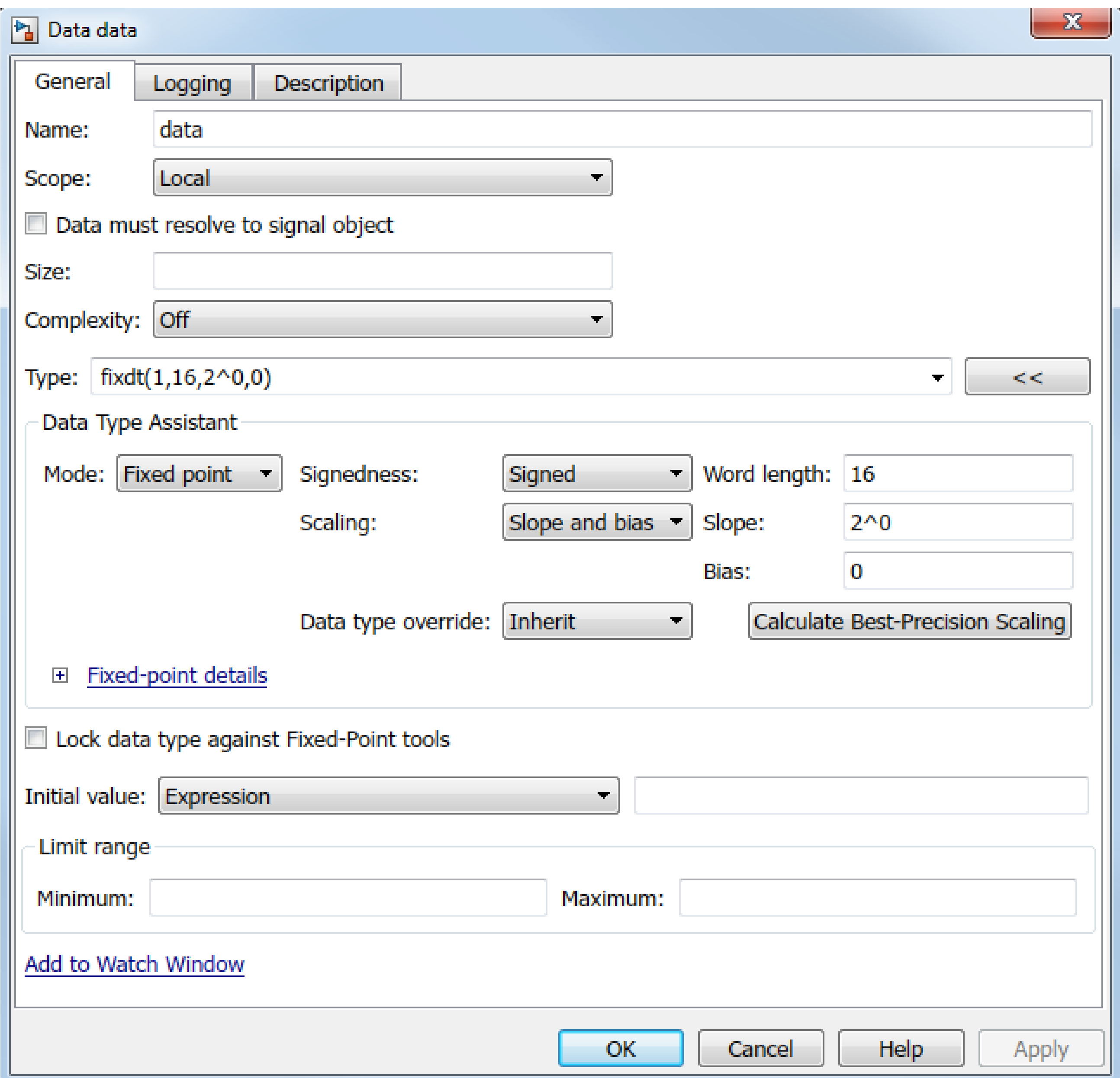
The smallest value that you can set for **Minimum** is `-inf`. The largest value that you can set for **Maximum** is `inf`.

You can specify the minimum and maximum values through a MATLAB expression. For more information, see “Specify Data Properties by Using MATLAB Expressions” on page 10-18.

Note A Simulink model uses the **Maximum** and **Minimum** properties to calculate best-precision scaling for fixed-point data types. Before you select **Calculate Best-Precision Scaling**, specify a minimum or maximum value. For more information, see “Calculate best-precision scaling” on page 10-13.

Fixed-Point Data Properties

In the Model Explorer, when you set the Data Type Assistant **Mode** to `Fixed point`, the Data Type Assistant displays fields for specifying additional information about your fixed-point data.



Signedness

Specifies whether the fixed-point data is `Signed` or `Unsigned`. Signed data can represent positive and negative values. Unsigned data represents positive values only. The default setting is `Signed`.

Word length

Specifies the bit size of the word that holds the quantized integer. Large word sizes represent large values with greater precision than small word sizes. The default value is 16.

- Word length can be any integer from 0 through 128 for chart-level data of these scopes:
  - `Input`

  - `Output`
  - `Parameter`
  - `Data Store Memory`
- For other Stateflow data, word length can be any integer from 0 through 32.

You can specify the word length through a MATLAB expression. For more information, see “Specify Data Properties by Using MATLAB Expressions” on page 10-18.

Scaling

Specifies the method for scaling your fixed-point data to avoid overflow conditions and minimize quantization errors. The default method is `Binary point` scaling.

Setting	Description
`Binary point`	If you select this mode, the Data Type Assistant displays the **Fraction length** field, which specifies the binary point location. **Fraction length** can be any integer. The default value is 0. A positive integer moves the binary point left of the rightmost bit by that amount. A negative integer moves the binary point farther right of the rightmost bit.
`Slope and bias`	If you select this mode, the Data Type Assistant displays fields for entering the **Slope** and **Bias** for the fixed-point encoding scheme. **Slope** can be any positive real number. The default value is 1.0. **Bias** can be any real number. The default value is 0.0. You can enter slope and bias as expressions that contain parameters you define in the MATLAB base workspace.

Whenever possible, use binary-point scaling to simplify the implementation of fixed-point data in generated code. Operations with fixed-point data that use binary-point scaling are performed with simple bit shifts and eliminate expensive code implementations required for separate slope and bias values. For more information about fixed-point scaling, see “Scaling” (Fixed-Point Designer).

You can specify **Fraction length**, **Slope**, and **Bias** through a MATLAB expression. For more information, see “Specify Data Properties by Using MATLAB Expressions” on page 10-18.

Data type override

Specifies whether to inherit the data type override setting of the Fixed-Point Tool that applies to this model. If the data does not inherit the model-wide setting, the specified data type applies.

Calculate best-precision scaling

Specifies whether to calculate the best-precision values for `Binary point` and `Slope and bias` scaling, based on the values in the **Minimum** and **Maximum** properties.

To calculate best-precision scaling values:

1. Specify **Maximum** and **Minimum** properties.
2. Click **Calculate Best-Precision Scaling**.

The best-precision scaling values are displayed in the **Fraction length** field or the **Slope** and **Bias** fields. For more information, see “Maximize Precision” (Fixed-Point Designer).

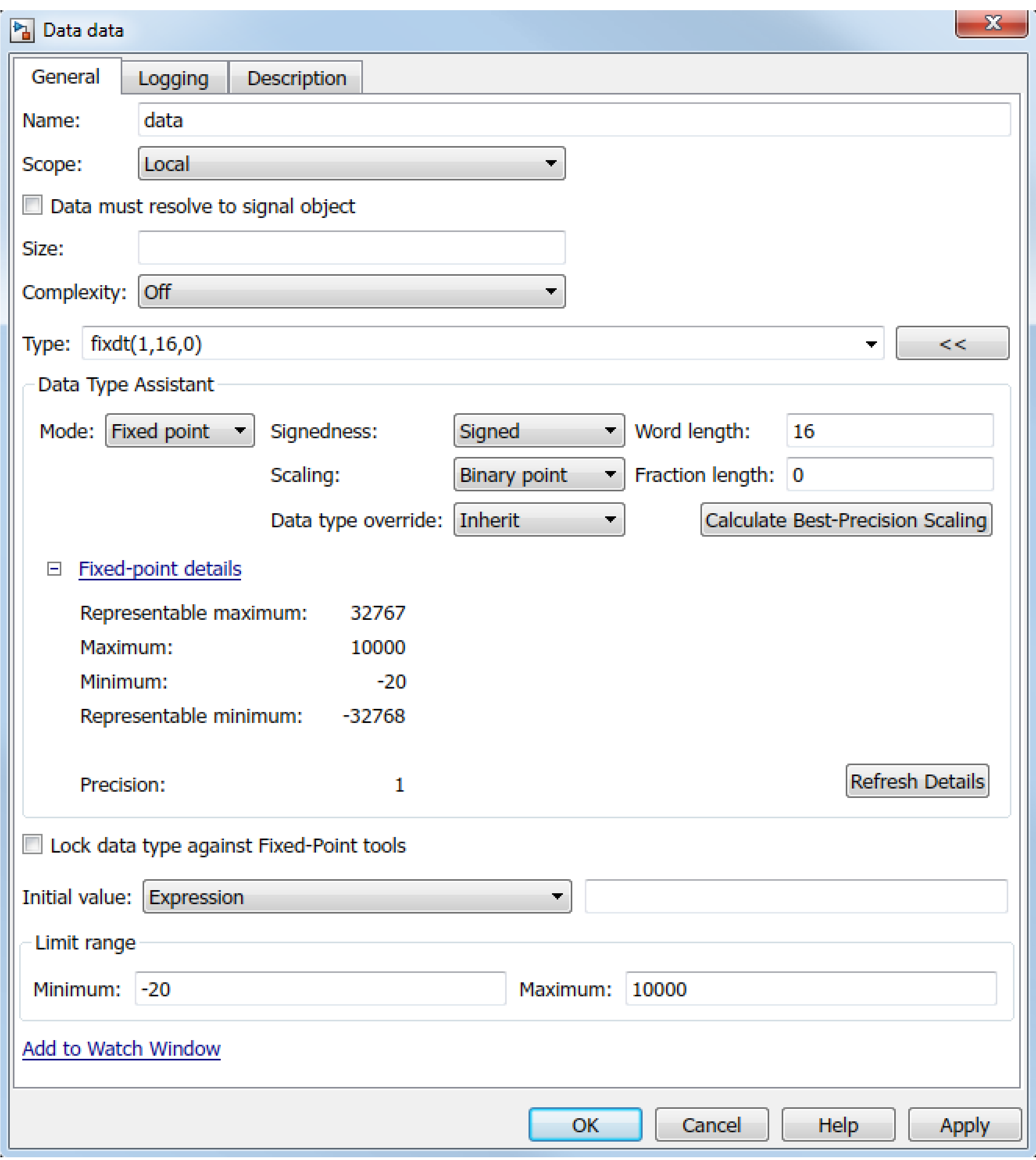
Note The **Maximum** and **Minimum** properties do not apply to `Constant` and `Parameter` scopes. For `Constant`, Simulink software calculates the scaling values based on the **Initial value** setting. The software cannot calculate best-precision scaling for data of `Parameter` scope.

Fixed-point details

Displays information about the fixed-point data type that is defined in the Data Type Assistant:

- `Minimum` and `Maximum` show the same values that you specify in the **Minimum** and **Maximum** properties.
- `Representable minimum`, `Representable maximum`, and `Precision` show the minimum value, maximum value, and precision that the fixed-point data type can represent.

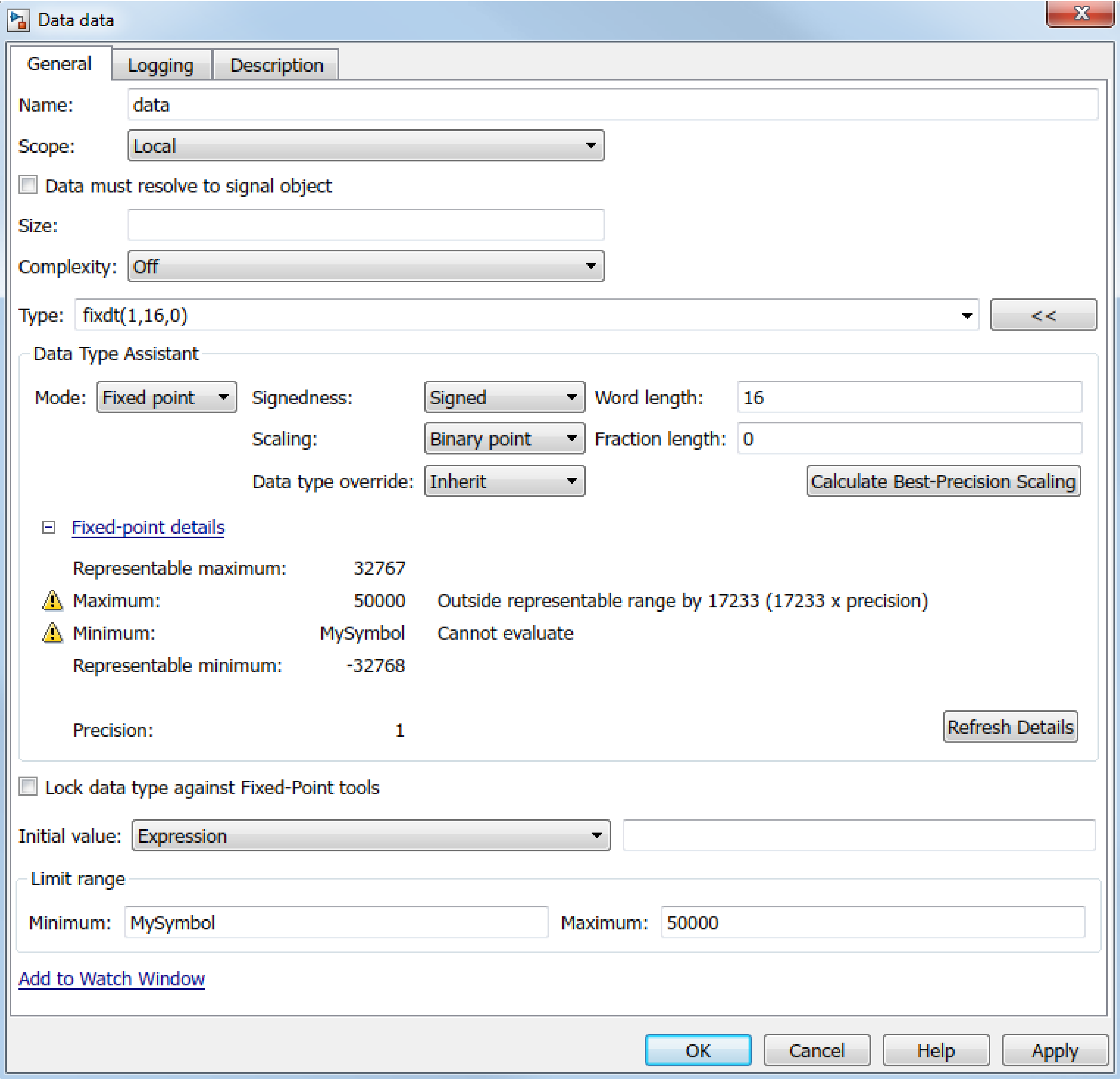
If the value of a field cannot be determined without first compiling the model, the **Fixed-point details** subpane shows the value as `Unknown`.



The values displayed by the **Fixed-point details** subpane *do not* automatically update if you change the values that define the fixed-point data type. To update the values shown in the **Fixed-point details** subpane, click **Refresh Details**.

Clicking **Refresh Details** does not modify the model. It changes only the display. To apply the displayed values, click **Apply** or **OK**.

The **Fixed-point details** subpane indicates any error resulting from the fixed-point data type specification. For example, this figure shows two errors.



The row labeled `Maximum` indicates that the value specified by the **Maximum** property is not representable by the fixed-point data type. To correct the error, make one of these modifications so the fixed-point data type can represent the maximum value:

- Decrease the value in the **Maximum** property.
- Increase **Word length**.
- Decrease **Fraction length**.

The row labeled `Minimum` shows the error `Cannot evaluate` because evaluating the expression `MySymbol`, specified by the **Minimum** property, does not return a numeric value. When an expression does not evaluate successfully, the **Fixed-point details** subpane shows the unevaluated expression (truncating to 10 characters as needed) in place of the unavailable value. To correct this error, define `MySymbol` in the base workspace to provide a numeric value. If you click **Refresh Details**, the error indicator and description are removed and the value of `MySymbol` appears in place of the unevaluated text.

Logging Properties

You can set logging properties for data in:

- The **Properties** tab of the **Property Inspector**.
- The **Logging** tab of the Model Explorer.

Log signal data

Whether to enable signal logging. Signal logging saves the values of the data object to the MATLAB workspace during simulation. For more information, see "Log Simulation Output for States and Data" on page 11-13.

Logging name

Signal name used to log the data object.

- To use the name of the data object, select `Use signal name` (default).
- To specify a different name, select `Custom` and enter the custom logging name.

Limit data points to last

Whether to limit the number of data points to log to the specified maximum. For example, if you set the maximum number of data points to 5000, the chart logs only the last 5000 data points generated by the simulation.

Decimation

Whether to limit the amount of logged data by skipping samples using the specified decimation interval. For example, if you set a decimation interval of 2, the chart logs every other sample.

Test point

Whether to set the data object as a test point that you can monitor with a floating scope during simulation. You can also log test point values to the MATLAB workspace. For more information, see "Monitor Test Points in Stateflow Charts" on page 11-38.

Additional Properties

You can set additional data properties in:

- The **Info** tab of the **Property Inspector**.
- The **Description** tab of the Model Explorer.

Save final value to base workspace

Assigns the value of the data object to a variable of the same name in the MATLAB base workspace at the end of simulation. This option is available only in the Model Explorer for charts that use C as the action language. For more information, see “Model Workspaces” (Simulink).

Units

Units of measurement associated with the data object. The unit in this field resides with the data object in the Stateflow hierarchy. This property is available only in the Model Explorer for C charts.

Description

Description of the data object.

Document link

Link to online documentation for the data object. You can enter a web URL address or a MATLAB command that displays documentation as an HTML file or as text in the MATLAB Command Window. When you click the **Document link** hyperlink, Stateflow evaluates the link and displays the documentation.

Default Data Property Values

When you leave a property field blank, Stateflow assumes a default value.

Property		Default Value
“Size” on page 10-7		`–1` (inherited), for inputs, parameters, and function outputs `1` (scalar), for other data objects
“First index” on page 10-8		0
“Initial value” on page 10-8		0.0
“Limit range” on page 10-10	**Minimum**	`-inf`
	Maximum	`inf`
“Fixed-Point Data Properties” on page 10-10	**Word length**	16
	Fraction length	0

Property		Default Value
	Slope	1.0
	Bias	0.0

Specify Data Properties by Using MATLAB Expressions

In the **Property Inspector** and Model Explorer, you can enter MATLAB expressions as values for these properties:

- “Size” on page 10-7
- “Type” on page 10-8
- “Initial value” on page 10-8
- “Limit range” on page 10-10: **Minimum** and **Maximum**
- “Fixed-Point Data Properties” on page 10-10: **Word length**, **Fraction length**, **Slope**, and **Bias**

Expressions can contain a mix of numeric values, constants, parameters, variables, arithmetic operations, parameters, constants, arithmetic operators, and calls to MATLAB functions. For example, you can use these functions to specify data properties.

Property	Function	Description
Size	`size`	Returns the size of a data object
Type	`type` on page 10-23	Returns the type of a data object
	`fixdt`	Returns a `Simulink.NumericType` object that describes a fixed-point or floating-point data type
	`fi`	Returns a fixed-point numeric object
Minimum	`min`	Returns the smallest element or elements of an array
Maximum	`max`	Returns the largest element or elements of an array

For more information, see “Specify Data Size by Using Expressions” on page 10-27 and “Derive Data Types from Other Data Objects” on page 10-23.

See Also

Objects
`Stateflow.Data` | `Simulink.AliasType` | `Simulink.NumericType`

Functions
`fi` | `fixdt` | `max` | `min` | `size`

Tools
Model Explorer

More About

- “Add Stateflow Data” on page 10-2

- “Guidelines for Naming Stateflow Objects” on page 1-66
- “Specify Type of Stateflow Data” on page 10-20
- “Specify Size of Stateflow Data” on page 10-26

Specify Type of Stateflow Data

The term data type refers to how computers represent information in memory. The data type determines the amount of storage allocated to data, the method of encoding a data value as a pattern of binary digits, and the operations available for manipulating the data.

Specify Data Type by Using the Data Type Assistant

You can specify the type of a data object in either the **Property Inspector** or the Model Explorer. In the **Type** field, select a type from the drop-down list or enter an expression that evaluates to a data type. For more information, see “Set Data Properties” on page 10-5.

Alternatively, use the Data Type Assistant to specify a data **Mode** and select the data type based on that mode:

1. In the Model Explorer, on the **Data** pane, click the **Show data type assistant** button ››.
2. Select a **Mode** from the drop-down list. The list of available modes depends on the scope of the data object.

Scope	Modes
`Local`	`Inherit` (available only in charts that use MATLAB as the action language), `Built in`, `Fixed point`, `Enumerated`, `Bus object`, `Expression`
`Constant`	`Built in`, `Fixed point`, `Expression`
`Parameter`	`Inherit`, `Built in`, `Fixed point`, `Enumerated`, `Bus object`, `Expression`
`Input`	`Inherit`, `Built in`, `Fixed point`, `Enumerated`, `Bus object`, `Expression`
`Output`	`Inherit`, `Built in`, `Fixed point`, `Enumerated`, `Bus object`, `Expression`
`Data Store Memory`	`Inherit`

3. Specify additional information based on the mode. The Data Type Assistant populates the **Type** field based on your specification.

<table>
<tr><th>Mode</th><th>Data Types</th></tr>
<tr><td><code>Inherit</code></td><td>You cannot specify a data type. You inherit the data type based on the scope that you select for the data object:

For charts that use MATLAB as the action language, if scope is <code>Local</code>, the data infers its type from the context of the MATLAB code in the chart.
If the scope is <code>Parameter</code>, the data inherits its type from the associated parameter, which you can define in the Simulink model or in the MATLAB base workspace. See “Share Parameters with Simulink and the MATLAB Workspace” on page 10-32.
If the scope is <code>Input</code>, the data inherits its type from the Simulink input signal on the designated input port. See “Share Input and Output Data with Simulink” on page 10-30.
If the scope is <code>Output</code>, the data inherits its type from the Simulink output signal on the designated output port. See “Share Input and Output Data with Simulink” on page 10-30.

Note Avoid inheriting data types from output signals. See “Avoid Inheriting Output Data Properties from Simulink Blocks” on page 10-3.
If the scope is <code>Data Store Memory</code>, the data inherits its type from the Simulink data store to which you bind the data object. See “Access Data Store Memory from a Chart” on page 10-34.

For more information, see “Inherit Data Types from Simulink Objects” on page 10-23.</td></tr>
</table>

Mode	Data Types
`Built in`	Specify a data type from the drop-down list of supported data types: • `double`: 64-bit double-precision floating point. • `single`: 32-bit single-precision floating point. • `half`: A half-precision data type occupies 16 bits of memory, but its floating-point representation enables it to handle wider dynamic ranges than integer or fixed-point data types of the same size. See "The Half-Precision Data Type in Simulink" (Fixed-Point Designer). • `int64`: 64-bit signed integer. • `int32`: 32-bit signed integer. • `int16`: 16-bit signed integer. • `int8`: 8-bit signed integer. • `uint64`: 64-bit unsigned 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`. Supported only in charts that use C as the action language. The `ml` data type provides Stateflow data with the benefits of the MATLAB environment, including the ability to assign the Stateflow data object to a MATLAB variable or pass it as an argument to a MATLAB function. `ml` data cannot have a scope outside the Stateflow hierarchy. That is, it cannot have a scope of `Input` or `Output`. For more information, see "ml Data Type" on page 14-23. • `string`: String. For more information, see "Manage Textual Information by Using Strings" on page 21-2.
`Fixed point`	Specify this information about the fixed-point data: • **Signedness**: Whether the data is signed or unsigned • **Word length**: Bit size of the word that holds the quantized integer. Large word sizes represent large values with greater precision than small word sizes. The default value is 16. • **Scaling**: Method for scaling your fixed-point data to avoid overflow conditions and minimize quantization errors. The default method is `Binary point`. For information, see "Fixed-Point Data Properties" on page 10-10.
`Enumerated`	Specify the class name for the enumerated data type. For more information, see "Define Enumerated Data Types" on page 20-5.
`Bus object`	Specify the name of a `Simulink.Bus` object to associate with the Stateflow bus object structure. Click **Edit** to create or edit a bus object in the Type Editor. You can also inherit bus object properties from Simulink signals.

Mode	Data Types
`Expression`	Specify an expression that evaluates to a data type. Use one of these expressions: • A call to the `fixdt` function to create a `Simulink.NumericType` object that describes a fixed-point or floating-point data type. See “Specify Fixed-Point Data” on page 23-2. • A call to the `type` operator to specify the type of previously defined data. See “Derive Data Types from Other Data Objects” on page 10-23. • A `Simulink.AliasType` object that defines a data type alias in the MATLAB base workspace. See “Specify Data Types by Using a Simulink Alias” on page 10-24. For more information, see “Specify Data Properties by Using MATLAB Expressions” on page 10-18.

4 To save the data type settings, click **Apply**.

The Data Type Assistant is available only through the Model Explorer.

Inherit Data Types from Simulink Objects

When you select `Inherit: Same as Simulink` from the **Type** drop-down list, data objects of scope `Input`, `Output`, `Parameter`, and `Data Store Memory` inherit their data types from Simulink objects.

Scope	Description
`Input`	Inherits type from the Simulink input signal connected to corresponding input port in chart.
`Output`	Inherits type from the Simulink output signal connected to corresponding output port in chart. Avoid inheriting data types from output signals. Values that back-propagate from Simulink blocks can be unpredictable.
`Parameter`	Inherits type from the corresponding MATLAB base workspace variable or Simulink parameter in a masked subsystem.
`Data Store Memory`	Inherits type from the corresponding Simulink data store.

To determine the data types that the objects inherit:

1 Build the Simulink model.

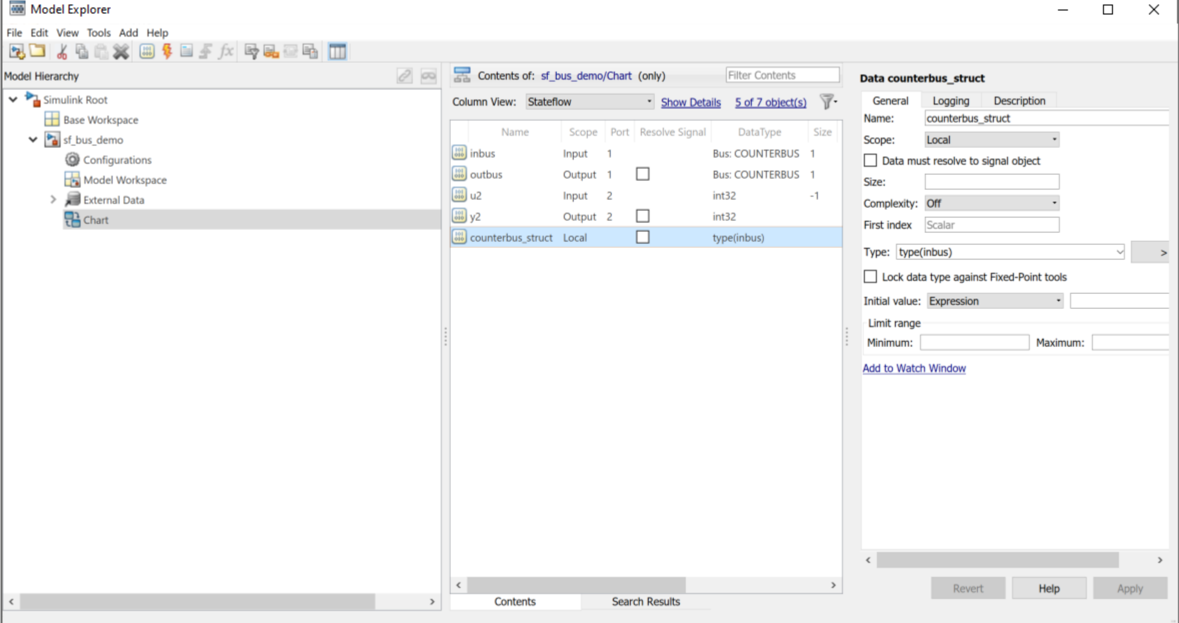
2 Open the Model Explorer.

3 In the **Contents** pane, examine the **CompiledType** column.

Derive Data Types from Other Data Objects

You can use the `type` operator to derive data types from other Stateflow data objects. For example, the model `sf_bus_demo` uses the data type of the input structure `inbus` to define the data type of the local structure `counterbus_struct` by using this expression:

```
type(inbus)
```



Because `inbus` derives its type from the `Simulink.Bus` object `COUNTERBUS`, `counterbus_struct` also derives its data type from `COUNTERBUS`. After you compile your model, the **CompiledType** column of the Model Explorer shows the type used in the compiled simulation application.

For more information about this example, see "Integrate Custom Structures in Stateflow Charts" on page 26-11.

Specify Data Types by Using a Simulink Alias

You can specify the type of Stateflow data by using a Simulink data type alias. For more information, see `Simulink.AliasType`.

For example, suppose that you want to define a data type alias `MyFloat` that corresponds to the built-in data type `single`. At the MATLAB command prompt, enter:

```
MyFloat = Simulink.AliasType;
MyFloat.BaseType = "single";
```

To use this alias to specify the type of a data object, select the object in the **Property Inspector** or the Model Explorer. In the **Type** field, enter the alias name `MyFloat`.

After you build your model, the **CompiledType** column of the Model Explorer shows the type used in the compiled simulation application.

Note Stateflow blocks do not support code generation if one of the data uses an alias type and is variable size. This limitation does not apply to chart-level input, output, or local data. For more information on defining variable-size data, see "Declare Variable-Size Data in Stateflow Charts" on page 19-9.

See Also

`fixdt` | `Simulink.AliasType` | `Simulink.NumericType`

More About

- “Set Data Properties” on page 10-5
- “About Data Types in Simulink” (Simulink)

Specify Size of Stateflow Data

In a Stateflow chart in a Simulink model, you specify the size of a data object by:

- Setting the **Size** property, as described in "Set Data Properties" on page 10-5. For more information, see "Size" on page 10-7.
- Setting the `Props.Array.Size` property through the Stateflow API. For more information, see `Stateflow.Data`.

Use one of these methods to specify the size:

- Inherit the size from a Simulink signal or from its definition in the Stateflow chart.
- Enter a numeric value.
- Enter a MATLAB expression.

Support for each 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	Only in charts that use MATLAB as the action language	Yes	Yes
Constant	No	Yes	Yes
Parameter	Yes	Yes	Yes
Input	Yes	Yes	Yes
Output	Yes	Yes	Yes
Data store memory	Yes	No	No

Inherit Data Size

To configure a Stateflow data object to inherit its size from the corresponding Simulink signal or its definition in the chart, specify a size of `–1`. After you simulate or build your model, you can find the inherited size of the data in the Model Explorer, under the **CompiledSize** column.

Note Charts cannot inherit data sizes from Simulink frame-based signals. For more information, see "Sample- and Frame-Based Concepts" (DSP System Toolbox).

Specify Data Size by Using Numeric Values

When you specify data size by entering a numeric value, follow these guidelines:

- To specify a scalar, enter `1` or leave the field blank.
- To specify an *n*-by-1 column vector, enter *n*.
- To specify a 1-by-*n* row vector, enter `[1 n]`.

- To specify an *n*-by-*m* matrix, enter `[n m]`, where *m* and *n* are greater than `1`.
- To specify an *n*-dimensional array, enter $[d_1\ d_2\ \cdots\ d_n]$, where d_i is the size of the ith dimension.

In charts that use C as the action language, one-dimensional Stateflow vectors are compatible with Simulink row or column vectors of the same size. For example, a Stateflow input data of size `3` is compatible with a Simulink row vector of size `[1 3]` or a column vector of size `[3 1]`.

Specify Data Size by Using Expressions

You can specify data size by entering a MATLAB expression that evaluates to one of the size specifications described in “Specify Data Size by Using Numeric Values” on page 10-26. These guidelines also apply:

- Expressions can contain a mix of numeric values, constants, parameters, variables, arithmetic operations, and calls to MATLAB functions.
- Expressions that specify the size of a dimension must evaluate to a positive integer value.
- Expressions can only combine compatible values. For example, integers can only be combined with other integers of the same type or with scalar doubles.
- 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. For more information, see “Notation for Enumerated Values” on page 20-3.
- You cannot use a MATLAB expression to:
  - Specify inherited data size. Do not use expressions that evaluate to `-1`.
  - Specify the size of Stateflow input data that accepts frame-based data from Simulink. For more information, see “Sample- and Frame-Based Concepts” (DSP System Toolbox).

Examples of Valid Data Size Expressions

These examples are valid MATLAB expressions for specifying data size in your chart:

- `K+3`, where `K` is a chart-level Stateflow constant or parameter.
- `N/2`, where `N` is a variable in the MATLAB base workspace.
- `[P Q]`, where `P` and `Q` are Simulink parameters. Charts that use C as the action language propagate these symbolic dimensions throughout the model. See “Propagate Symbolic Dimensions of Stateflow Data” on page 10-28.
- `2*Colors.Red`, where `Red` is an enumerated value of type `Colors`.
- `size(u)`, where `u` is a chart-level variable. The function `size` enables you to specify the size of one data object based on the size of another data object. This type of expression is useful in a library chart that you reuse with data of different sizes. In other situations, you can improve the clarity of your chart by avoiding the `size` function and specifying the size of the data directly.
- `floor((a*b)/c)`, where `a` and `c` are scalars of type `int16` and `b` is a scalar of type `double`.
- `[fi(2,1,16,2) fi(4,1,16,2)]`. This expression specifies a data size of `[2 4]` by calling the function `fi`. This function returns signed fixed-point numbers with a word length of 16 and a fraction length of 2.

Avoid Variables That Can Lead to Naming Conflicts

When a model contains multiple variables with identical names, the variable with the highest priority is used to specify size.

Priority	Variable
1	Mask parameter
2	Model workspace variable
3	MATLAB base workspace variable
4	Stateflow data

To avoid confusion, do not specify data size by using a variable name that you define in multiple levels of your model.

Propagate Symbolic Dimensions of Stateflow Data

When you select the model configuration parameter **Allow symbolic dimension specification**, charts that use C as the action language can propagate the symbolic dimensions of Stateflow data throughout the model. If you have Embedded Coder, the symbolic dimensions go into the generated code for ERT targets. Specify the size of the symbolic dimensions by using Simulink parameters with one of these storage classes:

- `Define` or `ImportedDefine` with a specified header file
- `CompilerFlag`
- A user-defined custom storage class that defines data as a macro in a specified header file

For more information, see “Allow symbolic dimension specification” (Simulink) and “Implement Symbolic Dimensions for Array Sizes in Generated Code” (Embedded Coder).

Stateflow charts that use MATLAB as the action language do not support symbolic dimension propagation. To specify data size by using Simulink parameters, clear the **Allow symbolic dimension specification** check box.

See Also

`fi` | `size`

More About

- “Vectors and Matrices in Stateflow Charts” on page 19-2
- “Stateflow Data Properties” on page 10-5
- “Reference Values by Name by Using Enumerated Data” on page 20-2

Specify Units for Stateflow Data

Units for Input and Output Data

Stateflow charts in Simulink models support the specification of physical units as properties for data inputs and outputs. Specify units by using the **Unit (e.g., m, m/s^2, N*m)** parameter for input or output data on charts, state transition tables, or truth tables. When you start typing in the field, this parameter provides matching suggestions for units that Simulink supports. By default, the property is set to inherit the unit from the Simulink signal on the corresponding input or output port. If you select the **Data must resolve to signal object** property for output data, you cannot specify units. In this case, output data is assigned the same unit type as the Simulink signal connected to the output port.

To display the units on the Simulink lines in the model, in the **Debug** tab, select **Information Overlays** > **Port Units**.

Consistency Checking

Stateflow checks the consistency of the signal line unit from Simulink with the unit setting for the corresponding input or output data in the Stateflow block. If the units do not match, Stateflow displays a warning during model update.

Units for Stateflow Limitations

The unit property settings do not affect the execution of the Stateflow block. Stateflow checks only consistency with the corresponding Simulink signal line connected to the input or output. It does not check consistency of assignments inside the Stateflow blocks. For example, Stateflow does not warn against an assignment of an input with unit set to `ft` to an output with unit set to `m`. Stateflow does not perform unit conversions.

See Also

More About

- “Unit Specification in Simulink Models” (Simulink)

Share Data with Simulink and the MATLAB Workspace

Stateflow charts interface with the other blocks in a Simulink model by:

- Sharing data through input and output connections.
- Importing initial data values from the MATLAB base workspace.
- Saving final data values to the MATLAB base workspace.

Charts also can access Simulink parameters and data stores. For more information, see “Share Parameters with Simulink and the MATLAB Workspace” on page 10-32 and “Access Data Store Memory from a Chart” on page 10-34.

Share Input and Output Data with Simulink

Data flows from Simulink into a Stateflow chart through input ports. Data flows from a Stateflow chart into Simulink through output ports.

To define input or output data in a chart:

1. Add a data object to the chart, as described in “Add Stateflow Data” on page 10-2.
2. Set the **Scope** property for the data object.

    - To define input data, set **Scope** to `Input Data`. An input port appears on the left side of the chart block.
    - To define output data, set **Scope** to `Output Data`. An output port appears on the right side of the chart block.

    By default, **Port** values appear in the order in which you add data objects. You can change these assignments by modifying the **Port** property of the data. When you change the **Port** property for an input or output data object, the **Port** values for the remaining input or output data objects automatically renumber.
3. Set the data type of the data object, as described in “Specify Type of Stateflow Data” on page 10-20.
4. Set the size of the data object, as described in “Specify Size of Stateflow Data” on page 10-26.

Note You cannot set the type or size of Stateflow input data to accept frame-based data from Simulink.

Initialize Data from the MATLAB Base Workspace

You can import the initial value of a data symbol by defining it in the MATLAB base workspace and in the Stateflow hierarchy.

1. Define and initialize a variable in the MATLAB base workspace.
2. In the Stateflow hierarchy, define a data object with the same name as the MATLAB variable.

3. Select the **Allow initial value to resolve to a parameter** property for the data object.

When the simulation starts, data resolution occurs. During this process, the Stateflow data object gets its initial value from the associated MATLAB variable.

One-dimensional Stateflow arrays are compatible with MATLAB row and column vectors of the same size. For example, a Stateflow vector of size `5` is compatible with a MATLAB row vector of size `[1,5]` or column vector of size `[5,1]`. Each element of the Stateflow array initializes to the same value as the corresponding element of the array in the MATLAB base workspace.

The time of initialization depends on the data parent and scope of the Stateflow data object.

Data Parent	Scope	Initialization Time
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 entry
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

Save Data to the MATLAB Base Workspace

At the end of simulation, a Stateflow chart that uses C as the action language can save the final value of a data object to the MATLAB base workspace.

1. Open the Model Explorer. In the **Modeling** tab, select **Model Explorer**.
2. Double-click the data object in the **Contents** pane.
3. In the **Description** pane of the Data properties dialog box, select **Save final value to base workspace**.

This option is available for data symbols of all scopes except `Constant` and `Parameter`.

See Also

More About

- “Add Stateflow Data” on page 10-2
- “Set Data Properties” on page 10-5
- “Share Parameters with Simulink and the MATLAB Workspace” on page 10-32
- “Access Data Store Memory from a Chart” on page 10-34

Share Parameters with Simulink and the MATLAB Workspace

A parameter is a constant data object that you can:

- Define in the MATLAB base workspace.
- Derive from a Simulink block parameter that you define and initialize in a mask.

Use parameters to avoid hard-coding data values and properties. Share Simulink parameters with charts to maintain consistency with your Simulink model.

You can access parameter values in multiple Stateflow objects in a chart such as states, MATLAB functions, and truth tables. You can include parameters in expressions defining data properties such as:

- Size
- Type
- Initial Value
- Minimum and Maximum
- Fixed-Point Data Properties

For more information, see “Specify Data Properties by Using MATLAB Expressions” on page 10-18

Initialize Parameters from the MATLAB Base Workspace

You can initialize a parameter by defining it in the MATLAB base workspace and in the Stateflow hierarchy.

1. Define and initialize a variable in the MATLAB base workspace.
2. In the Stateflow hierarchy, define a data object with the same name as the MATLAB variable.
3. Set the scope of the Stateflow data object to **Parameter**.

When the simulation starts, data resolution occurs. During this process, the Stateflow parameter gets its value from the associated MATLAB variable.

Share Simulink Parameters with Charts

You can share a parameter from a Simulink subsystem containing a Stateflow chart by creating a mask for the subsystem.

1. In the Simulink mask editor for the parent subsystem, define and initialize a Simulink parameter.
2. In the Stateflow hierarchy, define a data object with the same name as the Simulink parameter.
3. Set the scope of the Stateflow data object to **Parameter**.

When the simulation starts, Simulink tries to resolve the Stateflow data object to a parameter at the lowest-level masked subsystem. If unsuccessful, Simulink moves up the model hierarchy to resolve the data object to a parameter at higher-level masked subsystems.

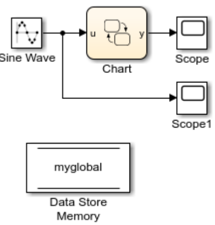
See Also

More About

- "Create a Mask to Share Parameters with Simulink" on page 25-36
- "Add Stateflow Data" on page 10-2
- "Set Data Properties" on page 10-5

Access Data Store Memory from a Chart

A Simulink model implements global variables as data stores, either as Data Store Memory blocks or as instances of `Simulink.Signal` objects. You can use data stores to share data between multiple Simulink blocks without explicit input or output connections to pass data from one block to another. Stateflow charts share global data with Simulink models by reading from and writing to data store memory symbolically.



To access global data from a chart, bind a Stateflow data object to a Simulink data store. After you create the binding, the Stateflow data object becomes a symbolic representation of the Simulink data store memory. You can then use this symbolic object to store and retrieve global data. Stateflow can access data stores in Simulink that have unbounded dimensions.

Local and Global Data Store Memory

Stateflow charts can interface with local and global data stores.

- Local data stores are visible to all blocks in one model. To interact with a local data store, a chart must reside in the model where you define the local data store. You can define a local data store by adding a Data Store Memory block to a model or by creating a Simulink signal object.
- Global data stores have a broader scope that crosses model reference boundaries. To interact with global data stores, a chart must reside in the top model where you define the global data store or in a model that the top model references. You implement global data stores as Simulink signal objects.

For more information, see “Local and Global Data Stores” (Simulink).

Bind Stateflow Data to Data Stores

1. To define the Simulink data store memory, add a Data Store Memory block to your model or create a Simulink signal object. For more information, see “Data Stores with Data Store Memory Blocks” (Simulink) and “Data Stores with Signal Objects” (Simulink).
2. Add a data object to the Stateflow chart, as described in “Add Stateflow Data” on page 10-2.
3. Set the **Name** property as the name of the Simulink data store memory to which you want to bind the Stateflow data object.
4. Set the **Scope** property to `Data Store Memory`.

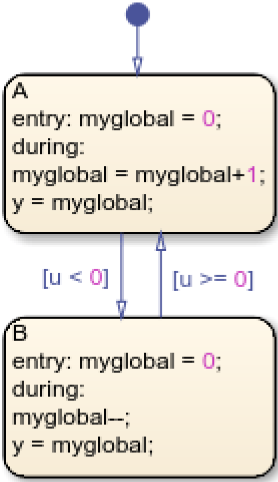
The Stateflow data object inherits all additional properties from the data store memory to which you bind the object.

Multiple local and global data stores with the same name can exist in the same model hierarchy. In this situation, the Stateflow data object binds to the data store that is the nearest ancestor.

Store and Retrieve Global Data

After binding a Stateflow data object to a Simulink data store, you can store and retrieve global data in state and transition actions. The data object acts as a global variable that you reference by its symbolic name. When you store numeric values in this variable, you are writing to the Simulink data store memory. When you retrieve numeric values from this variable, you are reading from the data store memory.

For example, in this chart, the state actions read from and write to a Data Store Memory block called `myglobal`.



Best Practices for Using Data Stores

Data Store Properties in Charts

When you bind a Stateflow data object to a data store, the Stateflow object inherits all of its properties from the data store. To ensure that properties propagate correctly, when you create the Simulink data stores:

- Specify a data type other than `auto`.
- Minimize the use of automatic-mode properties.

Share Data Store Memory Across Multiple Models

To access a global data store from multiple models:

- Verify that your models do not contain any Data Store Memory blocks. You can include Data Store Read and Data Store Write blocks.
- In the MATLAB base workspace, create a `Simulink.Signal` object with these attributes:
  - Set **Data type** to an explicit data type. The data type cannot be `Auto`.
  - Fully specify **Dimensions**. The signal dimensions cannot be –1 or `Inherited`.
  - Fully specify **Complexity**. The complexity cannot be `Auto`.

  - Set **Storage class** to `ExportedGlobal`.
- In each chart that shares the data, bind a Stateflow data object to the Simulink data store.

Write to Data Store Memory Before Reading

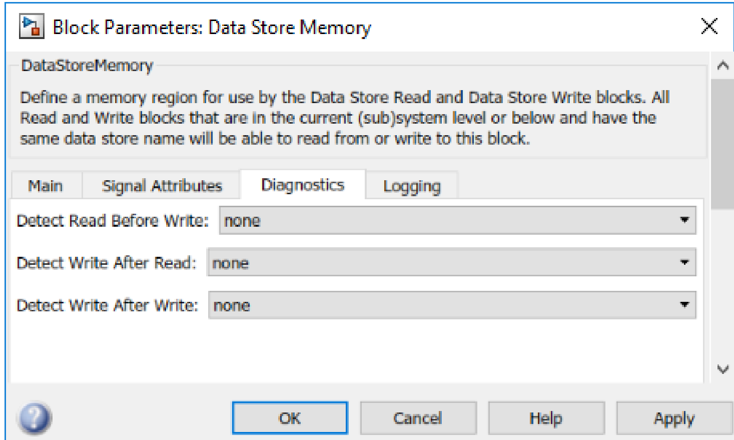
To avoid algorithm latency, write to data store memory before reading from it. Otherwise, the read actions retrieve the value that was stored in the previous time step, rather than the value computed and stored in the current time step. When unconnected blocks share global data while running at different rates:

- Segregate read actions into separate blocks from write actions.
- Assign priorities to blocks so that your model invokes write blocks before read blocks. For more information, see “Control and Display Execution Order” (Simulink).

To avoid situations when multiple reads and writes occur unintentionally in the same time step, enable the Data Store Memory block diagnostics to:

- **Detect Read Before Write**
- **Detect Write After Read**
- **Detect Write After Write**

If you use a data store memory block as a persistent global storage area for accumulating values across time steps, avoid unnecessary warnings by disabling the Data Store Memory block diagnostics. For more information, see “Data Store Diagnostics” (Simulink).



See Also

`Simulink.Signal` | Data Store Memory | Data Store Read | Data Store Write

More About

- “Add Stateflow Data” on page 10-2
- “Data Store Basics” (Simulink)
- “Model Global Data by Creating Data Stores” (Simulink)
- “Data Store Diagnostics” (Simulink)
- “Control and Display Execution Order” (Simulink)

Handle Integer Overflow for Chart Data

When Integer Overflow Can Occur

For some arithmetic operations, a processor may need to take an *n*-bit fixed-point value and store it in *m* bits, where $m \neq n$. If $m < n$, the reduced range of the value can cause an overflow for an arithmetic operation. Some processors identify this overflow as `Inf` or `NaN`. Other processors, especially digital signal processors (DSPs), handle overflows by saturating or wrapping the value.

For more information about saturation and wrapping for integer overflow, see "Saturation and Wrapping" (Fixed-Point Designer).

Support for Handling Integer Overflow in Charts

For Stateflow charts in Simulink models, you can control whether or not saturation occurs for integer overflow. To control overflow handling, set the **Saturate on integer overflow** chart property, as described in "Specify Properties for Stateflow Charts" on page 1-19.

Chart Property Setting	When to Use This Setting	Overflow Handling	Example of the Result
Selected	Overflow is possible for data in your chart and you want explicit saturation protection in the generated code.	Overflows saturate to either the minimum or maximum value that the data type can represent.	An overflow associated with a signed 8-bit integer saturates to –128 or +127 in the generated code.
Cleared	You want to optimize efficiency of the generated code.	The handling of overflows depends on the C compiler that you use for generating code.	The number 130 does not fit in a signed 8-bit integer and wraps to –126 in the generated code.

Arithmetic operations for which you can enable saturation protection are:

- Unary minus: `–a`
- Binary operations: `a + b`, `a – b`, `a * b`, `a / b`, `a ^ b`
- Assignment operations: `a += b`, `a –=b`, `a *= b`, `a /= b`
- In C charts, increment and decrement operations: `++`, `--`

When you select **Saturate on integer overflow**, be aware that:

- Saturation applies to all intermediate operations, not just the output or final result.
- The code generator can detect some cases when overflow is not possible. In these cases, the generated code does not include saturation protection.

To determine whether clearing the **Saturate on integer overflow** check box is a safe option, perform a careful analysis of your logic, including simulation if necessary. If saturation is necessary in only some sections of the logic, encapsulate that logic in atomic subcharts or MATLAB functions and define a different set of saturation settings for those units.

Effect of Integer Promotion Rules on Saturation

Charts use ANSI® C rules for integer promotion.

- All arithmetic operations use a data type that has the same word length as the target word size. Therefore, the intermediate data type in a chained arithmetic operation can be different from the data type of the operands or the final result.
- For operands with integer types smaller than the target word size, promotion to a larger type of the same word length as the target size occurs. This implicit cast occurs before any arithmetic operations take place.

  For example, when the target word size is 32 bits, an implicit cast to `int32` occurs for operands with a type of `uint8`, `uint16`, `int8`, or `int16` before any arithmetic operations occur.

Suppose that you have the following expression, where `y`, `u1`, `u2`, and `u3` are of `uint8` type:

```
y = (u1 + u2) - u3;
```

Based on integer promotion rules, that expression is equivalent to the following statements:

```
uint8_T u1, u2, u3, y;
int32_T tmp, result;
tmp = (int32_T) u1 + (int32_T) u2;
result = tmp - (int32_T) u3;
y = (uint8_T) result;
```

For each calculation, the following data types and saturation limits apply.

Calculation	Data Type	Saturation Limits
tmp	int32	(MIN_INT32, MAX_INT32)
result	int32	(MIN_INT32, MAX_INT32)
y	uint8	(MIN_UINT8, MAX_UINT8)

Suppose that `u1`, `u2`, and `u3` are equal to 200. Because the saturation limits depend on the intermediate data types and not the operand types, you get the following values:

- `tmp` is 400.
- `result` is 200.
- `y` is 200.

Impact of Saturation on Error Checks

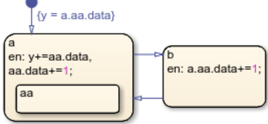
If you enable the chart property **Saturate on integer overflow** and set the configuration parameter **Wrap on overflow** to `error` or `warning`, the Stateflow chart does not flag cases of integer overflow during simulation. However, the chart continues to flag division-by-zero operations and out-of-range data violations based on minimum and maximum range checks.

Identify Data by Using Dot Notation

To specify the path from the parent state to a data object, a qualified data name uses dot notation. Dot notation is a way to identify data at a specific level of the Stateflow chart hierarchy. The first part of a qualified data name identifies the parent object. Subsequent parts identify the children along a hierarchical path.

For example, in this chart, the symbol `data` resides in the substate `aa` of the state `a`. The state and transition actions use qualified data names to refer to this symbol.

- In the default transition, the action uses the qualified data name `a.aa.data` to specify a path from the chart to the top-level state `a`, to the substate `aa`, and finally to `data`.
- In state `a`, the entry action uses the qualified data name `aa.data` to specify a path from the substate `aa` to `data`.
- In state `b`, the entry action uses the qualified data name `a.aa.data` to specify a path from the chart to the state `a`, to the substate `aa`, and then to `data`.



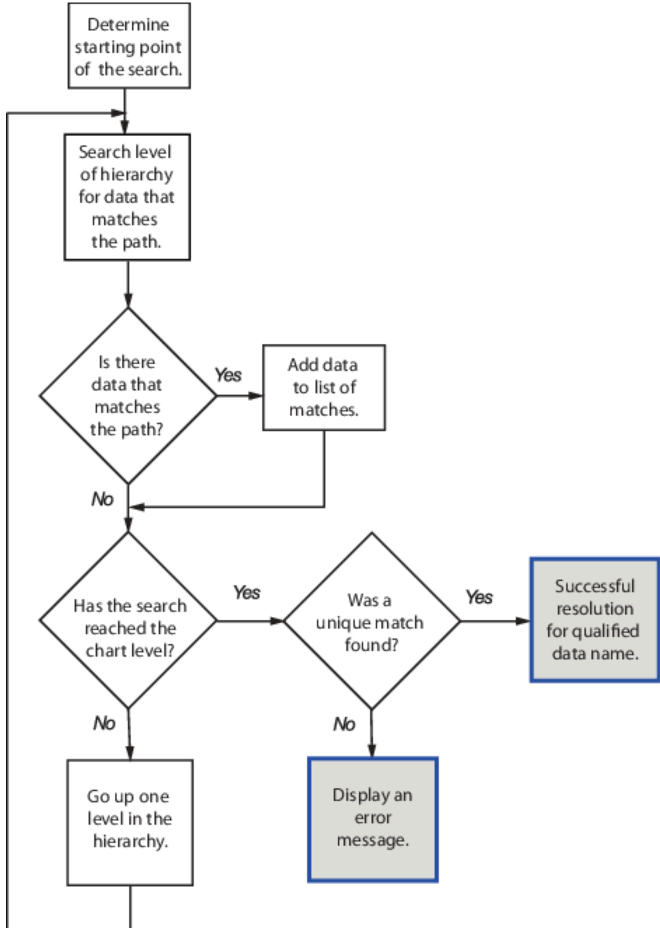
Resolution of Qualified Data Names

During simulation, Stateflow resolves the qualified data name by performing a localized search of the chart hierarchy for a matching data object. The search begins at the hierarchy level where the qualified data name appears:

- For a state action, the starting point is the state containing the action.
- For a transition label, the starting point is the parent of the transition source.

The resolution process searches each level of the chart hierarchy for a path to the data. If a data object matches the path, the process adds that data object to the list of possible matches. Then, the process continues the search one level higher in the hierarchy. The resolution process stops after it searches the chart level of the hierarchy. If a unique match exists, the qualified data name resolves to the matching path. Otherwise, the resolution process fails. Simulation stops, and you see an error message.

This flow chart illustrates the different stages in the process for resolving qualified data names.



Best Practices for Using Dot Notation

Resolving qualified data names:

- Does not perform an exhaustive search of all data.
- Does not stop after finding the first match.

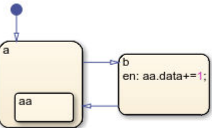
To improve the chances of finding a unique search result when resolving qualified data names:

- Use specific paths in qualified data names.
- Give states unique names.
- Use states and boxes as enclosures to limit the scope of the path resolution search.

Examples of Qualified Data Name Resolution

Search Produces No Matches

In this chart, the entry action in state `b` contains the qualified data name `aa.data`. If the symbol `data` resides in state `aa`, then Stateflow cannot resolve the qualified data name.



This table lists the different stages in the resolution process for the qualified data name `aa.data`.

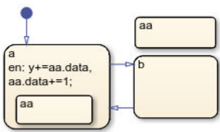
Stage	Description	Result
1	Starting in state `b`, search for an object `aa` that contains `data`.	No match found.
2	Move up to the next level of the hierarchy (the chart level). Search for an object `aa` that contains `data`.	No match found.

The search ends at the chart level with no match found for `aa.data`, resulting in an error.

To avoid this error, in the entry action of state `b`, specify the data with the more specific qualified data name `a.aa.data`.

Search Produces Multiple Matches

In this chart, the entry action in state `a` contains two instances of the qualified data name `aa.data`. If both states named `aa` contain a data object named `data`, then Stateflow cannot resolve the qualified data name.



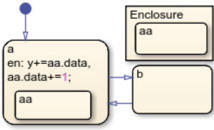
This table lists the different stages in the resolution process for the qualified data name `aa.data`.

Stage	Description	Result
1	Starting in state `a`, search for an object `aa` that contains `data`.	Match found.
2	Move up to the next level of the hierarchy (the chart level). Search for an object `aa` that contains `data`.	Match found.

The search ends at the chart level with two matches found for `aa.data`, resulting in an error.

To avoid this error:

- Use a more specific qualified data name. For instance:
  - To specify the data object in the substate of state `a`, use the qualified data name `a.aa.data`.
  - To specify the data object in the top-level state `aa`, use the qualified data name `/aa.data`.
- Rename one of the states containing `data`.
- Enclose the top-level state `aa` in a box or in another state. Adding an enclosure prevents the search process from detecting data in the top-level state.



See Also

More About

- “Represent Operating Modes by Using States” on page 1-26
- “Transition Between Operating Modes” on page 1-37
- “Use State Hierarchy to Design Multilevel State Complexity” on page 1-33

Resolve Data Properties from Simulink Signal Objects

Stateflow® local and output data in charts can explicitly inherit properties from `Simulink.Signal` objects in the model workspace or base workspace. This process is called signal resolution and requires that the resolved signal have the same name as the chart output or local data.

For information about Simulink® signal resolution, see “Symbol Resolution” (Simulink) and “Symbol Resolution Process” (Simulink).

Inherited Properties

When Stateflow local or output data resolve to Simulink signal objects, they inherit these properties:

- Size
- Complexity
- Type
- Minimum value
- Maximum value
- Initial value
- Storage class

Storage class controls the appearance of chart data in the generated code. See “Organize Parameter Data into a Structure by Using Struct Storage Class” (Embedded Coder).

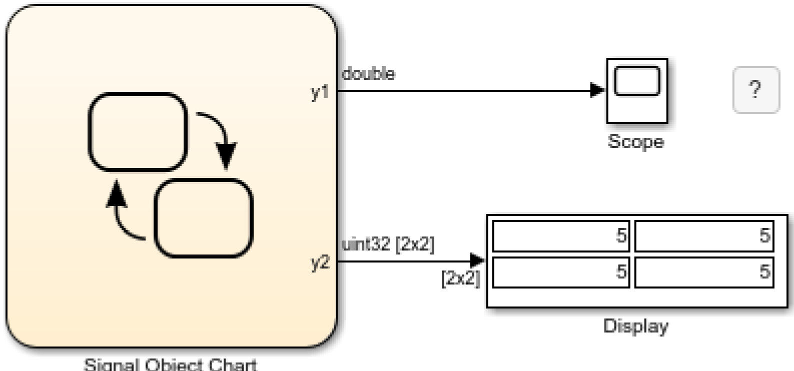
Enable Signal Resolution

To enable explicit signal resolution:

1. Open the Configuration Parameters dialog box and, in the **Diagnostics** > **Data Validity** pane, set **Signal resolution** to a value other than `None`. For more information about the other options, see “Signal resolution” (Simulink).
2. In the model workspace, base workspace, or data dictionary, define a `Simulink.Signal` object with the properties you want your Stateflow data to inherit. For more information about creating Simulink signals, see `Simulink.Signal` (Simulink).
3. Add chart output or local data. Use the same name for your data as for the `Simulink.Signal` object.
4. Enable the **Data must resolve to signal object** property, as described in “Set Data Properties” on page 10-5. After you select this property, the dialog box removes or dims the properties that your data inherits from the signal.

A Simple Example

This model shows how a chart resolves local and output data to `Simulink.Signal` objects.

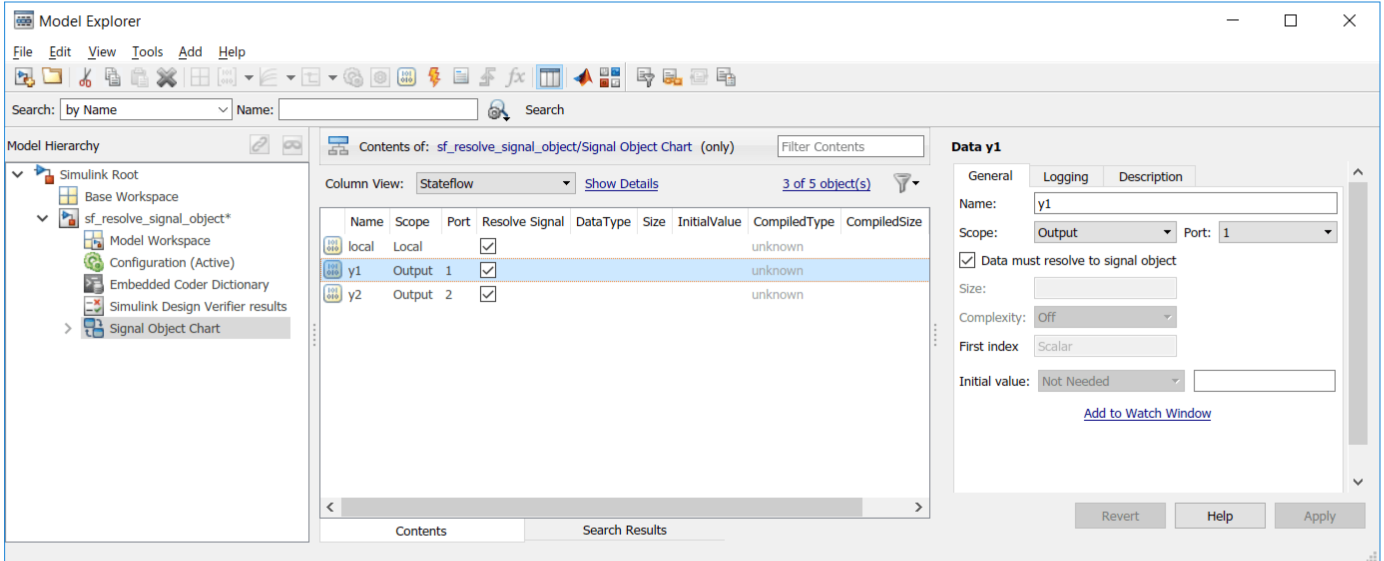


Copyright 2007-2021 The MathWorks, Inc.

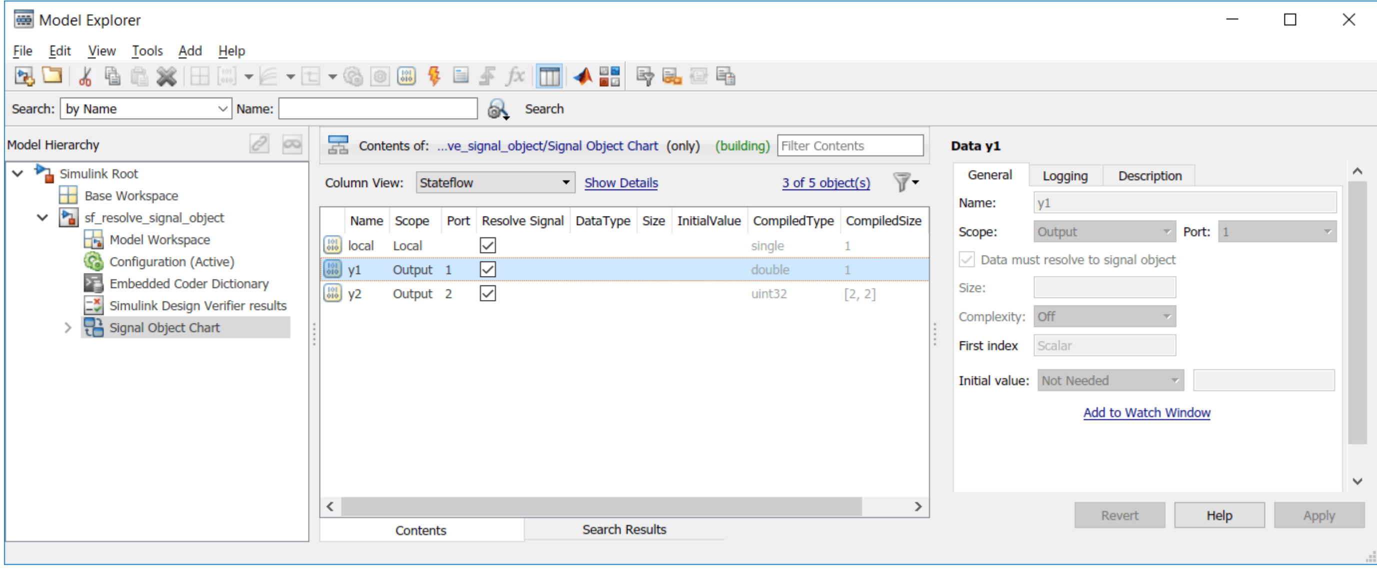
In the base workspace, there are three `Simulink.Signal` objects, each with a different set of properties.

- `y1` has these properties: **Type** = `double`, **Dimensions** = `1`, and **Storage Class** = `Model default`.
- `y2` has these properties: **Type** = `uint32`, **Dimensions** = `[2 2]`, and **Storage Class** = `Auto`.
- `local` has these properties: **Type** = `single`, **Dimensions** = `1`, and **Storage Class** = `ExportedGlobal`.

The chart contains three data objects — two outputs and a local variable — that resolve to a signal with the same name.



When you build the model, each data object inherits the properties of the identically named signal.



The generated code declares the data based on the storage class that the data inherits from the associated Simulink signal. For example, the header file below declares local to be an exported global variable:

```
/*
 * Exported States
 *
 * Note: Exported states are block states with an exported global
 * storage class designation.  Code generation will declare the memory for these
 * states and exports their symbols.
 *
 */
extern real32_T local;                  /* '<Root>/Signal Object Chart' */
```

See Also

`Simulink.Signal`

More About

- “Symbol Resolution” (Simulink)
- “Organize Parameter Data into a Structure by Using Struct Storage Class” (Embedded Coder)

Active State Data

- “Monitor State Activity Through Active State Data” on page 11-2
- “View State Activity by Using the Simulation Data Inspector” on page 11-7
- “View Stateflow States in the Logic Analyzer” on page 11-10
- “Log Simulation Output for States and Data” on page 11-13
- “Log Data in Library Charts” on page 11-20
- “Check State Activity by Using the in Operator” on page 11-24
- “Simplify Stateflow Charts by Incorporating Active State Output” on page 11-30
- “Model An Intersection Of One-Way Streets” on page 11-35
- “Monitor Test Points in Stateflow Charts” on page 11-38

Monitor State Activity Through Active State Data

Active state data can simplify the design of some Stateflow charts because you do not have to maintain data that is highly correlated to the chart hierarchy. When you enable active state data, Stateflow reports state activity through an output port to Simulink or as local data in your chart. Using active state data, you can:

- Avoid manual data updates reflecting chart activity.
- View chart activity by using a scope, the Simulation Data Inspector, or the Logic Analyzer.
- Log chart activity for diagnostics.
- Drive other Simulink subsystems.

Types of Active State Data

When you enable active state data, Stateflow creates a Boolean or enumerated data object to match the activity type.

Activity Type	Active State Data Type	Description
Self activity	Boolean	Is the state active?
Child activity	Enumeration	Which child is active?
Leaf state activity	Enumeration	Which leaf state is active?

For self-activity of a chart or state, the data value is `true` when active and `false` when inactive. For child and leaf state activity, the data is an enumerated type. Stateflow can define the enumeration class or you can create the definition manually. For more information, see "Define State Activity Enumeration Type" on page 11-4.

You can enable active state data for a Stateflow chart, state, state transition table, or atomic subchart. This table lists the activity types supported by each kind of Stateflow object.

Stateflow Object	Self-Activity	Child Activity	Leaf State Activity
Charts	Not supported	Supported	Supported
States with exclusive (OR) decomposition	Supported	Supported	Supported
States with parallel (AND) decomposition	Supported	Not supported	Not supported
Atomic subcharts	Supported at the container level	Supported inside the subchart	Supported inside the subchart
State transition tables	Not supported	Supported	Supported

Enable Active State Data

You can enable active state data in the **Property Inspector**, the Model Explorer, or the properties dialog box for the Stateflow object to monitor.

To use the **Property Inspector**:

1. In the **Modeling** tab, under **Design Data**, select **Property Inspector**.
2. In the Stateflow Editor, select the Stateflow object to monitor.
3. In the **Monitoring** section of the **Property Inspector**, select **Create output for monitoring** and edit the active state data properties listed below.

To use the Model Explorer:

1. In the **Modeling** tab, under **Design Data**, select **Model Explorer**.
2. In the **Model Hierarchy** pane, double-click the Stateflow object to monitor.
3. In the **Dialog** pane, select **Create output for monitoring** and edit the active state data properties listed below.

To use the properties dialog box:

1. In the Stateflow Editor, right-click the Stateflow object to monitor.
2. Select **Properties**.
3. In the properties dialog box, select **Create output for monitoring** and edit the active state data properties listed below.

Activity Type

Type of state activity to monitor. Choose from these options:

- `Self activity`
- `Child activity`
- `Leaf state activity`

Data Name

Name of the active state data object. For more information, see “Guidelines for Naming Stateflow Objects” on page 1-66.

Enum Name

Name of the enumerated data type for the active state data object. This property applies only to child and leaf state activity.

Define Enumerated Type Manually

Specifies whether you define the enumerated data type manually. This property applies only to child and leaf state activity. For more information, see “Define State Activity Enumeration Type” on page 11-4.

Set Scope for Active State Data

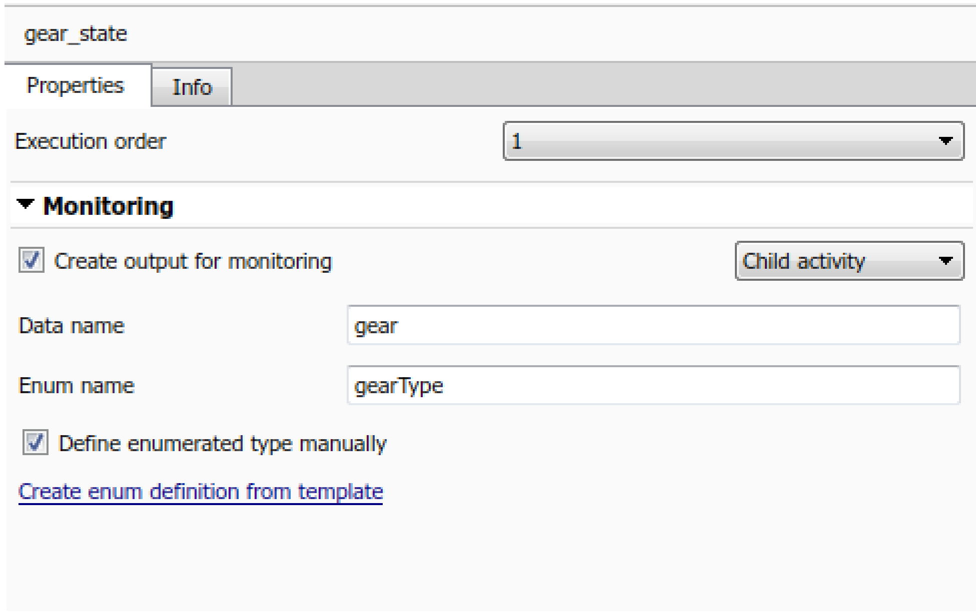
By default, active state data has a scope of `Output`. Stateflow creates an output port on the chart block in the Simulink model.

To access active state data inside a Stateflow chart, in the **Symbols** pane or in the Model Explorer, change the scope to `Local`. For more information, see “Set Data Properties” on page 10-5.

You can specify information for code generation by binding the local active state data to a `Simulink.Signal` object. Modify the properties of the object through the `CoderInfo` property. For more information, see `Simulink.CoderInfo`.

Define State Activity Enumeration Type

By default, Stateflow defines the enumeration data type for child and leaf activity. If you select the **Define enumerated type manually** check box and no enumeration data type definition exists, then Stateflow provides a link to create a definition. Clicking the **Create enum definition from template** link generates a customizable definition.



The enumeration data type definition contains one enumerated value for each state name. By default, the definition contains an enumerated value that serves as the default value for the enumeration data type and indicates that no substate is active. For example, in the model `sf_car`, the state `gear_state` contains four child states that correspond to the gears in a car: `first`, `second`, `third`, `fourth`. To open this example, enter:

```
openExample("stateflow/AutomaticTransmissionWithActiveStateDataExample")
```

The model specifies the child activity data type with this enumeration class definition:

```
classdef gearType < Simulink.IntEnumType
   enumeration
        None(0),
        first(1),
        second(2),
        third(3),
        fourth(4)
   end
...
end
```

You can customize this definition by reordering the enumerated values or renaming the default value. Do not rename the enumerated values that correspond to states or use the `getDefaultValue`

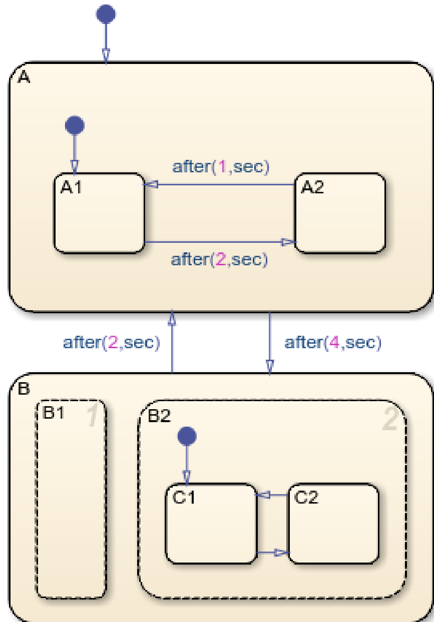
method to specify a different default value. For more information, see “Define Enumerated Data Types” on page 20-5.

Tip The base storage type for automatically created enumerations defaults to `Native Integer`. To use a smaller memory footprint, open the Configuration Parameters dialog box and, in the **Optimization** pane, change the value of the **Base storage type for automatically created enumerations** parameter. For more information, see Base storage type for automatically created enumerations (Simulink Coder).

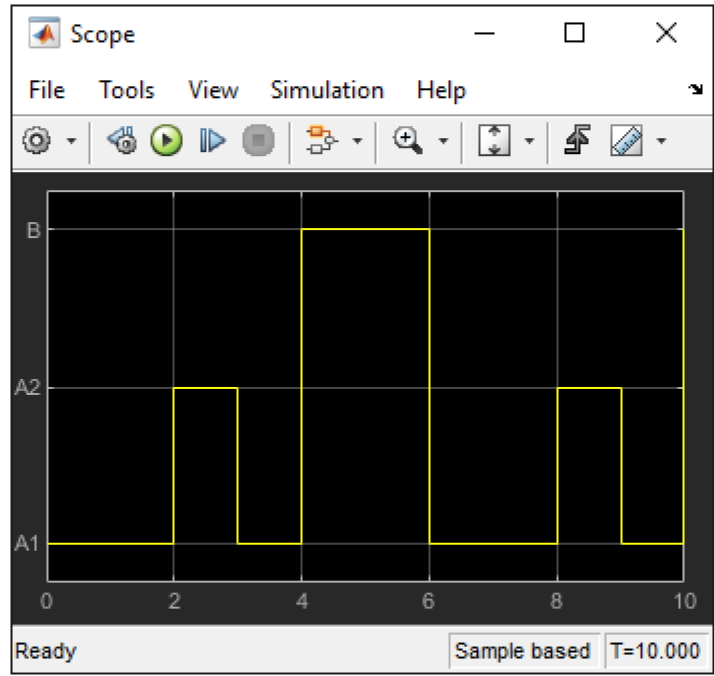
State Activity and Parallel States

In states with parallel (AND) decomposition, child activity and leaf state activity are not available because the parallel substates are active simultaneously.

When you enable leaf state activity in a chart or state, a substate with parallel (AND) decomposition is treated as a leaf state. For example, suppose that you enable leaf state activity for this chart. Because state `B` has parallel decomposition, its substates `B1` and `B2` are active simultaneously so `B` is treated as a leaf state of the chart.



During simulation, a scope connected to the active state output data shows the enumerated values for the leaf states `A1`, `A2`, and `B`.



Limitations for Active State Data

- Enabling child activity output for states that have no children results in an error at compilation and run time.
- You cannot enable child or leaf state activity in charts or states with parallel decomposition. To check state activity in substates of parallel states, use the `in` operator. For more information, see "Check State Activity by Using the in Operator" on page 11-24.
- Active state data is not supported in charts that use classic or Mealy semantics when the chart property **Initialize outputs every time chart wakes up** is enabled. For more information, see "Initialize outputs every time chart wakes up" on page 1-21.

See Also

Objects
`Simulink.Signal` | `Simulink.CoderInfo`

Tools
Model Explorer

More About

- "Simplify Stateflow Charts by Incorporating Active State Output" on page 11-30
- "View State Activity by Using the Simulation Data Inspector" on page 11-7
- "Check State Activity by Using the in Operator" on page 11-24
- "Guidelines for Naming Stateflow Objects" on page 1-66
- "Define State Activity Enumeration Type" on page 11-4
- Base storage type for automatically created enumerations (Simulink Coder)

View State Activity by Using the Simulation Data Inspector

You can use the **Simulation Data Inspector** to log state activity and data for your Stateflow chart. With the Simulation Data Inspector, you can view and compare:

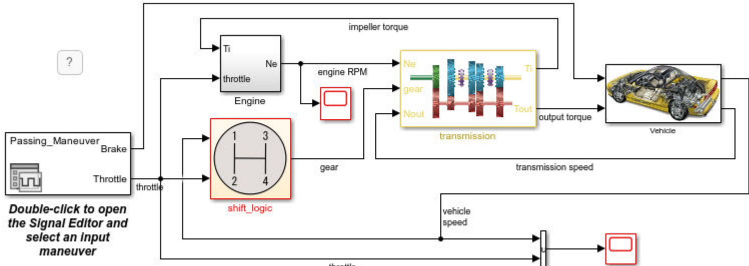
- Data from your chart
- Child and leaf state activity for your chart
- Self, child, and leaf state activity for your states

Add Signals and States for Logging

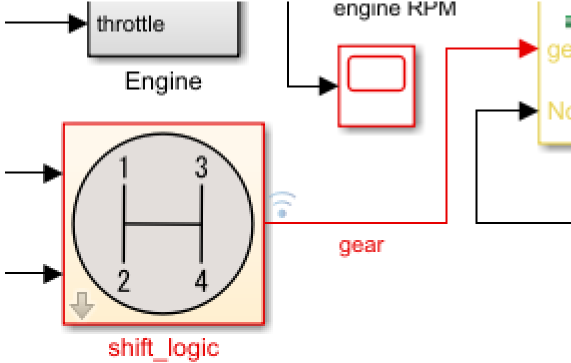
In this example, you use the Simulation Data Inspector to monitor the active state data for the Stateflow chart in the model `sf_car`.

1 Open the model `sf_car`.

```
openExample("stateflow/AutomaticTransmissionWithActiveStateDataExample")
```

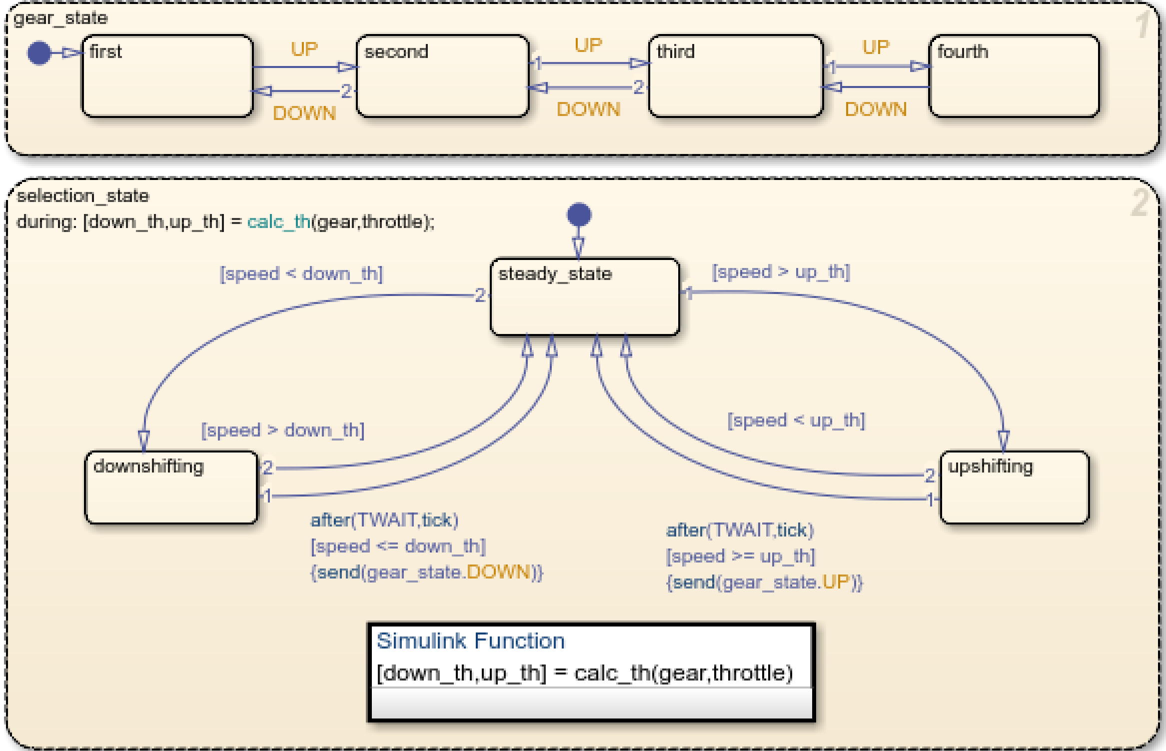


2 In the Simulink Editor, click the `gear` signal. Then, in the **Simulation** tab, select **Log Signals**.



The logging badge appears above the signal to indicate that the data from the signal is logged when you run the model.

3 Open the `shift_logic` chart by clicking the arrow in the bottom-left corner of the block.



4 Select the state `gear_state`. Then, in the **Simulation** tab, click **Log Child Activity**. The logging badge appears in the corner of the state.

5 In the `selection_state` state, select the `downshifting` substate. Then, in the **Simulation** tab, click **Log Self Activity**.

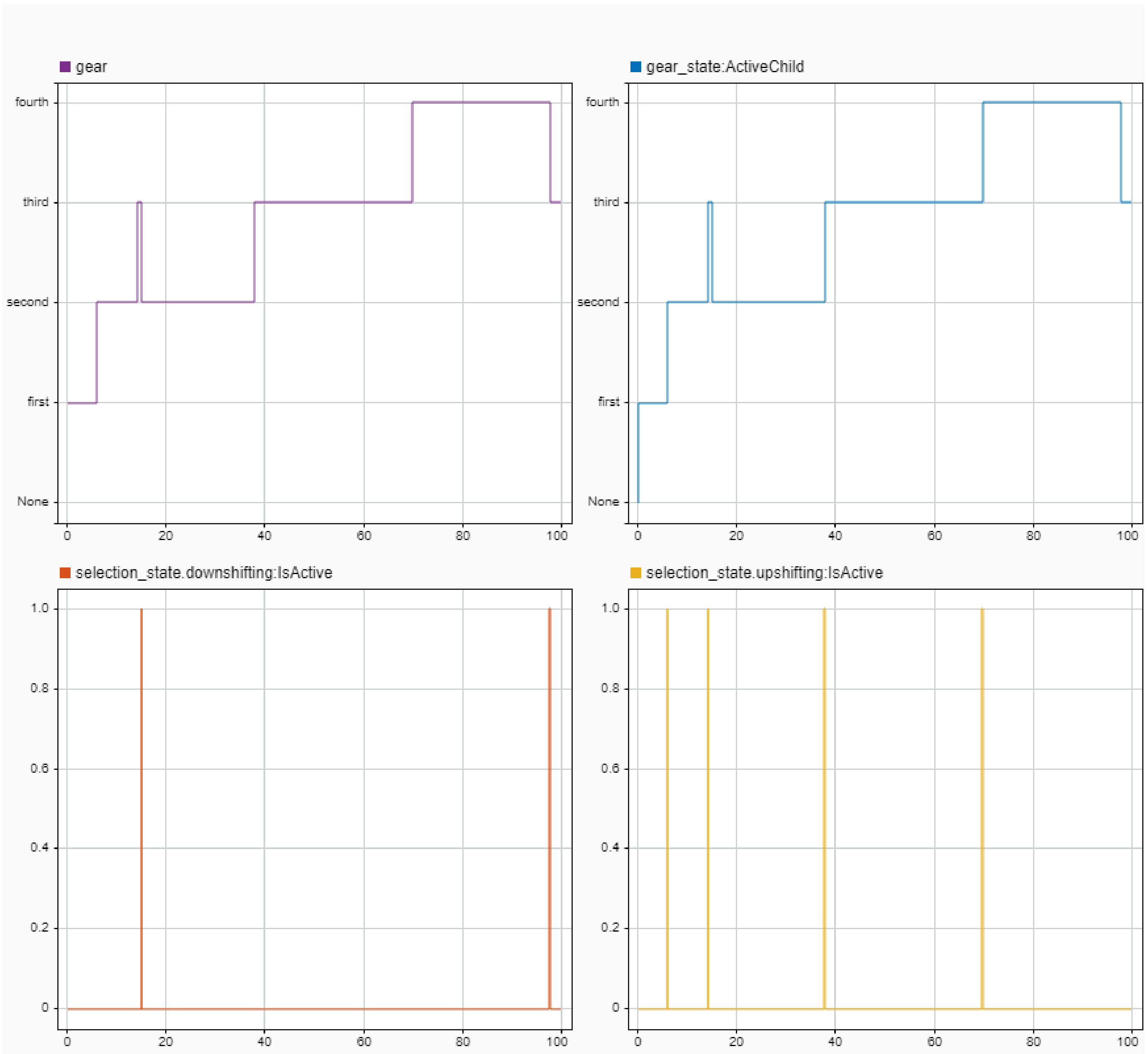
6 Repeat the previous step for the `upshifting` substate.

View Logged Output in Simulation Data Inspector

1 Simulate the model.

2 In the **Simulation** tab, under **Review Results**, select **Data Inspector**. When you simulate the model, the icon is highlighted to indicate that the Simulation Data Inspector has new simulation data.

3 In the Simulation Data Inspector, under **Visualizations and Layouts**, select the four-plot grid layout. Then, in the **Inspect** pane, select a signal for each plot. For more information, see “Inspect Simulation Data” (Simulink).



See Also

Simulation Data Inspector

More About

- “Monitor State Activity Through Active State Data” on page 11-2
- “Simplify Stateflow Charts by Incorporating Active State Output” on page 11-30
- “Inspect Simulation Data” (Simulink)
- “Compare Simulation Data” (Simulink)

View Stateflow States in the Logic Analyzer

Use the **Logic Analyzer** to visualize and measure transitions and states over time. With the **Logic Analyzer**, you can visualize:

- Output data from your chart
- Child and leaf state activity for your chart
- Self, child, and leaf state activity for your states

You can measure the output over time and add triggers to identify the output values at specified events.

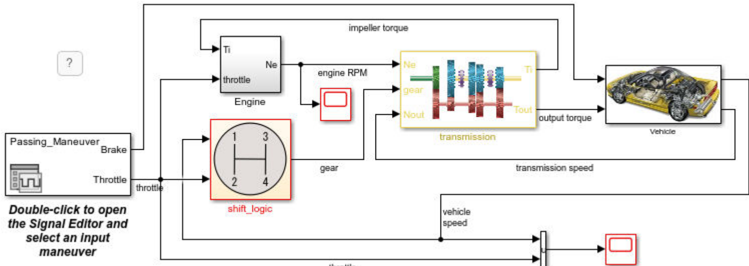
Note To use the **Logic Analyzer**, you must have DSP System Toolbox, SoC Blockset, or HDL Verifier.

Add Signals and States for Logging

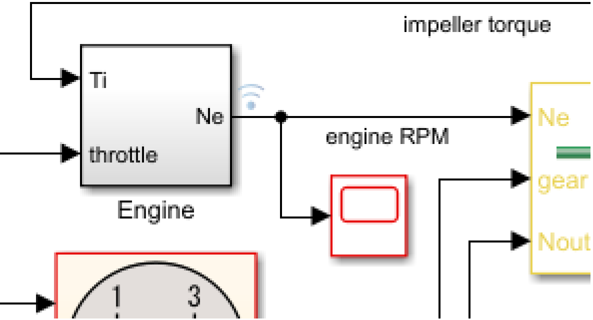
In this example, you use the **Logic Analyzer** to visualize the behavior of the engine RPM, the transmission and vehicle speed, and the gear state in the model `sf_car`.

1. Open the model `sf_car`.

```
openExample("stateflow/AutomaticTransmissionWithActiveStateDataExample")
```



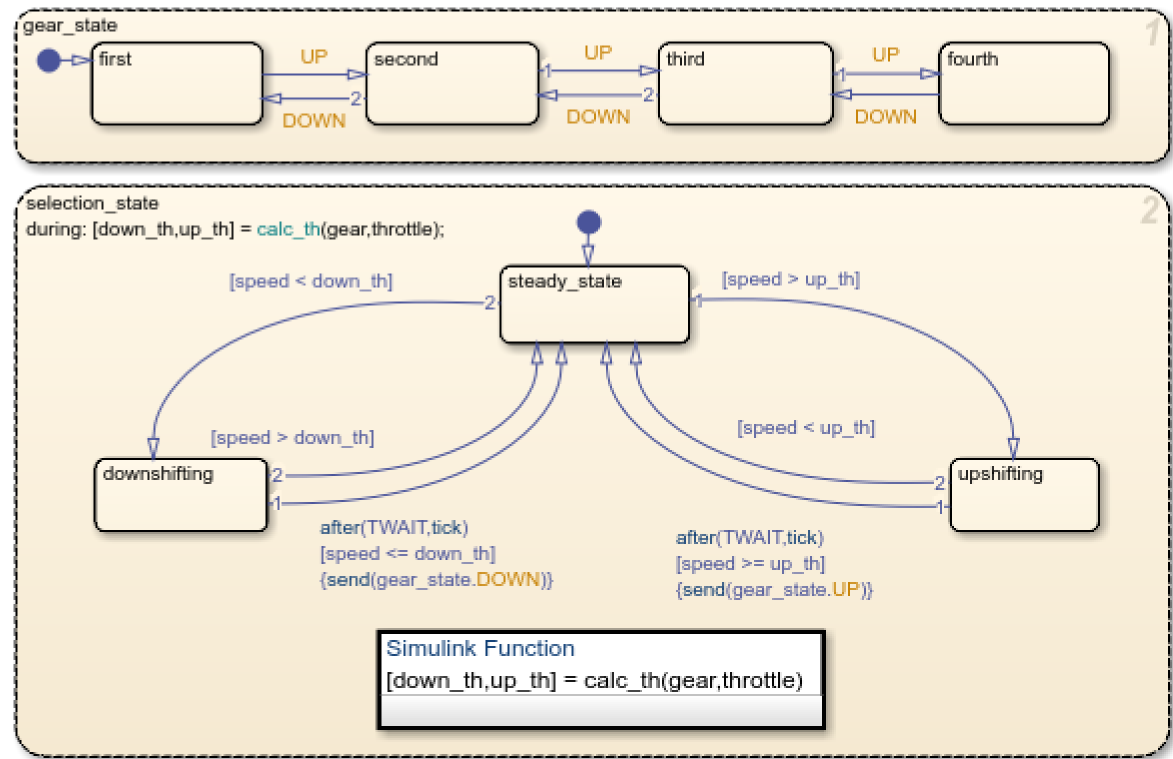
2. In the Simulink Editor, click the `engine RPM` signal. Then, in the **Simulation** tab, select **Log Signals**.



The logging badge appears above the signal to indicate that the data from the signal is logged when you run the model.

3 Repeat the previous step for the `transmission speed` and `vehicle speed` signals.

4 Open the `shift_logic` chart by clicking the arrow in the bottom-left corner of the block.



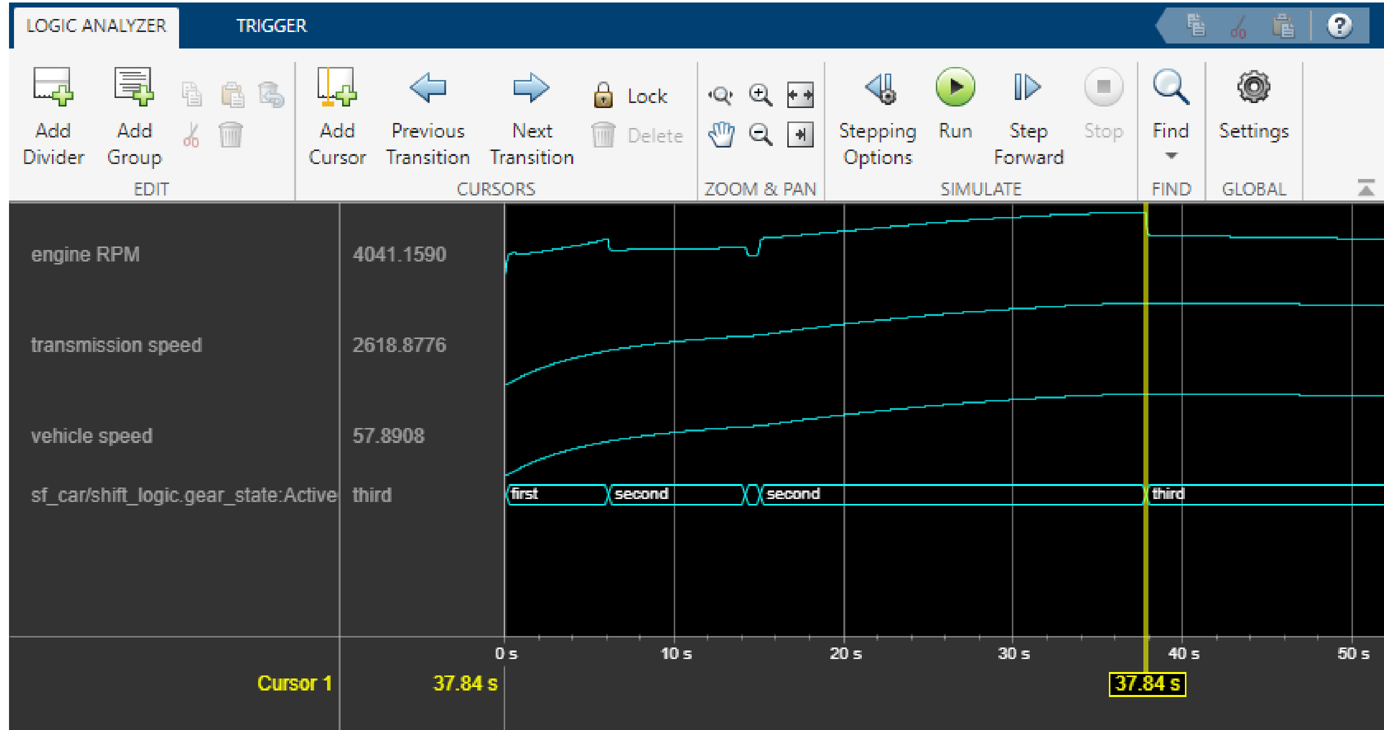
5 Select the state `gear_state`. Then, in the **Simulation** tab, click **Log Child Activity**. The logging badge appears in the corner of the state.

View Logged Output in Logic Analyzer

1 Simulate the model.

2 In the **Simulation** tab, under **Review Results**, select **Logic Analyzer**. When you simulate the model, the icon is highlighted to indicate that the **Logic Analyzer** has new simulation data.

3 In the **Logic Analyzer** window, drag the yellow cursor to see the signal values at different points in the simulation. For example, you can see the reaction of the engine RPM as the car gears change. For more information, see “Inspect and Measure Transitions Using the Logic Analyzer” (DSP System Toolbox).



See Also

Logic Analyzer

More About

- “Log Simulation Output for States and Data” on page 11-13
- “Inspect and Measure Transitions Using the Logic Analyzer” (DSP System Toolbox)
- “View State Activity by Using the Simulation Data Inspector” on page 11-7

Log Simulation Output for States and Data

When you simulate a Stateflow chart in a Simulink model, you can log values for local, output, and active state data into a `Simulink.SimulationData.Dataset` object. After simulation, you can access this object through the **Simulation Data Inspector**, the **Logic Analyzer**, or in the MATLAB workspace. The workflow for logging data is:

1. Enable signal logging for the chart and choose a logging format. See "Enable Signal Logging" on page 11-13.
2. Configure states and data for signal logging. See "Configure States and Data for Logging" on page 11-13.
3. Simulate the chart.
4. Access the logged data. See "Access Signal Logging Data" on page 11-15.

Enable Signal Logging

Signal logging is enabled by default for models and charts. To disable or reenable signal logging:

1. Open the Configuration Parameters dialog box.
2. In the **Data Import/Export** pane, select **Signal logging** to enable logging for the chart. To disable logging, clear **Signal logging**. For more information, see Signal logging (Simulink).
3. (Optional) Specify a custom name for the signal logging object. The default name is `logsout`. Using this object, you can access the logging data in a MATLAB workspace variable. For more information, see "Save Signal Data Using Signal Logging" (Simulink).
4. (Optional) In the **Format** field, select a signal logging format. Options include:

   - `Array`
   - `Structure`
   - `Structure with time`
   - `Dataset`

   The default setting is `Dataset`. For more information, see "Time, State, and Output Data Format" (Simulink).

Configure States and Data for Logging

You can set logging properties for states, local data, and output data from inside the chart, through the Stateflow Signal Logging dialog box, or programmatically from the command line.

Log Individual States and Data

Configure logging properties for one state or data object at a time through the **Property Inspector**, the Model Explorer, or the properties dialog box for the state or data object. Select the **Logging** tab and modify properties as needed. For more information, see "Logging Properties" on page 10-16.

For example:

1. Open the `sf_semantics_hotel_checkin` model:

   ```
   openExample("stateflow/SemanticsHotelCheckinExample")
   ```

For more information about this example, see “How Stateflow Objects Interact During Execution” on page 1-8.

2. Open the `Hotel` chart.
3. Open the **Symbols** pane. In the **Simulation** tab, in **Prepare**, click **Symbols Pane**.
4. Open the **Property Inspector**. In the **Simulation** tab, in **Prepare**, click **Property Inspector**.
5. Configure the `service` local data for logging.

   a. In the **Symbols** pane, select `service`.
   b. In the **Property Inspector**, under **Logging**, select the **Log signal data** check box.
6. Configure the `Dining_area` state for logging.

   a. On the Stateflow Editor, select the `Dining_area` state.
   b. In the **Simulation** tab, in **Prepare**, select **Log Self Activity**. Alternatively, in the **Property Inspector**,under **Logging**, select the **Log self activity** check box.
   c. By default, the logging name for this state is the hierarchical signal name `Check_in.Checked_in.Executive_suite.Dining_area`. To assign a shorter name to the state, set **Logging Name** to `Custom` and enter `Dining Room`.

Log Multiple Signals

Configure logging properties for multiple states and data objects through the Stateflow Signal Logging dialog box. Select which chart objects to log from a list of all states, local, and output data. For more information, see “Logging Properties” on page 10-16.

For example:

1. Open the `sf_semantics_hotel_checkin` model:

```
openExample("stateflow/SemanticsHotelCheckinExample")
```

2. Open the `Hotel` chart.
3. To log multiple signals, press and hold shift to select the states for logging. In the **Simulation** tab, under **Prepare**, select **Log Self Activity**.
4. The logging badge marks logged signals in the model.

Add an Output Port

You can add an output port to monitor chart activity. From the Stateflow Editor, in the **Simulation** tab, click **Add Output Port**. A new port appears on your Stateflow chart. Connect this port to a viewer to monitor the chart child activity.

Log Chart Signals by Using the Command-Line API

Configure logging properties for states and data objects programmatically from the command line. To enable logging for a states or data object, get a handle for the object and set its `LoggingInfo.DataLogging` property to `1`. For more information on the Stateflow Programmatic Interface, see “Overview of the Stateflow API”.

For example:

1 Open the `sf_semantics_hotel_checkin` model:

```
openExample("stateflow/SemanticsHotelCheckinExample")
```

2 Access the `Stateflow.State` object that corresponds to the `Dining_area` state:

```
diningState = find(sfroot,"-isa","Stateflow.State",Name="Dining_area");
```

3 Access the `Stateflow.Data` object that corresponds to the local data `service`:

```
serviceData = find(sfroot,"-isa","Stateflow.Data",Name="service");
```

4 Enable logging for the `Dining_area` state and the `service` data:

```
diningState.LoggingInfo.DataLogging = true;
serviceData.LoggingInfo.DataLogging = true;
```

5 Change the logging name of the `Dining_area` state to the custom name `Dining Room`:

```
% Enable custom naming
diningState.LoggingInfo.NameMode = "Custom";

% Enter the custom name
diningState.LoggingInfo.LoggingName = "Dining Room";
```

Access Signal Logging Data

During simulation, Stateflow saves logged data in a `Simulink.SimulationData.Dataset` signal logging object.

For example, suppose that you configure the `sf_semantics_hotel_checkin` model to log the `service` local data and the activity of the `Dining_area` state. After starting the simulation, you check into the hotel by toggling the first switch twice and order room service by toggling the second switch multiple times. After stopping the simulation, you can view the logged data through the Simulation Data Inspector, Logic Analyzer, or in the MATLAB workspace.

View Logged Data Through the Simulation Data Inspector

To open the Simulation Data Inspector, in the **Simulation** tab, select **Data Inspector**. When you simulate the model, the icon is highlighted to indicate that the Simulation Data Inspector has new simulation data. For more information, see “View State Activity by Using the Simulation Data Inspector” on page 11-7.

View Logged Data Through the Logic Analyzer

To open the **Logic Analyzer**, in the **Simulation** tab, select **Logic Analyzer**. When you simulate the model, the icon is highlighted to indicate that the **Logic Analyzer** has new simulation data. For more information, see “View Stateflow States in the Logic Analyzer” on page 11-10.

Note To use the **Logic Analyzer**, you must have DSP System Toolbox, SoC Blockset, or HDL Verifier.

View Logged Data in the MATLAB Workspace

1 To access the signal logging object, at the MATLAB command prompt, enter:

```
logsout = out.logsout
```

```
logsout = 

Simulink.SimulationData.Dataset 'logsout' with 2 elements

                        Name         BlockPath                       
                        ___________  _________________________________
    1  [1x1 State]      Dining Room  sf_semantics_hotel_checkin/Hotel
    2  [1x1 Data ]      service      sf_semantics_hotel_checkin/Hotel
```

2 To access logged elements, use the `get` function. You can access logged elements by name, index, or block path.

```
diningLog = get(logsout,"Dining Room")

diningLog = 

  Stateflow.SimulationData.State
  Package: Stateflow.SimulationData

  Properties:
         Name: 'Dining Room'
    BlockPath: [1×1 Simulink.SimulationData.BlockPath]
       Values: [1×1 timeseries]

serviceLog = get(logsout,"service")

serviceLog = 

  Stateflow.SimulationData.Data
  Package: Stateflow.SimulationData

  Properties:
         Name: 'service'
    BlockPath: [1×1 Simulink.SimulationData.BlockPath]
       Values: [1×1 timeseries]
```

3 To access the logged data and time of each logged element, use the `Values.Data` and `Values.Time` properties. For example:

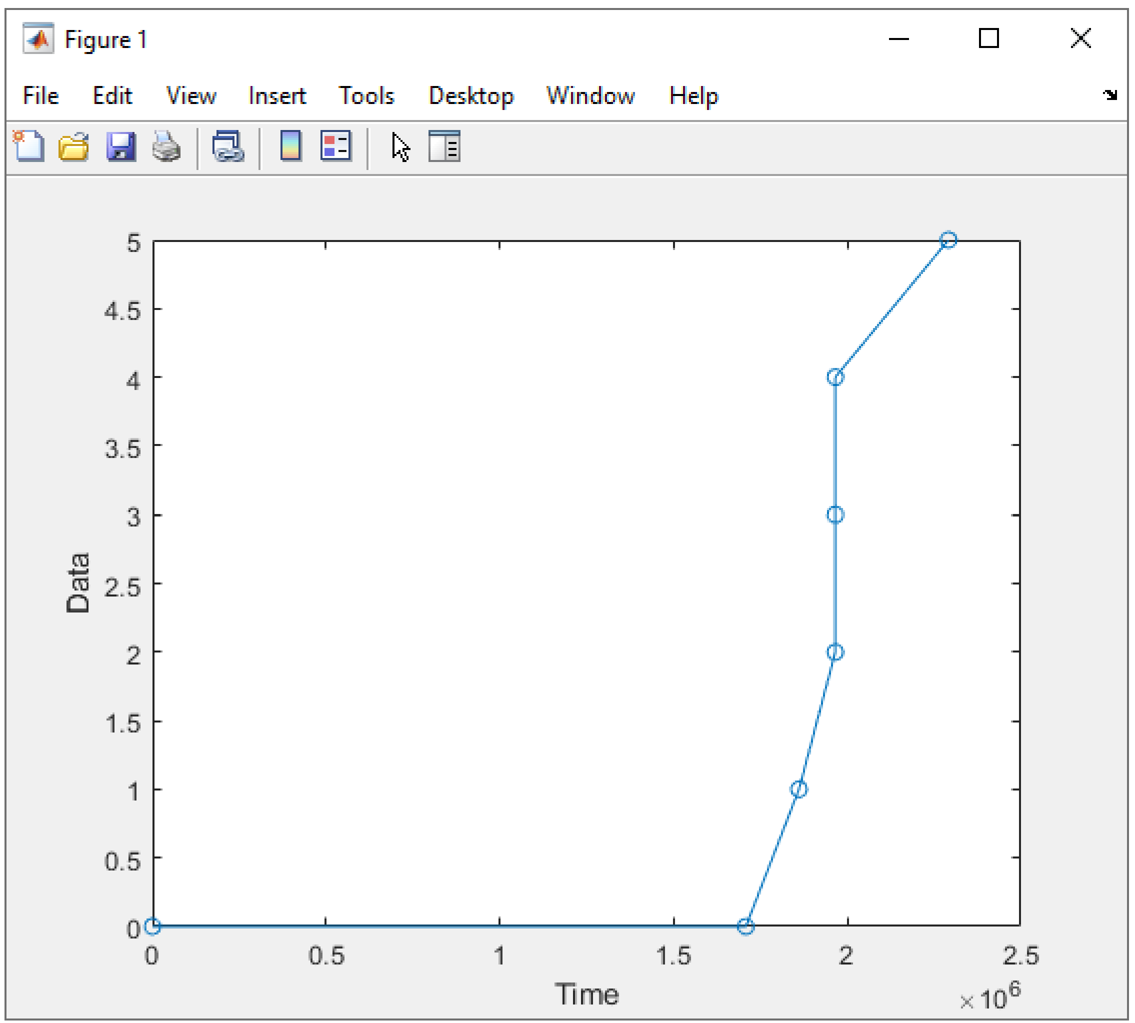
- Arrange logged data in tabular form by using the `table` function.

  ```
  T1 = table(diningLog.Values.Time,diningLog.Values.Data);
  T1.Properties.VariableNames = ["Time","Data"]

  T1 = 

    6×2 table

           Time       Data
        __________    ____

                 0     0  
        1.8607e+06     1  
        1.9653e+06     0  
        1.9653e+06     1  
        1.9653e+06     0  
        2.2912e+06     1  

  T2 = table(serviceLog.Values.Time,serviceLog.Values.Data);
  T2.Properties.VariableNames = ["Time","Data"]
  ```

```
T2 =

  6×2 table

       Time        Data
    ___________    ____

              0     0
     1.7076e+06     0
     1.8607e+06     1
     1.9653e+06     2
     1.9653e+06     3
     1.9653e+06     4
     2.2912e+06     5
```

- View logged data in a figure window by using the `plot` function.

```
X = serviceLog.Values.Time;
Y = serviceLog.Values.Data;
plot(X,Y,"-o")
xlabel("Time")
ylabel("Data")
```



- Export logged data to an Excel® spreadsheet by passing an array of logged values to the `xlswrite` function:

  ```
  A = [double(diningLog.Values.Time) double(diningLog.Values.Data)];
  xlswrite("dining_log.xls",A);
  ```

Note The signal logging object records a data point every time that the Stateflow chart writes to the data you are logging, even if the data does not change value. For instance, in this example, the data points with values of `0` correspond to when the chart initializes the local data `service` to `0` at time `0` and when a default transition sets `service` to `0` at time `1.7076e+06`.

Log Multidimensional Data

Stateflow logs each update to a multidimensional signal as a single change. For example, updating two elements of a matrix `A` separately

```
A[1][1] = 1;
A[1][2] = 1;
```

produces two different changes in the logged data. In contrast, updating a matrix `A` in a single command

```
A = 1;
```

produces a single change in the logged data, even though the command implies `A[i][j] = 1` for all values of `i` and `j`.

Limitations on Logging Data

When simulating models in external mode, logging of Stateflow data is not supported.

If you log state activity or data from a chart with **Fast Restart** enabled, any run after the first run duplicates the first logged data points. When you run algorithms that process these data points, you must account for this duplication.

See Also

Apps
Simulation Data Inspector | **Logic Analyzer**

Objects
`Simulink.SimulationData.Dataset` | `Stateflow.SimulationData.Data` | `Stateflow.SimulationData.State`

Functions
`plot` | `table` | `xlswrite`

Tools
Signal Properties

More About

- “Save Signal Data Using Signal Logging” (Simulink)

Log Data in Library Charts

In Simulink, you can create your own block libraries as a way to reuse the functionality of blocks or subsystems in one or more models. Similarly, you can reuse a set of Stateflow algorithms by encapsulating the functionality in a library chart.

As with other Simulink block libraries, you can specialize each instance of chart library blocks in your model to use different data types, sample times, and other properties. Library instances that inherit the same properties can reuse generated code.

For more information about Simulink block libraries, see “Custom Libraries” (Simulink).

How Library Log Settings Influence Linked Instances

Chart instances inherit logging properties from the library chart to which they are linked. You can override logging properties in the instance, but only for signals you select in the library. You cannot select additional signals to log from the instance.

Override Logging Properties in Chart Instances

To override properties of logged signals in chart instances, use one of the following approaches.

Approach	How To Use
Simulink Signal Logging Selector dialog box	See “Override Logging Properties with the Logging Selector” on page 11-20
Command-line interface	See “Override Logging Properties with the Command-Line API” on page 11-22

Override Logging Properties in Atomic Subcharts

The model `sf_atomic_sensor_pair` simulates a redundant sensor pair as atomic subcharts `Sensor1` and `Sensor2` in the chart `RedundantSensors`. Each atomic subchart contains instances of the states `Fail`, `FailOnce`, and `OK` from the library chart `sf_atomic_sensor_lib`.

Override Logging Properties with the Logging Selector

1. Open the example library `sf_atomic_sensor_lib`.

```
openExample("stateflow/ModelingARedundantSensorPairUsingAtomicSubchartExample", ...
    supportingFile="sf_atomic_sensor_lib");
```

2. Unlock the library. In the **Simulation** tab, click **Locked Library**.
3. In the Simulink Editor, select the Stateflow `SingleSensor` chart. In the **Simulation** tab, click **Log States from List**.
4. In Stateflow Signal Logging dialog box, set the following logging properties, then click **OK**.

For Signal:	What to Specify:
`Fail`	• Select the **Log signal data** check box. • Change **Logging name** to the custom name `LogFail`. • Click **Apply**.
`FailOnce`	• Select the **Log signal data** check box. • Change **Logging name** to the custom name `LogFailOnce`. • Click **Apply**.
`OK`	• Select the **Log signal data** check box. • Change **Logging name** to the custom name `LogOK`. • Click **Apply**.

5 Open the model `sf_atomic_sensor_pair`. This model contains two instances of the library chart.

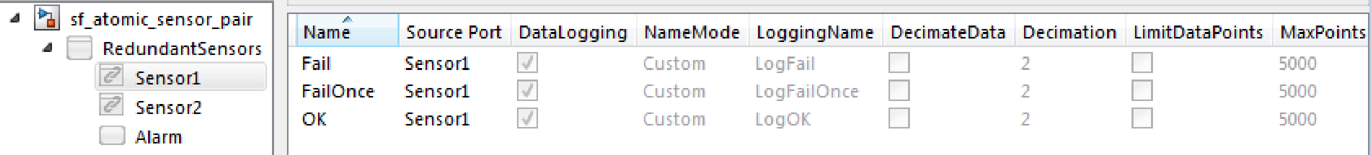
```
openExample("stateflow/ModelingARedundantSensorPairUsingAtomicSubchartExample")
```

6 Open the Configuration Parameters dialog box.

7 In the **Data Import/Export** pane, click **Configure Signals to Log** to open the Simulink Signal Logging Selector.

8 In the **Model Hierarchy** pane, expand `RedundantSensors`, and click `Sensor1` and `Sensor2`.

Each instance inherits logging properties from the library chart.



9 Override some logging properties for `Sensor1`:

a In the **Model Hierarchy** pane, select `Sensor1`.

b Change **Logging Mode** to `Override signals`.

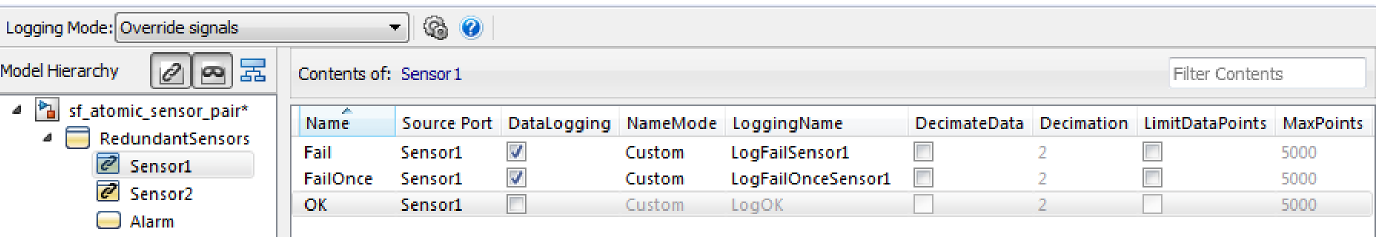
The selector clears all **DataLogging** check boxes for the model.

c Enable logging only for the `Fail` and `FailOnce` states in `Sensor1`:

Select **DataLogging** for these two signals. Leave **DataLogging** cleared for the `OK` signal.

d Append the text `Sensor1` to the logging names for `Fail` and `FailOnce`:

Double-click the logging names for signals `Fail` and `FailOnce`, and rename them `LogFailSensor1` and `LogFailOnceSensor1`, respectively.



Override Logging Properties with the Command-Line API

1. Open the example library `sf_atomic_sensor_lib`.

```
openExample("stateflow/ModelingARedundantSensorPairUsingAtomicSubchartExample", ...
    supportingFile="sf_atomic_sensor_lib");
```

2. Unlock the library.

```
library = find(sfroot,"-isa","Stateflow.Machine", ...
    Name="sf_atomic_sensor_lib");
library.Locked = false;
```

3. Log the signals `Fail`, `FailOnce`, and `OK` in the `SingleSensor` chart using these commands:

```
states = find(library,"-isa","Stateflow.State");

for i = 1: length(states)
   states(i).LoggingInfo.DataLogging = true;
end
```

4. Open the model `sf_atomic_sensor_pair`. This model contains two instances of the library chart.

```
open_system("sf_atomic_sensor_pair")
```

5. Create a `ModelLoggingInfo` object for the model.

   This object contains a vector `Signals` that stores all logged signals.

```
logInfo = Simulink.SimulationData.ModelLoggingInfo.createFromModel('sf_atomic_sensor_pair')

logInfo = 

  ModelLoggingInfo with properties:

                     Model: 'sf_atomic_sensor_pair'
               LoggingMode: 'OverrideSignals'
    LogAsSpecifiedByModels: {}
                   Signals: [1×6 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`

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. For example, 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.

```
failidx = logInfo.findSignal(failPath);
failOnceidx = logInfo.findSignal(failOncePath);
OKidx = logInfo.findSignal(OKPath);
```

8. Override some logging properties for the signals in `Sensor1`:

   a. Disable logging for signal `OK`:

```
logInfo.Signals(OKidx).LoggingInfo.DataLogging = false;
```

b Append the text `Sensor1` to the logging names for `Fail` and `FailOnce`:

```
% Enable custom naming
logInfo.Signals(failidx).LoggingInfo.NameMode = true;
logInfo.Signals(failOnceidx).LoggingInfo.NameMode = true;

% Enter the custom name
logInfo.Signals(failidx).LoggingInfo.LoggingName = "LogFailSensor1";
logInfo.Signals(failOnceidx).LoggingInfo.LoggingName = "LogFailOnceSensor1";
```

9 Apply the changes:

```
set_param(bdroot,DataLoggingOverride=logInfo);
```

See Also

`Simulink.SimulationData.ModelLoggingInfo` | `Simulink.SimulationData.BlockPath`

Check State Activity by Using the in Operator

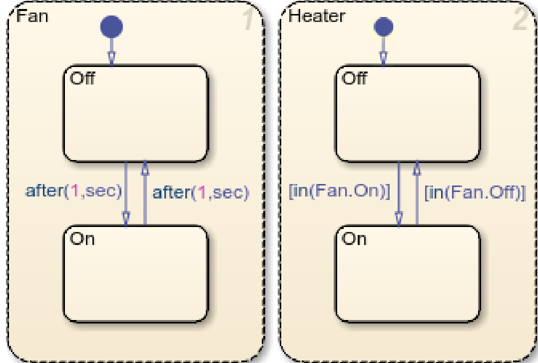
In a Stateflow chart with parallel state decomposition, substates can be active at the same time. To coordinate the behavior of different parallel states, one state can check the substate activity of another state and react accordingly. For example, one state can keep its substates synchronized with the substates of the other state.

The in Operator

To check if a state is active in a given time step, call the `in` operator in state and transition actions. The `in` operator takes a qualified state name `state_name` and returns a Boolean output. If state `state_name` is active, `in` returns a value of 1 (`true`). Otherwise, `in` returns a value of 0 (`false`).

```
in(state_name)
```

For example, in this chart, `Fan` and `Heater` are parallel (AND) states. Each state has a pair of substates, `On` and `Off`. Every second, the active substate of the state `Fan` alternates between `Fan.Off` and `Fan.On`. In the state `Heater`, the conditions on the transitions check the substate activity in `Fan` and keep the states synchronized. A change of active substate in `Fan` causes a corresponding change of active substate in `Heater`.



Resolution of State Activity

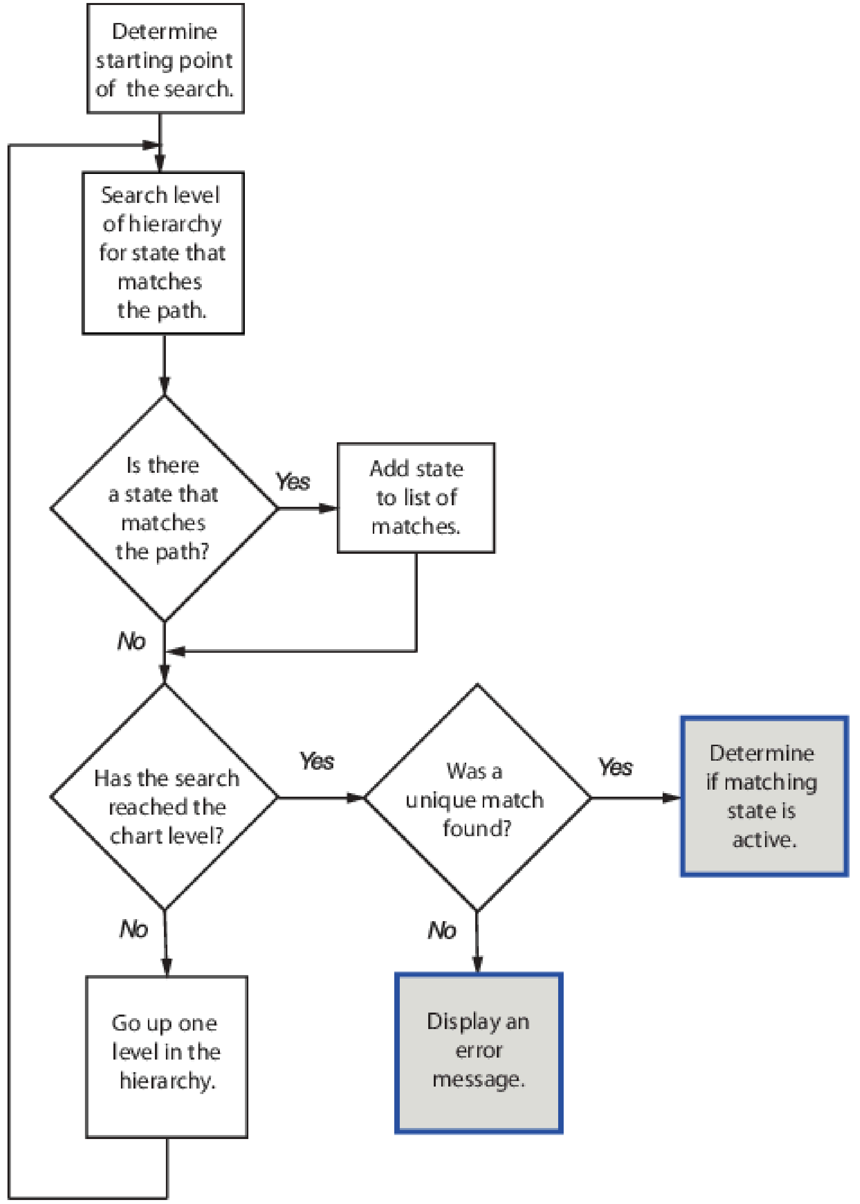
Checking state activity is a two-part process. First, the Stateflow chart resolves the qualified state name by performing a localized search of the chart hierarchy for a matching state. Then, the chart determines if the matching state is active.

The search begins at the hierarchy level where the `in` operator is called with the qualified state name:

- For a state action, the starting point is the state containing the action.
- For a transition label, the starting point is the parent of the transition source.

The resolution process searches each level of the chart hierarchy for a path to the state. If a state matches the path, the process adds that state to the list of possible matches. Then, the process continues the search one level higher in the hierarchy. The resolution process stops after it searches the chart level of the hierarchy. If a unique match exists, the chart checks if the matching state is active. Otherwise, the resolution process fails. Simulation stops with an error.

This flow chart illustrates the different stages in the process for checking state activity.



Best Practices for Checking State Activity

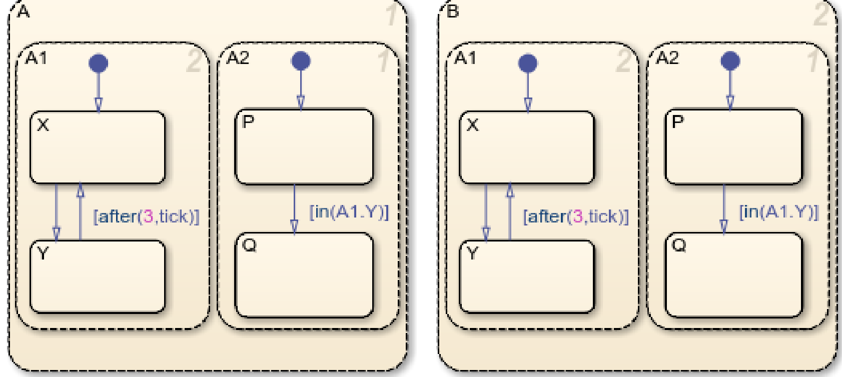
To resolve the state activity, a Stateflow chart does not perform an exhaustive search for all states and does not stop after finding the first match. To improve the chances of finding a unique search result:

- Use specific paths in qualified data names.
- Give states unique names.
- Use states and boxes as enclosures to limit the scope of the path resolution search.

Examples of State Activity Resolution

Search Finds Local Copy of Substate

This chart contains two parallel states, A and B. Each state has a pair of substates, A1 and A2. A1 has substates X and Y, while A2 has substates P and Q. In A.A2 and in B.A2, the condition in(A1.Y) guards the transition from P to Q.



The chart resolves each qualified state name as the local copy of the substate Y:

- In the state A, the condition in(A1.Y) checks the activity of state A.A1.Y.
- In the state B, the condition in(A1.Y) checks the activity of state B.A1.Y.

For example, this table lists the different stages in the resolution process for the transition condition in state A.

Stage	Description	Result
1	Starting in state A.A2, the chart searches for the state A.A2.A1.Y.	No match found.
2	Moving up to the next level of the hierarchy (state A), the chart searches for the state A.A1.Y	Match found.
3	Moving up to the next level of the hierarchy (the chart level), the chart searches for the state A1.Y	No match found.

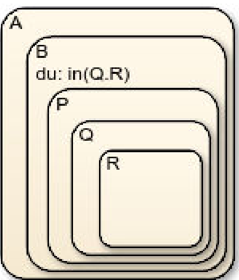
The search ends with a single match found. Because the resolution algorithm localizes the scope of the search, the in operator guarding the transition in A.A2 detects only the state A.A1.Y. The in operator guarding the transition in B.A2 detects only the state B.A1.Y.

To check the state activity of the other copy of Y, use more specific qualified state names:

- In state A, use the expression in(B.A1.Y).
- In state B, use the expression in(A.A1.Y).

Search Produces No Matches

In this chart, the during action in state A.B contains the expression in(Q.R). Stateflow cannot resolve the qualified state name Q.R.



This table lists the different stages in the resolution process.

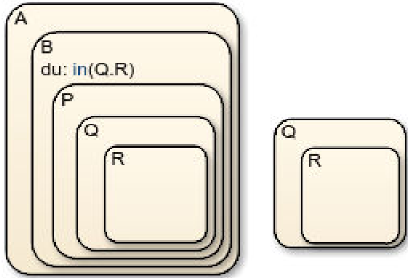
Stage	Description	Result
1	Starting in state `A.B`, the chart searches for the state `A.B.Q.R`.	No match found.
2	Moving up to the next level of the hierarchy (state `A`), the chart searches for the state `A.Q.R`.	No match found.
3	Moving up to the next level of the hierarchy (the chart level), the chart searches for the state `Q.R`.	No match found.

The search ends at the chart level with no match found for `Q.R`, resulting in an error.

To avoid this error, use a more specific qualified state name. For instance, check state activity by using the expression `in(P.Q.R)`.

Search Finds the Wrong State

In this chart, the `during` action in state `A.B` contains the expression `in(Q.R)`. When resolving the qualified state name `Q.R`, Stateflow cannot detect the substate `A.B.P.Q.R`.



This table lists the different stages in the resolution process.

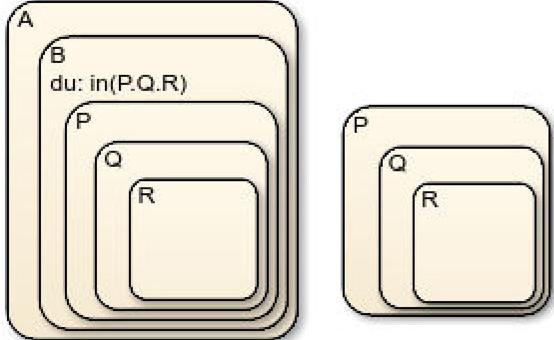
Stage	Description	Result
1	Starting in state `A.B`, the chart searches for the state `A.B.Q.R`.	No match found
2	Moving up to the next level of the hierarchy (state `A`), the chart searches for the state `A.Q.R`.	No match found.
3	Moving up to the next level of the hierarchy (the chart level), the chart searches for the state `Q.R`.	Match found.

The search ends with a single match found. The in operator detects only the substate R of the top-level state Q.

To check the state activity of A.B.P.Q.R, use a more specific qualified state name. For instance, use the expression in(P.Q.R).

Search Produces Multiple Matches

In this chart, the during action in state A.B contains the expression in(P.Q.R). Stateflow cannot resolve the qualified state name P.Q.R.



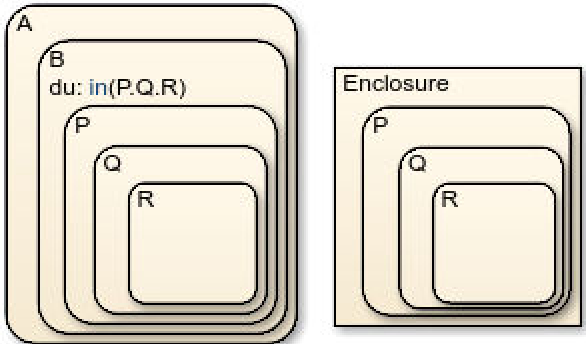
This table lists the different stages in the resolution process.

Stage	Description	Result
1	Starting in state A.B, search for the state A.B.P.Q.R.	Match found
2	Moving up to the next level of the hierarchy (state A), the chart searches for the state A.P.Q.R.	No match found.
3	Moving up to the next level of the hierarchy (the chart level), the chart searches for the state P.Q.R.	Match found.

The search ends at the chart level with two matches found for P.Q.R, resulting in an error.

To avoid this error:

- Use a more specific qualified state name. For example, to check the substate activity inside B, use the expression in(B.P.Q.R).
- Rename one of the matching states.
- Enclose the top-level state P in a box or another state. Adding an enclosure prevents the search process from detecting substates in the top-level state.



See Also

`in` | `after`

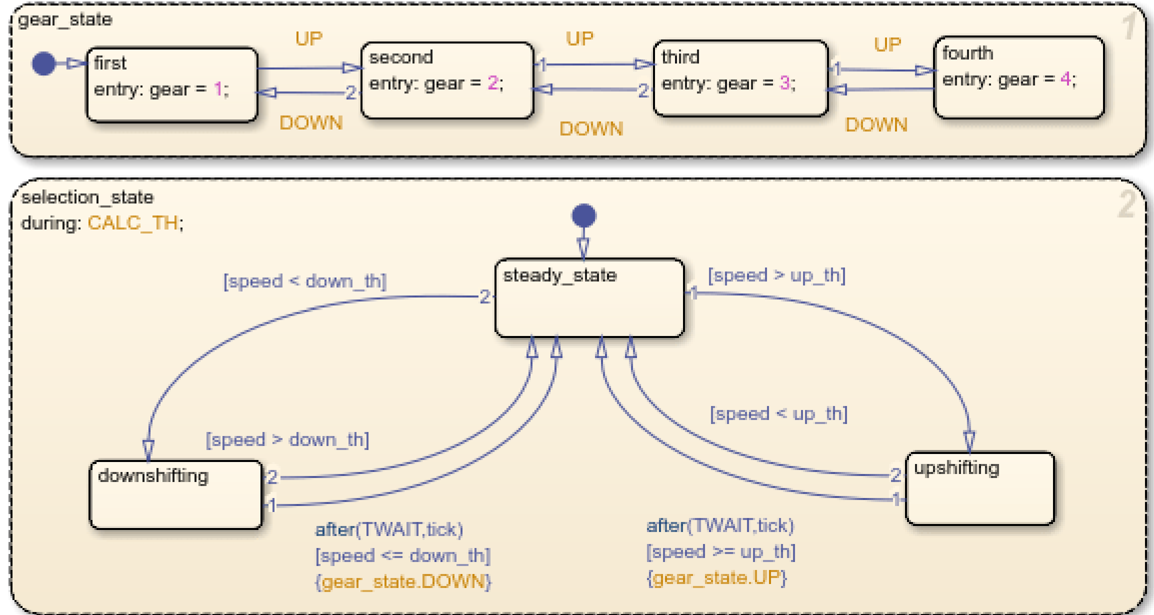
More About

- "Use State Hierarchy to Design Multilevel State Complexity" on page 1-33
- "Monitor State Activity Through Active State Data" on page 11-2

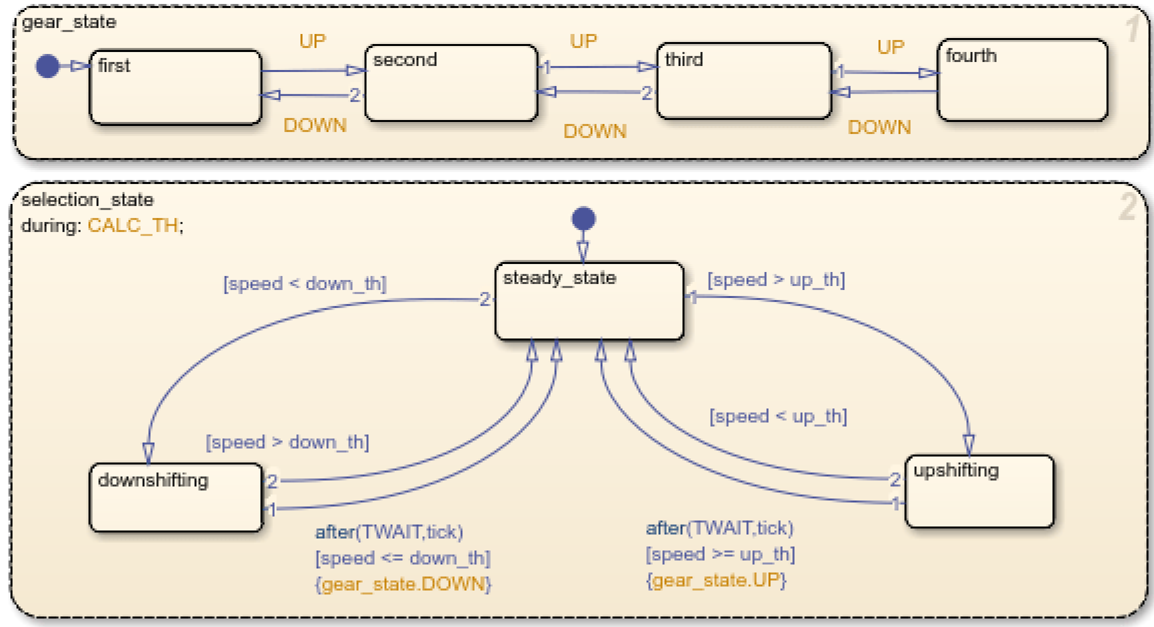
Simplify Stateflow Charts by Incorporating Active State Output

Active state data can simplify the design of some Stateflow® charts because you do not have to maintain data that is highly correlated to the chart hierarchy. When you enable active state data, Stateflow reports state activity through an output port to Simulink® or as local data in your chart. This example shows how to simplify the design of a Stateflow chart by adding active state output data. For more information, see “Monitor State Activity Through Active State Data” on page 11-2.

In the legacy model `old_sf_car`, the Stateflow chart shift_logic tracks child state activity in `gear_state` by updating the value of the output data `gear`.



By incorporating active state data, the model `sf_car` avoids manual data updates reflecting chart activity. Instead, the chart outputs child state activity automatically through the active state output gear.



Modify the Model

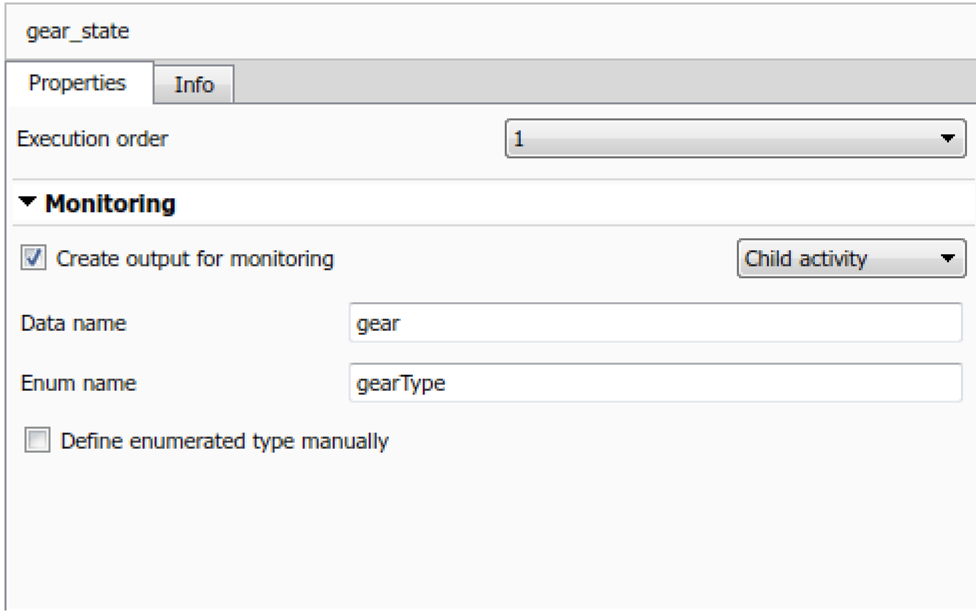
To simplify the design of the `old_sf_car` model, eliminate data that is highly correlated to the chart hierarchy and enable automatic monitoring of child state activity in `gear_state`.

Step 1: Eliminate Manual Tracking of State Activity

1. In the model `old_sf_car`, open the chart `shift_logic`.
2. To open the **Symbols** pane. In the **Modeling** tab, select **Symbols Pane**.
3. In each substate of `gear_state`, delete the entry action assigning a value to the output data variable `gear`.
4. In the **Symbols** pane, right-click the output variable `gear` and select `Delete`.

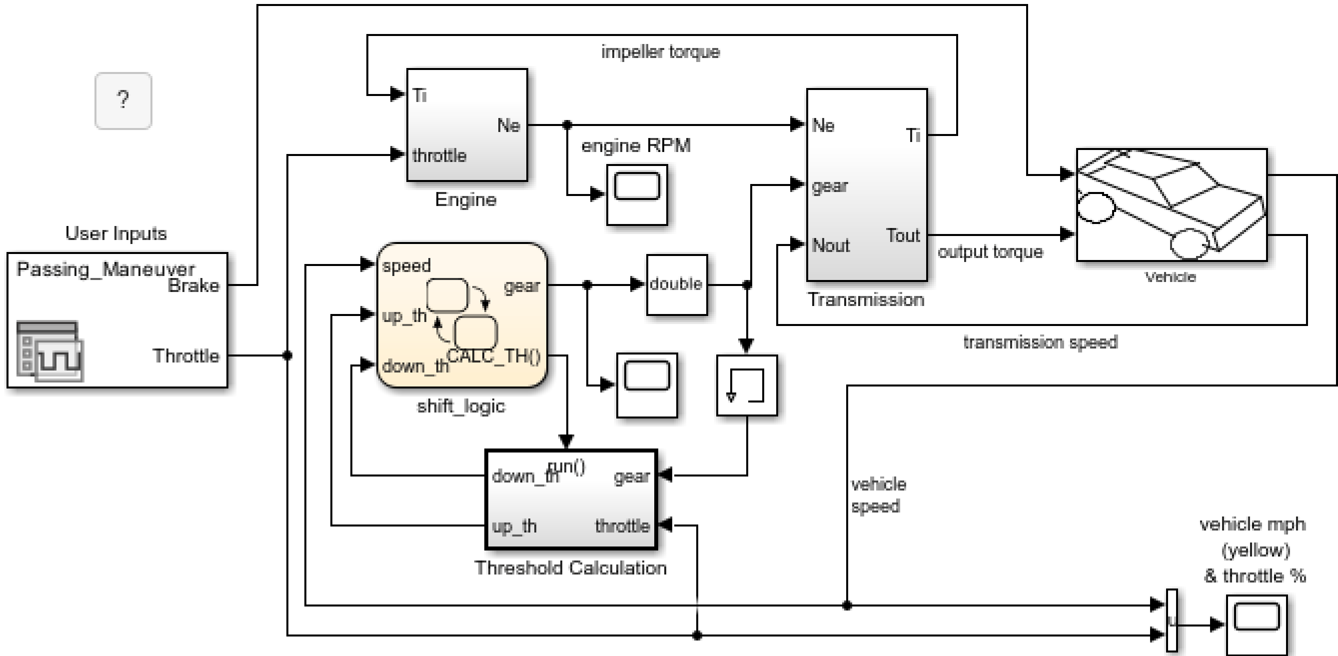
Step 2: Enable Active State Output

1. Open the **Property Inspector**. In the **Modeling** tab, select **Property Inspector**.
2. In the Stateflow Editor, select the state `gear_state`.
3. In the **Property Inspector**, select the **Create output for monitoring** check box and choose `Child activity`.
4. In the **Data name** field, enter the name `gear` of the active state data.
5. In the **Enum name** field, enter the name `gearType` of the enumeration data type for the active state data.



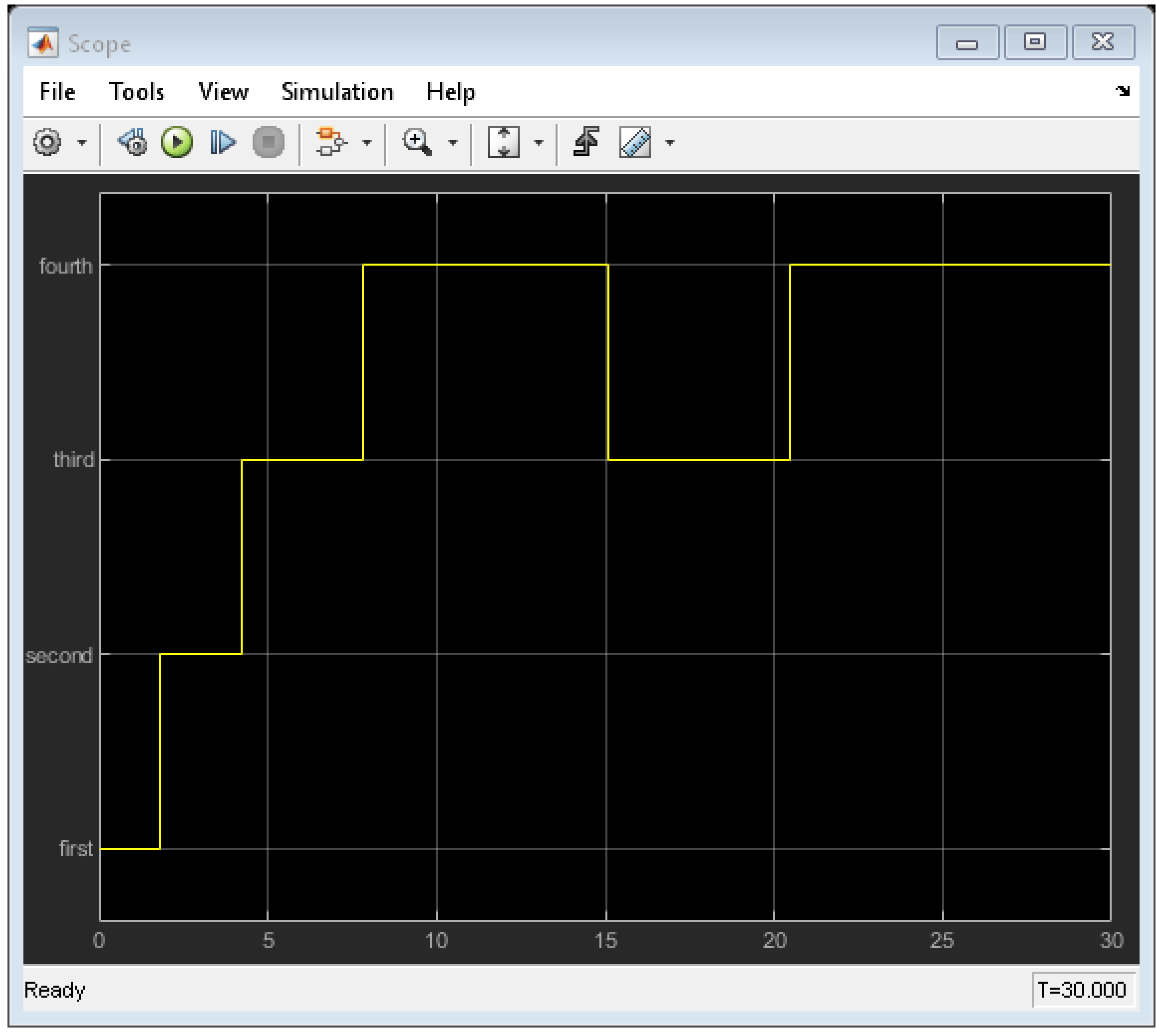
Step 3: Connect Signal to Simulink Blocks

1. In the Simulink model, add a Cast To Double block. This block converts the enumerated output from the Stateflow chart to a signal of type `double`. For more information, see Data Type Conversion (Simulink).
2. Connect the output signal `gear` from the `shift_logic` chart to the Cast To Double block.
3. Connect the output signal from the Cast To Double block to the Transmission subsystem.
4. Add a Memory (Simulink) block. This block prevents an algebraic loop between the Stateflow chart and the Threshold Calculation subsystem.
5. Make a second connection from the output signal from the Cast To Double block to the Memory block.
6. Connect the output of the Memory block to the Threshold Calculation subsystem.



View Simulation Results

The output signal `gear` is an enumerated type managed by Stateflow. You can view the active state output signal gear during simulation by connecting the chart to a Scope block. The names of the enumerated values match the names of the substates in `gear_state`. The additional enumerated value of `None` indicates time steps when no child is active.



See Also

Data Type Conversion | Memory

More About

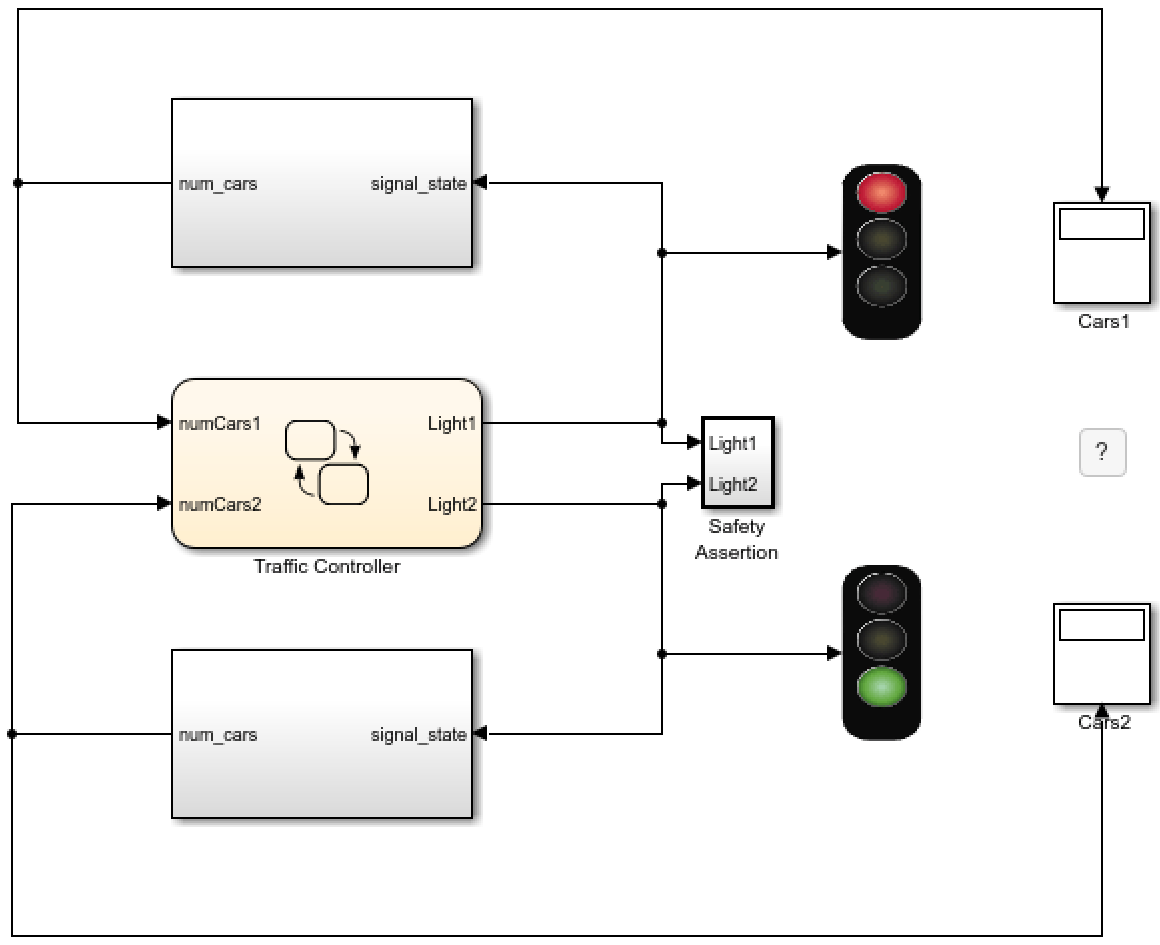
- “Monitor State Activity Through Active State Data” on page 11-2
- “View State Activity by Using the Simulation Data Inspector” on page 11-7
- “Manage Symbols in the Stateflow Editor” on page 25-14

Model An Intersection Of One-Way Streets

This example models an intersection of one-way roads controlled by a Stateflow® traffic light system. The Stateflow chart tracks the state of each traffic light by using active state output. The behavior of the traffic lights is controlled by parameters on the Stateflow mask.

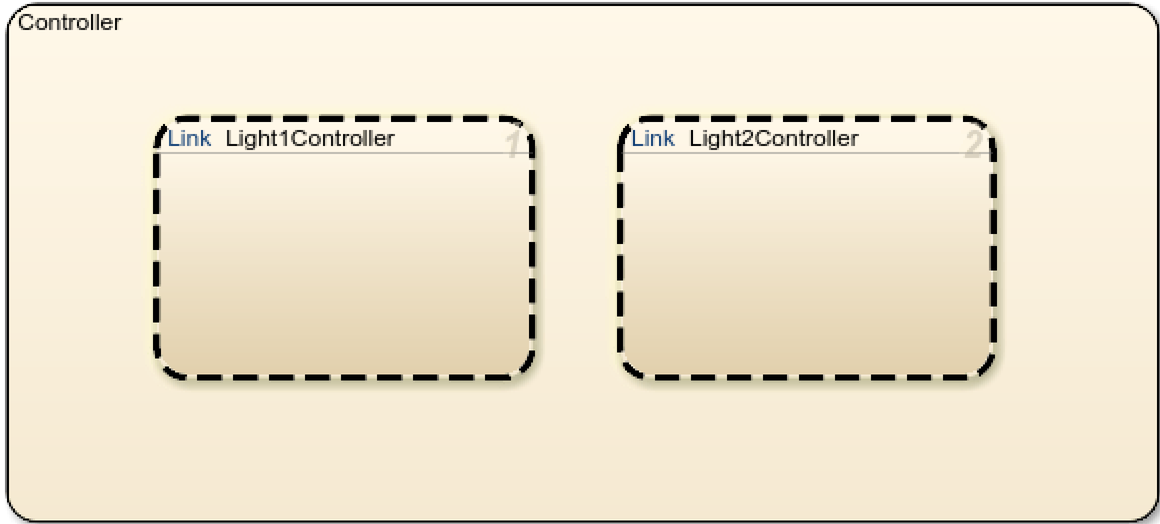
Intersection Model

The phase of the animated traffic lights is determined by the output data from the Stateflow chart. The value of the output data corresponds to the active child of the substates `Light1Controller` and `Light2Controller`, respectively.



Traffic Controller

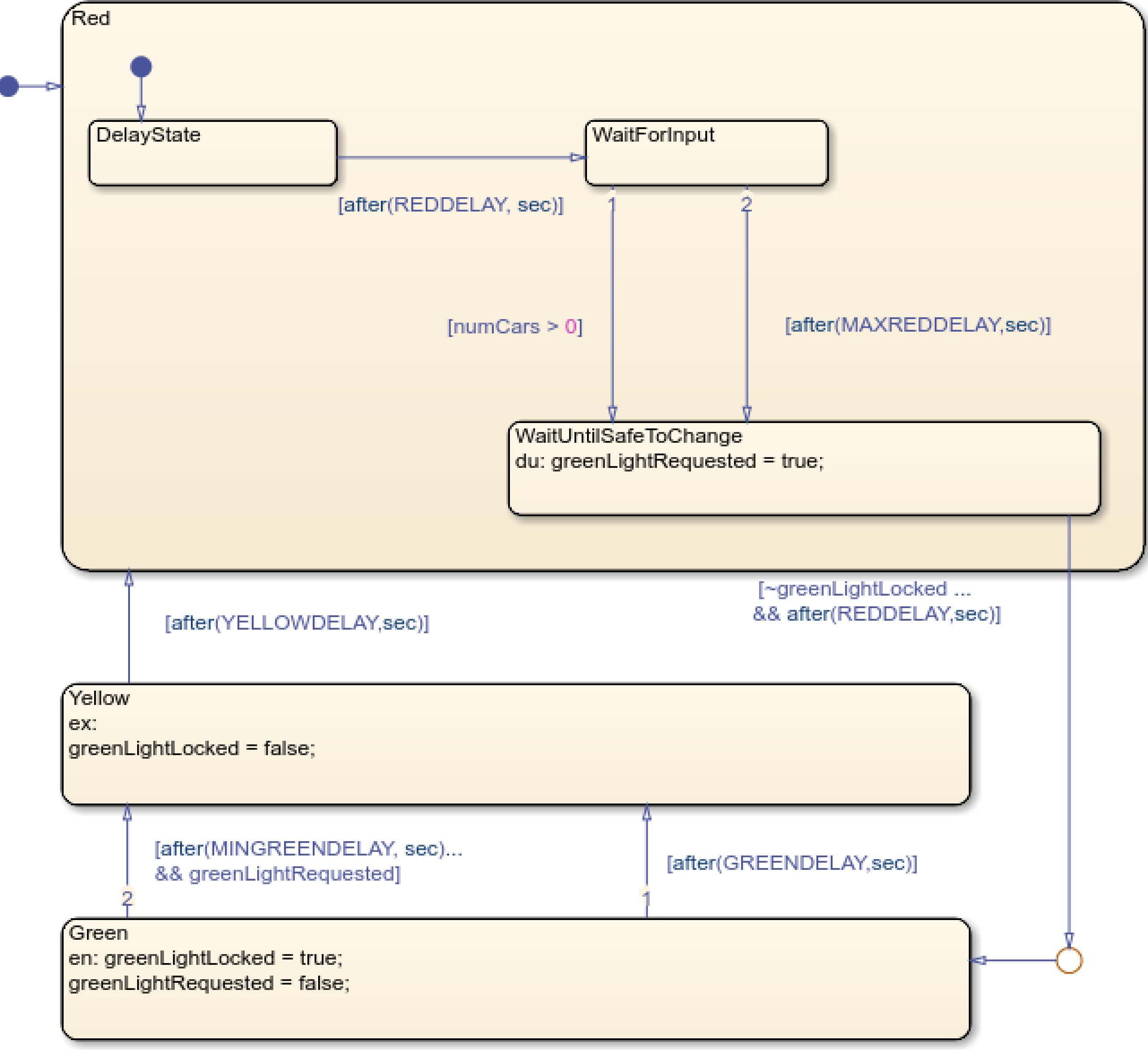
The Stateflow chart Traffic Controller manages two traffic controllers in parallel. Each controller determines the phase of the downstream traffic light based on traffic congestion at the intersection, an input from Simulink®, and mask parameters for the chart. For more information, see “Create a Mask to Share Parameters with Simulink” on page 25-36.



Active State Output

The child activity of both `Light1Controller` and `Light2Controller` is output to Simulink through a data of enumerated type. Stateflow manages this data automatically. For more information, see “Monitor State Activity Through Active State Data” on page 11-2.

- Open one of the Light controllers (for instance, `Light1Controller`).
- Right click and select Properties.
- Notice that the **Create output for monitoring** option is selected and set to `Child activity`.
- The field **Data name** corresponds to the name of the output data on the linked instance.
- This output is mapped to a chart level output called `Light1`.



Simulation

Simulate the model to see the traffic light blocks animate.

See Also

More About

- “Monitor Chart Activity by Using Active State Data”
- “Monitor State Activity Through Active State Data” on page 11-2
- “Model Distributed Traffic Control System by Using Messages” on page 13-20

Monitor Test Points in Stateflow Charts

This example shows how to specify data or states as test points that you can plot using a Floating Scope and Scope Viewer (Simulink) block during simulation. You can designate states or data with these properties as test points:

- **Scope** — Output or local
- **Size** — Scalar, one-dimensional, or two-dimensional
- **Type** — Any data type except `ml`
- **Location** — Descendant of a Stateflow chart

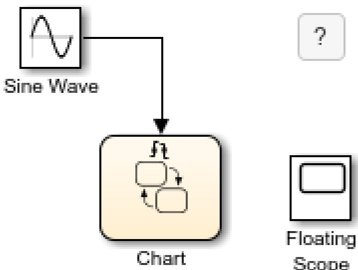
For more information about using test points in Simulink®, see “Configure Signals as Test Points” (Simulink).

Alternatively, you can log values for local, output, and active state data and view this logged output by using the Simulation Data Inspector or the Logic Analyzer. For more information, see “Log Simulation Output for States and Data” on page 11-13.

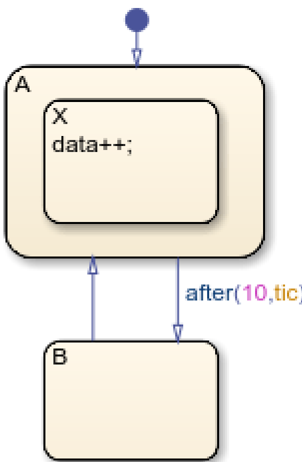
Set Test Points for Stateflow States and Data with the Property Inspector

You can specify individual data or states as test points by setting their **Test Point** property as described in “Specify Properties for States” on page 1-30 or “Set Data Properties” on page 10-5.

1. Open the model.



In the Stateflow chart, state `A` and its substate `X` become active on the first `tic` event. After 10 `tic` events, state `B` becomes active. On the next event, state `A` and substate `X` become active again and the cycle repeats.



The substate X is the parent of the local data data. Initially, data is equal to zero. While X is active, the entry and during actions increment data. After 10 tic events, when state B becomes active, data mantains a value of 10. Then, when state A and substate X become active again, the value of X reinitializes to zero and the cycle repeats.

2. Open the **Property Inspector**. In the **Modeling** tab, select **Property Inspector**.

3. Select state A. In the **Logging** section of the **Property Inspector**, select **Test Point**.

4. Repeat step 3 for state X and B.

5. Open the **Symbols** pane. In the **Modeling** tab, select **Symbols Pane**.

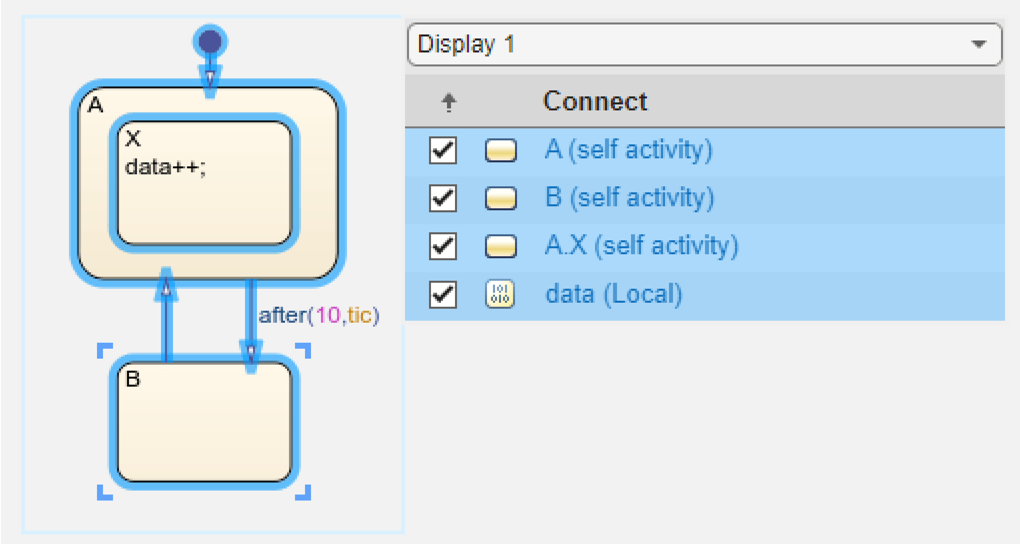
6. In the **Symbols** pane, under states A and X, select the local data data. Then, in the **Logging** section of the **Property Inspector**, select **Test Point**.

Monitor Data Values and State Self Activity Using a Floating Scope

1. In the Simulink model, open the Floating Scope block and select **Simulation** > **Signal Selector**.

2. Open the Stateflow chart.

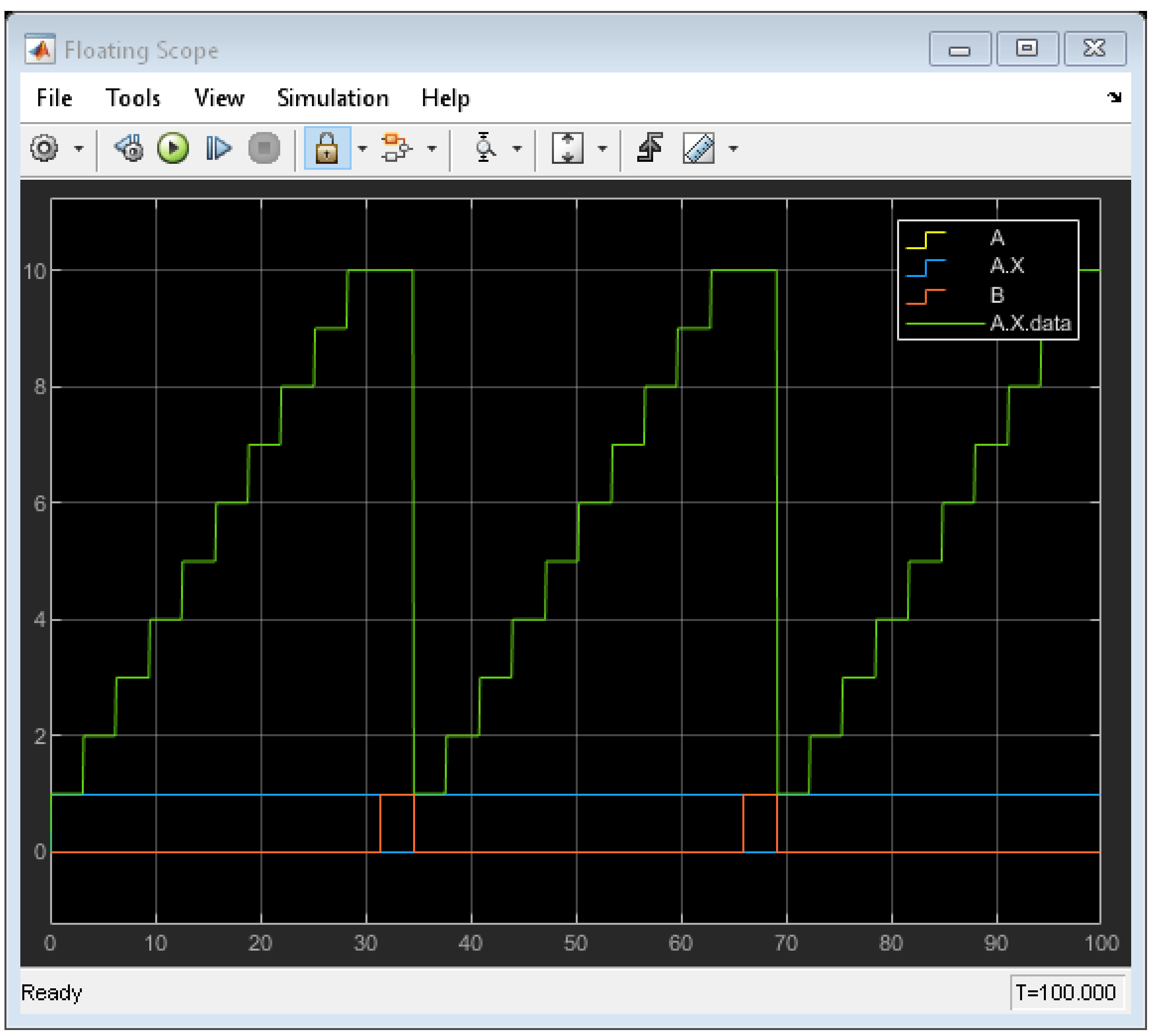
3. Open the Signal Selector by clicking and dragging a box around the chart.



4. In the Signal Selector, select the state or data to connect to the Floating Scope block. In this example, the states `A`, `B`, and `A.X` and the local data `data` are already selected in the Signal Selector.

5. Click the **X** button in the upper-right corner of the canvas.

6. Simulate the model.



See Also

Floating Scope and Scope Viewer

More About

- “Log Simulation Output for States and Data” on page 11-13
- “Represent Operating Modes by Using States” on page 1-26
- “Set Data Properties” on page 10-5
- “Configure Signals as Test Points” (Simulink)

Define Events

- “Synchronize Model Components by Broadcasting Events” on page 12-2
- “Set Properties for an Event” on page 12-5
- “Activate a Stateflow Chart by Sending Input Events” on page 12-8
- “Control States in Charts Enabled by Function-Call Input Events” on page 12-12
- “Activate a Simulink Block by Sending Output Events” on page 12-15
- “Broadcast Local Events to Synchronize Parallel States” on page 12-25
- “Control Chart Behavior by Using Implicit Events” on page 12-28
- “Yo-Yo Control of Satellites” on page 12-31

Synchronize Model Components by Broadcasting Events

An event is a Stateflow object that can trigger actions in one of these objects:

- A parallel state in a Stateflow chart
- Another Stateflow chart
- A Simulink triggered or function-call subsystem

For simulation purposes, there is no limit to the number of events in a Stateflow chart. However, for code generation, the underlying C compiler enforces a theoretical limit of 2^{31}-1 events.

Types of Events

An implicit event is a built-in event that is broadcast during chart execution. These events are implicit because you do not define or trigger them explicitly. For more information, see “Control Chart Behavior by Using Implicit Events” on page 12-28.

An explicit event is an event that you define explicitly. Explicit events can have one of these types.

Type	Description
Input Event	Event that is broadcast to a Stateflow chart from outside the chart. For more information, see “Activate a Stateflow Chart by Sending Input Events” on page 12-8 and “Design Human-Machine Interface Logic by Using Stateflow Charts” on page 31-24.
Local Event	Event that can occur anywhere in a Stateflow chart but is visible only in the parent object and its descendants. Local events are not supported in standalone Stateflow charts in MATLAB. For more information, see “Broadcast Local Events to Synchronize Parallel States” on page 12-25.
Output Event	Event that occurs in a Stateflow chart but is broadcast to a Simulink block. Output events are not supported in standalone Stateflow charts in MATLAB. For more information, see “Activate a Simulink Block by Sending Output Events” on page 12-15.

You can define local events at these levels of the Stateflow hierarchy.

Level of Hierarchy	Visibility
Chart	Local event is visible to the chart and all its states and substates.
Subchart	Local event is visible to the subchart and all its states and substates.
State	Local event is visible to the state and all its substates.

Define Events in a Chart

You can add events to a Stateflow chart by using the **Symbols** pane, the Stateflow Editor menu, or the Model Explorer.

Add Events Through the Symbols Pane

1. In the **Modeling** tab, under **Design Data**, select **Symbols Pane**.

2 Click the **Create Event** icon .
3 In the row for the new event, under **Type**, click the icon and choose:

   - `Input Event`
   - `Local Event`
   - `Output Event`
4 Edit the name of the event.
5 For input and output events, click the **Port** field and choose a port number.
6 To specify properties for the event, open the **Property Inspector**. In the **Symbols** pane, right-click the row for the event and select **Explore**. For more information, see "Set Properties for an Event" on page 12-5.

Add Events by Using the Stateflow Editor Menu

1 In a Stateflow chart in a Simulink model, select the menu option corresponding to the type of the event that you want to add.

Type	Menu Option
Input Event	In the **Modeling** tab, under **Design Data**, click **Event Input**.
Output Event	In the **Modeling** tab, under **Design Data**, click **Event Output**.
Local Event	In the **Modeling** tab, under **Design Data**, click **Local Event**.

2 In the Event dialog box, specify data properties. For more information, see "Set Properties for an Event" on page 12-5.

Add Events Through the Model Explorer

1 In the **Modeling** tab, under **Design Data**, select **Model Explorer**.
2 In the **Model Hierarchy** pane, select the object in the Stateflow hierarchy where you want to make the new event visible. The object that you select becomes the parent of the new event.
3 In the Model Explorer menu, select **Add > Event**. The new event with a default definition appears in the **Contents** pane of the Model Explorer.
4 In the **Event** pane, specify the properties of the event. For more information, see "Set Properties for an Event" on page 12-5.

Access Event Information from a Stateflow Chart

You can display the properties of an input or local event, or open the destination of an output event directly from a Stateflow chart. Right-click the state or transition that contains the event of interest and select **Explore**. A context menu lists the names and scopes of all resolved symbols in the state or transition. Selecting an input or local event from the context menu displays its properties in the Model Explorer. Selecting an output event from the context menu opens the Simulink subsystem or Stateflow chart associated with the event.

Best Practices for Using Events in Stateflow Charts

Use the send Command to Broadcast Explicit Events in Actions

To broadcast local or output events in state or transition actions, use the `send` operator. For example, to broadcast an output event when a transition is valid, avoid using the name of the event as a condition action.

```
{output_event;}
```

Instead, call the `send` operator.

```
{send(output_event);}
```

Although both actions are valid, using the `send` operator enhances readability of a chart and ensures that explicit events are not mistaken for data.

Avoid Using Explicit Events to Trigger Conditional Actions

Use conditions on transitions instead of events when you want to:

- Represent conditional statements, for example, `[x < 1]` or `[x == 0]`.
- Represent a change of data value, for example, `[hasChanged(x)]`.

Avoid Using the Implicit Event enter to Check State Activity

To check state activity, use the `in` operator instead of the implicit event `enter`. For more information, see “Check State Activity by Using the in Operator” on page 11-24.

Do Not Mix Edge-Triggered and Function-Call Input Events in a Chart

Mixing input events that use edge triggers and function calls results in a compile-time error.

See Also

`enter` | `in` | `send`

More About

- “Set Properties for an Event” on page 12-5
- “Broadcast Local Events to Synchronize Parallel States” on page 12-25
- “Activate a Stateflow Chart by Sending Input Events” on page 12-8
- “Activate a Simulink Block by Sending Output Events” on page 12-15
- “Guidelines for Naming Stateflow Objects” on page 1-66
- “Identify Data by Using Dot Notation” on page 10-39

Set Properties for an Event

An event is a Stateflow object that can trigger actions in a parallel state, another Stateflow chart, or a Simulink triggered or function-call subsystem. For more information, see “Synchronize Model Components by Broadcasting Events” on page 12-2.

When you create Stateflow charts in Simulink models, you can modify event properties in the **Property Inspector** or the Model Explorer.

To use the **Property Inspector**:

1. In the **Modeling** tab, under **Design Data**, select **Symbols Pane** and **Property Inspector**.
2. In the **Symbols** pane, select the event.
3. In the **Property Inspector**, edit the event properties.

To use the Model Explorer:

1. In the **Modeling** tab, under **Design Data**, select **Model Explorer**.
2. In the **Model Hierarchy** pane, select the parent of the event.
3. In the **Contents** pane, select the event.
4. In the **Dialog** pane, edit the event properties.

You can also modify these properties programmatically by using `Stateflow.Event` objects. For more information about the Stateflow programmatic interface, see “Overview of the Stateflow API”.

Stateflow Event Properties

Name

Name of the event. Actions reference events by their names. Names must begin with an alphabetic character, cannot include spaces, and cannot be shared by sibling events. For more information, see “Guidelines for Naming Stateflow Objects” on page 1-66.

Scope

Scope of the event. The scope specifies where the event occurs relative to the parent object.

Scope	Description
`Local`	Event that can occur anywhere in a Stateflow machine but is visible only in the parent object and its descendants. For more information, see “Broadcast Local Events” on page 12-25.
`Input from Simulink`	Event that occurs in a Simulink block but is broadcast to a Stateflow chart. For more information, see “Activate a Stateflow Chart by Sending Input Events” on page 12-8.
`Output to Simulink`	Event that occurs in a Stateflow chart but is broadcast to a Simulink block. For more information, see “Activate a Simulink Block by Sending Output Events” on page 12-15.

Port

Index of the port associated with the event. This property applies only to input and output events.

- For input events, port is the index of the input signal that triggers the event. For more information, see "Association of Input Events with Control Signals" on page 12-10.
- For output events, port is the index of the signal that outputs this event. For more information, see "Association of Output Events with Output Ports" on page 12-24.

Trigger

Type of signal that triggers an input or output event. For more information, see "Activate a Stateflow Chart by Sending Input Events" on page 12-8 and "Activate a Simulink Block by Sending Output Events" on page 12-15.

Debugger Breakpoints

Option for setting debugger breakpoints at the start or end of an event broadcast. Available breakpoints depend on the type of the event.

Type of Event	Start of Broadcast	End of Broadcast
Local Event	Available	Available
Input Event	Available	Not available
Output Event	Not available	Not available

For more information, see "Set Breakpoints to Debug Charts" on page 30-2.

Description

Description of the event.

Document Link

Link to online documentation for the event. You can enter a web URL address or a MATLAB command that displays documentation as an HTML file or as text in the MATLAB Command Window. When you click the **Document link** hyperlink, Stateflow displays the documentation.

See Also

Objects
`Stateflow.Event`

Tools
Model Explorer

More About

- "Synchronize Model Components by Broadcasting Events" on page 12-2
- "Broadcast Local Events" on page 12-25
- "Activate a Stateflow Chart by Sending Input Events" on page 12-8
- "Activate a Simulink Block by Sending Output Events" on page 12-15

- "Guidelines for Naming Stateflow Objects" on page 1-66
- "Identify Data by Using Dot Notation" on page 10-39

Activate a Stateflow Chart by Sending Input Events

An input event occurs outside a Stateflow chart but is visible only in that chart. This type of event enables other Simulink blocks, including other Stateflow charts, to notify a specific chart of events that occur outside it. To define an input event:

1. Add an event to the Stateflow chart, as described in "Define Events in a Chart" on page 12-2.
2. Set the **Scope** property for the event to `Input from Simulink`. A single trigger port appears at the top of the Stateflow block in the Simulink model.
3. An input event can activate a Stateflow chart through a change in a control signal (an edge trigger) or a function call from a Simulink block.

   - To specify an edge-triggered input event, set the **Trigger** property to one of these options:

     - `Rising`
     - `Falling`
     - `Either`
   - To specify a function-call input event, set the **Trigger** property to `Function call`.

   You cannot mix edge-triggered and function-call input events in the same Stateflow chart. Mixing these input events results in a compile-time error.

For more information, see "Synchronize Model Components by Broadcasting Events" on page 12-2.

Activate a Stateflow Chart by Using Edge Triggers

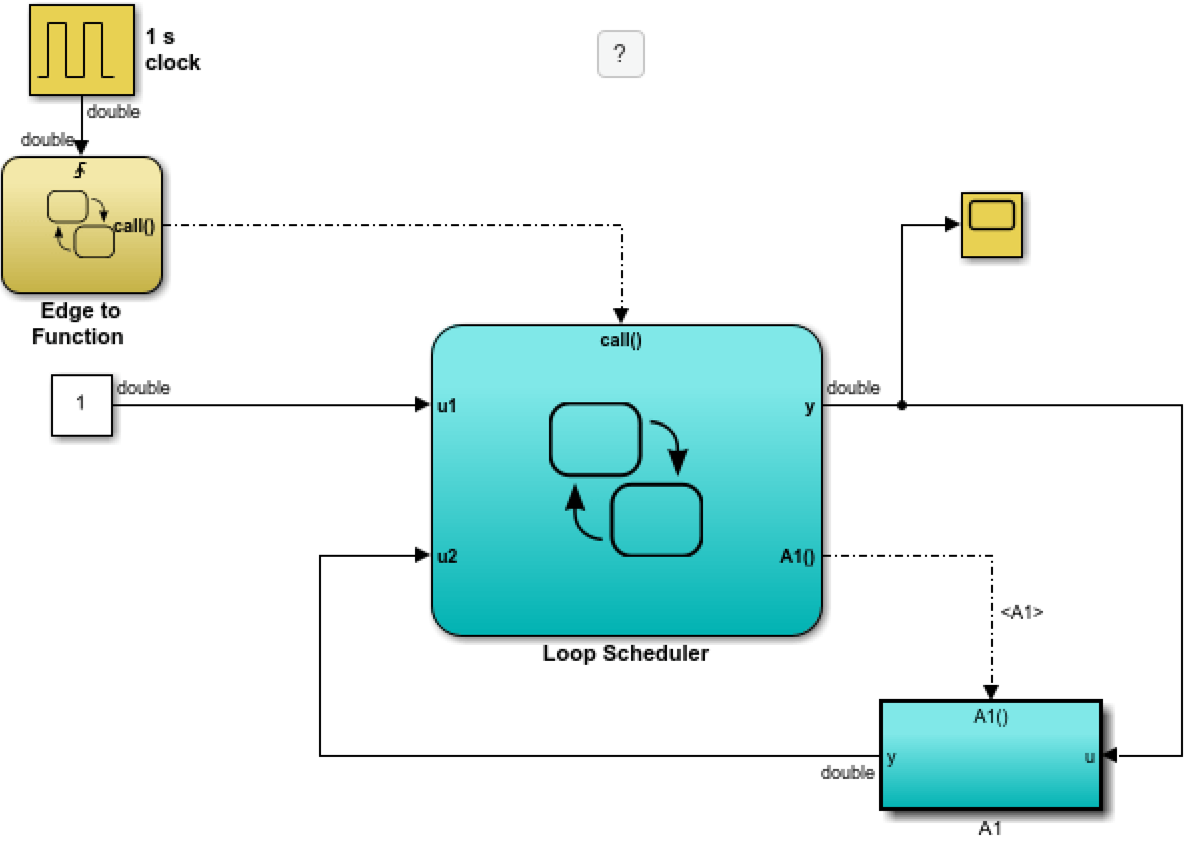
An edge-triggered input event causes a Stateflow chart to execute during the current time step of simulation. With this type of input event, a change in a control signal acts as a trigger.

Edge Trigger Type	Description
`Rising`	Rising edge trigger. Chart is activated when the control signal changes from either zero or a negative value to a positive value.
`Falling`	Falling edge trigger. Chart is activated when the control signal changes from a positive value to either zero or a negative value.
`Either`	Either rising or falling edge trigger. Chart is activated when the control signal crosses zero as it changes in either direction.

In all cases, the value of the control signal must cross zero to be a valid edge trigger. For example, a signal that changes from -1 to 1 is a valid rising edge trigger. A signal that changes from 1 to 2 is not a valid rising edge trigger.

When to Use Edge-Triggered Input Events

Use an edge-triggered input event to activate a chart when your model requires regular or periodic chart execution. For example, in this model, an edge-triggered input event activates the Edge to Function chart at regular intervals. For more information, see "Schedule a Subsystem Multiple Times in a Single Step" on page 27-6.



Behavior of Multiple Edge-Triggered Input Events

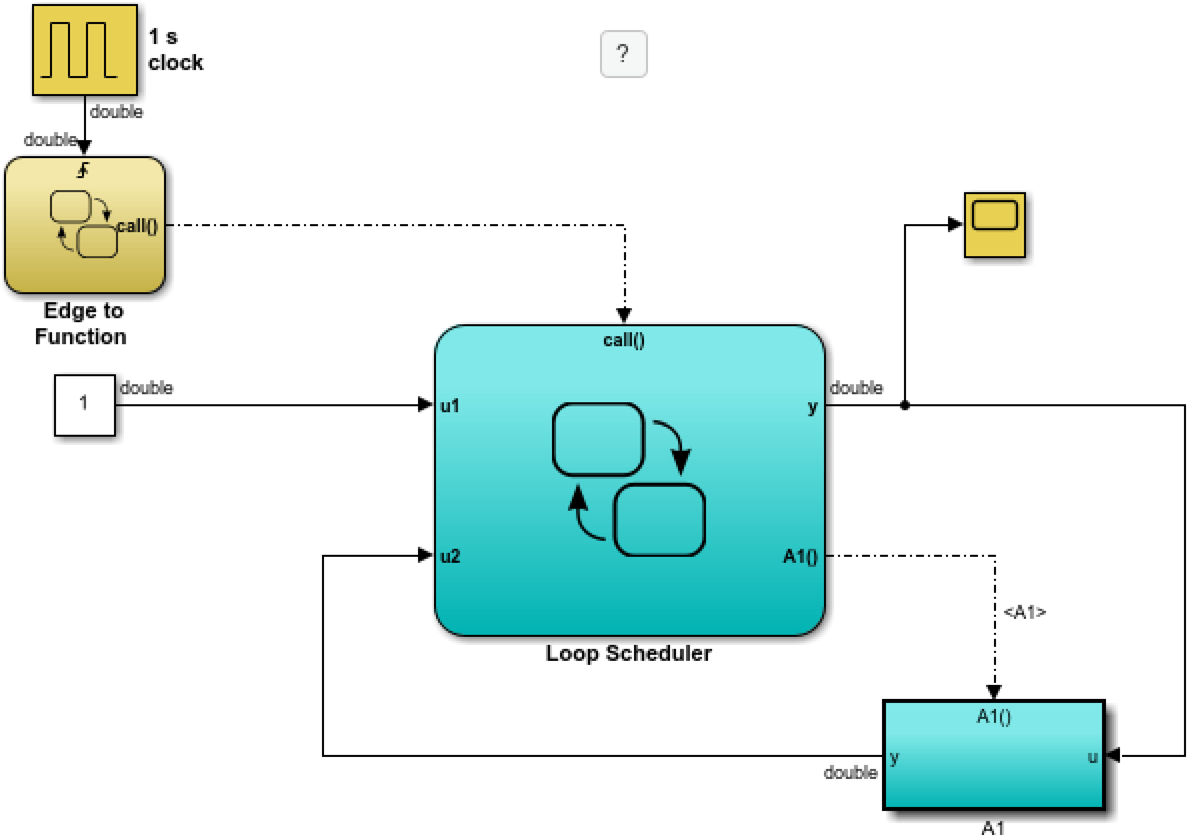
At any given time step, input events are checked in ascending order based on their port numbers. The chart awakens once for each valid event. For edge-triggered input events, multiple edges can occur in the same time step, waking the chart more than once in that time step. In this situation, the events wake the chart in ascending order based on their port numbers.

Activate a Stateflow Chart by Using Function Calls

A function-call input event causes a Stateflow chart to execute during the current time step of simulation. With this type of input event, you must also define a function-call output event for the block that calls the Stateflow chart.

When to Use Function-Call Input Events

Use a function-call input event to activate a chart when your model requires access to output data from the chart in the same time step as the function call. For example, in this model, a function-call input event activates the Looping Scheduler chart. For more information, see “Schedule a Subsystem Multiple Times in a Single Step” on page 27-6.



Behavior of Multiple Function-Call Input Events

For function-call input events, only one trigger event exists. The caller of the event explicitly calls and executes the chart. Only one function call is valid in a single time step.

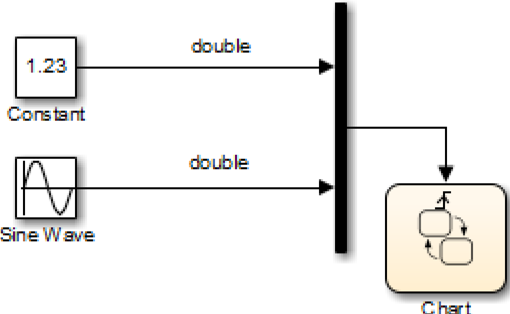
Association of Input Events with Control Signals

When you define one or more input events in a chart, a single trigger port appears on the top side of the chart block. Multiple external Simulink blocks can trigger the input events through a vector of signals connected to the trigger port. The **Port** property of an input event specifies the index into the control signal vector that connects to the trigger port.

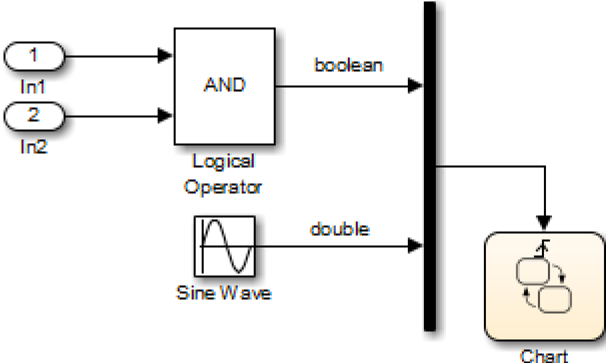
By default, **Port** values appear in the order that you add input events. You can change these assignments by modifying the **Port** property of the events. When you change the **Port** property for an input event, the **Port** values for the remaining input events automatically renumber.

Data Types Allowed for Input Events

For multiple input events to a trigger port, all signals must have the same data type. Using signals of different data types as input events results in an error during simulation. For example, you can mux two input signals of type `double` to use as input events to a chart.



You cannot mux two input signals of different data types, such as `boolean` and `double`.



See Also

More About

- “Synchronize Model Components by Broadcasting Events” on page 12-2
- “Set Properties for an Event” on page 12-5
- “Control States in Charts Enabled by Function-Call Input Events” on page 12-12
- “Schedule a Subsystem Multiple Times in a Single Step” on page 27-6
- “View Differences Between Stateflow Messages, Events, and Data” on page 13-14

Control States in Charts Enabled by Function-Call Input Events

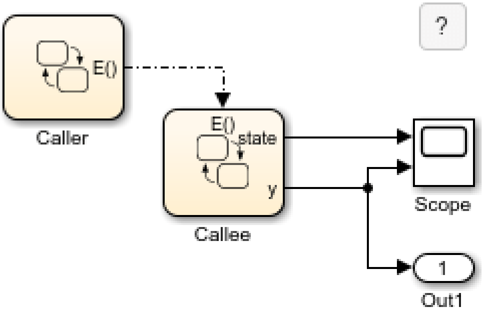
In a Simulink® model, when a Stateflow® chart is enabled by a function-call input event, you can control the state of the chart by setting the **States When Enabling** chart property. This property determines the values of states and data when the input event reenables the chart:

- `Held` — Maintain most recent values of the states and data.
- `Reset` — Revert to the initial values of the states and data.

For new charts, the default setting is `Held`. For more information, see “Activate a Stateflow Chart by Sending Input Events” on page 12-8.

Example of a Chart Enabled by a Function-Call Input Event

In this model, the `Caller` chart uses the event `E` to wake up and execute the `Callee` chart.



The `Caller` chart contains two states, `A` and `B`. When you bind the output event `E` in state `A`:

- Entering `A` enables the `Callee` chart.
- Exiting `A` disables the `Callee` chart.
- Reentering `A` reenables the `Callee` chart.

The temporal logic operator after changes the active state every ten time steps, so the `Callee` chart is repeatedly enabled and disabled.

The `Callee` chart contains two states, `C` and `D`. Each time that the chart executes, the output data `y` increments by one. The state `C` is initially active. After one time step, the value of `y` is positive and the chart takes the transition to state `D`.

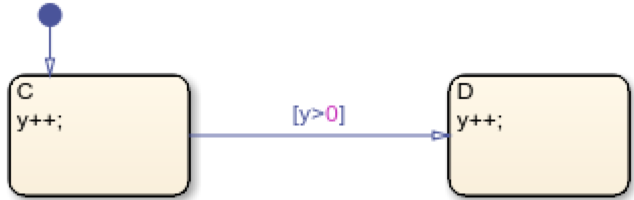


Chart Simulation When Property Is `Held`

In the `Callee` chart, the **States When Enabling** property is set to `Held`. During simulation, when the function-call input event reenables the chart at times $t = 20$ and $t = 40$, state `D` stays active and output `y` maintains its most recent value.

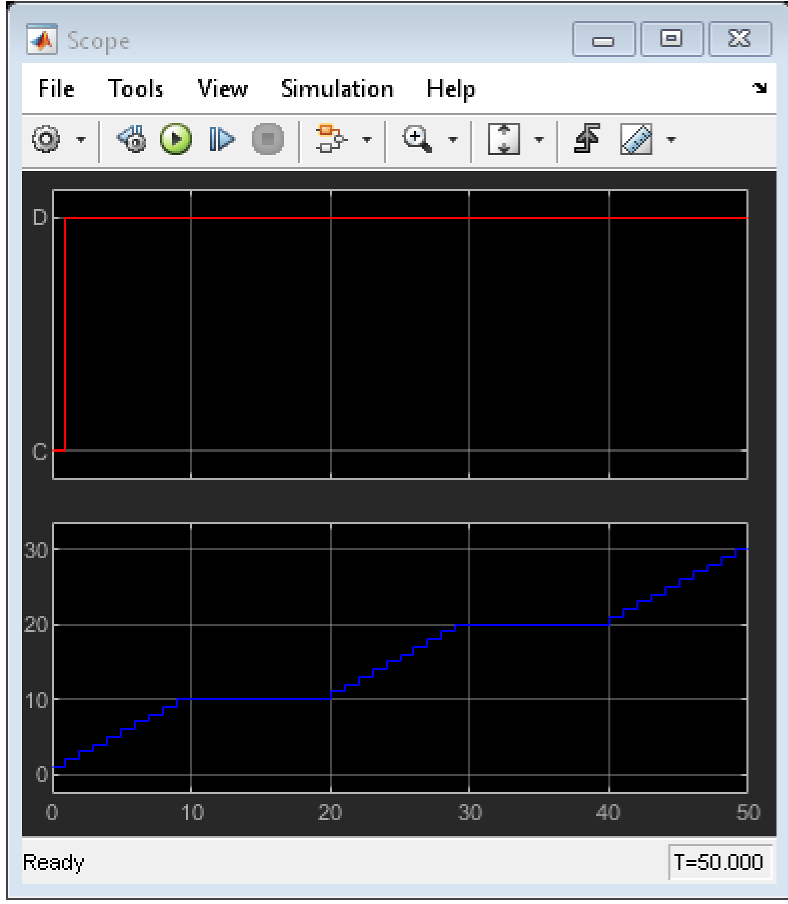
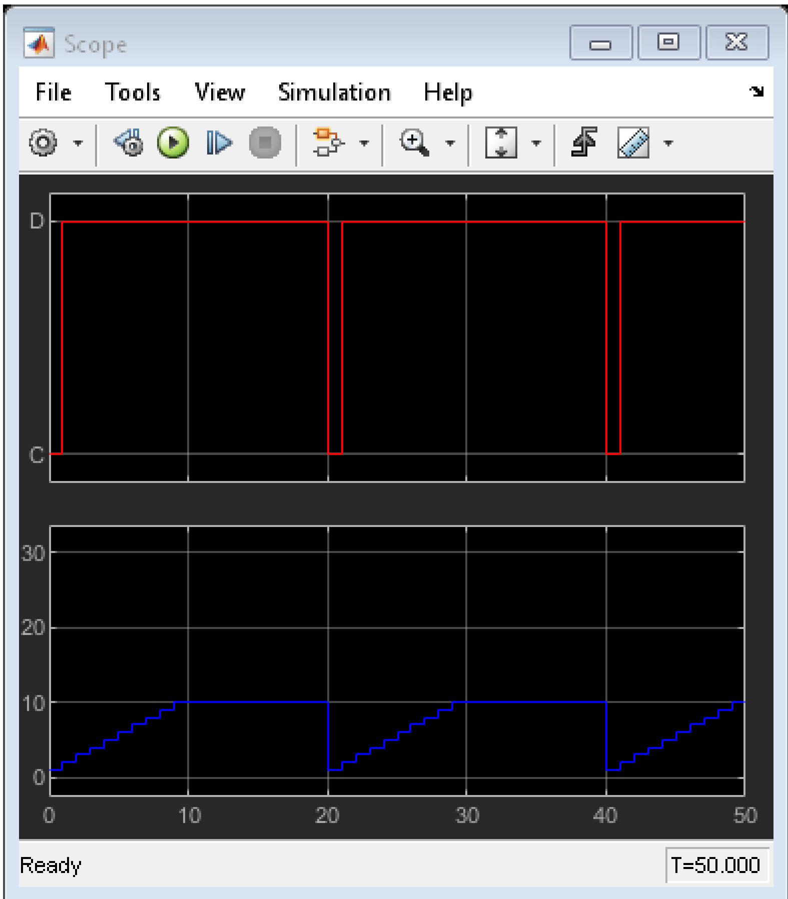


Chart Simulation When Property is `Reset`

In the `Callee` chart, change the **States When Enabling** property to `Reset`. During simulation, when the function-call input event reenables the chart at times $t = 20$ and $t = 40$, state `C` becomes active and output `y` reverts to its initial value of zero.



See Also

More About

- “Synchronize Model Components by Broadcasting Events” on page 12-2
- “Activate a Stateflow Chart by Sending Input Events” on page 12-8
- “Model References” (Simulink)

Activate a Simulink Block by Sending Output Events

An output event is an event that occurs in a Stateflow chart but is visible in Simulink blocks outside the chart. This type of event enables a chart to notify other blocks in a model about events that occur in the chart. To define an output event:

1. Add an event to the Stateflow chart, as described in "Define Events in a Chart" on page 12-2.
2. Set the **Scope** property for the event to `Output to Simulink`. For each output event that you define, an output port appears on the Stateflow block.
3. An output event can activate other blocks in the model through a change in a control signal (an edge trigger) or a function call to a Simulink block.

   - To specify an edge-triggered output event, set the **Trigger** property to `Either Edge`.
   - To specify a function-call output event, set the **Trigger** property to `Function call`.

For more information, see "Synchronize Model Components by Broadcasting Events" on page 12-2.

Broadcast Output Events

To broadcast output events from one chart to another, use the operator `send`. The format of an output event broadcast is

```
send(event_name)
```

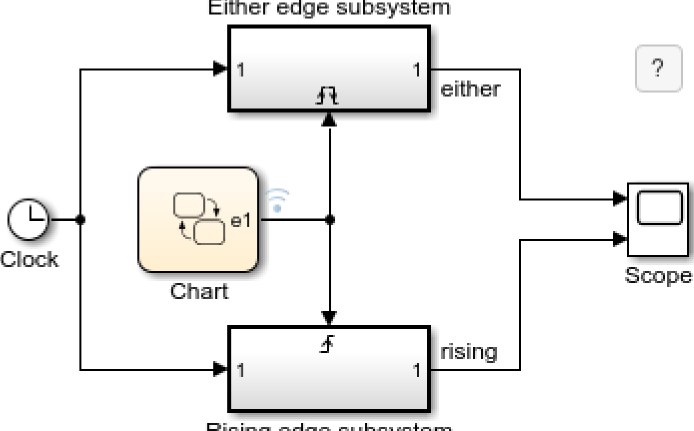
where *`event_name`* is an output event.

Activate a Simulink Block by Using Edge Triggers

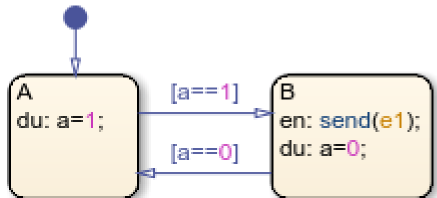
An edge-triggered output event activates a Simulink block to execute during the current time step of simulation. With this type of output event, a change in a control signal acts as a trigger. For more information, see "Using Triggered Subsystems" (Simulink).

When to Use Edge-Triggered Output Events

To activate a Simulink subsystem when your model requires regular or periodic subsystem execution, use an edge-triggered output event. For example, this model uses an edge-triggered output event to activate two triggered subsystems at regular intervals.

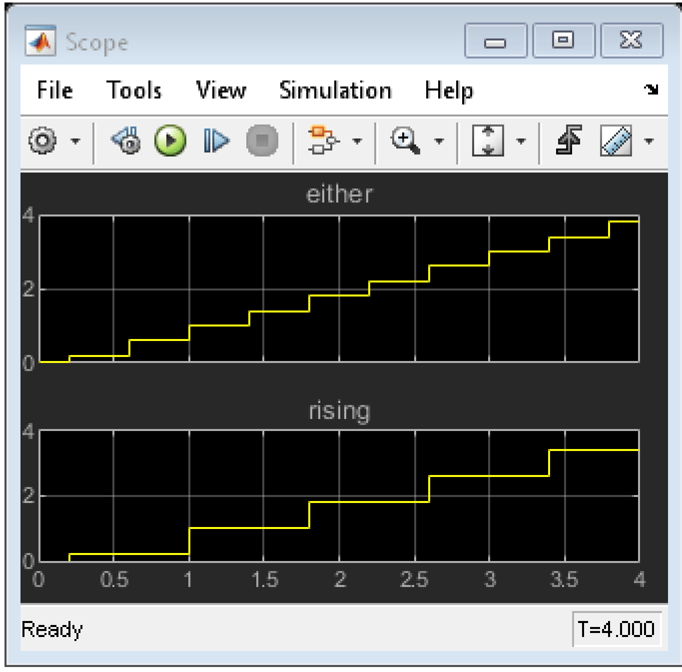


The chart contains the edge-triggered output event `e1` which alternates between 0 and 1 during simulation.



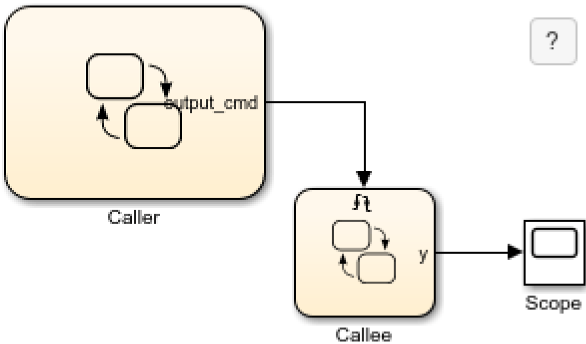
In a Stateflow chart, the **Trigger** property of an edge-triggered output event is always `Either Edge`. Simulink triggered subsystems can have a `Rising`, `Falling`, or `Either` edge trigger. The model shows the difference between triggering an `Either` edge subsystem from a `Rising` edge subsystem:

- The output event triggers the `Either` edge subsystem on every broadcast. The trigger occurs when the event signal switches from 0 to 1 or from 1 to 0.
- The output event triggers the `Rising` edge subsystem on every other broadcast. The trigger occurs only when the event signal switches from 0 to 1.

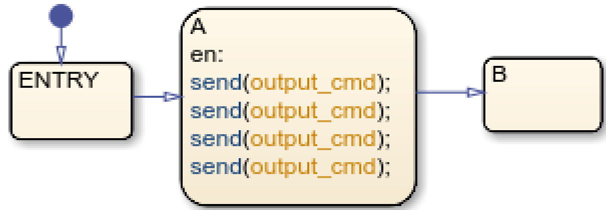


Queuing Behavior of Multiple Edge-Triggered Output Events

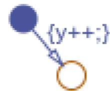
A chart dispatches only one broadcast of an edge-triggered output event for each time step. When there are multiple broadcasts in a single time step, the chart dispatches one broadcast and queues up the remaining broadcasts for dispatch in successive time steps. For example, in this model, the Caller chart uses the edge-triggered output event `output_cmd` to activate the Callee chart.



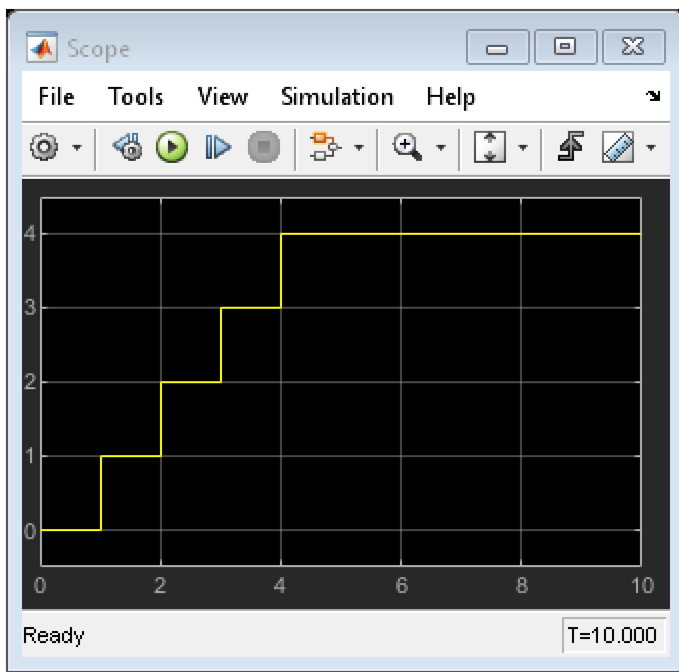
The Caller chart tries to broadcast the same edge-triggered output event four times in a single time step.



Each time the Callee chart is activated, the output data y increments by one.



When you simulate the model, at time $t = 1$, the Caller chart dispatches one of the four output events. The Callee chart executes once during that time step. The Caller chart queues up the other three event broadcasts for future dispatch at a time $t = 2$, $t = 3$, and $t = 4$. As a result, the value of y grows in increments of one at time $t = 1$, $t = 2$, $t = 3$, and $t = 4$.



Activate a Simulink Block by Using Function Calls

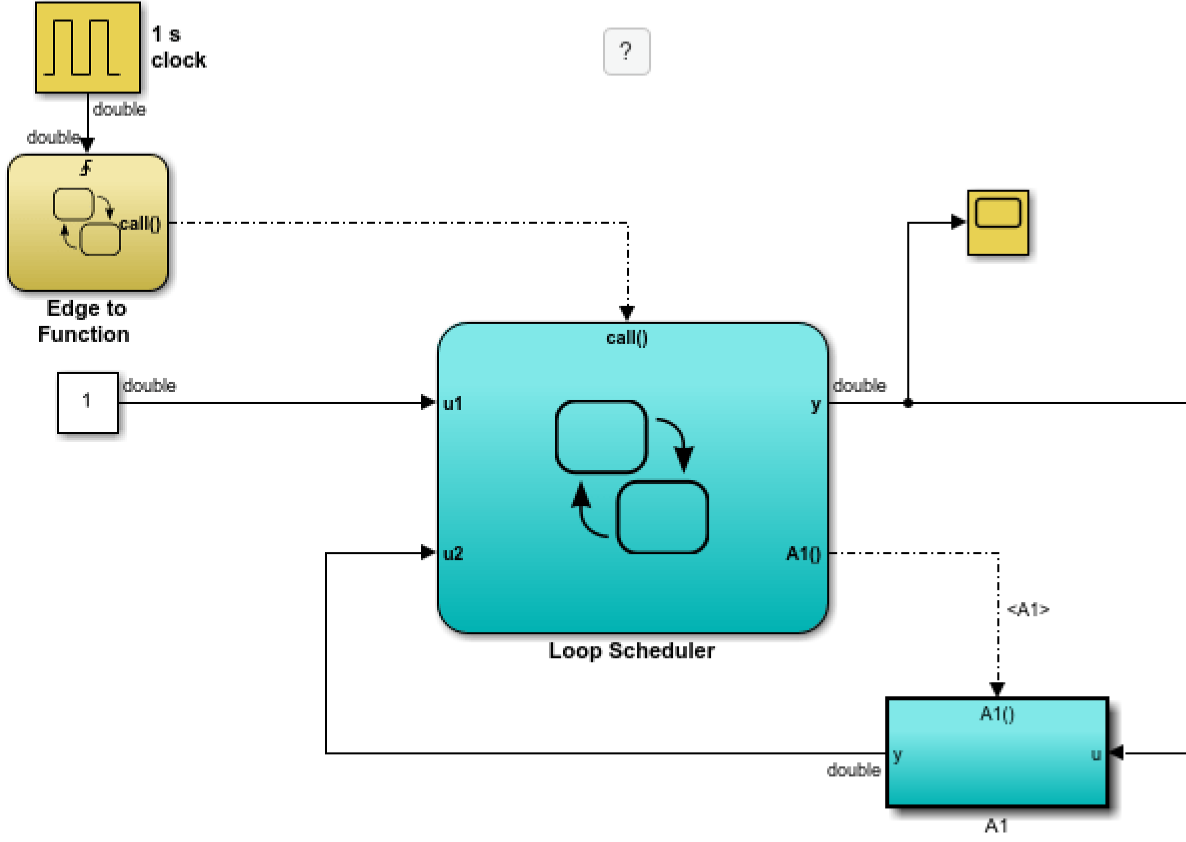
A function-call output event activates a Simulink block to execute during the current time step of simulation. This type of output event works only on blocks that you can trigger with a function call. For more information, see “Using Function-Call Subsystems” (Simulink).

When to Use Function-Call Output Events

Use a function-call output event to activate a Simulink block when your model requires access to output data from the block in the same time step as the function call. For example, this model contains two function-call output events:

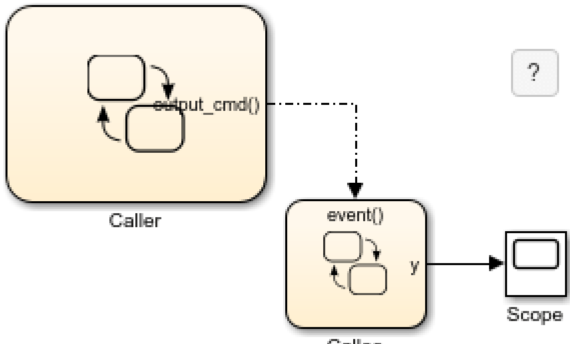
- In the Edge to Function chart, the output event `call` activates the Looping Scheduler chart.
- In the Looping Scheduler chart, the output event `A1` activates a Simulink subsystem.

For more information, see “Schedule a Subsystem Multiple Times in a Single Step” on page 27-6.

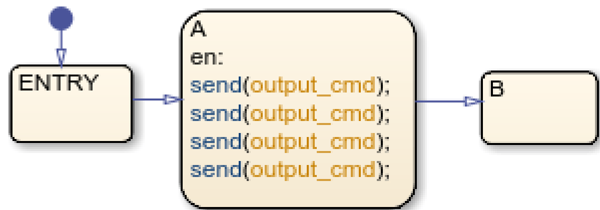


Interleaving Behavior of Multiple Function-Call Output Events

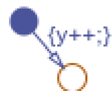
When there are multiple broadcasts of a function-call output event in a single time step, the chart dispatches all the broadcasts in that time step. Execution of function-call subsystems is interleaved with the execution of the chart, so that output from the function-call subsystem is available immediately in the chart. For example, in this model, the Caller chart uses the function-call output event `output_cmd` to activate the Callee chart.



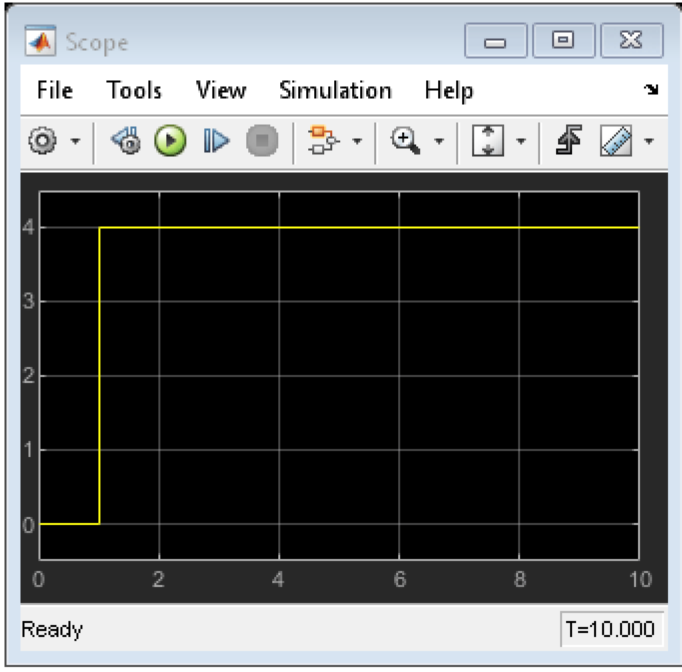
The Caller chart tries to broadcast the same function-call output event four times in a single time step.



Each time the Callee chart is activated, the output data y increments by one.



When you simulate the model, the Caller chart dispatches all four output events at time $t = 1$. The Callee chart executes four times during that time step. Execution of the Callee chart is interleaved with execution of the Caller chart so that output from the Callee chart is immediately available. As a result, the value of | y | increases from 0 to 4 at time $t = 1$.



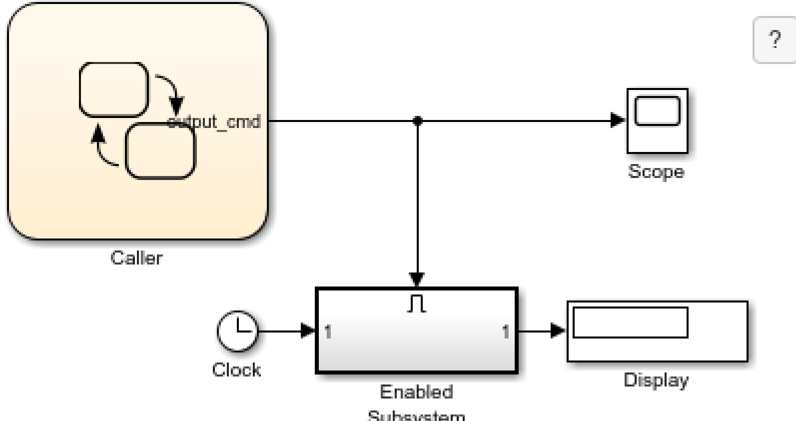
Approximate a Function Call by Using Edge-Triggered Events

If you cannot use a function-call output event, such as for HDL code generation, you can approximate a function call by using:

- An edge-triggered output event
- An enabled subsystem
- Two consecutive event broadcasts

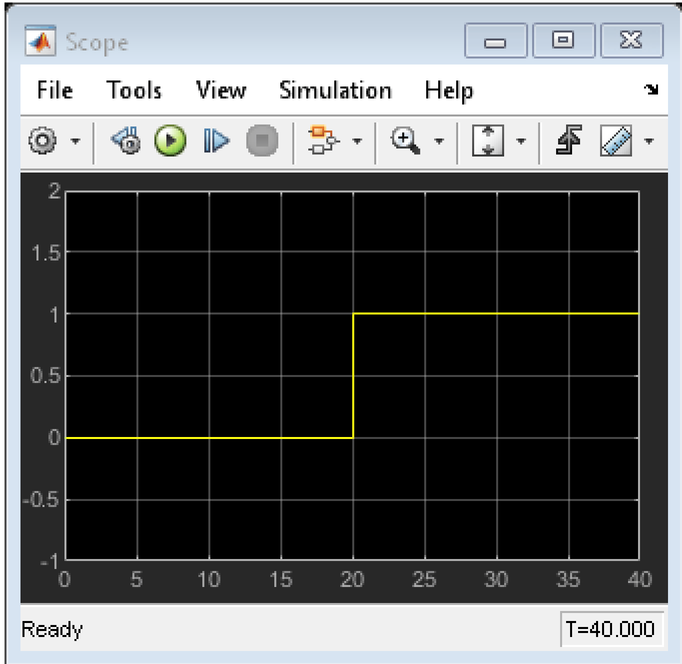
The queuing behavior of consecutive edge-triggered output events enables you to approximate a function call with an enabled subsystem.

For example, in this model, the edge-triggered output event `output_cmd` activates the enabled subsystem.

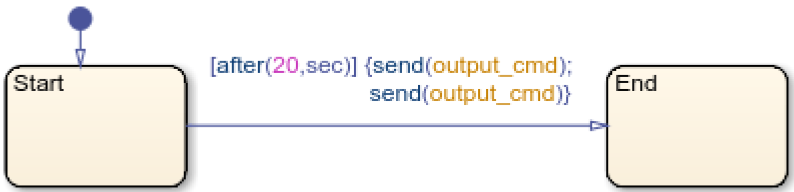


The Caller chart broadcasts the edge-triggered output event by using the `send` operator.

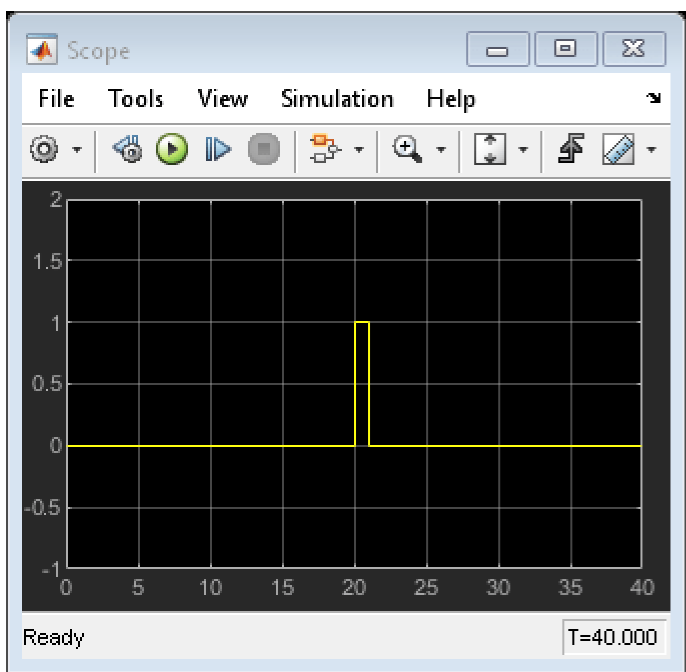
When simulation starts, the value of the trigger signal is 0. At time t = 20, the chart dispatches `output_cmd`, changing the value of the trigger signal to 1. The enabled subsystem becomes active and executes during that time step. Because no other event broadcasts occur, the enabled subsystem continues to execute at every time step until simulation ends at t= 40. The Display block shows a final value of 40.



To approximate a function call, add a second event broadcast in the same action.



When simulation starts, the value of the trigger signal is 0. At time *t* = 20, the chart dispatches `output_cmd`, changing the value of the trigger signal to 1. The enabled subsystem becomes active and executes during that time step. The chart queues up the second event for dispatch at the next time step. At time *t*= 21, the chart dispatches the second output event, which changes the value of the trigger signal back to 0. The enabled subsystem stops executing and the Display block shows a final value of 20.



Although you can approximate a function call, there is a subtle difference in execution behavior. Execution of a function-call subsystem occurs *during* execution of the chart action that provides the trigger. Execution of an enabled subsystem occurs *after* execution of the chart action is complete.

Association of Output Events with Output Ports

When you define an output event in a chart, an output event port appears on the right side of a chart block. Output events must be scalar, but you can define multiple output events in a chart. The **Port** property of an output event specifies the position of the output port.

By default, **Port** values appear in the order in which you add output events. You can change these assignments by modifying the **Port** property of the events. When you change the **Port** property for an output event, the **Port** values for the remaining output events automatically renumber.

See Also

`send`

More About

- “Synchronize Model Components by Broadcasting Events” on page 12-2
- “Set Properties for an Event” on page 12-5
- “Schedule a Subsystem Multiple Times in a Single Step” on page 27-6
- “View Differences Between Stateflow Messages, Events, and Data” on page 13-14
- “Using Triggered Subsystems” (Simulink)
- “Using Function-Call Subsystems” (Simulink)

Broadcast Local Events to Synchronize Parallel States

A local event is an event that occurs in a Stateflow chart and is visible only in the chart. This type of event enables parallel (AND) states in the same chart to synchronize with one another, so that actions in one state trigger actions in the other state. An action in one chart cannot broadcast local events to states in another chart. To define a local event:

1. Add an event to the Stateflow chart, as described in “Define Events in a Chart” on page 12-2.
2. Set the **Scope** property for the event to `Local`.

Local events are not supported in standalone Stateflow charts in MATLAB. For more information, see “Synchronize Model Components by Broadcasting Events” on page 12-2.

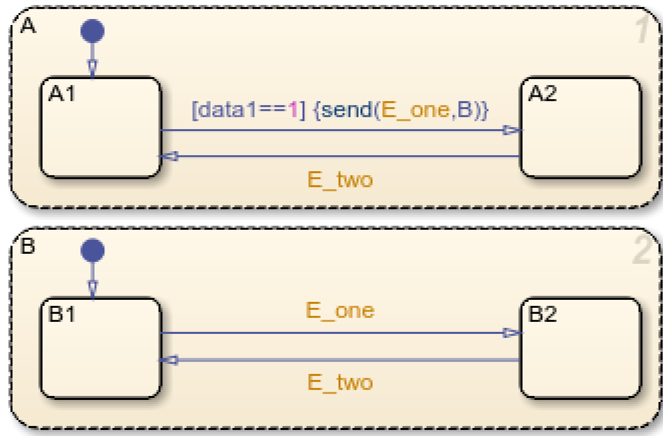
Broadcast Local Events

A directed event broadcast sends a local event directly from one state to another by using the operator `send`:

```
send(event_name,state_name)
```

`event_name` is a local event and *`state_name`* is a receiving state. The local event is broadcast directly to the receiving state and any of its substates. The local event must be visible to both the sending state and the receiving state. The receiving state must be active during the event broadcast.

For example, this chart contains two parallel (AND) states, `A` and `B`. The local event `E_one` belongs to the chart and is visible to both states. In state `A`, the transition from substate `A1` to substate `A2` uses a directed event broadcast of the form `send(E_one,B)` to send the local event `E_one` to state `B`. In `B`, the event triggers the transition from substate `B1` to substate `B2`. Therefore, the active substates in `A` and `B` are synchronized. For more information on the semantics of this example, see “Directed Event Broadcast Using Send” on page A-35.



The *`state_name`* argument can include a full hierarchy path to the state. For example, if the state `A` contains the state `A1`, you can send an event `E` to state `A1` with this broadcast:

```
send(E,A.A1)
```

Tip Do not include the chart name in the full hierarchy path to a state.

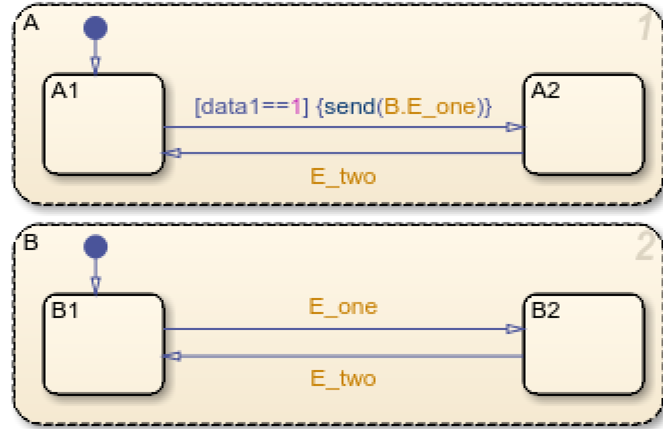
Use Qualified Event Names in Event Broadcasts

To broadcast a local event that is not visible to the sending state, use the operator `send` with a qualified event name:

```
send(state_name.event_name)
```

`event_name` is a local event that is owned by the receiving state *`state_name`*. The local event is broadcast directly to the receiving state and any of its substates. The local event is visible to the receiving state, but not to the sending state. The receiving state must be active during the event broadcast.

For example, this chart contains two parallel (AND) states, `A` and `B`. The local event `E_one` belongs to state `B` and is visible only to that state. In state `A`, the transition from substate `A1` to substate `A2` uses a directed event broadcast of the form `send(B.E_one)` to send the local event `E_one` to state `B`. In `B`, the event triggers the transition from substate `B1` to substate `B2`. Therefore, the active substates in `A` and `B` are synchronized. For more information on the semantics of this example, see “Directed Event Broadcast Using Qualified Event Name” on page A-35.



The *`state_name`* argument can include a full hierarchy path to the receiving state. Do not use the chart name in the full path name of the state. For example, suppose that the state `A` contains the state `A1`, and that `A1` owns the local event `E`. You can send event `E` to state `A1` with this broadcast:

```
send(A.A1.E)
```

Undirected Event Broadcasts

An undirected event broadcast sends a local event to all states in which it is visible by using the name of the event as a condition action:

```
event_name;
```

or by calling the operator `send` without specifying a receiving state:

```
send(event_name)
```

`event_name` is a local event that is visible to the sending state.

When possible, use directed event broadcasts instead of undirected event broadcasts. Directed event broadcasts prevent unwanted recursion during simulation and improve the efficiency of generated code. For more information, see “Avoid Unwanted Recursion in a Chart” on page 30-43.

Diagnostic for Detecting Undirected Local Event Broadcasts

During simulation, Stateflow charts can detect undirected local event broadcasts. To control the level of diagnostic action, open the Configuration Parameters dialog box and, in the **Diagnostics > Stateflow** pane, set the **Undirected event broadcasts** parameter to `none`, `warning`, or `error`. The default setting is `warning`. For more information, see “Undirected event broadcasts” (Simulink).

See Also

`send`

More About

- “Synchronize Model Components by Broadcasting Events” on page 12-2
- “Set Properties for an Event” on page 12-5
- “Broadcast Local Events in Parallel States” on page A-35

Control Chart Behavior by Using Implicit Events

Implicit events are built-in events that occur during chart execution when:

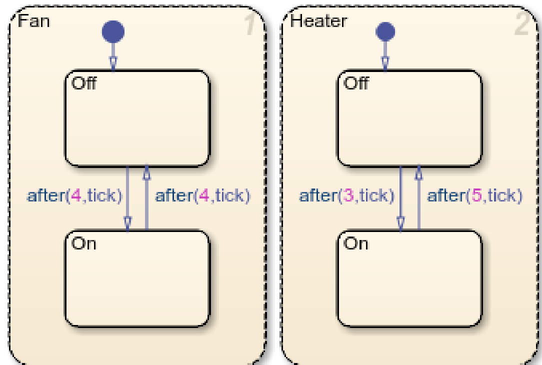
- The chart wakes up.
- The chart enters a state and the state becomes active.
- The chart exits a state and the state becomes inactive.
- The chart assigns a value to an internal data object.

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.

Implicit Events Based on Chart Execution

The keyword `tick` specifies the implicit event generated when a chart wakes up in a discrete-time simulation.

For example, in this chart, `Fan` and `Heater` are parallel (AND) states. Each state has a pair of substates, `On` and `Off`. Initially, the substates `Fan.Off` and `Heater.Off` are active. Each time the chart wakes up, it generates a `tick` event. The third `tick` triggers the transition from `Heater.Off` to `Heater.On`. Similarly, the fourth `tick` triggers the transition from `Fan.Off` to `Fan.On`. On the eighth `tick`, the chart transitions back to `Fan.Off` and `Heater.Off`.



For information about the temporal logic operator `after`, see “Control Chart Execution by Using Temporal Logic” on page 14-35.

Note The `tick` event refers to the chart containing the action being evaluated. The event cannot refer to a different chart.

Implicit Events Based on Data and States

In Stateflow charts in Simulink models, these operators generate implicit events when a chart sets the value of a variable or when a chart enters or exits a state.

Operator	Syntax	Description	Example

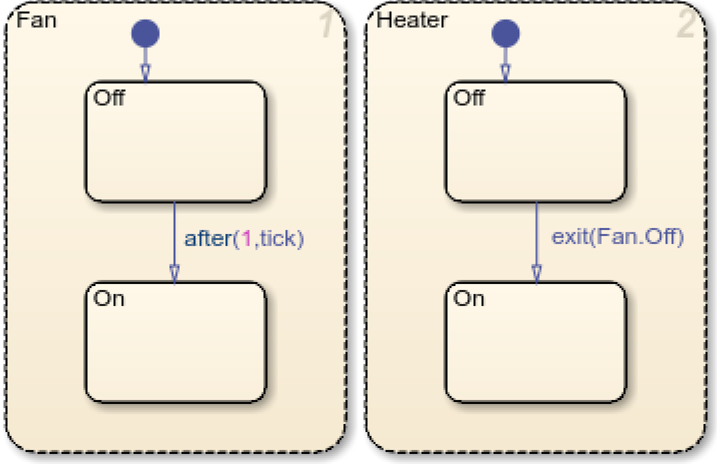
`change`	`change(`*`data_name`*`)`
	
	`chg(`*`data_name`*`)`
		Generates an implicit local event when the chart sets the value of the variable *`data_name`*.	Define an implicit local event when a state or transition action writes a value to the variable `Engine.rpm`.
			
			`change(Engine.rpm)`

`enter`	`enter(`*`state_name`*`)`
	
	`en(`*`state_name`*`)`
		Generates an implicit local event when the specified state *`state_name`* becomes active.	Define an implicit local event when the chart execution enters the state `Fan.On`.
			
			`enter(Fan.On)`

`exit`	`exit(`*`state_name`*`)`
	
	`ex(`*`state_name`*`)`
		Generates an implicit local event when the specified state *`state_name`* becomes inactive.	Define an implicit local event when the chart execution exits the state `Fan.Off`.
			
			`exit(Fan.Off)`

If more than one state or data object has the same name, use dot notation to qualify the name of the state. For more information, see "Identify Data by Using Dot Notation" on page 10-39.

For example, in this chart, `Fan` and `Heater` are parallel (AND) states. Each state has a pair of substates, `On` and `Off`. Initially, the substates `Fan.Off` and `Heater.Off` are active. When the chart wakes up, it generates a `tick` event that triggers the transition from `Fan.Off` to `Fan.On`. When the `Fan.Off` becomes inactive, the chart generates another implicit event that triggers the transition from `Heater.Off` to `Heater.On`. When the chart execution ends, the substates `Fan.On` and `Heater.On` are active.



Note If the same implicit event triggers multiple transitions in parallel states, the order in which the transitions execute does not necessarily match the execution order of the parallel states. To avoid unexpected behavior and ensure that the transitions execute in the order specified for the parallel states, do not use implicit events. Instead, use transition conditions that call operators such as `in` or `hasChanged`. For more information, see "Check State Activity by Using the in Operator" on page 11-24 and "Detect Changes in Data and Expression Values" on page 14-63.

See Also

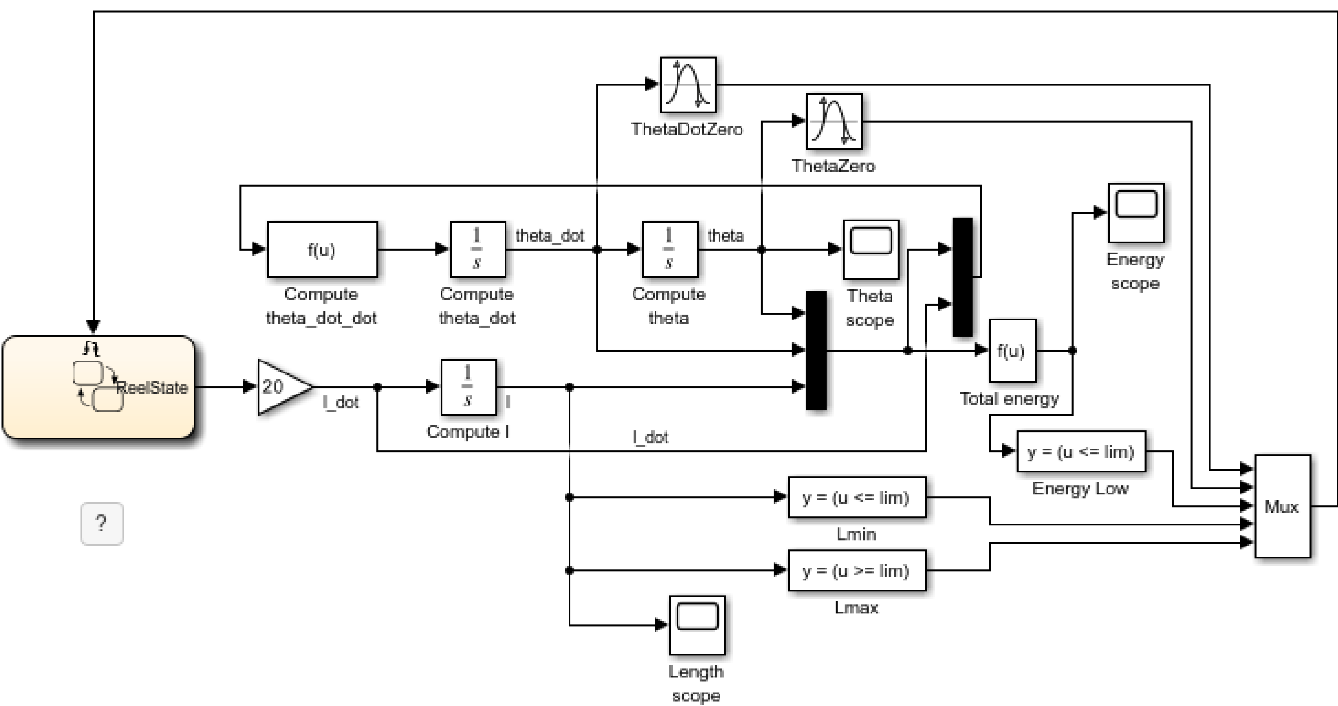
`after` | `change` | `enter` | `exit` | `hasChanged` | `in`

More About

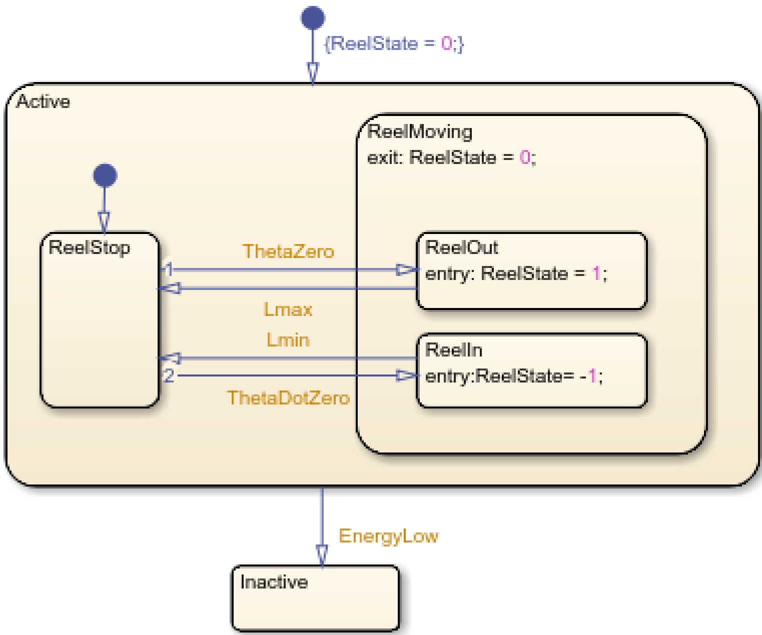
- “Use Events to Execute Charts” on page 2-40
- “Detect Changes in Data and Expression Values” on page 14-63
- “Check State Activity by Using the in Operator” on page 11-24
- “Control Chart Execution by Using Temporal Logic” on page 14-35

Yo-Yo Control of Satellites

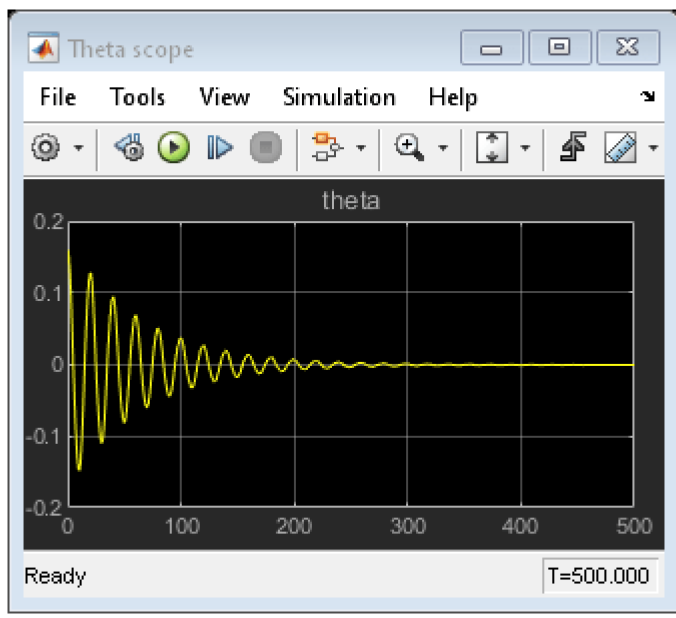
This example shows how to model the control system in a tethered satellite system. The satellite system consists of a small satellite attached by a long tether to an orbiting platform. When the tethered satellite oscillates, it behaves like a pendulum that exhibits too much libration. To stabilize the pendulum, the control system changes the length of the tether by reeling it out to its maximum length when the satellite is in the middle of its arc (which decreases its angular acceleration) and by reeling it in when the satellite has an angular velocity equal to zero.



Stateflow® is used to control when the tether is reeled in or reeled out using input events from Simulink®.



When the satellite is in the middle of its swing ($\theta = 0$), the state `ReelOut` becomes active. When the satellite has been reeled out as far as it can, the state `ReelStop` becomes active. When the angular velocity of the satellite reaches zero, the `ReelIn` state becomes active. When the tether is as short as possible, the `ReelStop` state becomes active once again. Finally, if the total energy of the satellite is too low, the system is deactivated by entering the `Inactive` state.



References

[1] Dabney, James B. and Harman, Thomas L. *Mastering Simulink*, 2003.

See Also

More About

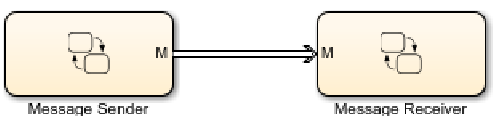
- "Synchronize Model Components by Broadcasting Events" on page 12-2

Messages

- “Communicate with Stateflow Charts by Sending Messages” on page 13-2
- “Set Properties for a Message” on page 13-5
- “Control Message Activity in Stateflow Charts” on page 13-9
- “View Differences Between Stateflow Messages, Events, and Data” on page 13-14
- “Model Distributed Traffic Control System by Using Messages” on page 13-20
- “Use the Sequence Viewer to Visualize Messages, Events, and Entities” on page 13-24
- “Build a Shared Communication Channel with Multiple Senders and Receivers” on page 13-33
- “Model Wireless Message Communication with Packet Loss and Channel Failure” on page 13-39
- “Model an Ethernet Communication Network with CSMA/CD Protocol” on page 13-49

Communicate with Stateflow Charts by Sending Messages

To communicate within and between Stateflow charts in a Simulink model, use messages. A message is a Stateflow object that communicates data locally or between charts. From a sender chart, you can send or forward a message containing data. In the receiving chart, a queue receives the message and holds it until the chart can evaluate it.



Messages combine some of the functionality of data and events. Like data, messages can transmit numeric and textual information. Like events, messages can trigger transition and state actions. However:

- Messages do not trigger charts to wake up. Instead, messages are queued until the chart wakes up. When the chart wakes up, it can respond to the messages in the queue.
- Messages are not lost if the receiver chart cannot respond immediately.

For more information, see “View Differences Between Stateflow Messages, Events, and Data” on page 13-14.

When a chart transition or state action evaluates a message, the chart determines if the queue contains any messages. If it does, the chart removes the message from the queue. The message remains valid until the end of the time step or until the chart forwards or discards it. While the message is valid, other transitions or actions can access the message data and the chart does not remove another message from the queue. The chart destroys all valid messages at the end of the current time step.

Define Messages in a Chart

You can add messages to a Stateflow chart by using the **Symbols** pane, the Stateflow Editor menu, or the Model Explorer.

Add Messages Through the Symbols Pane

1. In the **Modeling** tab, under **Design Data**, select **Symbols Pane**.
2. Click the **Create Message** icon .
3. In the row for the new message, under **Type**, click the icon and choose:
   - `Input Message`
   - `Local Message`
   - `Output Message`
4. Edit the name of the message.
5. For input and output messages, click the **Port** field and choose a port number.
6. To specify properties for the message, open the **Property Inspector**. In the **Symbols** pane, right-click the row for the message and select **Explore**. For more information, see “Set Properties for a Message” on page 13-5.

Add Messages by Using the Stateflow Editor Menu

1. In the Stateflow Editor, select the option corresponding to the scope of the message that you want to add.

Scope	Option
Input	In the **Modeling** tab, under **Design Data**, select **Message Input**.
Output	In the **Modeling** tab, under **Design Data**, select **Message Output**.
Local	In the **Modeling** tab, under **Design Data**, select **Message**.

2. In the Message dialog box, specify data properties. For more information, see "Set Properties for a Message" on page 13-5.

Add Messages Through the Model Explorer

1. In the **Modeling** tab, under **Design Data**, select **Model Explorer**.
2. In the **Model Hierarchy** pane, select the object in the Stateflow hierarchy where you want to make the new message visible. The object that you select becomes the parent of the new message.
3. In the Model Explorer menu, select **Add** > **Message**. The new message with a default definition appears in the **Contents** pane of the Model Explorer.
4. In the **Message** pane, specify the properties of the message. For more information, see "Set Properties for a Message" on page 13-5.

Lifetime of a Stateflow Message

A Stateflow message has a finite lifetime. The lifetime begins when you send a message to a receiving queue with the `send` operator. The message remains in the queue until a transition or state `on` action evaluates it or the chart receives it by using the `receive` operator.

A message becomes valid when a chart evaluates or receives it. The message remains valid until:

- The end of the current time step, when the chart destroys any remaining valid messages.
- The chart forwards the message to another queue by using the `forward` operator. The message continues its lifetime in the new queue.
- The chart discards the message by using the `discard`.

While a message is valid, other transitions and actions can evaluate the message and access its data. To check if a message is valid, use the `isvalid` operator.

To view the interchange of messages during simulation, add a Sequence Viewer block to your Simulink model. The Sequence Viewer block displays:

- Sent messages
- Received messages
- Forwarded messages
- Dropped messages
- Destroyed messages
- Discarded messages

For more information, see “Use the Sequence Viewer to Visualize Messages, Events, and Entities” on page 13-24.

Limitations for Messages

You cannot use messages in:

- Moore charts
- Atomic subcharts
- Breakpoint condition expressions

In charts that use C as the action language, messages do not support multiword fixed-point data.

See Also

`discard` | `forward` | `isvalid` | `receive` | `send` | Queue | Sequence Viewer

More About

- “Set Properties for a Message” on page 13-5
- “Control Message Activity in Stateflow Charts” on page 13-9
- “Send Messages With String Data” on page 21-8
- “Use the Sequence Viewer to Visualize Messages, Events, and Entities” on page 13-24

Set Properties for a Message

A message is a Stateflow object that communicates data locally or between charts in a Simulink model. For more information, see “Communicate with Stateflow Charts by Sending Messages” on page 13-2.

When you create Stateflow charts in Simulink models, you can modify message properties in the **Property Inspector** or the Model Explorer.

To use the **Property Inspector**:

1. In the **Modeling** tab, under **Design Data**, select **Symbols Pane** and **Property Inspector**.
2. In the **Symbols** pane, select the message.
3. In the **Property Inspector**, edit the message properties.

To use the Model Explorer:

1. In the **Modeling** tab, under **Design Data**, select **Model Explorer**.
2. In the **Model Hierarchy** pane, select the parent of the message.
3. In the **Contents** pane, select the message.
4. In the **Dialog** pane, edit the message properties.

You can also modify these properties programmatically by using `Stateflow.Message` objects. For more information about the Stateflow programmatic interface, see “Overview of the Stateflow API”.

Stateflow Message Properties

Name

Name of the message. For more information, see “Guidelines for Naming Stateflow Objects” on page 1-66.

Scope

Scope of the message. The scope specifies where the message occurs relative to the parent object.

Scope	Description
`Input`	Message that is received from another block in the Simulink model. Each input message can use an internal receiving queue that is maintained by the Stateflow chart or an external receiving queue that is managed by a Queue block.
`Output`	Message that is sent through an output port to another block in the Simulink model.
`Local`	Message that is local to the Stateflow chart. The Stateflow chart maintains an internal receiving queue for each local message. When you send a local message, it is visible by state and transition actions in the same chart. You cannot send a local message outside the chart.

Port

Index of the port associated with the message. This property applies only to input and output messages.

Size

Size of the message data field. For more information, see “Specify Size of Stateflow Data” on page 10-26.

Complexity

Specifies whether the message data field accepts complex values.

Complexity Setting	Description
`Off`	Data field does not accept complex values.
`On`	Data field accepts complex values.
`Inherited`	Data field inherits the complexity setting from a Simulink block.

The default value is `Off`. For more information, see “Complex Data in Stateflow Charts” on page 24-2.

Type

Type of the message data field. To specify the data type:

- From the **Type** drop-down list, select a built-in type.
- In the **Type** field, enter an expression that evaluates to a data type.

Additionally, in the Model Explorer, you can open the Data Type Assistant by clicking the **Show data type assistant** button ››. Specify a data **Mode**, and then specify the data type based on that mode. For more information, see “Specify Type of Stateflow Data” on page 10-20.

Note In charts that use C as the action language, messages do not support multiword fixed-point data.

Initial Value

Initial value of the message data. Enter an expression or parameter defined in the Stateflow hierarchy, MATLAB base workspace, or Simulink masked subsystem. This property applies only to local and output messages.

If you do not specify a value, the default value for numeric data is 0. For enumerated data, the default value typically is the first one listed in the `enumeration` section of the definition. You can specify a different default enumerated value in the `methods` section of the definition. For more information, see “Define Enumerated Data Types” on page 20-5.

Priority

Priority for the message. If two distinct messages occur at the same time, this property determines which message is processed first. A smaller numeric value indicates a higher priority. This property is

visible only for local and output messages in discrete-event charts. For more information, see "Create Custom Queuing Systems Using Discrete-Event Stateflow Charts" (SimEvents).

Add to Watch Window

Enables watching the message queue and data field in the Stateflow Breakpoints and Watch window. For more information, see "View Data in the Breakpoints and Watch Window" on page 30-10.

Message Queue Properties

These properties define the behavior of receiving queues and apply only to input and local messages.

Use Internal Queue

Specifies that the Stateflow chart maintains an internal receiving queue for the input message. By default, this property is enabled. When you disable this property, you can connect the message input port to:

- A Queue block that manages an external queue in your Simulink model
- A root-level Inport block that enables messages to cross the model boundary

For more information on external message queues, see "Messages" (Simulink).

Queue Capacity

Specifies the maximum number of messages held in an internal receiving queue. If a chart sends a message when the queue is full, a queue overflow occurs. To avoid dropped messages, set the queue capacity high enough that incoming messages do not cause the queue to overflow. The maximum queue capacity is 2^{16}–1.

Queue Overflow Diagnostic

Specifies the level of diagnostic action when the number of incoming messages exceeds the queue capacity. The default option is `Error`.

Diagnostic Setting	Description
`Error`	When the queue overflows, the simulation stops with an error.
`Warning`	When the queue overflows, the queue drops the last message and simulation continues with a warning.
`None`	When the queue overflows, the queue drops the last message and simulation continues without issuing a warning.

Queue Type

Specifies the order in which messages are removed from the receiving queue. The default option is `FIFO`.

Queue Type Setting	Description
`FIFO`	First in, first out
`LIFO`	Last in, first out

Queue Type Setting	Description
`Priority`	Remove messages according to the value in the data field. Choosing this setting exposes the **Priority order** field, which has these options: • `Ascending` — The messages are removed in ascending order of the message data value. • `Descending` — The messages are removed in descending order of the message data value.

Additional Properties

You can set additional message properties in:

- The **Info** tab of the **Property Inspector**.
- The **Description** tab of the Model Explorer.

Description

Description of the message.

Document Link

Link to online documentation for the message. You can enter a web URL address or a MATLAB command that displays documentation as an HTML file or as text in the MATLAB Command Window. When you click the **Document link** hyperlink, Stateflow displays the documentation.

See Also

Blocks
Queue

Objects
`Stateflow.Message`

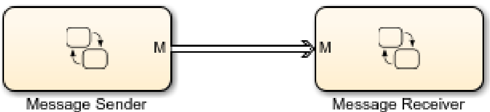
Tools
Model Explorer

More About

- “Communicate with Stateflow Charts by Sending Messages” on page 13-2
- “Guidelines for Naming Stateflow Objects” on page 1-66
- “Specify Size of Stateflow Data” on page 10-26
- “Specify Type of Stateflow Data” on page 10-20
- “Complex Data in Stateflow Charts” on page 24-2

Control Message Activity in Stateflow Charts

A message is a Stateflow object that communicates data locally or between charts in a Simulink model. From a sender chart, you can send or forward a message. In the receiving chart, a queue receives the message and holds it until the chart can evaluate it.



Using Stateflow operators, you can access message data, and send, receive, discard, or forward a message. You can also determine whether a message is valid and find the number of messages in a queue. For more information, see “Communicate with Stateflow Charts by Sending Messages” on page 13-2.

Access Message Data

Stateflow messages have a data field. To read or write to the message data field of a valid message, use dot notation syntax:

```
message_name.data
```

If you send a message without first assigning a value to the message data, the default value for numeric data is 0. For enumerated data, the default is the first value listed in the `enumeration` section of the definition, unless you specify otherwise in the `methods` section of the definition.

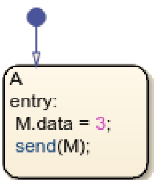
You cannot access message data for messages that are still in the queue or that have already been discarded.

Send a Message

To send an output or local message, use the `send` operator:

```
send(message_name)
```

For example, in this chart, the entry action in state `A` sends a message `M` with a data value of `3`. If the message scope is `Local`, then the message goes in the local receiving queue. If the message scope is `Output`, then the chart sends the message through the output port to another block in the Simulink model.



In a single time step, you can send multiple messages through an output port or to a local receiving queue.

If a chart sends a message that exceeds the capacity of the receiving queue, a queue overflow occurs. The result of the queue overflow depends on the type of receiving queue.

- When an overflow occurs in an internal queue, the Stateflow chart drops the new message. You can control the level of diagnostic action by setting the **Queue Overflow Diagnostic** property for the message. See “Queue Overflow Diagnostic” on page 13-7.
- When an overflow occurs in an external queue, the Queue block either drops the new message or overwrites the oldest message in the queue, depending on the configuration of the block. See “Overwrite the oldest element if queue is full” (Simulink). An overflow in an external queue always results in a warning.

Guard Transitions and Actions

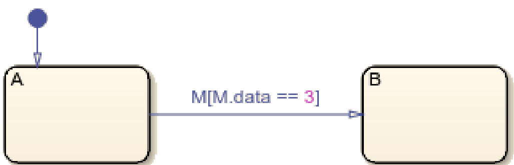
Messages can guard transitions or state actions of type `on`. During a time step, when the guarding message is evaluated for the first time, the chart removes the message from the queue and makes the message valid. While the message is valid, other transitions or actions can access the message data but they do not remove another message from the queue.

Guard a Transition with a Message

In this chart, a message `M` guards the transition from state `A` to state `B`. The transition occurs when both of these conditions are true:

- A message is present in the queue.
- The data value of the message is equal to 3.

If a message is not present or if the data value is not equal to 3, then the transition does not occur. If a message is present, it is removed from the queue regardless of whether the transition occurs.

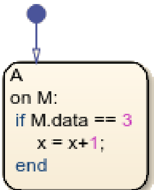


Guard a State on Action with a Message

In this chart, a message `M` guards the `on` action in state `A`. When state `A` becomes active, it increments the value of `x` if both of these conditions are true:

- A message is present in the queue.
- The data value of the message is equal to 3.

If a message is not present or if the data value is not equal to 3, then the value of `x` does not change. If a message is present, it is removed from the queue regardless of whether `x` is modified.



Receive a Message

To extract an input or local message from its receiving queue, use the `receive` operator:

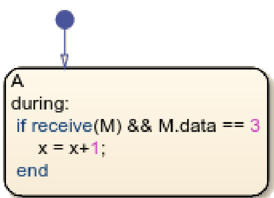
```
receive(message_name)
```

If a valid message `M` exists, `receive(M)` returns `true`. If a valid message does not exist but there is a message in the queue, then the chart removes the message from the queue and `receive(M)` returns `true`. If a valid message does not exist and there are no messages in the queue, `receive(M)` returns `false`.

For example, in this chart, the `during` action in state `A` checks the queue for message `M` and increments the value of `x` if both of these conditions are true:

- A message is present in the queue.
- The data value of the message is equal to 3.

If a message is not present or if the data value is not equal to 3, then the value of `x` does not change. If a message is present, the chart removes it from the queue regardless of the data value.



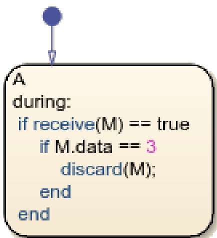
Discard a Message

To discard a valid input or local message, use the `discard` operator:

```
discard(message_name)
```

After a chart discards a message, it can remove another message from the queue in the same time step. A chart cannot access the data of a discarded message.

For example, in this chart, the `during` action in state `A` checks the queue for message `M`. If a message is present, the chart removes it from the queue. If the message has a data value equal to 3, the chart discards the message.



Forward a Message

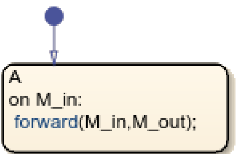
To forward a valid input or local message to a local queue or an output port, use the `forward` operator:

`forward(`*`message_in_name`*`,`*`message_out_name`*`)`

After a chart forwards a message, it can remove another message from the queue in the same time step.

Forward an Input Message

In this chart, state `A` checks the input queue for message `M_in`. If a message is present, the chart removes the message from the queue and forwards it to the output port `M_out`. After the chart forwards the message, the message is no longer valid in state `A`.



Forward a Local Message

In this chart, the transition between state `A` and state `B` checks the local queue for message `M_local`. If a message is present, the transition removes the message from the `M_local` message queue and forwards it to the output port `M_out`.

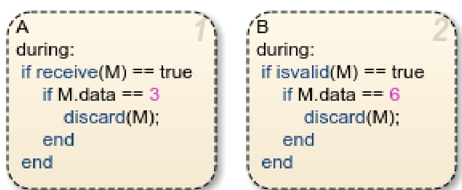
Determine if a Message Is Valid

To check if an input or local message is valid, use the `isvalid` operator:

`isvalid(`*`message_name`*`)`

A message is valid if the chart has removed it from the receiving queue and has not forwarded or discarded it.

For example, this chart first executes state `A`, as described in “Discard a Message” on page 13-11. When the chart executes state `B`, the `during` action checks that the message `M` is valid. If the message is valid and has a data value equal to 6, the chart discards the message.

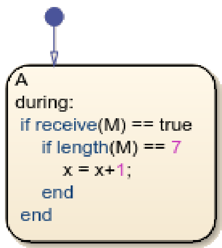


Determine the Length of the Queue

To check the number of messages in an internal receiving queue of an input or local message, use the `length` operator:

```
length(message_name)
```

For example, in this chart, the `during` action in state `A` checks the queue for message `M`. If a message is present, the chart removes it from the queue. If exactly seven messages remain in the queue, the chart increments the value of `x`.



The `length` operator is not supported for input messages that use external receiving queues managed by a Queue block.

See Also

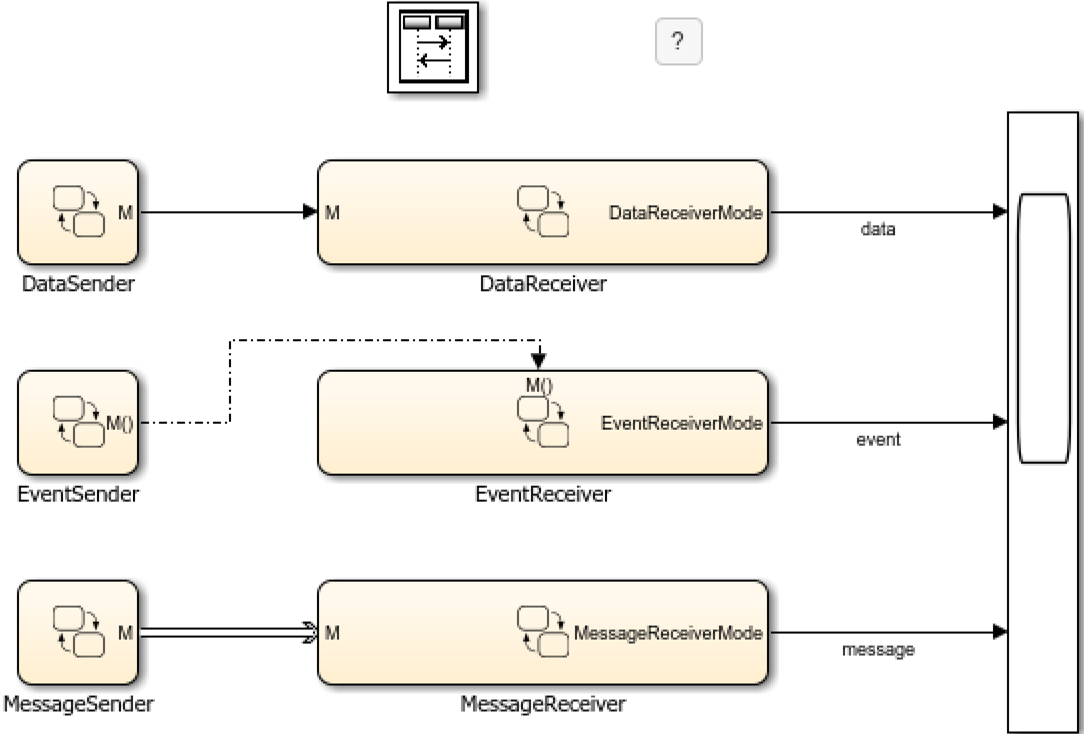
`discard` | `forward` | `isvalid` | `length` | `receive` | `send` | Queue

More About

- “Communicate with Stateflow Charts by Sending Messages” on page 13-2
- “Set Properties for a Message” on page 13-5
- “Use the Sequence Viewer to Visualize Messages, Events, and Entities” on page 13-24
- “View Differences Between Stateflow Messages, Events, and Data” on page 13-14

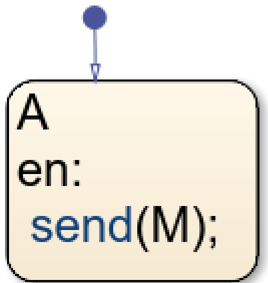
View Differences Between Stateflow Messages, Events, and Data

This example compares the behavior of messages, events, and data in Stateflow®.



Sender Charts

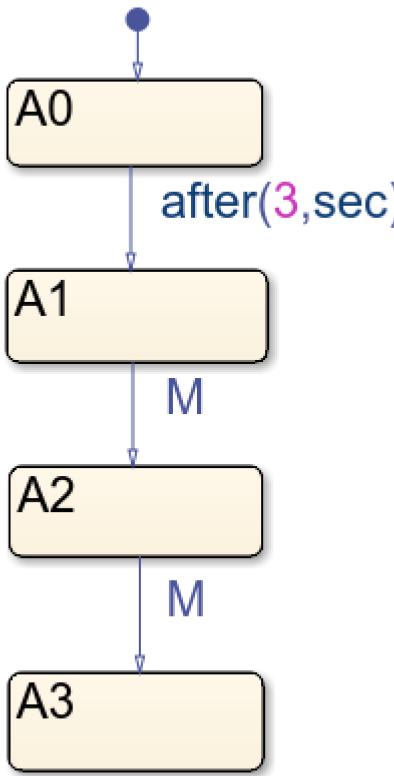
This model has three sender charts: `DataSender`, `EventSender`, and `MessageSender`. Each sender chart has one state. In the entry action of the state, the charts assign a value to data, send a function-call event, or send a message.



Receiver Charts

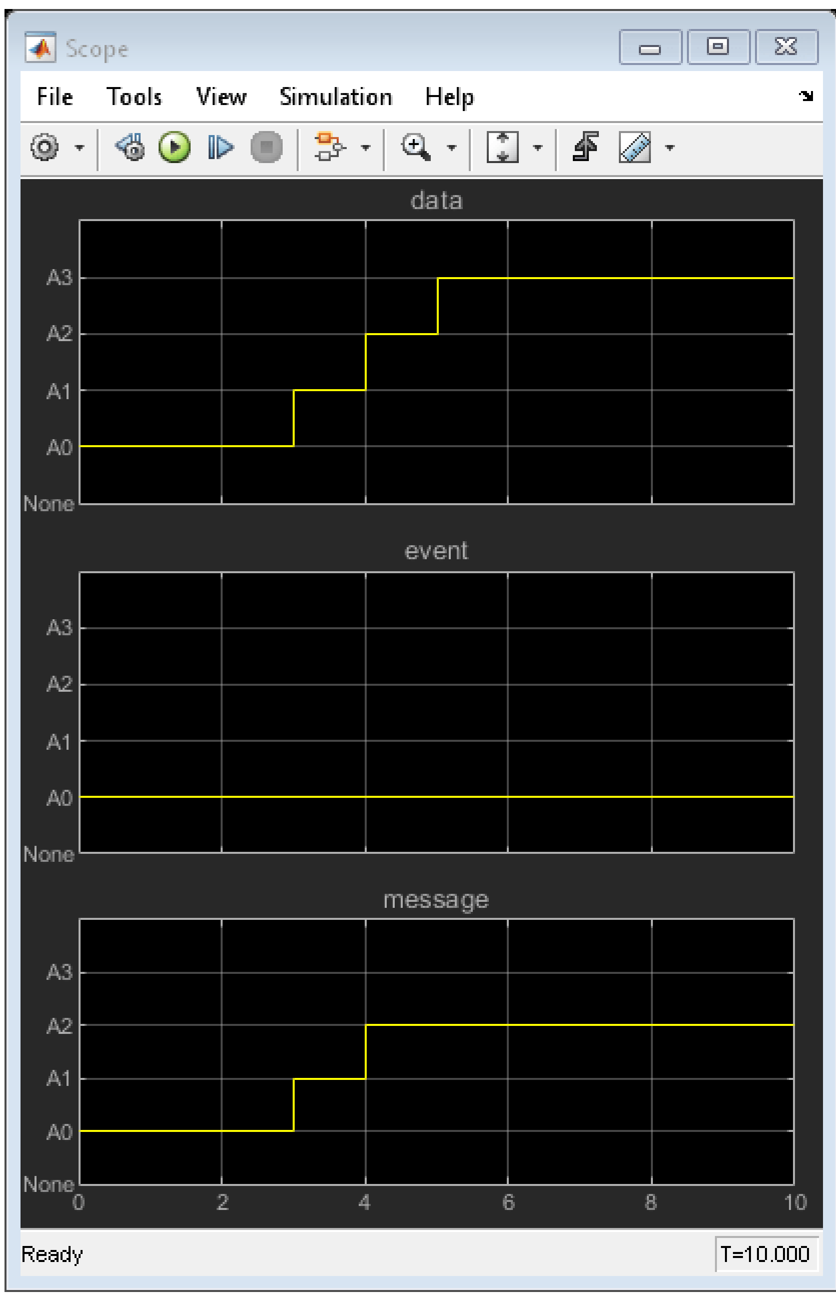
For each of the sender charts, there is a corresponding receiver chart. Each receiver chart has a state diagram with states `A0`, `A1`, `A2`, and `A3`. The implicit event `after(3,sec)` triggers the transition

from `A0` to `A1`. The data, event, or message from the corresponding sender chart guards the transitions between `A1`, `A2`, and `A3`.



Scope Output

Each receiver chart has active state output enabled and connected to a scope. The scope shows which states are active in each time step. This output highlights the difference in behavior between output data, events, and messages.

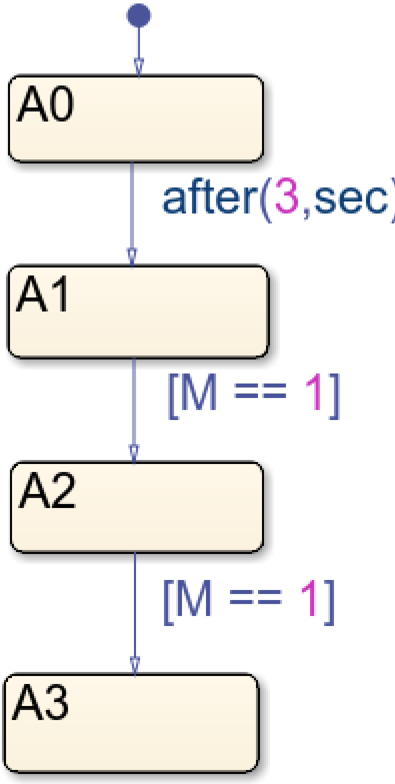


Behavior of Data

The `DataSender` chart assigns a value of 1 to the output data `M`, which connects as an input to the `DataReceiver` chart.

The `DataReceiver` chart executes once at every time step. At the start of simulation, state `A0` is active. At time `t=3`, the transition from `A0` to `A1` occurs. At time `t=4`, the chart tests whether `M` equals 1. This condition is true, so the chart transitions from `A1` to `A2`. At time `t=5`, `M` still equals `1`, so the chart transitions from `A2` to `A3`. On the scope, you see that `DataReceiver` changes states three times.

After data is assigned a value, it holds its value throughout the simulation. Therefore, each time that the DataReceiver evaluates the condition [M == 1], it transitions to a new state.

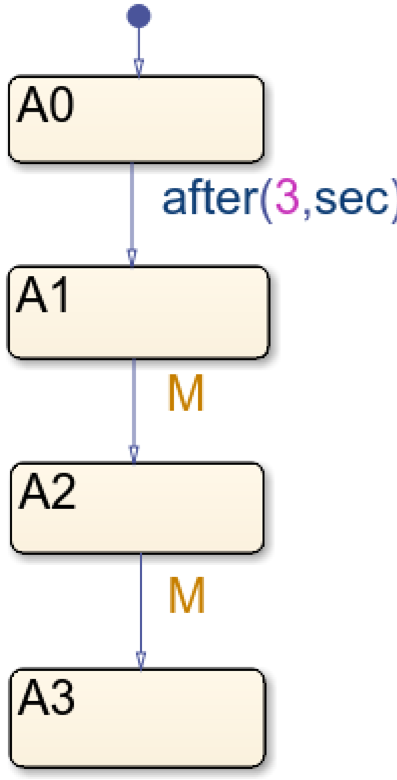


Behavior of Event

The EventSender chart uses the command send(M) to send a function-call output event to wake up the EventReceiver chart.

The EventReceiver chart executes only when the input event M wakes up the chart. At the start of simulation, state A0 is active. The transition from A0 to A1 is based on absolute-time temporal logic and is not valid at time t=0. A0 remains active and the chart goes back to sleep. Because EventSender sends the event M only once, EventReceiver does not wake up again. On the scope, you see that EventReceiver never transitions out of A0.

Events do not remain valid across time steps, so the receiving chart has only one chance to respond to the event. When EventSender sends the event, EventReceiver is not ready to respond to it. The opportunity for EventReceiver to transition in response to the event is lost.

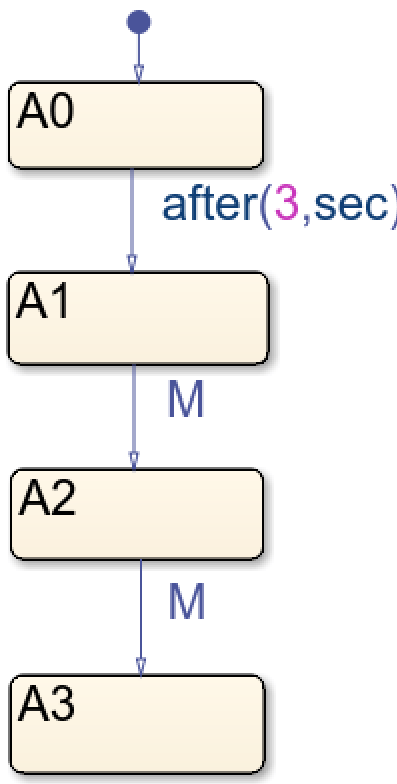


Behavior of Message

The MessageSender chart uses the syntax send(M) to send a message through the output message port. The message goes into the input message queue of the MessageReceiver chart. The message waits in the queue until MessageReceiver evaluates it.

The MessageReceiver chart executes once at every time step. At the start of simulation, state A0 is active. At time t=3, the transition from A0 to A1 occurs. At time t=4, the chart determines that M is present in the queue, so it takes the transition to A2. At the end of the time step, the chart removes M from the queue. At time t=5, there is no message present in the queue, so the chart does not transition to A3. A2 remains the active state. On the scope, you see that MessageReceiver changes state only two times.

Unlike events, messages are queued. The receiving chart can choose to respond to a message anytime after it was sent. Unlike data, the message does not remain valid indefinitely. The message is destroyed at the end of the time step.



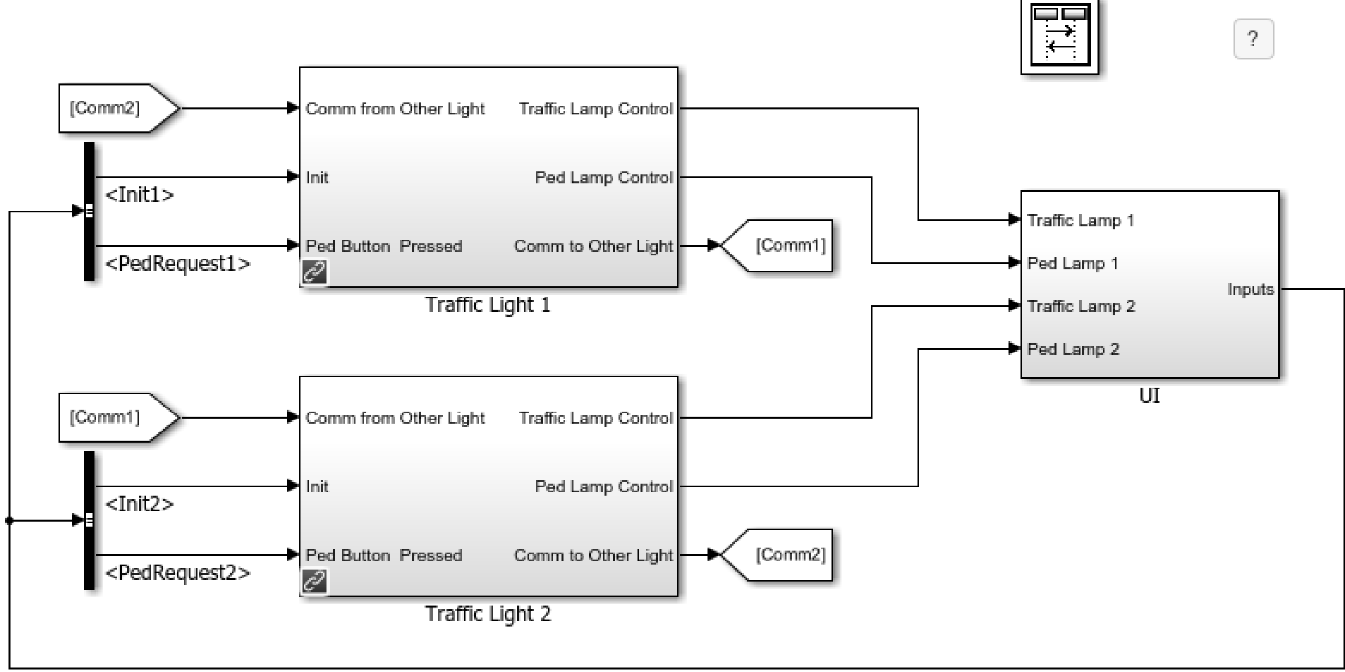
See Also

More About

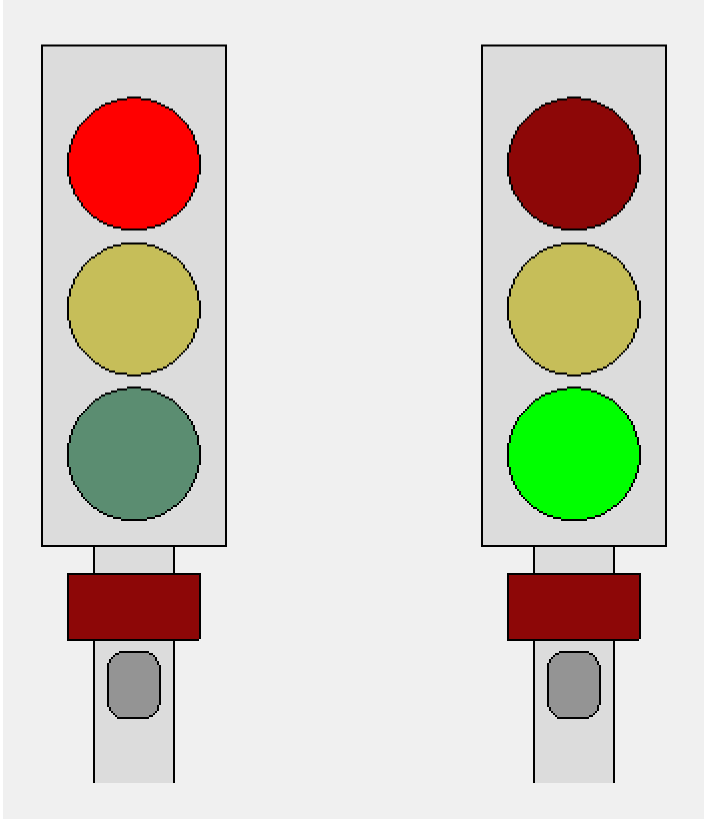
- “Share Data with Simulink and the MATLAB Workspace” on page 10-30
- “Synchronize Model Components by Broadcasting Events” on page 12-2
- “Communicate with Stateflow Charts by Sending Messages” on page 13-2

Model Distributed Traffic Control System by Using Messages

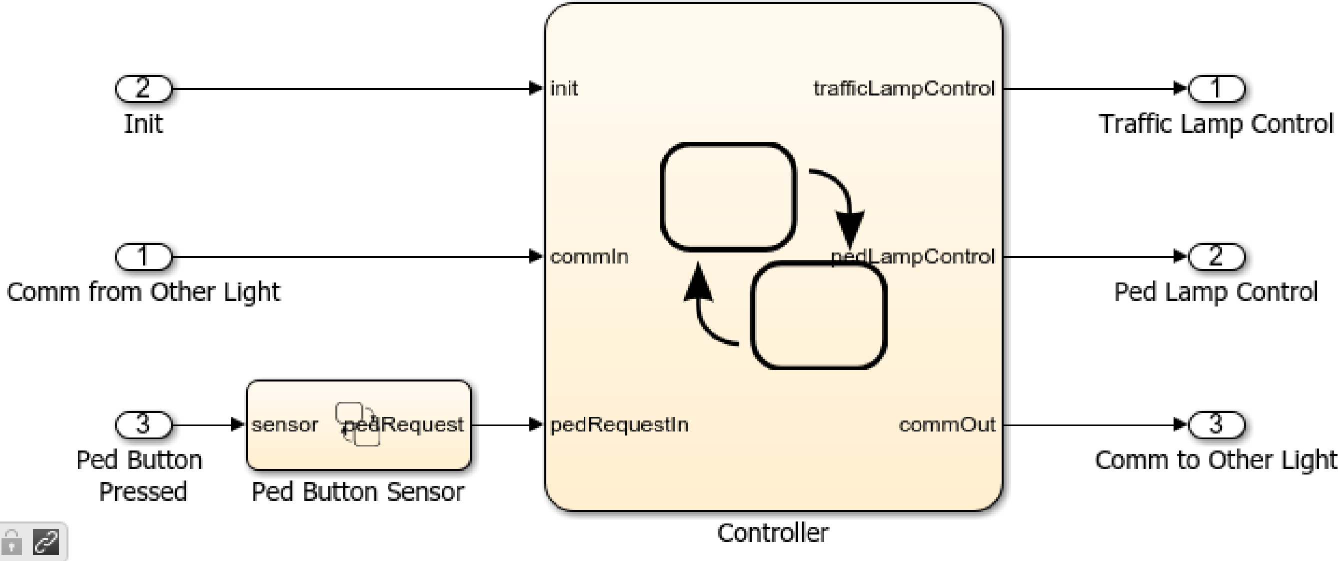
This example shows how to model a distributed control system for an intersection of one-way roads. To coordinate the state of the traffic lights, the two charts communicate with each other by using messages. The design of the two charts is identical.



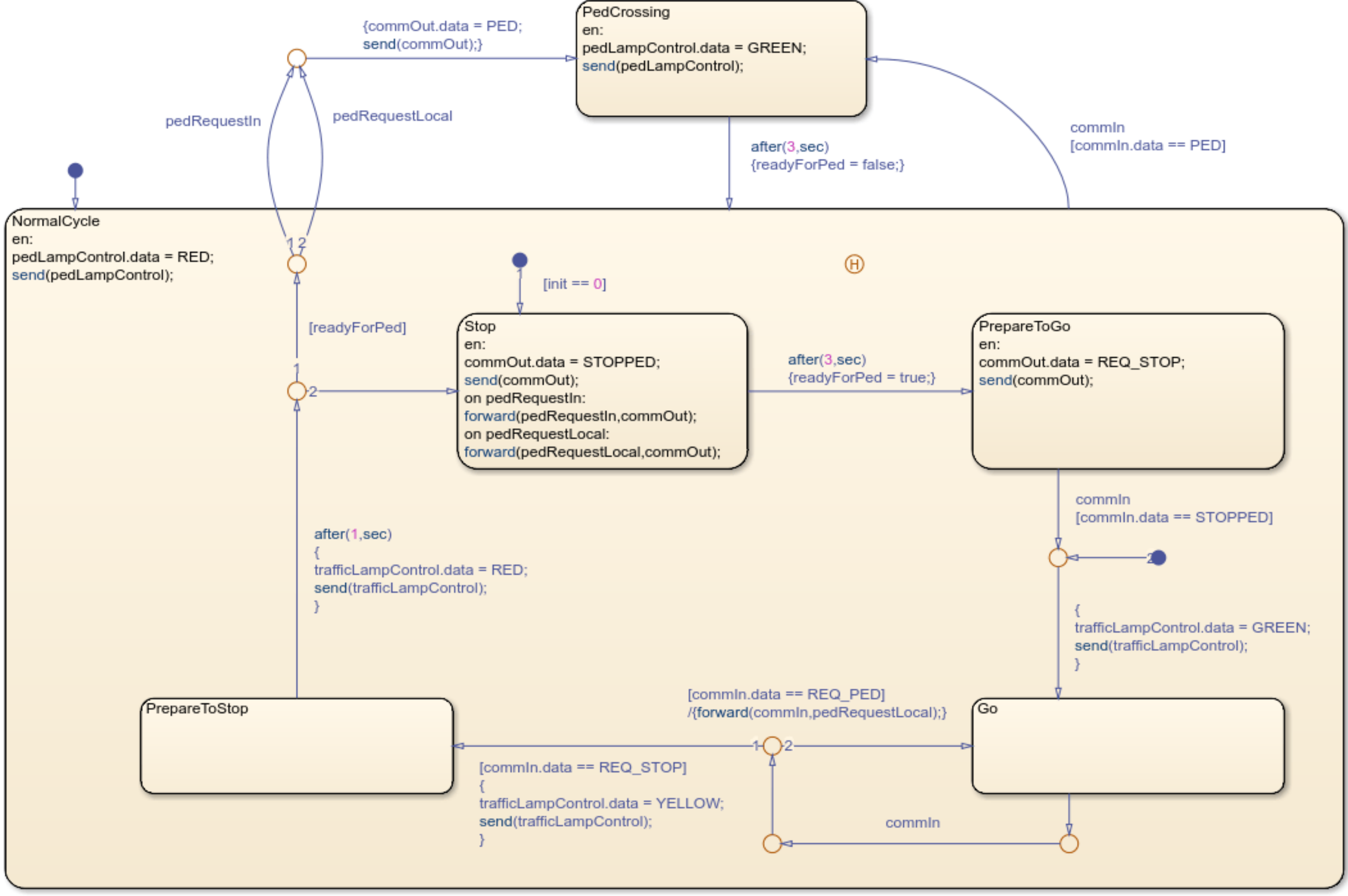
You can interact with the traffic signals through a MATLAB® UI. To request a pedestrian crossing, click one of the dark gray request buttons at the bottom of the traffic signal.



The controller for each road is implemented by the traffic light controller subsystems Traffic Light 1 and Traffic Light 2.



In each subsystem, the Controller chart describes the main logic of the traffic controller through the various states of the traffic signal.



This model takes advantage of these semantic features of messages:

- Messages are not discarded if they are not acted upon immediately. For example, in this model, pedestrian requests are queued up until the controller can react to a request when the traffic light turns red.
- You can set up message loops between different components. These loops do not result in algebraic loops in your model.
- Normally, input messages are destroyed at the end of the time step in which they are evaluated. However, you can preserve these input messages for use at a later time by temporarily forwarding them to a local "holding" queue. For example, when the Controller chart exits the `Go` state, it uses the local queue `pedRequestLocal` to store pedestrian requests made on the other road. The chart checks for those requests later, when it exits the `PrepareToStop` state.

To change the speed of the simulation, in the **Simulation** tab, select **Run** > **Simulation Pacing**. In the Simulation Pacing Options dialog box, adjust the slider setting. For more information, see "Simulation Pacing" (Simulink).

See Also

More About

- "Communicate with Stateflow Charts by Sending Messages" on page 13-2
- "Use the Sequence Viewer to Visualize Messages, Events, and Entities" on page 13-24
- "View Differences Between Stateflow Messages, Events, and Data" on page 13-14
- "Simulation Pacing" (Simulink)

Use the Sequence Viewer to Visualize Messages, Events, and Entities

To see the interchange of messages and events between the blocks from the Simulink Messages & Events library, Stateflow charts in Simulink models, and SimEvents blocks, you can:

- Use the **Sequence Viewer** tool from the Simulink toolstrip.
- Add a Sequence Viewer block to your Simulink model.

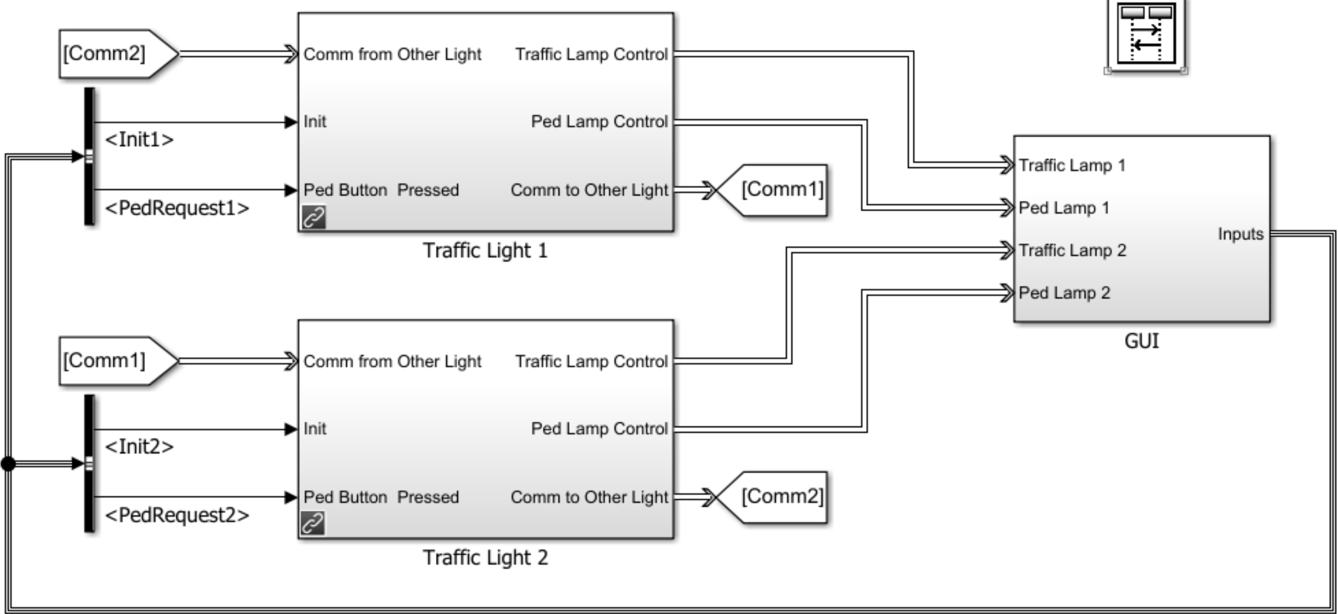
The Sequence Viewer allows you to visualize message transition events and the data that the messages carry. In the Sequence Viewer, you can view event data related to Stateflow chart execution and the exchange of messages between Stateflow charts. The Sequence Viewer window shows messages as they are created, sent, forwarded, received, and destroyed at different times during model execution. The Sequence Viewer window also displays state activity, transitions, and function calls to Stateflow graphical functions, Simulink functions, and MATLAB functions.

With the Sequence Viewer, you can also visualize the movement of entities between blocks when simulating SimEvents models. All SimEvents blocks that can store entities appear as lifelines in the Sequence Viewer window. Entities moving between these blocks appear as lines with arrows. You can view calls to Simulink Function blocks and to MATLAB Function blocks.

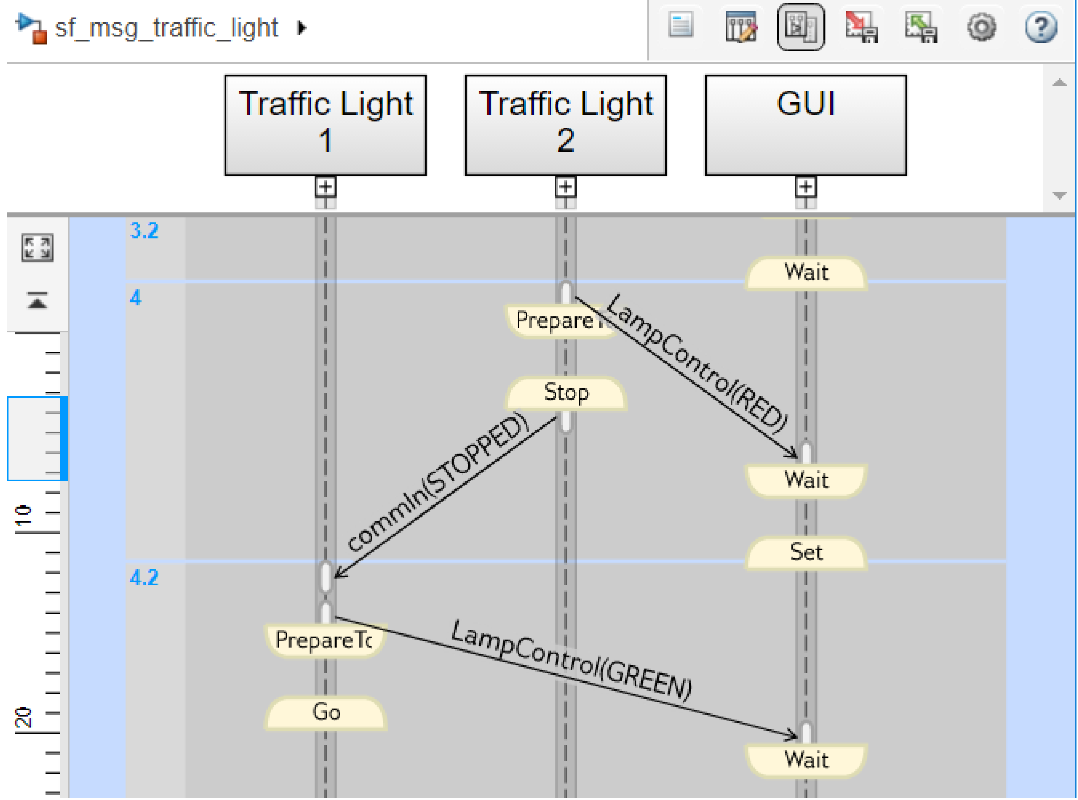
You can add a Sequence Viewer block to the top level of a model or any subsystem. If you place a Sequence Viewer block in a subsystem that does not have messages, events, or state activity, the Sequence Viewer window informs you that there is nothing to display.

For instance, open the Stateflow example `sf_msg_traffic_light`.

```
openExample("stateflow/ModelingADistributedTrafficControlSystemUsingMessageExample")
```



This model has three Simulink subsystems: Traffic Light 1, Traffic Light 2, and GUI. During simulation, the Stateflow charts in these subsystems exchange data by sending messages. As messages pass through the system, you can view them in the Sequence Viewer window. The Sequence Viewer window represents each block in the model as a vertical lifeline with simulation time progressing downward.



Components of the Sequence Viewer Window

Navigation Toolbar

At the top of the Sequence Viewer window, a navigation toolbar displays the model hierarchy path. Using the toolbar buttons, you can:

- Show or hide the **Property Inspector**.
- Select an automatic or manual layout.
- Show or hide inactive lifelines.
- Save Sequence Viewer settings.
- Restore Sequence Viewer settings.
- Configure Sequence Viewer parameters.
- Access the Sequence Viewer documentation.

Property Inspector

In the **Property Inspector**, you can choose filters to show or hide:

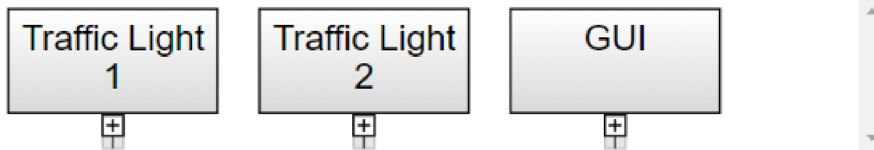
- Events
- Messages
- Function Calls
- State Changes and Transitions

Header Pane

The header pane below the Sequence Viewer toolbar shows lifeline headers containing the names of the corresponding blocks in a model.

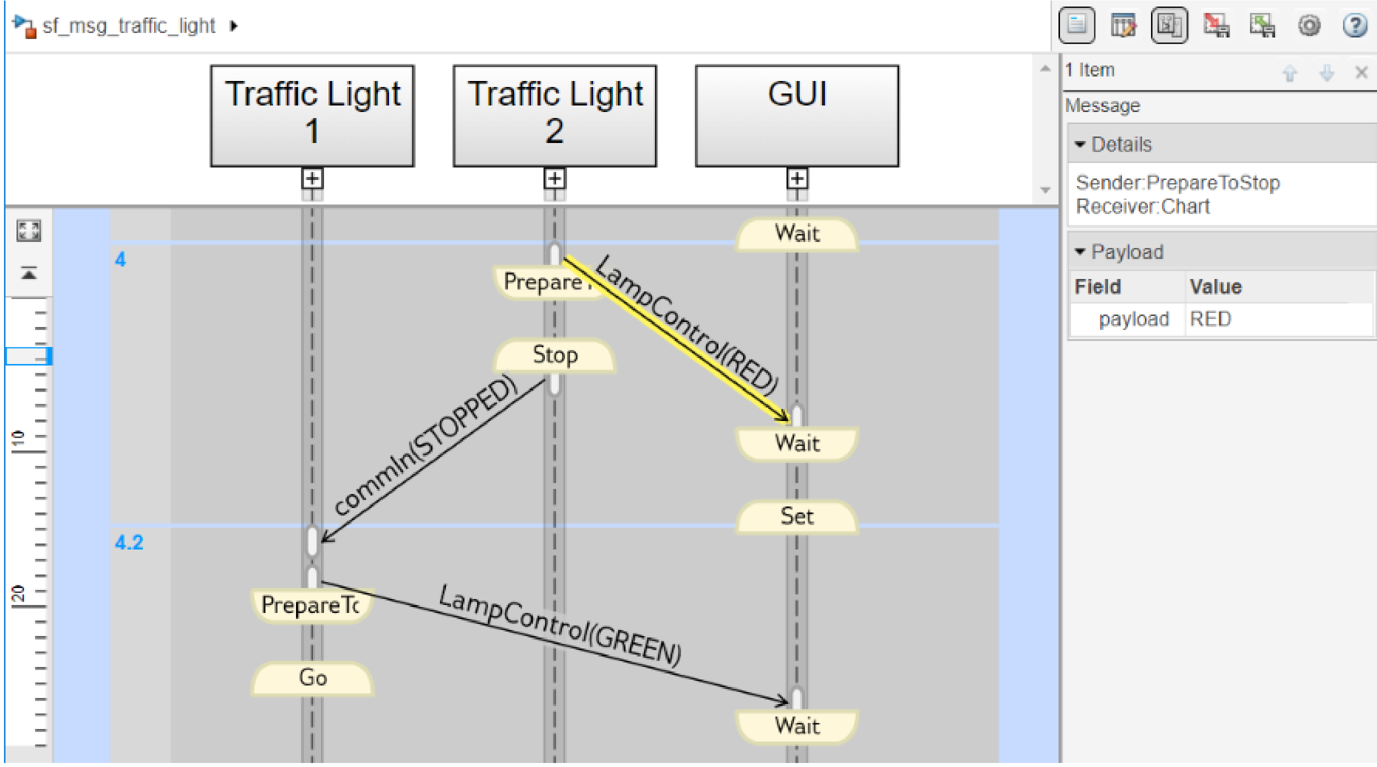
- Gray rectangular headers correspond to subsystems.
- White rectangular headers correspond to masked subsystems.
- Yellow headers with rounded corners correspond to Stateflow charts.

To open a block in the model, click the name in the corresponding lifeline header. To show or hide a lifeline, double-click the corresponding header. To resize a lifeline header, click and drag its right-hand side. To fit all lifeline headers in the Sequence Viewer window, press the space bar.



Message Pane

Below the header pane is the message pane. The message pane displays messages, events, and function calls between lifelines as arrows from the sender to the receiver. To display sender, receiver, and payload information in the **Property Inspector**, click the arrow corresponding to the message, event, or function call.



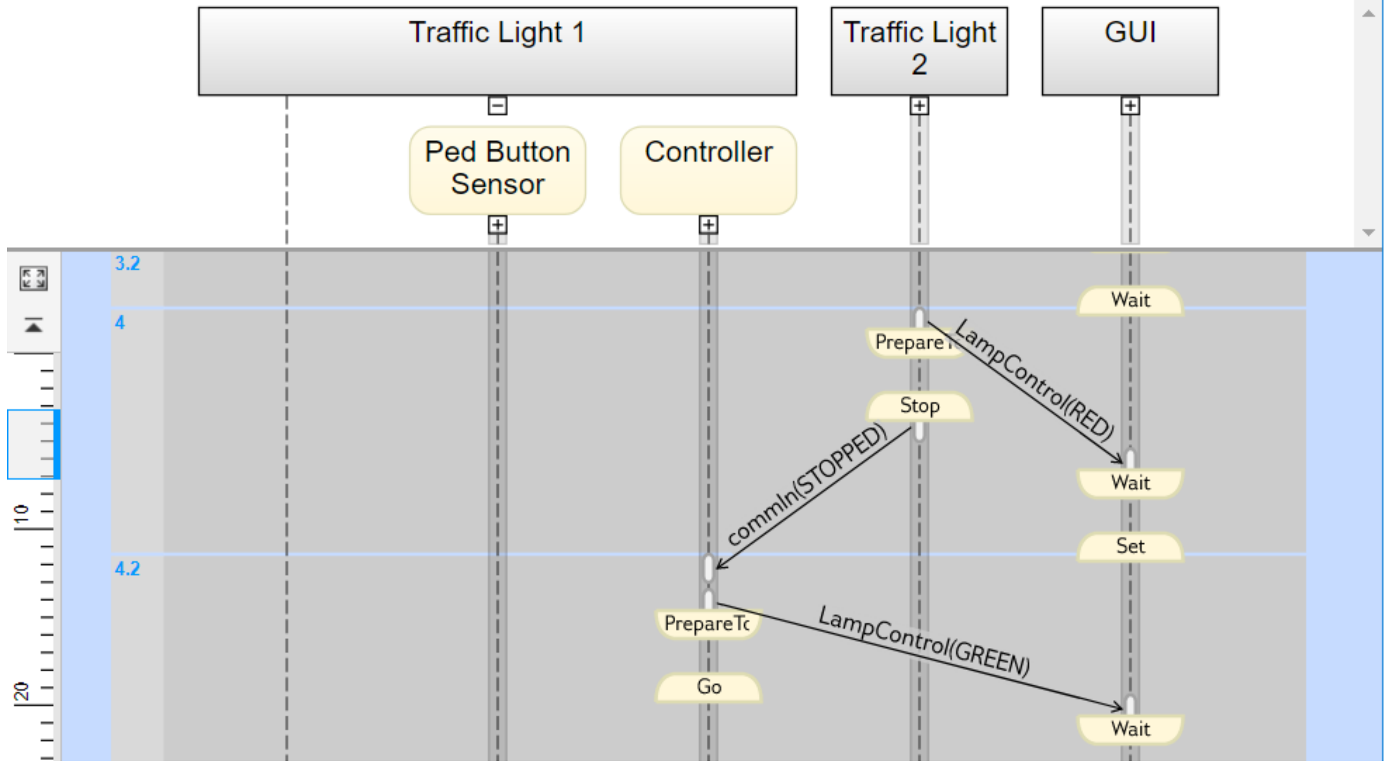
Navigate the Lifeline Hierarchy

In the Sequence Viewer window, the hierarchy of lifelines corresponds to the model hierarchy. When you pause or stop the model, you can expand or contract lifelines and change the root of focus for the viewer.

Expand a Parent Lifeline

In the message pane, a thick, gray lifeline indicates that you can expand the lifeline to see its children. To show the children of a lifeline, click the expander icon ⊞ below the header or double-click the parent lifeline.

For example, expanding the lifeline for the Traffic Light 1 block reveals two new lifelines corresponding to the Stateflow charts Ped Button Sensor and Controller.

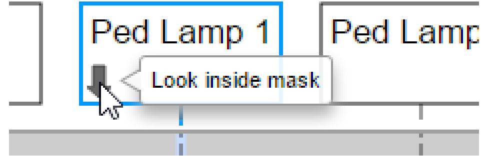


Expand a Masked Subsystem Lifeline

The Sequence Viewer window displays masked subsystems as white blocks. To show the children of a masked subsystem, point over the bottom left corner of the lifeline header and click the arrow.

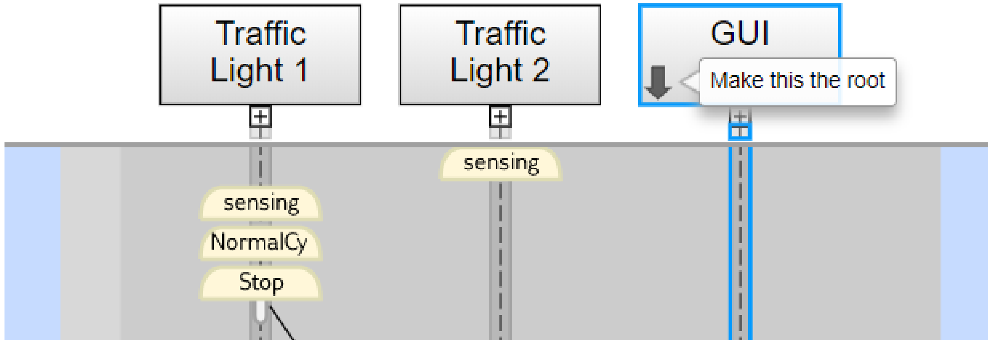
For example, the GUI subsystem contains four masked subsystems: Traffic Lamp 1,Traffic Lamp 2, Ped Lamp 1, and Ped Lamp 2.

You can display the child lifelines in these masked subsystems by clicking the arrow in the parent lifeline header.

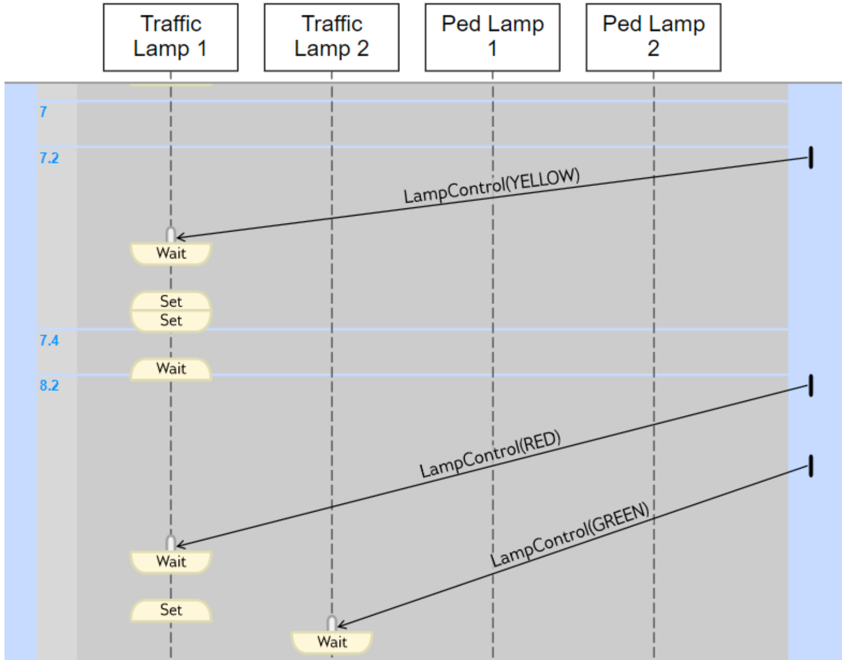


Change Root of Focus

To make a lifeline the root of focus for the viewer, point over the bottom left corner of the lifeline header and click the arrow. Alternatively, you can use the navigation toolbar at the top of the Sequence Viewer window to move the current root up and down the lifeline hierarchy. To move the current root up one level, press the **Esc** key.



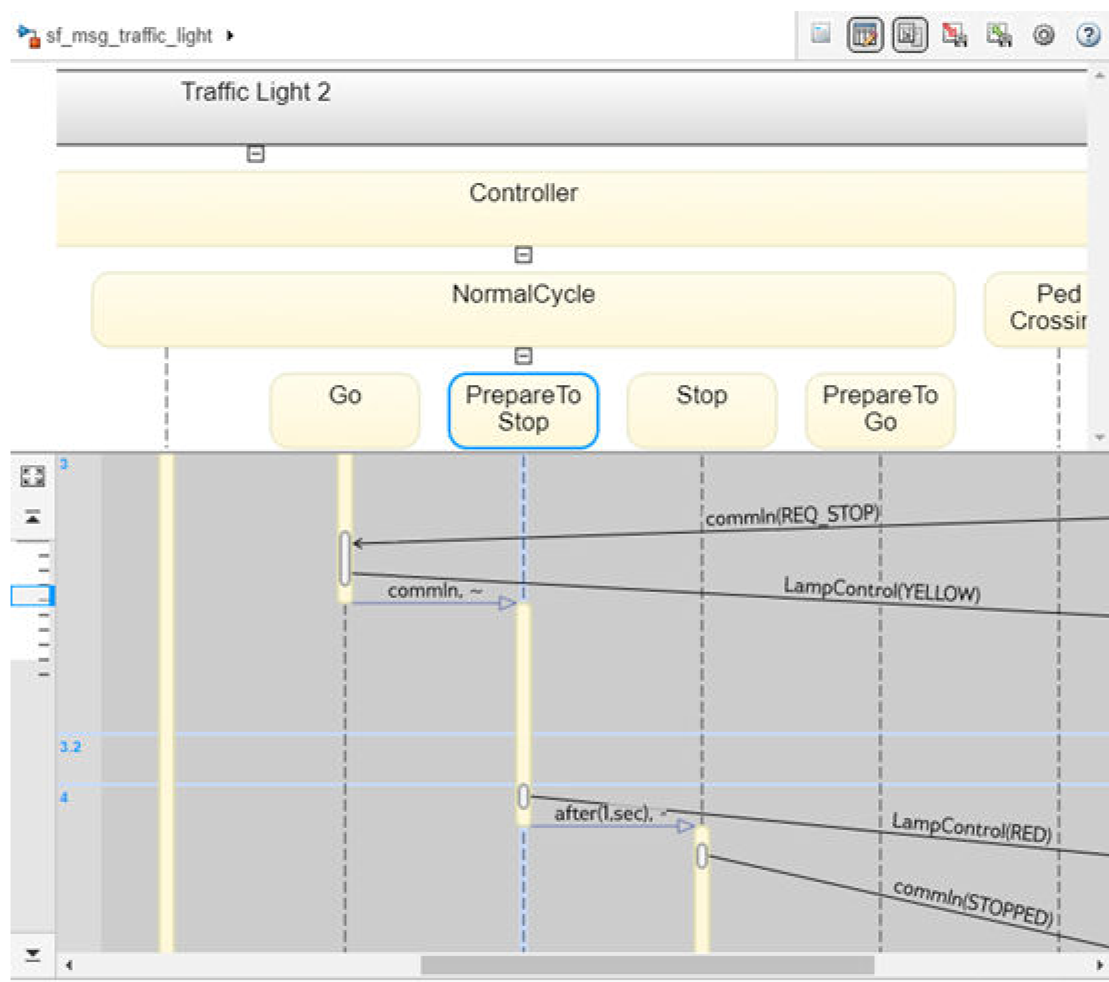
The Sequence Viewer window displays the current root lifeline path and shows its child lifelines. Any external events and messages are displayed as entering or exiting through vertical slots in the diagram gutter. When you point to a slot in the diagram gutter, a tooltip displays the name of the sending or receiving block.



View State Activity and Transitions

To see state activity and transitions in the Sequence Viewer window, expand the state hierarchy until you have reached the lowest child state. Vertical yellow bars show which state is active. Blue horizontal arrows denote the transitions between states.

In this example, you can see a transition from `Go` to `PrepareToStop` followed, after 1 second, by a transition to `Stop`.



To display the start state, end state, and full transition label in the **Property Inspector**, click the arrow corresponding to the transition.

To display information about the interactions that occur while a state is active, click the yellow bar corresponding to the state. In the **Property Inspector**, use the **Search Up** and **Search Down** buttons to move through the transitions, messages, events, and function calls that take place while the state is active.

View Function Calls

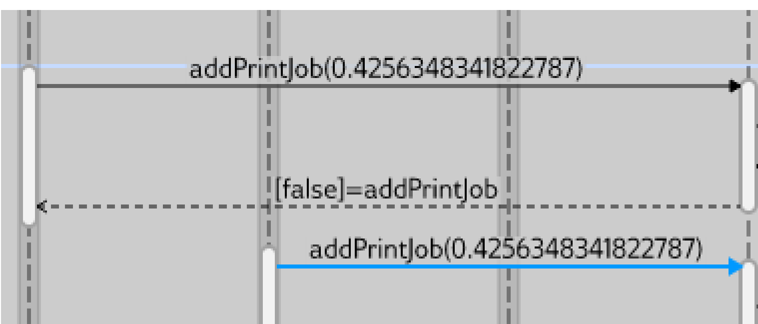
The Sequence Viewer displays function calls and replies. This table lists the type of support for each type of function call.

Function Call Type	Support
Calls to Simulink Function blocks	Fully supported.

Function Call Type	Support
Calls to Stateflow graphical or Stateflow MATLAB functions	• Scoped — Select the **Export chart level functions** chart option. Use the `chartName.functionName` dot notation. • Global — Select the **Treat exported functions as globally visible** chart option. You do not need the dot notation.
Calls to function-call subsystems	Fully supported and displayed.
Calls from MATLAB Function block	Supports displaying function call events with the limitation of calls crossing model reference boundaries.

The Sequence Viewer window displays function calls as solid arrows labeled with the format `function_name(argument_list)`. Replies to function calls are displayed as dashed arrows labeled with the format `[argument_list]=function_name`.

For example, in the model `slexPrinterExample`, a subsystem calls the Simulink Function block `addPrinterJob`. The function block replies with an output value of `false`.

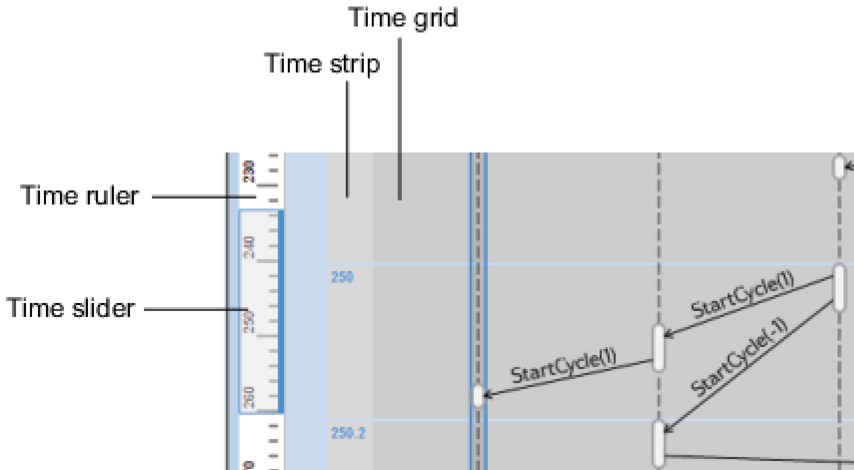


To open this example, enter:

```
openExample("stateflow/ShareFcnsAcrossSLandSFExample")
```

Simulation Time in the Sequence Viewer Window

The Sequence Viewer window shows events vertically, ordered in time. Multiple events in Simulink can happen at the same time. Conversely, there can be long periods of time during simulation with no events. As a consequence, the Sequence Viewer window shows time by using a combination of linear and nonlinear displays. The time ruler shows linear simulation time. The time grid shows time in a nonlinear fashion. Each time grid row, bordered by two blue lines, contains events that occur at the same simulation time. The time strip provides the times of the events in that grid row.



To show events in a specific simulation time range, use the scroll wheel or drag the time slider up and down the time ruler. To navigate to the beginning or end of the simulation, click the **Go to first event** or **Go to last event** buttons. To see the entire simulation duration on the time ruler, click the **Fit to view** button.

When using a variable step solver, you can adjust the precision of the time ruler. In the Model Explorer, on the **Main** tab of the Sequence Viewer Block Parameters pane, adjust the value of the **Time Precision for Variable Step** field.

Redisplay of Information in the Sequence Viewer Window

The Sequence Viewer saves the order and states of lifelines between simulation runs. When you close and reopen the Sequence Viewer window, it preserves the last open lifeline state. To save a particular viewer state, click the **Save Settings** button in the toolbar. Saving the model then saves that state information across sessions. To load the saved settings, click the **Restore Settings** button.

You can modify the **Time Precision for Variable Step** and **History** parameters only between simulations. You can access the buttons in the toolbar before simulation or when the simulation is paused. During a simulation, the buttons in the toolbar are disabled.

See Also

Blocks
Sequence Viewer

Tools
Sequence Viewer

More About

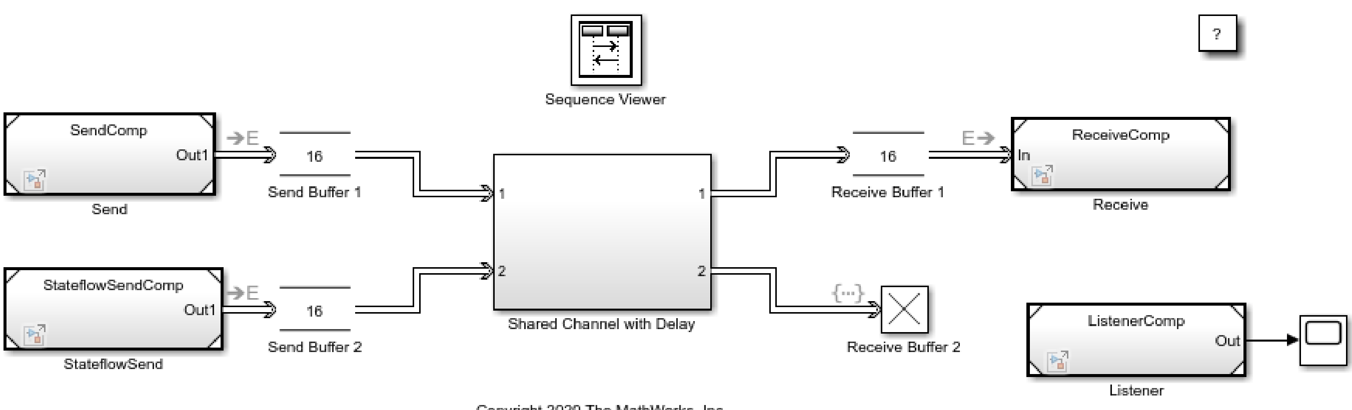
- “Communicate with Stateflow Charts by Sending Messages” on page 13-2
- “Model Distributed Traffic Control System by Using Messages” on page 13-20

Build a Shared Communication Channel with Multiple Senders and Receivers

This example shows how to model communication through a shared channel with multiple senders and receivers by using Simulink® messages, SimEvents®, and Stateflow®.

For an overview about messages, see “Simulink Messages Overview” (Simulink).

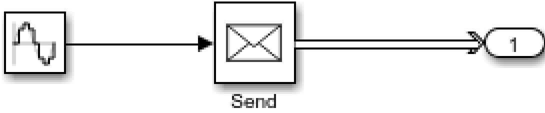
In this model, there are two software components that send messages and two components that receive messages. The shared channel transmits messages with an added delay. SimEvents® blocks are used to create custom communication behavior by merging the message lines, and copying and delaying messages. A Stateflow chart is used in a send component to send messages based on a decision logic.



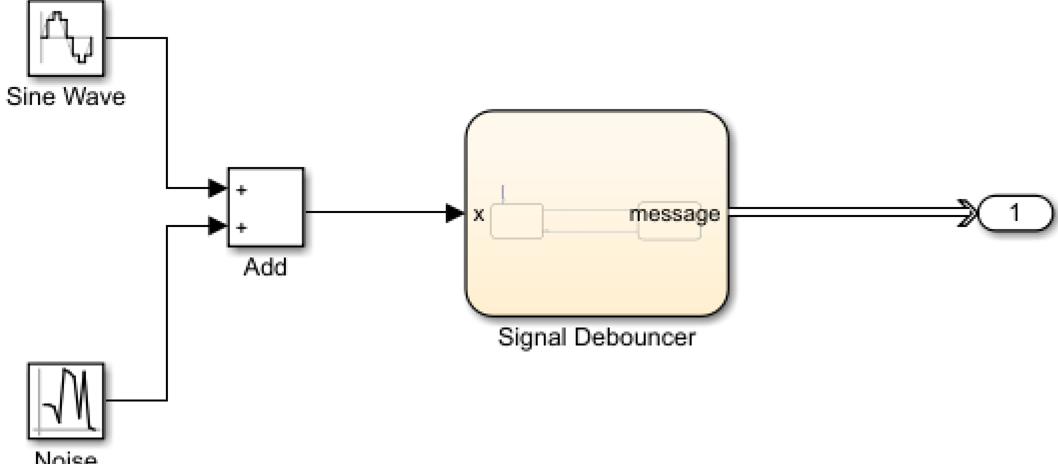
Create Components to Send Messages

In the model, there are two software components that output messages, Send and StateflowSend.

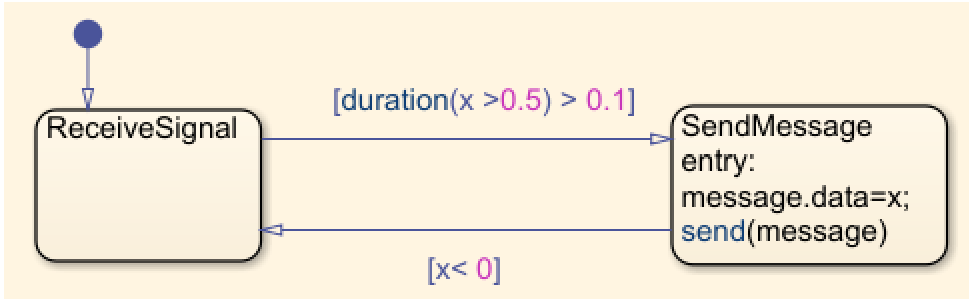
In the Send component, the Sine Wave block is the signal source. The block generates a sine wave signal with an amplitude of `1`. The sample time for the block is `0.1`. The Send block converts the signal to a message that carries the signal value as data. The Send component sends messages to Send Buffer 1.



In the StateflowSend component, another Sine Wave block generates a sine wave signal and a Noise block injects noise into the signal. The Noise block outputs a signal whose values are generated from a Gaussian distribution with mean of `0` and variance of `1`. The sample time of the block is `0.1`.



The Stateflow chart represents a simple logic that filters the signal and decides whether to send messages. If the value of the signal is greater than `0.5` for a duration greater than `0.1`, then the chart sends a message that carries the signal value. If the signal value is below `0`, then the chart transitions to the `ReceiveSignal` state. The StateflowSend component sends messages to Send Buffer 2.



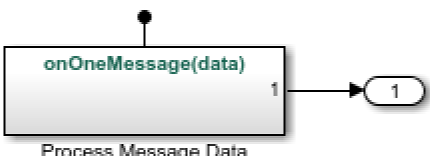
For more information about creating message interfaces, see "Establish Message Send and Receive Interfaces Between Software Components" (Simulink).

Create Components to Receive Messages

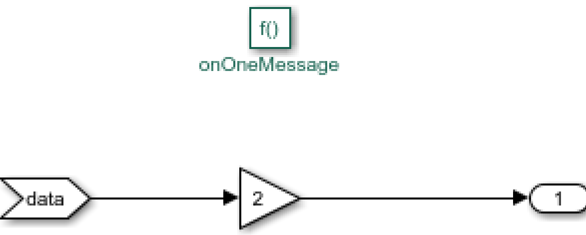
In the model, there are two software components that receive messages, Receive and Listener.

In the Receive component, a Receive block receives messages and converts the message data to signal values.

In the Listener component, there is a Simulink Function block. The block displays the function, onOneMessage(data), on the block face.

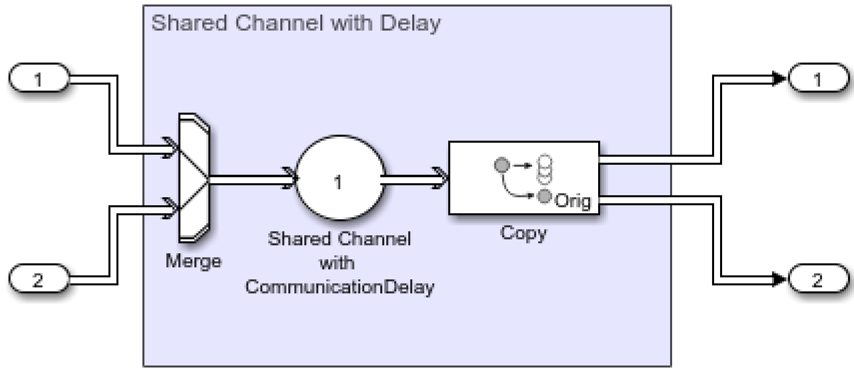


When a message arrives at Receive Buffer 2, the Listener block is notified and it takes the argument `data`, which is the value from the message data, as the input signal. In the block, `data` values are multiplied by `2`. The block outputs the new data value.



Routing Messages using SimEvents®

In the shared channel, the message paths originating from the two message-sending components are merged to represent a shared communication channel.



A SimEvents® Entity Input Switch block merges the message lines. In the block:

- **Number of input ports** specifies the number of message lines to be merged. The parameter value is `2` for two message paths.
- **Active port selection** specifies how to select the active port for message departure. If you select `All`, all of the messages arriving at the block are able to depart the block from the output port. If you select `Switch`, you can specify the logic that selects the active port for message departure. For this example, the parameter is set to `All`.

A SimEvents® Entity Server block is used to represent message transmission delay in the shared channel. In the block:

- **Capacity** is set to `1`, which specifies how many messages can be processed at a time.

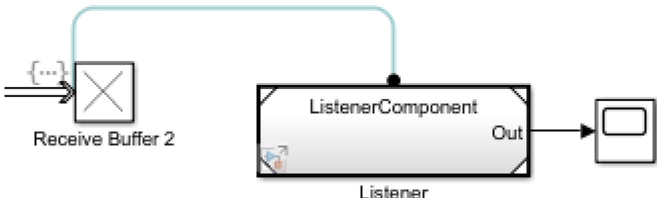
- **Service time value** is set to 1, which specifies how long it takes to process a message

A SimEvents® Entity Replicator block is used to generate identical copies of messages. In the block:

- **Replicas depart from** specifies if the copies leave the block from separate output ports or the same output port as the original messages. The parameter is set to `Separate output ports`.
- **Number of replicas** is set to 1, which specifies the number of copies generated for each message.
- **Hold original entity until all replicas depart** holds the original message in the block until all of its copies depart the block.

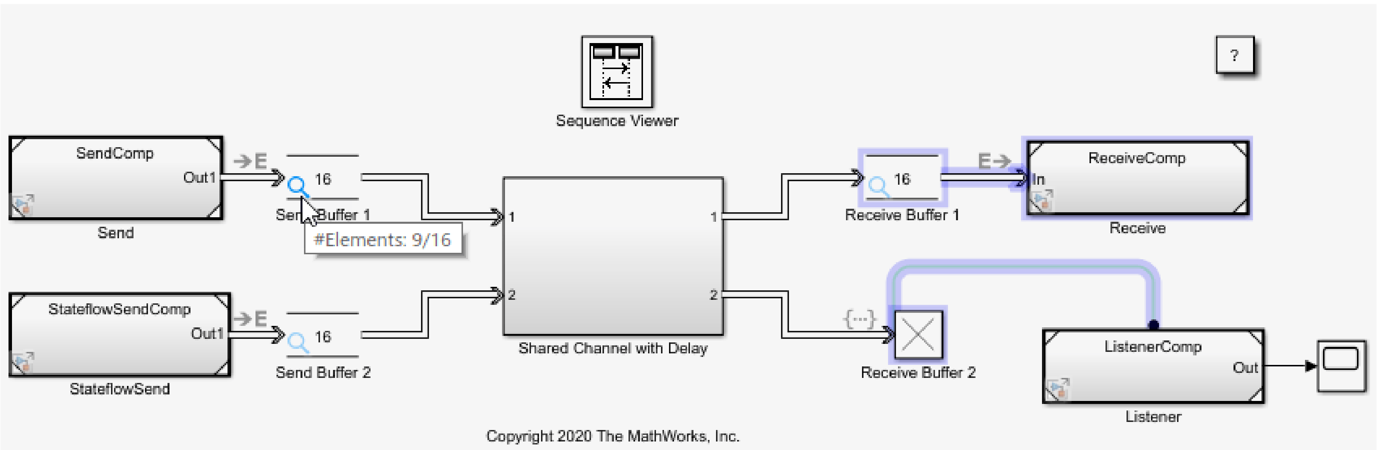
A SimEvents® Entity Terminator block is used to model Receive Buffer 2. In the block:

- Under the **Event actions** tab, in the **Entry action** field, you can specify MATLAB code that performs calculations or Simulink® function calls that are invoked when the message enters the block. In this example, `onOneMessage(entity)` is used to notify the Simulink Function block in the Listener component. To visualize the function call, under **Debug** tab, select **Information Overlays** and then **Function Connectors**.



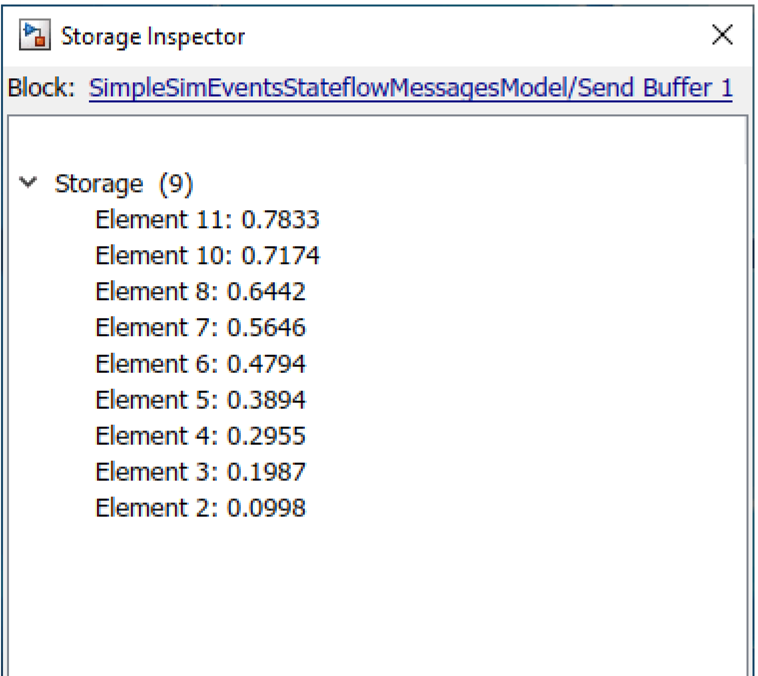
Simulate the Model and Review Results

Simulate the model. Observe that the animation highlights the messages flowing through the model. You can turn off the animation by right-clicking on the model canvas and setting **Animation Speed** to `None`.



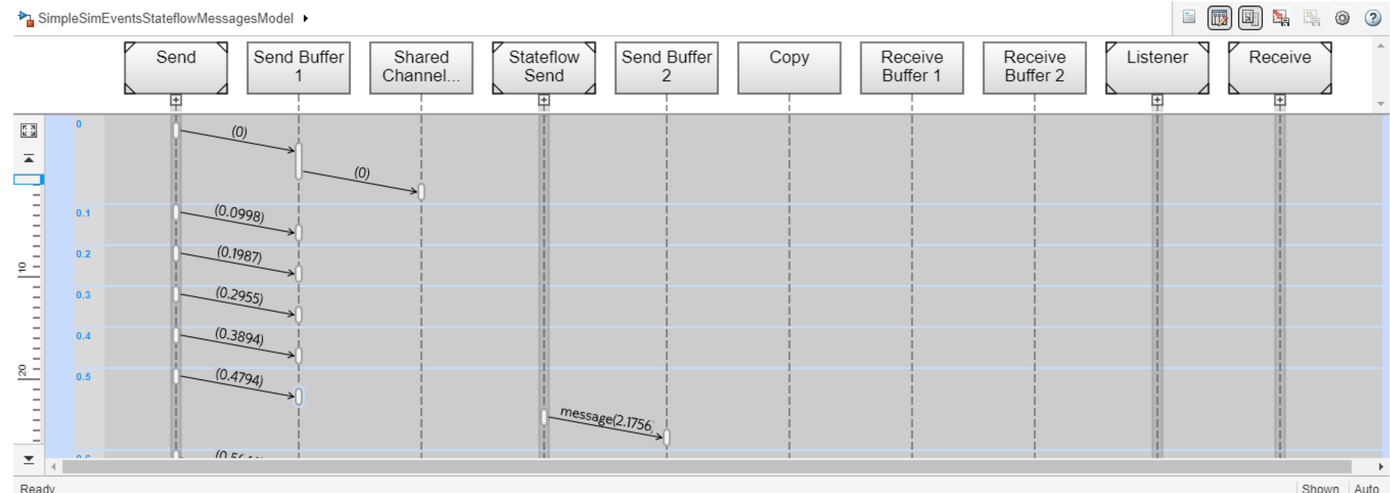
When you pause the animation, a magnifying glass appears on the blocks that store messages. If you point to the magnifying glass, you see the number of messages stored in the block.

To observe which messages are stored in the block, click the magnifying glass to open the Storage Inspector. For instance, the graphic below illustrates the messages stored in Send Buffer 1.



Turn the animation off and open the Sequence Viewer block to observe the Simulink Function calls and the flow of messages in the model.

For instance, observe the simulation time `0`, during which a message carrying value `0` is sent from the Send component to Send Buffer 1. From simulation time `0.1` to `0.5`, the Send component keeps sending messages to Send Buffer 1 with different data values. At time `0.5`, the StateflowSend component sends a message to Send Buffer 2. For more information about using the Sequence Viewer block, see “Use the Sequence Viewer to Visualize Messages, Events, and Entities” (Simulink).



See Also

Sine Wave | Send | Receive | Queue | Entity Input Switch (SimEvents) | Entity Server (SimEvents) | Entity Replicator (SimEvents) | Entity Terminator (SimEvents)

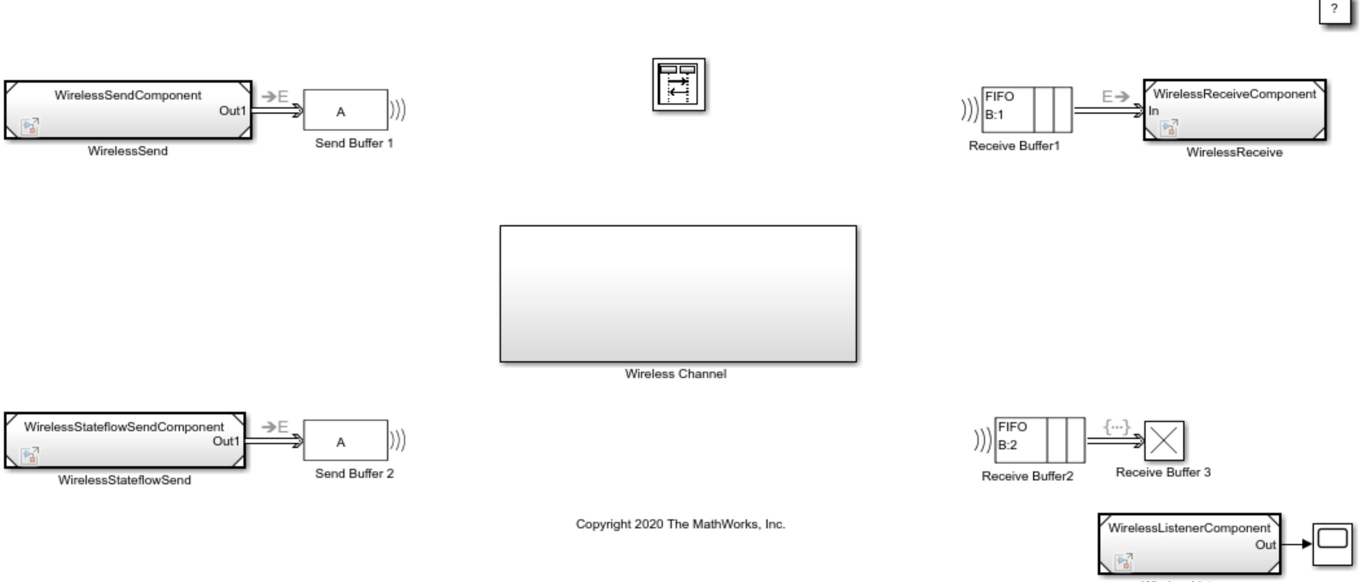
More About

- “Simulink Messages Overview” (Simulink)
- “Discrete-Event Simulation in Simulink Models” (SimEvents)
- “Model Wireless Message Communication with Packet Loss and Channel Failure” (Simulink)
- “Model an Ethernet Communication Network with CSMA/CD Protocol” (Simulink)

Model Wireless Message Communication with Packet Loss and Channel Failure

This example shows how to model wireless message communication with packet loss and channel failure by using Simulink® messages, Stateflow®, and SimEvents®.

In this model, there are two components that send messages and two components that receive messages. The messages are transmitted using a shared wireless channel with a transmission delay. A Stateflow® chart models message-sending logic in a wireless component and SimEvents® blocks model wireless message transmission, channel failure, and packet loss.



For an overview about messages, see “Simulink Messages Overview” (Simulink).

Create Components to Send and Receive Messages

In the model, there are two software components that output messages, WirelessSend and WirelessStateflowSend.

In the WirelessSend component, the Sine Wave block is the signal source. The Sine Wave block generates a sine wave with an amplitude of `1`. The block **Sample time** is set to `0.1`. The Send block converts the signal to a message that carries the data of the signal value. The WirelessSendComponent is connected to Send Buffer 1.

In the WirelessStateflowSend component, another Sine Wave block generates a sine wave signal and a Noise block is used to inject noise into the signal. The Noise block outputs a signal whose values are generated from a Gaussian distribution with mean of `0` and variance of `1`. The Stateflow® chart represents a simple logic that filters a signal and decides whether to send messages. The StateflowSend component sends messages to Send Buffer 2.

In the model, there are two software components that receive messages, WirelessReceive and WirelessListener.

In the WirelessReceive component, a Receive block receives messages and converts message data to signal values. The component is connected to Receive Buffer 1.

In the WirelessListener component, there is a Simulink Function block that runs the `onOneMessage(data)` function. When a message arrives at Receive Buffer 3, the Simulink Function block takes the argument `data`, which is the value from message data, as the input signal. In the block, the `data` values are multiplied by `2`. The block outputs the new data value.

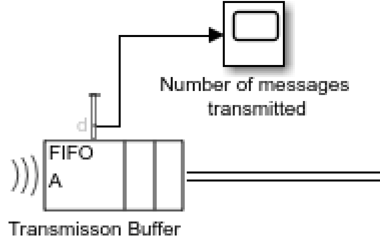
To learn more about creating these components, see “Build a Shared Communication Channel with Multiple Senders and Receivers” (Simulink).

Model Wireless Message Communication Using Multicasting

The WirelessSend and WirelessStateflowSend components send messages to Send Buffer 1 and Send Buffer 2, which are SimEvents® Entity Multicast blocks that can wirelessly transmit messages. The Transmission Buffer block is a SimEvents® multicast receive queue that can receive messages sent by Send Buffer 1 and Send Buffer 2.

To achieve this wireless communication between Send Buffer 1, Send Buffer 2, and the Transmission Buffer block that is inside the Wireless Channel block:

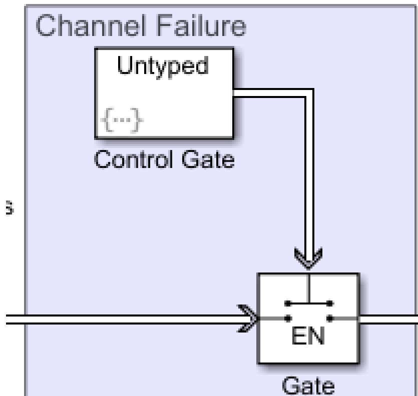
1. In the Send Buffer 1 and Send Buffer 2 blocks, set the **Multicast tag** parameter to `A`.
2. In the Transmission Buffer block, set the **Multicast tag** parameter to `A`.



The **Multicast tag** parameter defines from which Entity Multicast blocks the messages are received.

Model Channel Failure

A SimEvents® Entity Gate block is used to model channel failure. The block has two input ports. One input port is for incoming messages from Transmission Buffer. The second input port is a control port to decide when to open the gate.

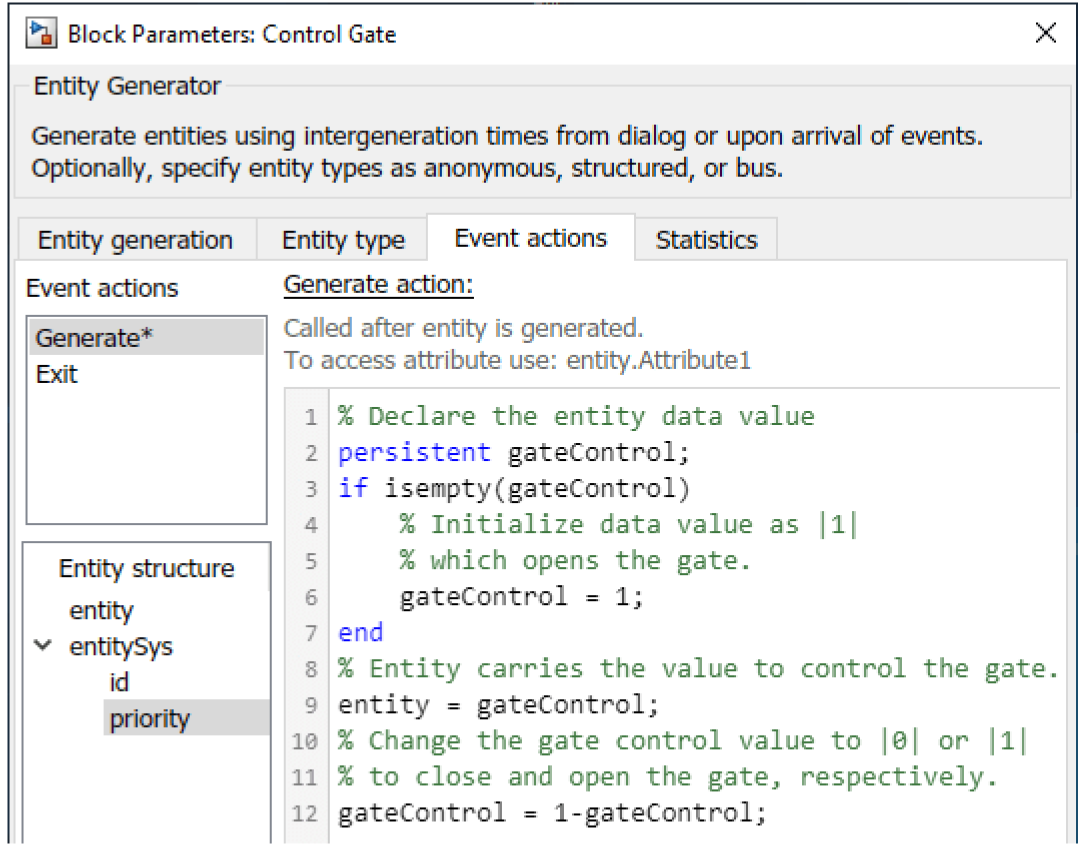


Set the **Operating mode** parameter for the Gate block to `Enable gate`. In this mode:

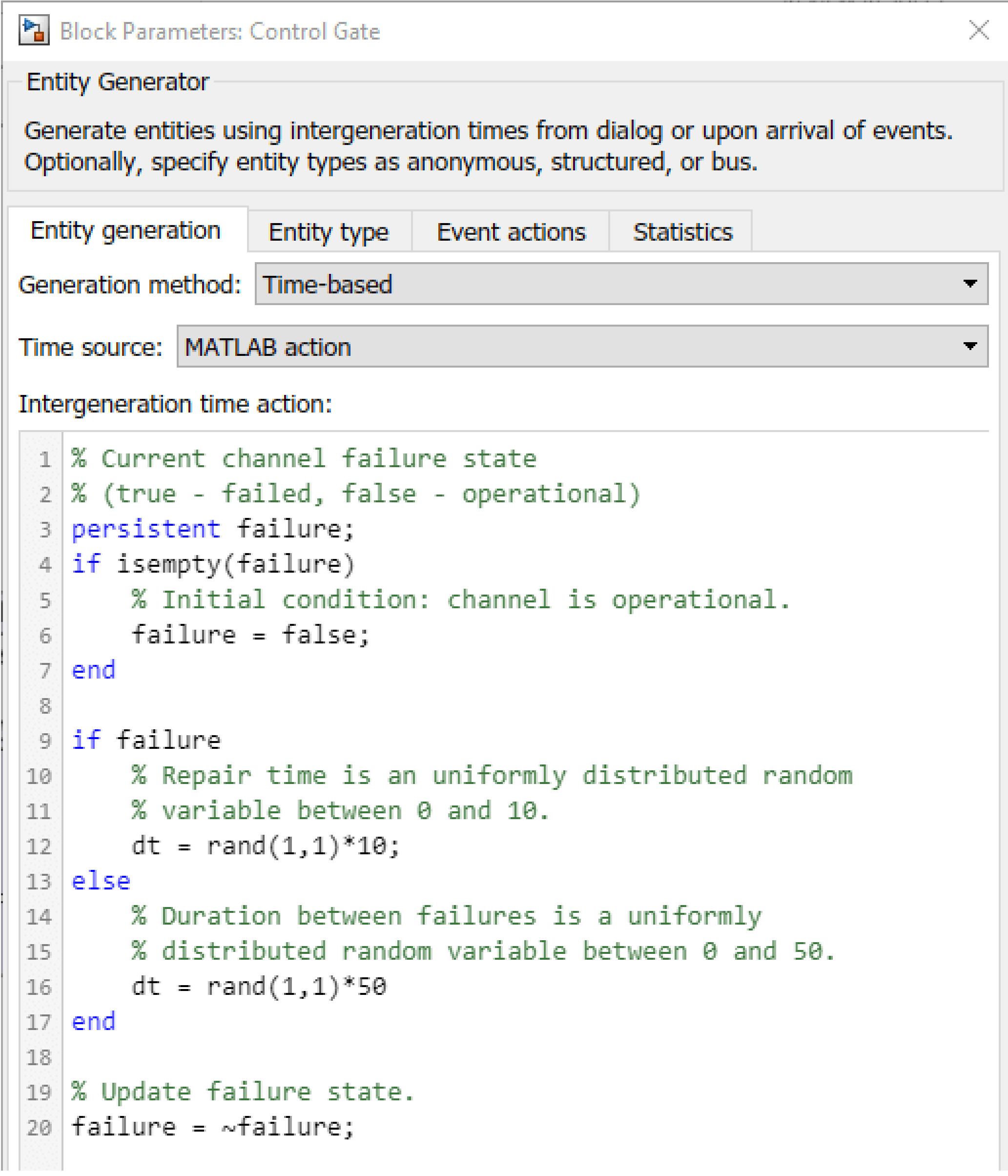
- The block opens the gate and permits messages to advance when it receives an entity carrying a value that is greater than `0` from its control port. This represents an operational channel.
- The block closes the gate and blocks messages passing if an entity carries data whose value is less than or equal to `0`. This represents a channel failure.

To control the Gate block, you can use the SimEvents® Entity Generator block, which is labeled Control Gate in this example, to generate entities carrying different data values.

In the Control Gate block, in **Event actions**, in the **Generate action** field, the code below is used to generate entities to open and close the Gate block. Initially, entity data is `1` and the gate is open and the channel is in operational state. When a new entity is generated, its value changes to `0`, which closes the gate. Every generated entity changes the status of the gate from open to closed or from closed to open.



In the Control Gate block, in the **Intergeneration time action** field, the code below is used to represent the operational and failed state of the channel. The code initializes the channel as operational. `dt` is the entity intergeneration time and is used to change the status of channel because each generated entity changes the status of the Gate block.

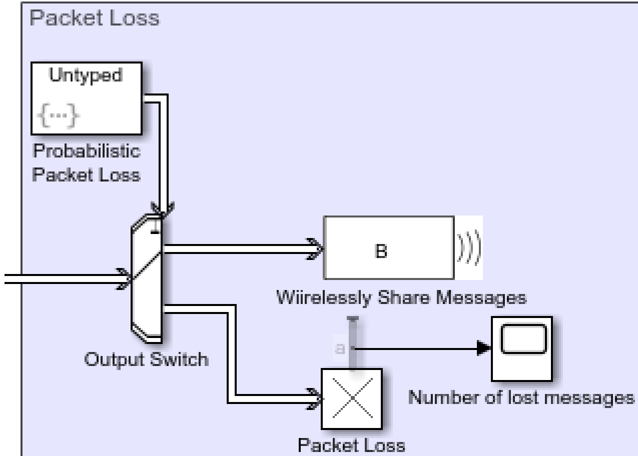


In the code, the repair time is generated from a uniform distribution that takes values between `0` and `10`. The time interval between the failures is generated from another uniform distribution that takes values between `0` and `50`.

Model Packet Loss

To model the packet loss, a SimEvents® Entity Output Switch block is used.

The block has two input ports. One input port accepts messages. The other input port accepts entities that determine the output port selection. If the entity is set to `1`, the block selects output port `1` to forward messages to the Wirelessly Share Messages block. If the entity is set to `2`, the block selects output port `2`, which is connected to an Entity Terminator block that represents packet loss.



In the Output Switch block:

- The **Number of output ports** is set to `2`.
- To determine which output is selected, the **Switching criterion** is set to `From control port` and **Initial port selection** is set to `1`.

To model a `0.1` probability of packet loss, in the Probabilistic Packet Loss block, select the **Event actions** tab, and in the **Generate action** field includes this code:

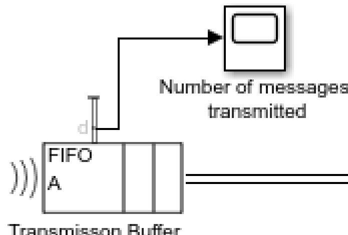
```
persistent rngInit;
if isempty(rngInit)
    seed = 12345;
    rng(seed);
    rngInit = true;
end

% Pattern: Uniform distribution
% m: Minimum, M: Maximum
m = 0; M = 1;
x = m + (M - m) * rand;

% x is generated from uniform distribution and
% takes values between |0| and |1|.
if x > 0.1
   % Entity carries data |1| and this forces Output switch to select
   % output |1| to forward entities to receive components.
   entity  = 1;
else
   % Entity carries data |2| and this forces Output switch to select
   % output |2| and this represents a packet loss.
    entity = 1;
end
```

This means that entities entering the control port have a `0.9` probability of being set to `1`, which makes the block output messages to the Wirelessly Share Messages block.

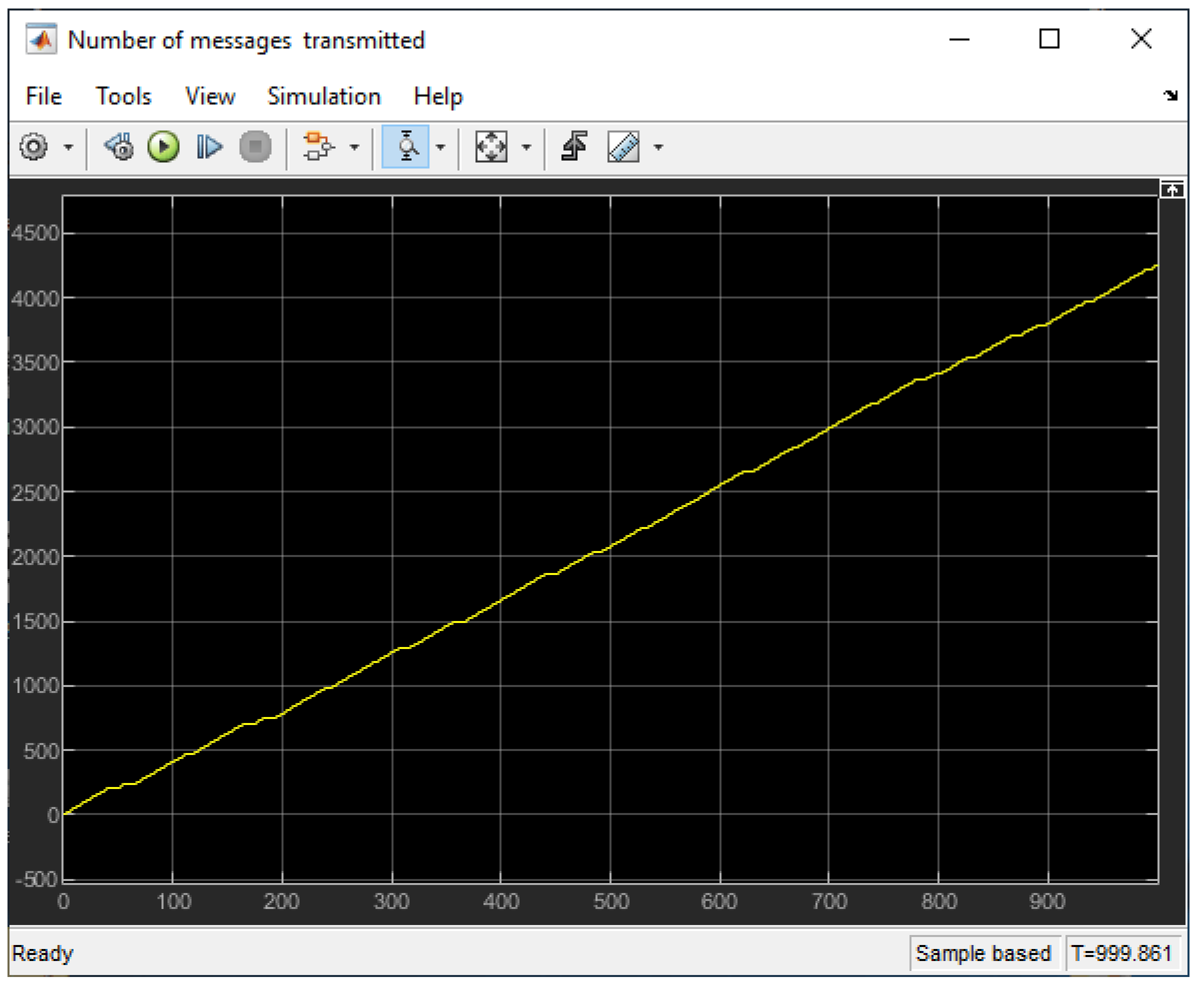
Simulate the Model and Review results

Simulate the model.

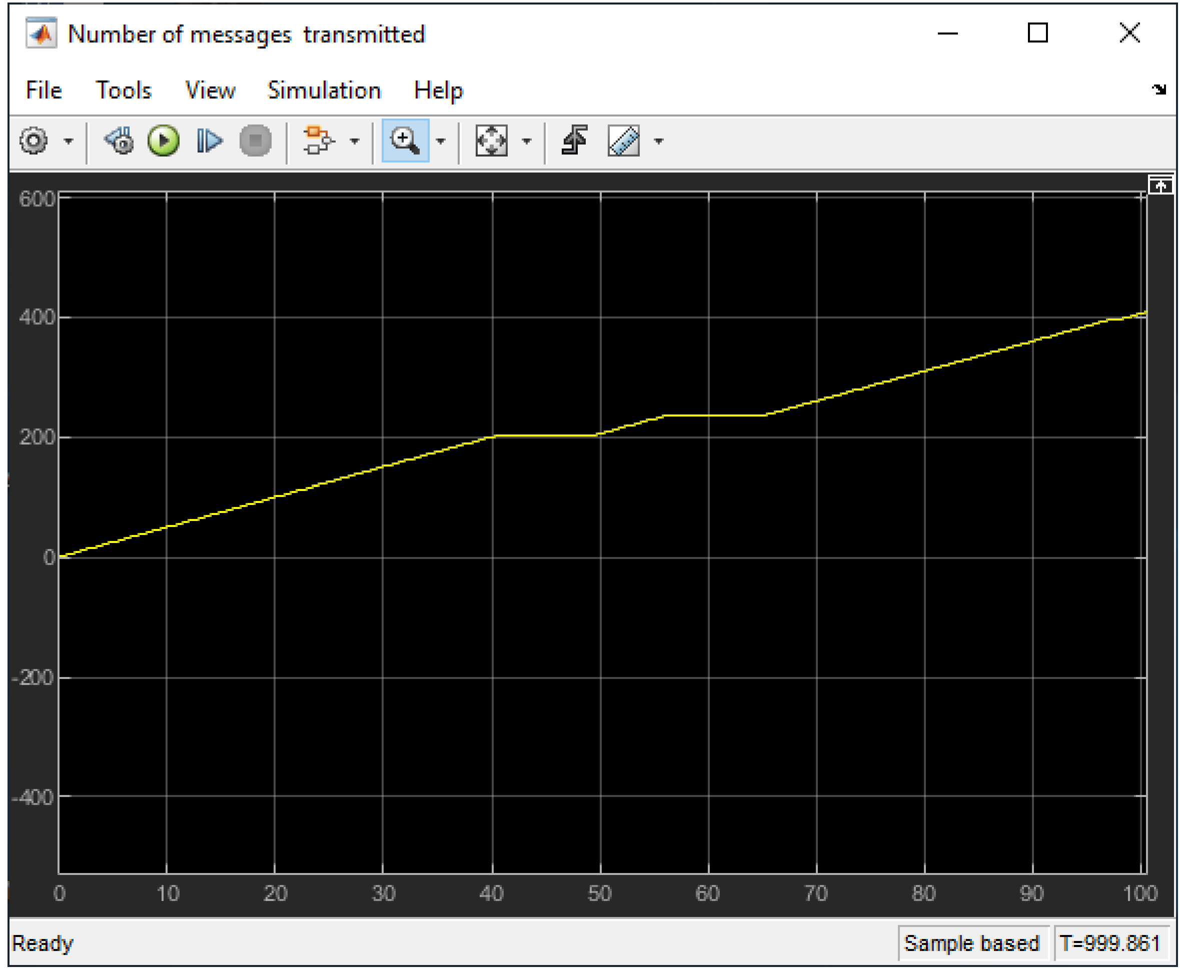
- Open the Scope block connected top the Transmission Buffer block. The block displays the total number of messages transmitted through the shared channel.



4255 messages are transmitted through the channel.

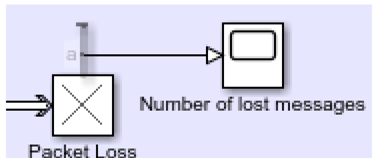


The plot also displays the channel failures. For example, zoom into the first 100 seconds. Observer that the channel failure occurs between 40 and 49 during which message transmission is blocked.

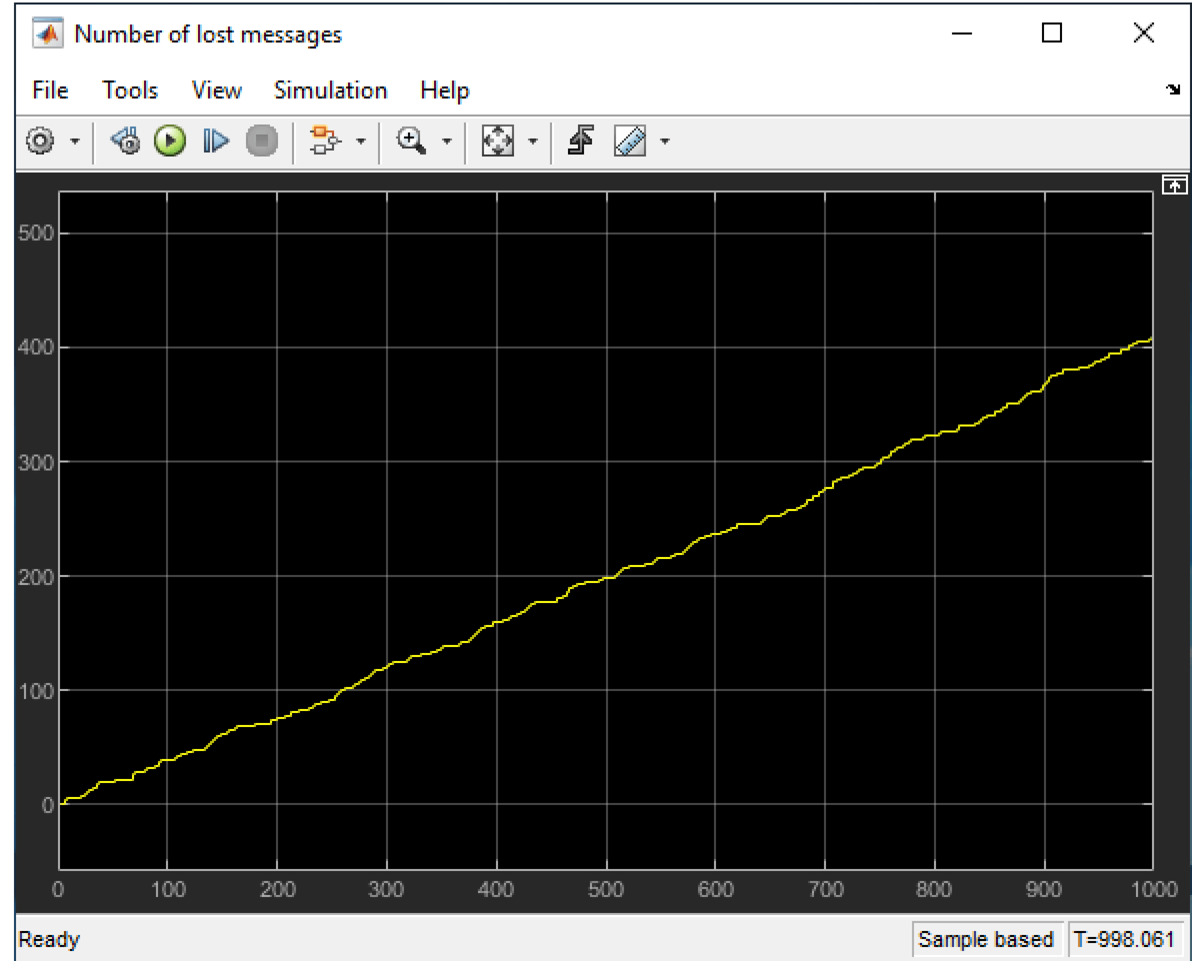


Open the Data Inspector to visualize the entities that control the Gate. Entity data changes from `1` to `0` for each generated entity.

To see the number of lost messages, open the Scope block connected to the Packet Loss block.



409 messages are lost during transmission. This is 9.6 percent of the messages.



See Also

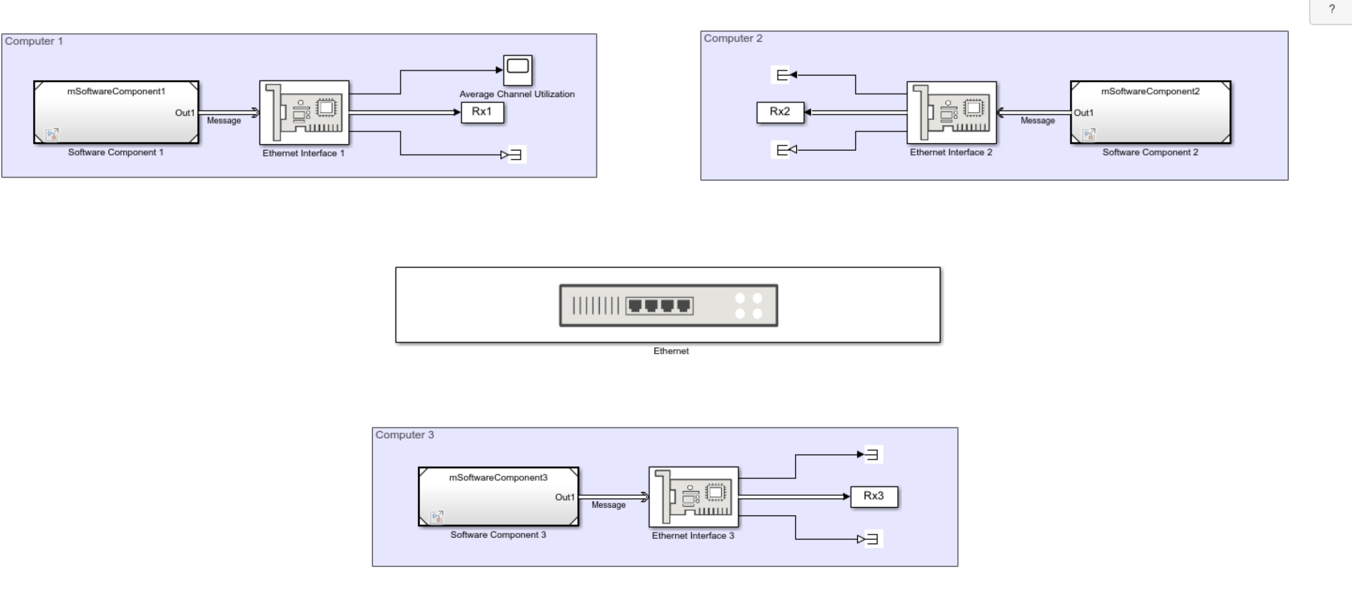
Sine Wave | Send | Receive | Queue | Entity Terminator (SimEvents) | Entity Output Switch (SimEvents) | Entity Gate (SimEvents) | Entity Multicast (SimEvents)

More About

- “Simulink Messages Overview” (Simulink)
- “Discrete-Event Simulation in Simulink Models” (SimEvents)
- “Build a Shared Communication Channel with Multiple Senders and Receivers” (Simulink)
- “Model an Ethernet Communication Network with CSMA/CD Protocol” (Simulink)

Model an Ethernet Communication Network with CSMA/CD Protocol

This example shows how to model an Ethernet communication network with CSMA/CD protocol using Simulink® messages and SimEvents®. In the example, there are three computers that communicate through an Ethernet communication network. Each computer has a software component that generates data and an Ethernet interface for communication. Each computer attempts to send the data to another computer with a unique MAC address. An Ethernet interface controls the interaction between a computer and the network by using a CSMA/CD communication protocol. The protocol is used to respond to collisions that occur when multiple computers send data simultaneously. The Ethernet component represents the network and the connection between the computers.

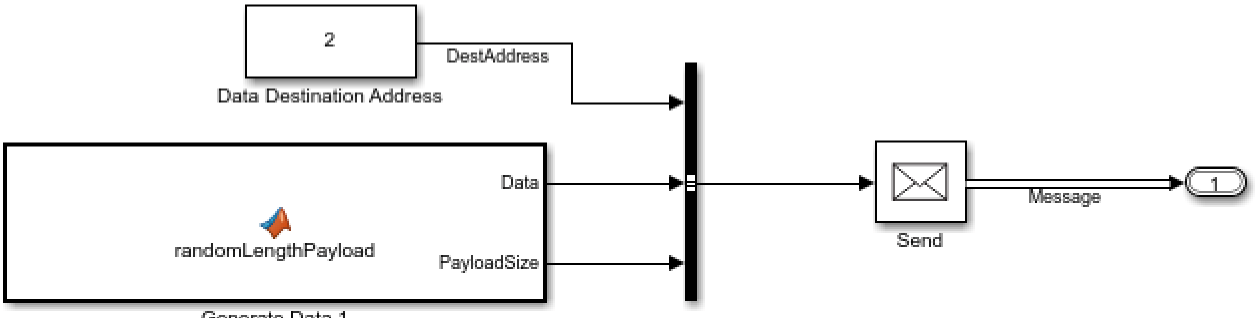


Software Components

In the model, each software component generates data (payload) and combines the data, its size, and its destination into a message. Then, the message is sent to the Ethernet interface for communication.

In each Software Component subsystem:

- A MATLAB Function block generates data with a size between `46` and `1500` bytes [1].
- A Constant block assigns destination addresses to data.
- A Bus Creator block converts the `Data`, `PayloadSize`, and `DestAddress` signals to a nonvirtual bus object called `dataPacket`.
- A Send block converts `dataPacket` to a message.
- An Outport block sends the message to the Ethernet interface for communication.

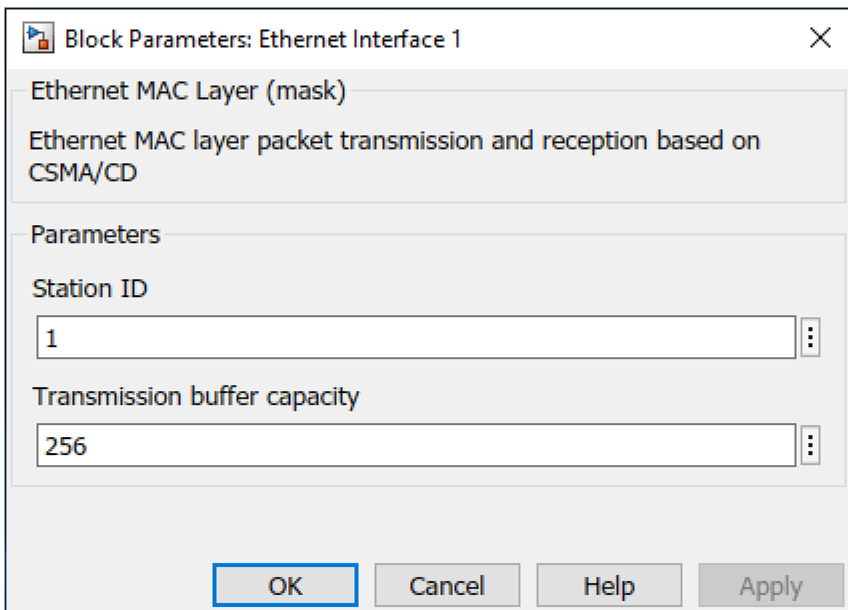


Each computer generates data with a different rate. You can change the data generation rate from the sample time of the MATLAB Function block.

To learn the basics of creating message send and receive interfaces, see “Establish Message Send and Receive Interfaces Between Software Components” (Simulink).

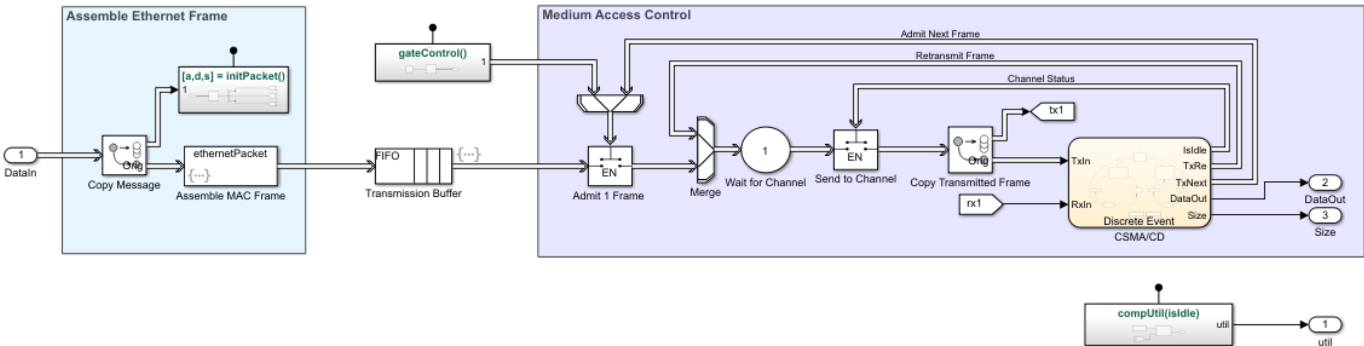
Ethernet Interface

Double-click Ethernet Interface 1. Observe that you can specify the **Station ID** and **Transmission buffer capacity**.



The Ethernet Interface subsystems have three main parts:

1. Assemble Ethernet Frame — Converts an incoming message to an Ethernet (MAC) frame.
2. Transmission Buffer — Stores Ethernet frames for transmission.
3. Medium Access Control — Implements a CSMA/CD protocol for packet transmission [2].

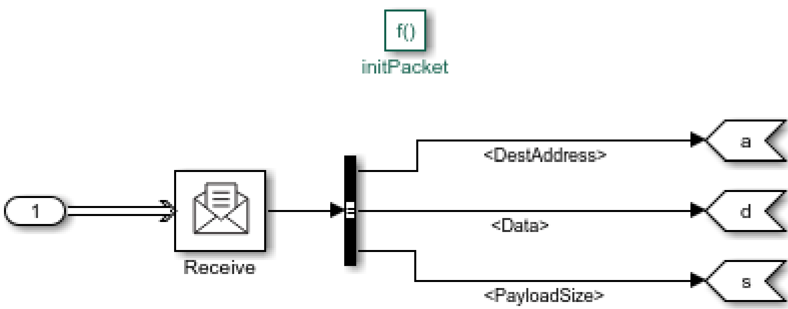


Assemble Ethernet Frame

The Assemble Ethernet Frame blocks convert messages to Ethernet frames by attaching Ethernet-specific attributes to the message [1].

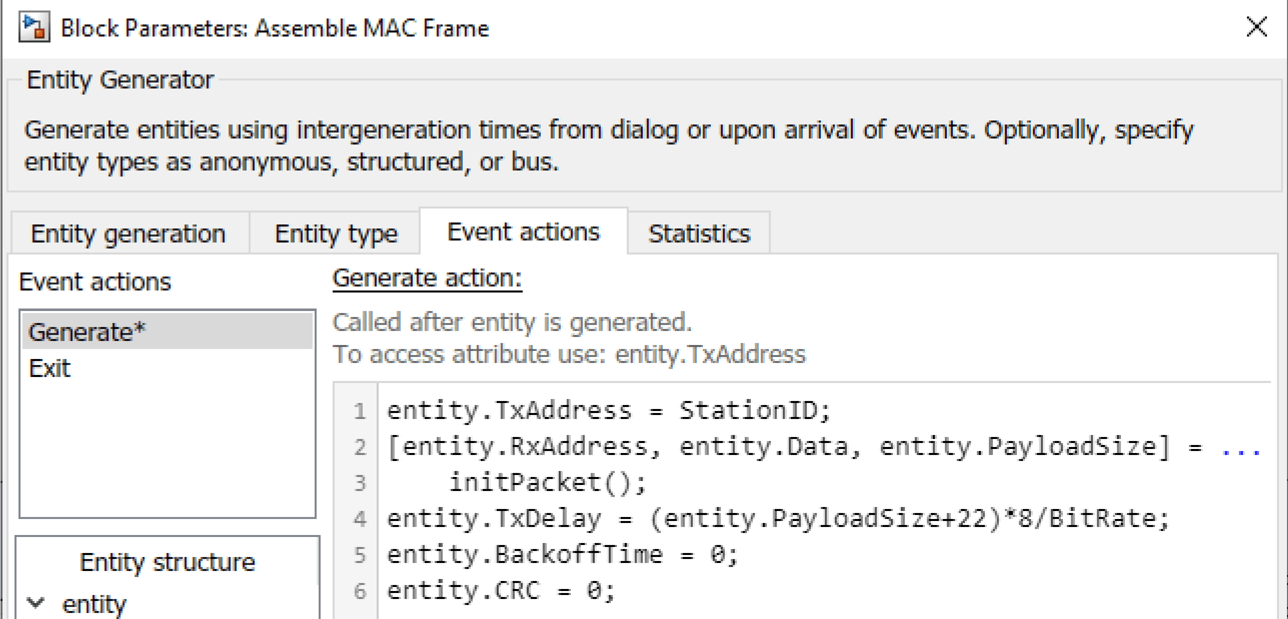
In the packet assembly process:

- A SimEvents® Entity Replicator block labeled Copy Message copies an incoming message. The original message is forwarded to a SimEvents® Entity Generator block labeled Assemble MAC Frame. Because the Entity Generator block **Generation method** parameter is set to `Event-based`, it immediately produces an entity when the original message arrives at the block. A copy of the message is forwarded to a Simulink Function block with the `initPacket()` function. The terms *message* and *entity* are used interchangeably between Simulink® and SimEvents®.
- The Simulink Function block transfers the data, its size, and its destination address to the Assemble MAC Frame block for frame assembly.



- The Assemble MAC Frame block generates the Ethernet frames that carry both Ethernet-specific attributes and values transferred from the Simulink Function block.

Assemble MAC Frame block calls the `initPacket()` function as an action that is invoked by each frame generation event.



These are the attributes of the generated Ethernet frame:

- `entity.TxAddress` is `StationID`.
- `entity.RxAddress`, `entity.Data`, and `entity.PayloadSize` are assigned the values from the Simulink Function block.
- `entity.TxDelay` is the transmission delay. It is defined by the payload size and the bitrate. The `Bitrate` parameter is specified by an initialization function in the Model Properties.
- `entity.CRC` is the cyclic redundancy check for error detection.

Transmission Buffer

The transmission buffer stores entities before transmission by using a first-in-first-out (FIFO) policy. The buffer is modeled by a Queue block.

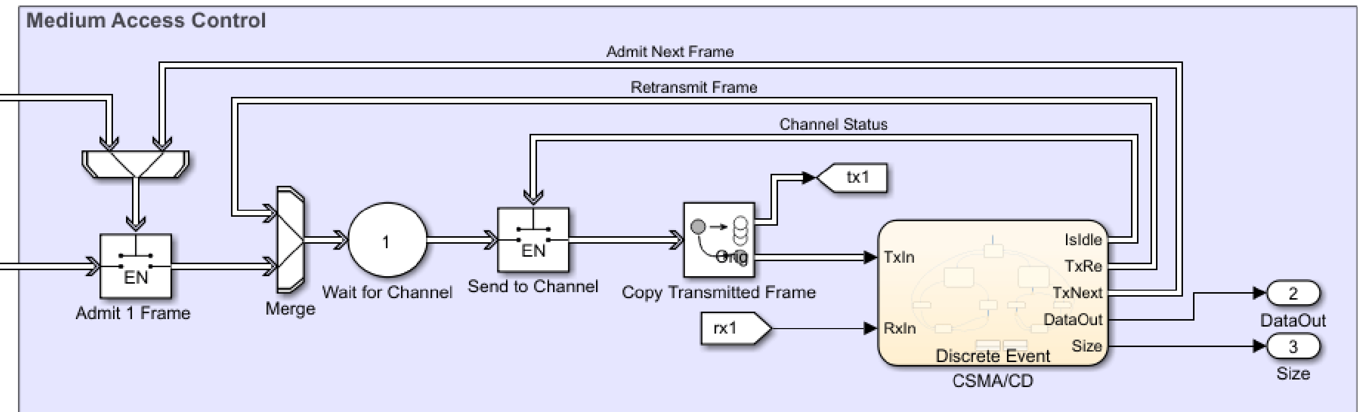
The capacity of the queue is determined by the **Transmission buffer capacity** parameter.

Medium Access Control

The Medium Access Control blocks are modeled by using six SimEvents® blocks.

- An Entity Gate block labeled Admit 1 Frame is configured as an enabled gate with two input ports. One input port allows frames from the Transmission Buffer block. The other input port is called the control port, which accepts messages from the CSMA/CD block. The block allows one frame to advance when it receives a message with a positive value from CSMA/CD block.
- An Entity Input Switch block labeled Merge merges two paths. One input port accepts new frames admitted by the Admit 1 frame block and the other input port accepts frames for retransmission that are sent by the CSMA/CD block.
- An Entity Server block labeled Wait for Channel models the back off time of a frame before its retransmission through the channel.

- Another Entity Gate block labeled Send to Channel opens the gate to accept frames when the channel is idle. The channel status is communicated by the CSMA/CD chart.
- An Entity Replicator block labeled Copy Transmitted Frame generates a copy of the frame. One frame is forwarded to the Ethernet network, and the other is forwarded to the CSMA/CD chart.
- A Discrete-Event Chart block labeled CSMA/CD represents the state machine that models the CSMA/CD protocol.



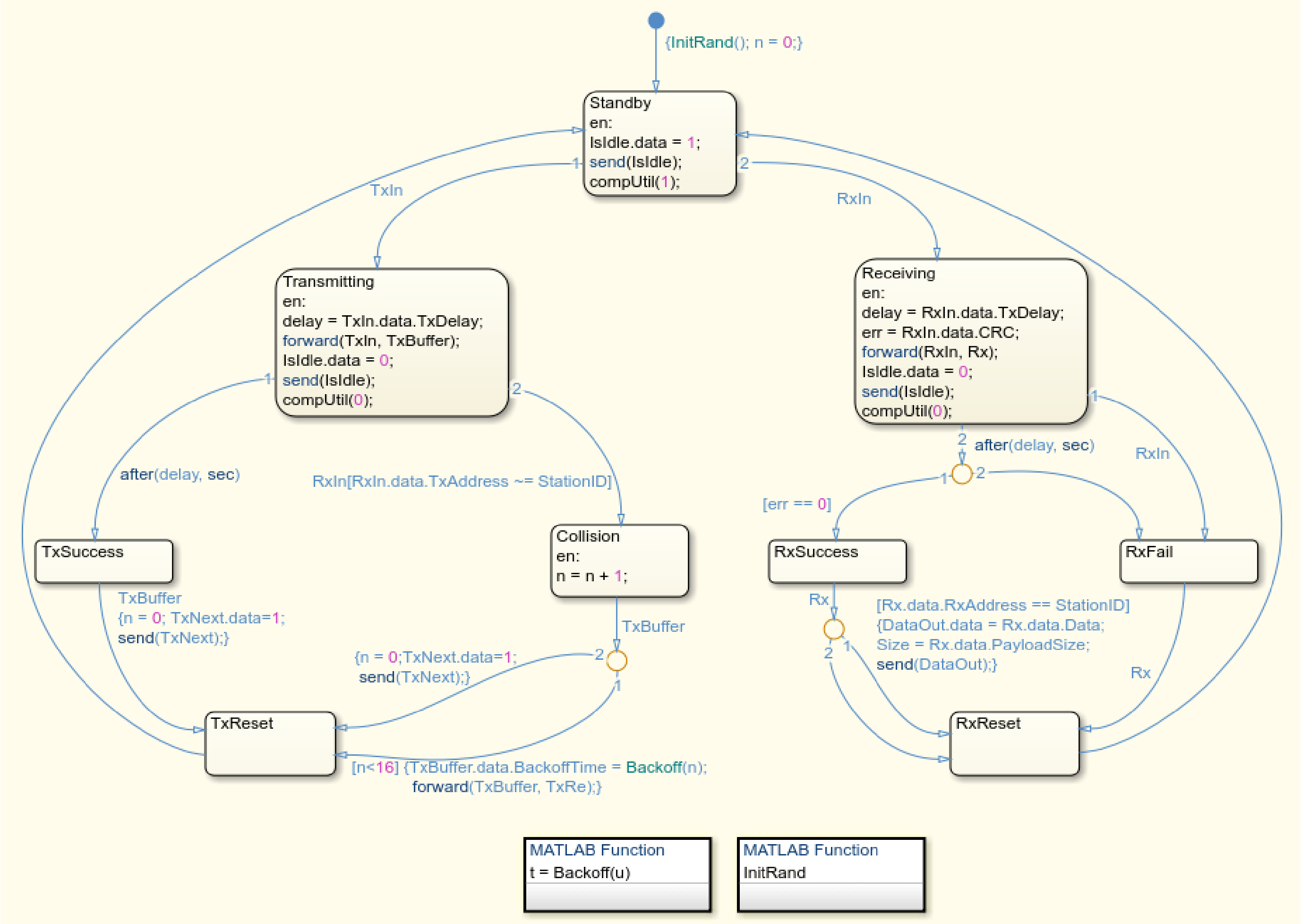
CSMA/CD Protocol

The CSMA/CD protocol [2] is modeled by a Discrete-Event Chart block that has two inputs:

- `TxIn` — Copy of the transmitted frame.
- `RxIn` — Received frame from the Ethernet network.

The chart has five outputs:

- `IsIdle` — Opens the Send to Channel gate to accept frames when the value is `1`, and closes the gate when the value is `0`.
- `TxRe` — Retransmitted frame that is forwarded to the Merge block if there is a collision detected during its transmission.
- `TxNext` — Opens the Admit 1 Frame gate to accept new frames when the value is `1`.
- `DataOut` — Received data.
- `Size` — Size of the received data.



Transmitting and Receiving Messages

The block is initially in the `Standby` state and the channel is idle.

If the block is transmitting, after a delay, the block attempts to transmit the message and `Isle.data` is set to `0` to declare that the channel is in use.

If the transmission is successful, the block sets `TxNext.data` to `1` to allow a new message into the channel and resets to the `Standby` state.

If there is a collision, the block resends the message after delaying it for a random back off time. *n* is the counter for retransmissions. The block retransmits a message a maximum of `16` times. If all of the retransmission attempts are unsuccessful, then the block terminates the message and allows the entry of a new message. Then it resets to `StandBy`.

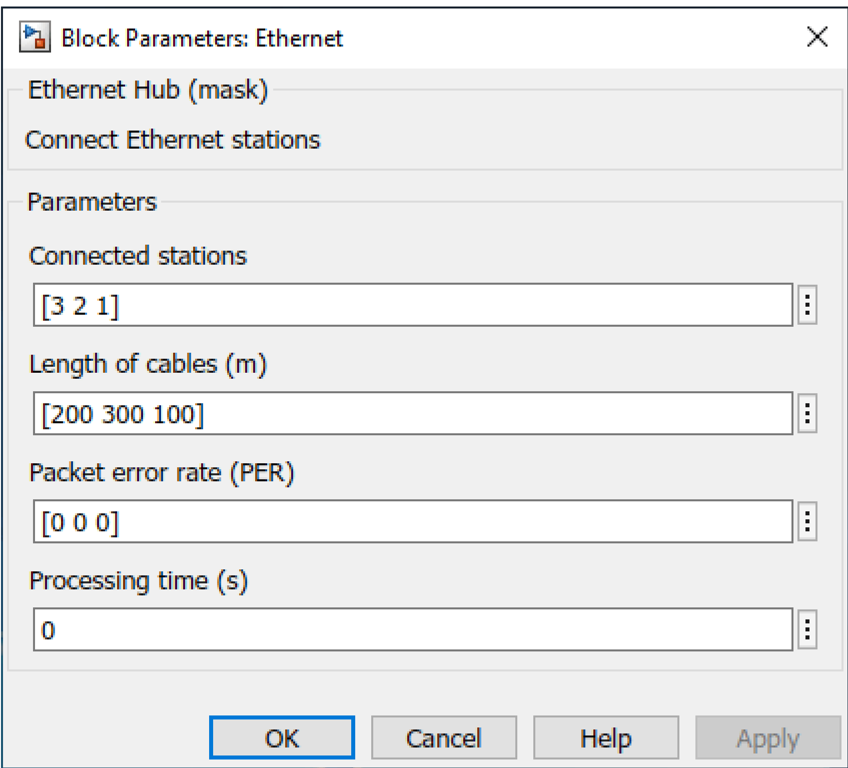
Similarly, the block can receive messages from other computers. If there is no error, the messages are successfully received and the block outputs the received data and its size.

Ethernet Hub

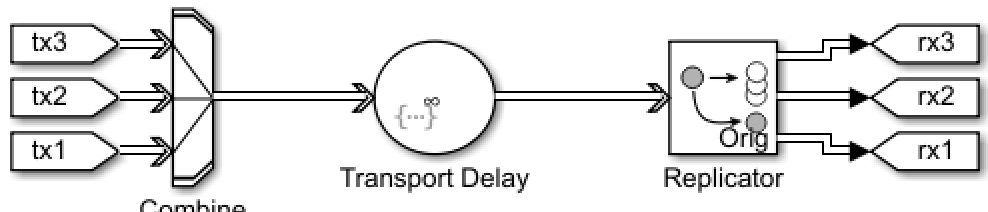
The Ethernet component represents the communication network and the cabled connections of the computers to the network.

Double-click the Ethernet block to see its parameters.

- **Connected stations** — These values are assigned to `Stations`, which is a vector with the station IDs as elements.
- **Length of cables (m)** — These values are assigned to `CableLength` and represent the length of the cables, in meters, for each computer connected to the hub.
- **Packet error rate (PER)** — These values are assigned to `PER` and represent the rate of error in message transmission for each computer.
- **Processing time (s)** — These values are assigned to `ProcessingTime` and it represents the channel transmission delay.

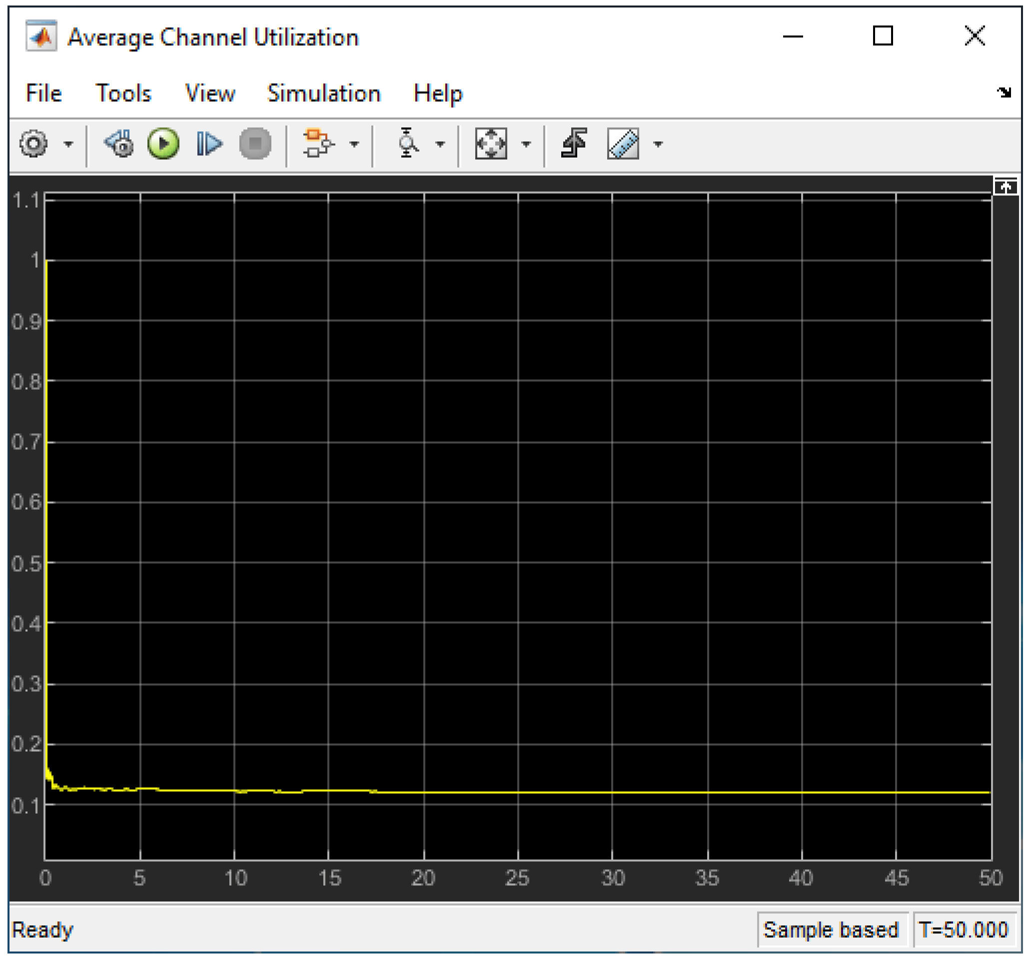


Three SimEvents® blocks are used to model the Ethernet network. The three computer connections are merged by using an Entity Input Switch block. An Entity Server block is used to model the channel transmission delay based on the cable length. An Entity Replicator block copies the transmitted message and forwards it to the three computers.

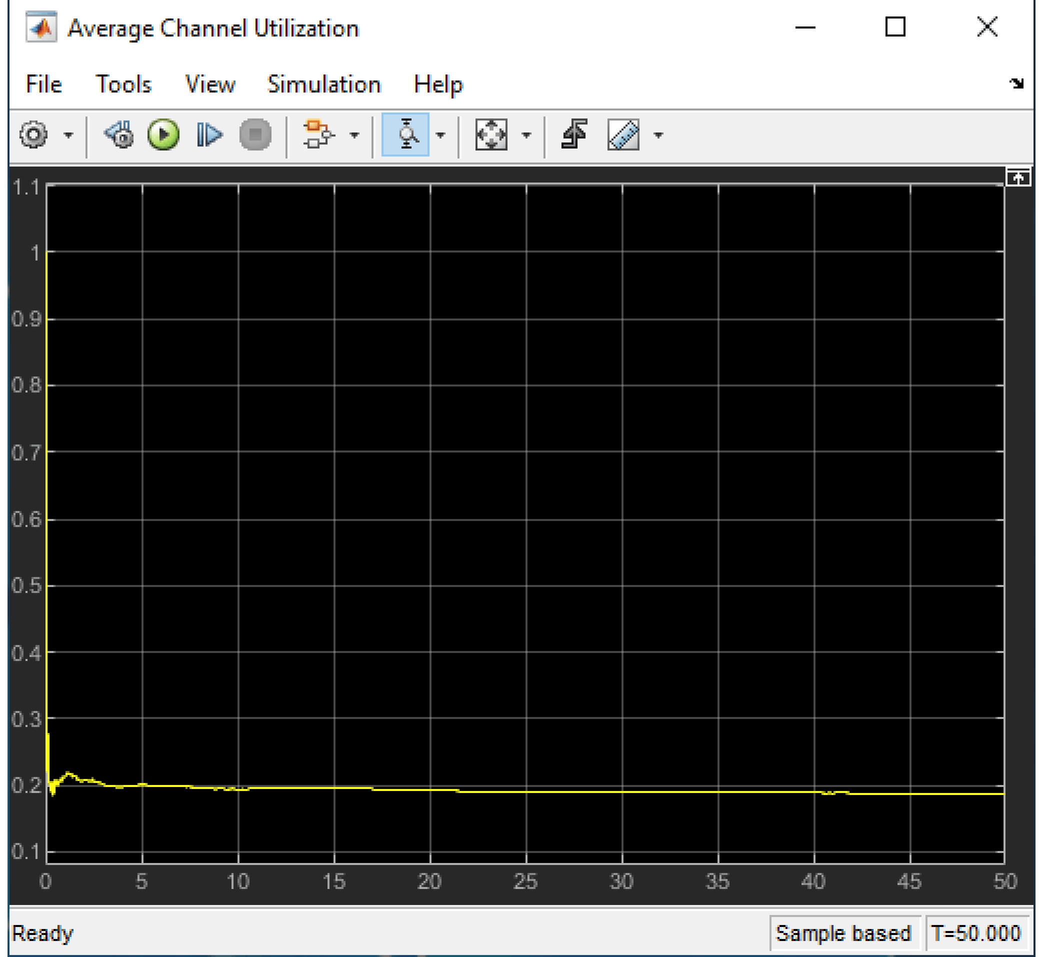


Simulate the Model and Review the Results

Simulate the model and open the Scope block that displays the average channel utilization. The channel utilization converges to approximately `0.12`.



Open Software Component 1 as a top model and change the data generation rate by setting the **Sample time** of the Generate Data 1 block to `0.01`. Run the simulation again and observe that the channel utilization increases to `0.2`.

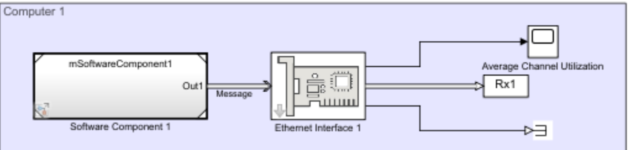


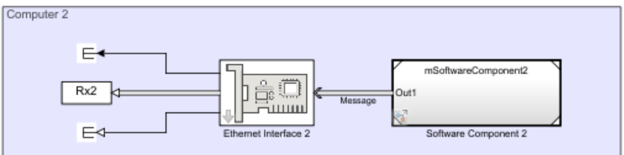
Connect New Computers to the Network

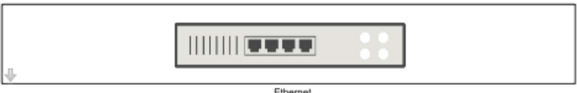
You can connect more computers to the network.

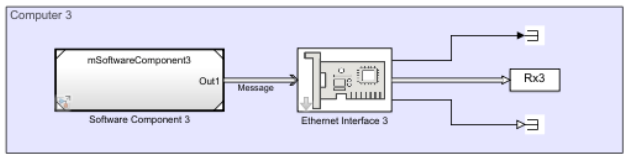
To add a new computer to the network:

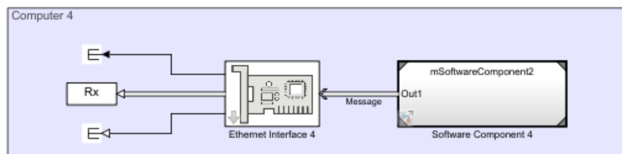
- Copy an existing computer and assign a new ID by double-clicking the Ethernet Interface block. In this example, new computer has ID 4.





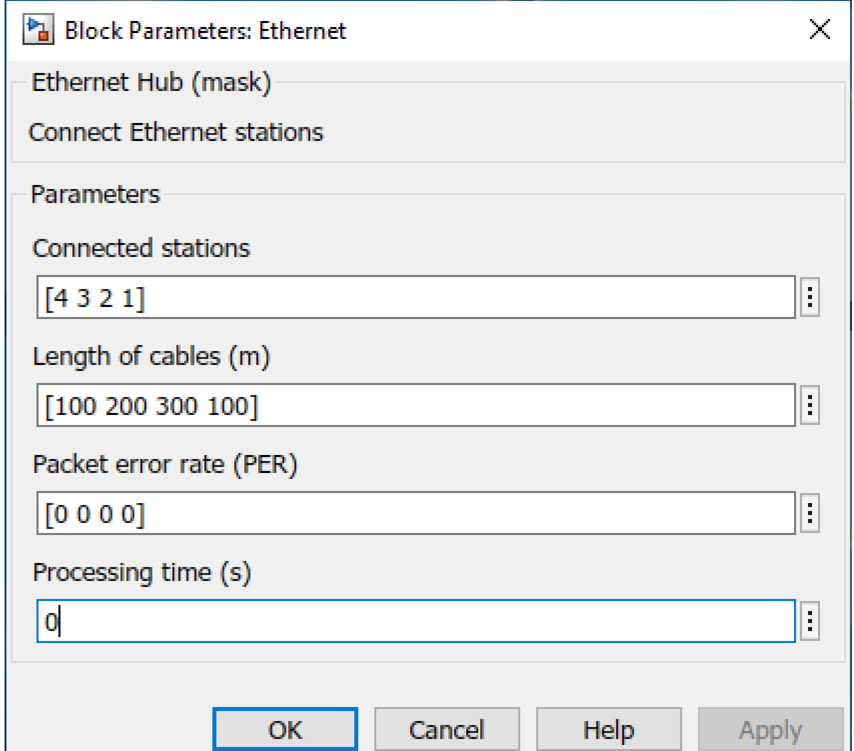






Copyright 2019 The MathWorks, Inc.

- Double-click the Ethernet block and add a station ID, cable length, and packet error rate for the new computer.



References

1 Ethernet frame - Wikipedia (https://en.wikipedia.org/wiki/Ethernet_frame)

2 Carrier-sense multiple access with collision detection - Wikipedia (https://en.wikipedia.org/wiki/Carrier-sense_multiple_access_with_collision_detection)

See Also

Send | Receive | Queue | Entity Input Switch (SimEvents) | Entity Replicator (SimEvents) | Discrete-Event Chart (SimEvents) | Entity Generator (SimEvents) | Entity Gate (SimEvents)

More About

- “Simulink Messages Overview” (Simulink)
- “Discrete-Event Simulation in Simulink Models” (SimEvents)
- “Build a Shared Communication Channel with Multiple Senders and Receivers” (Simulink)

Use Actions in Charts

- “Eliminate Redundant Code by Combining State Actions” on page 14-2
- “Operations for Stateflow Data” on page 14-4
- “Supported Symbols in Actions” on page 14-11
- “Call Extrinsic MATLAB Functions in Stateflow Charts” on page 14-13
- “Call C Library Functions in C Charts” on page 14-16
- “Access MATLAB Functions and Workspace Data in C Charts” on page 14-20
- “Control Function-Call Subsystems by Using bind Actions” on page 14-29
- “Control Chart Execution by Using Temporal Logic” on page 14-35
- “Model Bang-Bang Temperature Control System” on page 14-51
- “Control Oscillations by Using the duration Operator” on page 14-55
- “Implement an Automatic Transmission Gear System by Using the duration Operator” on page 14-58
- “Count Events by Using the temporalCount Operator” on page 14-61
- “Detect Changes in Data and Expression Values” on page 14-63
- “Design a Game by Using Stateflow” on page 14-75
- “Model an Automatic Transmission Controller” on page 14-79
- “Vehicle Electrical and Climate Control Systems” on page 14-90
- “Developing the Apollo Lunar Module Digital Autopilot” on page 14-96
- “Design a Guidance System in MATLAB and Simulink” on page 14-105

Eliminate Redundant Code by Combining State Actions

You can combine `entry`, `during`, and `exit` actions that execute the same tasks in a state.

By combining state actions that execute the same tasks, you eliminate redundant code. For example:

Separate Actions	Equivalent Combined Actions

`entry:`
`  y = 0;`
`  y=y+1;`
`during: y=y+1;`
	`entry: y = 0;`
	`entry, during: y=y+1;`

`en:`
`  fcn1();`
`  fcn2();`
`du: fcn1();`
`ex: fcn1();`
	`en, du, ex: fcn1();`
	`en: fcn2();`

Combining state actions this way produces the same chart execution behavior (semantics) and generates the same code as the equivalent separate actions.

How to Combine State Actions

Combine a set of `entry`, `during`, and/or `exit` actions that perform the same task as a comma-separated list in a state. Here is the syntax:

```
entry, during, exit: task1; task2;...taskN;
```

You can also use the equivalent abbreviations:

```
en, du, ex: task1; task2;...taskN;
```

Valid Combinations

You can use any combination of the three actions. For example, the following combinations are valid:

- en, du:
- en, ex:
- du, ex:
- en, du, ex:

You can combine actions in any order in the comma-separated list. For example, `en, du:` gives the same result as `du, en:`.

Invalid Combinations

You cannot combine two or more actions of the same type. For example, the following combinations are invalid:

- en, en:
- ex, en, ex:
- du, du, ex:

If you combine multiple actions of the same type, you receive a warning that the chart executes the action only once.

Order of Execution of Combined Actions

States execute combined actions in the same order as they execute separate actions:

1. `Entry` actions first, from top to bottom in the order they appear in the state
2. `During` actions second, from top to bottom
3. `Exit` actions last, from top to bottom

The order in which you combine actions does not affect state execution behavior. For example:

Combined Actions	Order of Execution

A
en: y = 0;
en, du: y = y+1;
	1 `en: y = 0;`
	2 `en: y = y+1;`
	3 `du: y = y+1;`

B
en, du: y = y+1;
en: y = 0;
	1 `en: y = y+1;`
	2 `en: y = 0;`
	3 `du: y = y+1;`

C
du, en: y = y+1;
en: y = 0;
	1 `en: y = y+1;`
	2 `en: y = 0;`
	3 `du: y = y+1;`

D
du, en: y = y+1;
en, ex: y = 10;
	1 `en: y = y+1;`
	2 `en: y = 10;`
	3 `du: y = y+1;`
	4 `ex: y = 10;`

Rules for Combining State Actions

- Do not combine multiple actions of the same type.
- Do not create data, events, or messages that have the same name as the action keywords: `entry`, `en`, `during`, `du`, `exit`, `ex`.

See Also

More About

- “Represent Operating Modes by Using States” on page 1-26

Operations for Stateflow Data

Stateflow charts in Simulink models have an action language property that defines the operations that you can use in state and transition actions. The language properties are:

- MATLAB as the action language.
- C as the action language.

For more information, see “Differences Between MATLAB and C as Action Language Syntax” on page 15-4.

Binary Operations

This table summarizes the interpretation of all binary operations in Stateflow charts according to their order of precedence (0 = highest, 10 = lowest). Binary operations are left associative so that, in any expression, operators with the same precedence are evaluated from left to right. The order of evaluation for other operations is unspecified. For example, in this assignment

```
A = f() > g();
```

the order of evaluation of `f()` and `g()` is unspecified. For more predictable results, it is good coding practice to split expressions that depend on the order of evaluation into multiple statements.

Operation	Precedence	MATLAB as the Action Language	C as the Action Language
`a ^ b`	0	Power.	Power. This operation is equivalent to the C library function `pow`. Operands are first cast to floating-point numbers. For more information, see “Call C Library Functions” on page 14-16. Enable this operation by clearing the **Enable C-bit operations** chart property. For more information, see “Enable C-bit operations” on page 1-21.
`a * b`	1	Multiplication.	Multiplication.
`a / b`	1	Division.	Division.
`a %% b`	1	Not supported. Use the `rem` or `mod` function.	Remainder. Noninteger operands are first cast to integers.
`a + b`	2	Addition.	Addition.
`a - b`	2	Subtraction.	Subtraction.
`a >> b`	3	Not supported. Use the `bitshift` function.	Shift `a` to the right by `b` bits. For more information, see “Bitwise Operations” on page 14-8.

<table>
<tr><th>Operation</th><th>Precedence</th><th>MATLAB as the Action Language</th><th>C as the Action Language</th></tr>
<tr><td><code>a << b</code></td><td>3</td><td>Not supported. Use the <code>bitshift</code> function.</td><td>Shift <code>a</code> to the left by <code>b</code> bits. For more information, see “Bitwise Operations” on page 14-8.</td></tr>
<tr><td><code>a > b</code></td><td>4</td><td>Comparison, greater than.</td><td>Comparison, greater than.</td></tr>
<tr><td><code>a < b</code></td><td>4</td><td>Comparison, less than.</td><td>Comparison, less than.</td></tr>
<tr><td><code>a >= b</code></td><td>4</td><td>Comparison, greater than or equal to.</td><td>Comparison, greater than or equal to.</td></tr>
<tr><td><code>a <= b</code></td><td>4</td><td>Comparison, less than or equal to.</td><td>Comparison, less than or equal to.</td></tr>
<tr><td><code>a == b</code></td><td>5</td><td>Comparison, equal to.</td><td>Comparison, equal to.</td></tr>
<tr><td><code>a ~= b</code></td><td>5</td><td>Comparison, not equal to.</td><td>Comparison, not equal to.</td></tr>
<tr><td><code>a != b</code></td><td>5</td><td>Not supported. Use the operation <code>a ~= b</code>.</td><td>Comparison, not equal to.</td></tr>
<tr><td><code>a <> b</code></td><td>5</td><td>Not supported. Use the operation <code>a ~= b</code>.</td><td>Comparison, not equal to.</td></tr>
<tr><td><code>a & b</code></td><td>6</td><td>Logical AND. For bitwise AND, use the <code>bitand</code> function.</td><td>• Bitwise AND (default). Enable this operation by selecting the Enable C-bit operations chart property.
• Logical AND. Enable this operation by clearing the Enable C-bit operations chart property.

For more information, see “Bitwise Operations” on page 14-8 and “Enable C-bit operations” on page 1-21.</td></tr>
<tr><td><code>a ^ b</code></td><td>7</td><td>Not supported. For bitwise XOR, use the <code>bitxor</code> function.</td><td>Bitwise XOR (default). Enable this operation by selecting the Enable C-bit operations chart property. For more information, see “Bitwise Operations” on page 14-8 and “Enable C-bit operations” on page 1-21.</td></tr>
<tr><td><code>a | b</code></td><td>8</td><td>Logical OR. For bitwise OR, use the <code>bitor</code> function.</td><td>• Bitwise OR (default). Enable this operation by selecting the Enable C-bit operations chart property.
• Logical OR. Enable this operation by clearing the Enable C-bit operations chart property.

For more information, see “Bitwise Operations” on page 14-8 and “Enable C-bit operations” on page 1-21.</td></tr>
<tr><td><code>a && b</code></td><td>9</td><td>Logical AND.</td><td>Logical AND.</td></tr>
<tr><td><code>a || b</code></td><td>10</td><td>Logical OR.</td><td>Logical OR.</td></tr>
</table>

Unary Operations and Actions

This table summarizes the interpretation of all unary operations and actions in Stateflow charts. Unary operations:

- Have higher precedence than the binary operators.
- Are right associative so that, in any expression, they are evaluated from right to left.

Operation	MATLAB as the Action Language	C as the Action Language
`~a`	Logical NOT. For bitwise NOT, use the `bitcmp` function.	• Bitwise NOT (default). Enable this operation by selecting the **Enable C-bit operations** chart property. • Logical NOT. Enable this operation by clearing the **Enable C-bit operations** chart property. For more information, see "Bitwise Operations" on page 14-8 and "Enable C-bit operations" on page 1-21.
`!a`	Not supported. Use the operation `~a`.	Logical NOT.
`-a`	Negative.	Negative.
`a++`	Not supported. Use the expression `a = a +1`.	Increment. Equivalent to `a = a+1`.
`a--`	Not supported. Use the expression `a = a-1`.	Decrement. Equivalent to `a = a-1`.

Assignment Operations

This table summarizes the interpretation of assignment operations in Stateflow charts.

Operation	MATLAB as the Action Language	C as the Action Language
`a = b`	Simple assignment.	Simple assignment.
`a := b`	Not supported. Use type cast operations to override fixed-point promotion rules. See "Type Cast Operations" on page 14-7.	Assignment of fixed-point numbers. See "Override Fixed-Point Promotion in C Charts" on page 23-11.
`a += b`	Not supported. Use the expression `a = a +b`.	Equivalent to `a = a+b`.
`a -= b`	Not supported. Use the expression `a = a-b`.	Equivalent to `a = a-b`.
`a *= b`	Not supported. Use the expression `a = a*b`.	Equivalent to `a = a*b`.
`a /= b`	Not supported. Use the expression `a = a/b`.	Equivalent to `a = a/b`.
`a %%= b`	Not supported. Use the expression `a = mod(a,b)` or `a = rem(a,b)`.	Equivalent to `a = a%%b`.

Operation	MATLAB as the Action Language	C as the Action Language
a &= b	Not supported. Use the expression a = bitand(a,b).	Equivalent to a = a&b (bitwise AND). Enable this operation by selecting the **Enable C-bit operations** chart property. For more information, see "Bitwise Operations" on page 14-8 and "Enable C-bit operations" on page 1-21.
a ^= b	Not supported. Use the expression a = bitxor(a,b).	Equivalent to a = a^b (bitwise XOR). Enable this operation by selecting the **Enable C-bit operations** chart property. For more information, see "Bitwise Operations" on page 14-8 and "Enable C-bit operations" on page 1-21.
a |= b	Not supported. Use the expression a = bitor(a,b).	Equivalent to a = a|b (bitwise OR). Enable this operation by selecting the **Enable C-bit operations** chart property. For more information, see "Bitwise Operations" on page 14-8 and "Enable C-bit operations" on page 1-21.

Type Cast Operations

To convert a value of one type to another type, use type cast operations. You can cast data to an explicit type or to the type of another variable.

Cast to Explicit Data Type

To cast a numeric expression to an explicit data type, use one of these type conversion functions: double, single, int8, int16, int32, int64, uint8, uint16, uint32, uint64, and boolean. For example, this statement casts the expression x+3 to a 16-bit unsigned integer and assigns the value to the data y:

```
y = uint16(x+3);
```

Alternatively, in charts that use MATLAB as the action language, you can use the cast function and specify "double", "single", "int8", "int16", "int32", "int64", "uint8", "uint16", "uint32", "uint64", or "logical" as an input argument. For example, this statement casts the expression x+3 to a 16-bit unsigned integer and assigns the value to y:

```
y = cast(x+3,"uint16");
```

To cast an expression to a fixed-point type, charts that use MATLAB as the action language support calling the fi function. For example, this statement casts the expression x+3 as a signed fixed-point value with a word length of eight bits and a fraction length of three bits:

```
y = fi(x+3,1,8,3);
```

In charts that use C as the action language, call the cast function using a fixdt expression as an argument. For example, this statement casts the expression x+3 as a signed fixed-point value with a word length of eight bits and a fraction length of three bits:

```
y = cast(x+3,fixdt(1,8,3));
```

Cast Type Based on Other Data

To make type casting easier, you can convert the type of a numeric expression to the same type as another Stateflow data.

In charts that use MATLAB as the action language, call the `cast` function with the keyword `"like"`. For example, this statement converts the value of `x+3` to the same type as that of data `z` and assigns the value to `y`:

```
y = cast(x+3,"like",z);
```

In charts that use C as the action language, the `type` operator returns the type of an existing Stateflow data. Use this return value in place of an explicit type in a `cast` operation. For example, this statement converts the value of `x+3` to the same type as that of data `z` and assigns the value to `y`:

```
y = cast(x+3,type(z));
```

Bitwise Operations

This table summarizes the interpretation of all bitwise operations in Stateflow charts that use C as the action language.

Operation	Description
`a & b`	Bitwise AND.
`a | b`	Bitwise OR.
`a ^ b`	Bitwise XOR.
`~a`	Bitwise NOT.
`a >> b`	Shift `a` to the right by `b` bits.
`a << b`	Shift `a` to the left by `b` bits.

Except for the bit shift operations `a >> b` and `a << b`, you must enable all bitwise operations by selecting the **Enable C-bit operations** chart property. See “Enable C-bit operations” on page 1-21.

Bitwise operations work on integers at the binary level. Noninteger operands are first cast to integers. Integer operands follow C promotion rules to determine the intermediate value of the result. This intermediate value is then cast to the type that you specify for the result of the operation.

Note Bitwise operations are not supported in charts that use MATLAB as the action language. Instead, use the functions `bitand`, `bitor`, `bitxor`, `bitnot`, or `bitshift`.

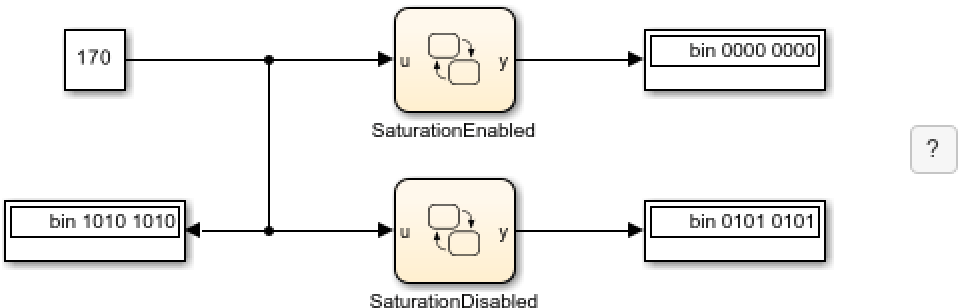
Bitwise Operations and Integer Overflows

The implicit cast used to assign the intermediate value of a bitwise operation can result in an overflow. To preserve the rightmost bits of the result and avoid unexpected behavior, disable the chart property **Saturate on Integer Overflows**.

For example, both charts in this model compute the bitwise operation `y = ~u`. The charts compute the intermediate value for this operation by using the target integer size of 32 bits, so the 24 leftmost bits in this value are all ones. When the charts assign the intermediate value to `y`, the cast to `uint8`

causes an integer overflow. The output from each chart depends on how the chart handles integer overflows.

- If **Saturate on Integer Overflow** is enabled, the chart saturates the result of the bitwise operation and outputs a value of zero.
- If **Saturate on Integer Overflow** is disabled, the chart wraps the result of the bitwise operation and outputs its eight rightmost bits.



For more information, see “Saturate on integer overflow” on page 1-22.

Pointer and Address Operations

This table summarizes the interpretation of pointer and address operations in Stateflow charts that use C as the action language.

Operation	Description
`&a`	Address operation. Use with custom code and Stateflow variables.
`*a`	Pointer operation. Use only with custom code variables.

For example, the model `sf_bus_demo` contains a custom C function that takes pointers as arguments. When the chart calls the custom code function, it uses the & operation to pass the Stateflow data by address. For more information, see “Integrate Custom Structures in Stateflow Charts” on page 26-11.

Pointer and address operations are not supported in charts that use MATLAB as the action language. Pointers to structures should only be used in read-only mode and are only valid during the call in which they are passed.

Replace Operations with Application Implementations

If you have Embedded Coder or Simulink Coder, you can configure the code generator to apply a code replacement library (CRL) during code generation. The code generator changes the code that it generates for operations to meet application requirements. With Embedded Coder, you can develop and apply custom code replacement libraries.

Operation entries of the code replacement library can specify integral or fixed-point operand and result patterns. You can use operation entries for these operations:

- Addition +

- Subtraction `-`
- Multiplication `*`
- Division `/`

For example, in this expression, you can replace the addition operator `+` with a target-specific implementation if `u1`, `u2`, and `y` have types that permit a match with an addition entry in the code replacement library:

```
y = u1 + u2
```

C chart semantics limit operator entry matching because the chart uses the target integer size as its intermediate type in arithmetic expressions. For example, this arithmetic expression computes the intermediate addition into a target integer:

```
y = (u1 + u2) % 3
```

If the target integer size is 32 bits, then you cannot replace this expression with an addition operator from the code replacement library and produce a signed 16-bit result without a loss of precision.

For more information about using code replacement libraries that MathWorks® provides, see “What Is Code Replacement?” (Simulink Coder) and “Code Replacement Libraries” (Simulink Coder). For information about developing custom code replacement libraries, see “What Is Code Replacement Customization?” (Embedded Coder) and “Code You Can Replace From Simulink Models” (Embedded Coder).

See Also

More About

- “Differences Between MATLAB and C as Action Language Syntax” on page 15-4
- “Specify Properties for Stateflow Charts” on page 1-19
- “Operations for Vectors and Matrices in Stateflow” on page 19-4
- “Operations for Complex Data in Stateflow” on page 24-4
- “Operations for Fixed-Point Data in Stateflow” on page 23-8

Supported Symbols in Actions

Boolean Symbols, true and false

Use the symbols `true` and `false` to represent Boolean constants. You can use these symbols as scalars in expressions. Examples include:

```
cooling_fan = true;
heating_fan = false;
```

Tip These symbols are case-sensitive. Therefore, `TRUE` and `FALSE` are not Boolean symbols.

Do not use `true` and `false` in the following cases. Otherwise, error messages appear.

- Left-hand side of assignment statements
  - `true++;`
  - `false += 3;`
  - `[true, false] = my_function(x);`
- Argument of the `change` implicit event (see “Control Chart Behavior by Using Implicit Events” on page 12-28)
  - `change(true);`
  - `chg(false);`
- Indexing into a vector or matrix (see “Operations for Vectors and Matrices in Stateflow” on page 19-4)
  - `x = true[1];`
  - `y = false[1][1];`

Note If you define `true` and `false` as Stateflow data objects, your custom definitions of `true` and `false` override the built-in Boolean constants.

Comment Symbols, %, //, /*

Use the symbols `%`, `//`, and `/*` to represent comments as shown in these examples:

```
% MATLAB comment line
// C++ comment line
/* C comment line */
```

You can also include comments in generated code for an embedded target (see “Model Configuration Parameters: Comments” (Simulink Coder). C chart comments in generated code use multibyte character code. Therefore, you can have code comments with characters for non-English alphabets, such as Japanese Kanji characters.

Hexadecimal Notation Symbols, 0xFF

C charts support C style hexadecimal notation, for example, `0xFF`. You can use hexadecimal values wherever you can use decimal values.

Infinity Symbol, inf

Use the MATLAB symbol `inf` to represent infinity in C charts. Calculations like `n/0`, where `n` is any nonzero real value, result in `inf`.

Note If you define `inf` as a Stateflow data object, your custom definition of `inf` overrides the built-in value.

Line Continuation Symbol, ...

Use the characters `...` at the end of a line to indicate that the expression continues on the next line. For example, you can use the line continuation symbol in a state action:

```
entry: total1 = 0, total2 = 0, ...
  total3 = 0;
```

MATLAB Display Symbol, ;

Omitting the semicolon after an expression displays the results of the expression in the Diagnostic Viewer. If you use a semicolon, the results do not appear.

Single-Precision Floating-Point Number Symbol, F

Use a trailing `F` to specify single-precision floating-point numbers in C charts. For example, you can use the action statement `x = 4.56F;` to specify a single-precision constant with the value 4.56. If a trailing `F` does not appear with a number, double precision applies.

Time Symbol, t

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.

Call Extrinsic MATLAB Functions in Stateflow Charts

Stateflow charts in Simulink models have an action language property that defines the syntax for state and transition actions. An icon in the lower-left corner of the chart canvas indicates the action language for the chart.

- MATLAB as the action language.
- C as the action language.

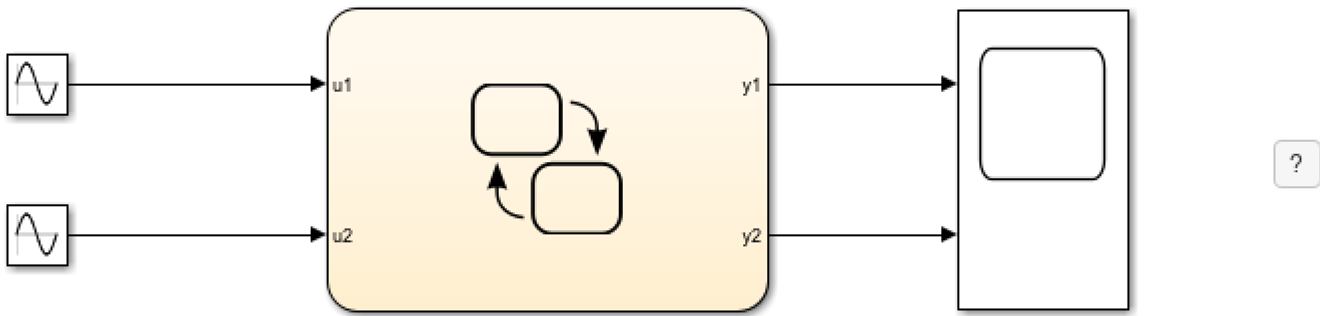
In charts that use C as the action language, you can call built-in MATLAB functions and access MATLAB workspace variables by using the `ml` namespace operator or the `ml` function. For more information, see “Access MATLAB Functions and Workspace Data in C Charts” on page 14-20.

In charts that use MATLAB as the action language, you can call MATLAB functions supported for code generation directly. To call extrinsic functions that are not supported for code generation, you must use the `coder.extrinsic` function. When you declare a function with `coder.extrinisic(`*`function_name`*`)`, Stateflow creates a call to the function during simulation. In a Stateflow chart, you only declare `coder.extrinsic` once. You cannot declare reserved keywords with `coder.extrinsic`. For more information, see “Guidelines for Naming Stateflow Objects” on page 1-66.

For charts that include atomic subcharts, you must declare functions that are not supported for code generation with `coder.extrinsic` separately within the atomic subchart.

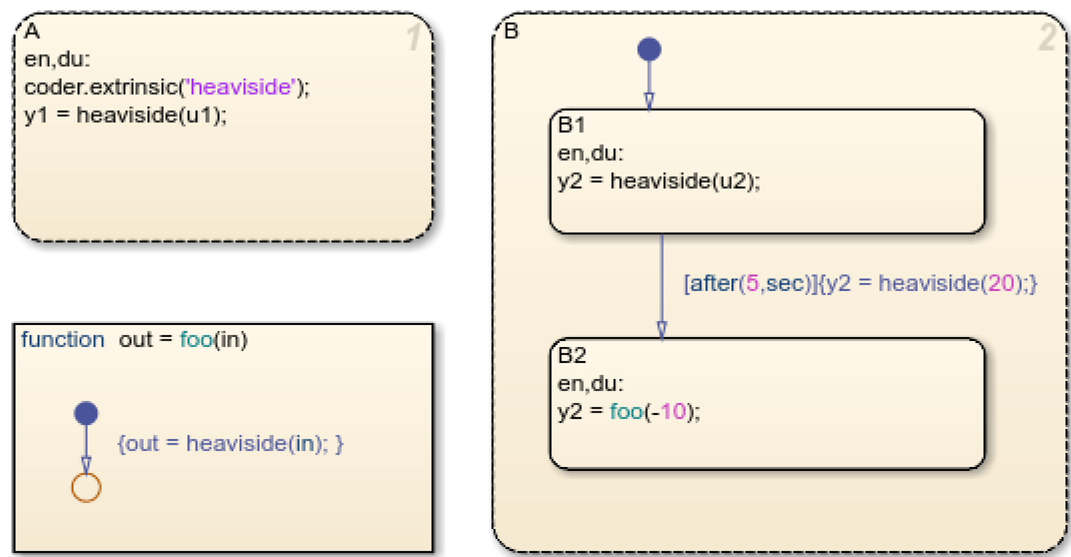
Use the coder.extrinsic Function

To enable calls to the extrinsic function `heaviside (Symbolic Math Toolbox)`, this model uses `coder.extrinsic`.



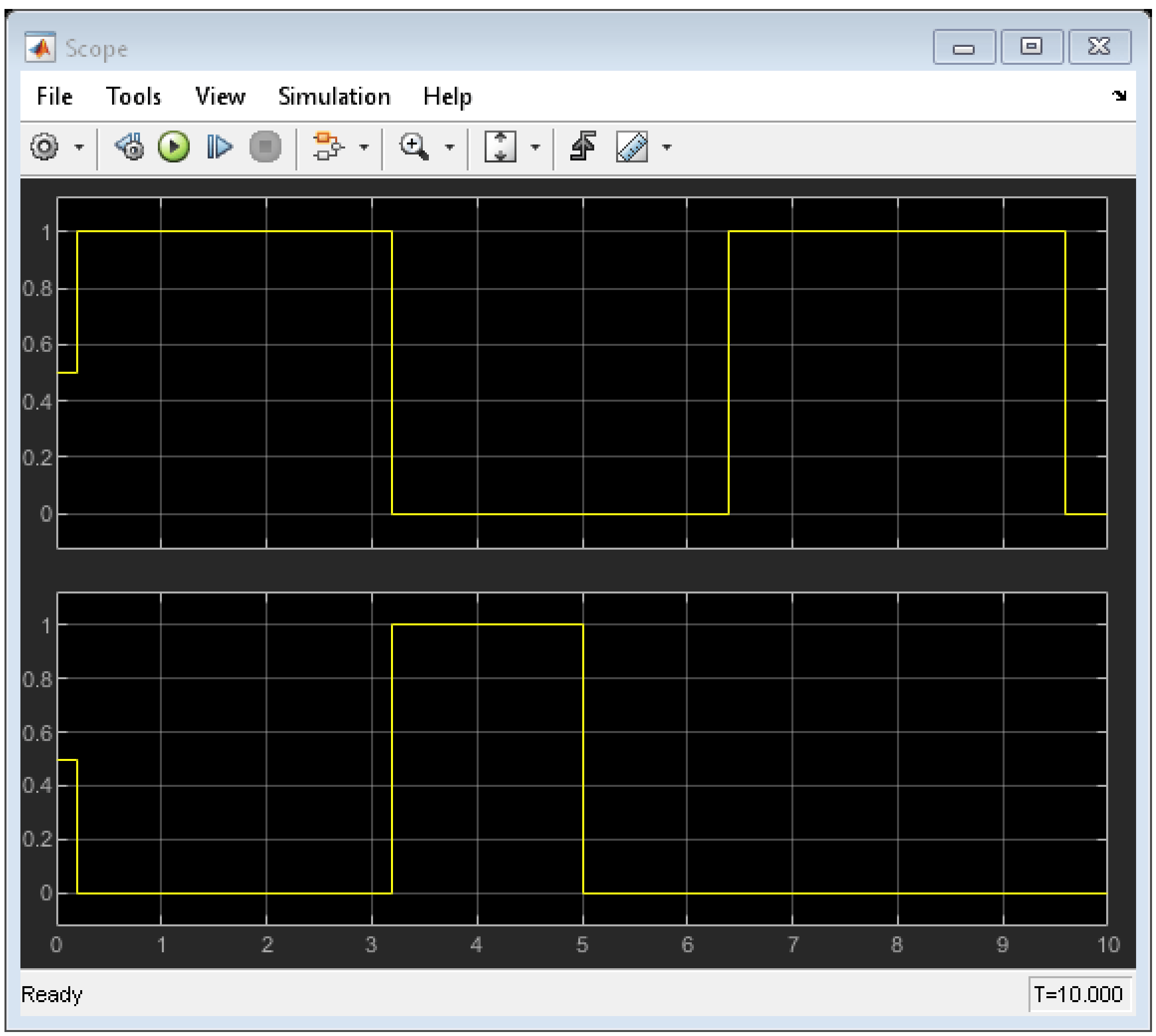
The chart contains two parallel states, A and B, and one graphical function block, `foo`. State A declares the function `heaviside`, which is not supported for code generation, by using `coder.extrinsic`. State B and the graphical function block also use `heaviside` without `coder.extrinsic`.

The input for state A is `u1`, a sine wave, and the input for state B is `u2`, a cosine wave. The graphical function `out` outputs the value of the `heaviside` function for the input `in`.



You only need to declare `heaviside` once in your chart using `coder.extrinsic`. After this you can use the `heaviside` function anywhere within your chart without `coder.extrinsic`. When generating code the functions that you declare using `coder.extrinsic` will have a call to the extrinsic function, and that function will not appear in the generated code.

To visualize the result of this chart, open the scope.



See Also

`heaviside` | `coder.extrinsic`

More About

- "Access MATLAB Functions and Workspace Data in C Charts" on page 14-20
- "Generate Code for Global Data" (MATLAB Coder)
- "Functions and Objects Supported for C/C++ Code Generation" (Simulink)

Call C Library Functions in C Charts

Stateflow charts in Simulink models have an action language property that defines the syntax for state and transition actions. An icon in the lower-left corner of the chart canvas indicates the action language for the chart.

- MATLAB as the action language.
- C as the action language.

Call C Library Functions

You can call this subset of the C Math Library functions:

abs* **	acos**	asin**	atan**	atan2**	ceil**
cos**	cosh**	exp**	fabs	floor**	fmod**
labs	ldexp**	log**	log10**	pow**	rand
sin**	sinh**	sqrt**	tan**	tanh**	
* The Stateflow abs function goes beyond that of its standard C counterpart with its own built-in functionality. For more information, see “Call the abs Function” on page 14-16.					
** You can also replace calls to the C Math Library with application-specific implementations for this subset of functions. For more information, see “Replacement of Math Library Functions with Application Implementations” on page 14-17.					

When you call these functions, double precision applies unless all the input arguments are explicitly single precision. When a type mismatch occurs, a cast of the input arguments to the expected type replace the original arguments. For example, if you call the `sin` function with an integer argument, a cast of the input argument to a floating-point number of type `double` replaces the original argument.

Note Because the input arguments to the C library functions are first cast to floating-point numbers, function calls with arguments of type `int64` or `uint64` can result in loss of precision.

If you call other C library functions not listed above, open the Configuration Parameters dialog box and, in the **Simulation Target** pane, enter the appropriate `#include` statements, as described in “Configure Custom Code for Your Model” on page 28-4.

Call the abs Function

Interpretation of the Stateflow `abs` function goes beyond the standard C version to include integer and floating-point arguments of all types as follows:

- If `x` is an integer of type `int32` or `int64`, the standard C function `abs` applies to `x`, or `abs(x)`.
- If `x` is an integer of type `int16` or `int8`, the standard C `abs` function applies to a cast of `x` as an integer of type `int32`, or `abs((int32)x)`.
- If `x` is a floating-point number of type `double`, the standard C function `fabs` applies to `x`, or `fabs(x)`.

- If x is a floating-point number of type single, the standard C function fabs applies to a cast of x as a double, or fabs((double)x).
- If x is a fixed-point number, the standard C function fabs applies to a cast of the fixed-point number as a double, or fabs((double) V_x), where V_x is the real-world value of x.

If you want to use the abs function in the strict sense of standard C, cast its argument or return values to integer types. See “Type Cast Operations” on page 14-7.

Note If you declare x in custom code, the standard C abs function applies in all cases. For instructions on inserting custom code into charts, see “Reuse Custom Code in Stateflow Charts” on page 28-2.

Call min and max Functions

You can call min and max by emitting the following macros automatically at the top of generated code.

```
#define min(x1,x2) ((x1) > (x2) ? (x2):(x1))
#define max(x1,x2) ((x1) > (x2) ? (x1):(x2))
```

To allow compatibility with user graphical functions named min() or max(), generated code uses a mangled name of the following form: <prefix>_min. However, if you export min() or max() graphical functions to other charts in your model, the name of these functions can no longer be emitted with mangled names in generated code and conflict occurs. To avoid this conflict, rename the min() and max() graphical functions.

Replacement of Math Library Functions with Application Implementations

You can configure the code generator to change the code that it generates for math library functions such that the code meets application requirements. To do this you configure the code generator to apply a code replacement library (CRL) during code generation. If you have an Embedded Coder license, you can develop and apply custom code replacement libraries.

For more information about replacing code, using code replacement libraries that MathWorks provides, see “What Is Code Replacement?” (Simulink Coder) and “Code Replacement Libraries” (Simulink Coder). For information about developing custom code replacement libraries, see “What Is Code Replacement Customization?” (Embedded Coder) and “Code You Can Replace From Simulink Models” (Embedded Coder).

Call Custom C Code Functions

You can specify custom code functions for use in C charts for simulation and C code generation. For more information, see “Configure Custom Code for Your Model” on page 28-4.

Guidelines for Calling Custom C Functions in Your Chart

- Define a function by its name, any arguments in parentheses, and an optional semicolon.
- Pass parameters to user-written functions using single quotation marks. For example, func('string').

- An action can nest function calls.
- An action can invoke functions that return a scalar value (of type `double` in the case of MATLAB functions and of any type in the case of C user-written functions).

Guidelines for Writing Custom C Functions That Access Input Vectors

- Use the `sizeof` function to determine the length of an input vector.

  For example, your custom function can include a for-loop that uses `sizeof` as follows:

  ```
  for(i=0; i < sizeof(input); i++) {
  ......
  }
  ```
- If your custom function uses the value of the input vector length multiple times, include an input to your function that specifies the input vector length.

  For example, you can use `input_length` as the second input to a `sum` function as follows:

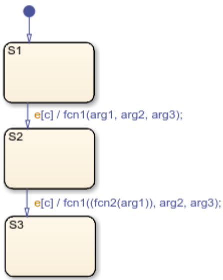
  ```
  int sum(double *input, double input_length)
  ```

  Your `sum` function can include a for-loop that iterates over all elements of the input vector:

  ```
  for(i=0; i < input_length; i++) {
  ......
  }
  ```

Function Call in Transition Action

Example formats of function calls using transition action notation appear in the following chart.



A function call to `fcn1` occurs with `arg1`, `arg2`, and `arg3` if the following are true:

- `S1` is active.
- Event `e` occurs.
- Condition `c` is true.
- The transition destination `S2` is valid.

The transition action in the transition from `S2` to `S3` shows a function call nested within another function call.

Function Call in State Action

Example formats of function calls using state action notation appear in the following chart.

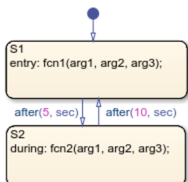


Chart execution occurs as follows:

1. When the default transition into `S1` occurs, `S1` becomes active.
2. The `entry` action, a function call to `fcn1` with the specified arguments, executes.
3. After 5 seconds of simulation time, `S1` becomes inactive and `S2` becomes active.
4. The `during` action, a function call to `fcn2` with the specified arguments, executes.
5. After 10 seconds of simulation time, `S2` becomes inactive and `S1` becomes active again.
6. Steps 2 through 5 repeat until the simulation ends.

Pass Arguments by Reference

A Stateflow action can pass arguments to a user-written function by reference rather than by value. In particular, an action can pass a pointer to a value rather than the value itself. For example, an action could contain the following call:

```
f(&x);
```

where `f` is a custom-code C function that expects a pointer to `x` as an argument.

If `x` is the name of a data item defined in the Stateflow hierarchy, the following rules apply:

- Do not use pointers to pass data items input from a Simulink model.

  If you need to pass an input item by reference, for example, an array, assign the item to a local data item and pass the local item by reference.
- If the data type of `x` is `boolean`, you must turn off the coder option **Use bitsets for storing state configuration**.
- If `x` is an array with its first index property set to 0 (see “Set Data Properties” on page 10-5), then you must call the function as follows.

  ```
  f(&(x[0]));
  ```

  This passes a pointer to the first element of `x` to the function.
- If `x` is an array with its first index property set to a nonzero number (for example, 1), the function must be called in the following way:

  ```
  f(&(x[1]));
  ```

  This passes a pointer to the first element of `x` to the function.

Access MATLAB Functions and Workspace Data in C Charts

Stateflow charts in Simulink models have an action language property that defines the syntax for state and transition actions. An icon in the lower-left corner of the chart canvas indicates the action language for the chart.

- MATLAB as the action language.
- C as the action language.

In charts that use C as the action language, you can call built-in MATLAB functions and access MATLAB workspace variables by using the `ml` namespace operator or the `ml` function.

Caution Because MATLAB functions are not available in a target environment, do not use the `ml` namespace operator and the `ml` function if you plan to build a code generation target.

---

ml Namespace Operator

For C charts, the `ml` namespace operator uses standard dot `(.)` notation to reference MATLAB variables and functions. For example, the statement `a = ml.x` returns the value of the MATLAB workspace variable `x` to the Stateflow data `a`.

For functions, the syntax is as follows:

```
[return_val1,return_val2,...] = ml.function_name(arg1,arg2,...)
```

For example, the statement `[a, b, c] = ml.`*`function`*`(x, y)` passes the return values from the MATLAB function *`function`* 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.f(ml.x[1][3],ml.y[3])`

  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 value returned by the call to `f` is assigned to element `(5,6,7)` of the workspace array, `v`. If `v` does not exist prior to simulation, it is automatically created in the MATLAB workspace.

ml Function

For C charts, you can use the `ml` function to specify calls to MATLAB functions. The format for the `ml` function call uses this notation:

```
ml(evalString,arg1,arg2,...);
```

`evalString` is an expression that is evaluated in the MATLAB workspace. It contains a MATLAB command (or a set of commands, each separated by a semicolon) to execute along with format specifiers (`%g`, `%f`, `%d`, etc.) that provide formatted substitution of the other arguments (*`arg1`*, *`arg2`*, etc.) into *`evalString`*.

The format specifiers used in `ml` functions are the same as those used in the C functions `printf` and `sprintf`. The `ml` function call is equivalent to calling the MATLAB `eval` function with the `ml` namespace operator if the arguments *`arg1, arg2,...`* are restricted to scalars or literals in the following command:

```
ml.eval(ml.sprintf(evalString,arg1,arg2,...))
```

Format specifiers used in the `ml` function must either match the data types of the arguments or the arguments must be of types that can be promoted to the type represented by the format specifier.

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 14-25.

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 object `a`. Because `x` is a workspace variable, and `sin(x)` is evaluated in the MATLAB workspace, you enter it directly as the string `"sin(x)"`.
- `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 object `sf_mat44`. you must define this data object as a 4-by-4 array before simulation. Otherwise, a size mismatch error occurs during run-time.
- `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 object `a`. Because `d1` is Stateflow data, its value is inserted in

the string argument "sin(%f)" using the format expression %f. If d1 = 1.5, the expression evaluated in the MATLAB workspace is sin(1.5).

- a = ml("f(%g,x,%f)",d1,d2)

  In this example, the expression is the *evalString* shown in the preceding format statement. Stateflow data d1 and d2 are inserted into the expression "f(%g,x,%f)" by using the format specifiers %g and %f, respectively.

ml Expressions

For C charts, you can mix ml namespace operator and ml function expressions along with Stateflow data in larger expressions. The following example squares the sine and cosine of an angle in workspace variable X and adds them:

```
a = ml.power(ml.sin(ml.X),2) + ml("power(cos(X),2)")
```

The first operand uses the ml namespace operator to call the sin function. Its argument is ml.X, since X is in the MATLAB workspace. The second operand uses the ml function. Because X is in the workspace, it appears in the *evalString* expression as X. The squaring of each operand is performed with the MATLAB power function, which takes two arguments: the value to square, and the power value, 2.

Expressions using the ml namespace operator and the ml function can be used as arguments for ml namespace operator and ml function expressions. The following example nests ml expressions at three different levels:

```
a = ml.power(ml.sin(ml.X + ml("cos(Y)")),2)
```

In composing your ml expressions, follow the levels of precedence set out in “Binary Operations” on page 14-4. Use parentheses around power expressions with the ^ operator when you use them in conjunction with other arithmetic operators.

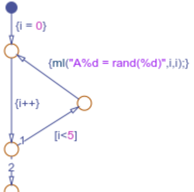
Stateflow software checks expressions for data size mismatches in your actions when you update or simulate the model. Because the return values for ml expressions are not known until run time, Stateflow software must infer the size of their return values. See “How Charts Infer the Return Size for ml Expressions” on page 14-25.

Which ml Should I Use?

In most cases, the notation of the ml namespace operator is more straightforward. However, using the ml function call does offer a few advantages:

- Use the ml function to dynamically construct workspace variables.

  The following flow chart creates four new MATLAB matrices:



The `for` loop creates four new matrix variables in the MATLAB workspace. The default transition initializes the Stateflow counter `i` to 0, while the transition segment between the top two junctions increments it by 1. If `i` is less than 5, the transition segment back to the top junction evaluates the `ml` function call `ml("A%d = rand(%d)",i,i)` for the current value of `i`. When `i` is greater than or equal to 5, the transition segment between the bottom two junctions occurs and execution stops.

The transition executes the following MATLAB commands, which create a workspace scalar (`A1`) and three matrices (`A2`, `A3`, `A4`):

```
A1 = rand(1)
A2 = rand(2)
A3 = rand(3)
A4 = rand(4)
```

- Use the `ml` function with full MATLAB notation.

  You cannot use full MATLAB notation with the `ml` namespace operator, as the following example shows:

```
ml.A = ml.magic(4);
B = ml("A + A'");
```

  This example sets the workspace variable `A` to a magic 4-by-4 matrix using the `ml` namespace operator. Stateflow data `B` is then set to the addition of `A` and its transpose matrix, `A'`, which produces a symmetric matrix. Because the `ml` namespace operator cannot evaluate the expression `A'`, the `ml` function is used instead. However, you can call the MATLAB function `transpose` with the `ml` namespace operator in the following equivalent expression:

```
ml.A = ml.magic(4);
B = ml.A + ml.transpose(ml.A)
```

  As another example, you cannot use arguments with cell arrays or subscript expressions involving colons with the `ml` namespace operator. However, these can be included in an `ml` function call.

ml Data Type

Stateflow data of type `ml` is typed internally with the MATLAB type `mxArray` for C charts. You can assign (store) any type of data available in the Stateflow hierarchy to a data of type `ml`. These types include any data type defined in the Stateflow hierarchy or returned from the MATLAB workspace with the `ml` namespace operator or `ml` function.

Rules for Using ml Data Type

These rules apply to Stateflow data of type `ml`:

- You can initialize `ml` data from the MATLAB workspace just like other data in the Stateflow hierarchy (see “Initialize Data from the MATLAB Base Workspace” on page 10-30).
- 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 14-25.

- You cannot define `ml` data with the scope **Constant**.

  This option is disabled in the Data properties dialog box and in the Model Explorer for Stateflow data of type `ml`.
- You can use `ml` data to build a simulation target but not to build an embeddable code generation target.
- If data of type `ml` contains an array, you can access the elements of the array via indexing with these rules:

  a You can index only arrays with numerical elements.

  b You can index numerical arrays only by their dimension.

  In other words, you can access only one-dimensional arrays by a single index value. You cannot access a multidimensional array with a single index value.

  c The first index value for each dimension of an array is 1, and not 0, as in C language arrays.

  In the examples that follow, `mldata` is a Stateflow data of type `ml`, `ws_num_array` is a 2-by-2 MATLAB workspace array with numerical values, and `ws_str_array` is a 2-by-2 MATLAB workspace array with character vector values.

  ```
  mldata = ml.ws_num_array; /* OK */
  n21 = mldata[2][1]; /* OK for numerical data of type ml */
  n21 = mldata[3]; /* NOT OK for 2-by-2 array data */
  mldata = ml.ws_str_array; /* OK */
  s21 = mldata[2][1]; /* NOT OK for character vector data of type ml*/
  ```
- `ml` data cannot have a scope outside a C chart; that is, you cannot define the scope of `ml` data as **Input from Simulink** or **Output to Simulink**.

Place Holder for Workspace Data

Both the `ml` namespace operator and the `ml` function can access data directly in the MATLAB workspace and return it to a C chart. However, maintaining data in the MATLAB workspace can present Stateflow users with conflicts with other data already resident in the workspace. Consequently, with the `ml` data type, you can maintain `ml` data in a chart and use it for MATLAB computations in C charts.

As an example, in the following statements, `mldata1` and `mldata2` are Stateflow data of type `ml`:

```
mldata1 = ml.rand(3);
mldata2 = ml.transpose(mldata1);
```

In the first line of this example, `mldata1` receives the return value of the MATLAB function `rand`, which, in this case, returns a 3-by-3 array of random numbers. Note that `mldata1` is not specified as an array or sized in any way. It can receive any MATLAB workspace data or the return of any MATLAB function because it is defined as a Stateflow data of type `ml`.

In the second line of the example, `mldata2`, also of Stateflow data type `ml`, receives the transpose matrix of the matrix in `mldata1`. It is assigned the return value of the MATLAB function `transpose` in which `mldata1` is the argument.

Note the differences in notation if the preceding example were to use MATLAB workspace data (`wsdata1` and `wsdata2`) instead of Stateflow `ml` data to hold the generated matrices:

```
ml.wsdata1 = ml.rand(3);
ml.wsdata2 = ml.transpose(ml.wsdata1);
```

In this case, each workspace data must be accessed through the `ml` namespace operator.

How Charts Infer the Return Size for ml Expressions

In C charts, Stateflow expressions using the `ml` namespace operator and the `ml` function evaluate in the MATLAB workspace at run time. The actual size of the data returned from the following expression types is known only at run time:

- MATLAB workspace data or functions using the `ml` namespace operator or the `ml` function call

  For example, the size of the return values from the expressions `ml.`*`var`*, `ml.`*`func`*`()`, or `ml(`*`evalString, arg1, arg2,...`*`)`, where *`var`* is a MATLAB workspace variable and *`func`* is a MATLAB function, cannot be known until run-time.
- Stateflow data of type `ml`
- Graphical functions that return Stateflow data of type `ml`

When these expressions appear in actions, Stateflow code generation creates temporary data to hold intermediate returns for evaluation of the full expression of which they are a part. Because the size of these return values is unknown until run time, Stateflow software must employ context rules to infer the sizes for creation of the temporary data.

During run time, if the actual returned value from one of these commands differs from the inferred size of the temporary variable that stores it, a size mismatch error appears. To prevent run-time errors, use the following guidelines to write actions with MATLAB commands or `ml` data:

Guideline	Example
Return sizes of MATLAB commands or data in an expression must match return sizes of peer expressions.	In the expression `ml.`*`func`*`() * (x + ml.y)`, if `x` is a 3-by-2 matrix, then `ml.`*`func`*`()` and `ml.y` are also assumed to evaluate to 3-by-2 matrices. If either returns a value of different size (other than a scalar), an error results during run-time.
Expressions that return a scalar never produce an error. You can combine matrices and scalars in larger expressions because MATLAB commands use scalar expansion.	In the expression `ml.x + y`, if `y` is a 3-by-2 matrix and `ml.x` returns a scalar, the resulting value is the result of adding the scalar value of `ml.x` to every member of `y` to produce a matrix with the size of `y`, that is, a 3-by-2 matrix. The same rule applies to subtraction (`-`), multiplication (`*`), division (`/`), and any other binary operations.

Guideline		Example
MATLAB commands or Stateflow data of type `ml` can be members of these independent levels of expression, for which resolution of return size is necessary:	Arguments The expression for each function argument is a larger expression for which the return size of MATLAB commands or Stateflow data of type `ml` must be determined.	In the expression `z + func(x + ml.y)`, the size of `ml.y` is independent of the size of `z`, because `ml.y` is used at the function argument level. However, the return size for `func(x + ml.y)` must match the size of `z`, because they are both at the same expression level.
	Array indices The expression for an array index is an independent level of expression that must be scalar in size.	In the expression `x + array[y]`, the size of `y` is independent of the size of `x` because `y` and `x` are at different levels of expression. Also, `y` must be a scalar.
The return size for an indexed array element access must be a scalar.		The expression `x[1][1]`, where `x` is a 3-by-2 array, must evaluate to a scalar.
MATLAB command or data elements used in an expression for the input argument of a MATLAB function called through the `ml` namespace operator are resolved for size. This resolution uses the rule for peer expressions (preceding rule 1) for the expression itself, because no size definition prototype is available.		In the function call `ml.func(x + ml.y)`, if `x` is a 3-by-2 array, `ml.y` must return a 3-by-2 array or a scalar.
MATLAB command or data elements used for the input argument for a graphical function in an expression are resolved for size by the function prototype.		If the graphical function `gfunc` has the prototype `gfunc(arg1)`, where `arg1` is a 2-by-3 Stateflow data array, the calling expression, `gfunc(ml.y + x)`, requires that both `ml.y` and `x` evaluate to 2-by-3 arrays (or scalars) during run-time.
`ml` function calls can take only scalar or character vector literal arguments. Any MATLAB command or data that specifies an argument for the `ml` function must return a scalar value.		In the expression `a = ml("sin(x)")`, the `ml` function calls the MATLAB function `sin` to evaluate the sine of `x` in the MATLAB workspace. Stateflow data variable `a` stores that result.
In an assignment, the size of the right-hand expression must match the size of the left-hand expression, with one exception. If the left-hand expression is a single MATLAB variable, such as `ml.x`, or Stateflow data of type `ml`, the right-hand expression determines the sizes of both expressions.		In the expression `s = ml.func(x)`, where `x` is a 3-by-2 matrix and `s` is scalar Stateflow data, `ml.func(x)` must return a scalar to match the left-hand expression, `s`. However, in the expression `ml.y = x + s`, where `x` is a 3-by-2 data array and `s` is scalar, the left-hand expression, workspace variable `y`, is assigned the size of a 3-by-2 array to match the size of the right-hand expression, `x+s`, a 3-by-2 array.

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 character vector workspace variable, produces a run-time error stating that `ml.`*`str`* is not a numeric type.

Special cases exist, in which no size checking occurs to resolve the size of MATLAB command or data expressions that are part of larger expressions. Use of the following expressions does not require enforcement of size checking at run-time:

- `ml.`*`var`*
- `ml.`*`func`*`()`
- `ml(`*`evalString`*`, `*`arg1`*`, `*`arg2`*`,...)`
- Stateflow data of type `ml`
- Graphical function returning a Stateflow data of type `ml`

In these cases, assignment of a return to the left-hand side of an assignment statement or a function argument occurs without checking for a size mismatch between the two:

- An assignment in which the left-hand side is a MATLAB workspace variable

  For example, in the expression `ml.x = ml.y`, `ml.y` is a MATLAB workspace variable of any size and type (structure, cell array, character vector, and so on).
- An assignment in which the left-hand side is a data of type `ml`

  For example, in the expression `m_x = ml.`*`func`*`()`, `m_x` is a Stateflow data of type `ml`.
- Input arguments of a MATLAB function

  For example, in the expression `ml.`*`func`*`(m_x, ml.x, `*`gfunc`*`())`, `m_x` is a Stateflow data of type `ml`, `ml.x` is a MATLAB workspace variable of any size and type, and *`gfunc`*`()` is a Stateflow

graphical function that returns a Stateflow data of type `ml`. Although size checking does not occur for the input type, if the passed-in data is not of the expected type, an error results from the function call `ml.`*`func`*`()`.

- Arguments for a graphical function that are specified as Stateflow data of type `ml` in its prototype statement

---

Note If you replace the inputs in the preceding cases with non-MATLAB numeric Stateflow data, conversion to an `ml` type occurs.

Control Function-Call Subsystems by Using bind Actions

You can bind specified data and events to a state by using `bind` actions. Events bound to a state can be broadcast only by the actions in that state or its children. You can also bind a function-call event to a state to enable or disable the function-call subsystem that the event triggers. The function-call subsystem enables when the state with the bound event is entered and disables when that state is exited. Execution of the function-call subsystem is fully bound to the activity of the state that calls it.

Bind a Function-Call Subsystem to a State

By default, a function-call subsystem is controlled by the chart in which the associated function call output event is defined. This association means that the function-call subsystem is enabled when the chart wakes up and remains active until the chart goes to sleep. To achieve a finer level of control, you can bind a function-call subsystem to a state within the chart hierarchy by using a `bind` action (see “Bind Actions” on page 1-29).

You can bind function-call output events to a state. When you create this type of binding, the function-call subsystem that is called by the event is also bound to the state. In this situation, the function-call subsystem is enabled when the state is entered and disabled when the state is exited.

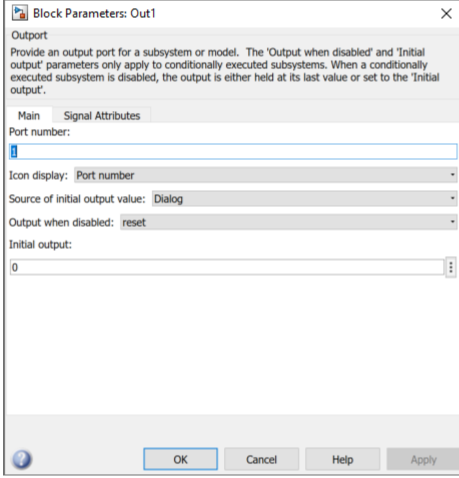
When you bind a function-call subsystem to a state, you can fine-tune the behavior of the subsystem when it is enabled and disabled, as described in the following sections:

- “Handle Outputs When the Subsystem is Disabled” on page 14-29
- “Control Behavior of States When the Subsystem is Enabled” on page 14-30

Handle Outputs When the Subsystem is Disabled

Although function-call subsystems do not execute while disabled, their output signals are available to other blocks in the model. If a function-call subsystem is bound to a state, you can hold its outputs at their values from the previous time step or reset the outputs to their initial values when the subsystem is disabled. Follow these steps:

1. Double-click the Outport block of the subsystem to open the Block Parameters dialog box.



2 Select an option for **Output when disabled**:

Select:	To:
`held`	Maintain most recent output value
`reset`	Reset output to its initial value

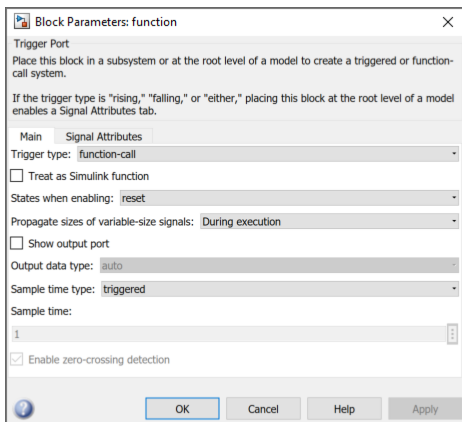
3 Click **OK** to record the settings.

Note Setting **Output when disabled** is meaningful only when the function-call subsystem is bound to a state, as described in “Bind a Function-Call Subsystem to a State” on page 14-29.

Control Behavior of States When the Subsystem is Enabled

If a function-call subsystem is bound to a state, you can hold the subsystem state variables at their values from the previous time step or reset the state variables to their initial conditions when the subsystem executes. In this way, the binding state gains full control of state variables for the function-call subsystem. Follow these steps:

1 Double-click the trigger port of the subsystem to open the Block Parameters dialog box.



2 Select an option for **States when enabling**:

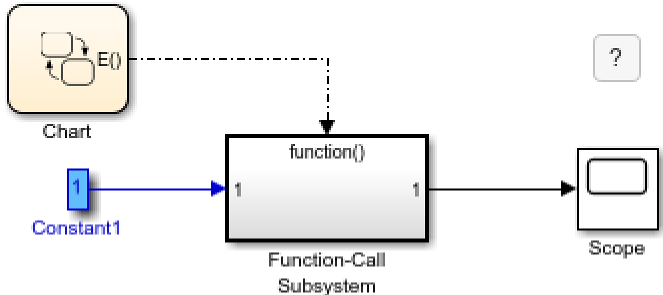
Select:	To:
`held`	Maintain most recent values of the states of the subsystem that contains the trigger port
`reset`	Revert to the initial conditions of the states of the subsystem that contains this trigger port
`inherit`	Inherit this setting from the parent subsystem of the function call initiator. If the parent of the initiator is the model root, the inherited setting is held. If the trigger has multiple initiators, the parents of all initiators must have the same setting: either all `held` or all `reset`.

3 Click **OK** to record the settings.

Note Setting **States when enabling** is meaningful only when the function-call subsystem is bound to a state, as described in “Bind a Function-Call Subsystem to a State” on page 14-29.

Bind a Function-Call Subsystem to a State

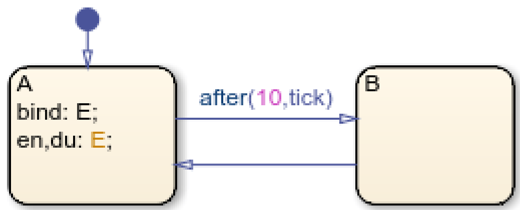
This model triggers a function-call subsystem with a trigger event `E` that binds to a state of a chart. The model uses a fixed-step solver with a fixed-step size of 1.



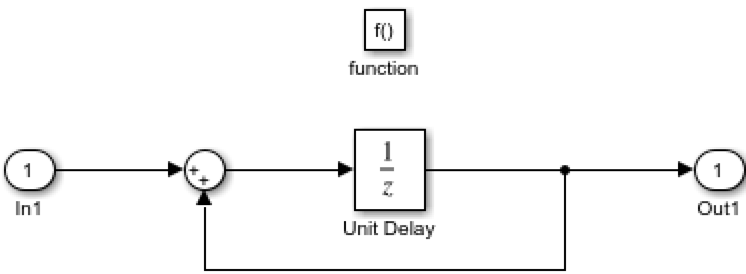
The chart contains two states. Event `E` binds to state `A` with the action

```
bind:E
```

Event `E` is defined for the chart with a scope of `Output to Simulink` and a trigger type of `function-call`.



The function-call subsystem contains a trigger port block, an input port, an output port, and a simple block diagram. The block diagram increments a counter by 1 at each time step, using a Unit Delay block.



The Block Parameters dialog box for the trigger port contains these settings:

- **Trigger type**: `function-call`.
- **States when enabling**: `reset`. This setting resets the state values for the function-call subsystem to zero when it is enabled.

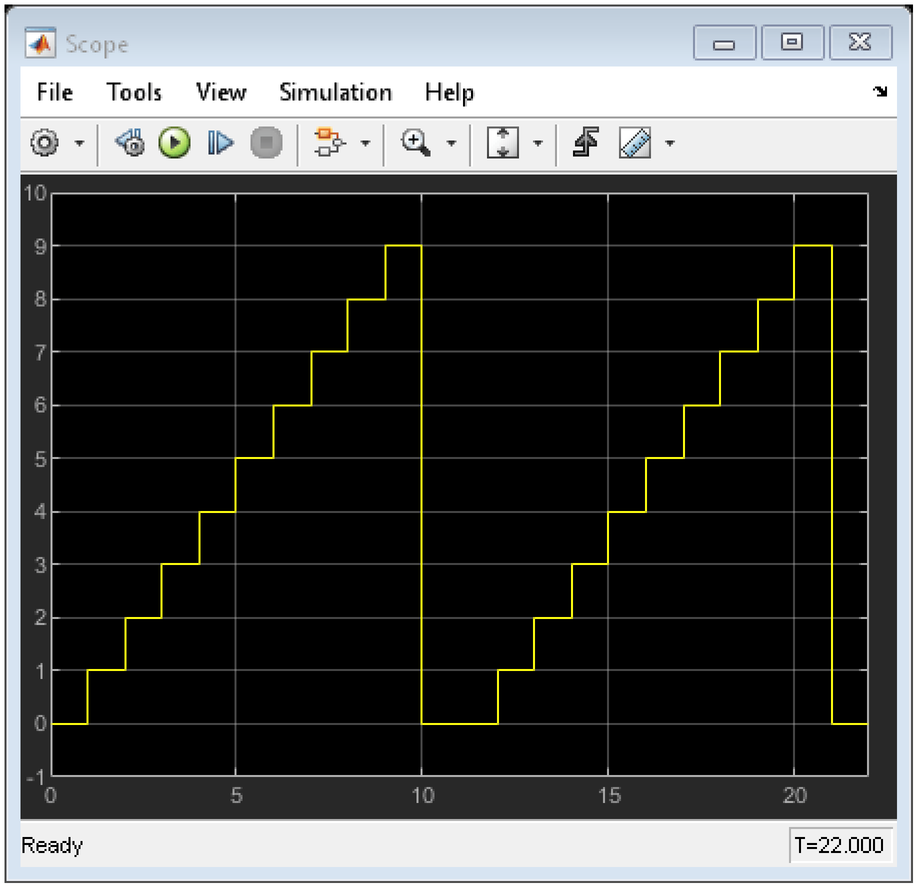
- **Sample time type**: `triggered`. This setting sets the function-call subsystem to execute only when it is triggered by a calling event while it is enabled.

Setting **Sample time type** to `periodic` enables the **Sample time** field below it, which defaults to 1. These settings force the function-call subsystem to execute for each time step specified in the **Sample time** field while it is enabled. To accomplish this, the state that binds the calling event for the function-call subsystem must send an event for the time step coinciding with the specified sampling rate in the **Sample time** field. States can send events with entry or during actions at the simulation sample rate.

- For fixed-step sampling, the **Sample time** value must be an integer multiple of the fixed-step size.
- For variable-step sampling, the Sample time value has no limitations.

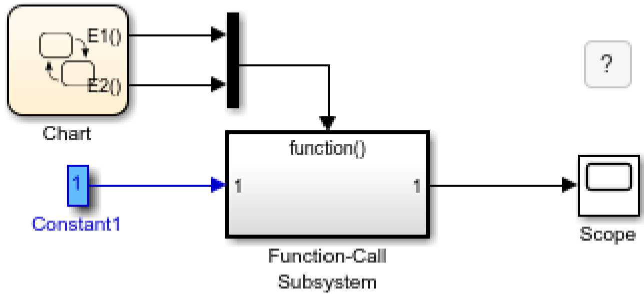
To see how a state controls a bound function-call subsystem, begin simulating the model.

- At time t = 0, the default transition to state `A` occurs. State `A` executes its bind and entry actions. The binding action binds event `E` to state `A`, enabling the function-call subsystem and resetting its state variables to 0. The entry action triggers the function-call subsystem and executes its block diagram. The block diagram increments a counter by 1 using a Unit Delay block. The Unit Delay block outputs a value of 0 and holds the new value of 1 until the next call to the subsystem.
- At time t = 1, the next update event from the model tests state `A` for an outgoing transition. The transition to state `B` does not occur because the temporal operator `after(10,tick)` allows the transition to be taken only after ten update events are received. State `A` remains active and its during action triggers the function-call subsystem. The Unit Delay block outputs its held value of 1. The subsystem also increments its counter to produce the value of 2, which the Unit Delay block holds until the next triggered execution.
- The next eight update events increment the subsystem output by one at each time step.
- At time t = 10, the transition from state `A` to state `B` occurs. Because the binding to state `A` is no longer active, the function-call subsystem is disabled, and its output drops back to 0.
- At time t = 11, the transition from state `B` to state `A` occurs. Again, the binding action enables the function-call subsystem. Subsequent update events increment the subsystem output by one at each time step until the next transition to state `B` occurs at time t = 21.

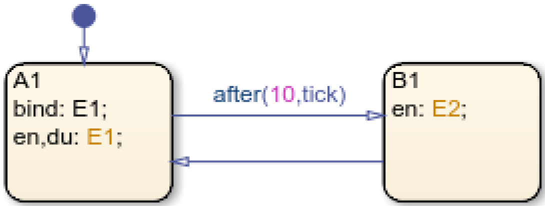


Avoid Muxed Trigger Events with Binding

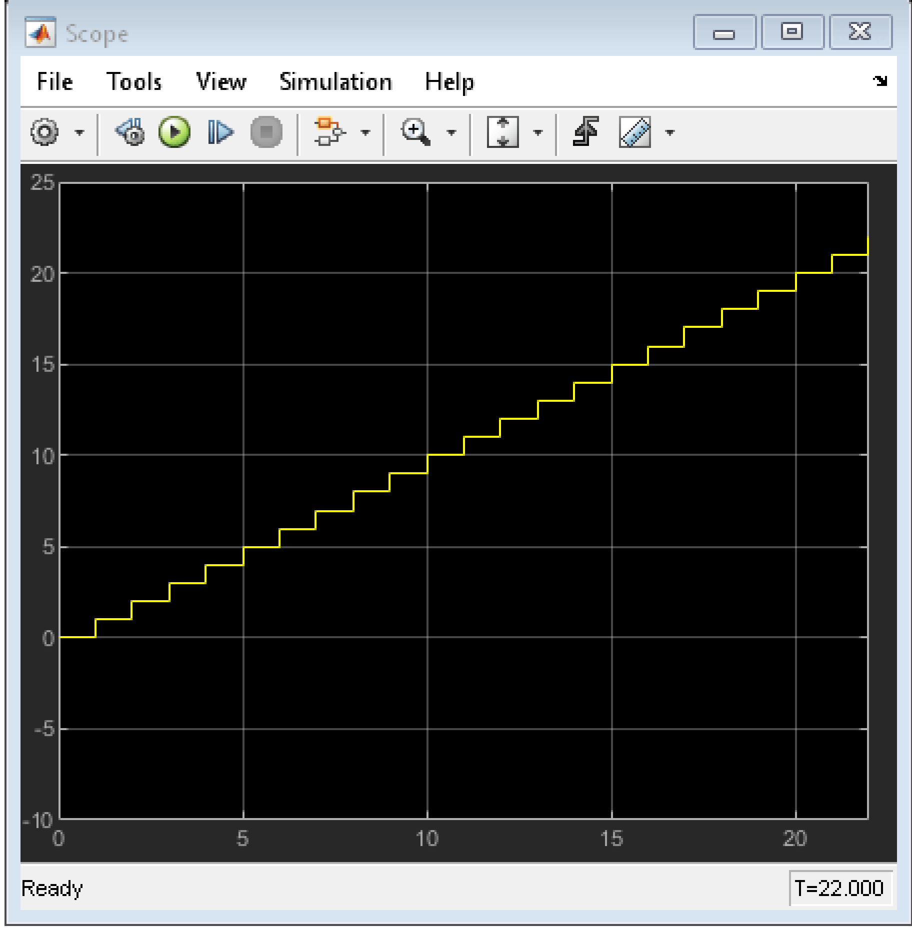
Binding events gives control of a function-call subsystem to a single state in a chart. This control does not work when you allow other events to trigger the function-call subsystem through a mux. For example, this model defines two function-call events to trigger a function-call subsystem using a Mux block.



In the chart, E1 binds to state A, but E2 does not. State B sends the triggering event E2 in its entry action.



When you simulate this model, the output does not reset when the transition from state A to state B occurs.



Binding is not recommended when you provide multiple trigger events to a function-call subsystem through a mux. Muxed trigger events can interfere with event binding and cause undefined behavior.

Control Chart Execution by Using Temporal Logic

Temporal logic controls the execution of a chart in terms of time. In state actions and transitions, you can use two types of temporal logic:

- Event-based temporal logic tracks recurring events. You can use any explicit or implicit event as a base event.
- Absolute-time temporal logic tracks the elapsed time since a state became active. The timing for absolute-time temporal logic operators depends on the type of Stateflow chart:
  - Charts in a Simulink model define absolute-time temporal logic in terms of simulation time.
  - Standalone charts in MATLAB define absolute-time temporal logic in terms of wall-clock time, which is limited to 1 millisecond precision.

Temporal Logic Operators

To define the behavior of a Stateflow chart based on temporal logic, use the operators listed in this table. These operators can appear in:

- State `on` actions
- Actions on transition paths that originate from a state

Each temporal logic operator has an associated state, which is the state in which the action appears or from which the transition path originates. The Stateflow chart resets the counter used by each operator every time that the associated state reactivates.

Operator	Syntax	Description	Example

`after`	`after(n,E)`
	
	`n` is a positive integer or an expression that evaluates to a positive integer value.
	
	`E` is the base event for the operator.
		Returns `true` if the event `E` has occurred at least `n` times since the associated state became active. Otherwise, the operator returns `false`.	Display a status message when the chart processes a broadcast of the event `E`, starting on the third broadcast of `E` after the state became active.
			
			`on after(3,E):`
			`   disp("ON");`

			Transition out of the associated state when the chart processes a broadcast of the event `E`, starting on the fifth broadcast of `E` after the state became active. `after(5,E)`

Operator	Syntax	Description	Example
	`after(n,tick)` `n` is a positive integer or an expression that evaluates to a positive integer value.	Returns `true` if the chart has woken up at least `n` times since the associated state became active. Otherwise, the operator returns `false`. The implicit event `tick` is not supported when a Stateflow chart in a Simulink model has input events. For more information, see “Control Chart Behavior by Using Implicit Events” on page 12-28.	Transition out of the associated state when the chart wakes up for at least the seventh time since the state became active, but only if the variable `temp` is greater than 98.6. `after(7,tick)[temp > 98.6]`

	`after(n,sec)`
	
	`after(n,msec)`
	
	`after(n,usec)`
	
	`n` is a positive real number or an expression that evaluates to a positive real value.
		Returns `true` if at least `n` units of time have elapsed since the associated state became active. Otherwise, the operator returns `false`.
		
		In charts in a Simulink model, specify time in seconds (`sec`), milliseconds (`msec`), or microseconds (`usec`).
		
		In standalone charts in MATLAB, specify time in seconds (`sec`). The operator creates a MATLAB `timer` object that generates an implicit event to wake up the chart. MATLAB `timer` objects are limited to 1 millisecond precision. For more information, see “Events in Standalone Charts” on page 2-42.
			Set the `temp` variable to `LOW` every time that the chart wakes up, starting when the associated state is active for at least 12.3 seconds.
			
			`on after(12.3,sec):`
			`   temp = LOW;`

`at`	`at(n,E)`
	
	`n` is a positive integer or an expression that evaluates to a positive integer value.
	
	`E` is the base event for the operator.
		Returns `true` if the event `E` has occurred exactly `n` times since the associated state became active. Otherwise, the operator returns `false`.	Display a status message when the chart processes the third broadcast of the event `E` after the state became active.
			
			`on at(3,E):`
			`   disp("ON");`
			
			Transition out of the associated state when the chart processes the fifth broadcast of the event `E` after the state became active.
			
			`at(5,E)`

Operator	Syntax	Description	Example
	`at(n,tick)` `n` is a positive integer or an expression that evaluates to a positive integer value.	Returns `true` if the chart has woken up exactly `n` times since the associated state became active. Otherwise, the operator returns `false`. The implicit event `tick` is not supported when a Stateflow chart in a Simulink model has input events. For more information, see "Control Chart Behavior by Using Implicit Events" on page 12-28.	Transition out of the associated state when the chart wakes up for the seventh time since the state became active, but only if the variable `temp` is greater than 98.6. `at(7,tick)[temp > 98.6]`

	`at(n,sec)`
	
	`n` is a positive real number or an expression that evaluates to a positive real value.
		Returns `true` if exactly `n` seconds have elapsed since the associated state became active. Otherwise, the operator returns `false`.
		
		Using `at` as an absolute-time temporal logic operator is supported only in standalone charts in MATLAB. The operator creates a MATLAB `timer` object that generates an implicit event to wake up the chart. MATLAB `timer` objects are limited to 1 millisecond precision. For more information, see "Events in Standalone Charts" on page 2-42.
			Set the `temp` variable to `HIGH` if the state has been active for exactly 12.3 seconds.
			
			`on at(12.3,sec):`
			`   temp = HIGH;`

`before`	`before(n,E)`
	
	`n` is a positive integer or an expression that evaluates to a positive integer value.
	
	`E` is the base event for the operator.
		Returns `true` if the event `E` has occurred fewer than `n` times since the associated state became active. Otherwise, the operator returns `false`.
		
		The temporal logic operator `before` is supported only in Stateflow charts in Simulink models.
			Display a status message when the chart processes the first and second broadcasts of the event `E` after the state became active.
			
			`on before(3,E):`
			`   disp("ON");`

			Transition out of the associated state when the chart processes a broadcast of the event `E`, but only if the state has been active for fewer than five broadcasts of `E`. `before(5,E)`

Operator	Syntax	Description	Example
	`before(n,tick)` `n` is a positive integer or an expression that evaluates to a positive integer value.	Returns `true` if the chart has woken up fewer than `n` times since the associated state became active. Otherwise, the operator returns `false`. The implicit event `tick` is not supported when a Stateflow chart in a Simulink model has input events. For more information, see “Control Chart Behavior by Using Implicit Events” on page 12-28. The temporal logic operator `before` is supported only in Stateflow charts in Simulink models.	Transition out of the associated state when the chart wakes up, but only if the variable `temp` is greater than 98.6 and the chart has woken up fewer than seven times since the state became active. `before(7,tick)[temp > 98.6]`

	`before(n,sec)`
	
	`before(n,msec)`
	
	`before(n,usec)`
	
	`n` is a positive real number or an expression that evaluates to a positive real value.
		Returns `true` if fewer than `n` units of time have elapsed since the associated state became active. Otherwise, the operator returns `false`.
		
		Specify time in seconds (`sec`), milliseconds (`msec`), or microseconds (`usec`).
		
		The temporal logic operator `before` is supported only in Stateflow charts in Simulink models.
			Set the `temp` variable to `MED` every time that the chart wakes up, but only if the associated state has been active for fewer 12.3 seconds.
			
			`on before(12.3,sec):`
			`   temp = MED;`

`every`	`every(n,E)`
	
	`n` is a positive integer or an expression that evaluates to a positive integer value.
	
	`E` is the base event for the operator.
		Returns `true` at every `n`th occurrence of the event `E` since the associated state became active. Otherwise, the operator returns `false`.	Display a status message when the chart processes every third broadcast of the event `E` after the state became active.
			
			`on every(3,E):`
			`   disp("ON");`

			Transition out of the associated state when the chart processes every fifth broadcast of the event `E` after the state became active. `every(5,E)`

Operator	Syntax	Description	Example
	`every(n,tick)` `n` is a positive integer or an expression that evaluates to a positive integer value.	Returns `true` at every `n`th time that the chart wakes up since the associated state became active. Otherwise, the operator returns `false`. The implicit event `tick` is not supported when a Stateflow chart in a Simulink model has input events. For more information, see “Control Chart Behavior by Using Implicit Events” on page 12-28.	Transition out of the associated state every seventh `tick` event since the state became active, but only if the variable `temp` is greater than 98.6. `every(7,tick)[temp > 98.6]`

	`every(n,sec)`
	
	`n` is a positive real number or an expression that evaluates to a positive real value.
		Returns `true` every `n` seconds since the associated state became active. Otherwise, the operator returns `false`.
		
		Using `every` as an absolute-time temporal logic operator is supported only in standalone charts in MATLAB. The operator creates a MATLAB `timer` object that generates an implicit event to wake up the chart. MATLAB `timer` objects are limited to 1 millisecond precision. For more information, see “Events in Standalone Charts” on page 2-42.
			Increment the `temp` variable by 5 every 12.3 seconds that the state is active.
			
			`on every(12.3,sec):`
			`   temp = temp+5;`

`temporal Count`	`temporalCount(E)`
	
	`E` is the base event for the operator.
		Returns the number of occurrences of the event `E` since the associated state became active.
		
		Using `temporalCount` as an event-based temporal logic operator is supported only in Stateflow charts in Simulink models.
			Access successive elements of the array `M` each time that the chart processes a broadcast of the event `E`.
			
			`on E:`
			`   y = M(temporalCount(E));`

	`temporalCount(tick )`	Returns the number of times that the chart has woken up since the associated state became active.
		
		The implicit event `tick` is not supported when a Stateflow chart in a Simulink model has input events.
		
		Using `temporalCount` as an event-based temporal logic operator is supported only in Stateflow charts in Simulink models.
			Store the value of the input data `u` in successive elements of the array `M`.
			
			`en,du:`
			`   M(temporalCount(tick)+1) = u;`

Operator	Syntax	Description	Example

	`temporalCount(sec)`
	
	`temporalCount(msec)`
	
	`temporalCount(usec)`
		Returns the length of time that has elapsed since the associated state became active.
		
		Specify time in seconds (`sec`), milliseconds (`msec`), or microseconds (`usec`).
			Store the number of milliseconds since the state became active.
			
			`en,du:`
			`   y = temporalCount(msec);`

`elapsed`	`elapsed(sec)`	Returns the length of time that has elapsed since the associated state became active.
		
		Equivalent to `temporalCount(sec)`.
			Store the number of seconds since the state became active.
			
			`en,du:`
			`   y = elapsed(sec);`

	`et`	An alternative way to execute `elapsed(sec)`.	When the chart processes a broadcast of the event `E`, transition out of the associated state and display the elapsed time since the state became active. `E{disp(et);}`
`count`	`count(C)` `C` is an expression that evaluates to `true` or `false`.	Returns the number of times that the chart has woken up since the conditional expression `C` became `true` and the associated state became active. The Stateflow chart resets the value of the `count` operator if the conditional expression `C` becomes `false` or if the associated state becomes inactive. In charts in a Simulink model, the value of `count` may depend on the step size. Changing the solver or step size for the model affects the results produced by the `count` operator.	Transition out of the associated state when the variable `x` has been greater than or equal to 2 for longer than five chart executions. `[count(x>=2) > 5]`

			Store the number of chart executions since the variable `x` became greater than 5.
			
			`en,du:`
			`   y = count(x>5);`

`duration`	`duration(C)`
	
	`duration(C,sec)`
	
	`duration(C,msec)`
	
	`duration(C,usec)`
	
	• `C` is an expression that evaluates to `true` or `false`.
		Returns the length of time that has elapsed since the conditional expression `C` became `true` and the associated state became active.
		
		Specify time in seconds (`sec`), milliseconds (`msec`), or microseconds (`usec`). The default unit is seconds.
		
		The Stateflow chart resets the value of the `duration` operator if the
			Transition out of the state when the variable `x` has been greater than or equal to 0 for longer than 0.1 seconds.
			
			`[duration(x>=0) > 0.1]`

Operator	Syntax	Description	Example

		conditional expression C becomes false or if the associated state becomes inactive.
		
		The temporal logic operator duration is not supported in standalone charts in MATLAB.
			Store the number of milliseconds since the variable x became greater than 5 and the state became active.
			
			`en,du:`
			`y = duration(x>5,msec);`

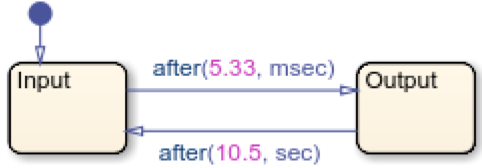
You can use quotation marks to enclose the keywords 'tick', 'sec', 'msec', and 'usec'. For example, after(5,'tick') is equivalent to after(5,tick).

Note The temporal logic operators after, at, before, and every compare the threshold n to an internal counter of integer type. If n is a fixed-point number defined by either a slope that is not an integer power of two or a nonzero bias, then the comparison can yield unexpected results due to rounding. For more information, see “Relational Operations for Fixed-Point Data” on page 23-19.

Examples of Temporal Logic

Define Time Delays

This example shows how to define two absolute time delays in a continuous-time chart.



The execution of the chart follows these steps:

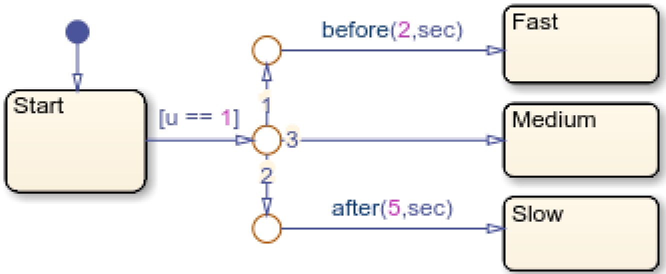
1. When the chart awakens, the state Input activates first.
2. After 5.33 milliseconds of simulation time, the transition from Input to Output occurs.
3. The state Input becomes inactive and the state Output becomes active.
4. After 10.5 seconds of simulation time, the transition from Output to Input occurs.
5. The state Output becomes inactive and the state Input becomes active.

Steps 2 through 5 are repeated until the simulation ends.

If a chart has a discrete sample time, any action in the chart occurs at integer multiples of this sample time. For example, if the Simulink® solver uses a fixed step of size 0.1 seconds, the first transition from state Input to state Output occurs at t = 0.1 seconds. This behavior applies because the solver does not wake the chart at exactly t = 5.33 milliseconds. Instead, the solver wakes the chart at integer multiples of 0.1 seconds, such as t = 0.0 and 0.1 seconds.

Detect Elapsed Time

In this example, a Step (Simulink) block provides a unit step input to a Stateflow chart.



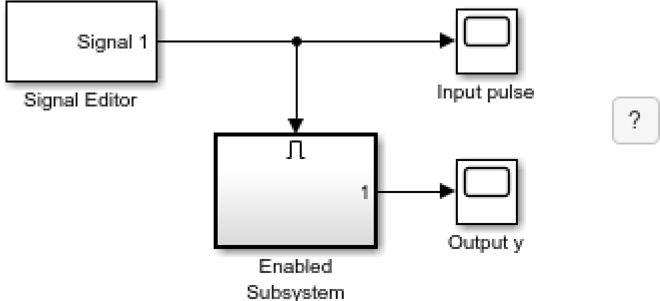
The chart determines when the input u equals 1:

- If the input equals 1 before t = 2 seconds, a transition occurs from `Start` to `Fast`.
- If the input equals 1 between t = 2 and t = 5 seconds, a transition occurs from `Start` to `Medium`.
- If the input equals 1 after t = 5 seconds, a transition occurs from `Start` to `Slow`.

Use Absolute-Time Temporal Logic in an Enabled Subsystem

You can use absolute-time temporal logic in a chart that resides in a conditionally executed subsystem. When the subsystem is disabled, the chart becomes inactive and the temporal logic operator pauses while the chart is asleep. The operator does not continue to count simulation time until the subsystem is reenabled and the chart is awake.

This model has an enabled subsystem with the **States when enabling** parameter set to `held`.

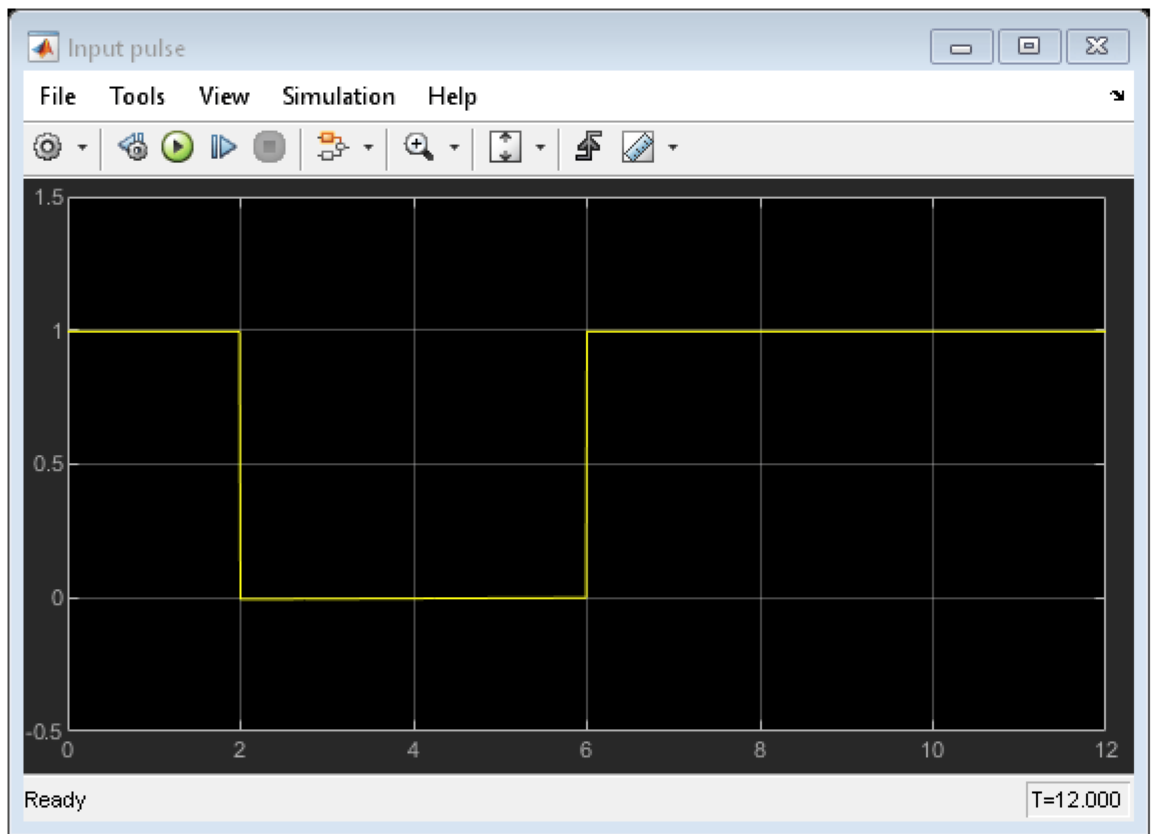


The subsystem contains a chart that uses the `after` operator to trigger a transition.

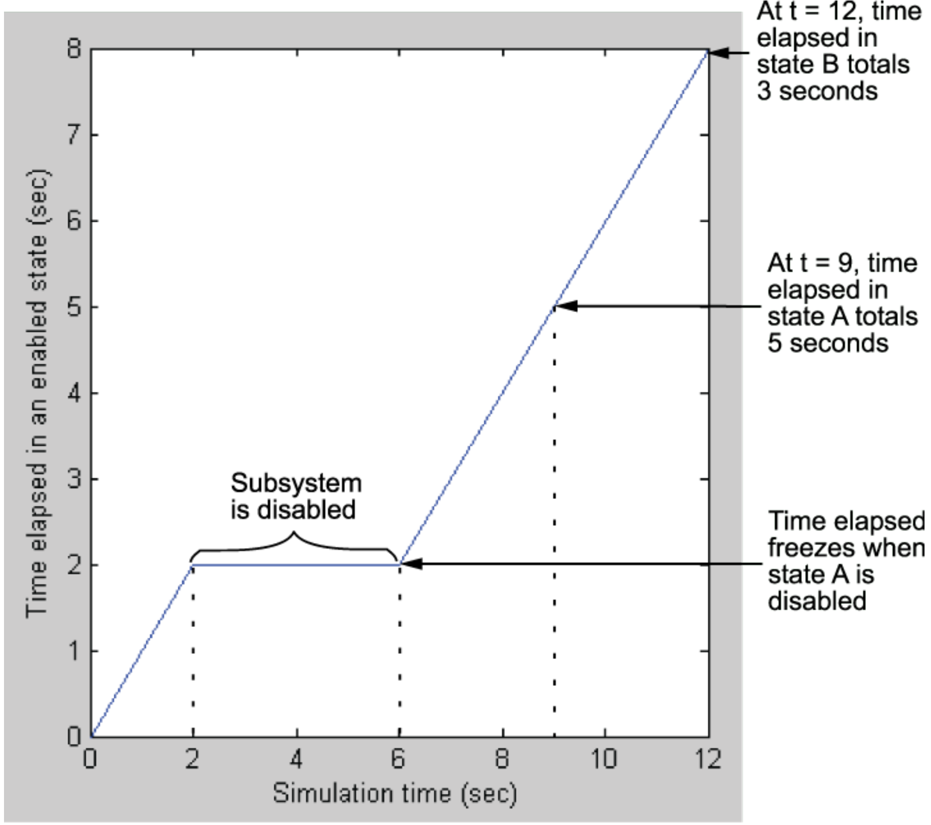
The Signal Editor (Simulink) block provides an input signal with these characteristics:

- The signal enables the subsystem at t = 0.

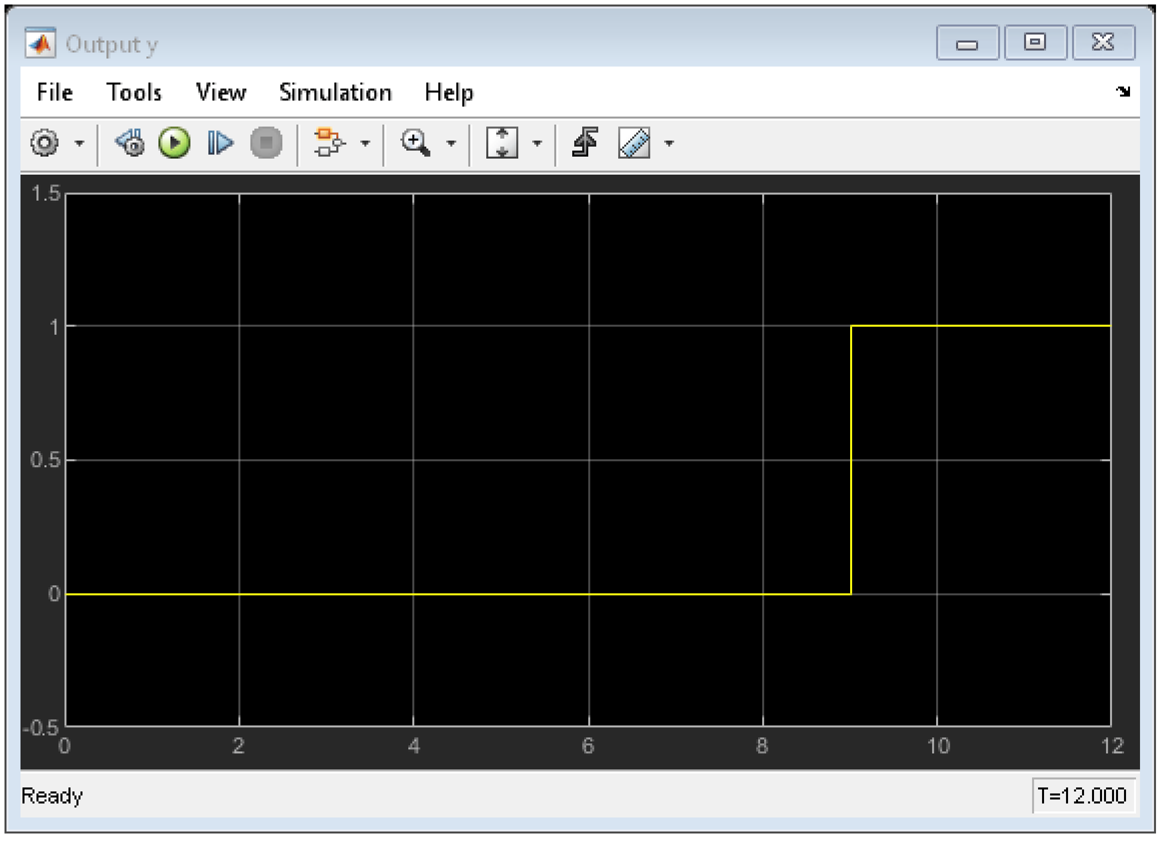
- The signal disables the subsystem at t = 2.
- The signal reenables the subsystem at t = 6.



This graph shows the total time elapsed in the chart. When the input signal enables the subsystem at time t = 0, state A becomes active. While the system is enabled, the elapsed time increases. When the subsystem is disabled at t = 2, the chart goes to sleep and the elapsed time stops increasing. For 2 < t < 6, the elapsed time stays frozen at 2 seconds because the system is disabled. When the chart wakes up at t = 6, elapsed time starts to increase again.



The transition from state A to state B depends on the elapsed time while state A is active, not on the simulation time. Therefore, the transition occurs at t = 9, when the elapsed time in state A equals 5 seconds. When the transition occurs, the output value y changes from 0 to 1.



This model behavior applies only to subsystems where you set the Enable block parameter **States when enabling** to `held`. If you set the parameter to `reset`, the chart reinitializes completely when the subsystem is reenabled. Default transitions execute and any temporal logic counters reset to 0.

Notation for Event-Based Temporal Logic in Transitions

In Stateflow charts in Simulink models, the operators `after`, `at`, and `before` support two distinct notations to express event-based temporal logic in a transition.

- Trigger notation defines a transition that depends only on the base event for the temporal logic operator. Trigger notation follows this syntax:

  ```
  temporalLogicOperator(n,E)[C]
  ```

  where:

  - *`temporalLogicOperator`* is a Boolean temporal logic operator.
  - `n` is the occurrence count of the operator.
  - `E` is the base event of the operator.
  - `C` is an optional condition expression.

  When you use trigger notation, the transition can occur only when the chart processes a broadcast of the base event `E`.
- Conditional notation defines a transition that depends on base and nonbase events. Conditional notation follows this syntax:

```
F[temporalLogicOperator(n,E) && C]
```

where:

- *temporalLogicOperator* is a Boolean temporal logic operator.
- n is the occurrence count of the operator.
- E is the base event of the operator.
- F is an optional nonbase event.
- C is an optional condition expression.

When you use conditional notation with a nonbase event F, the transition can occur only when the chart processes a broadcast of F. If you omit the nonbase event, the transition can occur when the chart is processing any explicit or implicit event.

Conditional notation for temporal logic operators is not supported in standalone charts in MATLAB.

For example, this transition label uses trigger notation to indicate a transition out of the associated state when the chart processes a broadcast of the base event E, starting on the fifth broadcast of E after the state became active.

```
after(5,E)
```

In contrast, this transition label uses conditional notation to indicate a transition out of the associated state when the state has been active for at least five broadcasts of the base event E, even if the chart is not processing a broadcast of E.

```
[after(5,E)]
```

Note The operator every supports trigger and conditional notations. However, both notations are equivalent for this operator. The transition labels every(5,E) and [every(5,E)] indicate a transition out of the associated state when the chart processes the kth broadcast of the base event E after the state became active, where k is a multiple of five.

Best Practices for Temporal Logic

Do Not Use Temporal Logic on Transition Paths Without A Source State

The value of a temporal logic operator depends on when its associated state became active. To ensure that every temporal logic operator has a unique associated state, only use these operators in:

- State on actions
- Actions on transition paths that originate from a state

Do not use temporal logic operators on default transitions or on transitions in graphical functions because these transitions do not originate from a state.

Use Absolute-time Temporal Logic Instead of tick in Charts in Simulink Models

In charts in a Simulink model, the value of delay expressions that use absolute-time temporal logic are semantically independent of the sample time of the model. In contrast, delay expressions that use temporal logic based on the implicit event tick depend on the step size used by the Simulink solver.

Additionally, absolute-time temporal logic is supported in charts that have input events. The implicit event `tick` is not supported when a Stateflow chart in a Simulink model has input events.

Do Not Use at for Absolute-Time Temporal Logic in Charts in Simulink Models

In charts in a Simulink model, using `at` as an absolute-time temporal logic operator is not supported. Instead, use the `after` operator. For example, suppose that you want to define a time delay using the expression `at(5.33,sec)`.

To prevent a run-time error, change the transition label to `after(5.33,sec)`.

Unexpected Results for Large Parameter Values

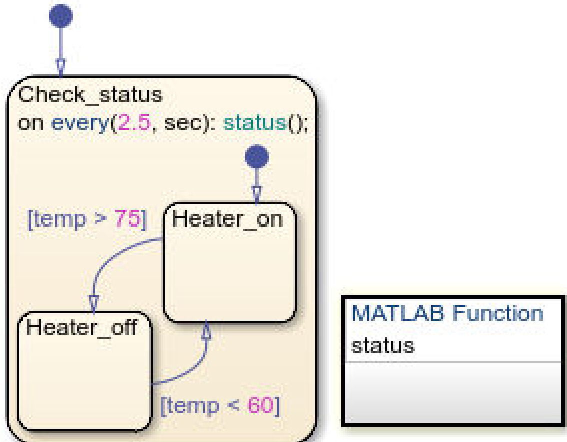
A Stateflow absolute time temporal logic condition such as `after(x,sec)` might not evaluate to `true` at the expected time after entering a state with the following conditions:

- The chart has a periodic discrete sample time.
- The chart logic makes the state remain active for greater than `2147418` units of time. The units of time are the smallest time units in any temporal logic expression used by that state. For example, if the state has two outgoing transitions, one using `after(x,sec)` and the other using `after(x,msec)`, the units of time are `msec (milliseconds)`.

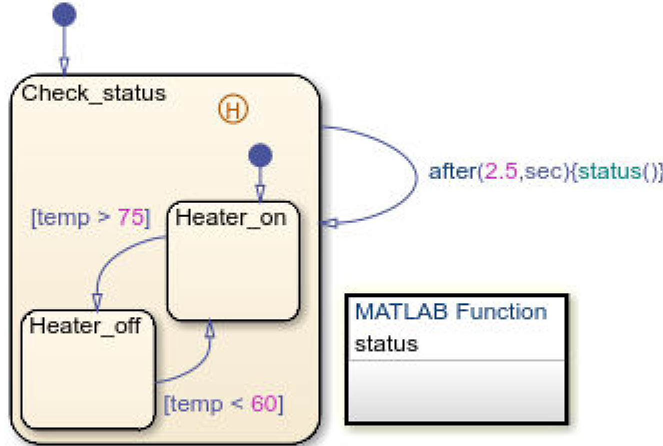
Typically, unexpected results occur when the length of time in the state is bigger than `2147418` units of time. However, this may change depending on the sample time of the chart.

Do Not Use every for Absolute-Time Temporal Logic in Charts in Simulink Models

In charts in a Simulink model, using `every` as an absolute-time temporal logic operator is not supported. Instead, use an outer self-loop transition with the `after` operator. For example, suppose that you want to print a status message for an active state every 2.5 seconds during chart execution.



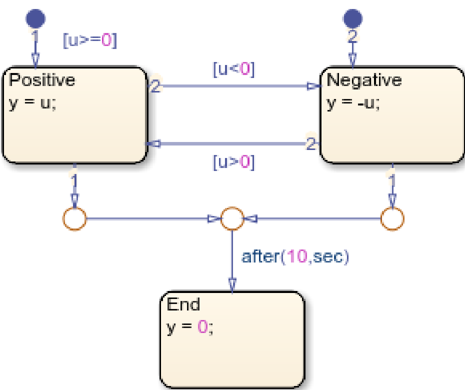
To prevent a run-time error, replace the state action with an outer self-loop transition.



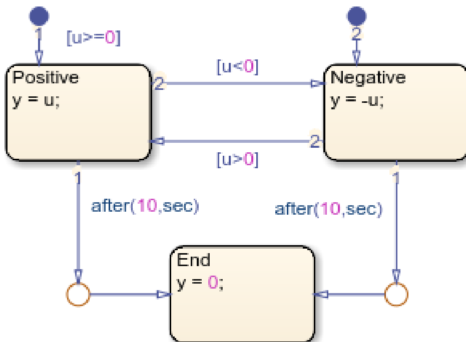
Add a history junction in the state so the chart remembers the state settings prior to each self-loop transition. See “Resume Prior Substate Activity by Using History Junctions” on page 1-58.

Do Not Use Temporal Logic in Transition Paths with Multiple Sources in Standalone Charts in MATLAB

Standalone charts in MATLAB do not support the use of temporal logic operators on transition paths that have more than one source state. For example, this standalone chart produces a run-time error because the temporal logic expression `after(10,sec)` triggers a transition path that has more than one source state.



To resolve the issue, use temporal logic expressions on separate transition paths, each with a single source state.

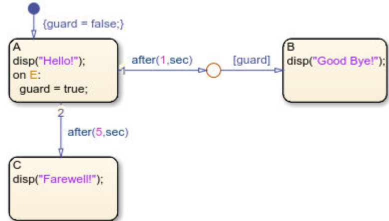


Avoid Mixing Absolute-time Temporal Logic and Conditions in Transition Paths of Standalone Charts in MATLAB

In standalone charts in MATLAB, the operators `after`, `at`, and `every` create MATLAB `timer` objects that generate implicit events to wake up the chart. Combining these operators with conditions on the same transition path can result in unintended behavior:

- If a condition on the transition path is false when the `timer` wakes up the chart, the chart performs the `during` and `on` actions of the active state.
- The chart does not reset the `timer` object associated with the operators `after` and `at`. If the condition on the transition path becomes true at a later time, the transition does not take place until another explicit or implicit event wakes up the chart.

For example, in this chart, the transition path from state `A` to state `B` combines the absolute-time temporal logic trigger `after(1,sec)` and the condition `[guard]`. The transition from state `A` to state `C` has the absolute-time temporal logic trigger `after(5,sec)`. Each transition is associated with a `timer` object that generates an implicit event. Initially, the local variable `guard` is `false`.



When you execute the chart, state A becomes active. The chart performs the `entry` action and displays the message `Hello!`. After 1 second, the `timer` associated with the transition from A to B wakes up the chart. Because the transition is not valid, the chart executes the `during` action in state A and displays the message `Hello!` a second time.

Suppose that, after 2 seconds, the chart receives the input event E. The chart executes the `on` action in state A and changes the value of `guard` to `true`. Because the chart does not reset the `timer` associated with the operator `after`, the transition from A to B does not take place until another event wakes up the chart.

After 5 seconds, the `timer` associated with the transition from A to C wakes up the chart. Because the transition from A to B is valid and has a higher execution order, the chart does not take the transition to state C or display the message `Farewell!`. Instead, state B becomes active and the chart displays the message `Good bye!`.

Use Charts with Discrete Sample Times for More Efficient Code Generation

The code generated for discrete charts that are not inside a triggered or enabled subsystem uses integer counters to track time instead of the time provided by Simulink. This behavior allows for more efficient code generation in terms of overhead and memory, and enables this code for use in software-in-the-loop (SIL) and processor-in-the-loop (PIL) simulation modes. For more information, see “SIL and PIL Simulations” (Embedded Coder).

See Also

`after` | `at` | `before` | `every` | `temporalCount` | `elapsed` | `count` | `duration` | `timer` | Signal Editor | Step

More About

- “Control Oscillations by Using the duration Operator” on page 14-55
- “Implement an Automatic Transmission Gear System by Using the duration Operator” on page 14-58
- “Count Events by Using the temporalCount Operator” on page 14-61
- “Events in Standalone Charts” on page 2-42
- “Resume Prior Substate Activity by Using History Junctions” on page 1-58

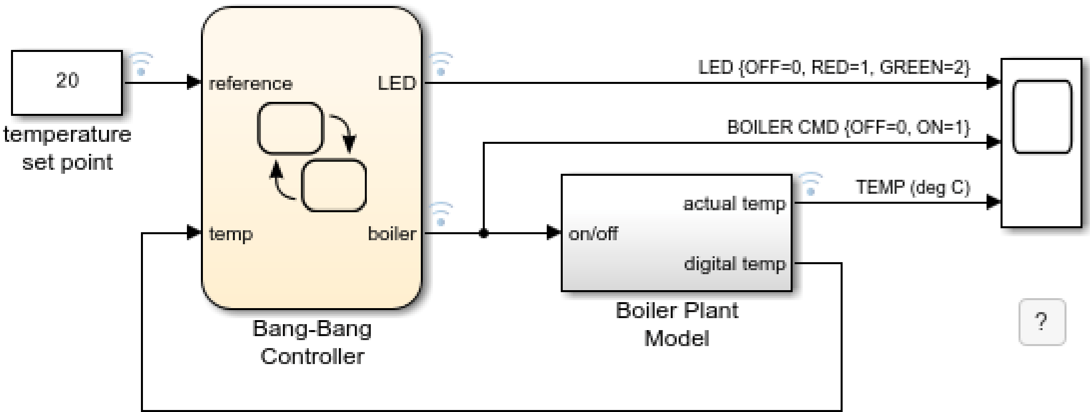
Model Bang-Bang Temperature Control System

This example shows how to model a bang-bang control system that regulates the temperature of a boiler. The model has two components:

- Boiler Plant Model is a Simulink® subsystem the models the boiler dynamics.
- Bang-Bang Controller is a Stateflow® chart that implements the bang-bang control logic.

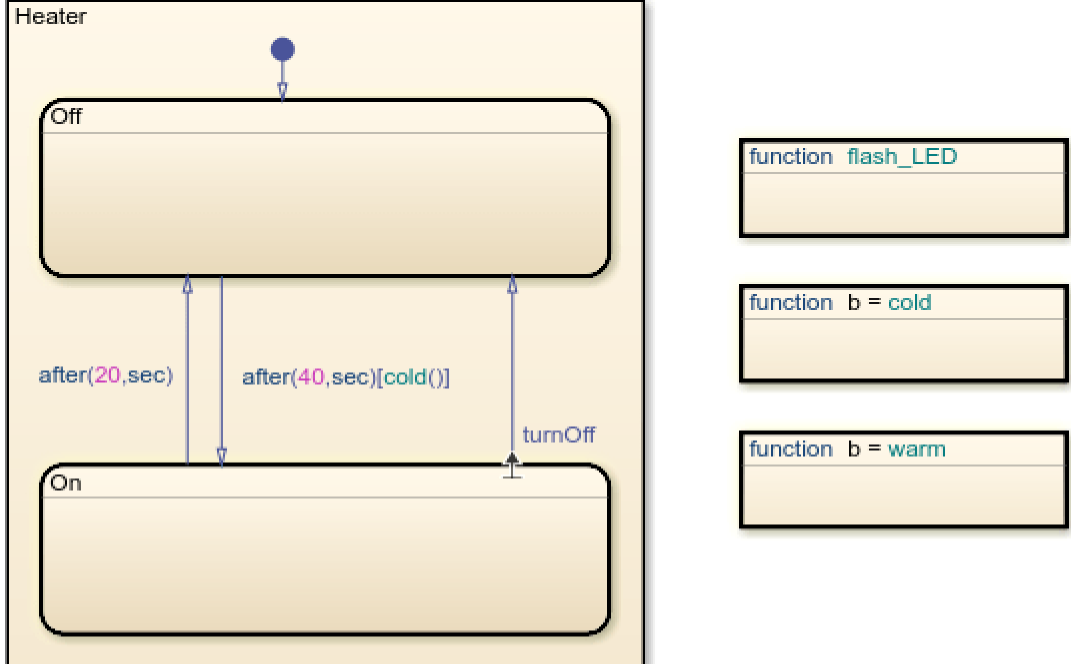
The chart uses:

- temporal logic to implement the timing of the bang-bang cycle
- 8-bit fixed-point data to represent the temperature of the boiler



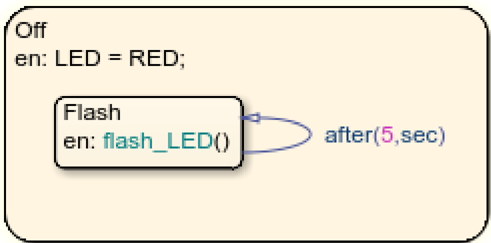
Implement Control Logic by Using `after` Operator

The Bang-Bang Controller determines when the boiler turns on or off. Initially, the boiler is off. After 40 seconds, if the boiler is cold, the boiler turns on. After 20 seconds, the boiler turns off and the bang-bang control cycle repeats.



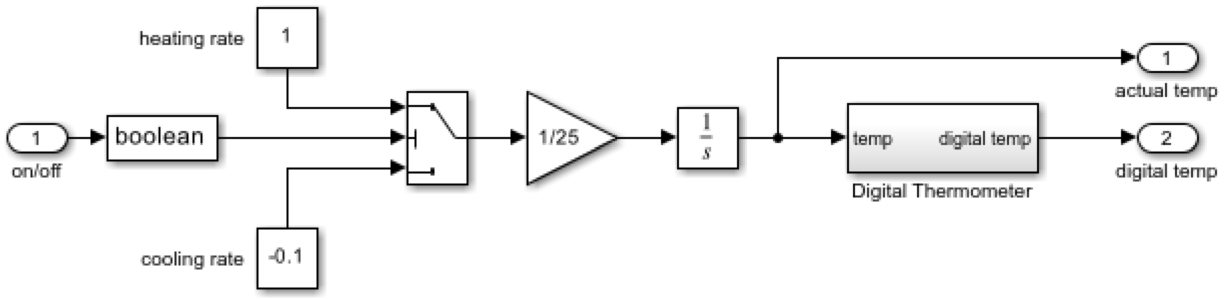
To control the transitions between the `On` and `Off` states, the chart calls the absolute-time temporal logic operator `after`. For example, the transition label `after(20,sec)` triggers the transition from `On` to `Off` after the `On` state is active for 20 seconds. The label `after(40,sec)[cold()]` causes the transition from `Off` to `On` to occur if the function `cold` returns `true` after the `Off` state is active for 40 seconds.

The `Off` state also uses temporal logic to control a status LED. Because Stateflow charts in Simulink models do not support the operator `every` for absolute time temporal logic, the state implements the operation of the LED by using a substate `Flash` with a self-loop transition. The transition label `after(5,sec)` triggers the entry action of the substate and causes the LED to flash every 5 seconds.

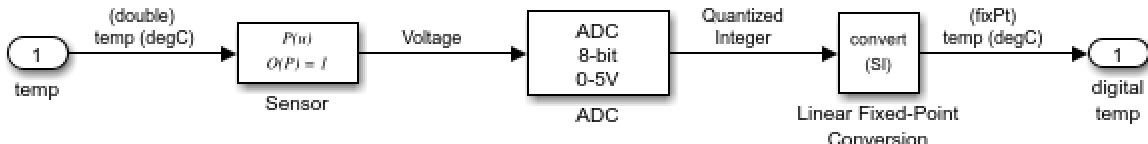


Process Floating-Point Data on an 8-Bit Processor

The Boiler Plant Model subsystem simulates the temperature reaction of the boiler during periods of heating or cooling.

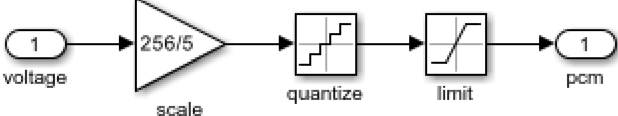


Depending on the output of the Bang-Bang Controller chart, the subsystem adds or subtracts a temperature increment (+1 for heating or –0.1 for cooling) to the previous boiler temperature and passes the result to the Digital Thermometer subsystem.



The Digital Thermometer subsystem converts the resulting temperature into an 8-bit fixed-point representation. The conversion occurs in three steps.

- The Sensor block converts input boiler temperature T_{actual} to an intermediate analog voltage output $V_{sensor} = 0.05 \cdot T_{actual} + 0.75$.
- The Analog-to-Digital Converter (ADC) subsystem digitizes the analog voltage from the sensor block by multiplying the voltage by $\frac{256}{5}$, rounding to the integer floor, and then limiting the result to a maximum of 255 (the largest unsigned 8-bit integer value). The subsystem outputs a quantized integer $Q = \lfloor \frac{256}{5} \cdot V_{sensor} \rfloor = \lfloor \frac{256 \times 0.05}{5} \cdot T_{actual} + \frac{256 \times 0.75}{5} \rfloor$.

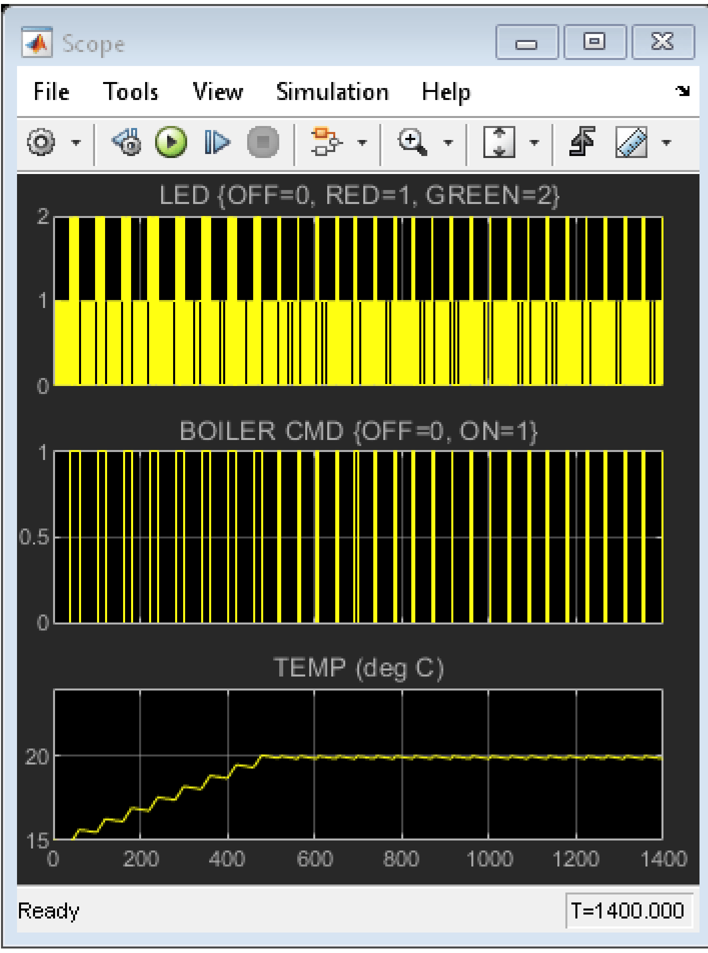


- The Linear Fixed-Point Conversion block inverts the combined transfer function of the Sensor and ADC blocks to encode the boiler temperature as a fixed-point number with a slope of $S = \frac{5}{256 \times 0.05} = 0.390625$ and a bias of $B = -\frac{0.75}{0.05} = -15$. These fixed-point parameters convert the 8-bit quantized integer Q into a digital coded temperature $T_{digital} = SQ + B = \frac{5}{256 \times 0.05} \cdot \lfloor \frac{256 \times 0.05}{5} \cdot T_{actual} + \frac{256 \times 0.75}{5} \rfloor - \frac{0.75}{0.05} \approx T_{actual}$.

The Bang-Bang Controller chart receives this digital coded temperature and interprets it as the unsigned 8-bit fixed-point data `temp`. The chart processes this temperature data in an 8-bit environment without any explicit conversions.

Examine Simulation Results

After simulation, the Simulink scope shows that the boiler reaches a temperature of 20 degrees Celsius after approximately 450 seconds (7.5 minutes). The bang-bang control logic effectively maintains that temperature for the rest of the simulation.



See Also

`after` | `every` | Data Type Conversion

More About

- “Control Chart Execution by Using Temporal Logic” on page 14-35
- “Fixed-Point Data in Stateflow Charts” on page 23-2
- “Model Bang-Bang Controller by Using a State Transition Table” on page 16-19

Control Oscillations by Using the duration Operator

The following example focuses on the gear logic of a car as it shifts from first gear to fourth gear.

When modeling the gear changes of this system, it is important to control the oscillation that occur. The model `sf_car` uses parallel state debouncer logic that controls which gear state is active. For more information about how debouncers work in Stateflow, see “Reduce Transient Signals by Using Debouncing Logic” on page 27-12.

You can simplify the debouncer logic by using the `duration` operator. You can see this simplification in the model `sf_car_using_duration`. The `duration` operator evaluates a condition expression and outputs the length of time that the expression has been `true`. When that length of time crosses a known time threshold, the state transitions to a higher or lower gear.

By removing the parallel state logic and using the `duration` operator, you can control oscillations with simpler Stateflow logic. The `duration` operator is supported only in Stateflow charts in a Simulink model.

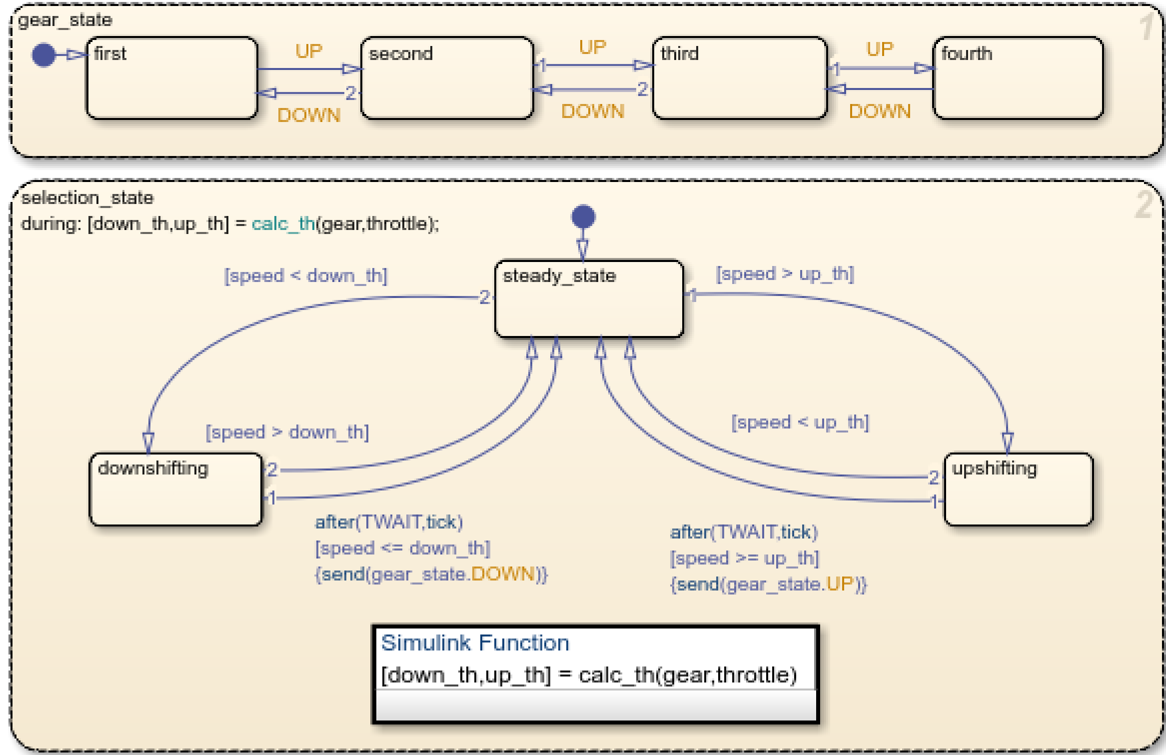
Control Oscillation with Parallel State Logic

Open the model `sf_car`.

```
openExample("stateflow/AutomaticTransmissionWithActiveStateDataExample")
```

Select the chart `shift_logic` and, in the **State Chart** tab, click **Look Under Mask**.

The Stateflow chart `shift_logic` controls which gear the car is in, given the speed of the car and how much throttle is being applied. Within `shift_logic` there are two parallel states: `gear_state` and `selection_state`. `gear_state` contains four exclusive states for each gear. `selection_state` determines whether the car is downshifting, upshifting, or remaining in its current gear.



In this Stateflow chart, for the car to move from first gear to second gear, the event `UP` must be sent from `selection_state` to `gear_state`. The event is sent when the speed crosses the threshold and remains higher than the threshold for the length of time determined by `TWAIT`. When the event `UP` is sent, `gear_state` transitions from `first` to `second`.

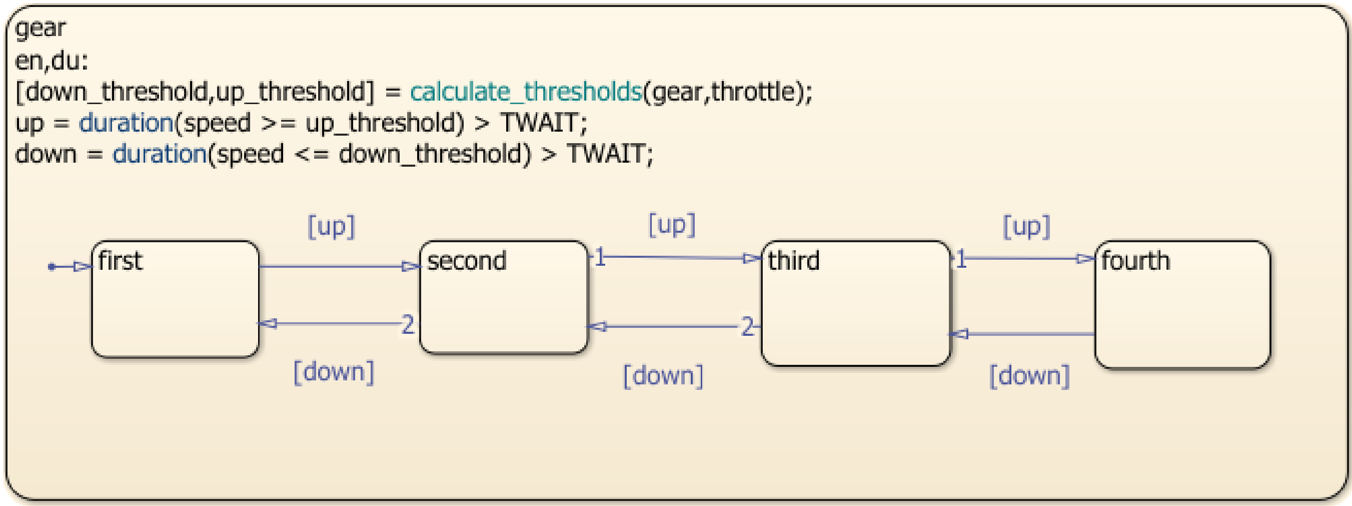
Control Oscillation with the duration Operator

Open the model `sf_car_using_duration`.

```
openExample("stateflow/AutomaticTransmissionUsingDurationOperatorExample")
```

Select the chart `Gear_Logic` and, in the **State Chart** tab, click **Look Under Mask**.

Within `Gear_Logic` there are four exclusive states for each gear. The local variables `up` and `down` guard the transitions between each state.



Simulink Function
[down_th,up_th] = calculate_thresholds(gear,throttle)

In this Stateflow chart, for the car to move from first gear to second gear, the condition `up` must be `true`. The condition `up` is defined as true if the length of time that the speed is greater than or equal to the threshold is greater than the length of time that is specified by `TWAIT`. The condition `down` is defined as true if the length of time that the speed is less than or equal to the threshold is greater than the length of time that is specified by `TWAIT`. The operator `duration` keeps track of the length of time that the speed has been above or below the threshold. When the `up` condition is met, the active state transitions from `first` to `second`.

By replacing the parallel state debouncer logic with the `duration` operator, you can create a simpler Stateflow chart to model the gear shifting.

See Also

`duration`

Related Examples

- “Reduce Transient Signals by Using Debouncing Logic” on page 27-12
- “Control Chart Execution by Using Temporal Logic” on page 14-35

Implement an Automatic Transmission Gear System by Using the duration Operator

This example models an automotive transmission system by using the Stateflow® temporal logic operator `duration` to automatically shift gears based on the vehicle throttle requirements and speed. For more information, see “Control Chart Execution by Using Temporal Logic” on page 14-35.

Model Description

There are five major blocks in this model.

- User Inputs: Provides two inputs to the model, brake and throttle.
- Engine: Calculates engine RPM based on impeller torque value and throttle.
- Gear_logic: Calculates next gear based on current gear, throttle, and current vehicle speed.
- Transmission: Calculates impeller and output torque based on RPM, gear and transmission speed.
- Vehicle: Calculates vehicle and transmission speed based on output torque and brake.

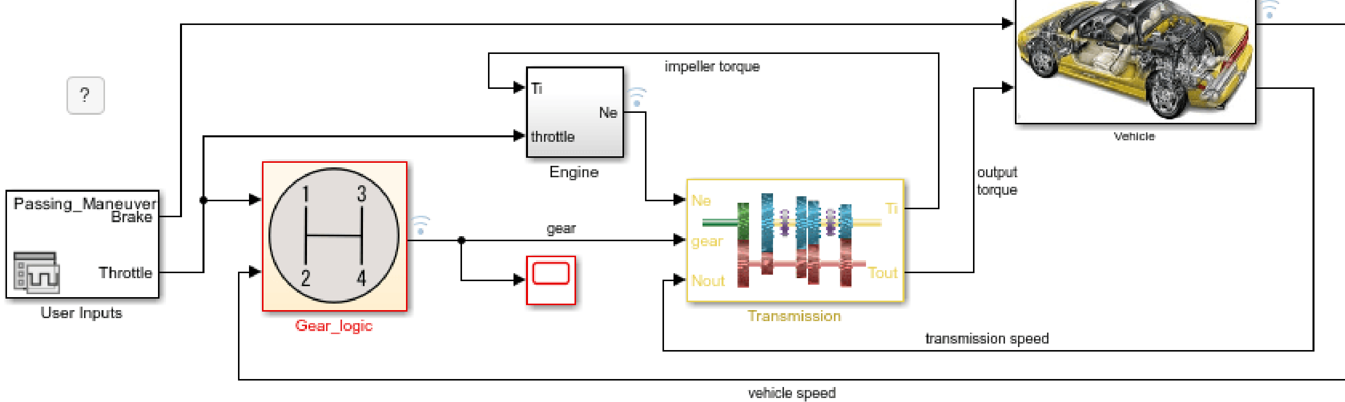
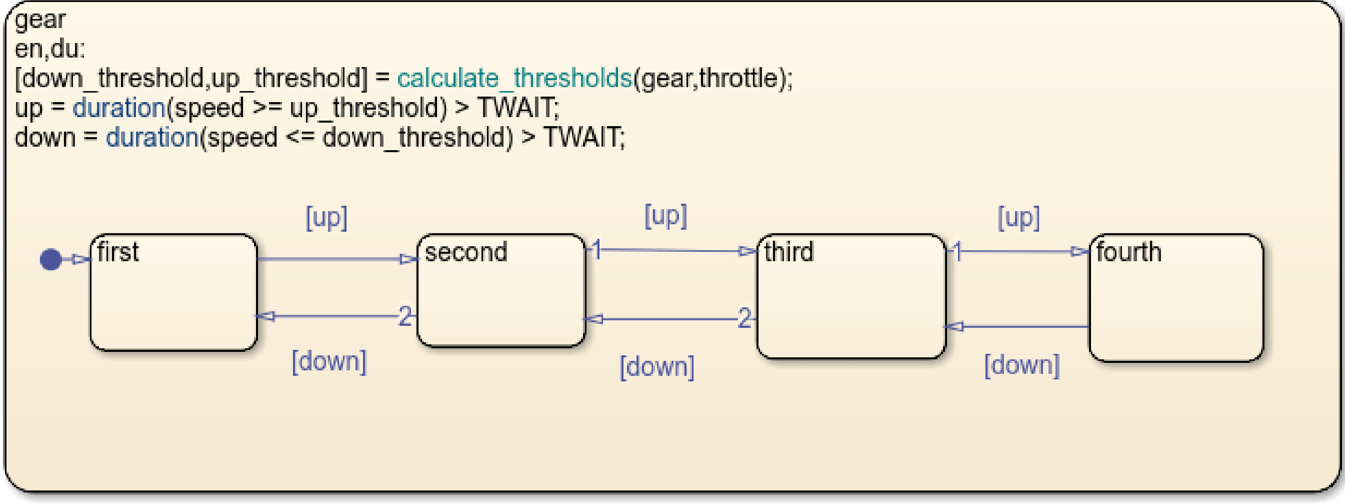


Chart Description

The Stateflow chart models the shifting of gears based on throttle and speed of the vehicle. The `down_threshold` and `up_threshold` outputs represent minimum and maximum speed values that throttle and current gear are able to handle. The Simulink function `calculate_thresholds` calculates these two values using `throttle` and `gear` as inputs. If the actual speed is higher than `up_threshold` for longer than `TWAIT`, then the chart transitions to higher gear. Conversely, if the actual speed is lower than `down_threshold` for longer than `TWAIT`, then the chart transitions to a lower gear. At each time step, the chart calls the `duration` operator to find the amount of time for which `speed` is higher than `up_threshold`. If this time exceeds `TWAIT` then boolean variable `up` is set which in turn transitions chart from current gear to a higher gear. Conversely the chart transitions to a lower gear based on the value of `down_threshold`.



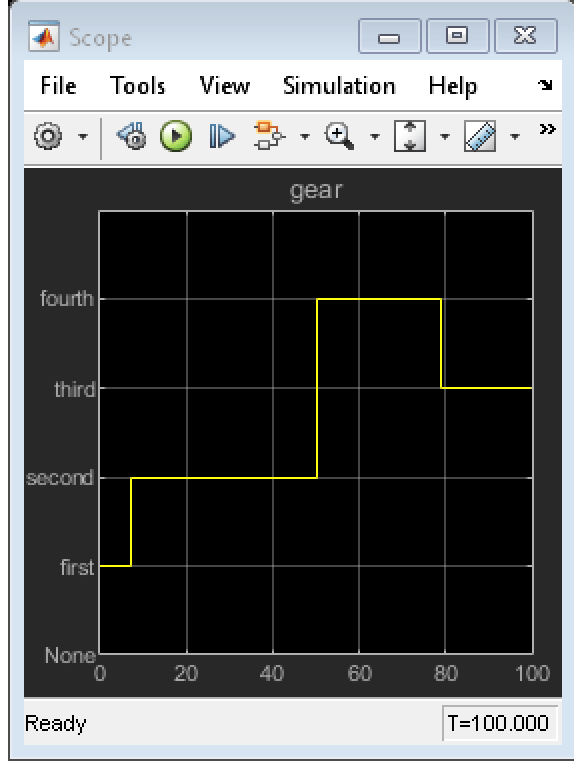
Simulink Function
[down_th,up_th] = calculate_thresholds(gear,throttle)

Active State Data

Active State Data is the enumerated data that represents the current active state during simulation. In this chart, the output data `gear` maintains the current active state which in turn represents the current gear. This data automatically updates when a transition is taken. The data is used by downstream blocks as well as by the Simulink® function `calculate_thresholds`. For more information, see “Monitor State Activity Through Active State Data” on page 11-2.

Simulation

To visualize these changes, simulate the model and open the scope.



See Also

`duration`

More About

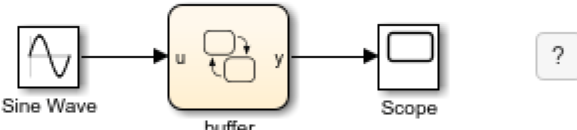
- “Control Chart Execution by Using Temporal Logic” on page 14-35
- “Control Oscillations by Using the duration Operator” on page 14-55
- “Monitor State Activity Through Active State Data” on page 11-2
- “Simplify Stateflow Charts by Incorporating Active State Output” on page 11-30

Count Events by Using the temporalCount Operator

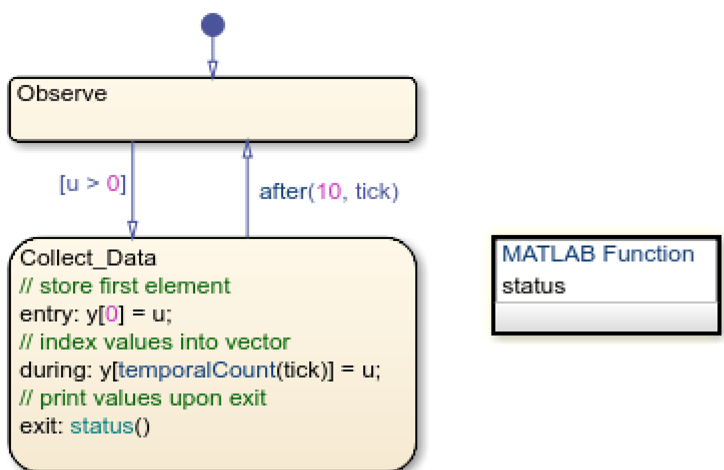
This example shows how to use the `temporalCount` operator to count occurrences of explicit and implicit events. For more information, see “Control Chart Execution by Using Temporal Logic” on page 14-35.

Collect and Store Input Data in a Vector

The Stateflow chart in this model collects and stores input data in a vector during simulation.



The chart contains two states and one MATLAB® function.



Simulate the Model

The execution of the chart consists of three stages.

Stage 1: Observation of Input Data

The chart wakes up 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

When 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` by using the `temporalCount` operator.

Stage 3: Display of Data Stored in the Vector

After the chart wakes up ten times, the data collection process ends. The chart calls the function `status` to display the vector data in the Diagnostic Viewer. Then, the chart takes the transition back to the state `Observe`.

See Also

`temporalCount`

More About

- “Control Chart Execution by Using Temporal Logic” on page 14-35
- “Control Chart Behavior by Using Implicit Events” on page 12-28
- “Vectors and Matrices in Stateflow Charts” on page 19-2

Detect Changes in Data and Expression Values

Stateflow charts can detect changes in the values of data and expressions between time steps. You can:

- Use change detection operators to determine when a variable changes to or from a value.
- Use edge detection operators to determine when an expression rises above or falls below a threshold.

To generate an implicit local event when the chart sets the value of a variable, use the `change` operator. For more information, see “Control Chart Behavior by Using Implicit Events” on page 12-28.

Change Detection Operators

To detect changes in Stateflow data, use the operators listed in this table.

Operator	Syntax	Description	Example
`hasChanged`	`tf = hasChanged(data_name)`	Returns 1 (`true`) if the value of `data_name` at the beginning of the current time step is different from the value of `data_name` at the beginning of the previous time step. Otherwise, the operator returns 0 (`false`).	Transition out of state if any element of the matrix `M` has changed value since the last time step or input event. `[hasChanged(M)]`

			Transition out of state if the element in row 1 and column 3 of the matrix `M` has changed value since the last time step or input event.
			In charts that use MATLAB as the action language, use:
			`[hasChanged(M(1,3))]`
			In charts that use C as the action language, use:
			`[hasChanged(M[0][2])]`

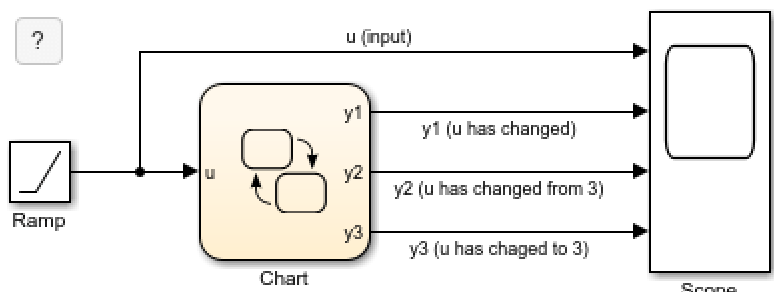
`hasChangedFrom`	`tf = hasChangedFrom(data_name,value)`	Returns 1 (`true`) if the value of `data_name` was equal to the specified `value` at the beginning of the previous time step and is a different value at the beginning of the current time step. Otherwise, the operator returns 0 (`false`).	Transition out of state if the previous value of the structure `struct` was equal to `structValue` and any field of `struct` has changed value since the last time step or input event. `[hasChangedFrom(struct,structValue)]`

Operator	Syntax	Description	Example
hasChangedTo	tf = hasChangedTo(data_name,value)	Returns 1 (true) if the value of data_name was not equal to the specified value at the beginning of the previous time step and is equal to value at the beginning of the current time step. Otherwise, the operator returns 0 (false).	Transition out of state if the structure field struct.field has changed to the value 5 since the last time step or input event. `[hasChangedTo(struct.field,5)]`

Note If multiple input events occur in the same time step, these operators can detect changes in data value between input events.

Example of Chart with Change Detection

This model shows how the operators hasChanged, hasChangedFrom, and hasChangedTo detect specific changes in an input signal. In this example, a Ramp (Simulink) 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 every time step. The chart analyzes the input signal u for these changes:

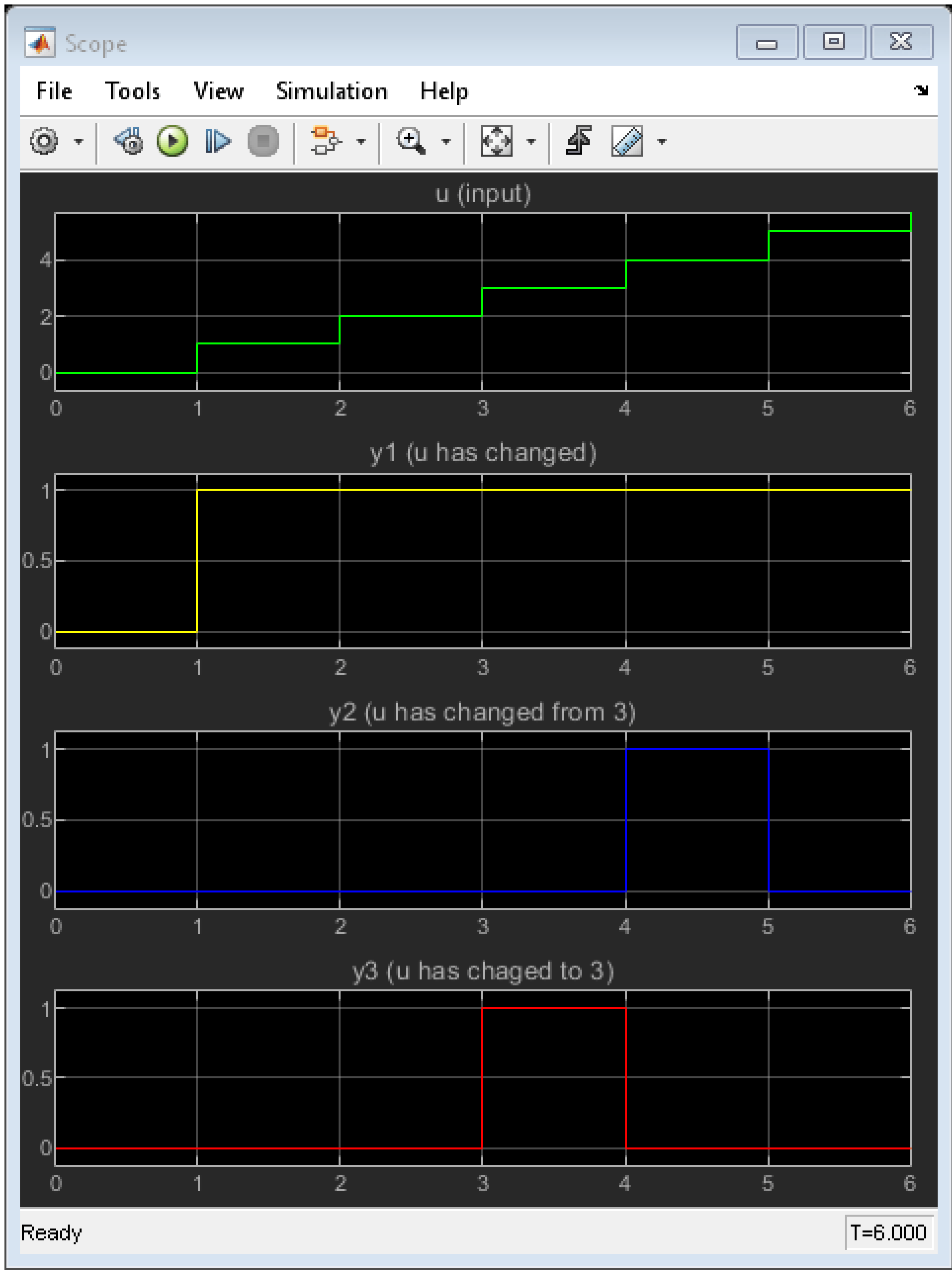
- Any change from the previous time step
- A change to the value 3
- A change from the value 3

To check the signal, the chart calls three change detection operators in a transition action. The chart outputs the return values as y1, y2, and y3.

```
{
y1 = hasChanged(u);
y2 = hasChangedFrom(u,3);
y3 = hasChangedTo(u,3);
}
```

During simulation, the Scope (Simulink) block shows the input and output signals for the chart.

- The value of `u` increases by 1 every time step.
- The value of `y1` changes from 0 to 1 at time `t = 1`. The value of `y1` remains 1 because `u` continues to change at each subsequent time step.
- The value of `y2` changes from 0 to 1 at time `t = 4` when the value of `u` changes from 3 to 4. The value of `y2` returns to 0 after one time step.
- The value of `y3` changes from 0 to 1 at time `t = 3` when the value of `u` changes from 2 to 3. The value of `y3` returns to 0 after one time step.



Limitations of Change Detection

The type of Stateflow chart determines the scope of the data supported for change detection:

- Standalone Stateflow charts in MATLAB: `Local` only
- Charts in Simulink that use MATLAB as the action language: `Input` only
- Charts in Simulink that use C as the action language: `Input`, `Output`, `Local`, or `Data Store Memory`

The argument `data_name` can be:

- A scalar variable.
- A matrix or an element of a matrix.
  - If `data_name` is a matrix, the operator returns `true` when it detects a change in any element of `data_name`.
  - Index elements of a matrix by using numbers or expressions that evaluate to a constant integer. See “Operations for Vectors and Matrices in Stateflow” on page 19-4.
- A structure or a field in a structure.
  - If `data_name` is a structure, the change detection operator returns `true` when it detects a change in any field of `data_name`.
  - Index fields in a structure by using dot notation. See “Index and Assign Values to Stateflow Structures” on page 26-7.
- Any valid combination of structure fields or matrix elements.

The argument `data_name` cannot be a nontrivial expression or a custom code variable.

Note Standalone charts in MATLAB do not support change detection on an element of a matrix or a field in a structure.

For the `hasChangedFrom` and `hasChangedTo` operators, the argument `value` can be any expression that resolves to a value that is comparable with `data_name`.

- If `data_name` is a scalar, then `value` must resolve to a scalar value.
- If `data_name` is a matrix, then `value` must resolve to a matrix value with the same dimensions as `data_name`.

  Alternatively, in a chart that uses C as the action language, `value` can resolve to a scalar value. The chart uses scalar expansion to compare `data_name` to a matrix whose elements are all equal to the value specified by `value`. See “Assign Values to All Elements of a Matrix” on page 19-6.
- If `data_name` is a structure, then `value` must resolve to a structure value whose field specification matches `data_name` exactly.

If you generate code from a chart that uses change detection operators and row-major array layout is enabled, code generation produces an error. Before generating code, enable column-major array layout. See “Select Array Layout for Matrices in Generated Code” on page 29-5.

Edge Detection Operators

To determine when an expression rises above or falls below a threshold, use the operators listed in this table.

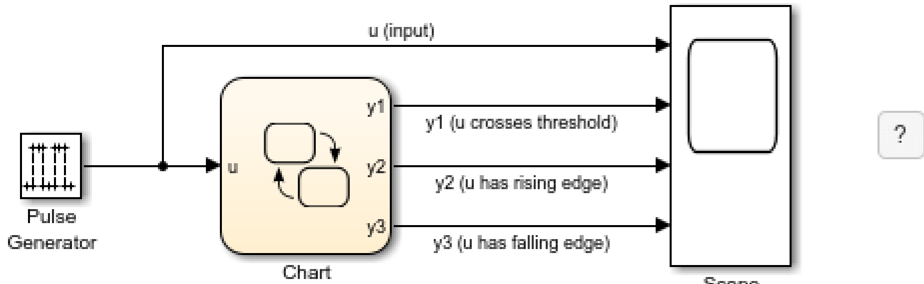
Operator	Syntax	Description	Example
`crossing`	`tf = crossing(expression)`	Returns 1 (`true`) if: • The previous value of `expression` was positive and its current value is zero or negative. • The previous value of `expression` was zero and its current value is nonzero. • The previous value of `expression` was negative and its current value is zero or positive. Otherwise, the operator returns 0 (`false`). This operator imitates the behavior of a Trigger block with **Trigger Type** set to `either`.	Transition out of state if the value of the input data `signal` crosses a threshold of 2.5. `[crossing(signal-2.5)]` The edge is detected when the value of the expression `signal-2.5` changes from positive to negative, from negative to positive, from zero to nonzero, or from nonzero to zero.
`falling`	`tf = falling(expression)`	Returns 1 (`true`) if: • The previous value of `expression` was positive and its current value is zero or negative. • The previous value of `expression` was zero and its current value is negative. Otherwise, the operator returns 0 (`false`). This operator imitates the behavior of a Trigger block with **Trigger Type** set to `falling`.	Transition out of state if the value of the input data `signal` falls below a threshold of 2.5. `[falling(signal-2.5)]` The falling edge is detected when the value of the expression `signal-2.5` changes from positive to negative, from positive to zero, or from zero to negative.

Operator	Syntax	Description	Example
rising	tf = rising(expression)	Returns 1 (true) if: • The previous value of expression was negative and its current value is zero or positive. • The previous value of expression was zero and its current value is positive. Otherwise, the operator returns 0 (false). This operator imitates the behavior of a Trigger block with **Trigger Type** set to rising.	Transition out of state if the value of the input data signal rises above a threshold of 2.5. [rising(signal-2.5)] The rising edge is detected when the value of the expression signal-2.5 changes from negative to positive, from negative to zero, or from zero to positive.

Note Like the Trigger block, these operators detect a single edge when the expression argument changes value from positive to zero to negative or from negative to zero to positive at three consecutive time steps. The edge occurs when the value of the expression becomes zero.

Example of Chart with Edge Detection

This model shows how the operators crossing, falling, and rising detect edges in an input signal. In this example, a Pulse Generator (Simulink) block sends a square wave to a chart.



The model uses a fixed-step solver with a step size of 1. The value of the input signal u alternates between 0 and 5 every two time steps. The chart analyzes the input signal u for these edges:

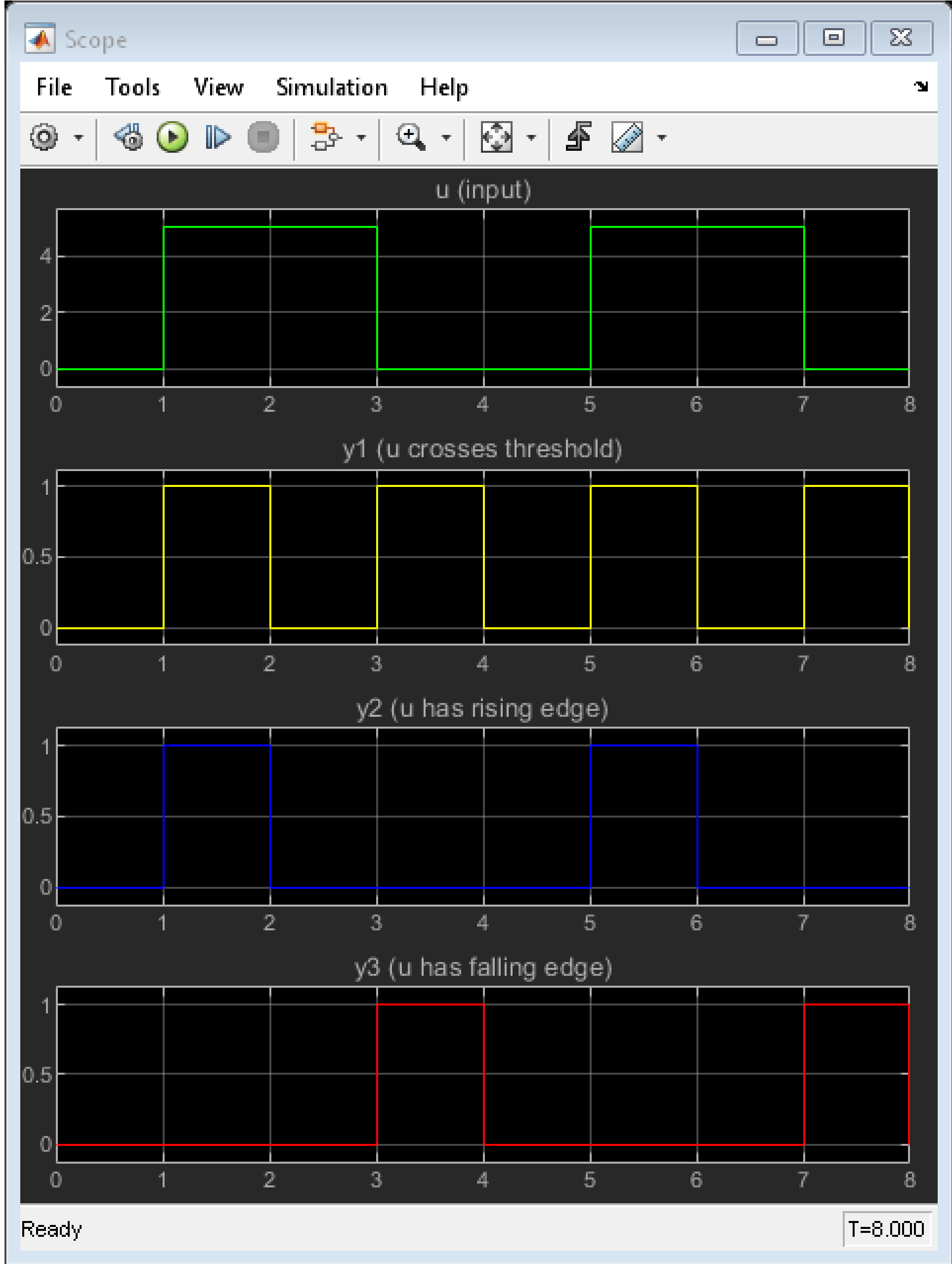
- A falling or rising edge crossing the threshold of 2.5
- An edge rising over the threshold of 2.5
- An edge falling under the threshold of 2.5

To check the signal, the chart calls three edge detection operators in a transition action. The chart outputs the return values as y1, y2, and y3.

```
{
y1 = crossing(u-2.5);
y2 = rising(u-2.5);
y3 = falling(u-2.5);
}
```

During simulation, the Scope (Simulink) block shows the input and output signals for the chart.

- The value of `u` alternates between 0 and 5 at every other time step.
- The value of `y1` changes from 0 to 1 at time `t = 1`, `3`, `5`, and `7`, when the value of the expression `u-2.5` changes sign. The value of `y1` returns to 0 after one time step.
- The value of `y2` changes from 0 to 1 at time `t = 1` and `5`, when the value of the expression `u-2.5` changes from negative to positive. The value of `y2` returns to 0 after one time step.
- The value of `y3` changes from 0 to 1 at time `t = 3` and `7`, when the value of the expression `u-2.5` changes from positive to negative. The value of `y3` returns to 0 after one time step.



Limitations of Edge Detection

Edge detection is supported only in Stateflow charts in Simulink models.

The argument `expression`:

- Must be a scalar-valued expression
- Can combine chart input data, constants, nontunable parameters, continuous-time local data, and state data from Simulink based states
- Can include addition, subtraction, and multiplication of scalar variables, elements of a matrix, fields in a structure, or any valid combination of structure fields and matrix elements

Index elements of a matrix by using numbers or expressions that evaluate to a constant integer.

Edge detection for continuous-time local data and state data from Simulink based states is supported only in transition conditions.

In atomic subcharts, map all input data that you use in edge detection expressions to input data or nontunable parameters in the main chart. Mapping these input data to output data, local data, or tunable parameters can result in undefined behavior.

Stateflow charts that use edge detection operators do not support operating points.

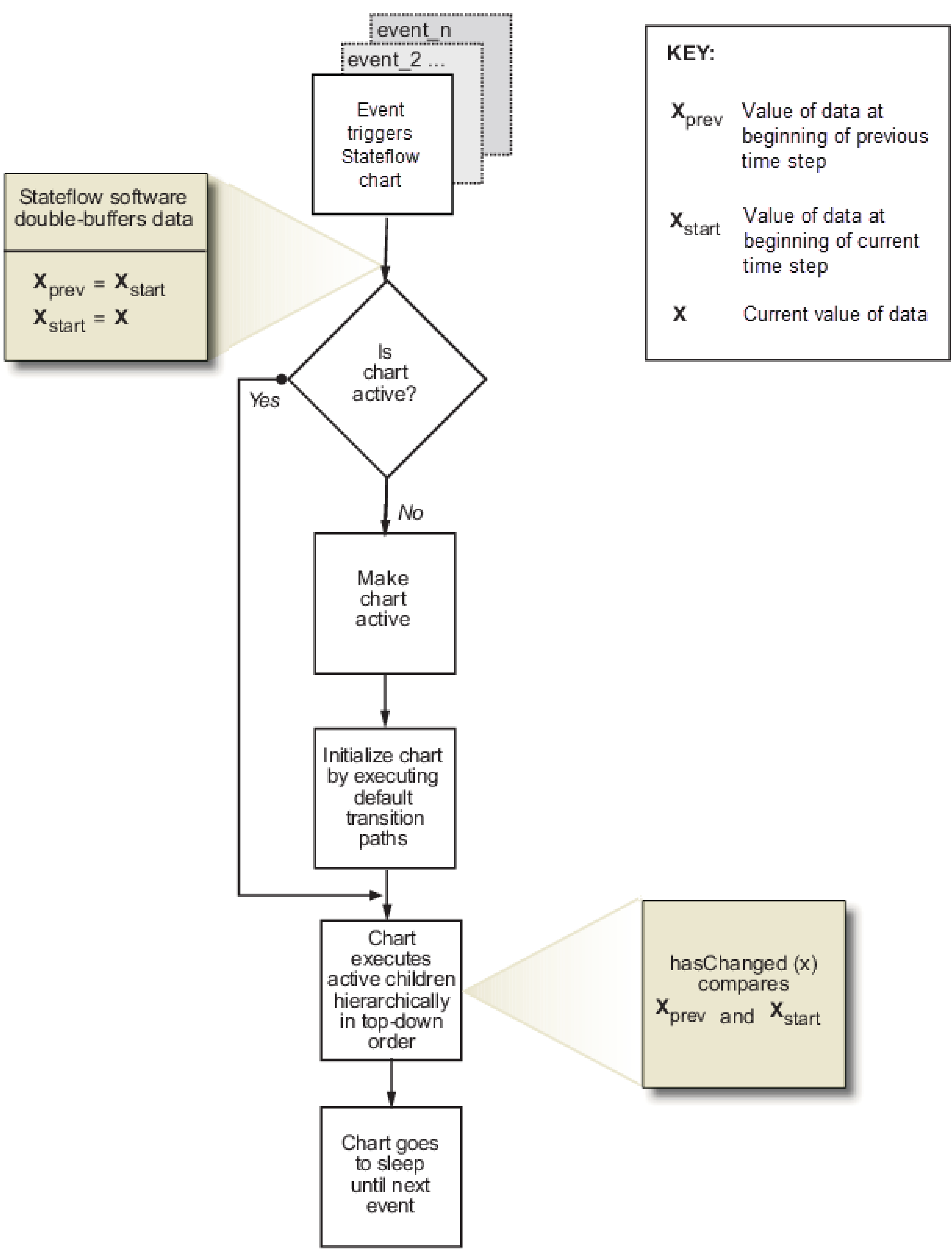
Implementation of Change and Edge Detection

A chart detects changes in chart data and expressions by evaluating the values at time step boundaries. The chart compares the value at the beginning of the previous execution step with the value at the beginning of the current execution step.

For example, when you invoke the `hasChanged` operator with an argument of `x`, the Stateflow chart double-buffers the values of `x` in local variables.

Local Buffer	Description
`x_prev`	Value of data `x` at the beginning of the last time step
`x_start`	Value of data `x` at the beginning of the current time step

To detect changes, the chart double-buffers data values *after* an event triggers the chart but *before* the chart begins execution. If the values of x_{prev} and x_{start} match, the change detection operator returns `false` to indicate that no change occurred; otherwise, it returns `true` to indicate a change. This diagram places these tasks in the context of the chart life cycle.


event_n
event_2 ...
Event triggers Stateflow chart
Stateflow software double-buffers data
$x_{prev} = x_{start}$
$x_{start} = x$
Is chart active?
Yes
No
Make chart active
Initialize chart by executing default transition paths
Chart executes active children hierarchically in top-down order
hasChanged (x) compares
x_{prev} and x_{start}
Chart goes to sleep until next event
KEY:
x_{prev} Value of data at beginning of previous time step
x_{start} Value of data at beginning of current time step
x Current value of data

Edge detection operators behave in a similar way, except that they compare the value of an expression at the beginning of the last time step (x_{prev}) with its current value (x). The difference in implementation allows continuous-time charts to detect edges in local data during minor time steps.

Transient Value Changes in Local Data

The change detection operators attempt to filter out transient changes in local chart variables by evaluating their values only at time boundaries. The chart evaluates the specified local variable only once at the end of the execution step. The return value of the change detection operators remains constant even if the value of the local variable fluctuates within a given time step. For example, suppose that in the current time step, the local variable `temp` changes from its value at the previous time step but then reverts to the original value. The operator `hasChanged(temp)` returns `false` for the next time step, indicating that no change occurred.

In contrast, the edge detection operators can detect edges in continuous-time local data during minor time steps. For example, suppose that `p` is a continuous-time local variable with a negative derivative. Then the operator `falling(p)` returns `true` during the minor time step when `p` changes sign from positive to negative.

Detect Value Changes Between Input Events or Super Step Iterations

When multiple input events occur in the same time step, or when you enable super step semantics, the chart updates the x_{prev} and x_{start} buffers every time it executes. The chart detects changes in value between input events and super step iterations even if the changes occur more than once in a given time step. For more information, see “Use Events to Execute Charts” on page 2-40 and “Super Step Semantics” on page 2-35.

See Also

`change` | `crossing` | `falling` | `hasChanged` | `hasChangedFrom` | `hasChangedTo` | `rising`

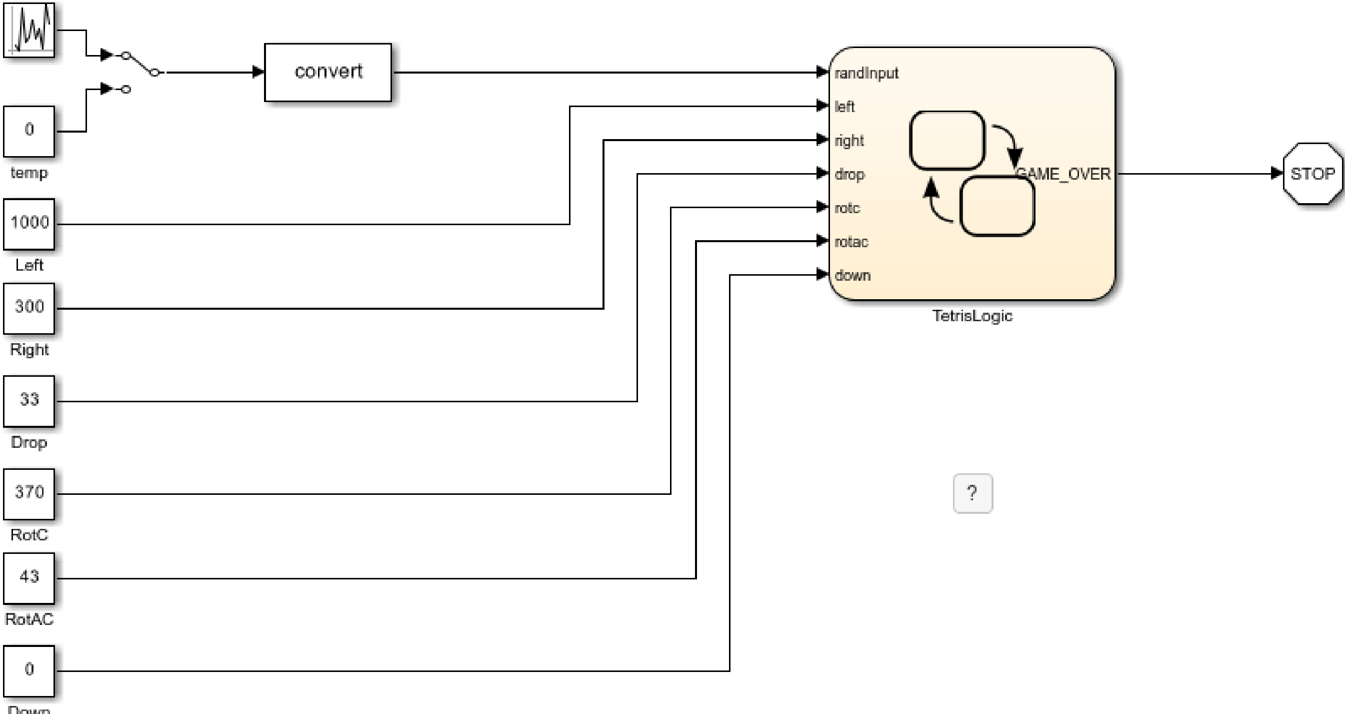
More About

- “Design a Game by Using Stateflow” on page 14-75
- “Control Chart Behavior by Using Implicit Events” on page 12-28
- “Use Events to Execute Charts” on page 2-40
- “Super Step Semantics” on page 2-35

Design a Game by Using Stateflow

This example shows how to implement the game of Tetris by using a Stateflow® chart. This model is a redesigned version of the classic Stateflow demo `sf_tetris`. The new design incorporates these programming paradigms:

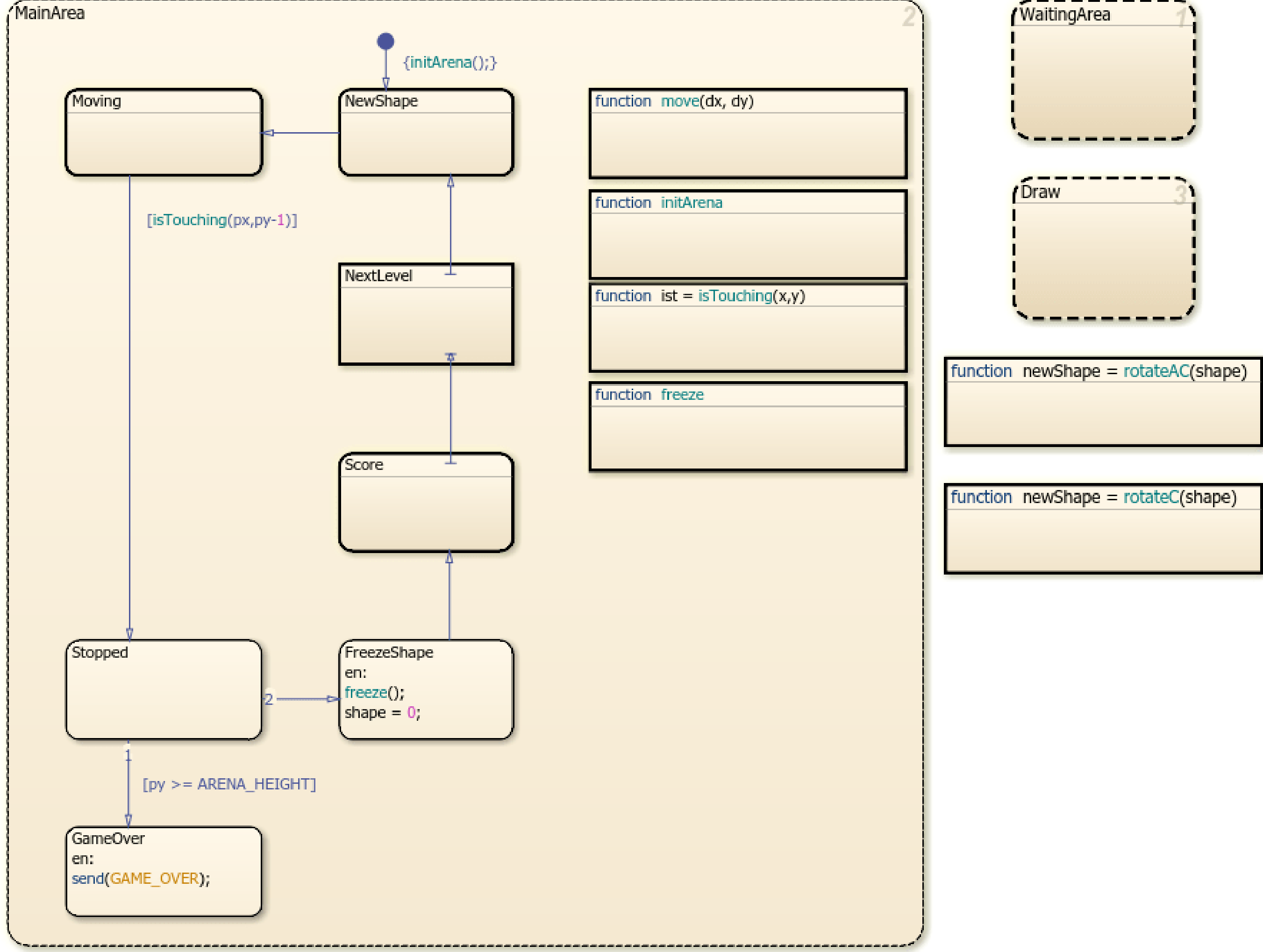
- Parallel decomposition to separate pre- and post-processing tasks from the main game control logic.
- State hierarchy and subcharts to provide semantic abstractions that simplify the design of the chart.
- Change detection operators to query for input from the keyboard.



Separate Subcomponents by Using Parallel Decomposition

The chart TetrisLogic implements the logic behind the game. The chart consists of three parallel states, which execute in this order:

- `WaitingArea` performs preprocessing tasks such as randomly generating the next *tetronimo* (a shape consisting of four squares). During simulation, the smaller square on the right of the game UI displays this tetronimo.
- `MainArea` implements the main control logic for the game. To represent the playing arena, this state uses a 21-by-12 array `arena`. At each simulation step, the chart updates the array based on the state of the game and the input from the player.
- `Draw` performs post-processing tasks such as calling the MATLAB® script `sf_tetris_gui`. This script displays the arena as an image and captures keystrokes from the player.



Simplify Chart Design by Using Hierarchy and Subcharts

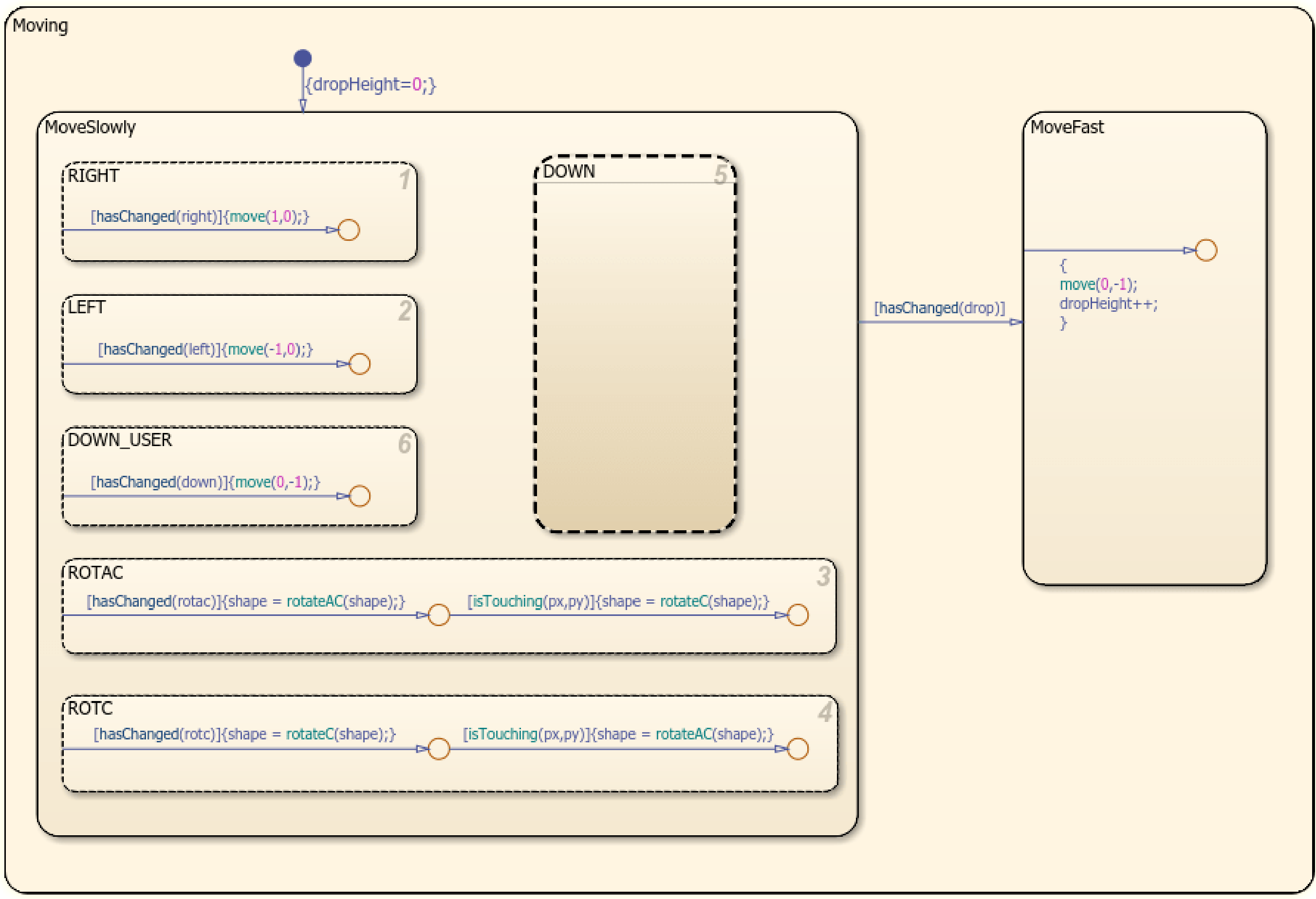
By using state hierarchy and subcharts, you can graphically abstract the game logic, present a high-level overview of the flow of the game, and hide the inner complexity of each stage of the game. For example, each substate of the parallel state `MainArea` represents a separate stage in the flow of the game.

- The game starts by generating a new tetronimo (substate `NewShape`).
- The tetronimo moves down or sideways, depending on the input from the player (substate `Moving`).
- When the tetronimo touches the bottom of the arena or an earlier tetronimo below it, the tetronimo stops moving (substate `Stopped`).
- If the tetronimo stops at too high a point on the arena, the game ends (substate `GameOver`). Otherwise, the chart freezes the tetronimo (substate `FreezeShape`), adjusts the score (substate `Score`), advances to the next level if necessary (substate `NextLevel`), and proceeds to the next tetronimo (substate `NewShape`).

Capture Keyboard Input Through Change Detection

The `Moving` subchart moves the tetronimo based on the input from the player. By default, the substate `MoveSlowly` is active. The tetronimo moves slowly down the playing arena while the parallel substates in `MoveSlowly` monitor the input from the keyboard. If the player presses the space bar, the substate `MoveFast` becomes active. The tetronimo drops quickly to the bottom of the arena.

To gather input from the keyboard, the subchart uses the change detection operator `hasChanged`. Every time that the player presses a key, `sf_tetris_gui` increments an input to the chart, which makes the corresponding `hasChanged` operator return a value of `true`. Because `MoveSlowly` has a parallel decomposition, the chart can process multiple keystrokes each time step.

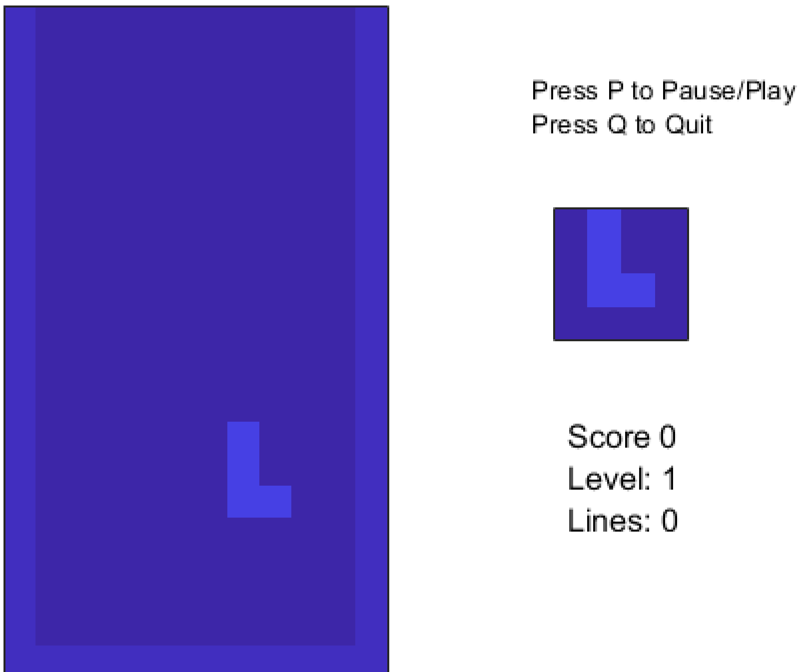


Key Mappings

To interact with the game UI, use these keys:

- Move left: **Left arrow** or **J**
- Move right: **Right arrow** or **L**
- Rotate clockwise: **Up arrow** or **I**
- Rotate counterclockwise: **Down arrow** or **K**

- Drop to bottom: **Space bar**
- To pause and resume play: **P**
- To quit: **Q**



See Also

`hasChanged`

More About

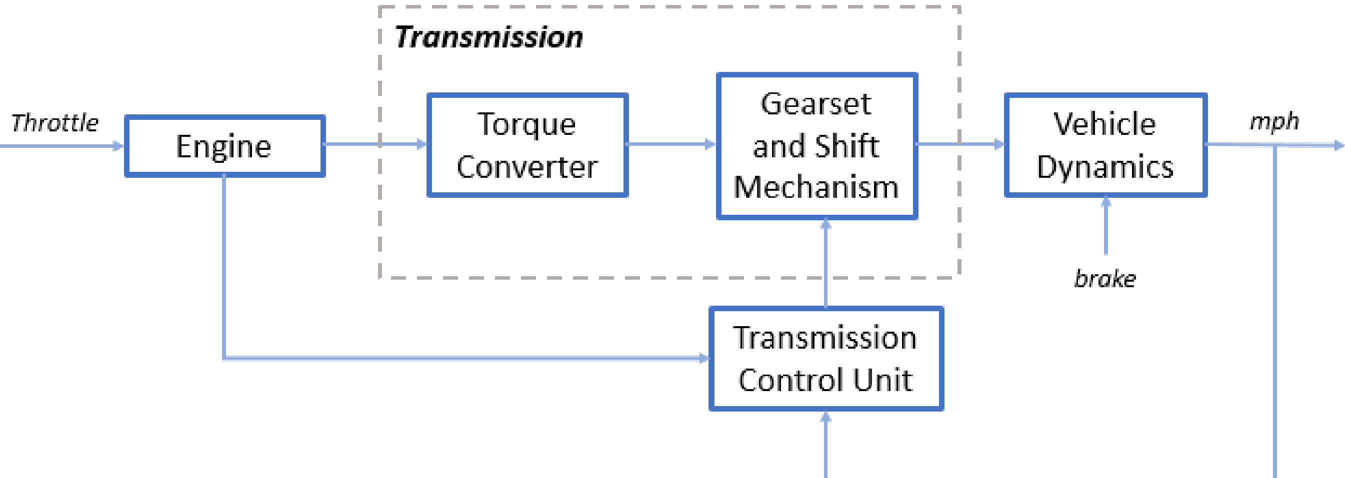
- "Define Exclusive and Parallel Modes by Using State Decomposition" on page 1-35
- "Use State Hierarchy to Design Multilevel State Complexity" on page 1-33
- "Encapsulate Modal Logic by Using Subcharts" on page 6-6
- "Detect Changes in Data and Expression Values" on page 14-63

Model an Automatic Transmission Controller

This example shows how to model an automotive drivetrain with Simulink®. Stateflow® enhances the Simulink model with its representation of the transmission control logic. Simulink provides a powerful environment for the modeling and simulation of dynamic systems and processes. In many systems, though, supervisory functions like changing modes or invoking new gain schedules must respond to events that may occur and conditions that develop over time. As a result, the environment requires a language capable of managing these multiple modes and developing conditions. In the following example, Stateflow shows its strength in this capacity by performing the function of gear selection in an automatic transmission. This function is combined with the drivetrain dynamics in a natural and intuitive manner by incorporating a Stateflow block in the Simulink block diagram.

Analysis and Physics

The figure below shows the power flow in a typical automotive drivetrain. Nonlinear ordinary differential equations model the engine, four-speed automatic transmission, and vehicle. The model discussed in this example directly implements the blocks from this figure as modular Simulink subsystems. On the other hand, the logic and decisions made in the Transmission Control Unit (TCU) do not lend themselves to well-formulated equations. TCU is better suited for a Stateflow representation. Stateflow monitors the events which correspond to important relationships within the system and takes the appropriate action as they occur.



The throttle opening is one of the inputs to the engine. The engine is connected to the impeller of the torque converter which couples it to the transmission (see Equation 1).

Equation 1

$$I_{ei}\dot{N}_e = T_e - T_i$$

$N_e =$ engine speed (RPM)

$I_{ei} =$ moment of inertia of the engine and the impeller

$T_e, T_i =$ engine and impeller torque

The input-output characteristics of the torque converter can be expressed as functions of the engine speed and the turbine speed. In this example, the direction of power flow is always assumed to be from the impeller to the turbine (see Equation 2).

Equation 2

$$T_i = \frac{N_e^2}{K^2}$$

$$K = f_2 \frac{N_{in}}{N_e} = \text{ K-factor (capacity)}$$

$$N_{in} = \text{ speed of turbine (torque converter output) = transmission input speed (RPM)}$$

$$R_{TQ} = f_3 \frac{N_{in}}{N_e} = \text{ torque ratio}$$

The transmission model is implemented via static gear ratios, assuming small shift times (see Equation 3).

Equation 3

$$R_{TR} = f_4(gear) = \text{ transmission ratio}$$

$$T_{out} = R_{TR} T_{in}$$

$$N_{in} = R_{TR} N_{out}$$

$$T_{in}, T_{out} = \text{ transmission input and output torques}$$

$$N_{in}, N_{out} = \text{ transmission input and output speed (RPM)}$$

The final drive, inertia, and a dynamically varying load constitute the vehicle dynamics (see Equation 4).

Equation 4

$$I_v \dot{N}_w = R_{fd}(T_{out} - T_{load})$$

$$I_v = \text{ vehicle inertia}$$

$$N_w = \text{ wheel speed (RPM)}$$

$$R_{fd} = \text{ final drive ratio}$$

$$T_{load} = f_5(N_w) = \text{ load torque}$$

The load torque includes both the road load and brake torque. The road load is the sum of frictional and aerodynamic losses (see Equation 5).

Equation 5

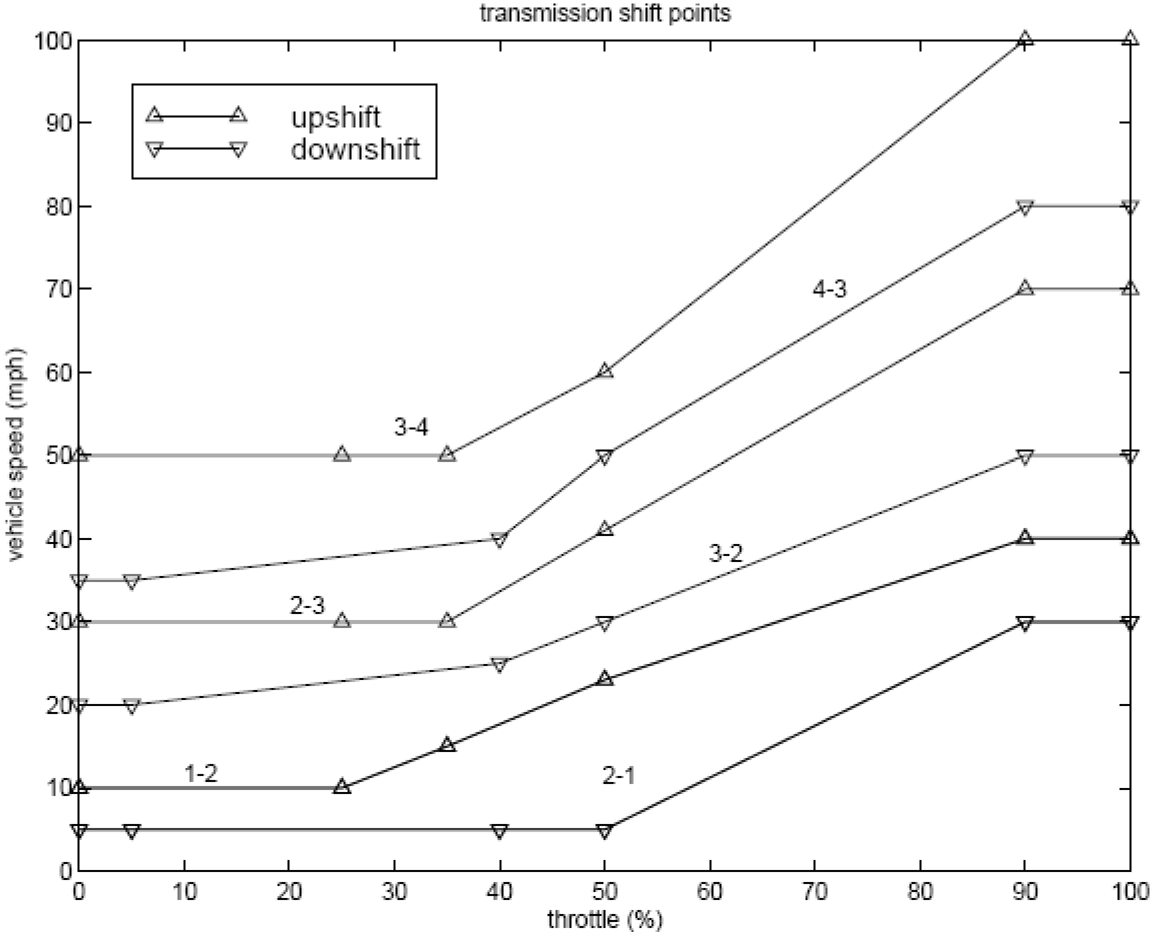
$$T_{load} = sgn(mph)(R_{load0} + R_{load2} mph^2 + T_{brake})$$

$R_{load0}, R_{load2} =$ friction and aerodynamic drag coefficients

$T_{load}, T_{brake} =$ load and brake torques

$mph =$ vehicle linear velocity

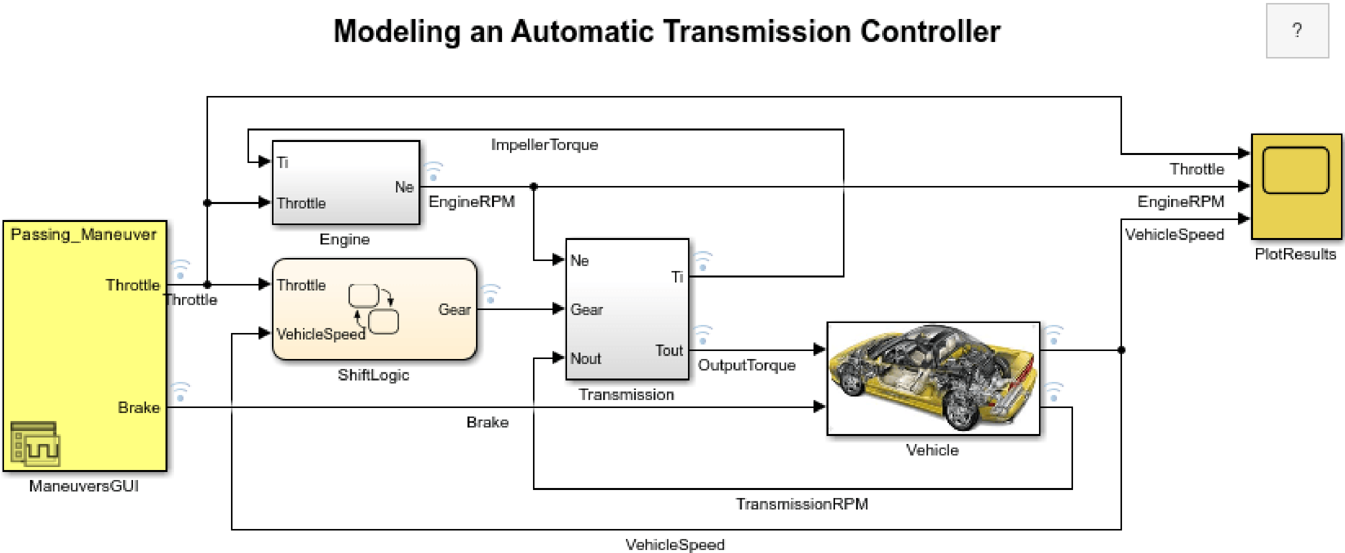
The model programs the shift points for the transmission according to the schedule shown in the figure below. For a given throttle in a given gear, there is a unique vehicle speed at which an upshift takes place. The simulation operates similarly for a downshift.



Modeling

When you open the model, the Initial conditions are set in the Model Workspace.

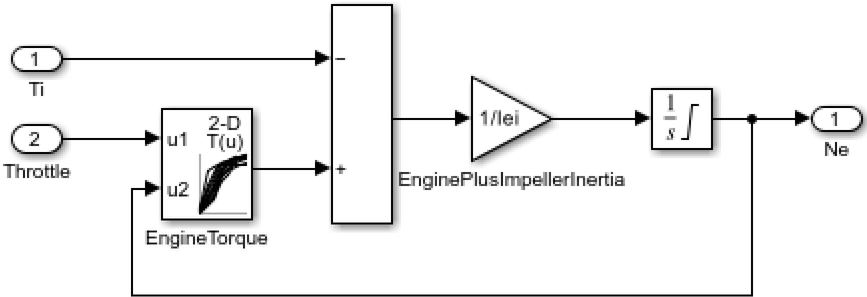
The top-level diagram of the model is shown in the figure below. To run the simulation, on the Simulation tab, click **Run**. Note that the model logs relevant data to MATLAB Workspace in a data structure called `sldemo_autotrans_output`. Logged signals have a blue indicator. After you run the simulation, you can view the components of the data structure by typing `sldemo_autotrans_output` in MATLAB Command Window. Also note that the units appear on the subsystem icons and signal lines.



Modeling

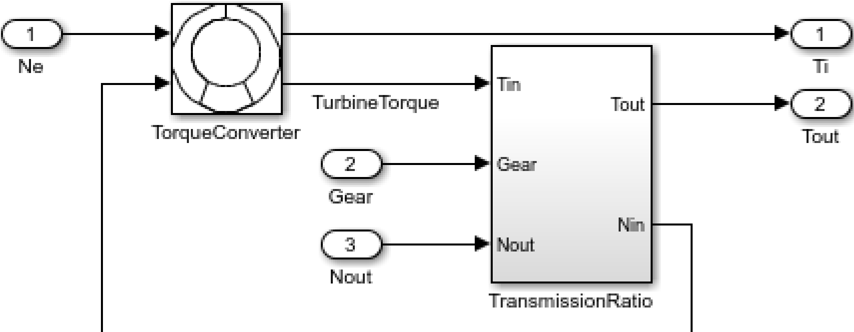
The Simulink model shown above is composed of modules which represent the engine, transmission, and the vehicle, with an additional shift logic block to control the transmission ratio. User inputs to the model are in the form of throttle (given in percent) and brake torque (given in ft-lb). The user inputs throttle and brake torques using the ManeuversGUI interface.

The Engine subsystem consists of a two-dimensional table that interpolates engine torque versus throttle and engine speed. The figure below shows the composite Engine subsystem. Double click on this subsystem in the model to view its structure.



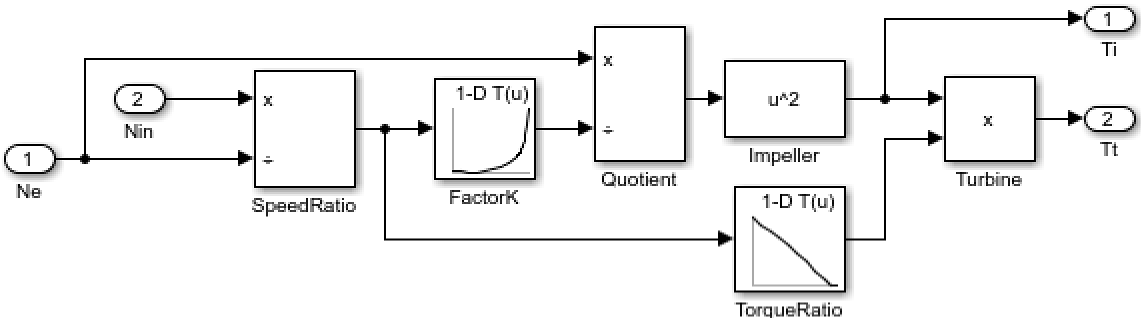
Engine

The TorqueConverter and the TransmissionRatio blocks make up the Transmission subsystem, as shown in the figure below. Double click on the Transmission subsystem in the model window to view its components.



Transmission

The TorqueConverter is a masked subsystem, which implements Equation 2. To open this subsystem, right click on it and select **Mask** > **Look Under Mask** from the drop-down menu. The mask requires a vector of speed ratios (`Nin/Ne`) and vectors of K-factor (`f2`) and torque ratio (`f3`). This figure shows the implementation of the TorqueConverter subsystem.

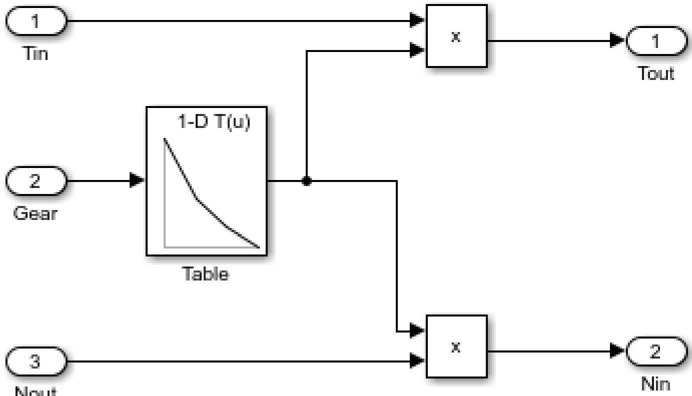


Torque Converter

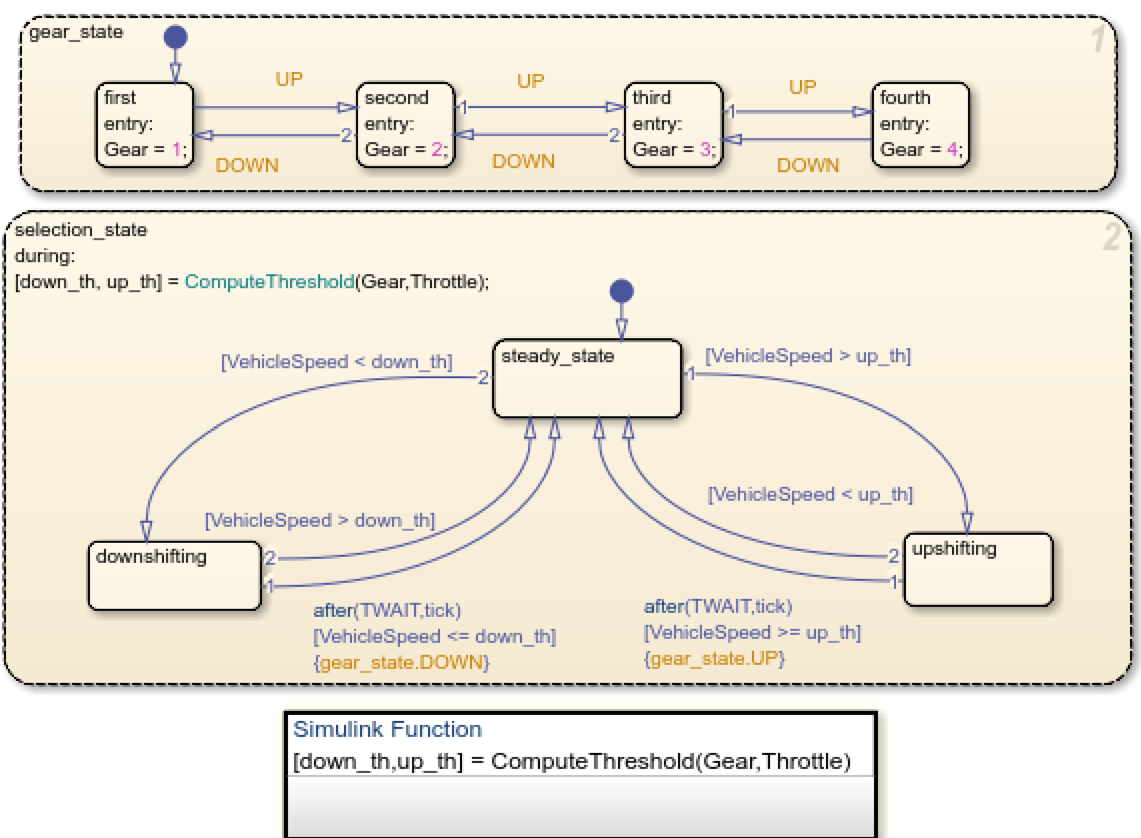
The transmission ratio block determines the ratio shown in Table 1 and computes the transmission output torque and input speed, as indicated in Equation 3. The figure that follows shows the block diagram for the subsystem that realizes this ratio in torque and speed.

Table 1: Transmission gear ratios

```
gear     Rtr = Nin/Ne
 1         2.393
 2         1.450
 3         1.000
 4         0.677
```



The Stateflow block labeled ShiftLogic implements gear selection for the transmission. Double click on ShiftLogic in the model window to open the Stateflow diagram. The Model Explorer is utilized to define the inputs as throttle and vehicle speed and the output as the desired gear number. Two dashed AND states keep track of the gear state and the state of the gear selection process. The overall chart is executed as a discrete-time system, sampled every 40 milliseconds. The Stateflow diagram shown below illustrates the functionality of the block.

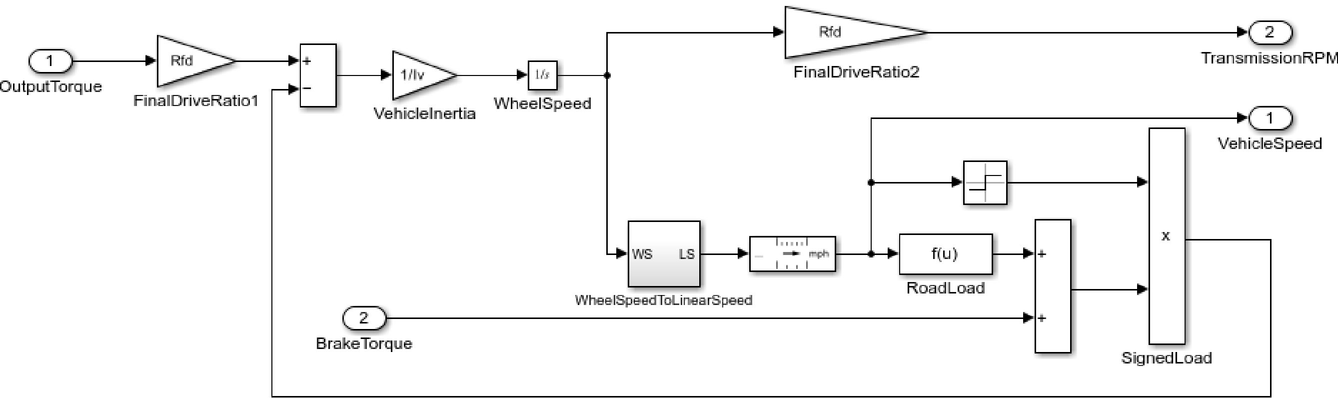


The shift logic behavior can be observed during simulation by enabling animation in the Stateflow debugger. The `selection_state` (always active) begins by performing the computations indicated in its `during` function. The model computes the upshift and downshift speed thresholds as a function of the instantaneous values of gear and throttle. While in steady_state, the model compares these values to the present vehicle speed to determine if a shift is required. If so, it enters one of the confirm states (`upshifting` or `downshifting`), which records the time of entry.

If the vehicle speed no longer satisfies the shift condition, while in the confirm state, the model ignores the shift and it transitions back to `steady_state`. This prevents extraneous shifts due to noise conditions. If the shift condition remains valid for a duration of `TWAIT` ticks, the model transitions through the lower junction and, depending on the current gear, it broadcasts one of the shift events. Subsequently, the model again activates `steady_state` after a transition through one of the central junctions. The shift event, which is broadcast to the `gear_selection` state, activates a transition to the appropriate new gear.

For example, if the vehicle is moving along in second gear with 25% throttle, the state `second` is active within `gear_state`, and `steady_state` is active in the `selection_state`. The `during` function of the latter, finds that an upshift should take place when the vehicle exceeds 30 mph. At the moment this becomes true, the model enters the `upshifting` state. While in this state, if the vehicle speed remains above 30 mph for `TWAIT` ticks, the model satisfies the transition condition leading down to the lower right junction. This also satisfies the condition [|gear == 2|] on the transition leading from here to `steady_state`, so the model now takes the overall transition from `upshifting` to `steady_state` and broadcasts the event `UP` as a transition action. Consequently, the transition from second to third is taken in `gear_state` which completes the shift logic.

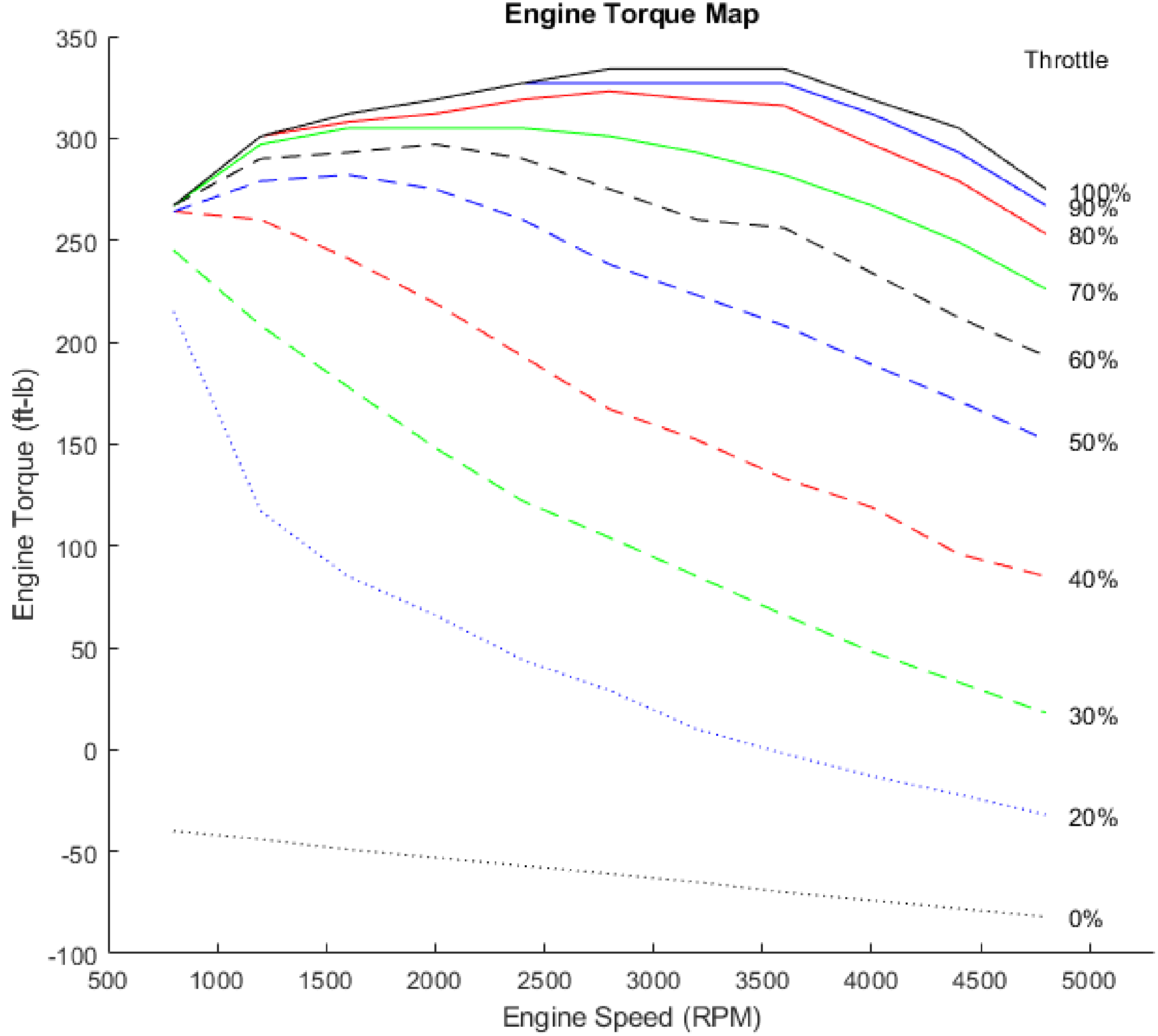
The Vehicle subsystem uses the net torque to compute the acceleration and integrate it to compute the vehicle speed, per Equation 4 and Equation 5. The Vehicle subsystem is masked. To see the structure of the Vehicle block, right click on it and select **Mask > Look Under Mask** from the drop-down menu. The parameters entered in the mask menu are the final drive ratio, the polynomial coefficients for drag friction and aerodynamic drag, the wheel radius, vehicle inertia, and initial transmission output speed.



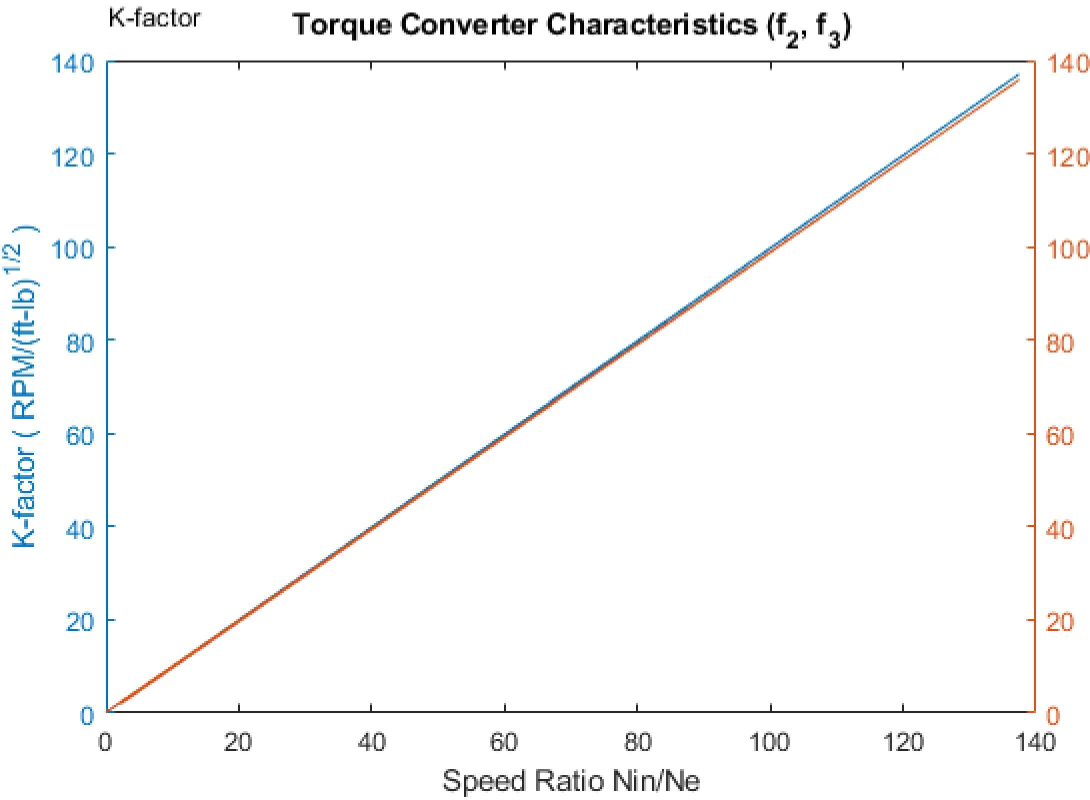
Vehicle

Results

The engine torque map, and torque converter characteristics used in the simulations are shown below.



Get the FactorK (second row) and the TorqueRatio (third row) vs SpeedRatio(first row)

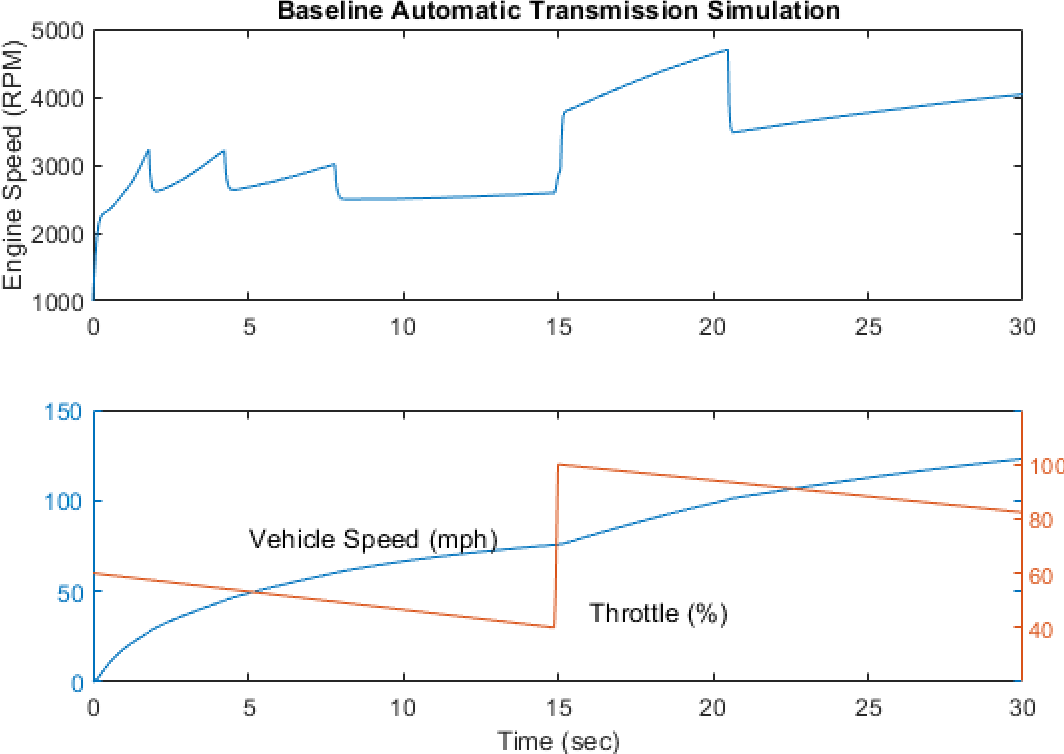


The first simulation (passing maneuver) uses the throttle schedule given in Table 2 (this data is interpolated linearly).

Table 2: Throttle schedule for first simulation (passing maneuver)

```
Time (sec)    Throttle (%)
  0             60
 14.9           40
 15            100
100              0
200              0
```

The first column corresponds to time; the second column corresponds to throttle opening in percent. In this case no brake is applied (brake torque is zero). The vehicle speed starts at zero and the engine at 1000 RPM. The following figure shows the plot for the baseline results, using the default parameters. As the driver steps to 60% throttle at `t=0`, the engine immediately responds by more than doubling its speed. This brings about a low speed ratio across the torque converter and, hence, a large torque ratio. The vehicle accelerates quickly (no tire slip is modeled) and both the engine and the vehicle gain speed until about `t = 2 sec`, at which time a 1-2 upshift occurs. The engine speed characteristically drops abruptly, then resumes its acceleration. The 2-3 and 3-4 upshifts take place at about four and eight seconds, respectively. Notice that the vehicle speed remains much smoother due to its large inertia.



At `t=15sec`, the driver steps the throttle to 100% as might be typical of a passing maneuver. The transmission downshifts to third gear and the engine jumps from about 2600 RPM to about 3700 RPM. The engine torque thus increases somewhat, as well as the mechanical advantage of the transmission. With continued heavy throttle, the vehicle accelerates to about 100 mph and then shifts into overdrive at about `t = 21 sec`. The vehicle cruises along in fourth gear for the remainder of the simulation. Double click on the ManeuversGUI block and use the graphical interface to vary the throttle and brake history.

Running Multiple Scenarios and Collecting Coverage

You can run the model for all scenarios while collecting coverage. To see a saved design study for running all of the scenarios of sldemo_autotrans, open `sldemo_autotrans_design_study.mldatx` in the Multiple Simulations panel of sldemo_autotrans.

After the design study has been opened, enable the model coverage and cumulative collection in the model settings.

```
set_param('sldemo_autotrans', 'CovEnable', 'on');

set_param('sldemo_autotrans', 'CovEnableCumulative', 'on');
```

Once this is set, click the `Run All (Coverage)` button on the Simulation tab on the Simulink toolstrip. Then check model coverage of the design cases.

Closing the Model

Close the model, clear generated data.

Conclusions

You can enhance this basic system in a modular manner, for example, by replacing the engine or transmission with a more complex model. You can build large systems within this structure via step-wise refinement. The seamless integration of Stateflow control logic with Simulink signal processing enables the construction of a model that is efficient and visually intuitive.

See Also

More About

- “Unit Specification in Simulink Models” (Simulink)
- “Powertrain Blockset”

Vehicle Electrical and Climate Control Systems

This example shows how to interface the vehicle climate control system with a model of the electrical system to examine the loading effects of the climate control system on the entire electrical system of the car.

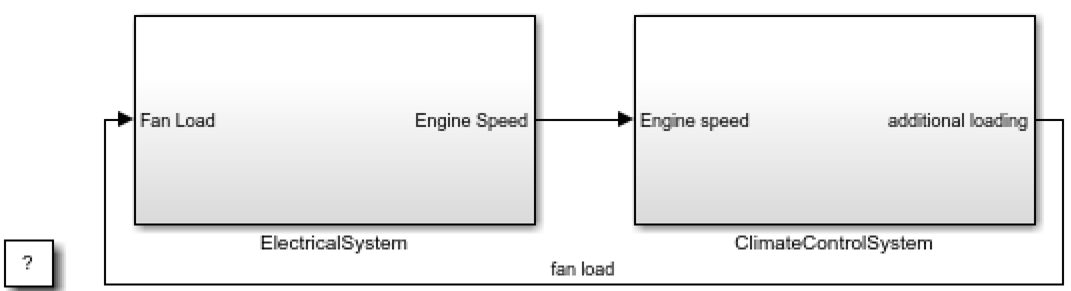


Figure 1: Vehicle Electrical and Climate Control System

The Climate Control System

Double clicking on the ClimateControlSystem subsystem will open the model of the climate control system. Here the user can enter the temperature value they would like the air in the car to reach by double clicking on the USER SETPOINT IN CELSIUS Block and entering the value into the dialog box. The EXTERNAL TEMPERATURE IN CELSIUS can also be set by the user in a similar way. The numerical display on the right hand side of the model shows the reading of a temperature sensor placed behind the driver's head. This is the temperature that the driver should be feeling. When the model is run and the climate control is active, it is this display box whose value changes showing the change of temperature in the car.

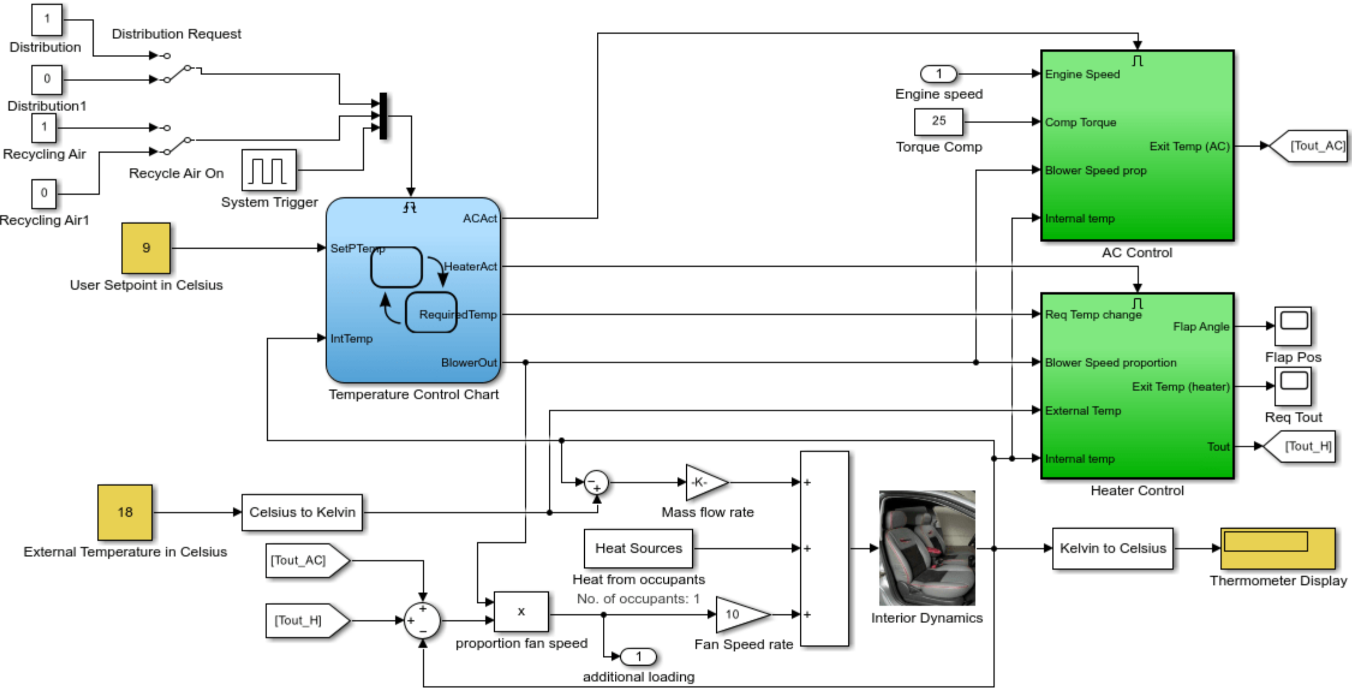


Figure 2: The automatic climate control system.

The Stateflow® Controller

The control of the system is implemented in Stateflow®. Double clicking on the Stateflow chart will show how this supervisory control logic has been formulated.

The **Heater_AC** state shows that when the user enters a setpoint temperature which greater than the current temperature in the car by at least 0.5 deg C, the heater system will be switched on. The heater will remain active until the current temperature in the car reaches to within 0.5 deg of the setpoint temperature. Similarly, when the user enters a setpoint which is 0.5 deg C (or more) lower than the current car temperature, the Air Conditioner is turned on and stays active until the temperature of the air in the car reaches to within 0.5 deg C of the setpoint temperature. After which, the system will switch off. The dead band of 0.5 deg has been implemented to avoid the problem of continuous switching.

In the **Blower** State, the larger the difference between the setpoint temperature and the current temperature, the harder the fan blows. This ensures that the temperature will reach the required value in a reasonable amount of time, despite the temperature difference. Once again, when the temperature of the air in the car reaches to within 0.5 deg C of the setpoint temperature, the system will switch off.

The Air Distribution(**AirDist**) and Recycling Air States(**Recyc_Air**) are controlled by the two switches that trigger the Stateflow chart. An internal transition has been implemented within these two states to facilitate effective defrosting of the windows when required. When the defrost state is activated, the recycling air is turned off.

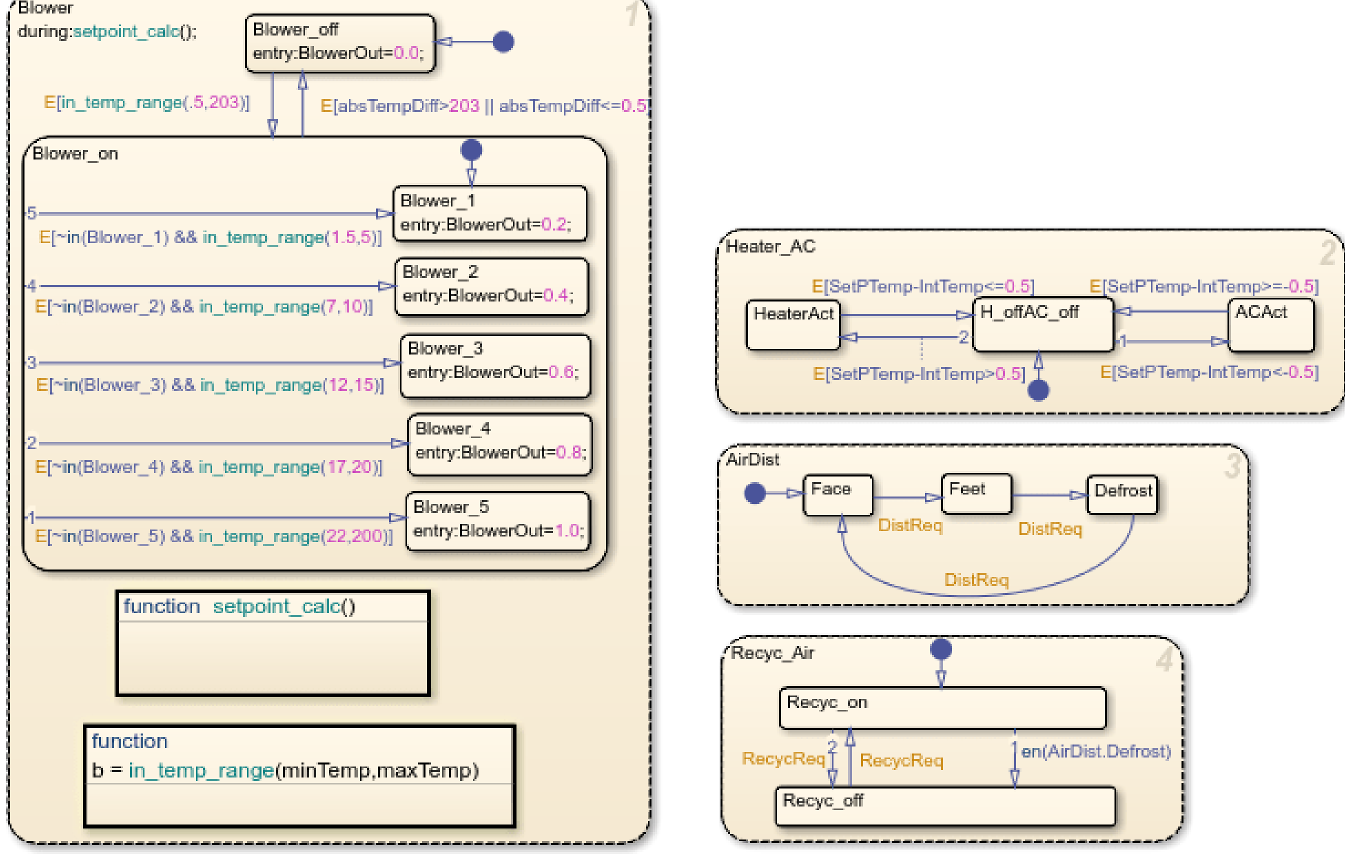


Figure 3: The supervisory control logic in Stateflow.

Heater and Air Conditioner Models

The heater model was built from the equation for a heat exchanger shown below:

```
Tout = Ts - (Ts-Tin)e^[(-pi*D*L*hc)/(m_dot*Cp)]
```

Where:

- Ts = constant (radiator wall temperature)
- D = 0.004m (channel diameter)
- L = 0.05m (radiator thickness)
- N = 30000 (Number of channels)
- k = 0.026 W/mK = constant (thermal conductivity of air)
- Cp = 1007 J/kgK = constant (specific heat of air)
- Laminar flow (hc = 3.66(k/D) = 23.8 W/m2K)

In addition, the effect of the heater flap is taken into account. Similar to the operation of the blower, the greater the temperature difference between the required setpoint temperature and the current temperature in the car, the more the heater flap is opened and the greater the heating effect.

The Air Conditioner system is one of the two places where the climate control model interfaces with the car's electrical system model. The compressor loads the engine of the car when the A/C system is active. The final temperature to exit from the A/C is calculated as follows:

y*(w*Tcomp) = m_dot*(h4-h1)

Where:

- y = efficiency
- m_dot = mass flow rate
- w = speed of the engine
- Tcomp = compressor torque
- h4, h1 = enthalpy

Here we have bang-bang control of the A/C system where the temperature of the air that exits the A/C is determined by the engine speed and compressor torque.

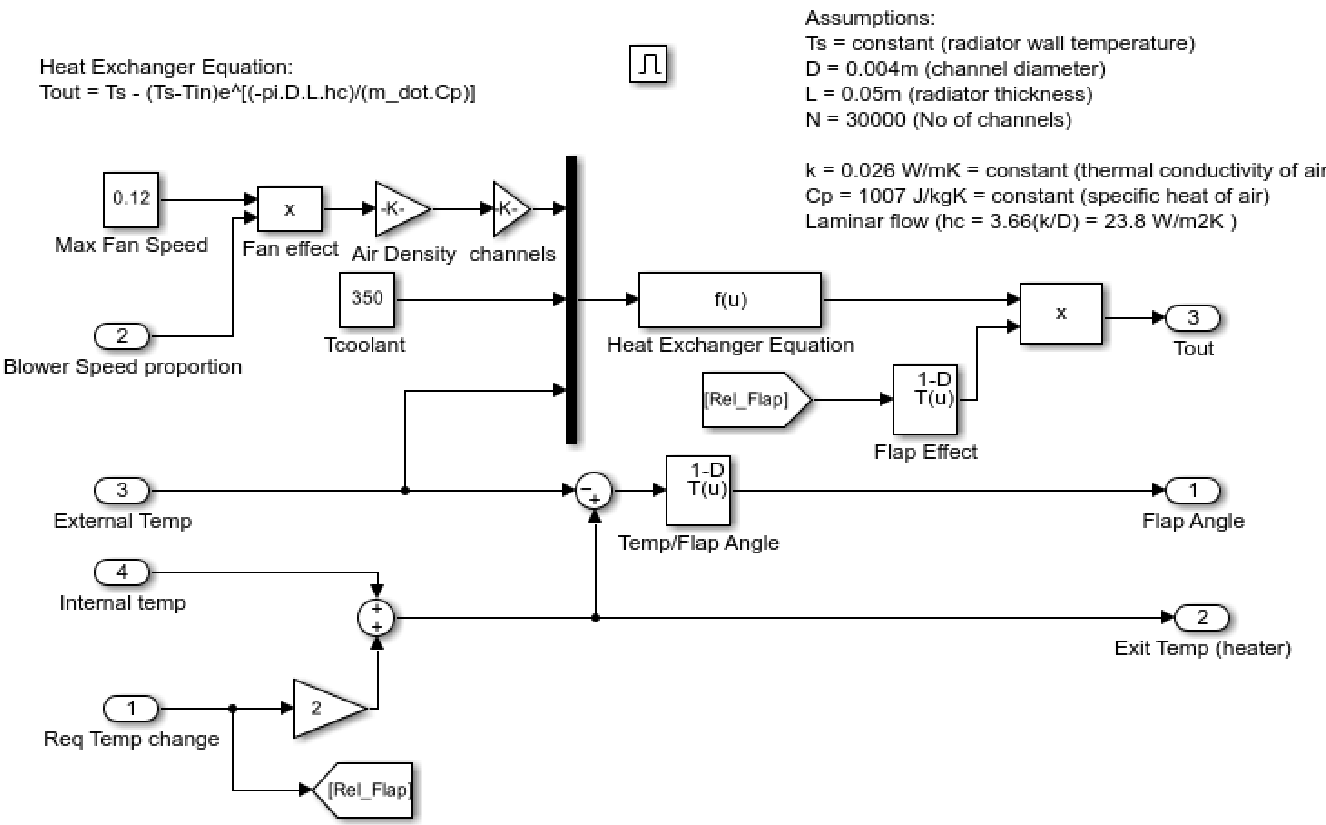


Figure 4: Heater control subsystem.

Heat Transfer Equation (from evaporator):

y.(w.Tcomp) = m_dot.(h4-h1)

y = efficiency
m_dot = mass flow rate
w = speed of the engine
Tcomp = compressor torque
h4, h1 = enthalpy

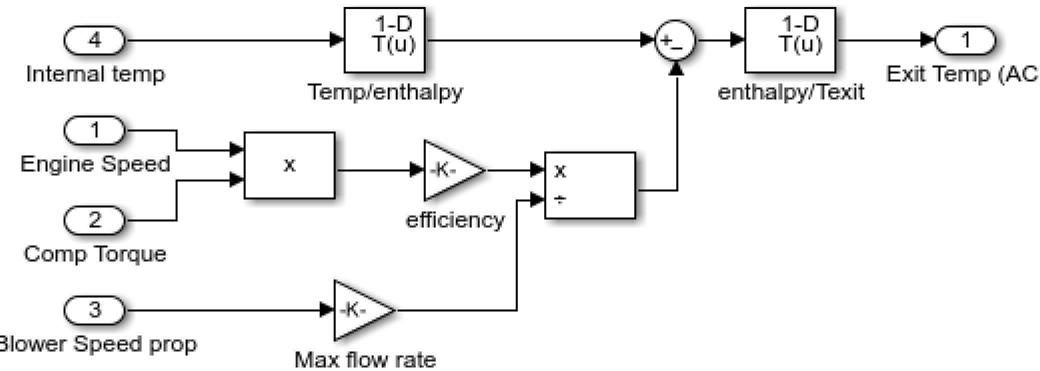


Figure 5: A/C control subsystem.

Heat Transfer in the Cabin

The temperature of the air felt by the driver is affected by all of these factors:

- The temperature of the air exiting the vents
- The temperature of the outside air
- The number of people in the car

These factors are inputs into the thermodynamic model of the interior of the cabin. We take into account the temperature of the air exiting the vents by calculating the difference between the vent air temperature and the current temperature inside the car and multiplying it by the fan speed proportion (mass flow rate). Then 100W of energy is added per person in the car. Lastly, the difference between the temperature of the outside air and the interior air temperature is multiplied by a lesser mass flow rate to account for the air radiating into the car from the outside.

The output of the interior dynamics model is fed to the display block as a measure of the temperature read by a sensor placed behind the driver's head.

The Electrical System

This electrical system models the car at idle speed. The PID controllers ensure that the car's alternator (modeled by a synchronous machine which has its field current regulated to control the output voltage) is also operating at the required speed. The alternator output is then fed through a 3-phase 6-pulse rectifier bridge to supply the voltage needed to charge the battery which supplies the voltage for the car's DC bus.

The fan used in the climate control system is fed off this DC bus as are the windscreen wipers, radio etc. As the difference between the setpoint temperature and the current temperature in the car drops, so does the fan speed and therefore so does the loading on the DC bus. The inclusion of feedback in the electrical system regulates the DC bus voltage.

The additional model of the car's electrical system allows for the changing of the engine speed. Changing the engine speed shows the effect on the DC bus voltage.

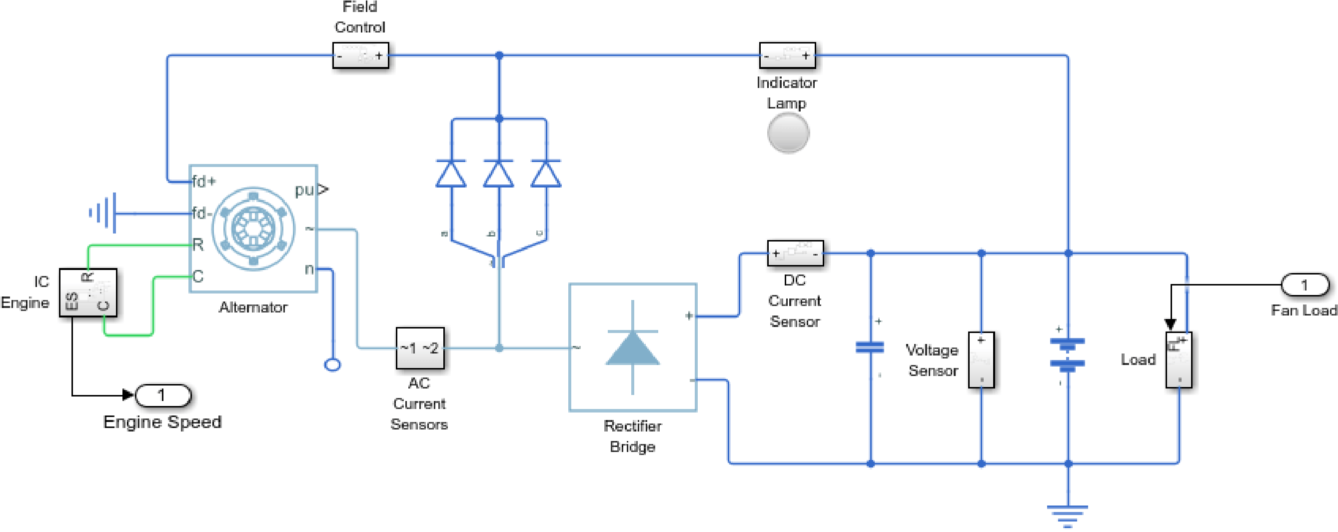


Figure 6: The electrical system

Developing the Apollo Lunar Module Digital Autopilot

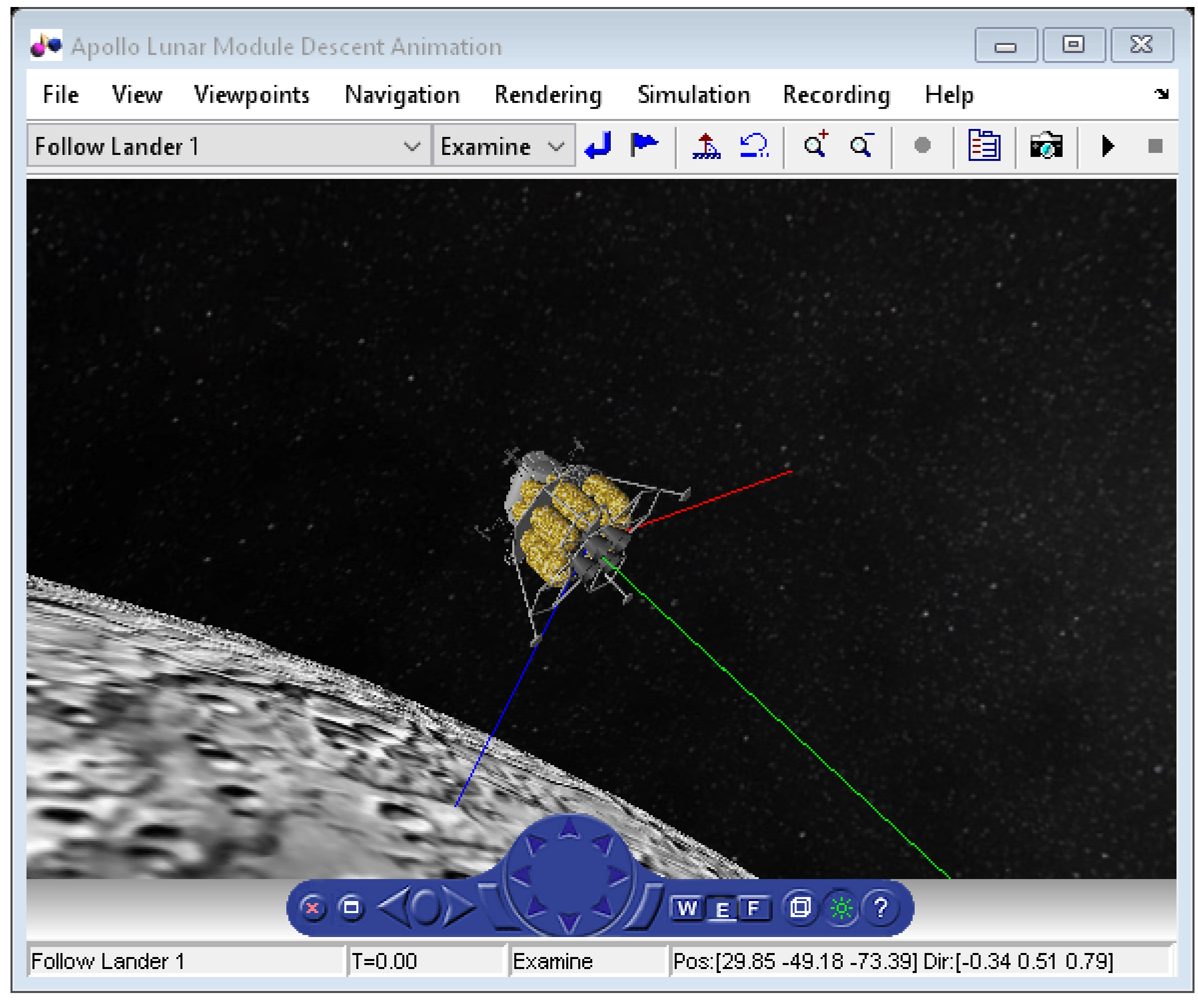
"Working on the design of the Lunar Module digital autopilot was the highlight of my career as an engineer. When Neil Armstrong stepped off the LM (Lunar Module) onto the moon's surface, every engineer who contributed to the Apollo program felt a sense of pride and accomplishment. We had succeeded in our goal. We had developed technology that never existed before, and through hard work and meticulous attention to detail, we had created a system that worked flawlessly." -Richard J. Gran, *The Apollo 11 Moon Landing: Spacecraft Design Then and Now*

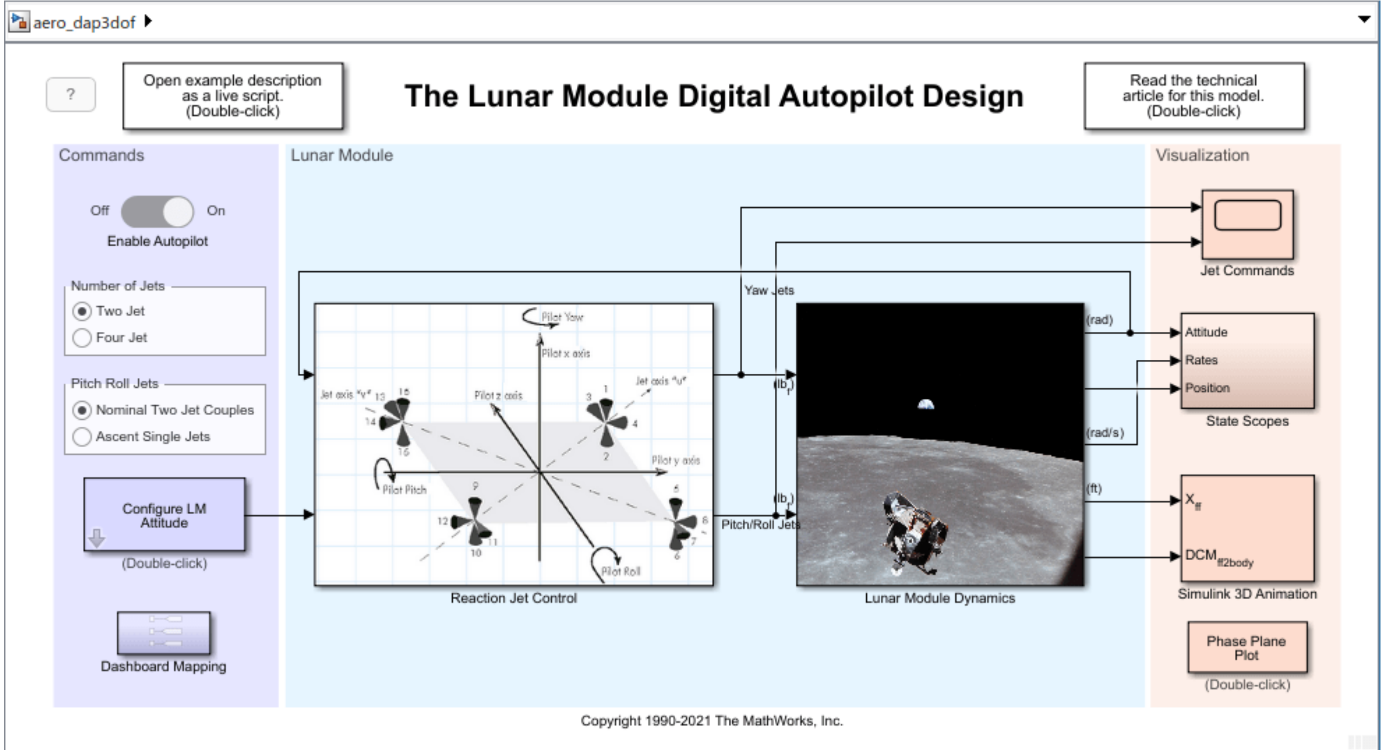
This example shows how Richard and the other engineers who worked on the Apollo Lunar Module digital autopilot design team could have done it using Simulink® and Aerospace Blockset™ if they had been available in 1961.

Model Description

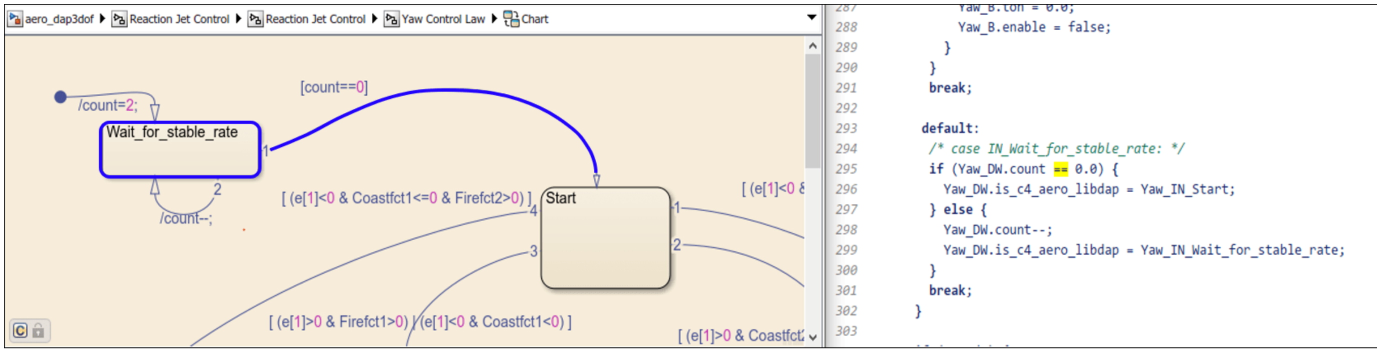
Developing the autopilot in Simulink takes a fraction of the time it took for the original design of the Apollo Lunar Module autopilot.

```
if ~bdIsLoaded("aero_dap3dof")
    open_system("aero_dap3dof");
end
```

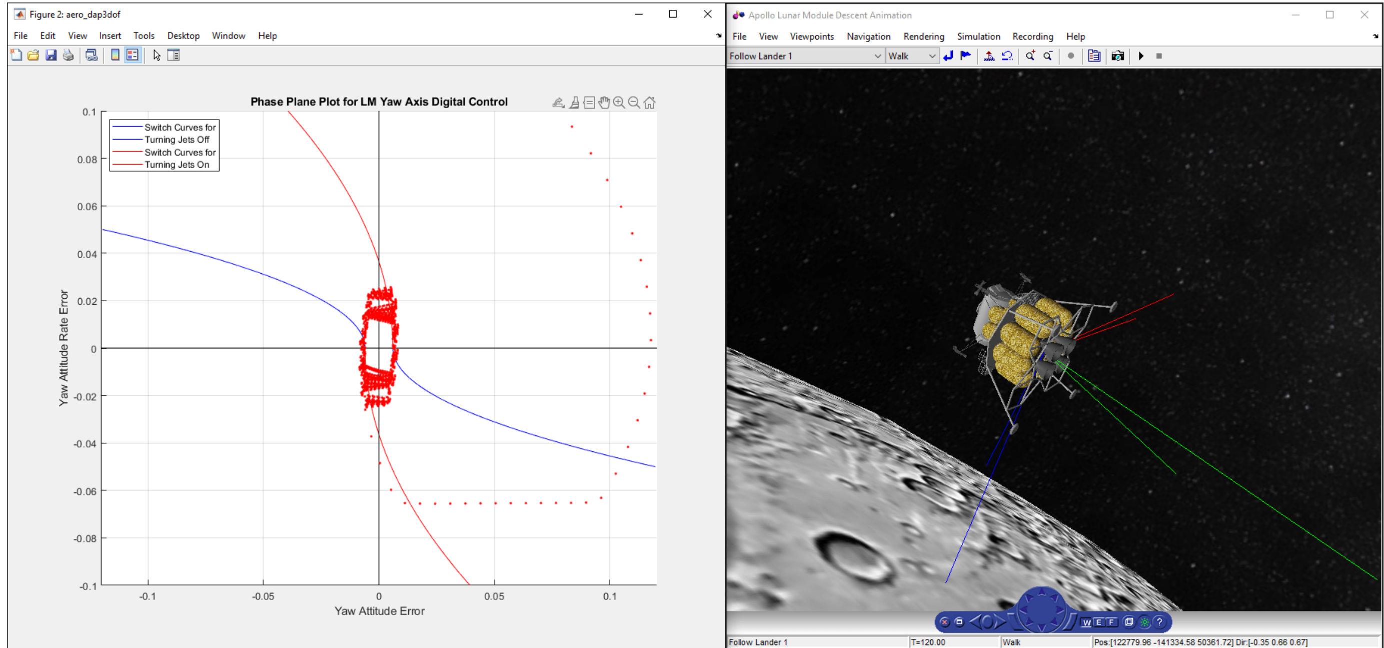

Apollo Lunar Module Descent Animation
File
View
Viewpoints
Navigation
Rendering
Simulation
Recording
Help
Follow Lander 1
Examine
W
E
F
Follow Lander 1
T=0.00
Examine
Pos:[29.85 -49.18 -73.39] Dir:[-0.34 0.51 0.79]



The Reaction Jet Control subsystem models the digital autopilot design proposed (and implemented) by MIT Instrumentation Laboratories (MIT IL), now called Draper Laboratory. A Stateflow® diagram in the model specifies the logic that implements the phase-plane control algorithm described in the technical article *The Apollo 11 Moon Landing: Spacecraft Design Then and Now*. Depending on which region of the diagram the Lunar Module is executing, the Stateflow diagram is in either a `Fire_region` or a `Coast_region`. Note, the transitions between these different regions depend on certain parameters. The Stateflow diagram determines whether to transition to another state and then computes which reaction jets to fire.

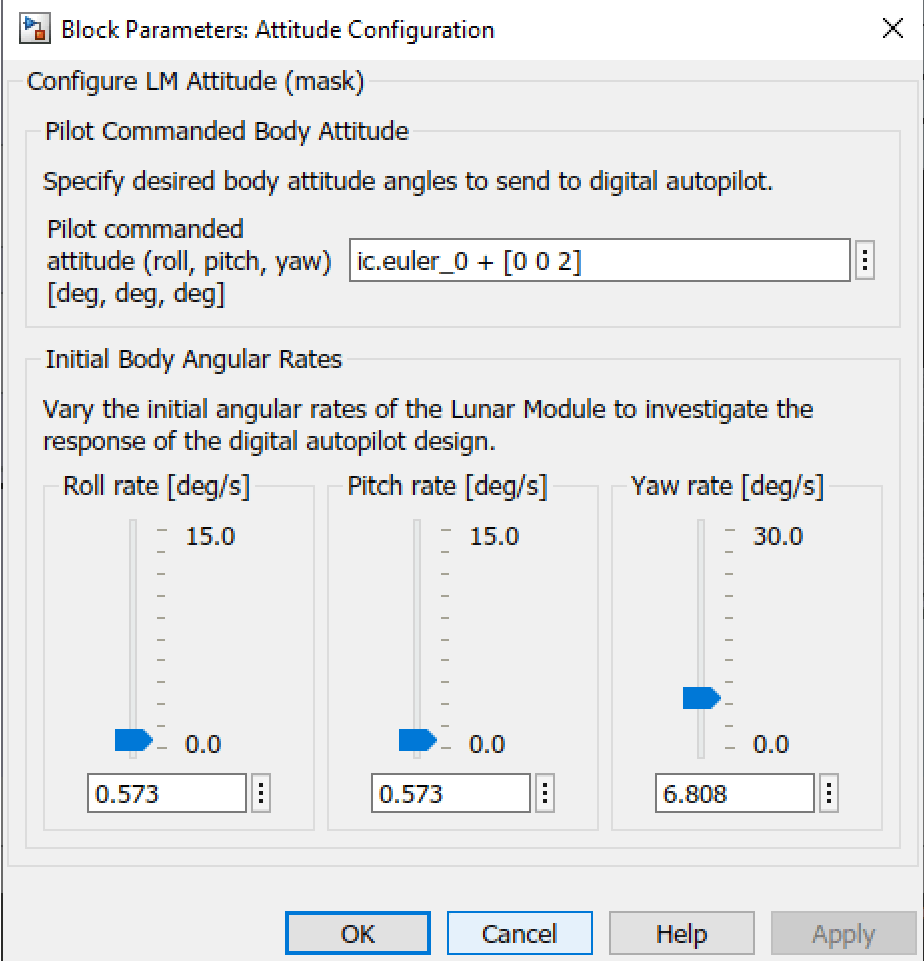


Translational and rotational dynamics of the Lunar Module are approximated in the `Lunar Module Dynamics` subsystem. Access various visualization methods of the Lunar Module states and autopilot performance in the `Visualization` area of the model, including Simulink scopes, animation with Simulink 3D Animation, and a phase plane plot.



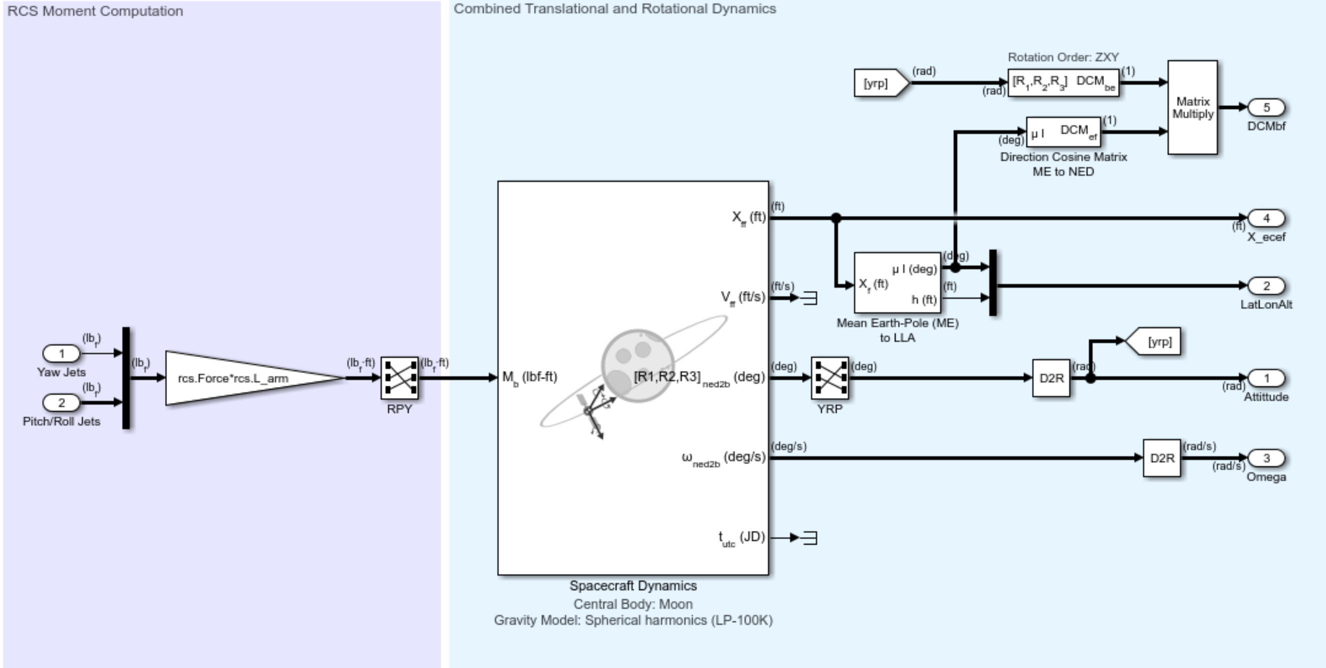
Interactive Controls

To interact with the Lunar Module model, vary autopilot settings and Lunar Module initial states in the `Commands` area. For example, to observe how the digital autopilot design handles increased initial body rates, use the slider components in `Configure LM Attitude`.



Mission Description

The LM digital autopilot has three degrees of freedom. This means that by design, the reaction jet thrusters are configured and commanded to rotate the vehicle without impacting the vehicle's orbital trajectory. Therefore, the translational dynamics in his model are approximated via orbit propagation using the Spacecraft Dynamics block from Aerospace Blockset. The block is configured to use Moon spherical harmonic gravity model LP-100K.



To demonstrate the digital autopilot design behavior, the "Descent Orbit Insertion" mission segment, just prior to the initiation of the powered descent, was selected from the *Apollo 11 Mission Report*.

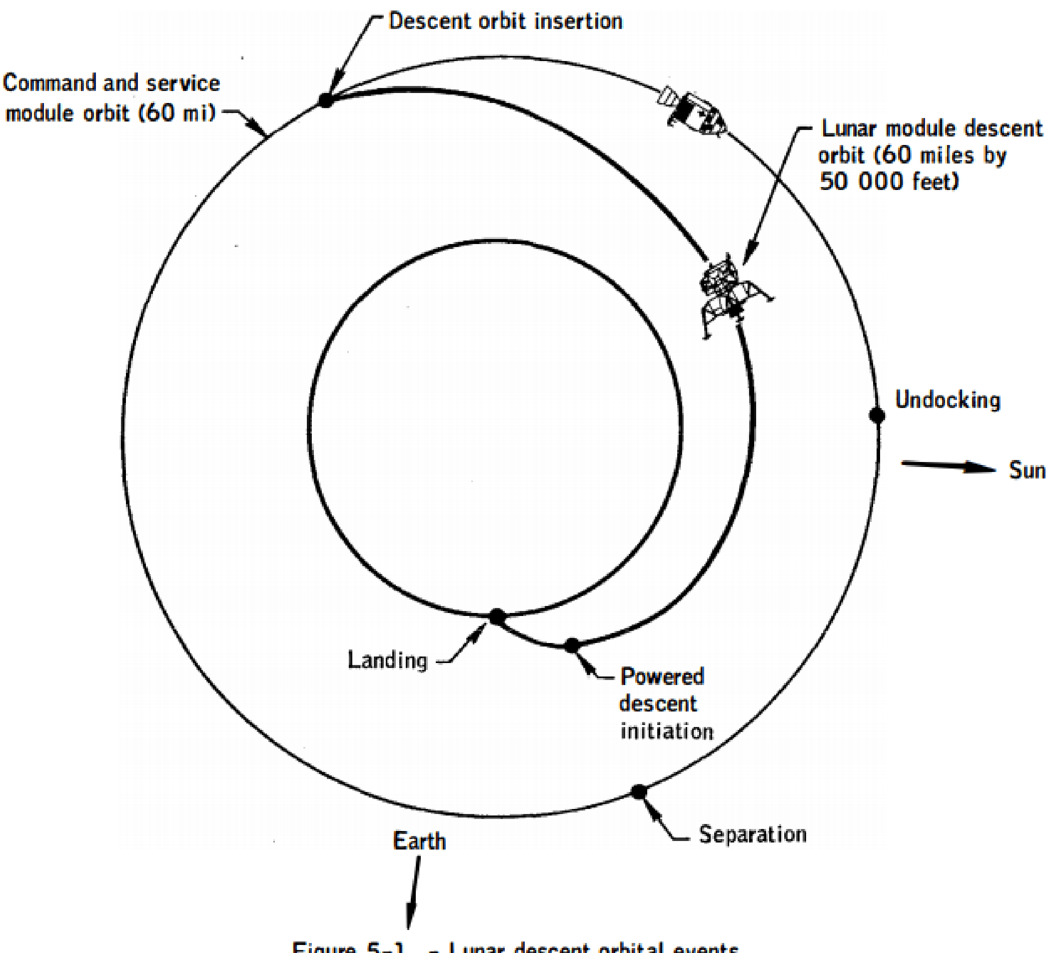


Figure 5-1. - Lunar descent orbital events.

(*Image Credit: NASA*)

The "Descent Orbit Insertion" burn began 101 hours, 36 minutes, and 14 seconds after lift-off and lasted 30 seconds. The burn set the Lunar module on a trajectory to lower its orbit from approximately 60 nautical miles to 50,000 ft over about an hour. At 50,000 ft, the Module initiated its powered descent.

Initialize the model `aero_dap3dof` with the approximate trajectory of the Lunar Module immediately after the descent orbit insertion burn.

```
mission.t_rangeZero             = datetime(1969,7,16,13,32,0); % lift-off
mission.t_descentInsertionStart = mission.t_rangeZero + hours(101) + minutes(36) + seconds(14);
mission.t_descentInsertion      = mission.t_descentInsertionStart + seconds(30);
mission.t_poweredDescentStart   = mission.t_rangeZero + hours(102) + minutes(33) + seconds(5.2);

disp(timetable([mission.t_rangeZero, mission.t_descentInsertionStart, ...
    mission.t_descentInsertion, mission.t_poweredDescentStart]', ...
    {'Range Zero (lift-off)', 'Descent Orbit Insertion (Engine ignition)', ...
    'Descent Orbit Insertion (Engine cutoff)', 'Powered Descent (Engine ignition)'}', VariableNar
```

```
         Time                           Mission Phase                  
    ____________________    ______________________________________________

    16-Jul-1969 13:32:00    {'Range Zero (lift-off)'                      }
    20-Jul-1969 19:08:14    {'Descent Orbit Insertion (Engine ignition)'}
    20-Jul-1969 19:08:44    {'Descent Orbit Insertion (Engine cutoff)'  }
    20-Jul-1969 20:05:05    {'Powered Descent (Engine ignition)'        }
```

The trajectory of the module at "Descent Orbit Insertion (Engine cutoff)" and "Powered Descent Initiation (Engine ignition)" is provided in the *Apollo 11 Mission Report* (Table 7-II.- Trajectory Parameters).

```
mission.Latitude_deg  = [-1.16, 1.02]';    % [deg]
mission.Longitude_deg = [-141.88, 39.39]'; % [deg]
mission.Altitude_mi   = [57.8, 6.4]';      % [nautical miles]
mission.Altitude_ft   = convlength(mission.Altitude_mi, 'naut mi', 'ft');
mission.Velocity_fps  = [5284.9, 5564.9]'; % [ft/s] (in Inertial frame)
mission.FlightPathAngle_deg = [-0.06, 0.03]'; % [deg] (measured upward from local horizontal plan
mission.HeadingAngle_deg = [-75.19 -101.23]'; % [deg] (measured East of North)
disp(table({'Range Zero (lift-off)'; 'Descent Orbit Insertion (Engine ignition)'}, ...
    mission.Latitude_deg, mission.Longitude_deg, mission.Altitude_mi, ...
    mission.Velocity_fps, mission.FlightPathAngle_deg, mission.HeadingAngle_deg, ...
    VariableNames=["Mission Phase", ...
    "Latitude (deg)", "Longitude (deg)", "Altitude (mi)", ...
    "Velocity (ft/s)", "Flight path angle (deg)", "Heading (deg)"]));
```

```
                   Mission Phase                    Latitude (deg)    Longitude (deg)    Altitu
    ______________________________________________    ______________    _______________    ______

    {'Range Zero (lift-off)'                    }        -1.16            -141.88            57
    {'Descent Orbit Insertion (Engine ignition)'}         1.02              39.39             6
```

Model Initialization

Initialize model parameters for the mission phase "Descent Orbit Insertion (Engine cutoff)" using the data defined above.

The intialization function `aero_dap3dofdata` requires information about the orientation of the Moon, which can be calculated using the Aerospace Blockset function `moonLibration`. This function requires "Ephemeris Data for Aerospace Toolbox". Use `aeroDataPackage` to install this data if it is not already installed.

```
mission.LibrationAngles_deg = moonLibration(juliandate(mission.t_descentInsertion), "405");
```

This example uses saved libration angle data corresponding with `t_descentInsertion`. Use the above command after installing the required ephemeris data.

```
mission.LibrationAngles_deg = [0.006379917345247; 0.382328074214300; 6.535718297208969];
```

Run the intialization function:

```
[moon, ic, vehicle, rcs] = aero_dap3dofdata(...
    mission.Latitude_deg(1), mission.Longitude_deg(1), mission.Altitude_ft(1), ...
    mission.Velocity_fps(1), mission.FlightPathAngle_deg(1), ...
    mission.HeadingAngle_deg(1), mission.LibrationAngles_deg)
```

```
moon = struct with fields:
    r_moon_eq: 5702428
```

```
       f_moon: 0.0012


ic = struct with fields:
       t_runtime: 120
    pos_inertial: [-3.6488e+06 -4.4381e+06 -1.9070e+06]
    vel_inertial: [4.0625e+03 -3.3792e+03 86.4867]
         euler_0: [-30 -10 -60]


vehicle = struct with fields:
    inertia_0: [3x3 double]
       mass_0: 33296


rcs = struct with fields:
     Force: 100
     L_arm: 5.5000
        DB: 0.0060
      tmin: 0.0140
     alph1: 0.0550
     alph2: 0.0039
     alph3: 0.0050
     alphu: 0.0063
     alphv: 7.8553e-04
    alphs1: 0.0055
    alphsu: 6.2855e-04
    alphsv: 7.8553e-05
    clockt: 0.0050
      delt: 0.1000
```

Closing Remarks

Building a digital autopilot was a daunting task in 1961 because there was very little industrial infrastructure for it - everything about it was in the process of being invented. Here is an excerpt from the technical article *The Apollo 11 Moon Landing: Spacecraft Design Then and Now*:

"One reason why the [autopilot's machine code] was so complex is that the number of jets that could be used to control the rotations about the pilot axes was large. A decision was made to change the axes that the autopilot was controlling to the "jet axes" shown in `aero_dap3dof`. This change dramatically reduced the number of lines of code and made it much easier to program the autopilot in the existing computer. Without this improvement, it would have been impossible to have the autopilot use only 2000 words of storage. The lesson of this change is that when engineers are given the opportunity to code the computer with the system they are designing, they can often modify the design to greatly improve the code."

References

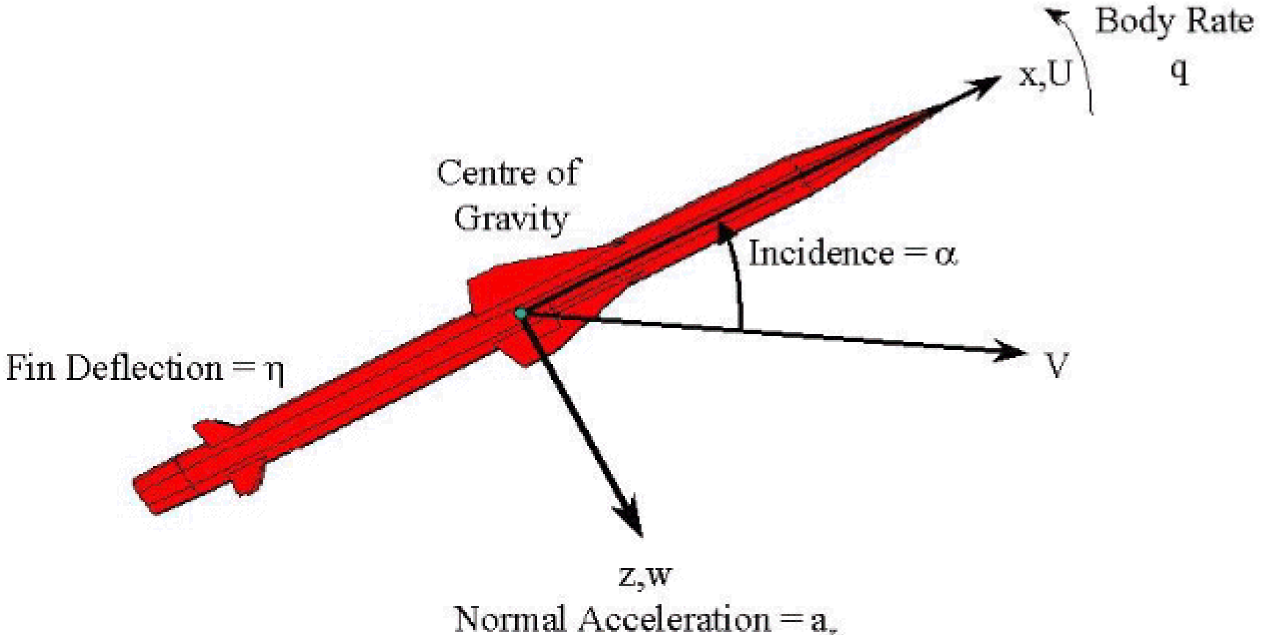
[1] National Aeronautics and Space Administration Manned Spacecraft Center, Mission Evaluation Team. (November 1969). *Apollo 11 Mission Report MSC-00171*. Retrieved from https://www.nasa.gov/specials/apollo50th/pdf/A11_MissionReport.pdf

[2] Richard J. Gran, MathWorks. (2019). *The Apollo 11 Moon Landing: Spacecraft Design Then and Now*. Retrieved from https://www.mathworks.com/company/newsletters/articles/the-apollo-11-moon-landing-spacecraft-design-then-and-now.html

Design a Guidance System in MATLAB and Simulink

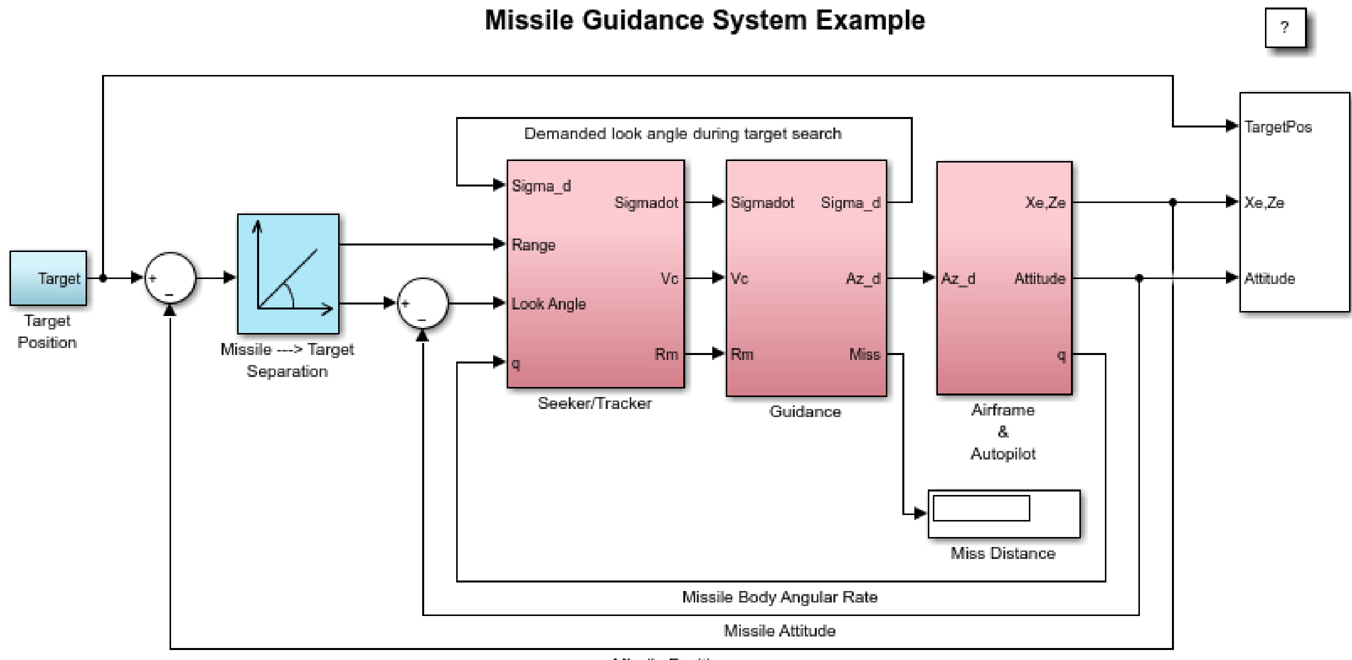
This example shows how to use the model of the missile airframe presented in a number of published papers (References [1], [2] and [3]) on the use of advanced control methods applied to missile autopilot design. The model represents a tail controlled missile traveling between Mach 2 and Mach 4, at altitudes ranging between 10,000ft (3,050m) and 60,000ft (18,290m), and with typical angles of attack ranging between +/-20 degrees.

Model of the Airframe Dynamics



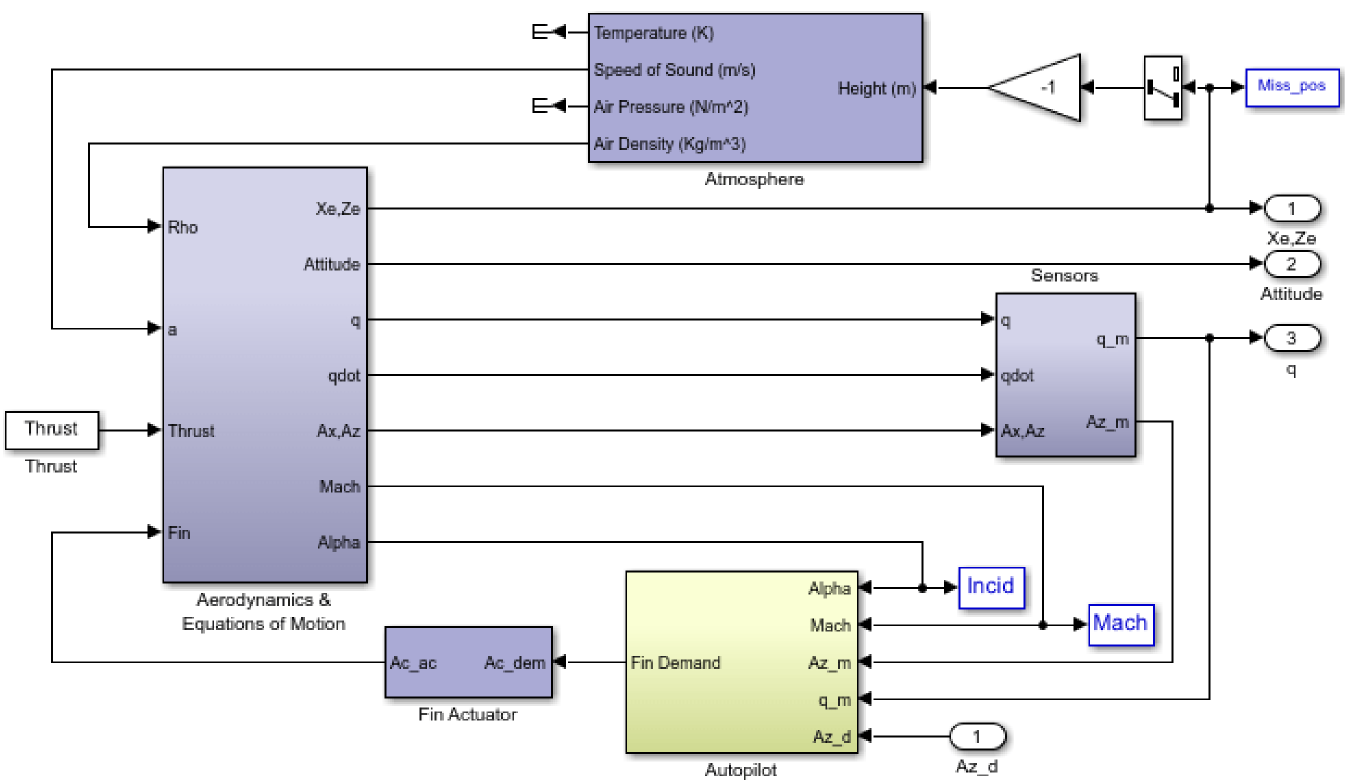
The core element of the model is a nonlinear representation of the rigid body dynamics of the airframe. The aerodynamic forces and moments acting on the missile body are generated from coefficients that are non-linear functions of both incidence and Mach number. The model can be created with Simulink® and the Aerospace Blockset™. The aim of this blockset is to provide reference components, such as atmosphere models, which will be common to all models irrespective of the airframe configuration. Simplified versions of the components available in the Aerospace Blockset are included with these examples to give you a sense of the potential for reuse available from standard block libraries.

Open the model.



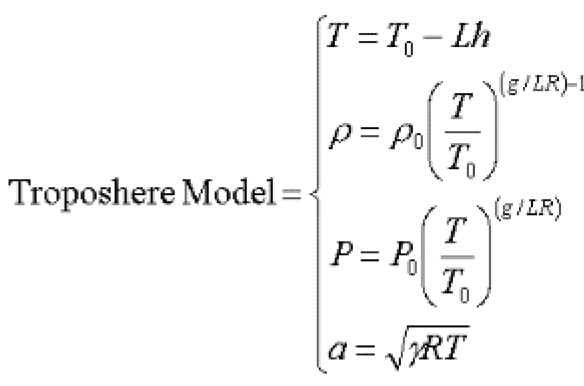
Representing the Airframe in Simulink

The airframe model consists of four principal subsystems, controlled through the acceleration-demand autopilot. The Atmosphere model calculates the change in atmospheric conditions with changing altitude, the Fin Actuator and Sensors models couple the autopilot to the airframe, and the Aerodynamics and Equations of Motion model calculates the magnitude of the forces and moments acting on the missile body, and integrates the equations of motion.

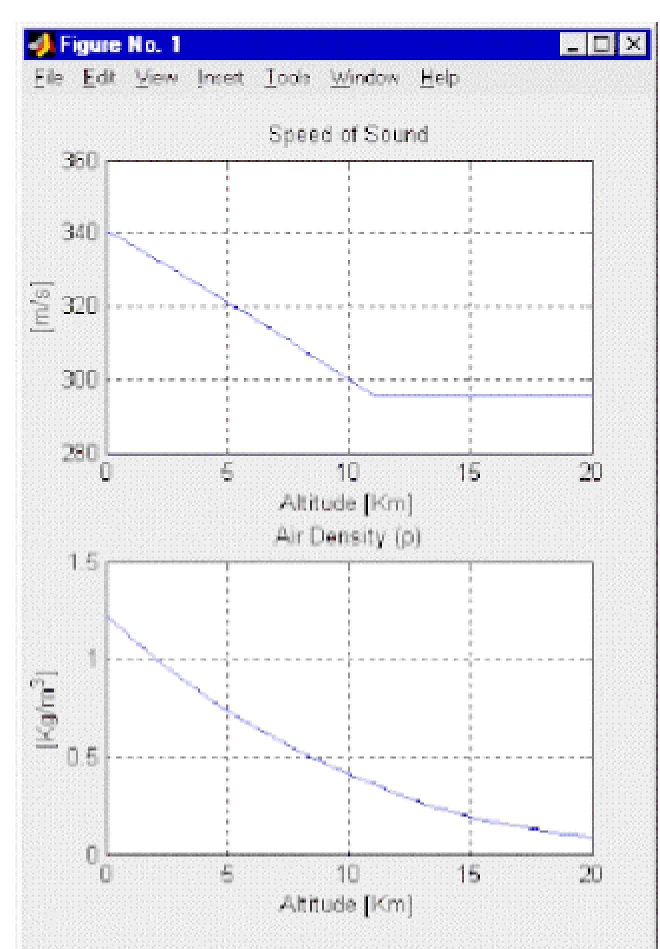


Temperature (K)
Speed of Sound (m/s)
Air Pressure (N/m^2)
Air Density (Kg/m^3)
Height (m)
Atmosphere
-1
Miss_pos
Rho
a
Thrust
Fin
Xe,Ze
Attitude
q
qdot
Ax,Az
Mach
Alpha
Aerodynamics &
Equations of Motion
Thrust
Thrust
1
Xe,Ze
2
Attitude
Sensors
q
qdot
Ax,Az
q_m
Az_m
3
q
Incid
Mach
Alpha
Mach
Az_m
q_m
Az_d
Fin Demand
Autopilot
1
Az_d
Ac_ac
Ac_dem
Fin Actuator

International Standard Atmosphere Model



$$\text{Troposhere Model} = \begin{cases} T = T_0 - Lh \\ \rho = \rho_0 \left(\dfrac{T}{T_0} \right)^{(g/LR)-1} \\ P = P_0 \left(\dfrac{T}{T_0} \right)^{(g/LR)} \\ a = \sqrt{\gamma R T} \end{cases}$$



T_0 = Absolute temperature at mean sea level [K]

ρ_0 = Air density at mean sea level [Kg/m^3]

P_0 = Static pressure at mean sea level [N/m^2]

h = Altitude [m]

T = Temperature at altitude h

ρ = Air density at altitude h

P = Static pressure at altitude h

a = Speed of sound at altitude h

L = Lapse rate [K/m]

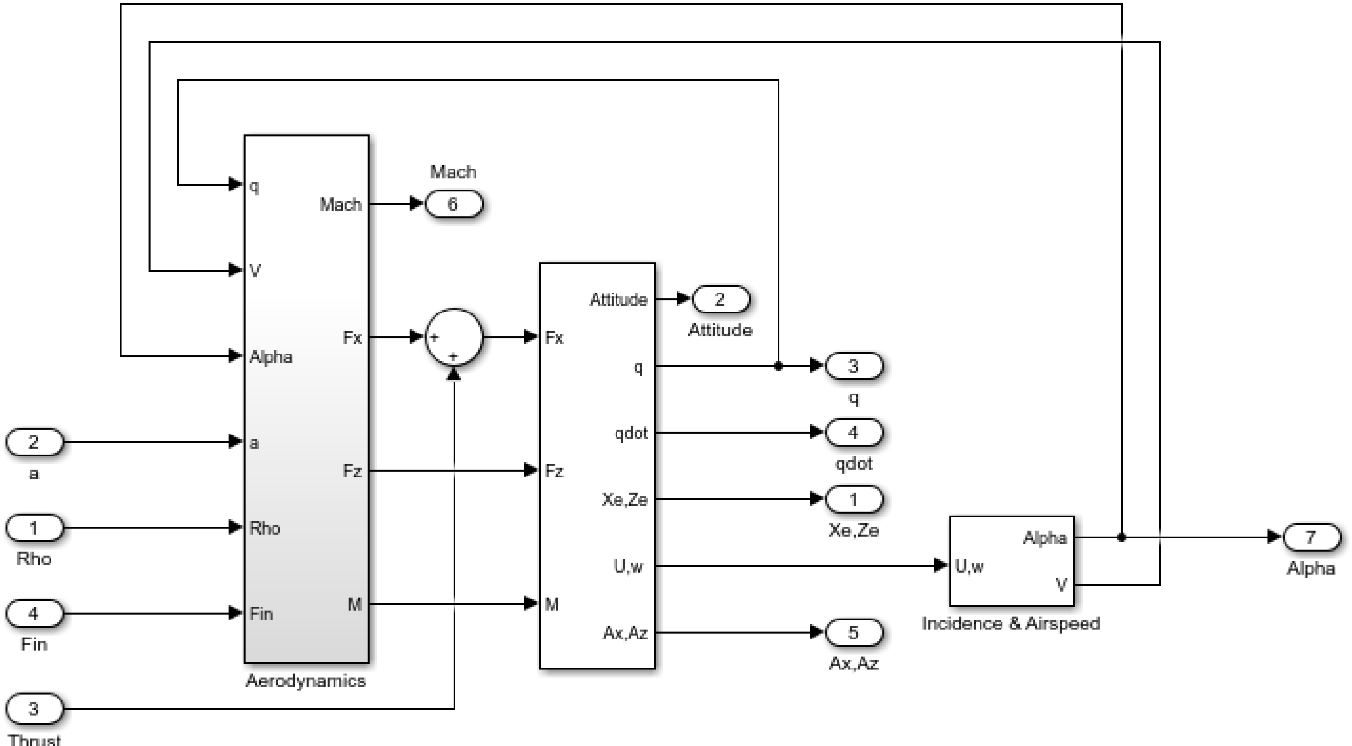
R = Characteristic gas constant [J/Kg/K]

g = Acceleration due to gravity [m/s^2]

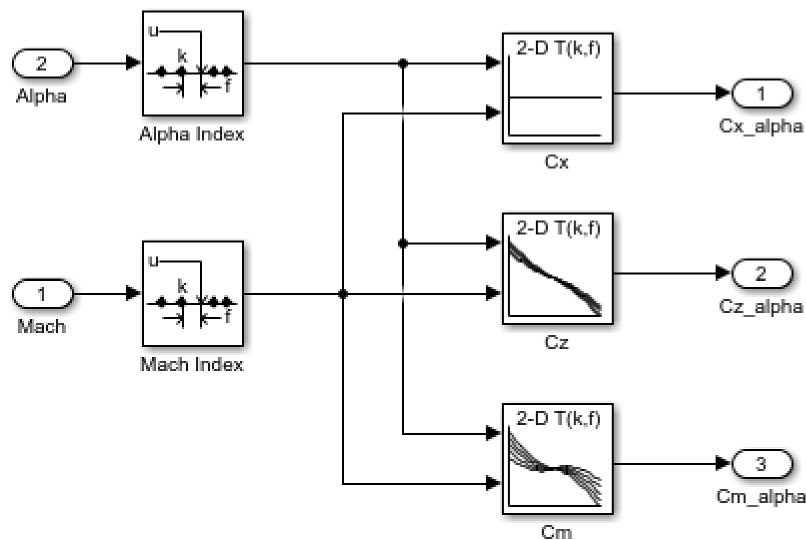
The Atmosphere Subsystem that is used is an approximation to the International Standard Atmosphere, and is split into two separate regions. The troposphere region lies between sea level and 11Km, and in this region there is assumed to be a linear temperature drop with changing altitude. Above the troposphere lies the lower stratosphere region ranging between 11Km and 20Km. In this region the temperature is assumed to remain constant.

Aerodynamic Coefficients for Constructing Forces and Moments

The Aerodynamics & Equations of Motion Subsystem generates the forces and moments applied to the missile in body axes, and integrates the equations of motion which define the linear and angular motion of the airframe.



The aerodynamic coefficients are stored in datasets, and during the simulation the value at the current operating condition is determined by interpolation using 2-D lookup table blocks.

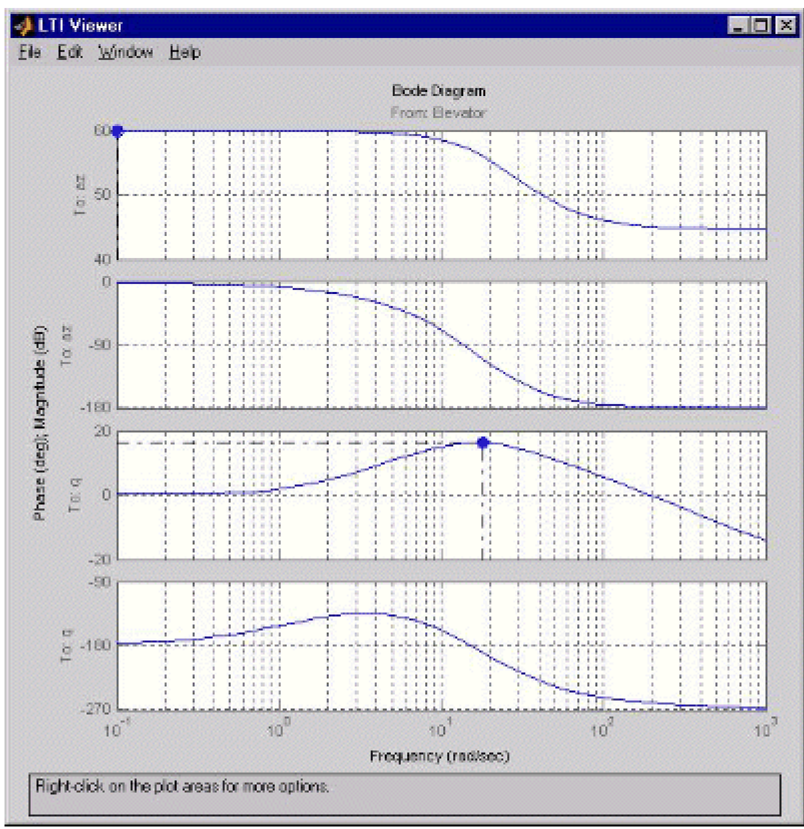


Classical Three Loop Autopilot Design

The aim of the missile autopilot is to control acceleration normal to the missile body. In this example the autopilot structure is a three loop design using measurements from an accelerometer placed

ahead of the center of gravity, and a rate gyro to provide additional damping. The controller gains are scheduled on incidence and Mach number, and are tuned for robust performance at an altitude of 10,000 ft.

To design the autopilot using classical design techniques requires that linear models of the airframe pitch dynamics be derived about a number of trimmed flight conditions. MATLAB® can determine the trim conditions, and derive linear state space models directly from the non-linear Simulink model, saving both time, and aiding in the validation of the model that has been created. The functions provided by the MATLAB Control System Toolbox™ and Simulink® Control Design™ allow the designer to visualize the behavior of the airframe open loop frequency (or time) responses. To see how to trim and linearize the airframe model you can run the companion example, "Airframe Trim and Linearize".



Airframe Frequency Response

Autopilot designs are carried out on a number of linear airframe models derived at varying flight conditions across the expected flight envelope. To implement the autopilot in the non-linear model involves storing the autopilot gains in 2 dimensional lookup tables, and incorporating an anti-windup gain to prevent integrator windup when the fin demands exceed the maximum limits. Testing the autopilot in the nonlinear Simulink model is then the best way to show satisfactory performance in the presence of non-linearities such as actuator fin and rate limits, and with the gains now dynamically varying with changing flight condition.

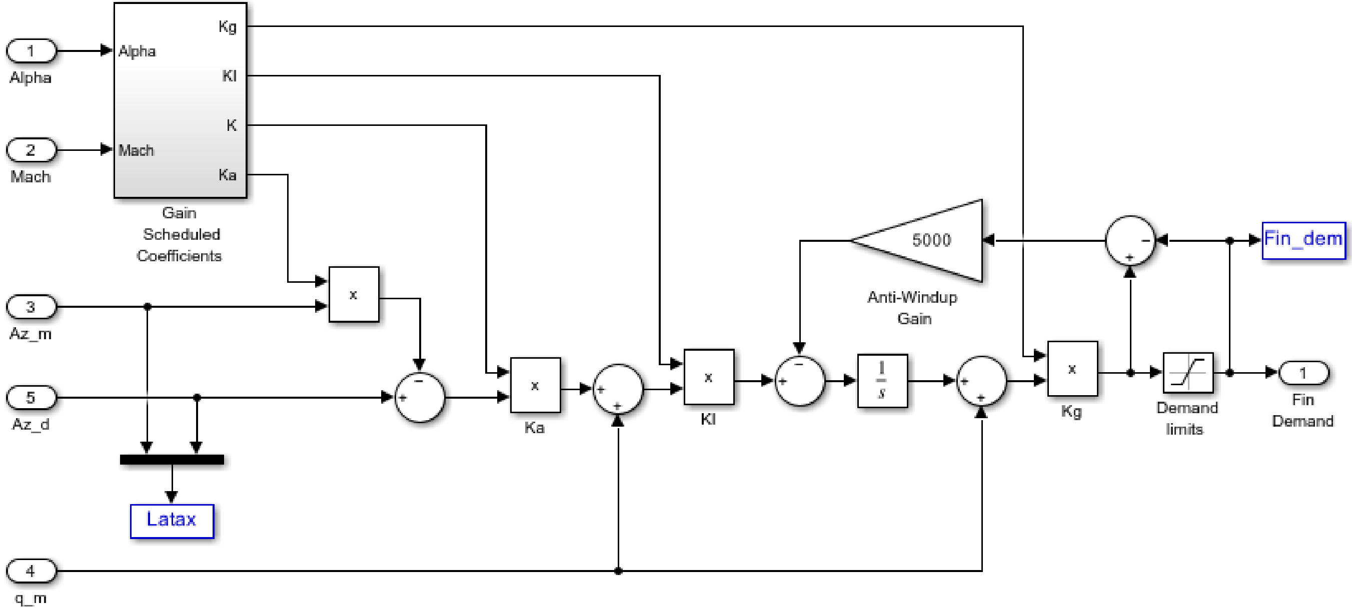
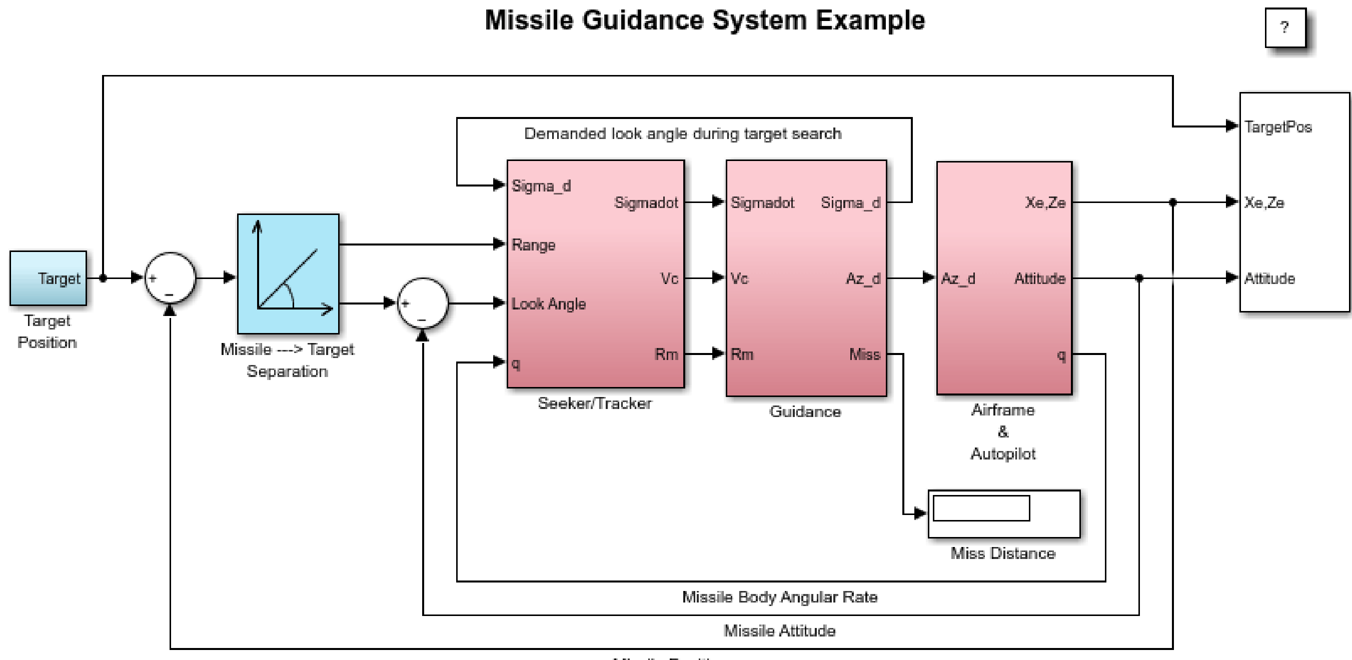


Figure: Simulink implementation of gain scheduled autopilot

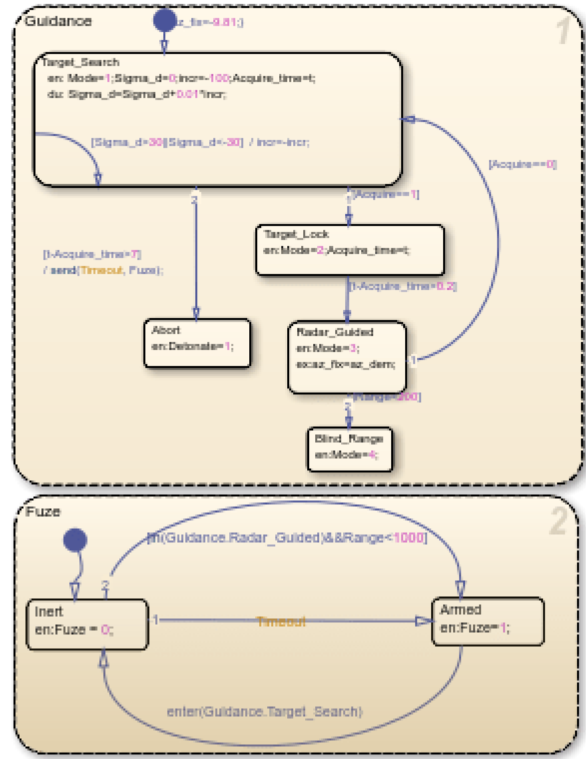
Homing Guidance Loop

The complete Homing Guidance Loop consists of a Seeker/Tracker Subsystem which returns measurements of the relative motion between the missile and target, and the Guidance Subsystem which generates normal acceleration demands which are passed to the autopilot. The autopilot is now part of an inner loop within the overall homing guidance system. Reference [4] provides information on the differing forms of guidance that are currently in use, and provides background information on the analysis techniques that are used to quantify guidance loop performance.



Guidance Subsystem

The function of the Guidance subsystem is to not only generate demands during closed loop tracking, but also perform an initial search to locate the target position. A Stateflow® model is used to control the transfer between these differing modes of operation. Switching between modes is triggered by events generated either in Simulink, or internal to the Stateflow model. Controlling the way the Simulink model then behaves is achieved by changing the value of the variable **Mode** that is passed out to Simulink. This variable is used to switch between the differing control demands that can be generated. During target search the Stateflow model controls the tracker directly by sending demands to the seeker gimbals (**Sigma**). Target acquisition is flagged by the tracker once the target lies within the beamwidth of the seeker (**Acquire**), and after a short delay closed loop guidance starts. Stateflow is an ideal tool for rapidly defining all the operational modes, whether they are for normal operation, or unusual situations. For example, the actions to be taken should there be loss of lock on the target, or should a target not be acquired during target search are catered for in this Stateflow diagram.



Proportional Navigation Guidance

Once the seeker has acquired the target a Proportional Navigation Guidance (PNG) law is used to guide the missile until impact. This form of guidance law has been used in guided missiles since the 1950s, and can be applied to radar, infrared or television guided missiles. The navigation law requires measurements of the closing velocity between the missile and target, which for a radar guided missile could be obtained using a Doppler tracking device, and an estimate for the rate of change of the inertial sightline angle.

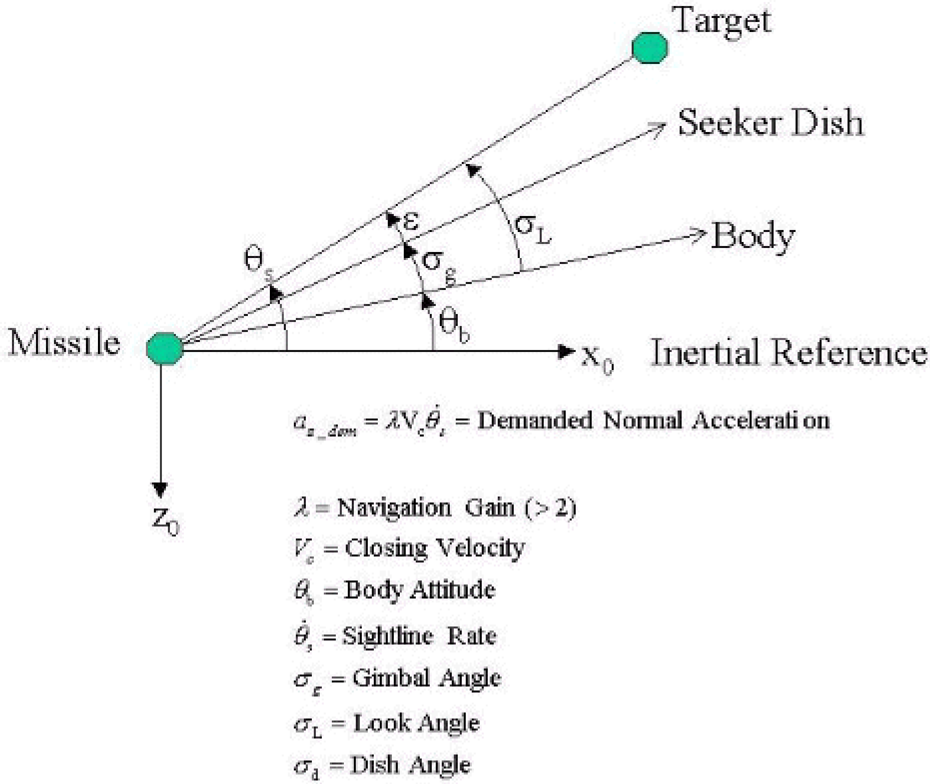
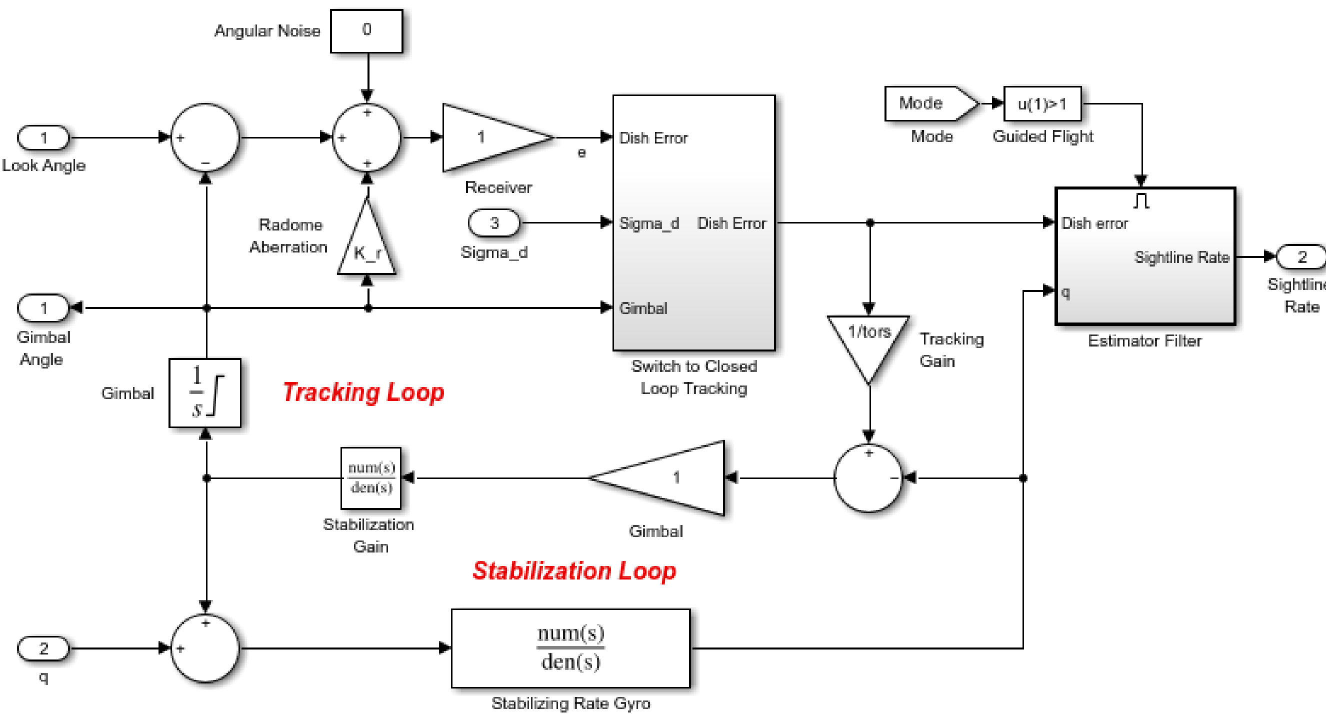


Figure: Proportional Navigation Guidance Law

Seeker/Tracker Subsystem

The aim of the Seeker/Tracker Subsystem is both to drive the seeker gimbals to keep the seeker dish aligned with the target, and to provide the guidance law with an estimate of the sightline rate. The tracker loop time constant **tors** is set to 0.05 seconds, and is chosen as a compromise between maximizing speed of response, and keeping the noise transmission to within acceptable levels. The stabilization loop aims to compensate for body rotation rates, and the gain **Ks**, which is the loop cross-over frequency, is set as high as possible subject to the limitations of the bandwidth of the stabilizing rate gyro. The sightline rate estimate is a filtered value of the sum of the rate of change of the dish angle measured by the stabilizing rate gyro, and an estimated value for the rate of change of the angular tracking error (**e**) measured by the receiver. In this example the bandwidth of the estimator filter is set to half that of the bandwidth of the autopilot.



Radome Aberration

For radar guided missiles a parasitic feedback effect that is commonly modelled is that of radome aberration. It occurs because the shape of the protective covering over the seeker distorts the returning signal, and then gives a false reading of the look angle to the target. Generally the amount of distortion is a nonlinear function of the current gimbal angle, but a commonly used approximation is to assume a linear relationship between the gimbal angle and the magnitude of the distortion. In the above system, the radome aberration is accounted for in the gain block labeled "Radome Aberration". Other parasitic effects, such as sensitivity in the rate gyros to normal acceleration, are also often modelled to test the robustness of the target tracker and estimator filters.

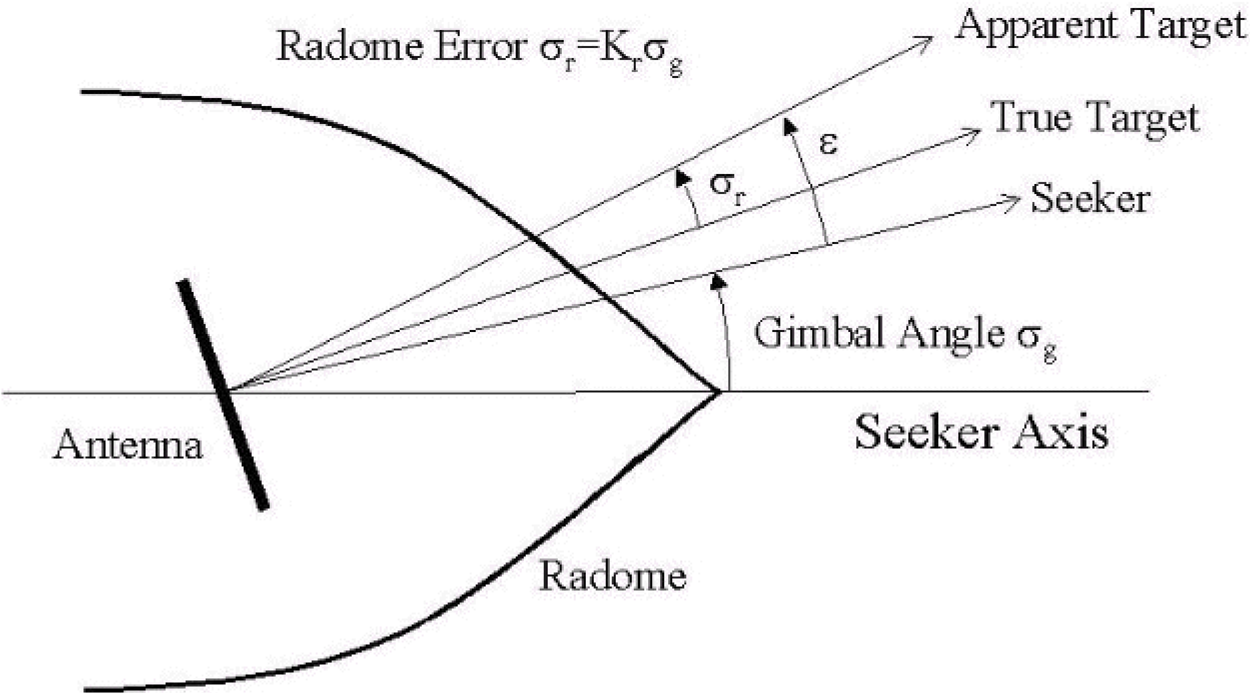
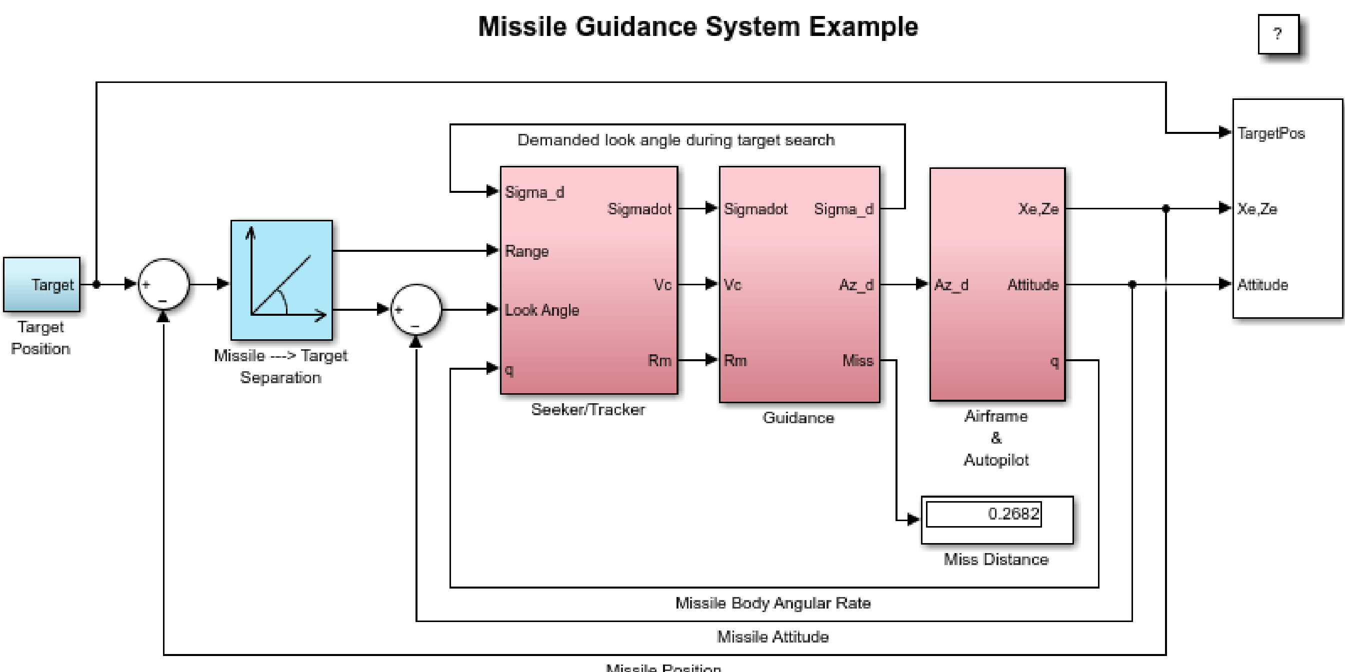


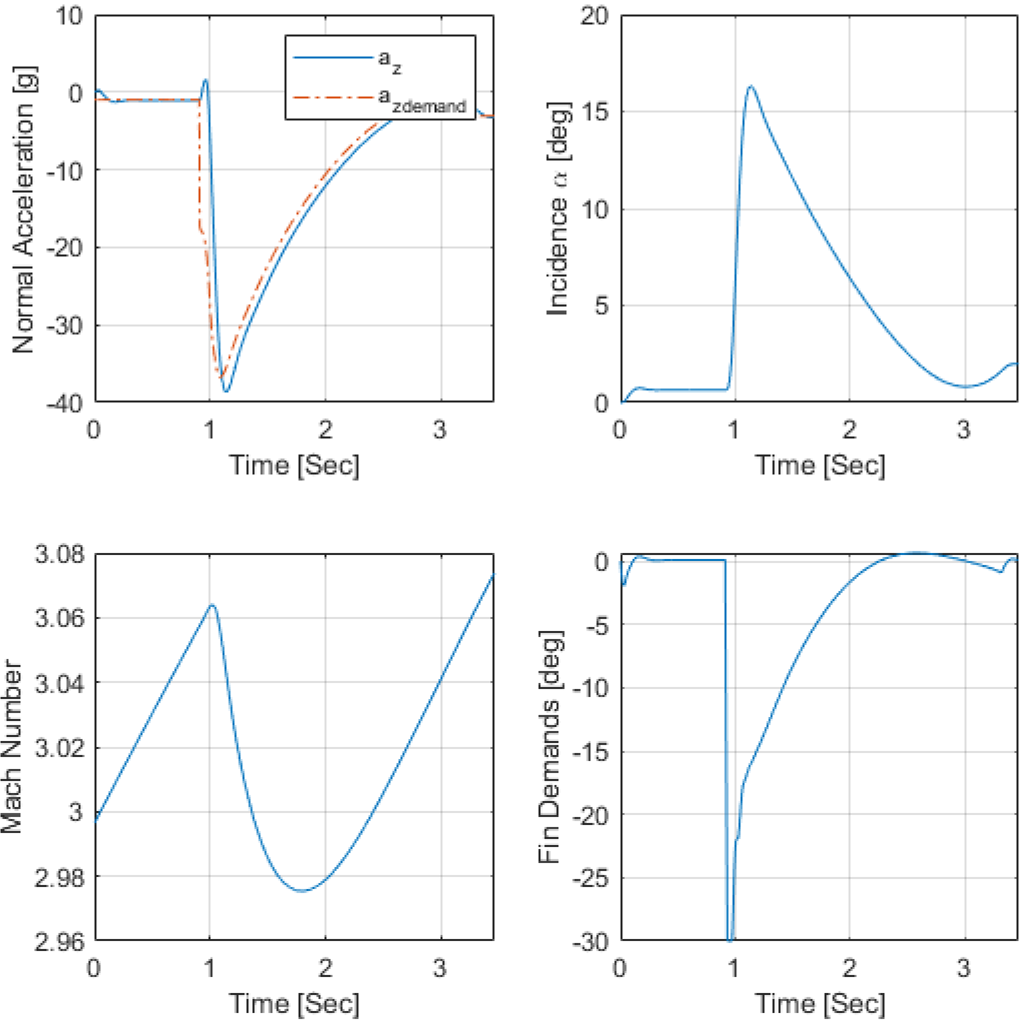
Figure: Radome aberration geometry

Running the Guidance Simulation

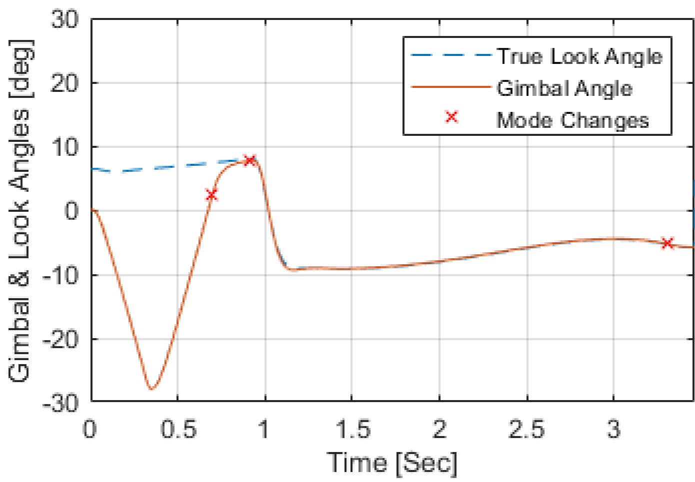
Now to show the performance of the overall system. In this case the target is defined to be traveling at a constant speed of 328m/s, on a reciprocal course to the initial missile heading, and 500m above the initial missile position. From the simulation results it can be determined that acquisition occurred 0.69 seconds into the engagement, with closed loop guidance starting after 0.89 seconds. Impact with the target occurred at 3.46 seconds, and the range to go at the point of closest approach was calculated to be 0.265m.

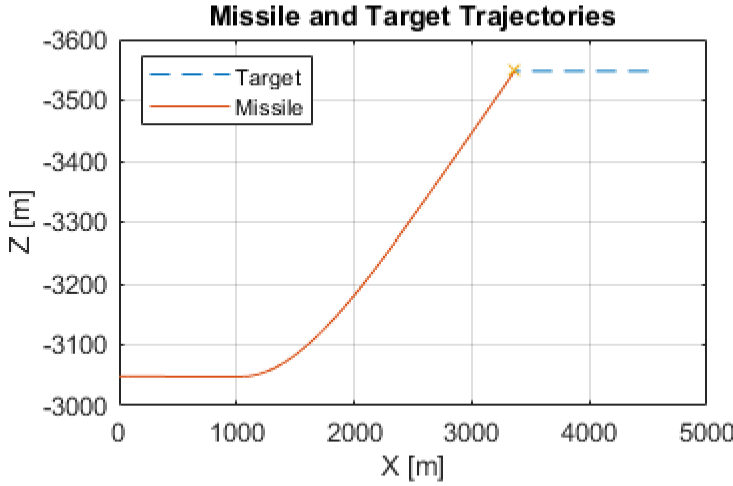
The aero_guid_plot.m script creates a performance analysis


Missile Guidance System Example
?
Demanded look angle during target search
Target
Target Position
Missile ---> Target Separation
Sigma_d
Range
Look Angle
q
Sigmadot
Vc
Rm
Seeker/Tracker
Sigmadot
Vc
Rm
Sigma_d
Az_d
Miss
Guidance
Az_d
Xe,Ze
Attitude
q
Airframe
&
Autopilot
0.2682
Miss Distance
TargetPos
Xe,Ze
Attitude
Missile Body Angular Rate
Missile Attitude
Missile Position
Copyright 1990-2014 The MathWorks, Inc.

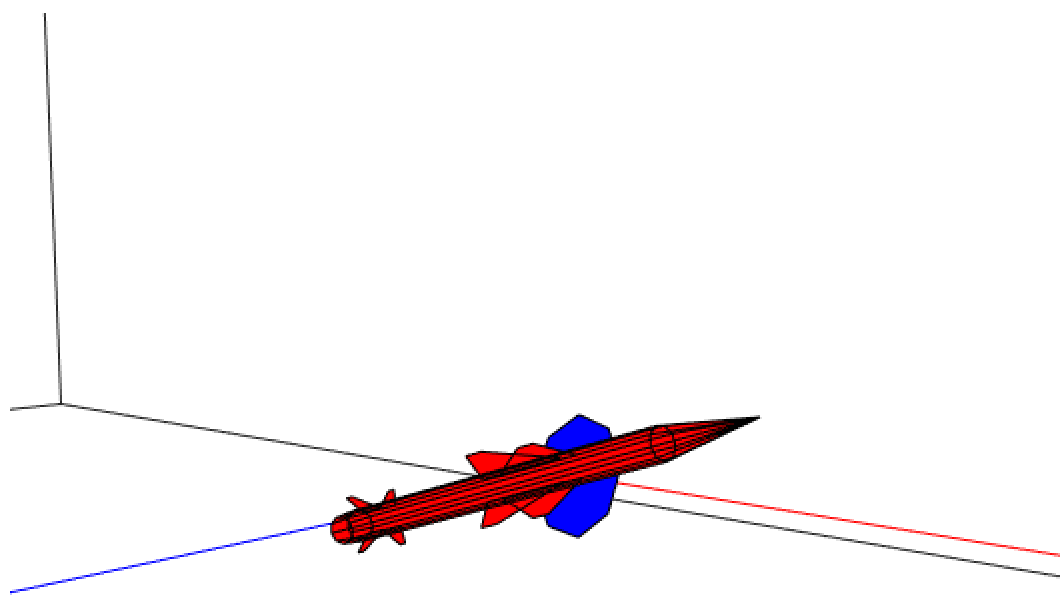


10
0
-10
-20
-30
-40
Normal Acceleration [g]
a_z
$a_{zdemand}$
0 1 2 3
Time [Sec]
20
15
10
5
0
Incidence α [deg]
0 1 2 3
Time [Sec]
3.08
3.06
3.04
3.02
3
2.98
2.96
Mach Number
0 1 2 3
Time [Sec]
0
-5
-10
-15
-20
-25
-30
Fin Demands [deg]
0 1 2 3
Time [Sec]





The animation block provides a visual reference for the simulation



References

1. "Robust LPV control with bounded parameter rates", S.Bennani, D.M.C. Willemsen, C.W. Scherer, AIAA-97-3641, August 1997.

2. "Full Envelope Missile Longitudinal Autopilot Design using the State-Dependent Riccati Equation Method", C.P.Mracek and J.R. Cloutier, AIAA-97-3767, August 1997.

3. "Gain-Scheduled Missile Autopilot Design Using Linear Parameter Varying Transformations", J.S.Shamma, J.R. Cloutier, Journal of Guidance, Control and Dynamics, Vol. 16, No. 2, March-April 1993.

4. "Modern Navigation, Guidance, and Control Processing Volume 2", Ching-Fang Lin, ISBN 0-13-596230-7, Prentice Hall, 1991.

MATLAB Syntax Support for States and Transitions

- “Modify the Action Language for a Chart” on page 15-2
- “Differences Between MATLAB and C as Action Language Syntax” on page 15-4

Modify the Action Language for a Chart

Stateflow charts in Simulink models have an action language property that defines the syntax for state and transition actions. An icon in the lower-left corner of the chart canvas indicates the action language for the chart.

- MATLAB as the action language.
- C as the action language.

You can change the action language of a chart in the **Action Language** box of the Chart properties dialog box. For more information, see "Differences Between MATLAB and C as Action Language Syntax" on page 15-4.

Change the Default Action Language

MATLAB is the default action language syntax for new Stateflow charts. To create a chart that uses C as the action language, enter:

```
sfnew -C
```

To change the default action language of new charts, use the `sfpref` function. For example, to set C as the action language for new charts, enter:

```
sfpref(ActionLanguage="C");
```

Auto Correction When Using MATLAB as the Action Language

Stateflow charts that use MATLAB as the action language automatically correct common C constructs to MATLAB syntax:

- Increment and decrement operations such as `a++` and `a--`. For example, `a++` is changed to `a = a +1`.
- Assignment operations such as `a += b`, `a –= b`, `a *= b`, and `a /= b`. For example, `a += b` is changed to `a = a+b`.
- Evaluation operations such as `a != b` and `!a`. For example, `a != b` is changed to `a ~= b`.
- Comment markers `//` and `/* */` are changed to `%`.

To disable this preference, use the `sfpref` function:

```
sfpref(EnableLabelAutoCorrectionForMAL=false);
```

C to MATLAB Syntax Conversion

For nonempty charts, after you change the action language property from C to MATLAB, a notification appears at the top of the chart. The notification provides the option to convert some of the C syntax to MATLAB syntax. In the notification, click the link to have Stateflow convert syntax in the chart. C syntax constructs that are converted include:

- Zero-based indexing.
- Increment and decrement operations such as `a++` and `a--`. For example, `a++` is changed to `a = a +1`.
- Assignment operations such as `a += b`, `a –= b`, `a *= b`, and `a /= b`. For example, `a += b` is changed to `a = a+b`.
- Binary operations such as `a %% b`, `a >> b`, and `a << b`. For example, `a %% b` is changed to `rem(a,b)`.
- Bitwise operations such as `a ^ b`, `a & b`, and `a | b`. For example, if the chart property **Enable C-bit operations** is selected, then `a ^ b` is changed to `bitxor(a,b)`.
- C style comment markers. For example, `//` and `/* */` are changed to `%`.

If the chart contains C constructs that cannot be converted to MATLAB, Stateflow shows a message in a dialog box. Click on the warnings link to display the warnings in the Diagnostic Viewer. Choose whether or not to continue with the conversion of supported syntax. C constructs not converted to MATLAB include:

- Explicit type casts with `cast` and `type`
- Operators such as &, * and `:=`
- Custom data
- Access to workspace variables using `ml` operator
- Functions not supported in code generation
- Hexadecimal and single precision notation
- Context-sensitive constants

See Also

`sfpref`

Related Examples

- “Differences Between MATLAB and C as Action Language Syntax” on page 15-4
- “Operations for Stateflow Data” on page 14-4
- “Operations for Vectors and Matrices in Stateflow” on page 19-4

Differences Between MATLAB and C as Action Language Syntax

Stateflow charts in Simulink models have an action language property that defines the syntax for state and transition actions. An icon in the lower-left corner of the chart canvas indicates the action language for the chart.

- MATLAB as the action language.
- C as the action language.

MATLAB is the default action language syntax for new Stateflow charts. To create a chart that uses C as the action language, enter:

```
sfnew -c
```

Compare Functionality of Action Languages

This table lists the most significant differences in functionality between the two action languages.

Functionality	MATLAB as the Action Language	C as the Action Language
Vector and matrix indexing	One-based indexing delimited by parentheses and commas. For example, `A(4,5)`. See “Operations for Vectors and Matrices in Stateflow” on page 19-4.	Zero-based indexing delimited by square brackets. For example, `A[3][4]`. See “Operations for Vectors and Matrices in Stateflow” on page 19-4.
C constructs: • Increment and decrement operations `a++` and `a--` • Assignment operations `a += b`, `a -= b`, `a *= b`, and `a /= b` • Evaluation operations `a != b` and `!a` • Binary operations `a %% b`, `a >> b`, `a << b`, `a & b`, and `a | b` • Comment markers `//` and `/* */`	Auto-correction to MATLAB syntax. For example, `a++` is corrected to `a = a+1`. See “Auto Correction When Using MATLAB as the Action Language” on page 15-2.	Supported. See “Operations for Stateflow Data” on page 14-4.
Conditional and loop control statements in state actions	Supported. For example, you can use `if`, `for`, and `while` statements in state actions. See “Loops and Conditional Statements”.	Not supported. For conditional and loop patterns, use graphical functions instead. See “Reuse Logic Patterns by Defining Graphical Functions” on page 6-9.

Functionality	MATLAB as the Action Language	C as the Action Language
Format of transition actions	Auto-correction encloses transition actions with braces `{}`. See “Transition Actions” on page 1-41.	Not required to enclose transition actions with braces `{}`. See “Transition Actions” on page 1-41.
Ordering of parallel states	Explicit ordering only. See “Execution Order for Parallel States” on page 2-46.	Explicit or implicit ordering. See “Execution Order for Parallel States” on page 2-46.
Variable-size data	Modify variable-size chart data in state and transition actions. For more information, see “Variable-Size Data in Charts That Use MATLAB as the Action Language” on page 19-9.	Modify variable-size chart data by using: • MATLAB functions • Simulink functions • Truth tables that use MATLAB as the action language All computations with variable-size data must occur inside these functions, and not directly in states or transitions. For more information, see “Variable-Size Data in Charts That Use C as the Action Language” on page 19-10.
Fixed-point constructs: • Special assignment operator `:=` • Context-sensitive constants such as `4.3C`	Not supported.	Supported. See “Override Fixed-Point Promotion in C Charts” on page 23-11 and “Fixed-Point Context-Sensitive Constants” on page 23-4.
Complex data	Use complex number notation `a + bi` or the `complex` operator. See “Operations for Complex Data in Stateflow” on page 24-4.	Use the `complex` operator. Complex number notation is not supported. See “Operations for Complex Data in Stateflow” on page 24-4.
Data type propagation	Follows MATLAB typing rules. For example, adding data of type `double` to data of type `int32` results in data of type `int32`.	Follows C typing rules. For example, adding data of type `double` to data of type `int32` results in data of type `double`.

Functionality	MATLAB as the Action Language	C as the Action Language
Explicit type cast operations	Use one of these casting forms: • MATLAB type conversion function. For example, `single(x)`. • `cast` function with a type keyword. For example, `cast(x,"int8")`. • `cast` function with the `"like"` keyword. For example, `cast(x,"like",z)`. The `type` operator is not supported. See “Type Cast Operations” on page 14-7.	Use one of these casting forms: • MATLAB type conversion function. For example, `uint16(x)`. • `cast` function with the `type` operator. For example, `cast(x,type(z))`. Type keywords for the `cast` function are not supported. See “Type Cast Operations” on page 14-7.
Scalar expansion	Not supported.	Supported. See “Assign Values to All Elements of a Matrix” on page 19-6.
String data	Use double quotes (`"..."`) as delimiters. See “Manage Textual Information by Using Strings” on page 21-2.	Use double (`"..."`) or single quotes (`'...'`) as delimiters. See “Manage Textual Information by Using Strings” on page 21-2.
Specification of data properties: • First index • Save final value to base workspace • Units	Not supported.	Supported. For more information, see: • “First index” on page 10-8 • “Save final value to base workspace” on page 10-17 • “Units” on page 10-17
Scope of data in graphical, truth table, and MATLAB functions	`Constant`, `Parameter`, `Input`, `Output`	`Local`, `Constant`, `Parameter`, `Input`, `Output`, `Temporary`
Dot notation for specifying states, local data, message, and local events inside MATLAB functions	Supported. See “Identify Data by Using Dot Notation” on page 10-39.	Not supported.

Functionality	MATLAB as the Action Language	C as the Action Language
Custom code functions and variables	Behavior depends on the **Import Custom Code** configuration parameter. • When you enable **Import Custom Code**, both custom code functions and variables are supported in states and transitions (default). • When you disable **Import Custom Code**, only custom code functions are supported. Use the `coder.ceval` function. See “Reuse Custom Code in Stateflow Charts” on page 28-2 and “Import custom code” (Simulink).	Custom code functions and variables are supported in states and transitions.
Structure parameters	Tunable and nontunable parameters are supported.	Only tunable parameters are supported.
Use of global `fimath` object	Supported.	Not supported.

Guidelines for Using MATLAB as the Action Language

Use one-based indexing for vectors and matrices

One-based indexing is consistent with MATLAB syntax. For more information, see “Indexing Notation” on page 19-4.

Use parentheses instead of brackets to index into vectors and matrices

This statement is valid:

```
a(2,5) = 0;
```

This statement is not valid:

```
a[2][5] = 0;
```

For more information, see “Indexing Notation” on page 19-4.

Use the MATLAB format for comments

Use `%` to specify comments in states and transitions for consistency with MATLAB. For example, the following comment is valid:

```
% This is a valid comment in the style of MATLAB
```

C style comments, such as `//` and `/* */`, are auto-corrected to use %.

Enclose transition actions with braces

This transition label contains a valid transition action:

```
E [x > 0] / {x = x+1;}
```

This transition label is incorrect, but is auto-corrected to the valid syntax.

```
E [x > 0] / x = x+1;
```

Do not use control flow logic in condition actions and transition actions

Control flow logic (such as `if`, `switch`, `for`, and `while` statements) is supported only in state actions. Use of control flow logic in condition actions or transition actions, result in a syntax error.

Do not declare global or persistent variables in state actions

The keywords `global` and `persistent` are not supported in state actions.

Assign an initial value to local and output data

When using MATLAB as the action language, data read without an initial value causes an error.

Include a type prefix for identifiers of enumerated values

The identifier `TrafficColors.Red` is valid, but `Red` is not.

To generate code from your model, use MATLAB language features supported for code generation

Otherwise, use `coder.extrinsic` to call unsupported functions, which gives the functionality that you want for simulation, but not in the generated code. For a list of supported features and functions, see “Language, Function, and Object Support” (Simulink).

See Also

`sfnew`

More About

- “Modify the Action Language for a Chart” on page 15-2
- “Operations for Stateflow Data” on page 14-4
- “Operations for Vectors and Matrices in Stateflow” on page 19-4
- “Operations for Complex Data in Stateflow” on page 24-4
- “Operations for Fixed-Point Data in Stateflow” on page 23-8
- “Execution Order for Parallel States” on page 2-46

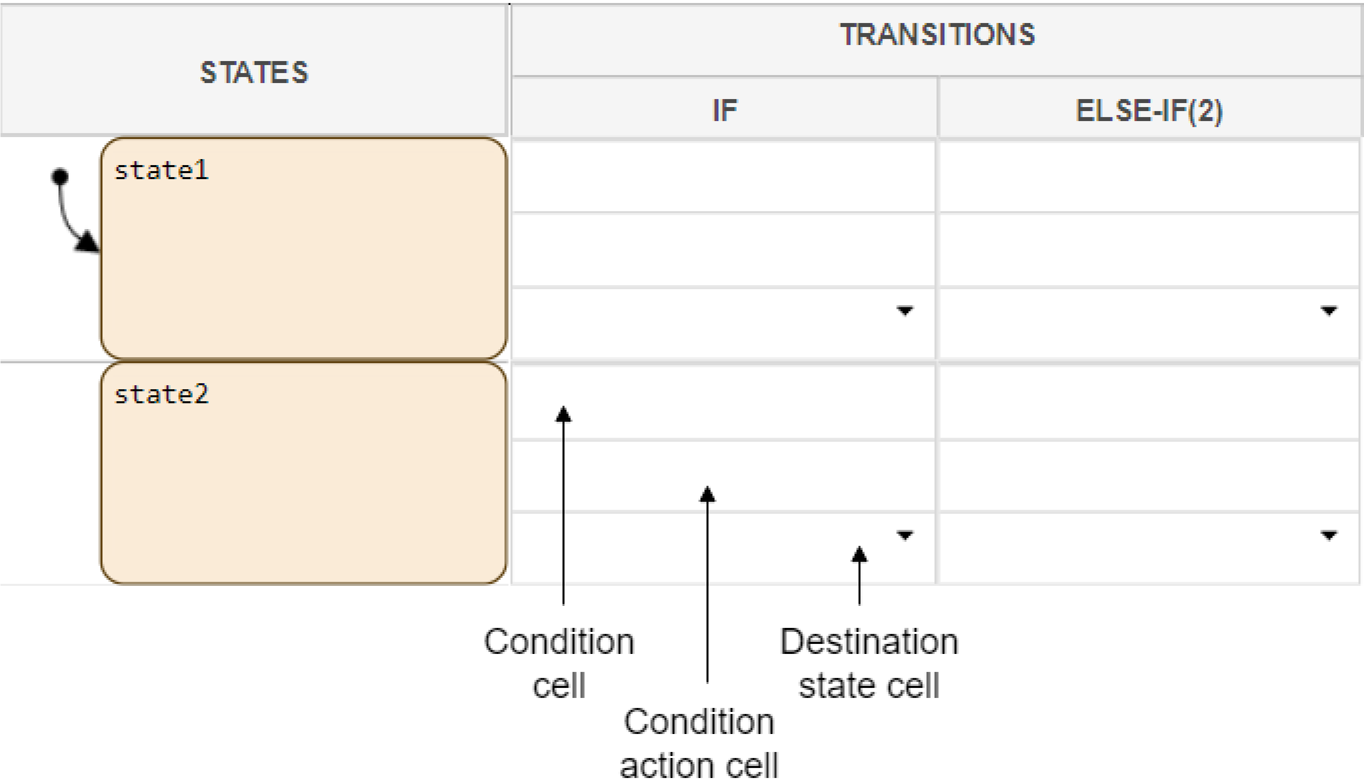
Tabular Expression of Modal Logic

- "Use State Transition Tables to Express Sequential Logic in Tabular Form" on page 16-2
- "Inspect the Design of State Transition Tables" on page 16-9
- "Debug Run-Time Errors in a State Transition Table" on page 16-16
- "Model Bang-Bang Controller by Using a State Transition Table" on page 16-19
- "Modeling a CD Player/Radio Using State Transition Tables" on page 16-30

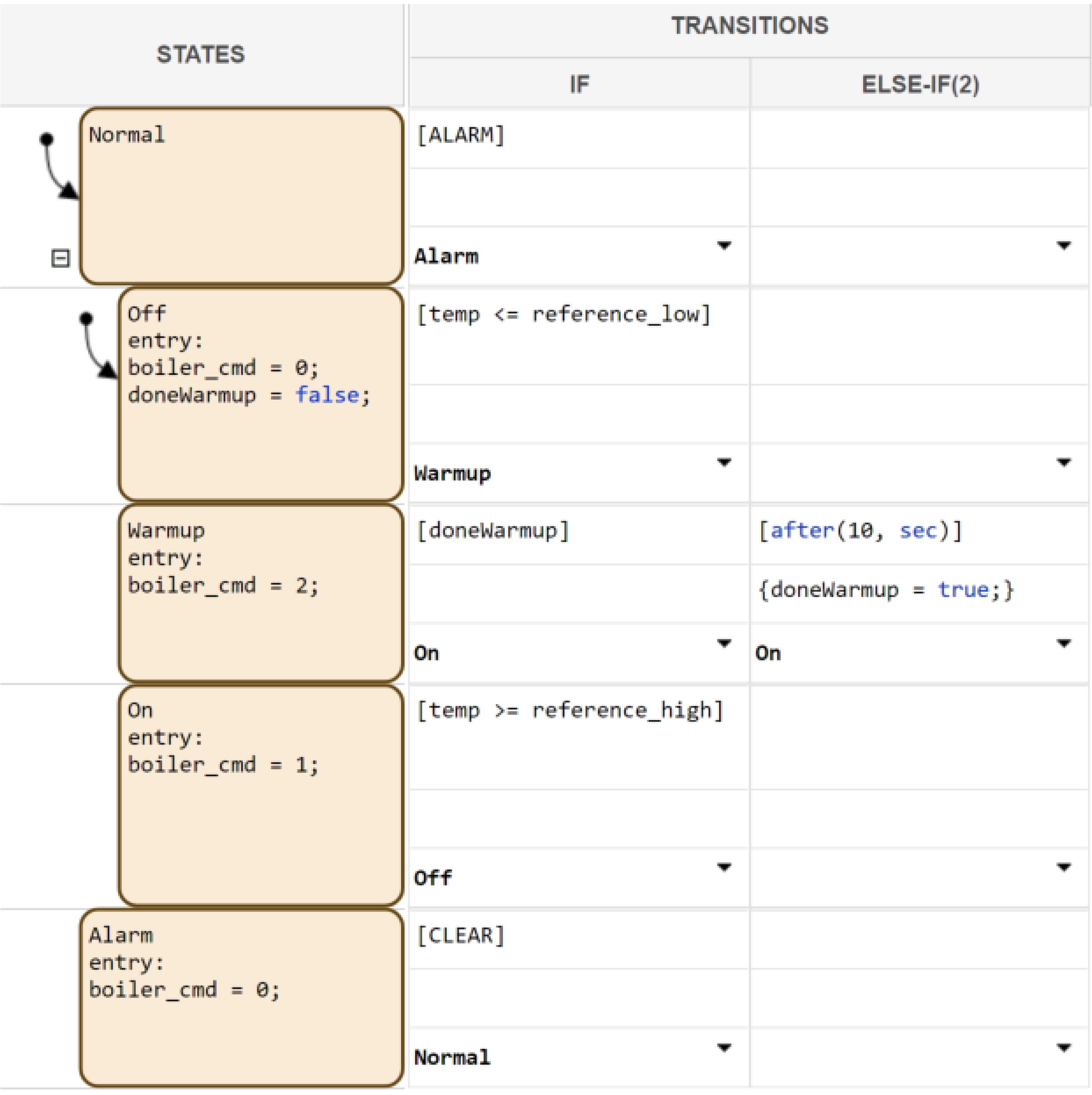
Use State Transition Tables to Express Sequential Logic in Tabular Form

The State Transition Table block represents a finite state machine for sequential modal logic in tabular format. Instead of drawing states and transitions in a Stateflow chart, you can use a state transition table to model a state machine in a concise, compact format that requires minimal maintenance of graphical objects.

In a state transition table, rows represent the states in your system. The transition columns specify the condition, condition action, and destination state for each outgoing transition from a state.



For example, this state transition table contains the modal logic for maintaining the temperature of a boiler between two set points, `reference_low` and `reference_high`. During normal operation, the boiler cycles between the states `Off`, `Warmup`, and `On`.

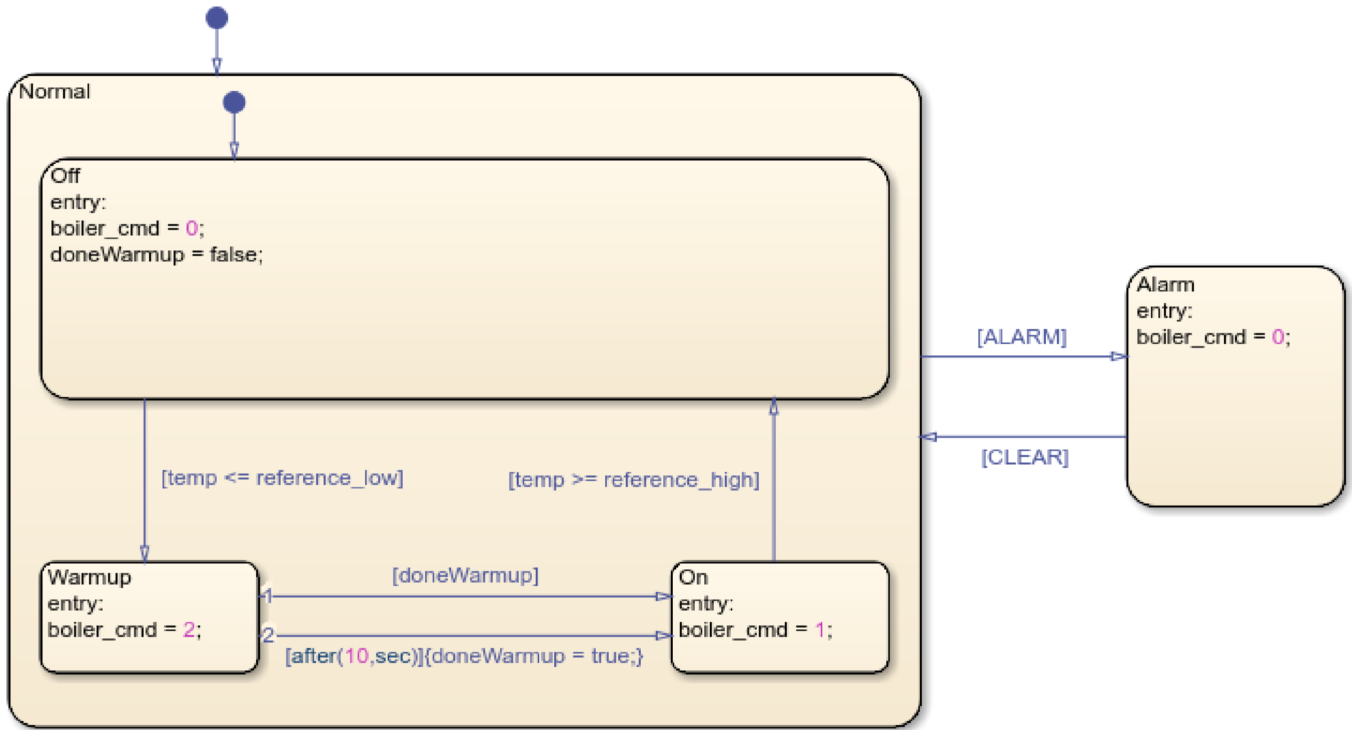


STATES	TRANSITIONS	
	IF	ELSE-IF(2)
Normal	[ALARM] **Alarm**	

Off
entry:
boiler_cmd = 0;
doneWarmup = false;
	[temp <= reference_low]
	
	Warmup	

Warmup entry: boiler_cmd = 2;	[doneWarmup] **On**	[after(10, sec)] {doneWarmup = true;} **On**
On entry: boiler_cmd = 1;	[temp >= reference_high] **Off**	
Alarm entry: boiler_cmd = 0;	[CLEAR] **Normal**	

The state transition table represents the same modal logic as this Stateflow chart.



For more information about this example, see “Model Bang-Bang Controller by Using a State Transition Table” on page 16-19.

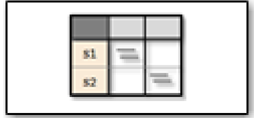
Program a State Transition Table

To create a State Transition Table:

1. Create a Simulink model that contains a State Transition Table block by calling the function `sfnew`.

   ```
   sfnew -STT
   ```

2. Double-click the State Transition Table block.

3. For each operating mode in your system, add a state row and enter a state label, as described in “Define Actions in a State” on page 1-27. To organize complex systems, define a hierarchy of states by adding child state rows below a parent state row.

   - To add a state row, select an existing state and, in the **Modeling** tab, choose from one of these options:

     - **Insert State Row** — Add a state at the same level of hierarchy.

  - **Insert Child State Row** — Add a state as a child of the selected state.
- To move a state row, click the top edge of the state and drag the state to a new location. As you drag the state, the editor displays a graphical cue that indicates the new position of the state.
- To model operating modes that are active at the same time, enable parallel (AND) decomposition in a parent state. For more information, see “Simulate Parallel States with a State Transition Table” on page 16-6.

4. To represent the direction of flow logic between states, specify conditions, condition actions, and destinations for the outgoing transitions from each state.

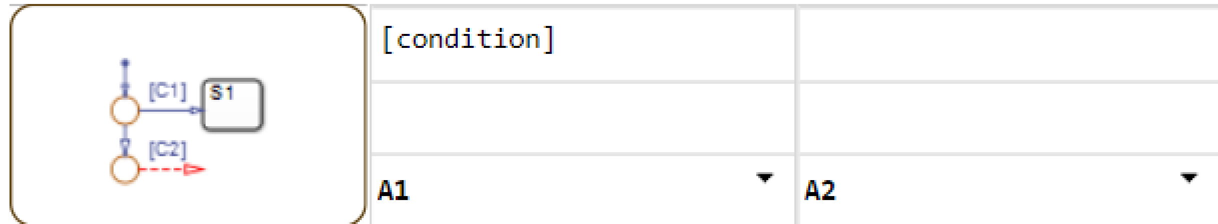
   - To add a transition column, right-click the header for an existing column and choose from one of these options:

     - **Append transition column** — Add a transition column to the right of the table.
     - **Insert transition column** — Add a transition column to the left of the selected column.
   - To move the condition, action, and destination cells for a transition, click the top edge of the condition cell and drag the transition left or right. The condition, action, and destination cells move together as a single unit.
   - To specify the destination of the transition, in the destination state cell, select the name of a state or one of these options:

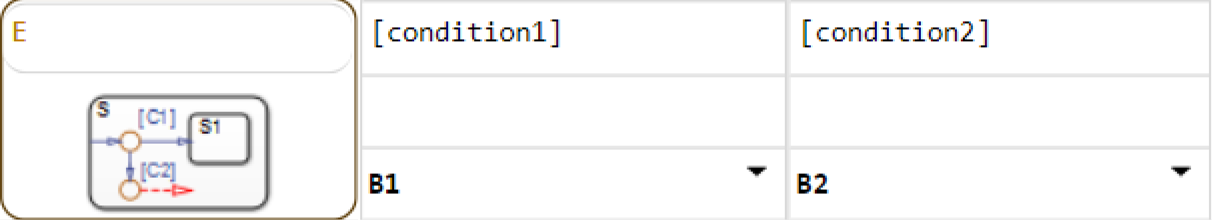
     - `$NEXT` — Create a transition to the next sibling state. This option is not available for the last substate in each level of the state hierarchy.
     - `$PREV` — Create a transition to the previous sibling state. This option is not available for the first substate in each level of the state hierarchy.
     - `$SELF` — Create a self-loop transition.
   - To comment out a transition, in the destination state cell, select `% IGNORE %`.

5. At each level of the hierarchy, mark the first state to become active when the parent becomes active. Select a state and, in the **Modeling** tab, select **Set as Default State**.

   Alternatively, to specify a default transition path with multiple branching points, in the **Modeling** tab, select **Insert Default Transition Row**. For example, this default transition row selects between two destinations, `A1` and `A2`, depending on the value of `condition`.



6. To specify an inner transition from the a parent state to one or more child states, in the **Modeling** tab, select **Insert Inner Transition Row**. For example, in this inner transition row, the input event `E` triggers an inner transition that selects between two destinations, `B1` and `B2`, depending on the values of `condition1` and `condition2`.



You must specify destination states in an inner transition row in the same order that the corresponding child states appear in the table. For instance, in the previous example, state B1 must appear above state B2.

7 If your system has inputs or outputs, or depends on any state variables, add input, output, and local data as described in "Add Stateflow Data" on page 10-2.

8 If your system reacts to event triggers or must trigger actions in your chart or other blocks in your model, add input and output events, as described in "Synchronize Model Components by Broadcasting Events" on page 12-2.

9 Connect the State Transition Table block to other blocks in the Simulink model by using input and output ports.

10 To simulate the model, click **Run**. During the simulation, the state transition table highlights the active states and transitions.

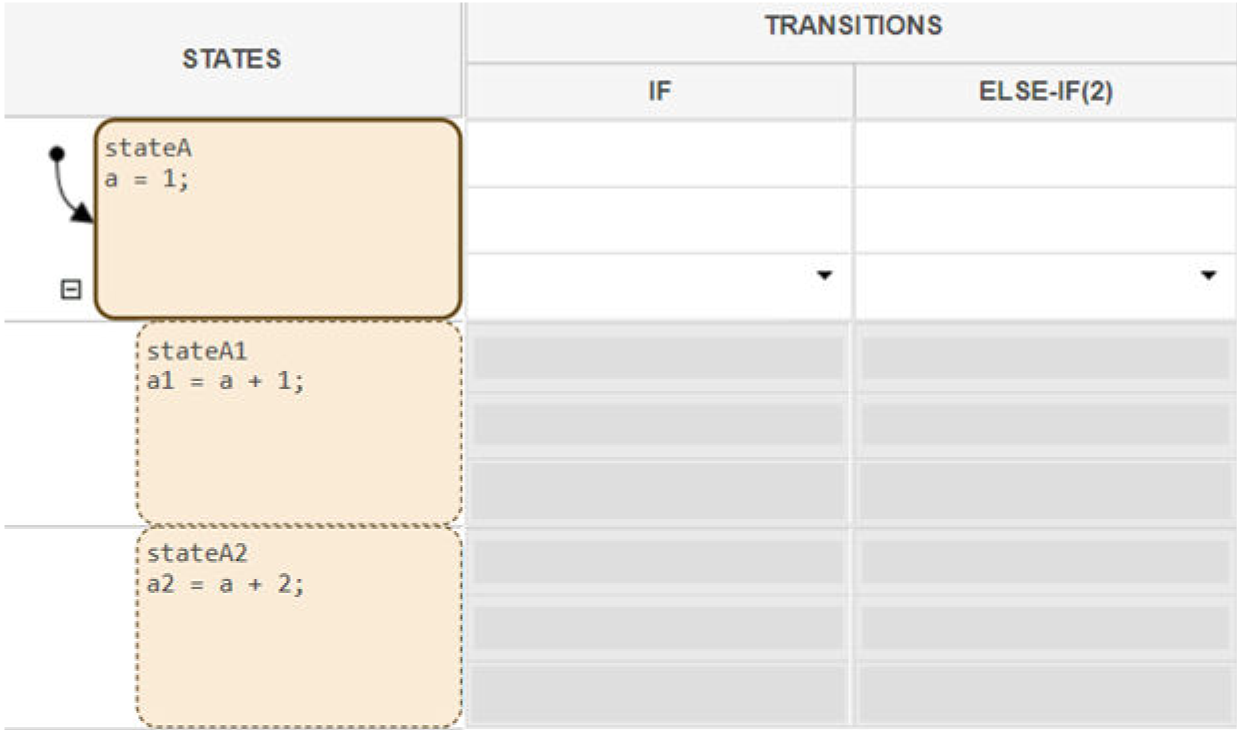
Simulate Parallel States with a State Transition Table

In Stateflow, the substates of states that use parallel decomposition are active simultaneously. For example, if State A uses parallel decomposition and has two substates, A1 and A2, both A1 and A2 are active at the same time. For more information on parallel states, see "Define Exclusive and Parallel Modes by Using State Decomposition" on page 1-35.

To use parallel states in State Transition Table blocks, enter the state transition table. To change the decomposition at the chart level, enter the state transition table, but do not select a state. In the **Modeling** tab, select **Decomposition** > **Parallel (AND)**. To give a state parallel decomposition, select the state whose decomposition you want to change. In the **Modeling** tab, select **Decomposition** > **Parallel (AND)**.

Note Before updating the decomposition of your chart or state to `Parallel (AND)`, all cells in the transition column of the state must be clear.

Parallel states have a dashed outline that indicates that they are active simultaneously.



Detect Errors in State Transition Tables

To run diagnostic checks on a state transition table, in the **Debug** tab, select **Update Model > Update Table**. The diagnostics tool statically parses the table to find errors such as:

- Unresolved symbols
- Unreachable states
- Default transition rows without an unconditional transition
- Transition cells with conditions or actions, but no destination
- Action text in a condition cell
- Inner transition rows that specify destination states in a different order than the corresponding states appear in the table

These error checks are also performed during simulation. For more information about debugging state transition tables, see “Debug Run-Time Errors in a State Transition Table” on page 16-16.

Specify Properties for State Transition Tables

State transition table properties specify how your state transition table interfaces with the Simulink model. You can modify these properties in the **Property Inspector**, the Model Explorer, or the State Transition Table properties dialog box.

To use the **Property Inspector**:

1. Open the State Transition Table block.

2 In the **Modeling** tab, under **Design Data**, select **Property Inspector**.
3 In the **Property Inspector**, edit the state transition table properties.

To use the Model Explorer:

1 In the **Modeling** tab, under **Design Data**, select **Model Explorer**.
2 In the **Model Hierarchy** pane, select the state transition table.
3 In the **Dialog** pane, edit the state transition table properties.

To use the State Transition Table properties dialog box:

1 Open the State Transition Table block.
2 In the **Modeling** tab, click **Table Properties**.
3 In the properties dialog box, edit the state transition table properties.

You can also modify state transition table properties programmatically by using `Stateflow.StateTransitionTableChart` objects. For more information about the Stateflow programmatic interface, see “Overview of the Stateflow API”.

Tip State transition table properties are a subset of the properties for Stateflow charts. For a description of each property, see “Specify Properties for Stateflow Charts” on page 1-19.

Guidelines for Using State Transition Tables

- State transition tables can use MATLAB or C as the action language. For more information, see “Differences Between MATLAB and C as Action Language Syntax” on page 15-4.
- State transition tables must have at least one state row and one transition column.
- State transition tables do not support these elements of Stateflow charts:
  - Supertransitions
  - Transition actions
  - Local events
  - Chart-level graphical, truth table, MATLAB, and Simulink functions

See Also

Blocks
State Transition Table

Objects
`Stateflow.StateTransitionTableChart`

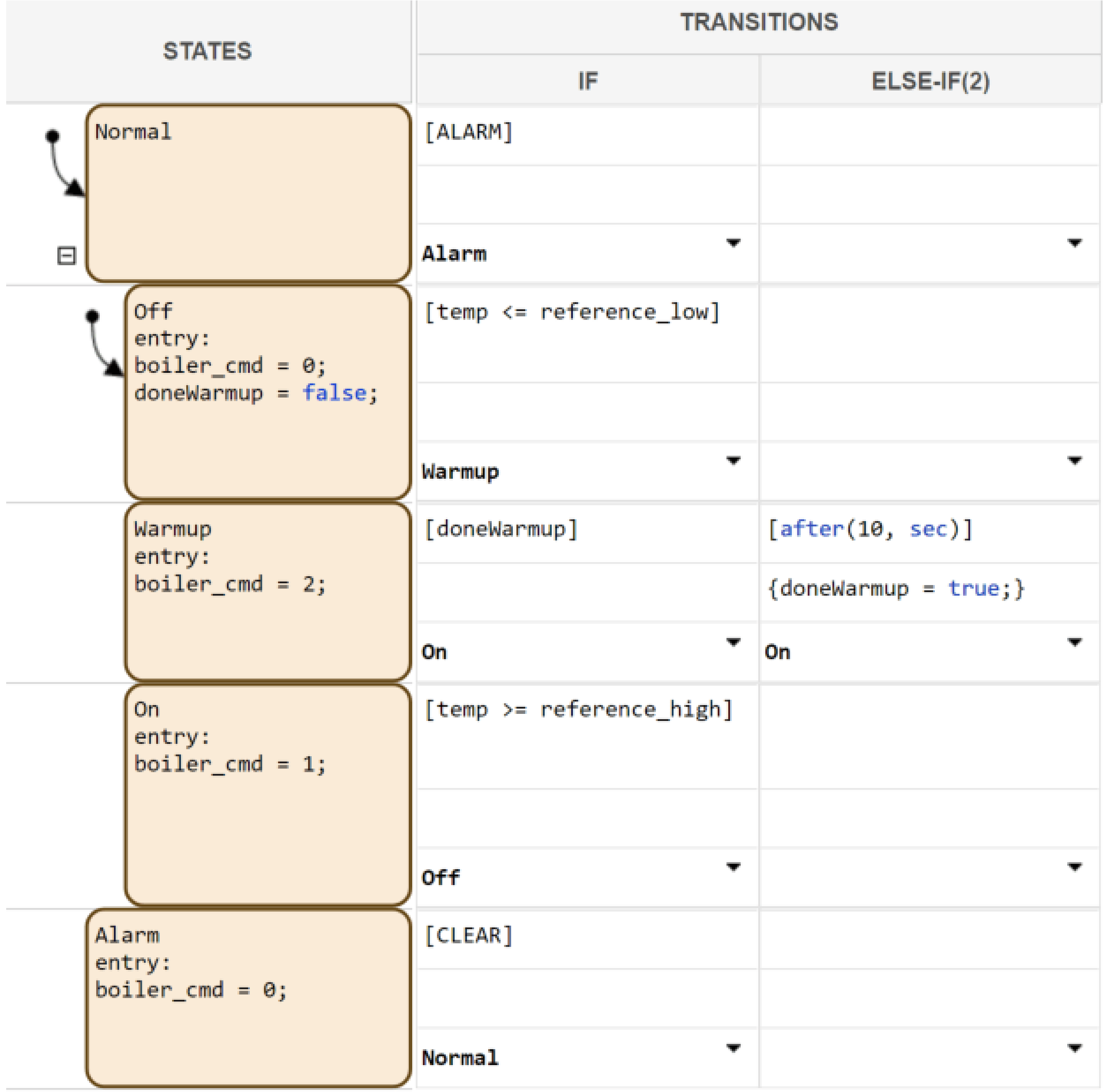
More About

- “Inspect the Design of State Transition Tables” on page 16-9
- “Debug Run-Time Errors in a State Transition Table” on page 16-16
- “Model Bang-Bang Controller by Using a State Transition Table” on page 16-19

Inspect the Design of State Transition Tables

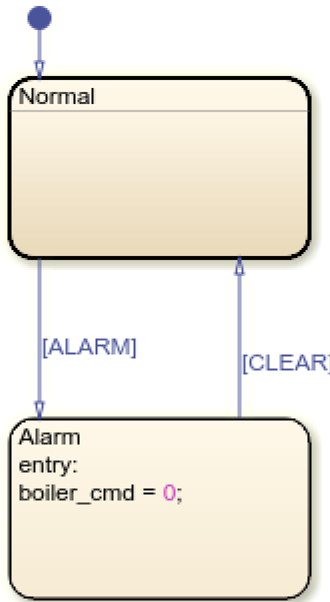
This example shows how to verify that the sequential modal logic in a state transition table behaves as intended. State transition tables model state machines in a concise, compact format that requires minimal maintenance of graphical objects. To examine the logic of a state transition table, you can display the contents of the table as a Stateflow® chart or as a state transition matrix. You can also use highlighting to mark the primary flow of logic in your table. For more information on state transition tables, see “Use State Transition Tables to Express Sequential Logic in Tabular Form” on page 16-2.

In this example, a state transition table contains the logic for maintaining the temperature of a boiler between two set points, `reference_low` and `reference_high`. During normal operation, the boiler cycles between the states `Off`, `Warmup`, and `On`. For more information on this example, see “Model Bang-Bang Controller by Using a State Transition Table” on page 16-19.

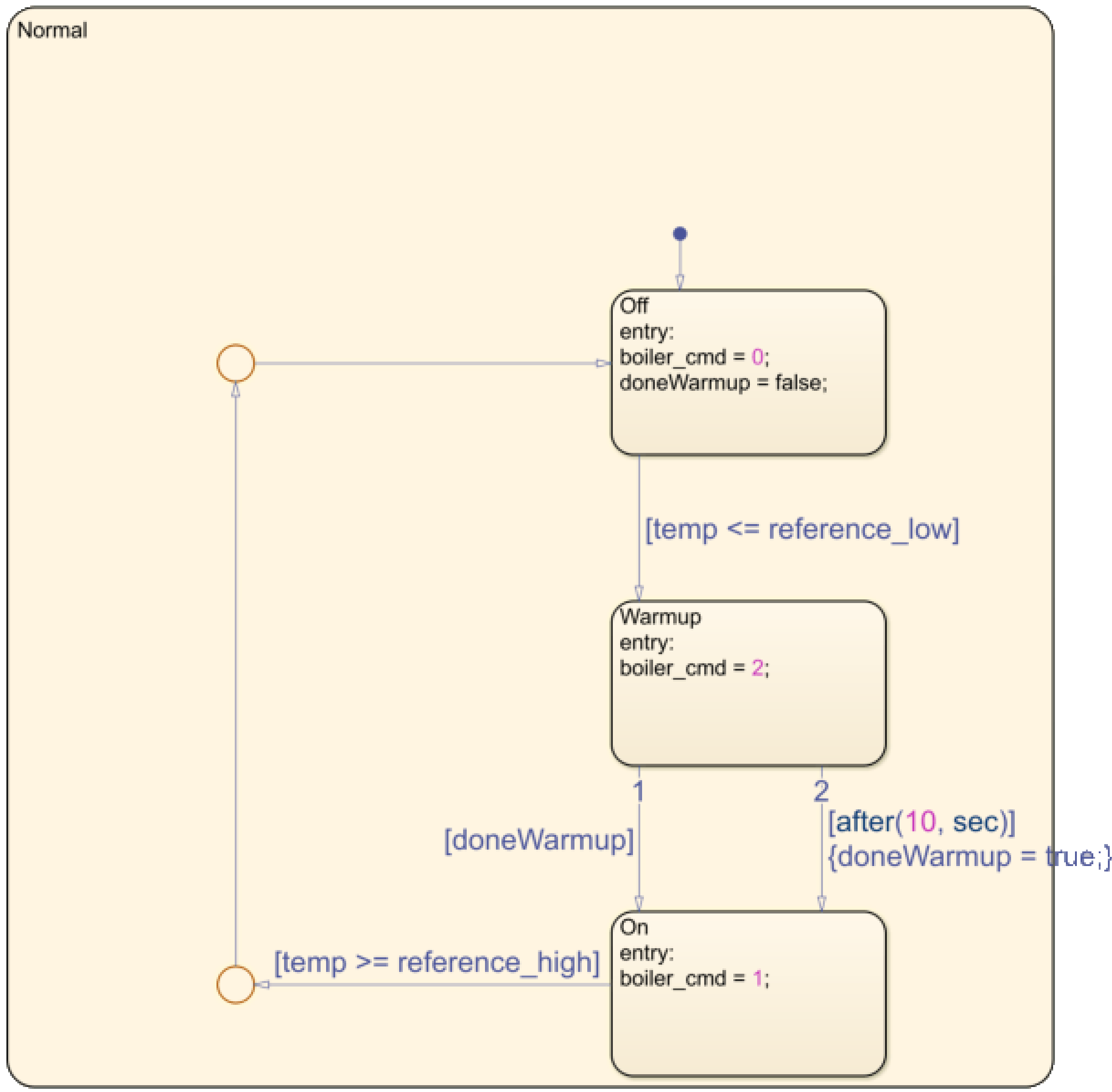


Display Table as Stateflow Chart

To see a read-only Stateflow chart that shows the logic for your state transition table in a graphical format, in the **Debug** tab, click **Show Auto Chart**.



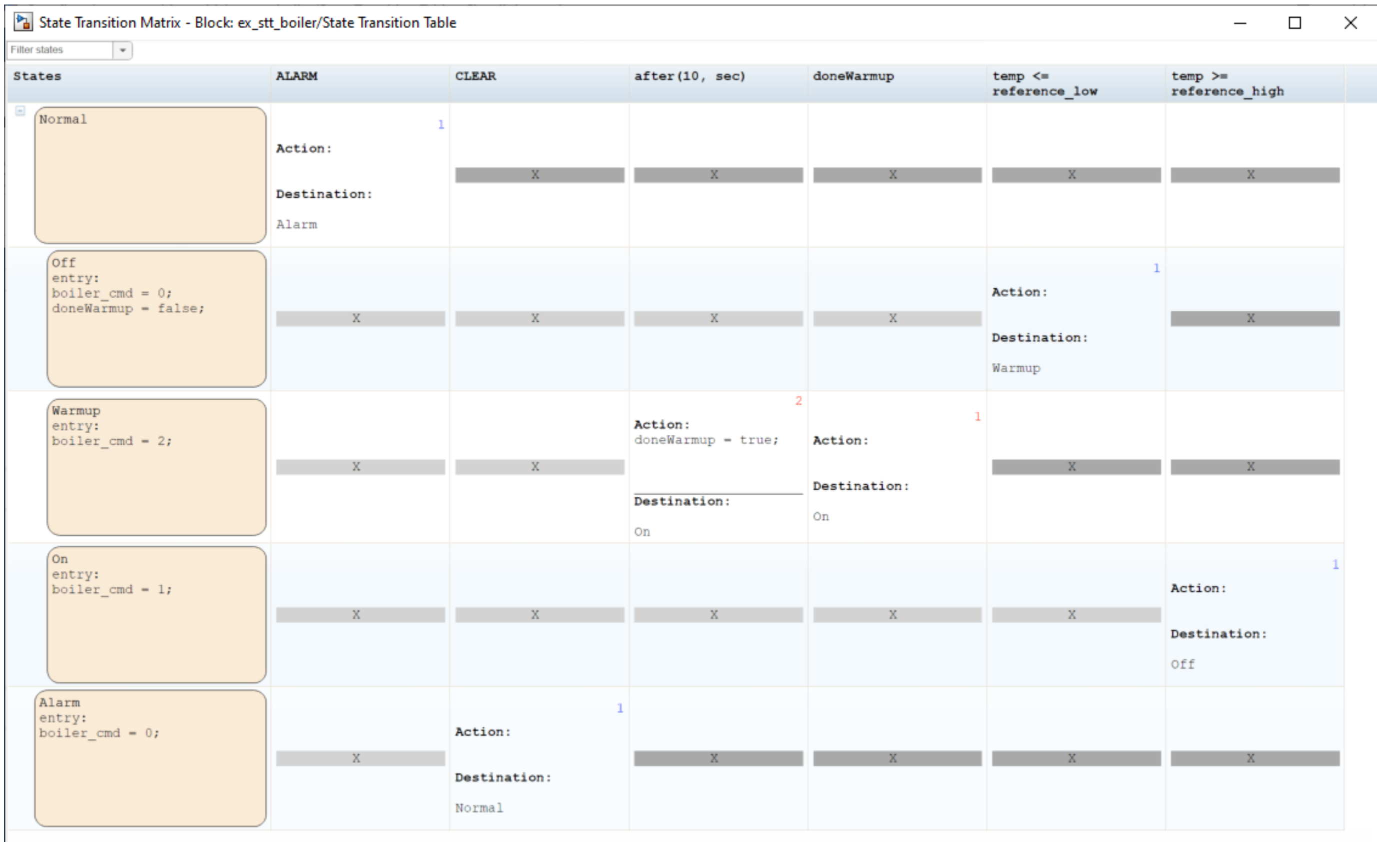
The automatically generated chart shows only the top-level states in your state transition table. To see the states and transitions at the next level of the chart hierarchy, double-click the subchart `Normal`.



If you modify the state transition table while the automatically generated chart is open, the chart reflects the changes that you make.

Display Table as State Transition Matrix

To generate a read-only state transition matrix that shows how the state transition table responds to various input conditions, in the **Debug** tab, click **Transition Matrix**.



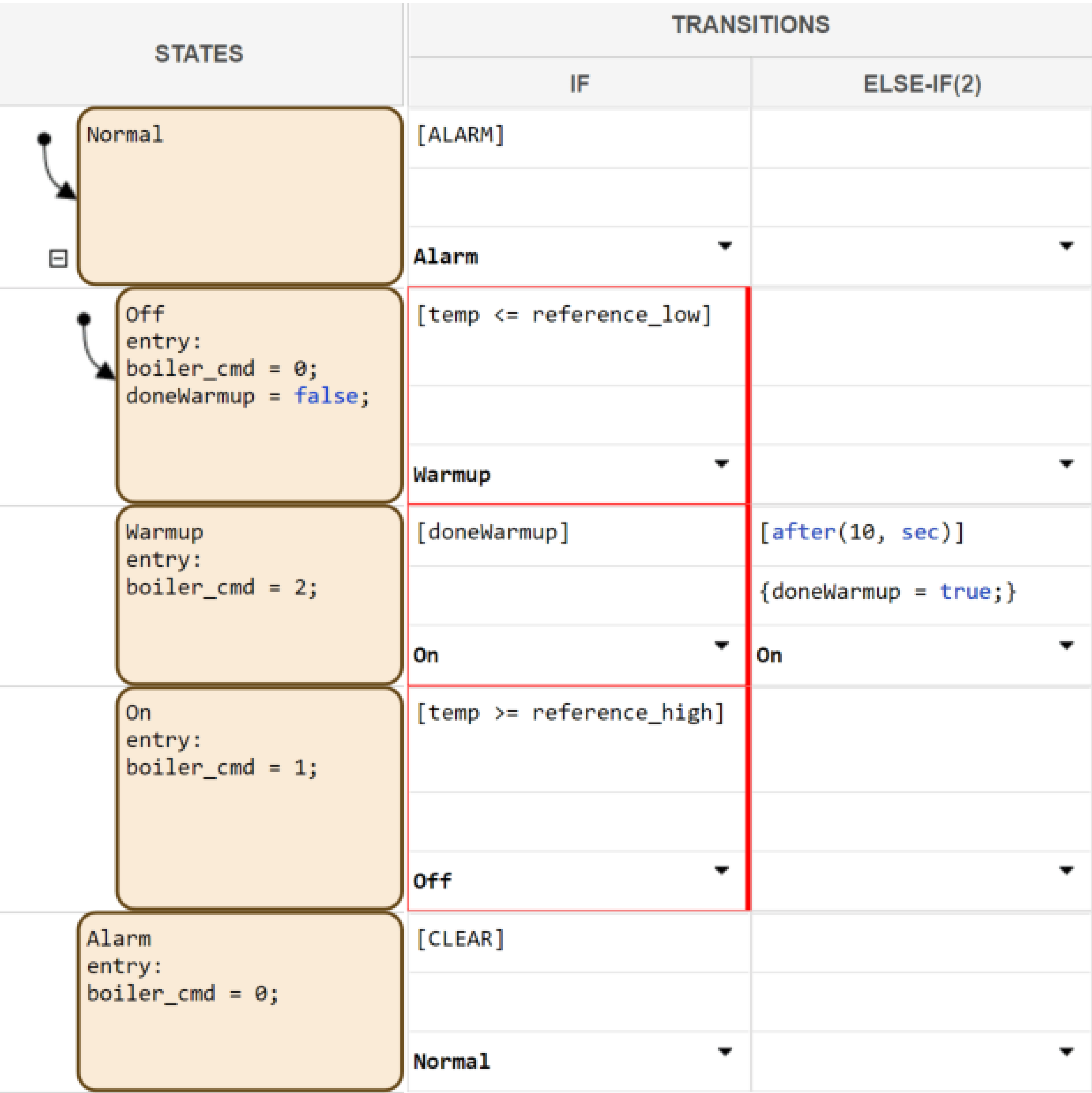
In the state transition matrix:

- Each row represents a state in the state transition table. The states appear in the same order as in the state transition table. To see only a subset of states, in the upper-left corner of the State Transition Matrix window, in the **Filter states** box, enter a state name or select a name from the drop-down list.
- Each column corresponds to a unique condition or event in the state transition table. The order of the columns depends on the number of states that respond to each condition or event. The conditions on the left of the matrix impact more states than conditions on the right of the matrix.
- Each cell lists the action and destination for a transition in the state transition table. An empty cell indicates that a condition or event does not impact a state. Empty cells to the left of a nonempty cell appear in light gray. Empty cells to the right of the last nonempty cell in a row appear in dark gray.
- The execution order of each transition appears in the upper-right corner of the cell. If the transitions in a row follow the same order as the columns of the matrix, the execution order appears in blue. Otherwise, the execution order appears in red.
- The state names, conditions, actions, and destinations are hyperlinks. To highlight the corresponding state, condition, action, or destination in the state transition table, click one of these hyperlinks.

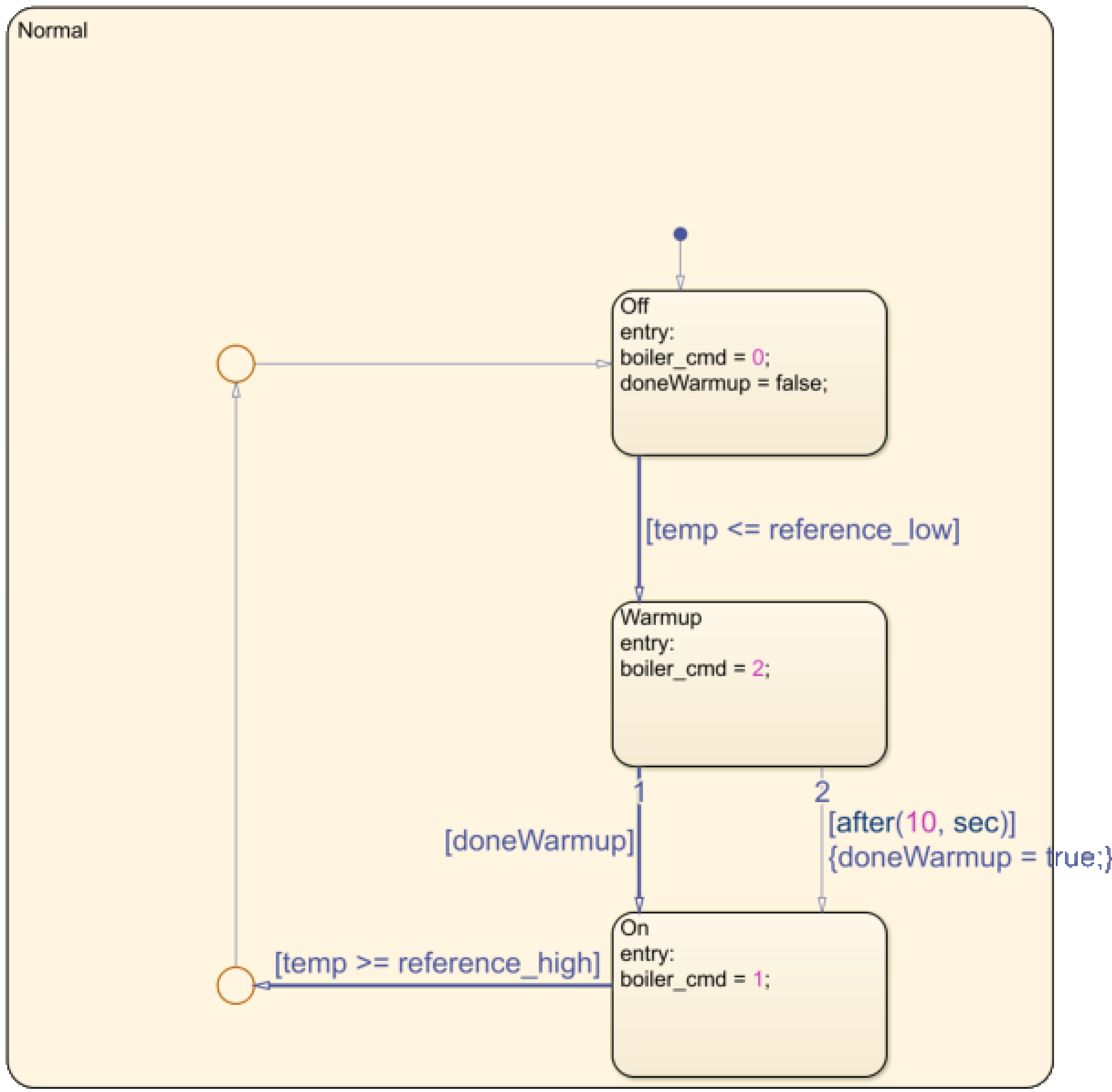
If you modify the state transition table while the state transition matrix is open, the matrix does not reflect the changes that you make. To see your changes, close and regenerate the matrix.

Highlight Primary Logic Flow

To mark a sequence of transitions that represent the primary flow of logic in your state transition table, right-click each transition cell and select **Mark as primary transition**. For example, you can mark the transitions between the Off, Warmup, and On states that describe the normal operation of the boiler. A red border indicates the primary transition cells.



The automatically generated chart highlights the primary transitions in blue.



The highlighting persists across MATLAB® sessions. To remove the highlighting, right-click each transition cell and clear the **Mark as primary transition** check box.

See Also

More About

- "Use State Transition Tables to Express Sequential Logic in Tabular Form" on page 16-2
- "Debug Run-Time Errors in a State Transition Table" on page 16-16
- "Model Bang-Bang Controller by Using a State Transition Table" on page 16-19

Debug Run-Time Errors in a State Transition Table

A state transition table represents a finite state machine for sequential modal logic in tabular format. Instead of drawing states and transitions in a Stateflow chart, you can use a state transition table to model a state machine in a concise, compact format that requires minimal maintenance of graphical objects. For more information, see “Use State Transition Tables to Express Sequential Logic in Tabular Form” on page 16-2.

Create the Model and the State Transition Table

1 Create a Simulink model with a new State Transition Table.

```
sfnew -STT
```

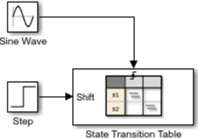
2 Add the following states and transitions to your table:

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

3. Add two inputs on page 10-2 from Simulink:

   - An event called SWITCH with a scope of **Input from Simulink** and a **Rising** edge trigger.
   - A data called Shift with a scope of **Input from Simulink**.

4. In the model view, connect a Sine Wave block as the SWITCH event and a Step block as the Shift data for your State Transition Table.



In the model, there is an event input and a data input. A Sine Wave block generates a repeating input event that corresponds with the Stateflow event SWITCH. The Step block generates a repeating pattern of 1 and 0 that corresponds with the Stateflow data object Shift. Ideally, the SWITCH event occurs at a frequency that allows at least one cycle through First, Second, and Third.

Debug the State Transition Table

To debug the table in “Create the Model and the State Transition Table” on page 16-16, follow these steps:

1. Right-click the Power_off state, and select **Set Breakpoint** > **On State Entry**.
2. Start the simulation.

   Because you specified a breakpoint on Power_off, execution stops at that point.

3. Move to the next step by clicking the Step In button, .
4. To see the data used and the current values, hover your cursor over the different table cells.

   Continue clicking the Step In button and watching the animating states. After each step, watch the chart animation to see the sequence of execution. Use the tooltips to see the data values.

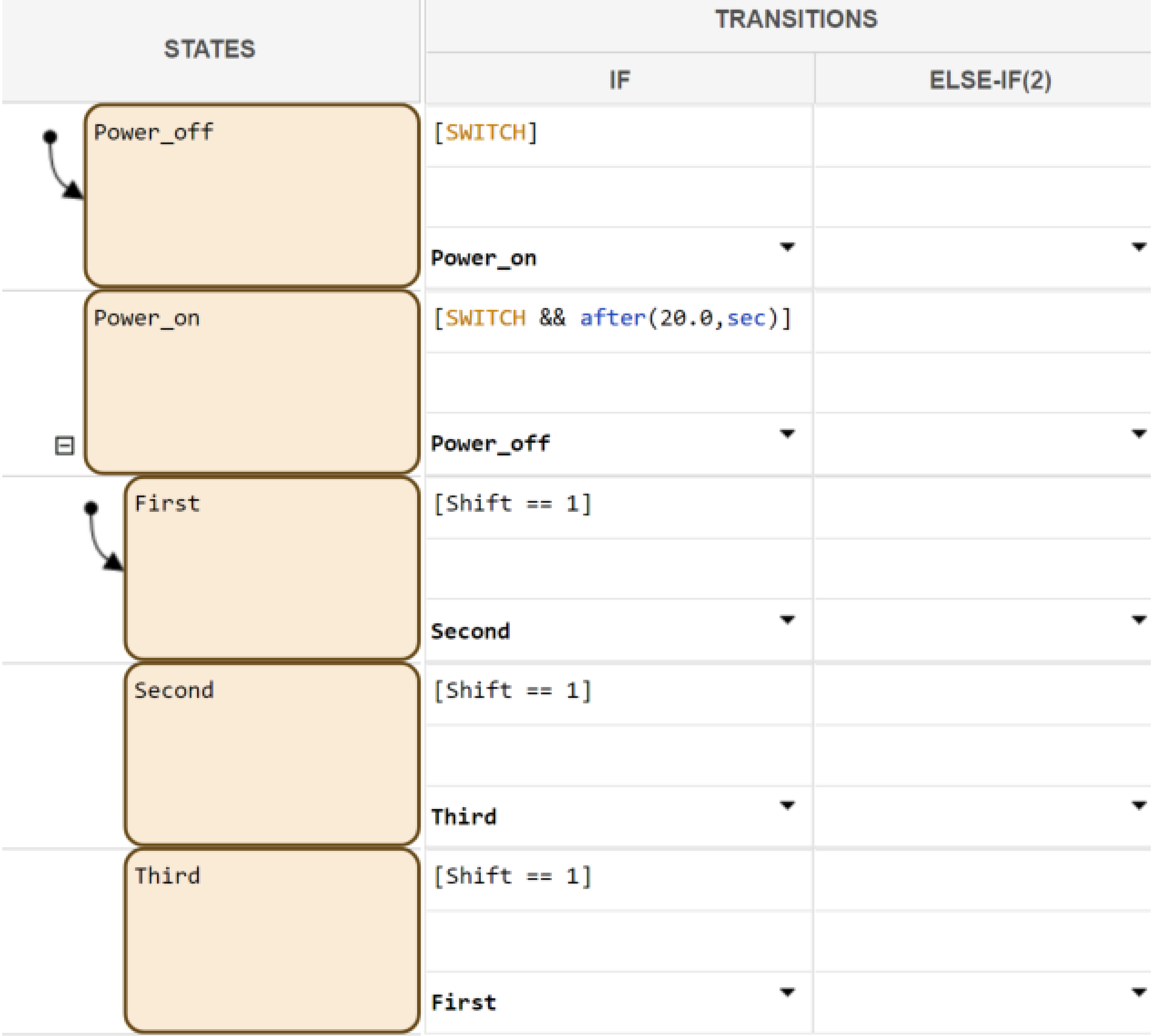
Single-stepping shows that the loop from First to Second to Third inside the state Power_on does not occur. The transition from Power_on to Power_off takes priority.

Correct the Run-Time Error

In “Debug the State Transition Table” on page 16-17, you step through a simulation of a state transition table and find an error. The event SWITCH drives the simulation, but the simulation time passes too quickly for the input data object Shift to have an effect.

To correct this error:

1 Stop the simulation so that you can edit the table.
2 Add the condition `after(20.0, sec)` to the transition from `Power_on` to `Power_off`.



Now the transition from `Power_on` to `Power_off` does not occur until 20 seconds have passed.
3 Begin simulation.
4 Click the Step In button repeatedly to observe the fixed behavior.

See Also

Related Examples

- "Use State Transition Tables to Express Sequential Logic in Tabular Form" on page 16-2
- "Model Bang-Bang Controller by Using a State Transition Table" on page 16-19

Model Bang-Bang Controller by Using a State Transition Table

A state transition table represents a finite state machine for sequential modal logic in tabular format. Instead of drawing states and transitions in a Stateflow chart, you can use a state transition table to model a state machine in a concise, compact format that requires minimal maintenance of graphical objects. For more information, see “Use State Transition Tables to Express Sequential Logic in Tabular Form” on page 16-2.

Design Requirements

This example shows how to model a bang-bang controller for temperature regulation of a boiler, using a state transition table. The controller must turn the boiler on and off to meet the following design requirements:

- High temperature cannot exceed 25 degrees Celsius.
- Low temperature cannot fall below 23 degrees Celsius.
- Steady-state operation requires a warm-up period of 10 seconds.
- When the alarm signal sounds, the boiler must shut down immediately.
- When the all-clear signal sounds, the boiler can turn on again.

Identify System Attributes

You can identify the operating modes and data requirements for the bang-bang controller based on its design requirements.

Operating Modes

The high-level operating modes for the boiler are:

- Normal operation, when no alarm signal sounds.
- Alarm state, during an alarm signal.

During normal operation, the boiler can be in one of three states:

- Off, when the temperature is above 25 degrees Celsius.
- Warm-up, during the first 10 seconds of being on.
- On, steady-state after 10 seconds of warm-up, when the temperature is below 23 degrees Celsius.

Data Requirements

The bang-bang controller requires the following data.

Scope	Description	Variable Name
Input	High temperature set point	`reference_high`
Input	Low temperature set point	`reference_low`
Input	Alarm indicator	`ALARM`
Input	All-clear indicator	`CLEAR`

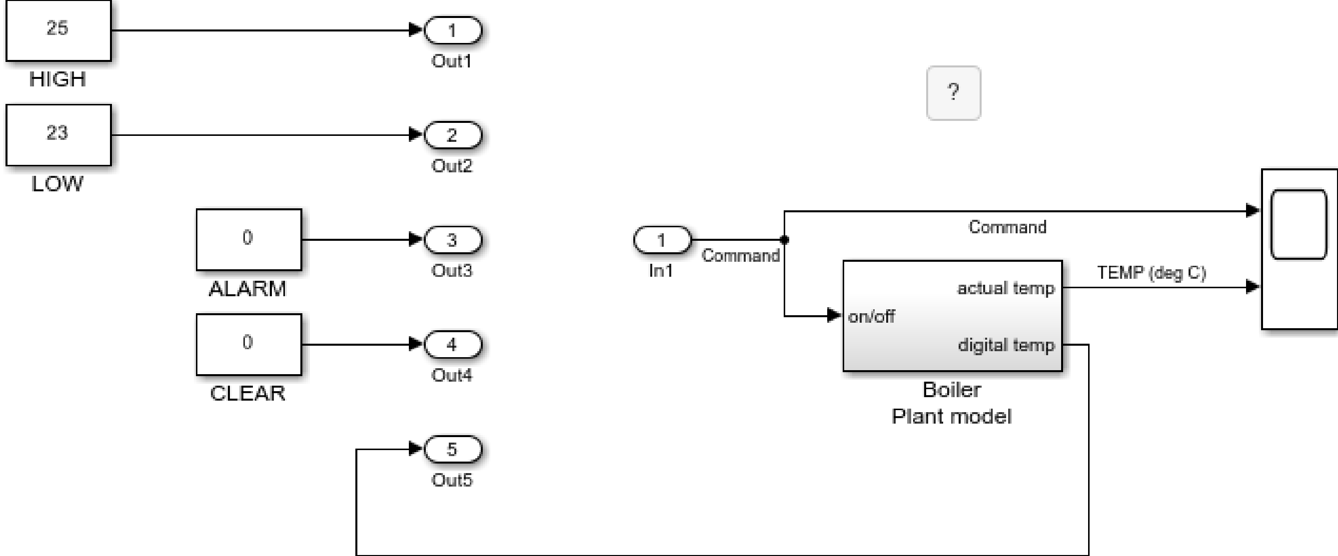
Scope	Description	Variable Name
Input	Current temperature of the boiler	`temp`
Local	Indicator that the boiler completed warm-up	`doneWarmup`
Output	Command to set the boiler mode: off, warm-up, or on	`boiler_cmd`

Add a New State Transition Table

In this exercise, you add a state transition table to a Simulink model that contains the required Simulink blocks, except for the bang-bang controller.

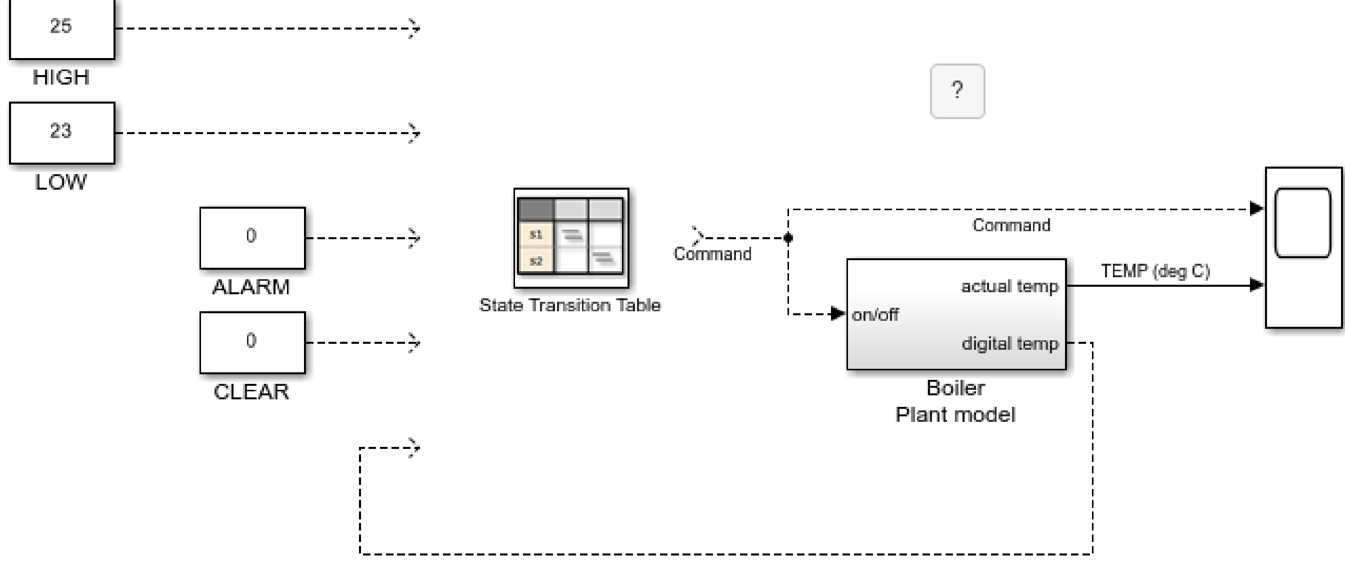
To implement the model yourself, follow these steps. Otherwise, you can open the completed model.

1. Open the example.



2. Delete the five output ports and the single input port.

3. Add a State Transition Table block to the model.



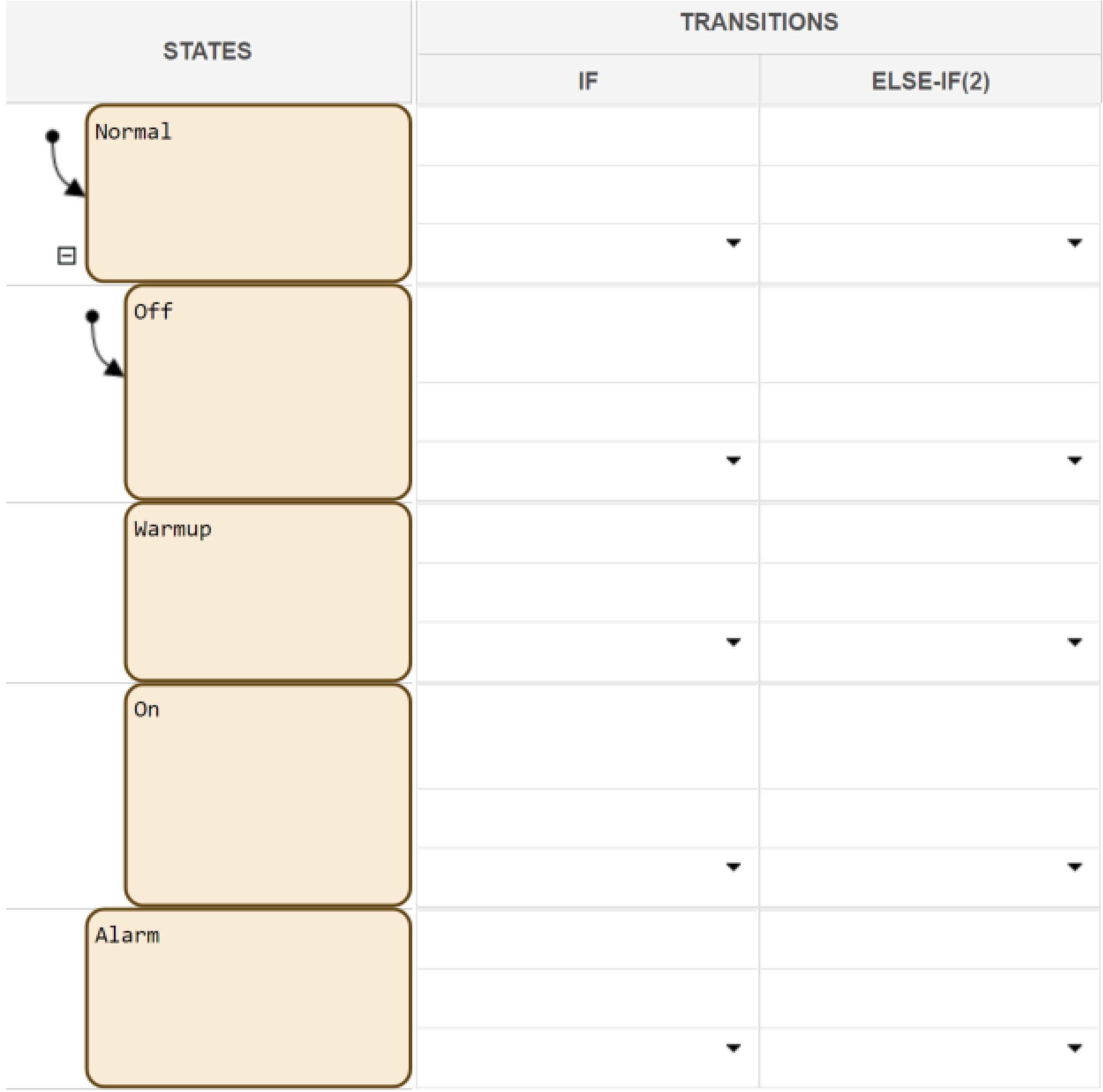
Add States and Hierarchy

To represent the operating modes of the boiler, add states and hierarchy to the state transition table.

1. Open the state transition table.
2. Represent the high-level operating modes: normal and alarm.
   - **a** Double-click `state1` and rename it `Normal`.
   - **b** Double-click `state2` and rename it `Alarm`.
3. Represent the three states of normal operation as substates of `Normal`:
   - **a** Right-click the `Normal` state, select **Insert Row** > **Child State Row**, and name the new state `Off`.
   - **b** Repeat step **a** two more times to create the child states `Warmup` and `On`, in that order.

   By default, when there is ambiguity, the top exclusive (OR) state at every level of hierarchy becomes active first. For this reason, the `Normal` and `Off` states appear with default transitions. This configuration meets the design requirements for this model. To set a default state, right-click the state and select **Set to default**.

Your state transition table looks like this table.



Now you are ready to specify actions for each state.

Specify State Actions

To describe the behavior that occurs in each state, specify state actions in the table. In this exercise, you initialize modes of operation as the boiler enters normal and alarm states, using the variables `boiler_cmd` and `doneWarmup` (described in “Data Requirements” on page 16-19).

1 In the following states, click after the state name, press **Enter**, and type the specified entry actions.

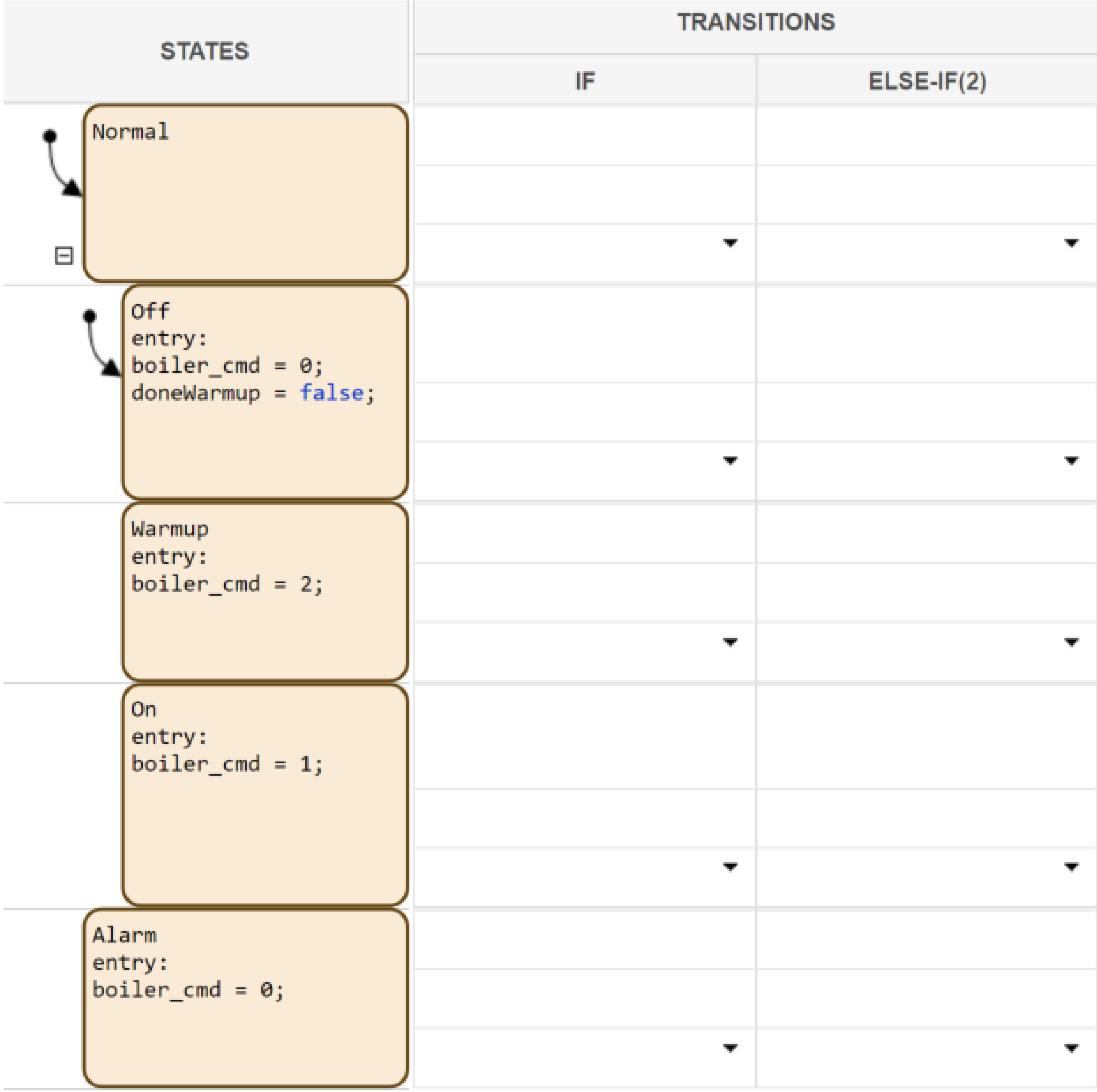
In State:	Type:	Resulting Behavior

`Off`	`entry:`
	`boiler_cmd = 0;`
	`doneWarmup = false;`
		Turns off the boiler and indicates that the boiler has not warmed up.

`Warmup`	`entry:` `boiler_cmd = 2;`	Starts warming up the boiler.
`On`	`entry:` `boiler_cmd = 1;`	Turns on the boiler.
`Alarm`	`entry:` `boiler_cmd = 0;`	Turns off the boiler.

2 Save the state transition table.

Your state transition table looks like this table.



Now you are ready to specify the conditions and actions for transitioning from one state to another state.

Specify Transition Conditions and Actions

To indicate when to change from one operating mode to another, specify transition conditions and actions in the table. In this exercise, you construct statements using variables described in “Data Requirements” on page 16-19.

1 In the `Normal` state row, enter:

if
[ALARM]
Alarm

During simulation:

a When first entered, the chart activates the `Normal` state.

b At each time step, normal operation cycles through the `Off`, `Warmup`, and `On` states until the ALARM condition is true.

c When the ALARM condition is true, the boiler transitions to the `Alarm` state and shuts down immediately.

2 In the `Off` state row, enter:

if
[temp <= reference_low]
Warmup

During simulation, when the current temperature of the boiler drops below 23 degrees Celsius, the boiler starts to warm up.

3 In the `Warmup` state row, enter:

if	else-if
[doneWarmup]	[after(10, sec)]
	{doneWarmup = true;}
On	**On**

During simulation, the boiler warms up for 10 seconds and then transitions to the `On` state.

4 In the `On` state row, enter:

if
[temp >= reference_high]
Off

During simulation, when the current temperature of the boiler rises above 25 degrees Celsius, the boiler shuts off.

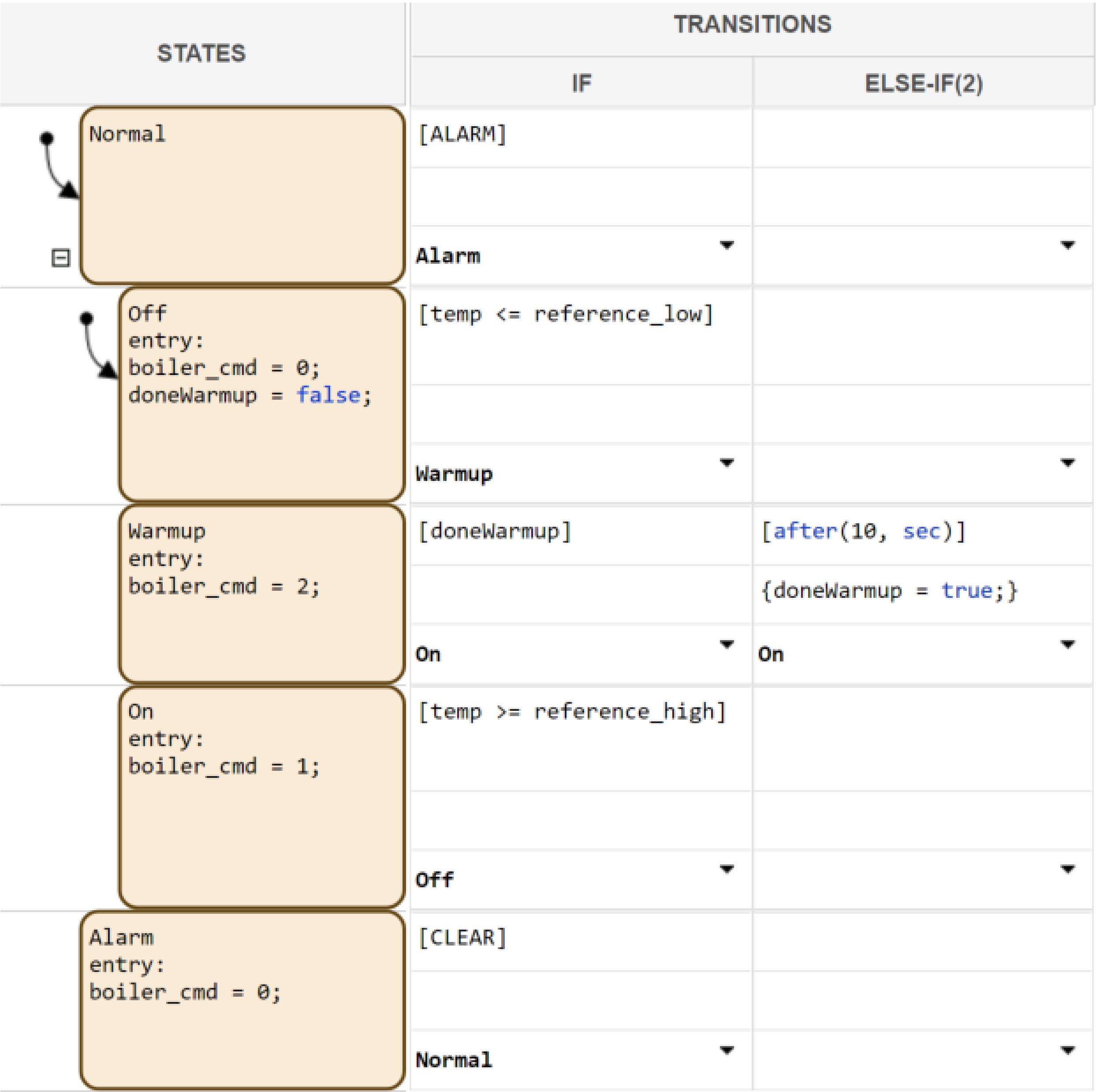
5 In the `Alarm` state row, enter:

if
[CLEAR]
Normal

During simulation, when the all-clear condition is true, the boiler returns to normal mode.

6 Save the state transition table.

Your state transition table looks like this table.



STATES	TRANSITIONS	
	IF	ELSE-IF(2)
Normal	[ALARM] Alarm	

Off
entry:
boiler_cmd = 0;
doneWarmup = false;
	[temp <= reference_low]
	
	Warmup	

Warmup entry: boiler_cmd = 2;	[doneWarmup] On	[after(10, sec)] {doneWarmup = true;} On
On entry: boiler_cmd = 1;	[temp >= reference_high] Off	
Alarm entry: boiler_cmd = 0;	[CLEAR] Normal	

Now you are ready to add data definitions using the Symbol Wizard.

Define Data

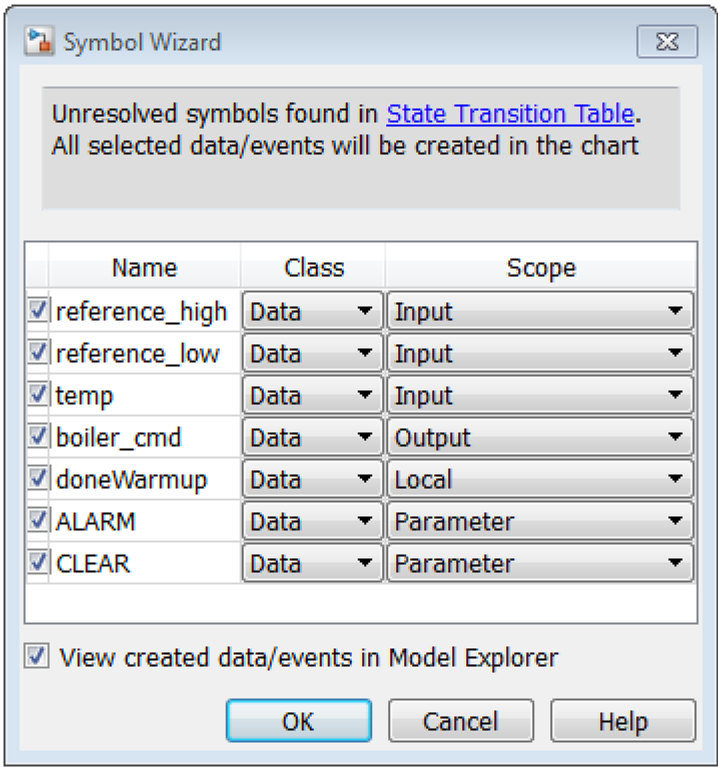
When you create a state transition table that uses MATLAB syntax, there are language requirements for C/C++ code generation. One of these requirements is that you define the size, type, and complexity of all MATLAB variables so that their properties can be determined at compile time. Even

though you have not yet explicitly defined the data in your state transition table, you can use the Symbol Wizard. During simulation, the Symbol Wizard alerts you to unresolved symbols, infers their properties, and adds the missing data to your table.

1 In the Simulink model , select **Run**.

Two dialog boxes appear:

- The Diagnostic Viewer indicates that you have unresolved symbols in the state transition table.
- The Symbol Wizard attempts to resolve the missing data. The wizard correctly infers the scope of all data except for the inputs `ALARM and CLEAR`.



2 In the Symbol Wizard, correct the scopes of `ALARM` and `CLEAR` by selecting **Input** from their Scope drop-down lists.

3 When the Model Explorer opens, verify that the Symbol Wizard added all required data definitions correctly.

Some of the inputs are assigned to the wrong ports.

4 In the Contents pane of the Model Explorer, reassign input ports as follows:

Assign:	To Port:
`reference_low`	2
`reference_high`	1
`temp`	5
`ALARM`	3

Assign:	To Port:
CLEAR	4

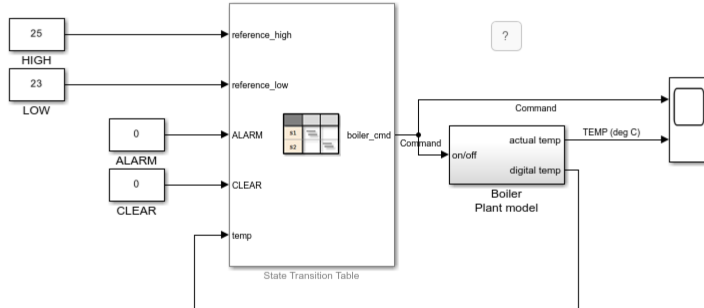
5 Save the state transition table.

6 Close the Diagnostic Viewer and the Model Explorer.

In the Simulink model, the inputs and outputs that you defined appear in the State Transition Table block. Now you are ready to connect these inputs and outputs to the Simulink signals and run the model.

Connect the Transition Table and Run the Model

1 In the Simulink model, connect the state transition table to the Simulink inputs and outputs:



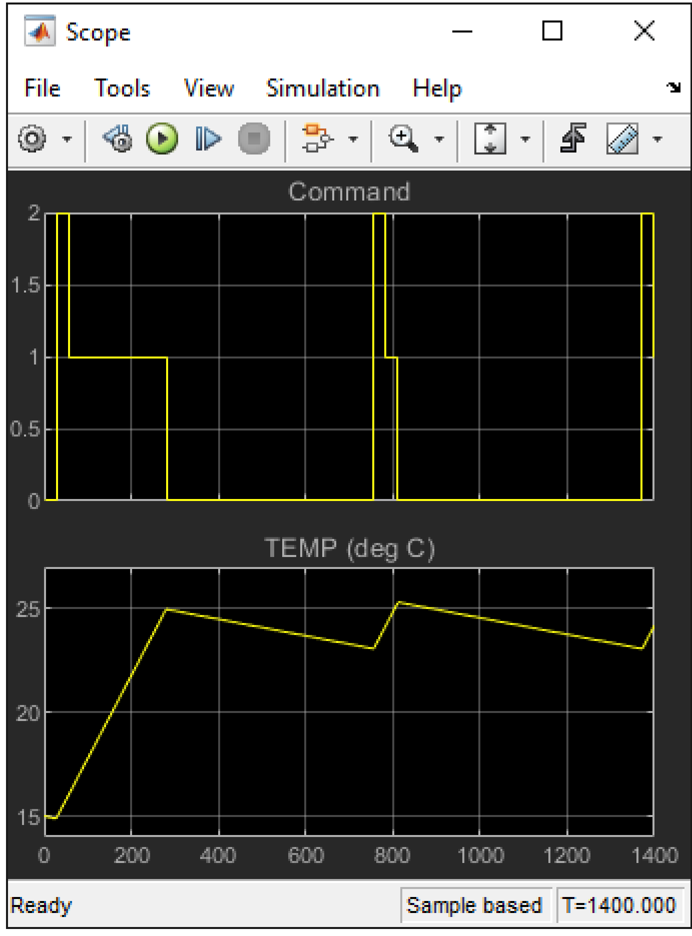
2 Save the model.

3 Reopen your state transition table.

4 Start the simulation by selecting **Run**.

As the simulation runs, you can watch the animation in the state transition table activate different states.

The following output appears in the Scope block.



When performing interactive debugging, you can set breakpoints on different states and view the data values at different points in the simulation. For more information about debugging, see “Set Breakpoints to Debug Charts” on page 30-2.

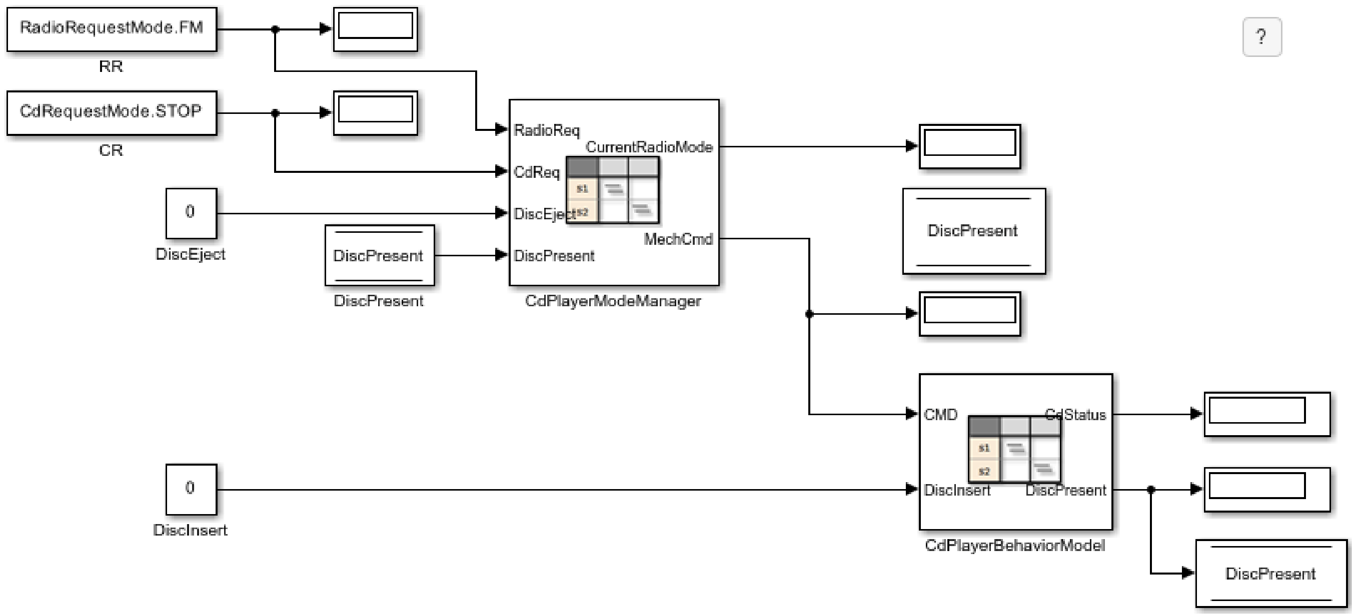
See Also

More About

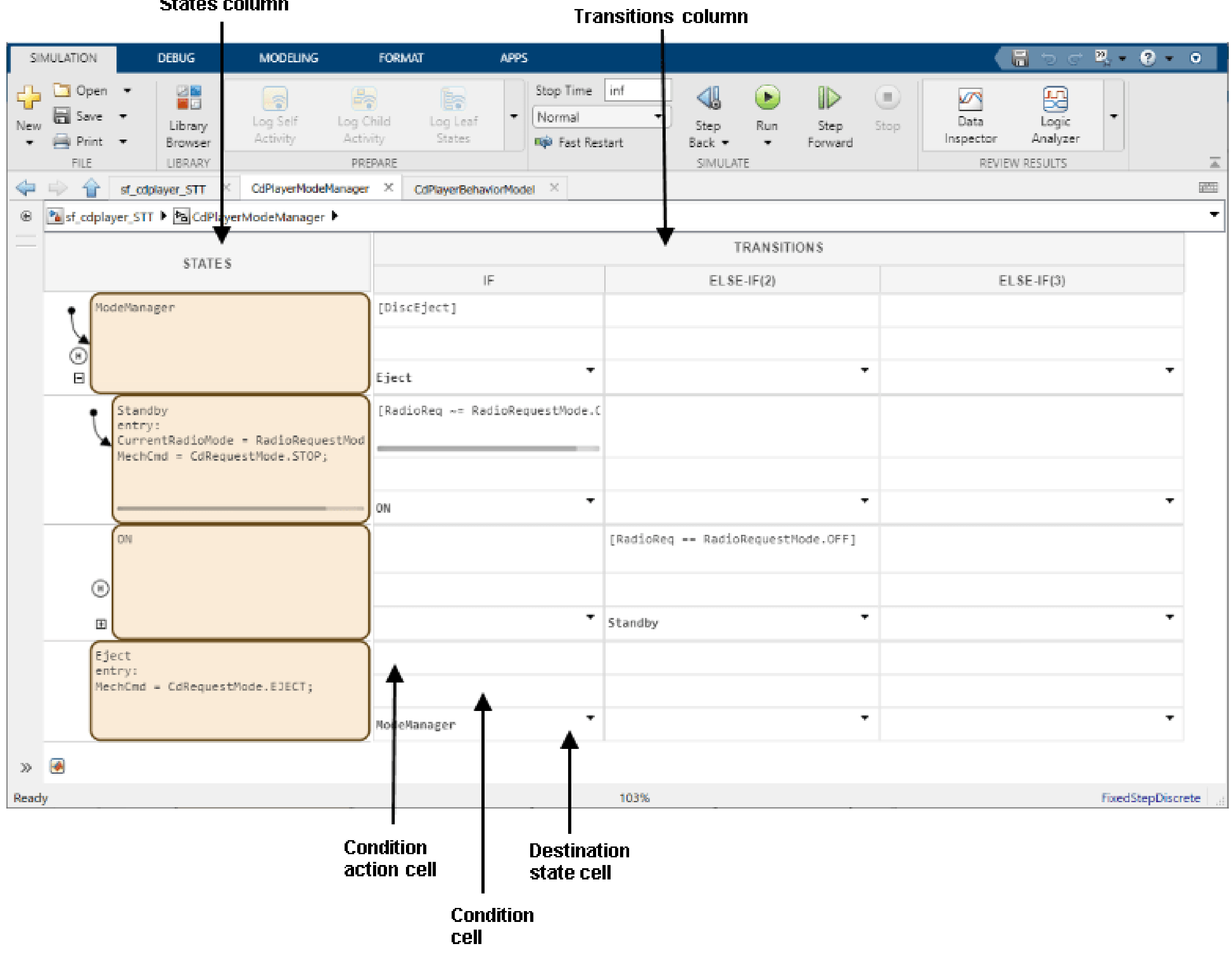
- “Use State Transition Tables to Express Sequential Logic in Tabular Form” on page 16-2
- “Inspect the Design of State Transition Tables” on page 16-9
- “Debug Run-Time Errors in a State Transition Table” on page 16-16

Modeling a CD Player/Radio Using State Transition Tables

This example shows a simple model of a CD Player/Radio logic that uses State Transition Tables in Stateflow®. This model is a reimplementation of `sf_cdplayer` using State Transition Tables.



The heart of the logic for controlling the CD Player/Radio is in the `CdPlayerModeManager` chart, which is designed using a State Transition Table. The table is used to represent modal logic in tabular form. It allows us to define various states and their hierarchical structure along with the transitions between the states.



State Cells

The left most column represents all the states in the table. Notice that states can be nested hierarchically. You can choose one of the states to be the default state at any given level. This is represented by a default transition drawn to the left of the state. You can also add a default transition row if there is logic involved in choosing the first state to enter.

You can add a history junction to a given state by right-clicking the state and selecting **Add history junction**. This enables the state to remember the last active state when it is re-entered instead of choosing the default state. For example, states **ModeManager** and **ON** have history junctions. A Stateflow chart can be automatically generated from this table view.

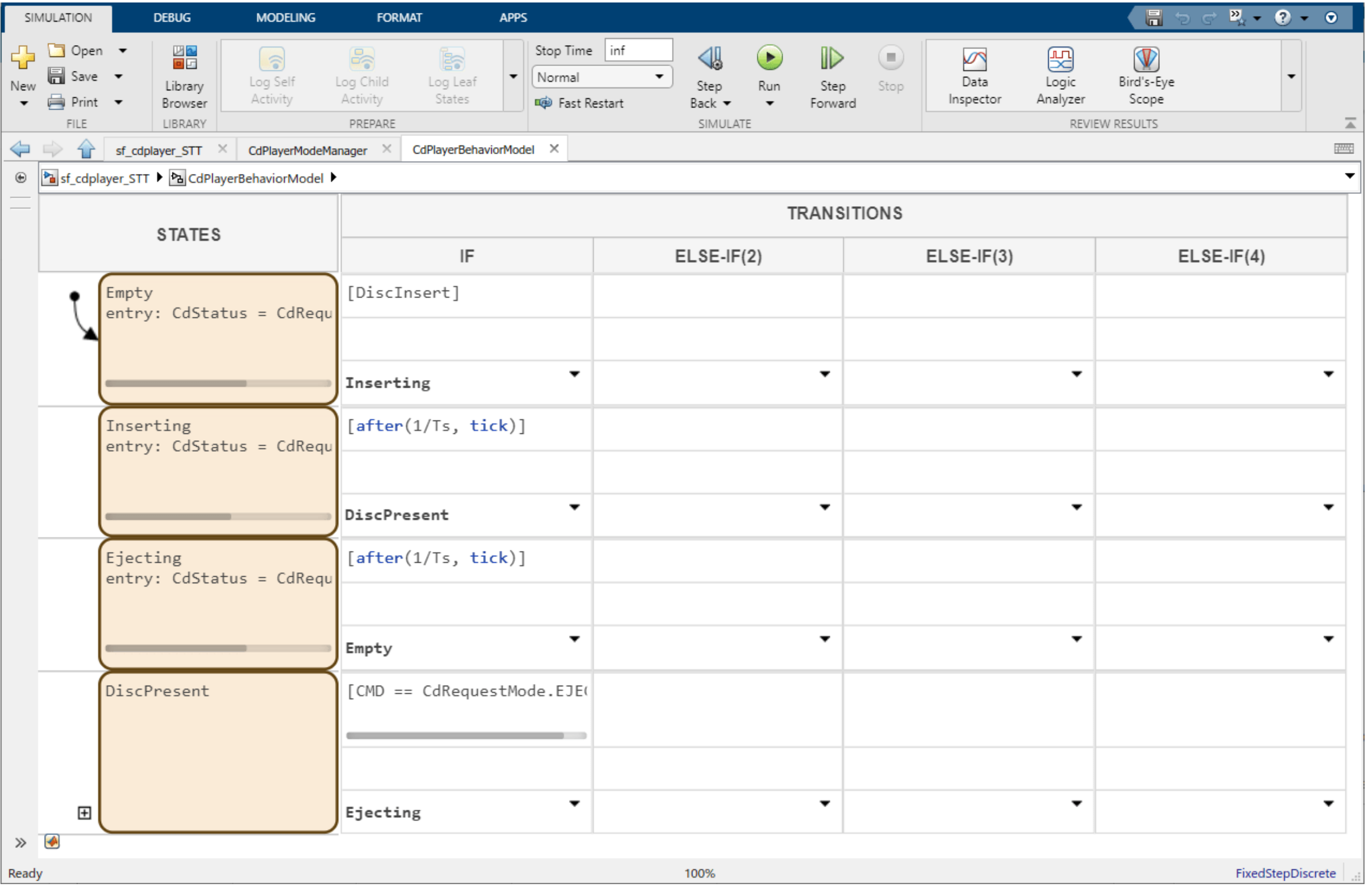
Transition Cells

The next set of columns represent the outer transitions from a state. Each row represents the outer transitions from a given state. Each of the transition cells is sub-divided into three sub-cells:

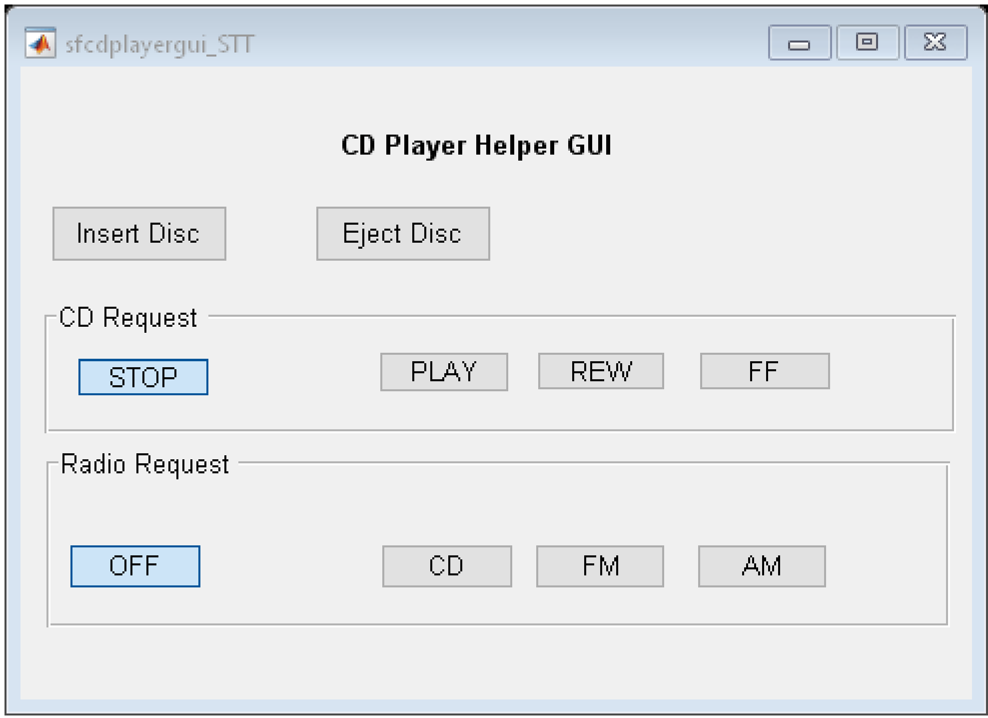
1. Condition cell: to specify a boolean condition which specifies when the transition is active
2. Condition Action cell: to specify the action to be taken when the transition is active
3. Destination cell: to specify the destination state for the transition. Notice that you can use special keywords such as `$NEXT`, `$PREV` to specify the destination relative to the current state.

This chart receives user inputs, such as whether a disk has been inserted and the choice for the radio mode (FM, AM, or CD). Then the chart determines the mechanical command to output. The data types of input and output data are defined as enumerated data types in the MATLAB® files `CdRequestMode.m` and `RadioRequestMode.m`.

The output command from the `CdPlayerModeManager` chart is processed by the chart `CdPlayerBehaviorModel` which models the behavior of the CD Player mechanism. This logic is also implemented using a State Transition Table.



A MATLAB UI is used to set the various CD/Radio modes.



See Also

More About

- "Use State Transition Tables to Express Sequential Logic in Tabular Form" on page 16-2
- "Model Media Player by Using Enumerated Data" on page 20-15

Make States Reusable with Atomic Subcharts

- “Create Reusable Subcomponents by Using Atomic Subcharts” on page 17-2
- “Guidelines for Using Atomic Subcharts” on page 17-7
- “Map Variables for Atomic Subcharts and Boxes” on page 17-11
- “Robot Trajectory Planning with Reusable Components” on page 17-24
- “Reuse a State Multiple Times in a Chart” on page 17-32
- “Reduce the Compilation Time of a Chart” on page 17-38
- “Divide a Chart into Separate Units” on page 17-41
- “Generate Separate Code for an Atomic Subchart” on page 17-44
- “Model a Redundant Sensor Pair by Using Atomic Subcharts” on page 17-48
- “Model an Elevator System by Using Atomic Subcharts” on page 17-52

Create Reusable Subcomponents by Using Atomic Subcharts

An *atomic subchart* is a graphical object that helps you to create independent subcomponents in a Stateflow chart. Atomic subcharts are not supported in standalone Stateflow charts in MATLAB.

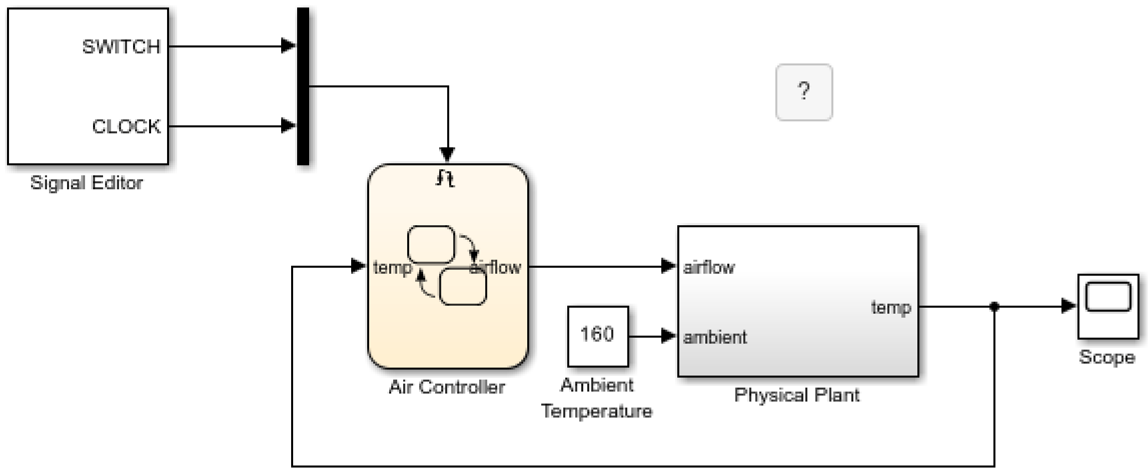
Atomic subcharts allow for:

- Reuse of the same state or subchart across multiple charts and models
- Faster simulation after making small changes to a chart with many states or levels of hierarchy
- Ease of team development when multiple people are working on different parts of the same chart
- Manual inspection of generated code for a specific state or subchart in a chart

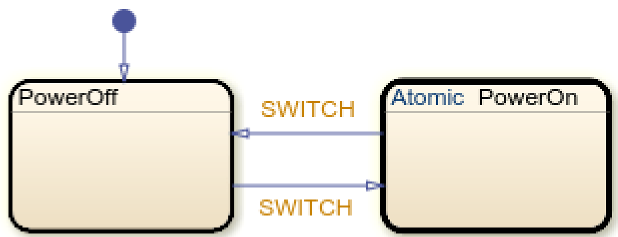
An atomic subchart looks opaque and includes the label **Atomic** in the upper-left corner. If you use a linked atomic subchart from a library, the label **Link** appears in the upper-left corner.

Example of an Atomic Subchart

This example illustrates the difference between a normal subchart and an atomic subchart.



In the Air Controller chart, `PowerOff` is a normal subchart. `PowerOn` is an atomic subchart. Both subcharts look opaque, but the `PowerOn` includes the label **Atomic** on the upper-left corner.



Benefits of Using Atomic Subcharts

Atomic subcharts combine the functionality of states on page 1-26, normal subcharts on page 6-6, and atomic subsystems (Simulink). Atomic subcharts:

- Behave as independent charts.
- Support usage as library links.
- Support the generation of reusable code.
- Allow mapping of inputs, outputs, parameters, data store memory, and input events.

Atomic subcharts do not support access to:

- Data at every level of the chart hierarchy.
- Event broadcasts outside the scope of the atomic subchart.

Atomic subcharts do not support explicit specification of sample time.

Create an Atomic Subchart

You can create an atomic subchart by converting an existing state or subchart or by linking a chart from a library model. After creating the atomic subchart, update the mapping of variables by right-clicking the atomic subchart and selecting **Subchart Mappings**. For more information, see “Map Variables for Atomic Subcharts and Boxes” on page 17-11.

Convert a State or Normal Subchart to an Atomic Subchart

To create an independent component that allows for faster debugging and code generation workflows, convert an existing state or subchart into an atomic subchart. In your chart, right-click a state or a normal subchart and select **Group & Subchart** > **Atomic Subchart**. The label **Atomic** appears in the upper-left corner of the subchart.

The new atomic subchart has a copy of every data object that the subchart accesses in the chart. Local data is copied as data store memory. The scope of other data, including input and output data, does not change.

Converting a state or subchart to an atomic subchart automatically replaces any supertransition into or out of the state or subchart with a transition that connects to an entry or exit port. Entry and exit ports enable your chart to transition across boundaries in the Stateflow hierarchy while isolating the logic for entering and exiting the atomic subchart. For more information, see “Create Entry and Exit Connections Across State Boundaries” on page 1-61 and “Robot Trajectory Planning with Reusable Components” on page 17-24.

For a list of issues that can prevent you from converting a state or subchart to an atomic subchart, see “Restrictions for Converting to Atomic Subcharts” on page 17-9.

Link an Atomic Subchart from a Library

To create a subcomponent for reuse across multiple charts and models, create a link from a library model. Copy a chart in a library model and paste it to a chart in another model. If the library chart contains any states, it appears as a linked atomic subchart with the label **Link** in the upper-left corner.

This modeling method minimizes maintenance of similar states. When you modify the atomic subchart in the library, your changes propagate to the links in all charts and models.

If the library chart contains only functions and no states, then it appears a linked atomic box in the chart. For more information, see “Reuse Functions by Using Atomic Boxes” on page 6-18.

Convert an Atomic Subchart to a Normal Subchart

Converting an atomic subchart back to a state or a normal subchart removes all of its variable mappings. The conversion merges subchart-parented data objects with the chart-parented data to which they map.

1. If the atomic subchart is a library link, right-click the atomic subchart and select **Library Link** > **Disable Link**.
2. To convert an atomic subchart back to a normal subchart, right-click the atomic subchart and clear the **Group & Subchart** > **Atomic Subchart** check box.
3. To convert the subchart back to a state, right-click the subchart and clear the **Group & Subchart** > **Subchart** check box.
4. If necessary, rearrange graphical objects in your chart.

You cannot convert an atomic subchart to a normal subchart if:

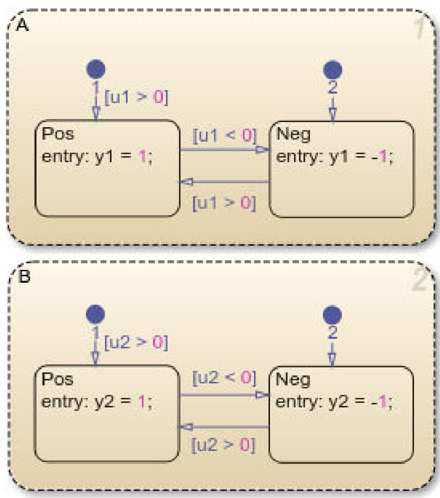
- The atomic subchart maps a parameter to an expression other than a single variable name. For example, mapping a parameter `data1` to one of these expressions prevents the conversion of an atomic subchart to a normal subchart:
  - `3`
  - `data2(3)`
  - `data2 + 3`
- Both of these conditions are true:
  - The atomic subchart contains MATLAB functions or truth table functions that use MATLAB as the action language.
  - The atomic subchart does not map each variable to a variable of the same name in the main chart.

When to Use Atomic Subcharts

Reuse State Logic

Suppose that you want to reuse the same state or subchart many times to facilitate large-scale modeling.

If you do not use atomic subcharts, you have to maintain each copy of a subcomponent manually. For example, this chart contains two states with a similar structure. The only difference between the two states is the names of variables. If you change the logic in state A, then you must make the same change in state B.



To enable reuse of subcomponents by using linked atomic subcharts, create a single copy of state A and store it as a chart in a library model. From that library, copy and paste the atomic subchart twice in your chart. Then update the mapping of subchart variables as needed.

When you change an atomic subchart in a library, the change propagates to all library links. For more information, see "Reuse a State Multiple Times in a Chart" on page 17-32.

Debug Charts Incrementally

Suppose that you want to test a sequence of changes in a chart that contains many states or several levels of hierarchy.

If you do not use atomic subcharts, when you make a small change to one part of a chart and start simulation, recompilation occurs for the entire chart. Because recompiling the entire chart can take a long time, you decide to make several changes before testing. However, if you find an error, you must step through all of your changes to identify the cause of the error.

In contrast, when you modify an atomic subchart, recompilation occurs for only the subchart and not for the entire chart. Incremental builds for simulation require less time to recompile. This reduction in compilation time enables you to test each individual change instead of waiting to test multiple changes at once. By testing each change individually, you can quickly identify a change that causes an error. For more information, see "Reduce the Compilation Time of a Chart" on page 17-38.

Develop Charts Used by Multiple People

Suppose that you want to break a chart into subcomponents because multiple people are working on different parts of the chart.

Without atomic subcharts, only one person at a time can edit the model. If someone edits one part of a chart while someone else edits another part of the same chart, you must merge those changes at submission time.

In contrast, you can store different parts of a chart as linked atomic subcharts. Because atomic subcharts behave as independent objects, different people can work on different parts of a chart without affecting the other parts of the chart. At submission time, no merge is necessary because the changes exist in separate models. For more information, see "Divide a Chart into Separate Units" on page 17-41.

Inspect Generated Code

Suppose that you want to inspect code generated by Simulink Coder or Embedded Coder manually for a specific part of a chart.

If you do not use atomic subcharts, you generate code for an entire model in one file. To find code for a specific part of the chart, you have to look through the entire file.

In contrast, you can specify that the code for an atomic subchart appears in a separate file. This method of code generation enables unit testing for a specific part of a chart. You avoid searching through unrelated code and focus only on the code that interests you. For more information, see "Generate Separate Code for an Atomic Subchart" on page 17-44.

See Also

Atomic Subsystem

More About

- "Encapsulate Modal Logic by Using Subcharts" on page 6-6
- "Map Variables for Atomic Subcharts and Boxes" on page 17-11
- "Model an Elevator System by Using Atomic Subcharts" on page 17-52
- "Model a Redundant Sensor Pair by Using Atomic Subcharts" on page 17-48

Guidelines for Using Atomic Subcharts

An atomic subchart is a graphical object that helps you to create independent subcomponents in a Stateflow chart. Atomic subcharts are not supported in standalone Stateflow charts in MATLAB. For more information, see “Create Reusable Subcomponents by Using Atomic Subcharts” on page 17-2.

Chart Properties and 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 Atomic Subcharts in Continuous-Time Charts

Continuous-time charts do not support atomic subcharts.

Avoid Using Atomic Subcharts in Charts That Execute at Initialization

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 can 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 2-10.

Use Consistent Settings for Super Step Semantics

When you use linked atomic subcharts, verify that your settings for super step semantics match the settings in the main chart. For more information, see “Super Step Semantics” on page 2-35.

Data in Atomic Subcharts

Define Data in an Atomic Subchart Explicitly

Be sure to define data that appears in an atomic subchart explicitly in the main chart. Atomic subcharts can only access main chart data whose size, type, and complexity are fully specified. For more information, see “Set Data Properties” on page 10-5.

Map Variables of Linked Atomic Subcharts

When you use linked atomic subcharts, map the variables so that data in the subchart corresponds to the correct data in the main chart. Map subchart variables manually if, when you add the subchart, the variables do not have the same names as the corresponding symbols in the main chart. For more information, see “Map Variables for Atomic Subcharts and Boxes” on page 17-11.

Match Size, Type, and Complexity of Variables in Linked Atomic Subcharts

Verify that the size, type, and complexity of variables in a subchart match the settings of the corresponding variables in the main chart. For more information, see “Map Variables for Atomic Subcharts and Boxes” on page 17-11.

Do Not Use Variable-Size Data in Atomic Subcharts

Atomic subcharts do not support variable-sized input or output arrays.

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.

Do Not Log Signals from Atomic Subcharts That Map Variables with Different Scopes

If an atomic subchart maps variables to variables at the main chart level with a different scope, you cannot log signals for the chart.

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")));
```

Events in Atomic Subcharts

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 12-4.

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.

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 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.

Functions and Atomic Subcharts

Export Chart-Level Functions If Called from an Atomic Subchart

If your atomic subchart contains a function call to a chart-level function, export that function by selecting **Export Chart Level Functions**. Do not export graphical functions from an atomic subchart that maps variables to variables at the main chart level with a different scope. For more information, see “Export Stateflow Functions for Reuse” on page 6-14.

Restrictions for Converting to Atomic Subcharts

Data, Graphical Functions, and Events

To convert a state or subchart to an atomic subchart, access to objects not parented by the state or subchart must be one of the following:

- Chart-level data
- Chart-level graphical functions
- Input events

If the state or subchart accesses a chart-level graphical function, the chart must export that function. For more information, see “Export Stateflow Functions for Reuse” on page 6-14.

Do not export graphical functions from an atomic subchart that maps variables to variables at the main chart level with a different scope.

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.

Event Broadcasts

The state or subchart that you want to convert to an atomic subchart cannot refer to:

- Local events that are outside the scope of that state or subchart
- Output events

The state or subchart you want to convert can refer to *input* events.

Messages

If a state or subchart contains messages, you cannot convert it to an atomic subchart.

Masked Library Chart

You cannot use a masked library chart containing mask parameters as an atomic subchart.

See Also

More About

- “Create Reusable Subcomponents by Using Atomic Subcharts” on page 17-2
- “Model an Elevator System by Using Atomic Subcharts” on page 17-52
- “Model a Redundant Sensor Pair by Using Atomic Subcharts” on page 17-48

Map Variables for Atomic Subcharts and Boxes

An atomic subchart is a graphical object that helps you create reusable subcomponents in a Stateflow chart. An atomic box is a graphical object that helps you share graphical, truth table, MATLAB, and Simulink functions across several charts. Atomic subcharts and boxes are not supported in standalone Stateflow charts in MATLAB. For more information, see "Create Reusable Subcomponents by Using Atomic Subcharts" on page 17-2 and "Reuse Functions by Using Atomic Boxes" on page 6-18.

To ensure that each symbol in your atomic subchart or box accesses the correct symbol in the main chart, edit the mapping of subchart symbols. Right-click the subchart or box and select **Subchart Mappings**. In the **Mappings** tab of the properties dialog box, use the **Main chart symbol** drop-down list to specify which symbol in the main chart corresponds to each symbol in the subchart. Alternatively, you can type an expression specifying:

- A field of a Stateflow structure. See "Index and Assign Values to Stateflow Structures" on page 26-7.
- An element of a vector or matrix. See "Operations for Vectors and Matrices in Stateflow" on page 19-4.
- Any valid combination of structure fields or matrix indices, such as `struct.field(1,2)` or `struct.field[0][1]`.

If you leave the **Main chart symbol** field empty, then Stateflow attempts to map the atomic subchart symbol to a main chart symbol with the same name.

You can map a symbol in the atomic subchart to a symbol in the main chart that has a different scope. This table lists the possible mappings.

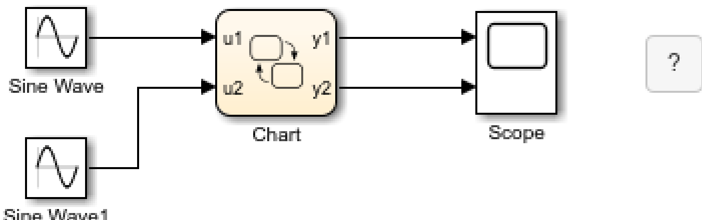
Atomic Subchart Symbol Scope	Main Chart Symbol Scope
Input	Input, Output, Local, Parameter
Output	Output, Local
Parameter	Parameter
Data Store Memory	Data Store Memory, Local
Input Event	Input Event

When you map data store memory in an atomic subchart to local data of enumerated type, you have two options for specifying the initial value of the data store memory:

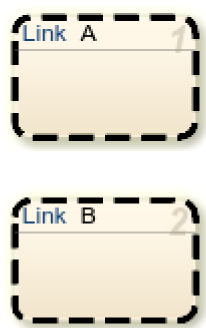
- In the Data properties dialog box, set the **Initial value** field for the chart-level local data.
- To apply the default value of the enumerated type, leave the **Initial value** field empty.

Map Input and Output Data for an Atomic Subchart

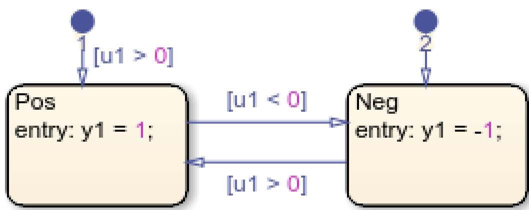
This model contains two Sine Wave blocks that supply input signals to a chart.



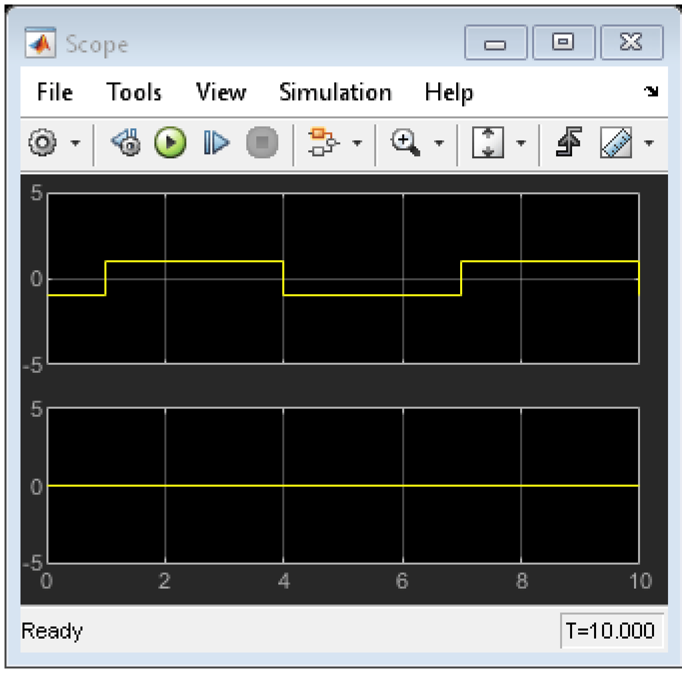
The chart consists of two linked atomic subcharts from the same library.



Both atomic subcharts contain saturator logic to convert an input sine wave to an output square wave of the same frequency.

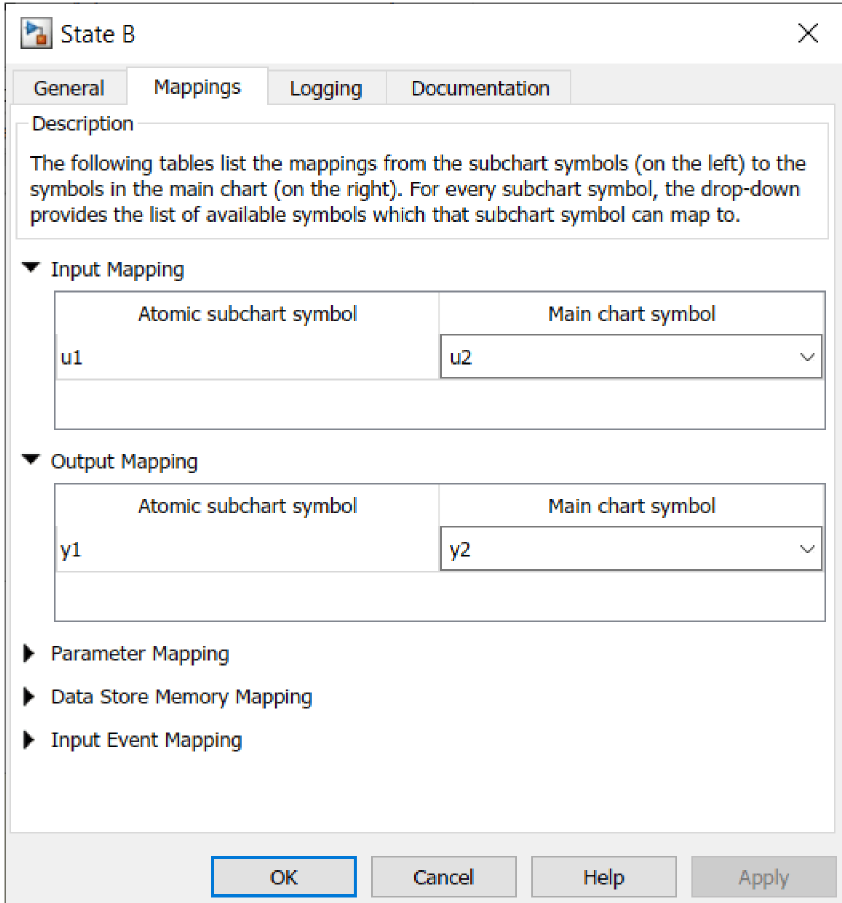


If you simulate the model, the output for y2 is zero.

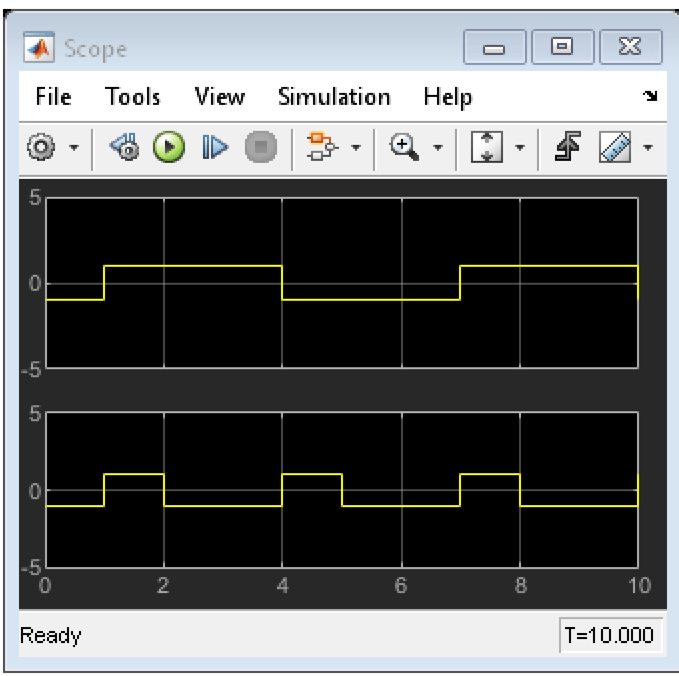


Because the symbols in atomic subchart A have the same name as the symbols u1 and y1 in the main chart, they map to the correct variables. The symbols in atomic subchart B do not map to u2 and y2 in the main chart, so you must edit the mapping.

1. Right-click subchart B and select **Subchart Mappings**.
2. Under **Input Mapping**, specify the main chart symbol for u1 to be u2.
3. Under **Output Mapping**, specify the main chart symbol for y1 to be y2.
4. Click **OK**.

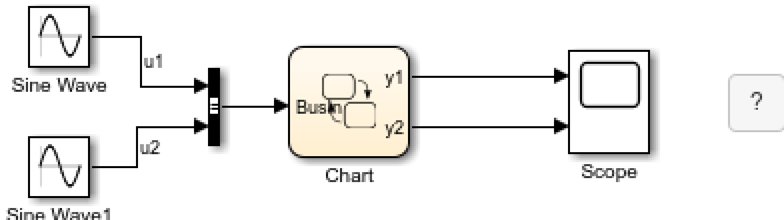


When you run the model again, you get these results.

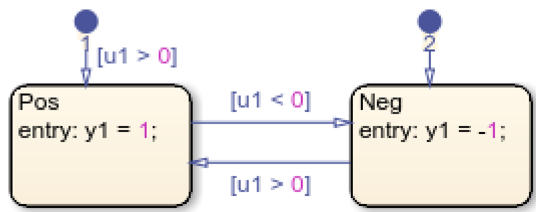


Map Atomic Subchart Variables to Bus Elements

This model contains two Sine Wave blocks that supply signals through a bus to a chart.



The chart consists of two linked atomic subcharts from the same library. Both atomic subcharts contain saturator logic to convert an input sine wave to an output square wave of the same frequency.

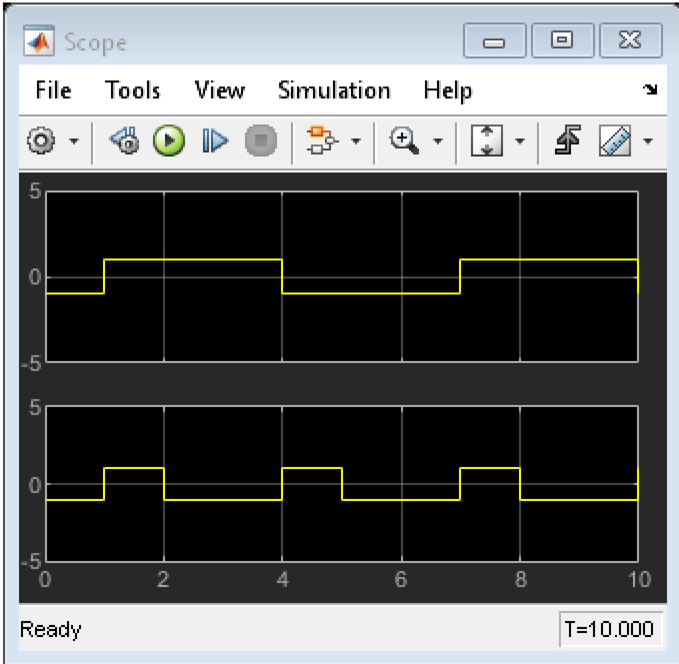


If you simulate the model, you get an error because the `u1` inputs in each subchart do not map to any variables in the main chart. To edit the mapping for `u1` in each subchart:

1 Right-click subchart A and select **Subchart Mappings**.

2 Under **Input Mapping**, specify the main chart symbol for `u1` to be the first element in the bus: `BusIn.u1`.
3 Click **OK**.
4 Repeat for subchart B, specifying the main chart symbol for `u1` to be the second element in the bus: `BusIn.u2`.

When you run the model again, you get these results.



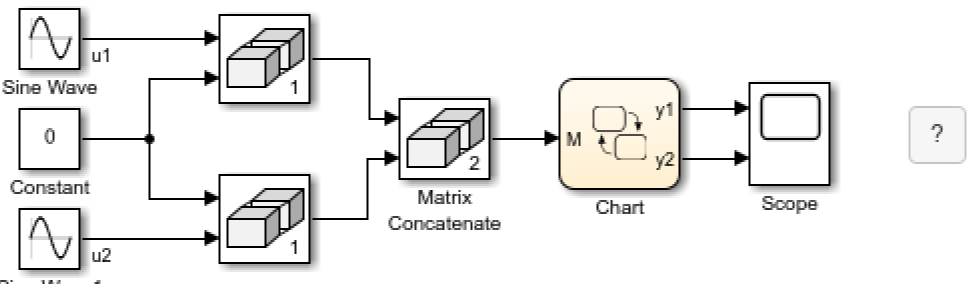
Map Atomic Subchart Variables to the Elements of a Matrix

When referring to elements of a vector or matrix, regardless of the action language of the chart, use:

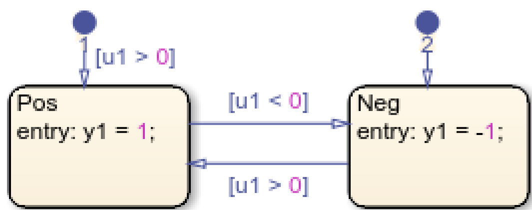
- One-based indexing delimited by parentheses and commas. For example, `A(4,5)`.
- Zero-based indexing delimited by brackets. For example, `A[3][4]`.

Indices can be numbers or parameters in the chart. The use of other expressions as indices is not supported.

For example, this model contains two Sine Wave blocks that supply signals through a diagonal matrix to a chart.



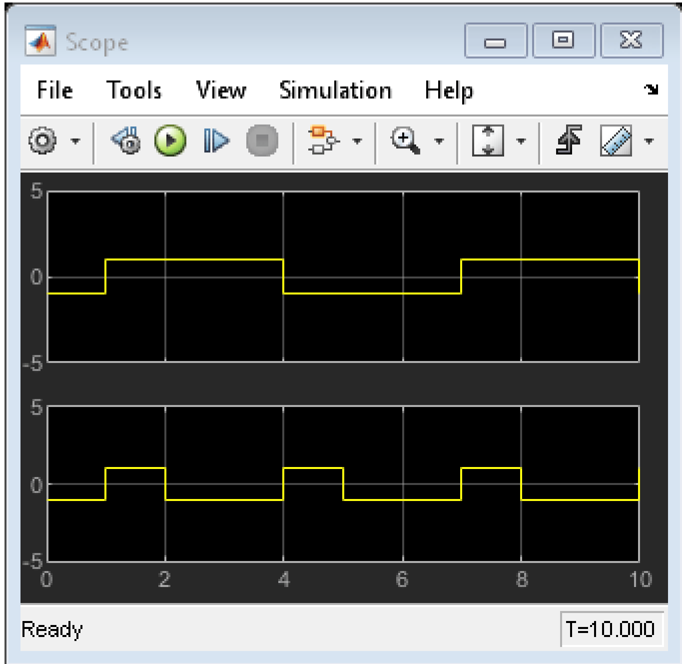
The chart consists of two linked atomic subcharts from the same library. Both atomic subcharts contain saturator logic to convert an input sine wave to an output square wave of the same frequency.



If you simulate the model, you get an error because the `u1` inputs in each subchart do not map to any variables in the main chart. To edit the mapping for `u1` in each subchart:

1. Right-click subchart A and select **Subchart Mappings**.
2. Under **Input Mapping**, specify the main chart symbol for `u1` to be the top-left element in the matrix. The zero-based indexing format for this element is `M[0][0]`.
3. Click **OK**.
4. Repeat for subchart B, specifying the main chart symbol for `u1` to be the bottom-right element in the matrix. The one-based indexing format for this element is `M(2,2)`.

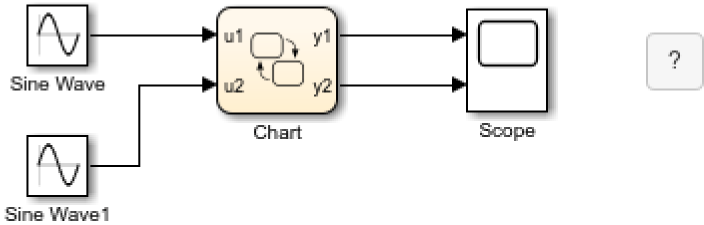
When you run the model again, you get these results.



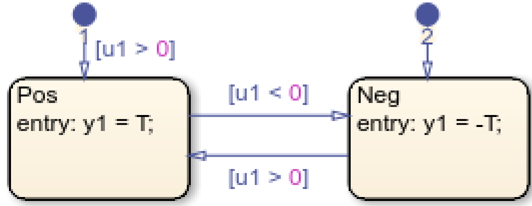
Map Atomic Subchart Parameters to Expressions

For parameters in an atomic subchart, you can specify an expression that combines constants, variables in the base workspace, and parameters in the main chart.

For example, this model contains two Sine Wave blocks that supply input signals to a chart.

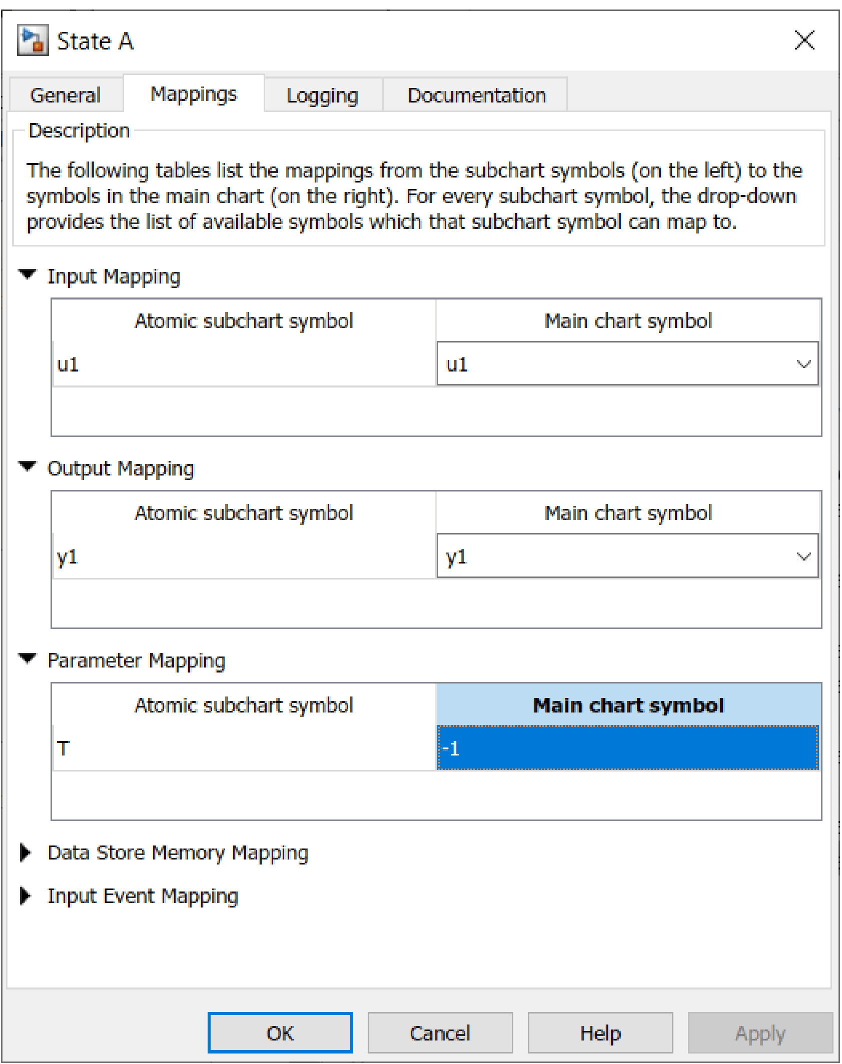


The chart consists of two linked atomic subchart from the same library. Both atomic subcharts contain saturator logic to convert an input sine wave to an output square wave of the same frequency.

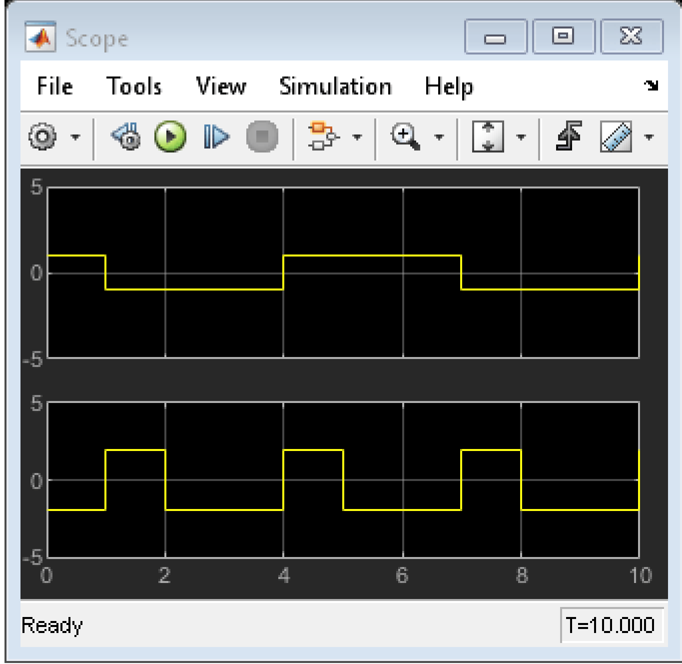


If you simulate the model, you get an error because the parameter T is undefined. To fix this error, specify an expression for T to evaluate in the main chart:

1. Right-click subchart A and select **Subchart Mappings**.
2. Under **Parameter Mapping**, as the value for T, enter -1.
3. Click **OK**.
4. Repeat for subchart B, specifying the value of T as 2.

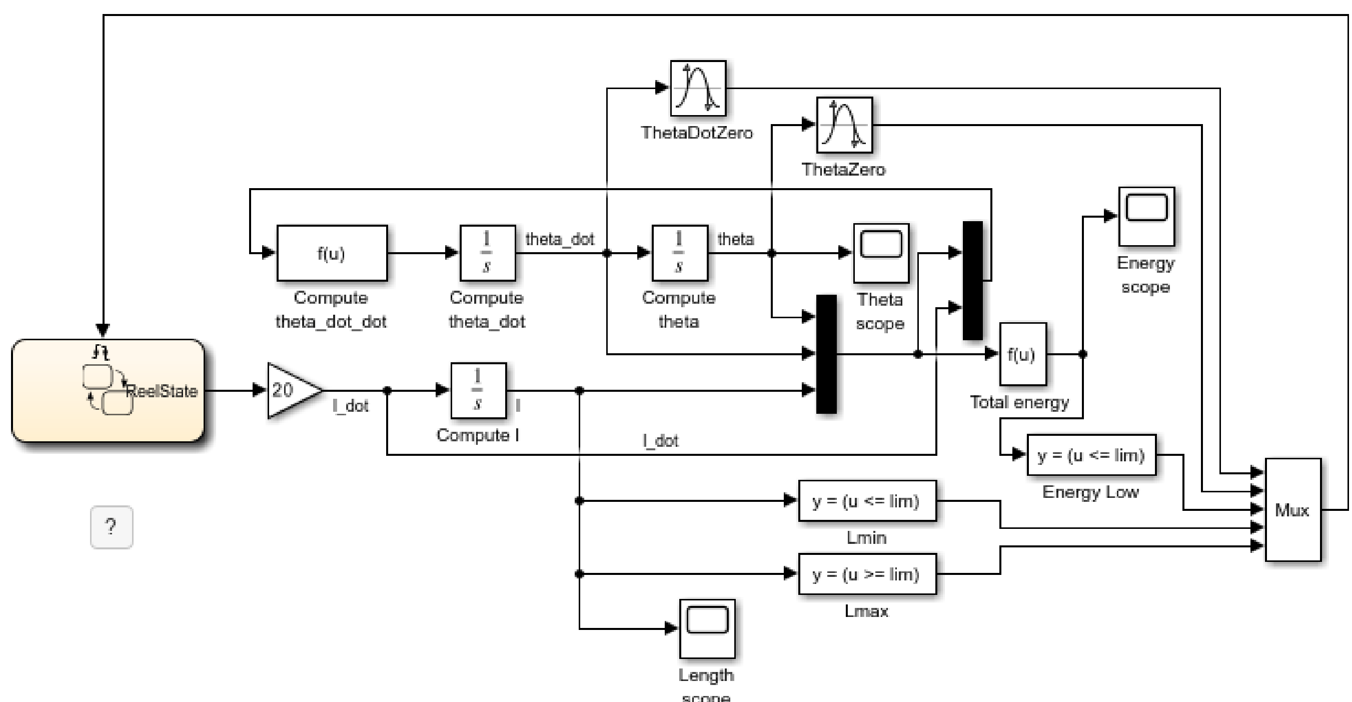


When you run the model again, you get these results.

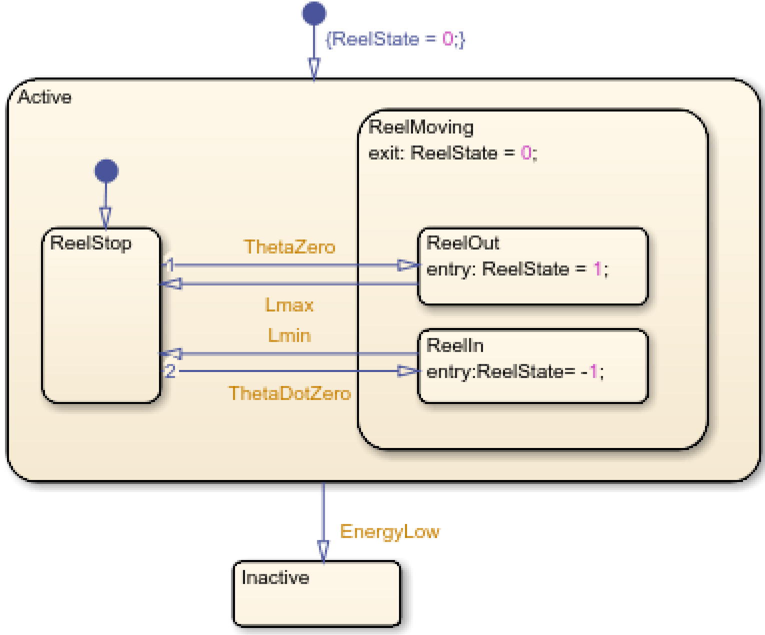


Map Input Events for an Atomic Subchart

This model contains a Mux block that supplies input events to a chart.

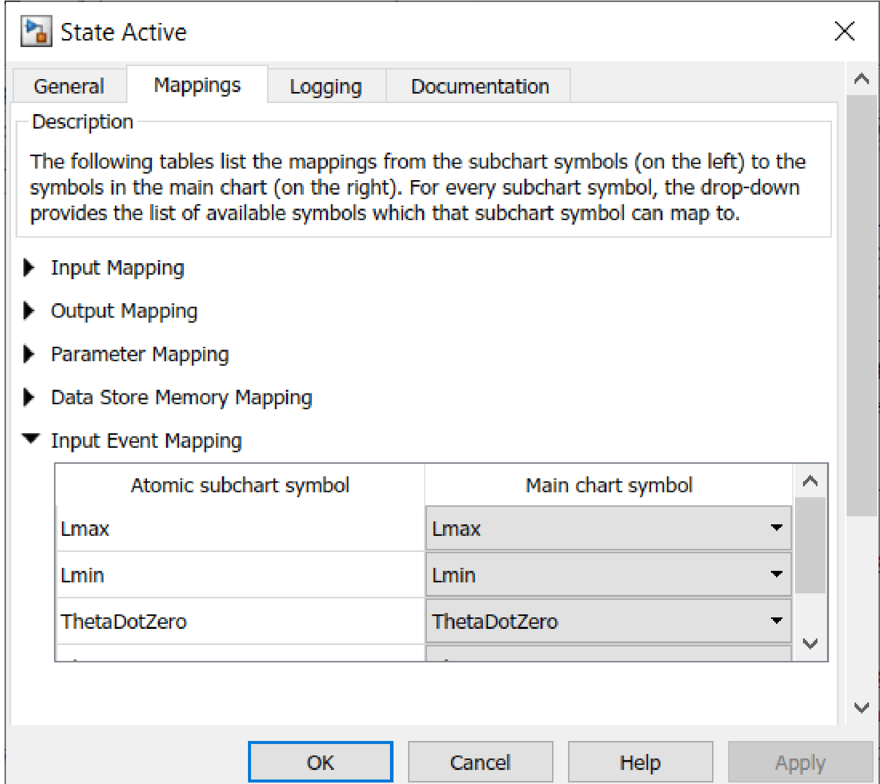


The chart contains two superstates: `Active` and `Inactive`. The `Active` state uses input events to guard transitions between different substates.



To convert the `Active` state to an atomic subchart:

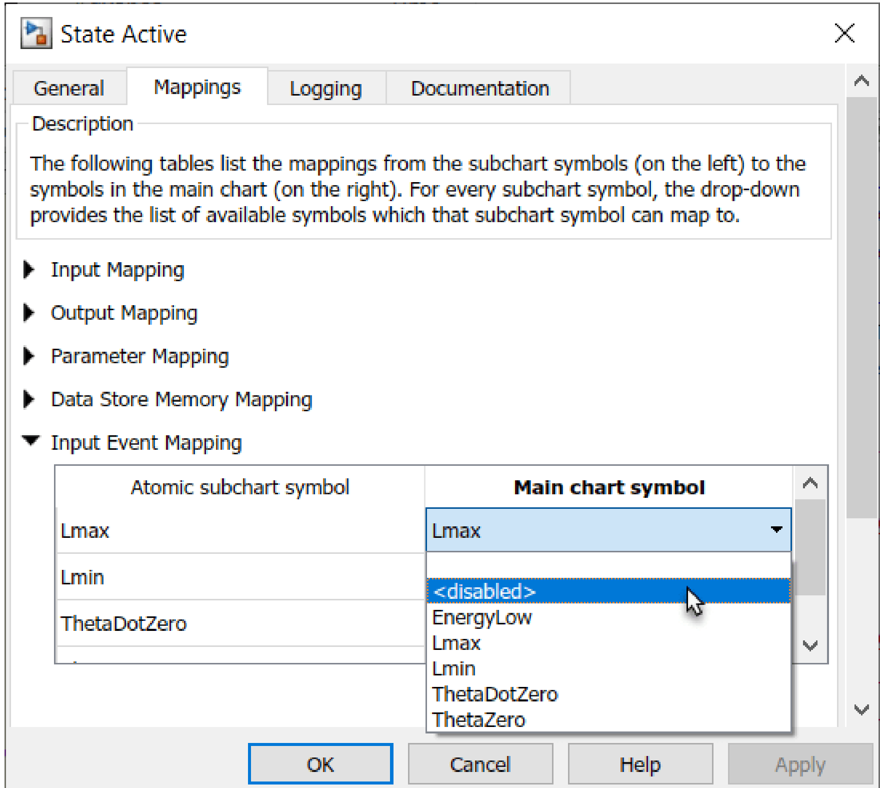
1. Right-click the `Active` state and select **Group & Subchart** > **Atomic Subchart**.
2. Right-click the atomic subchart and select **Subchart Mappings**.
3. Under **Input Event Mapping**, map each atomic subchart symbol to the corresponding input event in the main chart.
4. Click **OK**.



Disable Input Events for Atomic Subcharts

Not every input event in an atomic subchart has to correspond to an event in the main chart. For example, you can create a linked atomic subchart that does not use the entire set of events that are defined in the library chart. To disable an input event in an atomic subchart:

1. Right-click the atomic subchart and select **Subchart Mappings**.
2. Under **Input Event Mapping**, in the **Main chart symbol** drop-down list, select `<disabled>`.
3. Click **OK**.



See Also

More About

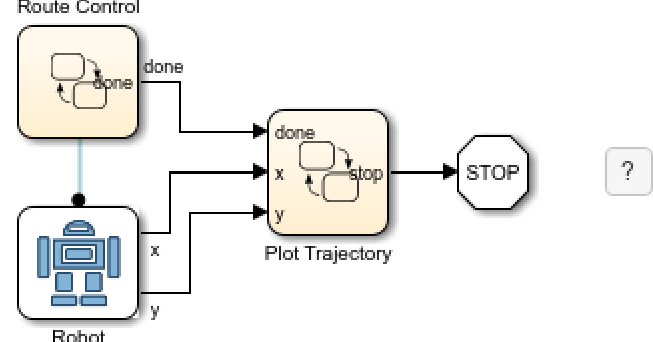
- “Create Reusable Subcomponents by Using Atomic Subcharts” on page 17-2
- “Reuse Functions by Using Atomic Boxes” on page 6-18
- “Index and Assign Values to Stateflow Structures” on page 26-7
- “Operations for Vectors and Matrices in Stateflow” on page 19-4

Robot Trajectory Planning with Reusable Components

This example shows how to use entry and exit ports to create multiple connections into and out of linked atomic subcharts. Entry and exit ports enable your chart to transition across boundaries in the Stateflow® hierarchy while isolating the logic for entering and exiting atomic subcharts. For more information about entry and exit ports, see “Create Entry and Exit Connections Across State Boundaries” on page 1-61.

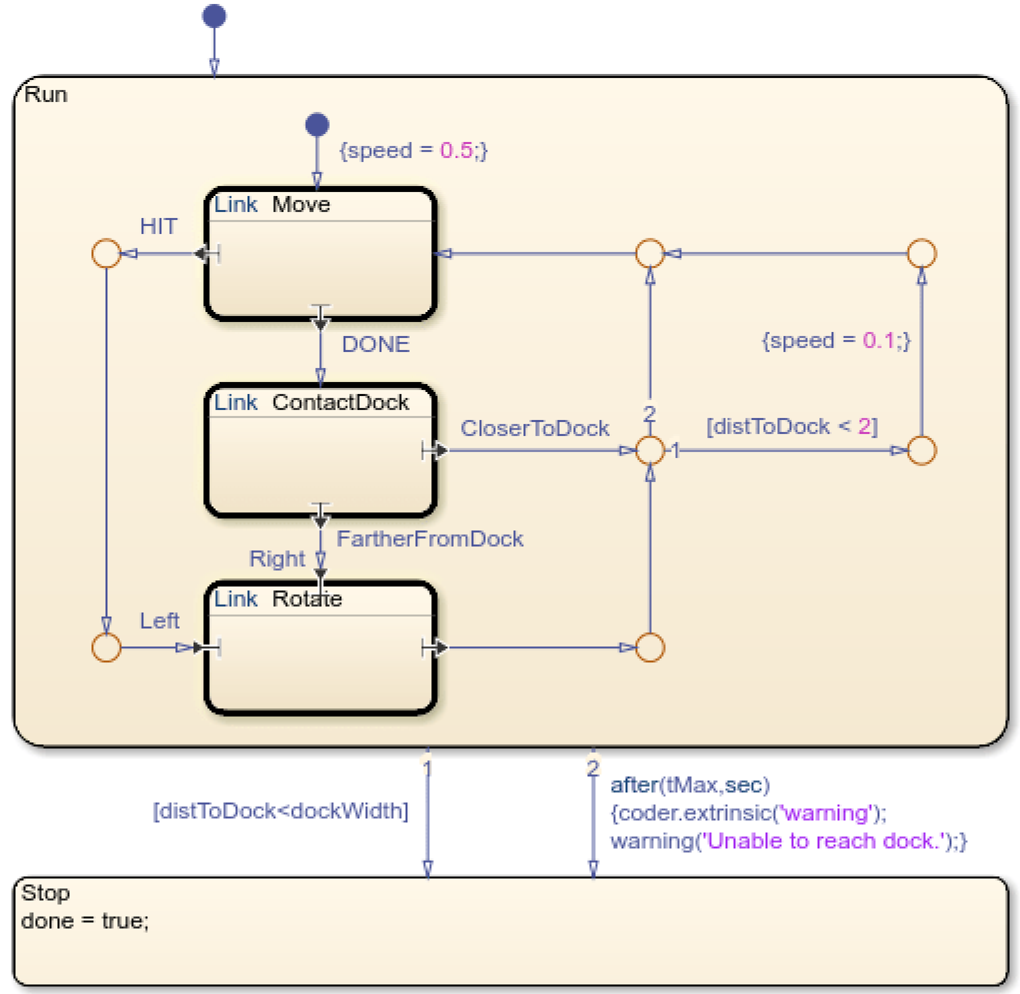
In this example, a Simulink® model simulates a robot that maneuvers through an obstacle course in search of a docking station. The model contains three Stateflow charts:

- `Route Control` defines the strategy used by the robot to search for the dock and navigate the obstacle course.
- `Robot` defines the physical characteristics of the robot, such as position and direction of motion, in relation to the dock and the obstacles that surround it.
- `Plot Trajectory` creates a visual representation of the path that the robot takes as it avoids obstacles and searches for the dock.



Define Search Strategy

The `Route Control` chart defines the strategy that the robot uses to search for the docking station. The chart consists of a combination of linked atomic subcharts from the Simulink library model `sfRobotExampleLib.slx`. The linked atomic subcharts behave as macros that instruct the robot to move forward, rotate left or right, and make radio contact with the dock. You can combine one or more instances of these subcharts to program your own search strategy.



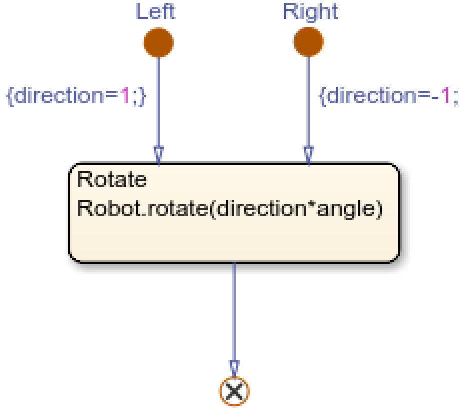
In this example, the robot moves in a straight line and makes radio contact with the dock at regular intervals. If the robot senses that it is moving away from the dock, it rotates 45 degrees to the right and continues searching. If the robot runs into an obstacle, it rotates 45 degrees to the left and continues searching. Initially, the robot moves with a speed of 0.5 meters per second. When the robot comes within 2 meters of the dock, it slows down to a speed of 0.1 meters per second. When the robot comes close to the docking station, the simulation stops. Otherwise, after `tMax` seconds, the simulation stops and returns a warning.

Create Multiple Entry Connections into Subchart

In the `Route Control` chart, the linked atomic subchart `Rotate` contains two entry ports labeled `Left` and `Right`. During simulation, the `Rotate` subchart becomes active when the chart takes a transition that leads to one of these entry ports.

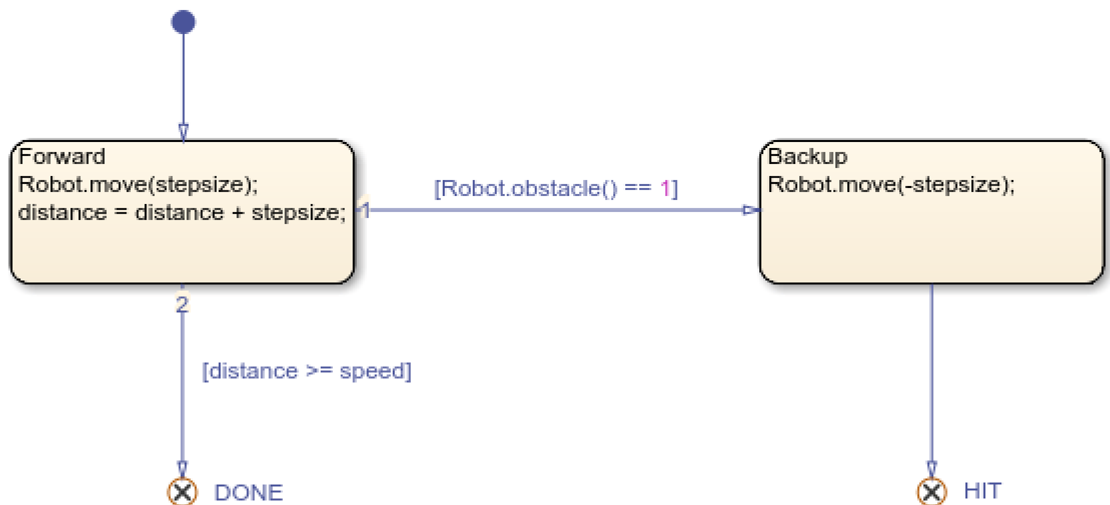
Inside the subchart, each entry port has a matching entry junction. The transitions that connect from these entry junctions to the state `Rotate` set the value of the local data object `direction` to `1` or `-1`. This value tells the robot whether to turn clockwise or counterclockwise. Then, the state `entry` action calls the `rotate` function in the `Robot` chart. This function changes the direction of motion for the robot by an angle of `direction*angle`. The parameter `angle` is specified as `pi/4` in the

Mappings tab of the properties dialog box for the subchart. For more information, see “Map Variables for Atomic Subcharts and Boxes” on page 17-11.



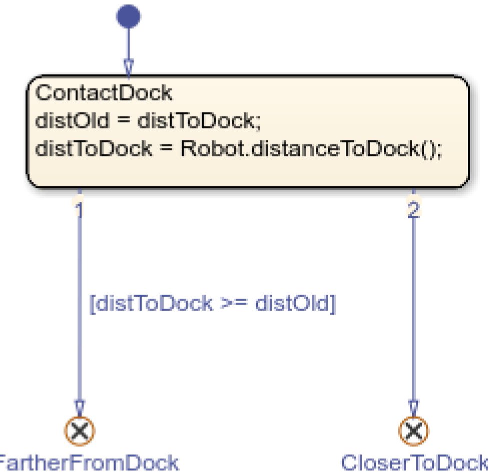
Create Multiple Exit Connections Out of Subchart

The linked atomic subchart `Move` contains two exit ports labeled `Done` and `HIT`. Inside the subchart, each exit port has a matching exit junction. During simulation, the state actions in the substate `Forward` call the `move` function in the `Robot` chart. This function changes the position of the robot by a distance of `step`, which is specified as `0.1` in the **Mappings** tab of the properties dialog box for the subchart. If the function `obstacle` indicates that the robot collided with an obstacle, the robot returns to the last safe position and the chart transitions out of the subchart by using the exit port `HIT`. Otherwise, the robot continues to move forward for a full second and the chart transitions out of the subchart by using the exit port `Done`.



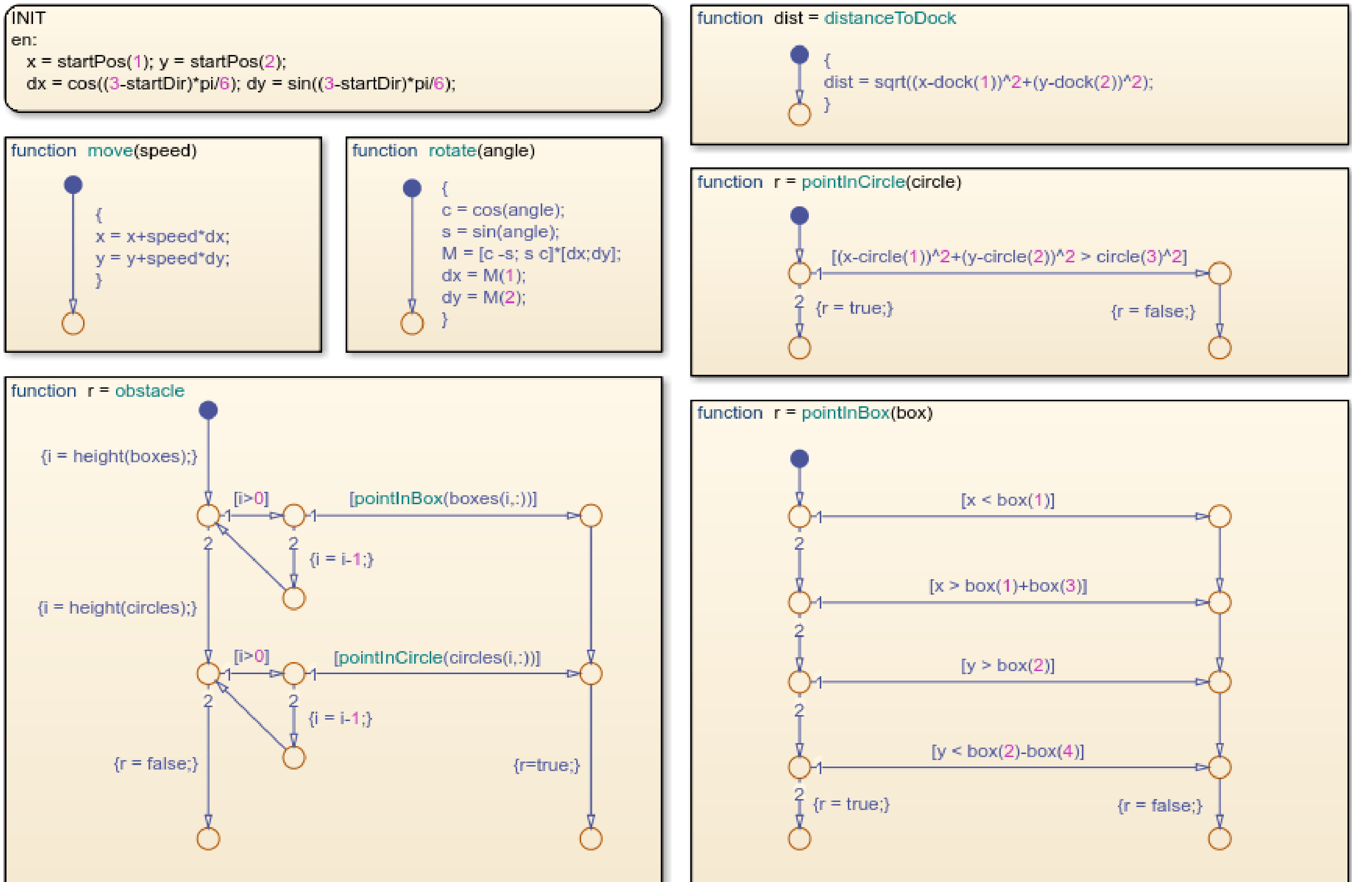
Similarly, the linked atomic subchart `ContactDock` contains two exit ports, `CloserToDock` and `FartherFromDock`. Inside the subchart, each exit port has a matching exit junction. During simulation, the state actions in the substate `ContactDock` call the `distanceToDock` function in the `Robot` chart. This function determines the distance from the robot to the docking station. If this distance decreased in the last time step, the chart transitions out of the subchart by using the exit

port CloserToDock. Otherwise, the chart transitions out of the subchart by using the exit port FartherFromDock.

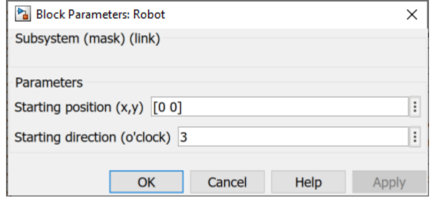


Model Physical Environment and Obstacle Course

The Robot chart maintains the position and direction of the robot. The chart also exports several functions to move and rotate the robot, determine the distance from the robot to the dock, and detect obstacles. By calling these functions, the Route Control chart never interacts directly with position and direction of the robot.



The mask parameters **Starting Position (x,y)** and **Starting direction (o'clock)** specify the values of `startPos` and `startDir`. To modify these values, open the Block Parameters dialog box by double-clicking the `Robot` chart. For more information, see “Create a Mask to Share Parameters with Simulink” on page 25-36.



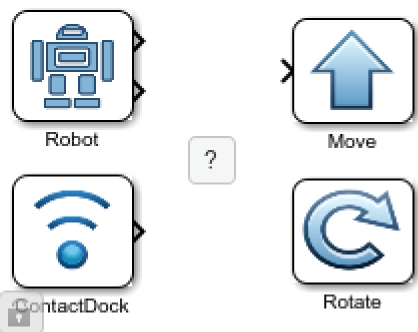
The workspace variables `dock`, `dockWidth`, `circles`, and `boxes` specify the position and size of the docking station and the obstacles around it.

- `dock` is a two-element vector that specifies the horizontal and vertical coordinates of the dock.
- `dockWidth` is a scalar that specifies the width of the dock.
- `circles` is an m-by-3 matrix that specifies the horizontal coordinate, the vertical coordinate, and the radius of each circular obstacle, where m is the number of circular obstacles.
- `boxes` is an n-by-4 matrix that specifies the horizontal coordinate, the vertical coordinate, the width, and the height of each rectangular obstacle, where n is the number of rectangular obstacles.

The obstacle course can contain arbitrarily many obstacles, but it must contain at least one circular and one rectangular obstacle. By default, the PreLoadFcn callback for the model in this example defines an obstacle course with three circular obstacles and three rectangular obstacles.

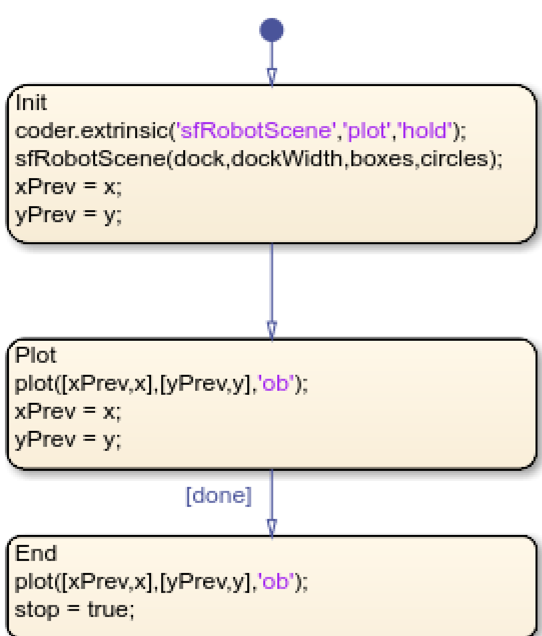
Package Reusable Components in a Library

The Robot chart is a library chart that is linked from the Simulink library model sfRobotExampleLib.slx. This library includes an all-in-one toolkit that defines this chart as well as the linked atomic subcharts for programming your own robot simulation. For more information, see “Custom Libraries” (Simulink).



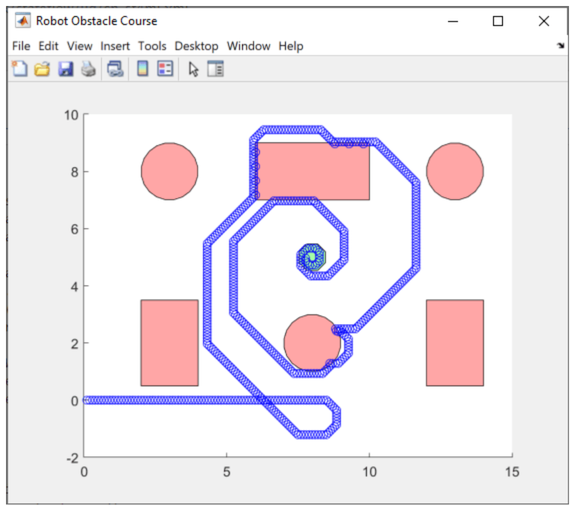
Plot Robot Trajectory

The chart Plot Trajectory reads the workspace variables dock, dockWidth, circles, and boxes, as well as the output signals x and y from the Robot chart and done from the Route Control chart to produce a visual representation of the obstacle course and the path that the robot takes as it searches for the dock.



When you start the simulation, the chart calls the helper function sfRobotScene to create a MATLAB® figure that shows the docking station in green and the obstacles in red. The code for this

function appears at the end of this example. Then, at each step of the simulation, the chart plots the location of the robot by using a blue circle. When the input signal `done` indicates the end of the simulation, the chart plots the final location of the robot and sets the value of the output signal `stop` to `true`, causing the `Stop` block to end the simulation.



To call `sfRobotScene`, `plot`, and `hold` as extrinsic functions, the chart uses the `coder.extrinsic` (Simulink) function. For more information, see “Call Extrinsic MATLAB Functions in Stateflow Charts” on page 14-13.

Display Elements of Obstacle Course

The helper function `sfRobotScene` creates a MATLAB figure that shows the docking station in green and the obstacles in red.

```
function sfRobotScene(dock,width,boxes,circles)

plot(nsidedpoly(8,Center=dock,Radius=2*width),FaceColor="green");
daspect([1 1 1])
hold on

for i = 1:height(circles)
    center = circles(i,1:2);
    radius = circles(i,3);
    plot(nsidedpoly(20,Center=center,Radius=radius),FaceColor="red");
end

for i = 1:height(boxes)
    box = boxes(i,:);
    X = [box(1) box(1)+box(3) box(1)+box(3) box(1)];
    Y = [box(2) box(2) box(2)-box(4) box(2)-box(4)];
    plot(polyshape(X,Y),FaceColor="red");
end

fig = gcf;
fig.Name = "Robot Obstacle Course";
fig.NumberTitle = 'off';
figure(fig)
```

```
end
```

See Also

`plot` | `hold` | `coder.extrinsic`

Related Examples

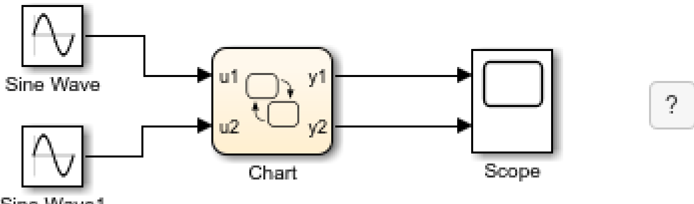
- “Create Entry and Exit Connections Across State Boundaries” on page 1-61
- “Map Variables for Atomic Subcharts and Boxes” on page 17-11
- “Create a Mask to Share Parameters with Simulink” on page 25-36
- “Call Extrinsic MATLAB Functions in Stateflow Charts” on page 14-13
- “Custom Libraries” (Simulink)

Reuse a State Multiple Times in a Chart

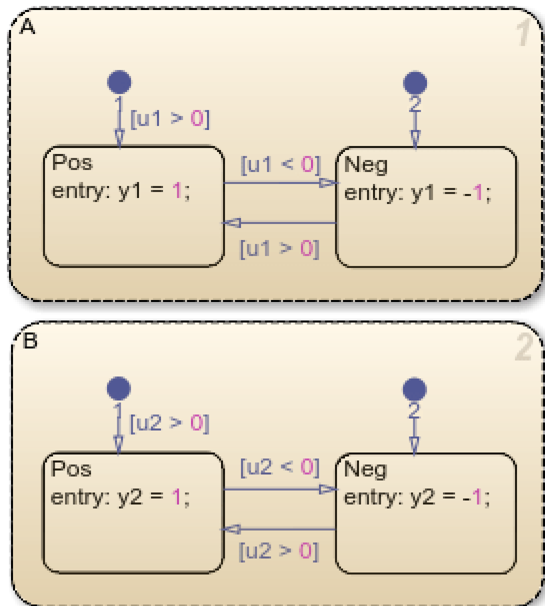
This example shows how to use linked atomic subcharts to repeat the same configuration of states and transitions multiple times in a Stateflow® chart. Atomic subcharts are not supported in standalone Stateflow charts in MATLAB®. For more information, see “Create Reusable Subcomponents by Using Atomic Subcharts” on page 17-2.

Original Model Without Atomic Subcharts

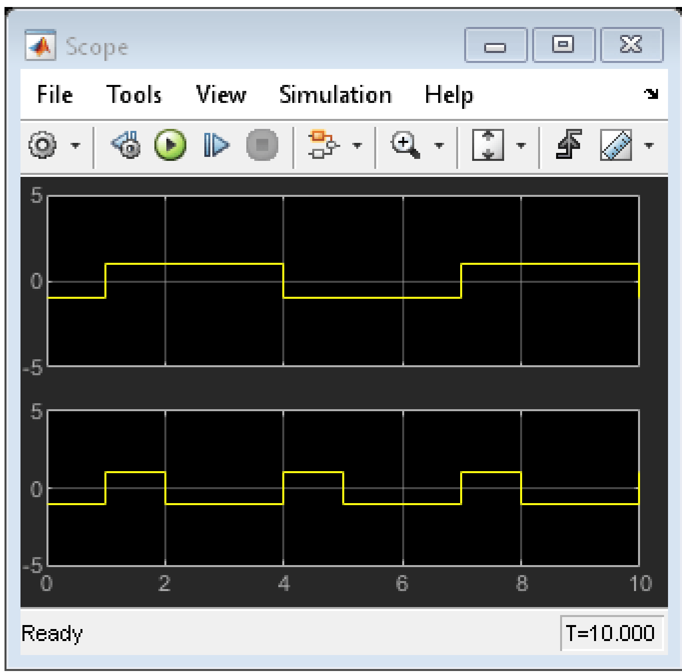
This model contains two Sine Wave (Simulink) blocks: one with a frequency of 1 radian per second, and the other with a frequency of 2 radians per second.



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 their input and output data.



Simulating the model produces these results.



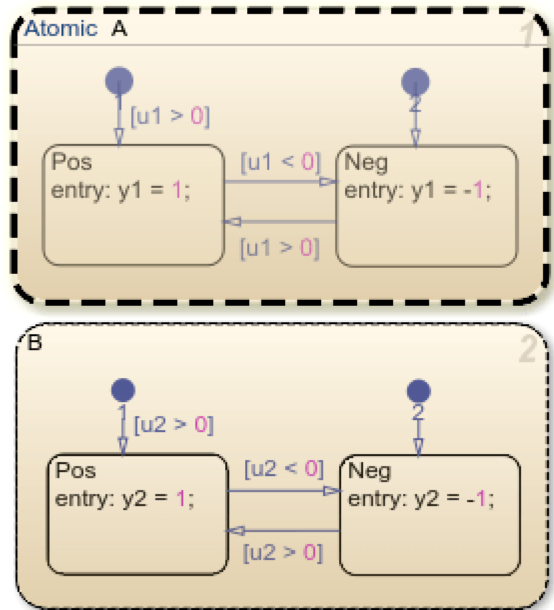
Because this example does not use atomic subcharts, you have to maintain each subcomponent manually. For example, if you change the logic in state A, then you must make the same change in state B.

In contrast, if you replace the states in this example with atomic subcharts, you can reuse the same object in your model and retain the same simulation results. You can save state A as an atomic subchart in a library model and then use multiple linked instances of that subchart in your chart. Changes in the library model propagate to all linked instances of the subchart.

Edit Model to Use Atomic Subcharts

Step 1: Convert a State to an Atomic Subchart

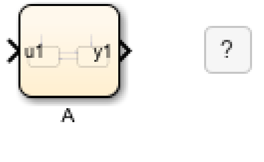
Right-click state A and select **Group & Subchart > Atomic Subchart**. State A changes to an atomic subchart and displays the label **Atomic** in the upper-left corner.



Step 2: Create a Library for the Atomic Subchart

1. Create a new library model.
2. Copy the atomic subchart and paste it in your library model.
3. Save your library model.

In the library model, the atomic subchart appears as an independent chart with an input port and an output port.

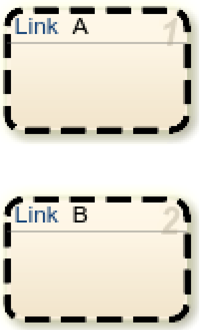


Copyright 2018-2020 The MathWorks, Inc.

Step 3: Replace States with Linked Atomic Subcharts

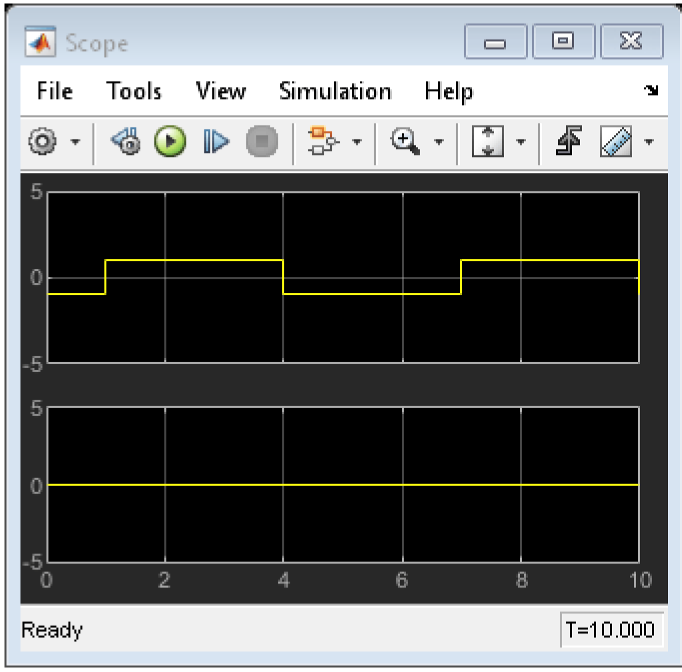
1. Delete both states in the chart.
2. Copy the atomic subchart in your library and paste it in your chart twice.
3. Change the name of the second atomic subchart to B.

Each linked atomic subchart appears opaque and contains the label **Link** in the upper-left corner.



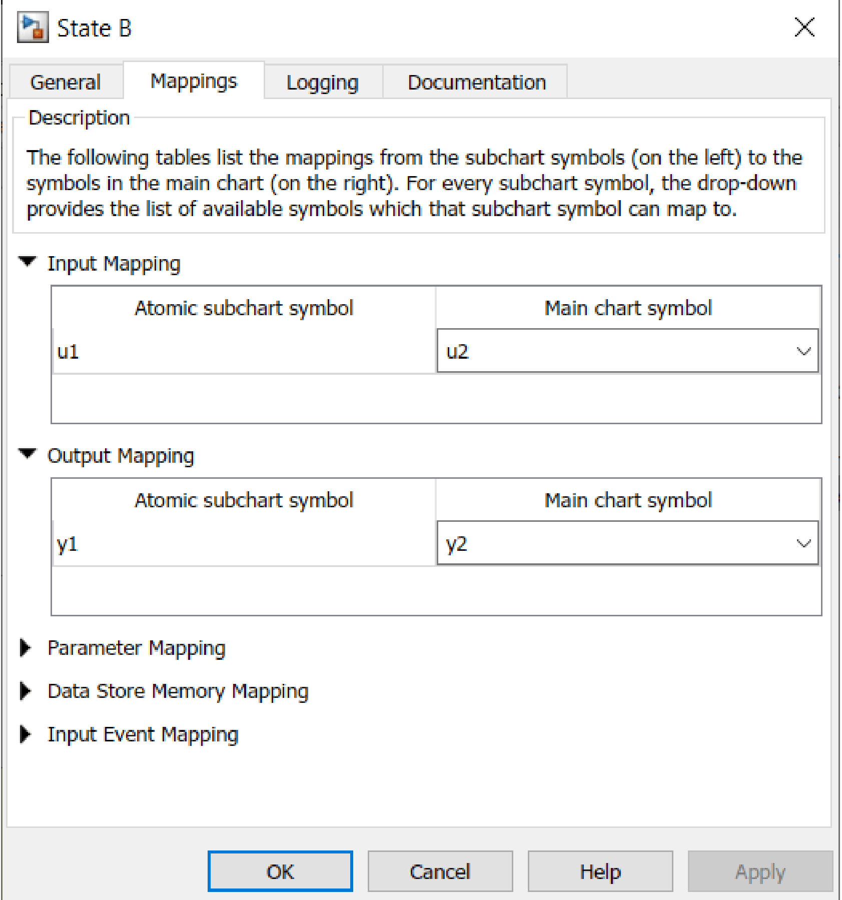
Step 4: Edit the Mapping of Input and Output Variables

If you simulate the model now, the output for y2 is zero. You also see warnings about unused data. These warnings appear because atomic subchart B uses u1 and y1 instead of u2 and y2.



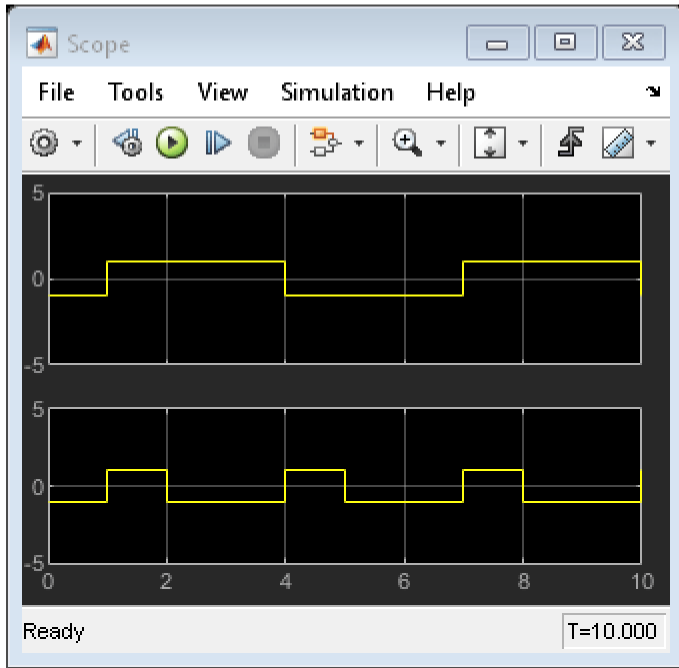
To fix these warnings, you must edit the mapping of input and output variables.

1. Right-click subchart B and select **Subchart Mappings**.
2. Under **Input Mapping**, specify the main chart symbol for u1 to be u2.
3. Under **Output Mapping**, specify the main chart symbol for y1 to be y2.
4. Click **OK**.



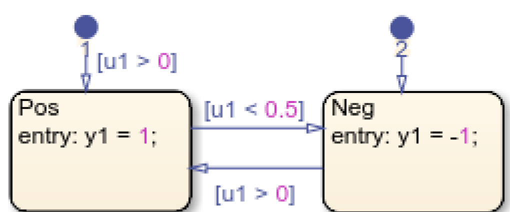
Run the New Model

When you simulate the new model, the results match those of the original design.



Propagate a Change in the Library Chart

Suppose that, in the library chart, you edit the transition from `Pos` to `Neg`.



This change propagates to all linked atomic subcharts in your main chart. You do not have to update each state individually.

See Also

Sine Wave

More About

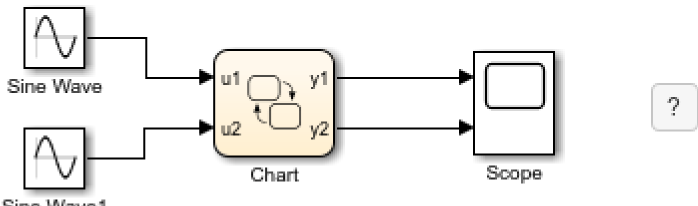
- "Create Reusable Subcomponents by Using Atomic Subcharts" on page 17-2
- "Map Variables for Atomic Subcharts and Boxes" on page 17-11
- "Model an Elevator System by Using Atomic Subcharts" on page 17-52
- "Model a Redundant Sensor Pair by Using Atomic Subcharts" on page 17-48

Reduce the Compilation Time of a Chart

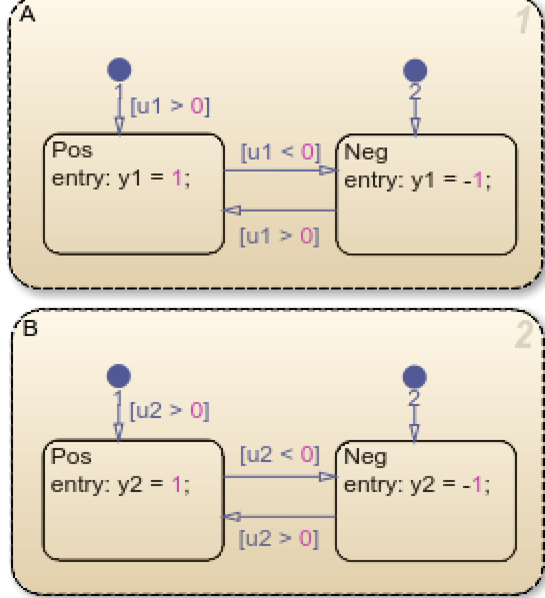
This example shows how to use atomic subcharts to reduce the compilation time when testing a sequence of changes in a Stateflow® chart. Atomic subcharts are not supported in standalone Stateflow charts in MATLAB®. For more information, see “Create Reusable Subcomponents by Using Atomic Subcharts” on page 17-2.

Original Model Without Atomic Subcharts

This model contains two Sine Wave (Simulink) blocks: one with a frequency of 1 radian per second, and the other with a frequency of 2 radians per second.



In the chart, each state uses saturator logic to convert the input sine wave to an output square wave of the same frequency.



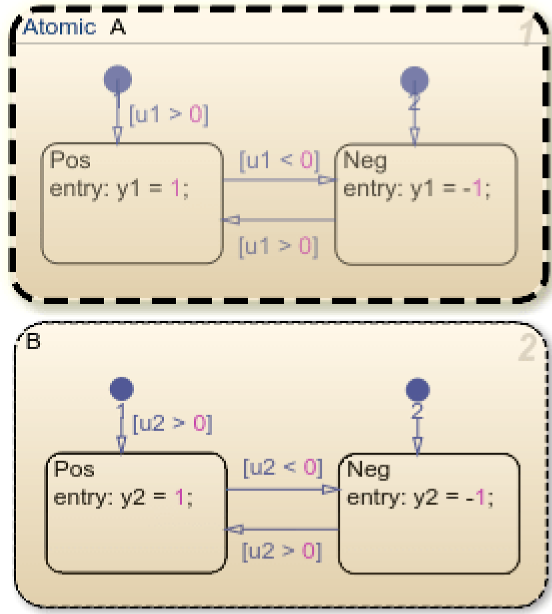
Because this example does not use atomic subcharts, every time that you make a change to the chart and start simulation, recompilation occurs for the entire chart.

In contrast, you can convert state A to an atomic subchart. When you modify the atomic subchart, recompilation occurs for only the subchart and not for the entire chart. As a result, incremental builds for simulation require less time to recompile.

Edit Model to Use Atomic Subcharts

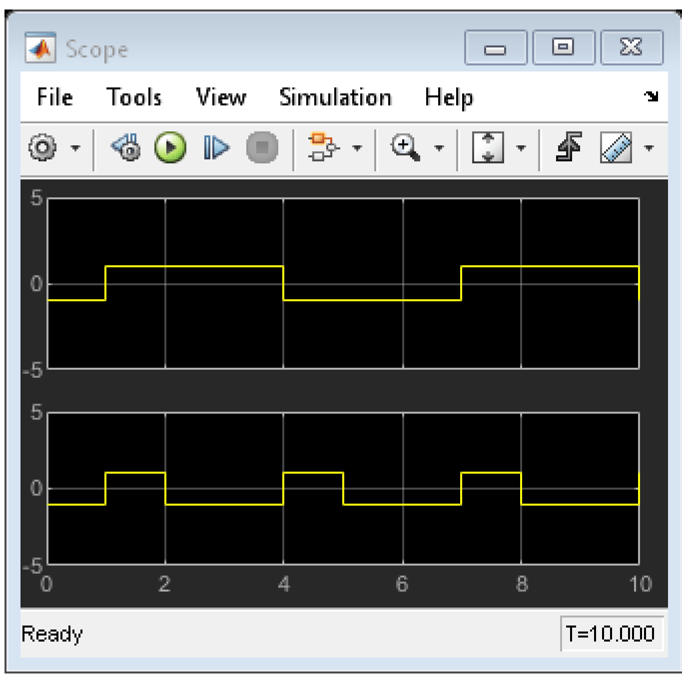
Step 1: Convert a State to an Atomic Subchart

Right-click state A and select **Group & Subchart > Atomic Subchart**. State A changes to an atomic subchart and displays the label **Atomic** in the upper-left corner.



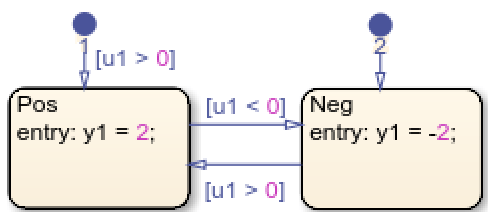
Step 2: Start the Simulation

Before simulating, compilation occurs for the entire chart.



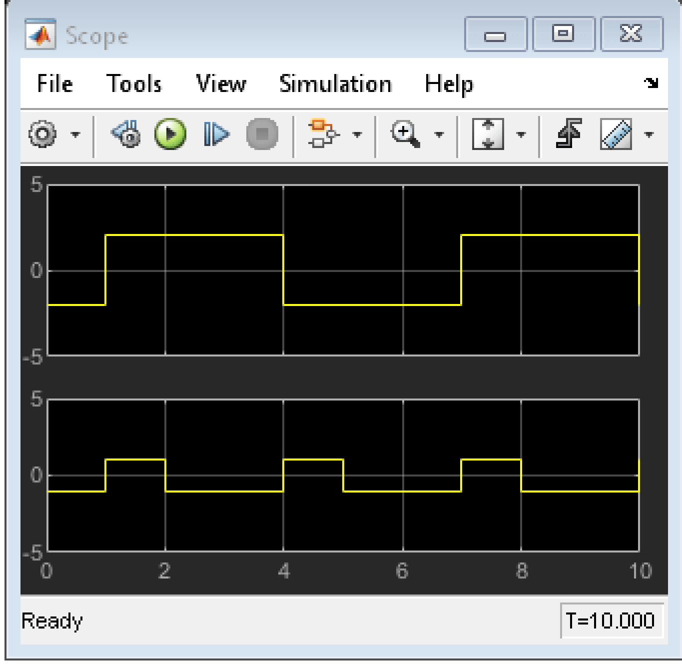
Step 3: Modify the Atomic Subchart

1. Double-click the atomic subchart A. The contents of the subchart appear in the Stateflow Editor.
2. In the state Pos, change the entry action to `y1 = 2;`
3. In the state Neg, change the entry action to `y1 = -2;`



Step 4: Restart the Simulation

After the changes to A, recompilation occurs only for the atomic subchart and not the entire chart.



See Also

Sine Wave

More About

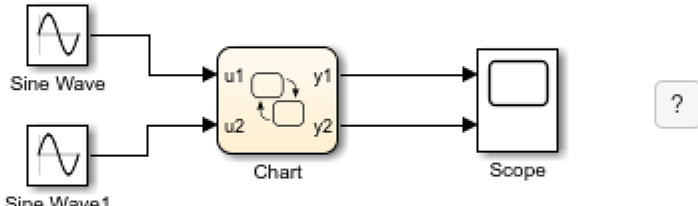
- “Create Reusable Subcomponents by Using Atomic Subcharts” on page 17-2
- “Reuse a State Multiple Times in a Chart” on page 17-32
- “Model an Elevator System by Using Atomic Subcharts” on page 17-52
- “Model a Redundant Sensor Pair by Using Atomic Subcharts” on page 17-48

Divide a Chart into Separate Units

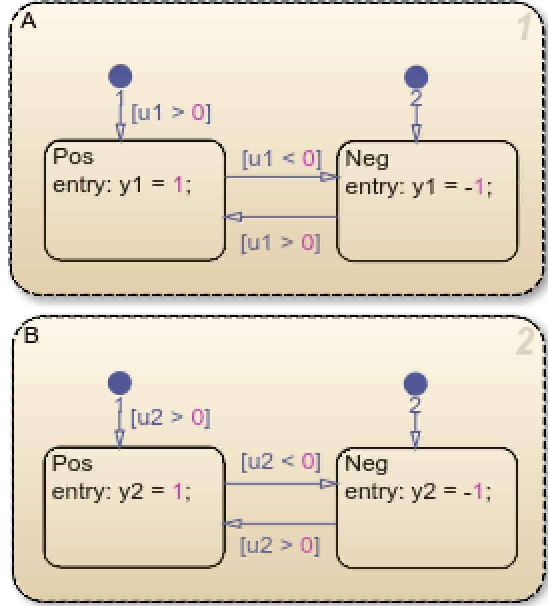
This example shows how to use linked atomic subcharts to break a Stateflow® chart into subcomponents so that multiple people can work on different parts of the chart. Atomic subcharts are not supported in standalone Stateflow charts in MATLAB®. For more information, see “Create Reusable Subcomponents by Using Atomic Subcharts” on page 17-2.

Original Model Without Atomic Subcharts

This model contains two Sine Wave (Simulink) blocks: one with a frequency of 1 radian per second, and the other with a frequency of 2 radians per second.



In the chart, each state uses saturator logic to convert the input sine wave to an output square wave of the same frequency.



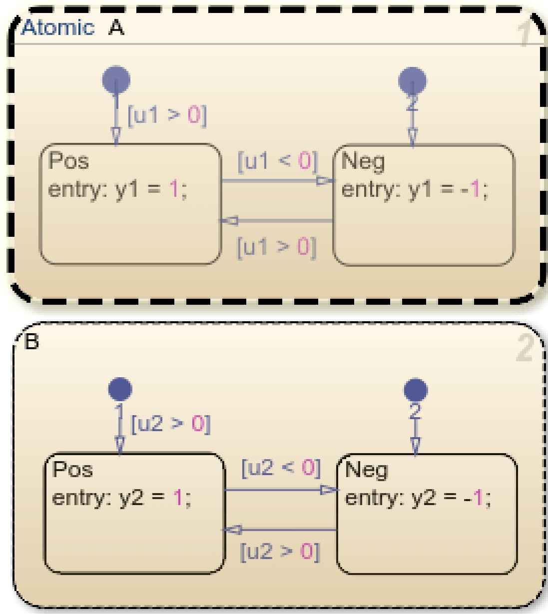
Because this example does not use atomic subcharts, only one person at a time can edit the model. If you edit state A while someone else edits state B, you must merge those changes at submission time.

In contrast, you can store different parts of this example as linked atomic subcharts. Because atomic subcharts behave as independent objects, different people can work on different parts of a chart without affecting the other parts of the chart. At submission time, no merge is necessary because the changes exist in separate models.

Edit Model to Use Atomic Subcharts

Step 1: Convert a State to an Atomic Subchart

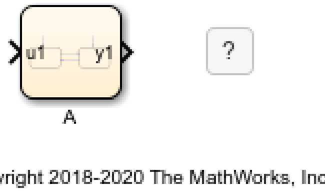
Right-click state A and select **Group & Subchart > Atomic Subchart**. State A changes to an atomic subchart and displays the label **Atomic** in the upper-left corner.



Step 2: Create a Library for the Atomic Subchart

1. Create a new library model.
2. Copy the atomic subchart and paste it in your library model.
3. Save your library model.

In the library model, the atomic subchart appears as an independent chart with an input port and an output port.

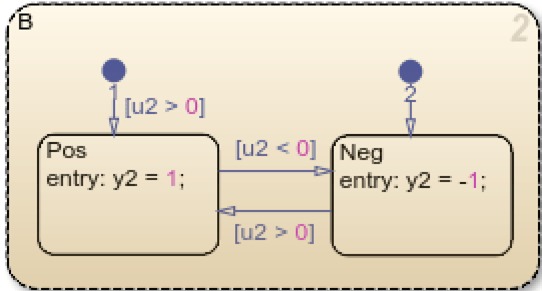


Copyright 2018-2020 The MathWorks, Inc.

Step 3: Replace State with Linked Atomic Subchart

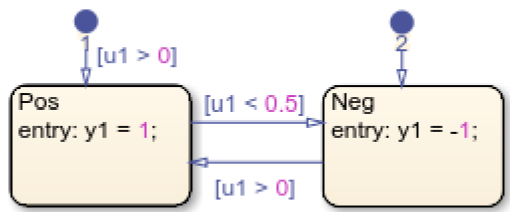
1. Delete state A in the chart.
2. Copy the atomic subchart in your library and paste it in your chart.

The linked atomic subchart appears opaque and contains the label **Link** in the upper-left corner.



Propagate a Change in the Library Chart

Suppose that, in the library chart, you edit the transition from `Pos` to `Neg`.



This change propagates to the linked atomic subchart in the main chart. If someone else edits the main chart, the changes are merged automatically.

See Also

Sine Wave

More About

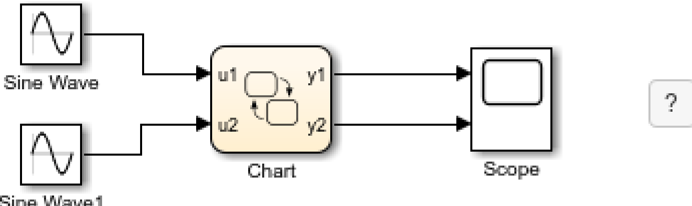
- “Create Reusable Subcomponents by Using Atomic Subcharts” on page 17-2
- “Reuse a State Multiple Times in a Chart” on page 17-32
- “Model an Elevator System by Using Atomic Subcharts” on page 17-52
- “Model a Redundant Sensor Pair by Using Atomic Subcharts” on page 17-48

Generate Separate Code for an Atomic Subchart

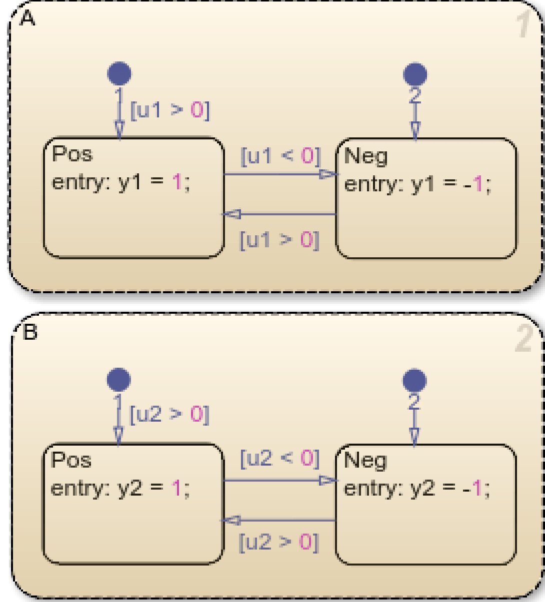
This example shows how to use atomic subcharts to generate code for individual parts of a Stateflow® chart. When you generate code for your chart, a separate file stores the code for the atomic subchart. Atomic subcharts are not supported in standalone Stateflow charts in MATLAB®. For more information, see "Create Reusable Subcomponents by Using Atomic Subcharts" on page 17-2.

Original Model Without Atomic Subcharts

This model contains two Sine Wave (Simulink) blocks: one with a frequency of 1 radian per second, and the other with a frequency of 2 radians per second.



In the chart, each state uses saturator logic to convert the input sine wave to an output square wave of the same frequency.



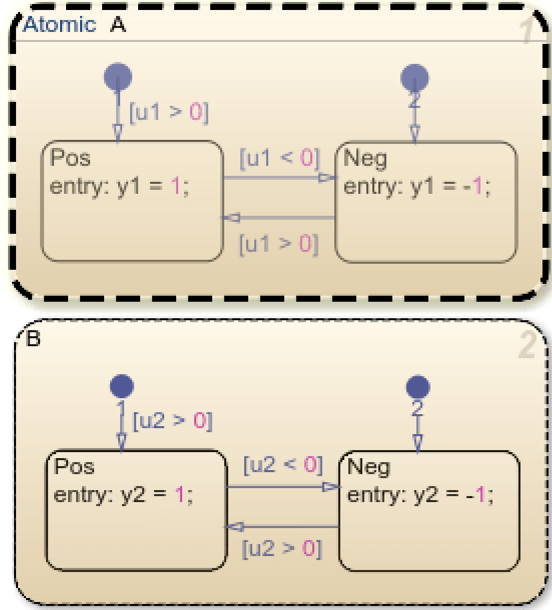
Because this example does not use atomic subcharts, generated code for the entire model is stored in one file. To find code for a specific part of the chart, you have to look through the entire file.

In contrast, you can convert state A to an atomic subchart and specify that the code for the subchart appears in a separate file. This method of code generation enables unit testing for a specific part of a chart. You avoid searching through unrelated code and focus only on the code that interests you.

Edit Model to Use Atomic Subcharts

Step 1: Convert a State to an Atomic Subchart

Right-click state A and select **Group & Subchart > Atomic Subchart**. State A changes to an atomic subchart and displays the label **Atomic** in the upper-left corner.



Step 2: Set Up a Standalone C File for the Atomic Subchart

1. Open the properties dialog box for subchart A by right-clicking the subchart and selecting **Properties**.
2. Set the **Code generation function packaging** property to `Reusable function`.
3. Set the **Code generation file name options** property to `User specified`.
4. In the **Code generation file name** box, enter `saturator` as the name of the file.
5. Click **OK**.

Step 3: Set Up the Code Generation Report

1. Open the Configuration Parameters dialog box.
2. In the **Code Generation** pane, set the **System target file** parameter to `ert.tlc`.
3. In the **Code Generation > Report** pane, select **Create code generation report**.
4. Under **Advanced parameters**, select **Model-to-code**.
5. Click **Apply**.

Step 4: Customize the Generated Function Names

In the Configuration Parameters dialog box, in the **Code Generation > Identifiers** pane, set the **Subsystem methods** parameter to the format scheme `$R$N$M$F`, where:

- `$R` is the root model name.

- $N is the block name.
- $M is the mangle token.
- $F is the type of interface function for the atomic subchart.

Generate Code

To generate code for your model by using Embedded Coder®, press **Ctrl+B**.

The code generation report contains links to the code generated from the chart (**Model files**) and the atomic subchart (**Subsystem files**). To inspect the code for the subchart, click the `saturator.c` hyperlink.

Code Generation Report

Find: Match Case

Contents

Summary
Subsystem Report
Code Interface Report
Traceability Report
Static Code Metrics Report
Code Replacements Report
Coder Assumptions

Generated Code

[-] **Main file**
ert_main.c

[-] **Model files**
ex_reuse_states.c
ex_reuse_states.h
ex_reuse_states_private.h
ex_reuse_states_types.h

[-] **Subsystem files**
saturator.c
saturator.h

[-] **Data files**
ex_reuse_states_data.c

[+] **Utility files (1)**

```
18
19 #include "saturator.h"
20
21 /* Include model header file for global data */
22 #include "ex_reuse_states.h"
23 #include "ex_reuse_states_private.h"
24
25 /* Named constants for Chart: '<S1>/A' */
26 #define ex_reuse_sta_IN_NO_ACTIVE_CHILD ((uint8_T)0U)
27 #define ex_reuse_states_IN_Neg         ((uint8_T)1U)
28 #define ex_reuse_states_IN_Pos         ((uint8_T)2U)
29
30 /* Function for Chart: '<S1>/A' */
31 void ex_reuse_states_A_during(real_T rtu_u1, real_T *rty_y1,
32   rtDW_A_ex_reuse_states *localDW)
33 {
34   /* During: Chart/A */
35   if (localDW->is_c3_ex_reuse_states == ex_reuse_states_IN_Neg) {
36     /* During 'Neg': '<S2>:9' */
37     if (rtu_u1 > 0.0) {
38       /* Transition: '<S2>:7' */
39       localDW->is_c3_ex_reuse_states = ex_reuse_states_IN_Pos;
40
41       /* Entry 'Pos': '<S2>:8' */
42       *rty_y1 = 1.0;
43     }
44   } else {
45     /* During 'Pos': '<S2>:8' */
46     if (rtu_u1 < 0.0) {
47       /* Transition: '<S2>:6' */
48       localDW->is_c3_ex_reuse_states = ex_reuse_states_IN_Neg;
49
50       /* Entry 'Neg': '<S2>:9' */
51       *rty_y1 = -1.0;
52     }
53   }
54 }
55
56 /* Function for Chart: '<S1>/A' */
```

OK Help

Line 31 shows that the `during` function generated for the atomic subchart has the name `ex_reuse_states_A_during`. This name follows the format scheme `$R$N$M$F` specified for Subsystem methods:

- The root model name is `ex_reuse_states`.
- The block name is `A`.
- The mangle token is empty.
- The type of interface function for the atomic subchart is `during`.

Note: The line numbers that appear in your code generation report can differ from the numbers shown.

See Also

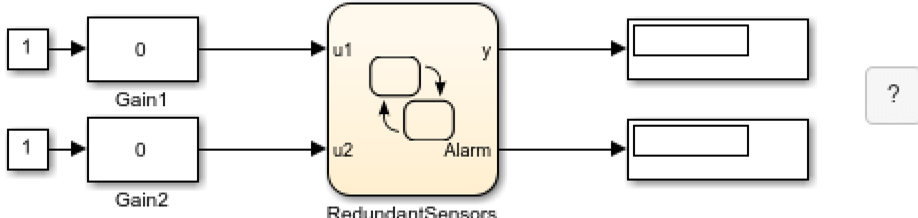
Sine Wave

More About

- "Create Reusable Subcomponents by Using Atomic Subcharts" on page 17-2
- "Reuse a State Multiple Times in a Chart" on page 17-32
- "Generate Code from Atomic Subcharts" on page 29-16

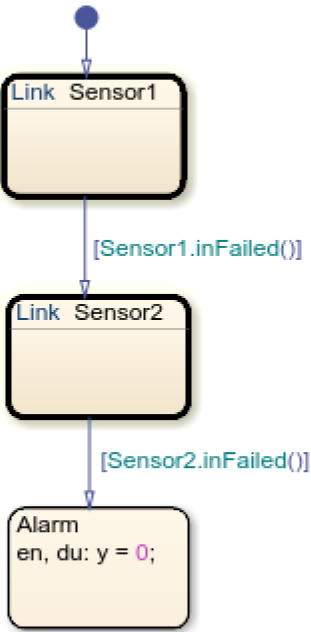
Model a Redundant Sensor Pair by Using Atomic Subcharts

This model shows how to model a redundant pair of sensors. By using atomic subcharts, you can compose a large Stateflow® chart from other charts that reside in a library file.



Main Chart

In this model, the chart RedundantSensors contains two linked atomic subcharts (`Sensor1` and `Sensor2`) and a state (`Alarm`).

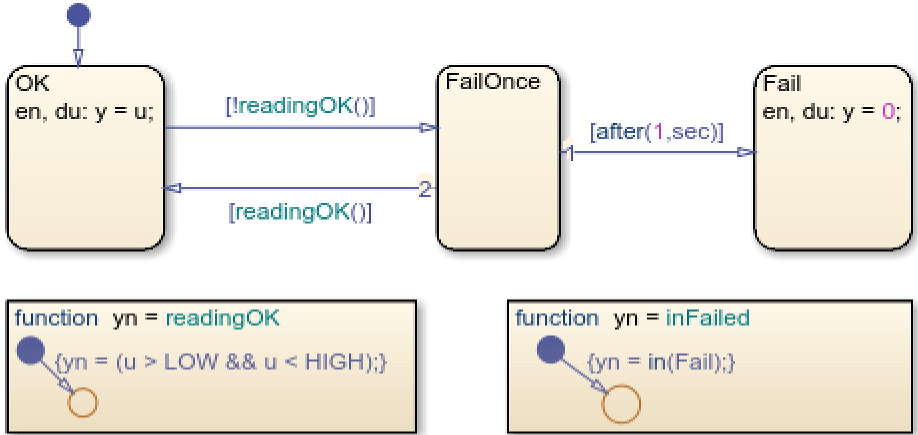


At the start of the simulation, the subchart `Sensor1` is active. This atomic subchart reads the input `u1`. If the input value remains between 75 and -75, `Sensor1` passes this value as the output of `y`. Otherwise, the sensor fails and subchart `Sensor2` becomes active.

In a similar way, `Sensor2` reads the input `u2` and checks that its value remains between 100 and -100. `Sensor2` passes this value as the output of `y`. Otherwise, the sensor fails and the chart transitions to the `Alarm` state.

Library Chart

The logic for both `Sensor1` and `Sensor2` is defined in a library model. In this model, the chart SingleSensor accepts an input `u` and provides a filtered sensor output `y`.



The chart detects out-of-range errors in the sensor input `u`. Initially, the sensor is in the state `OK`. If `u` goes out of range, the chart takes the transition from `OK` to the state `FailOnce`. If `u` stays out of range for longer than one second, then the chart transitions to the state `Fail`. In this case, the sensor outputs a constant value of zero. This pattern allows the sensor to ignore transient spikes in the sensor reading.

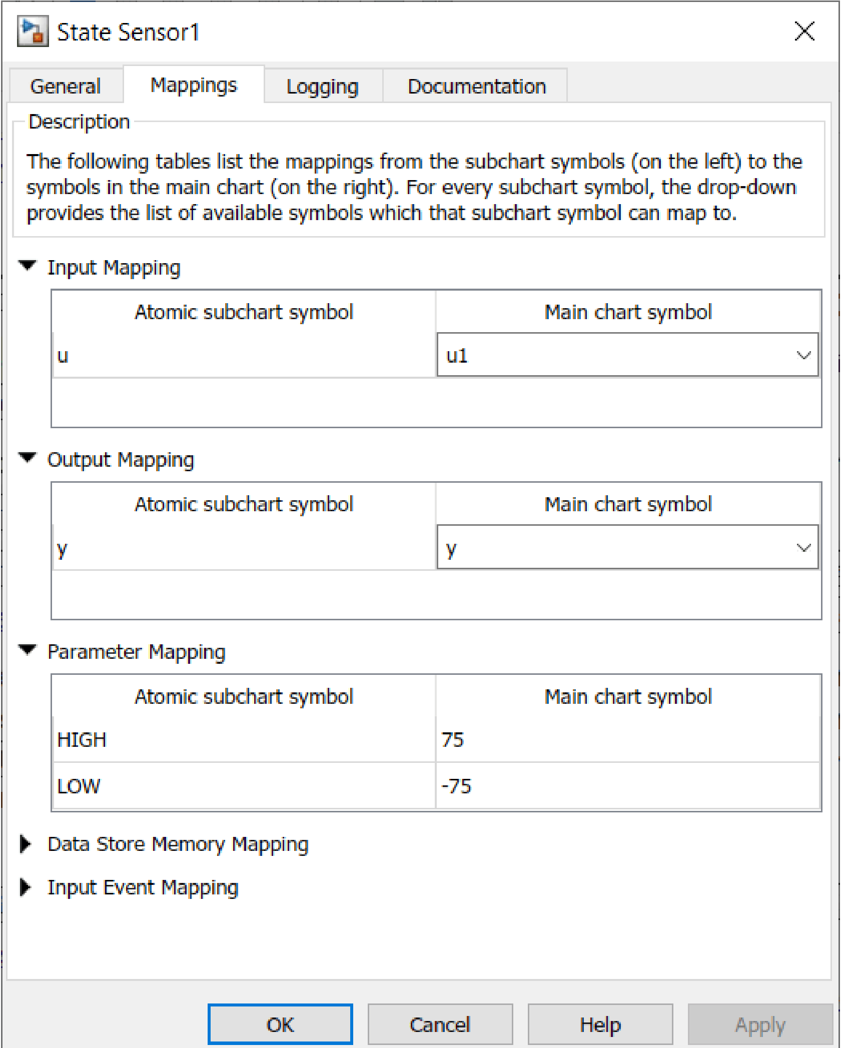
Map Inputs, Outputs, and Parameters

The chart RedundantSensors has two inputs (`u1` and `u2`), while the library chart that defines the atomic subcharts has only one input (`u`). To enable the atomic subcharts to access a different chart input, right-click each subchart and select **Subchart Mappings**. In the **Mappings** tab of the properties dialog box, you can:

- Specify which symbol in the main chart corresponds to each symbol in the subchart.
- Assign values to parameters defined in the subchart.

For example, in the case of subchart `Sensor1`:

- The subchart input `u` is mapped to the main chart input `u1`.
- The subchart output `u` is mapped to the main chart output `y`.
- The subchart parameters `HIGH` and `LOW` are assigned the values 75 and -75.

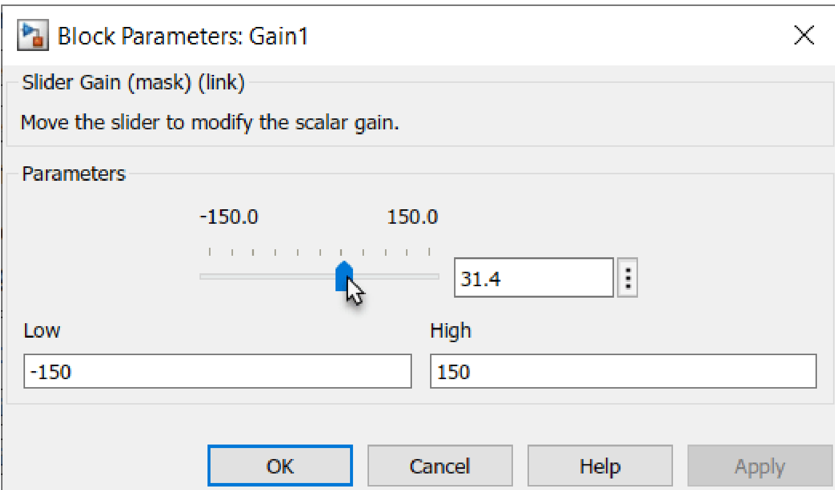


In the case of subchart `Sensor2`:

- The subchart input `u` is mapped to the main chart input `u2`.
- The subchart output `u` is mapped to the main chart output `y`.
- The subchart parameters `HIGH` and `LOW` are assigned the values 100 and -100.

Simulation Behavior

Initially, both inputs and both outputs to the chart are zero. To change the value of the chart inputs, double-click the Gain blocks and drag the slider.



As long as the value of u1 is between -75 and 75, the output value y tracks the input value u1. If the value of u1 exceeds these bounds, the value of y begins to track the input value u2. If the value of u2 falls outside the range from -100 to 100, y returns a value of zero and Alarm returns a value of one.

See Also

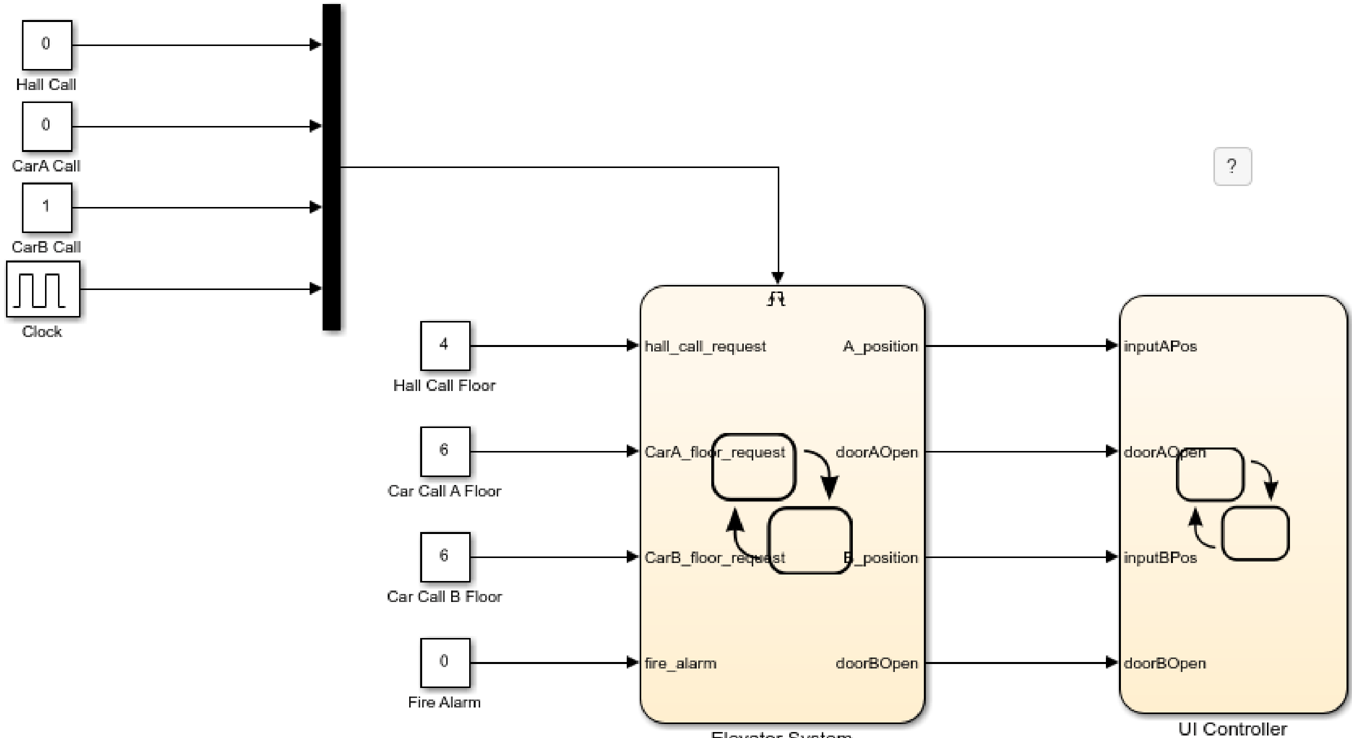
More About

- “Create Reusable Subcomponents by Using Atomic Subcharts” on page 17-2
- “Map Variables for Atomic Subcharts and Boxes” on page 17-11
- “Reuse Logic Patterns by Defining Graphical Functions” on page 6-9

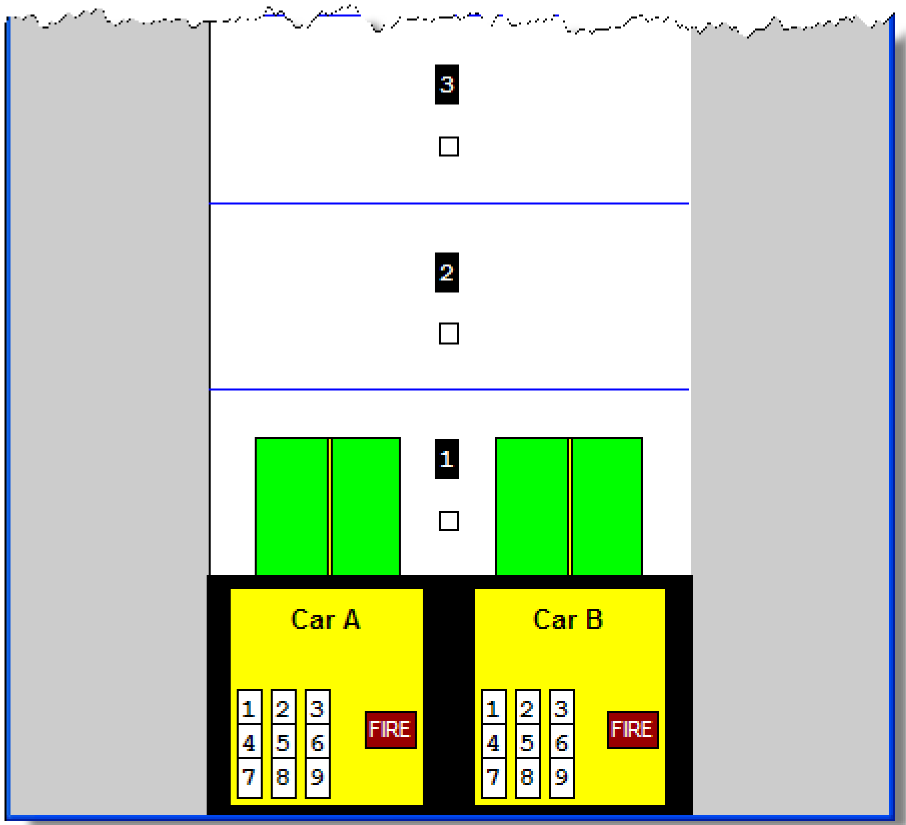
Model an Elevator System by Using Atomic Subcharts

This example shows how to model a two-car elevator system by using linked atomic subcharts in Stateflow®. The elevator system consists of a Simulink® model and a user interface (UI). The model contains two Stateflow charts:

- Elevator System models the core logic that delegates incoming requests from the UI to the nearest available elevator car. This chart contains a pair of atomic subcharts that implement identical logic for the cars.
- UI Controller processes information from the Elevator System chart and updates the UI display. In this chart, each atomic subchart determines when to move an elevator car and when to open its doors.



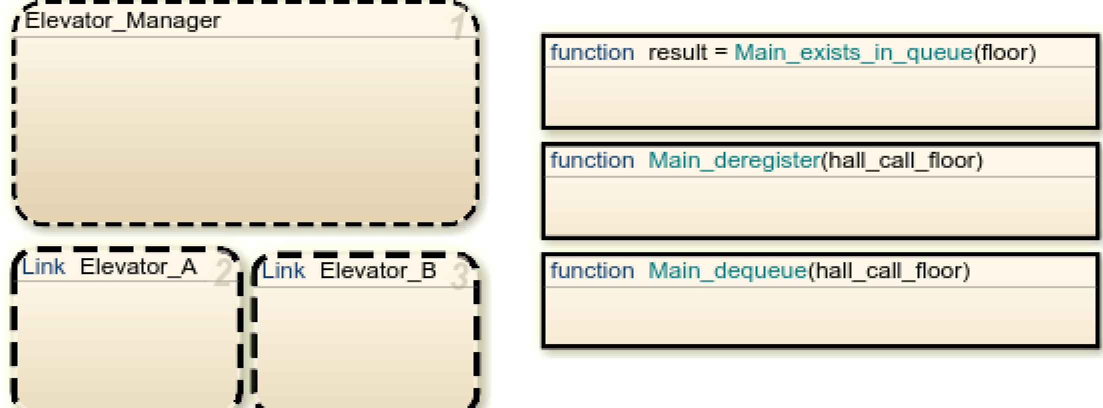
At the start of simulation, the model opens the UI. The UI shows two elevator cars that can stop at nine floors. At the bottom of the UI, two yellow rectangles represent the interior of the elevator cars. While the example is running, you call an elevator car, request a stop at a floor, or set off a fire alarm by clicking the buttons on each floor hallway and inside the elevator cars. The UI responds by modifying the input values and triggering input events for the Elevator System chart.



Manage Requests from User Interface

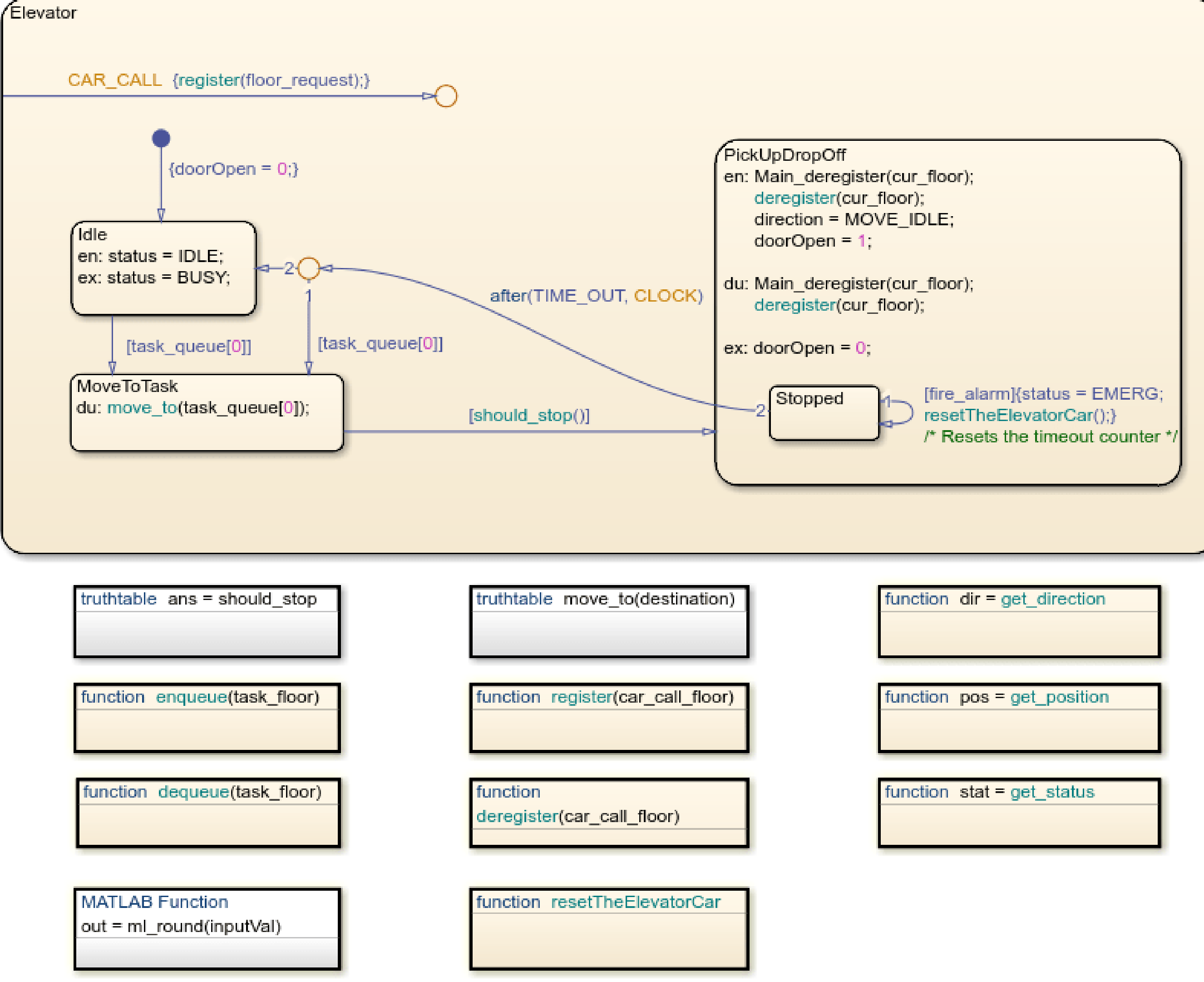
The Elevator System chart consists of three parallel subcharts. Each of these subcharts manages a queue of requests from the UI:

- The `Elevator_Manager` subchart implements the main control logic for the elevator system. This subchart manages the hall queue, which holds all the requests that are generated when you click a button in one of the floor hallways. The subchart processes these requests and delegates them to one of the elevator cars, depending on availability and proximity to the request.
- `Elevator_A` and `Elevator_B` represent the logic for the two elevator cars. Each car has its own queue that holds all of its floor requests. Floor requests are generated when you click a button inside the elevator car or when the `Elevator_Manager` delegates a request from the hall queue to the car.



Reuse Logic Patterns by Using Atomic Subcharts

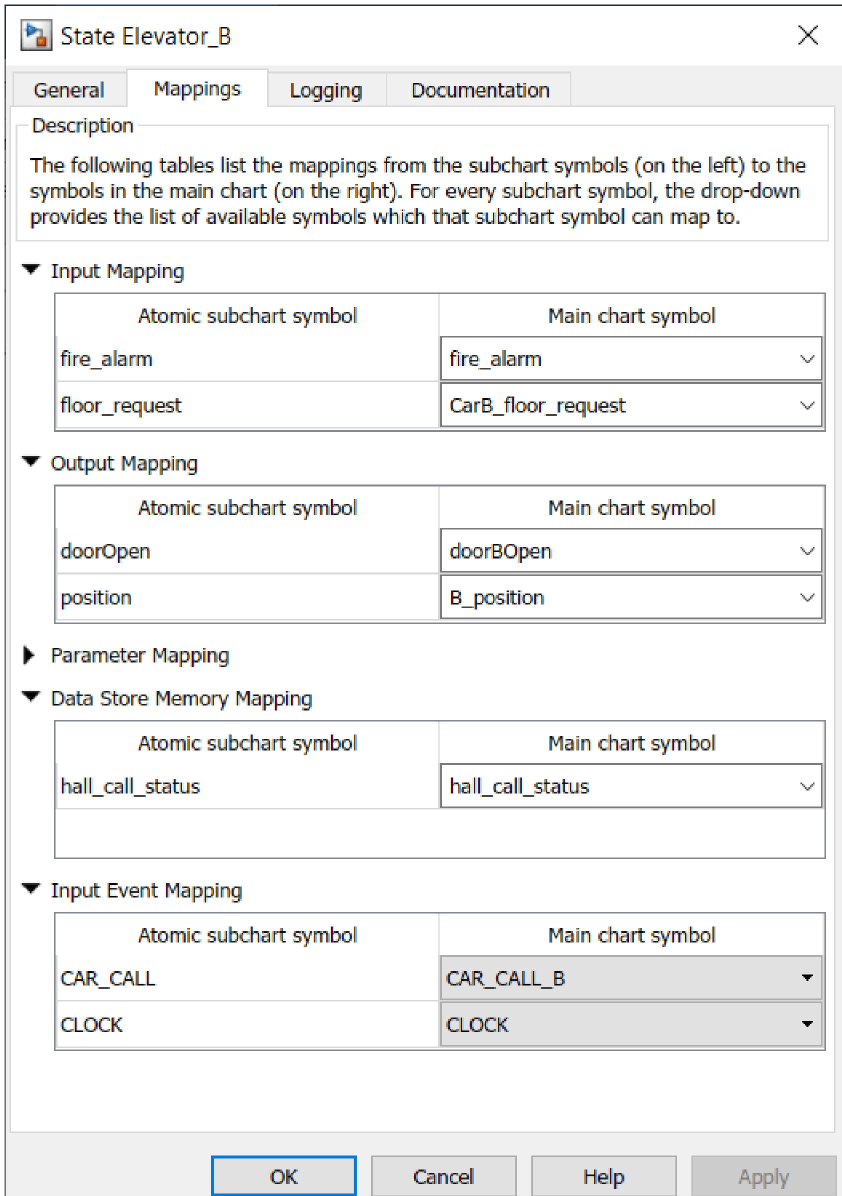
The elevator cars use identical logic to process their individual request queues. The Elevator System chart models their behavior by using linked atomic subcharts from a library model.



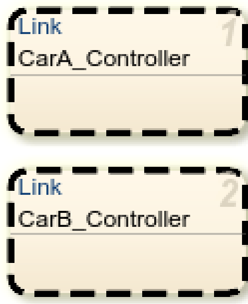
In the library model, the chart `Elevator` implements the logic for a generic elevator car. To program the subcharts `Elevator_A` and `Elevator_B` so that they control the appropriate car, you map data and events in each subchart to the corresponding data and events in the main chart. For instance, for `Elevator_B`:

- The subchart input `floor_request` maps to the chart input `CarB_floor_request`.
- The subchart output `position` maps to the chart output `B_position`.
- The subchart output `doorOpen` maps to the chart output `doorBOpen`.
- The subchart event `CAR_CALL` maps to the chart event `CAR_CALL_B`.

To see the mappings for each atomic subchart, right-click the subchart and select **Subchart Mappings**.



To control the UI display of each elevator car, the UI Controller chart uses two atomic subcharts linked from a library model.

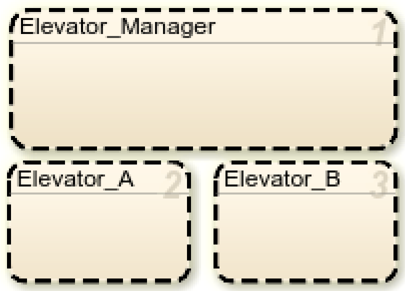


For more information about using atomic subcharts to encapsulate and reuse logic, see “Create Reusable Subcomponents by Using Atomic Subcharts” on page 17-2.

Incorporate Atomic Subcharts in Your Design

The model in this example is a redesigned version of an older model that does not use atomic subcharts. The original model uses separate subcharts to manage floor requests (subcharts `Elevator_A` and `Elevator_B` of the Elevator System chart) and to control the UI display of elevator cars (subcharts `CarA_Controller` and `CarB_Controller` of the UI Controller chart). In each case, the subcharts are nearly identical copies of one another. They differ only in the names of the data and events that they use.

To convert the duplicate subcharts to atomic subcharts, first make a library atomic subchart out of one of the subcharts. Then use linked instances of this library to replace the duplicate subcharts. For example, consider the duplicate elevator car subcharts of the Elevator System chart. These subcharts call several functions and local variables that are defined in the `Elevator_Manager` subchart. Before creating an atomic subchart, you must make these subcharts independent and self-contained units.



1. Migrate these functions from the `Elevator_Manager` subchart into the parent chart:

- `exists_in_queue`
- `deregister`
- `dequeue`

Rename these functions to distinguish them from the functions inside the elevator car subcharts.

2. Using the Model Explorer, migrate these variables from the `Elevator_Manager` subchart into the parent chart:

- `hall_call_queue`
- `hall_call_status`

3. In the Elevator System chart, set the **Export Chart Level Functions** chart property to `true`. For more information, see “Export Stateflow Functions for Reuse” on page 6-14.

4. Modify the `Elevator_Manager` and `Elevator_A` subcharts to use the migrated functions and variables.

5. Create a library atomic subchart from the `Elevator_A` subchart, as described in “Reuse a State Multiple Times in a Chart” on page 17-32.

6. To enable the atomic subchart to pass the position of the elevator car to the containing chart, change the **Scope** of the subchart data `position` from `Local` to `Output`.

7. Replace the two elevator subcharts with the linked atomic subchart. For each linked atomic subchart, map data and events to the parent chart. For more information, see “Map Variables for Atomic Subcharts and Boxes” on page 17-11.

See Also

More About

- “Create Reusable Subcomponents by Using Atomic Subcharts” on page 17-2
- “Export Stateflow Functions for Reuse” on page 6-14
- “Reuse a State Multiple Times in a Chart” on page 17-32
- “Map Variables for Atomic Subcharts and Boxes” on page 17-11

Save and Restore Simulations with Operating Points

- “Save and Restore Operating Points for Stateflow Charts” on page 18-2
- “Use Operating Points to Specify Initial State of Simulation” on page 18-7
- “Test Difficult-to-Reproduce Chart Configurations” on page 18-12
- “Test Chart with Fault Detection and Redundant Logic” on page 18-18

Save and Restore Operating Points for Stateflow Charts

An operating point is a snapshot of a Simulink model during simulation. If your model contains a Stateflow chart, the operating point includes information about:

- Active states
- Chart output data
- Chart, state, and function local data
- Persistent variables in MATLAB functions and truth tables

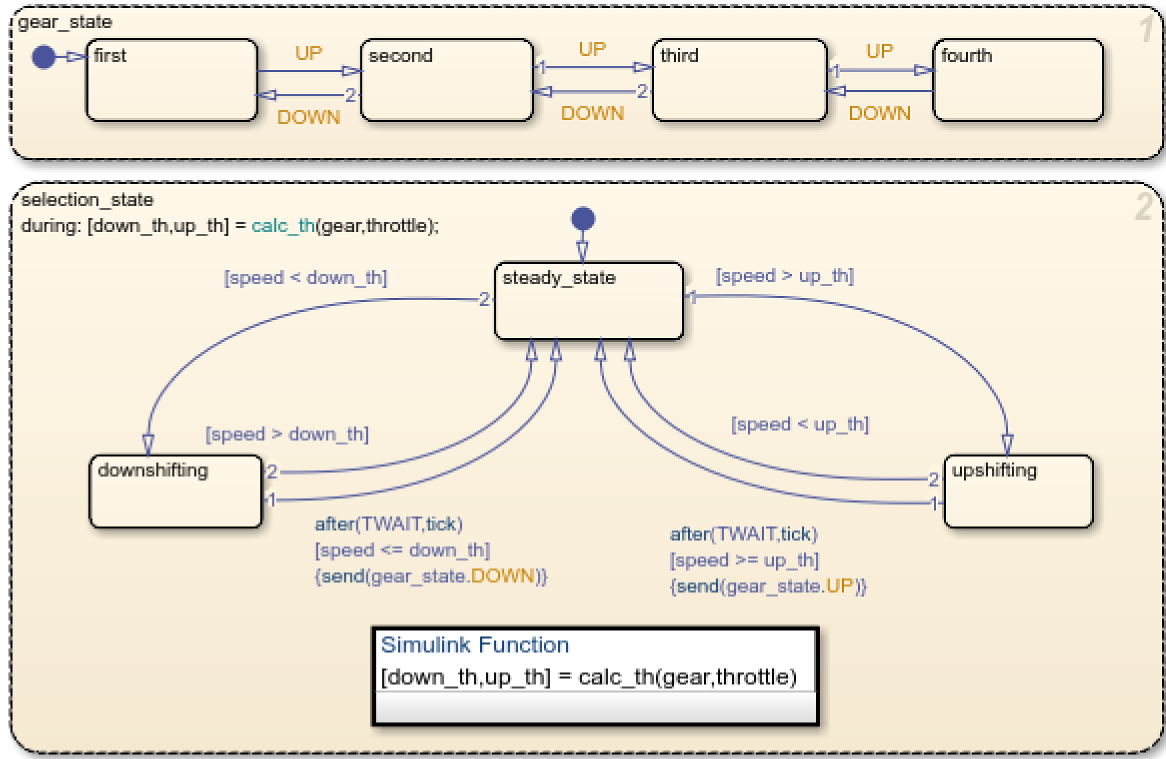
You can save the state of a model at a certain time step as an operating point and then use that operating point as the initial state for a simulation. For example, suppose that your simulation starts with an initialization phase. If you save an operating point after the model completes the initialization phase, you can use the operating point as the initial state for your model and get the results of a continuous simulation without starting from time $t = 0$. For more information, see “Use Operating Points to Specify Initial State of Simulation” on page 18-7.

You can also use operating points to test the response of a Stateflow chart to different settings, including configurations that are difficult to reach in simulation. You can modify an operating point by changing the values of local, output, or persistent data, the states that are currently active, and the previous state activity recorded by history junctions midway through a simulation. Then, you can use the modified operating point as a starting point of a simulation to test how the chart responds to your changes. For more information, see “Test Difficult-to-Reproduce Chart Configurations” on page 18-12 and “Test Chart with Fault Detection and Redundant Logic” on page 18-18.

Save Operating Points

To save the final operating point of a Simulink model, select the configuration parameters **Final states** and **Save final operating point**, specify a variable for the operating point for the model, and run a simulation. Then, access the operating point information for your Stateflow chart by calling the `get` function using the operating point for the model and the block path to the chart.

For example, suppose that `xFinal` is the operating point for the model `sf_car`, which contains the chart `shift_logic`.



To access the operating point information for this chart, enter:

```
op = get(xFinal,"sf_car/shift_logic")
```

```
op = 

  Block:    "shift_logic"    (handle)    (active)
  Path:     sf_car/shift_logic

  Contains:

    + gear_state         "State (AND)"          (active)
    + selection_state    "State (AND)"          (active)
      gear               "State output data"    gearType [1, 1]
      down_th            "Local scope data"     double [1, 1]
      up_th              "Local scope data"     double [1, 1]
```

The operating point lists the states, boxes, functions, and data in the chart in hierarchical order. If name conflicts exist, one or more underscores appear at the end of a name so that all objects have unique identifiers.

To access the operating point information for an object in the chart, use dot notation. For example, to access the operating point information for the top-level state `gear_state`, enter:

```
op.gear_state
```

```
ans = 

State: "gear_state"    (handle)    (active)
```

```
  Path:     sf_car/shift_logic/gear_state

  Contains:

    + first          "State (OR)"
    + fourth         "State (OR)"
    + second         "State (OR)"
    + third          "State (OR)"         (active)
```

Similarly, to access the operating point information for the chart output `gear`, enter:

```
op.gear
```

```
ans =

     Description: 'State output data'
        DataType: 'gearType'
            Size: '[1, 1]'
           Range: [1×1 struct]
    InitialValue: []
           Value: third
```

For more information about this example, see “Simulate Chart as a Simulink Block With Local Events”.

Note Stateless flow charts have an empty operating point because they do not contain states or persistent data.

Copy Operating Points

To create a copy of the operating point for a Stateflow chart, call the `clone` function. For example, suppose that you try to copy the operating point `op` by entering:

```
op1 = op;
op2 = clone(op);
```

In this case, `op1` refers to the same operating point as `op`, so modifying `op1` also modifies `op`. In contrast, `op2` refers to a different operating point that you can modify without modifying `op`.

Note The `clone` function copies the operating point information for the entire Stateflow chart. You cannot copy the operating point information for a state or data object.

Modify Operating Point Values

To test the response of your Stateflow chart to different settings, you can modify an operating point by changing the values of the local, output, or persistent data, the states that are currently active, and the previous state activity recorded by history junctions during a simulation.

Modify Data Values

To modify the values of local, output, or persistent data, change the `Value` property of the operating point. For example, suppose that `op` contains the operating point for a chart. To modify value of a chart output called `output`, enter:

```
op.output.Value = newValue;
```

When you modify data values:

- You cannot change the data type or size.
- Refer to elements of a vector or matrix by using one-based indexing delimited by parentheses and commas, even if your chart uses C as the action language.
- New values for numeric data must be within the range that you specify in the **Minimum** and **Maximum** parameters. For more information, see “Limit range” on page 10-10.
- For enumerated data types, you can choose only enumerated values from the type definition.
- You cannot modify the values of persistent data in custom C code or external MATLAB code.

Modify Current State Activity

To change the states that are currently active in an operating point, call the `setActive` function using a leaf state as an argument. This function maintains state consistency by:

- Exiting and entering the appropriate states
- Resetting temporal counters for newly active states
- Updating values of active state data
- Enabling or disabling function-call subsystems and Simulink functions that bind to states

However, the chart does not perform `exit` actions for the previously active states or `entry` actions for the newly active state. Additionally, the state does not reinitialize any state-parented local data. If you want these actions to occur, you must execute them separately. For example, if your state actions assign values to data, you must assign the values explicitly as described in “Modify Data Values” on page 18-4.

Modify Previous State Activity

To change the previously active state recorded by a history junction, call the `setPrevActiveChild` function using the state that contains the history junction as an argument. The state must not be active when you call this function.

Load Modified Operating Point Information

To load modified operating point information for a Stateflow chart into the operating point for a Simulink model, call the `set` function. For example, if `xFinal` is the operating point for the model `sf_car` and `op` contains the modified operating point information for the chart `shift_logic`, you can save the modified operating point `xModified` by entering:

```
xModified = set(xFinal,"sf_car/shift_logic",op);
```

Note The `set` function loads the operating point information for the entire Stateflow chart. You cannot load the operating point information for a state or data object.

Restore Operating Points

To use an operating point as the initial state for a simulation, set the configuration parameter **Initial state** and specify the variable name for the operating point for the model. When you simulate your

model, the simulation starts at the time of the operating point. For more information, see “Use Model Operating Point for Faster Simulation Workflow” (Simulink).

Limitations on Operating Points

Continuous-Time Charts

Operating points for continuous-time charts are read-only. You can save an operating point for a continuous-time chart and use it as the initial state for a simulation. However, you cannot modify the state activity or any data values in the operating point. For more information on continuous-time charts, see “Continuous-Time Modeling in Stateflow” on page 22-2.

Charts That Use Edge Detection

Stateflow charts that use edge detection operators do not support operating points. If your model contains a chart that uses edge detection, attempting to save the final operating point results in a compile-time error. For more information on edge detection operators, see “Detect Changes in Data and Expression Values” on page 14-63.

See Also

Model Settings
Initial state | Final states | Save final operating point

Objects
`Stateflow.op.BlockOperatingPoint` | `Stateflow.op.OperatingPointContainer` | `Stateflow.op.OperatingPointData`

Functions
`setActive` | `setPrevActiveChild` | `clone` | `get` | `set`

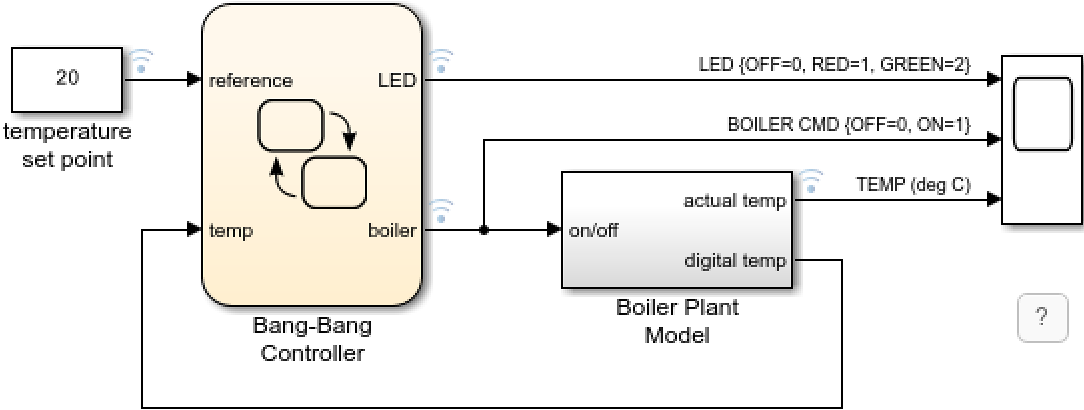
More About

- “Use Operating Points to Specify Initial State of Simulation” on page 18-7
- “Test Difficult-to-Reproduce Chart Configurations” on page 18-12
- “Test Chart with Fault Detection and Redundant Logic” on page 18-18
- “Use Model Operating Point for Faster Simulation Workflow” (Simulink)

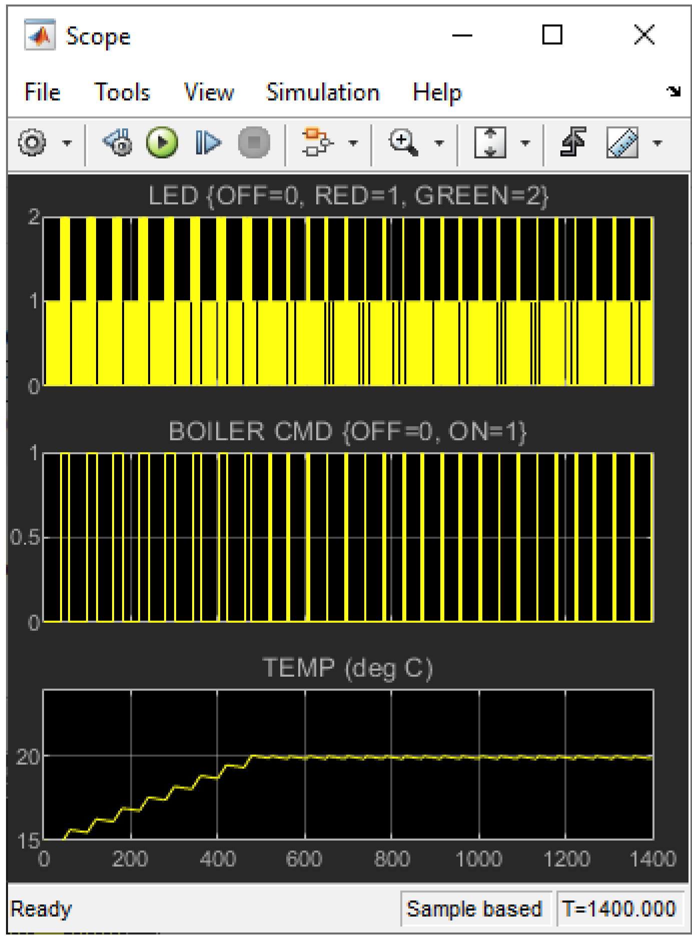
Use Operating Points to Specify Initial State of Simulation

This example shows how to use operating points to get the results of a continuous simulation without starting from time $t = 0$. An operating point is a snapshot of the state of a Simulink model during simulation. If your model contains a Stateflow chart, the operating point includes information about active states, output and local data, and persistent variables. For more information, see “Save and Restore Operating Points for Stateflow Charts” on page 18-2.

The `sf_boiler` model illustrates the logic of a bang-bang control system that regulates the temperature of a boiler.



This model simulates for 1400 seconds. During the initial phase of the simulation, the temperature of the boiler rises from 15℃ to 20℃. This initial phase of the simulation takes approximately 500 seconds.



To see how the system reacts to changes in the temperature set point after the initial phase of simulation, simulate the model and save the operating point at t =500. Then, load that operating point and run the simulation between t = 500 and t = 1400 using different temperature set points.

For more information about this model, see “Model Bang-Bang Temperature Control System” on page 14-51.

Save Operating Point for Initial Simulation Segment

1 Open the `sf_boiler` model.

```
openExample("stateflow/BangBangControlUsingTemporalLogicExample")
```

2 Set the model to save the final operating point. Open the Configuration Parameters dialog box and, in the **Data Import/Export** pane:

   a Select **Final states** and enter a name for the operating point. For this example, use `xFinal`.

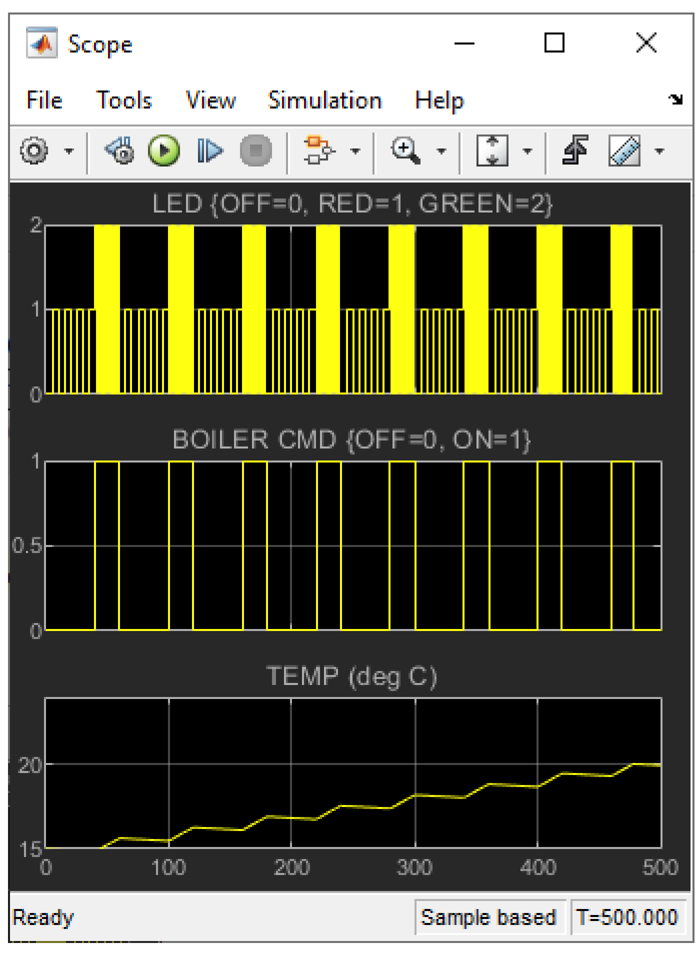
   b Select **Save final operating point**.

   c Click **OK**.

3 Set the stop time for this simulation segment. In the **Simulation** tab, set **Stop Time** to `500`.

4 Run the simulation.

When you simulate the model, you save the operating point at t = 500 in the variable `xFinal` in the MATLAB base workspace.



5 In the Configuration Parameters dialog box, in the **Data Import/Export** pane, clear the **Save final operating point** and **Final states** parameters. This step prevents you from overwriting the operating point you saved in the previous step.

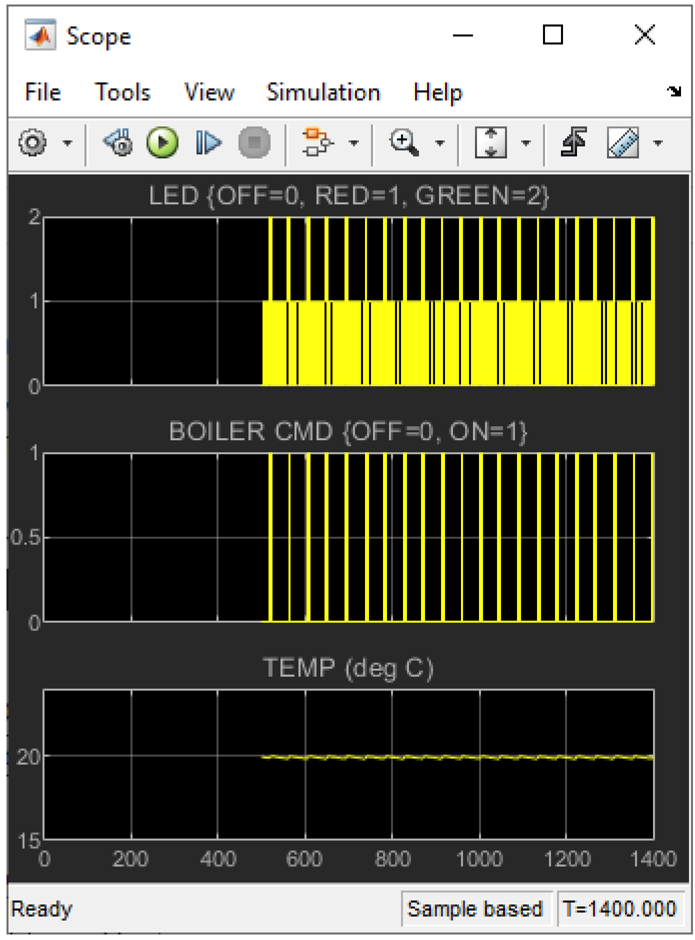
Start Simulation from Operation Point

1 Load the operating point as the initial state of the model. In the Configuration Parameters dialog box, in the **Data Import/Export** pane, select **Initial state** and enter the variable that contains the operating point of your chart, `xFinal`. Then, click **OK**.

2 Define the stop time for the new simulation segment. In the **Simulation** tab, set **Stop Time** to `1400`.

3 Run the simulation.

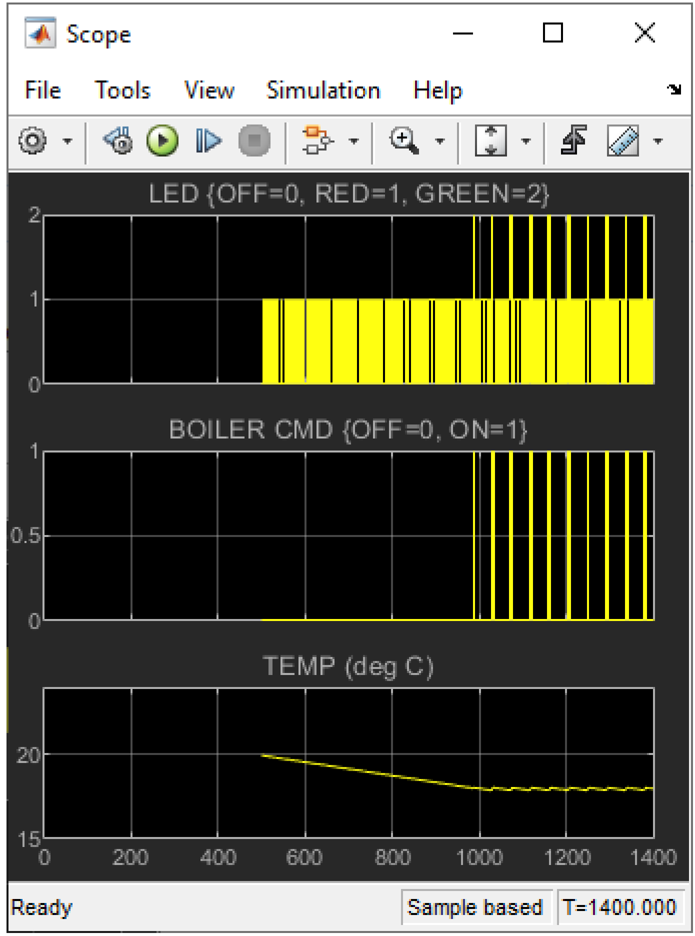
The model starts at t = 500 without repeating the initial phase of the simulation. The Scope block shows the chart output between t = 500 and t = 1400 using the original temperature set point of 20℃.



4 Change the value of the `temperature set point` block to `18`.

5 Run the simulation again.

The Scope block shows the chart output between t = 500 and t = 1400 using the modified temperature set point of 18℃.



See Also

Model Settings
Initial state | Final states | Save final operating point

Objects
`Stateflow.op.BlockOperatingPoint` | `Stateflow.op.OperatingPointContainer` | `Stateflow.op.OperatingPointData`

Functions
`setActive` | `setPrevActiveChild` | `clone` | `get` | `set`

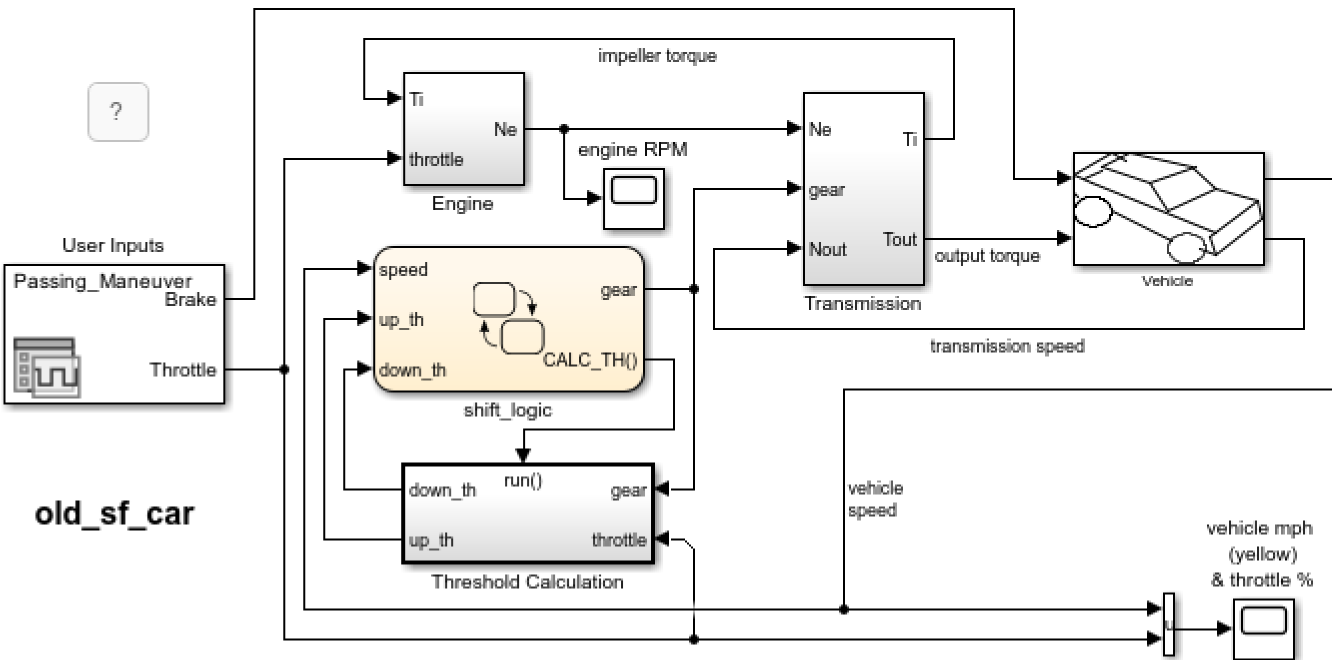
More About

- “Save and Restore Operating Points for Stateflow Charts” on page 18-2
- “Model Bang-Bang Temperature Control System” on page 14-51
- “Use Model Operating Point for Faster Simulation Workflow” (Simulink)

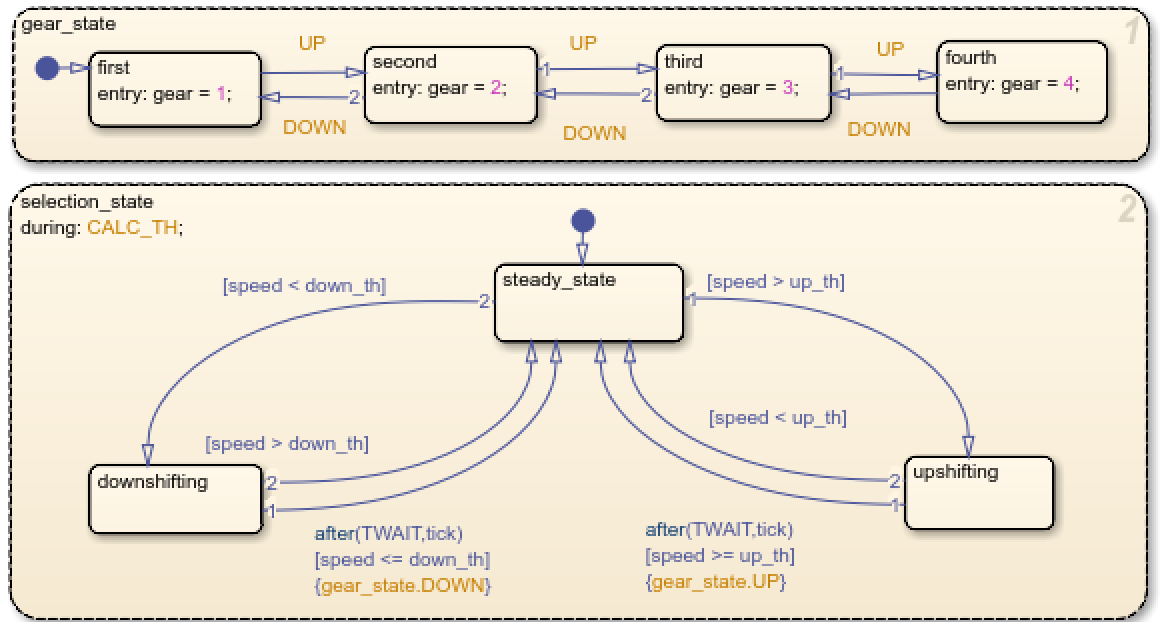
Test Difficult-to-Reproduce Chart Configurations

This example shows how to use operating points to test how a chart responds to a combination of active states and data values that is difficult to reproduce during simulation. An operating point is a snapshot of the state of a Simulink model during simulation. If your model contains a Stateflow chart, the operating point includes information about active states, output and local data, and persistent variables. For more information, see "Save and Restore Operating Points for Stateflow Charts" on page 18-2.

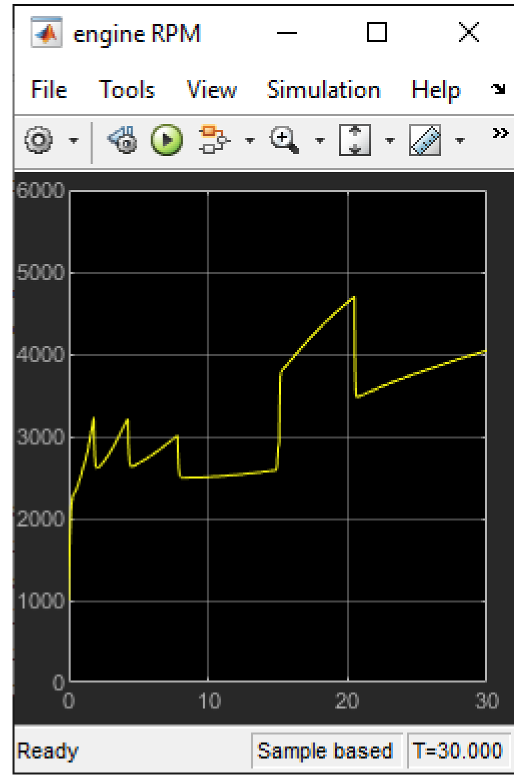
The model `old_sf_car` illustrates a four-speed automatic transmission system of a car.



In this model, the Stateflow chart `shift_logic` implements the logic for shifting between four gears. The substates of the parallel state `gear_shift` represent the gears in the transmission system. The entry actions in these substates set the value of the chart output `gear`. As a result, during normal simulation, the value of `gear` reflects which substate is active.



By default, the model simulates for 30 seconds.



To see what happens when the value of the chart output `gear` changes abruptly, simulate the model, save the operating point at $t = 10$, modify the operating point to reflect a change in the chart output, and then simulate again between $t = 10$ and $t = 30$.

Define Operating Point for Initial Segment

1 Open the model `old_sf_car`.

```
openExample("stateflow/AutomaticTransmissionLegacyExample")
```

2 Set the model to save the final operating point. Open the Configuration Parameters dialog box and, in the **Data Import/Export** pane:

   a Select **Final states** and enter a name for the operating point. For this example, use `xSteadyState`.

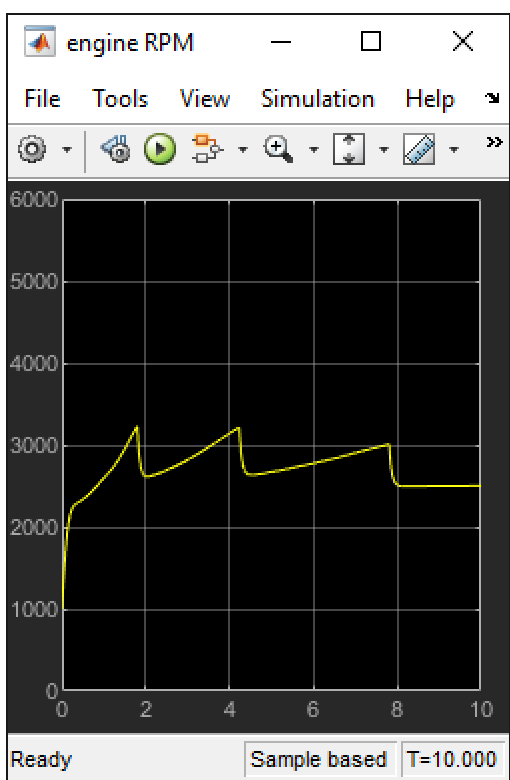
   b Select **Save final operating point**.

   c Click **OK**.

3 Set the stop time for this simulation segment. In the **Simulation** tab, set **Stop Time** to `10`.

4 Run the simulation.

When you simulate the model, you save the final operating point at $t = 10$ in the variable `xSteadyState` in the MATLAB base workspace. At $t = 10$, the engine operates at a steady-state value of 2500 RPM.



5 In the Configuration Parameters dialog box, in the **Data Import/Export** pane, clear the **Save final operating point** and **Final states** parameters. This step prevents you from overwriting the operating point you saved in the previous step.

Modify Operating Point for Change in Data Value

1 Access the `Stateflow.op.BlockOperatingPoint` object that contains the operating point information for the `shift_logic` chart.

```
blockpath = "old_sf_car/shift_logic";
op = get(xSteadyState,blockpath)
```

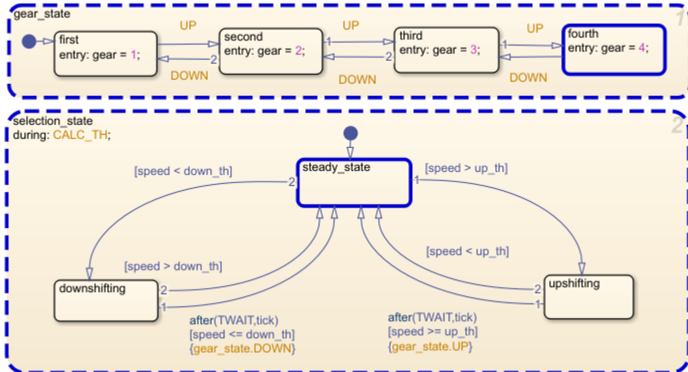
```
op = 

Block:    "shift_logic"    (handle)    (active)
  Path:     old_sf_car/shift_logic

  Contains:

    + gear_state          "State (AND)"          (active)
    + selection_state     "State (AND)"          (active)
      gear                "Block output data"    double [1, 1]
```

The operating point contains a list of states and data in hierarchical order.

2 Highlight the states that are active in the chart at $t = 10$.

```
highlightActiveStates(op)
```

Active states appear highlighted. The transmission system is in a steady state in fourth gear.



3 Access the `Stateflow.op.OperatingPointData` object that contains the operating point information for the chart output `gear`.

```
op.gear
```

```
ans = 

      Description: 'Block output data'
         DataType: 'double'
             Size: '[1, 1]'
            Range: [1x1 struct]
     InitialValue: [1x0 double]
            Value: 4
```

4 Change the value of `gear` to `1`.

```
op.gear.Value = 1;
```

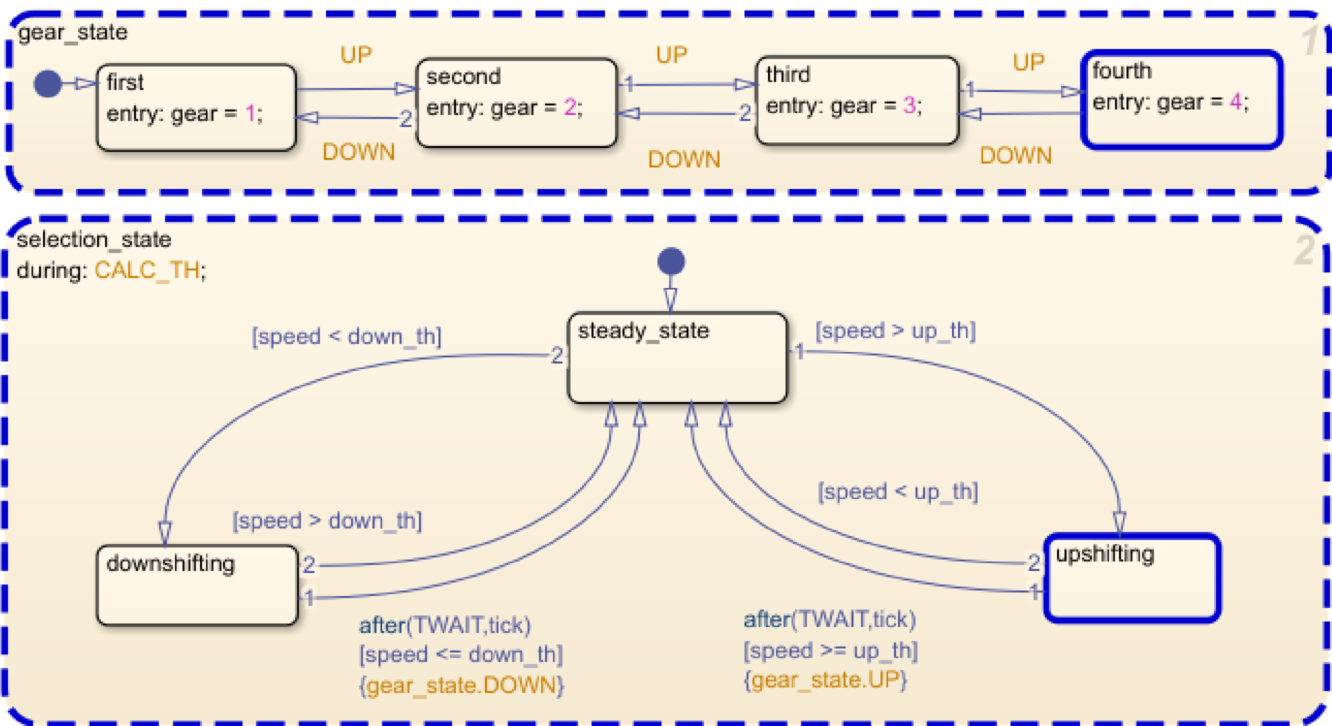
5 Save the modified operating point.

```
xModified = set(xSteadyState,blockpath,op);
```

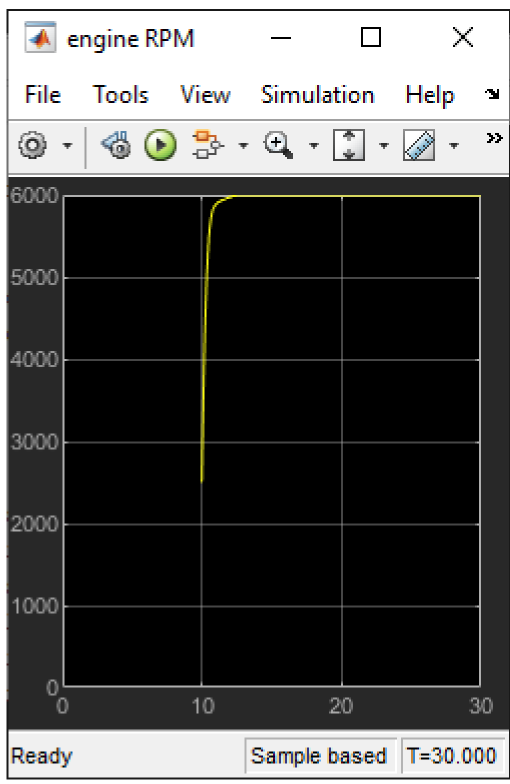
Test Model Behavior after Change in Data Value

1. Load the operating point as the initial state of the model. In the Configuration Parameters dialog box, in the **Data Import/Export** pane, select **Initial state** and enter the variable that contains the modified operating point of your chart, `xModified`. Then, click **OK**.
2. Define the stop time for the simulation segment to test. In the **Simulation** tab, set **Stop Time** to `30`.
3. Run the simulation.

   The chart animation shows that the system continually attempts to compensate for the high vehicle speed relative to the chart output `gear` by shifting up, but is unable to because the system is already in fourth gear.



   As a result, the engine RPM increases to 6000 RPM.



See Also

Model Settings
Initial state | **Final states** | **Save final operating point**

Objects
`Stateflow.op.BlockOperatingPoint` | `Stateflow.op.OperatingPointContainer` | `Stateflow.op.OperatingPointData`

Functions
`highlightActiveStates` | `get` | `set`

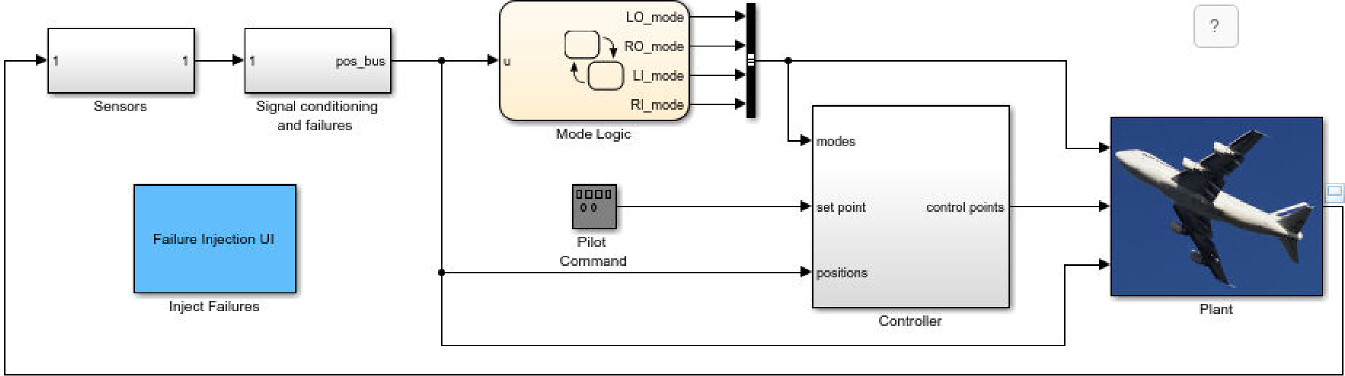
More About

- “Save and Restore Operating Points for Stateflow Charts” on page 18-2
- “Test Chart with Fault Detection and Redundant Logic” on page 18-18
- “Use Model Operating Point for Faster Simulation Workflow” (Simulink)

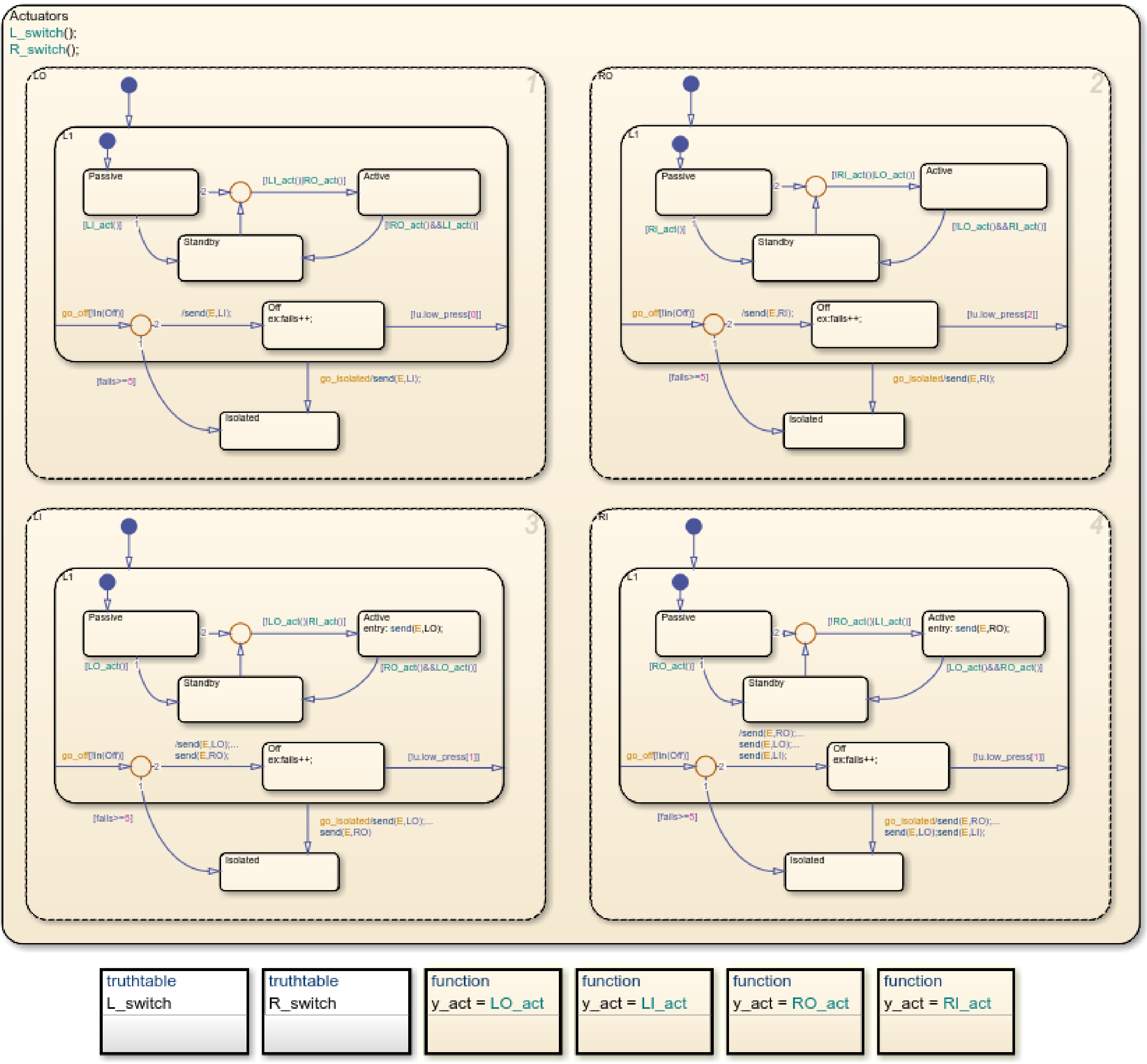
Test Chart with Fault Detection and Redundant Logic

This example shows how to use operating points to test the response of an aircraft elevator system to an actuator failure. An operating point is a snapshot of the state of a Simulink model during simulation. If your model contains a Stateflow chart, the operating point includes information about active states, output and local data, and persistent variables. For more information, see “Save and Restore Operating Points for Stateflow Charts” on page 18-2.

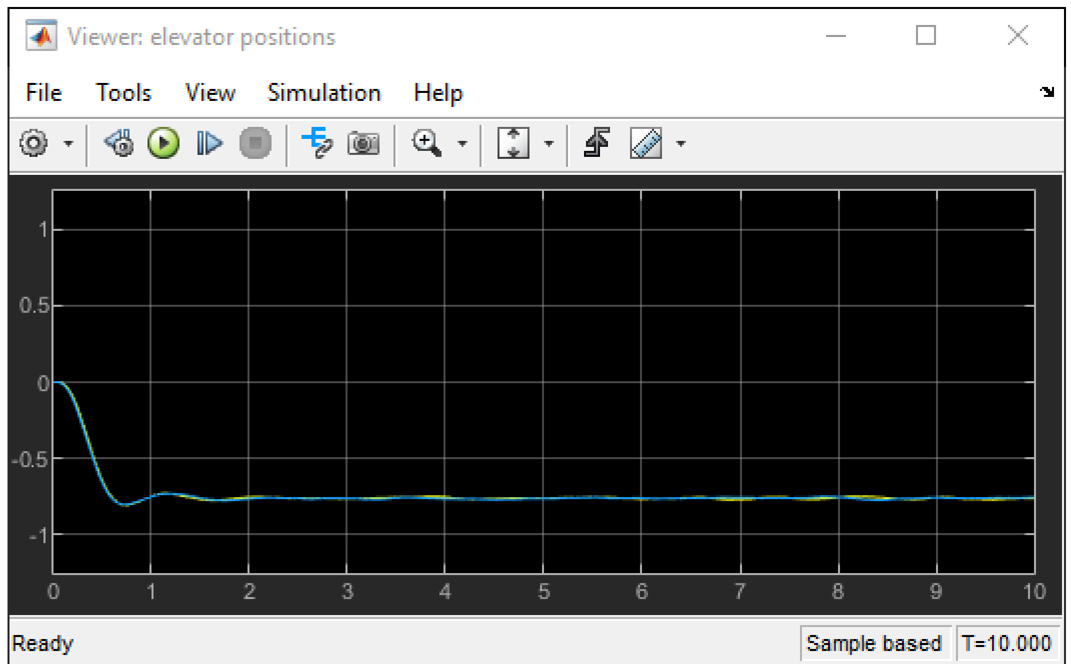
The model `sf_aircraft` demonstrates a fault detection, isolation, and recovery (FDIR) application for a pair of aircraft elevators controlled by redundant actuators.



The `Mode Logic` chart monitors the status of the four actuators. Each elevator has a primary, outer actuator (represented by the states `LO` and `RO`) and a secondary, inner actuator (represented by the states `LI` and `RI`). In normal operation, the outer actuators are active and the inner actuators are on standby.



When the actuators work correctly, the left and right elevators reach steady-state positions in 3 seconds.



To see what happens when one actuator fails, you can simulate the model, save the operating point at $t = 3$, modify the operating point to reflect an actuator failure, and then simulate again between $t = 3$ and $t = 10$.

For more information about this model, see “Detect Faults in Aircraft Elevator Control System” on page 27-19.

Define Operating Point for Steady State

1. Open the `sf_aircraft` model.

```
openExample("stateflow/FaultDetectionControlLogicInAnAircraftControlSystemExample")
```

2. Set the model to save the final operating point. Open the Configuration Parameters dialog box and, in the **Data Import/Export** pane:

   a. Select **Final states** and enter a name for the operating point. For this example, use `xSteadyState`.

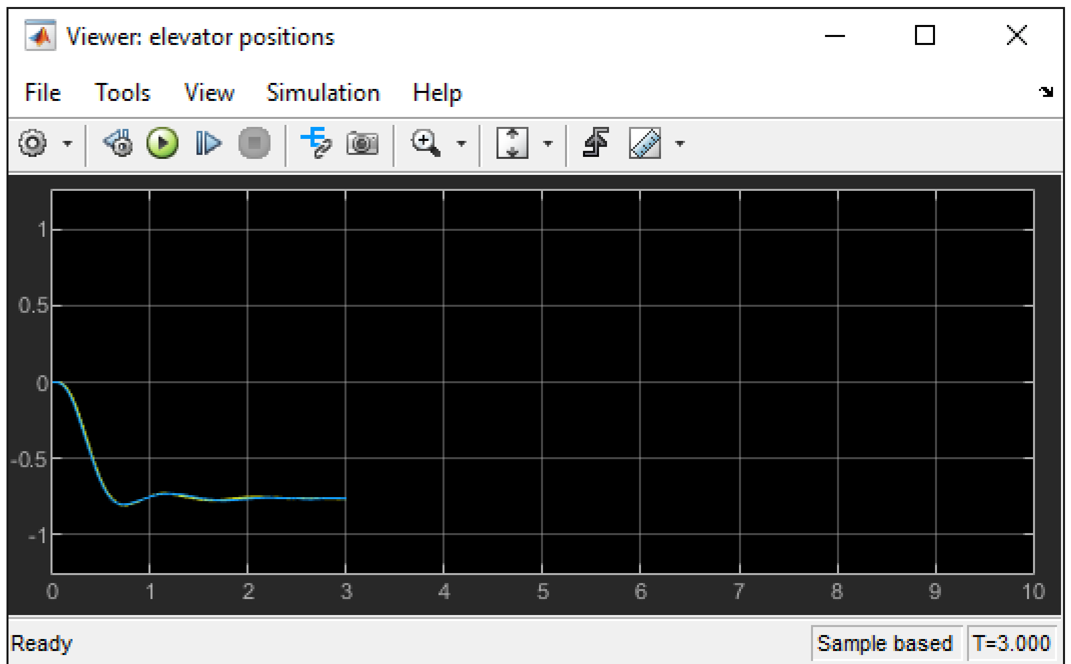
   b. Select **Save final operating point**.

   c. Click **OK**.

3. Set the stop time for this simulation segment. In the **Simulation** tab, set **Stop Time** to 3.

4. Run the simulation.

   When you simulate the model, you save the final operating point at $t = 3$ in the variable `xSteadyState` in the MATLAB base workspace.



5 In the Configuration Parameters dialog box, in the **Data Import/Export** pane, clear the **Save final operating point** and **Final states** parameters. This step prevents you from overwriting the operating point you saved in the previous step.

Modify Operating Point Values for Actuator Failure

1 Access the `Stateflow.op.BlockOperatingPoint` object that contains the operating point information for the `Mode Logic` chart.

```
blockpath = "sf_aircraft/Mode Logic";
op = get(xSteadyState,blockpath)
```

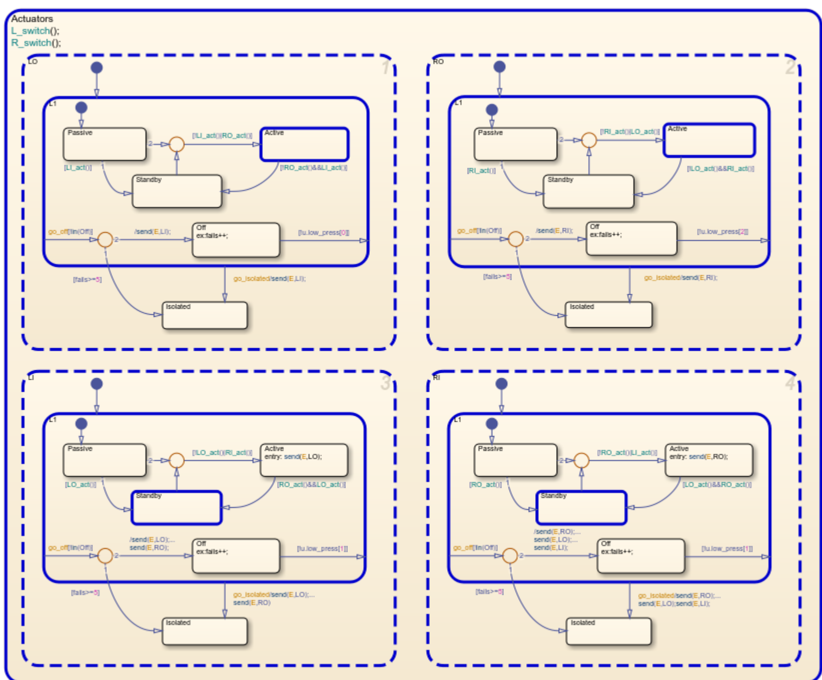
```
op = 

  Block:    "Mode Logic"    (handle)    (active)
  Path:     sf_aircraft/Mode Logic

  Contains:

    + Actuators          "State (OR)"                    (active)
    + LI_act             "Function"
    + LO_act             "Function"
    + L_switch           "Function"
    + RI_act             "Function"
    + RO_act             "Function"
    + R_switch           "Function"
      LI_mode            "State output data"        sf_aircraft_ModeType [1,1]
      LO_mode            "State output data"        sf_aircraft_ModeType [1,1]
      RI_mode            "State output data"        sf_aircraft_ModeType [1,1]
      RO_mode            "State output data"        sf_aircraft_ModeType [1,1]
```

The operating point contains a list of states, functions, and data in hierarchical order.

2 Highlight the states that are active in your chart at $t = 3$.

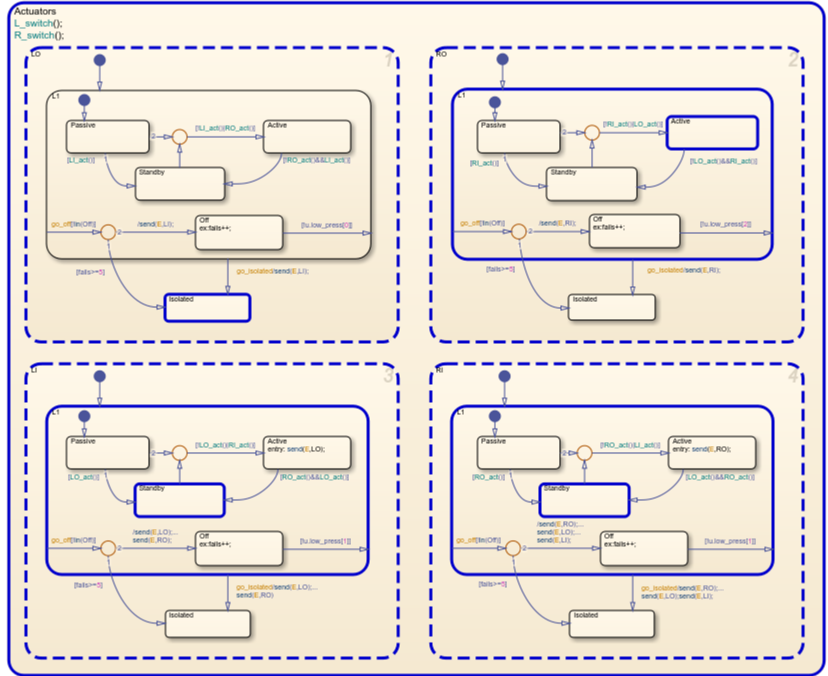
```
highlightActiveStates(op)
```

Active states appear highlighted. The two outer actuators are active and the two inner actuators are on standby.



3 Change the substate activity in the state LO to reflect a failure of the left-outer actuator.

```
setActive(op.Actuators.LO.Isolated)
```



Note The `setActive` function ensures that the chart exits and enters the appropriate states to maintain state consistency. However, the method does not perform `exit` actions for the previously active substate or `entry` actions for the newly active substate.

---

4. Save the modified operating point as the workspace variable `xFailure`.

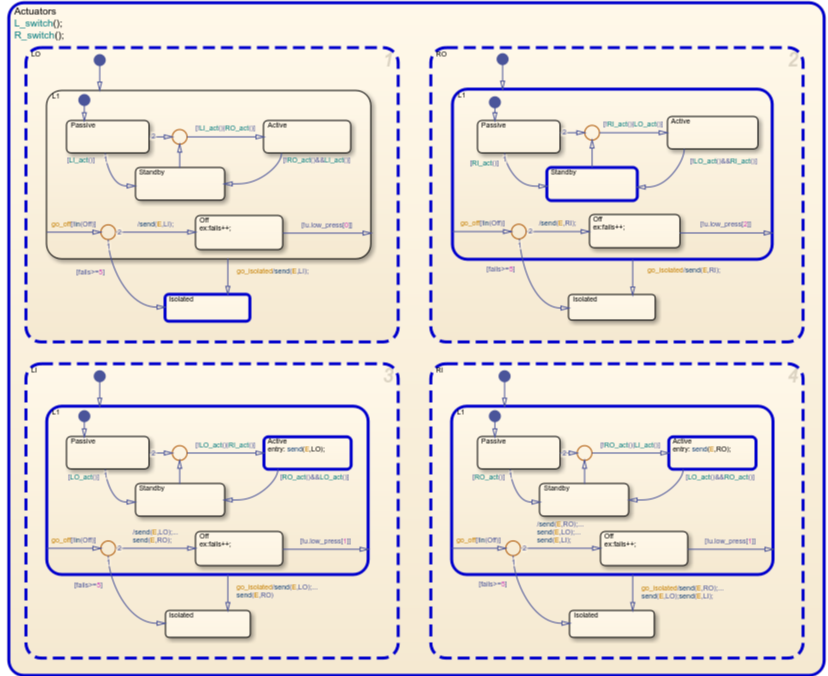
```
xFailure = set(xSteadyState,blockpath,op);
```

Test Model Behavior after Actuator Failure

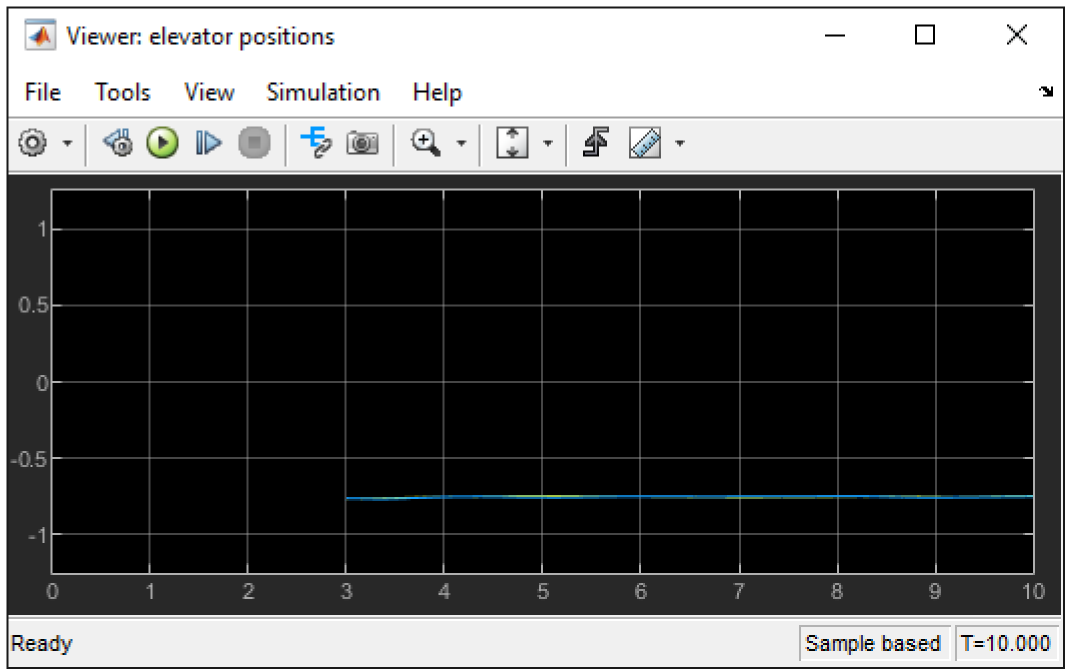
1. Load the operating point as the initial state of the model. In the Configuration Parameters dialog box, in the **Data Import/Export** pane, select **Initial state** and enter the variable that contains the modified operating point of your chart, `xFailure`. Then, click **OK**.
2. Define the stop time for the simulation segment to test. In the **Simulation** tab, set **Stop Time** to `10`.
3. Run the simulation.

   The chart animation shows how the other three actuators react to the failure of the left-outer actuator:

   - The left-inner actuator switches from standby to active to compensate for the left-outer actuator failure.
   - The right-inner actuator switches from standby to active because the same hydraulic line connects to both inner actuators.
   - The right-outer actuator switches from active to standby because only one actuator can be active for each elevator.



After the failure, both elevators continue to maintain steady-state positions.



See Also

Model Settings
Initial state | Final states | Save final operating point

Objects

`Stateflow.op.BlockOperatingPoint` | `Stateflow.op.OperatingPointContainer` | `Stateflow.op.OperatingPointData`

Functions

`highlightActiveStates` | `setActive` | `get` | `set`

More About

- "Save and Restore Operating Points for Stateflow Charts" on page 18-2
- "Detect Faults in Aircraft Elevator Control System" on page 27-19
- "Test Difficult-to-Reproduce Chart Configurations" on page 18-12
- "Use Model Operating Point for Faster Simulation Workflow" (Simulink)

Vectors and Matrices in Stateflow Charts

- “Vectors and Matrices in Stateflow Charts” on page 19-2
- “Operations for Vectors and Matrices in Stateflow” on page 19-4
- “Declare Variable-Size Data in Stateflow Charts” on page 19-9
- “Compute Output Based on Size of Input Signal” on page 19-12

Vectors and Matrices in Stateflow Charts

Vectors and matrices combine scalar data into a single, multidimensional data object. You can modify individual elements or perform arithmetic on entire vectors and matrices. For more information, see "Operations for Vectors and Matrices in Stateflow" on page 19-4.

Define Vector and Matrix Data

1. Add a data object to your chart, as described in "Add Stateflow Data" on page 10-2.
2. Set the **Size** property for the data object as the dimensions of the vector or matrix. See "Specify Size of Stateflow Data" on page 10-26. For example:
   - To specify a 4-by-1 column vector, enter `4`.
   - To specify a 1-by-4 row vector, enter `[1 4]`.
   - To specify a 3-by-3 matrix, enter `[3 3]`.
3. Set the **Initial value** property for the data object. See "Initial value" on page 10-8.
   - To specify a value of zero for all elements of the vector or matrix, leave the **Initial value** empty. If you do not specify an initial value, all elements are initialized to 0.
   - To specify the same value for all elements of the vector or matrix, enter a scalar value. All elements are initialized to the scalar value you specify.
   - To specify a different value for each element of the vector or matrix, enter an array of real values. For example:
     - To initialize a 4-by-1 column vector, you can enter `[1; 2; 3; 4]`.
     - To initialize a 1-by-4 row vector, you can enter `[1 2 3 4]`.
     - To initialize a 3-by-3 matrix, you can enter `[1 2 3; 4 5 6; 7 8 9]`.
4. Set the name, scope, base type, and other properties for the data object, as described in "Set Data Properties" on page 10-5.

You can specify the size and initial value of a vector or matrix by using an expression. Expressions can contain a mix of numeric values, constants, parameters, variables, arithmetic operations, and calls to MATLAB functions. For more information, see "Specify Data Properties by Using MATLAB Expressions" on page 10-18.

Where You Can Use Vectors and Matrices

You can define vectors and matrices at these levels of the Stateflow hierarchy:

- Charts
- Subcharts
- States
- Functions

You can use vectors and matrices to define:

- Input data
- Output data

- Local data
- Function inputs
- Function outputs

You can also use vectors and matrices as arguments for:

- State actions
- Transition actions
- MATLAB functions
- Truth table functions
- Graphical functions
- Simulink functions
- Change detection operators

Rules for Vectors and Matrices in Stateflow Charts

Use Operands of Equal Dimensions for Element-Wise Operations

If you perform element-wise operations on vectors or matrices with unequal dimensions, the chart generates a size mismatch error when you simulate the model. For more information, see "Operations for Vectors and Matrices in Stateflow" on page 19-4.

Do Not Define Vectors and Matrices with the ml Base Type

The `ml` base type supports only scalar data. If you define a vector or matrix with the `ml` base type, the chart generates an error when you simulate the model. For more information, see "ml Data Type" on page 14-23.

Do Not Use Complex Numbers to Set the Initial Values of Vectors and Matrices

If you initialize an element of a vector or matrix by using a complex number, the chart generates an error when you simulate the model. You can set the values of vectors and matrices to complex numbers after initialization. For more information, see "Complex Data in Stateflow Charts" on page 24-2.

Do Not Use Vectors and Matrices in Temporal Logic Operators

Because time is a scalar quantity, you cannot use a vector or matrix as an argument for a temporal logic operator. For more information, see "Control Chart Execution by Using Temporal Logic" on page 14-35.

See Also

More About

- "Operations for Vectors and Matrices in Stateflow" on page 19-4
- "Specify Size of Stateflow Data" on page 10-26
- "Reuse MATLAB Code by Defining MATLAB Functions" on page 7-2
- "Declare Variable-Size Data in Stateflow Charts" on page 19-9

Operations for Vectors and Matrices in Stateflow

Stateflow charts in Simulink models have an action language property that defines the syntax that you use to compute with vectors and matrices. The action language properties are:

- MATLAB as the action language.
- C as the action language.

For more information, see “Differences Between MATLAB and C as Action Language Syntax” on page 15-4.

Indexing Notation

In charts that use MATLAB as the action language, refer to elements of a vector or matrix by using one-based indexing delimited by parentheses. Separate indices for different dimensions with commas.

In charts that use C as the action language, refer to elements of a vector or matrix by using zero-based indexing delimited by brackets. Enclose indices for different dimensions in their own pair of brackets.

Example	MATLAB as the Action Language	C as the Action Language
The first element of a vector `V`	`V(1)`	`V[0]`
The `i`th element of a vector `V`	`V(i)`	`V[i-1]`
The element in row `4` and column `5` of a matrix `M`	`M(4,5)`	`M[3][4]`
The element in row `i` and column `j` of a matrix `M`	`M(i,j)`	`M[i-1][j-1]`

Binary Operations

This table summarizes the interpretation of all binary operations on vector and matrix operands according to their order of precedence (1 = highest, 3 = lowest). Binary operations are left associative so that, in any expression, operators with the same precedence are evaluated from left to right. Except for the matrix multiplication and division operators in charts that use MATLAB as the action language, all binary operators perform element-wise operations.

Operation	Precedence	MATLAB as the Action Language	C as the Action Language
`a * b`	1	Matrix multiplication.	Element-wise multiplication. For matrix multiplication, use the * operation in a MATLAB function. See “Perform Matrix Arithmetic by Using MATLAB Functions” on page 19-7.
`a .* b`	1	Element-wise multiplication.	Not supported. Use the operation `a * b`.

Operation	Precedence	MATLAB as the Action Language	C as the Action Language
a / b	1	Matrix right division.	Element-wise right division. For matrix right division, use the / operation in a MATLAB function. See “Perform Matrix Arithmetic by Using MATLAB Functions” on page 19-7.
a ./ b	1	Element-wise right division.	Not supported. Use the operation a / b.
a \ b	1	Matrix left division.	Not supported. Use the \ operation in a MATLAB function. See “Perform Matrix Arithmetic by Using MATLAB Functions” on page 19-7.
a .\ b	1	Element-wise left division.	Not supported. Use the .\ operation in a MATLAB function. See “Perform Matrix Arithmetic by Using MATLAB Functions” on page 19-7.
a + b	2	Addition.	Addition.
a - b	2	Subtraction.	Subtraction.
a == b	3	Comparison, equal to.	Comparison, equal to.
a ~= b	3	Comparison, not equal to.	Comparison, not equal to.
a != b	3	Not supported. Use the operation a ~= b.	Comparison, not equal to.
a <> b	3	Not supported. Use the operation a ~= b.	Comparison, not equal to.

Unary Operations and Actions

This table summarizes the interpretation of all unary operations and actions on vector and matrix operands. Unary operations:

- Have higher precedence than the binary operators.
- Are right associative so that, in any expression, they are evaluated from right to left.
- Perform element-wise operations.

Example	MATLAB as the Action Language	C as the Action Language
~a	Logical NOT. For bitwise NOT, use the bitcmp function.	• Bitwise NOT (default). Enable this operation by selecting the **Enable C-bit operations** chart property. • Logical NOT. Enable this operation by clearing the **Enable C-bit operations** chart property. For more information, see “Bitwise Operations” on page 14-8 and “Enable C-bit operations” on page 1-21.

Example	MATLAB as the Action Language	C as the Action Language
`!a`	Not supported. Use the operation `~a`.	Logical NOT.
`-a`	Negative.	Negative.
`a++`	Not supported.	Increment all elements of the vector or matrix. Equivalent to `a = a+1`.
`a--`	Not supported.	Decrement all elements of the vector or matrix. Equivalent to `a = a-1`.

Assignment Operations

This table summarizes the interpretation of assignment operations on vector and matrix operands.

Operation	MATLAB as the Action Language	C as the Action Language
`a = b`	Simple assignment.	Simple assignment.
`a += b`	Not supported. Use the expression `a = a+b`.	Equivalent to `a = a+b`.
`a -= b`	Not supported. Use the expression `a = a-b`.	Equivalent to `a = a-b`.
`a *= b`	Not supported. Use the expression `a = a*b`.	Equivalent to `a = a*b`.
`a /= b`	Not supported. Use the expression `a = a/b`.	Equivalent to `a = a/b`.

Assign Values to Individual Elements of a Matrix

You can assign a value to an individual entry of a vector or matrix by using the indexing syntax appropriate to the action language of the chart.

Example	MATLAB as the Action Language	C as the Action Language
Assign the value `10` to the first element of the vector `V`.	`V(1) = 10;`	`V[0] = 10;`
Assign the value 77 to the element in row 2 and column 9 of the matrix `M`.	`M(2,9) = 77;`	`M[1][8] = 77;`

Assign Values to All Elements of a Matrix

In charts that use MATLAB as the action language, you can use a single action to specify all of the elements of a vector or matrix. For example, this action assigns each element of the 2-by-3 matrix `A` to a different value:

```
A = [1 2 3; 4 5 6];
```

In charts that use C as the action language, you can use scalar expansion to set all of the elements of a vector or matrix to the same value. Scalar expansion converts scalar data to match the dimensions of vector or matrix data. For example, this action sets all of the elements of the matrix `A` to `10`:

```
A = 10;
```

Scalar expansion applies to all graphical, truth table, MATLAB, and Simulink functions. Suppose that you define the formal arguments of a function `f` as scalars. This table describes the rules of scalar expansion for the function call `y = f(u)`.

Output y	Input u	Result
Scalar	Scalar	No scalar expansion occurs.
Scalar	Vector or matrix	The chart generates a size mismatch error.
Vector or matrix	Scalar	The chart uses scalar expansion to assign the scalar output value of `f(u)` to every element of `y`: `y[i][j] = f(u)`
Vector or matrix	Vector or matrix	The chart uses scalar expansion to compute an output value for each element of `u` and assign it to the corresponding element of `y`: `y[i][j] = f(u[i][j])` If `y` and `u` do not have the same size, the chart generates a size mismatch error.

For functions with multiple outputs, the same rules apply unless the outputs and inputs are all vectors or matrices. In this case, the chart generates a size mismatch error and scalar expansion does not occur.

Charts that use MATLAB as the action language do not support scalar expansion.

Perform Matrix Arithmetic by Using MATLAB Functions

In charts that use C as the action language, the operations `*` and `/` perform element-wise multiplication and division. To perform standard matrix multiplication and division in a C chart, use a MATLAB function.

Suppose that you want to perform these operations on the square matrices `u1` and `u2`:

- Compute the standard matrix product `y1 = u1 * u2`.
- Solve the equation `u1 * y2 = u2`.
- Solve the equation `y3 * u1 = u2`.

To complete these calculations in a C chart, add a MATLAB function that runs this code:

```
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
```

Before calling the function, specify the properties for the input and output data, as described in "Set Data Properties" on page 10-5.

In charts that use MATLAB as the action language, the operations `*`, `/`, and `\` perform standard matrix multiplication and division. You can use these operations directly in state and transition actions.

See Also

More About

- "Vectors and Matrices in Stateflow Charts" on page 19-2
- "Specify Size of Stateflow Data" on page 10-26

Declare Variable-Size Data in Stateflow Charts

The size of most data in a Stateflow chart in a Simulink model is fixed at compile time and does not change at run time. In contrast, variable-size data can change size during simulation. For example, you can use variable-size data if the output from a Stateflow chart is an array whose size depends on the state of the chart.

Enable Support for Variable-Size Data

By default, Stateflow charts support variable-size data. To enable or disable this support for individual charts, modify the chart property **Support variable-size arrays**, as described in “Specify Properties for Stateflow Charts” on page 1-19. For more information, see “Support variable-size arrays” on page 1-22.

When you enable this chart property, your chart accepts variable-size input data from other blocks in the Simulink model. You can declare additional variable-size data objects by enabling the **Variable size** data property, as described in “Set Data Properties” on page 10-5. For more information, see “Variable size” on page 10-7.

Use the **Size** data property to explicitly specify the maximum size for each variable-size data object. For example, to specify a 2-D matrix where the maximum size is 2 for the first dimension and 4 for the second dimension, enter `[2 4]`.

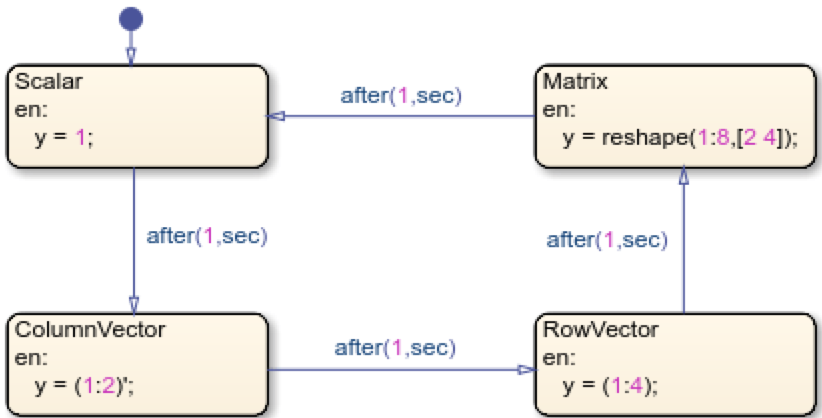
Alternatively, to inherit the maximum size of data, set **Size** to `-1`. For more information, see “Specify Size of Stateflow Data” on page 10-26.

Note You cannot enable variable-size data for scalar data.

Variable-Size Data in Charts That Use MATLAB as the Action Language

Charts that use MATLAB as the action language support variable-size chart-level input, local, and output data, as well as variable-size input and output data in graphical functions, MATLAB functions, and truth table functions.

You can modify variable-size chart data in state and transition actions. For example, in this chart that uses MATLAB as the action language, the state actions assign the variable-size chart output, `y`, to values that range in size from a scalar to a 2-by-4 matrix.



For more information about this example, see "Compute Output Based on Size of Input Signal" on page 19-12.

Variable-Size Data in Charts That Use C as the Action Language

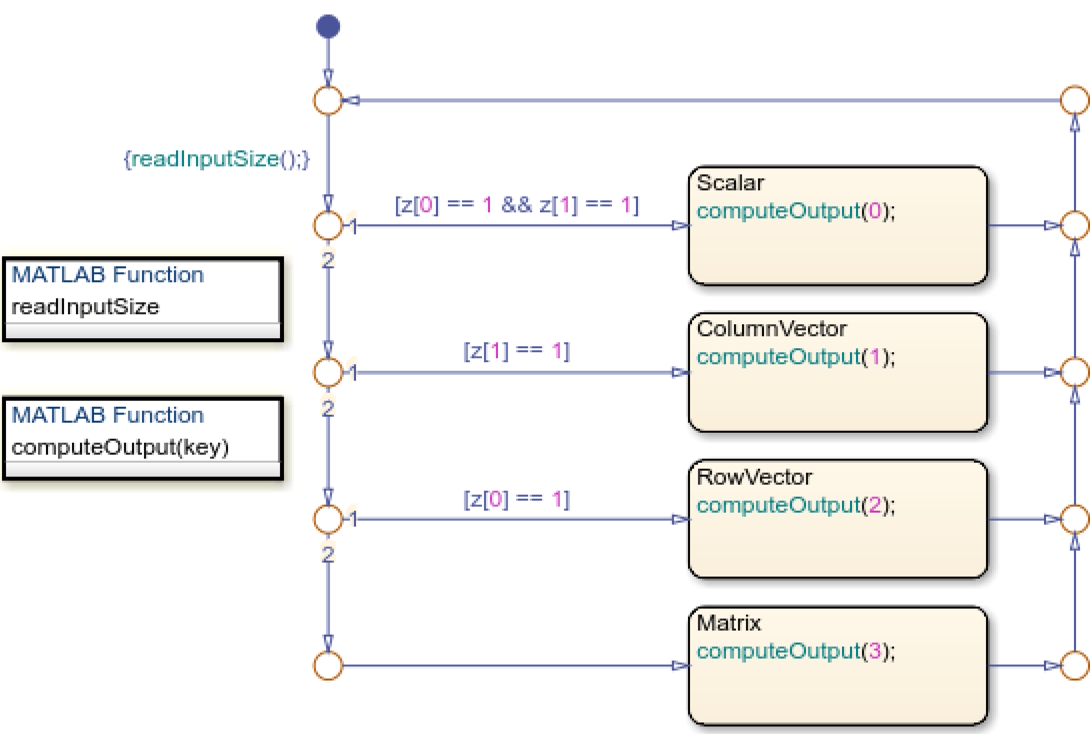
Charts that use C as the action language support variable-size chart-level input and output data, as well as variable-size input, local, temporary, and output data in MATLAB functions and truth table functions that use MATLAB as the action language.

You can only modify variable-size chart data by using:

- MATLAB functions
- Simulink functions
- Truth tables that use MATLAB as the action language

All computations with variable-size data must occur inside these functions, and not directly in states or transitions. For example, you can pass variable-size chart data to these functions from state and transition actions. Alternatively, MATLAB functions can access the variable-size chart data directly.

For example, in this chart that uses C as the action language, the MATLAB function `readInputSize` determines the size of the variable-size chart input. Then, the MATLAB function `computeOutput` applies a size-dependent algorithm to the chart input and sets the value of the variable-size chart output.



For more information about this example, see “Compute Output Based on Size of Input Signal” on page 19-12.

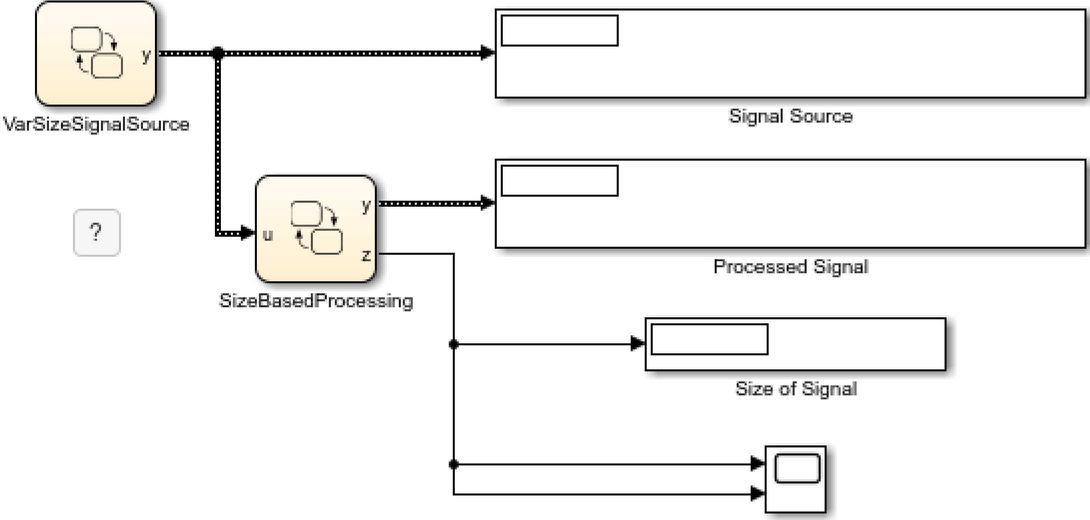
See Also

More About

- “Compute Output Based on Size of Input Signal” on page 19-12
- “Reuse MATLAB Code by Defining MATLAB Functions” on page 7-2
- “Reuse Simulink Functions in Stateflow Charts” on page 9-2
- “Use Truth Tables to Model Combinatorial Logic” on page 8-2

Compute Output Based on Size of Input Signal

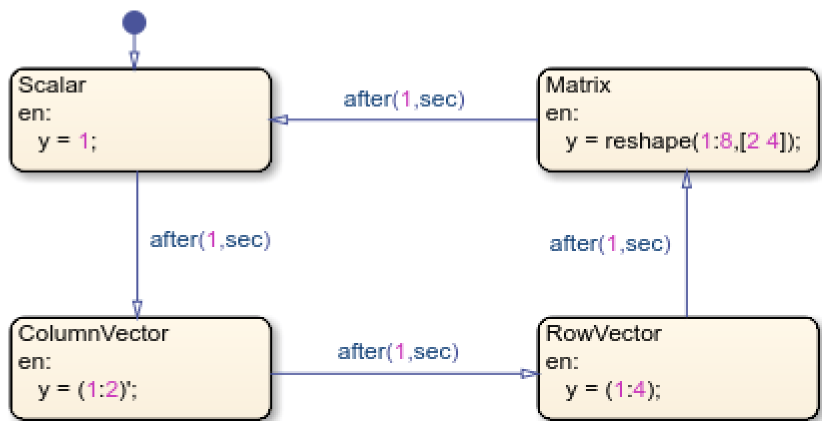
This example shows how to define variable-size output data in Stateflow® charts. In this model, the chart `VarSizeSignalSource` generates a variable-size signal. The chart `SizeBasedProcessing` analyzes this signal and produces a related variable-size signal. Display blocks show the values and size of the two signals.



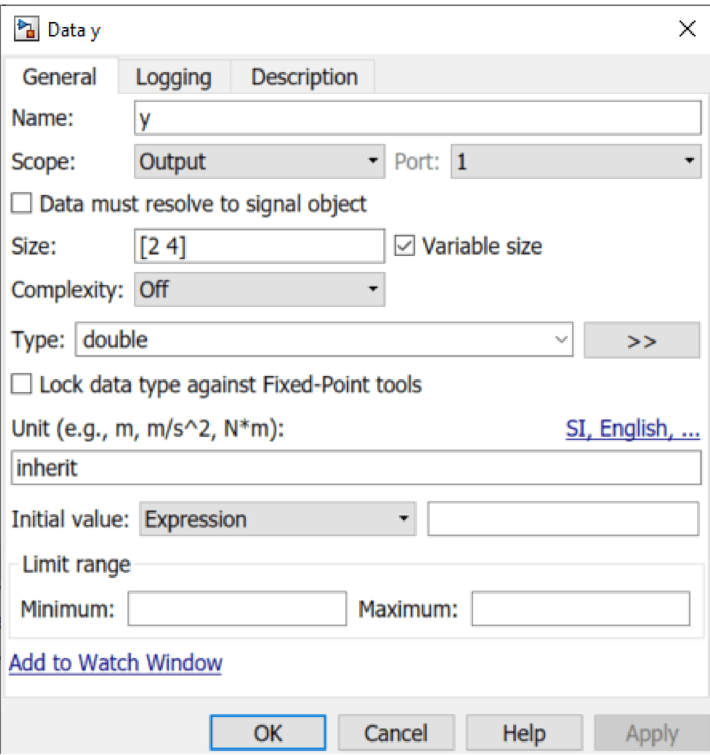
For more information on variable-size data, see "Declare Variable-Size Data in Stateflow Charts" on page 19-9.

Generate Variable-Size Output Data

The Stateflow chart `VarSizeSignalSource` uses MATLAB® as the action language. The temporal logic in this chart triggers the transitions between four states. Each state generates an output value with a different size: a scalar, a two-element column vector, a four-element row vector, and a 2-by-4 matrix.



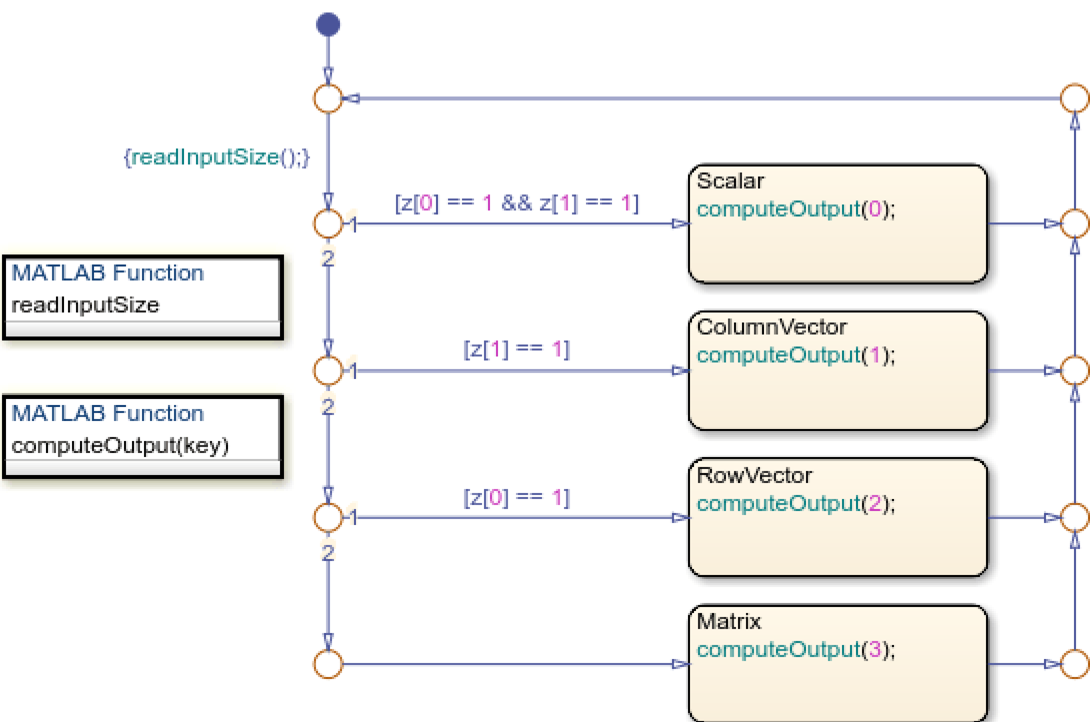
The chart behaves like a source block. It has no input and one variable-size output, `y`. For variable-size outputs, you must explicitly specify the upper bounds for each dimension. In this case, the **Variable size** property for `y` is enabled and its **Size** property is set to `[2 4]`, the maximum size for the signal.



In charts that use MATLAB as the action language, state and transition actions can read and write directly to variable-size data. For example, the entry actions of the states in this chart explicitly compute the value of `y`.

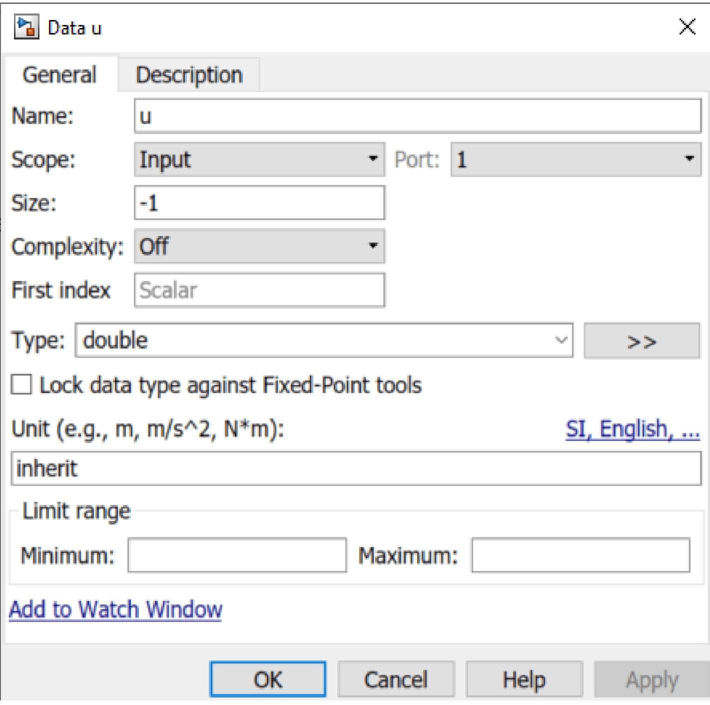
Process Variable-Size Input Data

The Stateflow chart `SizeBasedProcessing` uses C as the action language. These charts can exchange variable-size data with other charts and blocks in the model. However, state and transition actions in C charts cannot read from or write to variable-size data directly. All computations involving variable-size data must occur in MATLAB functions, Simulink® functions, and truth tables that use MATLAB as the action language.

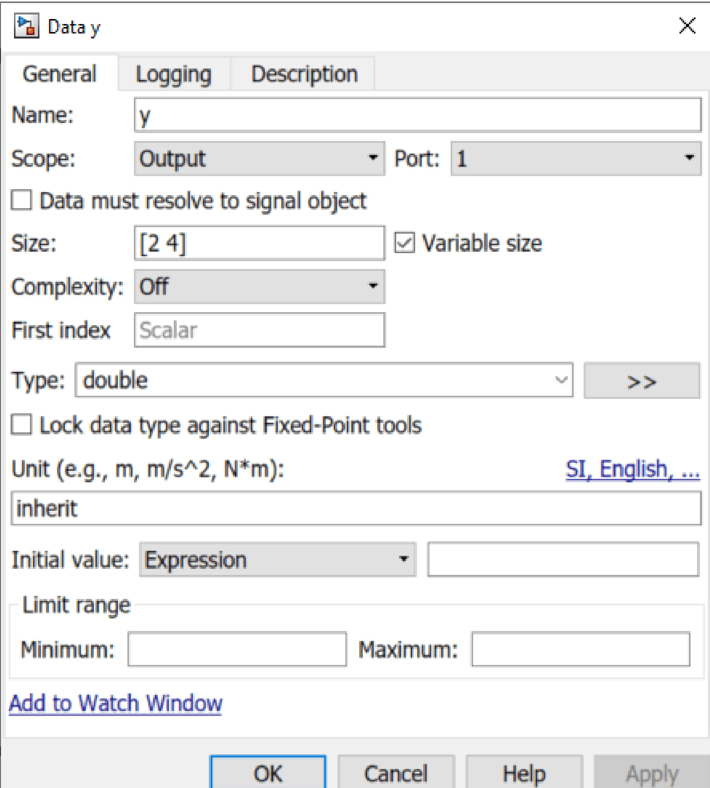


This chart has two variable-size data objects:

- Input `u` is the variable-size signal generated by the `VarSizeSignalSource` chart. Because the chart property **Support variable-size arrays** is enabled, this chart accepts variable-size input data from other blocks in the Simulink model. The **Size** property for `u` is set to `-1` to indicate that the chart inherits the maximum size for this input from the output in chart `VarSizeSignalSource`.



- Output y is a variable-size signal whose size and value depends on whether u is a scalar, a vector, or a matrix. The **Size** property for u is set to `[2 4]`, the maximum size for the signal.



Because this chart uses C as the action language, the names of the variable-size data do not appear in the state actions or transition logic. Instead, the transition logic in the chart calls the MATLAB function readInputSize to determine the size of the input u. Similarly, the actions in the states call the MATLAB function computeOutput to produce values of various size for the variable-size output y. Because MATLAB functions can access chart-level data directly, you do not have to pass the variable-size data as inputs or outputs to these functions.

Determine Size of Input

The MATLAB function isScalarInput determines the size of the chart input u. This signal, which is generated by chart VarSizeSignalSource, can be a scalar, a 2-by-1 column vector, a 1-by-4 row vector, or a 2-by-4 matrix. The function stores the dimensions of u as the chart-level output z.

```
function readInputSize
%#codegen
z = size(u);
end
```

Produce Variable-Size Output

The MATLAB function computeOutput computes the value of the chart output y based on the size and value of the chart input u.

- If u is a scalar, the function assigns to y a scalar value of zero.
- If u is a column vector, the function computes of the sine of each of its elements and stores them in y.
- If u is a row vector, the function computes of the cosine of each of its elements and stores them in y.
- If u is a matrix, the function computes of the square root of each of its elements and stores them in y.

In each case, the value of the output y has the same size as the input u.

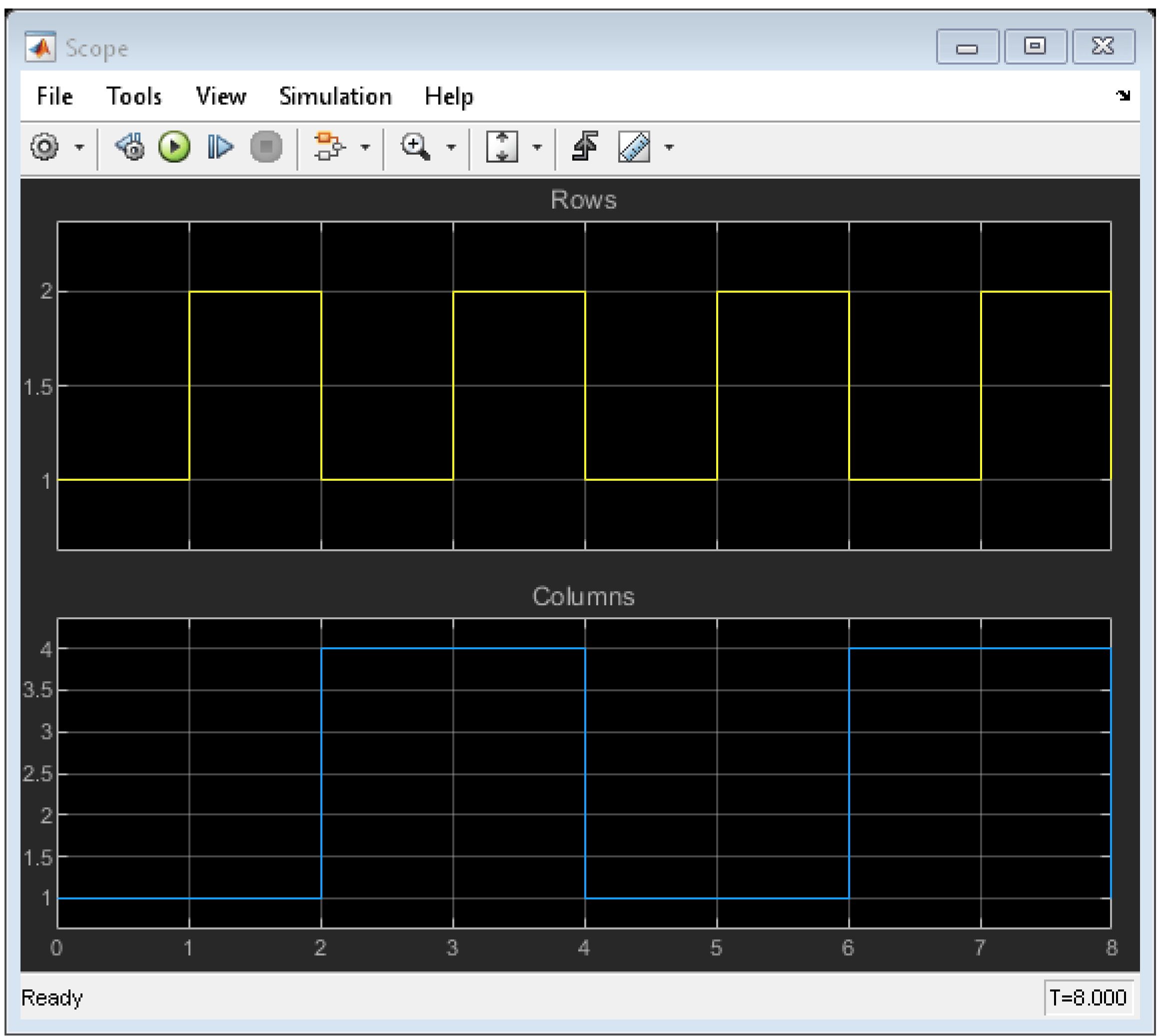
```
function computeOutput(key)
%#codegen
switch key
    case 0 % scalar
        y = 0;
    case 1 % column vector
        y = sin(u);
    case 2 % row vector
        y = cos(u);
    case 3 % matrix
        y = sqrt(u);
end
end
```

Simulate the Model

The tabs located above the Explorer Bar enable you to switch between the Simulink model and the two Stateflow charts. During simulation:

- The chart animations show the active state for each chart cycling between the Scalar, ColumnVector, RowVector, and VectorPartial, and Matrix states.

- In the Simulink model, the display blocks `Signal Source` and `Processed Signal` periodically show between one and eight values for the variable-size signals.
- The display block `Size of Signal` and the Scope block show the number of rows and columns in the variable-size signals.



See Also

`after`

More About

- "Declare Variable-Size Data in Stateflow Charts" on page 19-9
- "Control Chart Execution by Using Temporal Logic" on page 14-35
- "Reuse MATLAB Code by Defining MATLAB Functions" on page 7-2

Enumerated Data in Charts

- “Reference Values by Name by Using Enumerated Data” on page 20-2
- “Define Enumerated Data Types” on page 20-5
- “Best Practices for Using Enumerated Data” on page 20-8
- “Assign Enumerated Values in a Chart” on page 20-11
- “Model Media Player by Using Enumerated Data” on page 20-15

Reference Values by Name by Using Enumerated Data

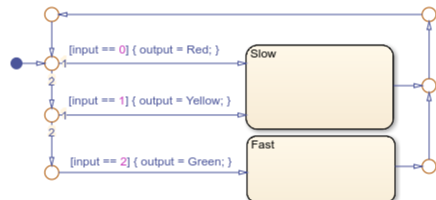
To enhance the readability of a Stateflow chart, use enumerated data. With enumerated data, you can:

- Create a restricted set of values and refer to those values by name.
- Group related values into separate data types.
- Avoid defining a long list of constants.

Enumerated data is supported in Stateflow charts in Simulink models.

Example of Enumerated Data

An enumerated data type is a finite collection of *enumerated values* consisting of a name and an underlying integer value. For example, this chart uses enumerated data to refer to a set of colors.



The enumerated data `output` is restricted to a finite set of values. You can refer to these values by their names: `Red`, `Yellow`, and `Green`.

Enumerated Value	Name	Integer Value
`Red(0)`	`Red`	`0`
`Yellow(1)`	`Yellow`	`1`
`Green(2)`	`Green`	`2`

This MATLAB file defines the enumerated data type `BasicColors` referenced by the chart.

```
classdef BasicColors < Simulink.IntEnumType
  enumeration
    Red(0)
    Yellow(1)
    Green(2)
  end
end
```

Computation with Enumerated Data

An enumerated data type does not function as a numeric type despite the existence of the underlying integer values. You cannot use enumerated values directly in a mathematical computation. You can use enumerated data to control chart behavior based on assignments and comparisons. To assign or compare enumerated data, use the operations listed in this table.

Example	Description
`a = exp`	Assignment of `exp` to `a`. `exp` must evaluate to an enumerated value.
`a == b`	Comparison, equality.
`a != b`	Comparison, inequality.

In a chart that uses C as the action language, you can compare enumerated data with different data types. Before the comparison, the chart casts the enumerated data to their underlying integer values.

Charts that use MATLAB as the action language cannot compare enumerated data with different data types.

Notation for Enumerated Values

To refer to an enumerated value, use prefixed or nonprefixed identifiers.

Prefixed Identifiers

To prevent name conflicts when referring to enumerated values in Stateflow charts, you can use prefixed identifiers of the form *`Type`*.*`Name`*. *`Type`* is an enumerated data type and *`Name`* is an enumerated value name. For example, suppose that you define three data types (`Colors`, `Temp`, and `Code`) that contain the enumerated name `Red`. By using prefixed notation, you can distinguish `Colors.Red` from `Temp.Red` and `Code.Red`.

Nonprefixed Identifiers

To minimize identifier length when referring to unique enumerated values, you can use nonprefixed enumerated value names. For example, suppose that the enumerated name `Red` belongs only to the data type `Colors`. You can then refer to this value with the nonprefixed identifier `Red`.

If your chart uses data types that contain identical enumerated names (such as `Colors.Red` and `Temp.Red`), use prefixed identifiers to prevent name conflicts.

Where to Use Enumerated Data

Use enumerated data at these levels of the Stateflow hierarchy:

- Chart
- Subchart
- State

Use enumerated data as arguments for:

- State actions
- Condition and transition actions
- Vector and matrix indexing
- MATLAB functions
- Graphical functions
- Simulink functions

- Truth Table blocks and truth table functions

If you have Simulink Coder installed, you can use enumerated data for simulation and code generation.

See Also

More About

- “Define Enumerated Data Types” on page 20-5
- “Assign Enumerated Values in a Chart” on page 20-11
- “Best Practices for Using Enumerated Data” on page 20-8
- “Simulink Enumerations” (Simulink)
- “Use Enumerated Data in Simulink Models” (Simulink)

Define Enumerated Data Types

To enhance the readability of a Stateflow chart, use enumerated data. With enumerated data, you can:

- Create a restricted set of values and refer to those values by name.
- Group related values into separate data types.
- Avoid defining a long list of constants.

Enumerated data is supported in Stateflow charts in Simulink models. For more information, see "Reference Values by Name by Using Enumerated Data" on page 20-2.

Before you can add enumerated data to a Stateflow chart, you must define an enumerated data type in a MATLAB class definition file. Create a different file for each enumerated type.

Elements of an Enumerated Data Type Definition

The enumerated data type definition consists of three sections of code.

Section of Code	Required?	Purpose
`classdef`	Yes	Provides the name of the enumerated data type
`enumeration`	Yes	Lists the enumerated values that the data type allows
`methods`	No	Provides methods that customize the data type

Define an Enumerated Data Type

1. Open a new file in which to store the data type definition. From the **Home** tab on the MATLAB toolstrip, select **New** > **Class**.
2. Complete the `classdef` section of the definition.

   ```
   classdef BasicColors < Simulink.IntEnumType
     ...
   end
   ```

   The `classdef` section defines an enumerated data type with the name `BasicColors`. Stateflow derives the data type from the built-in type `Simulink.IntEnumType`. The enumerated data type name must be unique among data type names and workspace variable names.
3. Define enumerated values in an `enumeration` section.

   ```
   classdef BasicColors < Simulink.IntEnumType
     enumeration
       Red(0)
       Yellow(1)
       Green(2)
     end
   end
   ```

   An enumerated type can define any number of values. The `enumeration` section lists the set of enumerated values that this data type allows. Each enumerated value consists of a name and an underlying integer value. Each name must be unique within its type, but can also appear in other

enumerated types. The default value is the first one in the list, unless you specify otherwise in the `methods` section of the definition.

4. (Optional) Customize the data type by using a `methods` section. The section can contain these methods:

   - `getDefaultValue` specifies a default enumerated value other than the first one in the list of allowed values.
   - `getDescription` specifies a description of the data type for code generated by Simulink Coder.
   - `getHeaderFile` specifies custom header file that contains the enumerated type definition in code generated by Simulink Coder.
   - `getDataScope` enables exporting or importing the enumerated type definition to or from a header file in code generated by Simulink Coder.
   - `addClassNameToEnumNames` enhances readability and prevents name conflicts with identifiers in code generated by Simulink Coder.

   For example, this MATLAB file presents a customized definition for the enumerated data type `BasicColors` that:

   - Specifies that the default enumerated value is the last one in the list of allowed values.
   - Includes a short description of the data type for code generated by Simulink Coder.
   - Imports the definition of the data type from a custom header file to prevent Simulink Coder from generating the definition.
   - Adds the name of the data type as a prefix to each enumeration member name in code generated by Simulink Coder.

```
classdef BasicColors < Simulink.IntEnumType
  enumeration
    Red(0)
    Yellow(1)
    Green(2)
  end

  methods (Static = true)
    function retVal = getDefaultValue()
      % GETDEFAULTVALUE Specifies the default enumeration member.
      % Return a valid member of this enumeration class to specify the default.
      % If you do not define this method, Simulink uses the first member.
      retVal = BasicColors.Green;
    end

    function retVal = getDescription()
      % GETDESCRIPTION Specifies a string to describe this enumerated type.
      retVal = 'This defines an enumerated type for colors';
    end

    function retVal = getHeaderFile()
      % GETHEADERFILE Specifies the file that defines this type in generated code.
      % The method getDataScope determines the significance of the specified file.
      retVal = 'imported_enum_type.h';
    end

    function retVal = getDataScope()
```

```
      % GETDATASCOPE Specifies whether generated code imports or exports this type.
      % Return one of these strings:
      % 'Auto':     define type in model_types.h, or import if header file specified
      % 'Exported': define type in a generated header file
      % 'Imported': import type definition from specified header file
      % If you do not define this method, DataScope is 'Auto' by default.
      retVal = 'Imported';
    end

    function retVal = addClassNameToEnumNames()
      % ADDCLASSNAMETOENUMNAMES Specifies whether to add the class name
      % as a prefix to enumeration member names in generated code.
      % Return true or false.
      % If you do not define this method, no prefix is added.
      retVal = true;
    end % function
  end % methods
end % classdef
```

5. Save the file on the MATLAB path. The name of the file must match exactly the name of the data type. For example, the definition for the data type `BasicColors` must reside in a file named `BasicColors.m`.

Tip To add a folder to the MATLAB search path, type `addpath` *`pathname`* at the command prompt.

Specify Data Type in the Property Inspector

When you add enumerated data to your chart, specify its type in the **Property Inspector**.

1. In the **Type** field, select `Enum: <class name>`.
2. Replace `<class name>` with the name of the data type. For example, you can enter `Enum: BasicColors` in the **Type** field.
3. (Optional) Enter an initial value for the enumerated data by using a prefixed identifier. The initial value must evaluate to a valid MATLAB expression. For more information on prefixed and nonprefixed identifiers, see “Notation for Enumerated Values” on page 20-3.

See Also

More About

- “Assign Enumerated Values in a Chart” on page 20-11
- “Best Practices for Using Enumerated Data” on page 20-8
- “Use Enumerated Data in Simulink Models” (Simulink)
- “Customize Simulink Enumeration” (Simulink)

Best Practices for Using Enumerated Data

To enhance the readability of a Stateflow chart, use enumerated data. With enumerated data, you can:

- Create a restricted set of values and refer to those values by name.
- Group related values into separate data types.
- Avoid defining a long list of constants.

Enumerated data is supported in Stateflow charts in Simulink models. For more information, see “Reference Values by Name by Using Enumerated Data” on page 20-2.

Guidelines for Defining Enumerated Data Types

Use Unique Name for Each Enumerated Type

To avoid name conflicts, the name of an enumerated data type cannot match the name of:

- Another data type
- A data object in the Stateflow chart
- A variable in the MATLAB base workspace

Use Same Name for Enumerated Type and Class Definition File

To enable resolution of enumerated data types for Simulink models, the name of the MATLAB file that contains the type definition must match the name of the data type.

Apply Changes in Enumerated Type Definition

When you update an enumerated data type definition for an open model, the changes do not take effect immediately. To see the effects of updating a data type definition:

1. Save and close the model.
2. Delete all instances of the data type from the MATLAB base workspace. To find these instances, type `whos` at the command prompt.
3. Open the model and start simulation or generate code by using Simulink Coder.

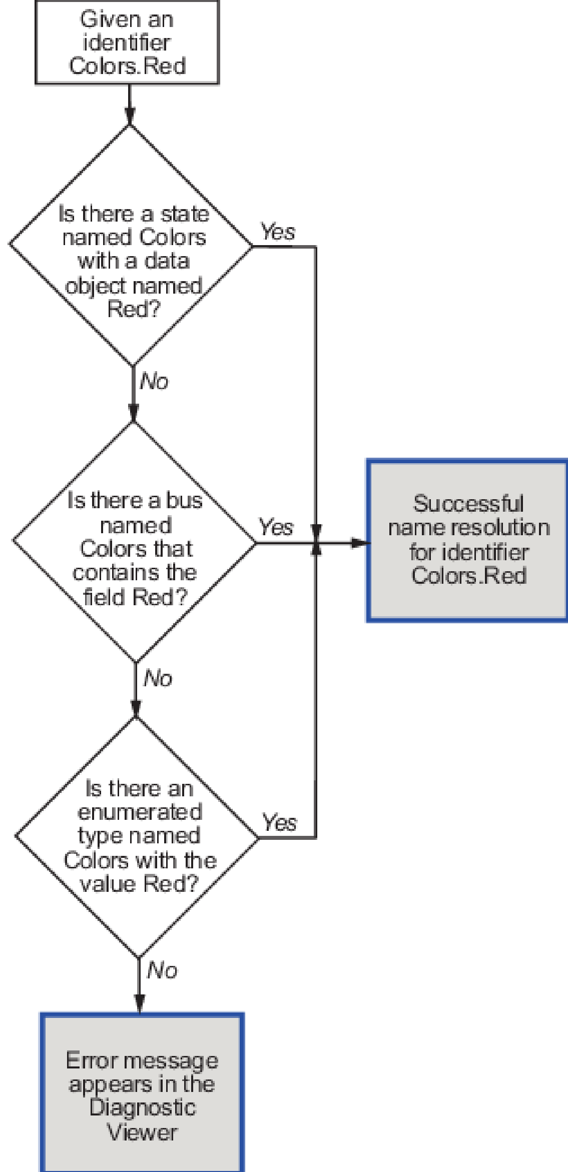
Guidelines for Referencing Enumerated Data

Ensure Unique Name Resolution for Nonprefixed Identifiers

If you use nonprefixed identifiers to refer to enumerated values in a chart, ensure that each enumerated name belongs to a unique enumerated data type.

Use Unique Identifiers for Enumerated Values

If an enumerated value uses the same identifier as a data object or a bus field, the chart does not resolve the identifier correctly. For example, this diagram shows the stages in which a chart tries to resolve the identifier `Colors.Red`.



Set Initial Values of Enumerated Data by Using Prefixed Identifiers

If you choose to set an initial value for enumerated data, you must use a prefixed identifier in the **Initial value** field of the **Property Inspector**. For example, `BasicColors.Red` is a valid identifier, but `Red` is not. The initial value must evaluate to a valid MATLAB expression.

Enhance Readability of Generated Code by Using Prefixed Identifiers

If you add prefixes to enumerated names in the generated code, you enhance readability and avoid name conflicts with global symbols. For details, see “Use Enumerated Data in Generated Code” (Simulink Coder).

Guidelines and Limitations for Enumerated Data

Do Not Enter Minimum or Maximum Values for Enumerated Data

For enumerated data, leave the **Minimum** and **Maximum** properties empty. The chart ignores any values that you enter for these properties.

Whether these fields appear in the **Property Inspector** depends on which **Type** field option you use to define enumerated data.

Type Field Option	Appearance of the Minimum and Maximum Fields
`Enum: <class name>`	Not available
`<data type expression>` or `Inherit from Simulink`	Available

Do Not Assign Enumerated Values to Constant Data

Because enumerated values are constants, assigning these values to constant data is redundant and unnecessary. If you try to assign enumerated values to constant data, an error appears.

Do Not Use ml Namespace Operator to Access Enumerated Data

The `ml` operator does not support enumerated data.

See Also

More About

- “Reference Values by Name by Using Enumerated Data” on page 20-2
- “Define Enumerated Data Types” on page 20-5
- “Assign Enumerated Values in a Chart” on page 20-11

Assign Enumerated Values in a Chart

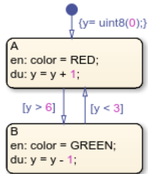
To enhance the readability of a Stateflow chart, use enumerated data. With enumerated data, you can:

- Create a restricted set of values and refer to those values by name.
- Group related values into separate data types.
- Avoid defining a long list of constants.

Enumerated data is supported in Stateflow charts in Simulink models. For more information, see "Reference Values by Name by Using Enumerated Data" on page 20-2.

Chart Behavior

This example shows how to build a chart that uses enumerated values to issue a status keyword.



During simulation, the chart action alternates between states A and B.

Execution of State A

- At the start of the simulation, state A is entered.
- State A executes the `entry` action by assigning the value RED to the enumerated data `color`.
- The data y increments once per time step (every 0.2 seconds) until the condition `[y > 6]` is true.
- The chart takes the transition from state A to state B.

Execution of State B

- After the transition from state A occurs, state B is entered.
- State B executes the `entry` action by assigning the value GREEN to the enumerated data `color`.
- The data y decrements once per time step (every 0.2 seconds) until the condition `[y < 3]` is true.
- The chart takes the transition from state B back to state A.

Build the Chart

Add States and Transitions to the Chart

1. To create a Simulink model with an empty chart, at the MATLAB command prompt, enter `sfnew`.
2. In the empty chart, add states A and B. At the text prompt, enter the appropriate action statements.
3. Add a default transition to state A and transitions between states A and B.
4. Double-click each transition. At the text prompt, enter the appropriate condition.

Define an Enumerated Data Type for the Chart

1. To create a file in which to store the data type definition, from the **Home** tab on the MATLAB toolstrip, select **New** > **Class**.
2. In the MATLAB Editor, enter:

```
classdef TrafficColors < Simulink.IntEnumType
  enumeration
    RED(0)
    GREEN(10)
  end
end
```

   The `classdef` section defines an integer-based enumerated data type named `TrafficColors`. The `enumeration` section contains the enumerated values that this data type allows followed by their underlying numeric value.
3. Save your file as `TrafficColors.m` in a folder on the MATLAB search path.

Define Chart Data

1. To resolve the undefined data, in the **Symbols** pane, click the **Resolve undefined symbols** icon . The Stateflow Editor assigns an appropriate scope to each symbol in the chart.

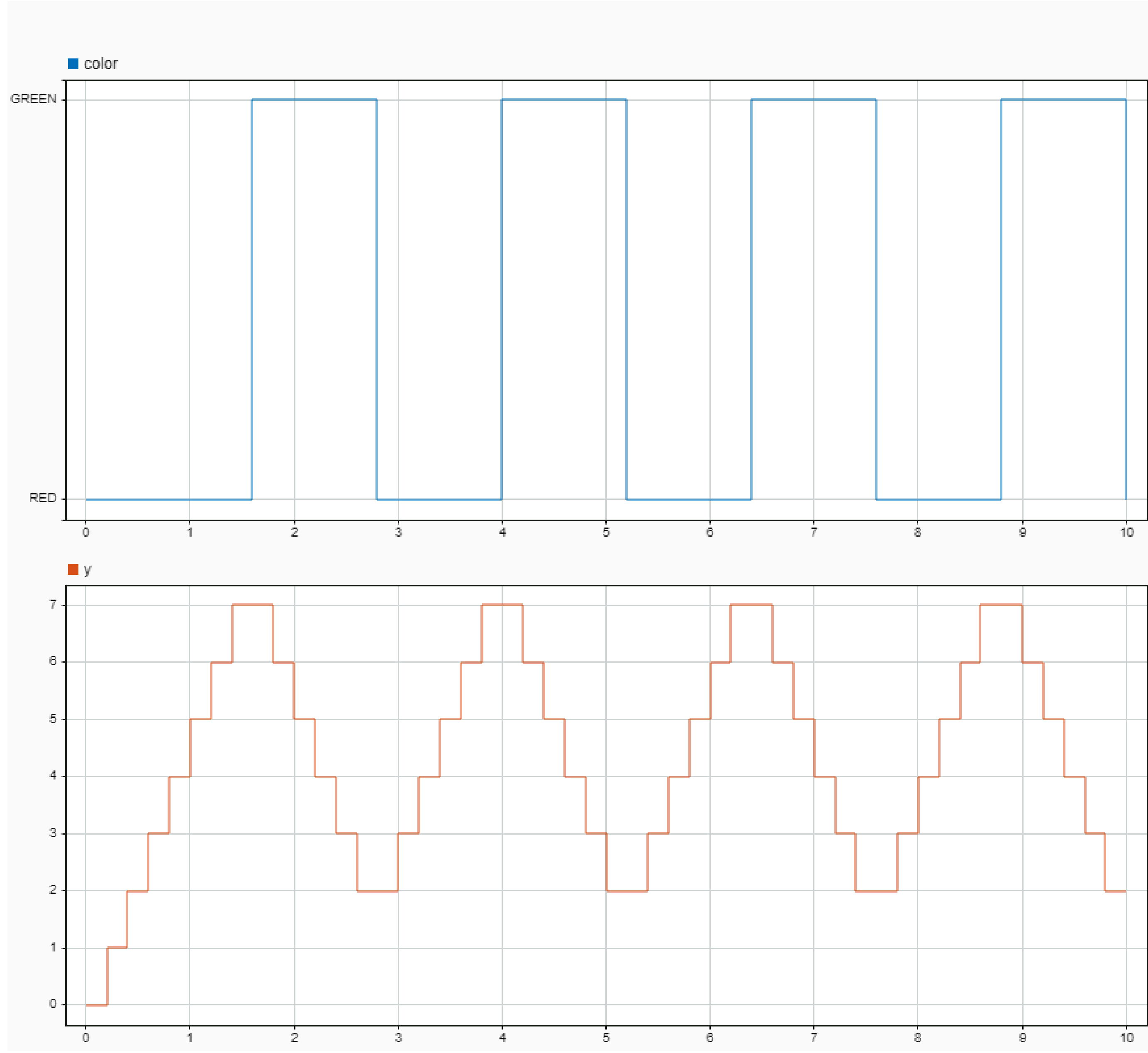
Symbol	Scope
color	Output Data
y	Local Data
GREEN	Parameter Data
RED	Parameter Data

2. To specify `color` as enumerated data, in the **Property Inspector**:
   - In the **Type** field, select `Enum: <class name>`. Replace `<class name>` with `TrafficColors`, the name of the data type that you defined previously.
   - Under **Logging**, select the **Log signal data** check box.
3. To set the scope and type of `y`, in the **Property Inspector**:
   - In the **Scope** field, select `Output`.
   - In the **Type** field, select `uint8`.
   - Under **Logging**, select the **Log signal data** check box.
4. In the **Symbols** pane, delete the symbols `GREEN` and `RED`. The Stateflow Editor incorrectly identified these symbols as parameters before you specified `color` as enumerated data.

View Logged Output

1. Simulate the model.
2. In the **Simulation** tab, under **Review Results**, select **Data Inspector** .

3. In the **Simulation Data Inspector**, in the **Inspect** pane, select the signals `color` and `y`. You can display the logged signals together or in separate axes. For more information, see "Inspect Simulation Data" (Simulink).



4. To access the logged data in the MATLAB workspace, call the signal logging object `logsout`. For example, at the command prompt, enter:

```
losgout = out.logsout;
colorLog = logsout.getElement("color");
Tbl = table(colorLog.Values.Time,colorLog.Values.Data);
Tbl.Properties.VariableNames = ["SimulationTime","Color"]
```

```
Tbl =

  9×2 table

    SimulationTime    Color
    ______________    _____

           0          RED  
         1.6          GREEN
         2.8          RED  
           4          GREEN
         5.2          RED  
         6.4          GREEN
         7.6          RED  
         8.8          GREEN
          10          RED  
```

See Also

Simulation Data Inspector

More About

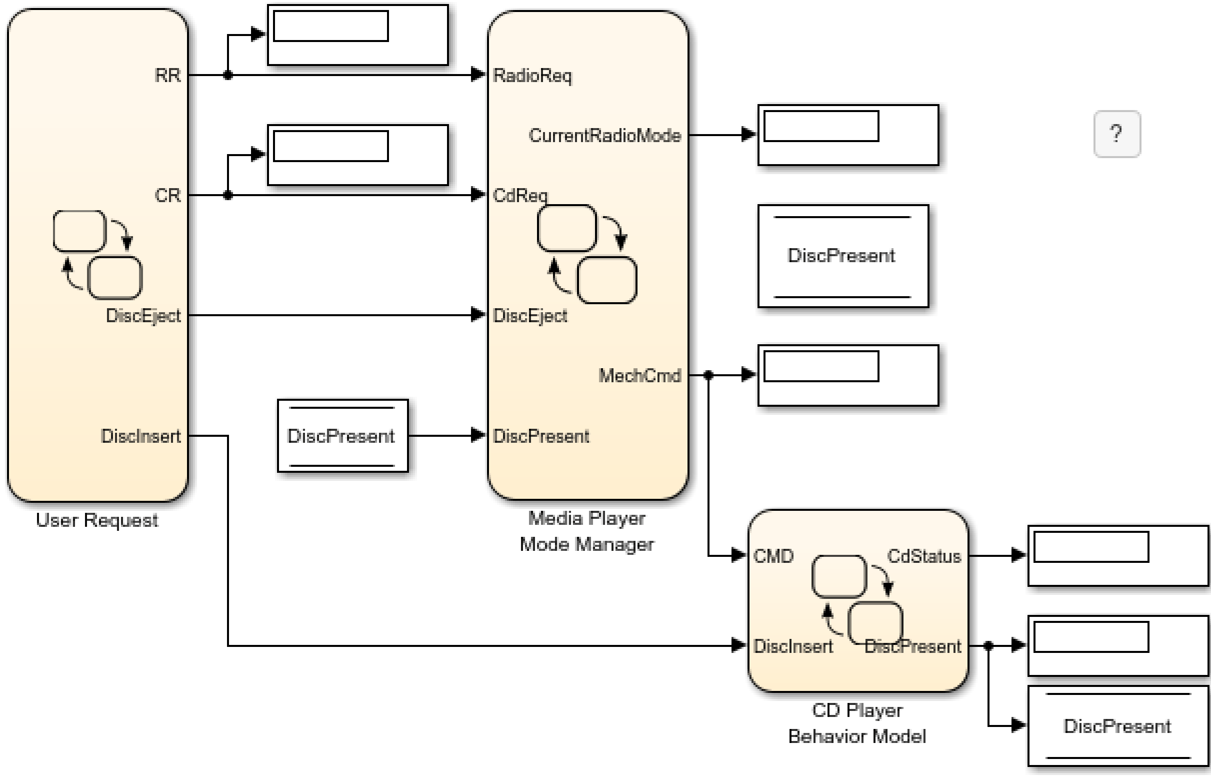
- “Define Enumerated Data Types” on page 20-5
- “Log Simulation Output for States and Data” on page 11-13
- “View Data in the Simulation Data Inspector” (Simulink)
- “Save Signal Data Using Signal Logging” (Simulink)
- “Inspect Simulation Data” (Simulink)

Model Media Player by Using Enumerated Data

This example shows how to model a media player by using enumerated data in Stateflow®. The media player consists of a Simulink® model and a MATLAB® user interface (UI). The model has these components:

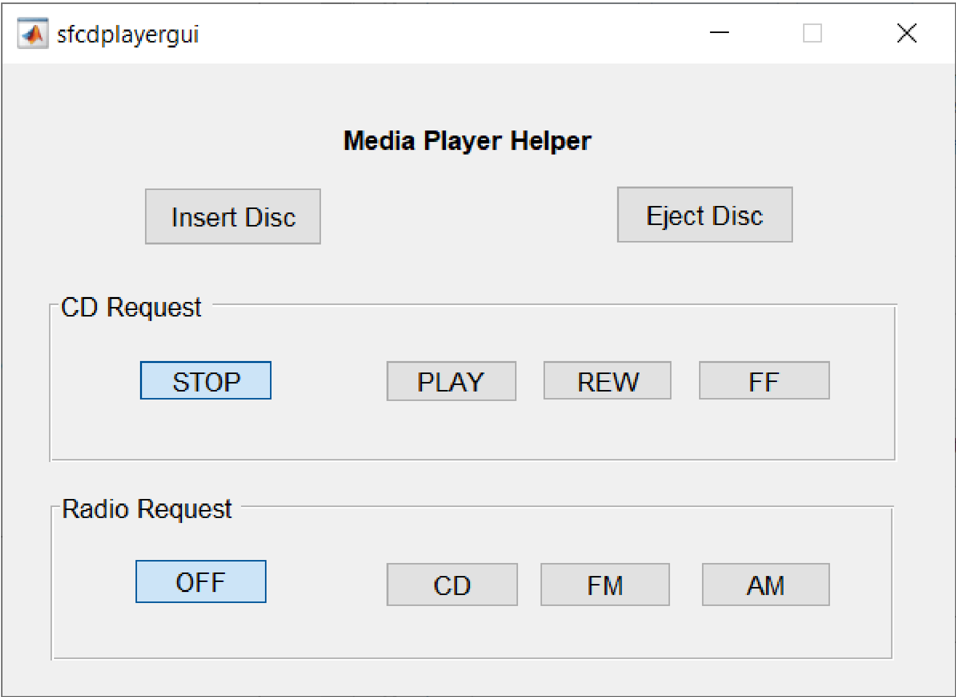
- User Request is a Stateflow chart that reads and stores user inputs from UI.
- Media Player Mode Manager is a Stateflow chart that determines whether the media player operates in AM radio, FM radio, or CD player mode.
- CD Player Behavior Model is a Stateflow chart that describes the behavior of the CD player component.

These charts use enumerated data to group related values into separate data types, reduce the amount of data, and enhance readability. For more information, see “Reference Values by Name by Using Enumerated Data” on page 20-2.



Create Groups of Related Data Values

The model uses two enumerated data types to group the possible operating modes for the media player and for its CD player component. The Media Player Helper UI separates these modes into two groups of buttons.



The **Radio Request** section contains buttons for selecting an operating mode for the media player. The enumerated values for the data type `RadioRequestMode` correspond to these media player operating modes:

- `OFF(0)`
- `CD(1)`
- `FM(2)`
- `AM(3)`

The **CD Request** section contains buttons for selecting an operating mode for the CD player component. The **Insert Disc** and **Eject Disc** buttons also affect this operating mode. The enumerated values for the data type `CdRequestMode` correspond to these CD player operating modes:

- `EMPTY(-2)`
- `DISCINSERT(-1)`
- `STOP(0)`
- `PLAY(1)`
- `REW(3)`
- `FF(4)`
- `EJECT(5)`

At the start of the model simulation, the Display blocks show the default settings of the media player. To change the enumerated values in the Display blocks, use the Media Player Helper to select other operating modes. For example:

1. In the **Radio Request** section, click **CD**. The Display blocks for enumerated data RR and `CurrentRadioMode` change from `OFF` to `CD`.

2. Click **Insert Disc**. The Display block for enumerated data `CdStatus` changes from `EMPTY` to `DISCINSERT` to `STOP`.
3. In the **CD Request** section, click **PLAY**. The Display blocks for enumerated data `CR`, `MechCmd`, and `CdStatus` change from `STOP` to `PLAY`.

Read Input from User Interface

The User Request chart reads requests from the Media Player Helper UI and stores the information as these outputs:

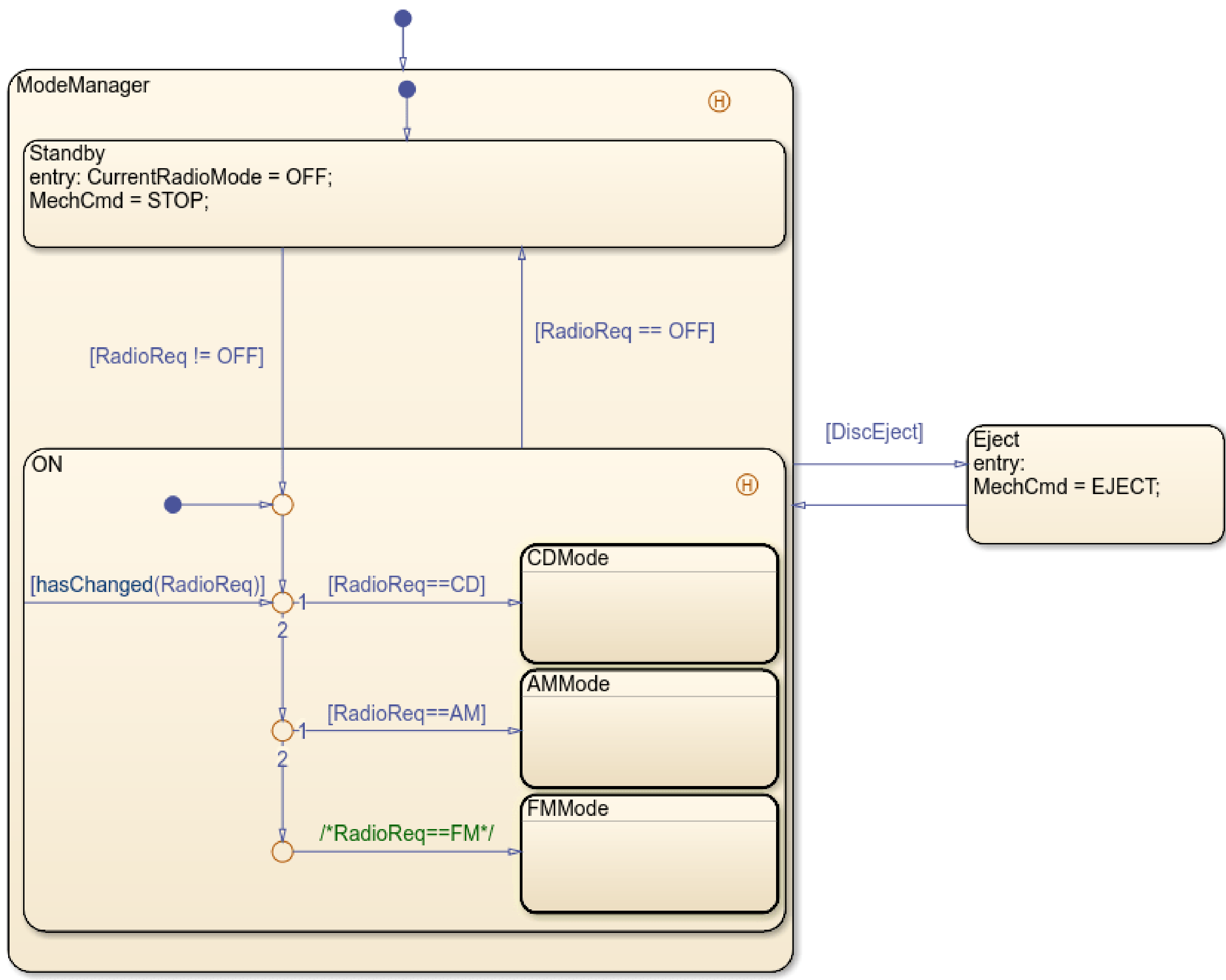
- `RR`: Enumerated data representing a **Radio Request** button.
- `CR`: Enumerated data representing a **CD Request** button.
- `DiscInsert`: Boolean data representing the **Insert Disc** button.
- `DiscEject`: Boolean data representing the **Eject Disc** button.

To read the input from the UI, the chart uses the `ml` namespace operator to call the function `sfcdplayerhelper` on the MATLAB path. For more information, see “Access MATLAB Functions and Workspace Data in C Charts” on page 14-20.

```
{
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');
}
```

Select Mode Based on Changes in Enumerated Data

The Media Player Mode Manager chart activates a subcomponent of the media player depending on the input from the User Request chart.



At the start of the model simulation, the `ModeManager` state is active. If the Boolean input data `DiscEject` becomes `true`, a transition to the `Eject` state occurs, followed by a transition back to the `ModeManager` state.

When `ModeManager` is active, the previously active substate (`Standby` or `ON`, as recorded by the history junction) becomes active. Subsequent transitions between the `Standby` and `ON` substates depend on the enumerated input data `RadioReq`:

- If `RadioReq` is `OFF`, the `Standby` substate is activated.
- If `RadioReq` is not `OFF`, the `ON` substate is activated.

In the `ON` substate, three subcharts represent the operating modes of the media player: CD player, AM radio, and FM radio. Each subchart corresponds to a different value of enumerated input data `RadioReq`:

- If `RadioReq` is `CD`, the `CDMode` subchart is activated. The subchart outputs `PLAY`, `REW`, `FF`, and `STOP` commands to the CD Player Behavior Model chart.

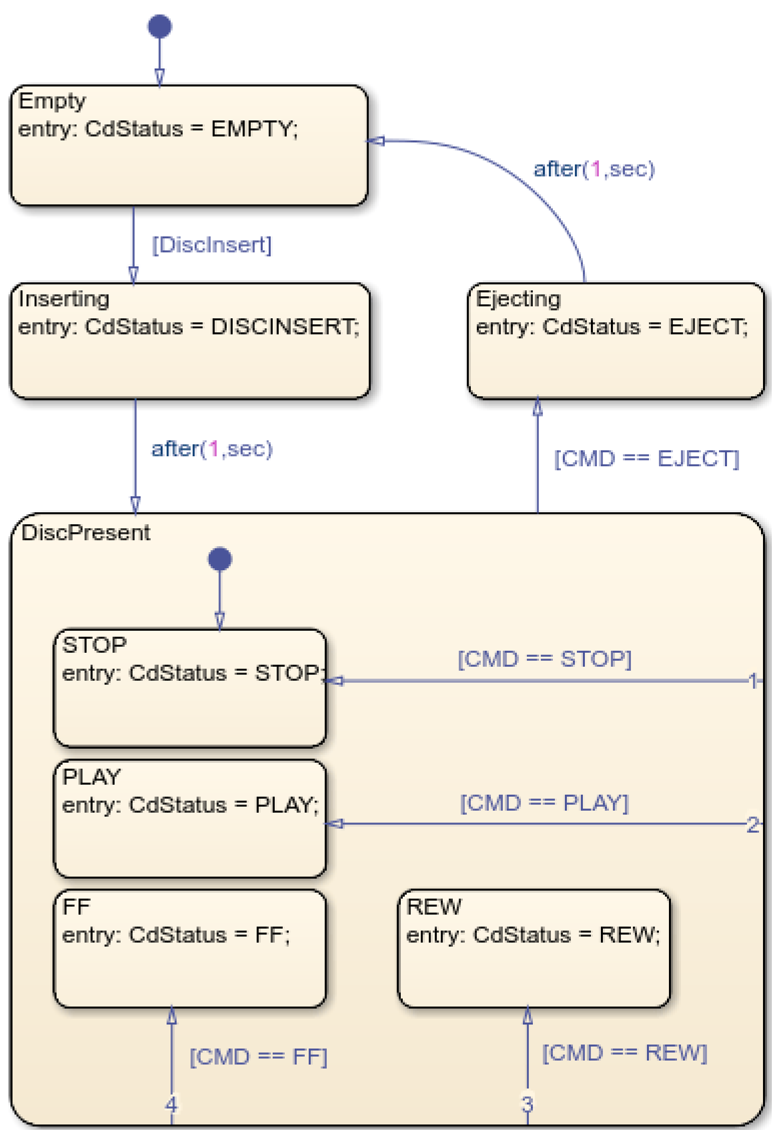
- If RadioReq is AM, the AMMode subchart is activated. The subchart outputs a STOP command to the CD Player Behavior Model chart.
- If RadioReq is FM, the FMMode subchart is activated. The subchart outputs a STOP command to the CD Player Behavior Model chart.

To scan for changes in the value of RadioReq, the inner transition inside the ON state calls the change detection operator hasChanged at every time step.

Control Timing of Transitions

The CD Player Behavior Model chart implements the behavior of the CD player mechanism depending on the input from the User Request and Media Player Mode Manager charts. To model the mechanical delays in the CD player, the chart uses the absolute-time temporal logic operator after. For instance:

- At the start of the model simulation, the Empty state is activated. If the Boolean input data DiscInsert is true, a transition to the Inserting state occurs. After a one-second delay, a transition to the DiscPresent state occurs.
- The DiscPresent state remains active until the input data CMD becomes EJECT. At that point, a transition to the Ejecting state occurs. After a one-second delay, a transition to the Empty state occurs.



Whenever a state transition occurs, the enumerated output data `CdStatus` changes value to reflect the status of the CD player:

- `CdStatus = EMPTY` when the active substate is `Empty` (CD player is empty).
- `CdStatus = DISCINSERT` when the active substate is `Inserting` (CD player is loading a disc).
- `CdStatus = EJECT` when the active substate is `Ejecting` (CD player is ejecting a disc).
- `CdStatus = STOP` when the active substate is `DiscPresent.STOP` (CD player is stopped).
- `CdStatus = PLAY` when the active substate is `DiscPresent.PLAY` (CD player is playing).
- `CdStatus = REW` when the active substate is `DiscPresent.REW` (CD player is rewinding).
- `CdStatus = FF` when the active substate is `DiscPresent.FF` (CD player is fast-forwarding).

See Also

`after` | `hasChanged`

More About

- “Reference Values by Name by Using Enumerated Data” on page 20-2
- “Access MATLAB Functions and Workspace Data in C Charts” on page 14-20
- “Detect Changes in Data and Expression Values” on page 14-63
- “Control Chart Execution by Using Temporal Logic” on page 14-35

String Data in Charts

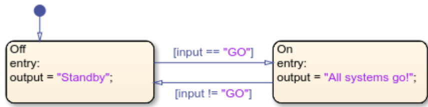
- “Manage Textual Information by Using Strings” on page 21-2
- “Log String Data to the Simulation Data Inspector” on page 21-5
- “Send Messages With String Data” on page 21-8
- “Share String Data with Custom C Code” on page 21-10
- “Simulate a Media Player” on page 21-14

Manage Textual Information by Using Strings

You can control chart behavior and output easy-to-read text by using strings to create textual data.

Creating Strings in Stateflow

In Stateflow, a string is a piece of text surrounded by double quotes (`"..."`). In addition, single-quoted strings (`'...'`) are supported in charts that use C as the action language. For example, this chart takes string data as input. Based on that input, the chart produces a corresponding string output.



To specify a string symbol, first open the **Property Inspector**. In the **Symbols** pane, select the data that you want to convert to a string. In the **Property Inspector**, set the **Type** to `string`. Stateflow dynamically allocates memory space for this type of data.

Computation with Strings

To manipulate string data in a Stateflow chart, use the operators in this table.

Goal	MATLAB Action Language Function	Example	C Action Language Function	Example

Concatenate two strings	`plus`	`h = "Hello,"`
		`w = " world!"`
		`x = plus(h,w)`
			`strcat`	`s1 = "State";`
				`s2 = "flow";`
				`dest = strcat(s1,s2);`

Determine the length of a string	`strlength`	`h = "Hello, world!"` `x = strlength(h)`	`strlen`	`L = strlen("Stateflow");`
Convert a string to a double	`str2double`	`X = str2double("-12.345");`	`str2double`	`X = str2double("-12.345");`
Convert numeric, Boolean, or enumerated data to string	`string`	`a = [1307];` `str = string(a)`	`tostring`	`dest = tostring(RED);`

String Truncation

You can also create string data with a maximum number of characters. To specify symbol as a string with a buffer size of *n* characters, set the **Type** field of the symbol to `stringtype(n)`. The text of the string can be shorter than the buffer, but if the string length exceeds the buffer size, then the text in the string is truncated. For example, if the **Type** property of the symbol `output` in the previous chart is `stringtype(10)`, then its value in the state `On` is truncated to `"All system"`.

You can enable the **String truncation checking** parameter to choose whether to stop simulation or generate a warning when a string exceeds the length specified by `stringtype(n)`.

String Truncation Checking	Description
`error`	Simulation stops with an error.
`warning`	String is truncated. Simulation continues with a warning.
`none`	String is truncated. Simulation continues with no error or warning.

Note Unlike C or C++, Stateflow interprets escape sequences as literal characters. For example, the string `"\n"` contains two characters, backslash and `n`, and not one newline character.

Differences Between Charts That Use MATLAB and C as the Action Language

Key differences between strings in charts that use C as the action language and charts that use MATLAB include:

- Charts that use MATLAB as the action language support only strings enclosed with double-quotes. In charts that use C as the action language, strings can use double quotes or single quotes.
- In charts that use MATLAB as the action language,`strcmp` returns `1 (true)` when the strings match. In charts that use C as the action language, `strcmp` returns `0`.
- In charts that use MATLAB as the action language, `strcmp` returns a boolean. In charts that use C as the action language, `strcmp` returns a double.
- To return the length of the string, use `strlength` in charts that use MATLAB as the action language and `strlen` in charts that use C as the action language.
- To concatenate a string, use the `+` operation in charts that use MATLAB as the action language and `strcat` in charts that use C as the action language.
- Charts that use MATLAB as the action language enforce complexity on the output variable when using `str2double`.
- Charts that use MATLAB as the action language support all comparison operations, such as `>`, `<`, or `==`.

Limitations

Parameter data cannot be strings.

The following limitations exist for charts that use MATLAB as the action language:

- Constant data cannot be strings.

- These operators are not supported:
  - `strcat`
  - `extract`
  - `extractBetween`
  - `sscanf`
  - `compose`
  - `append`
  - `pad`
  - `count`
  - `sfprint`
- Structures can only use these operators:
  - `isstring`
  - `strcmp`
  - `string`
  - `strlength`

  For more information about Stateflow structures, see “Access Bus Signals Through Stateflow Structures” on page 26-2.

See Also

`ascii2str` | `str2ascii` | `str2double` | `strcat` | `strcmp` | `strcpy` | `strlen` | `substr` | `tostring`

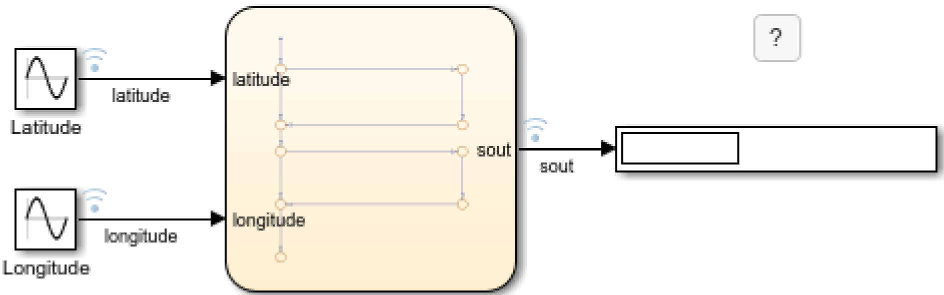
More About

- “Log String Data to the Simulation Data Inspector” on page 21-5
- “Send Messages With String Data” on page 21-8
- “Share String Data with Custom C Code” on page 21-10
- “Simulate a Media Player” on page 21-14
- “Simulink Strings” (Simulink)

Log String Data to the Simulation Data Inspector

This example shows how to build a Stateflow® chart that, based on numeric input data, concatenates string data into natural language output text. You can view the output text in the Simulation Data Inspector and the MATLAB® workspace. For more information about string data, see “Manage Textual Information by Using Strings” on page 21-2.

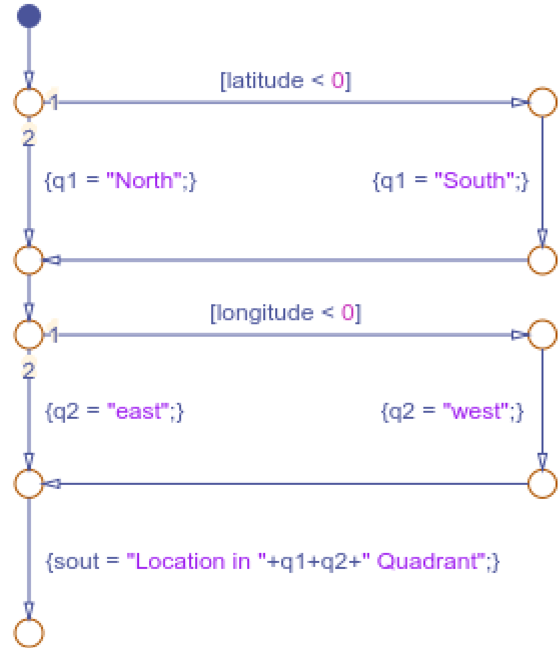
The model in this example uses Sine Wave blocks to provide the latitude and longitude of a point moving along a closed path. The Sine Wave blocks have an amplitude of 50, a bias of 0, and a frequency of 1. The `Latitude` block has a phase of 0, while the `Longitude` block has a phase of $\pi/2$.



The chart examines these coordinates and assigns the strings `q1` and `q2` according to these rules:

- If `latitude` and `longitude` are positive, `q1 = "North"` and `q2 = "east"`.
- If `latitude` is positive and `longitude` is negative, `q1 = "North"` and `q2 = "west"`.
- If `latitude` is negative and `longitude` is positive, `q1 = "South"` and `q2 = "east"`.
- If `longitude` and `longitude` are negative, `q1 = "South"` and `q2 = "west"`.

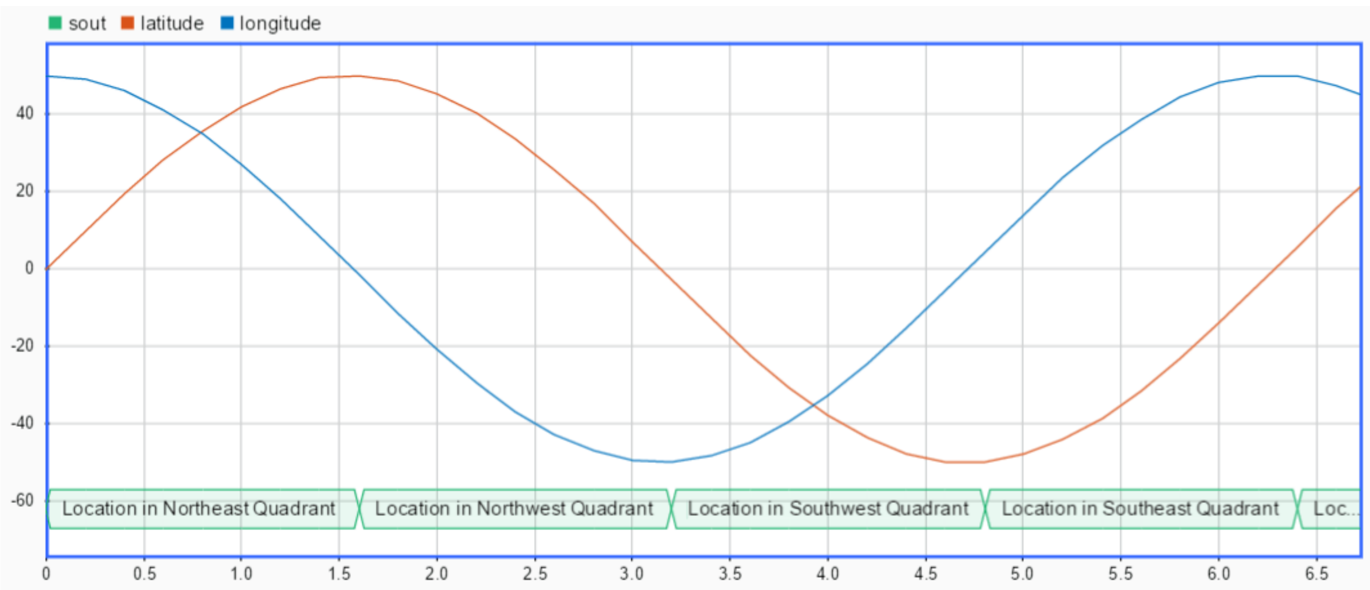
Then, the chart concatenates these strings into an output string `sout`.



View Simulation Results

During simulation, the model logs the signals coming into and out of the chart.

To examine the results of the simulation, open the Simulation Data Inspector and select the check boxes for the `latitude`, `longitude`, and `sout` signals. The `latitude` and `longitude` signals appear as sinusoidal curves. The `sout` signal appears as a transition plot. The transition plot displays the value of the string inside a band in which criss-crossed lines mark the changes in value.



To access the logged data in the MATLAB workspace, call the signal logging object `logsout`. Stateflow exports the string data `sout` as a MATLAB string scalar. For example, at the command prompt, enter:

```
Table = table(logsout.get("latitude").Values.Data, ...
    logsout.get("longitude").Values.Data, ...
    logsout.get("sout").Values.Data);
Table.Properties.VariableNames = ...
    ["Latitude","Longitude","QuadrantInfo"];
Table([4:8:30],:)
```

```
ans =

  4x3 table

    Latitude    Longitude             QuadrantInfo
    ________    _________    ________________________________

     28.232       41.267     "Location in Northeast Quadrant"
     40.425      -29.425     "Location in Northwest Quadrant"
    -30.593      -39.548     "Location in Southwest Quadrant"
    -38.638       31.735     "Location in Southeast Quadrant"
```

For more information on logging chart data, see “Log Simulation Output for States and Data” on page 11-13.

See Also

`plus` | `get` | `table`

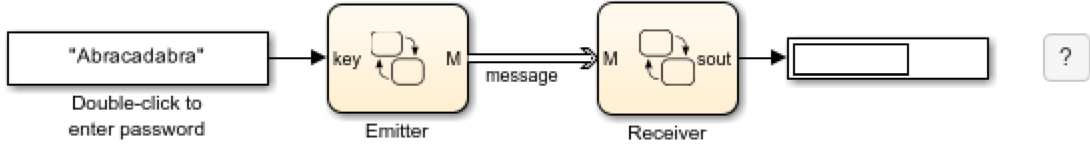
More About

- “Manage Textual Information by Using Strings” on page 21-2
- “Log Simulation Output for States and Data” on page 11-13
- “View Data in the Simulation Data Inspector” (Simulink)
- “Save Signal Data Using Signal Logging” (Simulink)

Send Messages With String Data

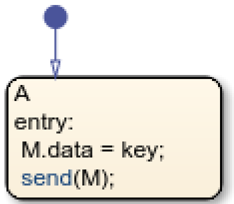
This example shows how to configure a pair of Stateflow® charts that communicate by sending messages that carry string data. For more information, see "Communicate with Stateflow Charts by Sending Messages" on page 13-2.

This model contains two Stateflow charts. During simulation, the `Emitter` chart reads an input string `key` from the String Constant block and sends a message to the `Receiver` chart. The message data consists of the input string `key`. The `Receiver` chart compares the string with a constant keyword and returns an output string that grants or denies access.



Emitter Chart

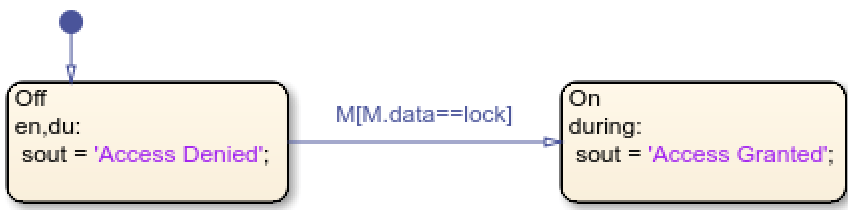
The `Emitter` chart consists of a single state. When the state becomes active, it sets the data for the output message `M` to the input value `key` and sends the message to the `Receiver` chart.



In this chart, `key` has type `Inherit: Same as Simulink` while `M` has type `string`.

Receiver Chart

The `Receiver` chart consists of two states joined by a transition. The input message `M` guards the transition. If there is a message present and its data value equals the constant string lock, then the state activity transitions from state `Off` to state `On`. The chart outputs the string value `"Access Granted"`. If there is no message present, or if the data value does not equal `lock`, the chart does not take the transition and the output value is `"Access Denied"`.

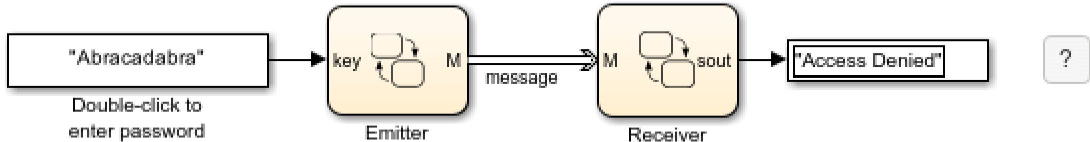


In this chart, `M` has type `Inherit: Same as Simulink`, while `lock` and `sout` have type `string`. The constant string `lock` contains a secret password, initially set to `"Open Sesame"`. You can change the value of lock in the **Value** field of the **Property Inspector**.

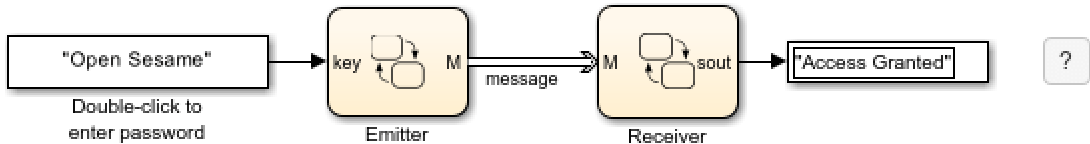
View Simulation Results

During simulation, the model responds to the password that you enter in the String Constant block:

- If you enter an incorrect password, such as `"Abracadabra"`, the model displays the output string `"Access Denied"`.



- If you enter the correct password, in this case, `"Open Sesame"`, the model displays the output string `"Access Granted"`.



See Also

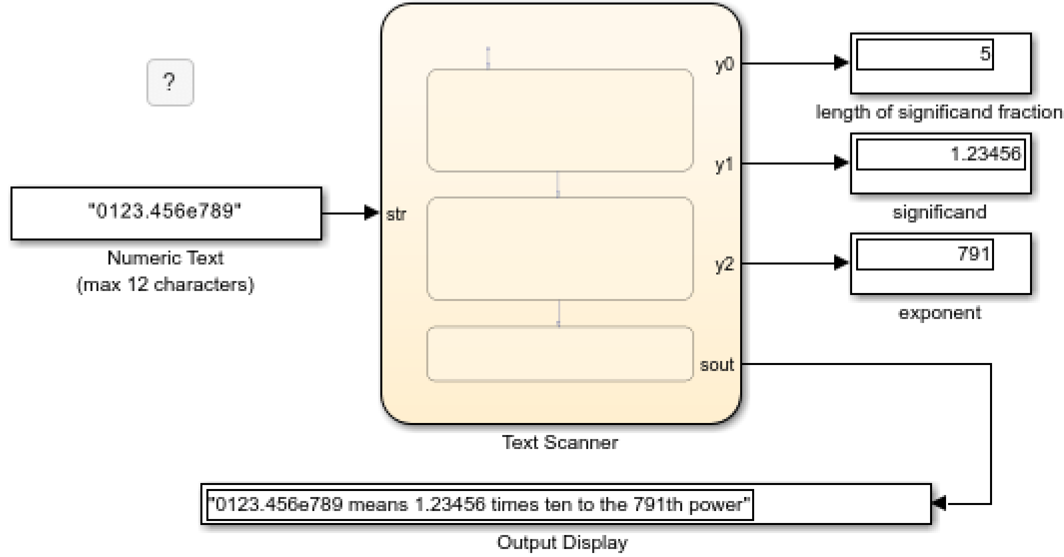
`send`

More About

- “Manage Textual Information by Using Strings” on page 21-2
- “Log String Data to the Simulation Data Inspector” on page 21-5
- “Share String Data with Custom C Code” on page 21-10
- “Simulate a Media Player” on page 21-14
- “Communicate with Stateflow Charts by Sending Messages” on page 13-2
- “View Differences Between Stateflow Messages, Events, and Data” on page 13-14

Share String Data with Custom C Code

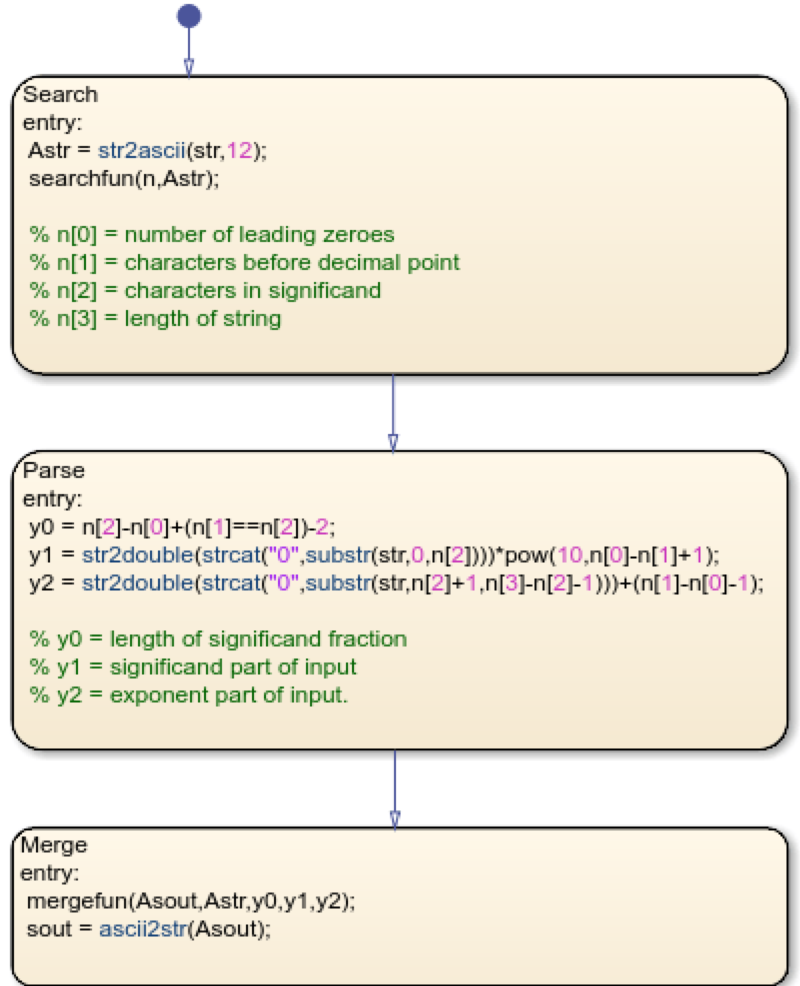
This example shows how to share string data between a Stateflow® chart and custom C code. You can export string data from a Stateflow chart to a C function by using the `str2ascii` operator. You can import the output of your C code as string data in a Stateflow chart by using the `ascii2str` operator. By sharing data with custom code, you can augment the capabilities of Stateflow and leverage the software to take advantage of your preexisting code. For more information, see “Reuse Custom Code in Stateflow Charts” on page 28-2.



This model contains a Stateflow chart that calls two functions from custom C code. During simulation, the chart takes as its input a string that contains text representing a floating-point number in exponential form. The chart consists of three states that:

- Search the input string for leading zeroes, a decimal point, and an `e`.
- Parse the string into double-precision numbers representing the significand and exponent parts of the input.
- Merge the numeric information into an output string expressing the input in scientific notation.

For example, if the input string is `"0123.456e789"`, then the chart outputs the string `"0123.456e789 means 1.23456 times ten to the 791th power"`.



Export String Data from Stateflow to C

You can use the `str2ascii` operator to convert string data into an array that you can export from a Stateflow chart to a custom C code function.

1. In the custom code function, declare the input variable as having type `char*`.
2. In the Stateflow chart, convert the string to an array of type `uint8` by calling the operator `str2ascii`.
3. Call the custom code function by passing the `uint8` array as an input.

For example, in the previous chart, the `Search` state converts the input string `str` to the `uint8` array `Asrt`. The `Search` state passes this array as an input to the custom code function `searchfun`:

```
extern void searchfun(int* nout, char* strin)
{
   nout[0] = strspn(strin,"0");
   nout[1] = strcspn(strin,".e");
   nout[2] = strcspn(strin,"e");
   nout[3] = strlen(strin);
}
```

The Search state calls this function with the command searchfun(n,Astr). The function populates the integer array n with these values:

- n[0] contains the number of leading zeroes in the input string str.
- n[1] contains the number of characters before the first instance of a decimal point or e. This result provides the number of characters before the decimal point in str.
- n[2] contains the number of characters before the first instance of e. This result provides the number of characters in the significand in str.
- n[3] contains the length of the input string str.

The Parse state uses these results to extract the values of the significand and exponent parts of the input.

Import String Data from C to Stateflow

You can import string data to a Stateflow chart by passing a pointer to an array of type uint8 as an input to a custom C function.

1. In the custom code function, declare the input variable containing the pointer as having type char*.
2. Save the output string data from the custom code function at the location indicated by the pointer.
3. In the Stateflow chart, convert the uint8 array to a string by calling the operator ascii2str.

For example, in the previous chart, the Merge state consolidates the numeric information obtained by the Parse state into an output string by calling the custom code function mergefun:

```
extern void mergefun(char* strout, char* strin, int in0, double in1, double in2)
{
   sprintf(strout, "%s means %1.*f times ten to the %dth power", strin, in0, in1, (int) in2);
}
```

The Merge state calls the mergefun function with the command mergefun(Asout,Astr,y0,y1,y2):

- Asout is an array of type uint8 pointing to the output of the custom function.
- Astr is an array of type uint8 corresponding to the input string to the chart.
- y0 is an integer containing the number of digits to the right of the decimal point in the significand.
- y1 and y2 are double-precision numbers representing the significand and exponent parts of the input.

The function mergefun calls the C library function sprintf, merging the contents of Astr, y1, and y2 and storing the result in the memory location indicated by Aout. The chart uses the operator ascii2str to convert this output to the string sout. In this way, the model imports the string constructed by the custom code function back into Stateflow.

See Also

ascii2str | str2ascii | str2double | strcat | substr

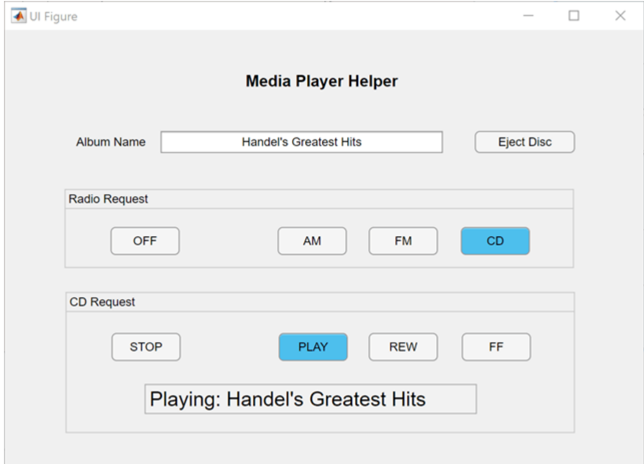
More About

- “Manage Textual Information by Using Strings” on page 21-2
- “Log String Data to the Simulation Data Inspector” on page 21-5
- “Send Messages With String Data” on page 21-8
- “Simulate a Media Player” on page 21-14
- “Reuse Custom Code in Stateflow Charts” on page 28-2

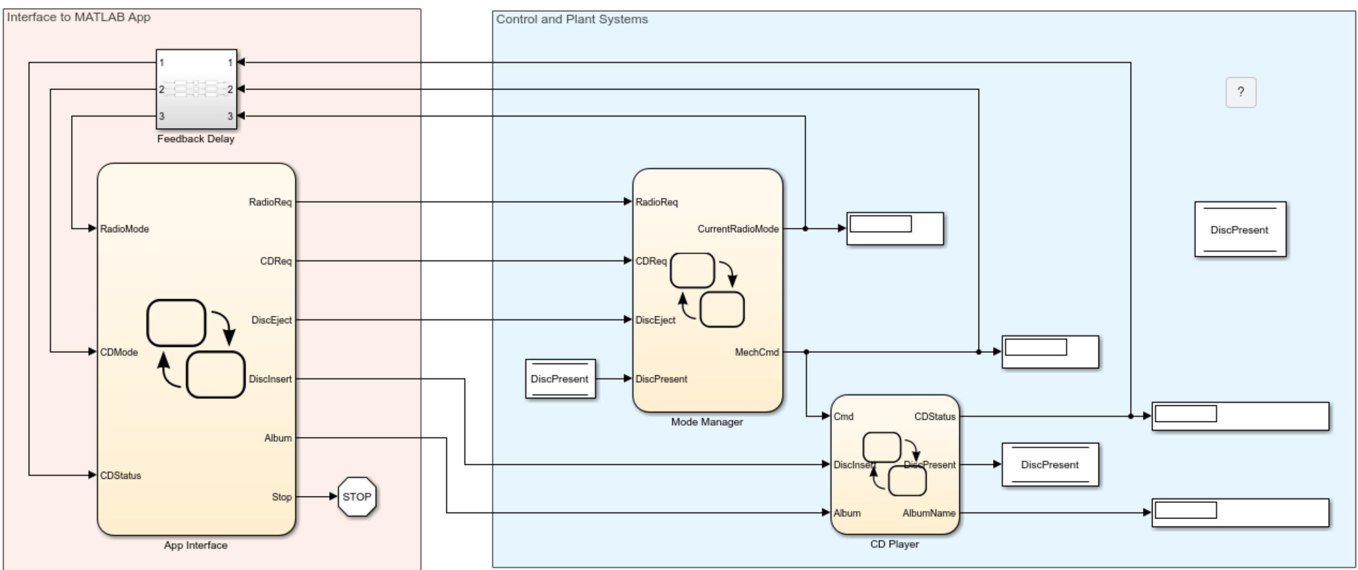
Simulate a Media Player

This example shows how to create an interface between a Stateflow® chart that uses C as the action language and a MATLAB® app created in App Designer. For more information on connecting a Stateflow chart that uses MATLAB as the action language to a MATLAB app, see "Model a Power Window Controller" on page 27-51.

In this example, a MATLAB app models the front end of a media player. During simulation, you can choose between the AM radio, FM radio, and CD player components of the media player. When the CD player is on, you can also select the playback mode.



The Stateflow chart `App Interface` provides a bidirectional connection between the MATLAB app and the control and plant systems in the Simulink® model. When you interact with the widgets in the app, the chart sends a corresponding command to the other charts in the model. These charts use strings to control the behavior of the media player and to provide natural language output messages that indicate its status. When the status of the media player changes, the chart changes the color of the buttons and updates the text field at the bottom of the app.



Run the Media Player Model

1. Open the Simulink model and click **Run**. The Media Player Helper app opens. The text field at the bottom of the app shows the status of the media player, `Standby (OFF)`.
2. In the **Radio Request** section, click **CD**. The media player status changes to `CD Player: Empty`.
3. Click **Insert Disc**. The media player status briefly says `Reading: Handel's Greatest Hits` before changing to `CD Player: Stopped`.
4. In the **CD Request** section, click **PLAY**. The media player status changes to `Playing: Handel's Greatest Hits` and music begins to play.
5. In the **CD Request** section, click **FF**. The music stops and chirping sounds begin. The media status changes to `Forward >> Handel's Greatest Hits`. The album name in this message scrolls forward across the display.
6. Use the Media Player Helper app to select other operating modes or to enter a different album name. For example, try playing to the albums `Training Deep Networks` or `Fun With State Machines`. To stop the simulation, close the Media Player Helper app.

Connect Chart to MATLAB App

The chart `App Interface` is already configured to communicate with the MATLAB app `sf_mediaplayer_app`. To create a bidirectional connection between your MATLAB app and a Stateflow chart that uses C as the action language, follow these steps. In the MATLAB app:

1. Create a custom property to interface with the chart during simulation. The app uses this property to access chart inputs, chart outputs, and local data. For more information, see “Share Data Within App Designer Apps”.
2. Modify the `startupFcn` callback for the app by adding a new input argument and storing its value as the property that you created in the previous step. For more information, see “Callbacks in App Designer”.

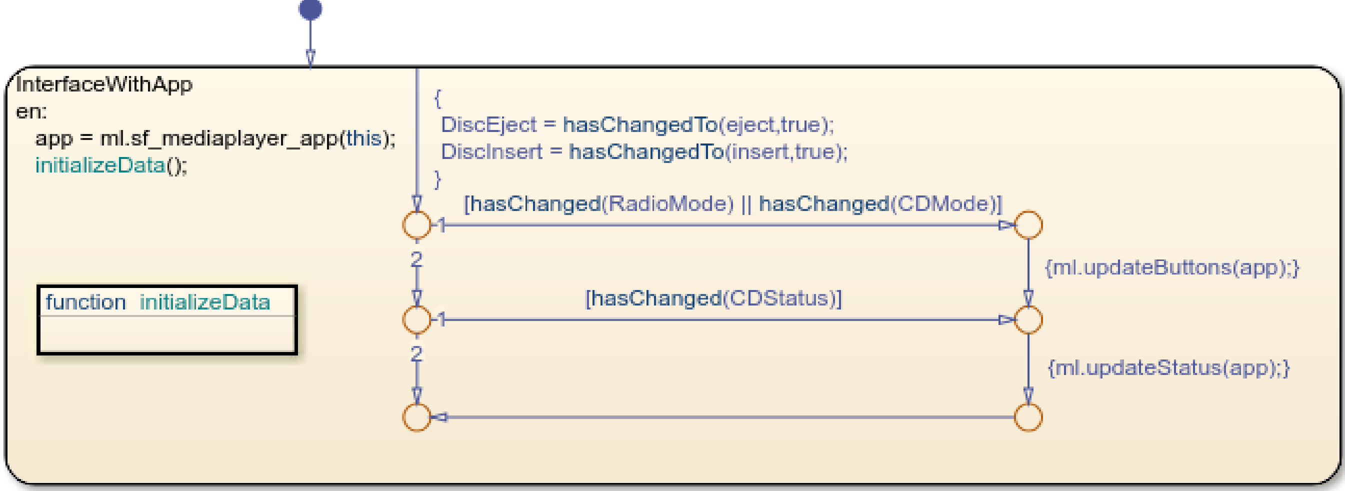
In the Stateflow chart:

1. Create a local data object to interface with the app. The chart uses this local data object as an argument when it calls helper functions in the app.
2. Set the type of the local data object you created in the previous step to `ml`. For more information, see “Specify Type of Stateflow Data” on page 10-20.
3. Run the app using the `ml` namespace operator to indicate that the app is extrinsic MATLAB code. Pass the keyword `this` as an argument to give the app access to the chart during simulation. Store the value returned by the function call to the app as the local data object that you created to interface with the app. For more information, see “Access MATLAB Functions and Workspace Data in C Charts” on page 14-20.

In this example, the Media Player Helper app uses a property called `chart` to interface with the chart `App Interface`. The app callbacks use this property to write to the chart outputs:

- When you insert or eject a disc, the `EjectButtonPushed` callback sets the values of `insert`, `eject`, and `Album`.
- When you click a button in the **Radio Request** section of the app, the corresponding callbacks set the value of `RadioReq`.
- When you click a button in the **CD Request** section of the app, the corresponding callbacks set the value of `CDReq`.
- When you close the app, the `UIFigureCloseRequest` callback sets the value of `Stop` to `true`.

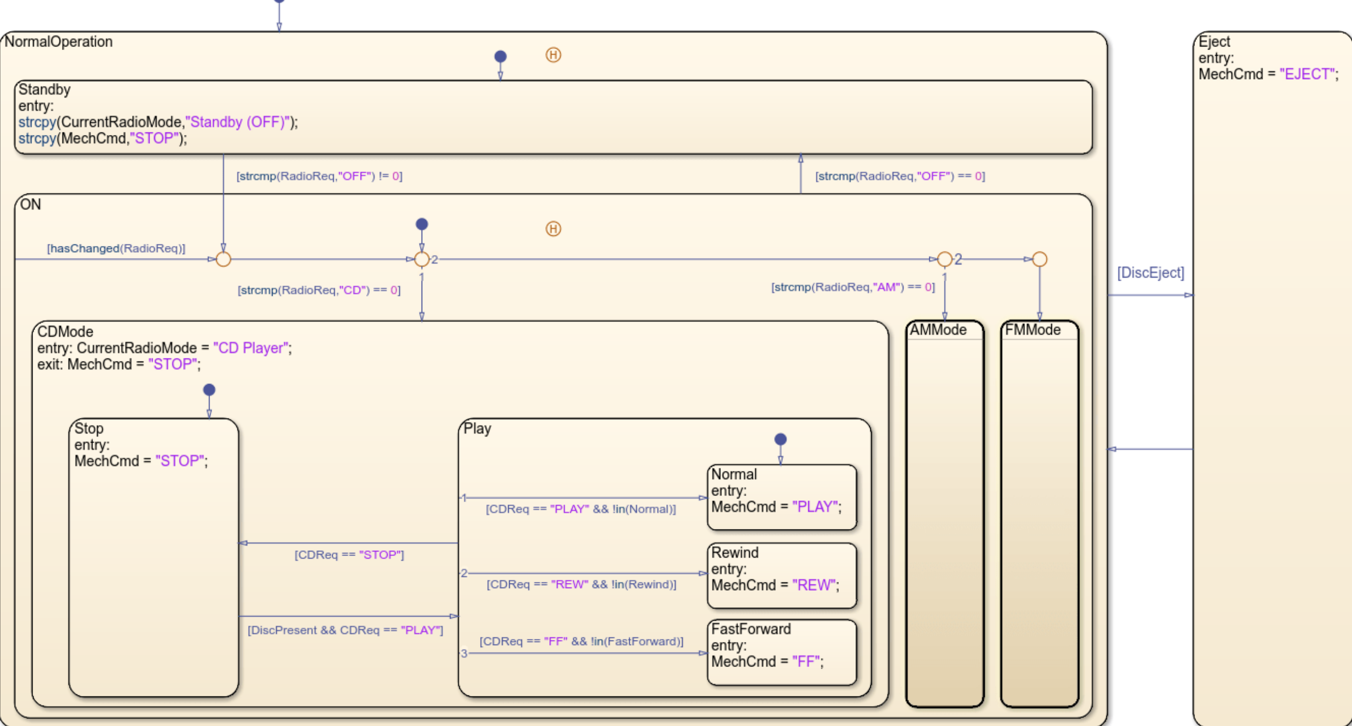
Conversely, in the chart, the entry actions in the `InterfaceWithApp` state run the app `sf_mediaplayer_app` and store the returned value as the local data object `app`. The chart uses this local data object when it calls the helper functions `updateButtons` and `updateStatus`. In the app, these helper functions change the color of the buttons and update the text field at the bottom of the app based on the value of the chart inputs `RadioMode`, `CDMode`, and `CDStatus`.



Manage Media Player Modes

The `Mode Manager` chart activates the appropriate subcomponent of the media player (AM radio, FM radio, or CD player) depending on the inputs received from the `App Interface` chart. The chart inputs `RadioReq` and `CDReq` contain string data that control the behavior of the chart. To evaluate the string data, the chart uses the string operator `strcmp` and its equivalent shorthand form `==`. The chart output `CurrentRadioMode` provides natural language output to the app, while `MechCmd`

controls the behavior of the CD player subcomponent. To assign values to these outputs, the chart uses the string operator `strcpy` and its equivalent shorthand form `=`.



At the start of simulation, the `NormalOperation` state becomes active. If the Boolean data `DiscEject` is `true`, a transition to the `Eject` state occurs, followed by a transition back to the `NormalOperation` state.

When `NormalOperation` is active, the previously active substate (`Standby` or `ON`) recorded by the history junction becomes active. Subsequent transitions between the `Standby` and `ON` substates depend on the value of the expression `strcmp(RadioReq,"OFF")`:

- If `strcmp` returns a value of zero, then `RadioReq` is `"OFF"` and the `Standby` substate is activated.
- If `strcmp` returns a nonzero value, then `RadioReq` is not `"OFF"` and the `ON` substate is activated.

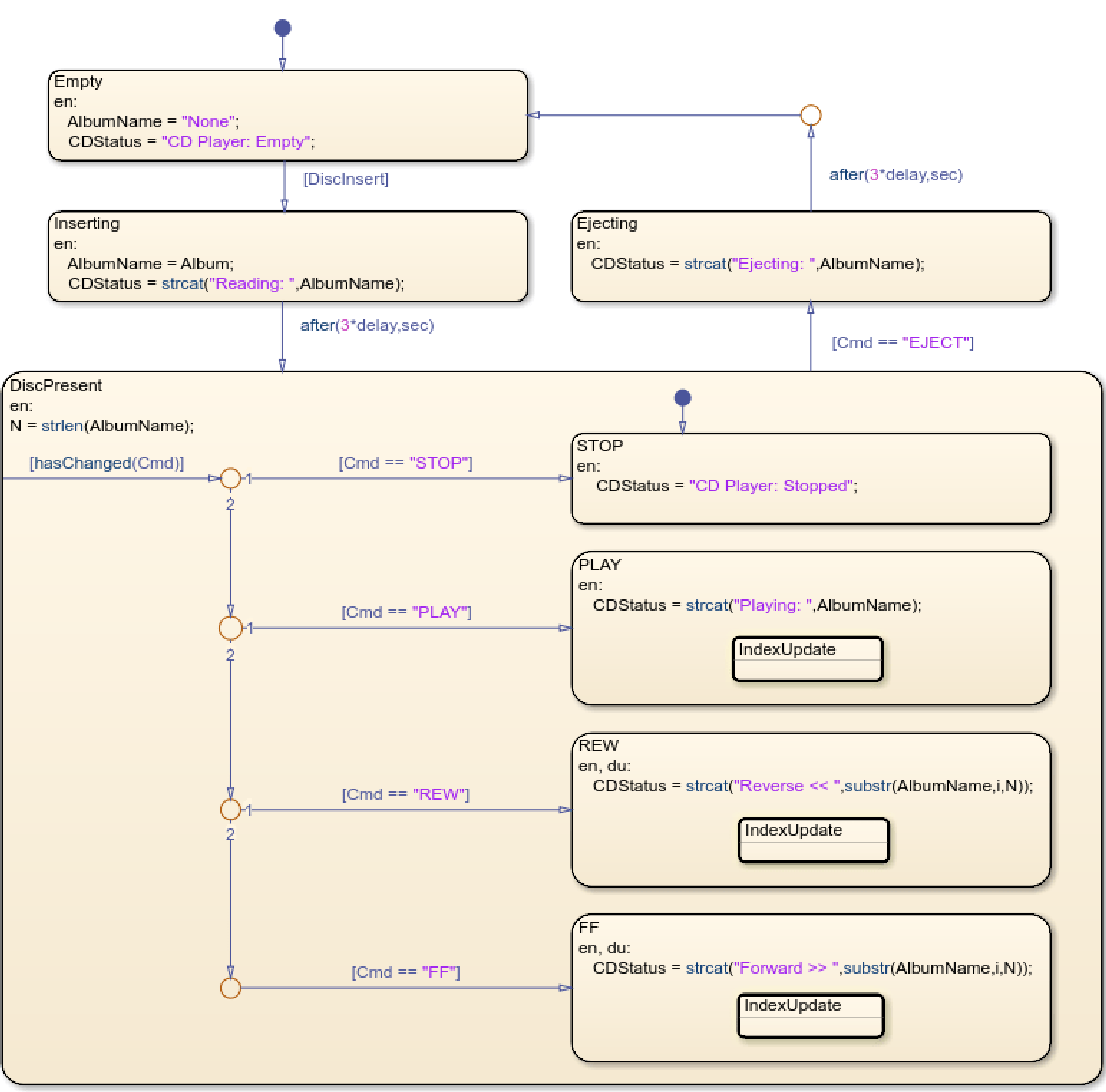
In the `ON` substate, three substates represent the operating modes of the media player: CD player, AM radio, and FM radio. Each substate corresponds to a different value of the input `RadioReq`. The inner transition inside the `ON` state uses the operator `hasChanged` to continually scan for changes in the value of `RadioReq`.

- If the value of `RadioReq` is `"CD"`, then the substate `CDMode` becomes active and the media player switches to CD player mode. The `Mode Manager` chart outputs `"PLAY"`, `"REW"`, `"FF"`, and `"STOP"` commands to the `CD Player` chart through the string data `MechCmd`.
- If the value of `RadioReq` is `"AM"`, then the substate `AMMode` becomes active and the media player switches to AM radio mode. The `Mode Manager` chart outputs a `"STOP"` command to the `CD Player` chart through the string data `MechCmd`.

- If the value of RadioReq is "FM", then the substate FMMode becomes active and the media player switches to FM radio mode. The Mode Manager chart outputs a "STOP" command to the CD Player chart through the string data MechCmd.

Manage CD Player Modes

The CD Player chart activates the appropriate operating mode for the CD player depending on the input received from the App Interface and Mode Manager charts. The chart inputs Cmd and Album contain string data that control the behavior of the chart. The chart output AlbumName provides natural language output to the app. To assign and compare the values of string data, the chart uses the shorthand operations = (see strcpy) and == (see strcmp). To produce text in the output string CDStatus, the chart uses the string operators strcat, strlen, and substr.



At the start of simulation, the `Empty` state is activated.

If the Boolean data `DiscInsert` is `true`, a transition to the `Inserting` state occurs. After a short time delay, a transition to the `DiscPresent` state occurs. The `DiscPresent` state remains active until the data `Cmd` becomes `"EJECT"`. At that point, a transition to the `Ejecting` state occurs. After a short time delay, a transition to the `Empty` state occurs. The temporal logic operator `after` controls the timing of the transitions during disc insertion and ejection.

When a state transition occurs, the entry action in the new state changes the value of `CDStatus` to reflect the status of the CD player. In the `FF` or `REW` substates, the during actions continually change the value of `CDStatus` to produce a scrolling motion effect.

- When the active state is `Empty`, the value of `CDStatus` is `"CD Player: Empty"`.
- When the active state is `Inserting`, the value of `CDStatus` is `"Reading: AlbumName"`.
- When the active state is `Ejecting`, the value of `CDStatus` is `"Ejecting: AlbumName"`.
- When the active state is `DiscPresent.STOP`, the value of `CDStatus` is `"CD Player: Stopped"`.
- When the active state is `DiscPresent.PLAY`, the value of `CDStatus` is `"Playing: AlbumName"`.
- When the active state is `DiscPresent.REW`, the value of `CDStatus` is `"Reverse << AlbumName"`, where *`AlbumName`* scrolls backward across the display.
- When the active state is `DiscPresent.FF`, the value of `CDStatus` is `"Forward >> AlbumName"`, where *`AlbumName`* scrolls forward across the display.

See Also

`after` | `hasChanged` | `strcat` | `strcmp` | `strcpy` | `strlen` | `substr`

More About

- “Manage Textual Information by Using Strings” on page 21-2
- “Detect Changes in Data and Expression Values” on page 14-63
- “Control Chart Execution by Using Temporal Logic” on page 14-35
- “Access MATLAB Functions and Workspace Data in C Charts” on page 14-20

- “Record and Play Audio”
- “Simulink Strings” (Simulink)

Continuous-Time Systems in Stateflow Charts

- "Continuous-Time Modeling in Stateflow" on page 22-2
- "Store Continuous State Information in Local Variables" on page 22-6
- "Model a Bouncing Ball in Continuous Time" on page 22-8
- "Model a DC Motor in Stateflow" on page 22-12
- "Model the Dynamics of Moving Billiard Balls" on page 22-14
- "Model Newton's Cradle" on page 22-19
- "Modeling Newton's Cradle with Virtual Reality" on page 22-23

Continuous-Time Modeling in Stateflow

Hybrid systems use modal logic to transition from one mode to another in response to physical events and conditions. In these systems, continuous-time dynamics govern each mode. A simple example of this type of hybrid system is a bouncing ball. The ball moves continuously through the air until it hits the ground, at which point a mode change or discontinuity occurs. As a result, the ball suddenly changes direction and velocity. For more information, see “Model a Bouncing Ball in Continuous Time” on page 22-8.

Simulate hybrid systems that respond to continuous and discrete mode changes by configuring Stateflow charts for continuous-time modeling. In a Stateflow chart, you can represent modal logic succinctly and intuitively as a series of states, transitions, or flow charts. You can also represent state information as continuous local variables with automatic access to time derivatives.

Continuous-time simulation is supported only in Stateflow charts in Simulink models. If your continuous system does not contain modal logic, consider using a Simulink model. For more information, see “Model a Continuous System” (Simulink).

Configure a Stateflow Chart for Continuous-Time Simulation

To enable continuous updating in a Stateflow chart, set the **Update method** chart property to `Continuous`, as described in “Specify Properties for Stateflow Charts” on page 1-19.

By default, zero-crossing detection is enabled. To disable this option, clear the **Enable zero-crossing detection** check box. For more information, see “Disable Zero-Crossing Detection” on page 22-3.

Note You cannot use Moore charts for continuous-time modeling.

Interaction with Simulink Solver

Maintain Mode in Minor Time Steps

During continuous-time simulation, a Stateflow chart updates its mode only in major time steps. In a minor time step, the chart computes outputs based on the state of the chart during the last major time step. For more information, see “Continuous Sample Time” (Simulink).

Compute Continuous State at Each Time Step

When you define local continuous variables, the Stateflow chart provides programmatic access to their derivatives. The Simulink solver computes the continuous state of the chart at the current time step based on the values of these variables and their derivatives at the previous time step. For more information, see “Continuous Versus Discrete Solvers” (Simulink).

Register Zero Crossings on State Transitions

To determine when a state transition occurs, a Stateflow chart registers a zero-crossing function with the Simulink solver. When Simulink detects a change of mode, the solver searches forward from the previous major time step to detect when the state transition occurred. For more information, see “Zero-Crossing Detection” (Simulink).

Disable Zero-Crossing Detection

Zero-crossing detection on state transitions can present a tradeoff between accuracy and performance. When detecting zero crossings, a Simulink model accurately simulates mode changes without unduly reducing step size. For systems that exhibit *chattering*, or frequent fluctuations between two modes of continuous operation, zero-crossing detection can potentially impact simulation time. Chattering requires a Simulink model to check for zero crossings in rapid succession, which can slow simulation. In these situations, you can:

- Disable zero-crossing detection.
- Choose a different zero-crossing detection algorithm for your chart.
- Modify parameters that control the frequency of zero crossings in your Simulink model.

Open the Configuration Parameters dialog box and, in the **Solver** pane, change the zero-crossing options for your model. For more information, see "Zero-Crossing Detection" (Simulink).

Guidelines for Continuous-Time Simulation

To maintain the integrity and smoothness of the results of a continuous-time simulation, constrain your charts to a restricted subset of Stateflow chart semantics. By restricting the semantics, the inputs do not depend on unpredictable factors such as:

- The number of minor intervals that the Simulink solver uses in each major time step.
- The number of iterations required to stabilize the integration and zero-crossings algorithms.

By minimizing these side effects, a Stateflow chart can maintain its state at minor time steps and update its state only during major time steps. Therefore, a Stateflow chart can compute outputs based on a constant state for continuous time.

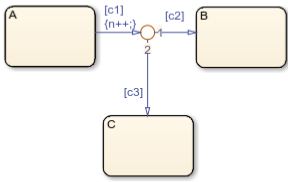
Continuous-Time Charts Must Have at Least One State

During continuous-time simulation, a chart updates its outputs at minor time steps corresponding to the `during` actions of the active state. A chart with no states produces no output. To mimic the behavior of a stateless chart in continuous time, create a single state that calls a graphical function in its `during` action.

Update Local Data in entry, exit, and Transition Actions

To maintain precision in continuous-time simulation, update discrete and continuous local data only during major time steps corresponding to state transitions. During state transitions, only these types of actions occur:

- State `exit` actions, which occur before leaving the state at the beginning of the transition.
- State `entry` actions, which occur after entering the new state at the end of the transition.
- Transition actions, which occur during the transition.
- Condition actions on a transition, but only if the transition directly reaches a state. For example, this chart executes the action `n++` even when conditions `c2` and `c3` are false. Because there is no state transition, the condition action updates `n` in a minor time step and results in an error.



Do not write to local continuous data in state `during` actions because these actions occur in minor time steps.

Compute Derivatives in State during Actions

In minor time steps, a continuous-time chart executes only state `during` actions. Because Simulink models read continuous-time derivatives during minor time steps, compute derivatives in `during` actions to provide the most current calculation.

Do Not Read Outputs or Derivatives in State during Actions or in Transition Conditions

In minor time steps, it is possible that outputs and derivatives do not reflect their most current values. To provide smooth outputs, compute values from local discrete data, local continuous data, and chart inputs.

Do Not Call Simulink Functions in State during Actions or in Transition Conditions

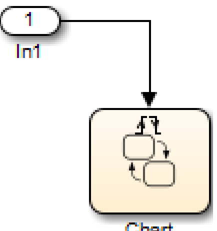
You cannot call Simulink functions during minor time steps. Instead, call Simulink functions only in actions that occur during major time steps: state `entry` or `exit` actions and transition actions. Calling Simulink functions in state `during` actions or in transition conditions results in an error during simulation. For more information, see “Reuse Simulink Functions in Stateflow Charts” on page 9-2.

Use Discrete Variables to Govern Conditions in during Actions

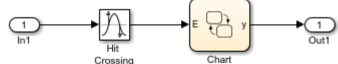
To prevent mode changes between major time steps, conditions that affect control flow in `during` actions depend on discrete variables. Discrete variables do not change value between major time steps.

Do Not Use Input Events

The presence of input events makes a chart behave like a triggered subsystem and unable to simulate in continuous time. For example, this model generates an error if the chart uses a continuous update method.



To mimic the behavior of an input event, pass the input signal through a Hit Crossing block as an input to the continuous-time chart.



Do Not Use Inner Transitions

When a mode change occurs during continuous-time simulation, the `entry` action of the destination state indicates to the Simulink model that a state transition occurred. With an inner transition, the chart never executes the `entry` action. For more information, see “Inner Transitions” on page 2-14.

Limit Use of Temporal Logic

Do not use event-based temporal logic because in continuous-time simulation, there is no concept of a tick. Use only absolute-time temporal logic for continuous-time simulation. For more information, see “Control Chart Execution by Using Temporal Logic” on page 14-35.

Do Not Use Change Detection Operators

To implement change detection, Stateflow buffers variables in a way that affects the behavior of charts between a minor time step and the next major time step.

Do Not Modify Operating Point Values

Operating points for continuous-time charts are read-only. You can save an operating point for a continuous-time chart and use it as the initial state for a simulation. However, you cannot modify the state activity or any data values in the operating point. For more information, see “Limitations on Operating Points” on page 18-6.

See Also

More About

- “Model a Bouncing Ball in Continuous Time” on page 22-8
- “Store Continuous State Information in Local Variables” on page 22-6
- “Compare Solvers” (Simulink)
- “Zero-Crossing Detection” (Simulink)

Store Continuous State Information in Local Variables

To compute a continuous state, you must determine its time derivative. You can represent this information by using local variables that are updated in continuous time. Continuous-time simulation is supported only in Stateflow charts in Simulink models. For more information, see “Continuous-Time Modeling in Stateflow” on page 22-2.

Define Continuous-Time Variables

1. Configure the chart to update in continuous time, as described in “Configure a Stateflow Chart for Continuous-Time Simulation” on page 22-2.
2. Add a data object to your chart, as described in “Add Stateflow Data” on page 10-2.
3. Set the **Scope** property for the data object to `Local`.
4. Set the **Update Method** property for the data object to `Continuous`.

In a Stateflow chart, continuous-time variables always have type `double`.

Compute Implicit Time Derivatives

For each continuous-time variable, Stateflow implicitly creates a variable to represent its time derivative. A chart denotes time derivative variables as *variable_name*`_dot`. For example, `data_dot` represents the time derivative of a continuous variable `data`. You can write to the time derivative variable in the `during` action of a state. The time derivative variable does not appear in the **Symbols** pane or in the Model Explorer.

Note Do not explicitly define variables with the suffix `_dot` in a chart configured for continuous-time simulation.

Expose Continuous State to a Simulink Model

In a Stateflow chart, you represent the continuous state by using local variables rather than inputs or outputs. To expose the continuous state to a Simulink model, you must explicitly assign the local variables to Stateflow outputs in the `during` action of a state.

Guidelines for Continuous-Time Variables

- Scope for continuous-time variables can be `Local` or `Output`.
- Define continuous-time variables at the chart level or below in the Stateflow hierarchy.
- Expose the continuous state of a chart by assigning the local continuous-time variable to a Stateflow output.

See Also

More About

- “Continuous-Time Modeling in Stateflow” on page 22-2

- "Model a Bouncing Ball in Continuous Time" on page 22-8

Model a Bouncing Ball in Continuous Time

This example shows how to configure a Stateflow® chart that simulates a bouncing ball in continuous time. The ball moves continuously through the air until it hits the ground, at which point a discontinuity occurs. As a result, the ball suddenly changes direction and velocity. For more information, see "Continuous-Time Modeling in Stateflow" on page 22-2.

The model `sf_bounce` contains a chart that updates in continuous time. Local variables describe the dynamics of a free-falling ball in terms of position and velocity. During simulation, the model uses zero-crossing detection to determine when the ball hits the ground.

Dynamics of a Bouncing Ball

You can specify how a ball falls freely under the force of gravity in terms of position p and velocity v with this system of first-order differential equations:

$$\dot{p} = v$$

$$\dot{v} = -9.81$$

When $p <= 0$, the ball hits the ground and bounces. You can model the bounce by updating the position and velocity of the ball:

- Reset the position to $p = 0$.
- Reset the velocity to the negative of its value just before the ball hit the ground.
- To account for energy loss, multiply the new velocity by a coefficient of distribution (-0.8).

Configure Chart for Continuous-Time Simulation

In the model, the BouncingBall chart implements modal logic to simulate the continuous dynamics of free fall and the discrete changes associated with bouncing. In the Chart properties dialog box, these settings enable the BouncingBall chart to simulate in continuous time:

- **Update method** is `Continuous` so the chart employs continuous-time simulation to model the dynamics of the bouncing ball.
- **Enable zero-crossing detection** is selected so the Simulink® solver can determine exactly when the ball hits the ground. Otherwise, the Simulink model cannot simulate the physics accurately and the ball appears to descend below ground.

Define Continuous-Time Variables

The BouncingBall chart has two continuous-time variables: `p` for position and `v` for velocity. For each of these variables:

- **Scope** is `Local`.
- **Type** is `double`.
- **Update Method** is `Continuous`.

To expose the continuous state of the chart to the Simulink model, the BouncingBall chart has two output variables: `p_out` and `v_out`. For each of these variables:

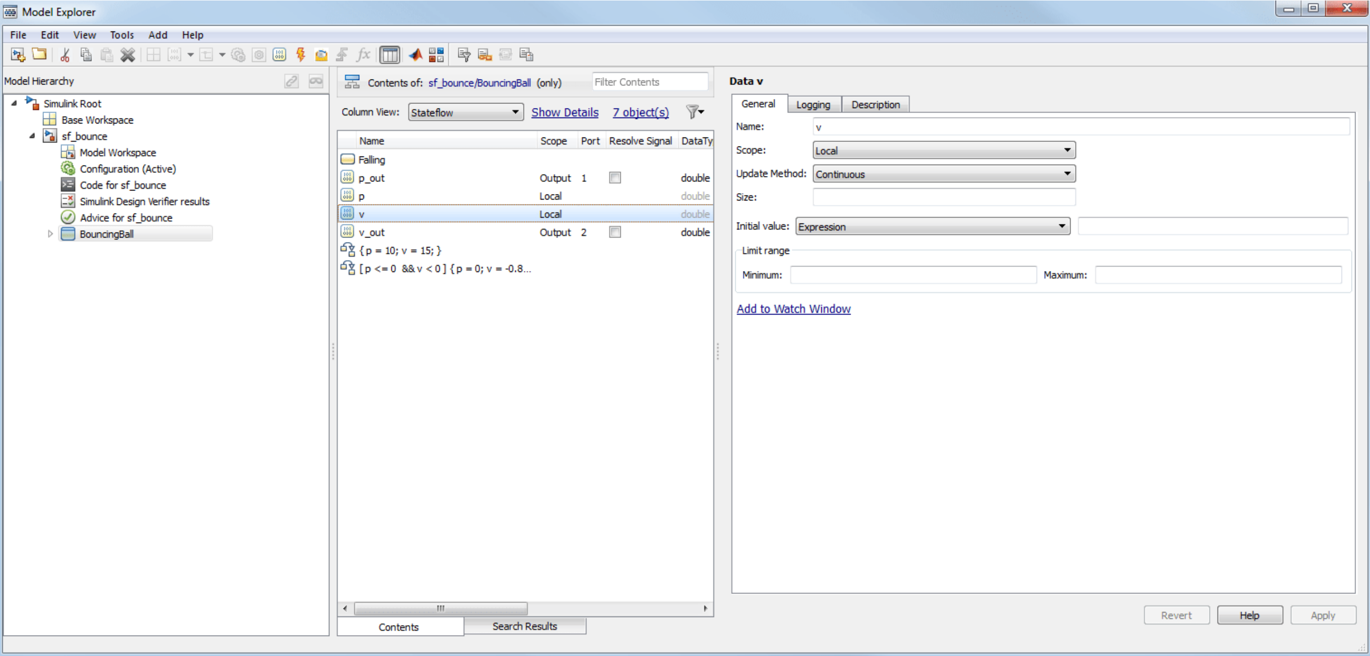
- **Scope** is `Output`.

- **Type** is `double`.
- **Update Method** is `Discrete`.

The chart defines the time derivative of continuous-time variables implicitly:

- `p_dot` is the derivative of position `p`.
- `v_dot` as the derivative of velocity `v`.

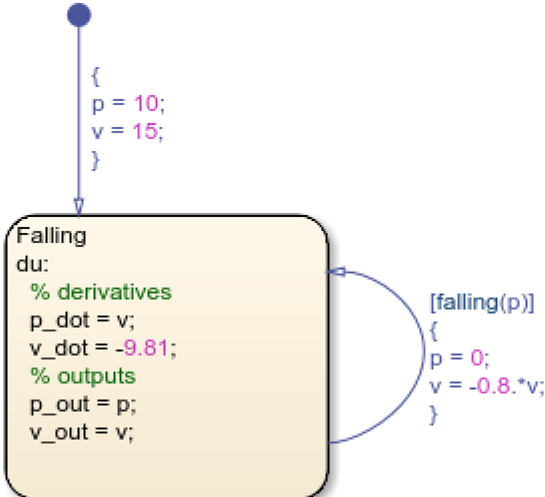
In the Model Explorer, you can view the continuous-time local variables and the corresponding outputs in the chart. Implicit derivative variables do not appear in the Model Explorer or in the **Symbols** pane.



Model Continuous Dynamics of Free Fall

The BouncingBall chart consists of a single state named `Falling` that numerically solves the differential equations for free fall. The default transition into the state sets the initial position to 10 m and the initial velocity to 15 m/s. The `during` actions in the state:

- Define the derivatives of position and velocity
- Assign the values of the position and velocity of the ball to the output variables `p_out` and `v_out`



Model Discrete Effects of the Bounce

The `Falling` state has a self-loop transition that models the discontinuity of the bounce as an instantaneous mode change when the ball suddenly reverses direction. The condition on the transition calls the edge detection operator `falling`. This operator determines when the ball hits the ground by detecting when the position crosses a threshold of zero and becomes negative. If the condition is valid, the condition action resets the position and velocity when the ball hits the ground.

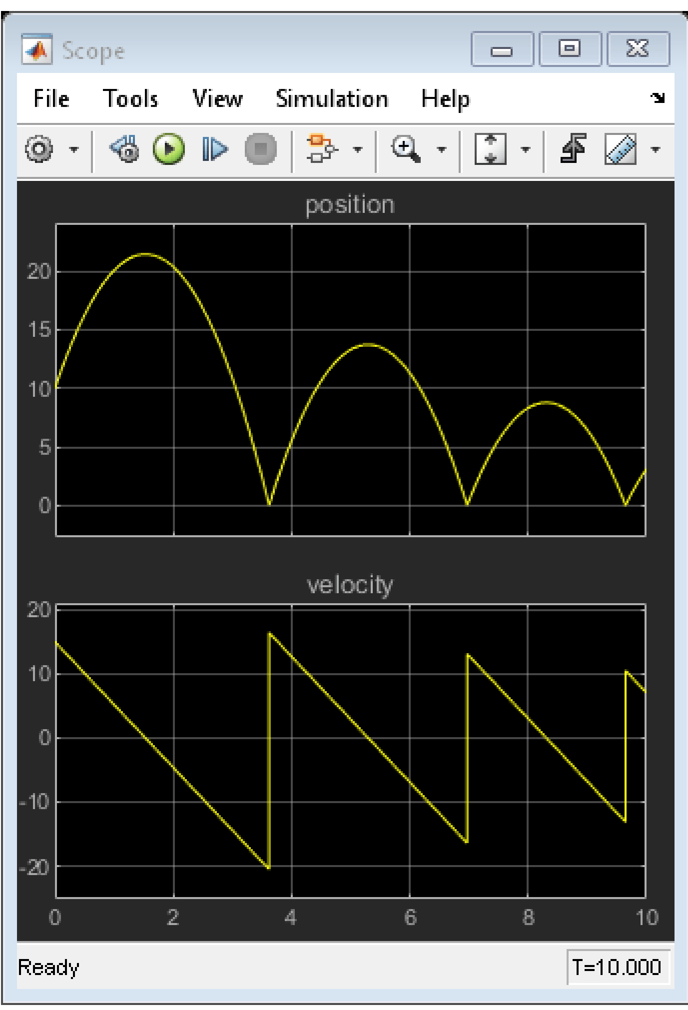
Validate Chart Semantics

The BouncingBall chart meets the design requirements defined in "Guidelines for Continuous-Time Simulation" on page 22-3. In particular, the chart:

- Initializes the local variables p and v on the default transition
- Assigns values to the derivatives p_dot and v_dot in a `during` action
- Writes to local variables p and v in a transition action
- Does not contain events, inner transitions, event-based temporal logic, or change detection operators

View Simulation Results

After you run the model, the Scope block graphs the position and the velocity of the ball. The position graph exhibits the expected bounce pattern.



See Also

More About

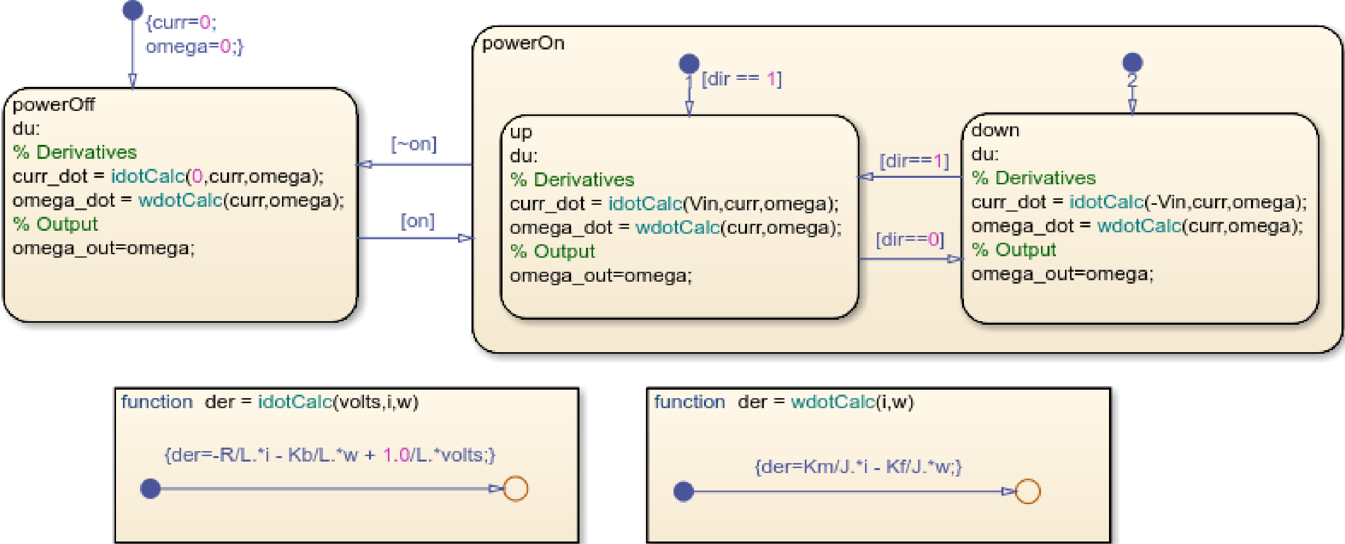
- "Continuous-Time Modeling in Stateflow" on page 22-2
- "Store Continuous State Information in Local Variables" on page 22-6

Model a DC Motor in Stateflow

This example shows the model of a permanent magnet DC motor. The mode logic and dynamics of the DC motor are both modeled using Stateflow®.

The DC motor state chart consists of two superstates: powerOn and powerOff. If the motor is powered on, it can be in one of two substates: up or down, signifying the direction of movement.

Note: This is a simplistic model of a DC motor. You can build more sophisticated DC motor models using Simscape™, which extends Simulink® with tools for modeling and simulating multidomain physical systems, such as those with mechanical, hydraulic, and electrical components.



The dynamics of the motor are defined directly in the state chart using graphical functions and change depending on the state of the motor. For example, when the motor is in the powerOff state, the voltage applied is equal to zero. When the motor is in the powerOn state, the voltage applied is either positive or negative, depending on the direction of the motor.

To review, the differential equations defining a permanent magnet DC motor are as follows:

$$\frac{di}{dt} = \frac{v_{app}(t) - R \cdot i(t) - K_b \cdot \omega(t)}{L}$$

$$\frac{d\omega}{dt} = \frac{K_m \cdot i(t) - K_f \cdot \omega(t)}{J}$$

where

$i = \text{current}$

$R = \text{resistance}$

$L = \text{inductance}$

K_b = EMF constant

ω = rotational speed of motor

v_{app} = applied voltage

K_f = damping constant

K_m = torque constant

See Also

More About

- “Continuous-Time Modeling in Stateflow” on page 22-2
- “Reuse Logic Patterns by Defining Graphical Functions” on page 6-9
- “Simscape”

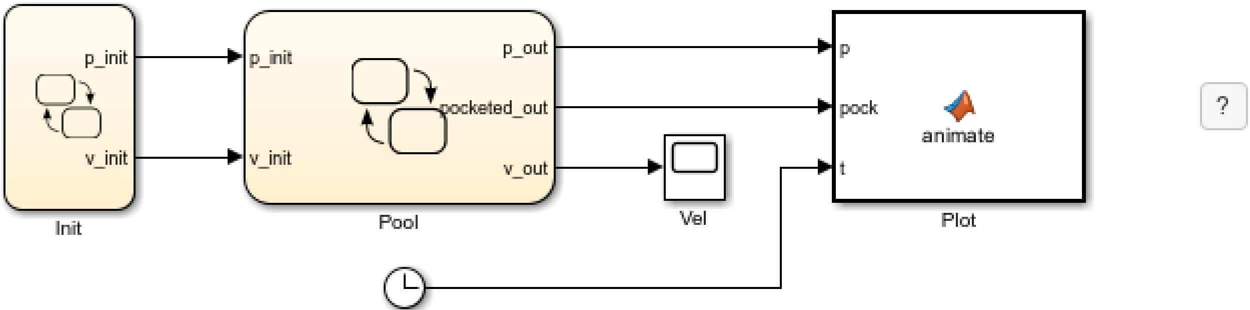
Model the Dynamics of Moving Billiard Balls

This example shows how to model the opening shot in a billiards game by using continuous-time matrix variables. In the model, a Stateflow® chart simulates the dynamics of a hybrid system that has a large number of discontinuities. For more information, see “Continuous-Time Modeling in Stateflow” on page 22-2.

When the simulation starts, a MATLAB® user interface (UI) shows a pool table with 15 billiard balls arranged in a triangular rack. The UI then prompts you to select the initial position and velocity of the cue ball. When the cue ball is released, the UI animates the motion of the billiard balls as they undergo a sequence of rapid collisions.

The model consists of:

- The Stateflow chart Init, which calls the function `sf_pool_plotter.m` to initialize the position and velocity of the cue ball based on the input from the UI.
- The Stateflow chart Pool, which calculates the two-dimensional dynamics of each billiard ball.
- The MATLAB Function block Plot, which calls the function `sf_pool_plotter.m` to animate the motion of the billiard balls during the simulation.
- The Scope block Vel, which displays the velocity of each billiard ball during the opening shot.



Calculate Continuous-Time Dynamics

To represent the dynamics of the billiard balls, the Pool chart makes several assumptions.

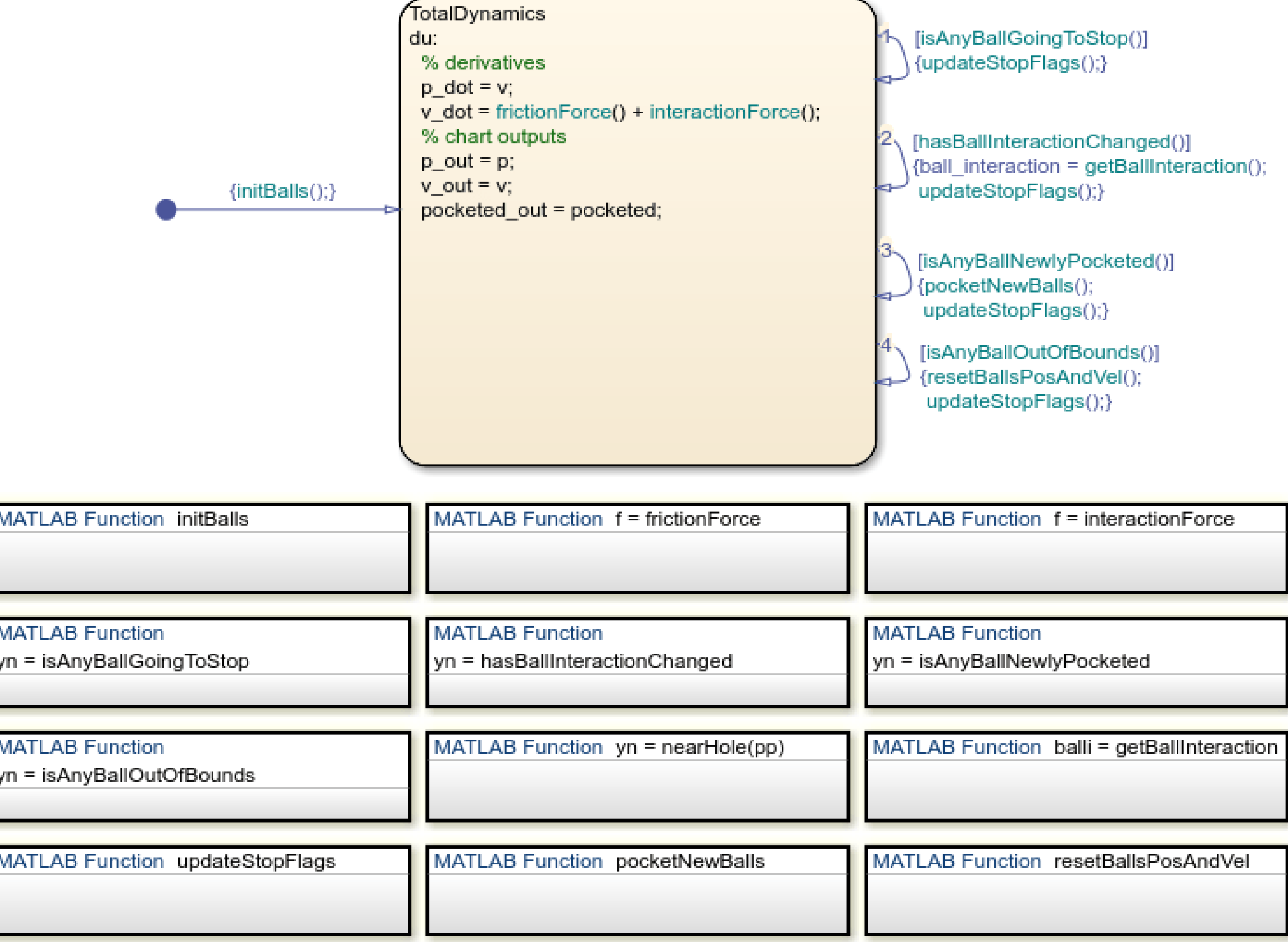
Continuous-Time Variables

The chart ignores the spin of the balls, so the state of the system is described completely by the positions and velocities of the balls. Each ball is assumed to have unit mass, so its position and velocity are described by the system of differential equations

$$\begin{cases} \dot{\mathbf{p}} = \mathbf{v} \\ \dot{\mathbf{v}} = \mathbf{F}_{\text{friction}} + \mathbf{F}_{\text{interaction}}, \end{cases}$$

where $\mathbf{F}_{\text{friction}}$ and $\mathbf{F}_{\text{interaction}}$ are the forces caused by friction with the pool table and by collisions with other balls.

To track the positions and velocities of the balls, the chart stores a pair of 16-by-2 matrices in the continuous-time variables `p` and `v`. In each matrix, the ith row represents the two-dimensional position or velocity of the ith ball.



Friction Model

To calculate the force of friction acting on each ball, the chart calls the MATLAB function `frictionForce`. This function implements a simplified friction model. Friction acts on each moving ball with a constant force opposite to the direction of motion. Because friction does not act on stationary balls, the force of friction on each ball is inherently modal:

$$\mathbf{F}_{\text{friction}} = \begin{cases} -\mu g \dfrac{\mathbf{v}}{\|\mathbf{v}\|} & \|\mathbf{v}\| > 0 \\ 0 & \text{otherwise,} \end{cases}$$

where μ is the coefficient of friction and g is the acceleration due to gravity.

Collision Dynamics

To determine the interactions caused by collisions between balls, the chart calls the MATLAB function `interactionForce`. This function implements a simple restoring force model when two balls come in contact with each other. The interaction force between the ith and j^{th} balls is modal:

$$\mathbf{F}_{\text{interaction}}(i,j) = \begin{cases} k_p(2R - \|\Delta\mathbf{p}\|)\dfrac{\Delta\mathbf{p}}{\|\Delta\mathbf{p}\|} - k_v\Delta\mathbf{v} & \|\Delta\mathbf{p}\| < 2R \\ 0 & \text{otherwise,} \end{cases}$$

where:

- R is the radius of each ball.
- k_p and k_v are constants of elasticity.
- $\Delta\mathbf{p} = \mathbf{p}_i - \mathbf{p}_j$ is the relative separation of the centers of the two balls.
- $\Delta\mathbf{v} = \mathbf{v}_i - \mathbf{v}_j$ is the relative difference in velocity between the two balls.

Because the balls are free to move in two dimensions, the chart uses the 16-by-16 Boolean matrix `ball_interaction` to account for all potential collisions. For example, when the ith and j^{th} balls are touching, the value of `ball_interaction(i,j)` is `true`. Otherwise, this value is `false`. Because collisions are symmetric in nature, the chart uses only the upper triangular portion of the matrix.

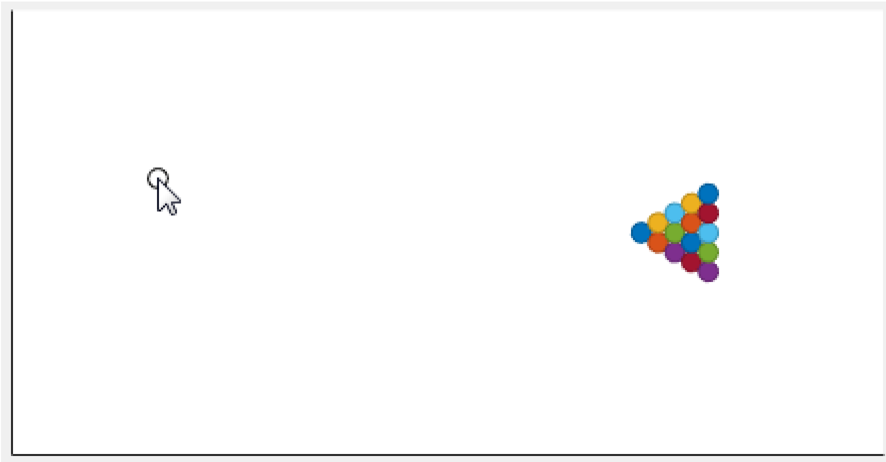
Perform Matrix Calculations in MATLAB Functions

To compute the two-dimensional dynamics of the billiard balls, the Pool chart calls several MATLAB functions that perform matrix calculations.

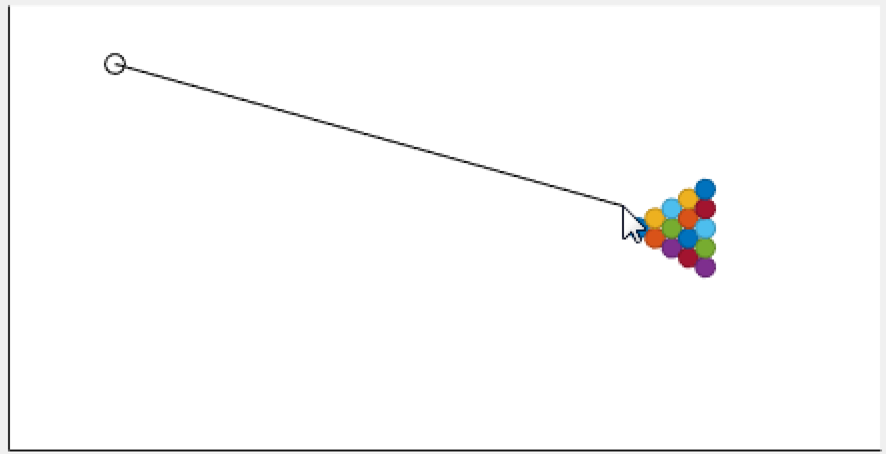
- `initBalls` initializes the position and velocity of every ball on the pool table.
- `frictionForce` calculates the friction force acting on each ball.
- `interactionForce` calculates the interaction force acting on each ball.
- `isAnyBallGoingToStop` returns a value of `1` if any ball stops moving. Otherwise, the function returns a value of `0`.
- `hasBallInteractionChanged` returns a value of `1` if any ball interactions change. Otherwise, the function returns a value of `0`.
- `isAnyBallNewlyPocketed` returns a value of `1` if any ball falls in a pocket. Otherwise, the function returns a value of `0`.
- `isAnyBallOutOfBounds` returns a value of `true` if any ball lies outside the boundary of the pool table. Otherwise, the function returns a value of `false`.
- `nearHole` returns a value of `true` if a ball is near a pocket on the pool table. Otherwise, the function returns a value of `false`.
- `getBallInteraction` returns a Boolean matrix that specifies whether any balls are in contact with each other.
- `updateStopFlags` keeps track of which balls have stopped moving and stores the result in the vector `stopped`.
- `pocketNewBalls` sets the velocity of each pocketed ball to `0`.
- `resetBallsPosAndVel` resets the position and velocity of any ball that lies outside the boundary of the pool table.

View the Simulation Results

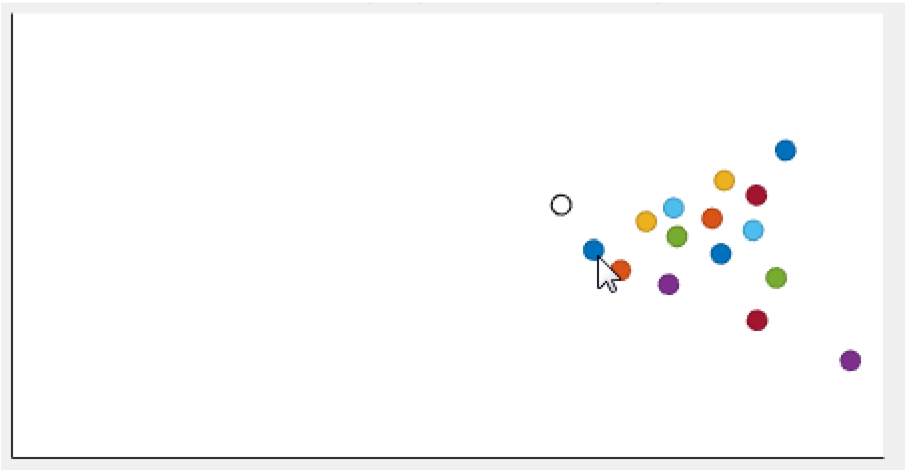
When you start the simulation, a UI shows a pool table with 15 billiard balls arranged at one end of the table. To specify the initial position of the cue ball, click anywhere on the pool table.



To specify the initial velocity of the cue ball, click a different spot on the pool table.



The model simulates the dynamics of the system and animates the motion of the billiard balls.



To stop the simulation, close the UI.

See Also

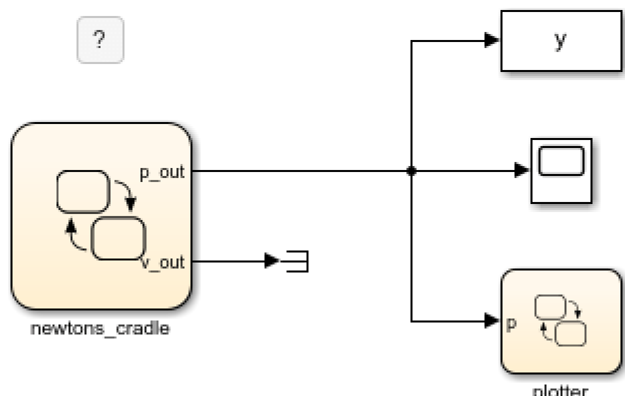
More About

- “Continuous-Time Modeling in Stateflow” on page 22-2
- “Vectors and Matrices in Stateflow Charts” on page 19-2
- “Operations for Vectors and Matrices in Stateflow” on page 19-4
- “Reuse MATLAB Code by Defining MATLAB Functions” on page 7-2

Model Newton's Cradle

This example shows how to model a popular toy called "Newton's cradle" which consists of a row of seven identical balls which are hung from a common height. At rest they are arranged such that they just touch each other. One or more balls from one end are then raised from their rest position and released.

An interesting consequence of elastic collisions between the balls is that the balls which are released seem to come to a stop and an equal number of balls from the other end get released (with almost the same energy as the incoming balls). The balls in the middle do not seem to move, although they are responsible for transferring momentum from one end to another.



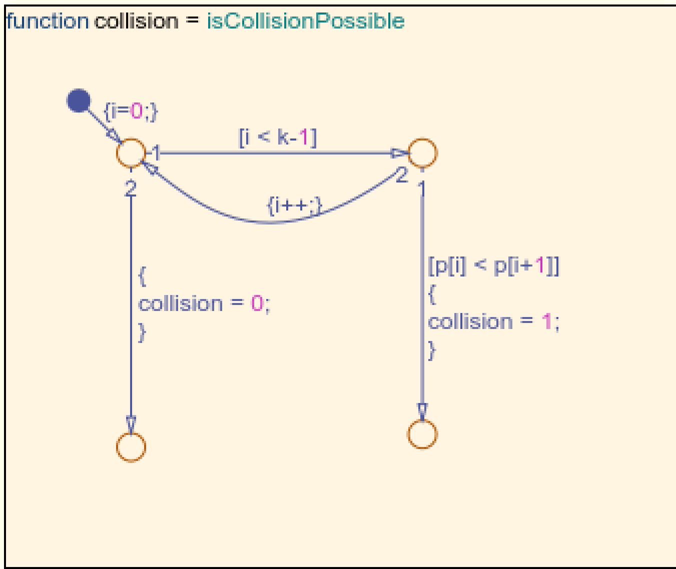
This model uses a simple elastic collision model to describe the interactions between the balls. The Stateflow® chart uses local variables to depict the continuous states of the system, namely the position `p` and the velocity `v`. Note that both these local variables are defined to have **Update method** as `continuous`. This allows you to refer to their derivatives as `p_dot` and `v_dot` respectively. Since the nominal dynamics of all the balls are identical, this example uses these vector assignments to represent the motion of all of the balls:

```
p_dot = v;
v_dot = -g/l*sin(p);
```

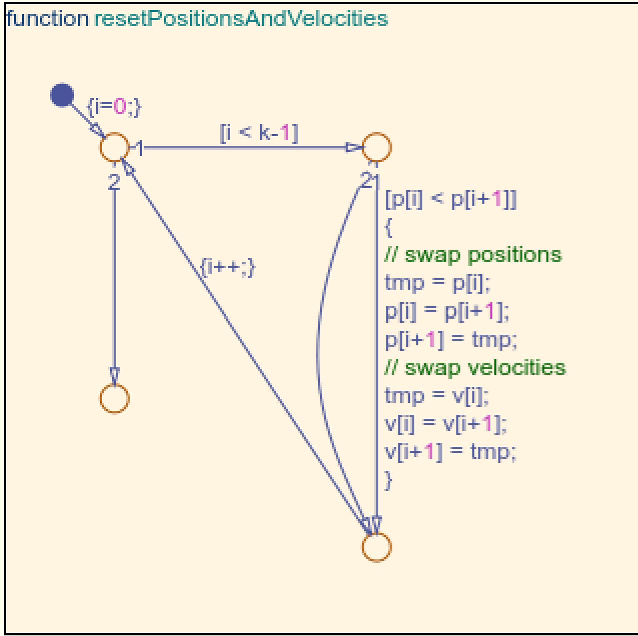
Note that `p_dot` and `v_dot` are not chart local variables. They are automatically created because `p` and `v` are defined to be continuous.

```
{
p = 0;
v = 0;
p[0] = pi/6;
p[1] = pi/6;

// uncomment the lines below to see
// some interesting behavior
// p[k-3] = -pi/6;
// p[k-2] = -pi/6;
// p[k-1] = -pi/6;
// v[0] = 2;
}

FreeDynamics
du:
  p_out = p;
  v_out = v;
  p_dot = v;
  v_dot = -g/l*sin(p);

[isCollisionPossible()]
{ resetPositionsAndVelocities(); }

function
collision = isCollisionPossible

function
resetPositionsAndVelocities
```

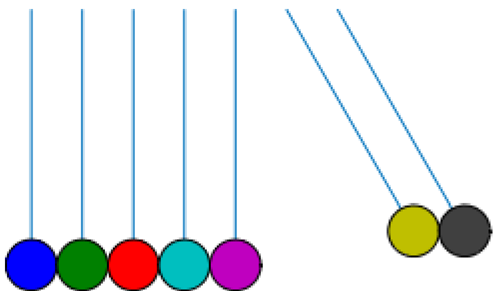
The model uses a simple for-loop to detect collisions between balls. In a one-dimensional setting, the chart only models collisions between successive balls with a single for loop.



The response to a collision is also expressed simply. Each collision is treated as a perfectly elastic instantaneous collision. The position and velocity are exchanged for each of the balls involved in the collision.



Simulating this model brings up a simple UI that shows the motion of the balls.



See Also

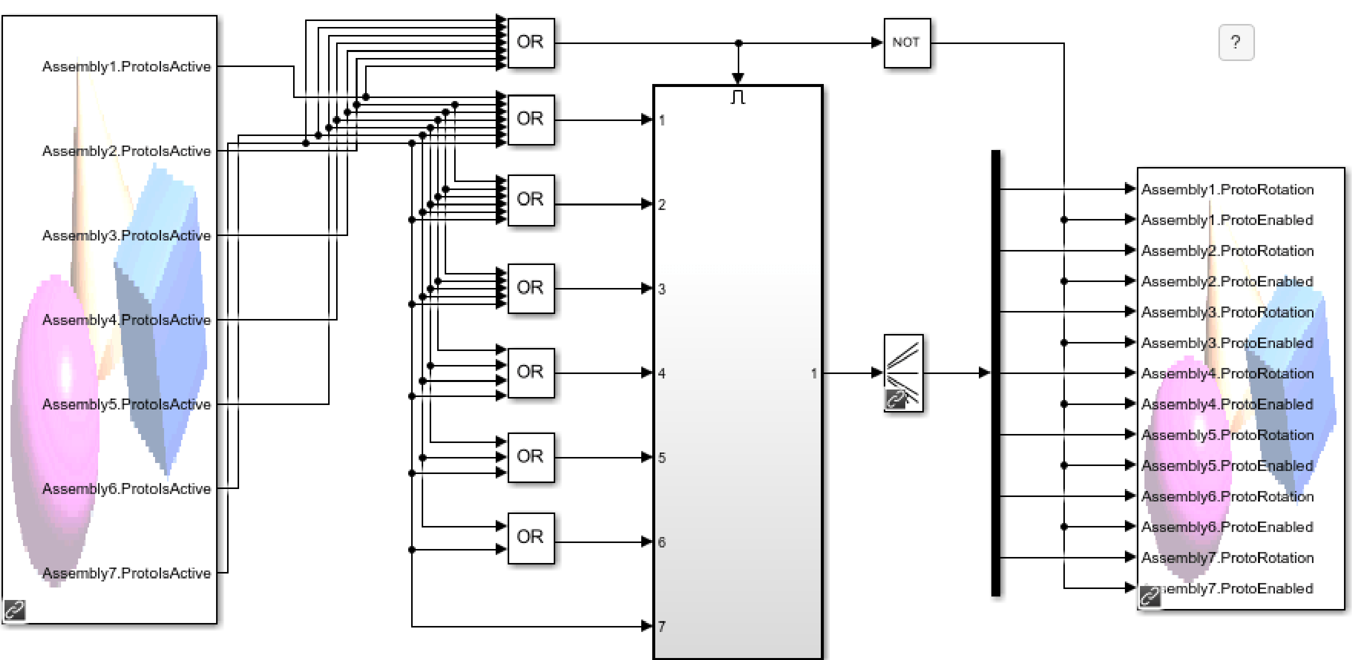
More About

- “Continuous-Time Modeling in Stateflow” on page 22-2
- “Reuse Logic Patterns by Defining Graphical Functions” on page 6-9
- “Modeling Newton's Cradle with Virtual Reality” on page 22-23

Modeling Newton's Cradle with Virtual Reality

This example shows how to model a popular toy called "Newton's cradle" which consists of a row of seven identical balls which are hung from a common height. At rest they are arranged such that they just touch each other. One or more balls from one end are then raised from their rest position and released.

An interesting consequence of elastic collisions between the balls is that the balls which are released seem to come to a stop and an equal number of balls from the other end get released (with almost the same energy as the incoming balls). The balls in the middle do not seem to move, although they are responsible for transferring momentum from one end to another.



This example uses a simple elastic collision model to describe the interactions between the balls. The Stateflow® chart uses local variables to depict the continuous states of the system, namely the position `p` and the velocity `v`. Note that both these local variables are defined to have **Update method** as `continuous`, so you can refer to their derivatives as `p_dot` and `v_dot` respectively. Since the nominal dynamics of all the balls are identical, this example uses these vector assignments to represent the motion of all of the balls:

```
p_dot = v;
v_dot = -g/l*sin(p);
```

Note that `p_dot` and `v_dot` are not chart local variables. They are automatically created because `p` and `v` are defined to be continuous.

```
{
p = 0;
v = 0;

p[0] = pi/3*input0;
p[1] = pi/3*input1;
p[2] = pi/3*input2;
p[3] = pi/3*input3;
p[4] = pi/3*input4;
p[5] = pi/3*input5;
p[6] = pi/3*input6;

// uncomment the lines below to see
// some interesting behavior
// p[k-3] = -pi/6;
//p[k-2] = -pi/6;
// p[k-1] = -pi/6;
// v[0] = 2;
}
```

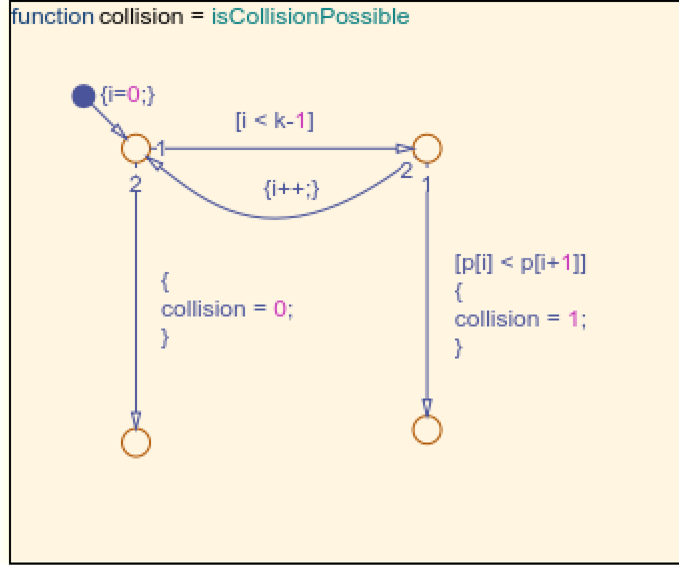
```
function
resetPositionsAndVelocities
```

```
function
collision = isCollisionPossible
```

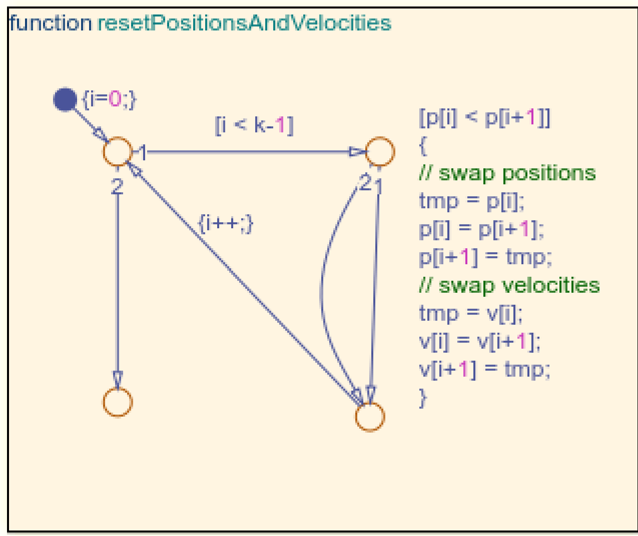
```
FreeDynamics
du:
  p_out = p;
  v_out = v;
  p_dot = v;
  v_dot = -g/l*sin(p);
```

```
[isCollisionPossible()]
{resetPositionsAndVelocities();}
```

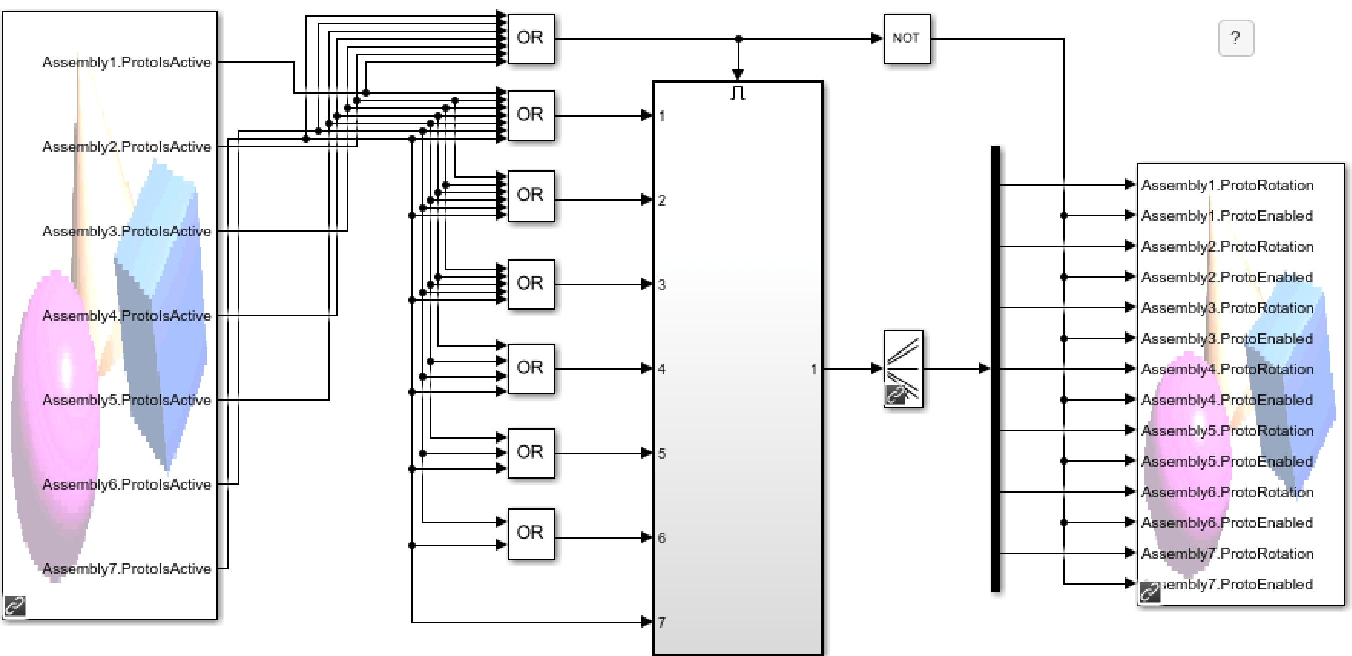
The model uses a simple for-loop to detect collisions between balls. In a one-dimensional setting, the chart only models collisions between successive balls with a single for loop.

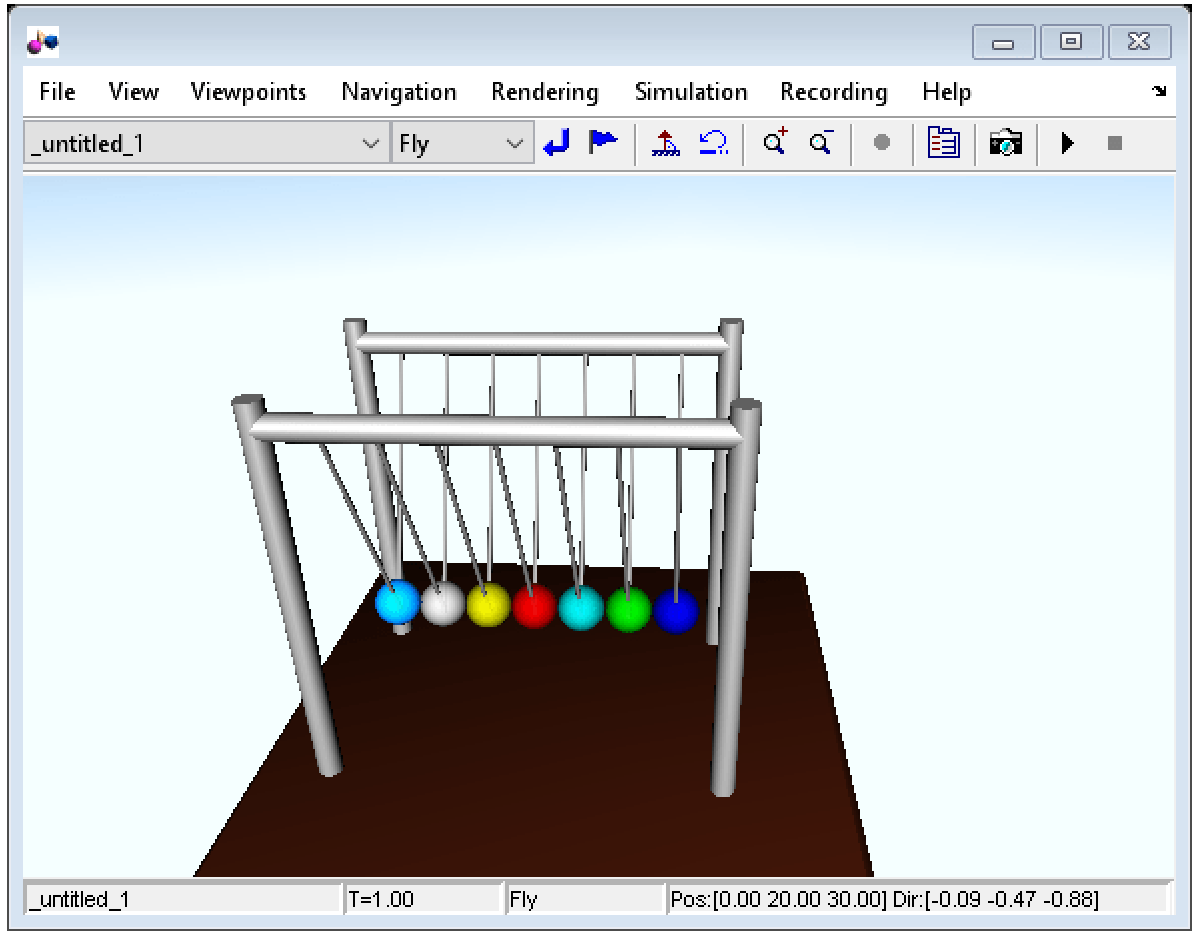


The response to a collision is also expressed simply. Each collision is treated as a perfectly elastic instantaneous collision. The position and velocity are exchanged for each of the balls involved in the collision.



Simulating this model creates a Simulink® 3D Animation™ which shows the motion of the balls. Double-click on any ball to start the simulation.





See Also

More About

- "Continuous-Time Modeling in Stateflow" on page 22-2
- "Reuse Logic Patterns by Defining Graphical Functions" on page 6-9
- "Model Newton's Cradle" on page 22-19
- "Simulink 3D Animation"

23

Fixed-Point Data in Stateflow Charts

- “Fixed-Point Data in Stateflow Charts” on page 23-2
- “Operations for Fixed-Point Data in Stateflow” on page 23-8
- “Build a Low-Pass Filter by Using Fixed-Point Data” on page 23-15
- “Fixed-Point Operations in Stateflow Charts” on page 23-19
- “Compare Fixed-Point and Floating-Point Computation in Mandelbrot Set” on page 23-25

Fixed-Point Data in Stateflow Charts

Fixed-point numbers use integers and integer arithmetic to approximate real numbers. They are an efficient means for performing computations involving real numbers without requiring floating-point support in underlying system hardware.

Fixed-Point Numbers

Fixed-point numbers represent real numbers by using the encoding scheme:

$$V \approx V_{approx} = SQ + B.$$

- V is a precise real-world value that you want to approximate with a fixed-point number.
- V_{approx} is the approximate real-world value that results from the fixed-point representation.
- Q is an integer that encodes the fixed-point number. This value is called the *quantized integer.*
- S is a coefficient that determines the precision of the fixed-point representation. This value is called the *slope*.
- B is an additive correction called the *bias*.

The quantized integer Q is the only part of the fixed-point representation that varies in value. In the generated code, the quantities S and B are constant and appear only as literal numbers or expressions. If a fixed-point number changes, its quantized integer Q changes but S and B remain unchanged.

To determine the quantized integer Q corresponding to a real-world value V, round the quantity $(V - B)/S$ to an integer. For example, to represent the number $V = 15.345$ in a fixed-point type with slope $S = 0.5$ and bias $B = 0.1$, you use the quantized integer

$$Q = \text{round}((V - B)/S) = \text{round}((15.345 - 0.1)/0.5) = \text{round}(30.49) = 30.$$

Because you round Q to an integer, you lose some precision in representing the number 15.345. The number that Q actually represents is

$$V_{approx} = SQ + B = 0.5 \times 30 + 0.1 = 15.1.$$

Using fixed-point numbers to represent real numbers with integers involves the loss of some precision. However, with a suitable choice of S and B, you can minimize this loss to acceptable levels. For instance, by changing the coding scheme to use $S = 0.25$ and $B = 0.1$, you can represent the number $V = 15.345$ with greater precision as:

$$Q = \text{round}((V - B)/S) = \text{round}((15.345 - 0.1)/0.25) = \text{round}(60.98) = 61$$

$$V_{approx} = SQ + B = 0.25 \times 61 + 0.1 = 15.35.$$

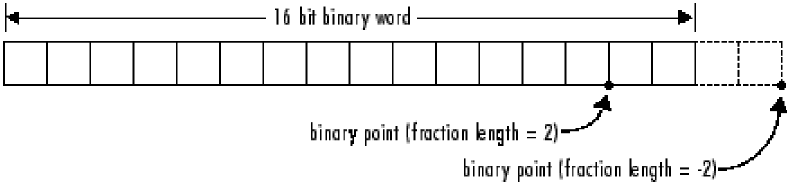
The difference between V_{approx} and V is always less than the slope S.

Specify Fixed-Point Data

In the Model Explorer, you can specify the fixed-point encoding for a data object by using the Data Type Assistant, as described in “Fixed-Point Data Properties” on page 10-10. Set the **Mode** field to `Fixed point` and specify these properties:

- **Signedness**: Choose `Signed` or `Unsigned`.
- **Word length**: Specify the bit size of the word that holds the quantized integer Q.
- **Scaling**: Choose `Binary point` or `Slope and bias`.

- If you select `Binary point`, the Data Type Assistant displays the **Fraction length** field, which specifies the binary point location. Choosing a **Fraction length** of n defines a fixed-point encoding with a slope of $S = 2^{-n}$ and a bias of $B = 0$.



- If you select `Slope and bias`, the Data Type Assistant displays fields for entering the **Slope** S and **Bias** B for the fixed-point encoding scheme.

Alternatively, you can specify the encoding for a fixed-point data object directly by using the `fixdt` function. In the **Property Inspector** or the Model Explorer, in the **Type** field, enter an expression in one of these formats:

```
fixdt(Signed, WordLength, FractionLength)
```

```
fixdt(Signed, WordLength, Slope, Bias)
```

Tip: Some encoding schemes are computationally expensive, particularly in multiplication and division operations. Selecting a slope that is an integer power of two and a zero bias avoids these computationally expensive instructions. Using binary point scaling is recommended.

Conversion Operations

Stateflow charts convert real numbers into fixed-point numbers during data initialization and as part of casting operations in the application. These conversions compute a quantized integer Q from a real number input. The type of conversion depends on the action language for the chart.

Conversion in Charts That Use MATLAB as the Action Language

In a chart that uses MATLAB as the action language, you define the method for all conversions through the fixed-point properties for the chart. See “Fixed-Point Properties” on page 1-23.

For example, if you set the **MATLAB Chart `fimath`** property to `Same as MATLAB`, then the chart rounds the resulting quantized integer to its nearest integer value.

Conversions in Charts That Use C as the Action Language

Charts that use C as the action language employ two methods for converting fixed-point data:

- Offline conversions initialize data during code generation. Offline conversions are designed to maximize accuracy. These conversions round the resulting quantized integer to its nearest integer value. Offline conversions are performed for initialization of data (variables and constants) in the Stateflow hierarchy and from the MATLAB workspace.
- Online conversions perform casting operations during run time. Online conversions are designed to maximize computational efficiency. They are faster and more efficient, but less precise than offline conversions. Instead of rounding Q to its nearest integer, online conversions round to the floor (except for division, which can round to 0, depending on the C compiler).

For example, this table illustrates the difference between offline and online conversions of real numbers to fixed-point numbers defined with a slope of $S = 2^{-4}$ and a bias of $B = 0$. For each real-world value V, the chart computes a quantized integer Q by rounding $(V\text{-}B)/S$ to the nearest integer (in offline conversion) or to the floor (in online conversion). For each conversion, $V_{approx} = QS + B$ is the approximate real-world value resulting from Q.

Real-World Value		Offline Conversion		Online Conversion	
V	**(*V-B*)/*S***	***Q***	V_{approx}	***Q***	V_{approx}
15.345	245.52	246	15.375	245	15.3125
3.45	55.2	55	3.4375	55	3.4375
1.0375	16.6	17	1.0625	16	1
2.06	32.96	33	2.0625	32	2

Fixed-Point Context-Sensitive Constants

In charts that use C as the action language, you can avoid explicit type casts by using fixed-point context-sensitive constants. These constants infer their type from the context in which they occur. They are written like ordinary numbers with the suffix `C` or `c`. For example, `4.3C` and `123.4c` are valid fixed-point context-sensitive constants that you can use in action statements.

Although fixed-point context-sensitive constants can appear in expressions with any data types (including integers and floating-point data), their main use is with fixed-point numbers. The algorithm that interprets the context-sensitive constant computes a type that provides maximum accuracy without overflow. The algorithm depends on:

- The operations in the expression
- The other data types in the context
- The value of the constant

Fixed-point context-sensitive constants infer their type according to these rules:

- In a casting operation, the constant has the type to which it is being cast.
- In a simple assignment operation of the form `a = b`:
  - If `b` is a context-sensitive constant, it has the same type as `a`.
  - If `b` is an addition or subtraction operation, then the constant has the same type as the other operand.
  - If `b` is a multiplication or division operation with a fixed-point operand, then the constant has the type that provides the best possible precision for a fixed-point result, as determined by the `fixptbestexp` function.
  - If `b` is a multiplication or division operation with a floating-point operand of type `double` or `single`, then the constant has the same type as the floating-point operand.
- In a special assignment operation of the form `a := b`:
  - If `b` is a context-sensitive constant, it has the same type as `a`.
  - If `b` is an arithmetic operation with a floating-point operand of type `double` or `single`, or if `a` is a floating-point data object, then the constant has the same type as the floating-point number.

  - If `b` is an addition or subtraction operation with a fixed-point operand and `a` is a fixed-point data object, then the constant has the same type as `a`.
  - If `b` is a multiplication or division operation with a fixed-point operand and `a` is a fixed-point data object, then the constant has the type that provides the best possible precision for a fixed-point result.
- As an argument in a function call, the constant has the same type as the formal argument.

You cannot use context-sensitive constants as both operands of a binary operation or as the leftmost operand of an assignment operation.

Tips for Using Fixed-Point Data

- Develop and test your application by using double- or single-precision floating-point numbers. Using double- or single-precision floating-point numbers does not limit the range or precision of your computations. Once your application works as designed, you can start substituting fixed-point data for double-precision data.
- Open the Configuration Parameters dialog box and, in the **Hardware Implementation** pane, set the integer word size for the simulation environment to the integer size of the intended target environment. Code generated by Stateflow uses this integer size to select result types for your fixed-point operations. See "Hardware Implementation Pane" (Simulink).
- When you simulate your model, use overflow detection to warn you when the result of a fixed-point operation exceeds the numeric capacity of its fixed-point type. Open the Configuration Parameters dialog box and, in the **Diagnostics** > **Data Validity** pane, set the **Wrap on overflow** and **Saturate on overflow** parameters to `error` or `warning`. If you encounter overflow errors in fixed-point data, increase the range of your data by:

  - Increasing the **Word length** value for the overflowing fixed-point data. For example, change the number of bits used to encode the fixed-point data from 16 to 32. This action changes the base integer type for *Q* from `int16` to `int32`.
  - Decreasing the **Fraction length** value (if using `Binary point` scaling) or increasing the **Slope** value (if using `Slope and bias` scaling). For example, decrease the **Fraction length** value from 4 to 1 (or, equivalently, increase the **Slope** value from $S = 2^{-4} = 0.0625$ to $S = 2^{-1} = 0.5$). This action increases the range of your fixed-point data but decreases the available precision.

  For more information, see "Detect Data Range Violations" on page 30-36.
- If you encounter issues with model behavior stemming from inadequate precision in your fixed-point data, increase the precision of your data by increasing the **Fraction length** value (if using `Binary point` scaling) or decreasing the **Slope** value (if using `Slope and bias` scaling). For example, increase the **Fraction length** value from 2 to 3 (or, equivalently, decrease the **Slope** value from $S = 2^{-2} = 0.25$ to $S = 2^{-3} = 0.125$). This action increases the precision of your fixed-point data but decreases the available range.
- In charts that use C as the action language, you can use a special assignment operation `:=` and context-sensitive constants to maintain as much precision as possible. See "Override Fixed-Point Promotion in C Charts" on page 23-11 and "Fixed-Point Context-Sensitive Constants" on page 23-4.

Note If you do not use context-sensitive constants with fixed-point types, noninteger numeric constants (constants that have a decimal point) can force fixed-point operations to produce floating-point results.

Automatic Scaling of Fixed-Point Data

Automatic scaling tools can change the settings of Stateflow fixed-point data. You can prevent automatic scaling by selecting the **Lock data type setting against changes by the fixed-point tools** option for the fixed-point data object. See “Lock data type against Fixed-Point tools” on page 10-8. For methods on autoscaling fixed-point data, see “Choosing a Range Collection Method” (Fixed-Point Designer).

Share Fixed-Point Data with Simulink Models

To share fixed-point data with Simulink models:

- Use the same property values to specify the data in the Stateflow chart and in the Simulink model. For an example of this method of sharing input data from a Simulink model, see “Model Bang-Bang Temperature Control System” on page 14-51.

  For some Simulink blocks, you can specify the type of input or output data directly. For example, you can specify the fixed-point data type for a Constant block directly in the **Output data type** field by using the `fixdt` function.
- Define the data as **Input** or **Output** in the Stateflow chart and instruct the sending or receiving block in the Simulink model to inherit its type from the chart data. In many blocks, you can set data types through inheritance from the driving block, or through back propagation from the next block. You can set the data type of a Simulink block to match the data type of the Stateflow port to which it connects.

  For example, you can set the Constant block to inherit its type from the Stateflow **Input to Simulink** port that it supplies. Set the **Output data type** block parameter to `Inherit via back propagation`.

Implementation of Fixed-Point Data in Stateflow

Stateflow charts define fixed-point data types from the values that you specify for S, B, and the base integer type for Q.

- For each fixed-point data, the chart defines an integer variable for Q in the generated code. This integer is the only part of a fixed-point number that changes in value. The available base types for Q are the unsigned integer types `uint8`, `uint16`, and `uint32`, and the signed integer types `int8`, `int16`, and `int32`. If a fixed-point number has a slope $S = 1$ and a bias $B = 0$, it is equivalent to its quantized integer Q and behaves exactly as its base integer type.
- The slope S is factored into a coefficient F with $1 \leq F < 2$ and an integer power of two with exponent E:
  $S = F \cdot 2^{E}$.
  If the fractional slope F is greater than 1, it is converted into a fixed-point number. Encoding schemes with $F > 1$ can be computationally expensive, particularly in multiplication and division operations. Setting $F = 1$ avoids these computationally expensive instructions. In this setting, scaling by a power of 2 is implemented as bit shifts, which are more efficient than multiply instructions. Therefore, using binary-point-only scaling, in which $F = 1$ and $B = 0$, is recommended.
- Operations for fixed-point types are implemented with solutions for the quantized integer as described in “Arithmetic Operations for Fixed-Point Data” on page 23-19. To generate efficient code, the fixed-point promotion rules choose values for slope and bias that cancel difficult terms in the solutions. See “Promotion Rules for Fixed-Point Operations” on page 23-20.

See Also

`fixdt` | `fixptbestexp`

More About

- "Fixed-Point Operations in Stateflow Charts" on page 23-19
- "Operations for Fixed-Point Data in Stateflow" on page 23-8
- "Build a Low-Pass Filter by Using Fixed-Point Data" on page 23-15
- "Set Data Properties" on page 10-5

Operations for Fixed-Point Data in Stateflow

Stateflow charts in Simulink models have an action language property that defines the syntax that you use to compute with fixed-point data:

- MATLAB as the action language.
- C as the action language.

For more information, see “Differences Between MATLAB and C as Action Language Syntax” on page 15-4.

Binary Operations

This table summarizes the interpretation of all binary operations on fixed-point operands according to their order of precedence (0 = highest, 9 = lowest). Binary operations are left associative so that, in any expression, operators with the same precedence are evaluated from left to right.

Operation	Precedence	MATLAB as the Action Language	C as the Action Language
`a ^ b`	0	Power. Not supported for fixed-point operands defined by using either a slope that is not an integer power of two or a nonzero bias. Exponent operand must be a constant whose value is a non-negative integer.	Power. Enable this operation by clearing the **Enable C-bit operations** chart property. See “Enable C-bit operations” on page 1-21.
`a * b`	1	Multiplication. For fixed-point operands defined by using either a slope that is not an integer power of two or a nonzero bias, specify a chart `fimath` object with `ProductMode` set to `SpecifyPrecision`. See “Multiplication” on page 23-22.	Multiplication. Not supported for fixed-point operands defined by using a nonzero bias. See “Multiplication” on page 23-22.
`a / b`	1	Division. Not supported for fixed-point operands defined by using either a slope that is not an integer power of two or a nonzero bias. See “Division” on page 23-22.	Division. Not supported for fixed-point operands defined by using a nonzero bias. See “Division” on page 23-22.
`a + b`	2	Addition. For fixed-point operands defined by using either a slope that is not an integer power of two or a nonzero bias, specify a chart `fimath` object with `SumMode` set to `SpecifyPrecision`. See “Addition and Subtraction” on page 23-21.	Addition. See “Addition and Subtraction” on page 23-21.

Operation	Precedence	MATLAB as the Action Language	C as the Action Language
`a - b`	2	Subtraction. For fixed-point operands defined by using either a slope that is not an integer power of two or a nonzero bias, specify a chart `fimath` object with `SumMode` set to `SpecifyPrecision`. See “Addition and Subtraction” on page 23-21.	Subtraction. See “Addition and Subtraction” on page 23-21.
`a > b`	3	Comparison, greater than. See “Relational Operations for Fixed-Point Data” on page 23-19.	Comparison, greater than. Not supported for fixed-point operands with mismatched biases. See “Relational Operations for Fixed-Point Data” on page 23-19.
`a < b`	3	Comparison, less than. See “Relational Operations for Fixed-Point Data” on page 23-19.	Comparison, less than. Not supported for fixed-point operands with mismatched biases. See “Relational Operations for Fixed-Point Data” on page 23-19.
`a >= b`	3	Comparison, greater than or equal to. See “Relational Operations for Fixed-Point Data” on page 23-19.	Comparison, greater than or equal to. Not supported for fixed-point operands with mismatched biases. See “Relational Operations for Fixed-Point Data” on page 23-19.
`a <= b`	3	Comparison, less than or equal to. See “Relational Operations for Fixed-Point Data” on page 23-19.	Comparison, less than or equal to. Not supported for fixed-point operands with mismatched biases. See “Relational Operations for Fixed-Point Data” on page 23-19.
`a == b`	4	Comparison, equal to. See “Relational Operations for Fixed-Point Data” on page 23-19.	Comparison, equal to. Not supported for fixed-point operands with mismatched biases. See “Relational Operations for Fixed-Point Data” on page 23-19.
`a ~= b`	4	Comparison, not equal to. See “Relational Operations for Fixed-Point Data” on page 23-19.	Comparison, not equal to. Not supported for fixed-point operands with mismatched biases. See “Relational Operations for Fixed-Point Data” on page 23-19.
`a != b`	4	Not supported. Use the operation `a ~= b`. See “Relational Operations for Fixed-Point Data” on page 23-19.	Comparison, not equal to. Not supported for fixed-point operands with mismatched biases. See “Relational Operations for Fixed-Point Data” on page 23-19.

Operation	Precedence	MATLAB as the Action Language	C as the Action Language
a <> b	4	Not supported. Use the operation a ~= b. See “Relational Operations for Fixed-Point Data” on page 23-19.	Comparison, not equal to. Not supported for fixed-point operands with mismatched biases. See “Relational Operations for Fixed-Point Data” on page 23-19.
a && b	8	Logical AND. See “Logical Operations for Fixed-Point Data” on page 23-20.	Logical AND. See “Logical Operations for Fixed-Point Data” on page 23-20.
a || b	9	Logical OR. See “Logical Operations for Fixed-Point Data” on page 23-20.	Logical OR. See “Logical Operations for Fixed-Point Data” on page 23-20.

Unary Operations and Actions

This table summarizes the interpretation of all unary operations and actions on fixed-point operands. Unary operations:

- Have higher precedence than binary operators.
- Are right associative so that, in any expression, they are evaluated from right to left.

Operation	MATLAB as the Action Language	C as the Action Language
~a	Not supported. Use the expression `a == cast(0,"like",a)`. See “Logical Operations for Fixed-Point Data” on page 23-20.	Logical NOT. Enable this operation by clearing the **Enable C-bit operations** chart property. See “Logical Operations for Fixed-Point Data” on page 23-20 and “Enable C-bit operations” on page 1-21.
!a	Not supported. Use the expression `a == cast(0,"like",a)`. See “Logical Operations for Fixed-Point Data” on page 23-20.	Logical NOT. See “Logical Operations for Fixed-Point Data” on page 23-20.
-a	Negative. See “Unary Minus” on page 23-23.	Negative. See “Unary Minus” on page 23-23.
a++	Not supported. Use the expression `a = a+1`.	Increment. Equivalent to `a = a+1`.
a--	Not supported. Use the expression `a = a-1`.	Decrement. Equivalent to `a = a-1`.

Assignment Operations

This table summarizes the interpretation of assignment operations on fixed-point operands.

Operation	MATLAB as the Action Language	C as the Action Language
a = b	Simple assignment.	Simple assignment.

Operation	MATLAB as the Action Language	C as the Action Language
a := b	Not supported. To override fixed-point promotion rules, use explicit type cast operations. See "Type Cast Operations" on page 14-7.	Special assignment that overrides fixed-point promotion rules. See "Override Fixed-Point Promotion in C Charts" on page 23-11.
a += b	Not supported. Use the expression a = a +b.	Equivalent to a = a+b.
a -= b	Not supported. Use the expression a = a-b.	Equivalent to a = a-b.
a *= b	Not supported. Use the expression a = a*b.	Equivalent to a = a*b.
a /= b	Not supported. Use the expression a = a/b.	Equivalent to a = a/b.

Override Fixed-Point Promotion in C Charts

In charts that use C as the action language, a simple assignment of the form a = b calculates an intermediate value for b according to the fixed-point promotion rules. Then this intermediate value is cast to the type of a by using an online conversion. See "Promotion Rules for Fixed-Point Operations" on page 23-20 and "Conversion Operations" on page 23-3. Simple assignments are most efficient when both types have equal bias and slopes that either are equal or are both powers of two.

In contrast, a special assignment of the form a := b overrides this behavior by initially using the type of a as the result type for the value of b.

- Constants in b are converted to the type of a by using offline conversions.
- The expression b can contain at most one arithmetic operator (+, -, *, or /). The result is determined by using an online conversion.
- If b contains anything other than an arithmetic operation or a constant, then the special assignment operation behaves like the simple assignment operation (=).

Use the special assignment operation := when you want to:

- Avoid an overflow in an arithmetic operation. For example, see "Avoid Overflow in Fixed-Point Addition" on page 23-11.
- Retain precision in a multiplication or division operation. For example, see "Improve Precision in Fixed-Point Division" on page 23-12.

Note Using the special assignment operation := can result in generated code that is less efficient than the code you generate by using the normal fixed-point promotion rules.

Avoid Overflow in Fixed-Point Addition

You can use the special assignment operation := to avoid overflow when performing an arithmetic operation on two fixed-point numbers. For example, consider a chart that computes the sum a+b where a = 2^{12}-1 = 4095 and b = 1.

Suppose that:

- Both inputs are signed 16-bit fixed-point numbers with three fraction bits (type `fixdt(1,16,3)`).
- The output `c` is a signed 32-bit fixed-point number with three fraction bits (type `fixdt(1,32,3)`).
- The integer word size for production targets is 16 bits.

Because the target integer size is 16 bits, the simple assignment `c = a+b` adds the inputs in 16 bits before casting the sum to 32 bits. The intermediate result is 4096, which, as a type `fixdt(1,16,3)` value, results in an overflow.

In contrast, the special assignment `c := a+b` casts the inputs to 32 bits before computing the sum. The result of 4096 is safely computed as a type `fixdt(1,32,3)` value without an overflow.

Improve Precision in Fixed-Point Division

You can use the special assignment operation `:=` to obtain a more precise result when multiplying or dividing two fixed-point numbers. For example, consider a chart that computes the ratio `a/b` where `a` = 2 and `b` = 3.

Suppose that:

- The input `a` is a fixed-point number with four fraction bits (type `fixdt(1,16,4)`).
- The input `b` is a fixed-point number with three fraction bits (type `fixdt(1,16,3)`).
- The output `c` is a signed 16-bit fixed-point number with six fraction bits (type `fixdt(1,16,6)`).

The inputs correspond to these slopes and quantized integers:

$S_a = 2^{-4}$, $Q_a = 32$

$S_b = 2^{-3}$, $Q_b = 24$.

The simple assignment `c = a/b` first calculates an intermediate value for `a/b` according to the fixed-point promotion rules. The quantized integer is rounded to the floor:

$S_{int} = S_a/S_b = 2^{-4}/2^{-3} = 2^{-1}$

$Q_{int} = Q_a/Q_b = 32/24 \approx 1$.

The intermediate result is then cast as a signed 16-digit fixed-point number with six fraction bits:

$S_c = 2^{-6} = 1/64$

$Q_c = S_{int}Q_{int}/S_c = 2^{-1}/2^{-6} = 2^5 = 32$.

Therefore, the approximate real-world value for `c` is $V_c \approx S_cQ_c = 32/64 = 0.5$. This result is not a good approximation of the actual value of 2/3.

In contrast, the special assignment `c := a/b` calculates `a/b` directly as a signed 16-digit fixed-point number with six fraction bits. Again, the quantized integer is rounded to the floor:

$S_c = 2^{-6} = 1/64$

$Q_c = (S_aQ_a)/(S_cS_bQ_b) = 128/3 \approx 42$.

Therefore, the approximate real-world value for `c` is $V_c \approx S_cQ_c = 42/64 = 0.6563$. This result is a better approximation to the actual value of 2/3.

Compare Results of Fixed-Point Arithmetic

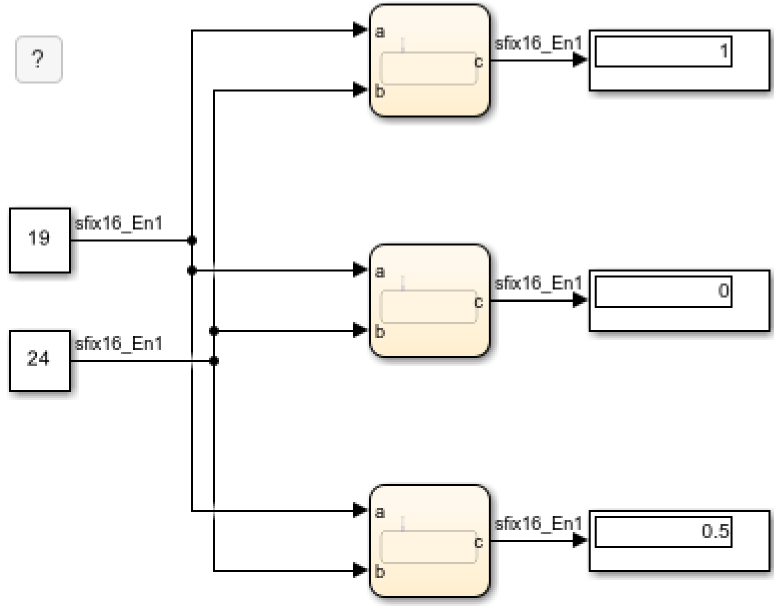
This example shows the difference between various implementations of fixed-point arithmetic in Stateflow charts. The model contains three charts that calculate the ratio `a/b` where `a` = 19 and `b` = 24. Both inputs are signed 16-digit fixed-point numbers with one fraction bit (type `fixdt(1,16,1)`). They correspond to these slopes and quantized integers:

$$S_a = 2^{-1}, Q_a = 38$$

$$S_{\mathtt{b}} = 2^{-1}, Q_{\mathtt{b}} = 48$$

The model calculates the value of `a/b` as a floating-point number of type `fixdt(1,16,1)` in three different ways:

- A type casting operation in a chart that uses MATLAB as the action language.
- A simple assignment operation in a chart that uses C as the action language.
- A special assignment operation in a chart that uses C as the action language.



Type Casting in Chart That Uses MATLAB as the Action Language

The chart at the top of the model computes an intermediate value for `a/b`. The quantized integer for the intermediate value is rounded to the nearest integer:

$$S_{\mathtt{int}} = S_{\mathtt{a}}/S_{\mathtt{b}} = 1$$

$$Q_{\mathtt{int}} = Q_{\mathtt{a}}/Q_{\mathtt{b}} = 38/48 \approx 1.$$

The intermediate value is then cast as a signed 16-digit fixed-point number `c` with one fraction bit:

$$S_{\mathtt{c}} = 2^{-1}$$

$$Q_{\mathtt{c}} = S_{\mathtt{int}} \cdot Q_{\mathtt{int}}/S_{\mathtt{c}} = 2.$$

The output value from this chart is

$$\tilde{V}_{\mathtt{c}} = S_{\mathtt{c}} \cdot Q_{\mathtt{c}} = 1.$$

Simple Assignment in Chart That Uses C as the Action Language

The middle chart also computes an intermediate value for `a/b`. In this case, the quantized integer for the intermediate value is rounded to the floor:

$$S_{\texttt{int}} = S_{\texttt{a}}/S_{\texttt{b}} = 1$$

$$Q_{\texttt{int}} = Q_{\texttt{a}}/Q_{\texttt{b}} = 38/48 \approx 0.$$

The intermediate value is then cast as a signed 16-digit fixed-point number `c` with one fraction bit:

$$S_{\texttt{c}} = 2^{-1}$$

$$Q_{\texttt{c}} = S_{\texttt{int}} \cdot Q_{\texttt{int}}/S_{\texttt{c}} = 0.$$

The output value from this chart is

$$\tilde{V}_{\texttt{c}} = S_{\texttt{c}} \cdot Q_{\texttt{c}} = 0.$$

Special Assignment in Chart That Uses C as the Action Language

The chart at the bottom of the model uses a special assignment of the form `c := a/b`. The value of the division is calculated directly as a signed 16-digit fixed-point number with one fraction bit. The quantized integer is rounded to the floor:

$$S_{\texttt{c}} = 2^{-1}$$

$$Q_{\texttt{c}} = (S_{\texttt{a}} \cdot Q_{\texttt{a}})/(S_{\texttt{c}} \cdot S_{\texttt{b}} \cdot Q_{\texttt{b}}) = 19/12 \approx 1.$$

Therefore, the output value from this chart is

$$\tilde{V}_{\texttt{c}} = S_{\texttt{c}} \cdot Q_{\texttt{c}} = 0.5.$$

The three results exhibit loss of precision compared to the floating-point answer of 19/24 = 0.7917. To minimize the loss of precision to an acceptable level in your application, adjust the encoding scheme in your fixed-point data.

See Also

More About

- “Fixed-Point Data in Stateflow Charts” on page 23-2
- “Fixed-Point Operations in Stateflow Charts” on page 23-19
- “Build a Low-Pass Filter by Using Fixed-Point Data” on page 23-15

Build a Low-Pass Filter by Using Fixed-Point Data

This example shows how to build a Stateflow® chart that uses fixed-point data to implement a low-pass Butterworth filter. By designing the filter with fixed-point data instead of floating-point data, you can simulate your model using less memory. For more information, see “Fixed-Point Data in Stateflow Charts” on page 23-2.

```
{
y = x*b0 + x_n1*b1 - y_n1*a1;
x_n1 = x;
y_n1 = y;
}
```

Build the Fixed-Point Butterworth Filter

The Low-Pass Filter chart is a stateless flow chart that accepts one input and provides one output. The chart contains these data symbols:

- `x` — **Scope**: `Input`, **Type**: `Inherit:Same as Simulink`
- `y` — **Scope**: `Output`, **Type**: `fixdt(1,16,10)`
- `x_n1` — **Scope**: `Local`, **Type**: `fixdt(1,16,12)`
- `y_n1` — **Scope**: `Local`, **Type**: `fixdt(1,16,10)`
- `b0` — **Scope**: `Parameter`, **Type**: `fixdt(1,16,15)`
- `b1` — **Scope**: `Parameter`, **Type**: `fixdt(1,16,15)`
- `a1` — **Scope**: `Parameter`, **Type**: `fixdt(1,16,15)`

The values of `b0`, `b1`, and `a1` are the coefficients of the low-pass Butterworth filter.

To build the Low-Pass Filter chart:

1. Create a Simulink® model with an empty Stateflow chart by entering `sfnew` at the MATLAB® command prompt.
2. In the Stateflow chart, add a flow chart with a single branch that assigns values to `y`, `x_n1`, and `y_n1`.
3. Add input, output, local, and parameter data to the chart, as described in “Add Stateflow Data” on page 10-2.

Define the Model Callback Function

Before loading the model, MATLAB calls the `butter` (Signal Processing Toolbox) function to compute the values for the parameters `b0`, `b1`, and `a1`. The function constructs a first-order low-pass Butterworth filter with a normalized cutoff frequency of `(2*pi*Fc/(Fs/2))` radians per second, where:

- The sampling frequency is `Fs` = 1000 Hz.

- The cutoff frequency is Fc = 50 Hz.

The function output B contains the numerator coefficients of the filter in descending powers of z. The function output A contains the denominator coefficients of the filter in descending powers of z.

```
Fs = 1000;
Fc = 50;
[B,A] = butter(1,2*pi*Fc/(Fs/2));
b0 = B(1);
b1 = B(2);
a1 = A(2);
```

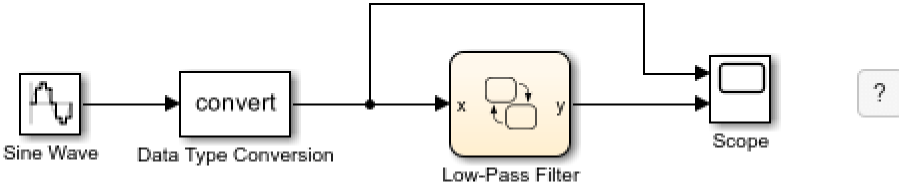
To define the preload callback for the model:

1. In the **Modeling** tab, under **Setup**, select **Model Settings** > **Model Properties**.
2. In the Model Properties dialog box, on the **Callbacks** tab, select **PreLoadFcn**.
3. Enter the MATLAB code for the preload function call.
4. Click **OK**.

To load the parameter values to the MATLAB workspace, save, close, and reopen the model.

Add Other Blocks to the Model

To complete the model, add a Sine Wave (Simulink) block, a Data Type Conversion (Simulink) block, and a Scope (Simulink) block. Connect and label the blocks according to this diagram.



Sine Wave block

The Sine Wave block outputs a floating-point signal. The block has these settings:

- **Sine type**: `Time based`
- **Time**: `Use simulation time`
- **Amplitude**: `1`
- **Bias**: `0`
- **Frequency**: `2*pi*Fc`
- **Phase**: `0`
- **Sample time**: `1/Fs`
- **Interpret vector parameters as 1-D**: `On`

Data Type Conversion block

The Data Type Conversion block converts the floating-point signal from the Sine Wave block to a fixed-point signal. By converting the signal to a fixed-point type, you can simulate your model using less memory. The block has these settings:

- **Output minimum**: `[]`
- **Output maximum**: `[]`
- **Output data type**: `fixdt(1,16,14)`
- **Lock output data type setting against changes by the fixed-point tools**: `Off`
- **Input and output to have equal**: `Real World Value (RWV)`
- **Integer rounding mode**: `Floor`
- **Saturate on integer overflow**: `Off`
- **Sample time**: `-1`

Scope block

The Scope block has two input ports that connect to the input and output signals for the Low-Pass Filter chart. To display the two signals separately, select a scope layout with two rows and one column.

Set Model Configuration Parameters

Because none of the blocks in the model have a continuous sample time, use a discrete solver with these configuration parameters:

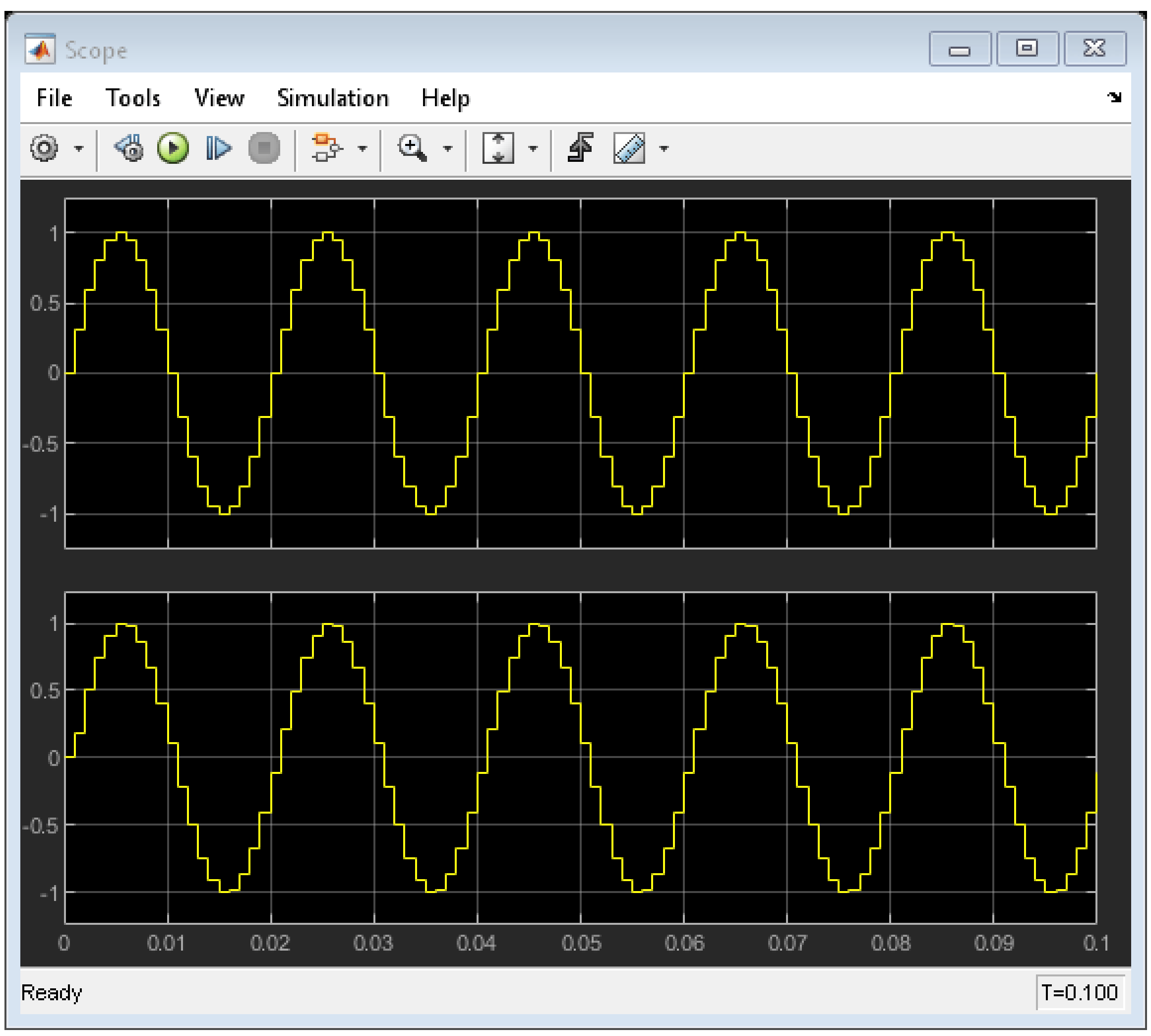
- **Stop time**: `0.1`
- **Type**: `Fixed-step`
- **Solver**: `discrete (no continuous states)`
- **Fixed-step size (fundamental sample time)**: `1/Fs`

To configure the model:

1. In the **Modeling** tab, under **Setup**, select **Model Settings**.
2. In the **Solver** pane, set the discrete solver parameters.
3. Click **OK**.

Run the Model

When you simulate the model, the Scope block displays two signals. The top signal shows the fixed-point version of the sine wave input to the chart. The bottom signal corresponds to the filtered output from the chart. The filter removes high-frequency values from the signal but allows low-frequency values to pass through the chart unchanged.



See Also

`sfnew` | `butter` | Sine Wave | Data Type Conversion | Scope

More About

- “Fixed-Point Data in Stateflow Charts” on page 23-2
- “Fixed-Point Operations in Stateflow Charts” on page 23-19
- “Operations for Fixed-Point Data in Stateflow” on page 23-8

Fixed-Point Operations in Stateflow Charts

Fixed-point numbers use integers and integer arithmetic to approximate real numbers. They enable you to perform computations involving real numbers without requiring floating-point support in underlying system hardware. For more information, see "Fixed-Point Data in Stateflow Charts" on page 23-2.

Arithmetic Operations for Fixed-Point Data

The general equation for a binary arithmetic operation between fixed-point operands is

```
c = a <op> b
```

where `a` and `b` are fixed-point numbers and `<op>` refers to addition, subtraction, multiplication, or division. The result of the operation is a fixed-point number `c` of the form

$$V_c \approx S_c Q_c + B_c.$$

The fixed-point type for `c` determines the slope S_c, the bias B_c, and the number of bits used to store the quantized integer Q_c. For each arithmetic operation, this table lists the value of the quantized integer Q_c in terms of the values of the operands (`a` and `b`) and the fixed-point type for `c`.

Operation		Slope and Bias Scaling	Binary-Point Scaling
Addition	`c = a+b`	Q_c = round($(S_a/S_c)Q_a + (S_b/S_c)Q_b + (B_a + B_b - B_c)/S_c$)	Q_c = round($(S_a/S_c)Q_a + (S_b/S_c)Q_b$)
Subtraction	`c = a-b`	Q_c = round($(S_a/S_c)Q_a - (S_b/S_c)Q_b - (B_a - B_b - B_c)/S_c$)	Q_c = round($(S_a/S_c)Q_a - (S_b/S_c)Q_b$)
Multiplicati on	`c = a*b`	Q_c = round($(S_aS_b/S_c)Q_aQ_b + (B_aS_b/S_c)Q_a + (B_bS_a/S_c)Q_b + (B_aB_b - B_c)/S_c$)	Q_c = round($(S_aS_b/S_c)Q_aQ_b$)
Division	`c = a/b`	Q_c = round($(S_aQ_a + B_a)/(S_c(S_bQ_b + B_b)) - (B_c/S_c)$)	Q_c = round($(S_a/(S_bS_c))Q_a/Q_b$)

To simplify these expressions and avoid computationally expensive operations, use binary-point scaling to encode all your fixed-point data. With this setting, the slope is an integer power of two and the bias is zero. Then, all fixed-point operations consist of integer arithmetic and bit shifts on the quantized integers.

Note The result of an arithmetic operation depends on the rounding method that computes the value of the quantized integer Q_c. For more information, see "Conversion Operations" on page 23-3.

Relational Operations for Fixed-Point Data

You can use relational operations to compare fixed-point data. Comparing fixed-point values of different types can yield unexpected results because each operand is converted to a common type for comparison. Because of rounding or overflow errors during the conversion, some values that appear to be equal are not equal as fixed-point numbers.

For example, suppose that `a` and `b` represent the real-world value V = 2.2 in two different fixed-point encoding schemes:

- `a` is a fixed-point number with slope S_a = 0.3 and bias B_a = 0.1. The quantized integer for `a` is:

$Q_a = (V - B_a)/S_a = (2.2 - 0.1)/0.3 = 7.$

- b is a fixed-point number with slope $S_b = 0.7$ and bias $B_b = 0.1$. The quantized integer for b is:
  $Q_b = (V - B_b)/S_b = (2.2 - 0.1)/0.7 = 3.$

To compare these values, the chart first converts them to a common fixed-point type with slope $S_{comp} = 1.06811 \cdot 10^{-5} \approx S_a/28087 \approx S_b \cdot 2^{-16}$ and bias $B_{comp} = 0.1$. (In this case, the slope S_{comp} arises from the approximate value of $S_a/S_b = 0.3/0.7 \approx 28087 \cdot 2^{-16}$.) In this common encoding scheme, a and b correspond to these quantized integers:
$Q_{a'} = S_a Q_a/S_{comp} = Q_a(S_a/S_{comp}) \approx 7 \cdot 28087 = 196609$
$Q_{b'} = S_b Q_b/S_{comp} = Q_b(S_b/S_{comp}) \approx 3 \cdot 2^{16} = 196608.$
After the conversion, the quantized integers are different. Although a and b represent the same real-world value, they are not equal as fixed-point numbers.

Note In charts that use C as the action language, comparisons of fixed-point operands with mismatched biases are not supported.

Logical Operations for Fixed-Point Data

In a logical operation, a fixed-point operand a is interpreted as `false` if it corresponds to the real-world value for zero in the fixed-point type of a. Otherwise, a is interpreted as `true`.

- In charts that use MATLAB as the action language, using a in a logical operation is equivalent to the expression `a ~= cast(0,"like",a)`.
- In charts that use C as the action language, using a in a logical operation is equivalent to the expression `a != 0c`, where `0c` is a fixed-point context-sensitive constant. See “Fixed-Point Context-Sensitive Constants” on page 23-4.

For example, suppose that a is a fixed-point number with a slope of $S_a = 0.25$ and a bias of $B_a = 5.1$. Using a in a logical operation is equivalent to testing whether the quantized integer Q_a satisfies the condition
$Q_a = \text{round}((0 - B_a)/S_a) = \text{round}(-5.1 / 0.25) = \text{round}(-20.4) = -20.$
Therefore, a is equivalent to `false` when its real-world approximation is
$V_a \approx S_a Q_a + B_a = 0.25 \cdot (-20) + 5.1 = 0.1.$

Promotion Rules for Fixed-Point Operations

The rules for selecting the numeric type used for the result of an operation on fixed-point numbers are called *fixed-point promotion rules*. These rules help to maintain computational efficiency and usability.

The fixed-point promotion rules determine a result type for an operation c = a <*op*> b by selecting the slope S_c, the bias B_c, and the number of bits w_c used to store the quantized integer Q_c. These parameters depend on the fixed-point types of the operands a and b, the operation <*op*> to be performed, and the action language property for the chart.

- In a chart that uses MATLAB as the action language, you control the fixed-point promotion rules through the fixed-point properties for the chart. See “Fixed-Point Properties” on page 1-23.
  - If you set the **MATLAB Chart `fimath`** property to `Same as MATLAB`, then arithmetic operations follow the default fixed-point promotion rules for MATLAB. See “Performing Fixed-Point Arithmetic” (Fixed-Point Designer).

    - If you specify a chart `fimath` object with `SumMode` and `ProductMode` set to `SpecifyPrecision`, then you can define the word length, slope, and bias for all sums and products explicitly. See "fimath Object Properties" (Fixed-Point Designer).
- In a chart that uses C as the action language, the fixed-point promotion rules determine the type for an intermediate value of the result. This intermediate value is then cast to the type that you specify for `c`.

  For all arithmetic operations, the default number of bits w_c used to store the quantized integer is the larger value between:

    - The maximum number of bits in the operand types (w_a and w_b).
    - The number of bits in the integer word size for the target machine (w_{int}).

  To specify the value of w_{int}, open the Configuration Parameters dialog box and, in the **Hardware Implementation** pane, set the **Device vendor** parameter to `Custom Processor` and the **int** parameter to the target integer word size. For more information, see "Hardware Implementation Pane" (Simulink).

  You can avoid overflow and improve the precision in your floating-point operations by using the special assignment operation of the form `c := a <op> b`. The special assignment operation does not follow the fixed-point promotion rules. Instead, the chart determines the result of the operation by using the type that you specify for `c`. See "Override Fixed-Point Promotion in C Charts" on page 23-11.

Addition and Subtraction

By default, charts that use MATLAB as the action language support addition and subtraction only on fixed-point data defined through binary-point scaling. If either operand is a signed fixed-point number, then the result is also signed. The choice of word length accommodates the integer and fraction parts of each operand in addition to a possible carry bit. The fraction length of the result is equal to the fraction length of the most precise operand. To perform addition and subtraction on fixed-point data defined by using either a slope that is not an integer power of two or a nonzero bias, specify a chart `fimath` object with `SumMode` set to `SpecifyPrecision`.

Charts that use C as the action language support addition and subtraction for operands of all fixed-point data types. The result is a signed fixed-point number only if both operands are signed. Mixing signed and unsigned operands can yield unexpected results and is not recommended. The slope of the result is equal to the slope of the least precise operand. To simplify calculations and yield efficient code, the biases of the two inputs are added for an addition operation and subtracted for a subtraction operation.

	a	b	MATLAB as the Action Language	C as the Action Language
Sign	s_a	s_b	$s_c = s_a \|\| s_b$	$s_c = s_a \&\& s_b$
Word length	w_a	w_b	$w_c = \max(w_a - f_a, w_b - f_b) + \max(f_a, f_b) + 1$	$w_c = \max(w_a, w_b, w_{int})$
Fraction length	f_a	f_b	$f_c = \max(f_a, f_b)$	$f_c = \min(f_a, f_b)$

	a	b	MATLAB as the Action Language	C as the Action Language
Slope	S_a (2^{-f_a} if using binary-point scaling)	S_b (2^{-f_b} if using binary-point scaling)	$S_c = \min(S_a, S_b)$	$S_c = \max(S_a, S_b)$
Bias	B_a (0 if using binary-point scaling)	B_b (0 if using binary-point scaling)	$B_c = 0$	$B_c = B_a + B_b$ for addition or $B_c = B_a - B_b$ for subtraction

Multiplication

By default, charts that use MATLAB as the action language support multiplication only on fixed-point data defined through binary-point scaling. If either operand is a signed fixed-point number, then the result is also signed. A full precision product requires a word length equal to the sum of the word lengths of the operands. The fraction length of a product is the sum of the fraction lengths of the operands. To perform multiplication on fixed-point data defined by using either a slope that is not an integer power of two or a nonzero bias, specify a chart `fimath` object with `ProductMode` set to `SpecifyPrecision`.

Charts that use C as the action language support multiplication only on fixed-point data operands defined by nonzero biases. The result is a signed fixed-point number only if both operands are signed. Mixing signed and unsigned operands can yield unexpected results and is not recommended. The slope of a product is the product of the slopes of the operands.

	a	b	MATLAB as the Action Language	C as the Action Language
Sign	s_a	s_b	$s_c = s_a \vert\vert s_b$	$s_c = s_a$ && s_b
Word length	w_a	w_b	$w_c = w_a + w_b$	$w_c = \max(w_a, w_b, w_{int})$
Fraction length	f_a	f_b	$f_c = f_a + f_b$	$f_c = f_a + f_b$
Slope	S_a (2^{-f_a} if using binary-point scaling)	S_b (2^{-f_b} if using binary-point scaling)	$S_c = S_a S_b$	$S_c = S_a S_b$
Bias	$B_a = 0$	$B_b = 0$	$B_c = 0$	$B_c = 0$

Division

Charts that use MATLAB as the action language support division only on fixed-point data defined through binary-point scaling. If either operand is a signed fixed-point number, then the result is also signed. A full precision quotient requires a word length equal to the maximum number of bits in the operands. The fraction length of a quotient is the difference of the fraction lengths of the operands.

Charts that use C as the action language support division for fixed-point data operands defined by nonzero biases. The result is a signed fixed-point number only if both operands are signed. Mixing signed and unsigned operands can yield unexpected results and is not recommended. The slope of a quotient is the quotient of the slopes of the operands.

	a	b	MATLAB as the Action Language	C as the Action Language
Sign	s_a	s_b	$s_c = s_a \|\| s_b$	$s_c = s_a$ && s_b
Word length	w_a	w_b	$w_c = \max(w_a, w_b)$	$w_c = \max(w_a, w_b, w_{int})$
Fraction length	f_a	f_b	$f_c = f_a - f_b$	$f_c = f_a - f_b$
Slope	S_a (2^{-f_a} if using binary-point scaling)	S_b (2^{-f_b} if using binary-point scaling)	$S_c = S_a/S_b$	$S_c = S_a/S_b$
Bias	$B_a = 0$	$B_b = 0$	$B_c = 0$	$B_c = 0$

Unary Minus

The only unary operation that requires a promotion of its result type is the unary minus operation `c = -a`. Taking the negative of an unsigned fixed-point number can yield unexpected results and is not recommended. The word size of the result depends on the action language property of the chart. The slope of the result is equal to the slope of the operand. The bias of the result type is the negative of the bias of the operand.

	a	MATLAB as the Action Language	C as the Action Language
Sign	s_a	$s_c = s_a$	$s_c = s_a$
Word length	w_a	$w_c = w_a$	$w_c = \max(w_a, w_{int})$
Fraction length	f_a	$f_c = f_a$	$f_c = f_a$
Slope	S_a (2^{-f_a} if using binary-point scaling)	$S_c = S_a$	$S_c = S_a$
Bias	B_a (0 if using binary-point scaling)	$B_c = -B_a$	$B_c = -B_a$

Arithmetic with Mixed Numeric Types

This table summarizes the fixed-point promotion rules for a binary operation between a fixed-point number and an operand of a different numeric type.

Numeric Type of Second Operand	MATLAB as the Action Language	C as the Action Language
Floating-point numbers: • `single` • `double`	Before performing the operation, the chart casts the floating-point operand to a fixed-point number. The type used for the cast depends on the operation: • Addition and subtraction operations use the same type as the fixed-point operand. • Multiplication operations use the same word length and signedness as the fixed-point operand, and the best precision fraction length for a fixed-point result. The result of the operation is a fixed-point number.	Before performing the operation, the chart casts the fixed-point operand to a floating-point number . The casting operation uses the same type (`single` or `double`) as the floating-point operand. The result of the operation is a floating-point number.
Integers: • `int64` • `int32` • `int16` • `int8` • `uint64` • `uint32` • `uint16` • `uint8`	The integer operand is treated as a fixed-point number of the same word length and signedness with slope $S = 1$ and bias $B = 0$. The result of the operation is a fixed-point number.	The integer operand is treated as a fixed-point number of the same word length and signedness with slope $S = 1$ and bias $B = 0$. The result of the operation is a fixed-point number.

See Also

More About

- “Fixed-Point Data in Stateflow Charts” on page 23-2
- “Operations for Fixed-Point Data in Stateflow” on page 23-8
- “Build a Low-Pass Filter by Using Fixed-Point Data” on page 23-15

Compare Fixed-Point and Floating-Point Computation in Mandelbrot Set

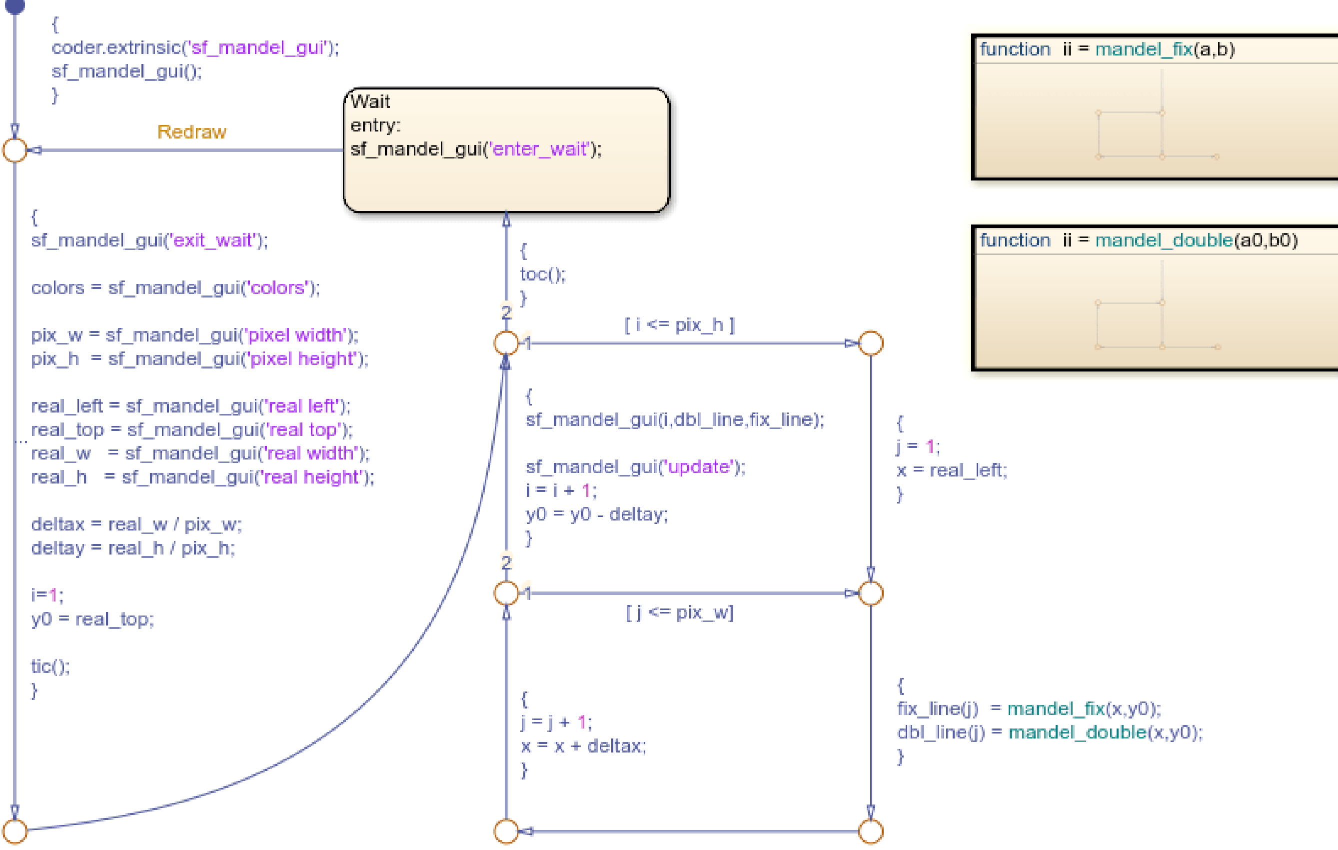
This example shows the difference between fixed-point numbers and floating-point numbers by computing the Mandelbrot set using each data type. For more information, see "Fixed-Point Data in Stateflow Charts" on page 23-2.

The Mandelbrot set is the set of complex numbers, c, for which this sequence does not diverge:

- $z_0 = 0$
- $z_{n+1} = z_n^2 + c$

In this example, a Stateflow® chart creates two images of the Mandelbrot set by using fixed-point and floating-point computations. The chart wakes and calls the helper function `sf_mandel_gui` to initialize the display and obtain the initial pixel and figure information. Then the chart iterates through each row and column in the figure. The pixels in each figure represent points in the complex plane.

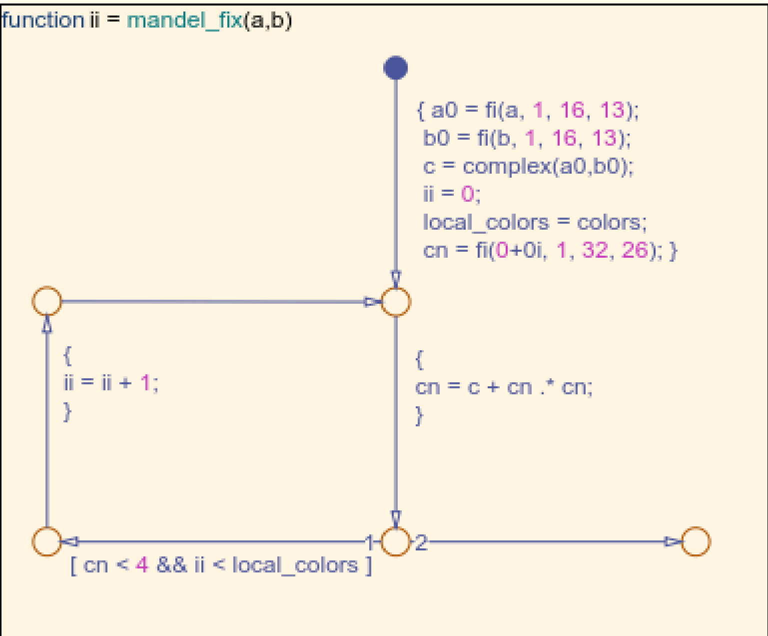
The *x*-coordinate of a pixel represents the real part of a complex number and the *y*-coordinate represents the imaginary part. For each pixel, the chart calls the graphical functions `mandel_fix` and `mandel_double` to compute the number of iterations it takes to determine whether the pixel is in the Mandelbrot set and maps the number to a color in a colormap. After the chart updates every pixel in the figures, the chart enters the `Wait` state and sleeps until you click on the image in the figure window to zoom into the figure.



Compute Mandelbrot Set by Using Graphical Functions

The `mandel_double` function uses floating-point computation to compute the number of iterations required to determine if the point is not in the Mandelbrot set. If the sequence does not diverge before the number of `local_colors` iterations, the function determines the point is in the Mandelbrot set.

The `mandel_fix` function uses fixed-point computation. `a0` and `b0` are fixed-point numeric objects constructed from the input values. `cn` is a fixed-point numeric object with twice the word length and fraction length as `a0` and `b0`.

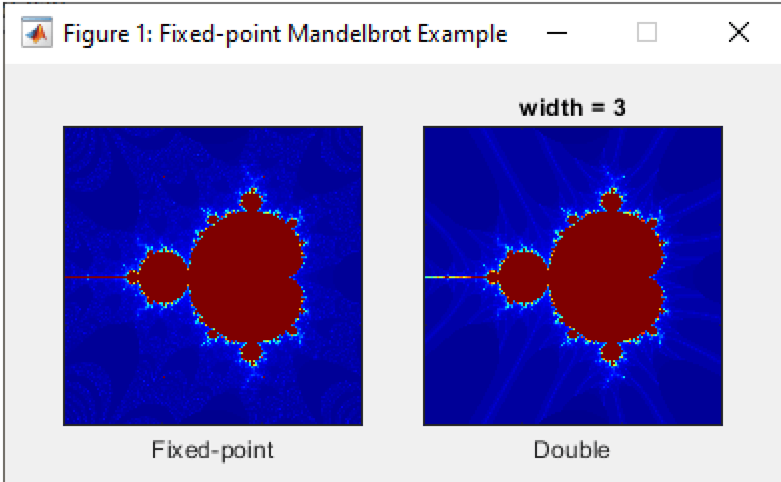


The fixed-point numeric object properties are specified in the **Property Inspector**, in the **Fixed-point properties** section. The chart uses a fixed-point numeric object with two's component overflow that rounds to negative infinity and has a word length of 32 and a fraction length of 26 for products and sums.

```
fimath('OverflowAction','Wrap',...
    'RoundingMethod','Floor',...
    'ProductMode', 'SpecifyPrecision',...
    'ProductWordLength', 32,...
    'ProductFractionLength', 26,...
    'SumMode', 'SpecifyPrecision',...
    'SumWordLength', 32,...
    'SumFractionLength', 26)
```

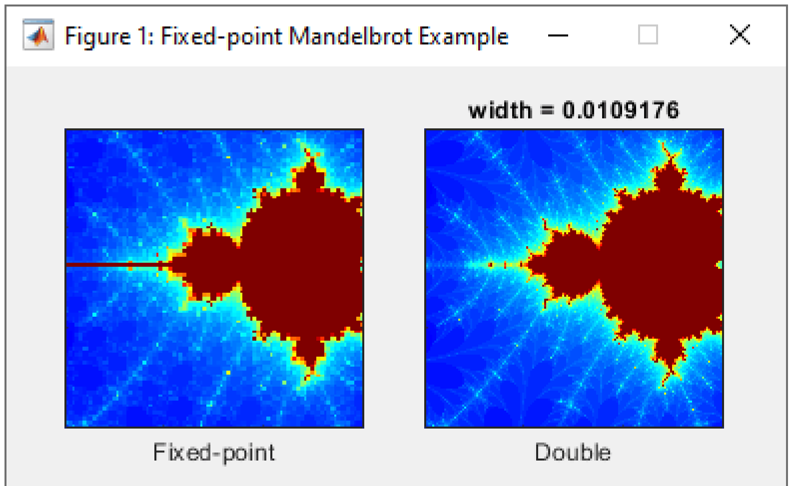
Run the Simulation

To start the simulation, press **Run**. In the figure window *x*-axis displays the interval from -2 to 1 and the *y*-axis displays the interval from -1.5 to 1.5. The color of the pixel corresponds to the number of iterations calculated in `mandel_fix` or `mandel_double`. Pixels that correspond to a complex number in the Mandelbrot set are dark red. Dark blue pixels indicate that `mandel_fix` or `mandel_double` required fewer iterations to determine the sequence diverged, and brighter colors indicate sequences that required more iterations to diverge.



To zoom in on a portion of the image, click and drag in the image window.

Observe the differences between the fixed-point image and the double image. The fixed-point image contains false positives due to the imprecision in comparison to floating point computations. Each computation in `mandel_fix` takes the floor of value as the rounding method. After many iterations, the rounding compounds and produces a round-off error that occurs when a point not in the Mandelbrot set does not produce a value of `r2` that surpasses the threshold of 4 in the maximum number of iterations.



To zoom out, restart the simulation. The simulation continues to run until you click **Stop**.

See Also

More About

- "Fixed-Point Data"

Complex Data

- “Complex Data in Stateflow Charts” on page 24-2
- “Operations for Complex Data in Stateflow” on page 24-4
- “Best Practices for Using Complex Data in C Charts” on page 24-7
- “Measure Frequency Response by Using Complex Data in Stateflow” on page 24-11
- “Detect Valid Transmission Data by Using Frame Synchronization” on page 24-17

Complex Data in Stateflow Charts

Complex data is data whose value is a complex number. For example, in a Stateflow chart in Simulink model, an input signal with the value `3 + 5i` is complex. See “Complex Signals” (Simulink).

Define Complex Data

1. Add a data object to your chart, as described in “Add Stateflow Data” on page 10-2.
2. Set the **Complexity** property for the data object to `On`. For more information, see “Complexity” on page 10-7.
3. Specify the name, scope, size, base type, and other properties for the data object, as described in “Set Data Properties” on page 10-5.
   - Complex data does not support the scope `Constant`.
   - Complex data does not support the base types `ml`, `struct`, and `boolean`.

When to Use Complex Data

Use complex data when you model applications in communication systems and digital signal processing. For example, you can use this design pattern to model a frame synchronization algorithm in a communication system:

1. Use Simulink blocks (such as filters) to process complex signals.
2. Use charts to implement mode logic for frame synchronization.
3. Let the charts access complex input and output data so that nested MATLAB functions can drive the mode logic.

For an example of modeling a frame synchronization algorithm, see “Detect Valid Transmission Data by Using Frame Synchronization” on page 24-17.

Note Continuous-time variables of complex type are *not* supported. For more information, see “Store Continuous State Information in Local Variables” on page 22-6.

Where You Can Use Complex Data

You can define complex data at these levels of the Stateflow hierarchy:

- Charts
- Subcharts
- States
- Functions

How You Can Use Complex Data

You can use complex data to define:

- Complex vectors

- Complex matrices

You can also use complex data as arguments for:

- State actions
- Transition actions
- MATLAB functions (see “Reuse MATLAB Code by Defining MATLAB Functions” on page 7-2)
- Truth table functions (see “Use Truth Tables to Model Combinatorial Logic” on page 8-2)
- Graphical functions (see “Reuse Logic Patterns by Defining Graphical Functions” on page 6-9)
- Change detection operators (see “Detect Changes in Data and Expression Values” on page 14-63)

Note Exported functions do not support complex data as arguments.

See Also

More About

- “Operations for Complex Data in Stateflow” on page 24-4
- “Rules for Using Complex Data in C Charts” on page 24-9
- “Best Practices for Using Complex Data in C Charts” on page 24-7

Operations for Complex Data in Stateflow

Stateflow charts in Simulink models have an action language property that defines the syntax that you use to compute with complex data. The action language properties are:

- MATLAB as the action language.
- C as the action language.

For more information, see "Differences Between MATLAB and C as Action Language Syntax" on page 15-4.

Notation for Complex Data

In charts that use MATLAB as the action language, you can define complex data by using complex number notation `a + bi`, where `a` and `b` are real numbers. For example, this statement assigns a value of 3+4i to `x`:

```
x = 3 + 4i;
```

Alternatively, you can define complex data by using the `complex` operator:

```
complex(<real_part>,<imag_part>)
```

<real_part> and *<imag_part>* are arguments that define the real and imaginary parts of the complex number, respectively. The two arguments must be real values or expressions that evaluate to real values. As in the preceding example, this statement assigns a value of 3+4i to `x`:

```
x = complex(3,4);
```

Charts that use C as the action language do not support complex number notation `a + bi`. To define a complex number based on two real values, use the `complex` operator.

Binary Operations

This table summarizes the interpretation of all binary operations on complex operands according to their order of precedence (1 = highest, 3 = lowest). Binary operations are left associative so that, in any expression, operators with the same precedence are evaluated from left to right.

Operation	Precedence	MATLAB as the Action Language	C as the Action Language
`a * b`	1	Multiplication.	Multiplication.
`a / b`	1	Division.	Not supported. Use the \ operation in a MATLAB function. See "Perform Complex Division with a MATLAB Function" on page 24-8.
`a + b`	2	Addition.	Addition.
`a - b`	2	Subtraction.	Subtraction.
`a == b`	3	Comparison, equal to.	Comparison, equal to.

Operation	Precedence	MATLAB as the Action Language	C as the Action Language
a ~= b	3	Comparison, not equal to.	Comparison, not equal to.
a != b	3	Not supported. Use the operation a ~= b.	Comparison, not equal to.
a <> b	3	Not supported. Use the operation a ~= b.	Comparison, not equal to.

Unary Operations and Actions

This table summarizes the interpretation of all unary operations and actions on complex data. Unary operations:

- Have higher precedence than the binary operators.
- Are right associative so that, in any expression, they are evaluated from right to left.

Operation	MATLAB as the Action Language	C as the Action Language
-a	Negative.	Negative.
a++	Not supported. Use the expression a = a+1.	Increment. Equivalent to a = a+1.
a--	Not supported. Use the expression a = a-1.	Decrement. Equivalent to a = a-1.

Assignment Operations

This table summarizes the interpretation of assignment operations in Stateflow charts.

Operation	MATLAB as the Action Language	C as the Action Language
a = b	Simple assignment.	Simple assignment.
a += b	Not supported. Use the expression a = a+b.	Equivalent to a = a+b.
a -= b	Not supported. Use the expression a = a-b.	Equivalent to a = a-b.
a *= b	Not supported. Use the expression a = a*b.	Equivalent to a = a*b.

Access Real and Imaginary Parts of a Complex Number

To access the real and imaginary parts of a complex number, use the `real` and `imag` operators.

real Operator

The `real` operator returns the value of the real part of a complex number:

```
real(<complex_expr>)
```

<complex_expr> is an expression that evaluates to a complex number. For example, if `frame(200)` evaluates to the complex number `8.23 + 4.56i`, this expression returns a value of `8.2300`:

```
real(frame(200))
```

imag Operator

The `imag` operator returns the value of the imaginary part of a complex number:

```
imag(<complex_expr>)
```

`<complex_expr>` is an expression that evaluates to a complex number. For example, if `frame(200)` evaluates to the complex number `8.23 + 4.56i`, this expression returns a value of `4.5600`:

```
imag(frame(200))
```

See Also

`complex` | `i` | `imag` | `real`

More About

- “Complex Data in Stateflow Charts” on page 24-2
- “Rules for Using Complex Data in C Charts” on page 24-9
- “Best Practices for Using Complex Data in C Charts” on page 24-7

Best Practices for Using Complex Data in C Charts

Complex data is data whose value is a complex number. For example, in a Stateflow chart in Simulink model, an input signal with the value `3 + 5i` is complex. See “Complex Data in Stateflow Charts” on page 24-2.

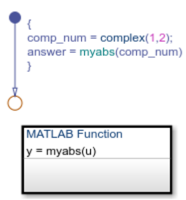
When you use complex data in Stateflow charts that use C as the action language, follow these best practices.

Perform Math Function Operations with a MATLAB Function

Math functions such as `sin`, `cos`, `min`, `max`, and `abs` do not work with complex data in C charts. However, you can use a MATLAB function in your chart to perform math function operations on complex data.

A Simple Example

In the following chart, a MATLAB function calculates the absolute value of a complex number:



The value of `comp_num` is `1+2i`. Calculating the absolute value gives an answer of `2.2361`.

How to Calculate Absolute Value

Suppose that you want to find the absolute value of a complex number. Follow these steps:

1. Add a MATLAB function to your chart with this signature:

   ```
   y = myabs(u)
   ```
2. Double-click the function box to open the editor.
3. In the editor, enter the code below:

   ```
   function y = myabs(u)
   %#codegen
   y = abs(u);
   ```

   The function `myabs` takes a complex input `u` and returns the absolute value as an output `y`.
4. Configure the input argument `u` to accept complex values.

   a. Open the Model Explorer.

   b. In the **Model Hierarchy** pane of the Model Explorer, navigate to the MATLAB function `myabs`.

   c. In the **Contents** pane of the Model Explorer, right-click the input argument `u` and select **Properties** from the context menu.

   d. In the Data properties dialog box, select `On` in the **Complexity** field and click **OK**.

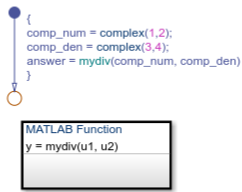
You cannot pass real values to function inputs of complex type. For details, see “Rules for Using Complex Data in C Charts” on page 24-9.

Perform Complex Division with a MATLAB Function

Division with complex operands is not available as a binary or assignment operation in C charts. However, you can use a MATLAB function in your chart to perform division on complex data.

A Simple Example

In the following chart, a MATLAB function performs division on two complex operands:



The values of `comp_num` and `comp_den` are `1+2i` and `3+4i`, respectively. Dividing these values gives an answer of `0.44+0.08i`.

How to Perform Complex Division

To divide two complex numbers:

1. Add a MATLAB function to your chart with this function signature:

   ```
   y = mydiv(u1, u2)
   ```
2. Double-click the function box to open the editor.
3. In the editor, enter the code below:

   ```
   function y = mydiv(u1, u2)
   %#codegen
   y = u1 / u2;
   ```

   The function `mydiv` takes two complex inputs, `u1` and `u2`, and returns the complex quotient of the two numbers as an output `y`.
4. Configure the input and output arguments to accept complex values.

   a. Open the Model Explorer.

   b. In the **Model Hierarchy** pane of the Model Explorer, navigate to the MATLAB function `mydiv`.

   c. For each input and output argument, follow these steps:

      i. In the **Contents** pane of the Model Explorer, right-click the argument and select **Properties** from the context menu.

      ii. In the Data properties dialog box, select `On` in the **Complexity** field and click **OK**.

You cannot pass real values to function inputs of complex type. For details, see “Rules for Using Complex Data in C Charts” on page 24-9.

Rules for Using Complex Data in C Charts

Complex data is data whose value is a complex number. For example, in a Stateflow chart in Simulink model, an input signal with the value `3 + 5i` is complex. See "Complex Data in Stateflow Charts" on page 24-2.

These rules apply when you use complex data in Stateflow charts that use C as the action language.

Do not use complex number notation in actions

C charts do not support complex number notation `(a + bi)`, where `a` and `b` are real numbers. Therefore, you cannot use complex number notation in state actions, transition conditions and actions, or any statements in C charts.

To define a complex number, use the `complex` operator as described in "Notation for Complex Data" on page 24-4.

Do not perform math function operations on complex data in C charts

Math operations such as `sin`, `cos`, `min`, `max`, and `abs` do not work with complex data in C charts. However, you can use MATLAB functions for these operations.

For more information, see "Perform Math Function Operations with a MATLAB Function" on page 24-7.

Mix complex and real operands only for addition, subtraction, and multiplication

If you mix operands for any other math operations in C charts, an error appears when you try to simulate your model.

To mix complex and real operands for division, you can use a MATLAB function as described in "Perform Complex Division with a MATLAB Function" on page 24-8.

Tip Another way to mix operands for division is to use the `complex`, `real`, and `imag` operators in C charts.

Suppose that you want to calculate `y = x1/x2`, where `x1` is complex and `x2` is real. You can rewrite this calculation as:

```
y = complex(real(x1)/x2, imag(x1)/x2)
```

For more information, see "Access Real and Imaginary Parts of a Complex Number" on page 24-5.

Do not define complex data with constant scope

If you define complex data with `Constant` scope, an error appears when you try to simulate your model.

Do not define complex data with `ml`, `struct`, or `boolean` base type

If you define complex data with `ml`, `struct`, or `boolean` base type, an error appears when you try to simulate your model.

Use only real values to set initial values of complex data

When you define the initial value for data that is complex, use only a real value. See "Additional Properties" on page 10-16 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 C chart passes real values to function inputs of complex type, an error appears when you try to simulate your model.

Do not use complex data with temporal logic operators

You cannot use complex data as an argument for temporal logic operators, because you cannot define time as a complex number.

See Also

More About

- "Complex Data in Stateflow Charts" on page 24-2
- "Operations for Complex Data in Stateflow" on page 24-4
- "Rules for Using Complex Data in C Charts" on page 24-9

Measure Frequency Response by Using Complex Data in Stateflow

This example shows how to emulate a spectrum analyzer that measures the frequency response of a continuous-time system driven by a complex sinusoidal signal. In this Simulink® model, the Plant block describes a second-order resonant system of the form

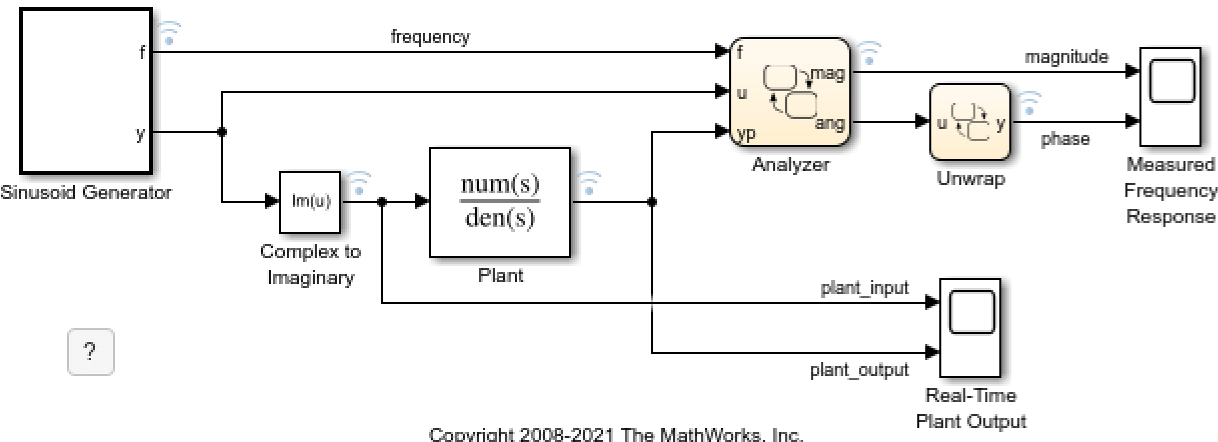
$$\frac{\omega^2}{s^2 + 2\zeta\omega s + \omega^2}$$

with a natural frequency of $\omega = 2\pi \cdot 150$ (or 150 Hz) and a damping ratio of $\zeta = 0.3$. Because the system is underdamped ($\zeta < 1$), the system has two complex conjugate poles. In a real-world application, replace this block with a subsystem composed of xPC D/A and A/D blocks that measure the response of a device under test (DUT).

The spectrum analyzer consists of these components:

- The Sinusoidal Generator block, which produces a complex sinusoidal signal of increasing frequency. Inside the block, a Stateflow® chart uses temporal logic to iterate over a range of frequencies.
- The Stateflow chart Analyzer, which calculates the frequency response (magnitude and phase angle) of the system at a specified frequency. The chart registers changes in the frequency by using change detection logic.
- The Stateflow chart Unwrap, which processes the measured phase angle and unwraps the result so that there are no sharp jumps between π and $-\pi$.

The spectrum analyzer displays the measured frequency response as a pair of discrete Bode plots.



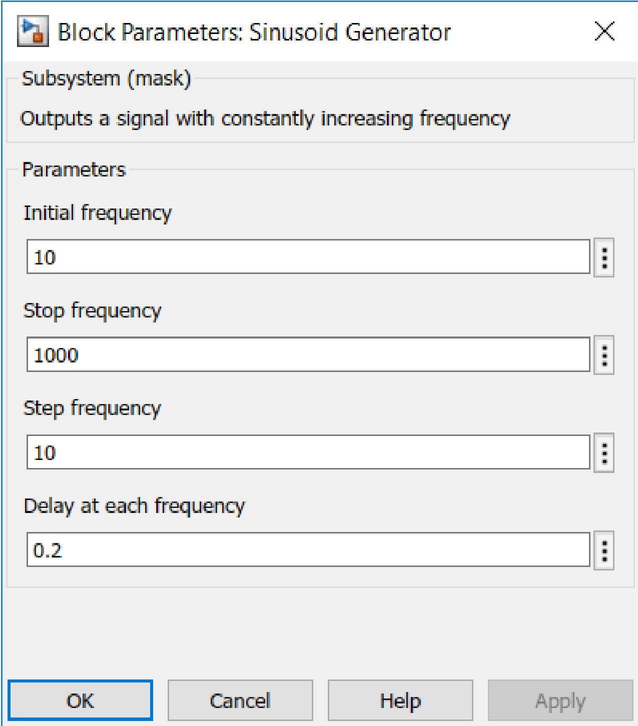
Generate Sinusoid Signal

The Sinusoid Generator block has two outputs:

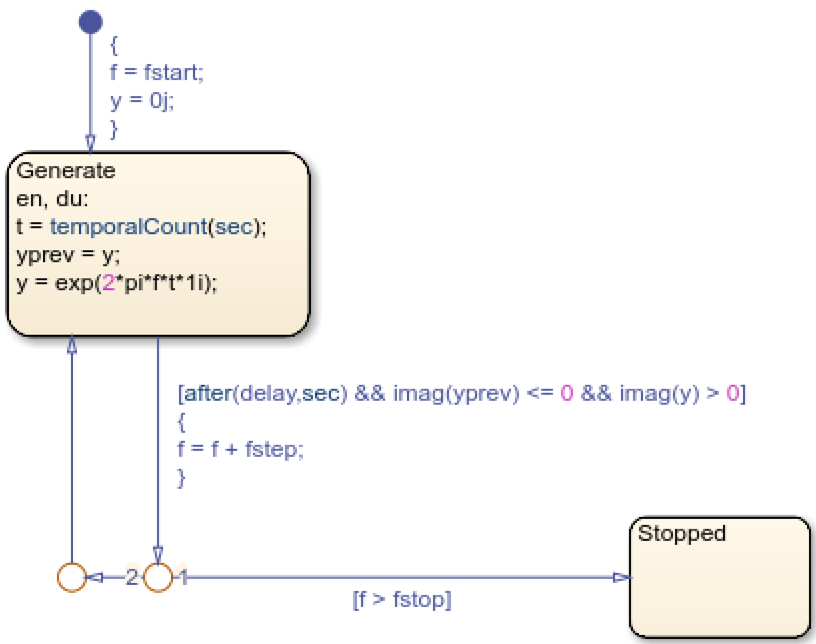
- A scalar `f` that represents the current frequency.

- A complex signal `y` that has a frequency of `f`.

You can control the behavior of the block by modifying its mask dialog box parameters. For example, the default values specify a sinusoidal signal with frequencies between 10Hz and 1000Hz. The block holds each frequency value for 0.2 seconds, and then increments the frequency by a step of 10Hz.



To control the timing of the signal generator, the block contains a Stateflow chart that applies absolute-time temporal logic.



During simulation, the chart goes through these stages:

- **Initialize Frequency**: The default transition sets the signal frequency `f` to the value of the parameter `fstart`. Specify the value of `fstart` in the Sinusoid Generator mask dialog box.
- **Generate Signal**: While the `Generate` state is active, the chart produces the complex signal `y = exp(2*pi*f*t*1i)` based on the frequency `f` and the time `t` since the last change in frequency. To determine the elapsed time (in seconds) since the state became active, the chart calls the temporal logic operator `temporalCount`.
- **Update Frequency**: After the `Generate` state is active for `delay` seconds, the chart transitions out of the state, increases the frequency `f` by `fstep`, and returns to the `Generate` state. To determine the timing of the frequency updates, the chart calls the temporal logic operator `after` and checks the sign of the imaginary part of the signal before and after the current time step. The sign checks ensure that the output signal completes a full cycle before the frequency is updated, preventing large changes in the output signal. Specify the values of `fstep` and `delay` in the Sinusoid Generator mask dialog box.
- **Stop Simulation**: When the frequency `f` reaches the value of the parameter `fstop`, the chart transitions to the `Stopped` state. The active state output `Stopped` triggers a Stop Simulation (Simulink) block and the simulation ends. Specify the value of `fstop` in the Sinusoid Generator mask dialog box.

Calculate Frequency Response

The Analyzer chart takes the complex signal `u` and the output `yp` from the Plant block and computes the magnitude and phase of the plant output.

```
Analyze
en:
y = 0i; yn = 0i;
du:
y = y + u.*yp;
yn = yn + u.*imag(u);

[hasChanged(f)]
{
mag = abs(y)/abs(yn);
ang = -angle(y) + pi/2;
}
```

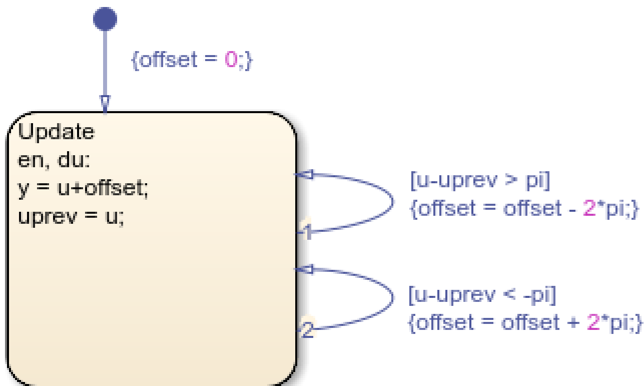
For each frequency value, the chart maintains two running sums:

- The sum `y` approximates the integral of the product of the plant output `yp` (a real number) and the complex signal `u`.
- The sum `yn` approximates the integral of the product of the plant input `imag(u)` and the complex signal `u`. This integral represents the accumulation of a hypothetical plant with a unit transfer function.

To detect a change in frequency, the chart uses the operator `hasChanged` to guard the self-transition on the state `Analyze`. When the frequency changes, the actions on this transition compute the magnitude and phase of the plant output by normalizing `y` with respect to `yn`.

Unwrap Measured Phase Angle

The Unwrap chart prevents the measured phase angle from changing by more than π radians in a single time step.



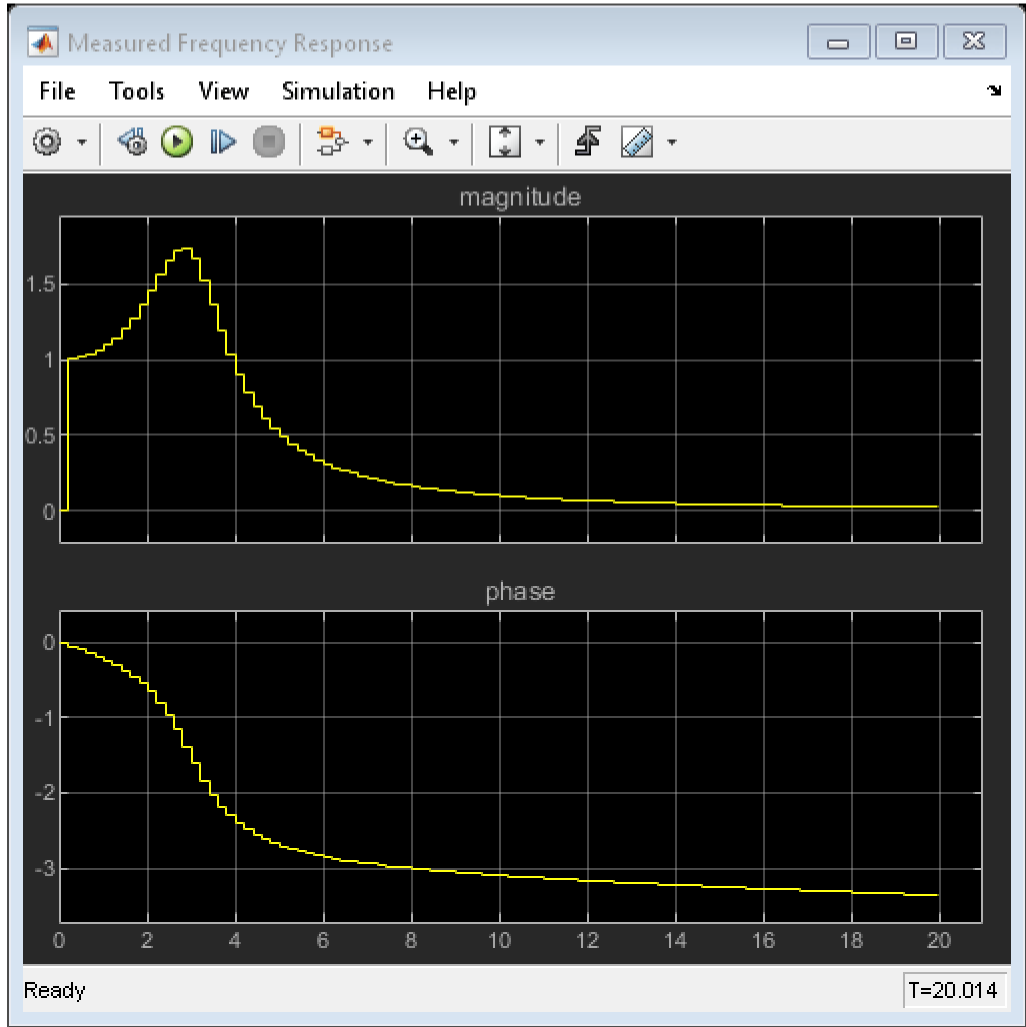
In this chart, the transitions test the change in the input `u` before computing the new output value `y`.

- If the input increases by more than π radians, the chart offsets the output by -2π radians.
- If the input decreases by more than $-\pi$ radians, the chart offsets the output by $+2\pi$ radians.

Examine Simulation Results

When you simulate the model, the scope block shows the frequency response of the system (magnitude and phase) as a function of simulation time.

- In the magnitude plot, the maximum value at $t \approx 3$ indicates the response of the Plant block to a resonant frequency. The peak amplitude is approximately 1.7.
- In the phase plot, the angle changes from 0 to $-\pi$ radians. Each complex pole of the system adds $-\pi/2$ radians to the phase angle.

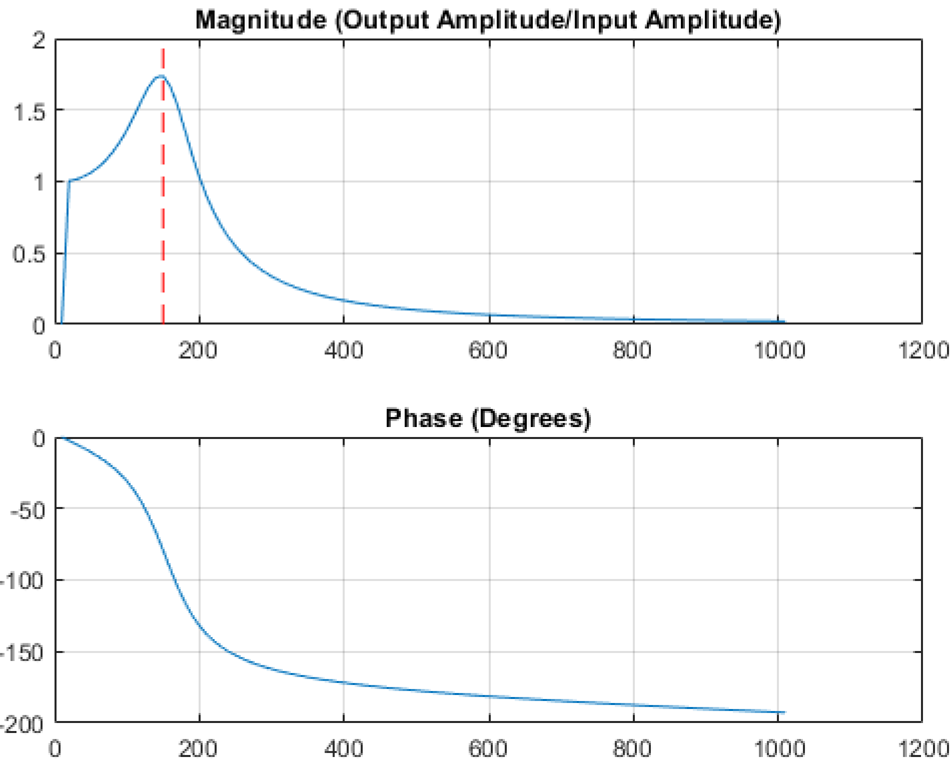


To determine the measured resonant frequency, plot the measured magnitude and phase against the plant input frequency. During simulation, the model saves these values in a signal logging object `logsout` in the MATLAB workspace. You can access the logged values by using the `get (Simulink)` method. For example, to create the Bode plot for the measured frequency response of the system and draw a red cursor at 150Hz, enter:

```
figure;
subplot(211);
plot(get(logsout,"frequency").Values.Data, ...
    get(logsout,"magnitude").Values.Data);
line([150 150],[0 2],Color="red",LineStyle="--");
grid on;
title("Magnitude (Output Amplitude/Input Amplitude)");
```

```
subplot(212);
plot(get(logsout,"frequency").Values.Data, ...
    get(logsout,"phase").Values.Data*180/pi)
grid on;
title("Phase (Degrees)");
```

The Bode plot shows that the measured resonant frequency is approximately 150Hz, matching the value predicted by the plant dynamics.



See Also

Stop Simulation | `after` | `get` | `hasChanged` | `temporalCount`

More About

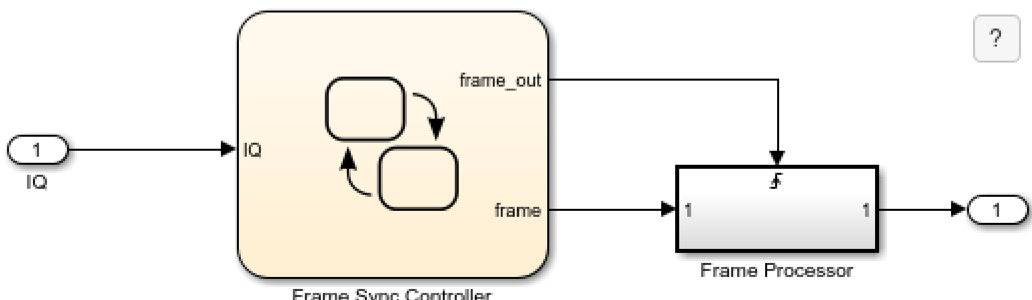
- “Complex Data in Stateflow Charts” on page 24-2
- “Control Chart Execution by Using Temporal Logic” on page 14-35
- “Detect Changes in Data and Expression Values” on page 14-63
- “Access Signal Logging Data” on page 11-15

Detect Valid Transmission Data by Using Frame Synchronization

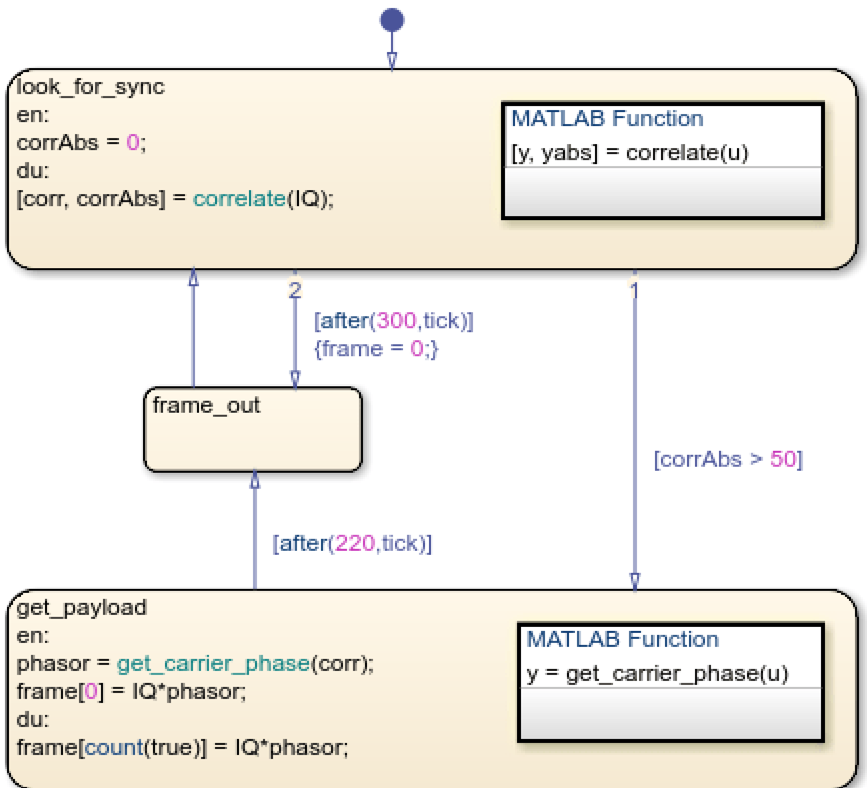
This example shows how to process complex data in a Stateflow® chart. The model uses vectors of complex data to find a fixed pattern in a signal from a communication system.

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 and achieves frame synchronization when the correlation between the input data and the fixed pattern is high.



In this example, the Stateflow chart Frame Sync Controller accepts a complex input signal `IQ` and searches for the fixed data pattern `trainSig`. After recognizing the start of a data frame, the chart stores the valid data in a complex output signal `frame`. This output signal is a vector of complex products between each valid data point and the phase angle of the carrier wave. The chart then passes the valid data to the Frame Processor subsystem.



Note that this model does not contain the rest of the communication system.

Synchronize Data Frame

In the Frame Sync Controller chart, the `look_for_sync` state starts the frame synchronization algorithm. At each time step, the MATLAB® function `correlate` calculates the correlation between the input signal `IQ` and the fixed data pattern `trainSig`. The function stores the complex correlation as `corr` and its absolute value 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.

- If `corrAbs` exceeds 50 percent, the correlation is high and the chart registers the start of a data frame. The chart takes the transition to the `get_payload` state and stores 220 valid data points in the complex vector `frame`.
- If `corrAbs` stays below 50 percent for 300 consecutive data points, the frame synchronization algorithm resets. The chart takes the transition to the `frame_out` state and triggers the Frame Processor subsystem. The chart then returns to the `look_for_sync` state.

Store Scalar Data in a Vector

When the Frame Sync Controller chart recognizes the start of a data frame, the `get_payload` state calls the MATLAB function `get_carrier_phase` to compute the phase angle of the carrier wave. The state stores this phase angle as the local data object `phasor`. Then the state collects the scalar values of the product of `IQ*phasor` in the vector `frame`. To avoid using an extra variable as an index counter, this state indexes into this vector by using the `count` operator:

- When the state becomes active, the `entry` action stores the initial value of the product in the first element of `frame`.
- In later time steps, the `during` action stores the next values of this product in successive elements of `frame`. The indexing expression `count(true)` returns the number of time steps since the state became active.

After 220 times steps, the transition condition `[after(220,tick)]` becomes `true` and the chart exits from the state. When the chart enters the `frame_out` state, the vector `frame` contains the values of 220 products. The chart passes this data to the Frame Processor subsystem and then returns to the `look_for_sync` state.

See Also

`after` | `count`

More About

- “Complex Data in Stateflow Charts” on page 24-2
- “Vectors and Matrices in Stateflow Charts” on page 19-2
- “Control Chart Execution by Using Temporal Logic” on page 14-35

Define Interfaces to Simulink Models and the MATLAB Workspace

- “Stateflow Editor Operations” on page 25-2
- “Manage Symbols in the Stateflow Editor” on page 25-14
- “Use the Model Explorer with Stateflow Objects” on page 25-22
- “Visualize Chart Execution with the Activity Profiler” on page 25-26
- “Connect Dashboard Blocks to Stateflow” on page 25-31
- “Reuse Charts in Models with Chart Libraries” on page 25-34
- “Create a Mask to Share Parameters with Simulink” on page 25-36

Stateflow Editor Operations

Stateflow Editor

The Stateflow Editor allows you to draw, zoom, modify, print, and save a Stateflow chart in your Simulink model. When you open a new Stateflow chart, the Stateflow Editor displays the chart.

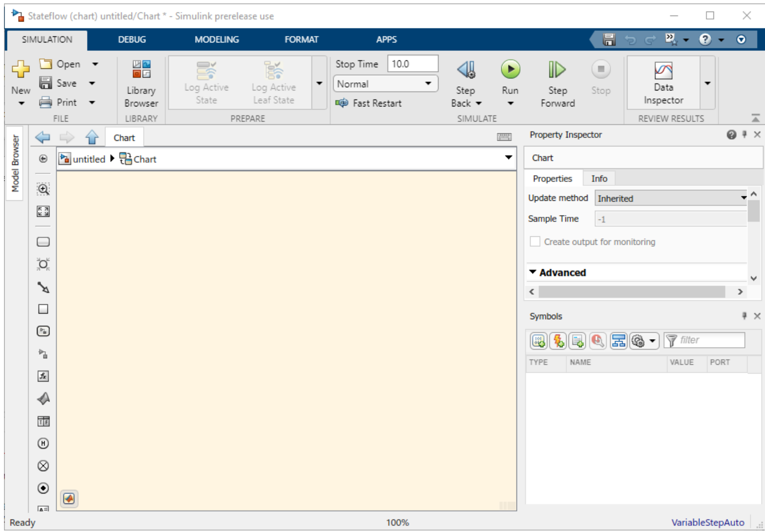
To open a new Stateflow chart in the Stateflow Editor, at the MATLAB command prompt, enter:

1

Command	Result
`sfnew`	Creates a chart with the default action language. For more information, see `sfnew`.
`sfnew -MATLAB`	Creates an empty chart with MATLAB as the action language.
`sfnew -C`	Creates an empty C chart.

   The Simulink Editor opens, and an empty chart is included on the canvas.

2 To open the Stateflow Editor, double-click the chart object.



The main components of the Stateflow Editor are the chart canvas, the object palette, the **Symbols** pane, and the **Property Inspector**.

- The chart canvas is a drawing area where you create a chart by combining states, transitions, and other graphical elements.
- On the left side of the canvas, the object palette displays a set of tools for adding graphical elements to your chart. For more information, see “Graphical Objects” on page 1-5.

  To add an object:

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

  - Drag the icon for the object into the drawing area.
  - Double-click the icon and then click multiple times in the drawing area to make copies of the object.
- On the right side of the canvas, in the **Symbols** pane, you add new data to the chart and resolve any undefined or unused symbols. Also on the right side of the canvas, above the **Symbols** pane, is the **Property Inspector**. Use the **Property Inspector** to change the properties for symbols listed in the **Symbols** pane. For more information, see "Manage Symbols in the Stateflow Editor" on page 25-14.

Undo and Redo Editor Operations

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

- To undo an operation in the chart, press **Ctrl + Z**.
- To redo an operation in the chart, press **Ctrl + Y**.

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.

  For more information about subcharting, see "Encapsulate Modal Logic by Using Subcharts" on page 6-6.
- 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 more information about supertransitions, see "Create a Supertransition That Enters a Subchart" on page 1-49and "Create a Supertransition That Exits a Subchart" on page 1-51.
- You cannot undo any changes made to the chart using the Stateflow API.

  For more information about the Stateflow API, see "Stateflow Programmatic Interface".

Specify Colors and Fonts in a Chart

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

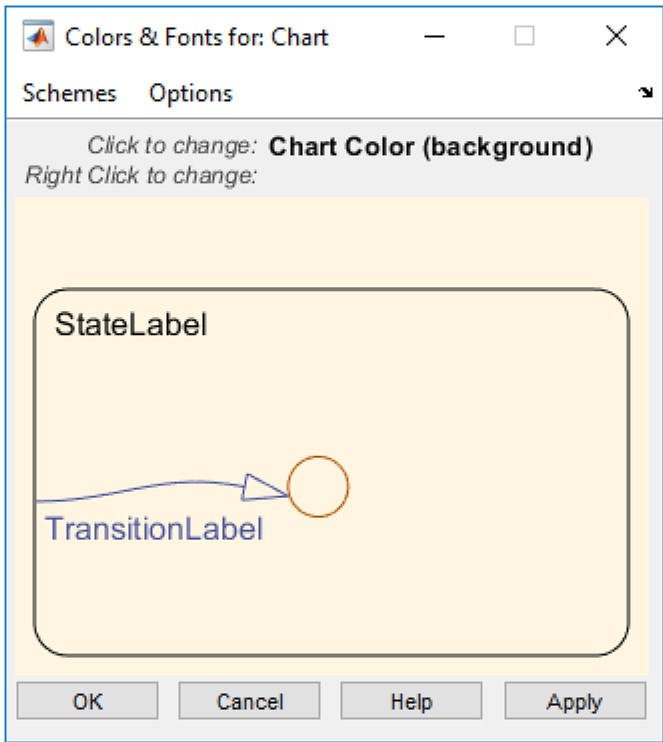
Change Size of a Single Element

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

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

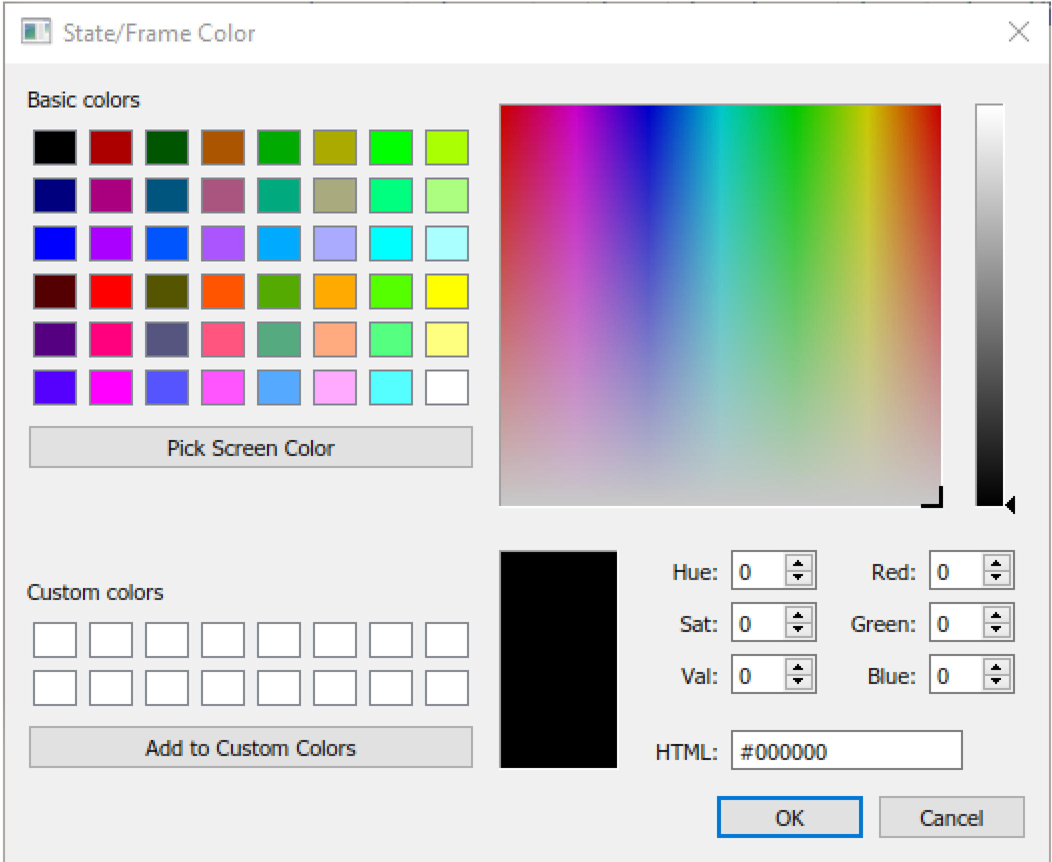
Change Global Display Options

Through the **Colors & Fonts** dialog box, you can specify a color scheme for the chart or specify colors and label fonts for different types of objects in a chart. To open the **Colors & Fonts** dialog box, in the **Format** tab, click **Style**.

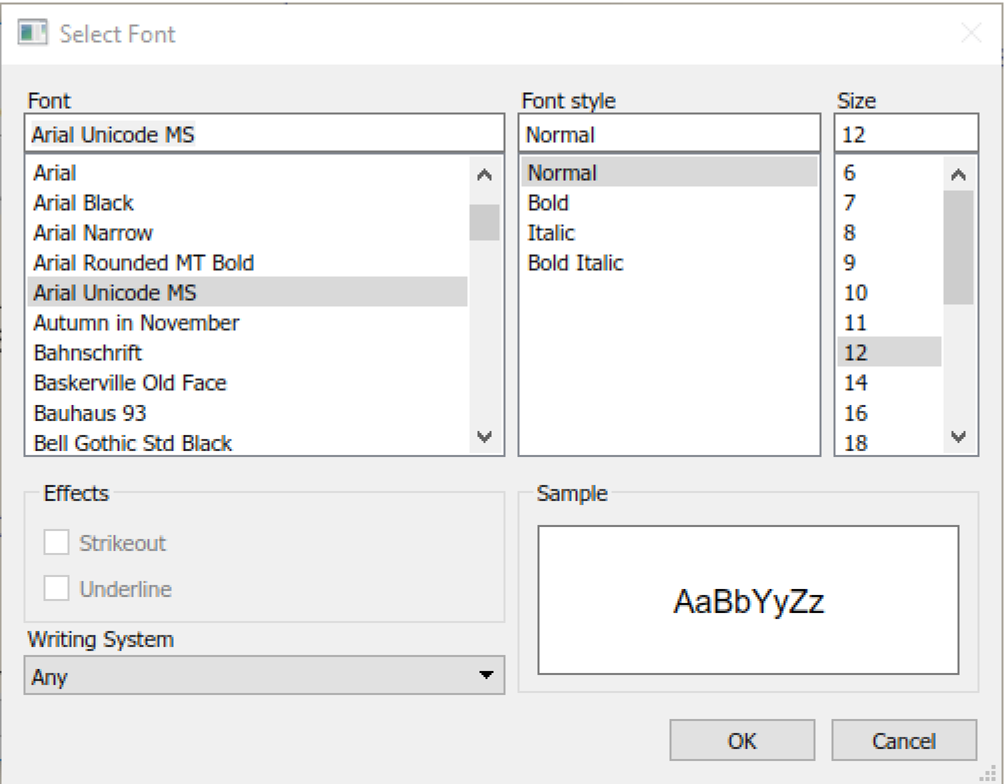


In the **Colors & Fonts** dialog box, the drawing area displays examples of the colors and label fonts specified by the current color scheme for the chart. You can choose a different color scheme from the **Schemes** menu. To modify the display options for a single type of chart element, position your pointer over the sample object.

- To change the color of the element, click the sample object and select a new color in the dialog box.



- To change the font of the element, right-click the sample object and select a new font, style, or size in the dialog box.



To apply the scheme to the chart, click **Apply**. To apply the scheme and close the dialog box, click **OK**.

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

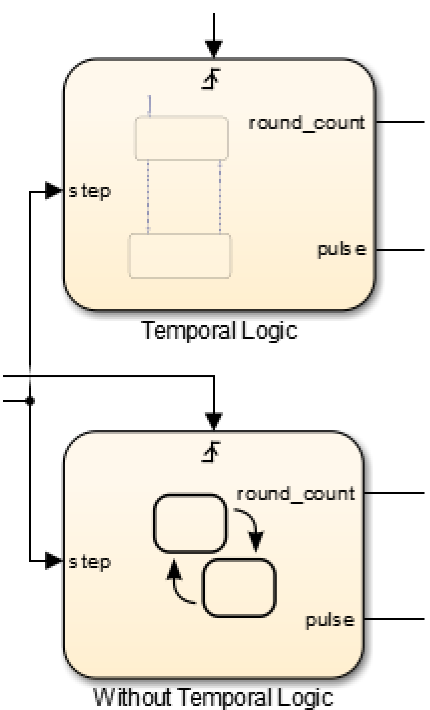
To save changes to the default color scheme, select **Options** > **Save defaults to disk**. If the modified scheme is not the default scheme, choosing **Save defaults to disk** has no effect.

Content Preview for Stateflow Objects

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

To turn on content preview for Stateflow charts and subcharts, right-click the chart and select **Format** > **Content Preview**. For Simulink functions, right-click the function and select **Content Preview**. For details on content preview in Simulink, see “Preview Content of Model Components” (Simulink).

For example, the `Temporal Logic` chart uses content preview. The chart `Without Temporal Logic` does not.



Intelligent Tab Completion for Stateflow Charts

Stateflow tab completion provides context-sensitive editing assistance. Tab completion helps you avoid typographical errors. It also helps you quickly select syntax-appropriate options for keywords, data, event, messages, and function names, without having to navigate the Model Explorer. In a Stateflow chart, to complete entries:

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

Additionally, you can:

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

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

Differentiate Elements of Action Language Syntax

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

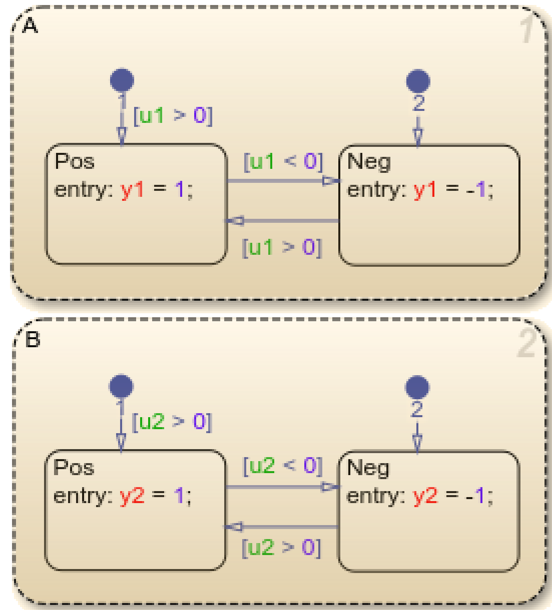
- Keywords
- Comments

- Events
- Messages
- Functions
- Strings
- Numbers
- Local data
- Constant data
- Input data
- Output data
- Parameter data
- Data Store Memory data

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

Default Syntax Highlighting

The following chart illustrates the default highlighting for language elements.



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

Edit Syntax Highlighting

1. In the Stateflow Editor, in the **Format** tab, click **Style** > **Syntax Highlighting**.

   The Syntax Highlight Preferences dialog box appears.

2 Click the color that you want to change, choose an alternative from the color palette, and click **Apply**.
3 Click **OK** to close the Syntax Highlight Preferences dialog box.

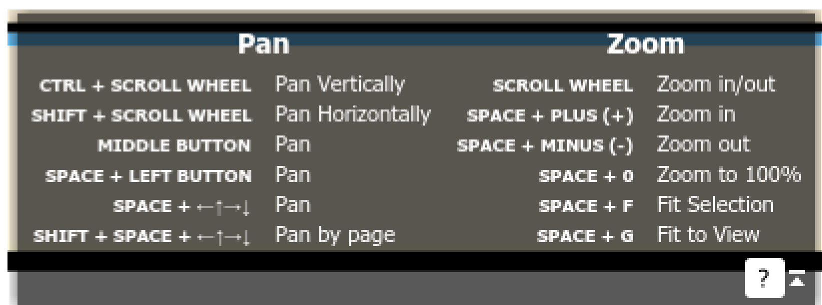
Enable and Disable Syntax Highlighting

1 In the Stateflow Editor, in the **Format** tab, click **Style** > **Syntax Highlighting**.

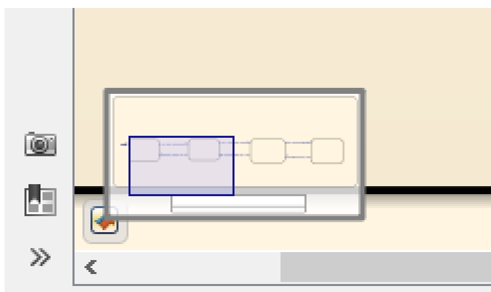
   The Syntax Highlight Preferences dialog box appears.
2 Select or clear **Enable syntax highlighting** and click **OK**.

Zoom and Navigate with the Miniature Map

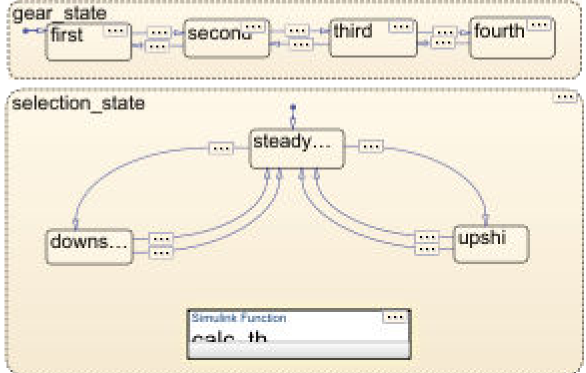
To zoom in or out of a Stateflow chart, use the scroll wheel on your mouse. When you zoom Press and hold **Space** to see a dialog with the keyboard shortcuts.



You can also press Space to view a miniature map of your chart in the bottom-left corner. As you zoom or navigate within your Stateflow chart, the blue square will move on the miniature map, indicating your location relative to the entire chart.



You can also click and drag the blue square on the miniature map to navigate within your Stateflow chart. As you zoom out, the text in your Stateflow chart adjusts its size to maintain readability.



Format Chart Objects

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

- Alignment
- Distribution
- Resizing

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

- States
- Functions
- Boxes
- Junctions

Some of these options appear only after selecting elements within your chart.

Align, Distribute, and Resize Chart Objects

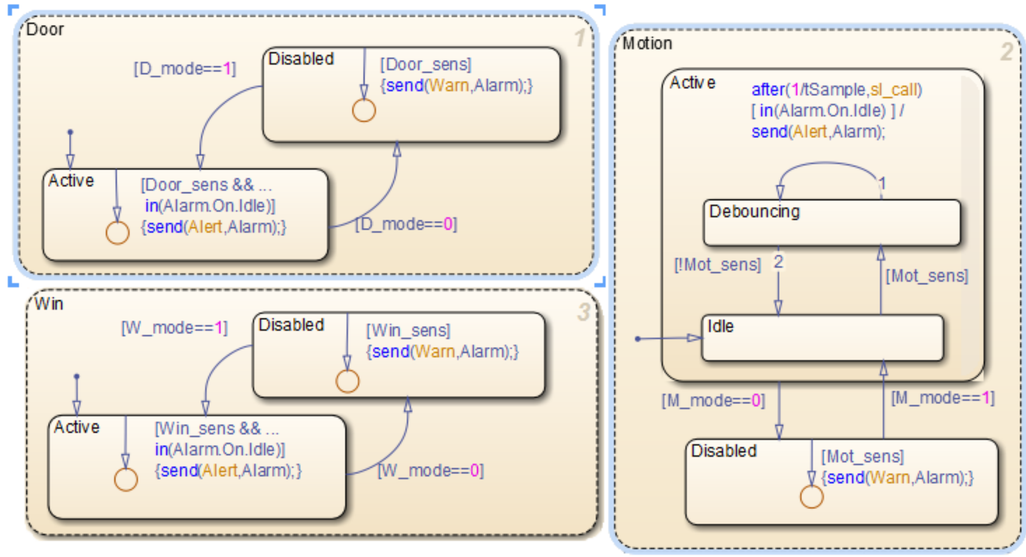
The basic steps to align, distribute, or resize chart objects are similar.

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

   To set explicit ordering, in the Chart properties dialog box, select **User-specified state/ transition execution order**.
2. Select the chart objects that you want to align, distribute, or resize.

   You can select objects in any order, one-by-one, or by drawing a box around them.
3. Decide which object to use as the anchor for aligning, distributing, or resizing other chart objects. This object is the reference object.

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



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

For more information about chart object distribution options, see “Options for Distributing Chart Objects” on page 25-11

Options for Distributing Chart Objects

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

Option	Description
Even Vertical Gaps	The vertical white space between any two objects is the same. The space restriction for **Distribute Vertically** applies.

Automatic Chart Formatting

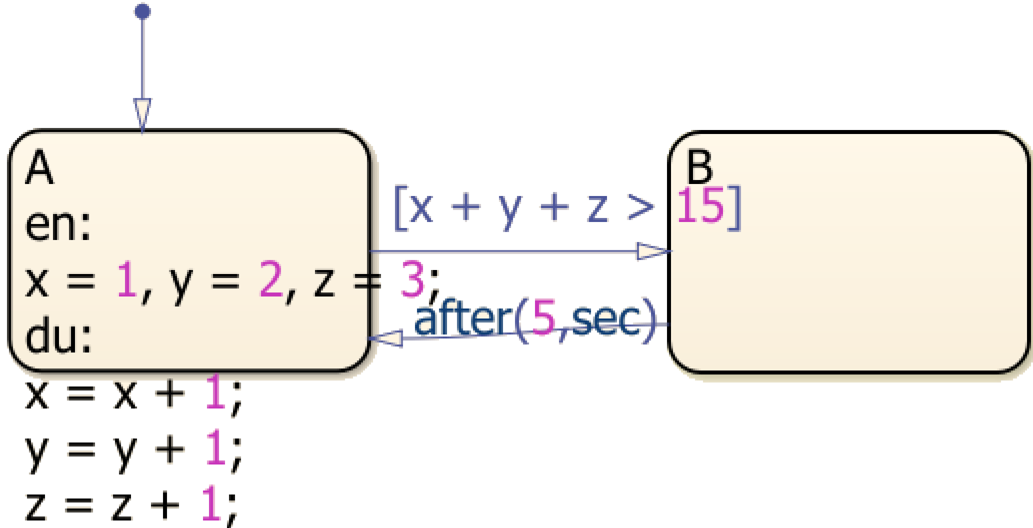
With Arrange Automatically, Stateflow arranges your charts to:

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

In the **Format** tab, click **Auto Arrange**.

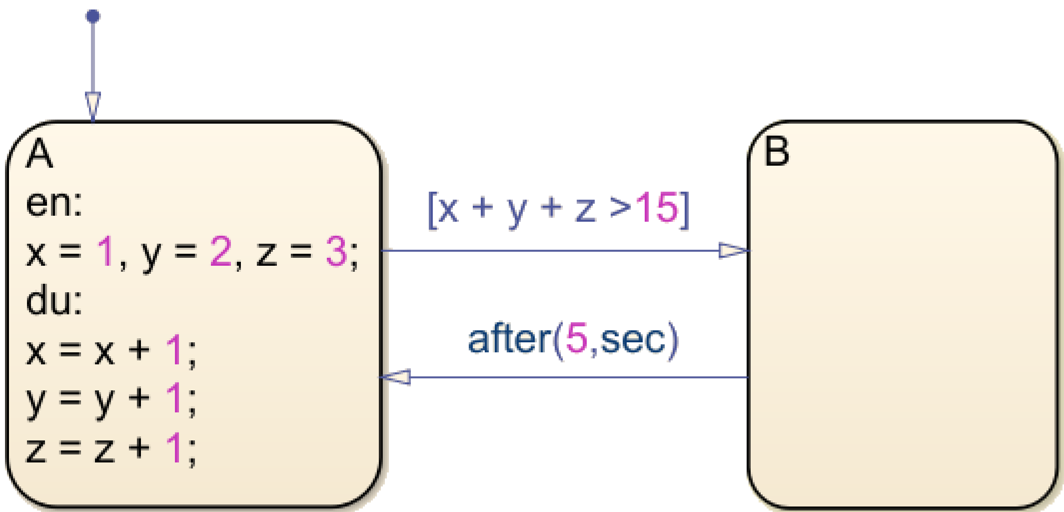
In this example, the chart has:

1. State actions that are outside of the boundary for state A.
2. A transition condition that overlaps state B.
3. A transition that is not horizontal.



After the layout has been automatically arranged:

1. The state actions are contained within state A.
2. The transition condition does not overlap into state B.
3. The lower transition is horizontal.



See Also

More About

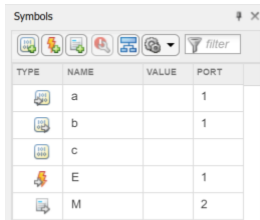
- "Add Stateflow Data" on page 10-2
- "Inspect and Modify Data and Messages While Debugging" on page 30-8
- "Execute and Unit Test Stateflow Chart Objects" on page 31-8

Manage Symbols in the Stateflow Editor

In the **Symbols** pane, you can view and manage data, events, and messages while working in the Stateflow Editor. In the **Modeling** tab, select **Symbols Pane**.

From the **Symbols** pane you can:

- Add and delete data, events, and messages.
- Set the object type and scope.
- Change the port number.
- Edit the name of an object and update all references to the object name in the chart.
- Undo and redo changes in type, name, and port number.
- Detect unused objects.
- Detect and fix unresolved objects.
- Trace between objects in the window and where the objects are used in the chart.
- View and edit object properties in the **Property Inspector**.



The rows in the **Symbols** pane display object hierarchy. Expand an object in the window to see data, events, and messages parented by that object. By default, all the nongraphical objects in a chart are listed in the window. To view only the objects that are used at the current level of hierarchy and below, select the icon. To search for specific symbols, type in the Filter search box .

Add and Modify Data, Events, and Messages

To add a nongraphical object to a Stateflow block, in the **Symbols** pane:

1 Select one of these icons.

Icon	Description
	Add data
	Add event
	Add message

2 In the row for the new object, under **TYPE**, choose the object type.
3 Edit the name of the object.
4 For input and output objects, under **PORT**, choose a port number.

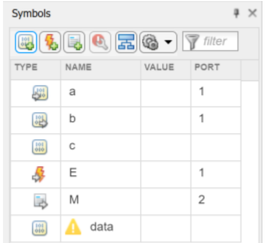
5 To view the object in the **Property Inspector**, right-click the object and select `Inspect`.
6 In the **Property Inspector**, modify the object properties.

After you add objects through the **Symbols** pane, the objects appear as unused until you reference them in your Stateflow design.

In the **Symbols** pane, you can modify the name, type, and port number of Stateflow objects. Edit the name of objects in the **NAME** field. When you rename an object, select **Shift+Enter** to rename all references to the object throughout the chart. To change the type or port number of an object, click the corresponding field and select from the available options. To delete an object, right-click the object and select `Delete`.

Detect Unused Data in the Symbols Pane

The **Symbols** pane indicates unused data, messages, functions, and events with a yellow warning icon. You can resolve undefined symbols by using the **Symbols** pane or the Symbol Wizard.



The **Symbols** pane does not detect inputs and outputs of MATLAB functions or parameters in atomic subcharts.

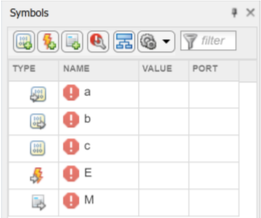
To delete unused objects, right-click the object in the **Symbols** pane and select **Delete**. By removing objects that have no effect on simulation, you can reduce the size of your model. In this chart, after you add `data`, it first appears as unused. After you reference `data` in the chart, the warning sign disappears.

Resolve Symbols Through the Symbols Pane

Symbols that appear in your chart but that you have not added as data, events, or messages are undefined or unresolved. As you edit your chart, Stateflow detects undefined symbols and marks them in the **Symbols** pane with a red error icon.

For each undefined symbol, the **Symbols** pane displays the class and scope inferred from the usage in the chart. You can resolve undefined symbols individually or collectively.

- To define a symbol with the inferred class and scope, click the error icon and select **Fix**.
- To define a symbol with a different class or scope, select another combination of class and scope from the **TYPE** drop-down list.
- To resolve all of the undefined symbols with their inferred classes and scopes, click the **Resolve undefined symbols** button.

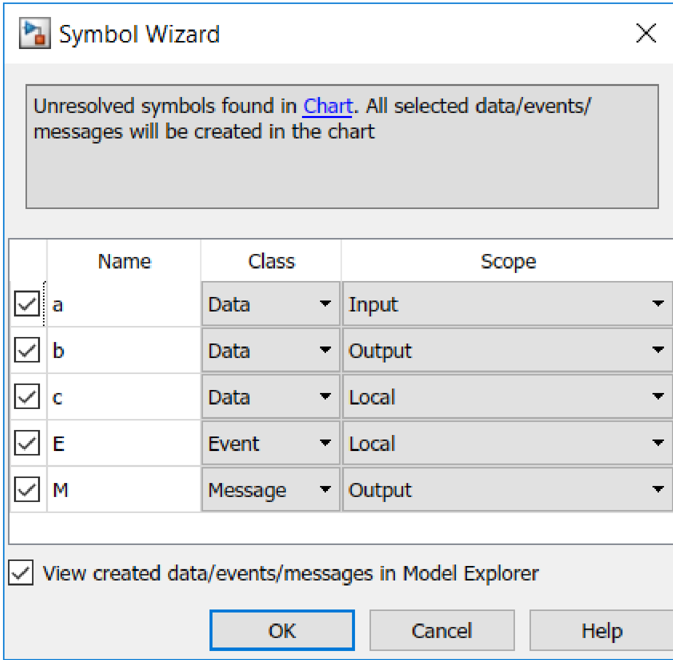


Resolve Symbols Through the Symbol Wizard

If your chart contains any undefined symbols when you update the chart, update the model, or simulate the model, the Symbol Wizard opens and lists the undefined symbols. For each undefined symbol, the **Class** and **Scope** columns display the class and scope inferred from the usage in the chart. You can accept, modify, or reject each symbol definition that the Symbol Wizard suggests.

- To accept a definition with the inferred class and scope, select the check box in front of the symbol.
- To modify a definition, select a different class or scope from the **Class** or **Scope** drop-down lists.
- To reject a definition, clear the check box in front of the symbol.

After you edit the symbol definitions, add the symbols to the Stateflow hierarchy by clicking **OK**.



Detect Symbol Definitions in Custom Code

Detection of symbols defined in custom code depends on the model configuration parameter **Import custom code**.

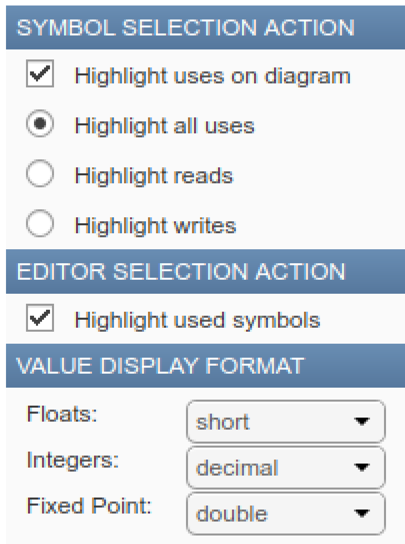
- If you select **Import custom code**, the Stateflow parser tries to find unresolved chart symbols in the custom code. If the custom code does not define these symbols, they appear in the Symbol Wizard.
- If you do not select **Import custom code**, the Stateflow parser considers unresolved data symbols in the chart as defined in the custom code. If the custom code does not define these symbols, simulating and generating code from the model results in an error.

The **Import custom code** option is not available for charts that use MATLAB as the action language. For more information, see "Import custom code" (Simulink).

Trace Data, Events, and Messages

Stateflow provides traceability between the chart and nongraphical symbols. When you select a symbol in the **Symbols** pane, Stateflow highlights sections of the chart that access that symbol. When you select an object in your chart, Stateflow highlights the symbols that the object uses.

To control when the objects and symbols are highlighted, select the preference button . A drop-down menu appears.



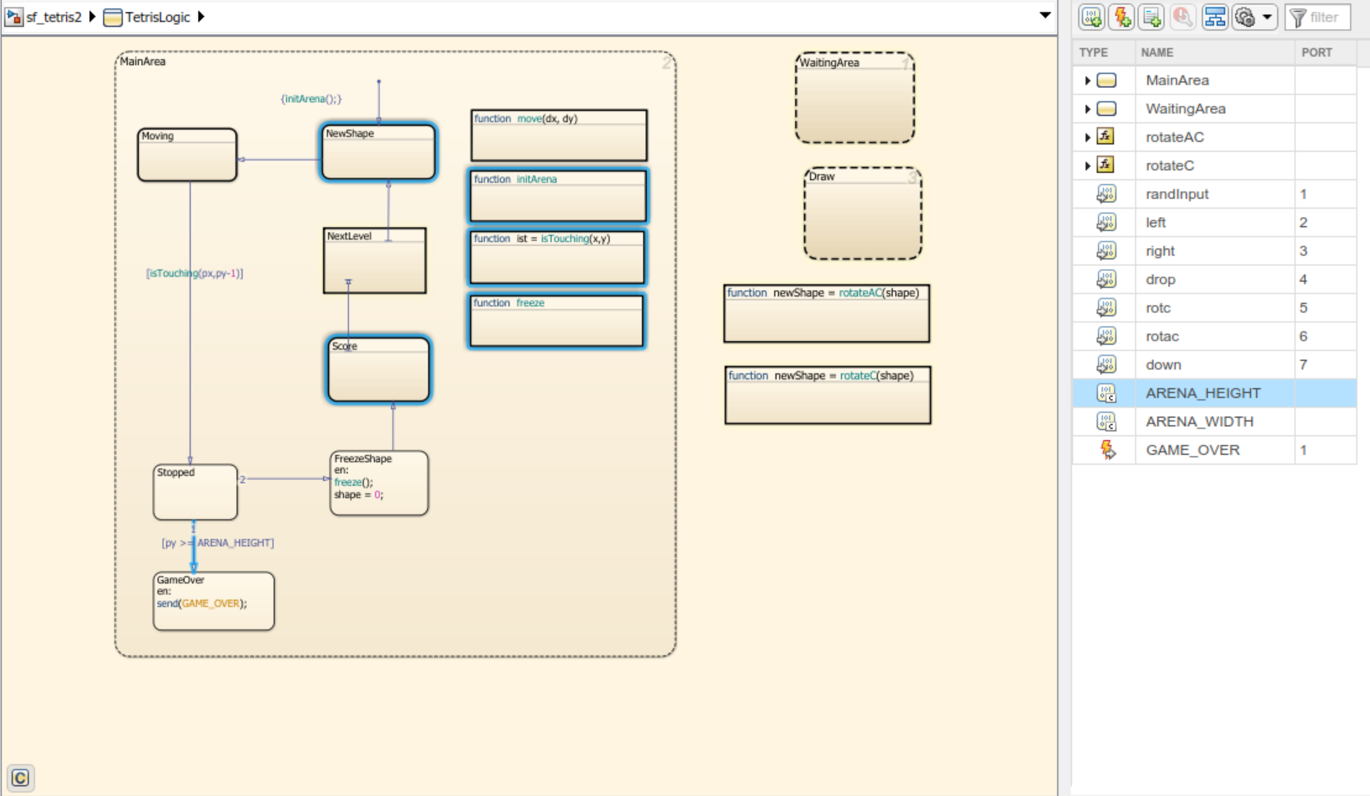
For Stateflow to highlight symbols in the **Symbols** pane that an object uses, select **Highlight used symbols**. If you want Stateflow to highlight objects in the chart that use an symbol, select **Highlight uses on diagram**. With **Highlight uses on diagram**, you can choose to highlight:

- All uses of the symbol in your chart.
- Objects from where the symbol is read.
- Objects to where the symbol is written.

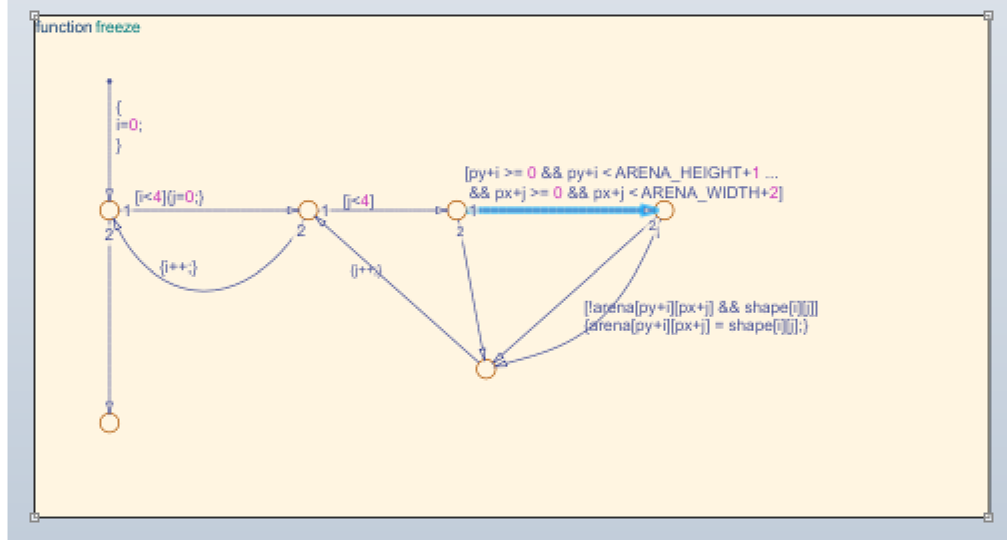
For example, open the model `sf_tetris2`.

```
openExample("stateflow/TetrisExample")
```

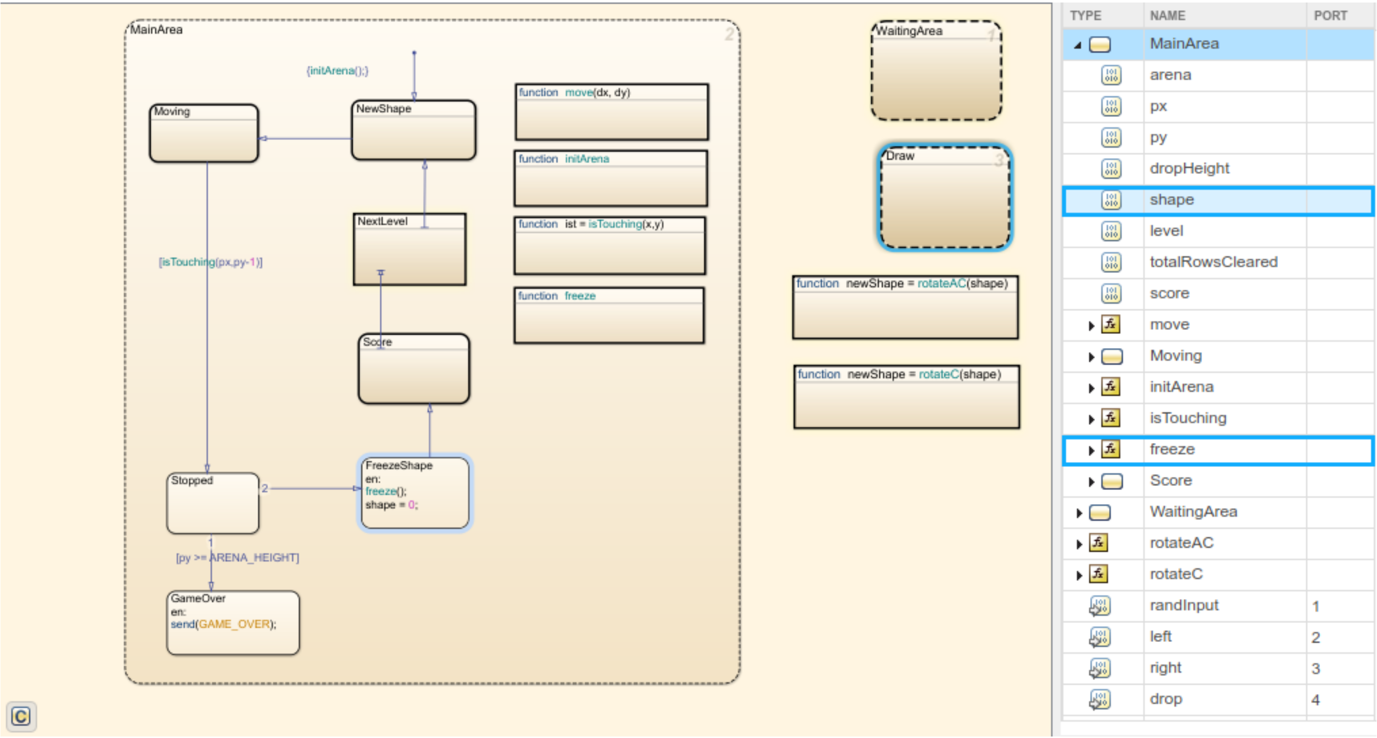
Double-click the chart `TetrisLogic`. In the **Symbols** pane, when you select constant `ARENA_HEIGHT`, the states and functions that use `ARENA_HEIGHT` are highlighted.



To see the uses of the constant `ARENA_HEIGHT`, open the function `freeze`.



You can also select a graphical object such as a state, transition, or function in the chart and view the symbols that the object uses. For example, in the chart `TetrisLogic`, expand the symbol `MainArea` in the **Symbols** pane. If you select the state `FreezeShape` in the chart, then the local data `shape` and the function `freeze()` are highlighted in the **Symbols** pane. This highlighting indicates that those objects are used inside the state `FreezeShape`.



When in debugging mode, the values of each data are displayed in the **VALUE** column of the **Symbols** pane. Stateflow updates the values periodically when the simulation is running. The value column highlights changes to data values as the changes occur. When the debugger is stopped at a breakpoint, you can update the initial value or change the value of a symbols in either the command prompt or the **Symbols** pane.

Data or Message	Update Initial Value	Update During Debugging
Input	No	No
Output	Yes	Yes
Parameter	No	No
Constant	Yes	No
Data Store Memory	No	Yes
Local	Yes	Yes

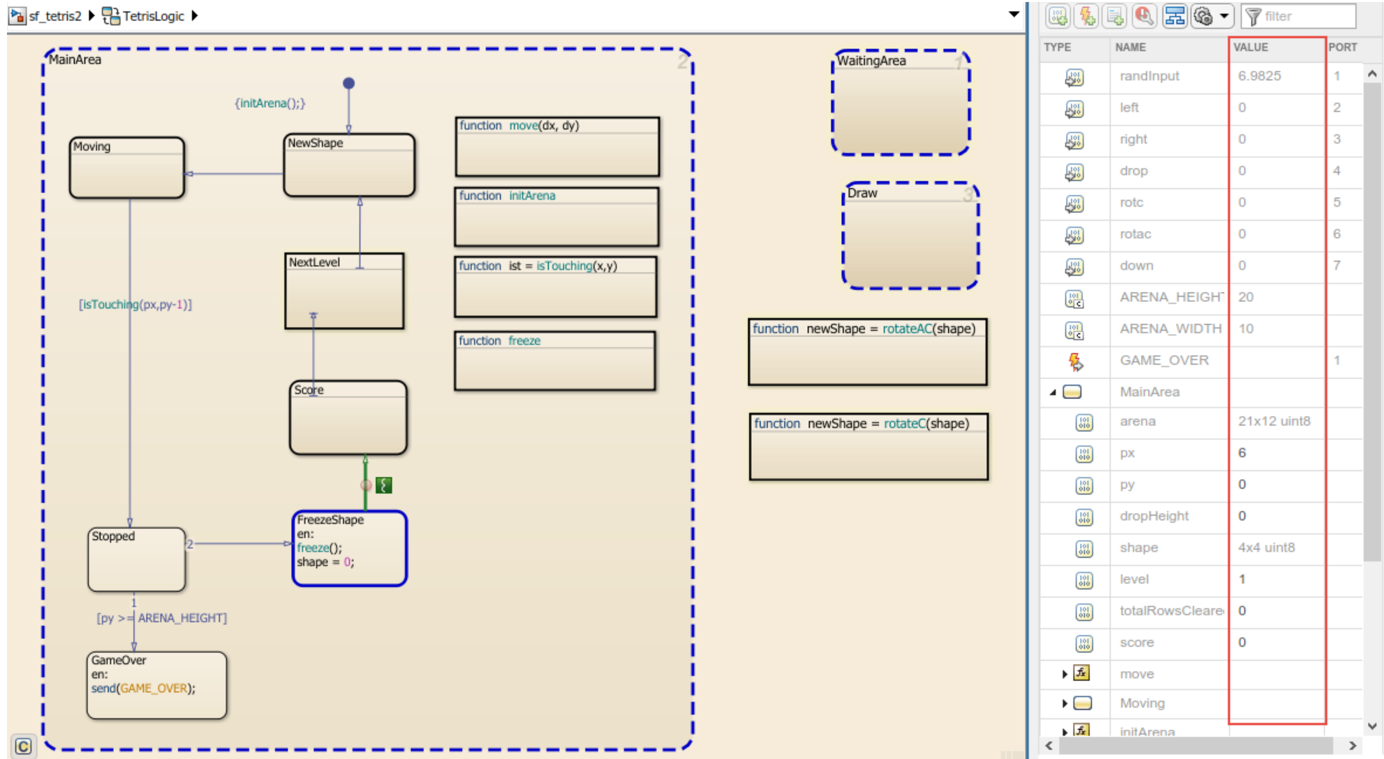
For bus elements, you can change the value of a symbols in either the command prompt or the **Symbols** pane.

Bus Element	Update Initial Value	Update During Debugging
Input	No	No
Output	No	Yes
Parameter	No	No
Constant	No	No
Data Store Memory	No	Yes

Bus Element	Update Initial Value	Update During Debugging
Local	No	Yes

In the **Symbols** pane multidimensional arrays appear as the data type and size of the array. If the array does not exceed more than 100 elements, hover over the symbol to view the elements. For arrays that contain more than 100 elements, view the elements by using the command prompt.

When simulation is paused, hover over messages in the canvas to view payloads in the queue. This is similar to the hover functionality on the canvas. For other non-scalar objects, the size and data type appear. To see these values, use the Watch window. See “Inspect and Modify Data and Messages While Debugging” on page 30-8 and “Track Data in the Watch List” on page 30-10.



Symbols Pane Limitations

- You cannot add data, events, or messages the **Symbols** pane if they are parented by a state or function. To add these types of objects, use the Model Explorer.
- When you modify the code in a MATLAB function, the changes are not updated in the **Symbols** pane until after you save the MATLAB function.
- You cannot undo or redo changes to input and output for MATLAB functions.
- You cannot recover deleted data, events, or messages from a state transition table.
- You cannot undo scope changes to data parented by graphical functions, MATLAB functions, and truth tables.
- You cannot undo renaming an object for truth tables.
- When you delete data for objects contained in a Simulink based state, the object remains in your Simulink based state and the data symbol is shown as undefined in the **Symbols** pane.

See Also

More About

- "Add Stateflow Data" on page 10-2
- "Set Data Properties" on page 10-5

Use the Model Explorer with Stateflow Objects

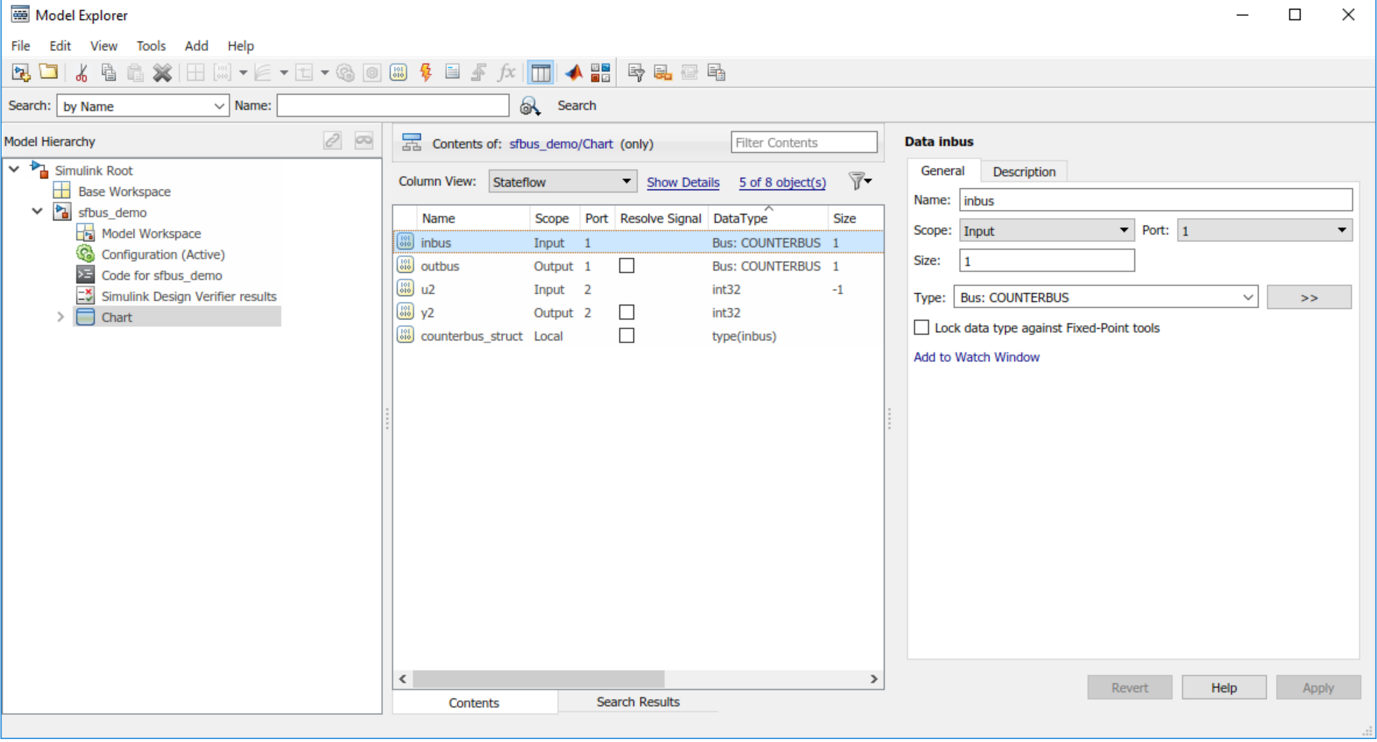
In this section...
“View Stateflow Objects in the Model Explorer” on page 25-22
“Edit Chart Objects in the Model Explorer” on page 25-23
“Add Data and Events in the Model Explorer” on page 25-23
“Rename Objects in the Model Explorer” on page 25-23
“Set Properties for Chart Objects in the Model Explorer” on page 25-23
“Move and Copy Data and Events in the Model Explorer” on page 25-24
“Change the Port Order of Input and Output Data and Events” on page 25-25
“Delete Data and Events in the Model Explorer” on page 25-25

View Stateflow Objects in the Model Explorer

You can use one of these methods for opening the Model Explorer:

- In the **Modeling** tab, select **Model Explorer**.
- Right-click an empty area in the chart and select **Explore**.

The Model Explorer appears something like this:



The main window has two panes: a **Model Hierarchy** pane on the left and a **Contents** pane on the right. When you open the Model Explorer, the Stateflow object you are editing appears highlighted in

the **Model Hierarchy** pane and its objects appear in the **Contents** pane. This example shows how the Model Explorer appears when opened from the chart.

The **Model Hierarchy** pane displays the elements of all loaded Simulink models, which includes Stateflow charts. A preceding plus (+) character for an object indicates that you can expand the display of its child objects by double-clicking the entry or by clicking the plus (+). A preceding minus (-) character for an object indicates that it has no child objects.

Clicking an entry in the **Model Hierarchy** pane selects that entry and displays its child objects in the **Contents** pane. A hypertext link to the currently selected object in the **Model Hierarchy** pane appears after the **Contents of:** label at the top of the **Contents** pane. Click this link to display that object in its native editor. In the preceding example, clicking the link `sfbus_demo/Chart` displays the contents of the chart in its editor.

Each type of object, whether in the **Model Hierarchy** or **Contents** pane, appears with an adjacent icon. Subcharted objects (states, boxes, or graphical functions) appear altered with shading.

The display of child objects in the **Contents** pane includes properties for each object, most of which are directly editable. You can also access the properties dialog box for an object from the Model Explorer. See “Set Properties for Chart Objects in the Model Explorer” on page 25-23 for more details.

Edit Chart Objects in the Model Explorer

To edit a chart object that appears in the **Model Hierarchy** pane of the Model Explorer:

1. Right-click the object.
2. Select **Open** from the context menu.

   The selected object appears highlighted in the chart.

Add Data and Events in the Model Explorer

To add data or events using the Model Explorer, see the following links:

- “Add Data Through the Model Explorer” on page 10-3
- “Add Events Through the Model Explorer” on page 12-3

Rename Objects in the Model Explorer

To rename a chart object in the Model Explorer:

1. Right-click the object row in the **Contents** pane of the Model Explorer and select **Rename**.

   The name of the selected object appears in a text edit box.
2. Change the name of the object and click outside the edit box.

Set Properties for Chart Objects in the Model Explorer

To change the property of an object in the **Contents** pane of the Model Explorer:

1. In the **Contents** pane, click in the row of the displayed object.
2. Click an individual entry for a property column in the highlighted row.

   - For text properties, such as the Name property, a text editing field with the current text value overlays the displayed value. Edit the field and press the **Enter** key or click anywhere outside the edit field to apply the changes.
   - For properties with enumerated entries, such as the Scope, Trigger, or Type properties, select from a drop-down combo box that overlays the displayed value.
   - For Boolean properties (properties that are set on or off), select or clear the box that appears in place of the displayed value.

To set all the properties for an object displayed in the **Model Hierarchy** or **Contents** pane of the Model Explorer:

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

   The properties dialog box for the object appears.
2. Edit the appropriate properties and click **Apply** or **OK**.

To display the properties dialog box dynamically for the selected object in the **Model Hierarchy** or **Contents** pane of the Model Explorer:

1. Select **View** > **Show Dialog Pane**.

   The properties dialog box for the selected object appears in the far right pane of the Model Explorer.

Move and Copy Data and Events in the Model Explorer

Note If you move an object to a level in the hierarchy that does not support the **Scope** property for that object, the **Scope** automatically changes to **Local**.

To move data and event objects to another parent:

1. Select the data or event to move in the **Contents** pane of the Model Explorer.

   You can select a contiguous block of items by highlighting the first (or last) item in the block and then using **Shift** + click for highlighting the last (or first) item.
2. Click and drag the highlighted objects from the **Contents** pane to a new location in the **Model Hierarchy** pane to change its parent.

   A shadow copy of the selected objects accompanies the mouse cursor during dragging. If no parent is chosen or the parent chosen is the current parent, the mouse cursor changes to an X enclosed in a circle, indicating an invalid choice.

To cut or copy the selected data or event:

1. Select the event or data to cut or copy in the **Contents** pane of the Model Explorer.
2. In the Model Explorer, select **Edit** > **Cut** or **Edit** > **Copy**.

If you select **Cut**, the selected items are deleted and then copied to the clipboard for copying elsewhere. If you select **Copy**, the selected items are left unchanged.

You can also right-click a single selection and select **Cut** or **Copy** from the context menu. The Model Explorer also uses the keyboard equivalents of **Ctrl+X** (Cut) and **Ctrl+C** (Copy) on a computer running the UNIX® or Windows® operating system.

3. Select a new parent object in the **Model Hierarchy** pane of the Model Explorer.
4. Select **Edit** > **Paste**. The cut items appear in the **Contents** pane of the Model Explorer.

   You can also paste the cut items by right-clicking an empty part of the **Contents** pane and selecting **Paste** from the context menu. The Model Explorer also uses the keyboard equivalent of **Ctrl+V** (Paste) on a computer running the UNIX or Windows operating system.

Change the Port Order of Input and Output Data and Events

Input data, output data, input events, and output events each have numerical sequences of port index numbers. You can change the order of indexing for event or data objects with a scope of **Input** or **Output** in the **Contents** pane of the Model Explorer as follows:

1. Select one of the input or output data or event objects.
2. Click the **Port** property for the object.
3. Enter a new value for the Port property for the object.

   The remaining objects in the affected sequence are automatically assigned a new value for their **Port** property.

Delete Data and Events in the Model Explorer

Delete data and event objects in the **Contents** pane of the Model Explorer as follows:

1. Select the object.
2. Press the **Delete** key.

   You can also select **Edit** > **Cut** or **Ctrl+X** from the keyboard to delete an object.

Visualize Chart Execution with the Activity Profiler

The Activity Profiler allows you to examine the behavior of your Stateflow chart. When you enable the Activity Profiler, after simulation, your Stateflow chart is highlighted to show which states were entered, transitions were taken, or functions were executed during the time that the simulation is running, also known as the simulation time. Additionally, you can see the duration of time that was spent in each state. To adjust the simulation time, in the Simulation tab, change the Stop time number.

With the Activity Profiler, you can quickly

1. Assess the behavior of your chart.
2. View states and transitions of your chart that are never entered or taken.

The Activity Profiler is not supported in referenced models or with fast restart mode.

Debug with the Activity Profiler

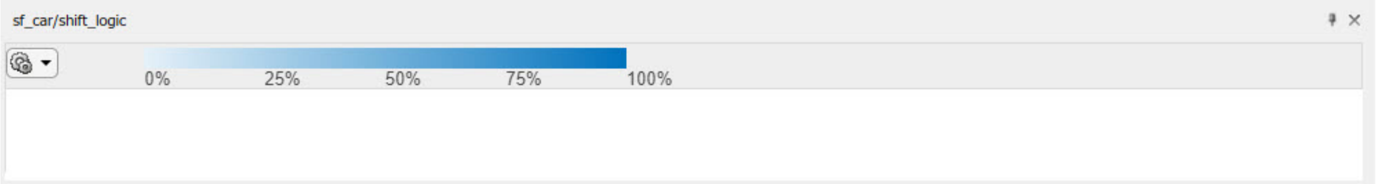
You can use the Activity Profiler as a complement to the Stateflow debugger. With the Activity Profiler, you can immediately see areas in your chart that were never reached or were constantly active, which can result in a faster debugging process. Additionally, you can view atomic states to see how often they are entered and how often their corresponding transitions are taken. With this information, it is easier to identify transition logic issues and solve problem areas in your chart such as:

- Transitions that are taken too often and serve no purpose
- Charts that are activated too often and slow down performance, such as an unexpected loop
- A bottleneck, such as a controller state that has multiple incoming transitions

After finding the specific problem areas, you can then set breakpoints to debug your chart. Without the Activity Profiler, you would have to set many breakpoints within your chart to pinpoint the problem area.

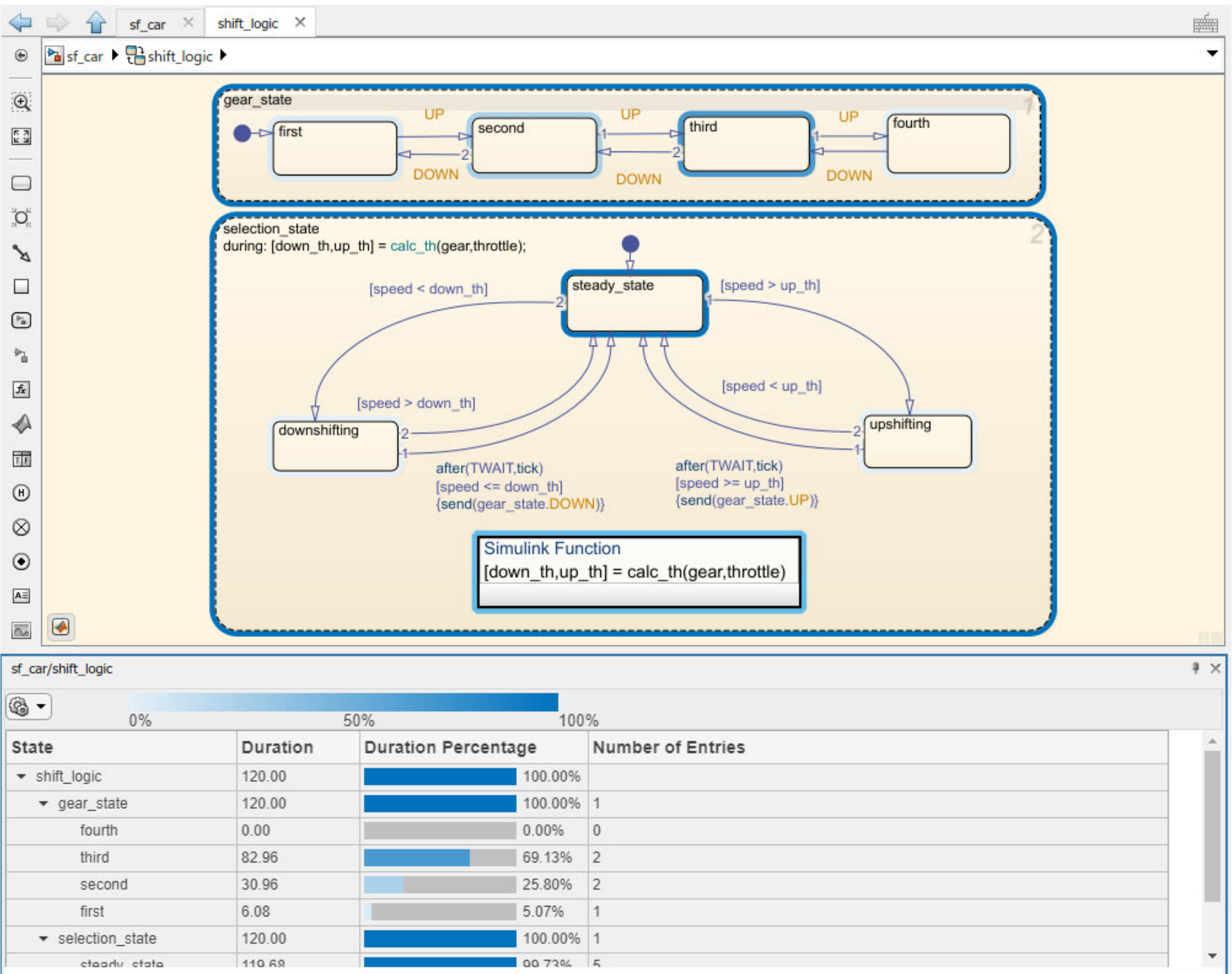
Enable the Activity Profiler

To enable the Activity Profiler, in the Stateflow editor, in the **Debug** tab, click **Activity Profiler**. On the bottom of the Stateflow Editor, the Activity Profiler pane appears below your chart.



To show how each state, transition, and function is executed, click **Run**. In the Stateflow editor, your chart is highlighted to show how many times a state is entered, a transition is taken, or a function is executed. The Activity Profiler pane shows four columns: State, Duration, Duration Percentage, and Number of Entries.

The Activity Profiler pane remains empty any time that the simulation is running, or paused for debugging. Once the simulation is complete the Activity Profiler pane populates and the canvas highlighting appears.



The **State** column lists the states in your Stateflow chart, along with their child states. The **Duration** column displays the duration of time (in seconds) spent in each state during the simulation. The **Duration Percentage** column shows a bar that represents what percentage of the runtime was spent in each state in relation to the parent chart. The **Number of Entries** column is the number of times each state was entered during simulation time.

You can also hover over a state or transition to see this data.

After running the simulation, you can toggle the highlighting and the Activity Profiler table off and on. In the Debug tab, open the Activity Profiler drop-down.

To turn off the canvas highlighting, clear `Canvas Highlight`. To turn off the Activity Profiler table, clear `Table View`.

Activity Profiler Preferences

The Activity Profiler is customizable. You have the option to highlight specific group of objects in the chart or change the color scheme. Specifying the highlighting allows you to focus on only one area. Changing the color scheme can help you visualize the data in different ways.

Highlighting Options

To change the highlighting options, in the Activity Profiler table, select the Activity Profiler preferences drop-down menu . Under **Canvas Highlight Options**, you can choose to enable highlighting for:

- States
- Transitions
- Functions

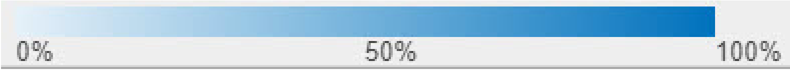
This is a global setting. If you change this setting in one model, all other models will only appear this way.

Color Scheme Options

The Stateflow Activity Profiler allows you to configure how to view the Activity Profiler through different color schemes. To change the color scheme, select the Activity Profiler preferences drop-down menu . Under **Color Scheme**, select a color. You can choose from the following color schemes:

- Blue
- Red
- Green
- HotCold
- Autumn
- Cool
- Jet
- Parula

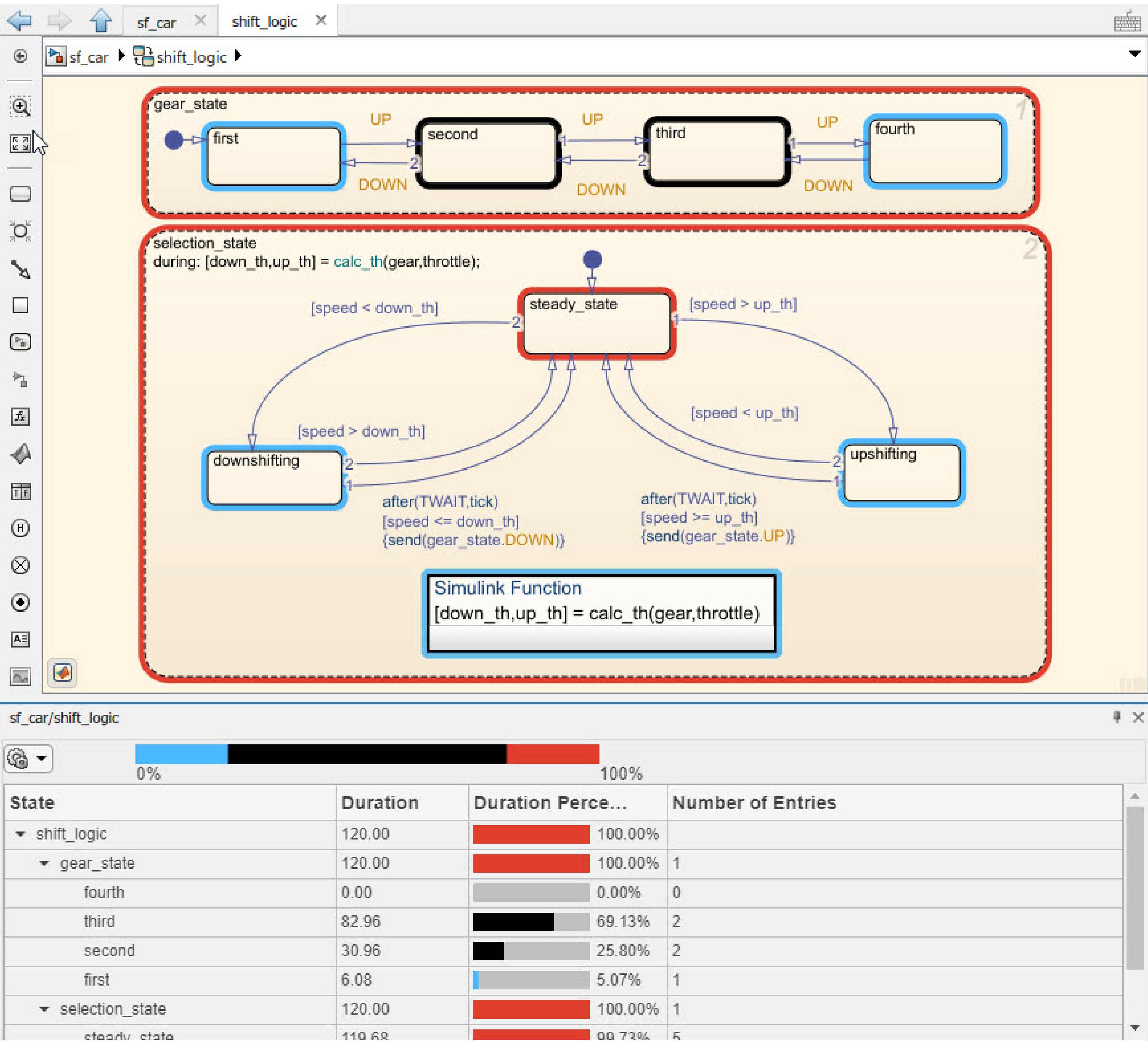
The table at the top of the Activity Profiler table shows how the colors appear on the map.



This legend for the `Blue` scheme shows how the color becomes gradually darker as the state is entered more often.

`HotCold` is a highlighting scheme that is used to visualize the top 80% and low 20% of objects in a Stateflow chart.

The legend for `HotCold` shows how states or transitions that are entered: 0% to 20% relative to the parent chart are highlighted blue. States that are entered 80% to 100% are highlighted in red.



In this chart, you can see that the states **upshifting**, **downshifting**, and **first** were active for 20% or less of the total runtime. The states **gear_state**, **selection_state**, and **steady_state** were all active for 80% or more of the runtime. The states **second** and **third** were active between 20 and 80% of the runtime. The state **fourth** was not entered at all and remains not highlighted.

Explore

To view only subcharts and their child states, in the Activity Profiler table, right-click on the subchart name and select **Explore**. The Activity Profiler table adjusts to only include data about that subchart

and its child states. Once you select the subchart as the current scope for the Activity Profiler pane, the duration percentage is in relation to the selected subchart.

See Also

More About

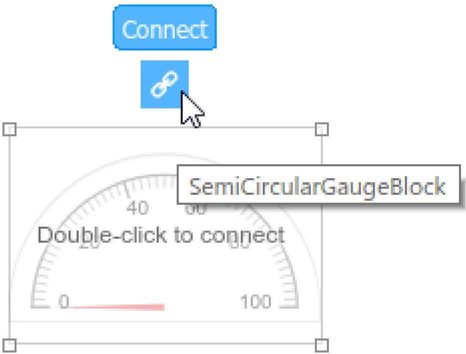
- “Manage Symbols in the Stateflow Editor” on page 25-14
- “Detect Common Modeling Errors During Simulation” on page 30-35
- “Set Data Properties” on page 10-5

Connect Dashboard Blocks to Stateflow

Dashboard blocks help you to control and visualize your Stateflow chart during simulation and while your simulation is paused. Dashboard blocks can be used to bind to Stateflow in order to:

- Monitor self, child, and leaf activity of a state.
- Monitor local and output data within states, transitions, or graphical functions.

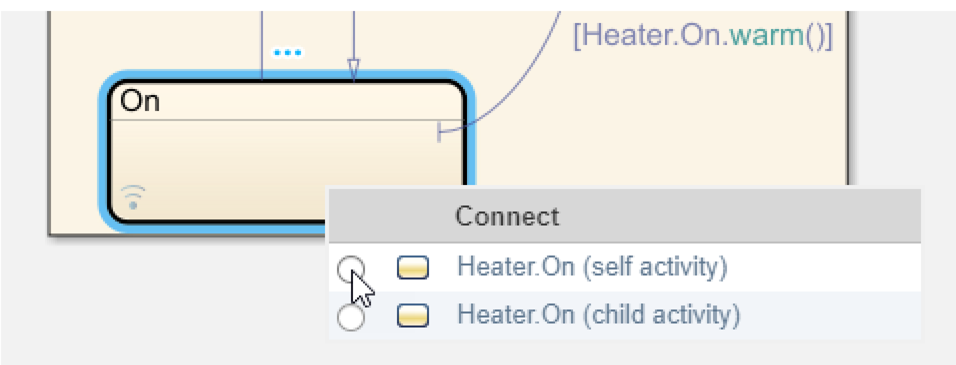
To connect a dashboard block to data or to a Stateflow state, point to the dashboard block. Above the block, a connect button appears.



Click the Connect button, and navigate to the Stateflow object you want to connect to.

Note The **Double-click to connect** feature is not supported in Stateflow.

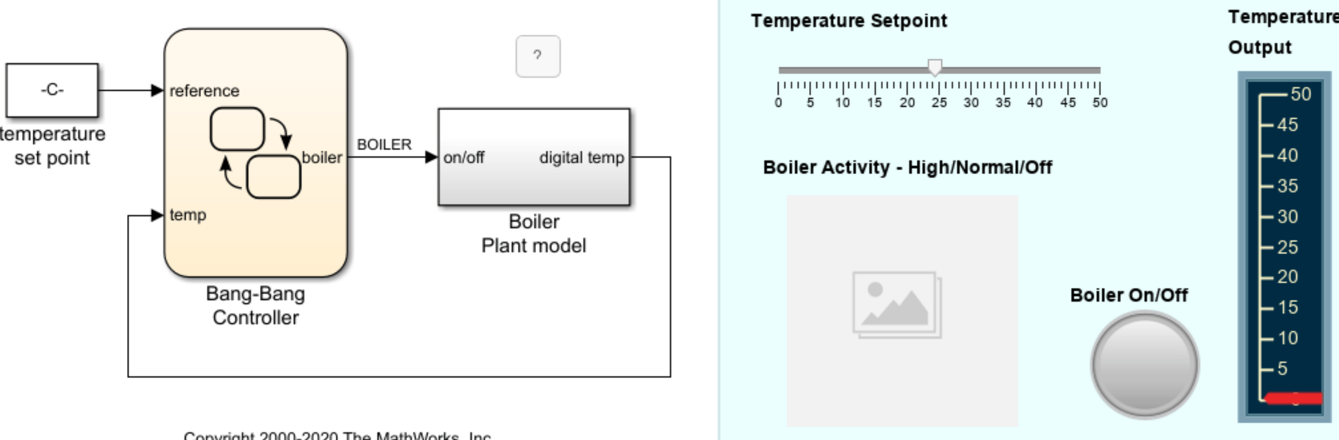
Click a state or transition, and choose which activity or data you want to connect.



Dashboard block connection is not supported for MATLAB Functions, Simulink Functions, or truth tables. For more information on dashboard blocks, see “Control Simulations with Interactive Displays” (Simulink).

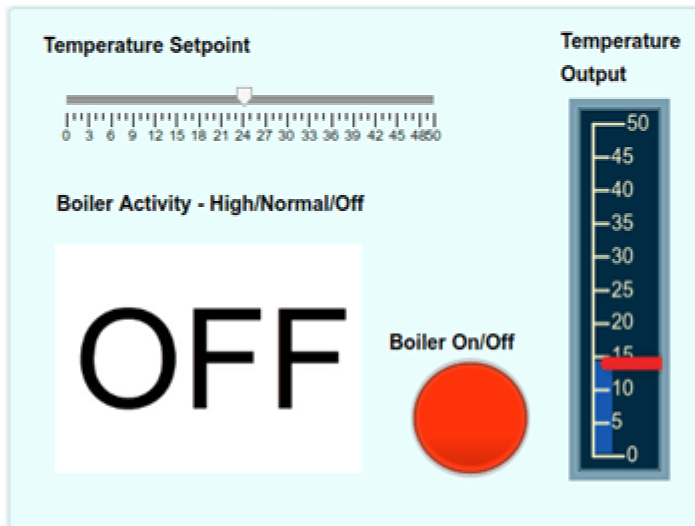
Monitor a Boiler with Dashboard Blocks

In this Stateflow chart, dashboard blocks are used to control the temperature setpoint for a boiler, visualize when the boiler is on or off and what mode it is operating in.

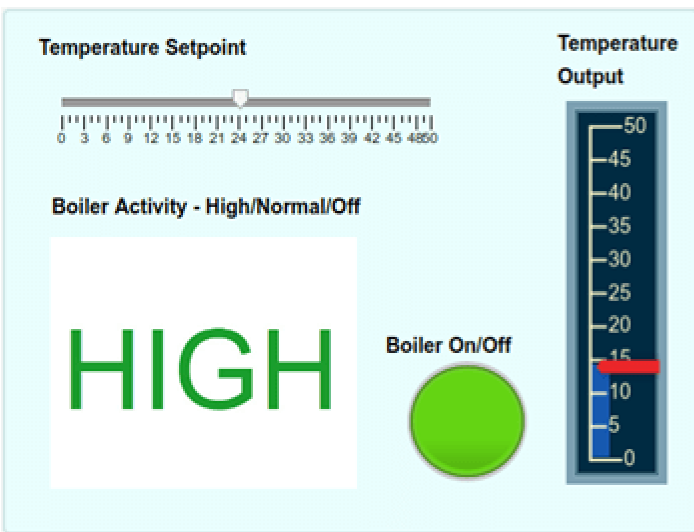


Dashboard blocks are also used to visualize the temperature output by the boiler, which should match the temperature setpoint.

When you simulate the model, the LED light flashes to show if the boiler is off or on. A red light indicates that the boiler is off.



A green light indicates that the boiler is on. The `Boiler Activity - Hight/Normal/Off` block shows you if the boiler is operating in high or normal mode.



See Also

More About

- “Decide How to Visualize Simulation Data” (Simulink)
- “Model Battery Management with Custom Code” on page 28-14

Reuse Charts in Models with Chart Libraries

In Simulink, you can create your own block libraries as a way to reuse the functionality of blocks or subsystems in one or more models. Similarly, you can reuse a set of Stateflow algorithms by encapsulating the functionality in a chart library.

As with other Simulink block libraries, you can specialize each instance of chart library blocks in your model to use different data types, sample times, and other properties. Library instances that inherit the same properties can reuse generated code.

For more information about Simulink block libraries, see “Custom Libraries” (Simulink).

Create Specialized Chart Libraries for Large-Scale Modeling

1. Add Stateflow charts with polymorphic logic to a Simulink model.

   Polymorphic logic is logic that can process data with different properties, such as type, size, and complexity.
2. Configure the charts to inherit the properties you want to specialize.

   For a list, see “Customize Properties of Library Blocks” on page 25-34.
3. Optionally, customize your charts using masking.
4. Simulate and debug your charts.
5. In Simulink, create a library model. In the **Simulation** tab, select **New** > **Library**
6. Copy or drag the charts into a library model.

Customize Properties of Library Blocks

You can customize instances of Stateflow library blocks by allowing them to inherit any of the following properties from Simulink.

Property	Inherits by Default?	How to Specify Inheritance
Type	Yes	Set the data type property to **Inherit: Same as Simulink**.
Size	Yes	Set the data size property to **-1**.
Complexity	Yes	Set the data complexity property to **Inherited**.
Limit range	No	Specify minimum and maximum values as Simulink parameters. For example, if minimum value = `aParam` and maximum value = `aParam` + 3, different instances of a Stateflow library block can resolve to different `aParam` parameters defined in their parent mask subsystems.

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

1. Events parented by a library chart are invalid. The Stateflow parser flags such events as errors.
2. To include a linked library chart within another library chart, the two library charts must be in separate libraries.
3. To include a linked library chart within a Simulink subsystem, first save the library chart within a subsystem and place that library subsystem in the Simulink subsystem.

Create a Mask to Share Parameters with Simulink

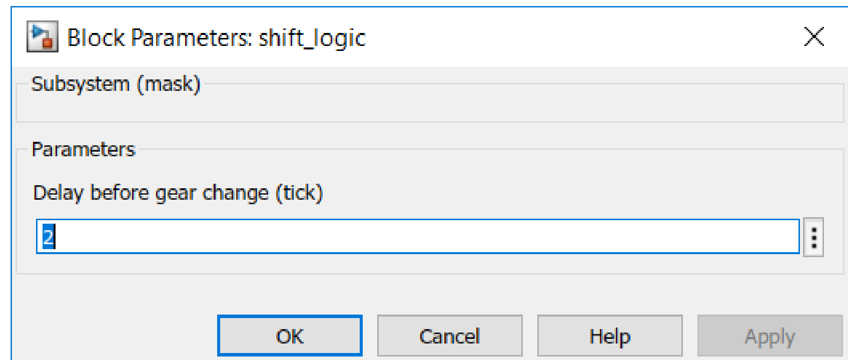
Creating masks for Stateflow charts, state transition tables, and truth tables simplifies how you use and share blocks in a Simulink model. The mask encapsulates the block by hiding the underlying logic and creates a user interface for the block. You can customize the block by:

- Changing the appearance with meaningful icons and ports.
- Creating a user interface for parameters.
- Adding customized documentation.

You decide which parameters to change through the mask user interface. You can provide meaningful descriptions of these parameters. For example, in the model `sf_car`, the `shift_logic` chart has a mask through which you can adjust the parameter `TWAIT`. To open the model, enter:

```
openExample("stateflow/AutomaticTransmissionWithActiveStateDataExample")
```

To open the Mask Parameters dialog box, double-click the Stateflow chart. This dialog box contains a parameter description "`Delay before gear change (tick)`" and a box to edit the value. This value is tied to the parameter `TWAIT` inside the mask. When you edit the value in this box, Stateflow assigns the new value to `TWAIT` during simulation.



You can create other types of user interfaces for the mask parameters, such as check boxes, context menus, and option buttons.

You can create masks on Stateflow blocks accessible from the Simulink library: charts, state transition tables, and truth tables. You cannot mask atomic subcharts, states, or any other objects within a chart.

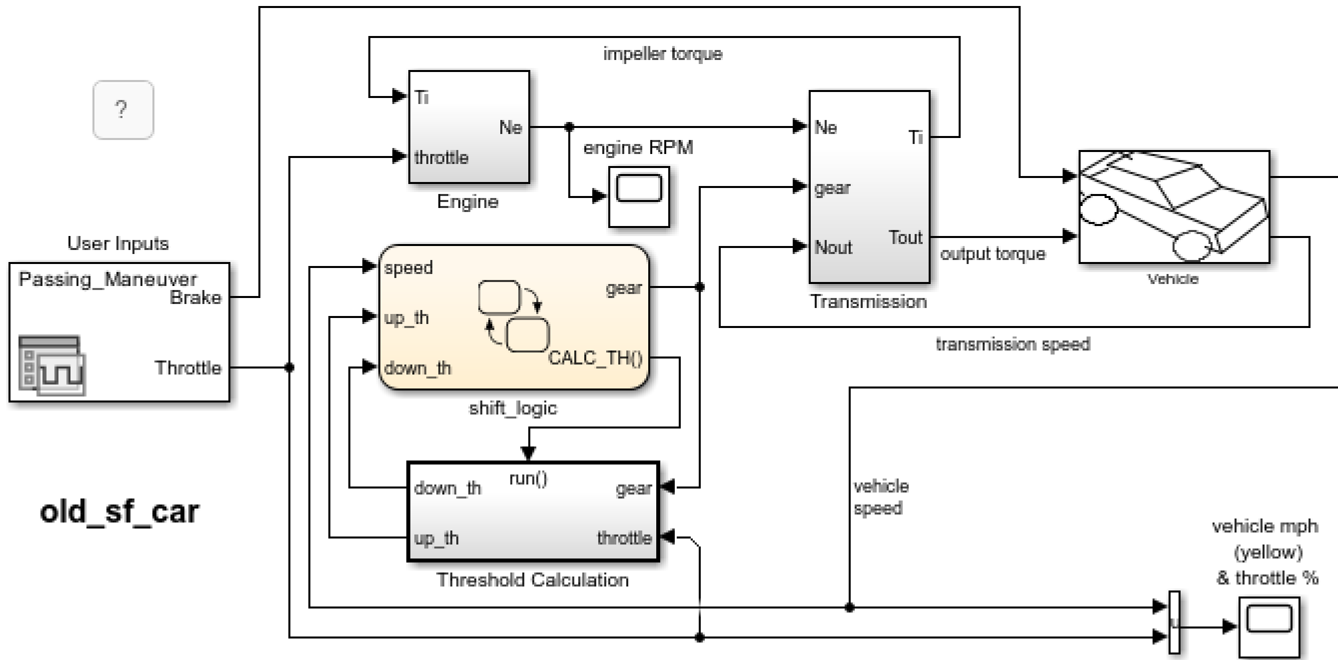
For more information, see “Create Block Masks” (Simulink).

Create a Mask for a Stateflow Chart

To create a mask for the Stateflow chart in the model `old_sf_car`:

1 Open the model.

```
openExample("stateflow/AutomaticTransmissionLegacyExample")
```



2 In the Simulink Editor, select the chart `shift_logic`.
3 Open the Mask Editor. In the **State Chart** tab, click **Create Mask**.

Add an Icon to the Mask

To customize the appearance of the block icon, use drawing commands or load an image. For more information, see “Draw Mask Icon Using Mask Drawing Commands” (Simulink).

1 In the Mask Editor, select the **Icon** tab.
2 In the edit box, enter:

```
image("shift_logic.svg")
```

3 Click **Save Mask**.

Add Parameters to the Mask

When you create a mask for a Stateflow block, you can define a custom interface for the block parameters. You provide access to the block parameters by defining corresponding parameters with the same name in the Mask Editor. A user interface to these parameters is then provided through a Mask Parameters dialog box. The mask parameters appear as editable fields in the Mask Parameters dialog box. Stateflow applies these values to the corresponding block parameters during simulation.

For example, the chart `shift_logic` has a parameter `TWAIT`. To add `TWAIT` as a parameter to the mask:

1 In the Mask Editor, select the **Parameters & Dialog** tab.
2 In the **Controls** pane, under **Parameter**, click the **Edit** icon.
3 Under **Prompt**, enter the prompt for the new mask parameter in the Mask Parameters dialog box:

```
Delay before gear change(tick)
```

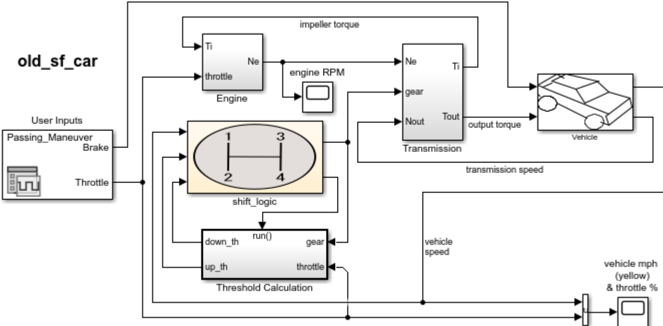
4 Under **Name**, enter the name of the parameter in the mask:

```
TWAIT
```

5 Click **Save Mask**.

View the New Mask

After creating a mask, the new icon for the `shift_logic` chart appears in the Simulink canvas. If you double-click the icon, the Mask Parameters dialog box opens. This dialog box has the prompt for the parameter `TWAIT`. The value in the edit box is assigned to the parameter `TWAIT` during simulation.



Look Under the Mask

You can view and edit the contents of a masked block by clicking the **Look inside mask** badge on the chart. The badge is a downward facing arrow in the lower-left corner of the chart. Alternatively, in the **State Chart** tab, click **Look Under Mask**. Looking under a mask does not unmask the block.

Edit the Mask

To edit a mask, in the **State Chart** tab, click **Edit Mask**. In the Mask Editor, you can modify the mask icon, change the parameters, or add documentation. To remove the mask, click **Unmask** in the lower corner of the Mask Editor. After you change a mask, click **Apply** to save the changes.

See Also

More About

- “Share Parameters with Simulink and the MATLAB Workspace” on page 10-32
- “Masking Fundamentals” (Simulink)
- “Mask Editor Overview” (Simulink)
- “Draw Mask Icon Using Mask Drawing Commands” (Simulink)

Structures and Bus Signals in Stateflow Charts

- "Access Bus Signals Through Stateflow Structures" on page 26-2
- "Index and Assign Values to Stateflow Structures" on page 26-7
- "Integrate Custom Structures in Stateflow Charts" on page 26-11

Access Bus Signals Through Stateflow Structures

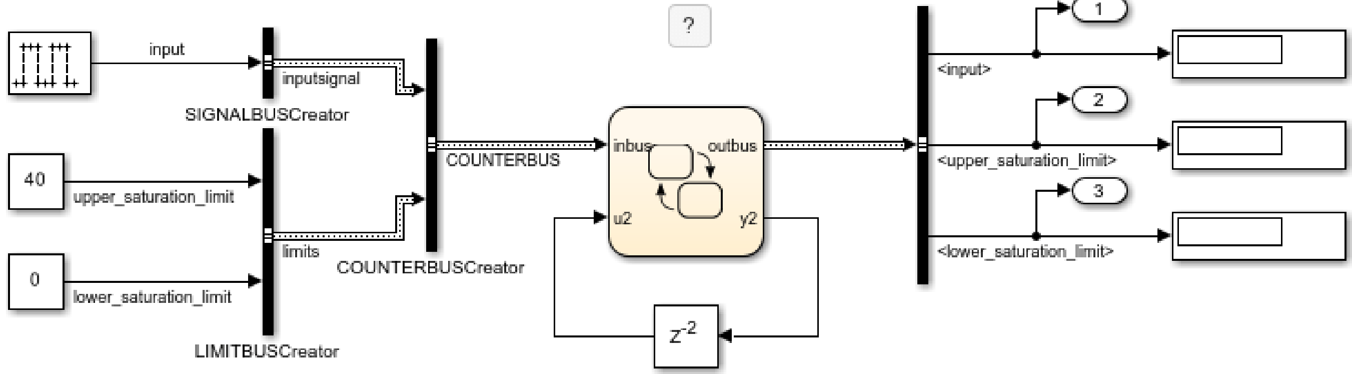
A Stateflow structure is a data type that you define from a `Simulink.Bus` object. Using Stateflow structures, you can bundle data of different size and type to create:

- Inputs and outputs that access Simulink bus signals from Stateflow charts, Truth Table blocks, and MATLAB Function blocks.
- Local data in Stateflow charts, truth tables, graphical functions, MATLAB functions, and boxes.
- Temporary data in Stateflow graphical functions, truth tables, and MATLAB functions.

For more information, see “Create and Specify Simulink.Bus Objects” (Simulink).

Example of Stateflow Structures

In this example, a Stateflow chart receives a bus input signal by using the structure `inbus` and outputs a bus signal from the structure `outbus`. The input signal comes from the Simulink Bus Creator block `COUNTERBUSCreator`, which bundles signals from two other Bus Creator blocks. The output structure `outbus` connects to a Simulink Bus Selector block. Both `inbus` and `outbus` derive their type from the `Simulink.Bus` object `COUNTERBUS`. For more information about this example, see “Integrate Custom Structures in Stateflow Charts” on page 26-11.



The elements of a Stateflow structure data type are called *fields*. Fields can be any combination of individual signals, muxed signals, vectors, and other structures (also called *substructures*). Each field has its own data type. The data type does not have to match the type of any other field in the structure. For example, in this model, each of the structures `inbus` and `outbus` has two fields:

- `inputsignal` is a substructure with one field, `input`.
- `limits` is a substructure with two fields, `upper_saturation_limit` and `lower_saturation_limit`.

Define Stateflow Structures

1. To define the structure data type, create a Simulink bus object in the base workspace, as described in “Create and Specify Simulink.Bus Objects” (Simulink).
2. Add a data object to the chart, as described in “Add Stateflow Data” on page 10-2.

To define temporary structures in truth tables, graphical functions, and MATLAB functions, add a data object *to your function*. For more information, see "Add Data Through the Model Explorer" on page 10-3.

3. Set the **Scope** property for the structure. Your choices are:
   - `Input`
   - `Output`
   - `Local`
   - `Parameter`
   - `Data Store Memory`
   - `Temporary` (Only in charts that use C as the action language)
4. Set the **Type** property for the structure. Depending on its scope, a Stateflow structure can have one of these data types.

<table>
<tr><th>Type</th><th>Description</th></tr>
<tr><td><code>Inherit: Same as Simulink</code></td><td>This option is available for input structures only. The input structure inherits its data type from the Simulink bus signal in your model that connects to it. The Simulink bus signal must be a nonvirtual bus. For more information, see "Virtual and Nonvirtual Buses" on page 26-5.

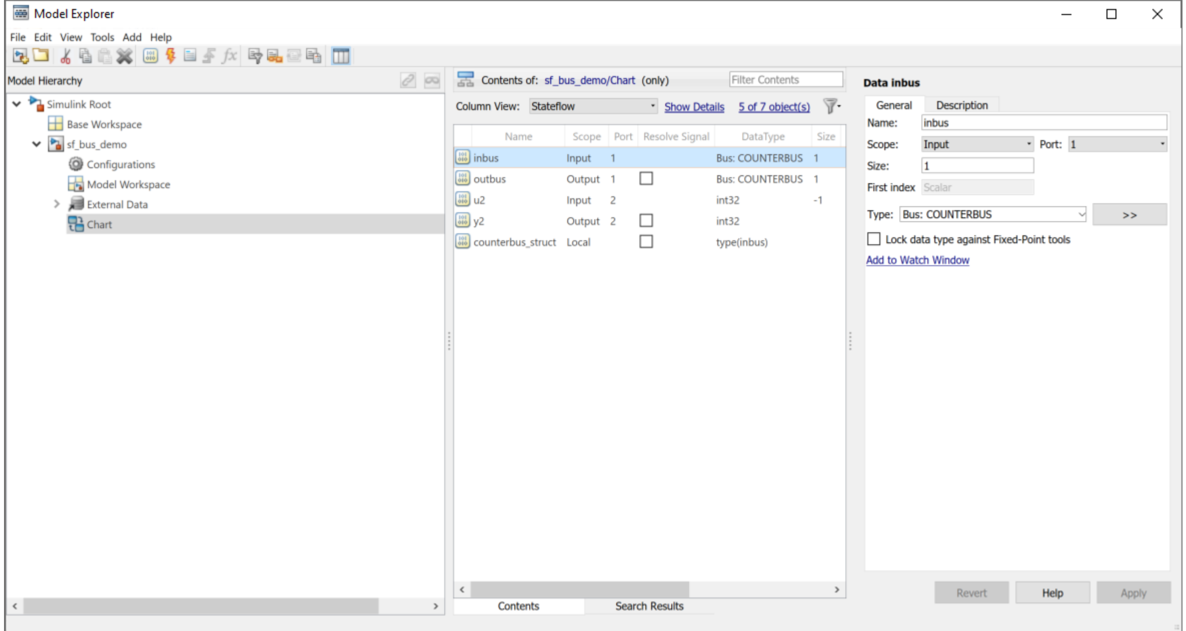
In the base workspace, specify a <code>Simulink.Bus</code> object with the same properties as the bus signal that connects to the Stateflow input structure. These properties must match:
• Number, name, and type of inputs
• Dimension
• Sample Time
• Complexity
• Sampling Mode

If the input signal comes from a Bus Creator block, in the Bus Creator dialog box, specify an appropriate bus object for Output data type field. When you specify the bus object, Simulink verifies that the properties of the <code>Simulink.Bus</code> object in the base workspace match the properties of the Simulink bus signal.</td></tr>
<tr><td><code>Bus: <object name></code></td><td>In the Type field, replace <<code>object name</code>> with the name of the <code>Simulink.Bus</code> object that defines the Stateflow structure.

For input or output structures, you are not required to specify the bus signal in your Simulink model that connects to the Stateflow structure. If you do specify a bus signal, its properties must match the <code>Simulink.Bus</code> object that defines the Stateflow structure.</td></tr>
</table>

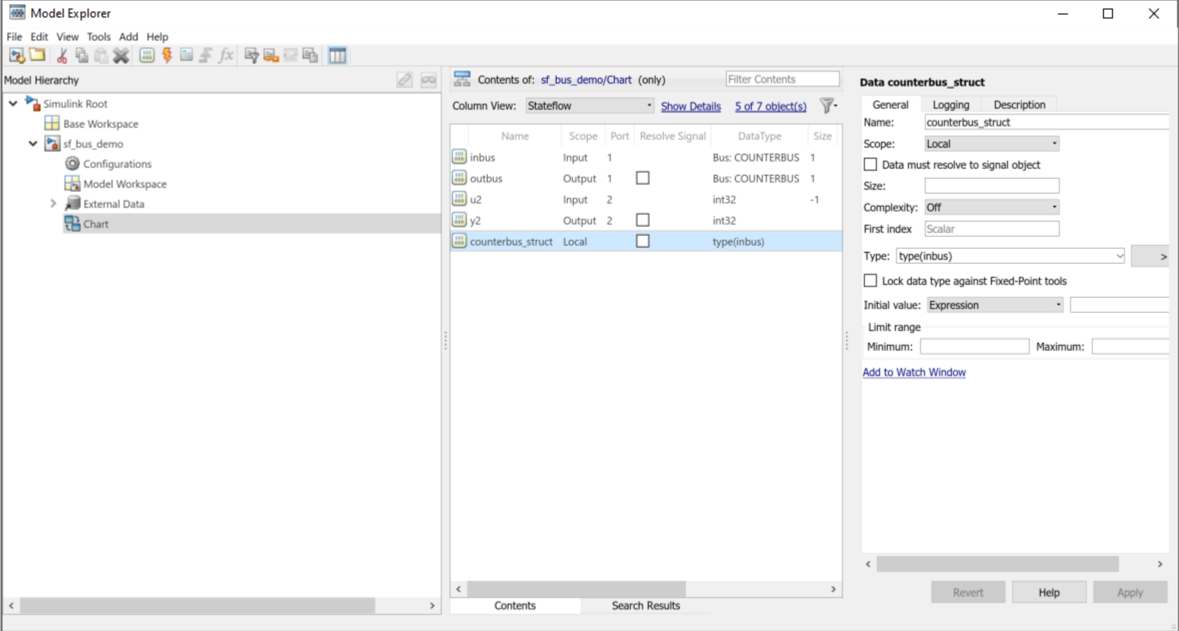
Type	Description
<date type expression>	In the **Type** field, replace <*data type expression*> with an expression that evaluates to a data type. For example: • Enter the name of the `Simulink.Bus` object that defines the Stateflow structure. • For structures with scopes other than `Output`, use the Stateflow `type` operator to copy the type of another structure. For more information, see “Specify Structure Types by Calling the type Operator” on page 26-4.

For example, in the `sf_bus_demo` model, the input structure `inbus` and the output structure `outbus` derive their type through a type specification of the form `Bus: COUNTERBUS`.



Specify Structure Types by Calling the type Operator

To specify structure types, you can use expressions that call the Stateflow `type` operator. This operator sets the type of one structure to the type of another structure in the Stateflow chart. For example, in the `sf_bus_demo` model, a `type` operator expression specifies the type of the local structure `counterbus_struct` in terms of the input structure `inbus`. Both structures are defined from the `Simulink.Bus` object `COUNTERBUS`. For more information, see “Derive Data Types from Other Data Objects” on page 10-23.



Virtual and Nonvirtual Buses

Simulink models support virtual and nonvirtual buses. Nonvirtual buses read their inputs from data structures stored in contiguous memory. Virtual buses read their inputs from noncontiguous memory. For more information, see “Composite Interface Guidelines” (Simulink).

Stateflow charts support only nonvirtual buses. Stateflow input structures can accept virtual bus signals and convert them to nonvirtual bus signals. Stateflow input structures cannot inherit properties from virtual bus signals. If the input to a chart is a virtual bus, set the **Type** property of the input structure through a type specification of the form `Bus: <object name>`.

Debug Structures

To debug a Stateflow structure, open the Stateflow Breakpoints and Watch window and examine the values of structure fields during simulation. To view the values of structure fields at the command line, use dot notation to index into the structure. For more information, see “Inspect and Modify Data and Messages While Debugging” on page 30-8.

Guidelines for Structure Data Types

- Define each structure from a `Simulink.Bus` object in the base workspace.
- Structures cannot have a constant scope.
- Structures of parameter scope must be tunable.

See Also

`Simulink.Bus`

More About

- “Index and Assign Values to Stateflow Structures” on page 26-7
- “Integrate Custom Structures in Stateflow Charts” on page 26-11
- “Add Stateflow Data” on page 10-2
- “Derive Data Types from Other Data Objects” on page 10-23
- “Inspect and Modify Data and Messages While Debugging” on page 30-8
- “Specify Bus Properties with Simulink.Bus Object Data Types” (Simulink)
- “Composite Interface Guidelines” (Simulink)

Index and Assign Values to Stateflow Structures

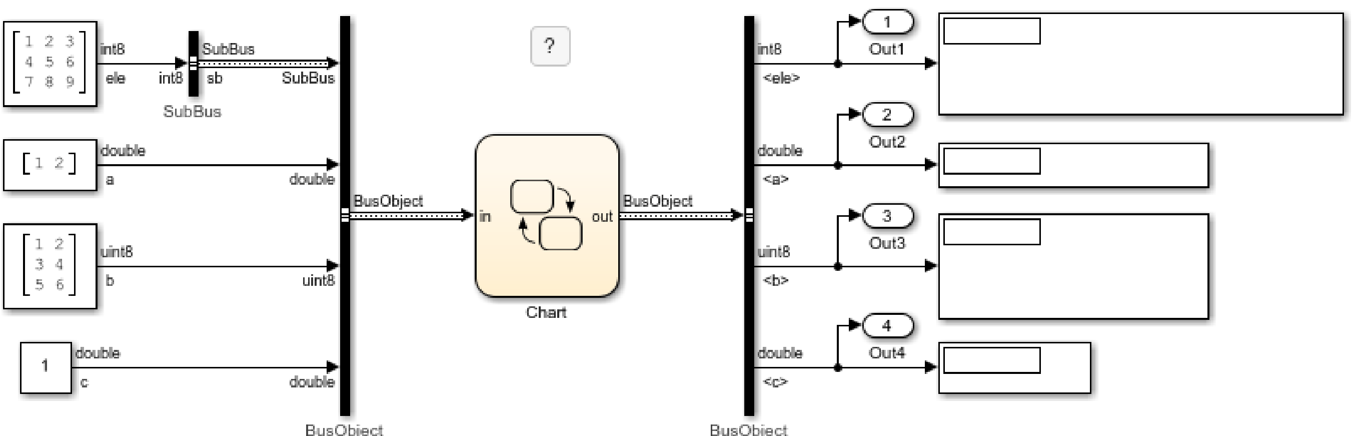
This example shows how to access and modify the contents of a Stateflow® structure or an array of Stateflow structures. A Stateflow structure is a data type that you define from a `Simulink.Bus (Simulink)` object. You can use Stateflow structures to bundle data of different sizes and types together into a single data object. For more information, see “Access Bus Signals Through Stateflow Structures” on page 26-2.

Index Substructures and Fields

To index substructures and fields of Stateflow structures, use dot notation. The first part of a name identifies the parent structure. Subsequent parts identify the children along a hierarchical path. The children can be individual fields or fields that contain other structures (also called substructures). The names of the fields of a Stateflow structure match the names of the elements of the `Simulink.Bus` object that defines the structure. When a field contains a vector, matrix, or array, you can access its elements by using the indexing notation supported by the action language of your chart.

For example, the chart in this model contains an input structure (`in`), an output structure (`out`), a local structure (`localbus`), and a local array of structures (`subBusArray`).

- The chart defines the input structure `in`, the output structure `out`, and the local structure `localbus` by using the `Simulink.Bus` object `BusObject`. These structures have four fields: `sb`, `a`, `b`, and `c`.
- The field `sb` is a substructure defined from the `Simulink.Bus` object `SubBus`. This substructure has one field called `ele`.
- The chart defines the local array of structures `subBusArray` by using the `Simulink.Bus` object `SubBus`. The array has size 4. Each element in the array is a structure with one field called `ele`.



This list illustrates expressions that combine dot notation and numeric indices based on the structure specifications for this example:

- `in.c` — Field `c` of the input structure `in`
- `in.a(1)` — First element of the vector field `a` of the input structure `in`

- `out.sb` — Substructure `sb` of the output structure `out`
- `out.sb.ele` — Field `ele` of the substructure `out.sb`
- `out.sb.ele(2,2)` — Element in the second row, second column of the field `ele` of the substructure `out.sb`
- `subBusArray(1)` — First element of the array of structures `subBusArray`
- `subBusArray(1).ele` — Field `ele` of the structure `subBusArray(1)`
- `subBusArray(1).ele(3,4)` — Element in the third row, fourth column of the field `ele` of structure `subBusArray(1)`

Because the chart uses MATLAB as the action language, you access the elements of the arrays in this example by using one-based indexing delimited by parentheses. In charts that use C as the action language, use zero-based indexing delimited by brackets. For more information, see “Operations for Vectors and Matrices in Stateflow” on page 19-4.

Assign Values to Structures and Fields

You can write to any Stateflow structure that has a scope other than `Input`. You can assign values to the entire structure, to a substructure, or to a single field.

- To assign one structure to another structure, define both structures from the same `Simulink.Bus` object in the base workspace.
- To assign one structure to a substructure of a different structure (or the other way around), define the structure and substructure from the same `Simulink.Bus` object.
- To assign a field of one structure to a field of another structure, the fields must have the same type and size. You can define the Stateflow structures from different `Simulink.Bus` objects.

For instance, the chart in this example makes these assignments:

- `localbus = sb2abc(in.sb)` — The structure `localbus` and the output argument of the MATLAB® function `sb2abc` are defined from the same `Simulink.Bus` object `BusObject`. The function decomposes its input into three components: a vector, a 3-by-2 matrix, and a scalar. The function returns these components as the fields `a`, `b`, and `c` of its output. For more information on this function, see “Access Simulink Bus Signals in MATLAB Functions” on page 7-13.
- `subBusArray(1) = in.sb` — The structure `subBusArray(1)` and the substructure `in.sb` are defined from the same `Simulink.Bus` object `SubBus`.
- `subBusArray(2) = abc2sb(in)` — The structure `subBusArray(2)` and the output argument of the graphical function `abc2sb` are defined from the same `Simulink.Bus` object `SubBus`. The function combines the values of the fields `a`, `b`, and `c` from its input and rearranges them in a 3-by-3 matrix of type `int8`. It returns this matrix as the field `ele` of its output.
- `subBusArray(3).ele = transpose(in.sb.ele)` — The field `subBusArray(3).ele` has the same type and size as the result of `transpose(in.sb.ele)`. Both are 3-by-3 matrices of type `int8`.
- `subBusArray(4).ele = int8(magic(3))` — The field `subBusArray(4).ele` has the same type and size as the result of `int8(magic(3))`. Both are 3-by-3 matrices of type `int8`.
- `out = localbus` — Both `out` and `localbus` are defined from the same `Simulink.Bus` object `BusObject`.
- `out.sb = subBusArray(idx)` — The substructure `out.sb` and the structure `subBusArray(idx)` are defined from the same `Simulink.Bus` object `SubBus`.

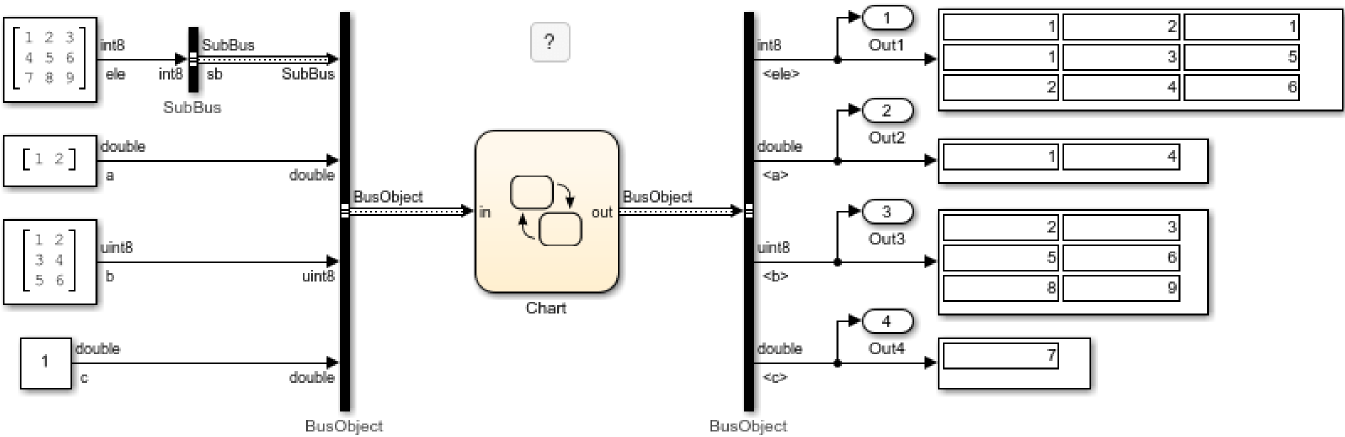
```
A
en:
localbus = sb2abc(in.sb);
subBusArray(1) = in.sb;
subBusArray(2) = abc2sb(in);
subBusArray(3).ele = transpose(in.sb.ele);
subBusArray(4).ele = int8(magic(3));
out = localbus;
out.sb = subBusArray(idx);
```

```
MATLAB Function  y = sb2abc(u)
```

```
function  y = abc2sb(u)
{
y.ele = int8([u.a(1) u.a(2) u.c; transpose(u.b)]);
}
```

Run the Simulation

When you simulate the example, the chart uses the values of the field `sb` of the input structure to populate the fields `a`, `b`, and `c` of the output structure. The parameter `idx` selects the element of the array of structures `subBusArray` to use as the substructure `sb` of the output. In this example, `idx` equals 2, so the chart uses the values of the fields `a`, `b`, `c` of the input structure to populate the substructure.



When you use other values for `idx`, the substructure `out.sb` contains the same values as `in.sb`, the transpose of `in.sb`, or a 3-by-3 magic square.

See Also

`Simulink.Bus`

More About

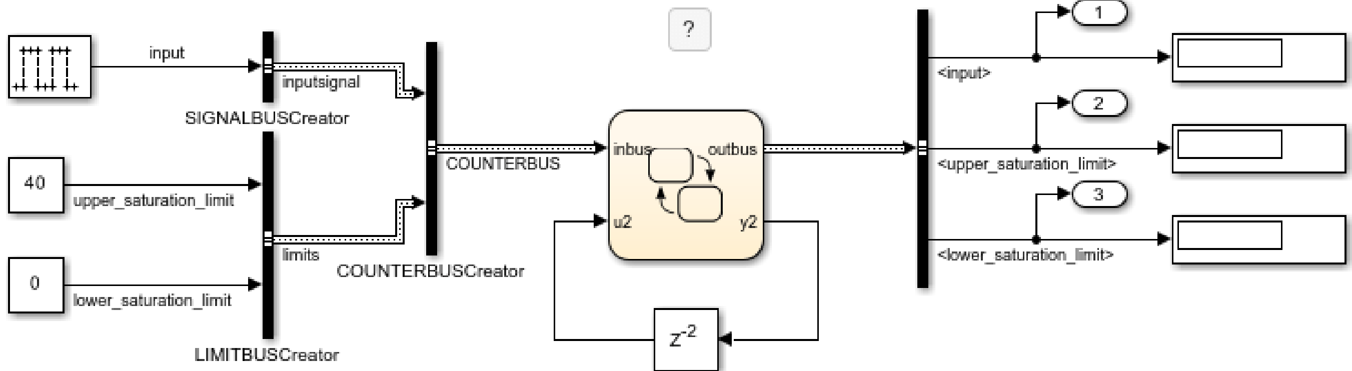
- "Access Bus Signals Through Stateflow Structures" on page 26-2
- "Identify Data by Using Dot Notation" on page 10-39
- "Operations for Vectors and Matrices in Stateflow" on page 19-4

- “Access Simulink Bus Signals in MATLAB Functions” on page 7-13

Integrate Custom Structures in Stateflow Charts

This example shows how to use structures from custom code in a Stateflow® chart. You can define structure typed data in C code and integrate it with Stateflow structures and Simulink® buses. By sharing data with custom code, you can augment the capabilities supported by Stateflow and take advantage of your preexisting code. For more information, see “Reuse Custom Code in Stateflow Charts” on page 28-2.

In this example, a Stateflow chart processes data from one Simulink bus and outputs the result to another Simulink bus. Both the input and output buses are defined by the `Simulink.Bus (Simulink)` object `COUNTERBUS`. In the chart, the Simulink buses interface with the Stateflow structures `inbus` and `outbus`. The chart calls a custom C function to write to the output structure `outbus`.

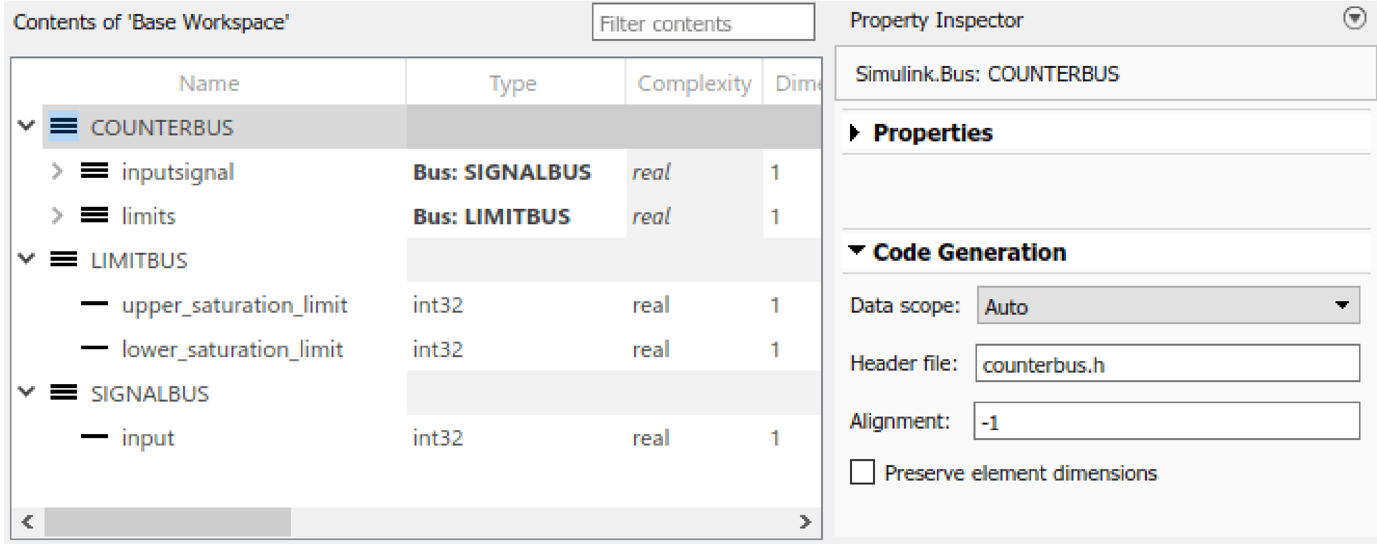


Define Custom Structures in C Code

1. In your C code, define a structure by creating a custom header file. The header file contains `typedef` declarations that define the properties of the custom structure. For example, in this model, the header file `counterbus.h` declares three custom structures:

```
...
typedef struct {
    int input;
} SIGNALBUS;

typedef struct {
    int upper_saturation_limit;
    int lower_saturation_limit;
} LIMITBUS;

typedef struct {
    SIGNALBUS inputsignal;
    LIMITBUS limits;
} COUNTERBUS;
...
```

2. In the Type Editor, define a `Simulink.Bus` object that matches each custom structure `typedef` declaration. In the **Header file** field of each `Simulink.Bus` object, enter the name of the header file that contains the matching `typedef` declaration.



3. Configure the Stateflow chart to include custom C code, as described in “Configure Custom Code for Your Model” on page 28-4.

4. Build and run your model.

Pass Stateflow Structures to Custom Code

When you call custom code functions that take structure pointers as arguments, pass the Stateflow structures by address. To pass the address of a Stateflow structure or one of its fields to a custom function, use the & operator and dot notation:

- `&outbus` provides the address of the Stateflow structure `outbus`.
- `&outbus.inputsignal` provides the address of the substructure `inputsignal` of the structure `outbus`.
- `&outbus.inputsignal.input` provides the address of the field `input` of the substructure `outbus.inputsignal`.

For more information, see “Index and Assign Values to Stateflow Structures” on page 26-7.

For instance, this example contains a custom C function `counterbusFcn` that takes structure pointers as arguments. The custom header file `counterbus.h` contains this function declaration:

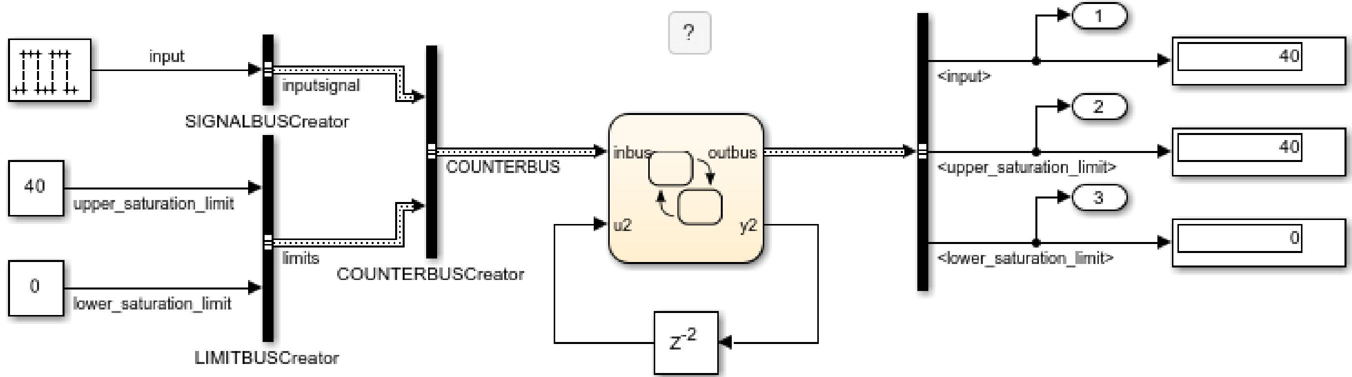
```
extern void counterbusFcn(COUNTERBUS *u1, int u2, COUNTERBUS *y1, int *y2);
```

The chart passes the addresses to the Stateflow structures `counterbus_struct` and `outbus` by using this function call:

```
counterbusFcn(&counterbus_struct, u2, &outbus, &y2);
```

```
{
counterbus_struct = inbus;
counterbus_struct.inputsignal = inbus.inputsignal;

counterbusFcn(&counterbus_struct, u2, &outbus, &y2);
}
```

The function reads the value of the chart input `u2` and the local structure `counterbus_struct`. It writes to the chart output `y2` and the output structure `outbus`.



See Also

`Simulink.Bus`

More About

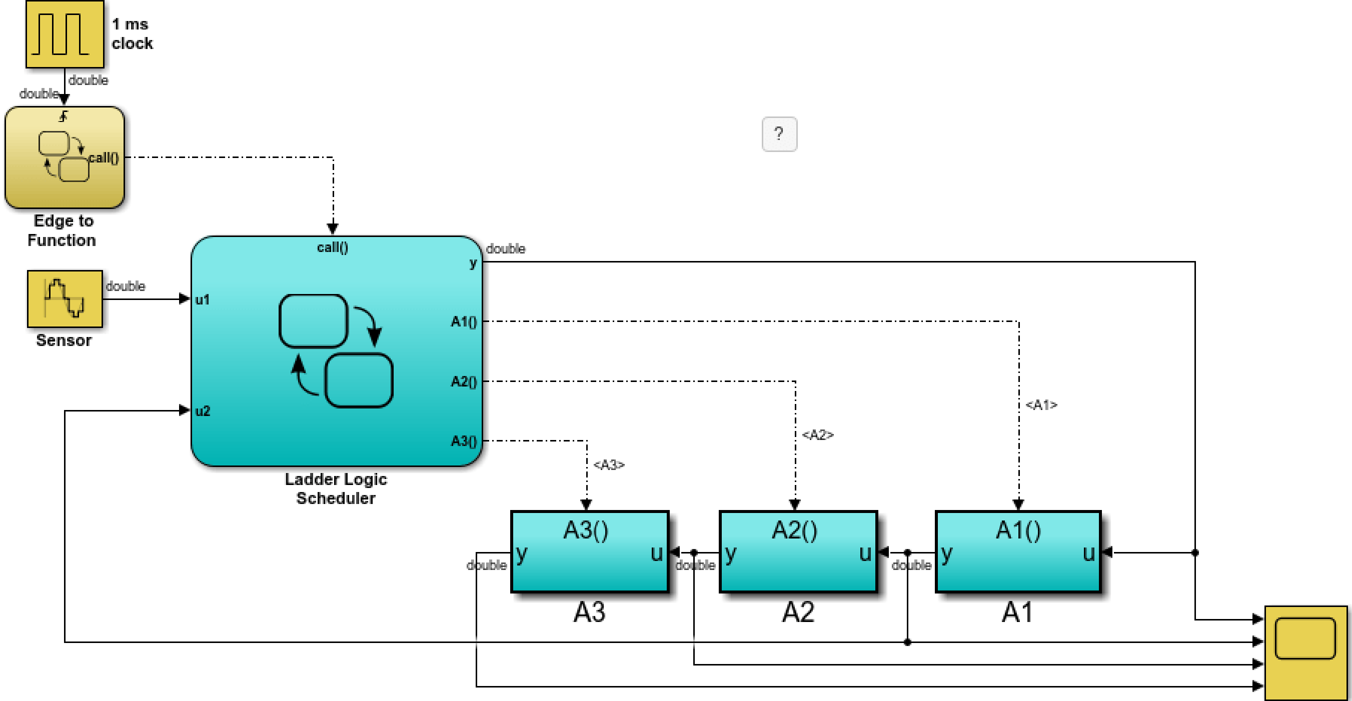
- “Access Bus Signals Through Stateflow Structures” on page 26-2
- “Index and Assign Values to Stateflow Structures” on page 26-7
- “Reuse Custom Code in Stateflow Charts” on page 28-2
- “Access Custom Code Variables and Functions in Stateflow Charts” on page 28-12
- “Integrate External Code by Using Model Configuration Parameters” (Simulink Coder)

Stateflow Design Patterns

- “Schedule Multiple Subsystems in a Single Step” on page 27-2
- “Schedule a Subsystem Multiple Times in a Single Step” on page 27-6
- “Schedule Subsystems to Execute at Specific Times” on page 27-9
- “Reduce Transient Signals by Using Debouncing Logic” on page 27-12
- “Detect Faults in Aircraft Elevator Control System” on page 27-19
- “Map Fault Conditions to Actions by Using Truth Tables” on page 27-24
- “Design for Isolation and Recovery in a Chart” on page 27-27
- “Launch Abort System” on page 27-31
- “Model a Fault-Tolerant Fuel Control System” on page 27-36
- “Model a Power Window Controller” on page 27-51
- “Model a Fitness Tracker” on page 27-59

Schedule Multiple Subsystems in a Single Step

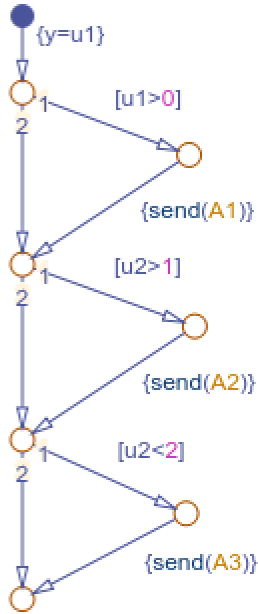
This example shows how to design a *ladder logic scheduler* in Stateflow®. The ladder logic scheduler design pattern allows you to specify the order in which multiple Simulink® subsystems execute in a single time step. Stateflow schedulers extend control of subsystem execution in a Simulink model, which determines order of execution implicitly based on block connectivity and sample time propagation.



Key Behavior of the Ladder Logic Scheduler

In this example, the Ladder Logic Scheduler chart broadcasts a series of function-call output events to execute three function-call subsystems (A1, A2, and A3). During each time step:

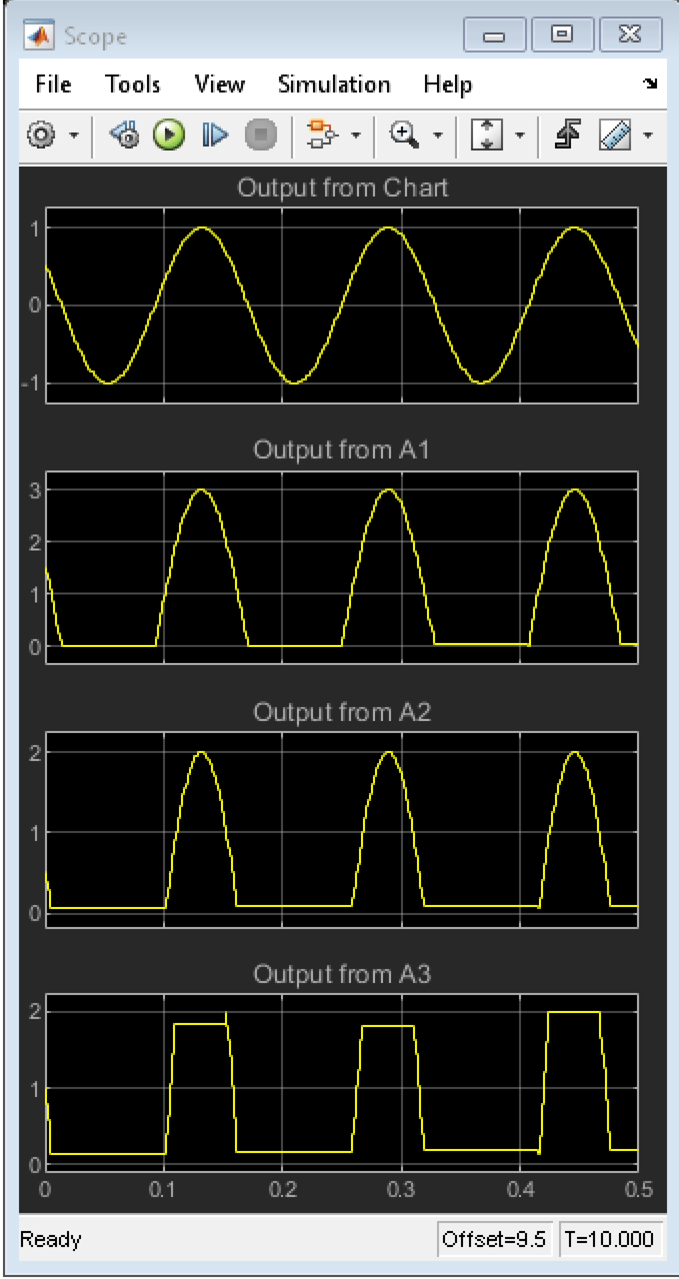
1. The Simulink model activates the Edge to Function chart at the rising edge of the 1-millisecond pulse generator.
2. The Edge to Function chart broadcasts the function-call output event `call` to activate Ladder Logic Scheduler chart.
3. The Ladder Logic Scheduler chart uses sequencing ladder logic to broadcast function-call output events based on the values of the input signals `u1` and `u2`.



The chart evaluates each condition sequentially. When a condition is valid, the chart calls the `send` operator to broadcast an output event. The corresponding subsystem computes its output and returns control back to the Ladder Logic Scheduler chart.

Run the Ladder Logic Scheduler

When you simulate the model, the scope shows the input and output of each function-call subsystem.



During each time step, the Ladder Logic Scheduler chart executes the subsystems depending on the values of the input signals `u1` and `u2`:

1. If `u1` is positive, the chart sends a function-call output event to execute subsystem A1. This subsystem multiplies the value of `u1` by a gain of 3 and passes this value back to the Ladder Logic Scheduler chart as input `u2`. Control returns to the next condition in the Ladder Logic Scheduler chart.
2. If `u2` is greater than 1, the chart sends a function-call output event to execute subsystem A2. This subsystem decreases the value of `u2` by 1. Control returns to the final condition in the Ladder Logic Scheduler chart.

3. If u2 is less than 2, the chart sends a function-call output event to execute subsystem A3. This subsystem multiplies its input by a gain of 2.

In the scope, horizontal segments indicate time steps when a subsystem does not execute.

See Also

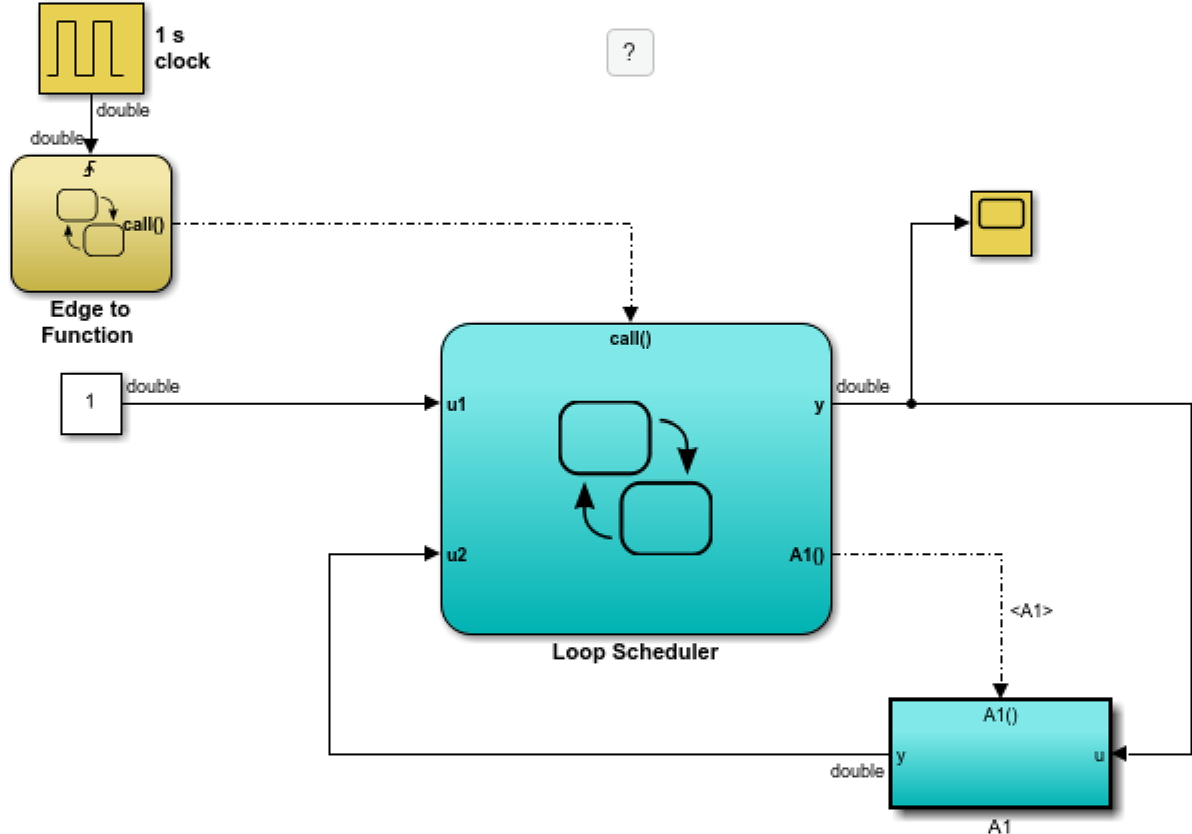
`send`

More About

- “Activate a Stateflow Chart by Sending Input Events” on page 12-8
- “Activate a Simulink Block by Sending Output Events” on page 12-15

Schedule a Subsystem Multiple Times in a Single Step

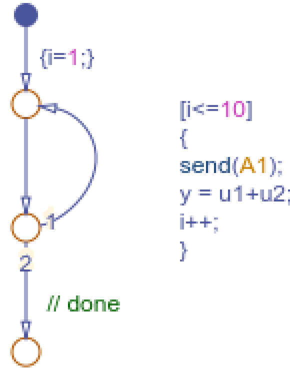
This example shows how to design a *loop scheduler* in Stateflow®. The loop scheduler design pattern allows you to execute a Simulink® subsystem multiple times in a single time step. Stateflow schedulers extend control of subsystem execution in a Simulink model, which determines order of execution implicitly based on block connectivity and sample time propagation.



Key Behavior of the Loop Scheduler

In this example, the Loop Scheduler chart broadcasts a function-call output event to execute the function-call subsystem A1 multiple times every time step. During each time step:

1. The Simulink model activates the Edge to Function chart at the rising edge of the 1-millisecond pulse generator.
2. The Edge to Function chart broadcasts the function-call output event `call` to activate the Loop Scheduler chart.
3. The Loop Scheduler chart calls the `send` operator to broadcast the function-call output event `A1` multiple times.



Each broadcast of the event `A1` executes the subsystem A1. The subsystem computes its output and returns control back to the Loop Scheduler chart.

Run the Loop Scheduler

When you simulate the model, the scope displays the value of `y` at each time step.

During each time step, the value of y increases by 25 because:

- The flow chart in the Loop Scheduler implements a for loop that iterates 10 times.
- In each iteration of the for loop, the chart increments y by 1 (the constant value of input u1).
- Each time that the chart broadcasts the output event to subsystem A1, the subsystem increments y by 1.5.

See Also

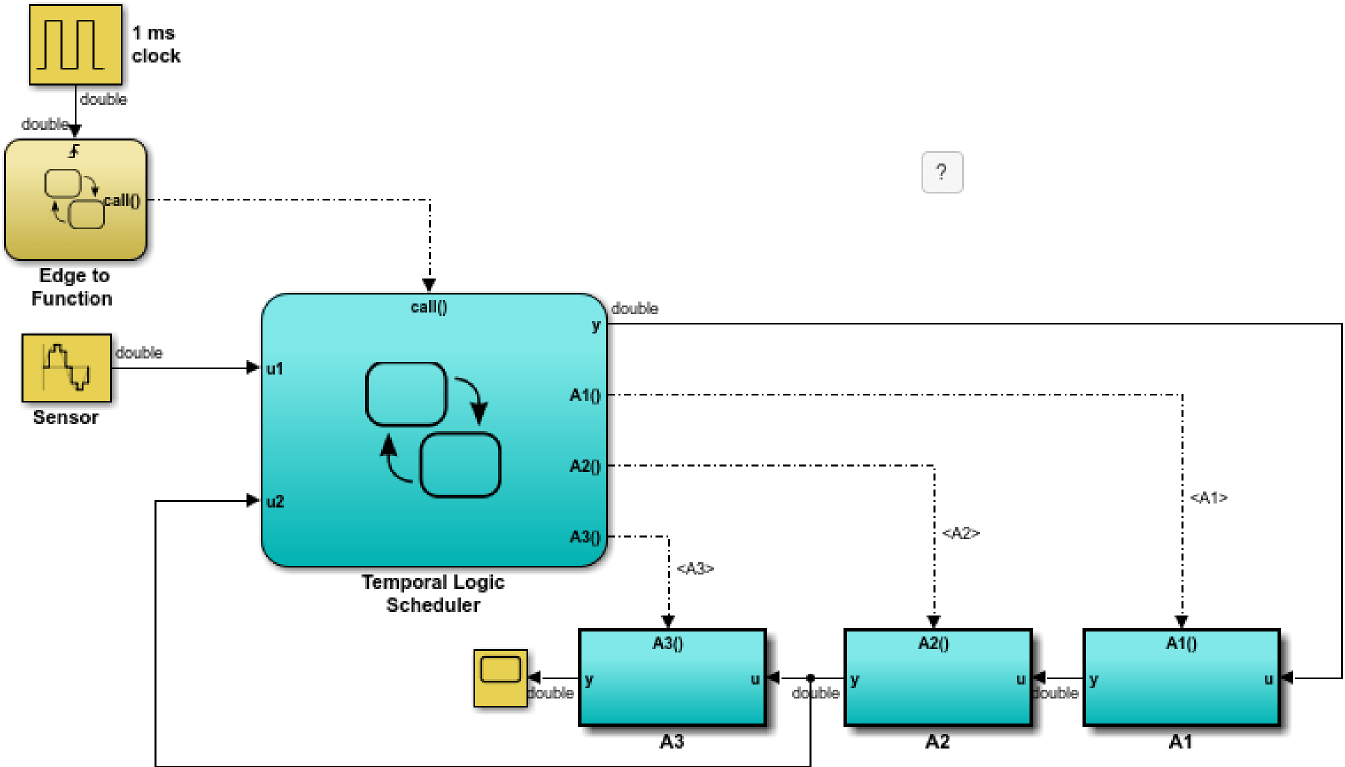
send

More About

- "Activate a Stateflow Chart by Sending Input Events" on page 12-8
- "Activate a Simulink Block by Sending Output Events" on page 12-15

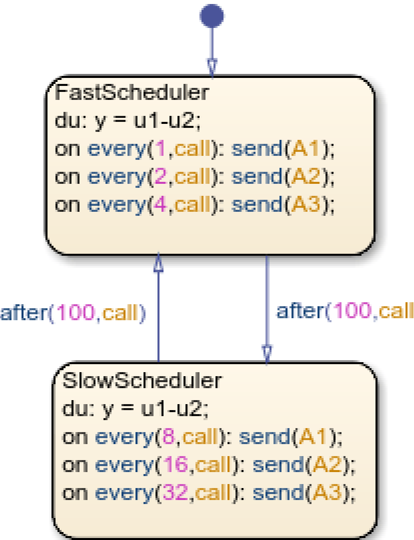
Schedule Subsystems to Execute at Specific Times

This example shows how to design a *temporal logic scheduler* in Stateflow®. The temporal logic scheduler design pattern allows you to schedule Simulink® subsystems to execute at specified times. Stateflow schedulers extend control of subsystem execution in a Simulink model, which determines order of execution implicitly based on block connectivity and sample time propagation.



Key Behavior of the Temporal Logic Scheduler

In this example, the Temporal Logic Scheduler chart contains two states that schedule the execution of three function-call subsystems (A1, A2, and A3) at different rates, as determined by the temporal logic operator `every`.



When the `FastScheduler` state is active, the chart schedules function calls to different Simulink subsystems at a fraction of the base rate at which the input event `call` wakes up the chart.

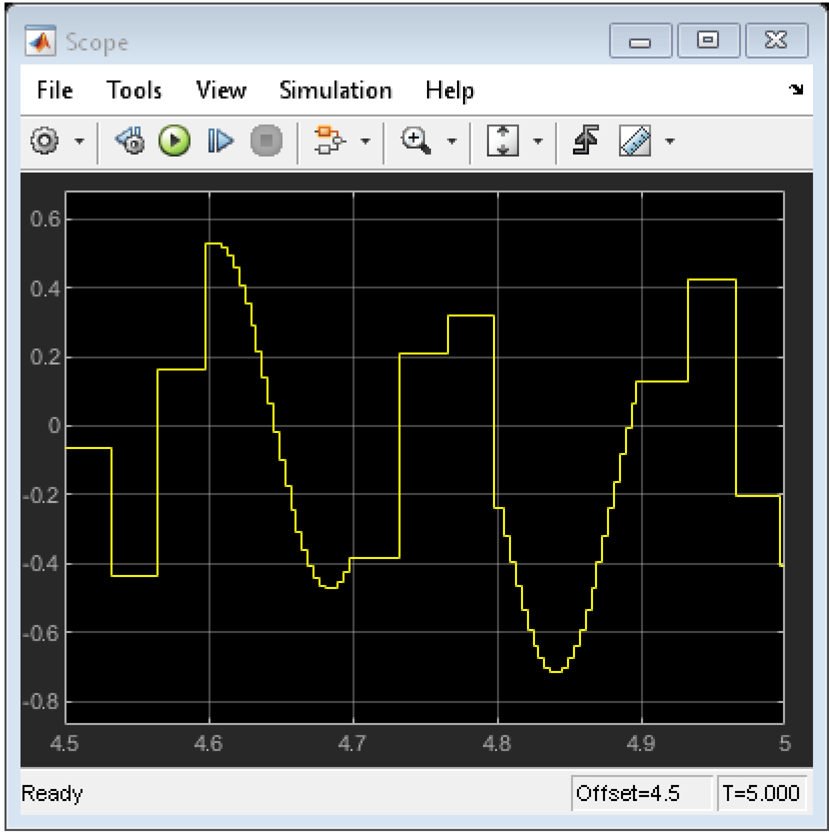
- The chart sends an event to execute subsystem A1 at the base rate.
- The chart sends an event to execute subsystem A2 at half the base rate.
- The chart sends an event to execute subsystem A3 at one quarter the base rate.

When the `SlowScheduler` state is active, the chart schedules function calls for A1, A2, and A3 at 1/8, 1/16, and 1/32 times the base rate.

The chart switches between the fast and slow execution modes after every 100 invocations of the `call` event.

Run the Temporal Logic Scheduler

When you simulate the model, the scope displays the value of y at each time step.



The changes in value illustrate the different rates of execution.

- When the chart executes the subsystems at a slow rate (for example, from $t = 4.5$ to $t = 4.6$, from $t = 4.7$ to $t = 4.8$, and from $t = 4.9$ to $t = 5.0$), the values change slowly.
- When the chart executes the subsystems at a fast rate (for example, from $t = 4.6$ to $t = 4.7$ and from $t = 4.8$ to $t = 4.9$), the values change rapidly.

See Also

every | send

More About

- “Control Chart Execution by Using Temporal Logic” on page 14-35
- “Activate a Stateflow Chart by Sending Input Events” on page 12-8
- “Activate a Simulink Block by Sending Output Events” on page 12-15

Reduce Transient Signals by Using Debouncing Logic

When a switch opens and closes, the switch contacts can bounce off each other before the switch completely transitions to an on or off state. The bouncing action can produce transient signals that do not represent a true change of state. Therefore, when modeling switch logic, it is important to filter out transient signals by using *debouncing* algorithms.

If you model a controller in a Stateflow chart, you do not want your switch logic to overwork the controller by turning it on and off in response to every transient signal it receives. To avoid this, design a Stateflow controller that uses temporal logic to debounce your input signals and determine whether a switch is actually on or off.

How to Debounce a Signal

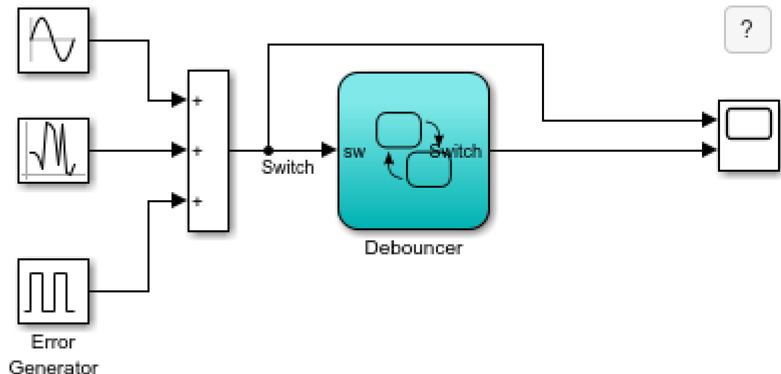
There are two ways to debounce a signal by using Stateflow:

1. Filter out transient signals by using the `duration` temporal operator.
2. Filter out transient signals by using an intermediate graphical state. Use intermediate graphical state for advanced filtering techniques, such as fault detection.

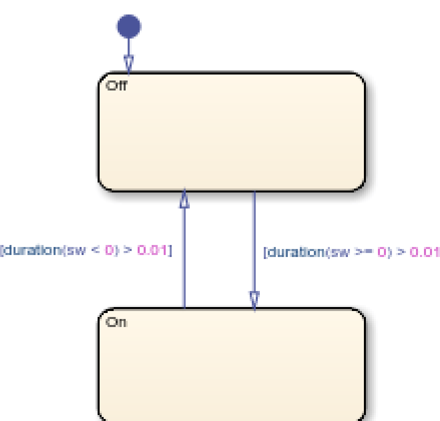
The `duration` operator is supported only in Stateflow charts in a Simulink model.

Debounce Signals with the duration Operator

This example illustrates a design pattern that uses the `duration` operator to filter out transient signals.



The Debouncer chart contains this logic.



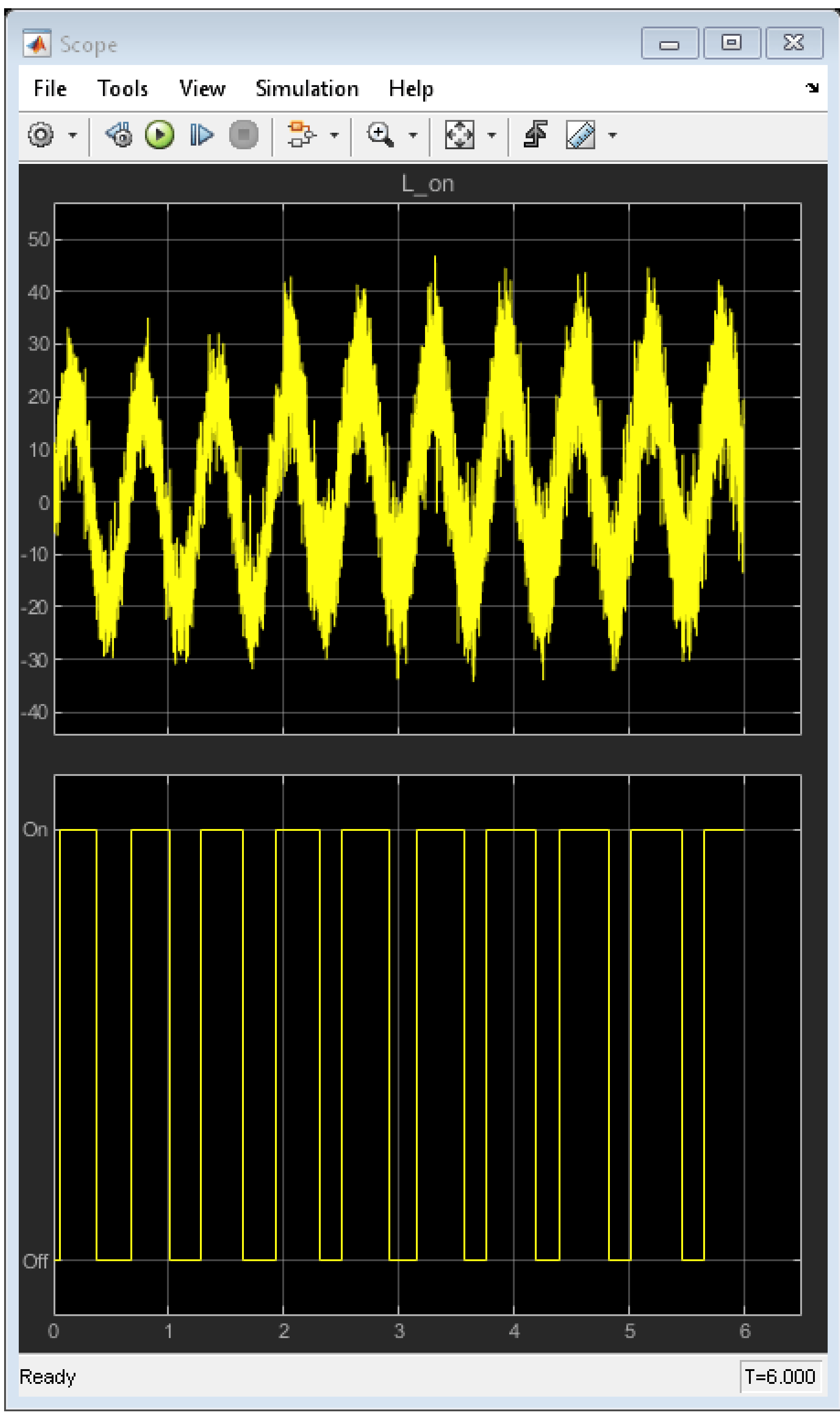
State Logic

The initial state for this model is `Off`. By using the `duration` operator, you can control which state is active based on how long the switch signal, `sw`, has been negative or nonnegative.

- When `sw` has been nonnegative for longer than 0.01 seconds, the switch moves from state `Off` to state `On`.
- When `sw` has been negative for longer than 0.01 seconds, the switch moves from state `On` to state `Off`.

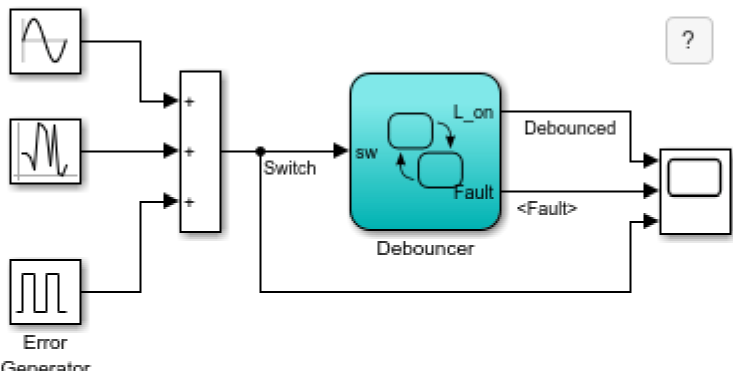
Run the Debouncer

1. Open the model.
2. Open the Scope block.
3. Open the Stateflow chart Debouncer.
4. Simulate the model. The scope shows how the debouncer isolates transient signals from the noisy input signal.

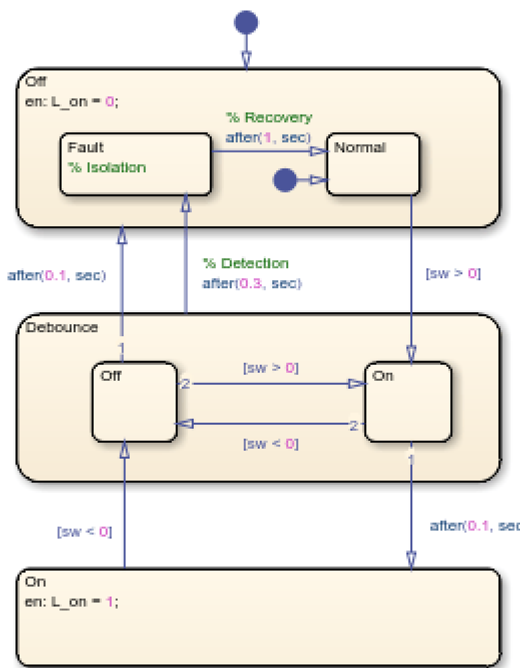


Debounce Signals with Fault Detection

This example illustrates a design pattern that uses an intermediate state to isolate transient signals. The debouncer design uses the `after` operator to implement absolute-time temporal logic. With this design pattern, you can also detect faults and allow your system time to recover.



The Debouncer chart contains this logic.



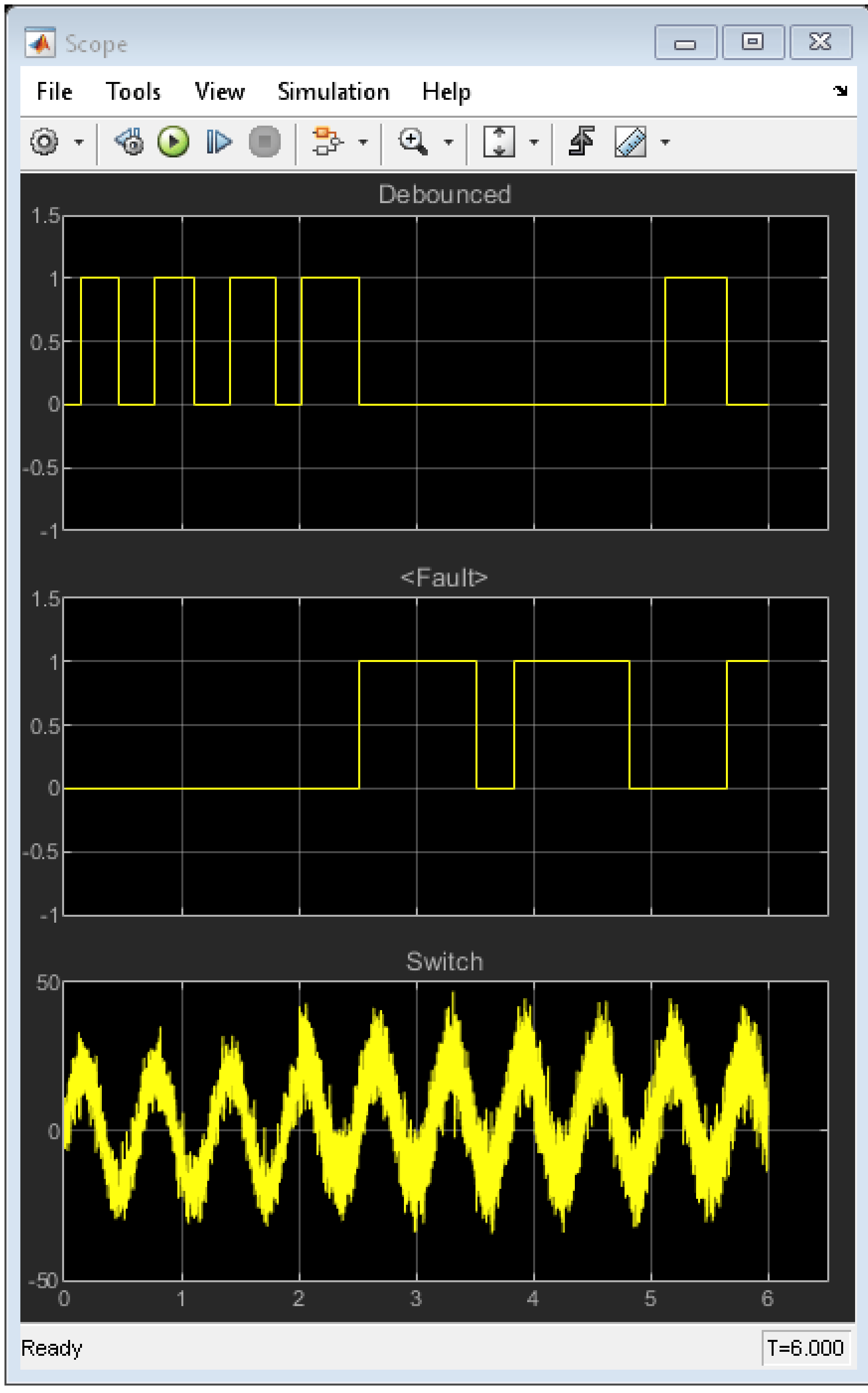
State Logic

The Debouncer chart contains an intermediate state called `Debounce`. This state isolates transient inputs by checking if the signal `sw` remains positive or negative, or if it fluctuates between zero crossings over a prescribed period.

- When `sw` has been positive for longer than 0.1 seconds, the switch moves to state `On`.
- When `sw` has been negative for longer than 0.1 seconds, the switch moves to state `Off`.
- When `sw` fluctuates between zero crossings for longer than 0.3 seconds, the switch moves to state `Off.Fault`, isolating `sw` as a transient signal and giving it time to recover.

Run the Debouncer

1. Open the model.
2. Open the Scope block.
3. Open the Stateflow chart Debouncer.
4. Simulate the model. The scope shows how the debouncer isolates transient signals from the noisy input signal.



Use Event-Based Temporal Logic

As an alternative to absolute-time temporal logic, you can apply event-based temporal logic to determine true state in the Debouncer chart by using the `after` operator. The keyword `tick` specifies and implicitly generates a local event when the chart awakens.

The Error Generator block in the sf_debouncer model generates a pulse signal every 0.001 second. Therefore, to convert the absolute-time temporal logic specified in the Debouncer chart to event-based logic, multiply the argument of the after operator by 1000, as indicated by this table.

Absolute Time-Based Logic	Event-Based Logic
after(0.1,sec)	after(100,tick)
after(0.3,sec)	after(300,tick)
after(1,sec)	after(1000,tick)

See Also

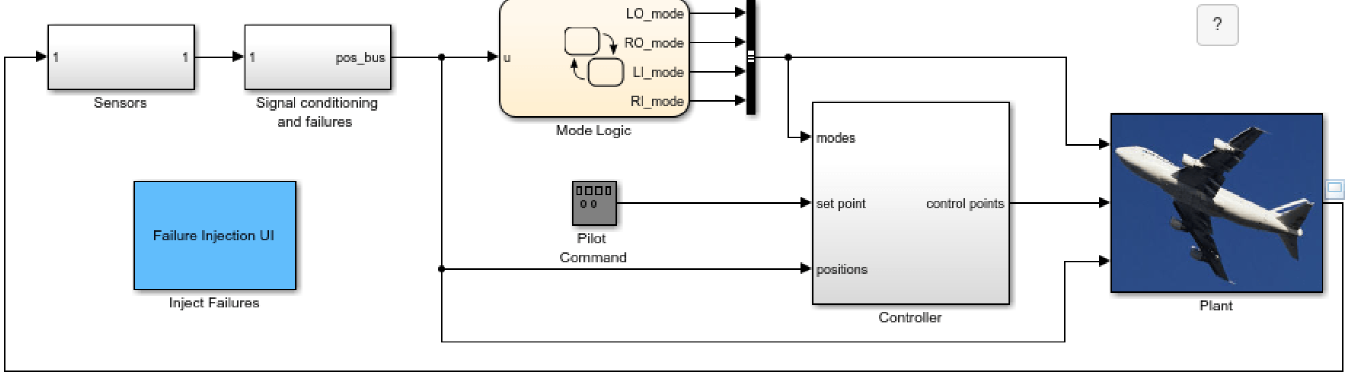
after | duration

More About

- “Control Chart Execution by Using Temporal Logic” on page 14-35
- “Control Chart Behavior by Using Implicit Events” on page 12-28

Detect Faults in Aircraft Elevator Control System

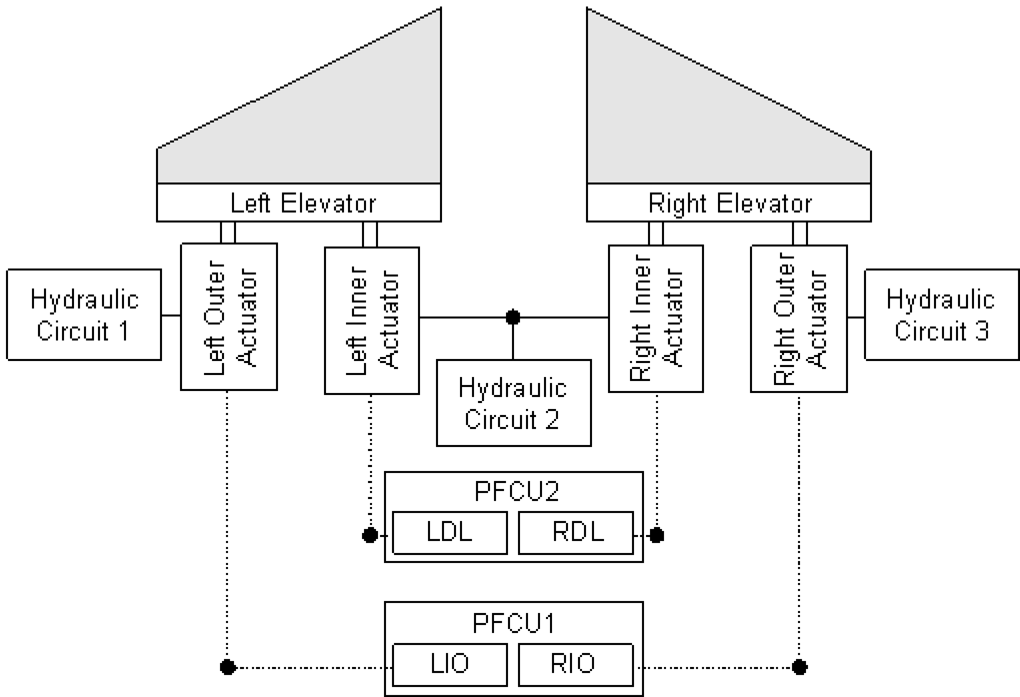
This example shows how to design a fault detection, isolation, and recovery (FDIR) application for a pair of aircraft elevators controlled by redundant actuators. This model uses the same fault detection control logic as the Avionics subsystem of the Aerospace Blockset™ example “HL-20 Project with Optional FlightGear Interface” (Aerospace Blockset).



Elevator Control System

A typical aircraft has two elevators, one on each side of the fuselage, attached on the horizontal tails. To enhance the safety of the aircraft, the elevator control system contains these redundant parts:

- Four independent hydraulic actuators (two actuators per elevator).
- Three hydraulic circuits that drive the actuators. Each outer actuator has a dedicated hydraulic circuit. The inner actuators share a hydraulic circuit.
- Two primary flight control units (PFCU).
- Two control modules per actuator: full range control law and limited/reduced range control law.



If the aircraft is flying perfectly level, then the actuator position should maintain a constant value. The fault detection system registers a failure in an actuator if:

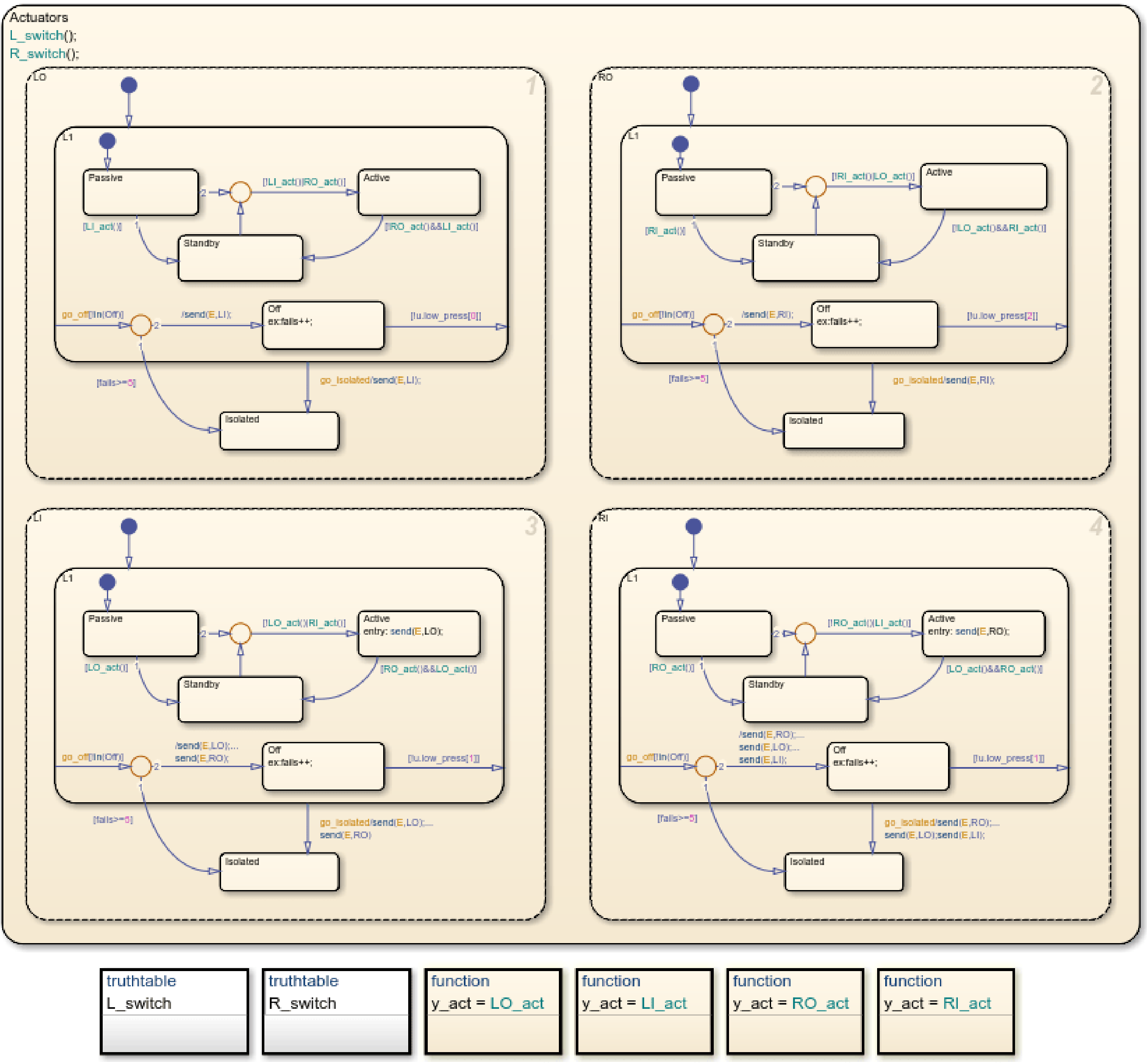
- The position of the actuator increases or decreases by 10 cm from this zero point.
- The actuator changes position rapidly (for instance, if the position changes at least 20 cm in 0.01 seconds).

The fault detection system also registers a fault in one of the hydraulic circuits if the pressure is out of bounds or if the pressure changes rapidly. In this example, the fault detection system checks that:

- The pressure in the hydraulic circuit is between 500 kPa and 2 MPa.
- The pressure changes no more than 100 kPa in 0.01 seconds.

Fault Detection Control Logic

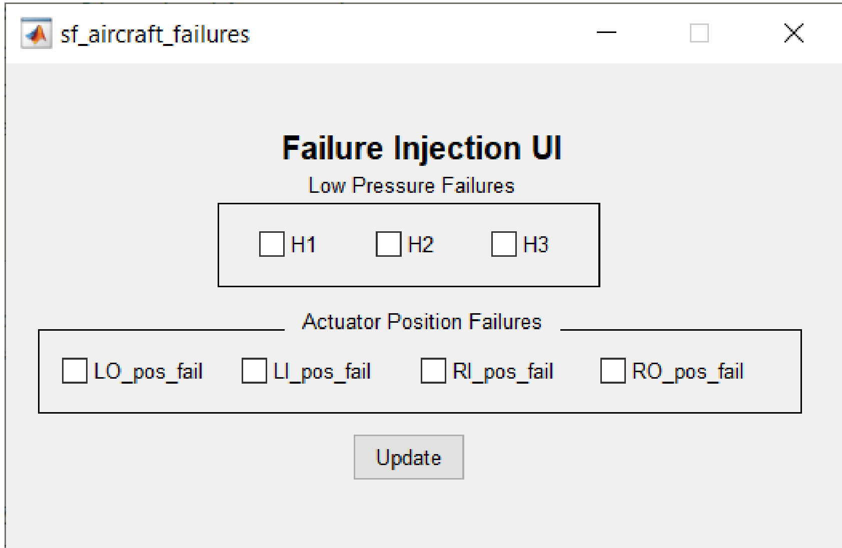
The Stateflow® chart Mode Logic defines the fault detection logic for the elevator control system. The chart contains a parallel substate for each actuator in the system. Each actuator can be in one of five modes: `Passive`, `Standby`, `Active`, `Off`, and `Isolated`. These operating modes are represented as substates of the parallel states.



By default, the outer actuators start in `Active` mode and the inner actuators start in `Standby` mode. If a failure is detected in the outer actuators or in the hydraulic circuits that are connected to them, the fault detection system responds by disabling the outer actuators and activating the inner actuators.

Inject Failures Into Fault Detection System

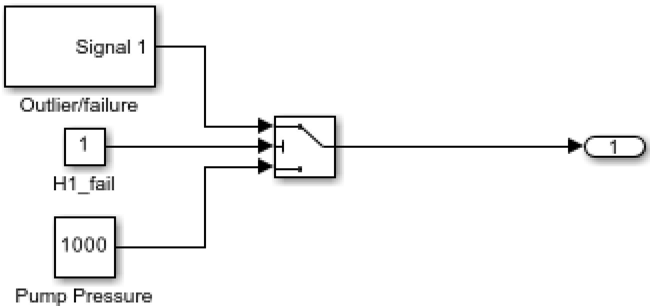
To experiment with the model, during simulation, you can introduce hydraulic circuit and actuator position failures into the fault detection system through the Failure Injection UI.



For example, to inject a failure in Hydraulic Circuit 1, select the `H1` check box and click **Update**. The UI runs this MATLAB® code to communicate with the Simulink® model:

```
function Inject_failure_Callback(hObject,eventdata,handles)

mname = gcs;

...

blockname = mname+ ...
    "/Signal conditioning and failures /Hydraulic Pressures/Measured "+ ...
    newline+"Hydraulic system 1 pressures/Hydraulic pressure/H1_fail";
val = get(handles.H1,"Value");

if val
    set_param(blockname,value="1");
else
    set_param(blockname,value="0");
end

...

end
```

This code turns on a switch in the Signal conditioning subsystem that causes the fault detection system to register a fault in the hydraulic circuit.



The chart Mode Logic responds to failures in the hydraulic circuits and actuators by using truth table functions and event broadcasting. For example, if the fault detection system registers an isolated failure in Hydraulic Circuit 1, then:

- The truth table function `L_switch` broadcasts the event `go_off` to the substate `LO`.
- The substate `LO` enters the `Off` mode and sends the event `E` to the substate `LI`.
- Because the substate `LO` is no longer in the `Active` mode, `LI` enters the `Active` mode.
- Because the substate `LI` is now in the active mode, `RI` enters the `Active` mode and sends a second event `E` to the substate `RO`.
- The substate `RO` enters the `Standby` mode.

After the fault detection systems registers a failure in Hydraulic Circuit 1, the left outer actuator is turned off, the right outer actuator is placed on standby, and the inner actuators are activated.

Recover from Hydraulic Failures

The fault detection control logic enables the system to recover from a hydraulic circuit failure. For example, to bring the Hydraulic Circuit 1 back online, in the Failure Injection UI, clear the `H1` check box and click **Update**. In the chart, the condition `!u.low_press[0]` becomes true, so the substate `LO` transitions from the `Off` mode to the `Standby` mode. As a result, the left outer actuator can then be activated in the event that the fault detection system registers another failure later in the simulation.

Isolate Actuators After Failures

When the fault detection system registers a failure in one of the actuators, that actuator can no longer be activated. In the chart Mode Logic, the failure of an actuator is represented by the substate `Isolated`. This substate has no outgoing transitions so once an actuator enters the `Isolated` state, it remains in that state for the rest of the simulation.

References

Pieter J. Mosterman and Jason Ghidella, "Model Reuse for the Training of Fault Scenarios in Aerospace," in *Proceedings of the AIAA® Modeling and Simulation Technologies Conference*, CD-ROM, paper 2004-4931, August 16 - 19, 2004, Rhode Island Convention Center, Providence, RI.

Jason R. Ghidella and Pieter J. Mosterman, "Applying Model-Based Design to a Fault Detection, Isolation, and Recovery System," in *Military Embedded Systems*, Summer, 2006.

See Also

More About

- “Synchronize Model Components by Broadcasting Events” on page 12-2
- “Use Truth Tables to Model Combinatorial Logic” on page 8-2
- “Reuse Logic Patterns by Defining Graphical Functions” on page 6-9
- “HL-20 Project with Optional FlightGear Interface” (Aerospace Blockset)

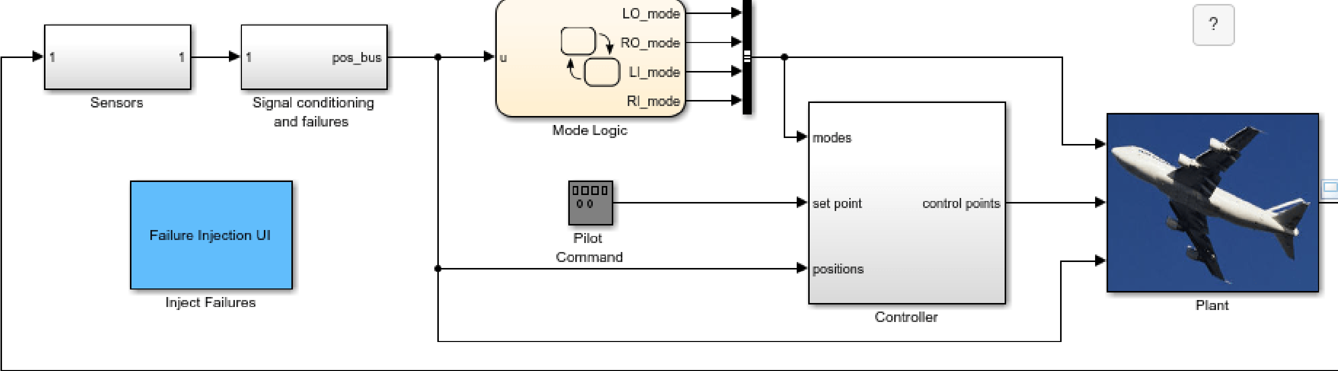
Map Fault Conditions to Actions by Using Truth Tables

This example shows how to use truth tables to map fault conditions of a system directly to their consequent actions.

Truth tables are only supported in Simulink®. For more information, see “Use Truth Tables to Model Combinatorial Logic” on page 8-2.

Detect Faults in a System

This model maps the fault conditions and actions of an aircraft elevator control system by using truth tables.



This list describes the requirements for the fault detection system in the model.

- Hydraulic pressure 1 failure — While there are no other failures, turn off the left outer actuator.
- Hydraulic pressure 2 failure — While there are no other failures, turn off the left inner actuator and the right inner actuator.
- Hydraulic pressure 3 failure — While there are no other failures, turn off the right outer actuator.
- Actuator position failure — While there are no other failures, isolate that specific actuator.
- Hydraulic pressure 1 and left outer actuator failures — While there are no other failures, turn off the left outer actuator.
- Hydraulic pressure 2 and left inner actuator failures — While there are no other failures, turn off the left inner actuator.
- Hydraulic pressure 3 and right outer actuator failures — While there are no other failures, turn off the right outer actuator.
- Multiple failures on left hydraulics and actuators — Isolate the left outer actuator and the left inner actuator.
- Multiple failures on right hydraulics and actuators — Isolate the right outer actuator and the right inner actuator.
- Intermittent actuator failures — If an actuator has been switched on and off five times during operation, isolate that specific actuator.

In the Mode Logic chart, a pair of truth table functions define the logic to satisfy these requirements. `L_switch` controls the left elevator and `R_switch` controls the right elevator. This truth table is for the left elevator.

sf_aircraft ▸ Mode Logic ▸ L_switch

Condition Table

	DESCRIPTION	CONDITION	D1	D2	D3	D4	D5	D6	D7
1	Hydraulic system 1 Low pressure (Left Outer line)	u.low_press[0]	T	T	F	F	-	-	-
2	Left Outer actuator position failed	u.L_pos_fail[0]	-	-	T	T	-	-	-
3	Hydraulic system 2 Low pressure (Inner line)	u.low_press[1]	F	-	F	-	T	-	-
4	Left Inner actuator position failed	u.L_pos_fail[1]	F	-	F	-	-	T	-
		ACTIONS: SPECIFY A ROW FROM THE ACTION TABLE	**2**	**3,5**	**3**	**3,5**	**4**	**5**	**Default**

Action Table

	DESCRIPTION	ACTION
1	Default - All ok, do nothing.	Default:
2	Hydraulic System 1 Failure. Turn off Left Outer Actuator	send(go_off,Actuators.LO);
3	Left Outer Actuator Failure. Isolate Left Outer Actuator	send(go_isolated,Actuators.LO);
4	Hydraulic System 2 Failure. Turn off Left Inner Actuator	send(go_off,Actuators.LI);
5	Left Inner Actuator Failure. Isolate Left Inner Actuator	send(go_isolated,Actuators.LI);

The first requirement indicates that if a failure is only detected in the hydraulic pressure 1 system, turn off the left outer actuator. In the truth table, this requirement is represented by the decision **D1**. If there is low pressure in the hydraulic system 1, then **D1** specifies that action 2 is performed. Action 2 sends an event `go_off` to the left actuator, `Actuators.LO`.

Similarly, the other requirements are mapped to the appropriate actions in the truth table. For example, if the left outer actuator fails, **D3** causes action 3. Action 3 sends the event `go_isolated` to `Actuators.LO` to isolate the left actuator.

The truth table functions are called at entry and during actions for the chart so that fault checks execute at each time step.

See Also

More About

- “Use Truth Tables to Model Combinatorial Logic” on page 8-2
- “Program a Truth Table” on page 8-8

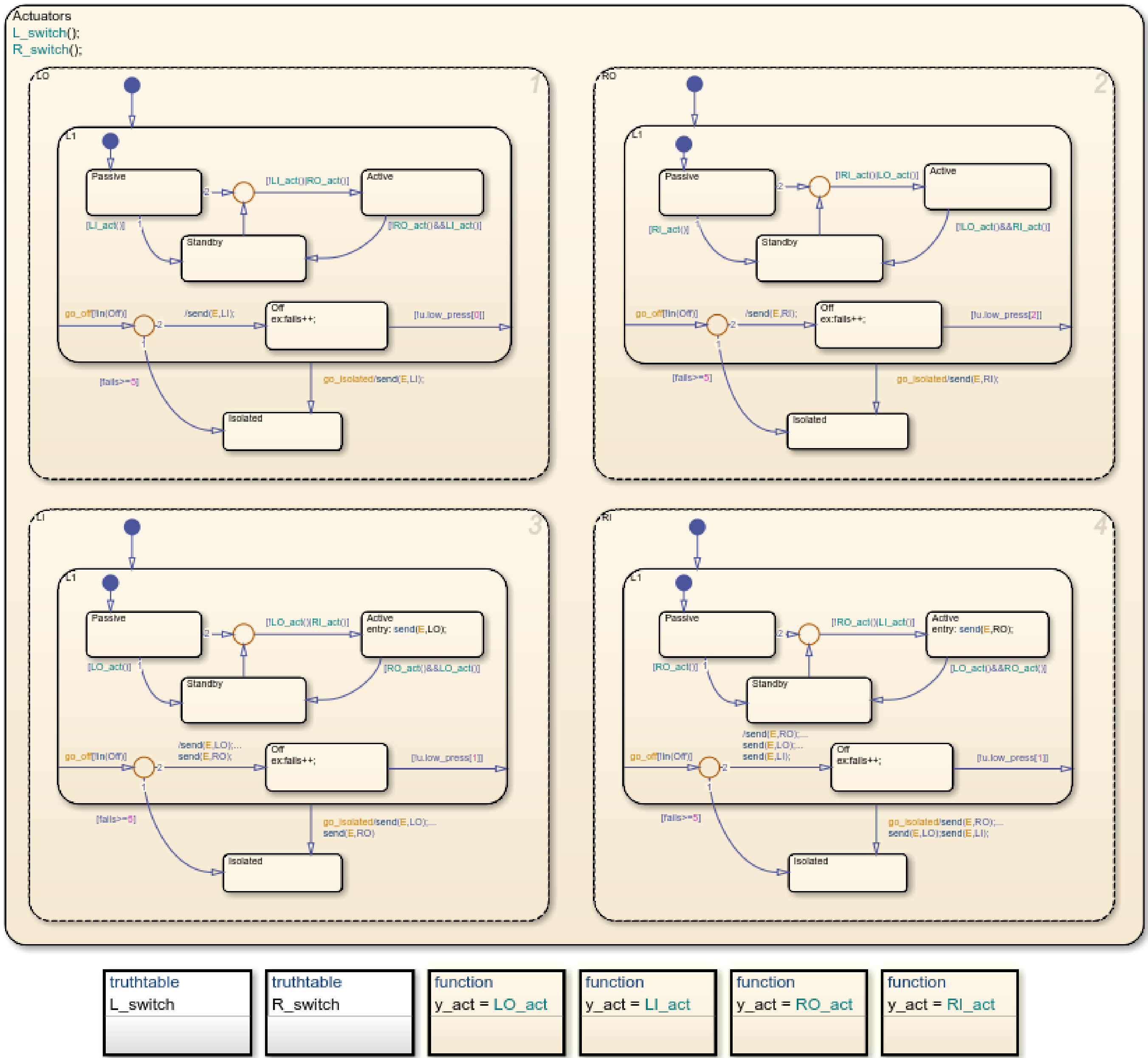
Design for Isolation and Recovery in a Chart

Mode Logic for the Elevator Actuators

This example shows how the model `sf_aircraft` uses the chart `Mode Logic` to detect system faults and recover from failure modes for an aircraft elevator control system. For more information on this model, see "Detect Faults in Aircraft Elevator Control System" on page 27-19. To open the model, enter:

```
openExample("stateflow/FaultDetectionControlLogicInAnAircraftControlSystemExample")
```

There are two elevators in the system, each with an outer and inner actuator. The `Actuators` state has a corresponding substate for each of the four actuators. An actuator has five modes: `Passive`, `Active`, `Standby`, `Off`, and `Isolated`. By default, the outer actuators are on, and the inner actuators are on standby. If a fault is detected in the outer actuators, the system responds to maintain stability by turning the outer actuators off and activating the inner actuators.



States for Failure and Isolation

Each actuator contains an `Off` state and an `Isolated` state. When the fault detection logic in one of the truth tables detects a failure, it broadcasts the event `go_off` or `go_isolated` to the failing actuator. For more information, see “Map Fault Conditions to Actions by Using Truth Tables” on page 27-24.

The `go_off` event instructs the failing actuator to transition to the `Off` state until the condition is resolved. The event `go_isolated` causes the failing actuator to transition to `Isolated`. Transitions to the `Isolated` state are from the superstate `L1`, which contains all the other operating modes. This state has no outgoing transitions, so that once an actuator has entered `Isolated` it remains there.

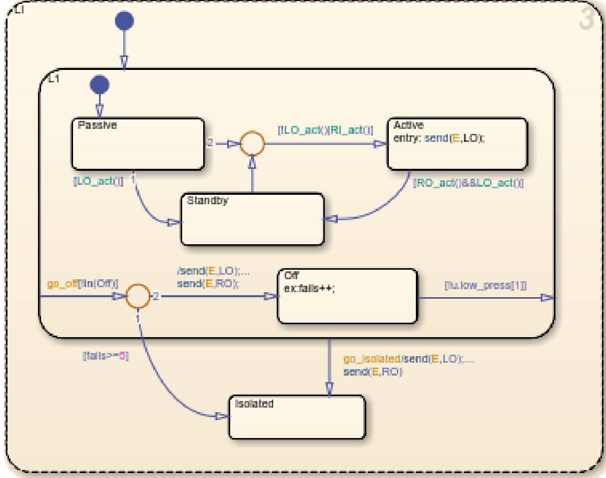
Intermittent failures that cause an actuator to fail 5 or more times, also cause a transition to `Isolated`. The variable `fails` logs the number of failures for an actuator by incrementing each time a transition occurs out of `Off`.

Transitions for Recovery

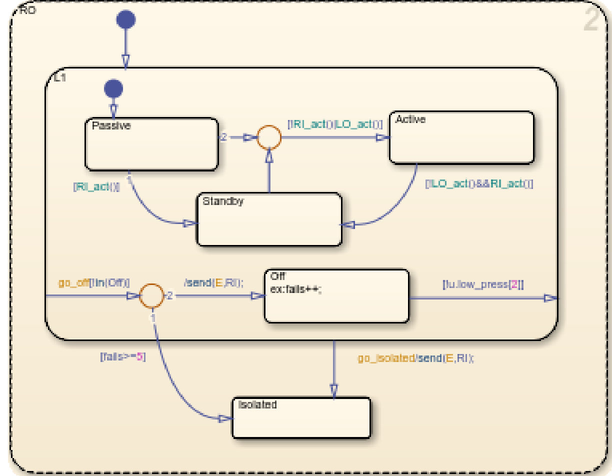
Transitions in the substates for each actuator account for recovery requirements of the elevator system. These requirements derive from rules for symmetry and safety of the elevators such as:

- Only one actuator for an elevator must be active at one time.
- Outer actuators have priority over the inner actuators.
- Actuator activity should be symmetric if possible.
- Switching between actuators must be kept to a minimum.

For example, one requirement of the system is if one outer actuator fails, then the other outer actuator must move to standby and the inner actuators take over. Consequently, there is a transition from each `Active` state to `Standby`, and vice versa.



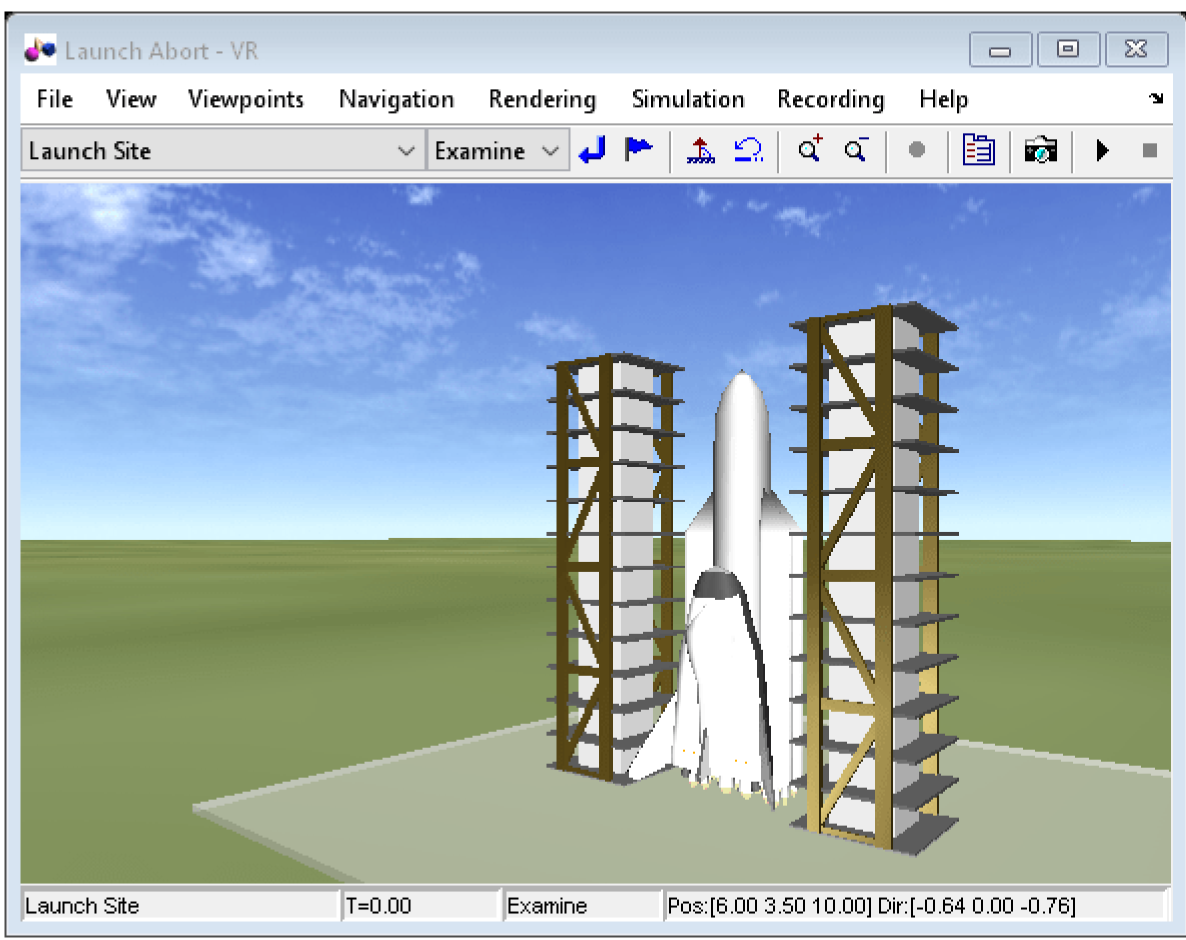
For the inner left actuator (`LI`), the transition to `Active` inside the `L1` superstate is conditionally based on `[!LO_act()|RI_act()]`. This causes the left inner actuator to turn on if the outer actuator (`LO`) has failed, or the right inner actuator (`RI`) has turned on.



Another consequence if LO fails and moves out of Active is a transition that occurs in the right outer actuator (RO). The RO state transitions inside the L1 superstate from Active to Standby. This satisfies the requirement of the outer actuators and inner actuators to work in symmetry.

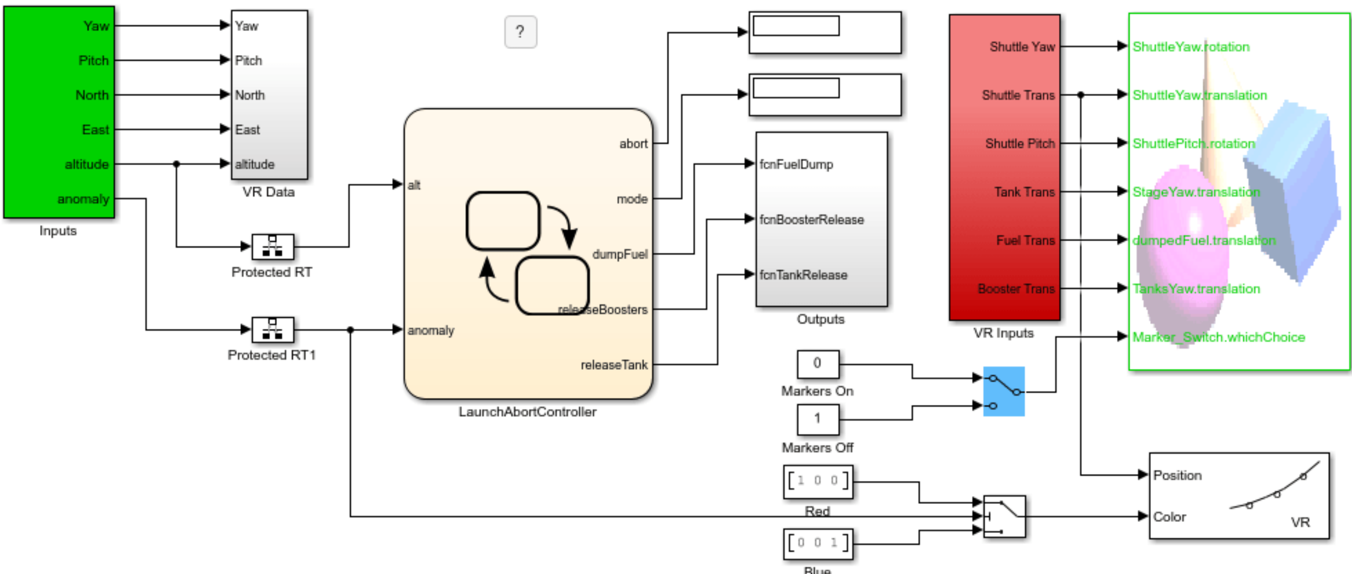
Launch Abort System

This example shows how to model a launch abort system for an orbiter heading into outer space. If a fault occurs during the launch, the orbiter aborts the launch. Depending on when the fault occurs, the orbiter returns to the launch site, returns to a downrange landing site, attempts to land after orbiting once around the Earth, or proceeds to a lower, stable orbit. A Simulink 3D Animation™ window displays a visualization of these steps. This simplified example does not model the dynamics of the fuel, boosters, and tank subsystems.



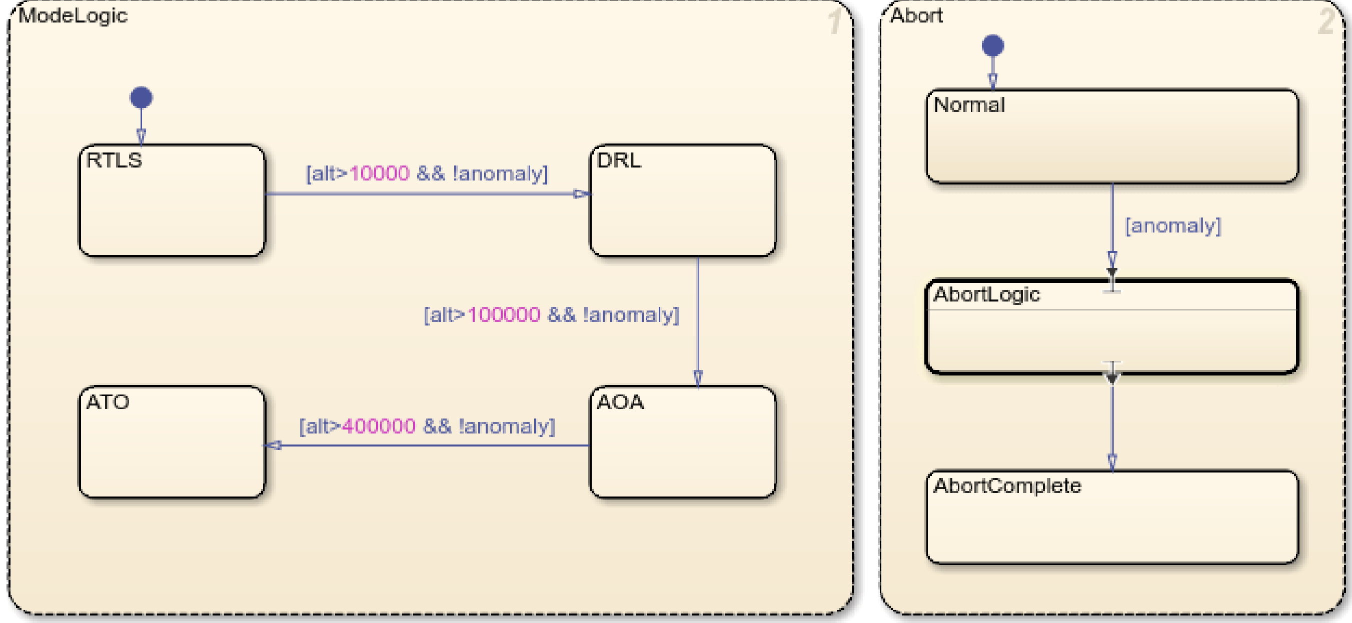
Monitor Orbiter Altitude

In this example, a Stateflow® chart monitors the flight of the orbiter and schedules the appropriate launch abort actions when the orbiter encounters an anomaly.



The chart consists of two parallel states: `ModeLogic` and `Abort`. The `ModeLogic` state contains four substates that describe the possible launch abort scenarios based on the altitude of the orbiter:

- Return to Launch Site (`RTLS`) — If the altitude is less than 10,000 meters, the orbiter dumps the fuel, releases the solid rocket boosters and the external tank, and returns to the launch site.
- Downrange landing (`DRL`) — If the altitude is between 10,000 and 100,000 meters, the orbiter releases the solid rocket boosters and the external tank and returns to a downrange landing site.
- Abort Once Around (`AOA`) — If the altitude is between 100,000 and 400,000 meters, the orbiter releases the external tank, circles the Earth once, and proceeds to re-entry.
- Abort to Orbit (`ATO`) — If the altitude is greater than 400,000 meters, the orbiter abandons the intended orbit and proceeds to a lower, stable orbit.



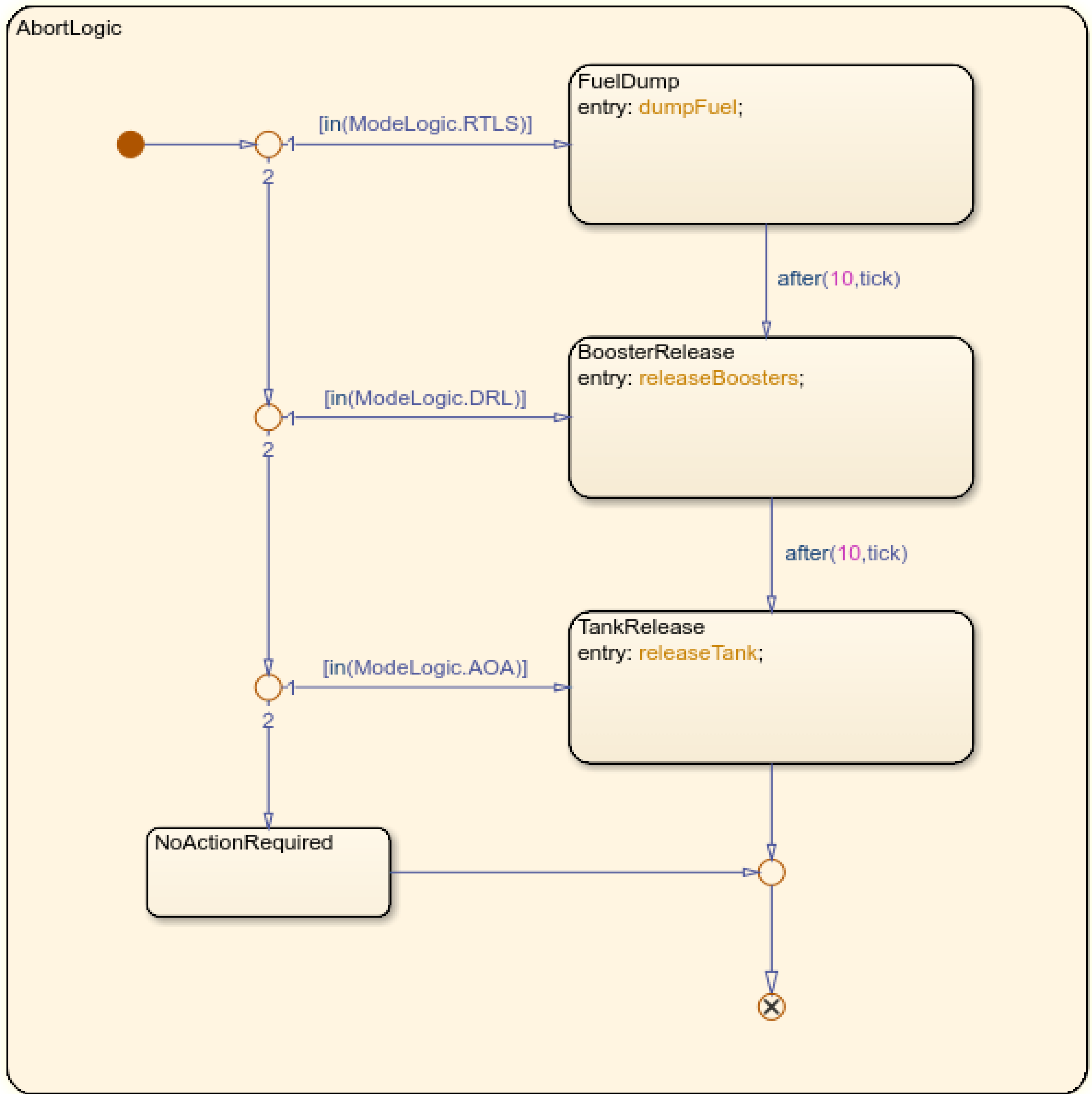
Schedule Launch Abort Actions

The `Abort` state controls the behavior of the orbiter when an anomaly is detected. If an anomaly occurs, the system transitions from the `Normal` substate to the `AbortLogic` subchart before transitioning to the `AbortComplete` substate.

The transition into the `AbortLogic` subchart connects to an entry port. Similarly, the transition out of the subchart begins at an exit port. Each port has a matching junction that marks the entry or exit point inside the subchart. The junctions isolate the internal logic of the subchart which, depending on the launch abort scenario, schedules three possible actions:

- Dump the fuel.
- Release the sold rocket boosters.
- Release the external tank.

If the orbiter is in the ATO scenario, none of these actions is required.

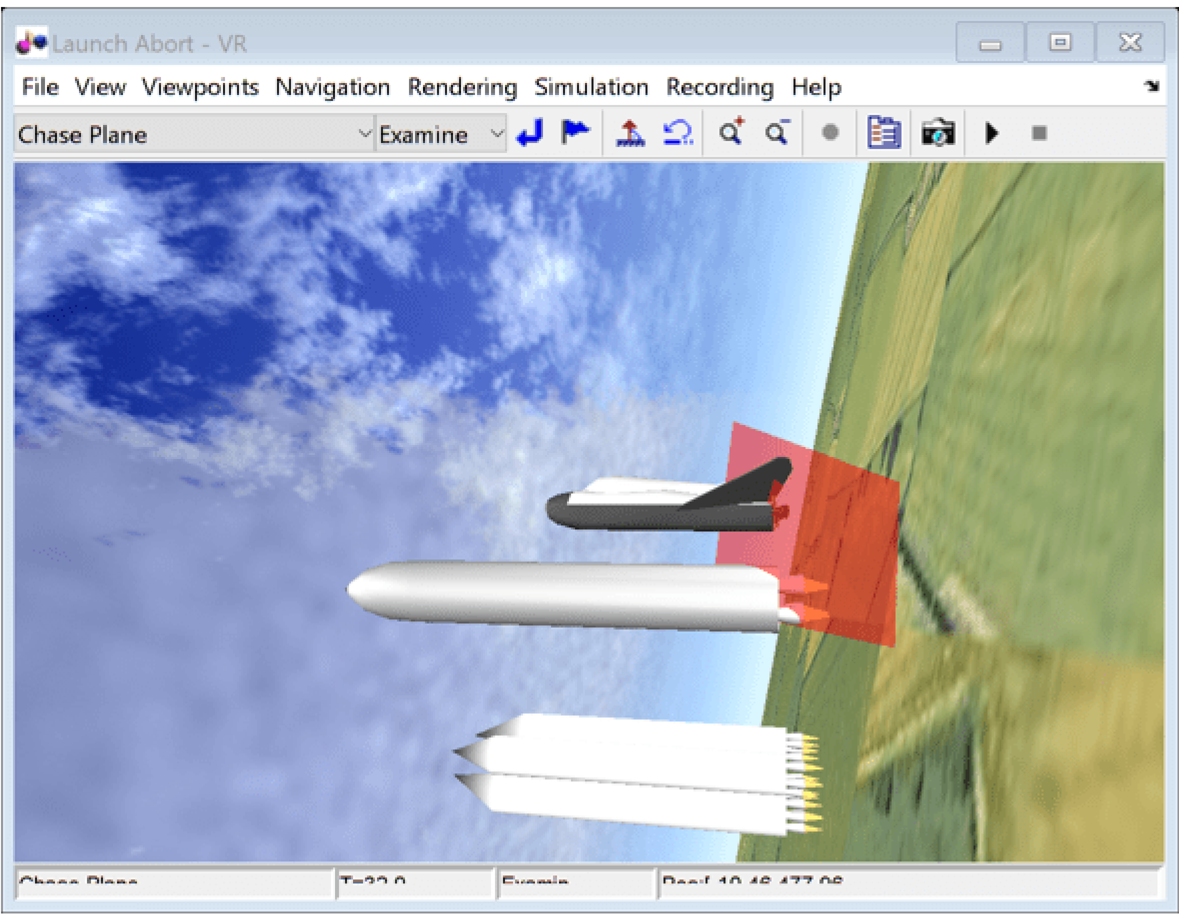


For more information about entry and exit ports, see "Create Entry and Exit Connections Across State Boundaries" on page 1-61.

Simulate the Model

To run a simulation of the model:

1. Double-click the `Inputs` block. In the Signal Editor dialog box, select a launch abort scenario from the **Active Scenario** list. The default scenario is `RTLS_Abort`.
2. Click **Run**. The Simulink 3D Animation window displays a visualization of the launch.
3. To view the orbiter from different perspectives, in the Simulink 3D Animation window, use the **Viewpoint** dropdown menu. For example, you can see entire flight of the orbiter by selecting `Chase Plane`.



Reference

Nelson, Douglas, John Bradford, and John Olds. "Abortability Metrics: Quantifying Intact Abort Mode Availability for Reusable Launch Vehicles." In *Space 2006*. San Jose, California: American Institute of Aeronautics and Astronautics, 2006. https://doi.org/10.2514/6.2006-7293.

See Also

More About

- “Create Entry and Exit Connections Across State Boundaries” on page 1-61
- “Check State Activity by Using the in Operator” on page 11-24
- “Control Chart Execution by Using Temporal Logic” on page 14-35
- “Simulink 3D Animation”

Model a Fault-Tolerant Fuel Control System

This example shows how to combine Stateflow® with Simulink® to efficiently model hybrid systems. This type of modeling is particularly useful for systems that have numerous possible operational modes based on discrete events. Traditional signal flow is handled in Simulink while changes in control configuration are implemented in Stateflow. The model described below represents a fuel control system for a gasoline engine. The system is highly robust in that individual sensor failures are detected and the control system is dynamically reconfigured for uninterrupted operation.

Analysis and Physics

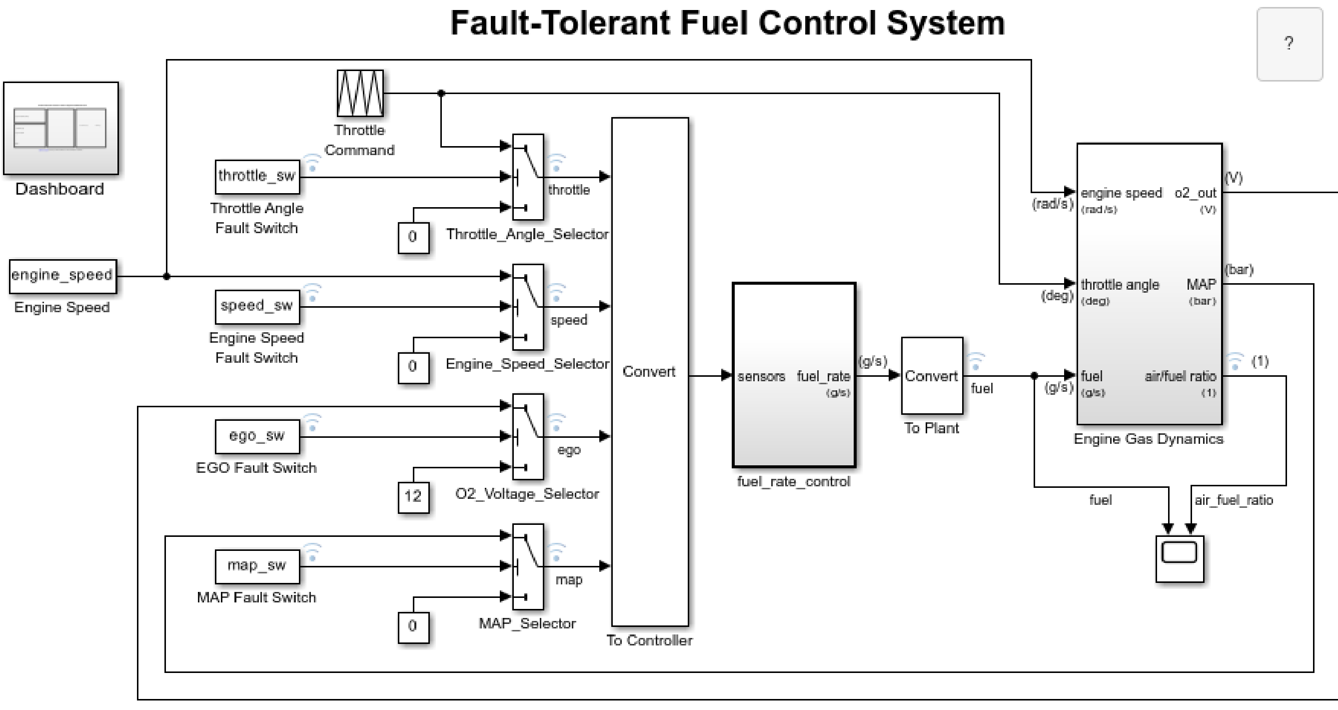
Physical and empirical relationships form the basis for the throttle and intake manifold dynamics of this model. The air-fuel ratio is computed by dividing the air mass flow rate (pumped from the intake manifold) by the fuel mass flow rate (injected at the valves). The ideal (i.e. stoichiometric) mixture ratio provides a good compromise between power, fuel economy, and emissions. The target air-fuel ratio for this system is 14.6. Typically, a sensor determines the amount of residual oxygen present in the exhaust gas (EGO). This gives a good indication of the mixture ratio and provides a feedback measurement for closed-loop control. If the sensor indicates a high oxygen level, the control law increases the fuel rate. When the sensor detects a fuel-rich mixture, corresponding to a very low level of residual oxygen, the controller decreases the fuel rate.

Modeling

Figure 1 shows the top level of the Simulink model. To open the model, click **Open Model**. Press the Play button in the model window toolbar to run the simulation. The model loads necessary data into the model workspace from `sldemo_fuelsys_data.m`. The model logs relevant data to MATLAB workspace in a data structure called `sldemo_fuelsys_output` and streams the data to the Simulation Data Inspector. Logged signals are marked with a blue indicator while streaming signals are marked with the light blue badge (see Figure 1).

Note that loading initial conditions into the model workspace keeps simulation data isolated from data in other open models that you may have open. This also helps avoid MATLAB workspace cluttering. To view the contents of the model workspace select Modeling > Model Explorer, and click on Model Workspace from the Model Hierarchy list.

Notice that units are visible on the model and subsystem icons and signal lines. Units are specified on the ports and on the bus object.



Copyright 1990-2022 The MathWorks, Inc.

Figure 1: Top-level diagram for the fuel control system model

The Dashboard subsystem (shown in Figure 2) allows you to interact with the model during simulation. The Fault Injection switches can be moved from the Normal to Fail position to simulate sensor failures, while the Engine Speed selector switch can be toggled to change the engine speed. The fuel and air/fuel ratio signals are visualized using the dashboard gauges and scopes to provide visual feedback during a simulation run.

Figure 2: Dashboard subsystem for the fuel control system model

The fuel_rate_control uses signals from the system's sensors to determine the fuel rate which gives a stoichiometric mixture. The fuel rate combines with the actual air flow in the engine gas dynamics model to determine the resulting mixture ratio as sensed at the exhaust.

You can selectively disable each of the four sensors (throttle angle, speed, EGO and manifold absolute pressure [MAP]) by using the slider switches in the dashboard subsystem, to simulate failures. Simulink accomplishes this by binding slider switches to the value parameter of the constant block. Double-click on the dashboard subsystem to open the control dashboard to change the position of the switch. Similarly, you can induce the failure condition of a high engine speed by toggling the engine speed switch on the dashboard subsystem. A Repeating Table block provides the throttle angle input and periodically repeats the sequence of data specified in the mask.

The fuel_rate_control block, shown in Figure 3, uses the sensor input and feedback signals to adjust the fuel rate to give a stoichiometric ratio. The model uses three subsystems to implement this strategy: control logic, airflow calculation, and fuel calculation. Under normal operation, the model estimates the airflow rate and multiplies the estimate by the reciprocal of the desired ratio to give the fuel rate. Feedback from the oxygen sensor provides a closed-loop adjustment of the rate estimation in order to maintain the ideal mixture ratio.

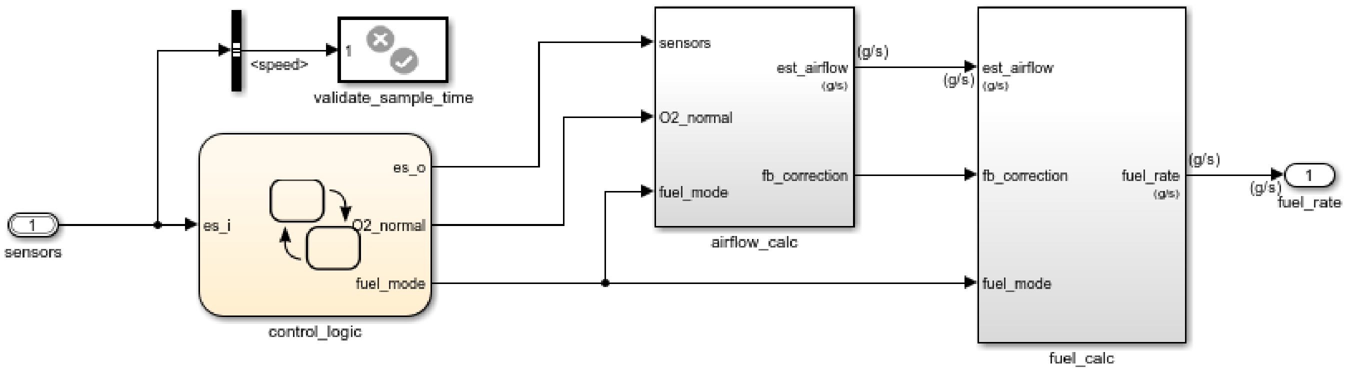


Figure 3: Fuel rate controller subsystem

Control Logic

A single Stateflow chart, consisting of a set of six parallel states, implements the control logic in its entirety. The four parallel states shown at the top of Figure 4 correspond to the four individual sensors. The remaining two parallel states at the bottom consider the status of the four sensors simultaneously and determine the overall system operating mode. The model synchronously calls the entire Stateflow diagram at a regular sample time interval of 0.01 sec. This permits the conditions for transitions to the correct mode to be tested on a timely basis.

To open the control_logic Stateflow chart, double-click on it in the fuel_rate_control subsystem.

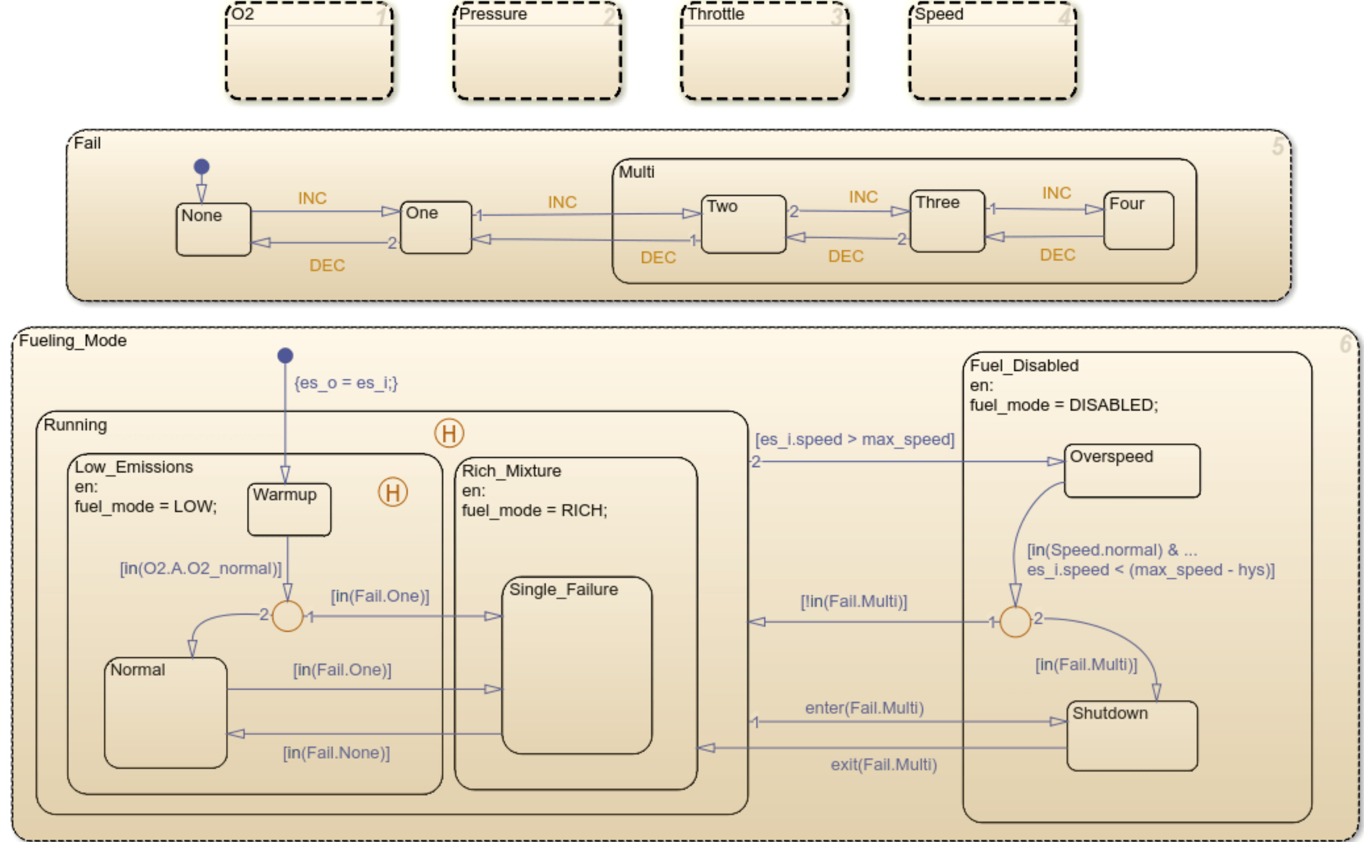


Figure 4: The control logic chart

When execution begins, all of the states start in their `normal` mode with the exception of the oxygen sensor (EGO). The `O2_warmup` state is entered initially until the warmup period is complete. The system detects throttle and pressure sensor failures when their measured values fall outside their nominal ranges. A manifold vacuum in the absence of a speed signal indicates a speed sensor failure. The oxygen sensor also has a nominal range for failure conditions but, because zero is both the minimum signal level and the bottom of the range, failure can be detected only when it exceeds the upper limit.

Regardless of which sensor fails, the model always generates the directed event broadcast `Fail.INC`. In this way the triggering of the universal sensor failure logic is independent of the sensor. The model also uses a corresponding sensor recovery event, `Fail.DEC`. The `Fail` state keeps track of the number of failed sensors. The counter increments on each `Fail.INC` event and decrements on each Fail.DEC event. The model uses a superstate, `Multi`, to group all cases where more than one sensor has failed.

The bottom parallel state represents the fueling mode of the engine. If a single sensor fails, operation continues but the air/fuel mixture is richer to allow smoother running at the cost of higher emissions. If more than one sensor has failed, the engine shuts down as a safety measure, since the air/fuel ratio cannot be controlled reliably.

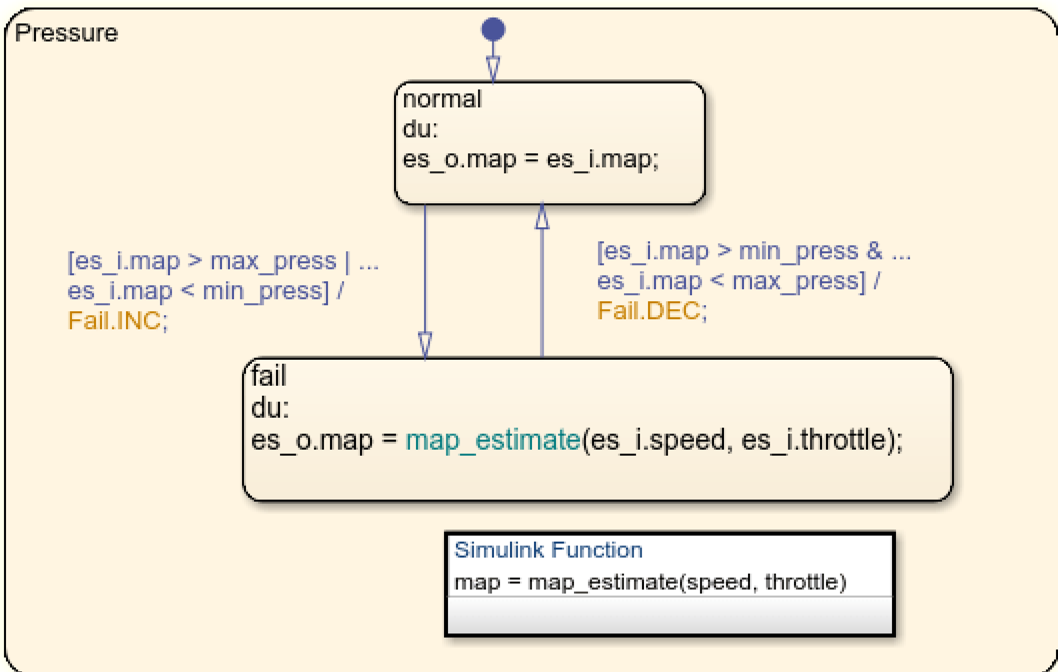
During the oxygen sensor warm-up, the model maintains the mixture at normal levels. If this is unsatisfactory, you can change the design by moving the warm-up state to within the `Rich_Mixture`

superstate. If a sensor failure occurs during the warm-up period, the `Single_Failure` state is entered after the warm-up time elapses. Otherwise, the `Normal` state is activated at this time.

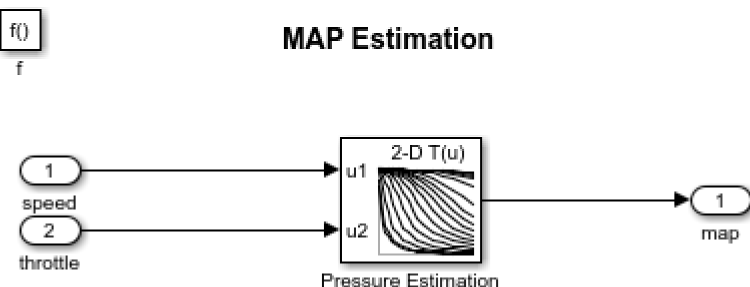
A protective overspeed feature has been added to the model by creating a new state in the `Fuel_Disabled` superstate. Through the use of history junctions, we assured that the chart returns to the appropriate state when the model exits the overspeed state. As the safety requirements for the engine become better specified, we can add additional shutdown states to the `Fuel_Disabled` superstate.

Sensor Correction

When a sensor fails, the model computes an estimate of the sensor. For example, open the pressure sensor calculation. Under normal sensor operation, the model uses the value of the pressure sensor. Otherwise, the model estimates the value.



The model computes an estimate of manifold pressure as a function of the engine speed and throttle position. To compute the value, the model uses a Simulink function inside Stateflow.



Airflow Calculation

The Airflow Calculation block (shown in Figure 6) is the location for the central control laws. This block is found inside the fuel_rate_control subsystem (open this block). The block estimates the intake air flow to determine the fuel rate which gives the appropriate air/fuel ratio. Closed-loop control adjusts the estimation according to the residual oxygen feedback in order to maintain the mixture ratio precisely. Even when a sensor failure mandates open-loop operation, the most recent closed-loop adjustment is retained to best meet the control objectives.

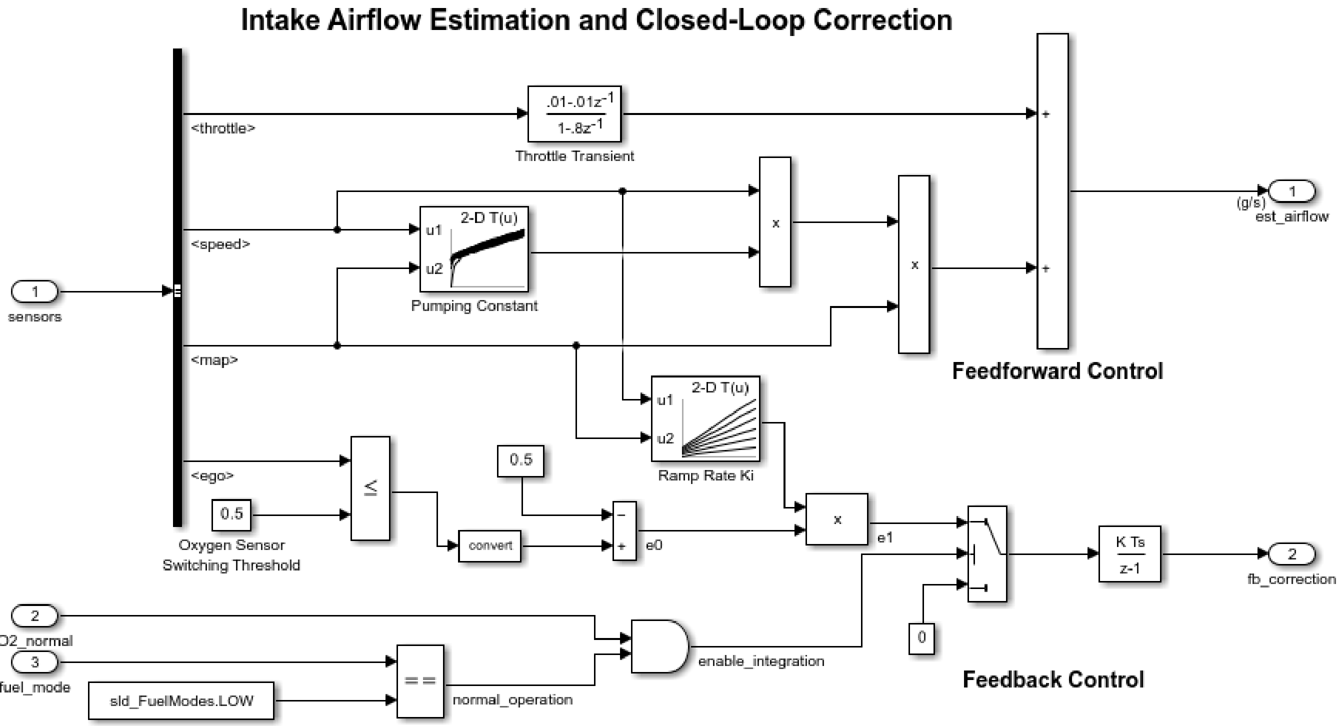


Figure 6: Airflow estimation and correction

Equation 1

The engine's intake air flow can be formulated as the product of the engine speed, the manifold pressure and a time-varying scale factor.

$$q = \frac{N}{4\pi} V_{cd} \nu \frac{P_m}{RT} = C_{pump}(N, P_m) N P_m = \text{ intake mass flow}$$

$N =$ engine angular speed (Rad/sec)

$V_{cd} =$ engine cylinder displacement volume

$\nu =$ volumetric efficiency

$P_m =$ manifold pressure

$R, T =$ specific gas constant, gas temperature

`Cpump` is computed by a lookup table and multiplied by the speed and pressure to form the initial flow estimate. During transients, the throttle rate, with the derivative approximated by a high-pass filter, corrects the air flow for filling dynamics. The control algorithm provides additional correction according to Equation 2.

Equation 2

$$e_0 = 0.5 \text{ for } EGO \leq 0.5$$

$$e_0 = -0.5 \text{ for } EGO > 0.5$$

$$e_1 = K_i(N, P_m)e_0 \text{ for } EGO \leq 0.5$$

$$\dot{e_2} = e_1 \text{ for LOW mode with valid EGO signal}$$

$$\dot{e_2} = 0 \text{ for RICH, DISABLE or EGO warmup}$$

$$e_0, e_1, e_2 = \text{ intermediate error signals}$$

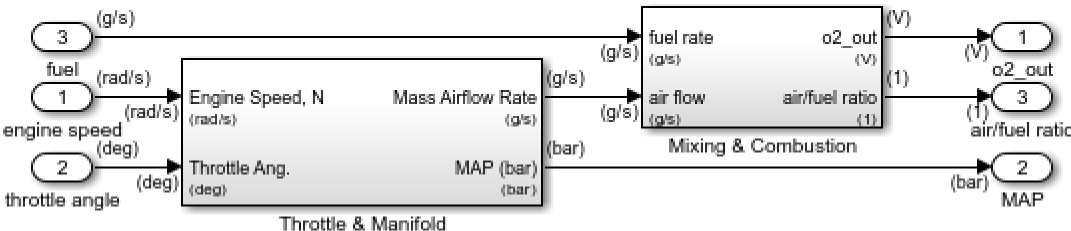


Figure 7: Engine Gas Dynamics subsystem

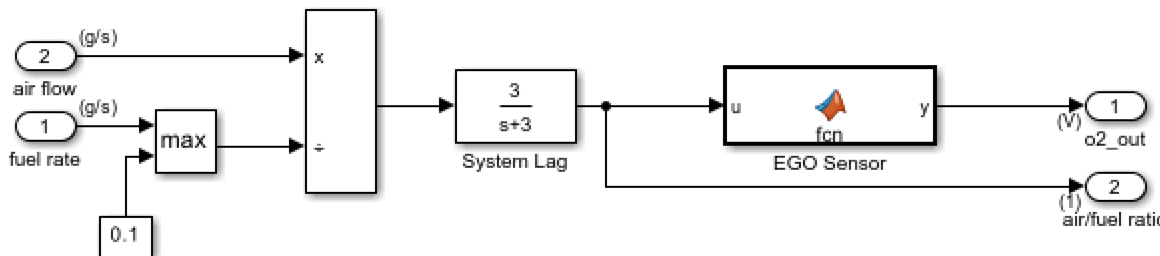


Figure 8: Mixing & Combustion block within the Engine Gas Dynamics subsystem

The nonlinear oxygen sensor (EGO Sensor block) is found inside the Mixing & Combustion block (see Figure 8) within the Engine Gas Dynamics subsystem (see Figure 7). EGO Sensor is modeled as a hyperbolic tangent function, and it provides a meaningful signal when in the vicinity of 0.5 volt. The raw error in the feedback loop is thus detected with a switching threshold, as indicated in Equation 2. If the air-fuel ratio is low (the mixture is lean), the original air estimate is too small and needs to be increased. Conversely, when the oxygen sensor output is high, the air estimate is too large and needs to be decreased. Integral control is utilized so that the correction term achieves a level that brings about zero steady-state error in the mixture ratio.

The normal closed-loop operation mode, LOW, adjusts the integrator dynamically to minimize the error. The integration is performed in discrete time, with updates every 10 milliseconds. When

operating open-loop however, in the RICH or O2 failure modes, the feedback error is ignored and the integrator is held. This gives the best correction based on the most recent valid feedback.

Fuel Calculation

The fuel_calc subsystem (within the fuel_rate_control subsystem, see Figure 9) sets the injector signal to match the given airflow calculation and fault status. The first input is the computed airflow estimation. This is multiplied with the target fuel/air ratio to get the commanded fuel rate. Normally the target is stoichiometric, i.e. equals the optimal air to fuel ratio of 14.6. When a sensor fault occurs, the Stateflow control logic sets the mode input to a value of 2 or 3 (RICH or DISABLED) so that the mixture is either slightly rich of stoichiometric or is shut down completely.

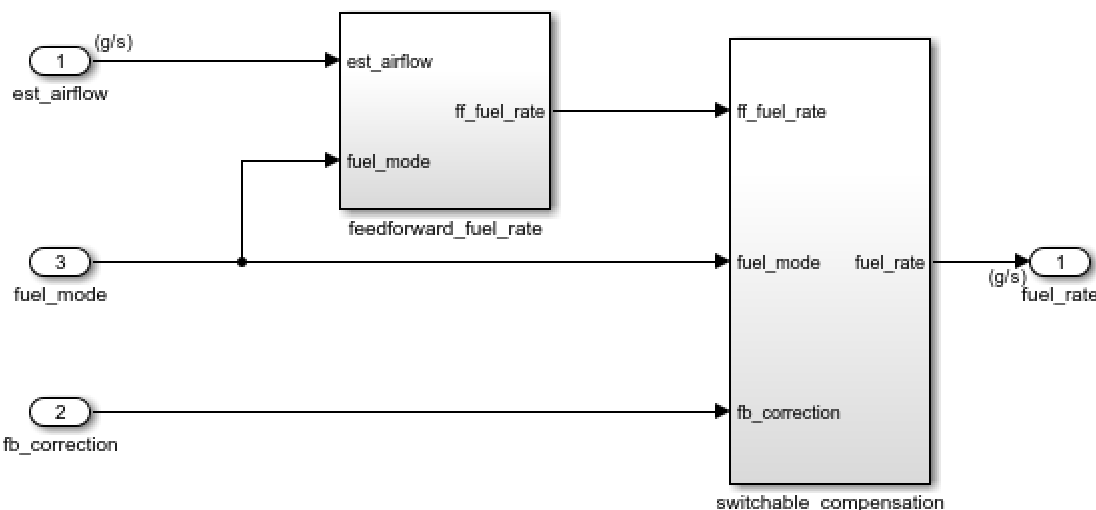


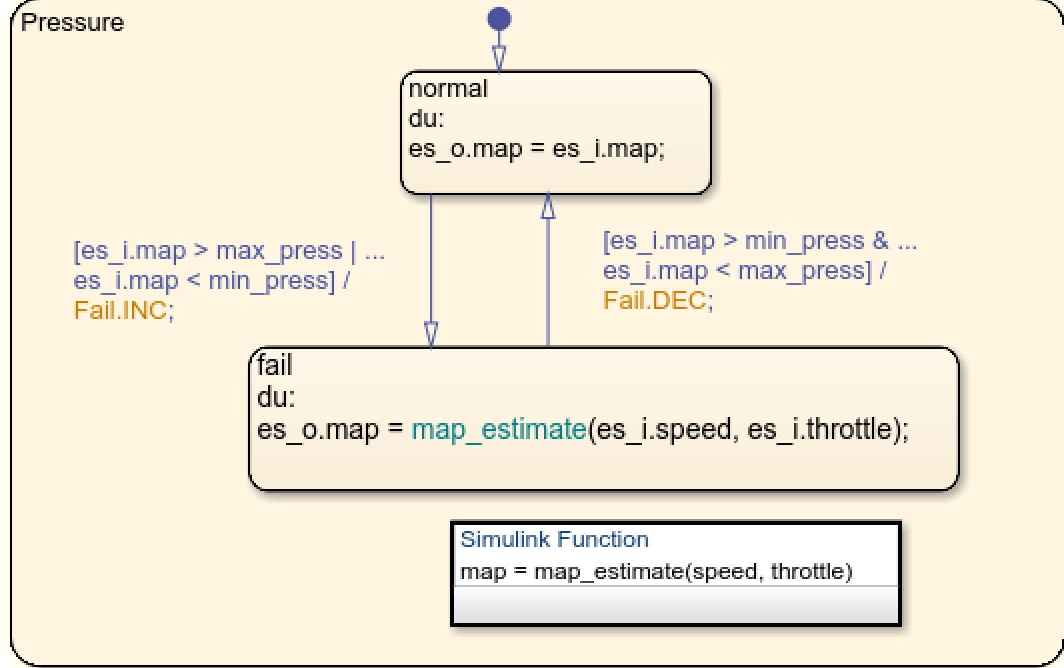
Figure 9: fuel_calc subsystem

The fuel_calc subsystem (Figure 9) employs adjustable compensation (Figure 10) in order to achieve different purposes in different modes. In normal operation, phase lead compensation of the feedback correction signal adds to the closed-loop stability margin. In RICH mode and during EGO sensor failure (open loop), however, the composite fuel signal is low-pass filtered to attenuate noise introduced in the estimation process. The end result is a signal representing the fuel flow rate which, in an actual system, would be translated to injector pulse times.

Loop Compensation and Filtering

2
fuel_mode
u1
case [LOW]:
case [RICH]:
default:
[RICH]
3
fb_correction
1
ff_fuel_rate
case: { }
fb_correction
fuel_rate
ff_fuel_rate
low_mode
[RICH]
case: { }
ff_fuel_rate
fuel_rate
fb_correction
rich_mode
default: { }
fuel_rate
disabled_mode
merge
1
fuel_rate

Figure 10: Switchable compensation subsystem



Results and Conclusions

Simulation results are shown in Figure 11 and Figure 12. The simulation is run with a throttle input that ramps from 10 to 20 degrees over a period of two seconds, then goes back to 10 degrees over the next two seconds. This cycle repeats continuously while the engine is held at a constant speed so

that the user can experiment with different fault conditions and failure modes. Click on a sensor fault switch in the dashboard subsystem to simulate the failure of the associated sensor. Repeat this operation to slide the switch back for normal operation.

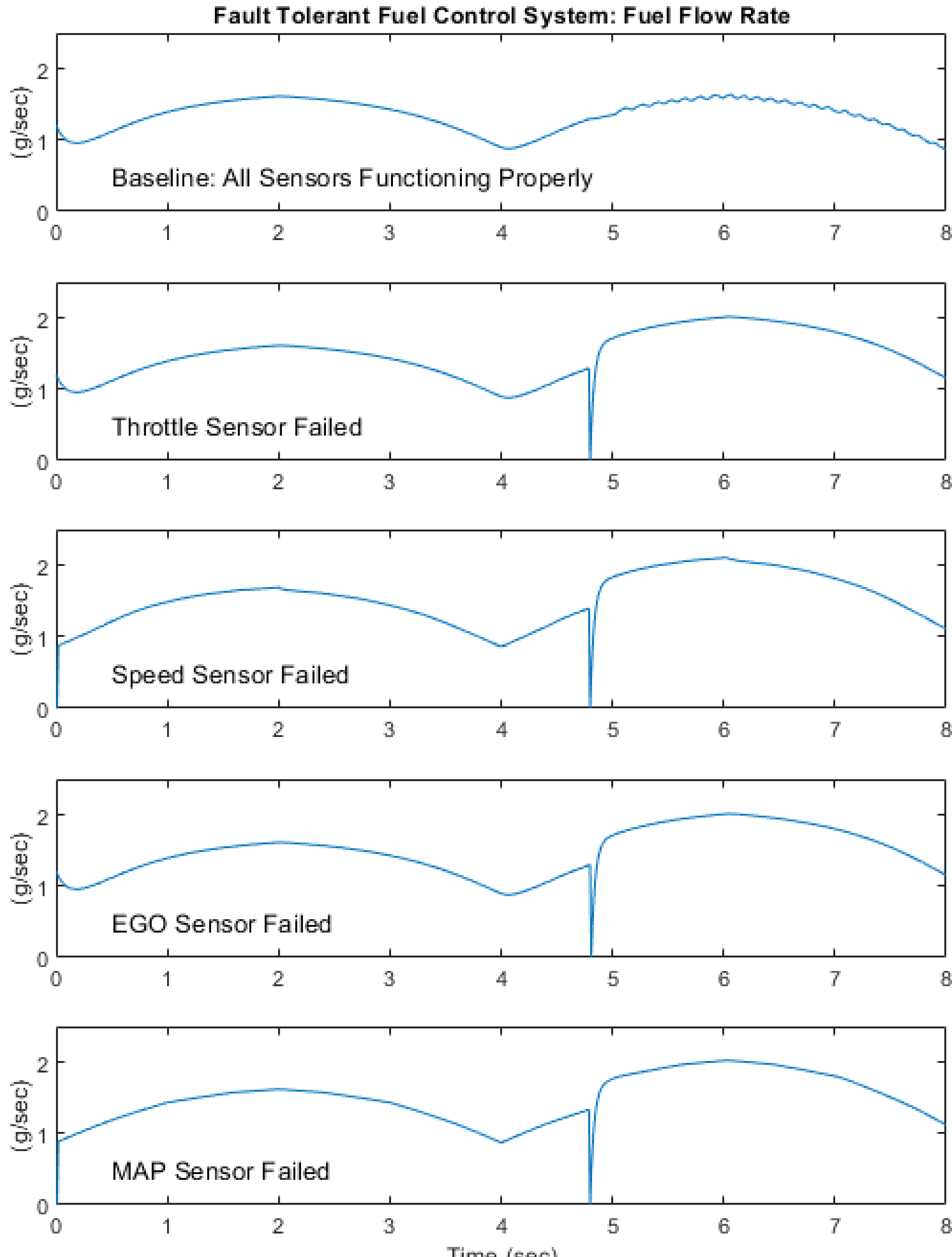


Figure 11: Comparing the fuel flow rate for different sensor failures

Figure 11 compares the fuel flow rate under fault-free conditions (baseline) with the rate applied in the presence of a single failure in each sensor individually. In each case note the nonlinear relationship between fuel flow and the triangular throttle command (shown in Figure 13). In the baseline case, the fuel rate is regulated tightly, exhibiting a small ripple due to the switching nature of the EGO sensor's input circuitry. In the other four cases the system operates open loop. The control strategy is proven effective in maintaining the correct fuel profile in the single-failure mode. In each of the fault conditions, the fuel rate is essentially 125% of the baseline flow, fulfilling the design objective of 80% rich.

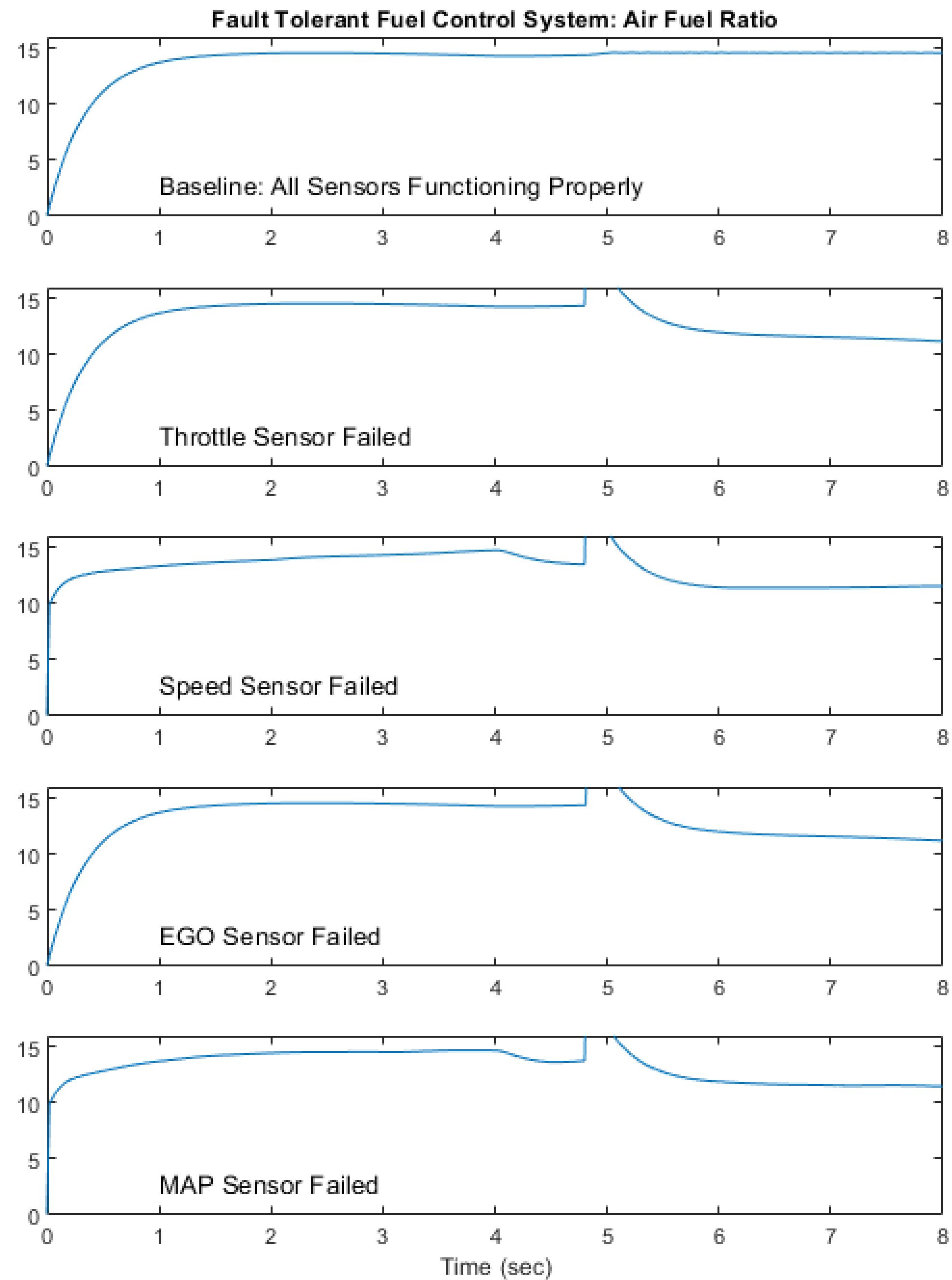


Figure 12: Comparing the air-fuel ratio for different sensor failures

Figure 12 plots the corresponding air/fuel ratio for each case. The baseline plot shows the effects of closed-loop operation. The mixture ratio is regulated very tightly to the stoichiometric objective of

14.6. The rich mixture ratio is shown in the bottom four plots of Figure 12. Although they are not tightly regulated, as in the closed-loop case, they approximate the objective of air/fuel (0.8*14.6=11.7).

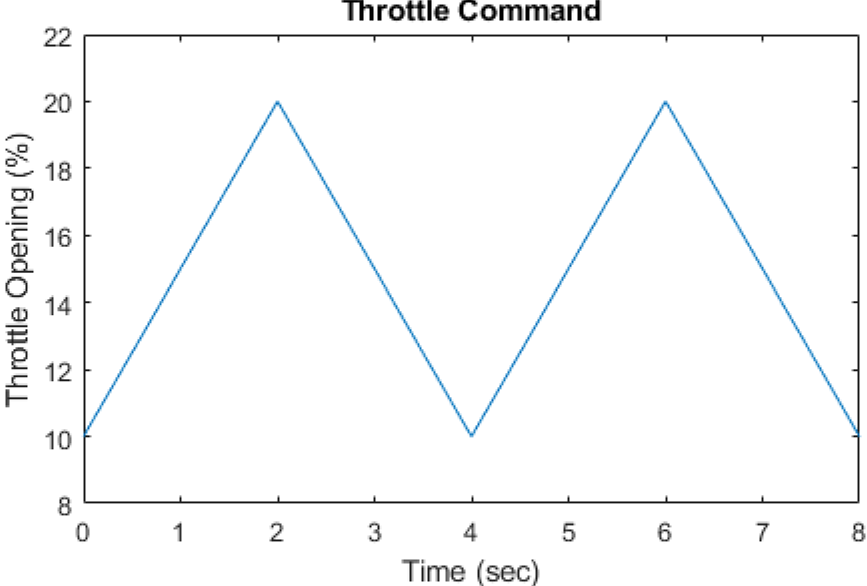


Figure 13: Throttle command

The transient behavior of the system is shown in Figure 14. With a constant 12 degree throttle angle and the system in steady-state, a throttle failure is introduced at t = 2 and corrected at t = 5. At the onset of the failure, the fuel rate increases immediately. The effects are seen at the exhaust as the rich ratio propagates through the system. The steady-state condition is then quickly recovered when closed-loop operation is restored.

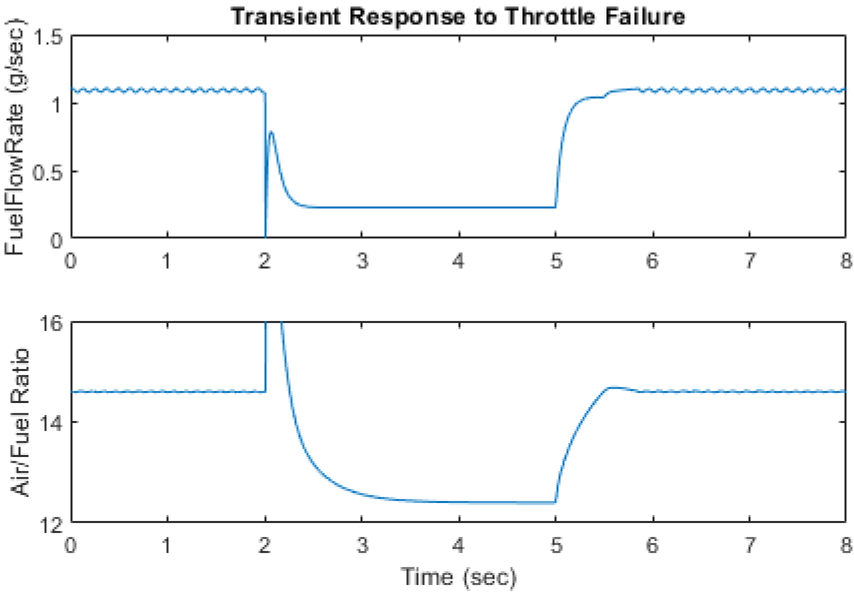


Figure 14: Transient response to fault detection

Remarks

If you enable animation in the Stateflow debugger, the state transitions are highlighted in the Stateflow diagram (see Figure 4) as the various states are activated. The sequence of activation is indicated by changing colors. This closely coupled synergy between Stateflow and Simulink fosters the modeling and development of complete control systems.

See Also

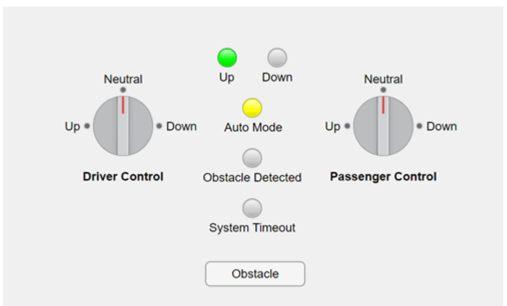
More About

- “Air-Fuel Ratio Control System with Fixed-Point Data” (Embedded Coder)
- “Air-Fuel Ratio Control System with Stateflow Charts” (Embedded Coder)

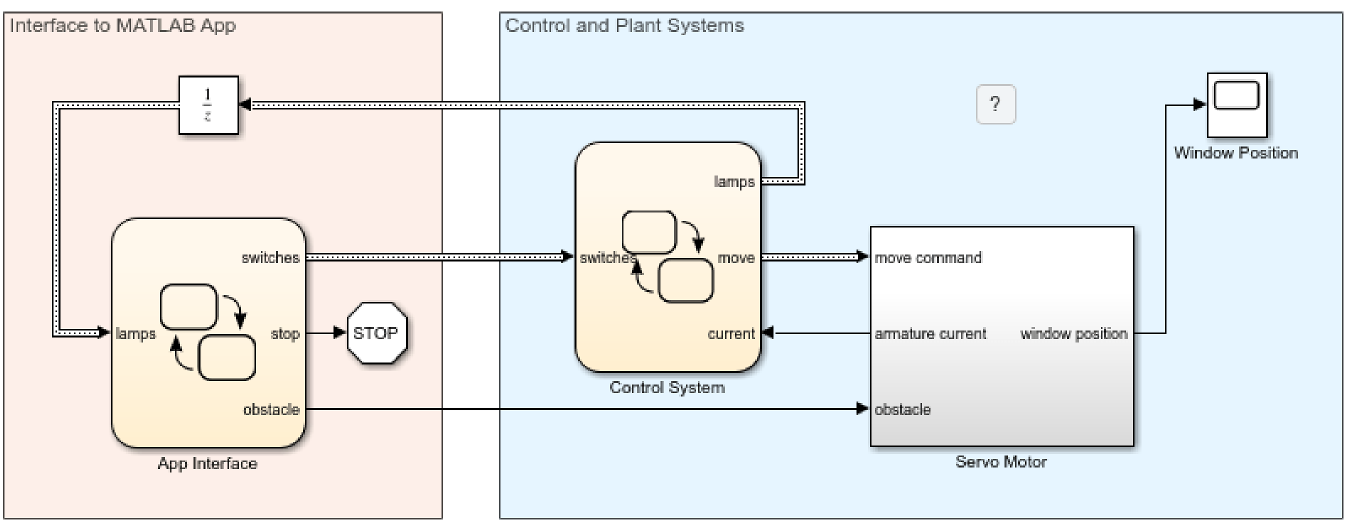
Model a Power Window Controller

This example shows how to create an interface between a Stateflow® chart that uses MATLAB® as the action language and a MATLAB app created in App Designer. For more information on connecting a Stateflow chart that uses C as the action language to a MATLAB app, see “Simulate a Media Player” on page 21-14.

In this example, an automotive power window system raises and lowers the passenger-side window in response to a pair of window control switches. The switches in the MATLAB app represent the controls on the driver and passenger doors. The app also contains several indicator lamps that monitor the status of the power window system and a button for introducing an obstacle in the path of the window.



The Stateflow chart `App Interface` provides a bidirectional connection between the MATLAB app and the control and plant systems in the Simulink® model. When you point a switch in the app to a new position, the chart sends a corresponding "Up," "Down," or "Neutral" command to the power window control system. Conversely, when the control system changes state, the chart enables or disables the corresponding status lamps in the app.



To run the example, open the Simulink model and click **Run**. The chart `App Interface` opens the app and initializes the control and plant systems in the power window system. To stop the simulation, click **Stop** or close the app.

Connect Chart to MATLAB App

The chart `App Interface` is already configured to communicate with the MATLAB app `sf_power_window_app`. To create a bidirectional connection between your MATLAB app and a Stateflow chart that uses MATLAB as the action language, follow these steps. In the MATLAB app:

1. Create a custom property to interface with the chart during simulation. The app uses this property to access chart inputs, chart outputs, and local data. For more information, see “Share Data Within App Designer Apps”.
2. Modify the `startupFcn` callback for the app by adding a new input argument and storing its value as the property that you created in the previous step. For more information, see “Callbacks in App Designer”.

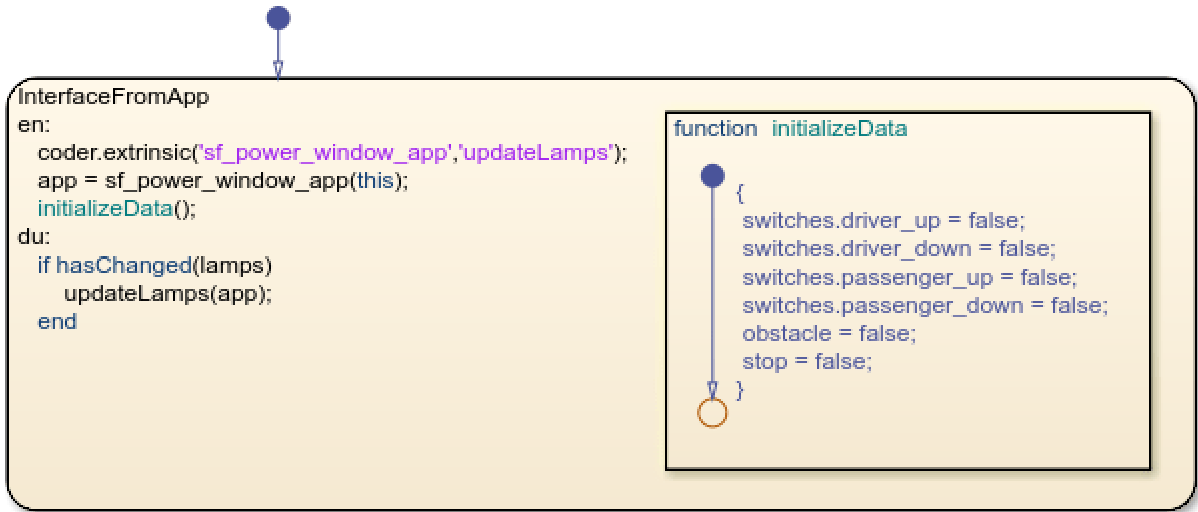
In the Stateflow chart:

1. Create a local data object to interface with the app. The chart uses this local data object as an argument when it calls helper functions in the app.
2. Set the type of the local data object you created in the previous step to `Inherit: From definition in chart`. For more information, see “Specify Type of Stateflow Data” on page 10-20.
3. Call the `coder.extrinsic` function to declare the app and any helper functions as extrinsic MATLAB code. For more information, see “Call Extrinsic MATLAB Functions in Stateflow Charts” on page 14-13.
4. Run the app using the keyword `this` as an argument to give the app access to the chart during simulation. Store the value returned by the function call to the app as the local data object that you created to interface with the app.

In this example, the power window app uses a property called `chart` to interface with the chart `App Interface`. The app callbacks use this property to write to the chart outputs:

- When you move the driver-side control switch to a new position, the `DriverControlValueChanged` callback sets the values of `switches.driver_up` and `switches.driver_down`.
- When you move the passenger-side control switch to a new position, the `PassengerControlValueChanged` callback sets the values of `switches.passenger_up` and `switches.passenger_down`.
- When you click the **Obstacle** button, the `ObstacleButtonPushed` callback sets the value of `obstacle` to `true`.
- When you close the app, the `UIFigureCloseRequest` callback sets the value of `stop` to `true`.

Conversely, in the chart, the entry actions in the `InterfaceWithApp` state run the app `sf_power_window_app` and store the returned value as the local data object `app`. The chart uses this local data object when it calls the helper function `updateLamps`. In the app, this helper function turns the lamps on and off based on the value of the chart input `lamps`.



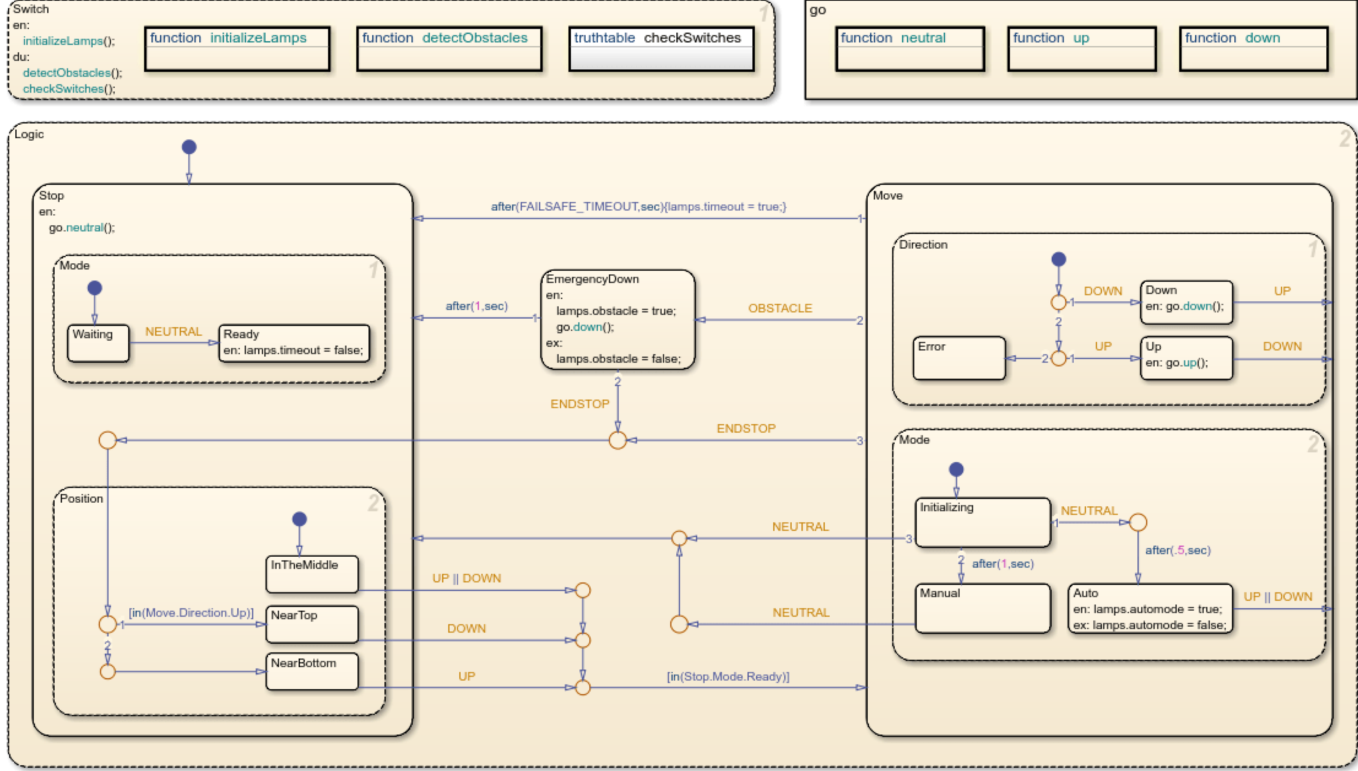
Control System Design

The power window control system uses mode logic to determine when the window should move and outputs a unified motion command to a servo motor. To lower production costs, the control system does not keep track of the window position. Instead, it relies on a signal from the servo motor to determine when the window is fully open or fully closed, or when it encounters an obstacle.

The control system meets these performance requirements:

1. The window must open or close completely in 5 seconds.
2. The motor must stop when the window reaches a fully opened or fully closed position.
3. The motor must be able to detect an obstacle when the window is moving up. When the motor detects an obstacle in the path of the window, the window must be lowered for one second or until the window is fully open.
4. The motor must stop after 10 seconds of continuous movement in any direction. This requirement provides a fail-safe protection for the window mechanism, motor, and drive.
5. If a control switch is pressed for less than half a second, or if it is pressed for longer than one second, the window must stop when the switch is released.
6. If a control switch is pressed for longer than half a second and released before one second, the window must open or close completely unless it is interrupted by a new command or by an obstacle. This requirement represents the automatic mode capability of the power window.
7. The driver-side control has priority over the passenger-side control.
8. Obstacle detection has priority over both driver-side and passenger-side controls.

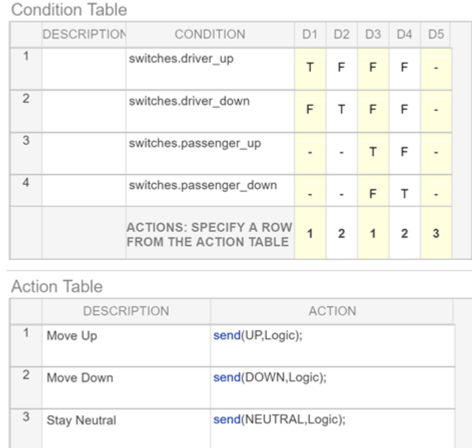
The Stateflow chart `Control System` models an event-driven controller that satisfies these requirements. The chart consists of two states (`Switch` and `Logic`) in parallel decomposition. These states react to changes in the chart inputs, determine the operating mode of the power window system, and manage the output signals that activate the servo motor.



Monitor Controller Input

The state `Switch` reads the values of the chart inputs and broadcasts local events to change the operating mode of the power window system. For more information, see “Broadcast Local Events to Synchronize Parallel States” on page 12-25.

At every time step of the simulation, the state calls the truth table function `checkSwitches` to determine the positions of the driver-side and passenger-side control switches. Depending on the value of the input structure `switches`, this function broadcasts the `UP`, `DOWN`, and `NEUTRAL` events. Because the function ignores any input from the passenger-side control when the driver-side control is not in the "Neutral" position, the driver-side control has priority over the passenger-side control, as specified by requirement 7.



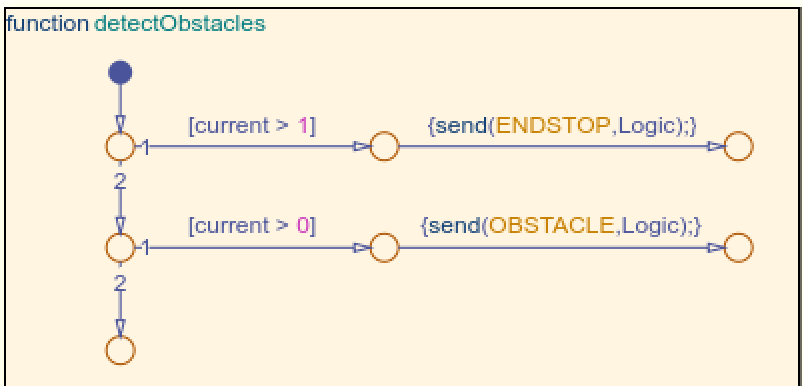
Condition Table

	DESCRIPTION	CONDITION	D1	D2	D3	D4	D5
1		switches.driver_up	T	F	F	F	-
2		switches.driver_down	F	T	F	F	-
3		switches.passenger_up	-	-	T	F	-
4		switches.passenger_down	-	-	F	T	-
		ACTIONS: SPECIFY A ROW FROM THE ACTION TABLE	1	2	1	2	3

Action Table

	DESCRIPTION	ACTION
1	Move Up	send(UP,Logic);
2	Move Down	send(DOWN,Logic);
3	Stay Neutral	send(NEUTRAL,Logic);

In a similar way, the state calls the graphical function `detectObstacles` to determine the strength of the armature current in the servo motor. If the value of `current` is small and nonzero, an obstacle is present so the function broadcasts the event `OBSTACLE`. In contrast, if the value of `current` is large, the window has reached a fully open or fully closed position so the function broadcasts the event `ENDSTOP`. Because the chart calls `detectObstacles` before `checkSwitches`, obstacle detection has priority over both driver-side and passenger-side controls, as specified by requirement 8.



Determine Operating Mode

The state `Logic` incorporates fault detection algorithms to protect the window hardware and any obstacles in the path of the window. The state contains three substates, `Stop`, `Move`, and `EmergencyDown`, that represent the operating modes of the power window system.

Initially, the state `Stop` is active. This state contains two parallel substates named `Mode` and `Position`.

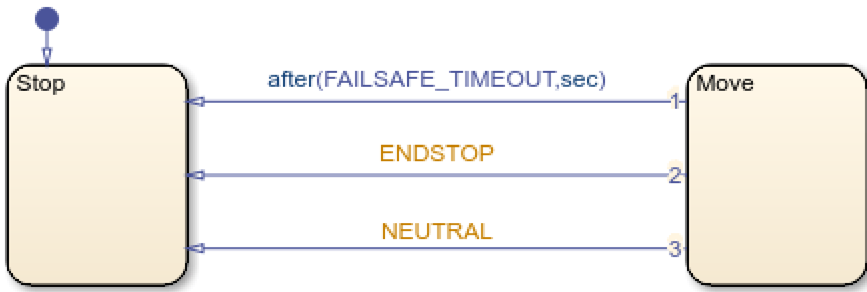
- `Mode` determines when the power system is ready to accept new commands from the control switches. The system is ready for new commands when both control switches are in the "Neutral" position.
- `Position` records whether the window is fully open, fully closed, or somewhere in the middle. The chart makes this determination by noting the direction in which the window is moving when the servo motor reaches the end of its range.

Stop remains active until a broadcast of the events UP or DOWN indicates a command from one of the control switches. As long as the window is not fully open or fully closed, these events trigger the transition to the state Move. However, the event UP is considered invalid when the window is already fully closed. Likewise, the event DOWN is invalid when the window is fully open.

The state Move is active whenever the window is in motion. This state implements several of the power window requirements related to automatic window movement and fault detection. The state has two parallel substates named Direction and Mode.

- Direction determines the direction in which the window should move and calls the functions go.up and go.down, as appropriate. These functions set the values of the output signals that control the servo motor and the "Up" and "Down" status lamps in the app.
- Mode implements the automatic and manual modes of the power window specified by requirements 5 and 6. This state has three exclusive substates (Initializing, Auto, and Manual). Initially, the substate Initializing is active. The substate waits for a broadcast of the NEUTRAL event, which indicates that the control switches have returned to the "Neutral" position. If the broadcast occurs within half a second of Initializing becoming active, the event triggers a transition to the Stop state, indicating that the window must stop moving. If the broadcast occurs after half a second but before one second of Initializing becoming active, the event triggers a transition to the substate Auto, indicating that the power window system is operating in its automatic mode. This substate remains active until it is interrupted by a broadcast of the events ENDSTOP (when the window is fully open or fully closed), OBSTACLE (when the window encounters an obstacle), or UP or DOWN (when the system receives a new command from one of the control switches). Finally, if the broadcast does not occur before one second of Initializing becoming active, the temporal logic expression after(1,sec) triggers the transition to the substate Manual. This substate remains active until a broadcast of the event NEUTRAL triggers the transition back to the Stop state.

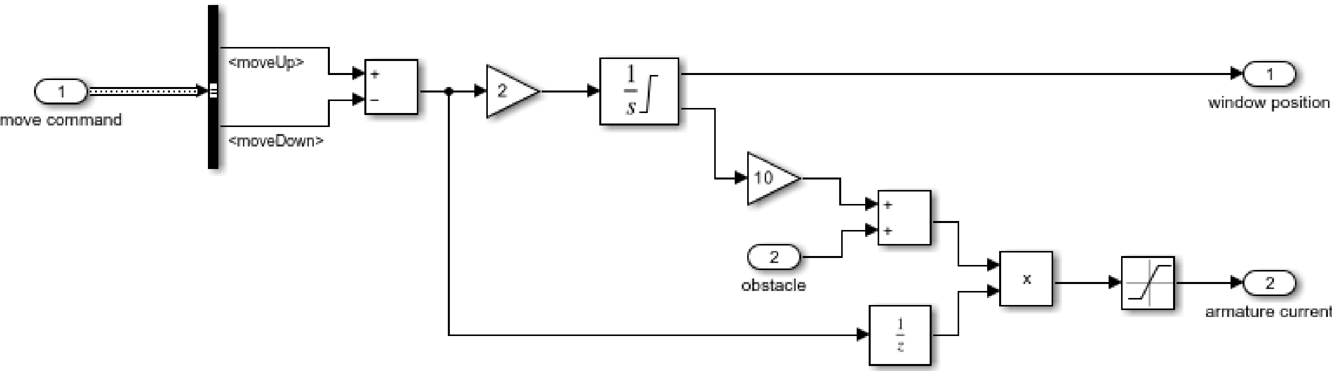
Independent of whether the system is in automatic or manual mode, the chart transitions directly from Move to Stop on the broadcast of the event ENDSTOP or when Move is active for longer than FAILSAFE_TIMEOUT seconds, as specified by requirements 2 and 4. By default, the value of this constant is set to 10.



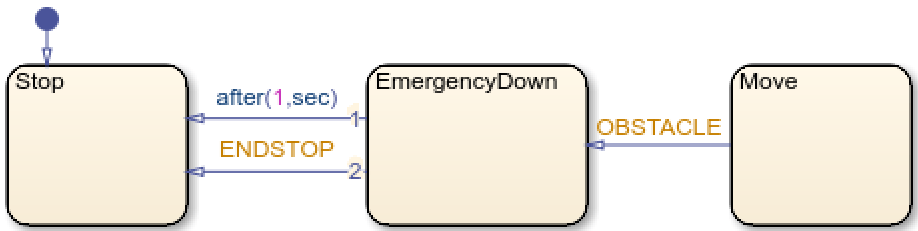
Detect Obstacles

When the window encounters an obstacle, the applied force on the window increases the load on the servo motor and causes a rise in the armature current. By monitoring for sharp increases in the armature current, the system detects obstacles in the path of the window.

In this example, a Simulink subsystem simulates the servo motor. The position of the window is computed by an Integrator (Simulink) block with saturation limits of 0 (fully open) and 10 (fully closed). Because the input to this block has a gain of 2, the window opens and closes completely in 5 seconds, as specified by requirement 1. When the Integrator block reaches a saturation point, the system output `armature current` increases to 10. This value indicates that the window is fully open or fully closed.



To introduce an obstacle in the path of the window, click the **Obstacle** button in the app while the window is moving up. The `App Interface` chart responds by sending a positive signal to the servo motor, which in turn produces a small rise in the armature current. In the `Control System` chart, the function `detectObstacles` registers this change in current and broadcasts the event `OBSTACLE`. In the `Logic` state, this event triggers the transition from the substate `Move` to the substate `EmergencyDown`. While this substate is active, the system moves the window down for one second or until the window is fully open. Then, the chart transitions back to the substate `Stop`, indicating that the window must stop moving, as specified by requirement 3.



See Also

`after` | `hasChanged` | `send` | `this` | `coder.extrinsic` | Integrator

More About

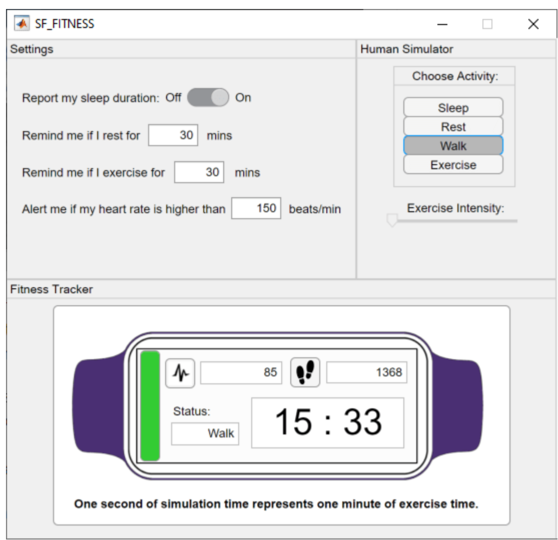
- “Simulate a Media Player” on page 21-14

- “Call Extrinsic MATLAB Functions in Stateflow Charts” on page 14-13
- “Broadcast Local Events to Synchronize Parallel States” on page 12-25
- “Control Chart Execution by Using Temporal Logic” on page 14-35
- “Develop Apps Using App Designer”

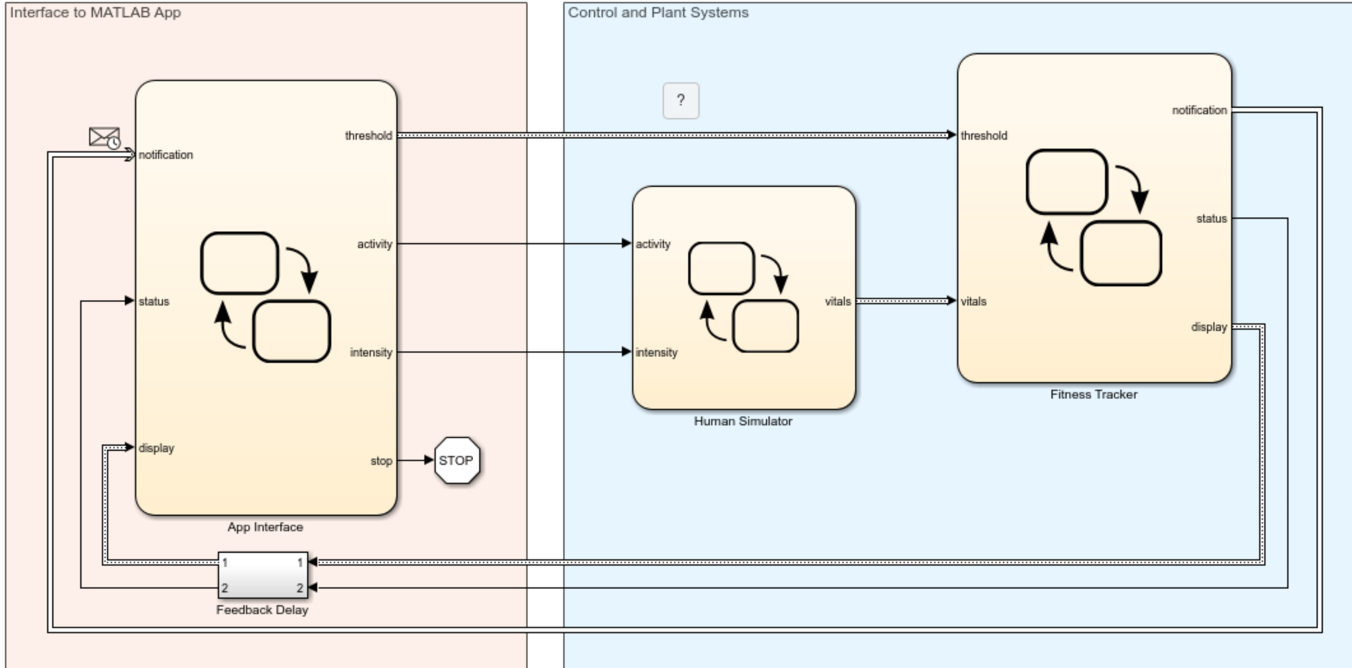
Model a Fitness Tracker

This example shows how to create an interface between a Stateflow® chart and a MATLAB® app created in App Designer. For other examples that show how to connect a Stateflow chart to a MATLAB app, see “Model a Power Window Controller” on page 27-51 and “Simulate a Media Player” on page 21-14. For a version of this example that uses standalone charts in MATLAB, see “Model a Fitness App by Using Standalone Charts” on page 31-35.

In this example, a MATLAB app models a fitness tracker. During simulation, you can adjust the settings for the tracker and select an activity (**Sleep**, **Rest**, **Walk**, or **Exercise**). When you choose **Exercise**, you can also set the intensity of your workout.



The Stateflow chart `App Interface` provides a bidirectional connection between the MATLAB app and the control and plant systems in the Simulink® model. When you interact with the widgets in the app, the chart communicates your selections to the other charts in the model. Conversely, the chart uses the output of the fitness tracker to update the numeric and text fields in the app. For example, when you click the **Rest** button on the app, the `App Interface` chart sets the value of the output `activity` to the enumerated value `Activity.Rest`. The `Human Simulator` chart responds by producing vital sign values that model a person at rest. The `Fitness Tracker` chart analyses these values and sets the output signal `status` to `Activity.Rest`. The `App Interface` chart monitors this signal and updates the contents of the **Status** field in the app to `Rest`.



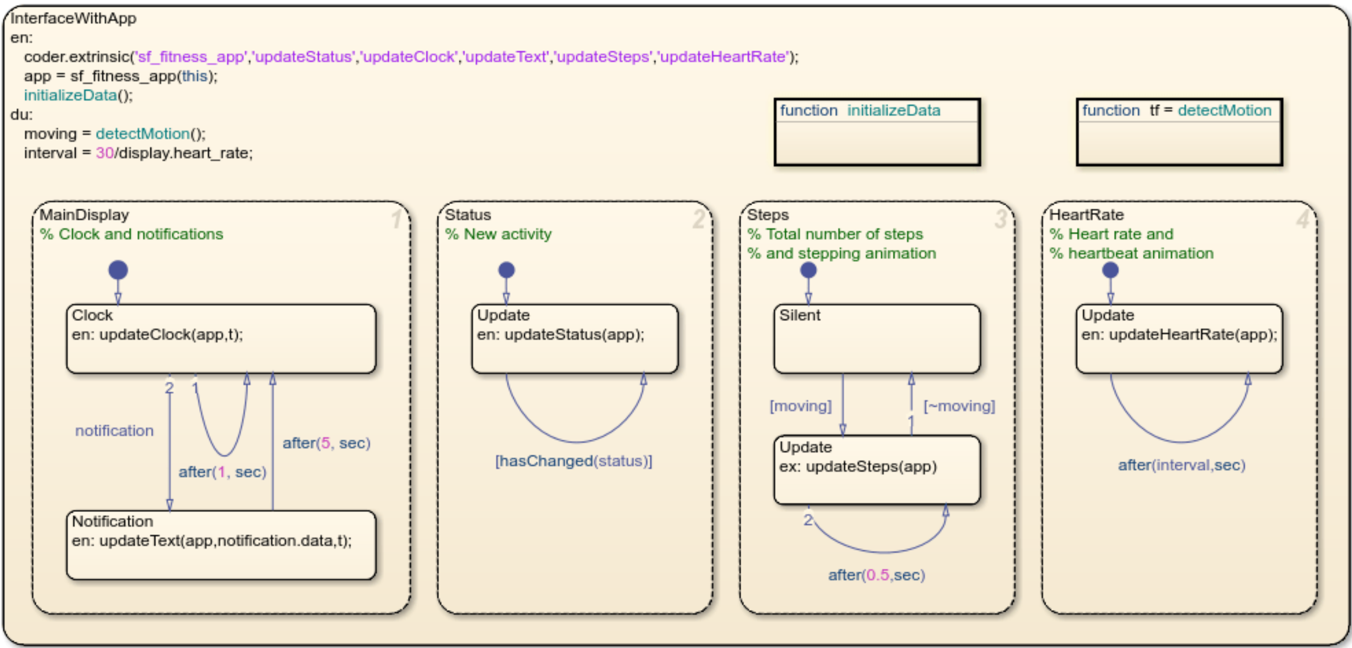
To run the example, open the Simulink model and click **Run**. The chart `App Interface` opens the app and initializes the `Human Simulator` and `Fitness Tracker` charts. While the example is running, one second of simulation represents one minute of exercise time. To stop the simulation, click **Stop** or close the app.

Connect Chart to MATLAB App

The chart `App Interface` is configured to communicate with the MATLAB app `sf_fitness_app`.

- The app uses a property called `chart` to interface with the chart `App Interface`. The app callbacks use this property to read the chart inputs and write to the chart outputs. For example, when you change the value of one of the fields in the **Settings** pane, a callback updates the value of the corresponding field of the output structure `threshold`. Similarly, when you select a new activity or change the intensity of your workout in the **Human Simulator** pane, a callback sets the value of the chart outputs `activity` and `intensity`. Finally, when you close the app, the `UIFigureCloseRequest` callback sets the value of the chart output `stop` to `true`.
- In the chart, the entry actions in the `InterfaceWithApp` state run the app `sf_fitness_app` and store the returned value as the local data object `app`. The chart uses this local data object when it calls the helper functions `updateStatus`, `updateClock`, `updateText`, `updateSteps`, and `updateHeartRate`. In the app, these helper functions change the contents of the activity status, clock, and step counter fields, and create the animation effects in the heartbeat and footstep displays. For example, when the chart receives a `notification` message, the substate `MainDisplay` calls the helper function `updateText`. This function replaces the contents of the clock display with a customized notification. After five seconds, the substate calls the helper function `updateClock` to restore the clock display.

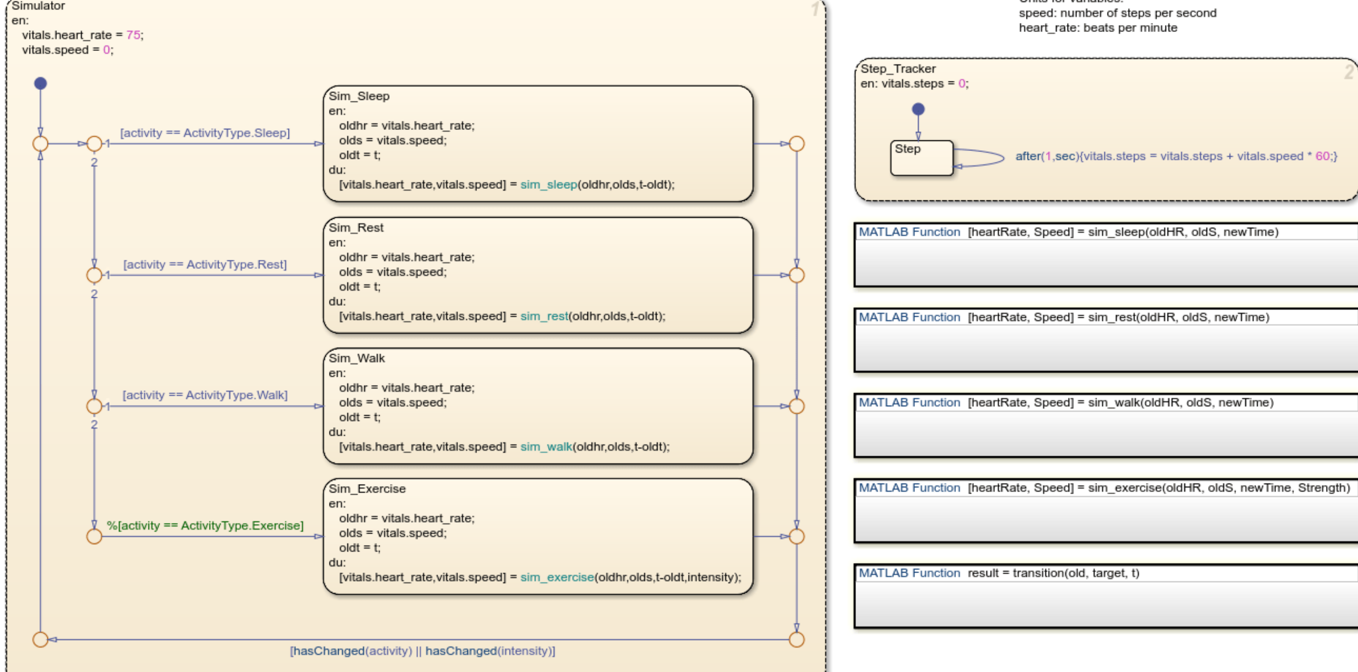
For more information on how to create a bidirectional connection between your MATLAB app and a Stateflow chart, see “Model a Power Window Controller” on page 27-51 and “Simulate a Media Player” on page 21-14.



The functions used to interact with the app are not supported for code generation, so the `InterfaceWithApp` state first calls the `coder.extrinsic` function to declare them as extrinsic MATLAB code. For more information, see “Call Extrinsic MATLAB Functions in Stateflow Charts” on page 14-13.

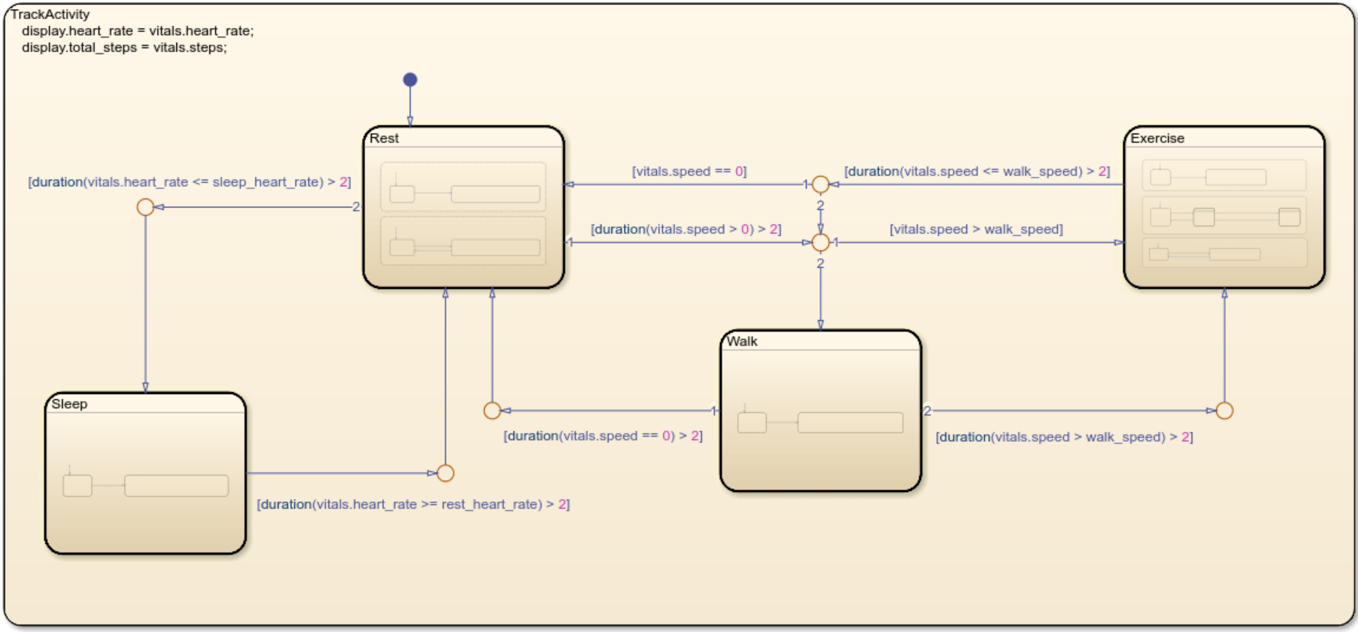
Simulate Vital Signs Based on Activity

The `Human Simulator` chart models the vital signs of a human engaged in the activity you select in the app. The chart uses the output structure `vitals` to relay these vital signs to the fitness tracker. The fields of the structure represent your heart rate, speed, and the number of steps that you have taken. When you select a new activity or adjust the intensity of your workout, the chart calls the function `transition` to ensure that these vital signs change gradually over time. To detect changes in activity or exercise intensity, the chart calls the `hasChanged` operator. For more information, see “Detect Changes in Data and Expression Values” on page 14-63.



Determine Fitness Tracker Output

The chart `Fitness Tracker` models the core logic of the fitness tracker. The chart consists of four subcharts that correspond to the possible activities. The chart registers your activity status based on the heart rate and speed produced by the `Human Simulator` chart and transitions between these subcharts. To filter out signal noise, the chart uses the `duration` operator to implement simple debouncing logic. For instance, when you are at rest, you can make some quick and sudden movements that do not correspond to exercise. The chart determines that you are walking or exercising only if your motion lasts longer than two minutes (or two seconds of simulation time). The chart monitors the active child state and passes this information to the `App Interface` chart through the output data `status`. For more information, see “Monitor State Activity Through Active State Data” on page 11-2.



The chart uses other temporal logic operators to track the amount of time you spend in each activity and determine when to send notifications to the app:

- The exit actions in each subchart call the `elapsed` operator to determine how long the subchart was active. The chart communicates this value, along with other information such as your heart rate and your total number of steps, to the `App Interface` chart through the output structure `display`.
- The chart uses the `after` operator to determine when you sleep or walk for longer than five minutes, rest or exercise for longer than the threshold you specify in the app, or exercise at a high intensity (taking more than 4 steps a second) for longer than 15 minutes. In each of these cases, the chart sends a `Notification` message. The `App Interface` chart receives this message and causes a notification to appear in the main display of the app. Depending on the type of notification, the notification button changes color.

See Also

Stop Simulation | `after` | `duration` | `elapsed` | `hasChanged` | `coder.extrinsic`

More About

- “Model a Fitness App by Using Standalone Charts” on page 31-35
- “Call Extrinsic MATLAB Functions in Stateflow Charts” on page 14-13
- “Control Chart Execution by Using Temporal Logic” on page 14-35
- “Communicate with Stateflow Charts by Sending Messages” on page 13-2
- “Monitor State Activity Through Active State Data” on page 11-2
- “Develop Apps Using App Designer”

Custom Code

- "Reuse Custom Code in Stateflow Charts" on page 28-2
- "Configure Custom Code in Library Models" on page 28-9
- "Access Custom Code Variables and Functions in Stateflow Charts" on page 28-12
- "Model Battery Management with Custom Code" on page 28-14
- "Access Custom C++ Code in Stateflow Charts" on page 28-21

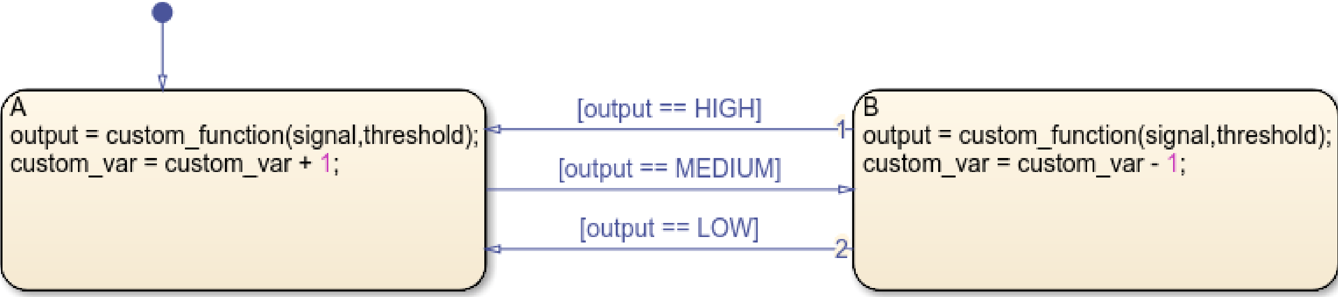
Reuse Custom Code in Stateflow Charts

You can integrate custom code written in C or C++ with Stateflow charts in Simulink models. By sharing data and functions between your custom code and your Stateflow chart, you can augment the capabilities of Stateflow and take advantage of your preexisting code.

Integrate Custom C Code in Stateflow Charts

This example shows how to use custom C code to define constants, variables, and functions that you can access in the charts in your model. For more information about integrating custom C++ code in your charts, see “Access Custom C++ Code in Stateflow Charts” on page 28-21.

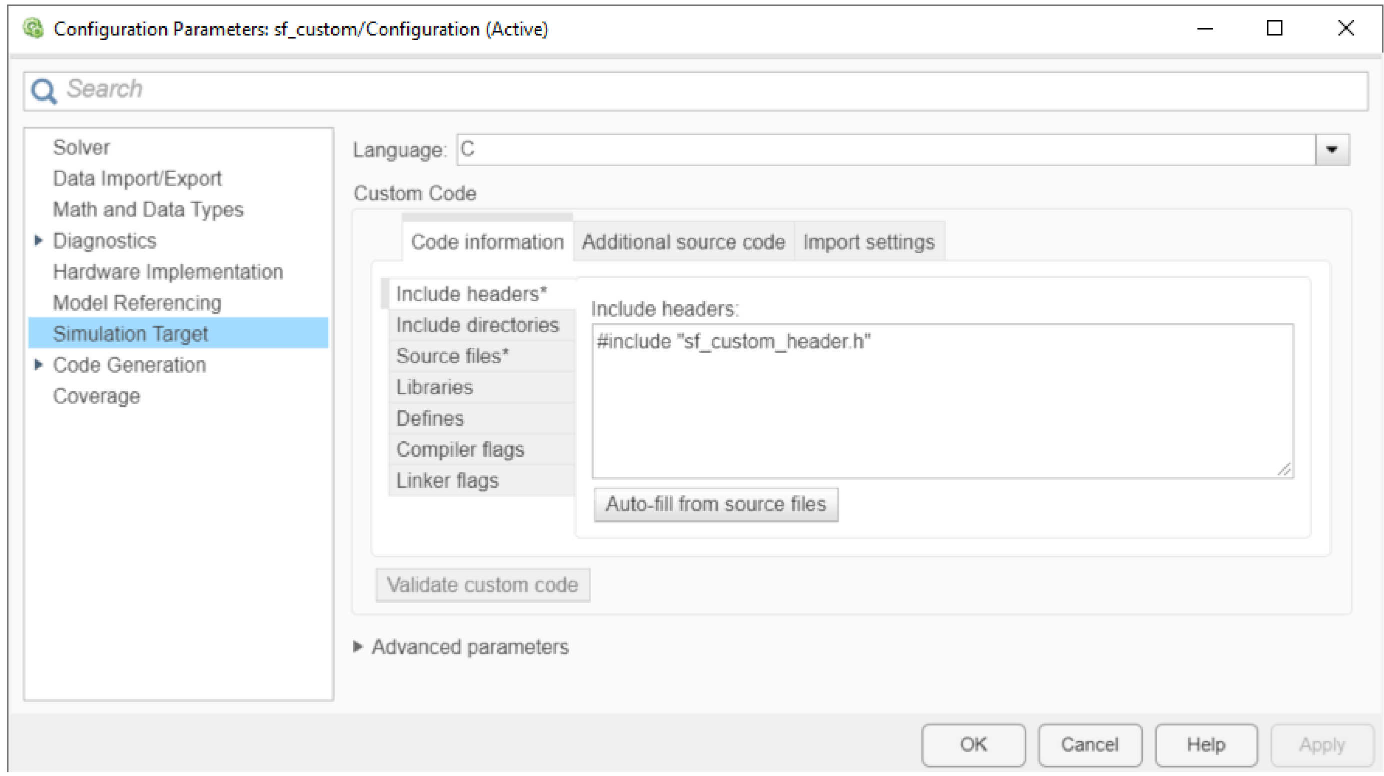
In this example, a Stateflow chart calls a custom code function named `custom_function`. This function reads the chart input `signal` and the local data `threshold`, compares their values, and returns one of three custom global constants named `HIGH`, `MEDIUM`, and `LOW`. The chart uses the value of the function to determine whether to transition to a new state after it increments or decrements the value of a custom variable named `custom_var`.



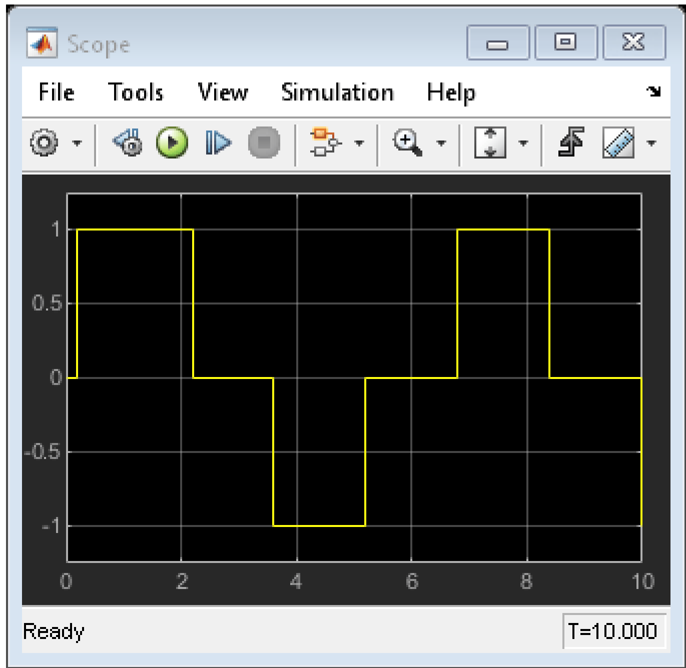
To see the custom code that this chart accesses, open the Configuration Parameters dialog box and, in the **Simulation Target** pane, select the **Code Information** tab.

- The **Include headers** parameter contains an `#include` statement that specifies the header file `sf_custom_header.h`. This file contains the definitions of three static global constants and the declarations for the variable `custom_var` and the function `custom_function`.
- The **Source files** parameter specifies the source file `sf_custom_source.c`. This file sets the initial value of `custom_var` to zero and defines the function `custom_function`.

Both of these files are located in the same folder that contains the model. To access custom code files in a different folder, use relative path names. For more information, see “Specify Relative Paths to Your Custom Code” on page 28-6.



When you simulate the model, Stateflow compiles the source file and the chart into a single S-function MEX file. Because the custom definitions appear at the top of the generated machine header file `sf_custom_sfun.h`, every chart in the model can access the custom code during simulation.



Configure Custom Code for Your Model

Specify Custom Code Settings for Simulation

To configure your model to access custom code during simulation, use the **Simulation Target** pane of the Configuration Parameters dialog box.

1. Open the Configuration Parameters dialog box.
2. In the **Simulation Target** pane, in the **Code Information** tab, specify these parameters:

   - **Include headers** — Enter the code to include at the top of the generated `*model*.h` header file, which declares custom functions and data in the generated code. The code appears at the top of all generated source code files and is visible to all generated code. For example, use this parameter to enter `#include` and `#define` statements. When you include a custom header file, you must enclose the file name in double quotes. For more information, see “Include headers” (Simulink).

   Note

   - Charts that use MATLAB as the action language do not support `#define` statements in custom code. To share constants between your chart and your custom code, use static global constants instead of macros.
   - The code you specify in this parameter can include `extern` declarations of variables or functions, such as `extern int x` or `extern void myfun(void)`. However, global variable or function definitions such as `int x` or `void myfun(void)` cause linking errors because they appear multiple times in the source files of the generated code.

- **Include directories** — Enter a space-separated list of the folder paths that contain custom header files that you include either directly in the **Header file** parameter or indirectly in the compiled target. For more information, see “Include directories” (Simulink).
- **Source files** — Enter a list of source files to compile and link into the target. You can separate source files with a comma, space, or new line. For more information, see “Source files” (Simulink).
- **Libraries** — Enter a space-separated list of static libraries that contain custom object code to link into the target. For more information, see “Libraries” (Simulink).
- **Defines** — Enter a space-separated list of preprocessor macro definitions to add to the generated code. For more information, see “Defines” (Simulink).
- **Compiler flags** — Enter additional compiler flags to be added to the compiler command line when your custom code is compiled. For more information, see “Compiler flags” (Simulink).
- **Linker flags** — Enter additional linker flags to be added to the linker command line when your custom code is linked. For more information, see “Linker flags” (Simulink).

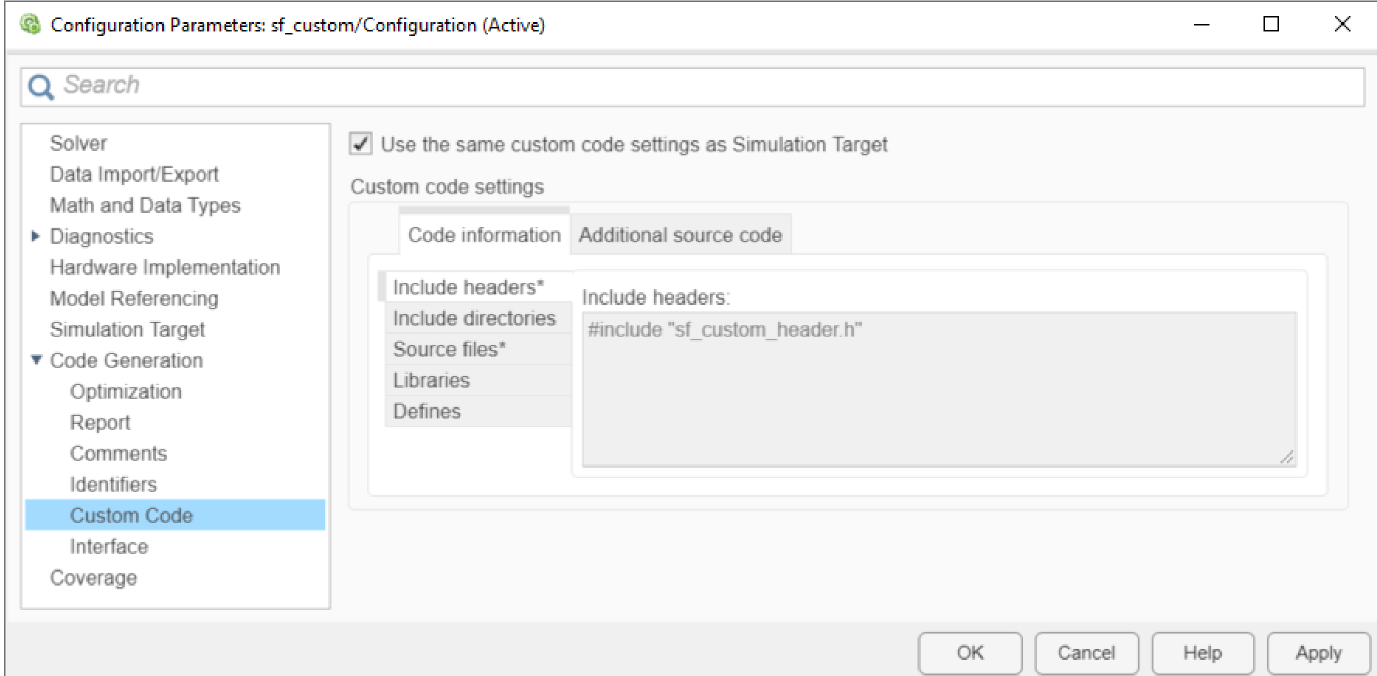
3. Under **Advanced parameters**, select **Import custom code** (Simulink).
4. If your model contains library charts, configure the custom code settings for each library model that contributes a chart to your model. For more information, see “Configure Custom Code in Library Models” on page 28-9.

For information on setting simulation options by using the command-line API, see “Set Configuration Parameters Programmatically” on page 29-18.

Specify Custom Code Settings for Code Generation

To configure your model to access custom code for code generation, use the **Code Generation > Custom Code** pane of the Configuration Parameters dialog box. When generating code, your model can use the same custom code settings that it uses for simulation or use unique custom code settings.

- To use the same custom code settings used for simulation, select **Use the same custom code settings as Simulation Target**. Specify the custom code settings in the **Simulation Target** pane as described in “Specify Custom Code Settings for Simulation” on page 28-4.
- To use unique custom code settings, clear **Use the same custom code settings as Simulation Target**. In the **Code Information** tab, specify custom code settings for code generation. For descriptions of the parameters in this tab, see “Specify Custom Code Settings for Simulation” on page 28-4.



For more information, see Use the same custom code settings as Simulation Target (Simulink Coder) and “Integrate External Code by Using Model Configuration Parameters” (Simulink Coder).

Call Custom Code Functions in States and Transitions

You can call custom code functions from the actions of any state or transition or from other functions.

To call a custom code function, use the signature specified by the function declaration in your header file. Include an actual argument value for each formal argument in the function signature:

```
return_val = function_name(arg1,arg2,...)
```

Note Do not share fixed-point data between your custom code and your Stateflow chart.

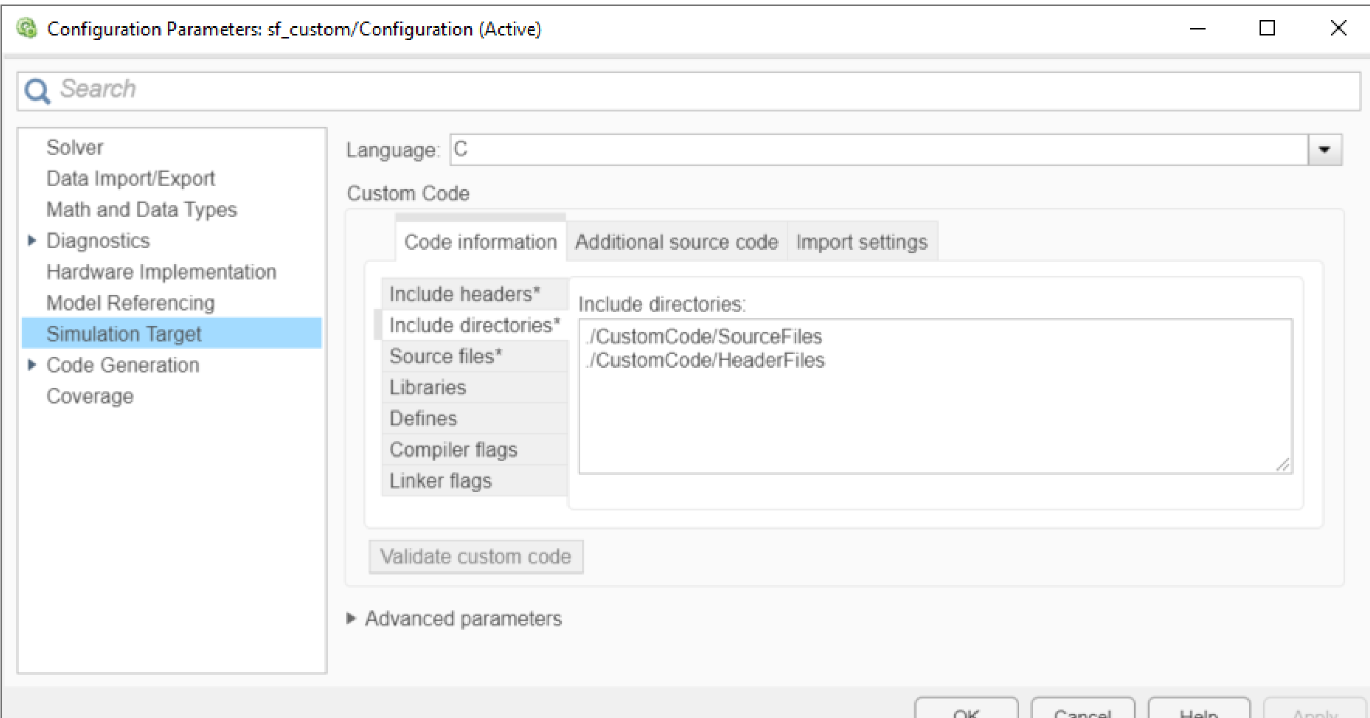
Specify Relative Paths to Your Custom Code

When you update your model or start the simulation, the model searches for the custom code files in these folders:

- The current folder
- The model folder (if this folder is 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

You can specify the location of your custom code by using paths relative to one of these folders. For instance, suppose that, in the previous example, you store the source and header files for your custom code in the subfolders `CustomCode/SourceFiles` and `CustomCode/HeaderFiles` of the

model folder. To access these files, use the **Include directories** parameter to specify the relative paths of the subfolders.



Alternatively, you can use relative path names to specify the header and source files individually:

- Under **Include headers**, enter:

```
#include "./CustomCode/HeaderFiles/sf_custom_code_constants_vars_fcns_hdr.h"
```

- Under **Source files**, enter:

```
./CustomCode/HeaderFiles/sf_custom_code_constants_vars_fcns_src.c
```

Guidelines for Relative Path Syntax

When you construct relative paths for custom code, follow these syntax guidelines:

- Use a single period (`.`) to indicate the starting point for the relative path.
- Use forward slashes (`/`) or backward slashes (`\`) as file separators, regardless of the current platform you are using.
- Enclose paths in double quotes (`"..."`) if they contain nonstandard path characters such as spaces or hyphens (`-`).
- Enclose expressions with dollar signs (`$...$`) to evaluate them in the MATLAB workspace. For example, suppose that `CustomCodeFolder` is a variable that you define in the MATLAB workspace as `"module1"`. If you specify your custom code files using the path name `.\work\source\$CustomCodeFolder$`, then the model searches for the custom code files in the folder `.\work\source\module1`.

See Also

More About

- “Model Battery Management with Custom Code” on page 28-14
- “Share String Data with Custom C Code” on page 21-10
- “Configure Custom Code in Library Models” on page 28-9
- “Integrate Custom Structures in Stateflow Charts” on page 26-11
- “Access Custom C++ Code in Stateflow Charts” on page 28-21

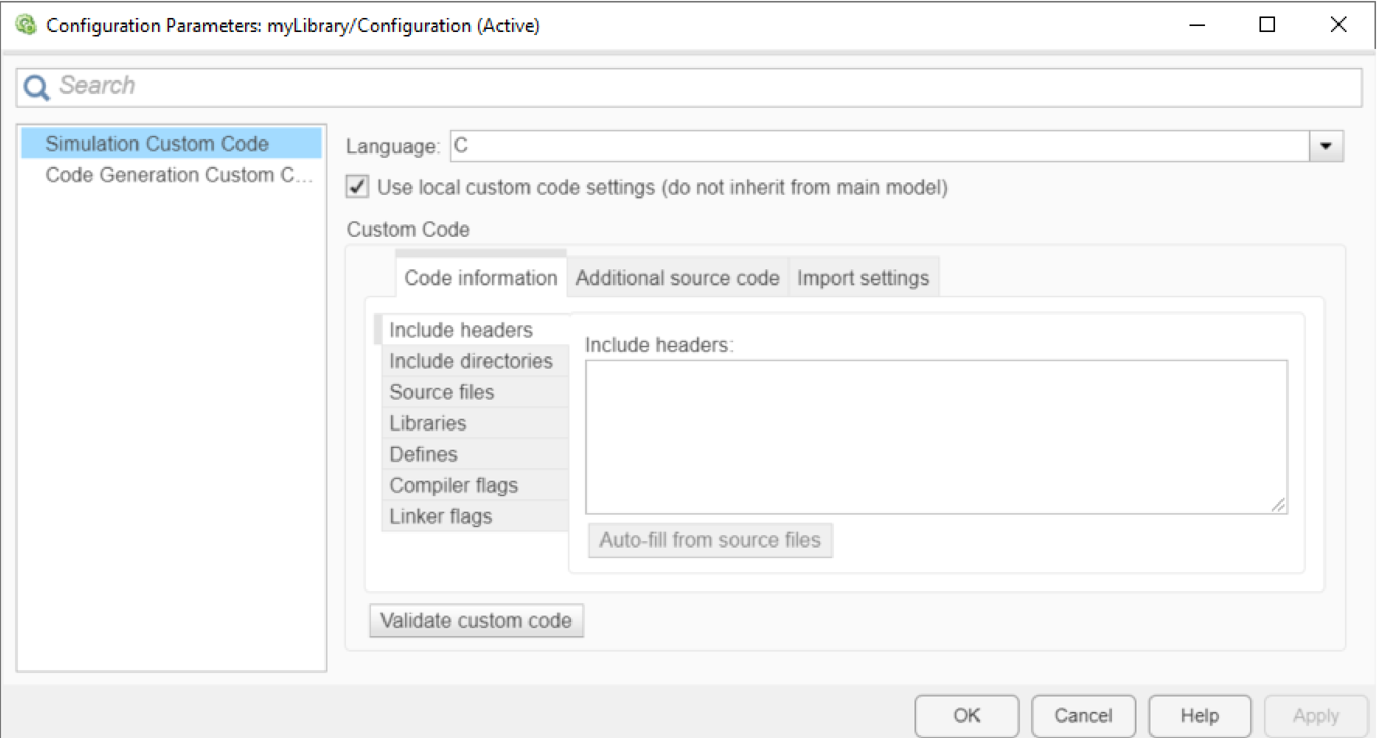
Configure Custom Code in Library Models

You can integrate custom code written in C or C++ with Stateflow charts in Simulink models. By sharing data and functions between your custom code and your Stateflow chart, you can augment the capabilities of Stateflow and take advantage of your preexisting code. For more information, see “Reuse Custom Code in Stateflow Charts” on page 28-2.

Configure Custom Code Settings for Simulation

To configure your library model to access custom code during simulation, use the **Simulation Custom Code** pane of the Configuration Parameters dialog box.

1 In the **Modeling** tab, under **Design**, select **Simulation Custom Code**.



2 A library model can inherit the custom code settings from the main model or use local custom code settings for simulation.

- To inherit the custom code settings from the main model, clear **Use local custom code settings (do not inherit from main model)**.
- To use local custom code settings for simulation, select **Use local custom code settings (do not inherit from main model)**.

For more information, see “Use local custom code settings (do not inherit from main model)” (Simulink).

3 To add custom code settings for simulation that are unique to your library model, in the **Code information** tab, specify these parameters:

- **Include headers** — Enter the code to include at the top of the generated `model.h` header file, which declares custom functions and data in the generated code. The code appears at the top of all generated source code files and is visible to all generated code. For example, use this parameter to enter `#include` and `#define` statements. When you include a custom header file, you must enclose the file name in double quotes. For more information, see "Include headers" (Simulink).

  Note

  - Charts that use MATLAB as the action language do not support `#define` statements in custom code. To share constants between your chart and your custom code, use static global constants instead of macros.
  - The code you specify in this parameter can include `extern` declarations of variables or functions, such as `extern int x` or `extern void myfun(void)`. However, global variable or function definitions such as `int x` or `void myfun(void)` cause linking errors because they appear multiple times in the source files of the generated code.

- **Include directories** — Enter a space-separated list of the folder paths that contain custom header files that you include either directly in the **Header file** parameter or indirectly in the compiled target. For more information, see "Include directories" (Simulink).
- **Source files** — Enter a list of source files to compile and link into the target. You can separate source files with a comma, space, or new line. For more information, see "Source files" (Simulink).
- **Libraries** — Enter a space-separated list of static libraries that contain custom object code to link into the target. For more information, see "Libraries" (Simulink).
- **Defines** — Enter a space-separated list of preprocessor macro definitions to add to the generated code. For more information, see "Defines" (Simulink).
- **Compiler flags** — Enter additional compiler flags to be added to the compiler command line when your custom code is compiled. For more information, see "Compiler flags" (Simulink).
- **Linker flags** — Enter additional linker flags to be added to the linker command line when your custom code is linked. For more information, see "Linker flags" (Simulink).

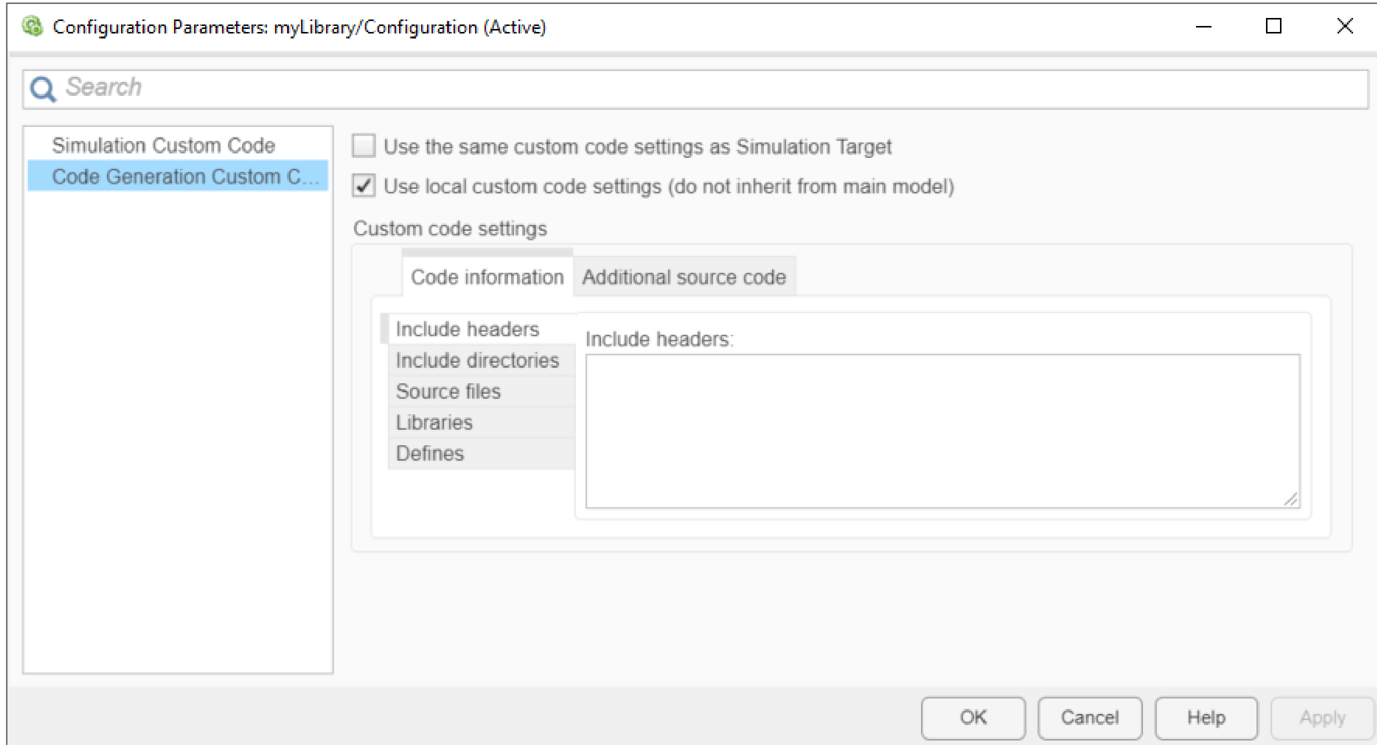
These settings apply only when you select **Use local custom code settings (do not inherit from main model)**.

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.

Configure Custom Code Settings for Code Generation

To configure your library model to access custom code during code generation, use the **Code Generation Custom Code** pane of the Configuration Parameters dialog box.

1. In the **Modeling** tab, under **Design**, select **Simulation Custom Code**.
2. Open the **Code Generation Custom Code** pane.



3 When generating code, a library model can use the same custom code settings that it uses for simulation or use unique custom code settings.

- To use the same custom code settings used for simulation, select **Use the same custom code settings as Simulation Target**. Specify the custom code settings in the **Simulation Custom Code** pane as described in "Configure Custom Code Settings for Simulation" on page 28-9.
- To use unique custom code settings, clear **Use the same custom code settings as Simulation Target**. In the **Code Information** tab, specify custom code settings for code generation. For descriptions of the parameters in this tab, see "Configure Custom Code Settings for Simulation" on page 28-9.

For more information, see Use the same custom code settings as Simulation Target (Simulink Coder) and "Integrate External Code by Using Model Configuration Parameters" (Simulink Coder).

See Also

More About

- "Reuse Custom Code in Stateflow Charts" on page 28-2
- "Model Battery Management with Custom Code" on page 28-14
- "Share String Data with Custom C Code" on page 21-10
- "Integrate Custom Structures in Stateflow Charts" on page 26-11

Access Custom Code Variables and Functions in Stateflow Charts

You can integrate custom code written in C or C++ with Stateflow charts in Simulink models. By sharing data and functions between your custom code and your Stateflow chart, you can augment the capabilities of Stateflow and take advantage of your preexisting code. For more information, see “Reuse Custom Code in Stateflow Charts” on page 28-2.

Custom Code Variables in Charts That Use MATLAB as the Action Language

You can read and write the following C code variables directly in your charts that use MATLAB as the action language.

Custom C Code Type	Description
`double`	Double-precision floating point
`single`	Single-precision floating point
`int8`	Signed 8-bit integer
`uint8`	Unsigned 8-bit integer
`int16`	Signed 16-bit integer
`uint16`	Unsigned 16-bit integer
`int32`	Signed 32-bit integer
`uint32`	Unsigned 32-bit integer

By right clicking on the Stateflow object that uses your custom code, you can access your custom code variable. After right clicking on the object, hover over **Explore**. Your custom code variable appears, denoted by `(C variable)`. Clicking the C variable allows you to access the custom code from MATLAB.

Custom Code Functions in Charts That Use MATLAB as the Action Language

You can use the following C function argument types directly in your charts that use MATLAB as the action language without using `coder.ceval`. For information on calling external code from MATLAB code by using `coder.ceval`, see “Call Custom C/C++ Code from the Generated Code” (MATLAB Coder).

Custom C Function Argument Type	Description
`double`	Double-precision floating point
`single`	Single-precision floating point
`int8`	Signed 8-bit integer
`uint8`	Unsigned 8-bit integer
`int16`	Signed 16-bit integer
`uint16`	Unsigned 16-bit integer

Custom C Function Argument Type	Description
`int32`	Signed 32-bit integer
`uint32`	Unsigned 32-bit integer

By right clicking on the Stateflow object that uses your custom code, you can access your custom code function. After right clicking on the object, hover over **Explore**. Your custom code function appears, denoted by `(C function)`. Clicking the C function allows you to access the custom code from MATLAB.

Accessing Enumerations in Custom Code

To include enumerations from your custom code in charts that use C as the action language:

1. Define your enumerations in a header file.
2. Open the Configuration Parameters dialog box.
3. In the **Simulation Target** pane, under **Advanced parameters**, select **Import custom code**.
4. In the **Code information** tab, include the header file that defines your enumerations.

See Also

More About

- "Reuse Custom Code in Stateflow Charts" on page 28-2
- "Model Battery Management with Custom Code" on page 28-14
- "Share String Data with Custom C Code" on page 21-10
- "Integrate Custom Structures in Stateflow Charts" on page 26-11

Model Battery Management with Custom Code

This example shows how to use custom C code with Stateflow® to model a system that manages battery percentage, also known as the state of charge (SOC).

With Stateflow you can integrate your custom C code into charts. Using custom C code in a Stateflow chart allows you to:

- Reuse existing algorithms that you have already coded.
- Use C code for low-level hardware operations, which may be difficult to implement with Stateflow.

Battery Management

This model represents several components of a battery management system. This system is designed to be implemented on a controller for battery powered devices, such as battery powered vehicle or a cell phone. The purpose of the battery management system is to limit the power demands on the battery and to ensure that the SOC does not get too high or too low. An SOC that is too high or too low would be detrimental to the health of the battery. Additionally, the model is designed to limit the discharge of the battery when the charge is low in a trade-off of performance for battery lifetime.

The battery management model achieves these goals with three different charts.

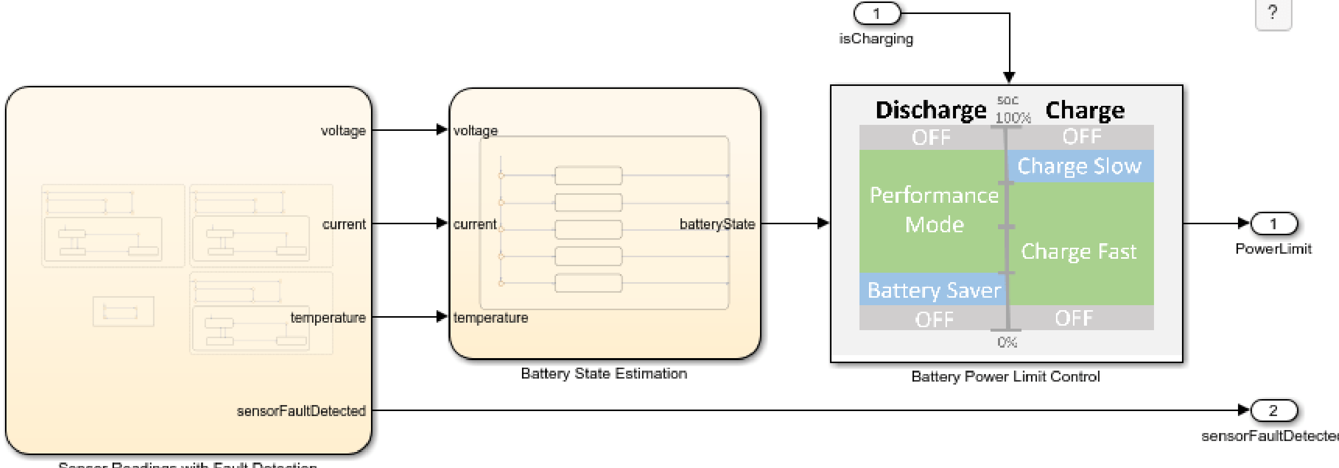
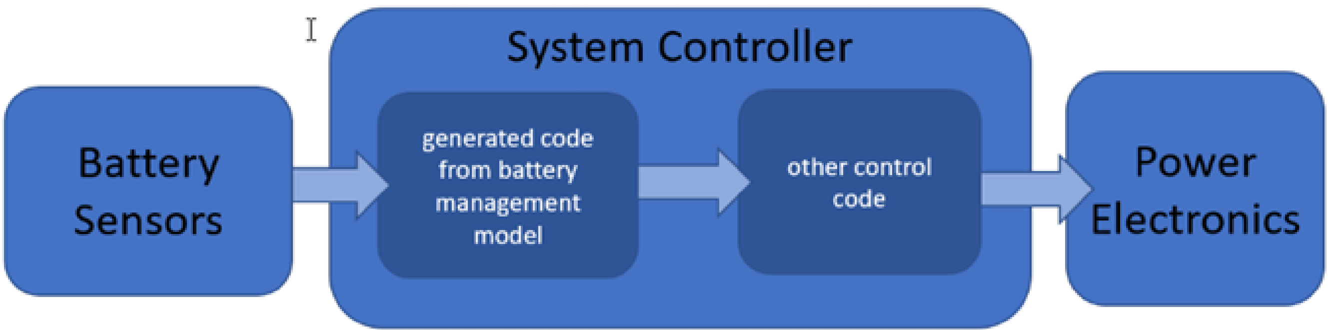


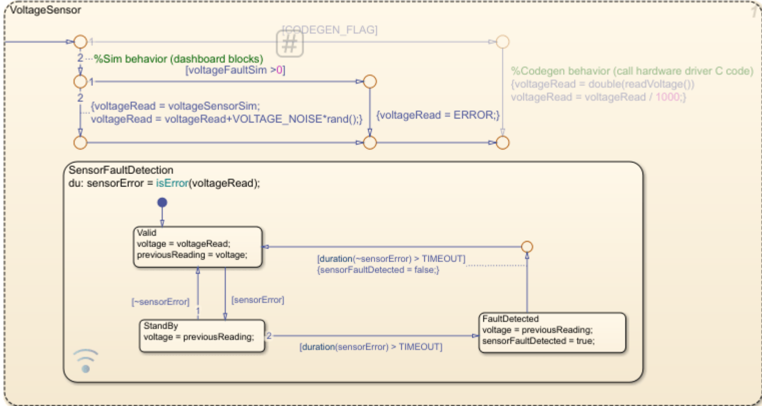
Chart `Sensor Readings with Fault Detection` reads the sensor values from the battery pack and reports out when the sensors is in a faulted state. Chart `Battery State Estimation` uses the sensor reading to estimate the SOC of the battery. Chart `Battery Power Limit Control` conserves the battery, protects the battery health, and keeps the SOC away from either extreme. The chart accomplishes these tasks by setting power limits for the controller.

With this model you can generate code and deploy that code to an embedded controller along with other control code that your system may need.



Simulate Communication with Hardware

The chart `Sensor Readings with Fault Detection` consists of three parallel states (`VoltageSensor`, `CurrentSensor`, and `TemperatureSensor`) that model the readings of the battery voltage, current, and temperature sensors. The three parallel states contain similar decision logic for choosing between simulation and code generation behavior. For example, when the parameter `CODEGEN_FLAG` is `false`, the `VoltageSensor` contains this logic for simulating the voltage readings.



When using the model for simulation, the Dashboard panel allows you to control the sensor readings for the system inputs. If the calls to the battery monitor timeout, an error code of `-9999` is returned from the function.

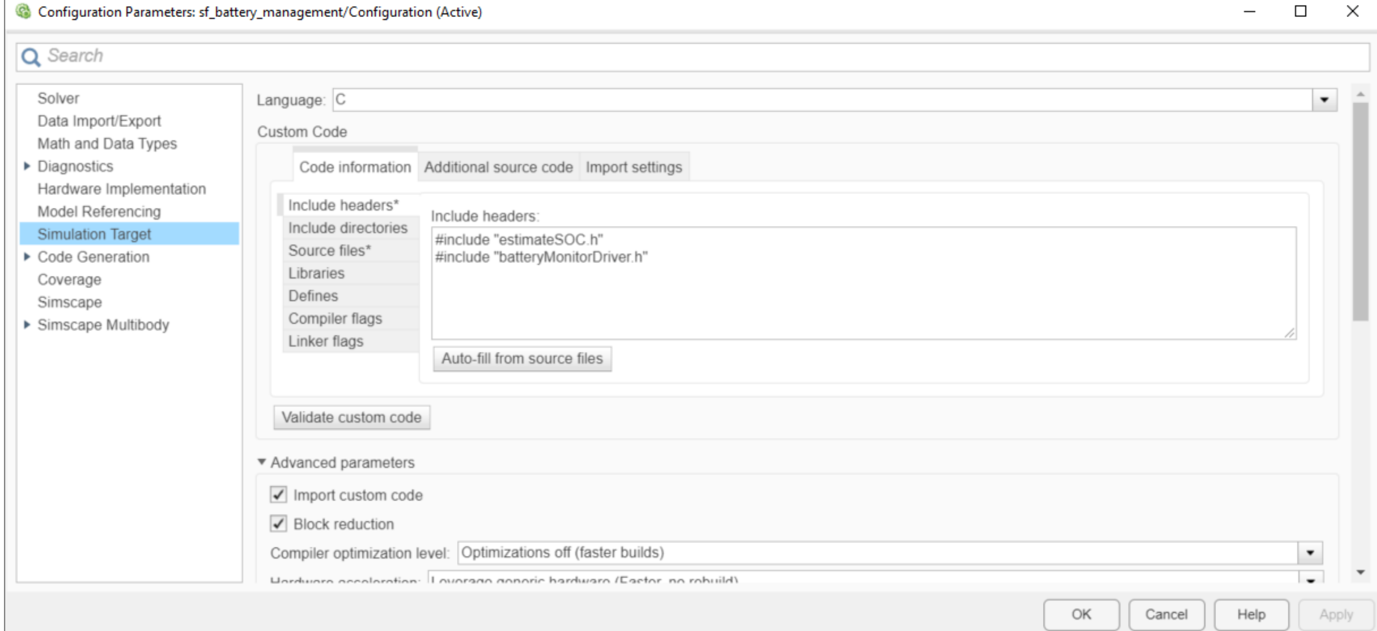
In each parallel state, the substate `SensorFaultDetection` handles the error signals returned by the sensors. In the event of a sensor error, `SensorFaultDetection` holds the last known valid sensor reading until the error code has been received for a certain amount of time. After this threshold is met, `SensorFaultDetection` sends a fault message and assumes it will be handled by the other control components of the controller.

The example includes two custom C code files: `batteryMonitorDriver.h` and `batteryMonitorDriver.c`. These files represent the device driver code that would be used to get sensor data from the system, including battery voltage, current, and temperature and are used for code generation. For more information, see Code Generation.

To simulate the model with the driver code:

1. Open the Configuration Parameters dialog box.
2. In the **Simulation Target** pane, specify the header file and source file.
3. Under **Advanced parameters**, select **Import custom code**.

For more information, see “Configure Custom Code for Your Model” on page 28-4.



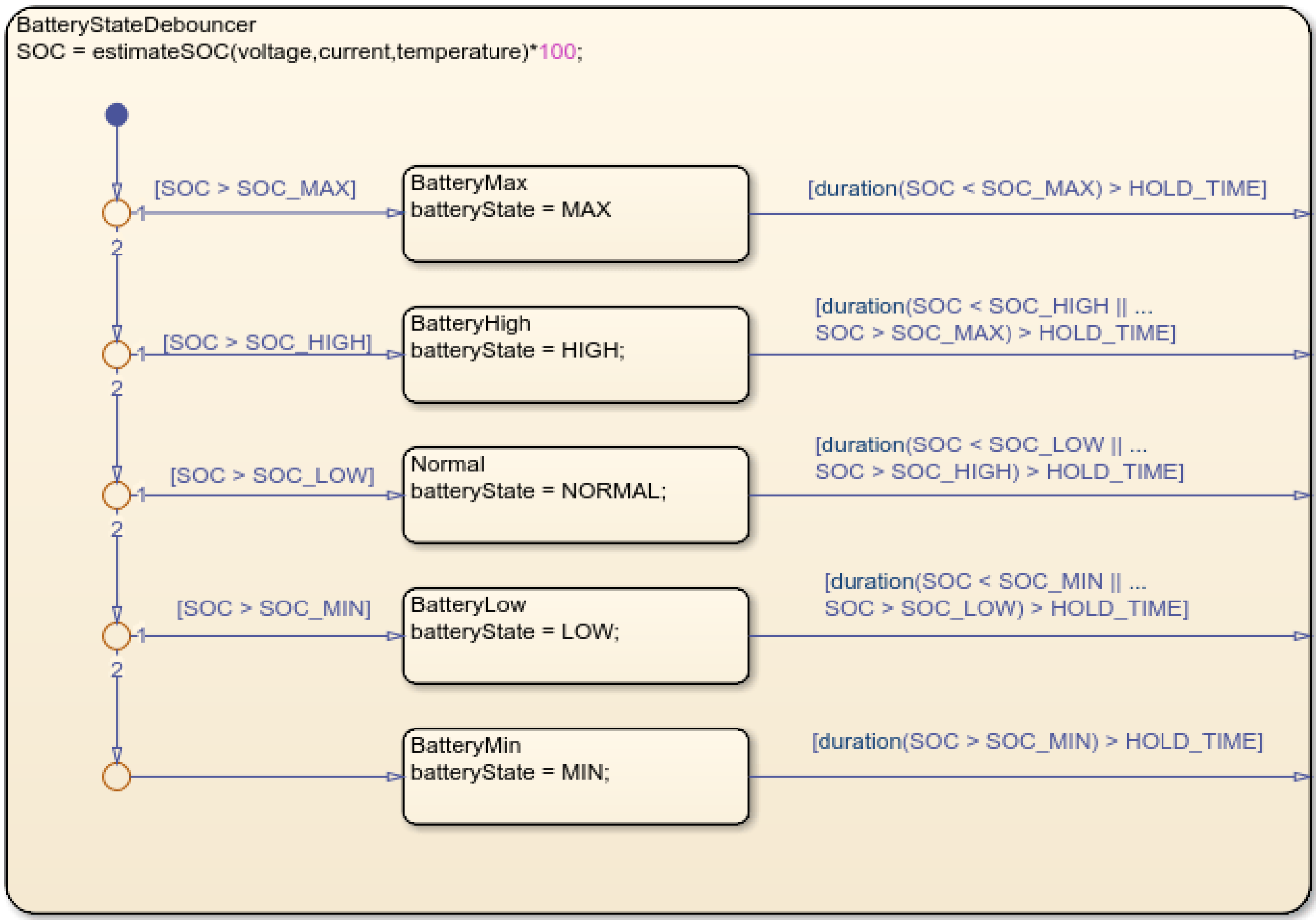
Estimate Battery State of Charge by Reusing Custom Code

To estimate the battery state of charge, the model utilizes a custom C code algorithm. The included file `estimateSOC.c` contains this code:

```
double estimateSOC(double V, double I, double T)
{
    double SOC;
    double r_int;
    double V_emf;

    r_int = batteryResistance(T); //internal resistance

    V_emf = voltage_EMF(V,I,r_int); //volage emf of battery

    SOC = calculateSOC(V,T);

    return SOC;
}
```

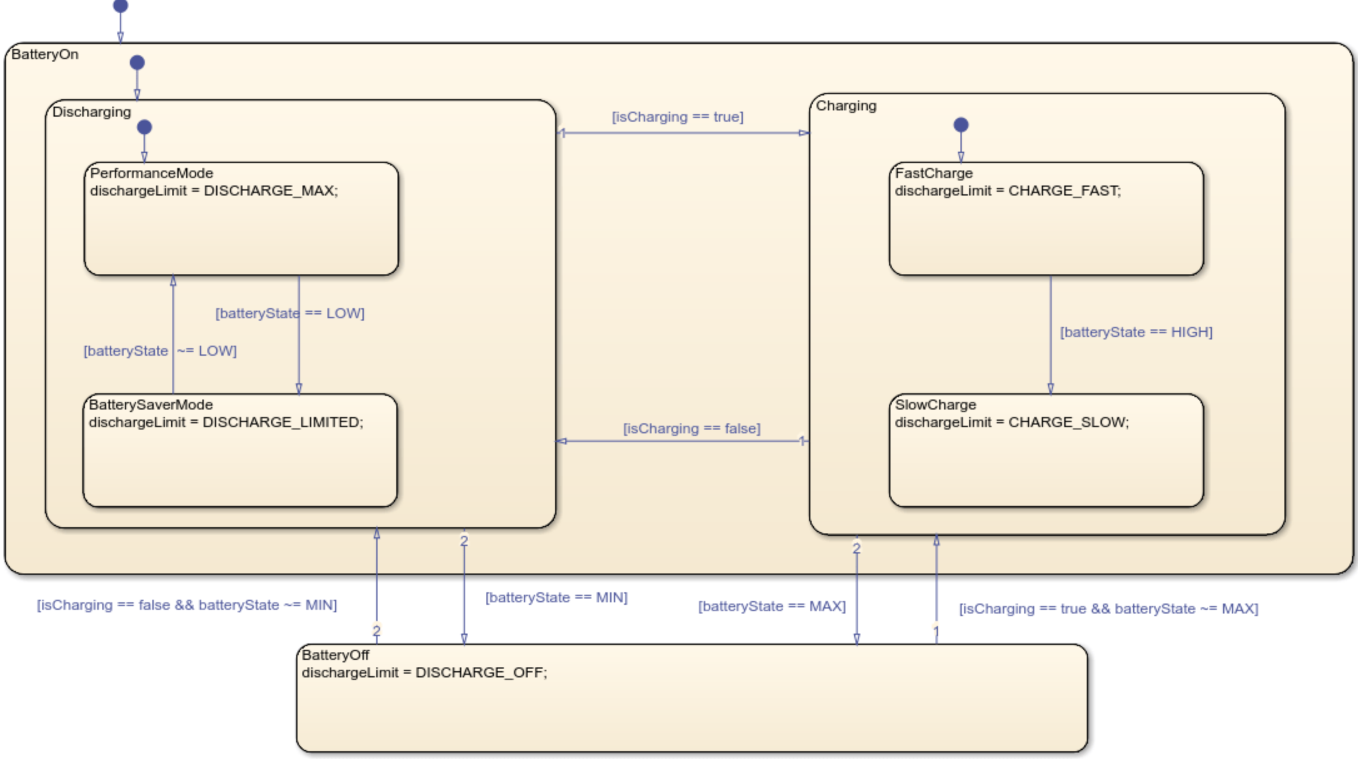
With this algorithm, you can easily call the C code function, rather than reimplementing it with Stateflow charts.

In order to account for the sensitivity of noise and change of current in the `estimateSOC` algorithm, Stateflow logic is used to implement a debouncing algorithm. This logic simplifies the SOC percentage into 5 ranges: `MAX`, `HIGH`, `NORMAL`, `LOW`, and `MIN`. These ranges prevent rapid fluctuation between different control states. The exit transitions from the child states go to the edge of the parent state. When these transitions are taken, Stateflow returns to the default transition of the parent state.



Logic to Control Device State of Charge

It is easier to design this control logic with a Stateflow chart, rather than implementing the logic control through custom code. This chart implements power limit on the battery based on the estimated battery state.

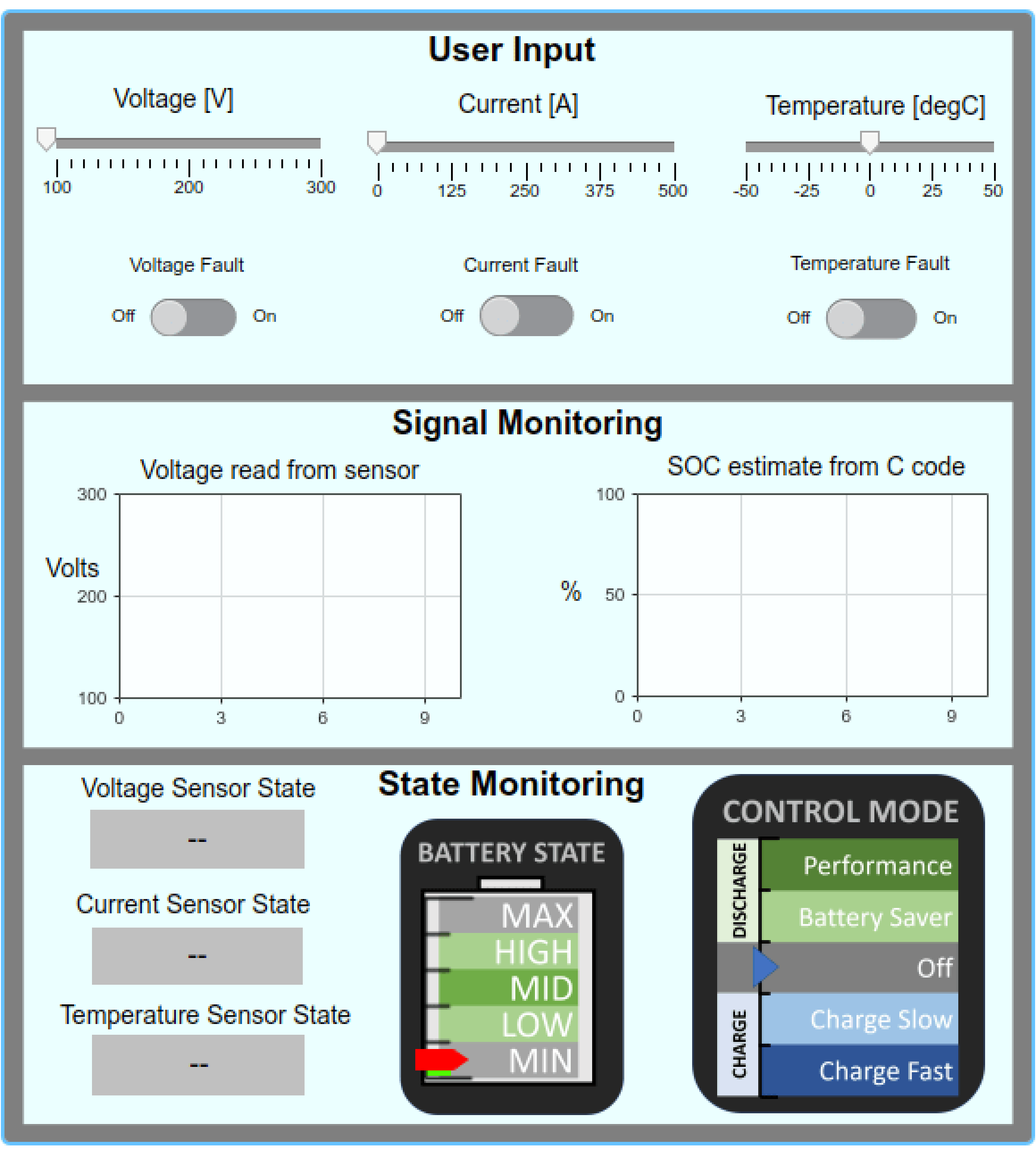


The chart represents five possible modes for power limits on the battery.

1. Performance Mode: Allow high power draw when battery charge is high.
2. Battery Saver Mode: Limit power draw on the battery for efficiency when charge is low.
3. Off: Do not allow Power Draw when battery is at state of charge limits.
4. Fast Charge: Quickly charge the battery when charge is low.
5. Slow Charge: Slowly charge the battery when charge is high for battery health benefits.

Simulate Using the Dashboard Panel

To test that the model behaves as expected, you can use the dashboard panel to simulate the voltage, current, and temperature readings. The switches allow you to simulate a sensor error to test the fault detection logic. The gauge and plot dashboard blocks are bound to the activity of stateflow charts to visualize internal states and data. You can move and minimize the dashboard panel while navigating the model. For more information on dashboard blocks, see “Control Simulations with Interactive Displays” (Simulink).



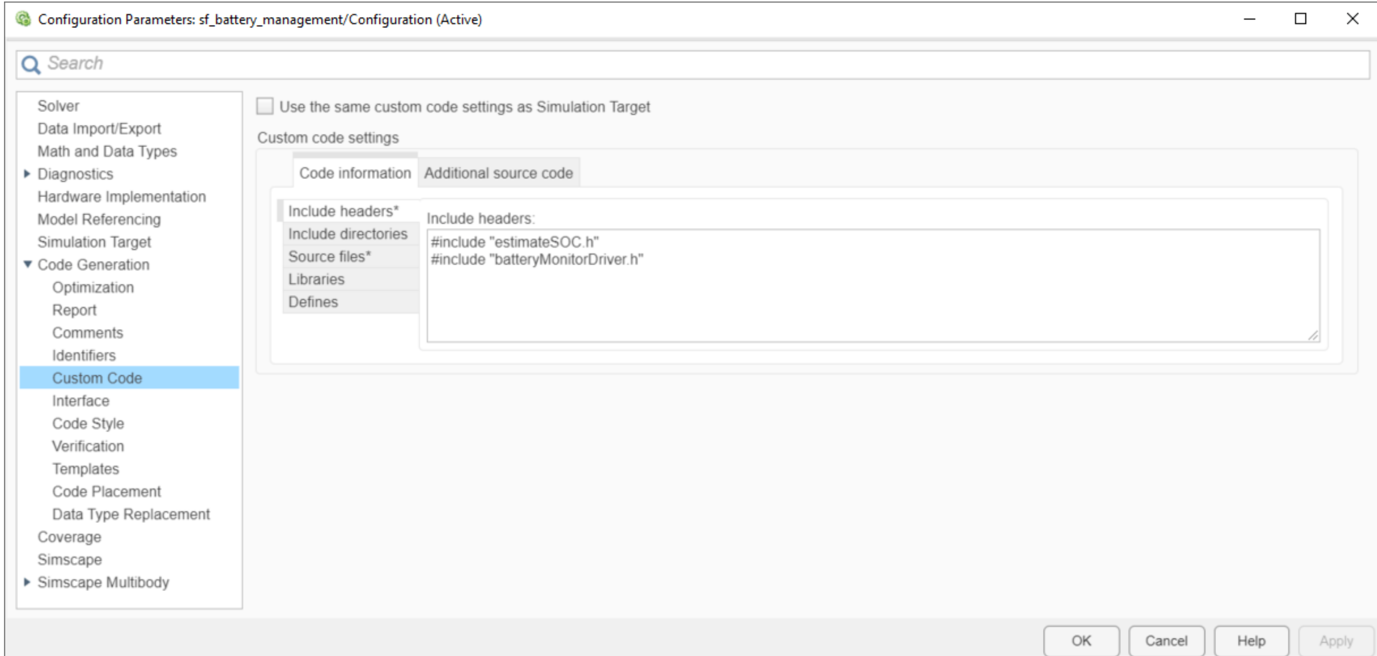
Code Generation

Inputs into the chart `Sensor Readings with Fault Detection` are provided with two C code files: `batteryMonitorDriver.h` and `batteryMonitorDriver.c`. These two files represent the device driver code that would be used to get sensor data from the system, including battery voltage, current, and temperature.

To use this model for code generation, the driver code must communicate with the external hardware. To enable this functionality, a variant transition using the control variable `CODEGEN_FLAG` allows the Stateflow chart to call the C code directly when generating code and simulate the sensor value with noise. In the Model Explorer, open the Base Workspace and set the value of `CODEGEN_FLAG` to `true`.

For more information on Stateflow Variants and variant transitions, see “Control Indicator Lamp Dimmer Using Variant Conditions” on page 29-9.

To compile the generated code with the driver code, open the Configuration Parameters dialog box and, in the **Code Generation > Custom Code** pane, specify the header file and source file. For more information, see “Configure Custom Code for Your Model” on page 28-4.



References

[1] Ramadass, P., B. Haran, R. E. White, and B. N. Popov. “Mathematical modeling of the capacity fade of Li-ion cells.” *Journal of Power Sources*. 123 (2003), pp. 230–240.

[2] Ning, G., B. Haran, and B. N. Popov. “Capacity fade study of lithium-ion batteries cycled at high discharge rates.” *Journal of Power Sources*. 117 (2003), pp. 160–169.

See Also

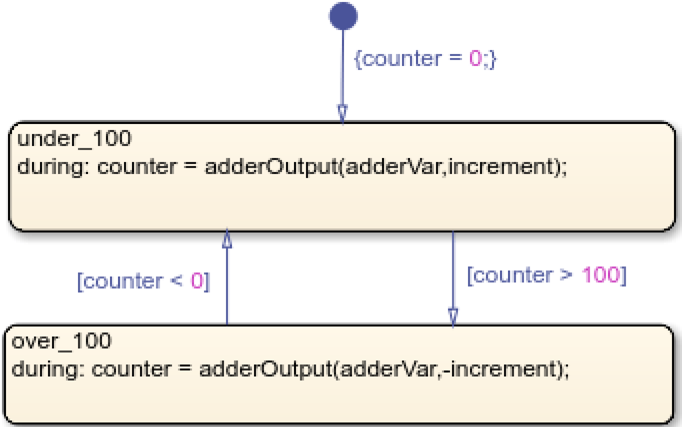
More About

- “Reuse Custom Code in Stateflow Charts” on page 28-2
- “Control Indicator Lamp Dimmer Using Variant Conditions” on page 29-9
- “Code Generation Using Simulink Coder” (Simulink Coder)
- “Code Generation Workflows with Embedded Coder” (Embedded Coder)
- “Tune and Experiment with Block Parameter Values” (Simulink)

Access Custom C++ Code in Stateflow Charts

This example shows how to integrate custom C++ code with Stateflow® charts in Simulink® models. By sharing data and functions between your custom code and your Stateflow charts, you can augment the capabilities of Stateflow and take advantage of your preexisting code. For more information, see "Reuse Custom Code in Stateflow Charts" on page 28-2.

In this example, a chart that uses C as the action language calls a custom code function named `adderOutput`. This function increments the value of the global custom variable `adderVar` by a specified amount. The chart stores the output of this function as the chart output `counter`. When `counter` is less than or equal to 100, the chart calls the custom function to increase the global variable by the chart input `increment`. When `counter` is greater than 100, the chart calls the custom function to decrease the global variable by the chart input `increment`.



Prepare Custom Code Files

Add a C function wrapper to execute each method in your C++ code. For instance, in this example, the C++ source file `adder_cpp.cpp` defines a class called `adder` that has two methods, `add_one` and `get_val`.

```
adder::adder()
{
    int_state = 0;
}

int adder::add_one(int increment)
{
       int_state += increment;
    return int_state;
}

int adder::get_val()
{
    return int_state;
}
```

To call these methods, the Stateflow chart uses the C function wrapper `adderOutput`.

```
double adderOutput(adder *obj, int increment)
{
       obj->add_one(increment);
       return obj->get_val();
}
```

The header file `adder_cpp.h` contains a prototype of this C function wrapper:

```
extern double adderOutput(adder *obj, int increment);
```

Choose a C++ Compiler

To view or change the default compiler, in the MATLAB® Command Window, enter:

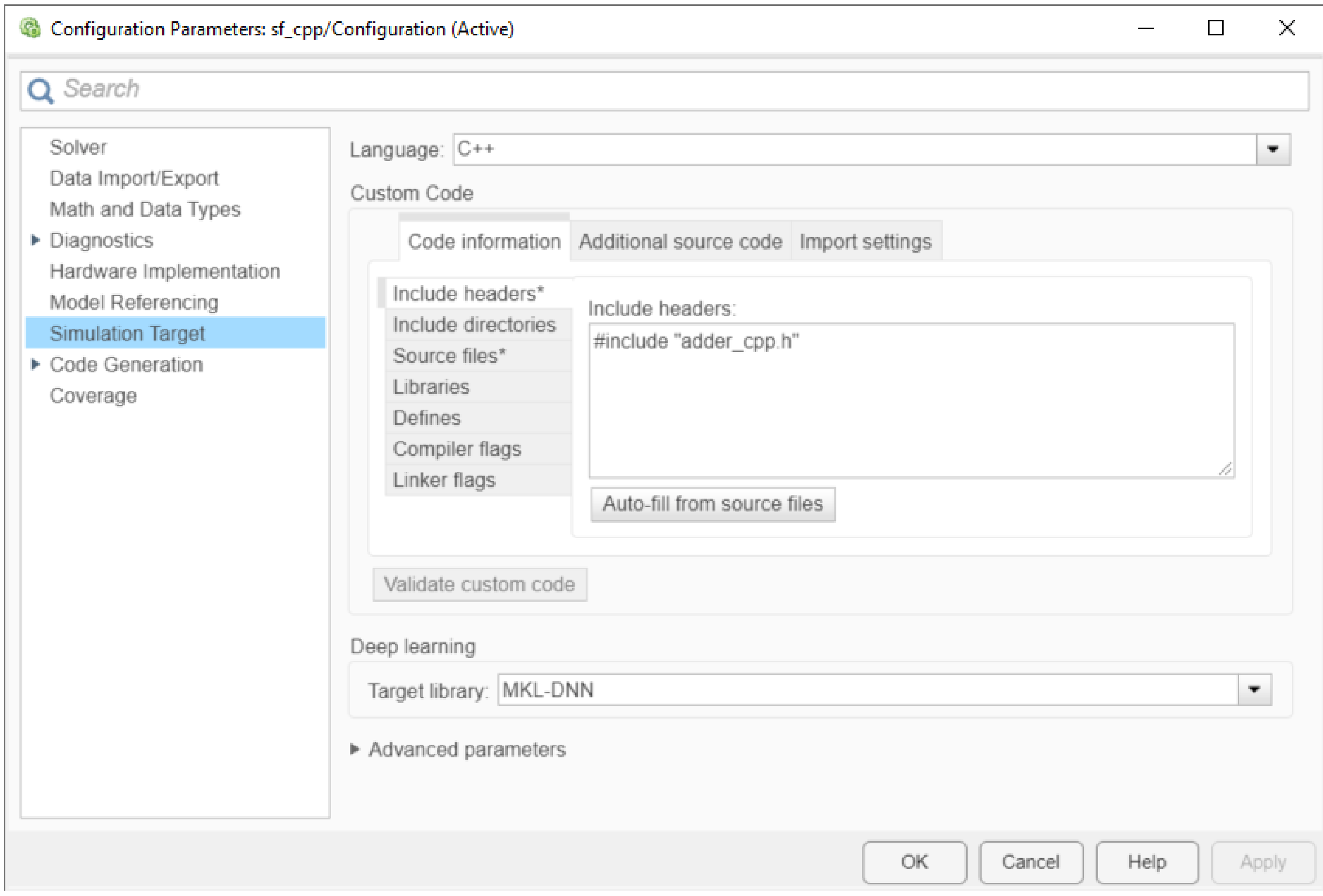
```
mex -setup c++
```

For more information, see "Choose a C++ Compiler". For a list of supported compilers, see Supported and Compatible Compilers.

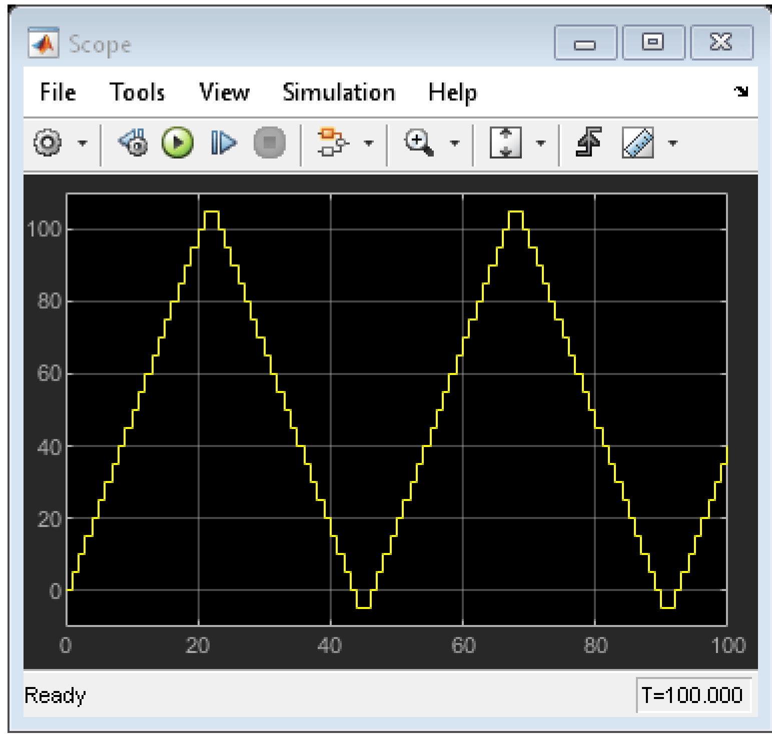
Include Custom Code Files for Simulation

Configure your simulation target and select C++ as the custom code language:

1. Open the Configuration Parameters dialog box.
2. In the **Simulation Target** pane, set the **Language** parameter to `C++`.
3. In the **Code Information** tab, specify your header and source files, as described in "Configure Custom Code for Your Model" on page 28-4.



When you simulate the model, the Scope block shows the value of the chart output `counter` increase or decrease by the value of the chart input `increment`.



C++ Code Generation

Select C++ as the code generation language and configure your model to use the same custom code settings specified for the simulation target:

1. Open the Configuration Parameters dialog box.
2. In the **Code Generation** pane, set the **Language** parameter to C++.
3. In the **Code Generation > Custom Code** pane, select **Use the same custom code settings as Simulation Target**.
4. Generate C++ code as described in “Generate Code Using Simulink Coder” (Simulink Coder) or “Generate Code Using Embedded Coder” (Embedded Coder).

See Also

More About

- “Reuse Custom Code in Stateflow Charts” on page 28-2
- “Choose a C++ Compiler”
- “Model Battery Management with Custom Code” on page 28-14

External Websites

- Supported and Compatible Compilers

Code Generation

- "Generate C or C++ Code from Stateflow Blocks" on page 29-2
- "Select Array Layout for Matrices in Generated Code" on page 29-5
- "Control Indicator Lamp Dimmer Using Variant Conditions" on page 29-9
- "Generate Code from Atomic Subcharts" on page 29-16
- "Set Configuration Parameters Programmatically" on page 29-18
- "Using Absolute Time Temporal Logic in Stateflow Charts" on page 29-19

Generate C or C++ Code from Stateflow Blocks

To generate C or C++ code from Simulink models that include a Stateflow chart, you must use Simulink Coder. In addition to Simulink Coder, you may use Embedded Coder to further enhance the generated code. Embedded Coder enhancements make your code more readable, more compact, and faster to execute.

When you generate code for a target, the Stateflow parser evaluates the graphical and nongraphical objects and data in each Stateflow machine against the supported chart notation and the action language syntax. For more information, see “Detect Common Modeling Errors During Simulation” on page 30-35.

Generate Code by Using Simulink Coder

Simulink Coder allows you to generate C and C++ code from models that contain Stateflow charts. You can then use the generated code for real-time and non-real-time applications, including:

- Simulation acceleration
- Rapid prototyping
- Hardware-in-the-loop (HIL) testing

Using Simulink Coder also allows you access to Classic Accelerator and Rapid Accelerator modes. Accelerator modes work by generating target code, which is then used for execution. For more information about these modes, see “How Acceleration Modes Work” (Simulink).

HIL testing allows you to test your controller design and determine if your physical system (plant) model is valid. For more information about HIL testing, see “Basics of Hardware-in-the-Loop simulation” (Simscape).

For more information, see “Generate Code Using Simulink Coder” (Simulink Coder).

Generate Code by Using Embedded Coder

With the addition of Embedded Coder you can generate C or C++ code that is more compact, easier to read, and faster to run. Embedded Coder additionally extends the abilities of Simulink Coder by allowing you control over generated functions, files, and data. Further, Embedded Coder enables easy integration for legacy code, data types, and calibration parameters. Embedded Coder supports software standards for:

- AUTOSAR
- MISRA C
- ASAP2

Embedded Coder also provides support packages with advanced optimizations and device drivers for specific hardware.

For more information, see “Generate Code Using Embedded Coder” (Embedded Coder).

Design Tips for Optimizing Generated Code for Stateflow Objects

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 "Specify Properties of Graphical Functions" on page 6-11.

Avoid Using Multiple Edge-Triggered Events in Stateflow Charts

If you use more than one trigger, you generate multiple code statements to handle rising or falling edge detections. If multiple triggers are required, use function-call events instead. For more information, see "Activate a Stateflow Chart by Sending Input Events" on page 12-8.

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 "Define Stateflow Structures" on page 26-2.

Use Discrete Sample Times

The code generated for discrete charts that are not inside a triggered or enabled subsystem uses integer counters to track time instead of Simulink provided time. The generated code uses less memory, and enables code for use in Software-in-the-Loop (SIL) and Processor-in-the-Loop (PIL) simulation modes.

Generate Code for Rapid Prototyping and Production Deployment

This table directs you to information about code generation based on your goals.

Goal	Simulink Coder Documentation	Embedded Coder Documentation
Generate C/C++ source code	"Source Code Generation" (Simulink Coder)	"Source Code Generation" (Embedded Coder)
Generate C/C++ source code and build executable	"Generated Code Compilation" (Simulink Coder)	"Generated Code Compilation" (Embedded Coder)
Integrate external code	"External Code Import" (Simulink Coder)	"External Code Import" (Embedded Coder)
Include external code only for library charts in a portable, self-contained library for use in multiple models	"Integrate External Code for Library Charts" (Simulink Coder)	"Integrate External Code for Library Charts" (Embedded Coder)
Optimize generated code	"Code Efficiency" (Simulink Coder)	"Code Efficiency" (Embedded Coder)

Traceability of Stateflow Objects in Generated Code

Traceability comments provide a way to:

- Verify generated code. You can identify which Stateflow object corresponds to a line of code and track code from different objects that you have or have not reviewed.
- Include comments in code generated for large-scale models. You can identify objects in generated code and avoid manually entering comments or descriptions.

To enable traceability comments, you must have Embedded Coder or HDL Coder software. For C/C++ code generation, comments appear in the generated code for embedded real-time (`ert`) based targets only. For more information, see "Trace Stateflow Elements in Generated Code" (Embedded Coder) and "Navigate Between Simulink Model and HDL Code by Using Traceability" (HDL Coder).

See Also

More About

- "Generate Code from Atomic Subcharts" on page 29-16
- "Select Array Layout for Matrices in Generated Code" on page 29-5
- "Generate Code Using Simulink Coder" (Simulink Coder)
- "Generate Code Using Embedded Coder" (Embedded Coder)

Select Array Layout for Matrices in Generated Code

When generating code from a Stateflow® chart, you can specify the array layout for matrices. For example, consider this matrix:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

By default, the code generator uses column-major layout to convert the matrix into a one-dimensional array and stores it in memory with this arrangement:

```
{1, 4, 2, 5, 3, 6}
```

If you select row-major layout, the code generator converts the matrix into a one-dimensional array and stores it in memory with this arrangement:

```
{1, 2, 3, 4, 5, 6}
```

If you have Embedded Coder®, you can preserve the multidimensionality of the matrix and store it as a two-dimensional array with this arrangement:

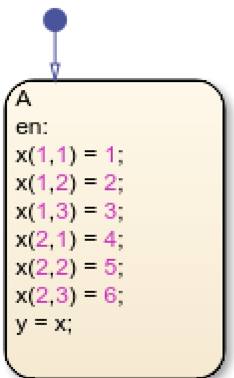
```
{{1, 2, 3}, {4, 5, 6}}
```

For more information, see “Code Generation of Matrices and Arrays” (Simulink Coder) and “Dimension Preservation of Multidimensional Arrays” (Embedded Coder).

Column-Major Array Layout

By default, the **Array Layout** configuration parameter for a Simulink® model is `Column-Major`. When you generate code from a model, the code generator flattens all matrix data into one-dimensional arrays in the column-major array layout.

For example, this Stateflow chart contains local data `x` of size `[2 3]`. The state actions index the elements in `x` by row and column number.



To generate code for this model:

1. In the **Apps** tab, select **Simulink Coder** or **Embedded Coder**.

2 In the **C Code** tab, click **Build**.

The file `sf_matrix_layout.c` implements the local data `x` in column-major layout with these lines of code:

```
...
sf_matrix_layout_DW.x[0] = 1.0;
sf_matrix_layout_DW.x[2] = 2.0;
sf_matrix_layout_DW.x[4] = 3.0;
sf_matrix_layout_DW.x[1] = 4.0;
sf_matrix_layout_DW.x[3] = 5.0;
sf_matrix_layout_DW.x[5] = 6.0;
...
```

The generated code refers to the elements of `x` by using only one index. The indices do not appear in increasing order.

Row-Major Array Layout

Row-major layout can improve the performance of certain algorithms. For example, see “Interpolation Algorithm for Row-Major Array Layout” (Embedded Coder).

To generate code that uses row-major array layout:

1. Open the Configuration Parameters dialog box.
2. In the **Code Generation** > **Interface** pane, set the **Array Layout** parameter to `Row-Major`.
3. Generate code as described in “Generate Code Using Simulink Coder” (Simulink Coder) or “Generate Code Using Embedded Coder” (Embedded Coder).

The file `sf_matrix_layout.c` implements the local data x with these lines of code:

```
...
sf_matrix_layout_DW.x[0] = 1.0;
sf_matrix_layout_DW.x[1] = 2.0;
sf_matrix_layout_DW.x[2] = 3.0;
sf_matrix_layout_DW.x[3] = 4.0;
sf_matrix_layout_DW.x[4] = 5.0;
sf_matrix_layout_DW.x[5] = 6.0;
...
```

The generated code refers to the elements of `x` by using only one index. The indices appear in increasing order.

When you enable row-major array layout, you can pass chart and message data as arguments to custom code functions in the row-major array layout. You can also use row-major as the default layout for custom code variables. To implement row-major as the default array layout for custom code functions and variables:

1. Open the Configuration Parameters dialog box.
2. In the **Code Generation** > **Interface** pane, set the **Array Layout** parameter to `Row-Major`.
3. In the **Simulation Target** pane, under **Advanced parameters**, select **Import custom code**.
4. In the **Import settings** tab, set the **Default function array layout** parameter to `Row-major`.

You can also specify row-major array layout for individual functions. In the **Simulation Target** pane, in the **Import settings** tab, click **Exception by function**. In the Array Layout for Custom Code

Functions window, you can add or remove functions and specify the individual array layout for each function.

If you enable row-major array layout in a chart that uses custom C code, global variables and arguments of custom code functions defined in the custom code must be scalars, vectors, or structures of scalars and vectors. Specify the size of an n-element vector as n, and not as [n 1] or [1 n].

When you enable row-major array layout in charts that use change detection operators, code generation produces an error. Before generating code in charts that use change detection operators, enable column-major array layout. See “Change Detection Operators” on page 14-63.

Multidimensional Array Layout

If you have Embedded Coder, you can generate code that preserves the multidimensionality of Stateflow data without flattening the data into one-dimensional arrays.

To generate code for the previous example using multidimensional array layout:

1. Enable row-major layout.
2. In the **Apps** tab, select **Embedded Coder**.
3. In the **C Code** tab, select **Code Interface > Default Code Mappings** to open the **Code Mappings** editor and the **Property Inspector**.
4. In the **Code Mappings** editor, on the **Data Defaults** tab, select the **Signals, states, and internal data** category and set the **Storage Class** as `Localizable`. If the Code Mappings editor is empty, navigate to the Simulink Model.
5. In the **Property Inspector**, in the **Code** section, select **PreserveDimensions**.
6. In the **C Code** tab, click **Build**.

The file `sf_matrix_layout.c` implements the local data x with these lines of code:

```
...
sf_matrix_layout_DW.x[0][0] = 1.0;
sf_matrix_layout_DW.x[0][1] = 2.0;
sf_matrix_layout_DW.x[0][2] = 3.0;
sf_matrix_layout_DW.x[1][0] = 4.0;
sf_matrix_layout_DW.x[1][1] = 5.0;
sf_matrix_layout_DW.x[1][2] = 6.0;
...
```

The generated code refers to the elements of x by using two indices.

Multidimensional array layout is available for:

- Constant and local data in Stateflow charts
- Message data in Stateflow charts
- Parameters and root-level inport and outport data in Simulink models

Multidimensional layout is not available for bus signals containing multidimensional array data.

Multidimensional layout is not supported in reusable charts or charts in reusable parent subsystems.

For more information, see "Preserve Dimensions of Multidimensional Arrays in Generated Code" (Embedded Coder).

See Also

Model Settings
Array layout | Default function array layout | Exception by function

More About

- "Generate C or C++ Code from Stateflow Blocks" on page 29-2
- "Code Generation of Matrices and Arrays" (Simulink Coder)
- "Interpolation Algorithm for Row-Major Array Layout" (Embedded Coder)
- "Dimension Preservation of Multidimensional Arrays" (Embedded Coder)
- "Preserve Dimensions of Multidimensional Arrays in Generated Code" (Embedded Coder)

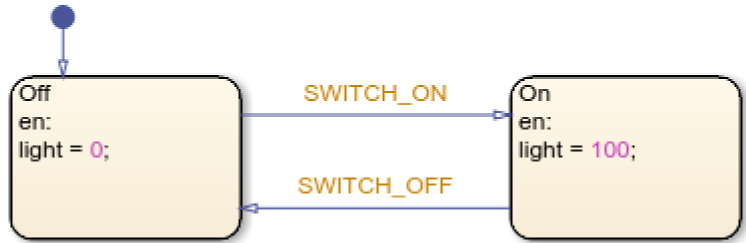
Control Indicator Lamp Dimmer Using Variant Conditions

This example shows how to use variant transitions to model multiple designs in a single Stateflow® chart. Each variant transition leads to a state that represents a variant configuration for your system. When you generate code from the chart, you can select which variant configuration you want to use. By default, the generated code only contains the code and data needed to execute the variant configuration you select.

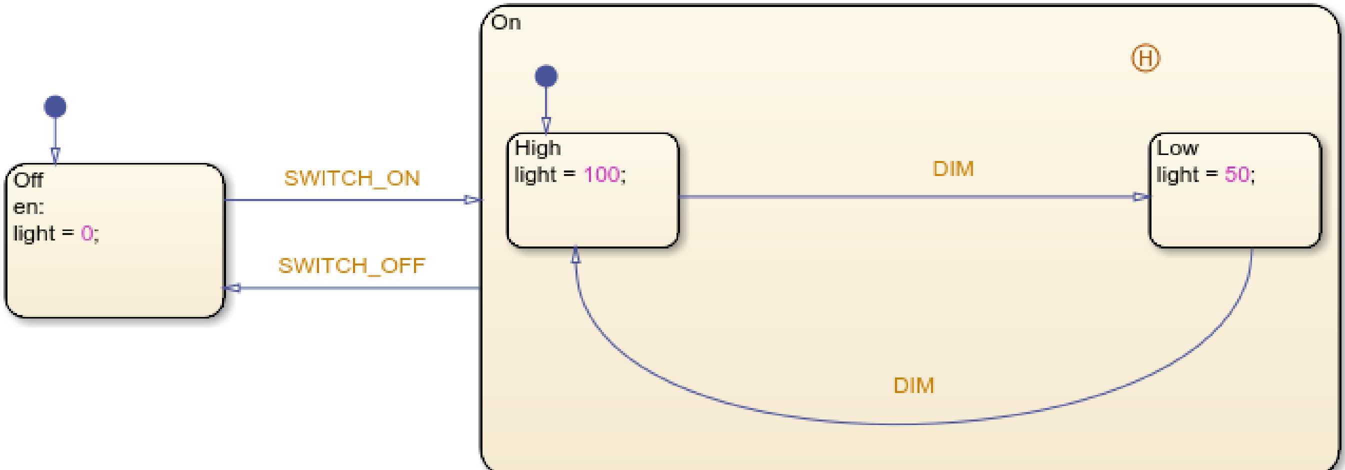
Variant Configurations for Lamp Control System

In this example, a Stateflow chart models a control system for a lamp. The chart uses variant configurations to combine three possible lamp designs.

The simplest design consists of a lamp with a single on-off switch. To model this design, you can create a Stateflow chart with two states, `Off` and `On`, that set the value of the chart output `light` to `0` or `100`, respectively. The chart transitions between these states when it receives the input events `SWITCH_ON` and `SWITCH_OFF`.

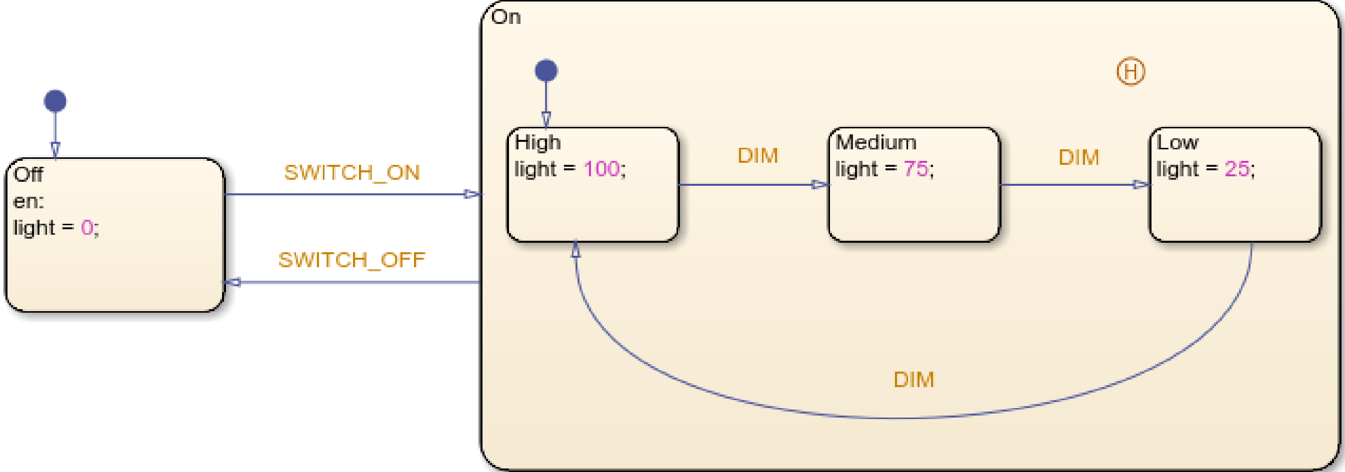


A more advanced lamp design includes a dimmer with two settings, high and low. To model this lamp design, you can add two substates to the state `On`. When the lamp is on, the substates `High` and `Low` set the value of the chart output `light` to `100` or `50`, respectively. The input event `DIM` triggers the transition between these substates.

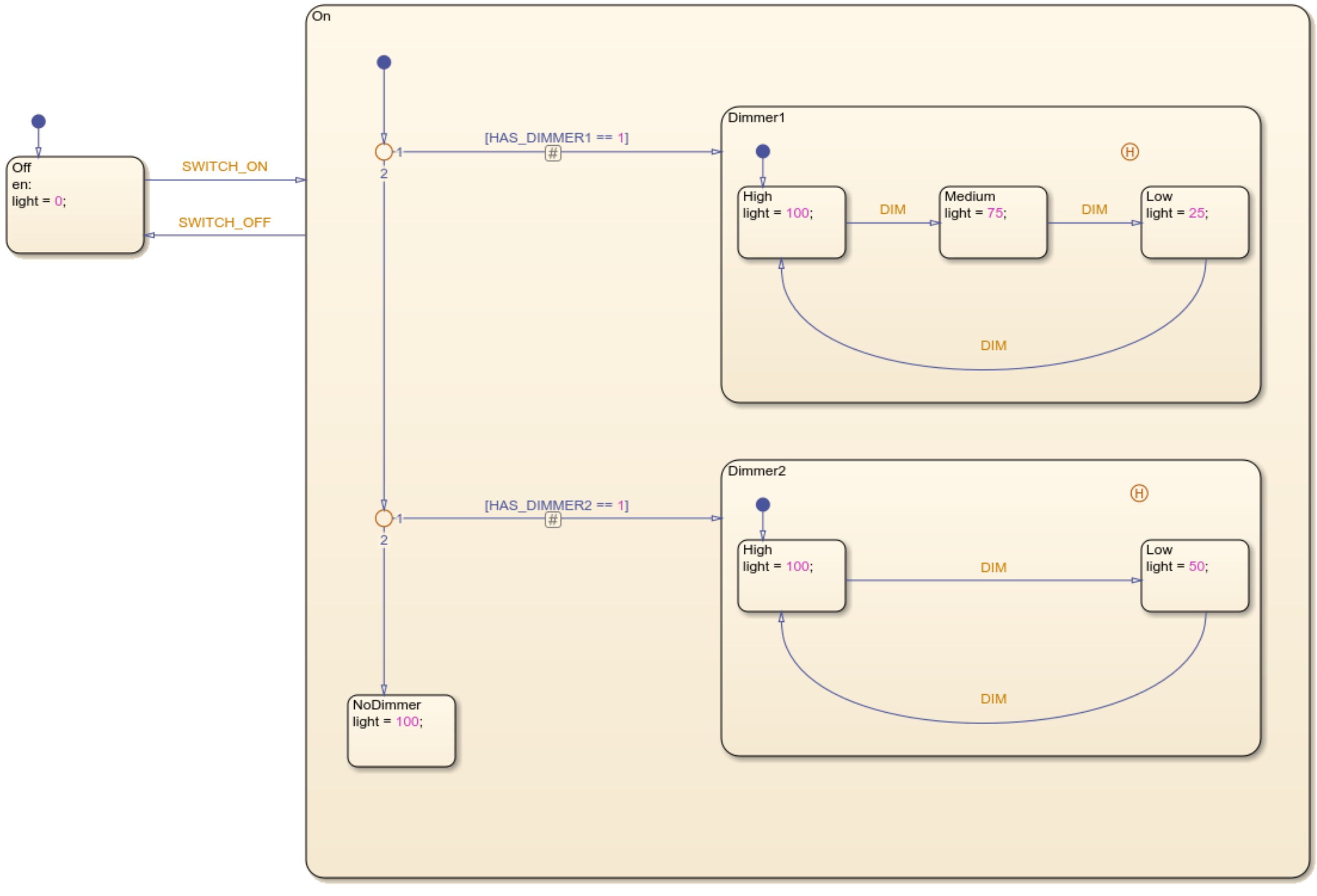


Finally, the third design uses a dimmer with three settings, high, medium, and low. In a chart that models this design, the state `On` contains three substates that set the value of the chart output `light`

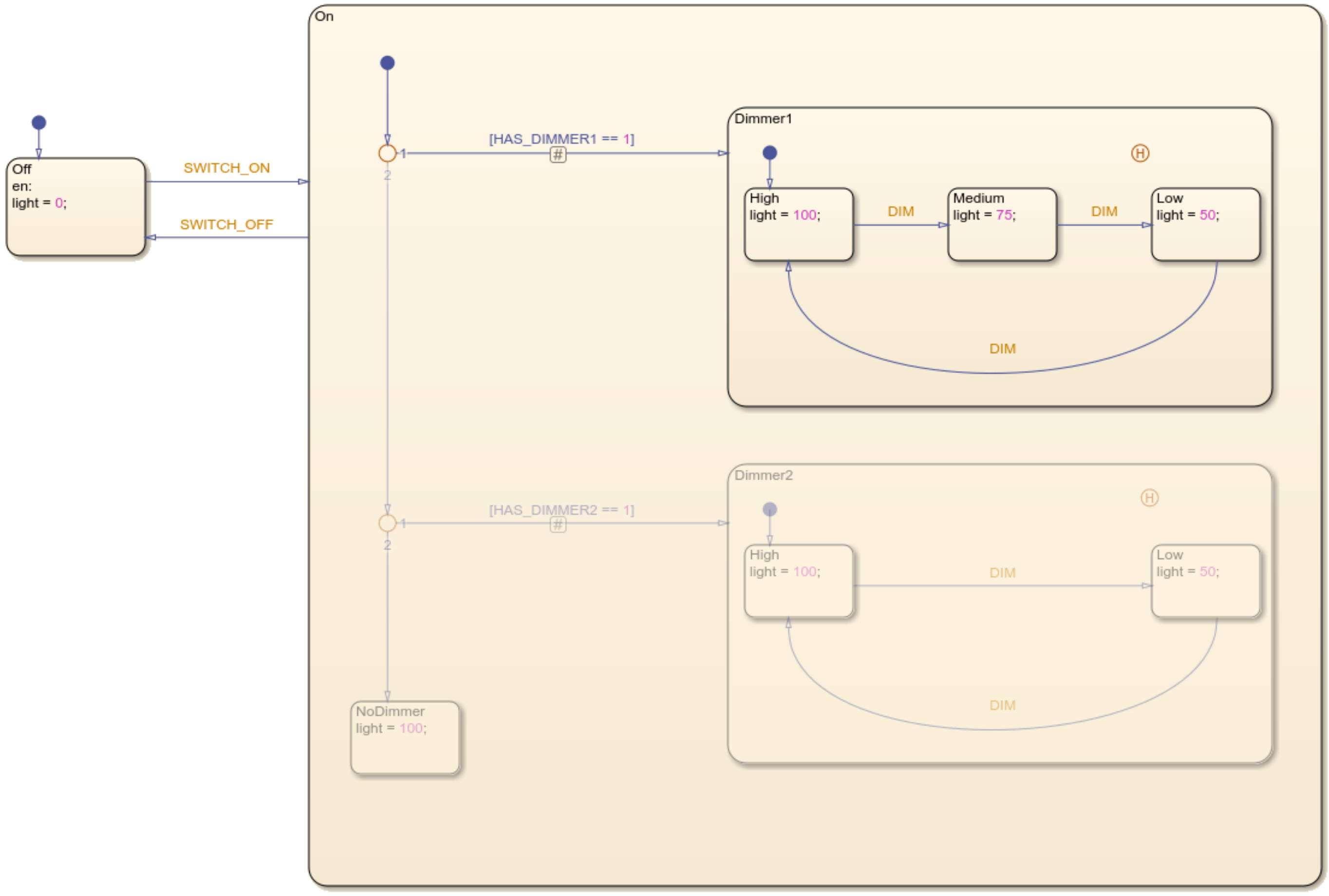
to 100, 75, or 50, respectively. The input event DIM triggers the transition from High to Medium, from Medium to Low, and from Low to High.



This example combines these designs into a single Stateflow chart. In the state On, the substates Dimmer1, Dimmer2, and NoDimmer represent the variant configurations for the system. A pair of variant transitions evaluate the parameters HAS_DIMMER1 and HAS_DIMMER2 and guard the entry into each of these substates.



When you simulate, compile, or generate code from the model, the conditions on the variant transitions determine which variant configuration is active. The portions of the chart that correspond to inactive variant configurations appear dimmed on the Stateflow canvas. For instance, in this chart, the variant configuration with three dimmer settings is active.



To change which variant configuration is currently active, modify the guarding parameters in the base workspace and update the model.

Generate Code for Variant Configurations

To generate code from your Stateflow chart, you must have Simulink Coder™ or Embedded Coder®. By default, the generated code only contains the code and data for the active variant configuration. For example, if you select the variant configuration with three dimmer settings, the generated code defines these constants:

```
/* Named constants for Chart: '<Root>/Lamp' */
#define sfVariantLam_IN_NO_ACTIVE_CHILD ((uint8_T)0U)
#define sfVariantLampE_event_SWITCH_OFF (1)
#define sfVariantLampEx_event_SWITCH_ON (2)
#define sfVariantLampExample_IN_High   ((uint8_T)1U)
#define sfVariantLampExample_IN_Low    ((uint8_T)2U)
#define sfVariantLampExample_IN_Medium ((uint8_T)3U)
#define sfVariantLampExample_IN_Off    ((uint8_T)1U)
#define sfVariantLampExample_IN_On     ((uint8_T)2U)
#define sfVariantLampExample_event_DIM (0)
```

If you are using Embedded Coder, you can include preprocessor conditional `#if` statements in your generated code by enabling the chart property **Variant activation time**, as described in “Specify

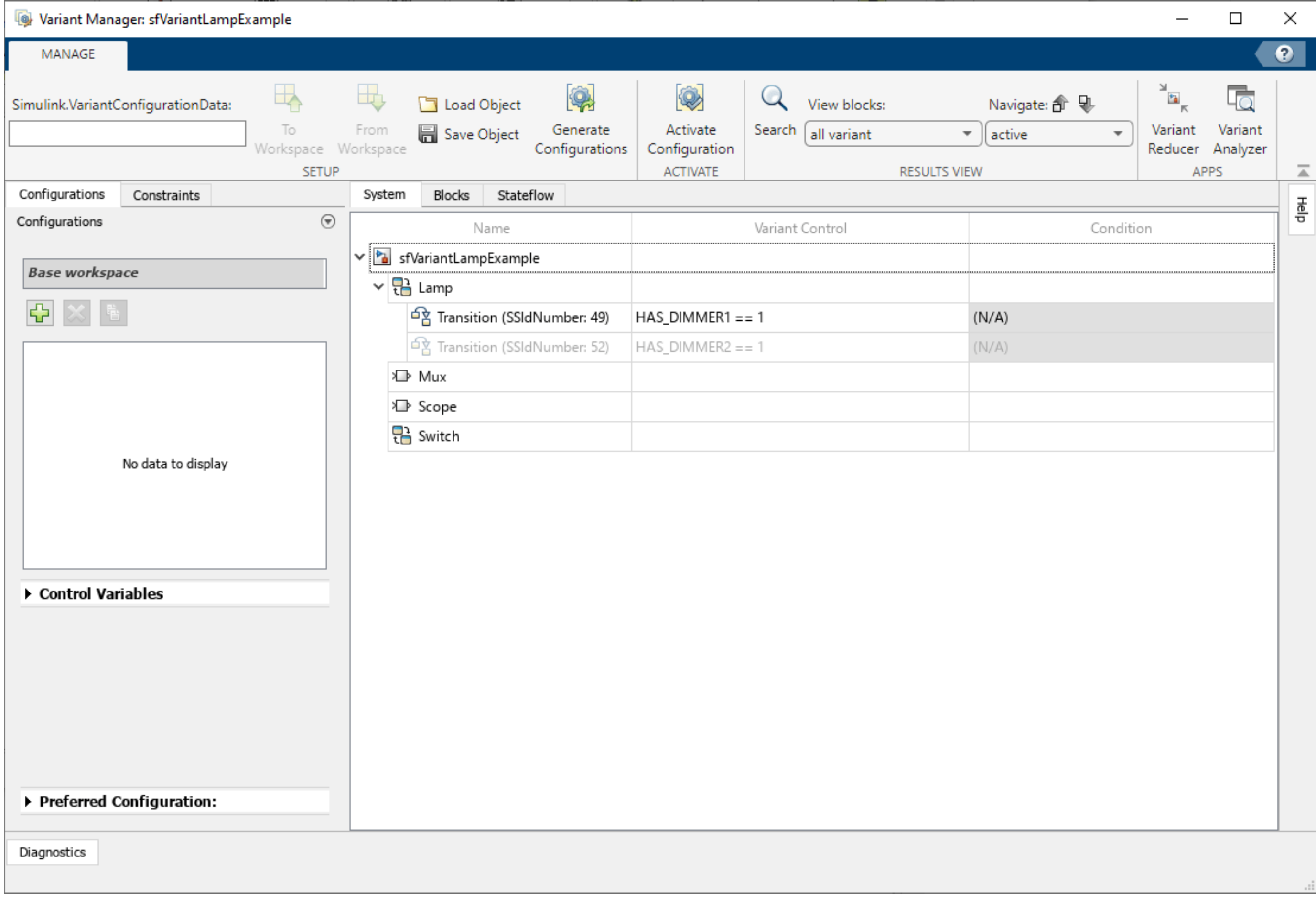
Properties for Stateflow Charts” on page 1-19. For example, if you enable this chart property, the generated code defines these constants regardless of which variant configuration is active:

```
/* Named constants for Chart: '<Root>/Lamp' */
#if HAS_DIMMER1 == 1 || HAS_DIMMER2 == 1
#define sfVariantLam_IN_NO_ACTIVE_CHILD ((uint8_T)0U)
#endif

#define sfVariantLampE_event_SWITCH_OFF (1)
#define sfVariantLampEx_event_SWITCH_ON (2)
#if HAS_DIMMER1 == 1 || HAS_DIMMER2 == 1
#define sfVariantLampExample_IN_High   ((uint8_T)1U)
#endif

#if HAS_DIMMER1 == 1 || HAS_DIMMER2 == 1
#define sfVariantLampExample_IN_Low    ((uint8_T)2U)
#endif

#if HAS_DIMMER1 == 1
#define sfVariantLampExample_IN_Medium ((uint8_T)3U)
#endif

#define sfVariantLampExample_IN_Off    ((uint8_T)1U)
#define sfVariantLampExample_IN_On     ((uint8_T)2U)
#define sfVariantLampExample_event_DIM (0)
```

For more information about generating code, see “Generate Code Using Simulink Coder” (Simulink Coder) and “Generate Code Using Embedded Coder” (Embedded Coder).

Manage Variant Configurations with the Variant Manager

The Variant Manager allows you to manage variant configurations in your model. To open the Variant Manager, on the **Modeling** tab, under **Design Data**, select **Variant Manager**.



For more information about how to use the Variant Manager, see “Variant Manager for Simulink” (Simulink).

Guidelines for Using Variant Transitions

To model several system designs in a single chart, represent each variant configuration with one or more states connected by variant transitions. Follow these guidelines:

- In the transition path to each variant configuration, you can combine variant transitions or transitions with empty labels.
- Guard each variant transition by using a Simulink® parameter or a MATLAB® variable defined in the base workspace. For more information, see “Share Parameters with Simulink and the MATLAB Workspace” on page 10-32 and “Options to Represent Variant Parameters in Generated Code” (Embedded Coder).
- Because the chart determines which variant configuration is active when you simulate, compile, or generate code from your model, variant transitions cannot contain event or message triggers, condition actions, or transition actions.

- To convert a transition to a variant transition, click the transition. Then, in the **Transition** tab, select **Variant Transition**. A variant transition badge appears at the midpoint of the variant transition.

See Also

More About

- “Generate C or C++ Code from Stateflow Blocks” on page 29-2
- “Share Parameters with Simulink and the MATLAB Workspace” on page 10-32
- “Generate Code Using Simulink Coder” (Simulink Coder)
- “Generate Code Using Embedded Coder” (Embedded Coder)
- “Variant Manager for Simulink” (Simulink)
- “Options to Represent Variant Parameters in Generated Code” (Embedded Coder)

Generate Code from Atomic Subcharts

To unit test a Stateflow chart in a Simulink model, first break the chart into smaller, independent components by using atomic subcharts. When you generate code for your chart, a separate file stores the code for the atomic subchart. Generating reusable code from atomic subcharts is useful for testing individual parts of your Stateflow chart. For more information, see “Create Reusable Subcomponents by Using Atomic Subcharts” on page 17-2.

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 these 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.
3. Open the Configuration Parameters dialog box.
4. In the **Code Generation** pane, set the **System target file** parameter to `ert.tlc`.
5. (OPTIONAL) Customize the generated function names for atomic subcharts. In the **Code Generation** > **Identifiers** pane, set the **Subsystem methods** parameter. Specify the format of the function names by using a combination of these tokens:
   - `$R` — root model name
   - `$F` — type of interface function for the atomic subchart
   - `$N` — block name
   - `$H` — subsystem index
   - `$M` — name-mangling text

For more information, see “Generate Separate Code for an Atomic Subchart” on page 17-44.

Generate Reusable Code for Linked Atomic Subcharts

To specify code generation parameters for linked atomic subcharts from the same library:

1. Open the library model that contains your atomic subchart.
2. Unlock the library.
3. Right-click the library chart and select **Block Parameters**.
4. In the dialog box, specify these 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.
5. In the **Code Generation** tab, set the **System target file** parameter to `ert.tlc`.

6 (OPTIONAL) Customize the generated function names for atomic subcharts. In the **Code Generation** > **Identifiers** tab, set the **Subsystem methods** parameter. Specify the format of the function names by using a combination of these tokens:

- `$R` — root model name
- `$F` — type of interface function for the atomic subchart
- `$N` — block name
- `$H` — subsystem index
- `$M` — name-mangling text

When you generate code for your model, a separate file stores the code for linked atomic subcharts from the same library.

See Also

More About

- “Create Reusable Subcomponents by Using Atomic Subcharts” on page 17-2
- “Generate C or C++ Code from Stateflow Blocks” on page 29-2
- “Generate Separate Code for an Atomic Subchart” on page 17-44

Set Configuration Parameters Programmatically

In Stateflow charts in Simulink models, you can use the command-line API to change the settings in the Configuration Parameters dialog box.

1. At the MATLAB command prompt, store the `Simulink.ConfigSet` object that contains the configuration parameters for the current model.

   ```
   configSet = getActiveConfigSet(gcs)
   ```
2. To get the current value of a configuration parameter, call the `get_param` function:

   ```
   get_param(configSet,"parameter_name")
   ```
3. To set a configuration parameter, call the `set_param` function:

   ```
   set_param(configSet,parameter_name=value)
   ```

For example, you can set the **Reserved names** parameter for simulation by entering:

```
configSet = getActiveConfigSet(gcs)
set_param(configSet,SimReservedNameArray=["abc","xyz"])
```

For more information on the configuration parameters that you can set programmatically, click the parameters listed on these pages and navigate to the **Command-Line Information** section:

- “Solver Pane” (Simulink)
- Data Import/Export (Simulink)
- Stateflow Diagnostics (Simulink)
- Simulation Target (Simulink)
- Code Generation (Simulink Coder)
- Code Generation Custom Code (Simulink Coder)

See Also

`get_param` | `set_param`

More About

- “Set Model Configuration Parameters for a Model” (Simulink)
- “Configure Code Generation Parameters for Model Programmatically” (Simulink Coder)
- “Recommended Settings Summary for Model Configuration Parameters” (Simulink Coder)

Using Absolute Time Temporal Logic in Stateflow Charts

When you use absolute time temporal logic in your Stateflow Chart blocks in your model for HDL code generation, use these settings.

For the sample rate of the chart:

- If you use seconds (`sec`), then the sample time must be an integer 65535 or lower, or a decimal between 65.535 and 0.001 with no more than three decimal places.
- If you use milliseconds (`msec`), the sample time must be a decimal between 65.535 and 0.001 with no more than three decimal places, or a decimal between 0.065535 and 0.000001 with no more than six decimal places.
- If you use microseconds (`usec`), the sample time must be a decimal between 0.065535 and 0.000001 with no more than six decimal places, or a decimal between 0.000065535 and 0.000000001 with no more than nine decimal places.
- If the sample time is an integer below 2^16, then use `sec`.
- If 1000 * sample time is an integer below 2^16, then use `sec` or `msec`.
- If 1000000 * sample time is an integer below 2^16, then use `msec` or `usec`.
- If 1000000000 * sample time is an integer below 2^16, then use `usec`.

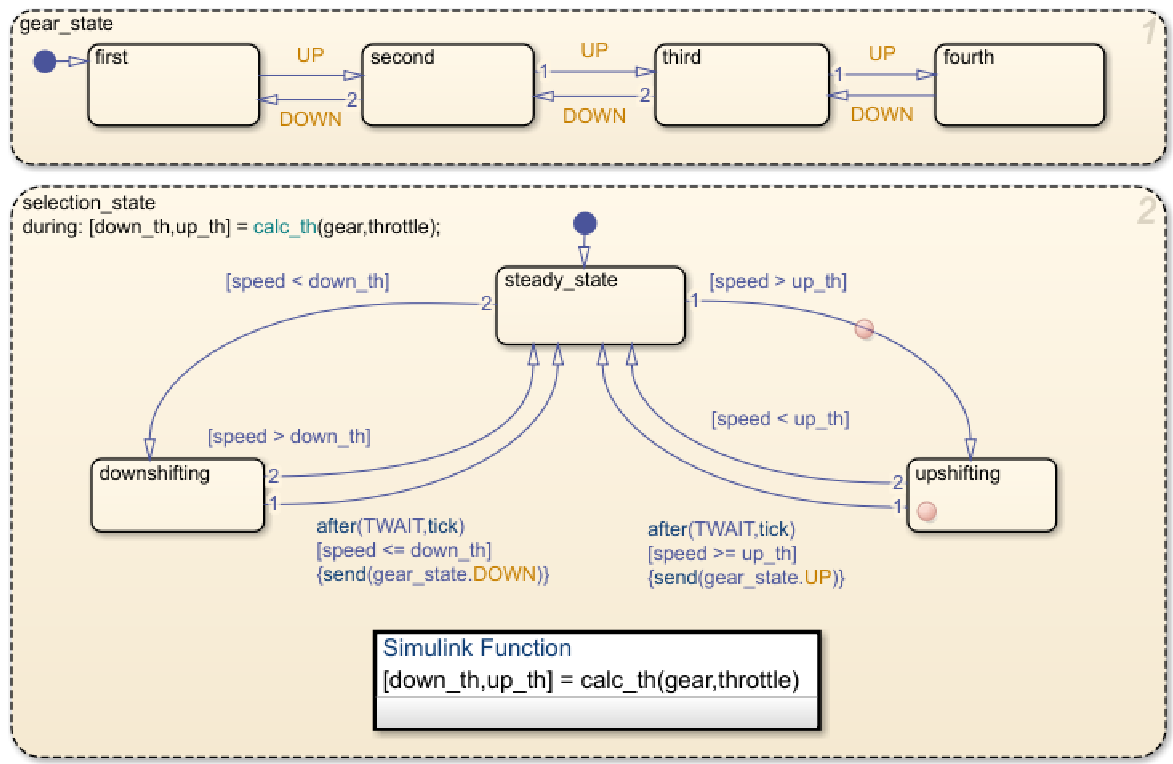
Debug and Test Stateflow Charts

- “Set Breakpoints to Debug Charts” on page 30-2
- “Inspect and Modify Data and Messages While Debugging” on page 30-8
- “Control Chart Execution After a Breakpoint” on page 30-15
- “Debug Run-Time Errors in a Chart” on page 30-19
- “Detect Modeling Errors During Edit Time” on page 30-21
- “Detect Common Modeling Errors During Simulation” on page 30-35
- “Animate Stateflow Charts” on page 30-40
- “Comment Out Objects in a Stateflow Chart” on page 30-41
- “Avoid Unwanted Recursion in a Chart” on page 30-43

Set Breakpoints to Debug Charts

You enable debugging for a Stateflow chart when you set a breakpoint. A breakpoint is a point on a Stateflow chart that pauses the simulation so you can examine the status of the chart. While simulation is paused, you can view Stateflow data, interact with the MATLAB workspace, and step through the simulation.

Breakpoints appear as circular red badges. For example, this chart contains breakpoints on the `upshifting` state and the transition from `steady_state` to `upshifting`.



Set a Breakpoint for a Stateflow Object

You can set breakpoints on charts, states, transitions, graphical or truth table functions, and events.

Breakpoints on Charts

To set a breakpoint on a chart, right-click inside the chart and select **Set Breakpoint on Chart Entry**. This type of breakpoint pauses the simulation before entering the chart.

To remove the breakpoint, right-click inside the chart and clear the **Set Breakpoint on Chart Entry** option.

Breakpoints on States and Transitions

You can set different types of breakpoints on states and transitions.

Object	**Breakpoint Type**
State	`On State Entry` — Pause the simulation before performing the state `entry` actions.
	`During State` — Pause the simulation before performing the state `during` actions.
	`On State Exit` — Pause the simulation after performing the state `exit` actions.
Transition	`When Transition is Tested` — Pause the simulation before testing that the transition is a valid path. If no condition exists on the transition, this breakpoint type is not available.
	`When Transition is Valid` — Pause the simulation after the transition is valid, but before taking the transition.

To set a breakpoint on a state or transition, right-click the state or transition and select **Set Breakpoint**. For states, the default breakpoints are `On State Entry` and `During State`. For transitions, the default breakpoint is `When Transition is Valid`. To change the type of breakpoint, click the breakpoint badge and select a different configuration of breakpoints. For more information, see "Change Breakpoint Types" on page 30-4.

To remove the breakpoint, right-click the state or transition and select **Clear Breakpoint**.

Breakpoints on Stateflow Functions

To set a breakpoint on a graphical or truth table function, right-click the function and select **Set Breakpoint During Function Call**. This type of breakpoint pauses the simulation before calling the function.

To remove the breakpoint, right-click the function and clear the **Set Breakpoint During Function Call** option.

Breakpoints on Events

You can select two types of breakpoints on events:

- `Start of Broadcast` — Pause the simulation before broadcasting the event.
- `End of Broadcast` — Pause the simulation after a Stateflow object reads the event.

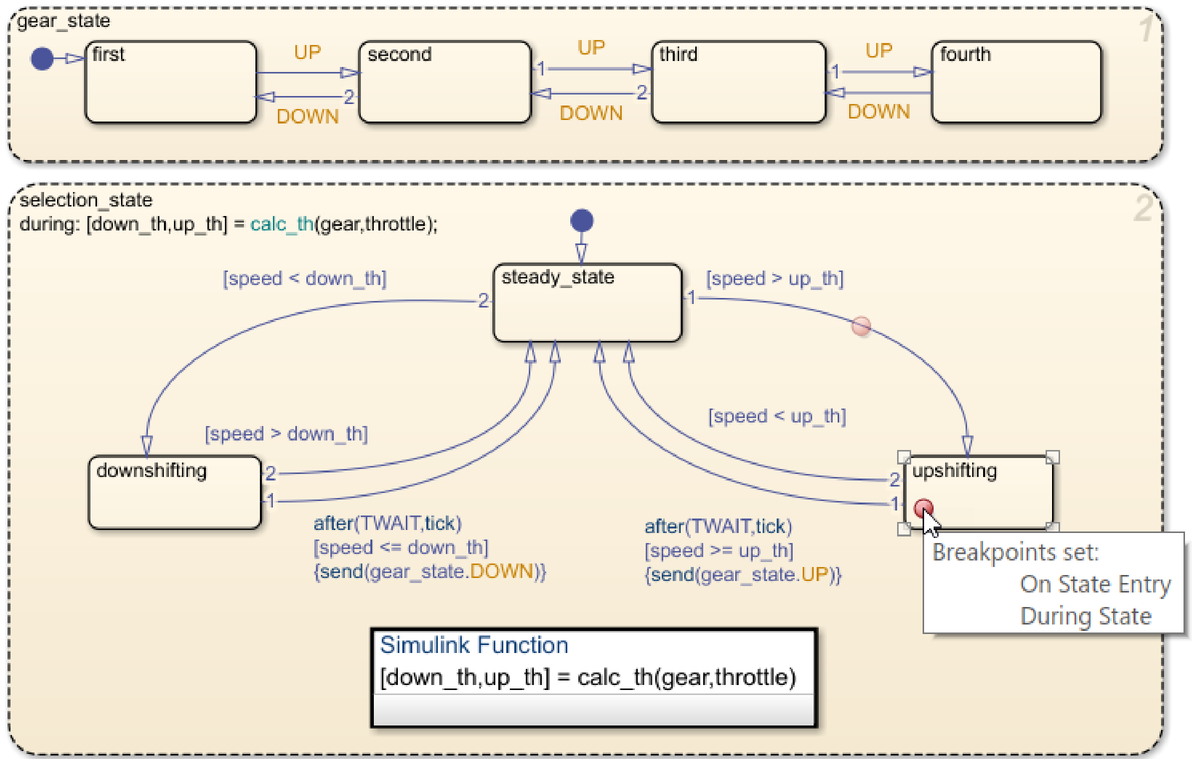
Available breakpoints depend on the scope of the event.

Scope of Event	**Start of Broadcast**	**End of Broadcast**
`Local`	Available	Available
`Input`	Available	Not available
`Output`	Not available	Not available

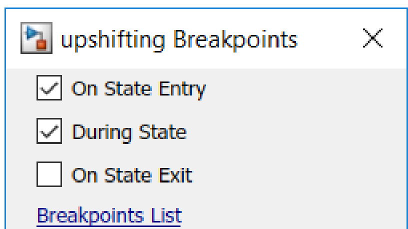
To set or clear breakpoints on an event, use the **Property Inspector** or the Model Explorer to modify the **Debugger Breakpoints** properties. For more information, see "Debugger Breakpoints" on page 12-6.

Change Breakpoint Types

A breakpoint badge can represent more than one type of breakpoint. To see a tooltip that lists the breakpoint types that are set on a Stateflow object, point to its badge. In this example, the badge on the state `upshifting` represents two breakpoint types: `On State Entry` and `During State`.



To change the type of breakpoint on an object, click the breakpoint badge. In the Breakpoints dialog box, you can select a different configuration of breakpoints, depending on the object type.



Clearing all of the check boxes in the Breakpoints dialog box removes the breakpoint.

Add Breakpoint Conditions

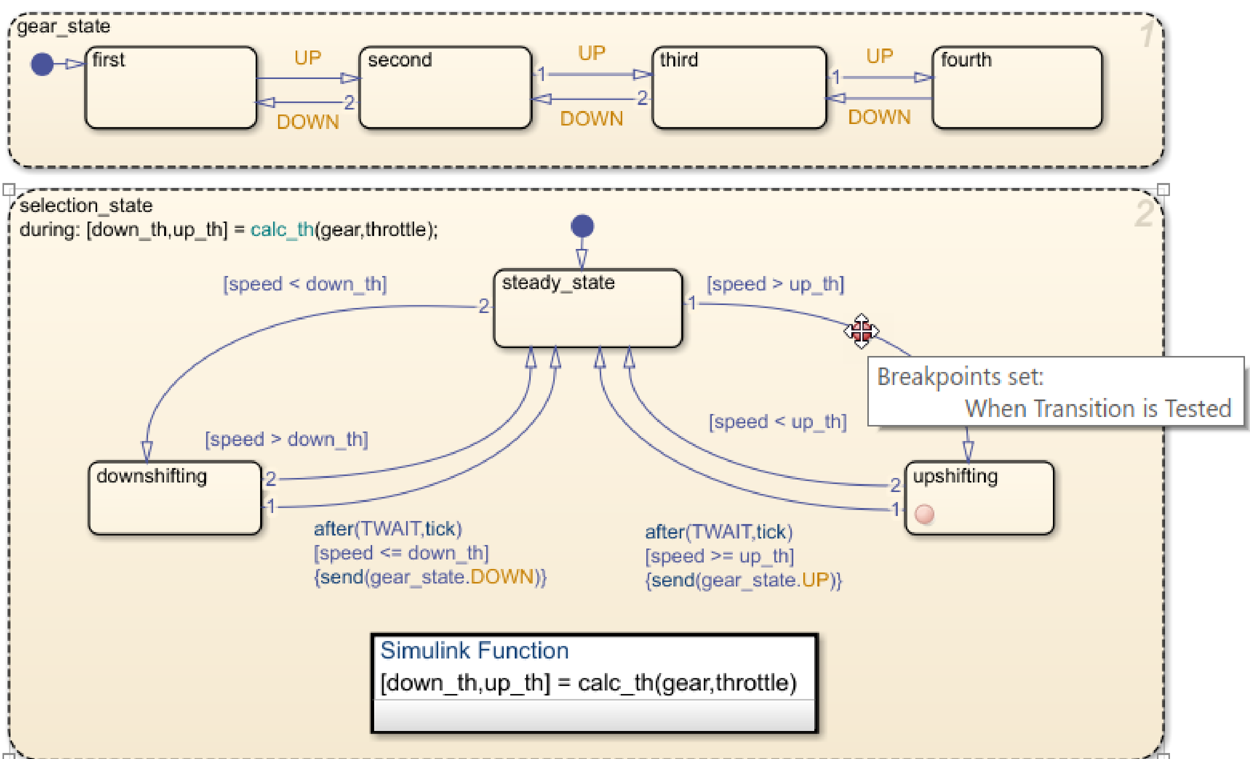
To limit the number of times that the simulation stops at a breakpoint, add a condition to the breakpoint. By default, a Stateflow chart pauses whenever it reaches a breakpoint. When you add a condition to a breakpoint, the chart pauses at the breakpoint only when the condition is true.

To add a condition to a breakpoint:

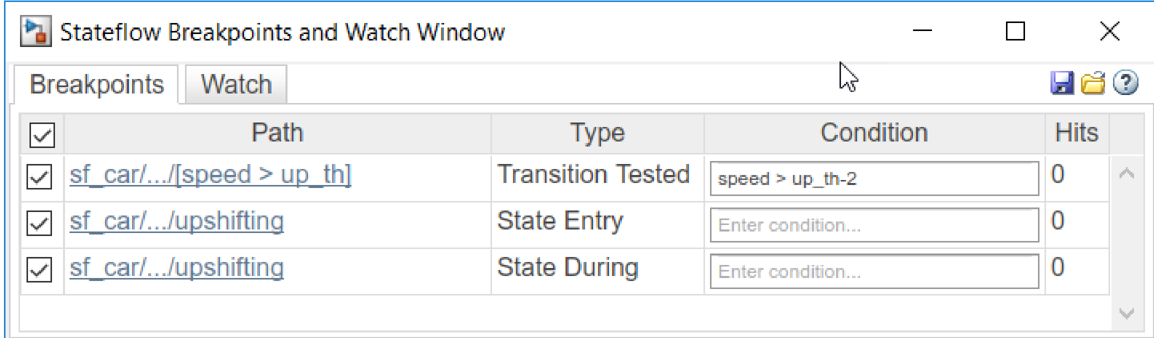
1. On the **Debug** tab, click **Breakpoints List** to open the Stateflow Breakpoints and Watch window. Alternatively, you can open the Breakpoints and Watch window by clicking the **Breakpoints List** link in the Breakpoints dialog box.
2. Select the **Breakpoints** tab. The **Breakpoints** tab lists all of the breakpoints in the chart. For more information, see “Manage Breakpoints Through the Breakpoints and Watch Window” on page 30-6.
3. Under the **Condition** column, enter a condition for the breakpoint. You can use any valid MATLAB expression that combines numerical values and Stateflow data objects that are in scope at the breakpoint.

Note You cannot use message data in a breakpoint condition expression.

For example, this chart has a breakpoint on the transition from `steady_state` to `upshifting`. This breakpoint stops the simulation every time that the transition is tested, even if the value of `speed` is far below `up_th`.



To inspect the chart before the transition is taken, you want the breakpoint to pause the simulation only when the value of `speed` is approaching the value of `up_th`. When you set the condition `speed > up_th-2` on the breakpoint, the simulation pauses only when the value of `speed` is within 2 of the value of `up_th`.



When the simulation pauses, you can inspect the values of the variables `speed` and `up_th` and step through the simulation. For more information, see “Inspect and Modify Data and Messages While Debugging” on page 30-8 and “Control Chart Execution After a Breakpoint” on page 30-15.

Manage Breakpoints Through the Breakpoints and Watch Window

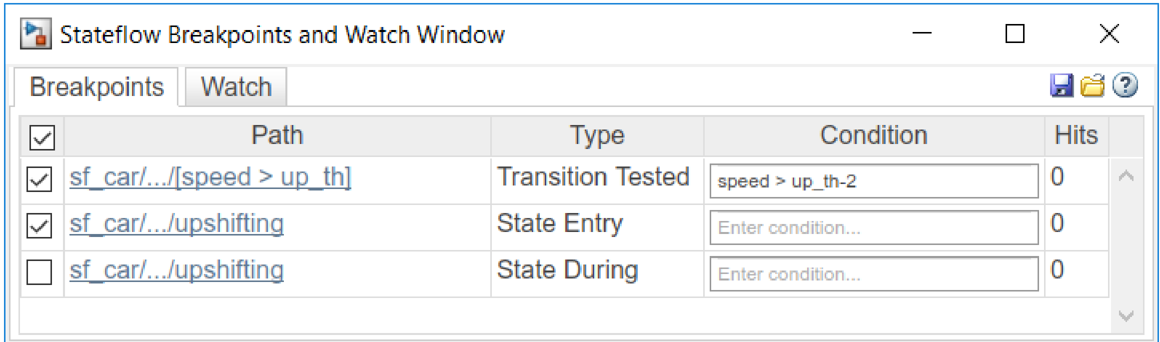
You can manage all of the breakpoints in the chart in the Stateflow Breakpoints and Watch window. To open the Breakpoints and Watch window, on the **Debug** tab, click **Breakpoints List**. Alternatively, open the Breakpoints dialog box and click the **Breakpoints List** link.

- To see a list of all of the breakpoints and their associated conditions, select the **Breakpoints** tab.
- To inspect data and message values, select the **Watch** tab. For more information, see “View Data in the Breakpoints and Watch Window” on page 30-10.

Tip You can also manage the breakpoints in your Stateflow chart by using the breakpoint list in the Simulink Editor. For more information, see “Debug Simulation Using Signal Breakpoints” (Simulink).

Disable and Reenable Breakpoints

To disable a breakpoint without deleting its associated condition, clear the check box next to the breakpoint name. For example, in this chart, the breakpoint on the `During State` breakpoint for the `upshifting` state is disabled.

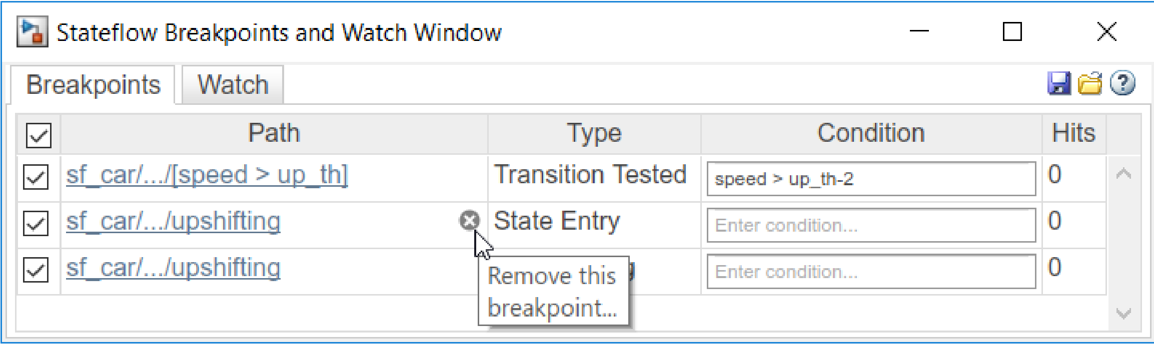


If you disable all the breakpoints for a graphical object, its breakpoint badge changes color from red to gray. If there is at least one breakpoint enabled for an object, the breakpoint badge remains red.

To reenable a breakpoint, select the box next to the breakpoint name. To disable or reenable all breakpoints, clear or select the check box at the top of the window.

Remove Breakpoints

To remove a breakpoint from the chart, point to the name of the breakpoint and click the **Remove this breakpoint** icon that appears to the right of the name. When you remove a breakpoint, you also delete its associated condition.



View Breakpoint Hits

The **Hits** column displays the number of times that the simulation has paused on each breakpoint. When you change the condition for a breakpoint, the chart resets the number of hits.

Save and Restore Breakpoints

Breakpoints persist during a MATLAB session. When you close a model, its breakpoints remain in the Breakpoints and Watch window. If you reopen a model during the same MATLAB session, all of the breakpoints and their associated conditions are restored.

You can save the breakpoint and watch data lists and reload them in a later MATLAB session. To save a snapshot of the breakpoint and watch data lists, at the top of the Breakpoints and Watch Window, click the **Save current breakpoints and watches** icon. To restore a snapshot, click the **Load breakpoints and watches** icon.

See Also

More About

- “Inspect and Modify Data and Messages While Debugging” on page 30-8
- “Control Chart Execution After a Breakpoint” on page 30-15
- “Debug Run-Time Errors in a Chart” on page 30-19

Inspect and Modify Data and Messages While Debugging

While your Stateflow chart is in debugging mode, you can examine the status of the chart by inspecting the values of data, messages, and temporal logic expressions. You can also test the design of the chart by modifying data values and sending local and output messages. This table summarizes the interfaces that you can use to perform these debugging tasks. For more information, see "Set Breakpoints to Debug Charts" on page 30-2.

Debugging Task	Stateflow Editor	Symbols Pane	Breakpoints and Watch Window	MATLAB Command Window
Inspect values of data and messages	Yes on page 30-8	Yes on page 30-9	Yes on page 30-10	Yes on page 30-11
Inspect temporal logic expressions	Yes on page 30-8	No	No	No
Modify values of data and messages	No	Yes on page 30-9	No	Yes on page 30-12
Send messages	No	No	No	Yes on page 30-13

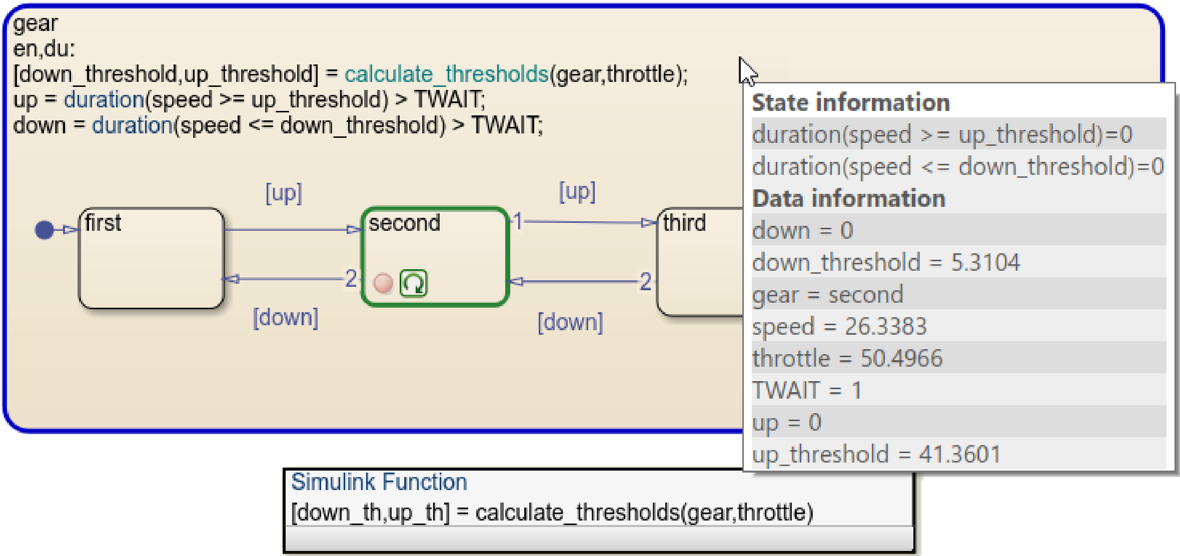
View Data in the Stateflow Editor

While the simulation is paused at a breakpoint, you can examine data values by pointing to a state, transition, or function in the chart. A tooltip displays the value of the data and the messages that the selected object uses.

Object Type	Tooltip Information
States and transitions	Values of data, messages, and temporal logic expressions that the object uses
Graphical, truth table, and MATLAB functions	Values of local data, messages, inputs, and outputs in the scope of the function

For example, the breakpoint in this chart pauses the simulation when the `second` state evaluates its `during` actions. Pointing to the superstate `gear` displays a tooltip that shows the values of:

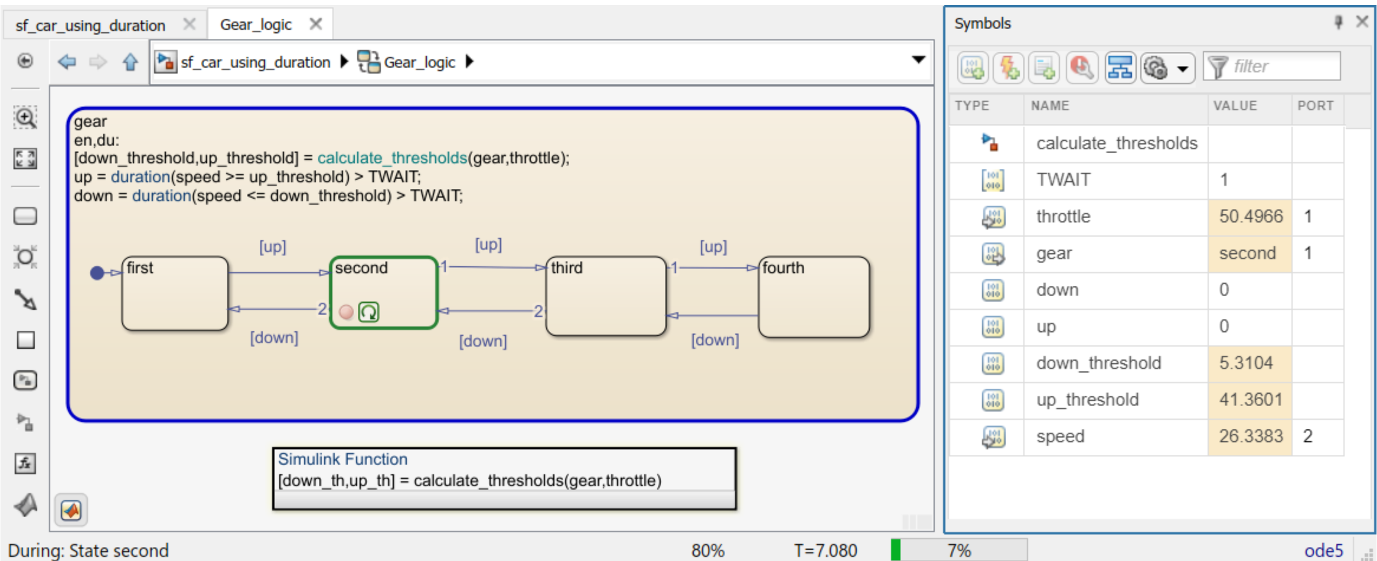
- Temporal logic expressions `duration(speed >= up_threshold)` and `duration(speed <= down_threshold)`.
- Data, including `speed`, `up_threshold`, and `up`.



Note If you select the chart properties **Export chart level functions** and **Treat exported functions as globally visible**, the tooltip does not display temporal logic data.

View and Modify Data in the Symbols Pane

While a chart is in debugging mode, the **Symbols** pane displays the value of each data and message object in the chart. For example, when this chart pauses at the breakpoint, you can see the values of all chart data listed in the **Value** column. The highlighted values changed during the last time step.



In the **Symbols** pane, you can change the value of:

- Data store memory, local, and output data.

- Local and output messages.

Click the **Value** field for a data or message object to enter a new value.

You cannot change the values of constants, parameters, or input data and messages during simulation.

For more information, see "Manage Symbols in the Stateflow Editor" on page 25-14.

View Data in the Breakpoints and Watch Window

In the Stateflow Breakpoints and Watch window, you can view current data and message values while the simulation is paused at a breakpoint. To open the Breakpoints and Watch window, on the **Debug** tab, click **Breakpoints List**. Alternatively, open the Breakpoints dialog box and click the **Breakpoints List** link.

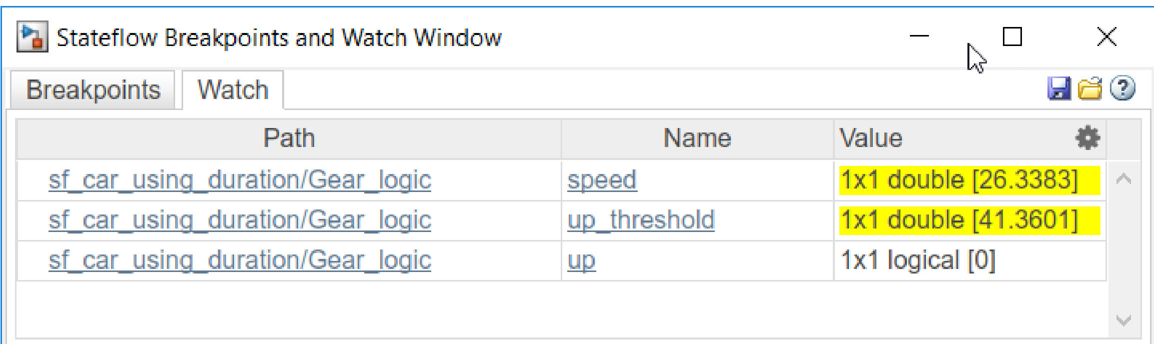
- To see a list of all of the breakpoints and their associated conditions, select the **Breakpoints** tab. For more information, see "Manage Breakpoints Through the Breakpoints and Watch Window" on page 30-6.
- To inspect data and message values, select the **Watch** tab.

Track Data in the Watch List

You can use the Breakpoints and Watch window to:

- Add data and message objects to a watch list.
- Track the values that changed since the last time step.
- Expand a message to view the message queue and message data values.

For example, you can add `speed`, `up_threshold`, and `up` to the watch list and track their values as you step through the simulation. The highlighting indicates that the values of `speed` and `up_threshold` changed during the last time step.

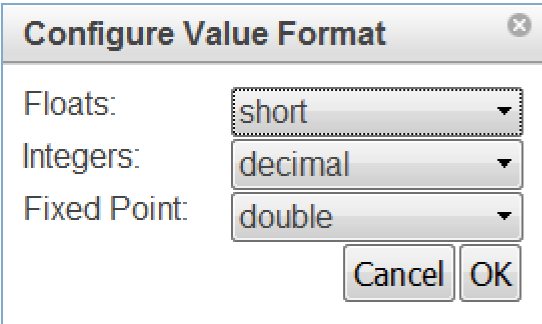


To add a data or message object to the watch list, open the **Property Inspector** or the Model Explorer. Select the data or message object you want to watch and click the **Add to Watch Window** link.

Alternatively, in the Stateflow Editor, right-click a state or transition that uses the data or message. Select **Add to Watch Window** and choose the variable name from the drop-down list.

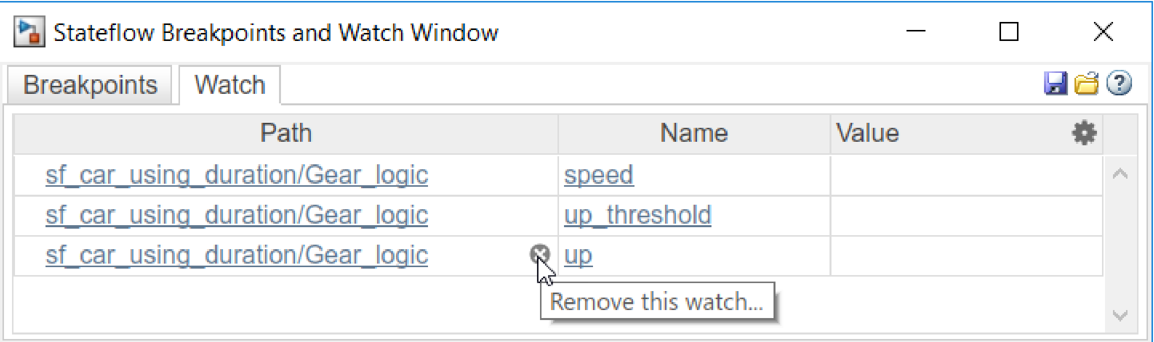
Format Watch Display

To change the format used to display watch data, select the gear icon at the top of the window. Use the drop-down lists to choose a MATLAB format for each data type.



Remove Data from the Watch List

To remove a data or message object from the watch list, point to the path for the watch data and click the **Remove this watch** icon that appears to the left of the variable name.



Save and Restore Watch Data

Watch data persists during a MATLAB session. When you close a model, its watch data list remains in the Breakpoints and Watch Window. If you reopen a model during the same MATLAB session, the watch data list for that model is restored.

You can save the breakpoint and watch data lists and reload them in a later MATLAB session. To save a snapshot of the breakpoint and watch data lists, at the top of the Breakpoints and Watch Window, click the **Save current breakpoints and watches** icon. To restore a snapshot, click the **Load breakpoints and watches** icon.

View and Modify Data in the MATLAB Command Window

While the simulation is paused at a breakpoint, the MATLAB command prompt changes to `debug>>`. At this prompt, you can inspect and change the values of Stateflow data, send local and output messages, and interact with the MATLAB workspace.

For example, suppose that the previous chart has reached a breakpoint. To view the data that is visible at the current scope, use the `whos` command.

```
whos
```

```
  Name                  Size            Bytes  Class       Attributes

  TWAIT                 1x1                 1  uint8
  down                  1x1                 1  logical
  down_th               1x1                 8  double
  down_threshold        1x1                 8  double
  gear                  1x1                 4  gearType
  speed                 1x1                 8  double
  throttle              1x1                 8  double
  up                    1x1                 1  logical
  up_th                 1x1                 8  double
  up_threshold          1x1                 8  double
```

To inspect the values of `speed` and `up_threshold`, enter:

```
speed
```

```
speed =

   26.3383
```

```
up_threshold
```

```
up_threshold =

   41.3601
```

Modify Data by Using the Debugging Prompt

At the debugging prompt, you can change the value of data store memory, local, and output data. For example, in the previous chart, you can change the value of `up_threshold`, `up`, and `gear`:

```
up_threshold = 25;
```

```
up = true;
```

```
gear = gearType.third;
```

Follow these rules when modifying data at the debugging prompt.

- To modify vectors and matrices, use MATLAB syntax for indexing, regardless of the action language property of your chart. See “Indexing Notation” on page 19-4.

  For example, to change the element in the diagonal of a 2-by-2 matrix `u`, enter:

  ```
  u(1,1) = 6.022e23;
  u(2,2) = 6.626e-34
  ```
- You can change the dimensions of variable-size data as long as the new size is within the dimension bounds specified for the data. For example, suppose that `v` is a variable-size array with a maximum size of `[16 16]`. To change the value of `v` to a 5-by-7 array of ones, enter:

  ```
  v = ones(5,7);
  ```
- To modify enumerated data, explicitly specify the enumerated type by using prefixed identifiers. See “Notation for Enumerated Values” on page 20-3.

  For example, suppose that `w` has an enumerated data type `Colors`. To change the value of `w` to the enumerated value `Red`, enter:

```
w = Colors.Red
```

- To modify numerical data, cast to an explicit data type by using a MATLAB type conversion function. Explicit casting is not required for data of a type `double`. See “Type Cast Operations” on page 14-7.

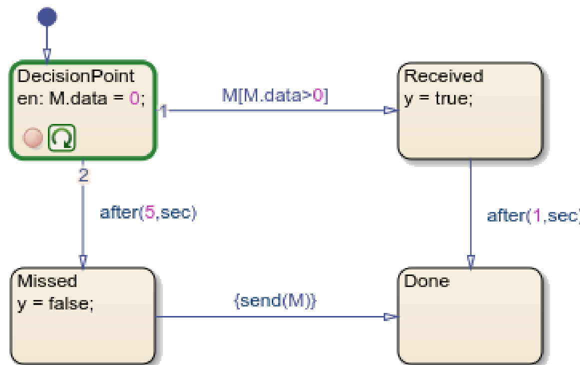
  For example, suppose that `x` has type `single`, `y` has type `int32`, and `z` has type `fixdt(1,16,12)`. To change the value of these data objects, enter:

  ```
  x = single(98.6);
  y = int32(100);
  z = fi(0.5413,1,16,12);
  ```

- You cannot change the values of constants, parameters, or input data at the debugging prompt.

Send Messages by Using the Debugging Prompt

At the debugging prompt, you can send local and output messages. For example, in this chart, the local message `M` determines which state becomes active after the state `DecisionPoint`. If the chart receives the message `M` with a positive value, the state `Received` becomes active and the chart outputs a value of `true`. Otherwise, the state `Missed` becomes active and the chart outputs a value of `false` .



The initial value of the message is zero. To change the value of the data field to a positive number and send the message to its local queue, enter:

```
M = 5;
send(M);
```

When you advance to the next step of the simulation, the message triggers the transition to the state `Received`. For more information, see “Control Chart Execution After a Breakpoint” on page 30-15.

Follow these rules when sending messages from the debugging prompt:

- To read or write to the message data field of a valid message, use the name of the message object. Do not use dot notation syntax.
- You can send a message from the debugging prompt only when the chart explicitly sends the message by calling the `send` operator.
- You cannot send input messages from the debugging prompt.

For more information, see “Control Message Activity in Stateflow Charts” on page 13-9.

Access the MATLAB Workspace While Debugging

You can enter other MATLAB commands at the debugging prompt, but the results are executed in the Stateflow workspace. For example, you can save all of the chart variables in a MAT-file by using the `save` function:

```
save(chartVars)
```

To enter a command in the MATLAB base workspace, use the `evalin` command with the first argument `"base"`. For example, to list the variables in the MATLAB workspace, use the command:

```
evalin("base","whos")
```

See Also

`whos` | `save` | `evalin` | `send`

More About

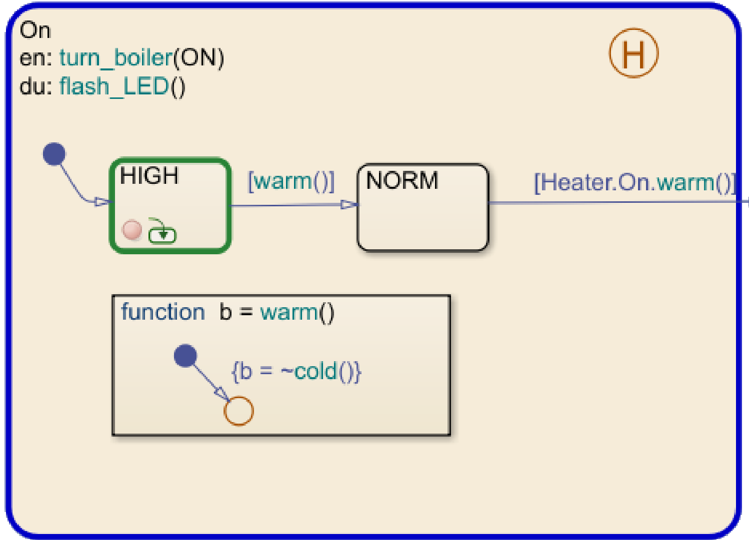
- "Set Breakpoints to Debug Charts" on page 30-2
- "Control Chart Execution After a Breakpoint" on page 30-15
- "Manage Symbols in the Stateflow Editor" on page 25-14

Control Chart Execution After a Breakpoint

When the simulation of a Stateflow chart pauses at a breakpoint, the chart enters debugging mode. You can examine the state of the chart and step through the simulation. For more information, see "Set Breakpoints to Debug Charts" on page 30-2.

Examine the State of the Chart

When a Stateflow chart enters debugging mode, the editor highlights the active elements in blue and the currently executing object in green. For example, this chart is paused at an `entry` breakpoint in the `HIGH` state. The active state (`On`) is highlighted in blue and the currently executing substate (`HIGH`) is highlighted in green.

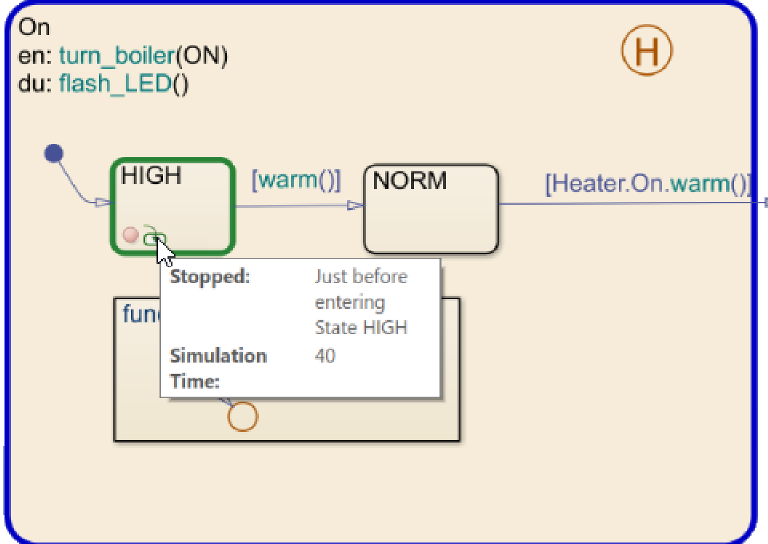


An execution status badge appears in the graphical object where the simulation is paused.

Badge	Description
	Simulation is paused before entering a chart or in a state `entry` action.
	Simulation is paused in a state `during` action, graphical function, or truth table function.
	Simulation is paused in a state `exit` action.
	Simulation is paused before testing a transition.
	Simulation is paused before taking a valid transition.

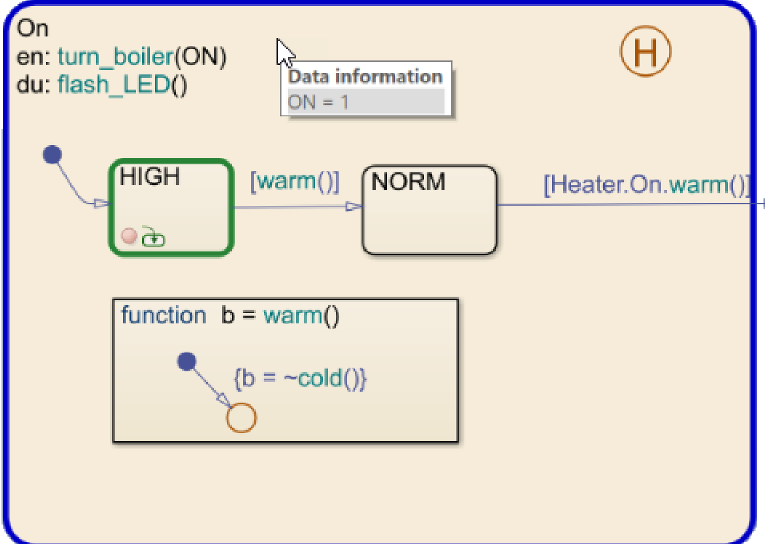
To see the execution status, point to the badge. A tooltip indicates:

- Where the simulation is paused
- The simulation time
- The current event (if the simulation is paused during a local or input event)



To view the values of chart, point to a chart object. A tooltip displays:

- The values of the data and messages that the selected object uses
- Temporal information (if the object contains a temporal logic operator)



For more information, see “Inspect and Modify Data and Messages While Debugging” on page 30-8.

Step Through the Simulation

When the chart is paused at a breakpoint, you can continue the simulation by using:

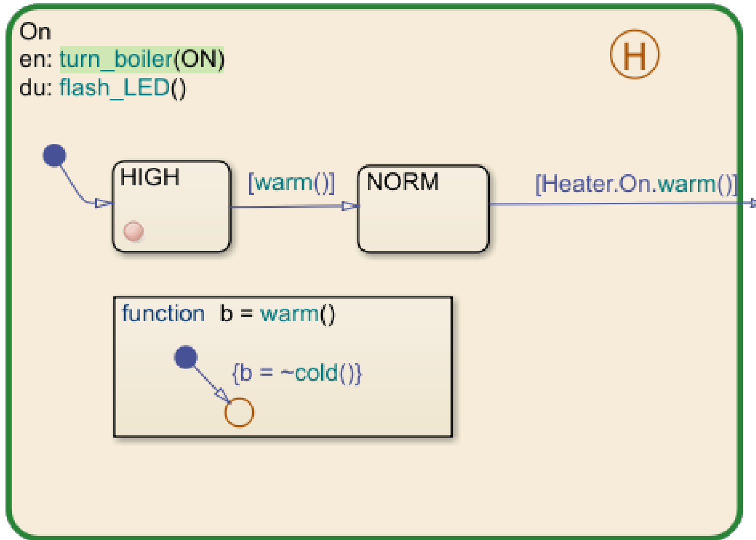
- Buttons in the **Debug** tab
- The MATLAB Command Window

- Keyboard shortcuts

Action	Debug Tab Button	MATLAB Command	Keyboard Shortcut	Description
Continue		`dbcont`	**Ctrl+T**	Continue simulation to the next breakpoint.
Step Forward				Exit debug mode and pause simulation before next time step.
Step Over		`dbstep`	**F10**	Advance to the next step in the chart execution. At the chart level, possible steps include: • Enter the chart • Test a transition • Execute a transition action • Activate a state • Execute a state action For more information, see "Execution of a Stateflow Chart" on page 2-12.
Step In		`dbstep in`	**F11**	From a state or transition action that calls a function, advance to the first executable statement in the function. From a statement in a function containing another function call, advance to the first executable statement in the second function. Otherwise, advance to the next step in the chart execution. (See **Step Over**.)
Step Out		`dbstep out`	**Shift+F11**	From a function call, return to the statement calling the function. Otherwise, continue simulation to the next breakpoint. (See **Continue**.)
Run to Cursor				In state or transition actions containing more than one statement, execute a group of statements together.

Action	Debug Tab Button	MATLAB Command	Keyboard Shortcut	Description
Stop		dbquit	**Ctrl+Shift+T**	Exit debug mode and stop simulation.

In state or transition actions that contain more than one statement, you can step through the individual statements one at a time by selecting **Step Over**. The Stateflow Editor highlights each statement before executing it.



To execute a group of statements together, click the last statement in the group and select **Run to Cursor**.

See Also

More About

- “Set Breakpoints to Debug Charts” on page 30-2
- “Inspect and Modify Data and Messages While Debugging” on page 30-8
- “Debug Run-Time Errors in a Chart” on page 30-19

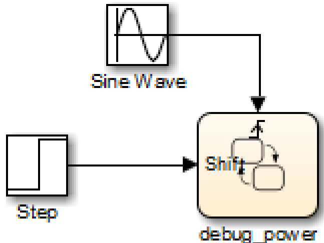
Debug Run-Time Errors in a Chart

In this section...
"Create the Model and the Stateflow Chart" on page 30-19
"Debug the Stateflow Chart" on page 30-20
"Correct the Run-Time Error" on page 30-20

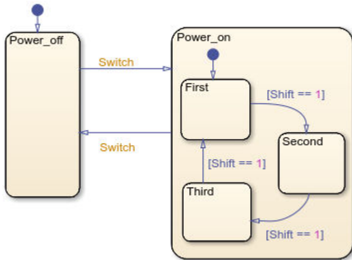
Create the Model and the Stateflow Chart

In this topic, you create a Simulink model with a Stateflow chart to debug. Follow these steps:

1. Create the following Simulink model:



2. Add the following states and transitions to your chart:



3. In your chart, add an event `Switch` with a scope of **Input from Simulink** and a **Rising** edge trigger.
4. Add a data `Shift` with a scope of **Input from Simulink**.

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

In the model, there is an event input and a data input. A Sine Wave block generates a repeating input event that corresponds with the Stateflow event `Switch`. The Step block generates a repeating pattern of 1 and 0 that corresponds with the Stateflow data object `Shift`. Ideally, the `Switch` event occurs at a frequency that allows at least one cycle through `First`, `Second`, and `Third`.

Debug the Stateflow Chart

To debug the chart in “Create the Model and the Stateflow Chart” on page 30-19, follow these steps:

1. Right-click in the chart, and select **Set Breakpoint on Chart Entry**.
2. Start the simulation.

   Because you specified a breakpoint on chart entry, execution stops at that point.
3. Click the Step In button, .

   The Step In button executes the next step and stops.
4. Continue clicking the Step In button and watching the animating chart.

   After each step, watch the chart animation to see the sequence of execution.

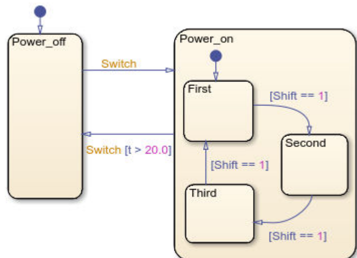
Single-stepping shows that the chart does not exhibit the desired behavior. The transitions from First to Second to Third inside the state Power_on are not occurring because the transition from Power_on to Power_off takes priority. The output display of code coverage also confirms this observation.

Correct the Run-Time Error

In “Debug the Stateflow Chart” on page 30-20, you step through a simulation of a chart and find an error: the event Switch drives the simulation but the simulation time passes too quickly for the input data object Shift to have an effect.

Correct this error as follows:

1. Stop the simulation so that you can edit the chart.
2. Add the condition [t > 20.0] to the transition from Power_on to Power_off.



   Now the transition from Power_on to Power_off does not occur until simulation time is greater than 20.0.
3. Begin simulation again.
4. Click the Step In button repeatedly to observe the new behavior.

Detect Modeling Errors During Edit Time

When edit-time checking is enabled, the Stateflow Editor detects potential errors and warnings as you work on your chart. By fixing these issues early in the design process, you can avoid compile-time or run-time warnings and errors.

The Stateflow Editor highlights objects that violate the edit-time checks in red (for errors) or orange (for warnings). When you point to an object that is highlighted and click the error or warning badge, a tooltip displays details and possible fixes.

Manage Edit-Time Checks

By default, edit-time checking and syntax error highlighting are enabled. To disable the edit-time checks, in the **Debug** tab, clear the **Diagnostics** > **Edit-Time Errors & Warnings** check box. Edit-time checks can also be disabled by using `edittime.setDisplayIssues` (Simulink).

This table lists edit-time checks that have an associated diagnostic configuration parameter on the **Diagnostics** > **Stateflow** pane of the Configuration Parameters dialog box.

Edit-Time Check Issue	Diagnostic Configuration Parameter
"Dangling transition" on page 30-24	"Unreachable execution path" (Simulink)
"Default transition path does not terminate in a state" on page 30-24	"No unconditional default transitions" (Simulink)
"Transition action precedes a condition action along this path" on page 30-25	"Transition action specified before condition action" (Simulink)
"Transition loops outside natural parent" on page 30-27	"Transition outside natural parent" (Simulink)
"Transition shadowing" on page 30-27	"Unreachable execution path" (Simulink)
"Unconditional path out of state with during actions or child states" on page 30-27	"Transition outside natural parent" (Simulink)
"Unexpected backtracking" on page 30-29	"Unexpected backtracking" (Simulink)
"Unreachable junction" on page 30-30	"Unreachable execution path" (Simulink)
"Unreachable port or junction" on page 30-33	"Unreachable execution path" (Simulink)
"Unreachable state" on page 30-23	"Unreachable execution path" (Simulink)

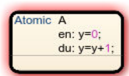
You can control the level of diagnostic action for these edit-time checks by setting the value of their configuration parameter to `error`, `warning`, or `none`. When you change the setting for a configuration parameter, the diagnostic level for the corresponding edit-time checks also changes. For example, if you set the **Unreachable execution path** configuration parameter to `none`, then the Stateflow Editor does not highlight dangling transitions, transition shadowing, or unreachable states.

Edit-Time Checks on States and Subcharts

Atomic subchart contains state actions

- **Issue:** State actions are not supported on atomic subcharts.

- **Diagnostic level:** Error.
- **Solution:** Delete the state actions or move them to a substate of the atomic subchart.

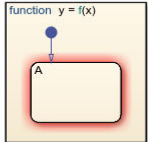


Default transition is missing

- **Issue:** In a chart or state with exclusive (OR) decomposition and at least two substates or junctions, a default transition is required to indicate where the execution begins.
- **Diagnostic level:** Error.
- **Solution:** Specify an initial state by adding a default transition. For more information, see “Use Default Transitions to Specify Initial Substate Activity” on page 1-44.

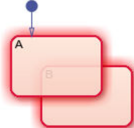
Graphical function contains a state

- **Issue:** Because graphical functions execute completely in a single time step, they must not contain any states.
- **Diagnostic level:** Error.
- **Solution:** Replace the states with junctions. For more information, see “Reuse Logic Patterns by Defining Graphical Functions” on page 6-9.



Invalid intersection

- **Issue:** States and junctions must not overlap in the Stateflow Editor.
- **Diagnostic level:** Error.
- **Solution:** Avoid intersections by separating the states and junctions.

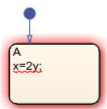


Monitoring leaf or child state activity of parallel states

- **Issue:** Charts and states with parallel (AND) decomposition do not support monitoring of leaf or child state activity because parallel substates are active simultaneously.
- **Diagnostic level:** Warning.
- **Solution:** Open the **Property Inspector** or the Model Explorer. Clear the **Create output for monitoring** check box or select `Self activity` from the drop-down list. For more information, see "Monitor State Activity Through Active State Data" on page 11-2.

State contains a syntax error

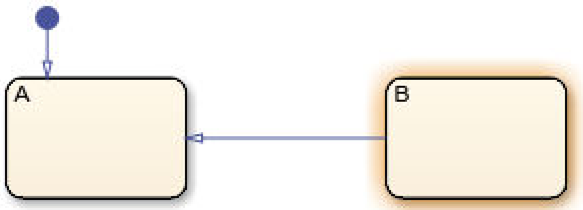
- **Issue:** A state action does not follow the Stateflow syntax rules. The Stateflow Editor underlines syntax errors with a red, wavy line. See also `Transition Contains a Syntax Error` on page 30-25.
- **Diagnostic level:** Error.
- **Solution:** Correct the syntax error in the state action. For more information, see "Define Actions in a State" on page 1-27.



Note In the parent chart, subcharts with syntax errors are highlighted in red and an error badge indicates the syntax issue. In the subchart editor, the syntax error is underlined in red, but there is no badge to indicate the issue.

Unreachable state

- **Issue:** A state is unreachable when no valid execution path leads to it.
- **Diagnostic level:** Depends on the configuration parameter "Unreachable execution path" (Simulink).
- **Solution:** Connect the unreachable state with a transition from a reachable source.



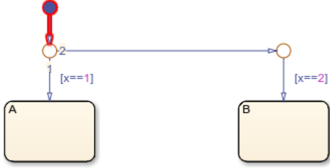
Edit-Time Checks on Transitions

Dangling transition

- **Issue:** Every transition must have a valid destination.
- **Diagnostic level:** Depends on the configuration parameter “Unreachable execution path” (Simulink).
- **Solution:** Connect the transition to a state, junction, or port. For more information, see “Transition Between Operating Modes” on page 1-37.

Default transition path does not terminate in a state

- **Issue:** In charts or states with exclusive (OR) decomposition and at least one substate:
  - Every branch of the default transition path must lead to a substate.
  - There must be a branch of the default transition path that is not guarded by a condition or triggered by an event.
- **Diagnostic level:** Depends on the configuration parameter “No unconditional default transitions” (Simulink).
- **Solution:** Terminate every branch of the default transition path in a substate. Ensure that one branch of the default transition path is not guarded by a condition or triggered by an event.



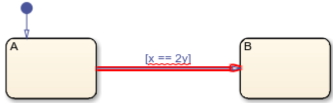
Invalid default transition path

- **Issue:** A default transition path must not exit the parent state.
- **Diagnostic level:** Error.

- **Solution:** Modify the default transition path so it stays within the parent state.

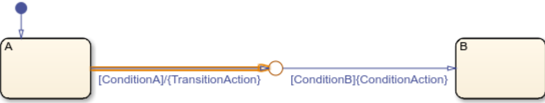
Transition contains a syntax error

- **Issue:** In a transition, a condition or action does not follow the Stateflow syntax rules. The Stateflow Editor underlines syntax errors with a red, wavy line. See also `State Contains a Syntax Error`. on page 30-23
- **Diagnostic level:** Error.
- **Solution:** Correct the syntax error in the transition condition or action. For more information, see “Define Actions in a Transition” on page 1-39.



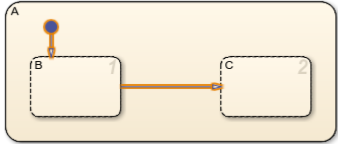
Transition action precedes a condition action along this path

- **Issue:** When a transition with a transition action is followed by a transition with a condition action, the actions are not executed in the order of the transitions. Stateflow charts execute condition actions when the associated condition is evaluated as true. In contrast, charts execute transition actions only when the transition path is fully executed. As a consequence, a chart takes a transition path, the condition actions occur before the transition actions.
- **Diagnostic level:** Depends on the configuration parameter “Transition action specified before condition action” (Simulink).
- **Solution:** Place the transition action after the last condition action on the path.



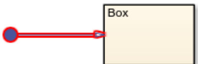
Transition begins or ends in a parallel state

- **Issue:** In charts and states with parallel (AND) decomposition, all sibling substates are active or inactive at the same time.
- **Diagnostic level:** Warning.
- **Solution:** Remove the transitions or change the decomposition of the parent state to exclusive (OR).



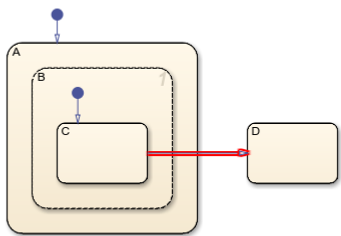
Transition connects to a box

- **Issue:** Transitions must connect only to states and junctions.
- **Diagnostic level:** Error.
- **Solution:** Move or delete the transitions attached to the box.



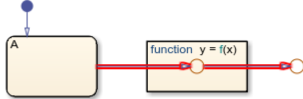
Transition crosses parallel states

- **Issue:** Standalone charts in MATLAB do not support transitions that cross the boundary of a parallel state.
- **Diagnostic level:** Error.
- **Solution:** Delete the transition crossing into or out of the parallel states.



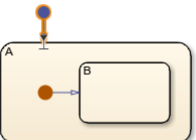
Transition enters or exits graphical function

- **Issue:** Transitions must not enter or exit a graphical function. Flow charts in graphical functions must be completely contained inside the function.
- **Diagnostic level:** Error.
- **Solution:** Delete the transition entering or exiting the graphical function.



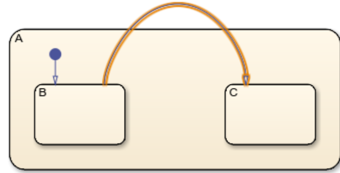
Transition is not connected to entry/exit port

- **Issue:** Transition is not connected to the entry or exit port near its source or destination.
- **Diagnostic level:** Warning.
- **Solution:** Connect the transition to the port or move the transition source or destination to a different location.



Transition loops outside natural parent

- **Issue:** If a transition goes outside the parent state between the source and destination, the chart executes the `exit` and `entry` actions of the parent state before the destination state becomes active.
- **Diagnostic level:** Depends on the configuration parameter “Transition outside natural parent” (Simulink).
- **Solution:** Move the transition so that it is contained within the parent state.



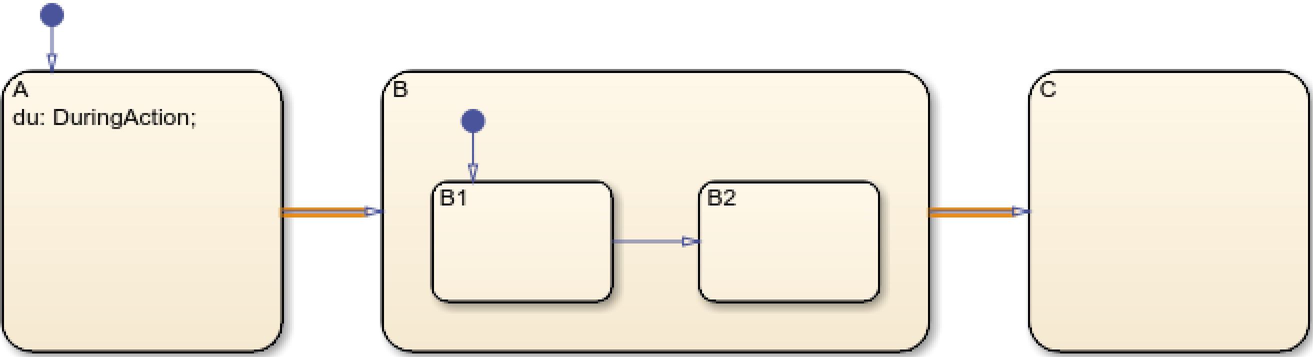
Transition shadowing

- **Issue:** When an unconditional transition executes before other outgoing transitions from the same source, it prevents the other transitions from executing.
- **Diagnostic level:** Depends on the configuration parameter “Unreachable execution path” (Simulink).
- **Solution:** Create no more than one unconditional transition from each state or junction. Explicitly specify that the unconditional transition executes after any transitions with conditions. For more information, see “Transition Evaluation Order” on page 2-28.

Unconditional path out of state with during actions or child states

- **Issue:** Unconditional transitions leading out of a state prevent the execution of the `during` actions in the state and the transitions between child states.
- **Diagnostic level:** Depends on the configuration parameter “Transition outside natural parent” (Simulink).

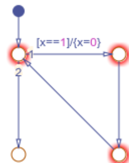
- **Solution:** Add a condition to the transition or remove during actions and child states from the state.



Edit-Time Checks on Junctions

Cycle contains transitions with transition actions

- **Issue:** Cycles should not contain transitions with transition actions.
- **Diagnostic level:** Error.
- **Solution:** Remove the transition action or remove the cycle by deleting a transition.

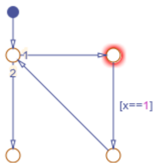


Invalid history junction

- **Issue:** A history junction is invalid when:
  - The history junction is contained in the chart level of the hierarchy.
  - The history junction is contained in a state with parallel (AND) decomposition.
  - The history junction is contained inside a graphical function.
  - There are multiple history junctions contained in the same state.
  - The history junction is the source of a transition.
- **Diagnostic level:** Error.
- **Solution:** Remove the history junction from the chart level of the hierarchy, a state with parallel (AND) decomposition, or a graphical function. Remove all but one history junction from the state. Move the transition source to a connective junction or a state. For more information, see “Resume Prior Substate Activity by Using History Junctions” on page 1-58.

Junction has no unconditional escape from cycle

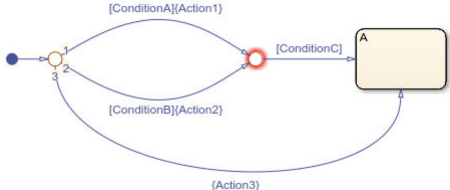
- **Issue:** A junction must have an unconditional escape path from a cycle to a state or terminating junction.
- **Diagnostic level:** Error.
- **Solution:** Create an unconditional path from the junction to a state or terminating junction.



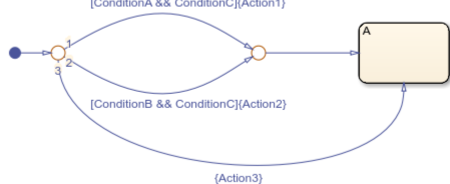
Unexpected backtracking

- **Issue:** Unexpected backtracking of control flow can occur when multiple transition paths from the same source lead to a junction and the junction does not have an unconditional path to a state or terminating junction.
- **Diagnostic level:** Depends on the configuration parameter "Unexpected backtracking" (Simulink).
- **Solution:** Create an unconditional path from the junction to a state or terminating junction. For more information, see "Backtrack in Flow Charts" on page A-28.

  For example, the highlighted junction in this chart does not have an unconditional path to state `A`. If `ConditionA` and `ConditionB` are true and `ConditionC` is false, the chart backtracks to the first junction in the path multiple times. As a result, the chart executes the three condition actions.



  To avoid backtracking, combine the conditions and create an unconditional path from the second junction to the destination state. After the change, the chart executes only one condition action.



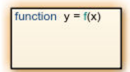
Unreachable junction

- **Issue:** A junction is unreachable when no valid execution path leads to it.
- **Diagnostic level:** Depends on the configuration parameter “Unreachable execution path” (Simulink).
- **Solution:** Connect the unreachable junction with a transition from a reachable source.

Edit-Time Checks on Functions

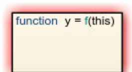
Function is unused

- **Issue:** A function is unused when a chart when there are no statements that call the function.
- **Diagnostic level:** Warning.
- **Solution:** Call the function from a state or transition action or from another function.



Invalid use of keywords as function arguments

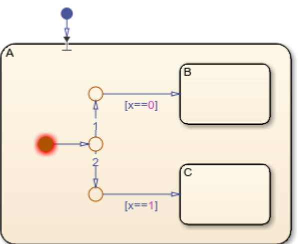
- **Issue:** A function definition uses a reserved keyword as an argument.
- **Diagnostic level:** Error.
- **Solution:** Rename the argument to the function. For a list of reserved keywords, see “Guidelines for Naming Stateflow Objects” on page 1-66.



Edit-Time Checks on Entry and Exit Ports

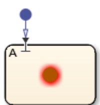
Entry junctions must have an unconditional path to a state

- **Issue:** An entry junction must have one transition path that is not guarded by a condition or triggered by an event.
- **Diagnostic level:** Error.
- **Solution:** Add an unconditional path from the entry junction to a state.



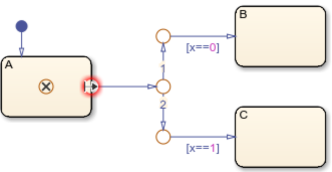
Entry junctions must have outgoing transitions

- **Issue:** An entry junction does not connect to an outgoing transition path.
- **Diagnostic level:** Error.
- **Solution:** Attach transitions to the entry junction or remove the junction.



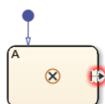
Exit ports must have an unconditional path to a state

- **Issue:** An exit port must have one transition path that is not guarded by a condition or triggered by an event.
- **Diagnostic level:** Error.
- **Solution:** Add an unconditional path from the exit port to a state.



Exit ports must have outgoing transitions

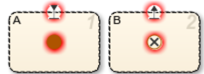
- **Issue:** An exit port does not connect to an outgoing transition path.
- **Diagnostic level:** Error.
- **Solution:** Attach transitions to the exit port or remove the port.



Invalid entry or exit junction

- **Issue:** Entry and exit junctions are supported only in exclusive (OR) states and atomic subcharts.

- **Diagnostic level:** Error.
- **Solution:** Move the junction to an exclusive (OR) state or atomic subchart or delete the junction.

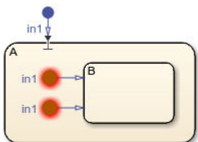


Missing entry or exit junction

- **Issue:** An entry or exit port must have a matching entry or exit junction.
- **Diagnostic level:** Error.
- **Solution:** Delete the port or create a matching junction with the same label.

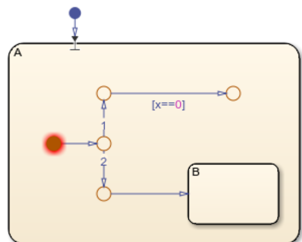
Multiple entry or exit junctions with same label

- **Issue:** Entry and exit junctions in the same parent must have unique labels.
- **Diagnostic level:** Error.
- **Solution:** Delete one of the junctions or change one of the labels.



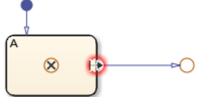
Path from entry junction contains a terminal junction

- **Issue:** Every path from an entry junction must lead to a state.
- **Diagnostic level:** Error.
- **Solution:** Replace the terminal junction with a state.



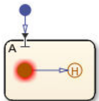
Path from exit port contains a terminal junction

- **Issue:** Every path from an exit port must lead to a state.
- **Diagnostic level:** Error.
- **Solution:** Replace the terminal junction with a state.



Transition path from an entry junction to a history junction

- **Issue:** Transition paths from entry junctions must not connect to history junctions.
- **Diagnostic level:** Error.
- **Solution:** Remove the path from the entry junction to the history junction.

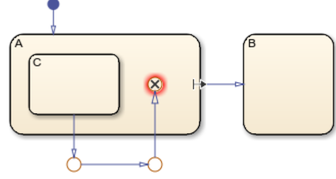


Transition path from an inner transition to an exit junction

- **Issue:** Inner transition paths must not connect to an exit junction.
- **Diagnostic level:** Error.
- **Solution:** Remove the path from the inner transition to the exit junction.

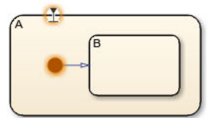
Transition path from entry junction or to exit junction must be contained in parent

- **Issue:** Transition paths that start at entry junctions or end at exit junctions must be contained in the parent state.
- **Diagnostic level:** Error.
- **Solution:** Modify the transition path to be contained in the parent state.



Unreachable port or junction

- **Issue:** A port or junction is unreachable when no valid execution path leads to it.
- **Diagnostic level:** Depends on the configuration parameter “Unreachable execution path” (Simulink).
- **Solution:** Connect the unreachable entry port or exit junction with a transition from a reachable source.



See Also

More About

- “Detect Common Modeling Errors During Simulation” on page 30-35
- “Modeling Guidelines for Stateflow Charts” on page 2-8
- “Stateflow Semantics” on page 2-2
- “Model Configuration Parameters: Stateflow Diagnostics” (Simulink)

Detect Common Modeling Errors During Simulation

To avoid common design errors, you can run diagnostic checks that test the completeness of your Stateflow chart during compilation and simulation. Stateflow diagnostics detect state inconsistencies, violations in data ranges, and cyclic behavior in Stateflow charts in Simulink models.

When you simulate a model, the Stateflow parser evaluates the graphical and nongraphical objects and data in each Stateflow machine against the supported chart notation and the action language syntax. You can also check the syntax of your chart by selecting **Update Chart** in the **Modeling** tab of the Stateflow Toolstrip.

If syntax errors exist in your chart, the chart automatically appears with the highlighted object that causes the first error. You can select the error in the diagnostic window to bring its source chart to the front with the source object highlighted. Any unresolved data or events in the chart are flagged in the Symbol Wizard.

Tip While you edit your chart, the Stateflow Editor displays potential causes for errors by highlighting objects in red or orange. For more information, see “Detect Modeling Errors During Edit Time” on page 30-21.

Detect State Inconsistencies

In a Stateflow chart, states are inconsistent if they violate one of these rules:

- An active state with exclusive (OR) decomposition and at least one substate has exactly one active substate.
- All substates of an active state with parallel (AND) decomposition are active.
- All substates of an inactive state are inactive regardless of the state decomposition.

For example, this chart has a state inconsistency because there is no default transition to indicate which substate becomes active first.

Adding an unconditional default transition to one of the states resolves the state inconsistency.

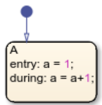
At compile time, Stateflow charts detect state inconsistencies caused by the omission of an unconditional default transition. To control the level of diagnostic action, open the Configuration Parameters dialog box and, in the **Diagnostics** > **Stateflow** pane, set the **No unconditional default transitions** parameter to `error`, `warning`, or `none`. The default setting is `error`. For more information, see “No unconditional default transitions” (Simulink).

Detect Data Range Violations

During simulation, a data range violation occurs when:

- An integer or fixed-point operation overflows the numeric capacity of its result type. See “Handle Integer Overflow for Chart Data” on page 10-37 and “Fixed-Point Operations in Stateflow Charts” on page 23-19.
- The value of a data object is outside the range of the values specified by the **Initial value**, **Minimum**, and **Maximum** properties. See “Initial value” on page 10-8 and “Limit range” on page 10-10.

For example, this chart contains local data `a` that has a **Minimum** value of 0 and a **Maximum** value of 2. The `entry` action in state `A` initializes `a` to 1. The `during` action increments the value of `a` by 1. After two time steps, the value of `a` exceeds its specified range, resulting in a data range violation.



At run time, Stateflow charts detect data range violations. To control the level of diagnostic action, open the Configuration Parameters dialog box and, in the **Diagnostics** > **Data Validity** pane, set these parameters to `error`, `warning`, or `none`:

- **Simulation range checking** detects violations based on minimum-and-maximum range checks. The default setting is `none`.
- **Wrap on overflow** and **Saturate on overflow** detect violations that occur when integer or fixed-point operations exceed the numeric capacity of their result type. The default setting is `warning`.

For more information, see “Simulation range checking” (Simulink), “Wrap on overflow” (Simulink), and “Saturate on overflow” (Simulink).

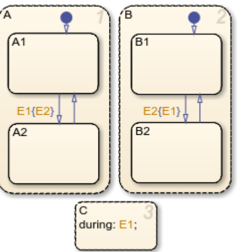
Detect Cyclic Behavior

Cyclic behavior occurs when a step or sequence of steps is repeated indefinitely during chart simulation.

For example, the actions in this chart produce an infinite cycle of recursive event broadcasts.

- The `during` action in state `C` broadcasts the event `E1`.
- The event `E1` triggers the transition from state `A.A1` to state `A.A2`. The condition action for this transition broadcasts the event `E2`.
- The event `E2` triggers the transition from state `B.B1` to state `B.B2`. The condition action for this transition broadcasts the event `E1`.

The event broadcasts in states `A` and `B` occur in condition actions, so the transitions do not take place until the chart processes the resulting events. The substates `A.A1` and `B.B1` remain active, so new event broadcasts continue to trigger the transitions and the process repeats indefinitely.



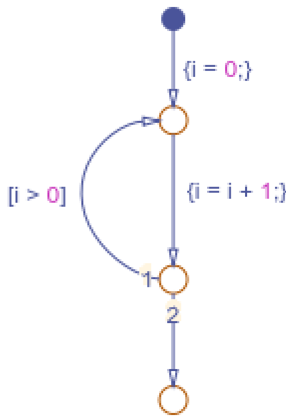
During chart simulation, Stateflow charts use cycle detection algorithms to detect a class of infinite recursions caused by event broadcasts. To enable cycle detection, open your Stateflow chart. In the **Debug** tab, select **Diagnostics** > **Detect Cyclical Behavior**. Cyclic behavior checking is selected by default.

Stateflow charts also detect undirected local event broadcasts. To control the level of diagnostic action, open the Configuration Parameters dialog box and, in the **Diagnostics** > **Stateflow** pane, set the **Undirected event broadcasts** parameter to `error`, `warning`, or `none`. The default setting is `warning`. For more information, see "Undirected event broadcasts" (Simulink).

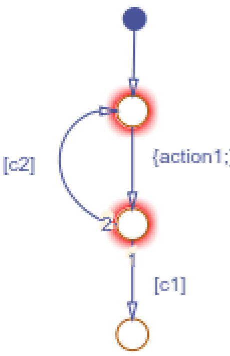
Tip Because undirected local event broadcasts can cause unwanted recursive behavior, use of the `send` operator to broadcast directed local events is recommended. For more information, see "Avoid Unwanted Recursion in a Chart" on page 30-43.

Fix Cyclic Behavior in Flow Charts

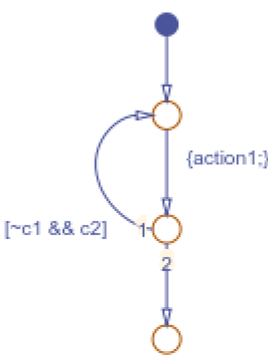
Run-time detection of cyclical behavior is limited to cases of recursion due to event broadcasts and does not extend to other types of cyclic behavior. For instance, cycle detection does not detect the infinite cycle in this flow chart. In this example, the default transition initializes the local data `i` to 0. The next transition segment increments `i`. The transition back to the first junction is valid when `i` is positive. Because this condition is always true, an infinite cycle results.



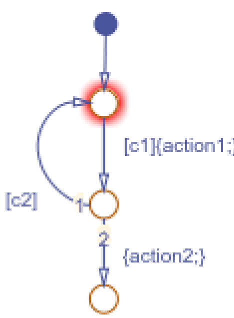
In many cases, cyclical behavior in flow charts are caused by junctions that do not have an unconditional escape path from a cycle. The Stateflow editor highlights these junctions in red. For example, in this flow chart, the top two junctions in the loop do not have an unconditional path to a terminal junction.



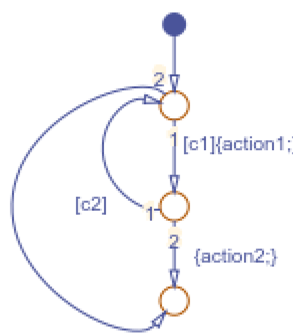
To resolve this issue, incorporate the negation of condition `c1` into the transition containing `c2`, so the resulting diagram has an unconditional path out of the loop.



Similarly, the top junction in this flow chart does not have an unconditional path to a terminal junction.



To resolve this issue, create an unconditional transition from the top junction to the terminal junction.



See Also

More About

- “Detect Modeling Errors During Edit Time” on page 30-21
- “Avoid Unwanted Recursion in a Chart” on page 30-43
- “Model Configuration Parameters: Stateflow Diagnostics” (Simulink)

Animate Stateflow Charts

Set Animation Speeds

During simulation, animation provides visual verification that your chart behaves as you expect. Animation highlights active objects in a chart as execution progresses. For charts in a Simulink model, you can control the speed of chart animation during simulation, or turn off animation. In the Stateflow Editor, in the **Debug** tab, under **Animation Speed**:

- `Lightning Fast`
- `Fast`
- `Medium`
- `Slow`
- `None`

`Lightning Fast` animation provides the fastest simulation speed by buffering the highlights. During `Lightning Fast` animation, the more recently highlighted objects are in a bolder, lighter blue. These highlights fade away as simulation time progresses.

The default animation speed, `Fast`, shows the active highlights at each time step. To add a delay with each time step, set the animation speed to `Medium` or `Slow`.

Maintain Highlighting

To maintain highlighting of active states in the chart after simulation ends, in the **Debug** tab, select **Maintain highlighting of active states after simulation ends** .

Disable Animation

Animation is enabled by default in Stateflow charts. To turn off animation for a chart, in the **Debug** tab, select **Remove animation highlighting** .

Animate Charts as Generated Code Executes on a Target System

If you have Simulink Coder, you can use external mode to establish communication between a Simulink model and generated code downloaded to and executing on a target system. Stateflow software can use the external mode communication channel to animate chart states. Also, you can designate chart data of local scope to be test points and view the test point data in floating scopes and signal viewers.

For more information, see:

- “External Mode Simulation by Using XCP Communication” (Simulink Coder)
- “External Mode Simulation with TCP/IP or Serial Communication” (Simulink Coder)

.

Comment Out Objects in a Stateflow Chart

In this section...

"Comment Out a Stateflow Object" on page 30-41
"How Commenting Affects the Chart and Model" on page 30-41
"Add Text to a Commented Object" on page 30-42
"Limitations on Commenting Objects" on page 30-42

Comment Out a Stateflow Object

Commenting out a Stateflow object excludes it from simulation. To comment out a Stateflow object, right-click the object and select **Comment Out**. Use commenting to:

- Debug a chart by making minor changes between simulation runs.
- Test and verify the effects of objects on simulation results.
- Create incremental changes for rapid, iterative design.

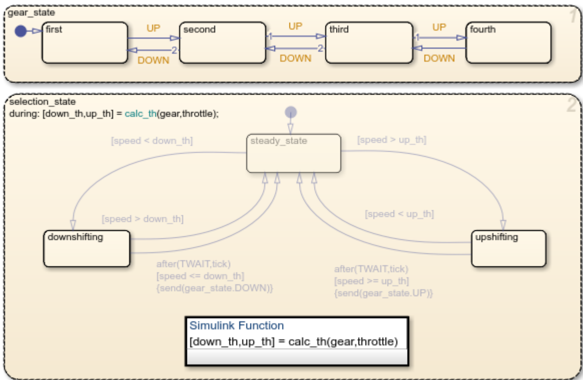
How Commenting Affects the Chart and Model

A commented object is not visible to the rest of the chart and the model. Commented objects in a chart are excluded from:

- Simulation
- Logging
- Code generation
- Animation
- Debugging
- Active state output

References to commented functions or states result in compile-time errors.

When you explicitly comment out a Stateflow object with **Comment Out**, the object appears gray with a badge . The software implicitly comments out some associated objects. For example, if you explicitly comment out a state or junction, all incoming and outgoing transitions are implicitly commented out. Implicitly commented objects also appear gray, but do not have a badge. For instance, in this chart, the state `steady_state` is explicitly commented. The transitions in and out of `steady_state` are implicitly commented.



To open this example, enter:

```
openExample("stateflow/AutomaticTransmissionWithActiveStateDataExample")
```

Explicitly Commented Stateflow Object	Implicit Results
States	All incoming and outgoing transitions, and child objects are implicitly commented out.
Transitions	None
Junctions	All connected transitions are implicitly commented out.
Functions	The software cannot invoke a commented function from any chart or model.
Data	You cannot explicitly comment out data. If you comment out the parent object, then the software cannot reference the data.
Events	You cannot explicitly comment out events. If you comment out the parent object, then the software cannot reference the event.

To uncomment an object, right-click the commented object and select **Uncomment**. All implicitly commented objects are restored as well. Implicitly commented objects cannot be uncommented directly.

Add Text to a Commented Object

Add a note to the commented object by clicking the badge %. Point to a badge to see the associated comments. You cannot add notes to implicitly commented objects.

Limitations on Commenting Objects

When you comment out an atomic subchart, the objects inside the chart do not appear implicitly commented. However, a commenting badge is displayed in the lower-left corner of the chart.

Avoid 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. This type of recursive behavior is desirable and efficient. For details, see "Broadcast Local Events to Synchronize Parallel States" on page 12-25.

However, unwanted recursion can also occur during chart execution. To avoid unwanted recursion, do not use recursive function calls or undirected local event broadcasts.

Recursive Function Calls

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.

Undirected Local Event Broadcasts

An undirected event broadcast sends a local event to all states in which it is visible. The format of an undirected event broadcast is

```
send(event_name)
```

where *`event_name`* is a local event.

To avoid recursive behavior, replace undirected event broadcasts with directed event broadcasts by using the syntax

```
send(event_name,state_name)
```

where *`event_name`* is a local event in the chart and *`state_name`* is a destination state.

- If the local event broadcast occurs in a state action, ensure that the destination state is not an ancestor of the source state in the chart hierarchy.
- If the local event broadcast occurs in a transition, ensure that:
  - The destination state is not an ancestor of the transition in the chart hierarchy.
  - The transition does not connect to the destination state.

For more information, see "Broadcast Local Events to Synchronize Parallel States" on page 12-25.

During simulation, Stateflow charts can detect undirected local event broadcasts. To control the level of diagnostic action, open the Configuration Parameters dialog box and, in the **Diagnostics > Stateflow** pane, set the **Undirected event broadcasts** parameter to `none`, `warning`, or `error`. The default setting is `warning`. For more information, see "Undirected event broadcasts" (Simulink).

See Also
`send`

More About

- “Broadcast Local Events to Synchronize Parallel States” on page 12-25
- “Detect Common Modeling Errors During Simulation” on page 30-35
- “Model Configuration Parameters: Stateflow Diagnostics” (Simulink)

Standalone Stateflow Charts for Execution in MATLAB

- “Create Stateflow Charts for Execution as MATLAB Objects” on page 31-2
- “Execute and Unit Test Stateflow Chart Objects” on page 31-8
- “Debug a Standalone Stateflow Chart” on page 31-13
- “Execute Stateflow Chart Objects Through Scripts and Models” on page 31-18
- “Design Human-Machine Interface Logic by Using Stateflow Charts” on page 31-24
- “Model a Communications Protocol by Using Chart Objects” on page 31-28
- “Implement a Financial Strategy by Using Stateflow” on page 31-32
- “Model a Fitness App by Using Standalone Charts” on page 31-35
- “Automate Control of Intelligent Vehicles by Using Stateflow Charts” on page 31-40
- “Model Bluetooth Low Energy Link Layer Using Stateflow” on page 31-44
- “Analog Triggered Data Acquisition Using Stateflow Charts” on page 31-47

Create Stateflow Charts for Execution as MATLAB Objects

To combine the advantages of state machine programming with the full functionality of MATLAB, create a standalone Stateflow chart outside of a Simulink model. Save the standalone chart with the extension `.sfx` and execute it as a MATLAB object. Refine your design by using chart animation and graphical debugging tools.

With standalone charts, you can create MATLAB applications such as:

- MATLAB App Designer user interfaces that use mode logic to manage the behavior of widgets. See "Design Human-Machine Interface Logic by Using Stateflow Charts" on page 31-24.
- Communication protocols and data stream processing applications that use sequential logic. See "Model a Communications Protocol by Using Chart Objects" on page 31-28.
- Data Acquisition Toolbox™ or Instrument Control Toolbox™ applications that use timer-based logic to monitor and control external tasks. See "Implement a Financial Strategy by Using Stateflow" on page 31-32.

These applications can be shared and executed without requiring a Stateflow license. For more information, see "Share Standalone Charts" on page 31-4.

Construct a Standalone Chart

To construct a standalone Stateflow chart, open the Stateflow Editor by using the `edit` function. For example, at the MATLAB Command Window, enter:

```
edit chart.sfx
```

If the file `chart.sfx` does not exist, the Stateflow Editor opens an empty chart with the name `chart`. Otherwise, the editor opens the chart defined by the `sfx` file.

In the Stateflow Editor, create a standalone chart by combining states, transitions, data, and other elements. For more information, see "Construct and Run a Stateflow Chart".

After you save the standalone chart, the `help` function displays information about executing it in MATLAB:

```
help chart.sfx
```

To close the standalone chart from the MATLAB Command Window, use the `sfclose` function:

```
sfclose chart
```

Create a Stateflow Chart Object

To execute a standalone chart in MATLAB, first create a Stateflow chart object. Use the name of the `sfx` file for the standalone chart as a function. Specify the initial values of data as name-value pairs. For example, suppose that you defined a standalone chart with data objects called `data1` and `data2`. Then this command creates the chart object `chartObject`, initializes `data1` and `data2`, and executes its default transition:

```
chartObject = chart(data1=value1,data2=value2)
```

To display chart information, such as the syntax for execution, the values of the chart data, and the list of active states, use the `disp` function:

```
disp(chartObject)
```

Execute a Standalone Chart

After you define a Stateflow chart object, you can execute the standalone chart by calling the `step` function (with data values, if necessary):

```
step(chartObject,data1=value1,data2=value2)
```

Alternatively, you can call one of the input event functions:

```
event_name(chartObject,data1=value1,data2=value2)
```

In either case, the values are assigned to local data before the chart executes.

If your chart has graphical or MATLAB functions, you can call them directly in the MATLAB Command Window. Calling a chart function does not execute the standalone chart.

```
function_name(chartObject,u1,u2)
```

Note If you use `nargin` in a graphical or MATLAB function in your chart, `nargin` counts the chart object as one of the input arguments. The value of `nargin` is the same whether you call the function from the chart or from the MATLAB Command Window.

You can execute a standalone chart without opening the Stateflow Editor. If the chart is open, then the Stateflow Editor highlights active states and transitions through chart animation.

For the purposes of debugging and unit testing, you can execute a standalone chart directly from the Stateflow Editor. During execution, you enter data values and broadcast events from the user interface. For more information, see “Execute and Unit Test Stateflow Chart Objects” on page 31-8.

You can execute a standalone chart from a MATLAB script, a Simulink model, or an App Designer user interface. For more information, see:

- “Execute Stateflow Chart Objects Through Scripts and Models” on page 31-18
- “Design Human-Machine Interface Logic by Using Stateflow Charts” on page 31-24

Stop Chart Execution

To stop executing a chart, destroy the chart object by calling the `delete` function:

```
delete(chartObject)
```

After the chart object is deleted, any handles to the chart object remain in the workspace, but become invalid. To remove the invalid handle from the workspace, use the command `clear`:

```
clear chartObject
```

If you clear a valid chart object handle and there are other handles to the same chart object, the chart object is not destroyed. For example, when you are executing a chart, the Stateflow Editor

contains internal handles to the chart object. Clearing the chart object handle from the workspace does not destroy the chart object or remove the chart animation highlighting in the editor. To reset the animation highlighting, right-click the chart canvas and select **Remove Highlighting**.

Share Standalone Charts

You can share standalone charts with collaborators who do not have a Stateflow license.

If your collaborators have the same or a later version of MATLAB than you have, they can execute your standalone charts as MATLAB objects without opening the Stateflow Editor. Chart animation and debugging are not supported. Run-time error messages do not link to the state or transition in the chart where the error occurs.

Note To run standalone charts that you saved in R2019a or R2019b, your collaborators must have the same version of MATLAB.

If your collaborators have an earlier version of MATLAB, export a standalone chart to a format that they can use. You can only export to R2019a and later releases. To complete the export process, you need access to the versions of Stateflow from which and to which you are exporting.

1. Using the later version of Stateflow, open the standalone chart.
2. On the **State Chart** tab, select **Save > Previous Version**.
3. In the Export to Previous Version dialog box, specify a file name for the exported chart.
4. From the **Save as type** list, select the earlier version to which to export the chart.
5. Click **Save**.
6. Using the earlier version of Stateflow, open and resave the exported chart.

To export a chart from the MATLAB Command Window, call the Stateflow function `exportToVersion`. For more information, see “Export Chart to an Earlier Version of MATLAB”.

Note Attempting to execute an exported chart before resaving it will result in an error.

Properties and Functions of Stateflow Chart Objects

A Stateflow chart object encapsulates data and operations in a single structure by providing:

- Private properties that contain the internal state variables for the standalone chart.
- A `step` function that calls the various operations implementing the chart semantics.

A chart object can have other properties and functions that correspond to the various elements present in the chart.

Standalone Chart Elements	Chart Object Elements
Local and constant data	Public properties
Input events	Functions that execute the chart

Standalone Chart Elements	Chart Object Elements
Graphical and MATLAB functions	Functions that you can call from the MATLAB Command Window

Chart Object Configuration Options

When you create a chart object, you can specify chart behavior by including these configuration options as name-value pairs.

Configuration Option	Description	Example
`-animationDelay`	Specify the delay that the chart animation uses to highlight each transition segment. The default value is `0.01` seconds. To produce a chart with no animation delays, set to zero.	Create a chart object that has slow animation by specifying one-second delays. `chartObject = chart('-animationDelay',1)`
`-enableAnimation`	Enable chart animation and debugging instrumentation. The default value is `true`.	Create a chart object that has animation and debugging instrumentation disabled. `chartObject = chart('-enableAnimation',false`
`-eventQueueSize`	Specify the size of the queue used for events and temporal logic operations. The default value is `20`. To disable the queuing of events, set to zero. For more information, see "Events in Standalone Charts" on page 2-42.	Create a chart object that ignores all events without warning if they occur when the chart is processing another operation. `chartObject = chart('-eventQueueSize',0)`
`-executeInitStep`	Enable the initial execution of default transitions. The default value is `true`.	Create a chart object but do not execute the default transition. `chartObject = chart('-executeInitStep',false`
`-warningOnUninitializedData`	Enable the warning about empty chart data after initializing the chart object. The default value is `true`.	Eliminate the warning when creating a chart object. `chartObject = chart('-warningOnUninitialized`

Initialization of Chart Data

In the Stateflow Editor, you can use the **Symbols** pane to specify initial values for chart data. When you create a chart object, chart data is initialized in alphabetical order according to its scope. Constant data is initialized first. Local data is initialized last.

If you use an expression to specify an initial value, then the chart attempts to resolve the expression by:

- Using the values of other data in the chart.
- Calling functions on the search path.

For example, suppose that you specify an initial value for the local data `x` by using the expression `y`. Then:

- If the chart has a constant called `y`, `y` is initialized before `x`. The local data `x` is assigned the same initial value as `y`.
- If the chart has a local data called `y`, `x` is initialized before `y`. The local data `x` is assigned to an empty array. If the configuration option `-warningOnUninitializedData` is set to `true`, a warning occurs.
- If the chart has no data named `y`, `x` is initialized by calling the function `y`. If the file `y.m` is not on the search path, this error occurs:

```
Undefined function or variable 'y'.
```

Stateflow does not search the MATLAB workspace to resolve initial values, so this error occurs even if there is a variable called `y` in the MATLAB workspace.

Capabilities and Limitations

Supported Features

- Classic chart semantics with MATLAB as the action language. You can use the full functionality of MATLAB, including those functions that are restricted for code generation in Simulink. See “Execute Stateflow Chart Objects Through Scripts and Models” on page 31-18.

Note In standalone Stateflow charts, the operating system command symbol `!` is not supported. To execute operating system commands, use the function `system`.

- Exclusive (OR) and Parallel (AND) state decomposition with hierarchy. See “Define Exclusive and Parallel Modes by Using State Decomposition” on page 1-35 and “Use State Hierarchy to Design Multilevel State Complexity” on page 1-33.
- Flow charts, graphical functions, and MATLAB functions. See “Reusable Components in Charts”.
- Conversion of MATLAB code to graphical functions by using the Pattern Wizard. See “Convert MATLAB Code into Stateflow Flow Charts” on page 3-17.
- Chart local and constant data without restriction to type. See “Execute and Unit Test Stateflow Chart Objects” on page 31-8.
- Input events. See “Design Human-Machine Interface Logic by Using Stateflow Charts” on page 31-24.
- Operators `hasChanged`, `hasChangedFrom`, and `hasChangedTo` that detect changes in the values of local data.

Note Standalone Stateflow charts do not support change detection on an element of a matrix or a field in a structure.

- Temporal logic operators:
  - `after`, `at`, and `every` operate on the number of input events, chart invocations (`tick`), and absolute time (`sec`). Use these operators in state `on` actions and as transition triggers.
  - `count` operates on the number of chart invocations (`tick`).
  - `temporalCount` operates on absolute time (`sec`, `msec`, and `usec`).
  - `elapsed` operates on absolute time (`sec`).

  Standalone charts define absolute-time temporal logic in terms of wall-clock time, which is limited to 1 millisecond precision.

- Function `getActiveStates` to access the states that are active during execution of the chart. To store the active states as a cell array, enter:

  ```
  states = getActiveStates(chartObject)
  ```
- Stateflow function `exportAsClass` that exports the standalone chart as the equivalent MATLAB class. Use this function to debug run-time errors that are otherwise difficult to diagnose. For example, suppose that you encounter an error while executing a Stateflow chart that controls a MATLAB application. If you export the chart as a MATLAB class file, you can replace the chart with the class in your application and diagnose the error by using the MATLAB debugger. To export the chart `chart.sfx` as a class file `chart.m`, enter:

  ```
  Stateflow.exportAsClass("chart.sfx")
  ```

  When you execute the MATLAB class, the Stateflow Editor does not animate the original chart.

Limitations

Content specific to Simulink:

- Sample time and continuous-time semantics.
- C action language.
- Simulink functions and Simulink subsystems as states.
- Input, output, and parameter data.
- Data store memory data.
- Output and local events.
- Input, output, and local messages.

Other limitations:

- No Mealy or Moore semantics.
- No State Transition Tables.
- No Truth Table functions.
- No state-parented local data or functions.
- No transition actions (actions that execute after the source state for the transition is exited but before the destination state is entered).

See Also

`disp` | `edit` | `exportAsClass` | `exportToVersion` | `help` | `sfclose`

More About

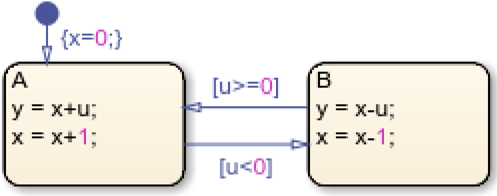
- “Execute and Unit Test Stateflow Chart Objects” on page 31-8
- “Execute Stateflow Chart Objects Through Scripts and Models” on page 31-18
- “Design Human-Machine Interface Logic by Using Stateflow Charts” on page 31-24
- “Implement a Financial Strategy by Using Stateflow” on page 31-32
- “Model a Communications Protocol by Using Chart Objects” on page 31-28

Execute and Unit Test Stateflow Chart Objects

A standalone Stateflow chart is a MATLAB class that defines the behavior of a finite state machine. Standalone charts implement classic chart semantics with MATLAB as the action language. You can program the chart by using the full functionality of MATLAB, including those functions that are restricted for code generation in Simulink. For more information, see “Create Stateflow Charts for Execution as MATLAB Objects” on page 31-2.

Example of a Standalone Stateflow Chart

The file `sf_chart.sfx` contains the standalone Stateflow chart `sf_chart`. The chart has local data `u`, `x`, and `y`.



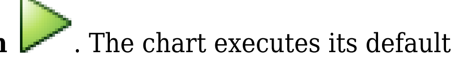
This example shows how to execute this chart from the Stateflow Editor and in the MATLAB Command Window.

Execute a Standalone Chart from the Stateflow Editor

To unit test and debug a standalone chart, you can execute the chart directly from the Stateflow Editor. During execution, you enter data values and broadcast events from the user interface.

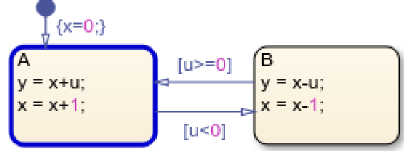
1 Open the chart in the Stateflow Editor:

```
edit sf_chart.sfx
```

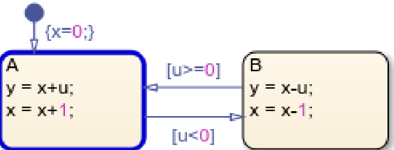
2 In the **Symbols** pane, enter a value of `u` = 1 and click **Run** . The chart executes its default transition and:

- Initializes `x` to the value of 0.
- Makes state `A` the active state.
- Assigns `y` to the value of 1.
- Increases the value of `x` to 1.

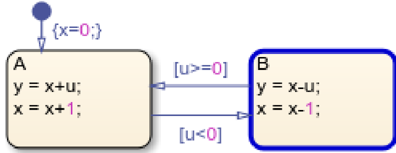
The chart animation highlights the active state `A`. The **Symbols** pane displays the values `u` = 1, `x` = 1, and `y` = 1. The chart maintains its current state and local data until the next execution command.



3 Click **Step**. Because the value of `u` does not satisfy the condition `[u<0]` to transition out of state `A`, this state remains active and the values of `x` and `y` increase to 2. The **Symbols** pane now displays the values `u` = 1, `x` = 2, and `y` = 2.

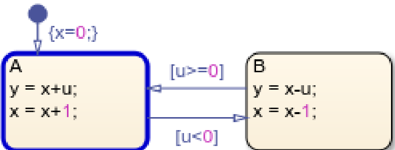


4 In the **Symbols** pane, enter a value of `u` = −1 and click **Step**. The negative data value triggers the transition to state `B`. The **Symbols** pane displays the values `u` = −1, `x` = 1, and `y` = 3.



5 You can modify the value of any chart data in the **Symbols** pane. For example, enter a value of `x` = 3. The chart will use this data value in the next time execution step.

6 Enter a value of `u` = 2 and click **Step**. The chart transitions back to state `A`. The **Symbols** pane displays the values `u` = 2, `x` = 4, and `y` = 5.



7 To stop the chart animation, click **Stop**.

To interrupt the execution and step through each action in the chart, add breakpoints before you execute the chart. For more information, see “Debug a Standalone Stateflow Chart” on page 31-13.

Execute a Standalone Chart in MATLAB

You can execute a standalone chart in MATLAB without opening the Stateflow Editor. If the chart is open, then the editor highlights active states and transitions through chart animation.

1 Open the chart in the Stateflow Editor. In the MATLAB Command Window, enter:

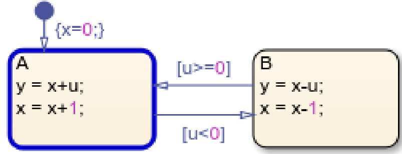
```
edit sf_chart.sfx
```

2 Create the Stateflow chart object by using the name of the `sfx` file for the standalone chart as a function. Specify the initial value for the data `u` as a name-value pair.

```
s = sf_chart(u=1)
```

```
Stateflow Chart

   Execution Function
    y = step(s)

   Local Data
      u          : 1
      x          : 1
      y          : 1
   Active States: {'A'}
```

This command creates the chart object `s`, executes the default transition, and initializes the values of `x` and `y`. The Stateflow Editor animates the chart and highlights the active state `A`.



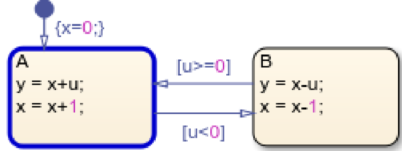
3 To execute the chart, call the `step` function. For example, suppose that you call the `step` function with a value of `u` = 1:

```
step(s,u=1)
```

```
disp(s)
```

```
Stateflow Chart

   Execution Function
    y = step(s)

   Local Data
      u          : 1
      x          : 2
      y          : 2
   Active States: {'A'}
```

Because the value of `u` does not satisfy the condition `[u<0]` to transition out of state `A`, this state remains active and the values of `x` and `y` increase to 2.



4 Execute the chart again, this time with a value of `u` = −1:

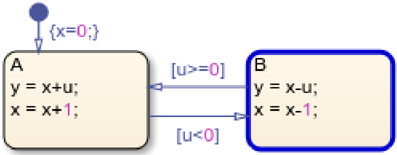
```
step(s,u=-1)
```

```
disp(s)
```

```
Stateflow Chart
```

```
Execution Function
 y = step(s)

Local Data
   u          : -1
   x          : 1
   y          : 3
Active States: {'B'}
```

The negative data value triggers the transition to state B. The value of x decreases to 1 and the value of y increases to 3.



5 To access the value of any chart data, use dot notation. For example, you can assign a value of 3 to the local data x by entering:

```
s.x = 3
```

Stateflow Chart

```
Execution Function
 y = step(s)

Local Data
   u          : -1
   x          : 3
   y          : 3
Active States: {'B'}
```

The standalone chart will use this data value in the next time execution step.

6 Execute the chart with a value of u = 2:

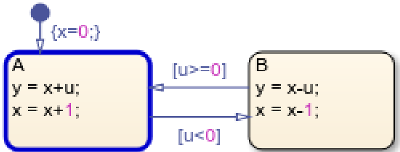
```
step(s,u=2)
```

```
disp(s)
```

Stateflow Chart

```
Execution Function

 y = step(s)

Local Data
   u          : 2
   x          : 4
   y          : 5
Active States: {'A'}
```

The chart transitions back to state A and modifies the values of x and y.



7 To stop the chart animation, delete the Stateflow chart object `s`:

```
delete(s)
```

Execute Multiple Chart Objects

You can execute multiple chart objects defined by the same standalone chart. Concurrent chart objects maintain their internal state independently, but remain associated to the same chart in the editor. The chart animation reflects the state of the chart object most recently executed. Executing multiple chart objects while the Stateflow Editor is open can produce confusing results and is not recommended.

See Also

`disp` | `edit` | `delete`

More About

- "Create Stateflow Charts for Execution as MATLAB Objects" on page 31-2
- "Debug a Standalone Stateflow Chart" on page 31-13
- "Execute Stateflow Chart Objects Through Scripts and Models" on page 31-18
- "Design Human-Machine Interface Logic by Using Stateflow Charts" on page 31-24

Debug a Standalone Stateflow Chart

A standalone Stateflow chart is a MATLAB class that defines the behavior of a finite state machine. Standalone charts implement classic chart semantics with MATLAB as the action language. You can program the chart by using the full functionality of MATLAB, including those functions that are restricted for code generation in Simulink. For more information, see "Create Stateflow Charts for Execution as MATLAB Objects" on page 31-2.

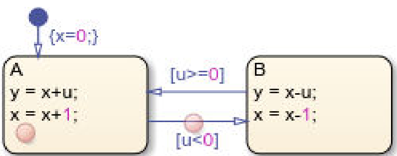
To enable debugging, set a breakpoint in the standalone chart or in a MATLAB script that executes the chart. Breakpoints pause the execution of a chart. While the execution is paused, you can step through each action in the chart, view data values, and interact with the MATLAB workspace to examine the state of the chart.

Note When debugging a standalone chart that you execute from a MATLAB script, first open the Stateflow Editor. Attempting to debug a standalone chart before opening the editor at least once can produce unexpected results.

For information on debugging Stateflow charts in Simulink models, see "Set Breakpoints to Debug Charts" on page 30-2.

Set and Clear Breakpoints

Breakpoints appear as circular red badges. For example, this chart contains breakpoints on the state `A` and on the transition from `A` to `B`.



You can set breakpoints on charts, states, and transitions.

Breakpoints on Charts

To set a breakpoint on a chart, right-click inside the chart and select **Set Breakpoint on Chart Entry**. This type of breakpoint pauses the execution before entering the chart.

To remove the breakpoint, right-click inside the chart and clear the **Set Breakpoint on Chart Entry** option.

Breakpoints on States and Transitions

You can set different types of breakpoints on states and transitions.

Object	Breakpoint Type
State	`On State Entry` — Pause the execution before performing the state `entry` actions.

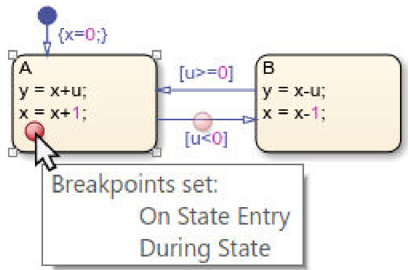
Object	Breakpoint Type
	`During State` — Pause the execution before performing the state `during` actions.
	`On State Exit` — Pause the execution after performing the state `exit` actions.
Transition	`When Transition is Tested` — Pause the execution before testing that the transition is a valid path. If no condition exists on the transition, this breakpoint type is not available.
	`When Transition is Valid` — Pause the execution after the transition is valid, but before taking the transition.

To set a breakpoint on a state or transition, right-click the state or transition and select **Set Breakpoint**. For states, the default breakpoints are `On State Entry` and `During State`. For transitions, the default breakpoint is `When Transition is Valid`. To change the type of breakpoint, click the breakpoint badge and select a different configuration of breakpoints. For more information, see "Manage Breakpoint Types and Conditions" on page 31-14.

To remove the breakpoint, right-click the state or transition and select **Clear Breakpoint**. To remove all of the breakpoints in a chart, right-click inside the chart and select **Clear All Breakpoints In Chart**.

Manage Breakpoint Types and Conditions

A breakpoint badge can represent more than one type of breakpoint. To see a tooltip that lists the breakpoint types that are set on a state or transition, point to its badge. In this example, the badge on the state `A` represents two breakpoint types: `On State Entry` and `During State`.

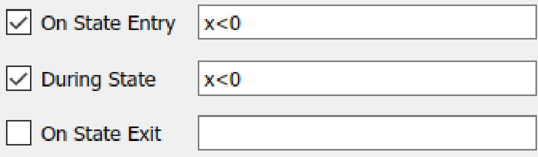


To change the type of breakpoint on an object, click the breakpoint badge. In the Breakpoints dialog box, you can select a different configuration of breakpoints, depending on the object type. Clearing all of the check boxes in the Breakpoints dialog box removes the breakpoint.

- [x] On State Entry
- [x] During State
- [ ] On State Exit

To limit the number of times that the execution stops at a breakpoint, add a condition to the breakpoint. By default, a Stateflow chart pauses whenever it reaches a breakpoint. When you add a condition to a breakpoint, the chart pauses at the breakpoint only when the condition is true. For

example, with these conditions, the breakpoints on state A pause the execution of the chart only when the value of x is negative.



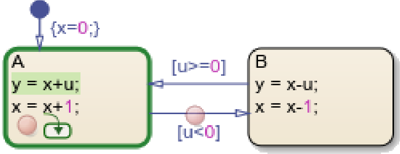
To specify a condition for the breakpoint, you can use any valid MATLAB expression that combines numerical values and Stateflow data objects that are in scope at the breakpoint.

Control Chart Execution After a Breakpoint

When execution stops at a breakpoint, Stateflow enters debugging mode.

- The MATLAB command prompt changes to K>>.
- The **Symbols** pane displays the value of each data object in the chart.
- The chart highlights active elements in blue and the currently executing object in green.

For example, when the execution stops at the breakpoint in state A, the border of the state and the first statement in the state `entry` action appear highlighted in green.



An execution status badge appears in the graphical object where execution pauses.

Badge	Description
	Execution is paused before entering a chart or in a state `entry` action.
	Execution is paused in a state `during` action.
	Execution is paused in a state `exit` action.
	Execution is paused before testing a transition.
	Execution is paused before taking a valid transition.

When the chart is paused at a breakpoint, you can continue the execution by using:

- Buttons in the **Debug** tab
- The MATLAB Command Window
- Keyboard shortcuts

Action	Debug Tab Button	MATLAB Command	Keyboard Shortcut	Description
Continue		`dbcont`	**Ctrl+T**	Continue execution to the next breakpoint.
Step Over		`dbstep`	**F10**	Advance to the next step in the chart execution. At the chart level, possible steps include: • Enter the chart • Execute a transition action • Activate a state • Execute a state action For more information, see "Execution of a Stateflow Chart" on page 2-12.
Step In		`dbstep in`	**F11**	From a state or transition action that calls a function, advance to the first executable statement in the function. From a statement in a function containing another function call, advance to the first executable statement in the second function. Otherwise, advance to the next step in the chart execution. (See the Step Over option.)
Step Out		`dbstep out`	**Shift+F11**	From a function call, return to the statement calling the function. Otherwise, continue execution to the next breakpoint. (See the Continue option.)
Stop		`dbquit`	**Ctrl+Shift+T**	Exit debug mode and interrupt the execution. When you execute the standalone chart from the Stateflow Editor, this action removes the chart object from the MATLAB workspace.

In state or transition actions containing more than one statement, you can step through the individual statements one at a time by selecting **Step Over**. The Stateflow Editor highlights each statement before executing it.

Note Because standalone charts define temporal logic in terms of wall-clock time, a temporal logic operator can become valid while a chart is paused at a breakpoint. In this case, the chart exits debugging mode and the execution continues to the next breakpoint.

Examine and Change Values of Chart Data

When Stateflow is in debug mode, the **Symbols** pane displays the value of each data object in the chart. You can also examine data values by pointing to a state or a transition in the chart. A tooltip displays the value of each data object used in the selected state or transition.

To test the behavior of your chart, in the **Symbols** pane, you can change the value of a data object during execution. Alternatively, at the debugging prompt, enter the new value by using the keyword `this` in place of the chart object name. For instance, to change the value of the local data `x`, enter:

```
this.x = 7
```

The new value appears in the **Symbols** pane.

Note When debugging a chart in a Simulink model, do not use the keyword `this`. Instead, you can access all Stateflow data directly at the debugging prompt. For more information, see “View and Modify Data in the MATLAB Command Window” on page 30-11.

See Also

More About

- “Create Stateflow Charts for Execution as MATLAB Objects” on page 31-2
- “Execute and Unit Test Stateflow Chart Objects” on page 31-8
- “Set Breakpoints to Debug Charts” on page 30-2

Execute Stateflow Chart Objects Through Scripts and Models

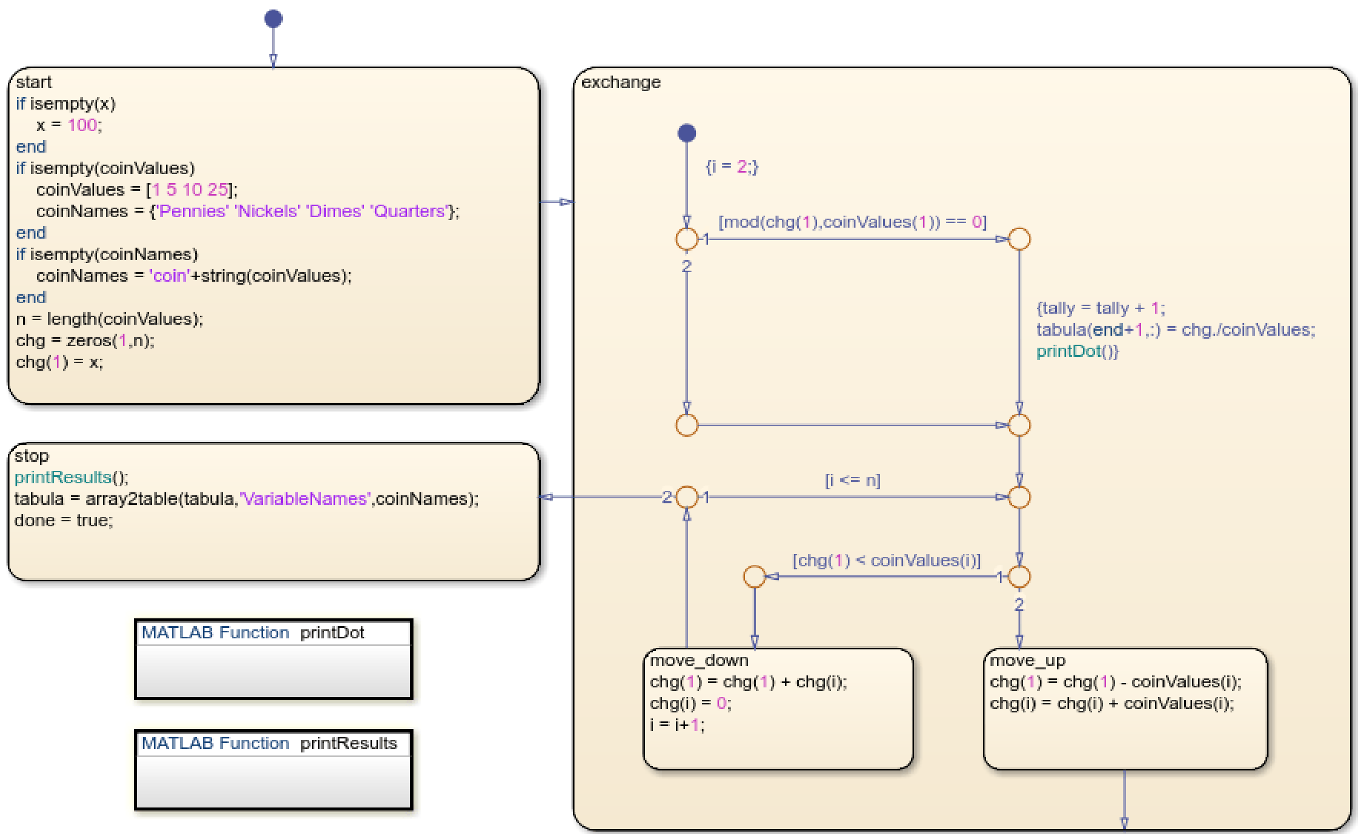
A standalone Stateflow chart is a MATLAB class that defines the behavior of a finite state machine. Standalone charts implement classic chart semantics with MATLAB as the action language. You can program the chart by using the full functionality of MATLAB, including those functions that are restricted for code generation in Simulink. For more information, see “Create Stateflow Charts for Execution as MATLAB Objects” on page 31-2.

This example shows how to execute a standalone Stateflow chart by using a MATLAB script or a Simulink model.

Count Ways to Make Change for Currency

The file `sf_change.sfx` defines a standalone Stateflow chart that counts the number of ways to make change for a given amount of money. The chart contains these data objects:

- `x` is the amount of money to change. The default value is 100.
- `coinValues` is a vector of coin denominations arranged in increasing order. `coinNames` is an array of corresponding coin names. The default values represent standard American coins (pennies, nickels, dimes, and quarters).
- `tally` is the number of valid change configurations.
- `tabula` is an array containing the different valid change configurations.
- `chg`, `done`, `i`, and `n` are local data used by the change-counting algorithm.
- `textWidth` and `quietMode` are local data that control how the chart displays its results.



Copyright 2018-2021 The MathWorks, Inc.

The chart begins with a potential change configuration consisting entirely of the lowest-value coins, specified by an index of 1. At each execution step, the state `exchange` modifies this configuration in one of two ways:

- The substate `move_up` exchanges some lowest-value coins for a coin with a higher value, specified by the index `i`.
- The substate `move_down` exchanges all of the coins with the value specified by the index `i` for lowest-value coins. Then `move_up` exchanges some lowest-value coins for a coin with a value specified by the index `i+1` or higher.

A potential change configuration is valid when the number of cents represented by the lowest-value coins is divisible by the value of that type of coin. When the chart encounters a new valid configuration, it increments `tally` and appends the new configuration to `tabula`.

When no more coin exchanges are possible, the state `stop` becomes active. This state prints the results of the computation, transforms the contents of `tabula` to a table, and sets the value of `done` to `true`.

Execute Standalone Chart in a MATLAB Script

To run the change-counting algorithm to completion, you must execute the standalone chart multiple times. For example, the MATLAB script `sf_change_script.m` creates a chart object `chartObj` and initializes the value of the local data `x` to 27. The configuration option `'-`

warningOnUninitializedData', which the script sets to false, eliminates the warning that tabula is an empty array in the new chart object. The while loop executes the chart until the local data done becomes true. Finally, the script displays the value of tabula.

```
chartObj = sf_change('-warningOnUninitializedData',false,x=27);

while ~chartObj.done
    step(chartObj);
end

disp(chartObj.tabula)
```

When you run this script, the standalone chart calculates the number of ways to make change for 27 cents by using standard American coins:

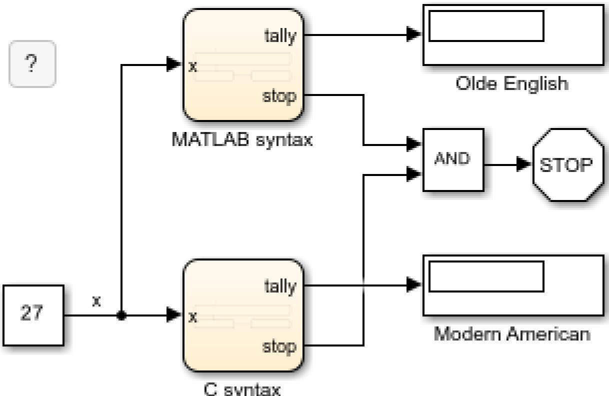
```
sf_change_script
```

```
.............
There are 13 ways to make change for 27 cents.
    Pennies    Nickels    Dimes    Quarters
    _______    _______    _____    ________

      27          0         0          0
      22          1         0          0
      17          2         0          0
      12          3         0          0
       7          4         0          0
       2          5         0          0
      17          0         1          0
      12          1         1          0
       7          2         1          0
       2          3         1          0
       7          0         2          0
       2          1         2          0
       2          0         0          1
```

To determine the number of ways to make change for a different amount, or to use a different system of currency, change the values of x and coinValues. For example, to use British currency, initialize coinValues to [1 2 5 10 20 25 50].

Execute Standalone Chart in a Simulink Model

You can execute a standalone Stateflow chart from within a Simulink model. For example, the model sf_change_model contains two Stateflow charts that use the standalone chart sf_change to count the number of ways to make change for 27 cents in two different currency systems. You can simulate the model, but the functions that execute the standalone chart do not support code generation.



Each chart contains these states:

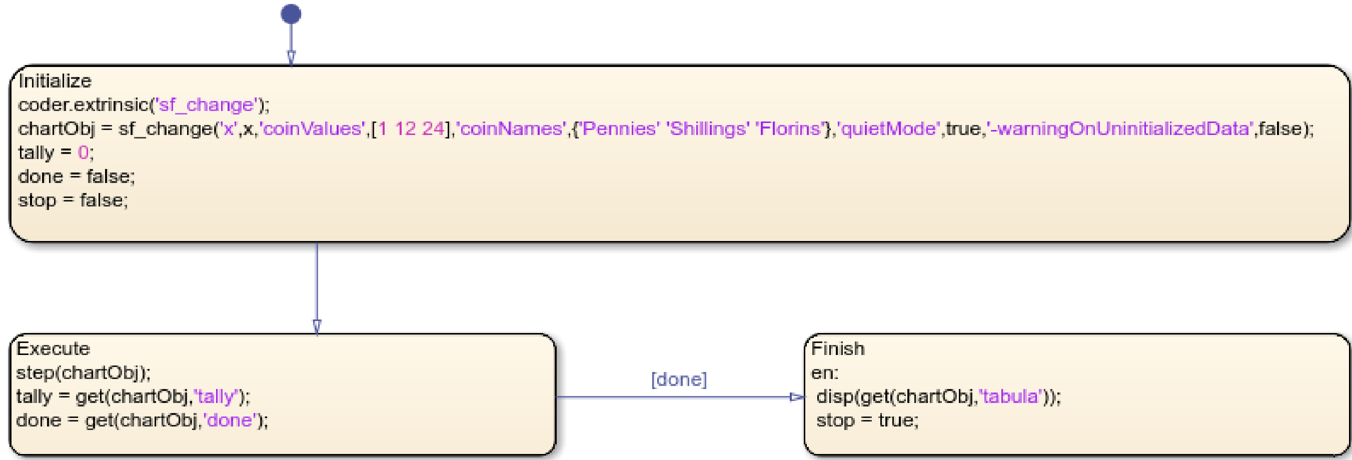
- `Initialize` creates a local chart object `chartObj` that implements the change-counting algorithm for the input value `x`.
- `Execute` calls the `step` function to execute the standalone chart and stores the result as the output data `tally`.
- `Finish` displays the results of the algorithm in the Diagnostic Viewer window and sets the value of the output data `done` to `true`.

When both charts reach their respective `Finish` state, the simulation of the model stops and the Display blocks show the final values of the two tallies.

Execution Using MATLAB as the Action Language

The chart `MATLAB syntax` uses MATLAB as the action language. To execute the standalone Stateflow chart, this chart must follow these guidelines:

- The local variable `chartObj` that contains the handle to the chart object has type `Inherit: From definition in chart`.
- Before creating the chart object, the `Initialize` state calls the `coder.extrinsic` function to declare `sf_change` as an extrinsic function that is restricted for code generation in Simulink. See “Call Extrinsic MATLAB Functions in Stateflow Charts” on page 14-13.
- The `Execute` and `Finish` states access the local data for the standalone chart by calling the `get` function.



When you simulate this chart with an input of `x` = 27, the Display block `Olde English` shows a tally of 4. The Diagnostic Viewer window shows these results:

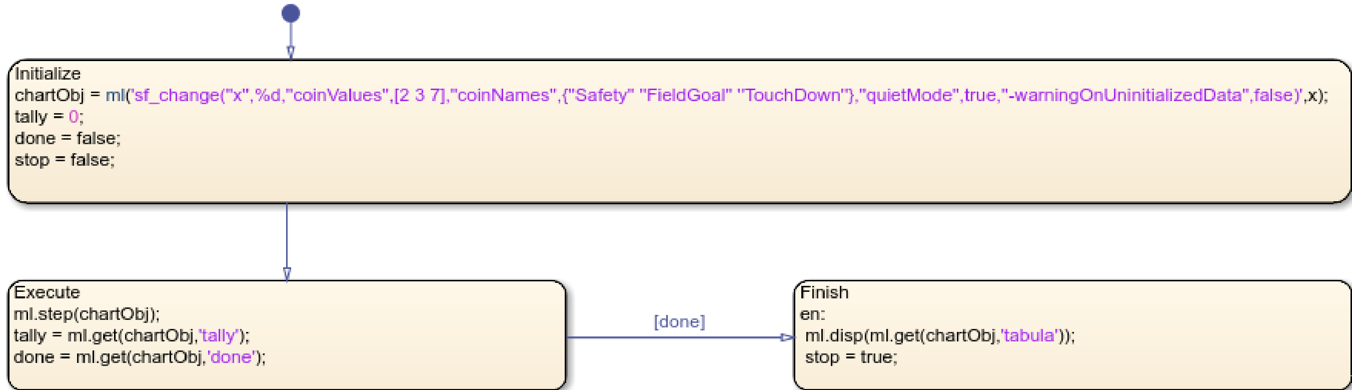
```
Pennies    Shillings    Florins
_______    _________    _______
  27           0           0
  15           1           0
   3           2           0
   3           0           1
```

Execution Using C as the Action Language

The chart `C syntax` uses C as the action language. To execute the standalone Stateflow chart, this chart must follow these guidelines:

- The local variable `chartObj` that contains the handle to the chart object has type `ml`.
- The `Initialize` state calls the `ml` function to create the chart object.
- The `Execute` and `Finish` states use the `ml` namespace operator to access the `step`, `get`, and `displ` functions to execute the standalone chart, access its local data, and display the results of the algorithm.

For more information, see “Access MATLAB Functions and Workspace Data in C Charts” on page 14-20.



When you simulate this chart with an input of x = 27, the Display block `Modern American` shows a tally of 13. The Diagnostic Viewer window shows these results:

```
Safety    FieldGoal    TouchDown
______    _________    _________
  12          1            0
   9          3            0
   6          5            0
   3          7            0
   0          9            0
  10          0            1
   7          2            1
   4          4            1
   1          6            1
   5          1            2
   2          3            2
   3          0            3
   0          2            3
```

See Also

More About

- “Create Stateflow Charts for Execution as MATLAB Objects” on page 31-2
- “Execute and Unit Test Stateflow Chart Objects” on page 31-8
- “Call Extrinsic MATLAB Functions in Stateflow Charts” on page 14-13
- “Access MATLAB Functions and Workspace Data in C Charts” on page 14-20

Design Human-Machine Interface Logic by Using Stateflow Charts

This example shows how to model the logic of a graphical user interface in a standalone Stateflow® chart. Standalone charts implement classic chart semantics with MATLAB® as the action language. You can program the chart by using the full functionality of MATLAB, including those functions that are restricted for code generation in Simulink®. For more information, see “Create Stateflow Charts for Execution as MATLAB Objects” on page 31-2.

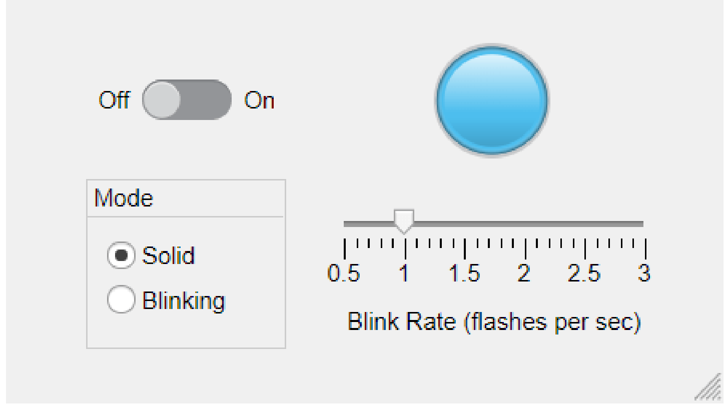
You can execute a standalone Stateflow chart by invoking its input events and using temporal operators. The event- and timer-driven execution workflow is suitable for designing the logic underlying human-machine interfaces (HMIs) and graphical user interfaces (UIs).

- When you use the MATLAB App Designer, callback functions from the interface widgets invoke events in the chart.
- In the Stateflow chart, temporal operators and local data control the properties of the user interface.

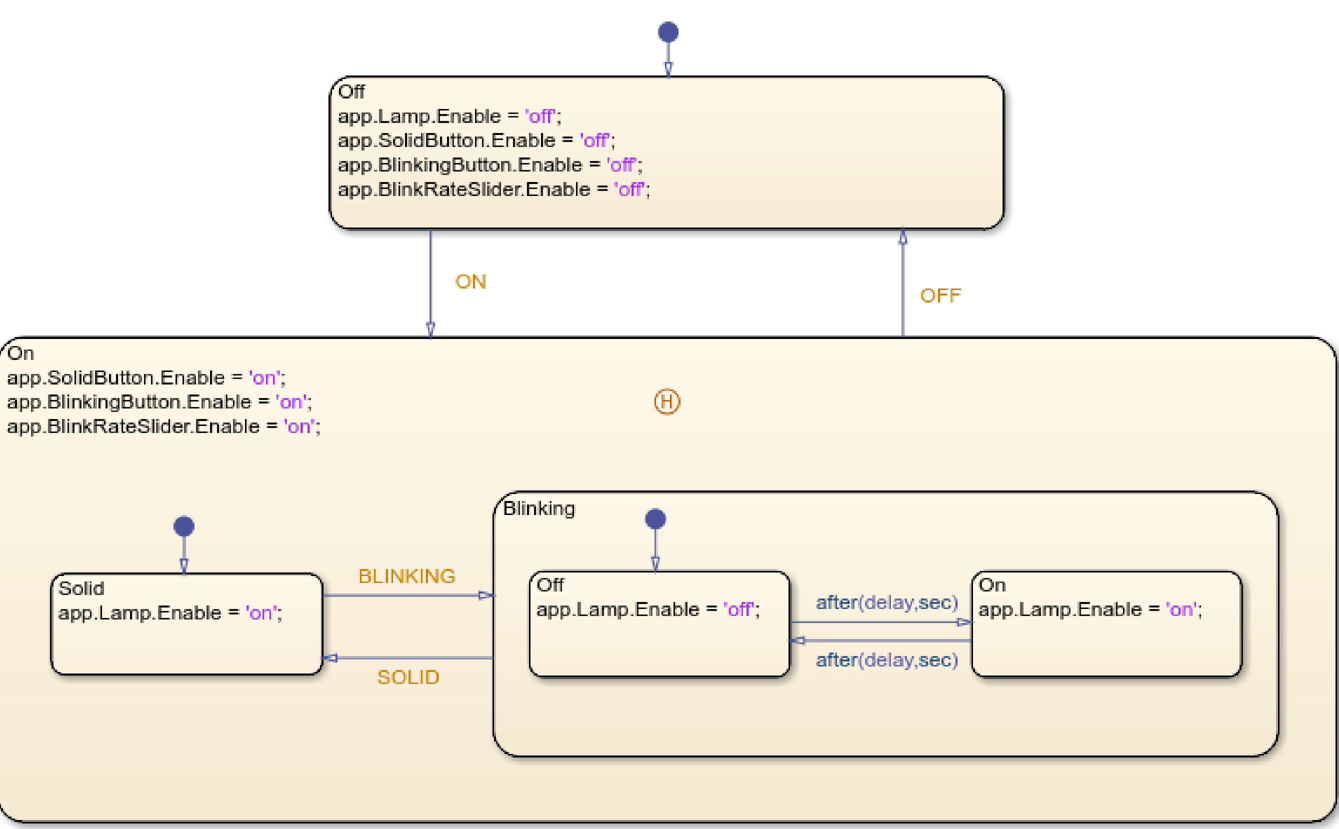
For more information on how to use MATLAB to create graphical user interfaces, see “Develop Apps Using App Designer”.

Control an App Designer User Interface

This user interface contains an On-Off switch that controls a lamp. When the switch is in the On position, the lamp lights up in one of two modes, solid or blinking, depending on the position of the Mode option button. You control the rate of blinking by moving the Blink Rate slider. To start the app, in the App Designer toolstrip, click **Run**.



The file `sf_lamp_logic.sfx` defines a standalone Stateflow chart that implements the logic for the user interface. The chart has input events (`ON`, `OFF`, `BLINKING`, and `SOLID`) and local data (`delay` and `app`). The actions in the chart control which widgets are accessible from each state. For instance, the actions in the `Off` state cause the Lamp widget, the Mode option buttons, and the Blink Rate slider in the user interface to appear dimmed.



In the `On` state, the substates `Solid` and `Blinking` denote the two modes of operation. To implement a blinking lamp, the chart relies on the temporal logic operator `after`. When the chart enters the state `Blinking.Off`, the expression `after(delay,sec)` on the outgoing transition creates a MATLAB timer object that executes the chart after a number of seconds. The chart then transitions to the state `Blinking.On` and creates another timer object to trigger the transition back to `Blinking.Off`. While the chart continually transitions between the two states, you can adjust the rate of blinking by changing the value of the local data delay or transition out of blinking mode by invoking the input events `SOLID` or `OFF`.

The history junction in the `On` state preserves information on the most recently active substate so that the user interface returns to the previous mode of operation when you turn on the lamp.

Execute Standalone Chart by Using Events

You can execute the standalone chart by calling its input event functions in the MATLAB Command Window. The Stateflow Editor shows the effects of each of these commands by highlighting active states and transitions through chart animation.

1. Create the chart object `L` and initialize the value of `delay` to 0.5. This value corresponds to a blinking rate of one flash per second (on for 0.5 seconds and off for 0.5 seconds).

```
L = sf_lamp_logic(delay=0.5);
```

2. Turn on the lamp.

```
ON(L)
```

3. Switch to blinking mode.

```
BLINKING(L)
```

4. Set the value of `delay` to 0.25. This value corresponds to a blinking rate of two flashes per second (on for 0.25 seconds and off for 0.25 seconds).

```
L.delay = 0.25;
```

5. Switch to solid mode.

```
SOLID(L)
```

6. Turn off the lamp.

```
OFF(L)
```

7. Delete the chart object `L` from the MATLAB workspace.

```
delete(L)
```

Connect Standalone Chart to User Interface

To establish a bidirectional connection between the user interface and the standalone Stateflow chart, open the App Designer window and select **Code View**.

1. In the App Designer window, create a private property `lampLogic` to store the handle to the Stateflow chart object.

```
properties (Access = private)
    lampLogic
end
```

2. Create a `StartupFcn` callback function that creates the chart object and sets its local data `app` to the user interface handle. Assign the chart object handle to the `lampLogic` private property.

```
function StartupFcn(app)
    app.lampLogic = sf_lamp_logic(delay=0.5,app=app);
end
```

3. Create a `CloseRequestFcn` callback function that deletes the chart object when you close the user interface.

```
function UIFigureCloseRequest(app, event)
    delete(app.lampLogic);
    delete(app);
end
```

4. For each one of the user interface widgets, add a callback function that invokes the appropriate event on the standalone chart.

- `ValueChangedFcn` callback function for Switch widget:

```
function SwitchValueChanged(app, event)
    value = app.Switch.Value;
    switch lower(value)
        case "off"
            OFF(app.lampLogic);
        case "on"
```

```
            ON(app.lampLogic);
    end
end
```

- `SelectionChangedFcn` callback function for Mode Button widget:

```
function ModeButtonGroupSelectionChanged(app, event)
    selectedButton = app.ModeButtonGroup.SelectedObject;
    if selectedButton == app.SolidButton
        SOLID(app.lampLogic);
    else
        BLINKING(app.lampLogic);
    end
end
```

- `ValueChangedFcn` callback function for Blink Rate Slider widget:

```
function BlinkRateSliderValueChanged(app, event)
    app.lampLogic.delay = round(0.5/app.BlinkRateSlider.Value,2);
end
```

When you run the user interface, you can observe the effects of adjusting the control widgets on the chart canvas and on the lamp widget.

See Also

`after`

More About

- “Create Stateflow Charts for Execution as MATLAB Objects” on page 31-2
- “Activate a Stateflow Chart by Sending Input Events” on page 12-8
- “Control Chart Execution by Using Temporal Logic” on page 14-35
- “Resume Prior Substate Activity by Using History Junctions” on page 1-58
- “Develop Apps Using App Designer”

Model a Communications Protocol by Using Chart Objects

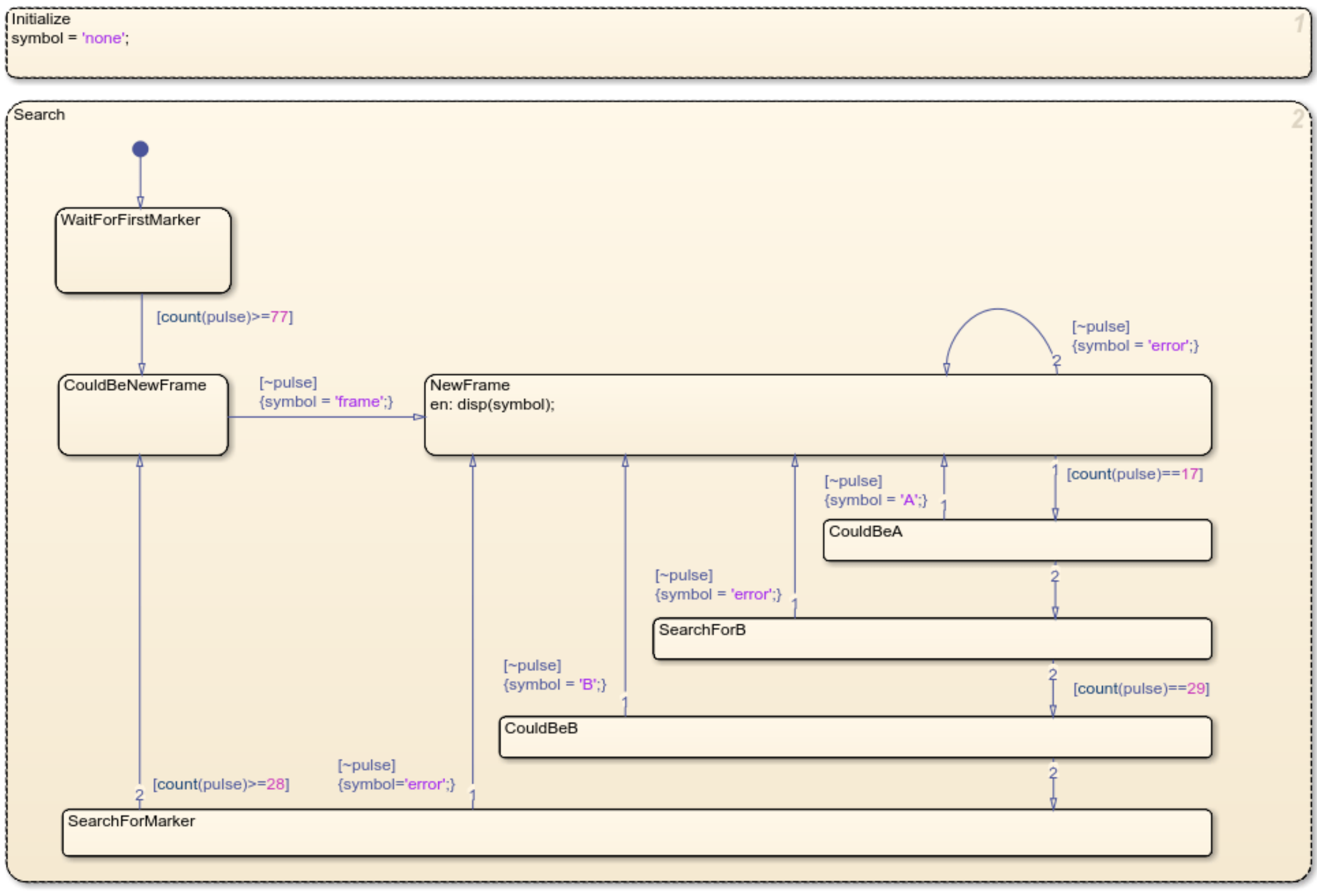
This example shows how to use a standalone Stateflow® chart to model a frame-synchronization and symbol-detection component in a communications system. Standalone charts implement classic chart semantics with MATLAB® as the action language. You can program the chart by using the full functionality of MATLAB, including those functions that are restricted for code generation in Simulink®. For more information, see “Create Stateflow Charts for Execution as MATLAB Objects” on page 31-2.

Implement a Symbol-Detection Algorithm

In this example, the input to the communications system consists of a binary signal of zeros and ones received every 10 milliseconds. The input signal can contain any combination of:

- A 770-ms pulse (77 consecutive ones) to mark the beginning and end of a frame of data and to ensure system synchronization.
- A 170-ms pulse (17 consecutive ones) to indicate symbol A.
- A 470-ms pulse (47 consecutive ones) to indicate symbol B.

The file `sf_frame_search.sfx` defines a standalone Stateflow chart that implements this communication protocol. The chart consists of two outer states in parallel decomposition. The `Initialize` state resets the value of the local data `symbol` at the start of each execution step. The `Search` state contains the logic that defines the symbol-detection algorithm. When this state detects one of the pulses allowed by the communication protocol, the name of the corresponding symbol is stored as `symbol` and displayed in the MATLAB Command Window. Parallel decomposition enables the chart to preprocess the input data. For more information, see “Define Exclusive and Parallel Modes by Using State Decomposition” on page 1-35.



To track the length of a pulse through several execution steps, the chart uses the `count` operator. This operator simplifies the design of the chart by eliminating the need for a manual counter. For example, the condition `[count(pulse)==17]` guards the outgoing transition from the substate `NewFrame`. This condition becomes true when the data `pulse` is one for 17 consecutive execution steps. In this case, the chart transitions to the `CouldBeA` substate. If this transition is followed by an input of zero, then the chart registers the reception of symbol A and transitions back to the `NewFrame` substate. Otherwise, the chart transitions to the `SearchForB` state from which the condition `[count(pulse)==29]` searches for an additional 29 ones to mark symbol B.

Execute Standalone Chart

In the MATLAB script `sf_frame_tester.m`, the sample code generates a short signal consisting of several valid pulses and one transmission error. The error consists of a 470-ms pulse that is too long to represent symbol A and too short to represent symbol B.

```
%% Test Symbol Detection Algorithm
% Generate a short signal consisting of several valid pulses and one
% transmission error.

f = sf_frame_search(pulse=0);         % create chart object
sendPulse(f,77);                      % frame marker
sendPulse(f,17);                      % A
```

```
sendPulse(f,47);                    % B
sendPulse(f,37);                    % transmission error
sendPulse(f,47);                    % B
sendPulse(f,17);                    % A
sendPulse(f,77);                    % frame marker
delete(f);                          % delete chart object

function sendPulse(f,n)
% Send a pulse of n ones and one zero to chart object f.

for i = 1:n
    step(f,pulse=1);
    printDot(1)
end

printDot(0)
step(f,pulse=0);

    function printDot(x)
        persistent k
        if isempty(k)
            k = 1;
        end

        if x == 0
            fprintf("\n");
            k = 1;
        elseif k == 50
            fprintf(".\n");
            k = 1;
        else
            fprintf(".");
            k = k+1;
        end
    end
end
```

Running the script produces these results in the MATLAB Command Window:

```
..................................................
...........................
frame
.................
A
...............................................
B
.....................................
error
...............................................
B
.................
A
..................................................
...........................
frame
```

During the simulation, the chart animation provides a visual indication of the runtime behavior of the algorithm.

See Also

`count`

More About

- “Create Stateflow Charts for Execution as MATLAB Objects” on page 31-2
- “Execute Stateflow Chart Objects Through Scripts and Models” on page 31-18
- “Control Chart Execution by Using Temporal Logic” on page 14-35
- “Define Exclusive and Parallel Modes by Using State Decomposition” on page 1-35

Implement a Financial Strategy by Using Stateflow

This example shows how to use a standalone Stateflow® chart to model a financial trading strategy known as *Bollinger Bands*. Standalone charts implement classic chart semantics with MATLAB® as the action language. You can program the chart by using the full functionality of MATLAB, including those functions that are restricted for code generation in Simulink®. For more information, see “Create Stateflow Charts for Execution as MATLAB Objects” on page 31-2.

Compute Bollinger Bands

The Bollinger Bands trading strategy is to maintain a moving average of `N` stock prices for some commodity and issuing trading instructions depending on the value of the stock:

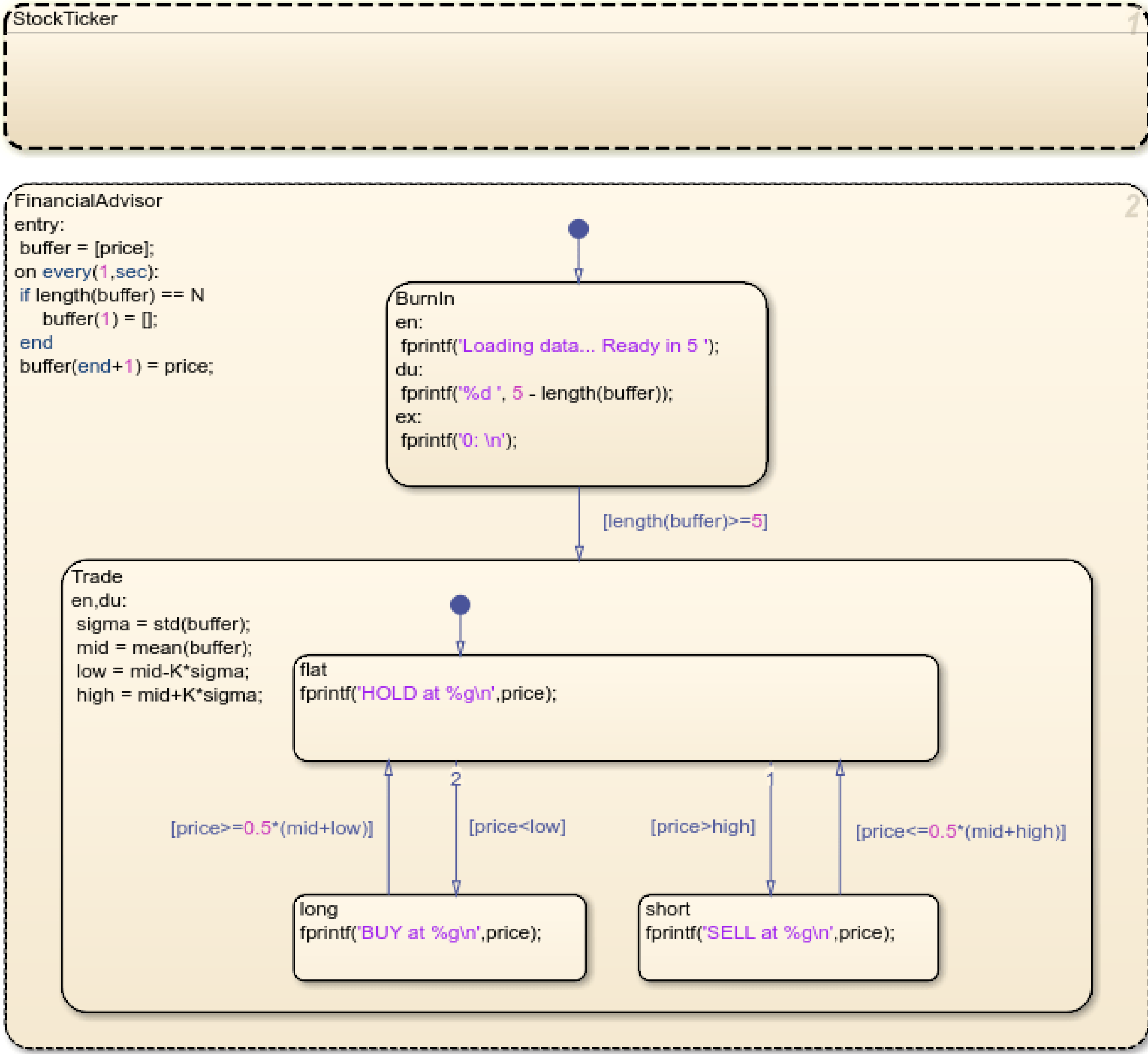
- "Buy" when the value of the stock drops `K` standard deviations below the moving average.
- "Sell" when the value of the stock rises `K` standard deviations above the moving average.
- "Hold" when the value of the stock is within `K` standard deviations of the moving average.

Typical implementations for this strategy use values of `N` = 20 and `K` = 2.

The file `sf_stock_watch.sfx` defines a standalone Stateflow chart that implements this financial strategy. The chart consists of two outer states in parallel decomposition.

- The `StockTicker` subchart records the current price of a stock. The subchart hides the details for calculating stock prices. To access real-time market data from financial data providers, one possible implementation involves the use of the Datafeed Toolbox™. For details, see “Datafeed Toolbox”.
- The `FinancialAdvisor` state uses the last `N` stock prices to compute high and low bands. Depending on the current price relative to these bands, the state generates "buy," "sell," or "hold" instructions. The action `on every(1,sec)` creates a MATLAB® timer to execute the chart every second. See “Control Chart Execution by Using Temporal Logic” on page 14-35.

Parallel decomposition is a common design pattern that enables your algorithm to preprocess input data. For more information, see “Define Exclusive and Parallel Modes by Using State Decomposition” on page 1-35.



Execute Standalone Chart

To execute the standalone chart, create a Stateflow chart object w:

```
w = sf_stock_watch();
```

The chart generates a stream of stock prices and issues "buy," "sell," or "hold" instructions.

```
Note: Chart execution continues until you delete the chart object.
Loading data... Ready in 5 4 3 2 1 0:
HOLD at 14.1942
SELL at 14.2802
SELL at 14.2471
```

```
HOLD at 14.2025
BUY at 14.1444
```

To stop the chart execution, delete the chart object w:

```
delete(w);
```

See Also

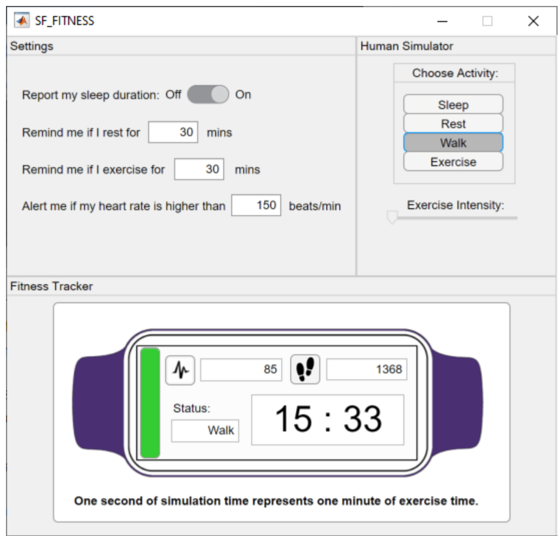
More About

- “Create Stateflow Charts for Execution as MATLAB Objects” on page 31-2
- “Control Chart Execution by Using Temporal Logic” on page 14-35
- “Define Exclusive and Parallel Modes by Using State Decomposition” on page 1-35
- “Datafeed Toolbox”

Model a Fitness App by Using Standalone Charts

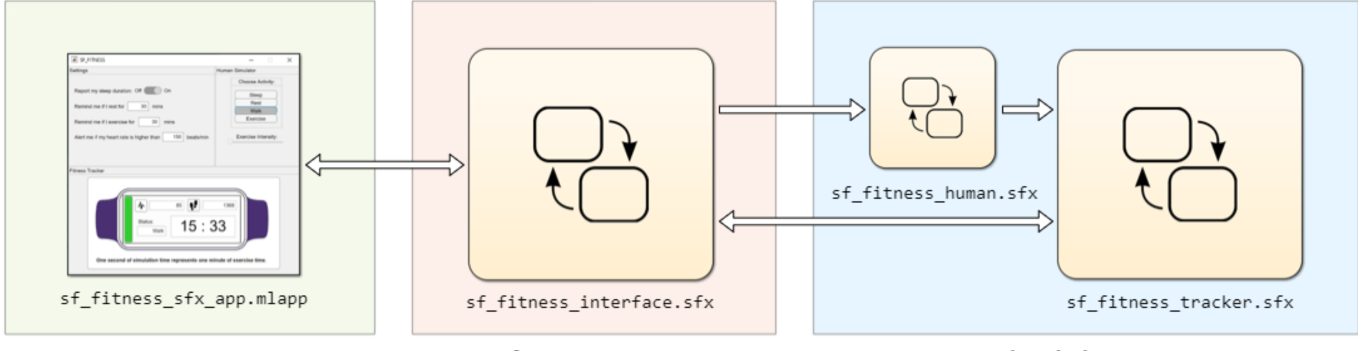
This example shows how to create an application composed of multiple standalone Stateflow® charts and a MATLAB® app. The standalone charts model the control and plant systems for the application and interface with the MATLAB app. For more information on connecting a standalone chart to a MATLAB app, see “Design Human-Machine Interface Logic by Using Stateflow Charts” on page 31-24. For a version of this example that uses Stateflow charts in a Simulink® model, see “Model a Fitness Tracker” on page 27-59.

In this example, a MATLAB app models a fitness tracker. When you run the app, you can adjust the settings for the tracker and select an activity (**Sleep**, **Rest**, **Walk**, or **Exercise**). When you choose **Exercise**, you can also set the intensity of your workout.



The standalone chart `sf_fitness_interface` provides a bidirectional connection between the MATLAB app and the other standalone charts in the example, `sf_fitness_human` and `sf_fitness_tracker`. These charts model a human simulator and provide the core logic for the fitness tracker, respectively. When you interact with the widgets in the app, the `sf_fitness_interface` chart communicates your selections to the other charts in the example. Conversely, the chart uses the output of the fitness tracker to update the numeric and text fields in the app.

This schematic diagram illustrates the transfer of information between the app and the charts in the example.



To start the example, run the `sf_fitness_sfx_app` app. The app creates a chart object for `sf_fitness_interface`. This chart, in turn, creates chart objects for the other two charts in the example. The chart also creates a MATLAB `timer` object that executes all three charts at a frequency proportional to the heart rate in the human simulator chart. While the example is running, one second of simulation represents one minute of exercise time. To stop the example, close the app.

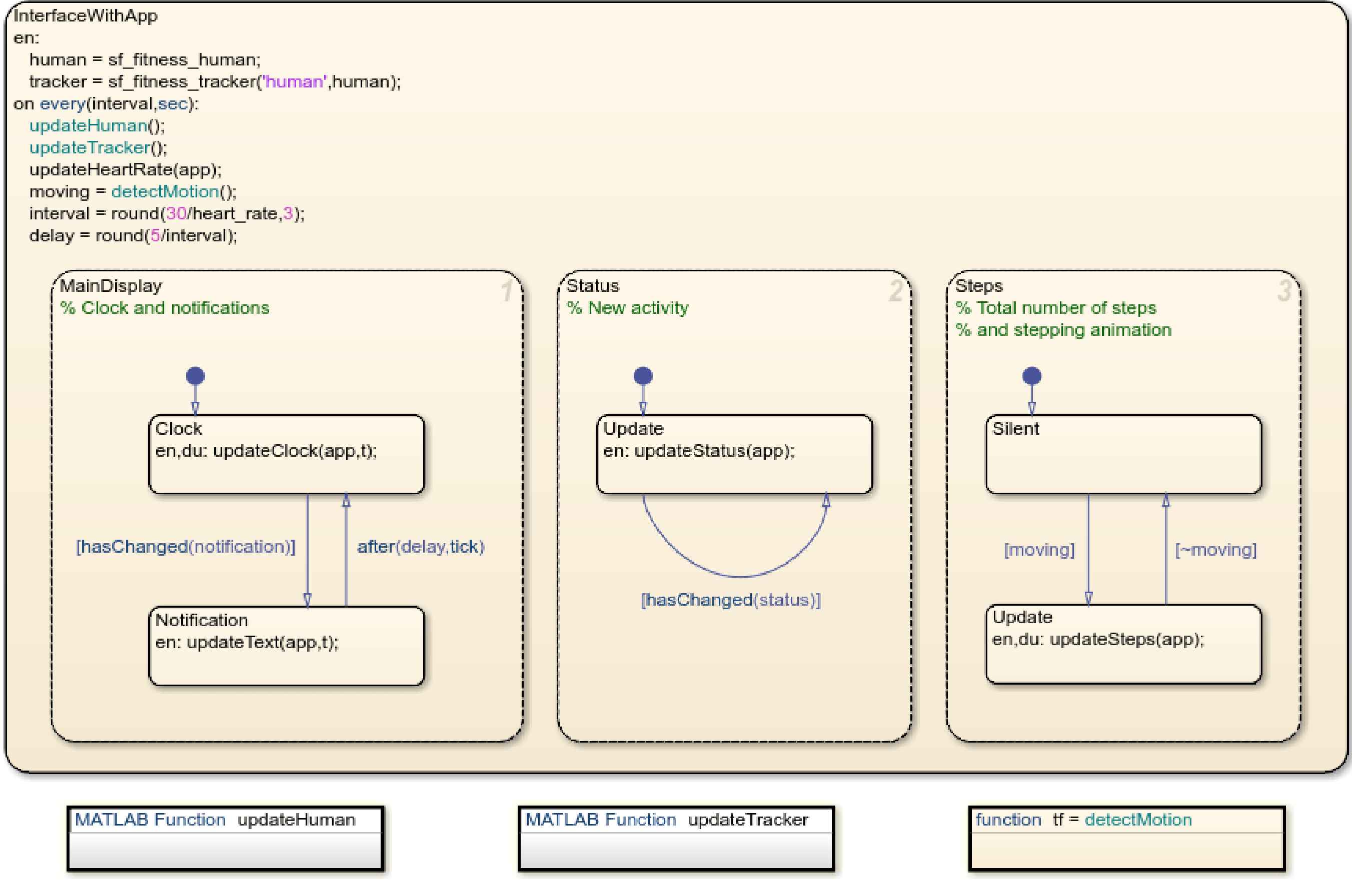
Connect Chart to MATLAB App

The chart `sf_fitness_interface` is configured to communicate with the MATLAB app `sf_fitness_sfx_app`.

- The chart uses the local data object `app` to interface with the MATLAB app. The chart uses this local data object when it calls the helper functions `updateStatus`, `updateClock`, `updateText`, `updateSteps`, and `updateHeartRate`. In the app, these helper functions change the contents of the activity status, clock, and step counter fields, and create the animation effects in the heartbeat and footstep displays. For example, when there is a new notification from the fitness tracker, the substate `MainDisplay` calls the helper function `updateText`. This function replaces the contents of the clock display with a customized notification. After a short delay, the substate calls the helper function `updateClock` to restore the clock display.

- The app uses a property called `chart` to interface with the chart. The app uses this property to read the chart local data. For example, the helper functions `updateHeartRate` and `updateSteps` read the chart local data `heart_rate` and `total_steps`, respectively. Additionally, when you close the app, the `UIFigureCloseRequest` callback uses the `chart` property to stop the execution of the charts in the example by deleting their chart objects.

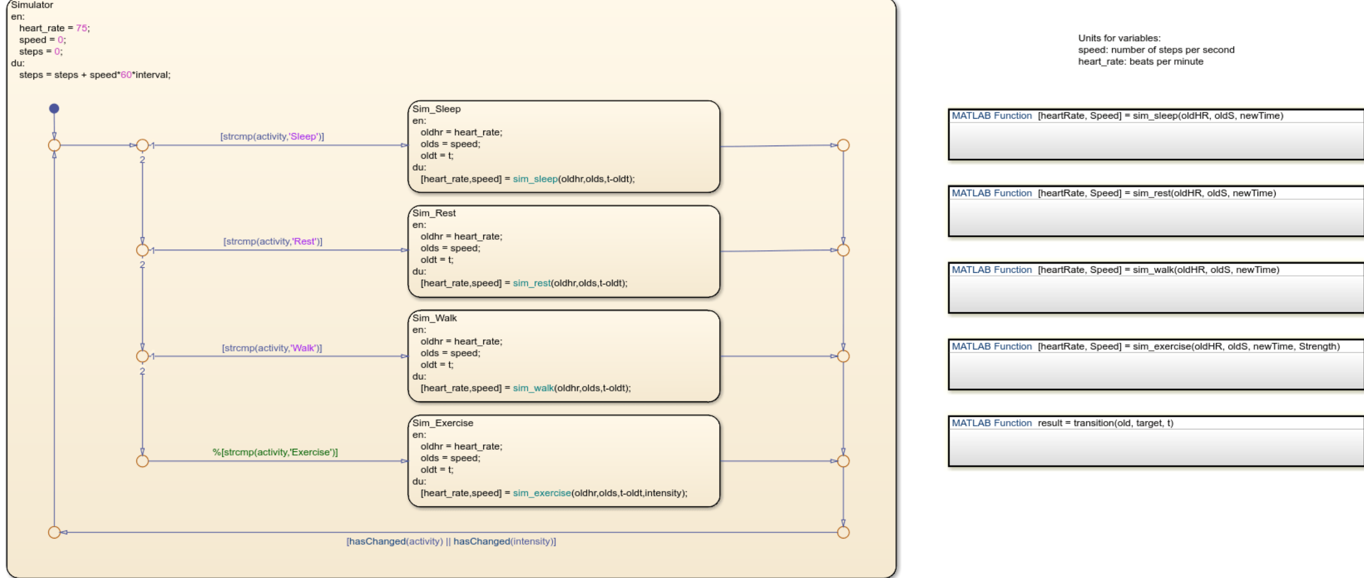
For more information on how to create a bidirectional connection between your MATLAB app and a standalone Stateflow chart, see “Design Human-Machine Interface Logic by Using Stateflow Charts” on page 31-24.



To establish communications with the human simulator and fitness tracker charts, the `sf_fitness_interface` chart saves their chart objects as the local data `human` and `tracker`. The chart-level MATLAB functions `updateHuman` and `updateTracker` use these objects to write to and read from the local data in the charts. For example, when you select a new activity or change the intensity of your workout in the **Human Simulator** pane of the app, `updateHuman` sets the value of the local data `activity` and `intensity` in the human simulator chart. Similarly, when you change the value of one of the fields in the **Settings** pane of the app, `updateTracker` updates the value of the corresponding local data in the fitness tracker chart.

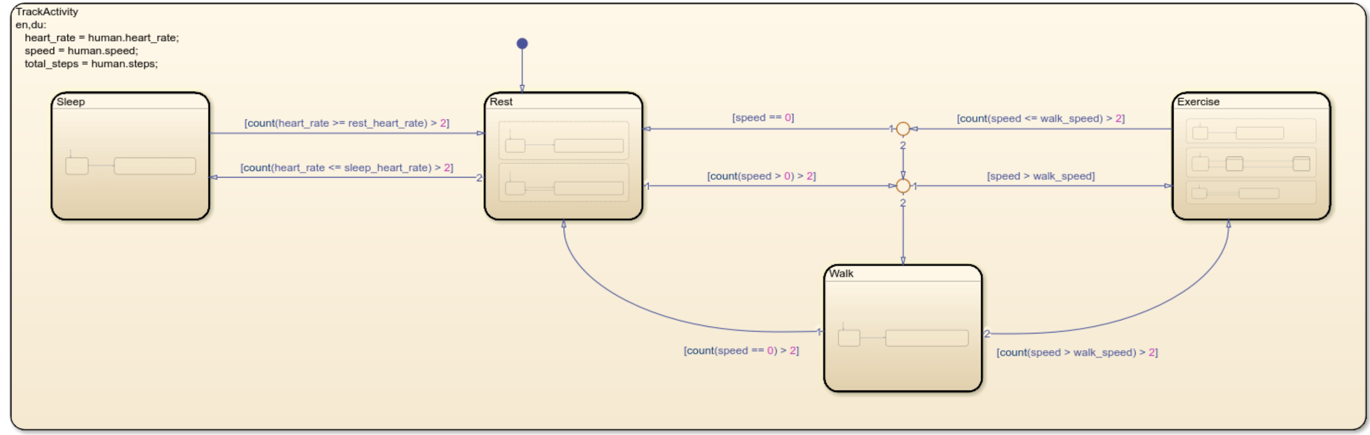
Simulate Vital Signs Based on Activity

The human simulator chart `sf_fitness_human` models the vital signs of a human engaged in the activity you select in the app. The chart stores these vital signs (representing your heart rate, speed, and the number of steps that you have taken) as local variables that the fitness tracker can access. When you select a new activity or adjust the intensity of your workout, the chart calls the function `transition` to ensure that these vital signs change gradually over time. To detect changes in activity or exercise intensity, the chart calls the `hasChanged` operator. For more information, see “Detect Changes in Data and Expression Values” on page 14-63.



Determine Fitness Tracker Output

The chart `sf_fitness_tracker` models the core logic of the fitness tracker. The chart consists of four subcharts that correspond to the possible activities. The chart registers your activity status based on the heart rate and speed produced by the human simulator chart and transitions between these subcharts. To filter out signal noise, the chart uses the `count` operator to implement simple debouncing logic. For instance, when you are at rest, you can make some quick and sudden movements that do not correspond to exercise. The chart determines that you are walking or exercising only if your motion lasts longer than two evaluations of the chart object.



The chart uses other temporal logic operators to track the amount of time you spend in each activity and determine when to send notifications to the app:

- The exit actions in each subchart call the `elapsed` operator to determine how long the subchart was active.
- The chart uses the `after` operator to determine when you sleep or walk for longer than five minutes, rest or exercise for longer than the threshold you specify in the app, or exercise at a high intensity (taking more than 4 steps a second) for longer than 15 minutes. In each of these cases,

the chart updates the value of the local data `notification`. The `sf_fitness_interface` chart reads this value and causes a notification to appear in the main display of the app. Depending on the type of notification, the notification button changes color.

See Also

`count` | `elapsed` | `hasChanged`

More About

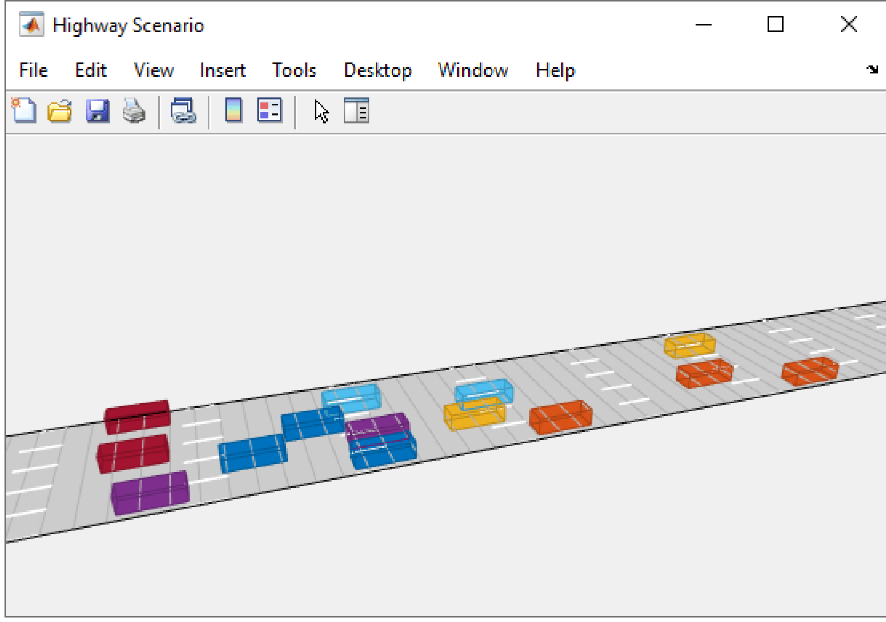
- "Model a Fitness Tracker" on page 27-59
- "Design Human-Machine Interface Logic by Using Stateflow Charts" on page 31-24
- "Detect Changes in Data and Expression Values" on page 14-63

Automate Control of Intelligent Vehicles by Using Stateflow Charts

This example shows how to model a highway scenario with intelligent vehicles that are controlled by the same decision logic. Each vehicle determines when to speed up, slow down, or change lanes based on the logic defined by a standalone Stateflow® chart. Because the driving conditions (including the relative position and speed of nearby vehicles) differ from vehicle to vehicle, separate chart objects in MATLAB® control the individual vehicles on the highway.

Open Driving Scenario

To start the example, run the script `sf_driver_demo.m`. The script displays a 3-D animation of a long highway and several vehicles. The view focuses on a single vehicle and its surroundings. As this vehicle moves along the highway, the standalone Stateflow chart `sf_driver` shows the decision logic that determines its actions.



Starting from a random position, each vehicle attempts to travel at a target speed. Because the target speeds are chosen at random, the vehicles can obstruct one another. In this situation, a vehicle will try to change lanes and resume its target speed.

The class file `HighwayScenario` defines a `drivingScenario (Automated Driving Toolbox)` object that represents the 3-D environment that contains the highway and the vehicles on it. To control the motion of the vehicles, the `drivingScenario` object creates an array of Stateflow chart objects. Each chart object controls a different vehicle in the simulation.

Execute Decision Logic for Vehicles

The Stateflow chart `sf_driver` consists of two top-level states, `LaneKeep` and `LaneChange`.

When the `LaneKeep` state is active, the corresponding vehicle stays in its lane of traffic. In this state, there are two possible substates:

- `Cruise` is active when the zone directly in front of the vehicle is empty and the vehicle can travel at its target speed.
- `Follow` becomes active when the zone directly in front of the vehicle is occupied and its target speed is faster than the speed of the vehicle in front. In this case, the vehicle is forced to slow down and attempt to change lanes.

When the `LaneChange` state is active, the corresponding vehicle attempts to change lanes. In this state, there are two possible substates:

- `Continue` is active when the zone next to the vehicle is empty and the vehicle can change lanes safely.
- `Abort` becomes active when the zone next to the vehicle is occupied. In this case, the vehicle is forced to remain in its lane.

```
{
frontF=0.9;
width=2.5;
positions = zeros(numCars,3);
velocities = zeros(numCars,3);
topLane = size(laneCenters,2);
myPos = positions(me,:);
}

LaneKeep

Cruise
myPos = positions(me,:);
[isZoneOccupied, zoneCar] = ...
    checkZone(myPos,0,0,maxSpeed,frontF,width+0.2,lWidth);
myVel = getVel(myVel, [maxSpeed 0 0],myPos,[nan nan nan], 0, maxSpeed,1);

[isZoneOccupied]

[~isZoneOccupied]
{isLaneChanging = false;}

Follow
{delay=0;
myPos=positions(me,:);}

[delay>20]

{
myPos=positions(me,:);
myLane = getLane(myPos(2),lWidth);
delay = delay+1;
}

{[isZoneOccupied,~] = ...
  checkZone(myPos,1,10,maxSpeed,frontF,width,lWidth);}

[~isZoneOccupied && (myLane < topLane)]

{[isZoneOccupied,~] = ...
  checkZone(myPos,-1,10,maxSpeed,frontF,width,lWidth);}

{deltaLane = 1;}

[~isZoneOccupied && myLane>1]

{deltaLane = -1;}

{[isZoneOccupied,zoneCar] = ...
  checkZone(myPos,0,0,maxSpeed,frontF,width,lWidth);}

{isLaneChanging = true;
isZoneOccupied = true;}

{myVel = getVel(myVel,velocities,myPos,positions,0,maxSpeed,zoneCar);}

MATLAB Function
[zoneOcc, idu] = checkZone(position,dLane,bSpot,desiredSpeed,frontFactor,sideWidth,laneWidth)

[~isLaneChanging]
{
deltaLane = 0;
errLane = 0;
isZoneOccupied = false;
}

function  myLane = getLane(yPos,laneWidth)

[isLaneChanging]
{
target=laneCenters(myLane+deltaLane);
slowCar = zoneCar;
}

MATLAB Function  v = getVel(vel,velocities,pos,positions,lane,maxSpeed,zoneCarIdx)

LaneChange
myPos=positions(me,:);
errLane = target-myPos(2);
if (deltaLane==1 & errLane<0.05) || (deltaLane==-1 & errLane>-0.05)
   isLaneChanging=false;
end
myVel = getVel(myVel, velocities, myPos, positions, errLane, maxSpeed, slowCar);

Continue
[isZoneOccupied, zoneCar] = ...
   checkZone(myPos,0.5*errLane,10,maxSpeed,0.5*frontF,width,lWidth);

[isZoneOccupied]

Abort
en:
deltaLane=-deltaLane;
target = laneCenters(myLane);
```

The transitions between the states `LaneKeep` and `LaneChange` are guarded by the value of `isLaneChanging`. In the `LaneKeep` state, the chart sets this local data to `true` when the substate `Follow` is active and there is enough room beside the vehicle to change lanes. In the `LaneChange` state, the chart sets this local data to `false` when the vehicle finishes changing lanes.

See Also

`drivingScenario`

More About

- “Create Stateflow Charts for Execution as MATLAB Objects” on page 31-2
- “Create Driving Scenario Programmatically” (Automated Driving Toolbox)
- “Create Actor and Vehicle Trajectories Programmatically” (Automated Driving Toolbox)
- “Define Road Layouts Programmatically” (Automated Driving Toolbox)

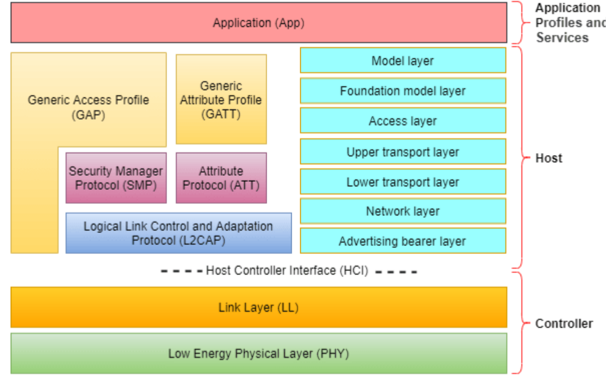
Model Bluetooth Low Energy Link Layer Using Stateflow

This example shows how to use a standalone Stateflow® chart to model the state machine in a Bluetooth® low energy (BLE) link layer.

Bluetooth technology is a wireless interface intended to replace the cables connecting portable and fixed electronic equipment. The Bluetooth Special Interest Group industry consortium defines two groups of standards for this technology: Bluetooth low energy (BLE) and Bluetooth basic rate/enhanced data rate (BR/EDR). BLE devices are characterized by low power consumption and low cost. These devices have an operating radio frequency in the range 2.4000 GHz to 2.4835 GHz. The operating band is divided into 40 channels, each with a bandwidth of 2 MHz. User data packets are transmitted using channels in the range from 0 to 36. Advertising data packets are transmitted in channels 37, 38, and 39.

The functionality of the BLE protocol stack is divided between three main layers:

- The controller layer includes the low energy physical layer (PHY), the link layer (LL), and the controller-side host controller interface (HCI). The state machine modeled by this example is part of the link layer in this portion of the BLE protocol stack.
- The host layer includes the host-side HCI, logical link control and adaptation protocol (L2CAP), attribute protocol (ATT), generic attribute profile (GATT), security manager protocol (SMP), and generic access profile (GAP). This layer also contains the BLE mesh stack, which consists of the advertising bearer, network, lower transport, upper transport, access, foundational model, and model layers.
- The application profiles and services layer (APP) is the user interface that defines usage profiles and enables interoperability between Bluetooth applications.



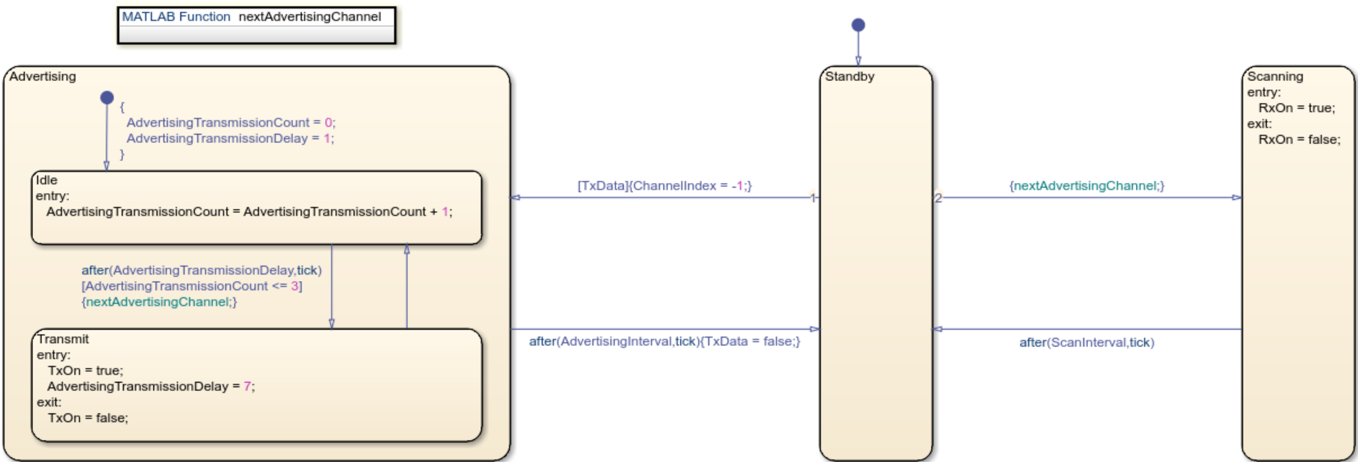
For more information, see “Bluetooth Technology Overview” (Bluetooth Toolbox), “Bluetooth Protocol Stack” (Bluetooth Toolbox), and “Bluetooth Mesh Networking” (Bluetooth Toolbox).

Determine the Operating Mode of the BLE Device

In the BLE protocol stack, the link layer manages the state of the radio to define the role of a device as advertiser or scanner. This layer interfaces directly with the PHY layer, which uses the three advertising channels (37, 38, and 39) to transmit and receive data packets.

In this example, the standalone Stateflow chart `sf_bluetooth.sfx` defines the operating modes of the link layer. The chart has three states: `Standby`, `Advertising`, and `Scanning`. Initially, the

Standby state is active, indicating that the device is idle. In the next time step, the chart transitions to the Advertising or Scanning state, depending on the value of the local data TxData. This value indicates whether there is data from the advertising bearer in the host layer available for transmission.



If there is data available, the chart transitions to the Advertising state. When this state is active, the device cycles through the three advertising channels and transmits the same data packet in each channel. The chart remains in the Advertising state for an advertising interval before it returns to the Standby state. The advertising interval is divided into three checkpoint timestamps that correspond to the three advertising channels. In this example, the advertising interval consists of a minimum of 20 milliseconds, with checkpoint timestamps at 1 millisecond (for channel 37), 9 milliseconds (for channel 38), and 17 milliseconds (for channel 39).

If there is no data available, the chart transitions to the Scanning state. When this state is active, the PHY layer passively scans one of the advertising channels for new data. If the PHY layer receives a data packet, the link layer passes it to the advertising bearer layer. The chart remains in the Scanning state for a scanning interval before it returns to the Standby state. At that point, if there is still no data available, the chart selects a new advertising channel and starts another scanning interval. In this example, the scanning interval consists of 50 milliseconds.

Simulate the BLE Link Layer

The script sf_bluetooth_demo.m creates a BLE link layer object bleLinkLayer and simulates 10 seconds of mesh communications over an advertising bearer. The link layer object relies on the standalone chart sf_bluetooth.sfx for control logic and on the link layer queue object bleQueue for storing data from the advertising bearer.

The simulation consists of 10,000 time steps. Each time step represents an execution of the link layer object (which corresponds to 1 millisecond of simulation time) and typically consists of these steps:

1. If the link layer queue is not empty, read a data packet. The data in the queue represents the advertising data packets that the link layer obtains from the advertising bearer layer.
2. Set the value of the chart data TxData and execute the standalone chart. In this step, the chart determines whether the device acts as an advertiser or scanner.
3. If the chart is in the Advertising.Transmit state, generate a BLE link layer advertising channel protocol data unit (PDU) using the data read from the link layer queue. The PDU represents a transmission by the PHY layer. For more information on generating and configuring

the advertising channel PDU, see `bleLLAdvertisingChannelPDU` (Bluetooth Toolbox) and `bleLLAdvertisingChannelPDUConfig` (Bluetooth Toolbox).

There are two exceptional cases:

- Every 1000 time steps, the script pushes data into the link layer queue before executing the link layer object. This data represents the advertising data packets that the link layer obtains from the advertising bearer layer.
- Every 2500 time steps, the script generates a BLE link layer advertising channel PDU that represents an advertising data packet received by the PHY layer. Then the script executes the link layer object. If the chart is in the `Scanning` state, the link layer object attempts to decode the PDU and, if the decoding is successful, passes the PDU to the advertising bearer layer. For more information on PDU decoding, see `bleLLAdvertisingChannelPDUDecode` (Bluetooth Toolbox).

When the simulation is complete, the script prints a summary that lists the number of data packets and bytes transmitted and received by the link layer.

```
Number of PDUs transmitted from link layer: 27
Number of bytes transmitted from link layer: 702
Number of PDUs received at link layer: 4
Number of bytes received at link layer: 112
```

During the simulation, the chart animation provides a visual indication of the runtime behavior of the algorithm. Note that chart animation slows down performance. To reduce the running time of the example, close the chart before running the script.

See Also

`bleLLAdvertisingChannelPDU` | `bleLLAdvertisingChannelPDUDecode` | `bleLLAdvertisingChannelPDUConfig`

More About

- “Create Stateflow Charts for Execution as MATLAB Objects” on page 31-2
- “Execute Stateflow Chart Objects Through Scripts and Models” on page 31-18
- “Bluetooth Technology Overview” (Bluetooth Toolbox)
- “Bluetooth Protocol Stack” (Bluetooth Toolbox)
- “Bluetooth Mesh Networking” (Bluetooth Toolbox)
- “Bluetooth Packet Structure” (Bluetooth Toolbox)

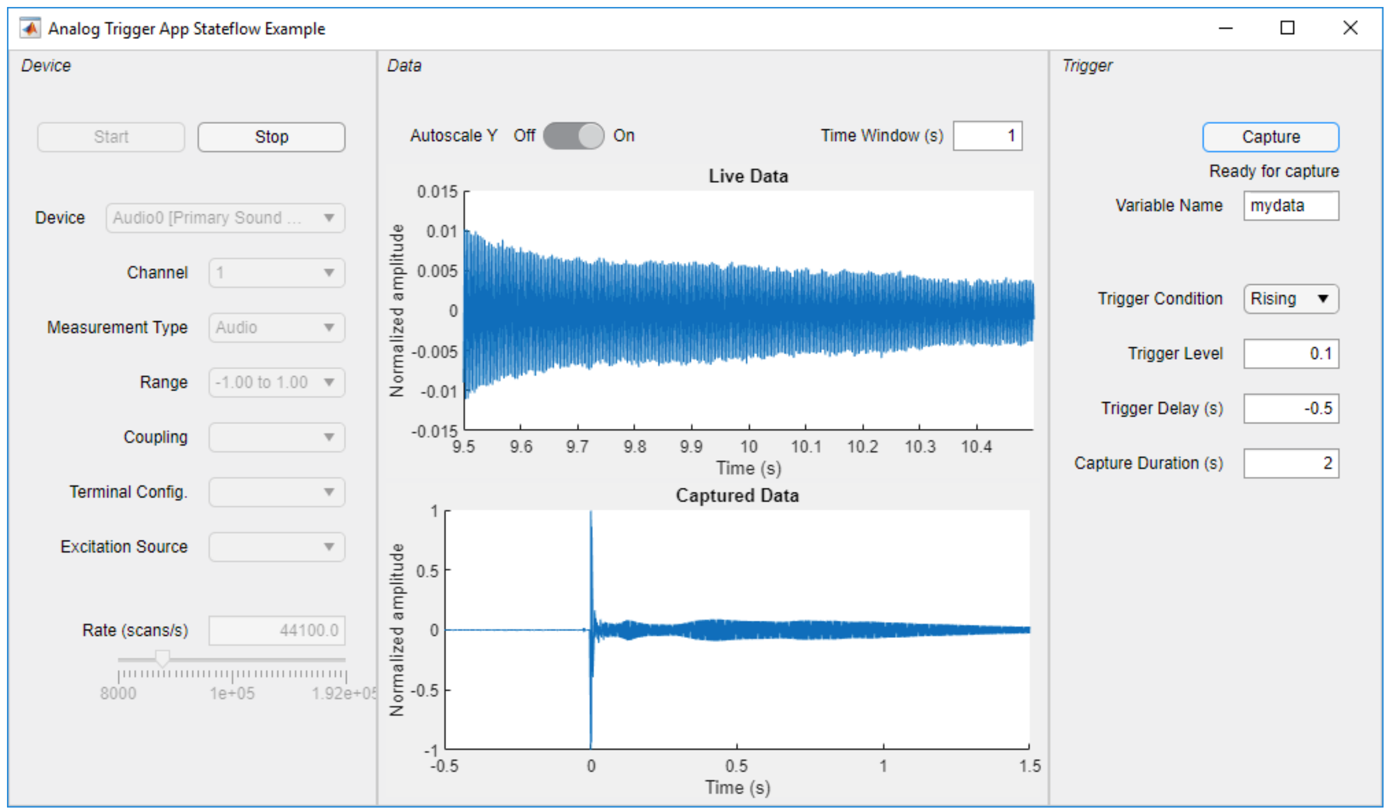
Analog Triggered Data Acquisition Using Stateflow Charts

This example shows how to create an analog-triggered data acquisition app by using Stateflow®, Data Acquisition Toolbox™, and App Designer.

Data Acquisition Toolbox provides functionality for acquiring measurement data from a DAQ device or audio soundcard. For certain applications, an analog-triggered acquisition that starts capturing or logging data based on a condition in the analog signal being measured is recommended. Software-analog triggered acquisition enables you to capture only a segment of interest out of a continuous stream of measurement data. For example, you can capture an audio recording when the signal level passes a certain threshold.

This example app, created by using App Designer and Stateflow, shows how to implement these operations:

- Control the app state logic by using a Stateflow chart.
- Discover available DAQ devices and select which device to use.
- Configure device acquisition parameters.
- Display a live plot in the app UI during acquisition.
- Perform a triggered data capture based on a programmable trigger condition.
- Save captured data to a MATLAB® base workspace variable.



By default, the app opens in design mode in App Designer. To run the app click the **Run** button or execute the app from the command line:

```
AnalogTriggerAppStateflow
```

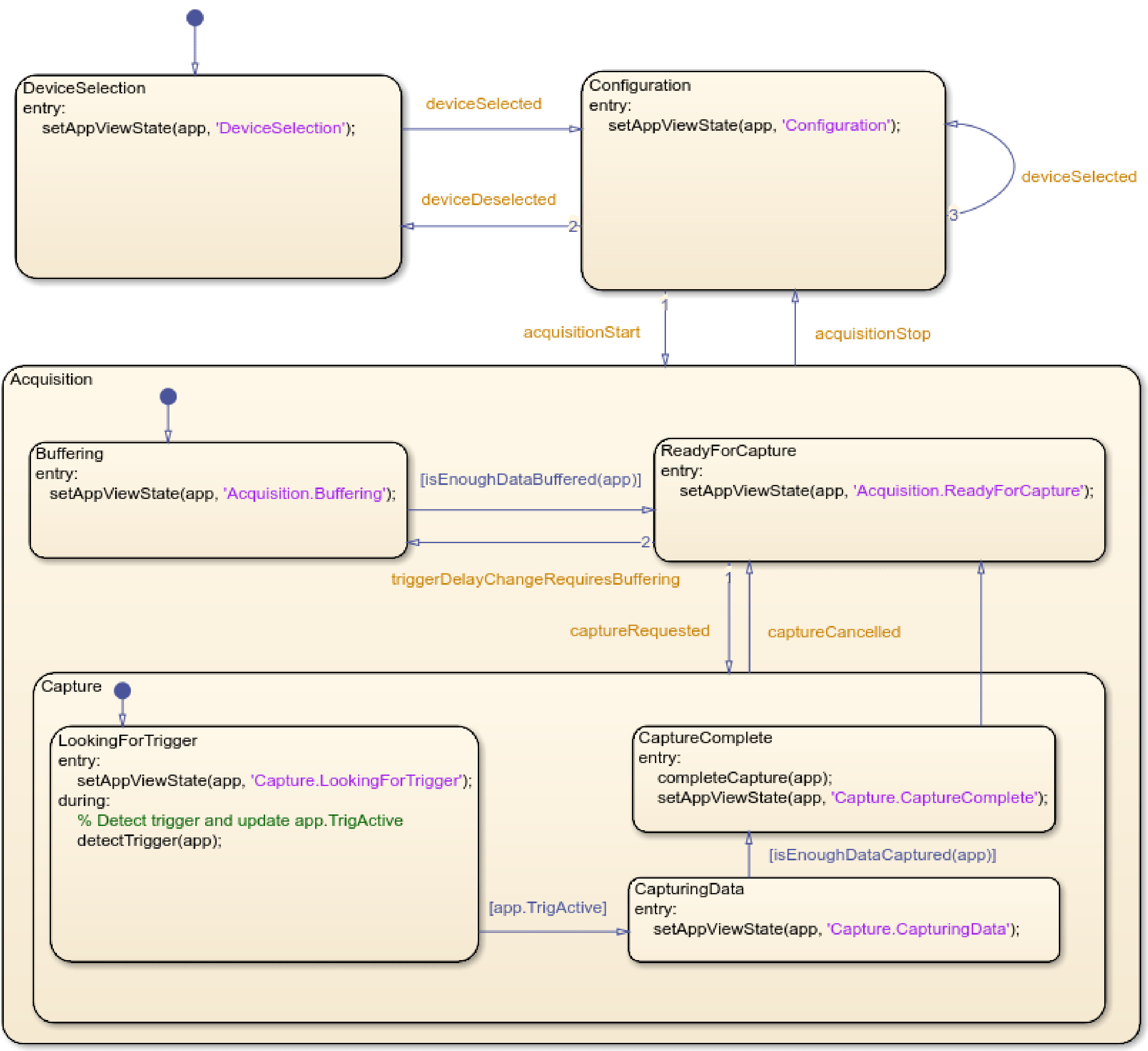
Requirements

This example app requires:

- MATLAB R2020a or later.
- Data Acquisition Toolbox (supported on Windows® only).
- Stateflow (for creating and editing charts only).
- A supported DAQ device or sound card. For example, any National Instruments or Measurement Computing device that supports analog input `Voltage` or `IEPE` measurements and background acquisition.
- Corresponding hardware support package and device drivers.

App States and the Stateflow Chart

When creating an app that has complex logic, consider the various states that correspond to the operating modes of the app. You can use a Stateflow chart to visualize and organize these app states. Use transitions between states to implement the control logic of your app. For example, the file `AnalogTriggerAppLogic.sfx` defines the Stateflow chart that controls the logic for this app. The chart can transition between states based on an action in the app UI or on a data-driven condition. For example, if you click the **Start** button, the chart transitions from the `Configuration` state to the `Acquisition` state. If the value of the signal crosses the specified trigger level, the chart transitions from the `LookingForTrigger` state to the `CapturingData` state.



Integrating the App with the Stateflow Chart

To establish a bidirectional connection between the MATLAB app and the Stateflow chart, in the `startupFcn` function of your app, create a chart object and store its handle in an app property.

```
app.Chart = AnalogTriggerAppLogic(app=app);
```

The app uses this handle to trigger state transitions in the chart. For example, when you click **Start**, the `StartButtonPushed` app callback function calls the `acquisitionStart` input event for the chart. This event triggers the transition from the `Configuration` state to the `Acquisition` state.

To evaluate transition conditions that are not events in the chart, the app calls the `step` function for the chart object. For example, while acquiring data from the device, the `dataAvailable_Callback`

app function periodically calls the `step` function. When the trigger condition is detected, the chart transitions from the `LookingForTrigger` State to the `CapturingData` state.

In the Stateflow chart, store the app object handle as chart local data. To share public properties and call public functions of the app, the Stateflow chart can use this handle in state actions, transition conditions, or transition actions.

See Also

Chart

More About

- “Design Human-Machine Interface Logic by Using Stateflow Charts” on page 31-24

Semantic Examples

Categories of Semantic Examples

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

"Transition Between Exclusive States" on page A-4

- "Transition from State to State with Events" on page A-4
- "Transition from a Substate to a Substate with Events" on page A-6

"Control Chart Execution by Using Condition Actions" on page A-8

- "Condition Action Behavior" on page A-8
- "Condition and Transition Action Behavior" on page A-8
- "Create Condition Actions Using a For-Loop" on page A-9
- "Broadcast Events to Parallel (AND) States Using Condition Actions" on page A-9
- "Avoid Cyclic Behavior" on page A-10

"Control Chart Execution by Using Default Transitions" on page A-12

- "Default Transition in Exclusive (OR) Decomposition" on page A-12
- "Default Transition to a Junction" on page A-12
- "Default Transition and a History Junction" on page A-13
- "Labeled Default Transitions" on page A-14

"Process Events in States Containing Inner Transitions" on page A-17

- "Process One Event in an Exclusive (OR) State" on page A-17
- "Process a Second Event in an Exclusive (OR) State" on page A-17
- "Process a Third Event in an Exclusive (OR) State" on page A-18
- "Process the First Event with an Inner Transition to a Connective Junction" on page A-18
- "Process a Second Event with an Inner Transition to a Connective Junction" on page A-19
- "Inner Transition to a History Junction" on page A-20

"Represent Multiple Paths by Using Connective Junctions" on page A-22

- "If-Then-Else Decision Construct" on page A-22
- "Self-Loop Transition" on page A-23
- "For-Loop Construct" on page A-24
- "Flow Chart Notation" on page A-24
- "Transition from a Common Source to Multiple Destinations" on page A-25
- "Transition from Multiple Sources to a Common Destination" on page A-27
- "Transition from a Source to a Destination Based on a Common Event" on page A-27

"Control Chart Execution by Using Event Actions in a Superstate" on page A-29

"Undirected Broadcast Events in Parallel States" on page A-30

- "Broadcast Events in State Actions" on page A-30

- "Broadcast Events in Transition Actions" on page A-31
- "Broadcast Events in Condition Actions" on page A-33

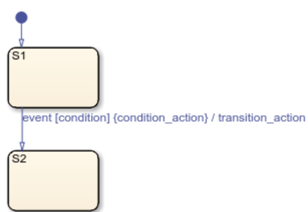
"Broadcast Local Events in Parallel States" on page A-35

- "Directed Event Broadcast Using Send" on page A-35
- "Directed Event Broadcast Using Qualified Event Name" on page A-35

Transition Between Exclusive States

Label Format for a State-to-State Transition

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

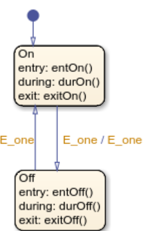


A chart executes this transition as follows:

1. When an event occurs, state `S1` checks for an outgoing transition with a matching event specified.
2. If a transition with a matching event is found, the condition for that transition (`[condition]`) is evaluated.
3. If the `condition` is true, `condition_action` is executed.
4. If there is a valid transition to the destination state, the transition is taken.
5. State `S1` is exited.
6. The `transition_action` is executed when the transition is taken.
7. State `S2` is entered.

Transition from State to State with Events

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



Process a First Event

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

1. The chart root checks to see if there is a valid transition as a result of `E_one`. A valid transition from state `On` to state `Off` is detected.
2. State `On` exit actions (`exitOn()`) execute and complete.
3. State `On` is marked inactive.

4 The event `E_one` is broadcast as the transition action.

   This second event `E_one` is processed, but because neither state is active, it has no effect. If the second broadcast of `E_one` resulted in a valid transition, it would preempt the processing of the first broadcast of `E_one`. See "Early Return Logic" on page 2-41.

5 State `Off` is marked active.

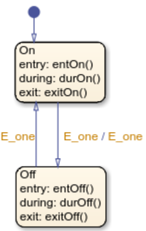
6 State `Off` entry actions (`entOff()`) execute and complete.

7 The chart goes back to sleep.

This sequence completes the execution of the Stateflow chart associated with event `E_one` when state `On` is initially active.

Process a Second Event

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



Initially, the chart is asleep. State `Off` is active. Event `E_one` occurs and awakens the chart, which processes the event from the root down through the hierarchy:

1 The chart root checks to see if there is a valid transition as a result of `E_one`.

   A valid transition from state `Off` to state `On` is detected.

2 State `Off` exit actions (`exitOff()`) execute and complete.

3 State `Off` is marked inactive.

4 State `On` is marked active.

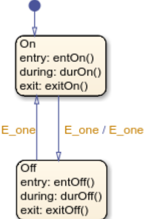
5 State `On` entry actions (`entOn()`) execute and complete.

6 The chart goes back to sleep.

This sequence completes the execution of the Stateflow chart associated with the second event `E_one` when state `Off` is initially active.

Process a Third Event

Using the same example, what happens when a third event, `E_two`, occurs?



Notice that the event E_two is not used explicitly in this example. However, its occurrence (or the occurrence of any event) does result in behavior. Initially, the chart is asleep and state On is active.

1. Event E_two occurs and awakens the chart.

   Event E_two is processed from the root of the chart down through the hierarchy of the chart.
2. The chart root checks to see if there is a valid transition as a result of E_two. There is none.
3. State On during actions (durOn()) execute and complete.
4. The chart goes back to sleep.

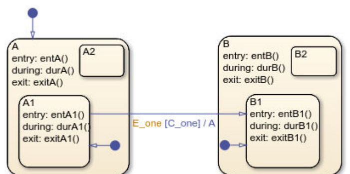
This sequence completes the execution of the Stateflow chart associated with event E_two when state On is initially active.

Tip Avoid using undirected local event broadcasts. Undirected local event broadcasts can cause unwanted recursive behavior in your chart. Instead, send local events by using directed broadcasts. For more information, see "Broadcast Local Events to Synchronize Parallel States" on page 12-25.

During simulation, Stateflow charts can detect undirected local event broadcasts. To control the level of diagnostic action, open the Configuration Parameters dialog box and, in the **Diagnostics** > **Stateflow** pane, set the **Undirected event broadcasts** parameter to none, warning, or error. The default setting is warning. For more information, see "Undirected event broadcasts" (Simulink).

Transition from a Substate to a Substate with Events

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



Initially, the chart is asleep. State A.A1 is active. Condition C_one is true. Event E_one occurs and awakens the chart, which processes the event from the root down through the hierarchy:

1. The chart root checks to see if there is a valid transition as a result of E_one. There is a valid transition from state A.A1 to state B.B1. (Condition C_one is true.)
2. State A during actions (durA()) execute and complete.
3. State A.A1 exit actions (exitA1()) execute and complete.
4. State A.A1 is marked inactive.
5. State A exit actions (exitA()) execute and complete.
6. State A is marked inactive.
7. The transition action, A, is executed and completed.
8. State B is marked active.
9. State B entry actions (entB()) execute and complete.
10. State B.B1 is marked active.

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

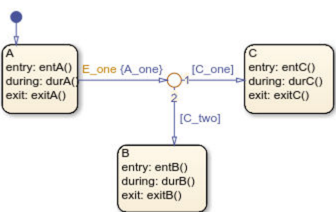
12 The chart goes back to sleep.

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

Control Chart Execution by Using Condition Actions

Condition Action Behavior

This example shows the behavior of a simple condition action in a transition path with multiple segments. The chart uses implicit ordering of outgoing transitions (see “Implicit Ordering” on page 2-28).



Initially, the chart is asleep. State A is active. Conditions C_one and C_two are false. Event E_one occurs and awakens the chart, which processes the event from the root down through the hierarchy:

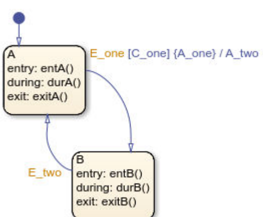
1. The chart root checks to see if there is a valid transition as a result of E_one. A valid transition segment from state A to a connective junction is detected. The condition action A_one is detected on the valid transition segment and is immediately executed and completed. State A is still active.
2. Because the conditions on the transition segments to possible destinations are false, none of the complete transitions is valid.
3. State A during actions (durA()) execute and complete.

   State A remains active.
4. The chart goes back to sleep.

This sequence completes the execution of this Stateflow chart associated with event E_one when state A is initially active.

Condition and Transition Action Behavior

This example shows the behavior of a simple condition and transition action specified on a transition from one exclusive (OR) state to another.



Initially, the chart is asleep. State A is active. Condition C_one is true. Event E_one occurs and awakens the chart, which processes the event from the root down through the hierarchy:

1. The chart root checks to see if there is a valid transition as a result of E_one. A valid transition from state A to state B is detected. The condition C_one is true. The condition action A_one is

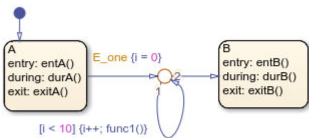
detected on the valid transition and is immediately executed and completed. State A is still active.

2. State A exit actions (`ExitA()`) execute and complete.
3. State A is marked inactive.
4. The transition action `A_two` is executed and completed.
5. State B is marked active.
6. State B entry actions (`entB()`) execute and complete.
7. The chart goes back to sleep.

This sequence completes the execution of this Stateflow chart associated with event `E_one` when state A is initially active.

Create Condition Actions Using a For-Loop

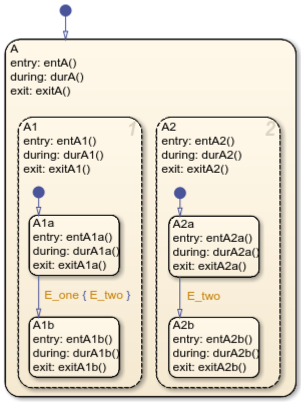
Condition actions and connective junctions are used to design a `for` loop construct. This example shows the use of a condition action and connective junction to create a `for` loop construct. The chart uses implicit ordering of outgoing transitions (see "Implicit Ordering" on page 2-28).



See "For-Loop Construct" on page A-24 to see the behavior of this example.

Broadcast Events to Parallel (AND) States Using Condition Actions

This example shows how to use condition actions to broadcast events immediately to parallel (AND) states. The chart uses implicit ordering of parallel states (see "Implicit Ordering of Parallel States" on page 2-47).



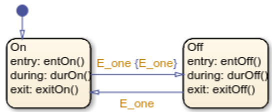
See "Broadcast Events in Condition Actions" on page A-33 to see the behavior of this example.

Tip Avoid using undirected local event broadcasts. Undirected local event broadcasts can cause unwanted recursive behavior in your chart. Instead, send local events by using directed broadcasts. For more information, see “Broadcast Local Events to Synchronize Parallel States” on page 12-25.

During simulation, Stateflow charts can detect undirected local event broadcasts. To control the level of diagnostic action, open the Configuration Parameters dialog box and, in the **Diagnostics** > **Stateflow** pane, set the **Undirected event broadcasts** parameter to `none`, `warning`, or `error`. The default setting is `warning`. For more information, see “Undirected event broadcasts” (Simulink).

Avoid Cyclic Behavior

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



Initially, the chart is asleep. State `On` is active. Event `E_one` occurs and awakens the chart, which processes the event from the root down through the hierarchy:

1. The chart root checks to see if there is a valid transition as a result of `E_one`.

   A valid transition from state `On` to state `Off` is detected.
2. The condition action on the transition broadcasts event `E_one`.
3. Event `E_one` is detected on the valid transition, which is immediately executed. State `On` is still active.
4. The broadcast of event `E_one` awakens the chart a second time.
5. Go to step 1.

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

Tip Avoid using undirected local event broadcasts. Undirected local event broadcasts can cause unwanted recursive behavior in your chart. Instead, send local events by using directed broadcasts. For more information, see “Broadcast Local Events to Synchronize Parallel States” on page 12-25.

During simulation, Stateflow charts can detect undirected local event broadcasts. To control the level of diagnostic action, open the Configuration Parameters dialog box and, in the **Diagnostics** > **Stateflow** pane, set the **Undirected event broadcasts** parameter to `none`, `warning`, or `error`. The default setting is `warning`. For more information, see “Undirected event broadcasts” (Simulink).

See Also

More About

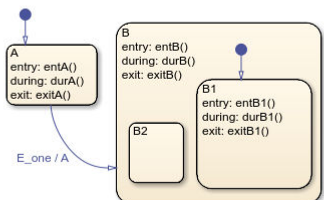
- “Transition Between Operating Modes” on page 1-37

- "Broadcast Local Events to Synchronize Parallel States" on page 12-25
- "Operations for Stateflow Data" on page 14-4

Control Chart Execution by Using Default Transitions

Default Transition in Exclusive (OR) Decomposition

This example shows a transition from an OR state to a superstate with exclusive (OR) decomposition, where a default transition to a substate is defined.



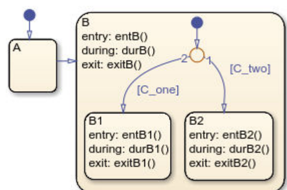
Initially, the chart is asleep. State A is active. Event E_one occurs and awakens the chart, which processes the event from the root down through the hierarchy:

1. The chart root checks to see if there is a valid transition as a result of E_one. There is a valid transition from state A to superstate B.
2. State A exit actions (exitA()) execute and complete.
3. State A is marked inactive.
4. The transition action, A, is executed and completed.
5. State B is marked active.
6. State B entry actions (entB()) execute and complete.
7. State B detects a valid default transition to state B.B1.
8. State B.B1 is marked active.
9. State B.B1 entry actions (entB1()) execute and complete.
10. The chart goes back to sleep.

This sequence completes the execution of this Stateflow chart associated with event E_one when state A is initially active.

Default Transition to a Junction

The following example shows the behavior of a default transition to a connective junction. The default transition to the junction is valid only when state B is first entered, not every time the chart wakes up.



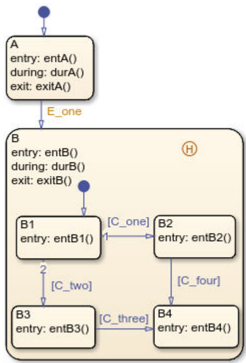
For this example, initially, the chart is asleep. State B.B1 is active. Condition [C_two] is true. An event occurs and awakens the chart, which processes the event from the root down through the hierarchy:

1. State B checks to see if there is a valid transition as a result of any event. There is none.
2. State B during actions (durB()) execute and complete.
3. State B1 checks to see if there is a valid transition as a result of any event. There is none.
4. State B1 during actions (durB1()) execute and complete.

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

Default Transition and a History Junction

This example shows the behavior of a superstate with a default transition and a history junction. The chart uses implicit ordering of outgoing transitions (see “Implicit Ordering” on page 2-28).



Initially, the chart is asleep. State A is active. A history junction records the fact that state B4 is the previously active substate of superstate B. Event E_one occurs and awakens the chart, which processes the event from the root down through the hierarchy:

1. The chart root checks to see if there is a valid transition as a result of E_one.

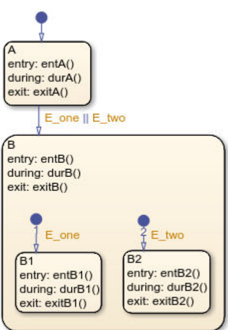
   There is a valid transition from state A to superstate B.
2. State A exit actions (exitA()) execute and complete.
3. State A is marked inactive.
4. State B is marked active.
5. State B entry actions (entB()) execute and complete.
6. State B uses the history junction to determine the substate destination of the transition into the superstate.

   The history junction indicates that substate B.B4 was the last active substate, which becomes the destination of the transition.
7. State B.B4 is marked active.
8. State B.B4 entry actions (entB4()) execute and complete.
9. The chart goes back to sleep.

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

Labeled Default Transitions

This example shows the use of a default transition with a label. The chart uses implicit ordering of outgoing transitions (see “Implicit Ordering” on page 2-28).



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.

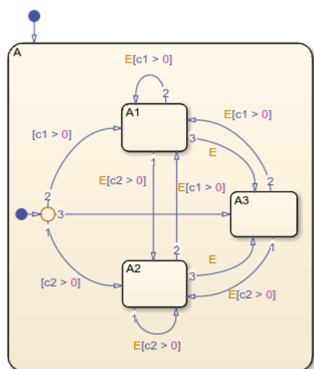
Control Chart Execution by Using Inner Transitions

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 A-15
- "After Using an Inner Transition to a Connective Junction" on page A-15
- "Using an Inner Transition to a History Junction" on page A-16

Before Using an Inner Transition

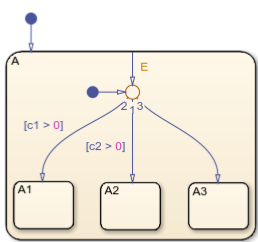
This chart is an example of how you can simplify logic using an inner transition.



Any event occurs and awakens the Stateflow chart. The default transition to the connective junction is valid. The destination of the transition is determined by `[c1 > 0]` and `[c2 > 0]`. If `[c1 > 0]` is true, the transition to `A1` is true. If `[c2 > 0]` is true, the transition to `A2` is valid. If neither `[c1 > 0]` nor `[c2 > 0]` is true, the transition to `A3` is valid. The transitions among `A1`, `A2`, and `A3` are determined by `E`, `[c1 > 0]`, and `[c2 > 0]`.

After Using an Inner Transition to a Connective Junction

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



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

You can simplify the chart by using an inner transition in place of the transitions among all the states in the original example. If state `A` is already active, the inner transition is used to reevaluate which of

the substates of state A is to be active. When event E occurs, the inner transition is potentially valid. If [c1 > 0] is true, the transition to A1 is valid. If [c2 > 0] is true, the transition to A2 is valid. If neither [c1 > 0] nor [c2 > 0] is true, the transition to A3 is valid. This chart design is simpler than the previous one.

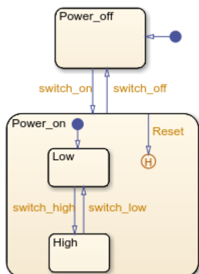
Note When you use an inner transition to a connective junction, an active substate can exit and reenter when the transition condition for that substate is valid. For example, if substate A1 is active and [c1 > 0] is true, the transition to A1 is valid. In this case:

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

See “Process the First Event with an Inner Transition to a Connective Junction” on page A-18 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 “Inner Transition to a History Junction” on page A-20 for more information on the semantics of this notation.

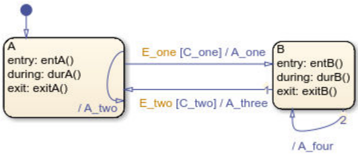
Process Events in States Containing Inner Transitions

Process Events with an Inner Transition in an Exclusive (OR) State

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

Process One Event in an Exclusive (OR) State

This example shows the behavior of an inner transition. The chart uses implicit ordering of outgoing transitions (see “Implicit Ordering” on page 2-28).



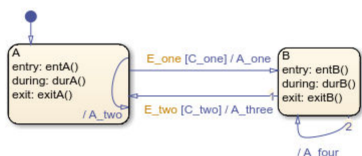
Initially, the chart is asleep. State A is active. Condition [C_one] is false. Event E_one occurs and awakens the chart, which processes the event from the root down through the hierarchy:

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, the chart does not execute the exit and entry actions for state A.
5. The chart goes back to sleep.

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

Process a Second Event in an Exclusive (OR) State

Using the previous example, this example shows what happens when a second event E_one occurs. The chart uses implicit ordering of outgoing transitions (see “Implicit Ordering” on page 2-28).



Initially, the chart is asleep. State A is still active. Condition [C_one] is true. Event E_one occurs and awakens the chart, which processes the event from the root down through the hierarchy:

1. The chart root checks to see if there is a valid transition as a result of E_one.

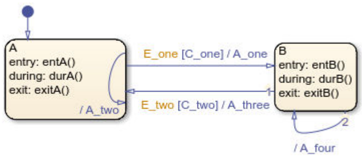
   The transition from state A to state B is now valid because [C_one] is true.
2. State A exit actions (exitA()) execute and complete.

3. State A is marked inactive.
4. The transition action A_one is executed and completed.
5. State B is marked active.
6. State B entry actions (entB()) execute and complete.
7. The chart goes back to sleep.

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

Process a Third Event in an Exclusive (OR) State

Using the previous example, this example shows what happens when a third event, E_two, occurs. The chart uses implicit ordering of outgoing transitions (see “Implicit Ordering” on page 2-28).



Initially, the chart is asleep. State B is now active. Condition [C_two] is false. Event E_two occurs and awakens the chart, which processes the event from the root down through the hierarchy:

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

   A potentially valid transition from state B to state A is detected. The transition is not valid because [C_two] is false. However, active state B has a valid self-loop transition.
2. State B exit actions (exitB()) execute and complete.
3. State B is marked inactive.
4. The self-loop transition action, A_four, executes and completes.
5. State B is marked active.
6. State B entry actions (entB()) execute and complete.
7. The chart goes back to sleep.

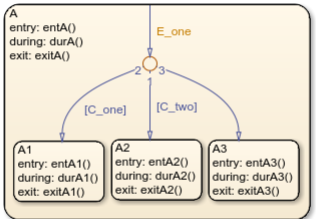
This sequence completes the execution of this Stateflow chart associated with event E_two. This example shows the difference in behavior between inner and self-loop transitions.

Process Events with an Inner Transition to a Connective Junction

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

Process the First Event with an Inner Transition to a Connective Junction

This example shows the behavior of an inner transition to a connective junction for the first event. The chart uses implicit ordering of outgoing transitions (see “Implicit Ordering” on page 2-28).



Initially, the chart is asleep. State A1 is active. Condition [C_two] is true. Event E_one occurs and awakens the chart, which processes the event from the root down through the hierarchy:

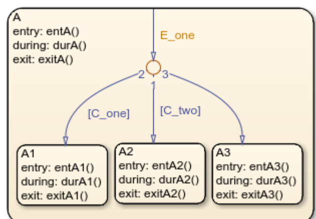
1. The chart root checks to see if there is a valid transition at the root level as a result of E_one. There is no valid transition.
2. State A during actions (durA()) execute and complete.
3. State A checks itself for valid transitions and detects that there is a valid inner transition to a connective junction.

   The conditions are evaluated to determine whether one of the transitions is valid. Because implicit ordering applies, the segments labeled with a condition are evaluated before the unlabeled segment. The evaluation starts from a 12 o'clock position on the junction and progresses in a clockwise manner. Because [C_two] is true, the inner transition to the junction and then to state A.A2 is valid.
4. State A.A1 exit actions (exitA1()) execute and complete.
5. State A.A1 is marked inactive.
6. State A.A2 is marked active.
7. State A.A2 entry actions (entA2()) execute and complete.
8. The chart goes back to sleep.

This sequence completes the execution of this Stateflow chart associated with event E_one when state A1 is active and condition [C_two] is true.

Process a Second Event with an Inner Transition to a Connective Junction

Continuing the previous example, this example shows the behavior of an inner transition to a junction when a second event E_one occurs. The chart uses implicit ordering of outgoing transitions (see “Implicit Ordering” on page 2-28).



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.

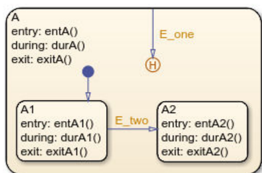
7 State A.A2 entry actions (entA2()) execute and complete.

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

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.

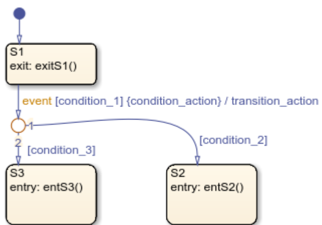
8 The chart goes back to sleep.

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.

Represent Multiple Paths by Using Connective Junctions

Label Format for Transition Segments

The label format for a transition segment entering a junction is the same as for transitions entering states, as shown in the following example. The chart uses implicit ordering of outgoing transitions (see “Implicit Ordering” on page 2-28).

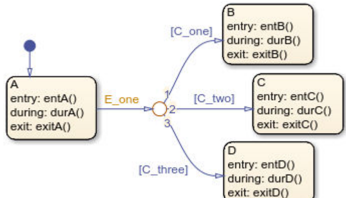


Execution of a transition in this example occurs as follows:

1. When an event occurs, state `S1` is checked for an outgoing transition with a matching event specified.
2. If a transition with a matching event is found, the transition condition for that transition (in brackets) is evaluated.
3. If `condition_1` evaluates to true, the condition action `condition_action` (in braces) is executed.
4. The outgoing transitions from the junction are checked for a valid transition. Since `condition_2` is true, a valid state-to-state transition from `S1` to `S2` exists.
5. State `S1` exit actions execute and complete.
6. State `S1` is marked inactive.
7. The transition action `transition_action` executes and completes.
8. The completed state-to-state transition from `S1` to `S2` occurs.
9. State `S2` is marked active.
10. State `S2` entry actions execute and complete.

If-Then-Else Decision Construct

This example shows the behavior of an `if-then-else` decision construct. The chart uses implicit ordering of outgoing transitions (see “Implicit Ordering” on page 2-28).



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

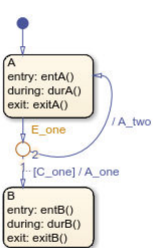
1. The chart root checks to see if there is a valid transition as a result of E_one.

   A valid transition segment from state A to the connective junction exists. Because implicit ordering applies, the transition segments beginning from a 12 o'clock position on the connective junction are evaluated for validity. The first transition segment, labeled with condition [C_one], is not valid. The next transition segment, labeled with the condition [C_two], is valid. The complete transition from state A to state C is valid.
2. State A exit actions (exitA()) execute and complete.
3. State A is marked inactive.
4. State C is marked active.
5. State C entry actions (entC()) execute and complete.
6. The chart goes back to sleep.

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

Self-Loop Transition

This example shows the behavior of a self-loop transition using a connective junction. The chart uses implicit ordering of outgoing transitions (see “Implicit Ordering” on page 2-28).



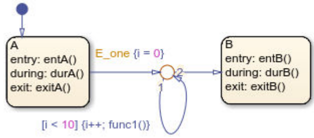
Initially, the chart is asleep. State A is active. Condition [C_one] is false. Event E_one occurs and awakens the chart, which processes the event from the root down through the hierarchy:

1. The chart root checks to see if there is a valid transition as a result of E_one. A valid transition segment from state A to the connective junction exists. Because implicit ordering applies, the transition segment labeled with a condition is evaluated for validity. Because the condition [C_one] is not valid, the complete transition from state A to state B is not valid. The transition segment from the connective junction back to state A is valid.
2. State A exit actions (exitA()) execute and complete.
3. State A is marked inactive.
4. The transition action A_two is executed and completed.
5. State A is marked active.
6. State A entry actions (entA()) execute and complete.
7. The chart goes back to sleep.

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

For-Loop Construct

This example shows the behavior of a `for` loop using a connective junction. The chart uses implicit ordering of outgoing transitions (see “Implicit Ordering” on page 2-28).



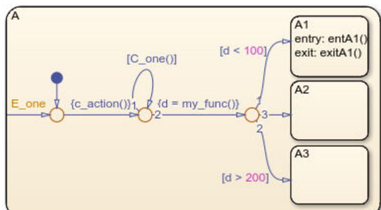
Initially, the chart is asleep. State `A` is active. Event `E_one` occurs and awakens the chart, which processes the event from the root down through the hierarchy:

1. The chart root checks to see if there is a valid transition as a result of `E_one`. There is a valid transition segment from state `A` to the connective junction. The transition segment condition action, `i = 0`, executes and completes. Of the two transition segments leaving the connective junction, the transition segment that is a self-loop back to the connective junction evaluates next for validity. That segment takes priority in evaluation because it has a condition, whereas the other segment is unlabeled. This evaluation behavior reflects implicit ordering of outgoing transitions in the chart.
2. The condition `[i < 10]` evaluates as true. The condition actions `i++` and a call to `func1` execute and complete until the condition becomes false. Because a connective junction is not a final destination, the transition destination is still unknown.
3. The unconditional segment to state `B` is now valid. The complete transition from state `A` to state `B` is valid.
4. State `A` exit actions (`exitA()`) execute and complete.
5. State `A` is marked inactive.
6. State `B` is marked active.
7. State `B` entry actions (`entB()`) execute and complete.
8. The chart goes back to sleep.

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

Flow Chart Notation

This example shows the behavior of a Stateflow chart that uses flow chart notation. The chart uses implicit ordering of outgoing transitions (see “Implicit Ordering” on page 2-28).



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:

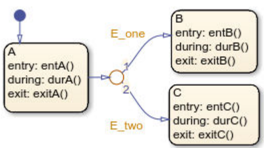
1. The chart root checks to see if there is a valid transition as a result of `E_one`. There is no valid transition.
2. State `A` checks itself for valid transitions and detects a valid inner transition to a connective junction.
3. The next possible segments of the transition are evaluated. Only one 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`.

Transition from a Common Source to Multiple Destinations

This example shows the behavior of transitions from a common source to multiple conditional destinations using a connective junction. The chart uses implicit ordering of outgoing transitions (see “Implicit Ordering” on page 2-28).



Initially, the chart is asleep. State `A` is active. Event `E_two` occurs and awakens the chart, which processes the event from the root down through the hierarchy:

1. The chart root checks to see if there is a valid transition as a result of `E_two`. A valid transition segment exists from state `A` to the connective junction. Because implicit ordering applies, evaluation of segments with equivalent label priority begins from a 12 o'clock position on the connective junction and progresses clockwise. The first transition segment, labeled with event `E_one`, is not valid. The next transition segment, labeled with event `E_two`, is valid. The complete transition from state `A` to state `C` is valid.

2 State A exit actions (exitA()) execute and complete.
3 State A is marked inactive.
4 State C is marked active.
5 State C entry actions (entC()) execute and complete.
6 The chart goes back to sleep.

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

Resolve Equally Valid Transition Paths

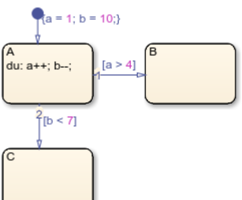
What Are Conflicting Transitions?

Conflicting transitions are two equally valid paths from the same source in a Stateflow chart during simulation. In the case of a conflict, the chart evaluates equally valid transitions based on ordering mode in the chart: explicit or implicit.

- For explicit ordering (the default mode), evaluation of conflicting transitions occurs based on the order you specify for each transition. For details, see “Explicit Ordering” on page 2-28.
- For implicit ordering in C charts, evaluation of conflicting transitions occurs based on internal rules described in “Implicit Ordering” on page 2-28.

Example of Conflicting Transitions

The following chart has two equally valid transition paths:



Conflict Resolution for Implicit Ordering

For implicit ordering, the chart evaluates multiple outgoing transitions with equal label priority in a clockwise progression starting from the twelve o'clock position on the state. In this case, the transition from state A to state B occurs.

Conflict Resolution for Explicit Ordering

For explicit ordering, the chart resolves the conflict by evaluating outgoing transitions in the order that you specify explicitly. For example, if you right-click the transition from state A to state C and select **Execution Order** > **1** from the context menu, the chart evaluates that transition first. In this case, the transition from state A to state C occurs.

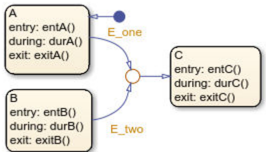
How the Transition Conflict Occurs

The default transition to state A assigns data a equal to 1 and data b equal to 10. The during action of state A increments a and decrements b during each time step. The transition from state A to state B is valid if the condition [a > 4] is true. The transition from state A to state C is valid if the

condition [b < 7] is true. During simulation, there is a time step where state A is active and both conditions are true. This issue is a transition conflict.

Transition from Multiple Sources to a Common Destination

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



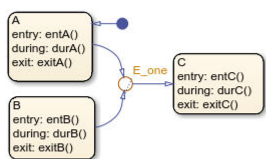
Initially, the chart is asleep. State A is active. Event E_one occurs and awakens the chart, which processes the event from the root down through the hierarchy:

1. The chart root checks to see if there is a valid transition as a result of E_one. A valid transition segment exists from state A to the connective junction and from the junction to state C.
2. State A exit actions (exitA()) execute and complete.
3. State A is marked inactive.
4. State C is marked active.
5. State C entry actions (entC()) execute and complete.
6. The chart goes back to sleep.

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

Transition from a Source to a Destination Based on a Common Event

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



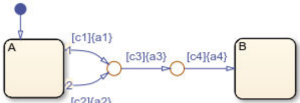
Initially, the chart is asleep. State B is active. Event E_one occurs and awakens the chart, which processes the event from the root down through the hierarchy:

1. The chart root checks to see if there is a valid transition as a result of E_one. A valid transition segment exists from state B to the connective junction and from the junction to state C.
2. State B exit actions (exitB()) execute and complete.
3. State B is marked inactive.
4. State C is marked active.
5. State C entry actions (entC()) execute and complete.
6. The chart goes back to sleep.

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

Backtrack in Flow Charts

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



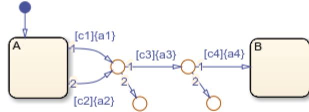
Initially, state A is active, conditions c1, c2, and c3 are true, and condition c4 is false:

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 expected 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.



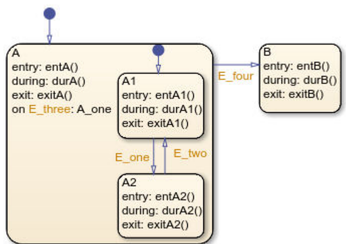
The terminating junctions allow flow to end if either c3 or c4 is not true. This design leaves state A active without executing unnecessary actions.

Additional Examples of Unintended Backtrack

Open this model to see additional examples of unintended backtracking in flow charts.

Control Chart Execution by Using Event Actions in a Superstate

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



Initially, the chart is asleep. State `A.A1` is active. Event `E_three` occurs and awakens the chart, which processes the event from the root down through the hierarchy:

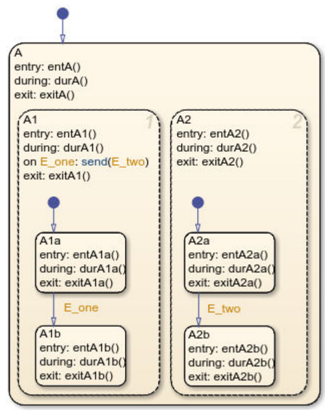
1. The chart root checks to see if there is a valid transition as a result of `E_three`. No valid transition exists.
2. State `A` during actions (`durA()`) execute and complete.
3. State `A` executes and completes the `on` event `E_three` action (`A_one`).
4. State `A` checks its children for valid transitions. No valid transitions exist.
5. State `A1` during actions (`durA1()`) execute and complete.
6. The chart goes back to sleep.

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

Undirected Broadcast Events in Parallel States

Broadcast Events in State Actions

This example shows the behavior of event broadcast actions in parallel states. The chart uses implicit ordering of parallel states (see “Implicit Ordering of Parallel States” on page 2-47).



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 evaluates its children 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. The `during` actions for state `A1a` (`durA1a()`) execute.
   f. State `A.A2` is evaluated. State `A.A2` `during` actions (`durA2()`) execute and complete. State `A.A2` checks for valid transitions. State `A.A2` has a valid transition as a result of `E_two` from state `A.A2`.A2a to state `A.A2.A2b`.
   g. State `A.A2.A2a` `exit` actions (`exitA2a()`) execute and complete.
   h. State `A.A2.A2a` is marked inactive.
   i. State `A.A2.A2b` is marked active.
   j. State `A.A2.A2b` `entry` actions (`entA2b()`) execute and complete.

4. The processing of E_one continues once the on event broadcast of E_two has been processed. State A.A1 checks for any valid transitions as a result of event E_one. A valid transition exists from state A.A1.A1a to state A.A1.A1b.
5. State A.A1.A1a executes and completes exit actions (exitA1a).
6. State A.A1.A1a is marked inactive.
7. State A.A1.A1b is marked active.
8. State A.A1.A1b entry actions (entA1b()) execute and complete.
9. Parallel state A.A2 is evaluated next. State A.A2 during actions (durA2()) execute and complete. There are no valid transitions as a result of E_one.
10. State A.A2.A2b during actions (durA2b()) execute and complete.

    State A.A2.A2b is now active as a result of the processing of the on event broadcast of E_two.
11. The chart goes back to sleep.

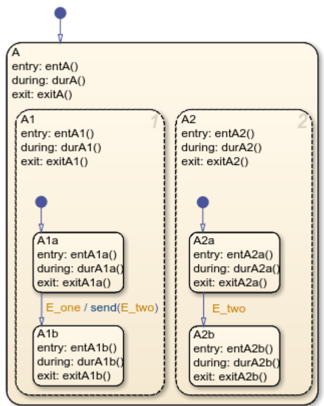
This sequence completes the execution of this Stateflow chart associated with event E_one and the on event broadcast to a parallel state of event E_two. The final chart activity is that parallel substates A.A1.A1b and A.A2.A2b are active.

Tip Avoid using undirected local event broadcasts. Undirected local event broadcasts can cause unwanted recursive behavior in your chart. Instead, send local events by using directed broadcasts. For more information, see “Broadcast Local Events to Synchronize Parallel States” on page 12-25.

During simulation, Stateflow charts can detect undirected local event broadcasts. To control the level of diagnostic action, open the Configuration Parameters dialog box and, in the **Diagnostics** > **Stateflow** pane, set the **Undirected event broadcasts** parameter to none, warning, or error. The default setting is warning. For more information, see “Undirected event broadcasts” (Simulink).

Broadcast Events in Transition Actions

This example shows the behavior of an event broadcast transition action that includes a nested event broadcast in a parallel state. The chart uses implicit ordering of parallel states (see “Implicit Ordering of Parallel States” on page 2-47).



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 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.A1during actions (durA1()) execute and complete.

4 State A.A1 checks for any valid transitions as a result of event E_one. There is a valid transition from state A.A1.A1a to state A.A1.A1b.

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

6 State A.A1.A1a is marked inactive.

Event E_two Preempts E_one

1 The transition action that broadcasts event E_two executes and completes:

   a The broadcast of event E_two now preempts the transition from state A1a to state A1b that event E_one triggers.

   b The broadcast of event E_two awakens the chart a second time. The chart root checks to see if there is a valid transition as a result of E_two. No valid transition exists.

   c State A during actions (durA()) execute and complete.

   d State A evaluates its children starting with state A.A1. State A.A1during actions (durA1()) execute and complete. State A.A1 is evaluated for valid transitions. There are no valid transitions as a result of E_two within state A1.

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

   f State A.A2.A2a exit actions (exitA2a()) execute and complete.

   g State A.A2.A2a is marked inactive.

   h State A.A2.A2b is marked active.

   i State A.A2.A2b entry actions (entA2b()) execute and complete.

Event E_one Processing Resumes

1 State A.A1.A1b is marked active.

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

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

4 State A.A2.A2b during actions (durA2b()) execute and complete.

   State A.A2.A2b is now active as a result of the processing of event broadcast E_two.

5 The chart goes back to sleep.

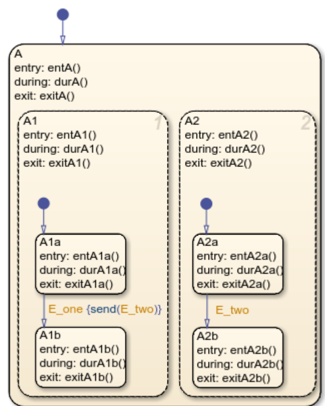
This sequence completes the execution of this Stateflow chart associated with event `E_one` and the event broadcast on a transition action to a parallel state of event `E_two`. The final chart activity is that parallel substates `A.A1.A1b` and `A.A2.A2b` are active.

Tip Avoid using undirected local event broadcasts. Undirected local event broadcasts can cause unwanted recursive behavior in your chart. Instead, send local events by using directed broadcasts. For more information, see "Broadcast Local Events to Synchronize Parallel States" on page 12-25.

During simulation, Stateflow charts can detect undirected local event broadcasts. To control the level of diagnostic action, open the Configuration Parameters dialog box and, in the **Diagnostics** > **Stateflow** pane, set the **Undirected event broadcasts** parameter to `none`, `warning`, or `error`. The default setting is `warning`. For more information, see "Undirected event broadcasts" (Simulink).

Broadcast Events in Condition Actions

This example shows the behavior of a condition action event broadcast in a parallel (AND) state. The chart uses implicit ordering of parallel states (see "Implicit Ordering of Parallel States" on page 2-47).



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. The children of state `A` 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 evaluates its children starting with state A.A1. State A.A1 during actions (durA1()) execute and complete. State A.A1 is evaluated for valid transitions. There are no valid transitions as a result of E_two within state A1.
   d State A1a during actions (durA1a()) execute.
   e State A.A2 is evaluated. State A.A2 during actions (durA2()) execute and complete. State A.A2 checks for valid transitions. State A.A2 has a valid transition as a result of E_two from state A.A2.A2a to state A.A2.A2b.
   f State A.A2.A2a exit actions (exitA2a()) execute and complete.
   g State A.A2.A2a is marked inactive.
   h State A.A2.A2b is marked active.
   i State A.A2.A2b entry actions (entA2b()) execute and complete.
5. State A.A1.A1a executes and completes exit actions (exitA1a).
6. State A.A1.A1a is marked inactive.
7. State A.A1.A1b is marked active.
8. State A.A1.A1b entry actions (entA1b()) execute and complete.
9. Parallel state A.A2 is evaluated next. State A.A2 during actions (durA2()) execute and complete. There are no valid transitions as a result of E_one.
10. State A.A2.A2b during actions (durA2b()) execute and complete.

    State A.A2.A2b is now active as a result of the processing of the condition action event broadcast of E_two.
11. The chart goes back to sleep.

This sequence completes the execution of this Stateflow chart associated with event E_one and the event broadcast on a condition action to a parallel state of event E_two. The final chart activity is that parallel substates A.A1.A1b and A.A2.A2b are active.

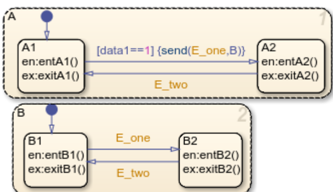
Tip Avoid using undirected local event broadcasts. Undirected local event broadcasts can cause unwanted recursive behavior in your chart. Instead, send local events by using directed broadcasts. For more information, see “Broadcast Local Events to Synchronize Parallel States” on page 12-25.

During simulation, Stateflow charts can detect undirected local event broadcasts. To control the level of diagnostic action, open the Configuration Parameters dialog box and, in the **Diagnostics** > **Stateflow** pane, set the **Undirected event broadcasts** parameter to none, warning, or error. The default setting is warning. For more information, see “Undirected event broadcasts” (Simulink).

Broadcast Local Events in Parallel States

Directed Event Broadcast Using Send

This example shows the behavior of directed event broadcast using the `send(event_name,state_name)` syntax on a transition. The chart uses implicit ordering of parallel states (see “Implicit Ordering of Parallel States” on page 2-47).



Initially, the chart is asleep. Parallel substates `A.A1` and `B.B1` are active, which implies that parallel (AND) superstates `A` and `B` are also active. The condition `[data1==1]` is true. The event `E_one` belongs to the chart and is visible to both `A` and `B`.

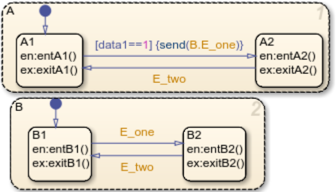
After waking up, the chart checks for valid transitions at every level of the hierarchy:

1. The chart root checks to see if there is a valid transition as a result of the event. There is no valid transition.
2. State `A` checks for any valid transitions as a result of the event. Because the condition `[data1==1]` is true, there is a valid transition from state `A.A1` to state `A.A2`.
3. The action `send(E_one,B)` executes:
   a. The broadcast of event `E_one` reaches state `B`. Because state `B` is active, that state receives the event broadcast and checks to see if there is a valid transition. There is a valid transition from `B.B1` to `B.B2`.
   b. State `B.B1` `exit` actions (`exitB1()`) execute and complete.
   c. State `B.B1` becomes inactive.
   d. State `B.B2` becomes active.
   e. State `B.B2` `entry` actions (`entB2()`) execute and complete.
4. State `A.A1` `exit` actions (`exitA1()`) execute and complete.
5. State `A.A1` becomes inactive.
6. State `A.A2` becomes active.
7. State `A.A2` `entry` actions (`entA2()`) execute and complete.

This sequence completes execution of a chart with a directed event broadcast to a parallel state.

Directed Event Broadcast Using Qualified Event Name

This example shows the behavior of directed event broadcast using a qualified event name on a transition. The chart uses implicit ordering of parallel states (see “Implicit Ordering of Parallel States” on page 2-47).



The only differences from the chart in “Directed Event Broadcast Using Send” on page A-35 are:

- The event `E_one` belongs to state `B` and is visible only to that state.
- The action `send(E_one,B)` is now `send(B.E_one)`.

  Using a qualified event name is necessary because `E_one` is not visible to state `A`.

After waking up, the chart checks for valid transitions at every level of the hierarchy:

1. The chart root checks to see if there is a valid transition as a result of the event. There is no valid transition.
2. State `A` checks for any valid transitions as a result of the event. Because the condition `[data1==1]` is true, there is a valid transition from state `A.A1` to state `A.A2`.
3. The action `send(B.E_one)` executes and completes:

   a. The broadcast of event `E_one` reaches state `B`. Because state `B` is active, that state receives the event broadcast and checks to see if there is a valid transition. There is a valid transition from `B.B1` to `B.B2`.

   b. State `B.B1` `exit` actions (`exitB1()`) execute and complete.

   c. State `B.B1` becomes inactive.

   d. State `B.B2` becomes active.

   e. State `B.B2` `entry` actions (`entB2()`) execute and complete.
4. State `A.A1` `exit` actions (`exitA1()`) execute and complete.
5. State `A.A1` becomes inactive.
6. State `A.A2` becomes active.
7. State `A.A2` `entry` actions (`entA2()`) execute and complete.

This sequence completes execution of a chart with a directed event broadcast using a qualified event name to a parallel state.

See Also
`send`

More About
- “Broadcast Local Events to Synchronize Parallel States” on page 12-25

Simulation Data Inspector

- "View Data in the Simulation Data Inspector" on page 32-2
- "Import Data from a CSV File into the Simulation Data Inspector" on page 32-11
- "Microsoft Excel Import, Export, and Logging Format" on page 32-15
- "Configure the Simulation Data Inspector" on page 32-23
- "How the Simulation Data Inspector Compares Data" on page 32-31
- "Save and Share Simulation Data Inspector Data and Views" on page 32-36
- "Inspect and Compare Data Programmatically" on page 32-42
- "Limit the Size of Logged Data" on page 32-48

View Data in the Simulation Data Inspector

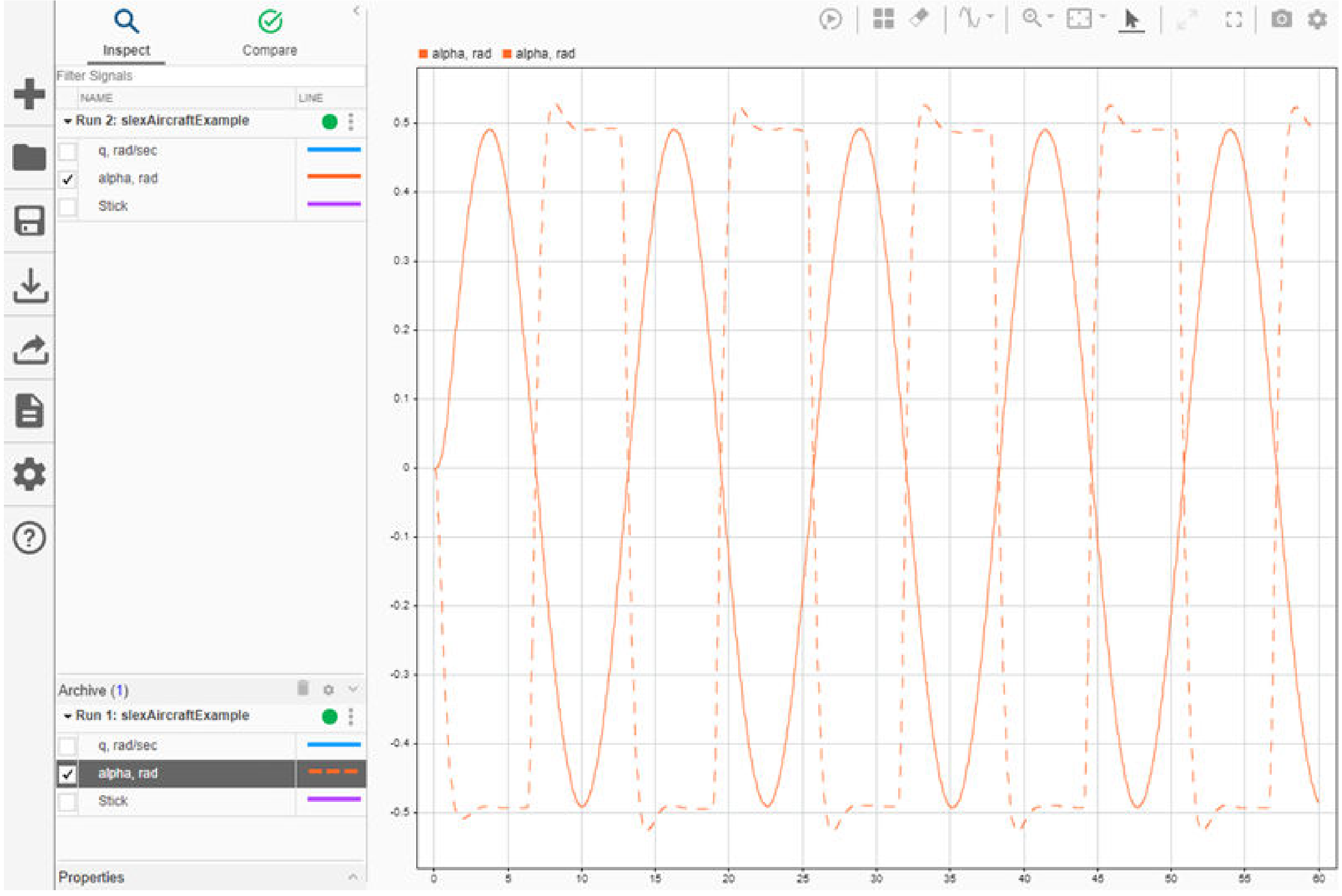
You can use the Simulation Data Inspector to visualize the data you generate throughout the design process. Simulation data that you log in a Simulink model logs to the Simulation Data inspector. You can also import test data and other recorded data into the Simulation Data Inspector to inspect and analyze it alongside the logged simulation data. The Simulation Data Inspector offers several types of plots, which allow you to easily create complex visualizations of your data.

View Logged Data

Logged signals as well as outputs and states logged using the `Dataset` format automatically log to the Simulation Data Inspector when you simulate a model. You can also record other kinds of simulation data so the data appears in the Simulation Data Inspector at the end of the simulation. To see states and output data logged using a format other than `Dataset` in the Simulation Data Inspector, open the Configuration Parameters dialog box and, in the **Data Import/Export** pane, select the **Record logged workspace data in Simulation Data Inspector** parameter.

Note When you log states and outputs using the `Structure` or `Array` format, you must also log time for the data to record to the Simulation Data Inspector.

The Simulation Data Inspector displays available data in the table in the **Inspect** pane. To plot a signal, select the check box next to the signal. You can modify the layout and add different visualizations to analyze the simulation data. For more information, see “Create Plots Using the Simulation Data Inspector” (Simulink).



The Simulation Data Inspector manages incoming simulation data using the archive. By default, the previous run moves to the archive when you start a new simulation. You can plot signals from the archive, or you can drag runs of interest back into the work area.

Import Data from the Workspace or a File

You can import data from the base workspace or from a file to view on its own or alongside simulation data. The Simulation Data Inspector supports all built-in data types and many data formats for importing data from the workspace. In general, whatever the format, sample values must be paired with sample times. The Simulation Data Inspector allows up to 8000 channels per signal in a run created from imported workspace data.

You can also import data from these types of files:

- MAT file
- CSV file — Format data as shown in “Import Data from a CSV File into the Simulation Data Inspector” (Simulink).
- Microsoft® Excel file — Format data as described in “Microsoft Excel Import, Export, and Logging Format” (Simulink).

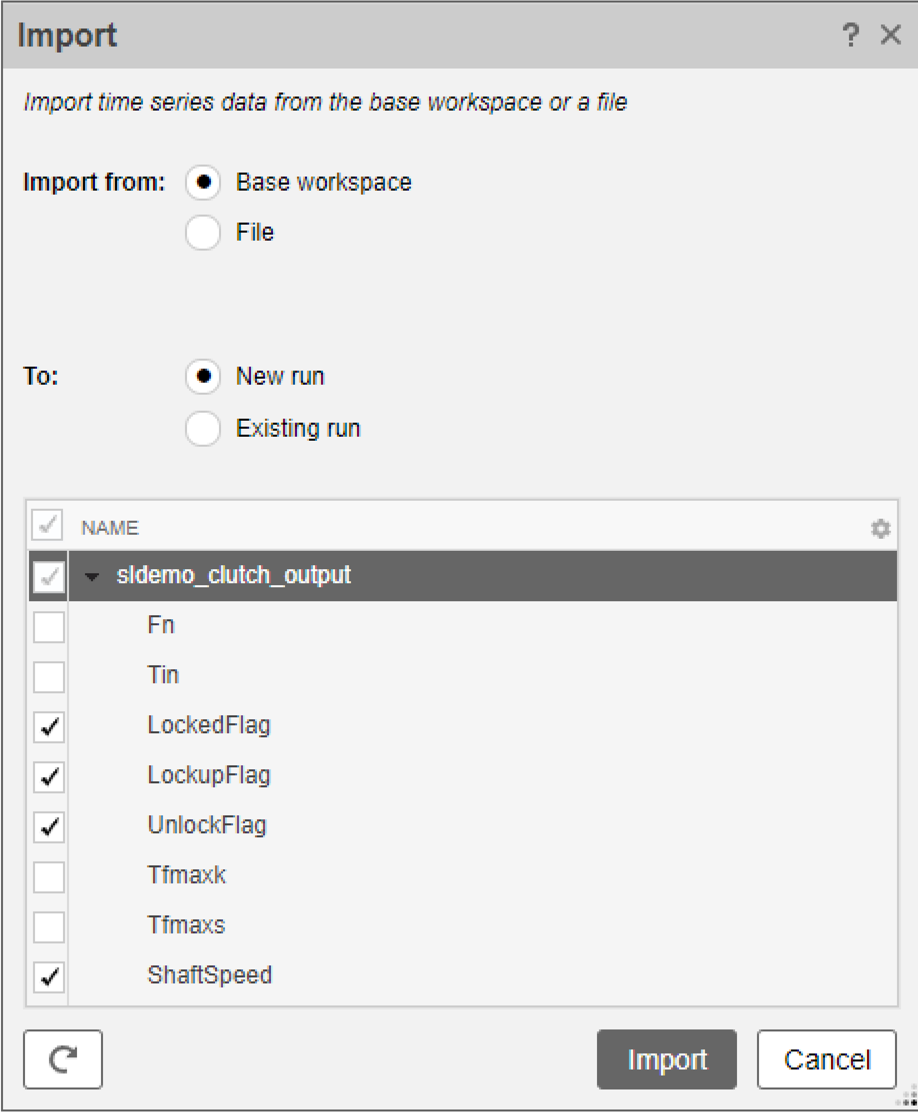
- MDF file — MDF file import is supported for Linux® and Windows operating systems. The MDF file must have a `.mdf`, `.mf4`, `.mf3`, `.data`, or `.dat` file extension and contain data with only integer and floating data types.
- ULG file — Flight log data import requires a UAV Toolbox license.

To import data from the workspace or from a file that is saved in a data or file format that the Simulation Data Inspector does not support, you can write your own workspace data or file reader to import the data using the `io.reader` class. You can also write a custom reader to use instead of the built-in reader for supported file types. For examples, see:

- "Import Data Using a Custom File Reader" (Simulink)
- "Import Workspace Variables Using a Custom Data Reader" (Simulink)

To import data, select the **Import** button in the Simulation Data Inspector.

In the Import dialog, you can choose to import data from the workspace or from a file. The table below the options shows data available for import. If you do not see your workspace variable or file contents in the table, that means the Simulation Data Inspector does not have a built-in or registered reader that supports that data. You can select which data to import using the check boxes, and you can choose whether to import that data into an existing run or a new run. To select all or none of the data, use the check box next to **NAME**.



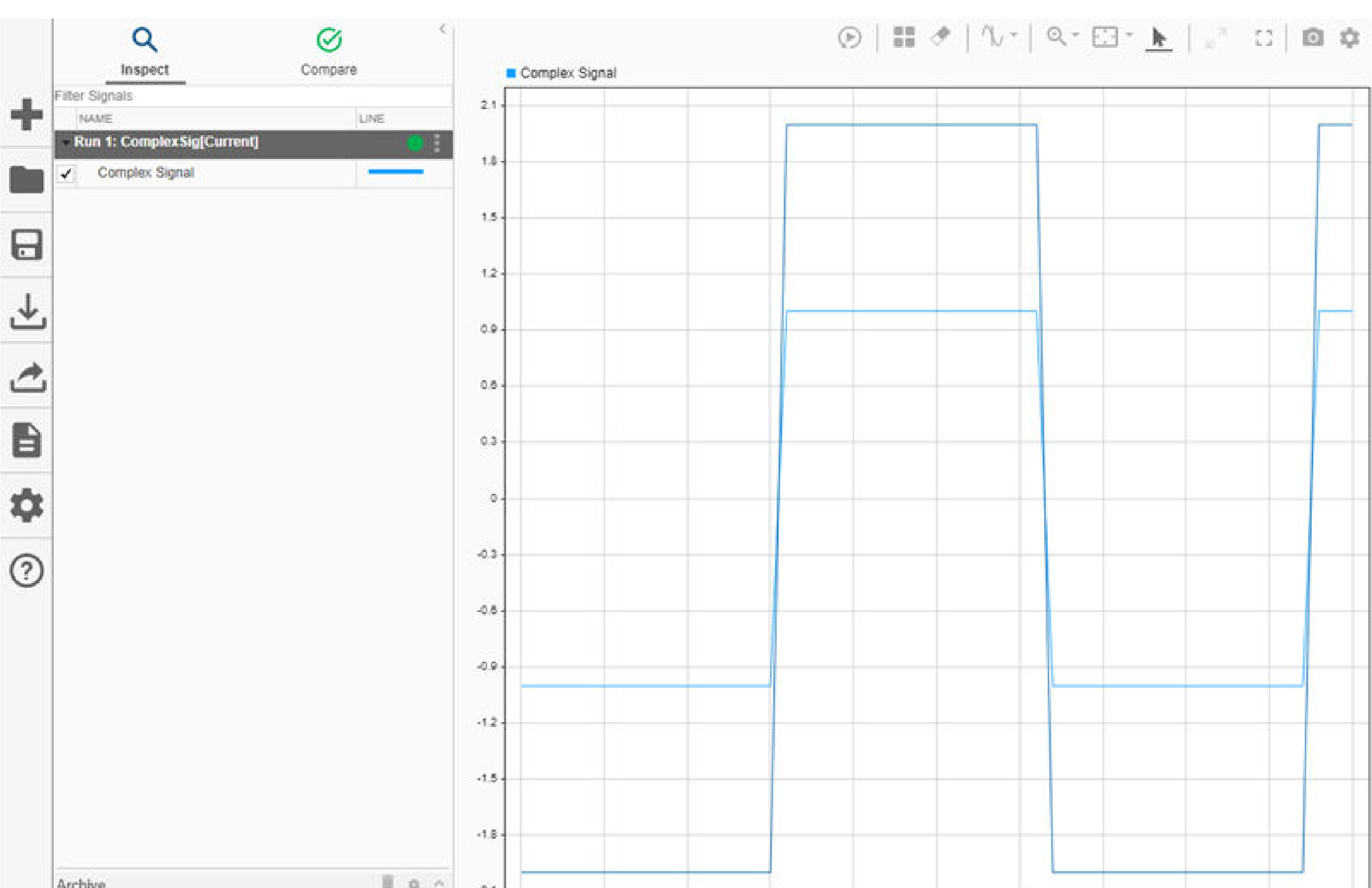
When you import data into a new run, the run always appears in the work area. You can manually move imported runs to the archive.

View Complex Data

To view complex data in the Simulation Data Inspector, import the data or log the signals to the Simulation Data Inspector. You can control how to visualize the complex signal using the **Properties** pane in the Simulation Data Inspector and in the **Instrumentation Properties** for the signal in the model. To access the **Instrumentation Properties** for a signal, right-click the logging badge for the signal and select **Properties**.

You can specify the **Complex Format** as Magnitude, Magnitude-Phase, Phase, or Real-Imaginary. If you select Magnitude-Phase or Real-Imaginary for the **Complex Format**, the Simulation Data Inspector plots both components of the signal when you select the check box for the signal. For signals in Real-Imaginary format, the **Line Color** specifies the color of the real component of the signal, and the imaginary component is a different shade of the **Line Color**. For example, the

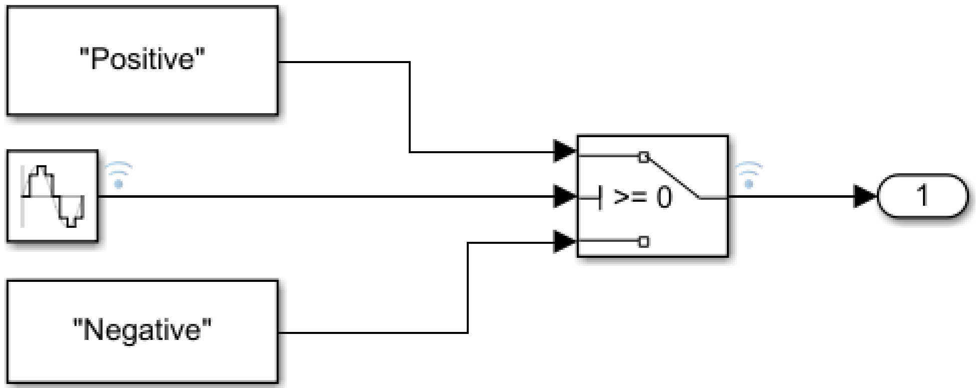
`Complex Signal` displays the real component of the signal in light blue, matching the **Line Color** parameter, and the imaginary component is shown in a darker shade of blue.



For signals in Magnitude-Phase format, the **Line Color** specifies the color of the magnitude component, and the phase is displayed in a different shade of the **Line Color**.

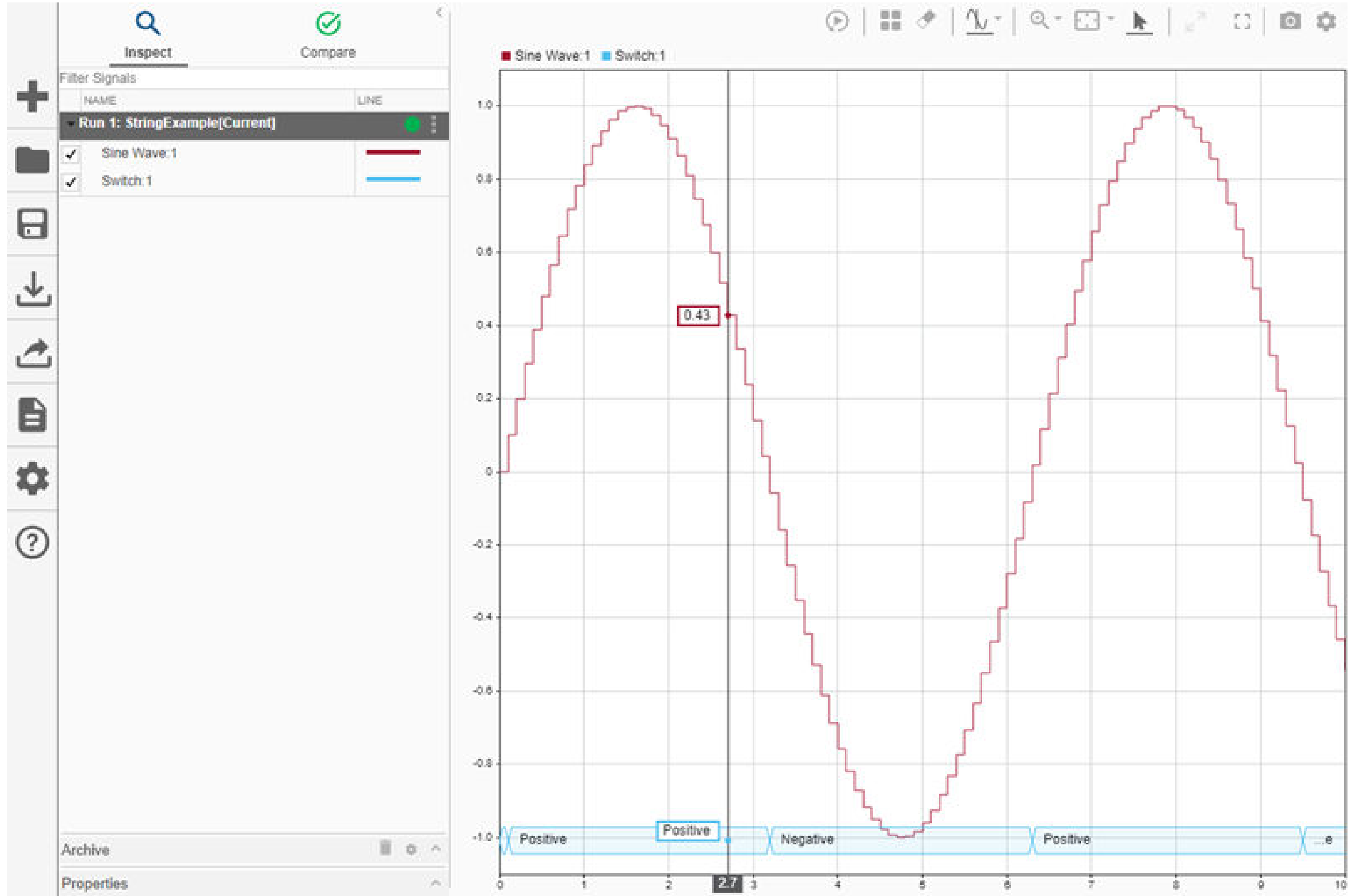
View String Data

You can log and view string data with your signal data in the Simulation Data Inspector. For example, consider this simple model. The value of the sine wave block controls whether the switch sends a string reading `Positive` or `Negative` to the output.

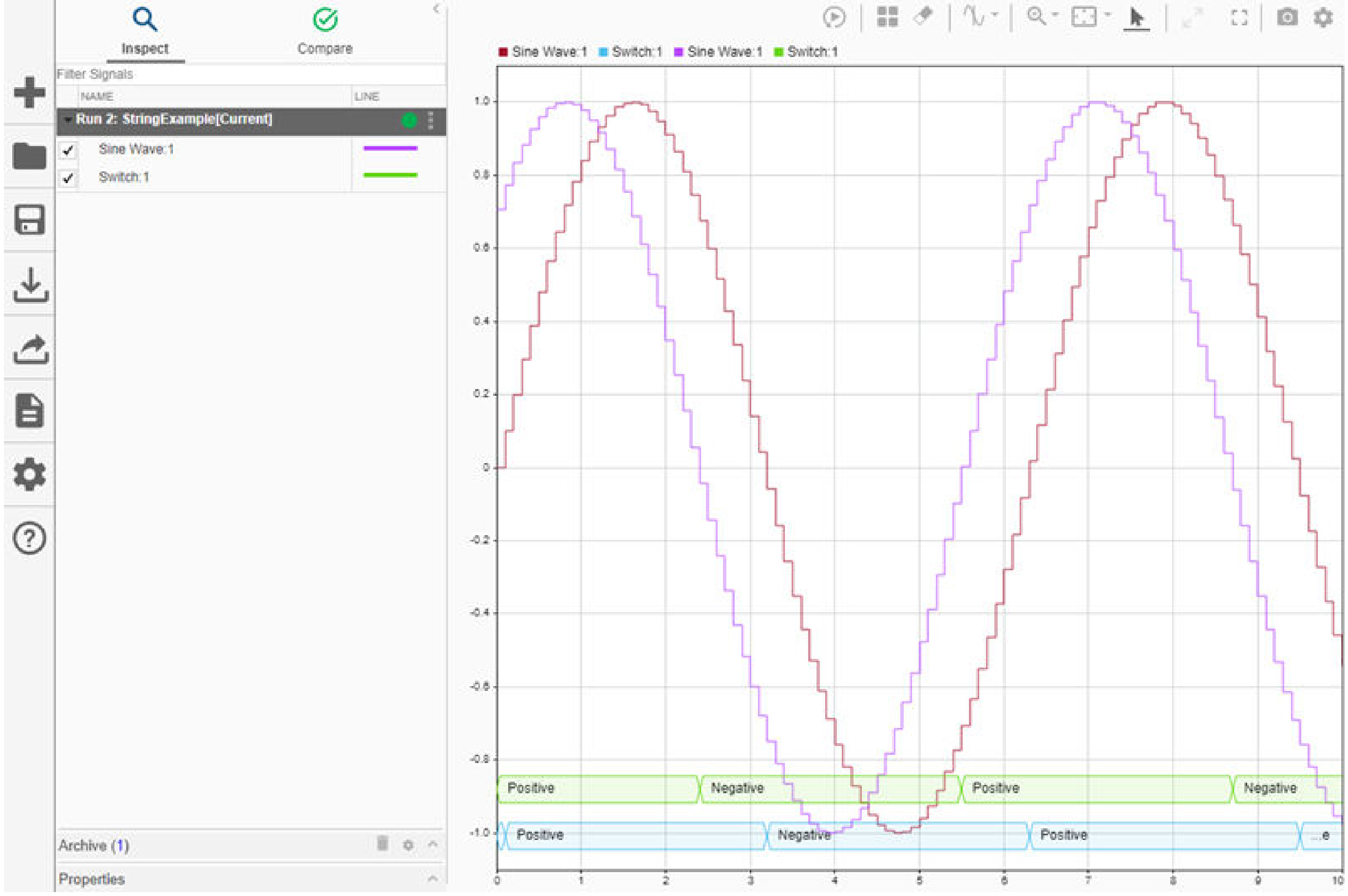


The plot shows the results of simulating the model. The string signal is shown at the bottom of the graphical viewing area. The value of the signal is displayed inside a band, and transitions in the string signal's value are marked with criss-crossed lines.

You can use cursors to inspect how the string signal values correspond with the sine signal's values.



When you plot multiple string signals on a plot, the signals stack in the order they were simulated or imported, with the most recent signal positioned at the top. For example, you might consider the effect of changing the phase of the sine wave controlling the switch.

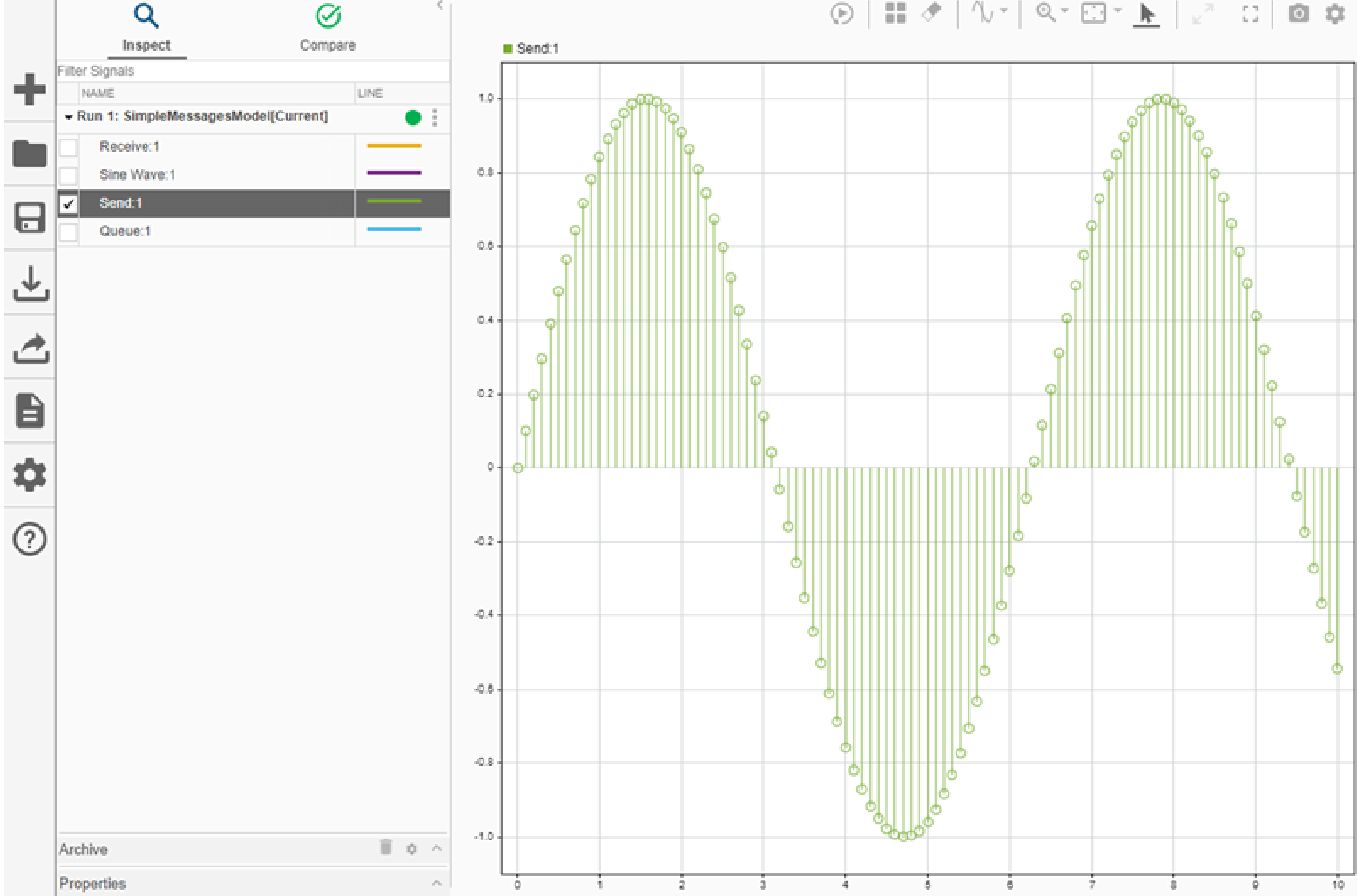


View Frame-Based Data

Processing data in frames rather than point by point provides a performance boost needed in some applications. To view frame-based data in the Simulation Data Inspector, you have to specify that the signal is frame-based in the **Instrumentation Properties** for the signal. To access the **Instrumentation Properties** dialog for a signal, right-click the signal's logging badge and select **Properties**. To specify a signal as frame-based, select **Columns as channels (frame based)** for **Input processing**.

View Event-Based Data

You can log or import event data to the Simulation Data Inspector. To view the logged event-based data, select the check box next to `Send: 1`. The Simulation Data Inspector displays the data as a stem plot, with each stem representing the number of events that occurred for a given sample time.



See Also

More About

- Inspect Simulation Data (Simulink)
- Compare Simulation Data (Simulink)
- Share Simulation Data Inspector Data and Views on page 32-36
- Decide How to Visualize Data (Simulink)
- Dataset Conversion for Logged Data (Simulink)

Import Data from a CSV File into the Simulation Data Inspector

To import data into the Simulation Data Inspector from a CSV file, format the data in the CSV file. Then, you can import the data using the Simulation Data Inspector UI or the `Simulink.sdi.createRun` function.

Tip When you want to import data from a CSV file where the data is formatted differently from the specification in this topic, you can write your own file reader for the Simulation Data Inspector using the `io.reader` class.

Basic File Format

In the simplest format, the first row in the CSV file is a header that lists the names of the signals in the file. The first column is time. The name for the time column must be `time`, and the time values must increase monotonically. The rows below the signal names list the signal values that correspond to each time step.

myData - Notepad

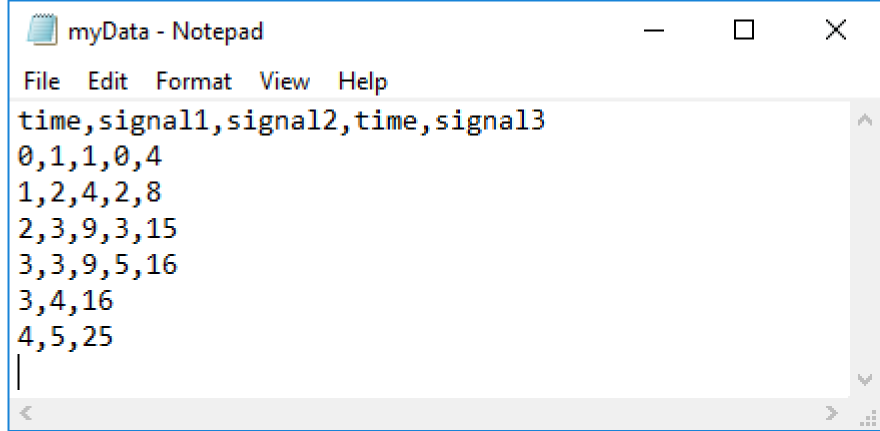
File Edit Format View Help

```
time,signal1,signal2,signal3
0,1,1,4
1,2,4,8
2,3,9,15
3,3,9,16
3,4,16,23
4,5,25,42
```

The import operation does not support time data that includes `Inf` or `NaN` values or signal data that includes `Inf` values. Empty or `NaN` signal values render as missing data. All built-in data types are supported.

Multiple Time Vectors

When your data includes signals with different time vectors, the file can include more than one time vector. Every time column must be named `time`. Time columns specify the sample times for signals to the right, up to the next time vector. For example, the first time column defines the time for `signal1` and `signal2`, and the second time column defines the time steps for `signal3`.



Signal columns must have the same number of data points as the associated time vector.

Signal Metadata

You can specify signal metadata in the CSV file to indicate the signal data type, units, interpolation method, block path, and port index. List metadata for each signal in rows between the signal name and the signal data. Label metadata according to this table.

Signal Property	Label	Value
Data type	`Type:`	Built-in data type.
Units	`Unit:`	Supported unit. For example, `Unit: m/s` specifies units of meters per second. For a list of supported units, enter `showunitslist` in the MATLAB Command Window.
Interpolation method	`Interp:`	`linear`, `zoh` for zero order hold, or `none`.
Block Path	`BlockPath:`	Path to the block that generated the signal.
Port Index	`PortIndex:`	Integer.

You can also import a signal with a data type defined by an enumeration class. Instead of using the `Type:` label, use the `Enum:` label and specify the value as the name of the enumeration class. The definition for the enumeration class must be saved on the MATLAB path.

When an imported file does not specify signal metadata, the Simulation Data Inspector assumes double data type and linear interpolation. You can specify the interpolation method as linear, zoh (zero-order hold), or none. If you do not specify units for the signals in your file, you can assign units to the signals in the Simulation Data Inspector after you import the file.

You can specify any combination of metadata for each signal. Leave a blank cell for signals with less specified metadata.

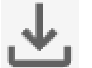
myData - Notepad

File Edit Format View Help

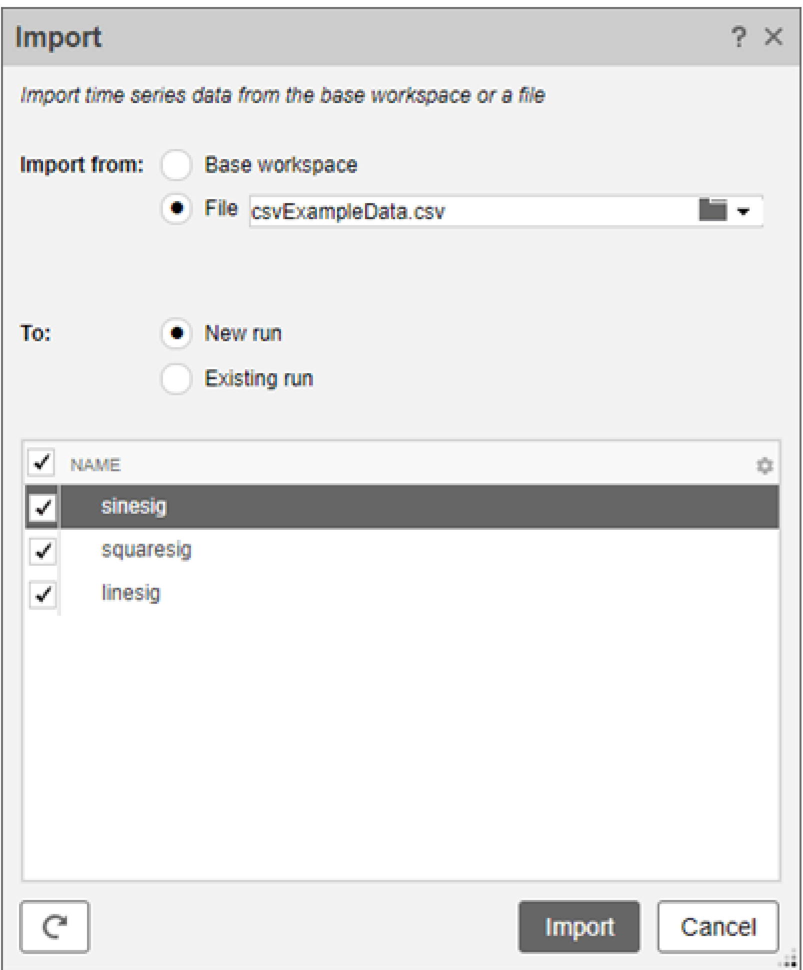
```
time,signal1,signal2,time,signal3
 ,Interp: zoh, , ,Interp: zoh
 ,Type: int8,Type: int32
 ,Unit: m, , ,Unit: m/s
0,1,1,0,4
1,2,4,2,8
2,3,9,3,15
3,3,9,5,16
3,4,16
4,5,25
```

Import Data from a CSV File

You can import data from a CSV file using the Simulation Data Inspector UI or using the `Simulink.sdi.createRun` function.

To import data using the UI, open the Simulation Data Inspector using the `Simulink.sdi.view` function or the **Data Inspector** button in the Simulink™ toolstrip. Then, click **Import** .

In the Import dialog, select the option to import data from a file and navigate in the file system to select the file. After you select the file, data available for import shows in the table. You can choose which signals to import and whether to import them to a new or existing run. This example imports all available signals to a new run. To select all or none of the signals, select or clear the check box next to `NAME`. After selecting the options, click the **Import** button.



When you import data into a new run using the UI, the new run name includes the run number followed by `Imported_Data`.

When you import data programmatically, you can specify the name of the imported run.

```
csvRunID = Simulink.sdi.createRun('CSV File Run','file','csvExampleData.csv');
```

See Also

Functions
`Simulink.sdi.createRun`

More About

- “View Data in the Simulation Data Inspector” (Simulink)
- “Microsoft Excel Import, Export, and Logging Format” (Simulink)
- “Import Data Using a Custom File Reader” (Simulink)

Microsoft Excel Import, Export, and Logging Format

Using the Simulation Data Inspector or Simulink Test, you can import data from a Microsoft Excel file or export data to a Microsoft Excel file. You can also log data to an Excel file using the Record block. The Simulation Data Inspector, Simulink Test, and the Record block all use the same file format, so you can use the same Microsoft Excel file with multiple applications.

Tip When the format of the data in your Excel file does not match the specification in this topic, you can write your own file reader to import the data using the `io.reader` class.

Basic File Format

In the simplest format, the first row in the Excel file is a header that lists the names of the signals in the file. The first column is time. The name for the time column must be `time`, and the time values must increase monotonically. The rows below the signal names list the signal values that correspond to each time step.

	A	B	C	D
1	time	signal1	signal2	signal3
2	0	1	1	4
3	1	2	4	8
4	2	3	9	15
5	3	3	9	16
6	3	4	16	23
7	4	5	25	42

The import operation does not support time data that includes `Inf` or `NaN` values or signal data that includes `Inf` values. Empty or `NaN` signal values imported from the Excel file render as missing data in the Simulation Data Inspector. All built-in data types are supported.

Multiple Time Vectors

When your data includes signals with different time vectors, the file can include more than one time vector. Every time column must be named `time`. Time columns specify the sample times for signals to the right, up to the next time vector. For example, the first time column defines the time for `signal1` and `signal2`, and the second time column defines the time steps for `signal3`.

	A	B	C	D	E
1	time	signal1	signal2	time	signal3
2	0	1	1	0	4
3	1	2	4	2	8
4	2	3	9	3	15
5	3	3	9	5	16
6	3	4	16		
7	4	5	25		

Signal columns must have the same number of data points as the associated time vector.

Signal Metadata

The file can include metadata for signals such as data type, units, and interpolation method. The metadata is used to determine how to plot the data, how to apply unit and data conversions, and how to compute comparison results. For more information about how metadata is used in comparisons, see “How the Simulation Data Inspector Compares Data” (Simulink).

Metadata for each signal is listed in rows between the signal names and the signal data. You can specify any combination of metadata for each signal. Leave a blank cell for signals with less specified metadata.

	A	B	C	D	E
1	time	signal1	signal2	time	signal3
2		Interp: zoh			Interp: zoh
3		Type: int8	Type: int32		
4		Unit: m			Unit: m/s
5	0	1	1	0	4
6	1	2	4	2	8
7	2	3	9	3	15
8	3	3	9	5	16
9	3	4	16		
10	4	5	25		

Label each piece of metadata according to this table. The table also indicates which tools and operations support each piece of metadata. When an imported file does not specify signal metadata, `double` data type, `linear` interpolation, and `union` synchronization are used.

Property Descriptions

Signal Property	Label	Values	Simulation Data Inspector Import	Record Block Logging and Simulation Data Inspector Export	Simulink Test Import and Export
Data type	`Type:`	Built-in data type.	**Supported**	**Supported**	**Supported**
Units	`Unit:`	Supported unit. For example, `Unit: m/s` specifies units of meters per second. For a list of supported units, enter `showunitslist` in the MATLAB Command Window.	**Supported**	**Supported**	**Supported**
Interpolation method	`Interp:`	`linear`, `zoh` for zero order hold, or `none`.	**Supported**	**Supported**	**Supported**
Synchronization method	`Sync:`	`union` or `intersection`.	**Supported**	**Not Supported** *Metadata not included in exported file.*	**Supported**
Relative tolerance	`RelTol:`	Percentage, represented as a decimal. For example, `RelTol: 0.1` specifies a 10% relative tolerance.	**Supported**	**Not Supported** *Metadata not included in exported file.*	**Supported**
Absolute tolerance	`AbsTol:`	Numeric value.	**Supported**	**Not Supported** *Metadata not included in exported file.*	**Supported**
Time tolerance	`TimeTol:`	Numeric value, in seconds.	**Supported**	**Not Supported** *Metadata not included in exported file.*	**Supported**

Signal Property	Label	Values	Simulation Data Inspector Import	Record Block Logging and Simulation Data Inspector Export	Simulink Test Import and Export
Leading tolerance	`LeadingTol:`	Numeric value, in seconds.	**Supported** *Only visible in Simulink Test.*	**Not Supported** *Metadata not included in exported file.*	**Supported**
Lagging tolerance	`LaggingTol:`	Numeric Value, in seconds.	**Supported** *Only visible in Simulink Test.*	**Not Supported** *Metadata not included in exported file.*	**Supported**
Block Path	`BlockPath:`	Path to the block that generated the signal.	**Supported**	**Supported**	**Supported**
Port Index	`PortIndex:`	Integer.	**Supported**	**Supported**	**Supported**
Name	`Name:`	Signal name	**Supported**	**Not Supported** *Metadata not included in exported file.*	**Supported**

User-Defined Data Types

In addition to built-in data types, you can use other labels in place of the `DataType:` label to specify fixed-point, enumerated, alias, and bus data types.

Property Descriptions

Data Type	Label	Values	Simulation Data Inspector Import	Record Block Logging and Simulation Data Inspector Export	Simulink Test Import and Export
Enumeration	`Enum:`	Name of the enumeration class.	**Supported** *Enumeration class definition must be saved on the MATLAB path.*	**Supported** *Enumeration class definition must be saved on the MATLAB path.*	**Supported** *Enumeration class definition must be saved on the MATLAB path.*
Alias	`Alias:`	Name of a `Simulink.AliasType` object in the MATLAB workspace.	**Supported** *For matrix and complex signals, specify the alias data type on the first channel.*	**Not Supported**	**Supported** *For matrix and complex signals, specify the alias data type on the first channel.*
Fixed-point	`Fixdt:`	• `fixdt` constructor. • Name of a `Simulink.NumericType` object in the MATLAB workspace. • Name of a fixed-point data type as described in “Fixed-Point Numbers in Simulink” (Fixed-Point Designer).	**Supported**	**Not Supported**	**Supported**
Bus	`Bus:`	Name of a `Simulink.Bus` object in the MATLAB workspace.	**Supported**	**Not Supported**	**Supported**

When you specify the type using the name of a `Simulink.Bus` object and the object is not in the MATLAB workspace, the data still imports from the file. However, individual signals in the bus use data types described in the file rather than data types defined in the `Simulink.Bus` object.

Complex, Multidimensional, and Bus Signals

You can import and export complex, multidimensional, and bus signals using an Excel file. The signal name for a column of data indicates whether that data is part of a complex, multidimensional, or bus signal. Excel file import and export do not support array of bus signals.

Note When you export data from a nonvirtual bus with variable-size signals to an Excel file, the variable-size signal data is expanded to individual channels, and the hierarchical nature of the data is lost. Data imported from this file is returned as a flat list.

Multidimensional signal names include index information in parentheses. For example, the signal name for a column might be `signal1(2,3)`. When you import data from a file that includes multidimensional signal data, elements in the data not included in the file take zero sample values with the same data type and complexity as the other elements.

Complex signal data is always in real-imaginary format. Signal names for columns containing complex signal data include `(real)` and `(imag)` to indicate which data each column contains. When you import data from a file that includes imaginary signal data without specifying values for the real component of that signal, the signal values for the real component default to zero.

Multidimensional signals can contain complex data. The signal name includes the indication for the index within the multidimensional signal and the real or imaginary tag. For example, `signal1(1,3)(real)`.

Dots in signal names specify the hierarchy for bus signals. For example:

- `bus.y.a`
- `bus.y.b`
- `bus.x`

	A	B	C	D	E
1	time	bus.y.a	bus.y.b	time	bus.x
2		Interp: zoh			Interp: zoh
3		Type: int8	Type: int32		
4		Unit: m			Unit: m/s
5	0	1	1	0	4
6	1	2	4	2	8
7	2	3	9	3	15
8	3	3	9	5	16
9	3	4	16		
10	4	5	25		

Tip When the name of your signal includes characters that could make it appear as though it were part of a matrix, complex signal, or bus, use the `Name` metadata option to specify the name you want the imported signal to use in the Simulation Data Inspector and Simulink Test.

Function-Call Signals

Signal data specified in columns before the first time column is imported as one or more function-call signals. The data in the column specifies the times at which the function-call signal was enabled. The imported signals have a value of 1 for the times specified in the column. The time values for function-call signals must be double, scalar, and real, and must increase monotonically.

When you export data from the Simulation Data Inspector, function-call signals are formatted the same as other signals, with a time column and a column for signal values.

Simulation Parameters

You can import data for parameter values used in simulation. In the Simulation Data Inspector, the parameter values are shown as signals. Simulink Test uses imported parameter values to specify values for those parameters in the tests it runs based on imported data.

Parameter data is specified using two or three columns. The first column specifies the parameter names, with the cell in the header row for that column labeled `Parameter:`. The second column specifies the value used for each parameter, with the cell in the header row labeled `Value:`. Parameter data may also include a third column that contains the block path associated with each parameter, with the cell in the header row labeled `BlockPath:`. Specify names, values, and block paths for parameters starting in the first row that contains signal data, below rows used to specify signal metadata. For example, this file specifies values for two parameters, `X` and `Y`.

	A	B	C	D	E	F	G
1	time	signal1	signal2	time	signal3	Parameter:	Value:
2		Interp: zoh			Interp: zoh		
3		Type: int8	Type: int32				
4		Unit: m			Unit: m/s		
5	0	1	1	0	4	X	2
6	1	2	4	2	8	Y	1.2
7	2	3	9	3	15		
8	3	3	9	5	16		
9	3	4	16				
10	4	5	25				

Multiple Runs

You can include data for multiple runs in a single file. Within a sheet, you can divide data into runs by labeling data with a simulation number and a source type, such as `Input` or `Output`. Specify the simulation number and source type as additional signal metadata, using the label `Simulation:` for the simulation number and the label `Source:` for the source type. The Simulation Data Inspector uses the simulation number and source type only to determine which signals belong in each run. Simulink Test uses the information to define inputs, parameters, and acceptance criteria for tests to run based on imported data.

You do not need to specify the simulation number and output type for every signal. Signals to the right of a signal with a simulation number and source use the same simulation number and source

until the next signal with a different source or simulation number. For example, this file defines data for two simulations and imports into four runs in the Simulation Data Inspector:

- **Run 1** contains `signal1` and `signal2`.
- **Run 2** contains `signal3`, X, and Y.
- **Run 3** contains `signal4`.
- **Run 4** contains `signal5`.

	A	B	C	D	E	F	G	H	I	J
1	time	signal1	signal2	time	signal3	Parameter:	Values:	time	signal4	signal5
2		Interp: zoh			Interp: zoh					
3		Type: int8	Type: int32							
4		Unit: m			Unit: m/s					
5		Simulation: 1							Simulation: 2	
6		Source: Input			Source: Output				Source: Input	Source: Output
7	0	1	1	0	4	X	2	1	2	1
8	1	2	4	2	8	Y	1.2	2	6	3
9	2	3	9	3	15			3	4	5
10	3	3	9	5	16			4	8	7
11	3	4	16					5	10	2
12	4	5	25							

You can also use sheets within the Microsoft Excel file to divide the data into runs and tests. When you do not specify simulation number and source information, the data on each sheet is imported into a separate run in the Simulation Data Inspector. When you export multiple runs from the Simulation Data Inspector, the data for each run is saved on a separate sheet. When you import a Microsoft Excel file that contains data on multiple sheets into Simulink Test, you are prompted to specify how to import the data.

See Also

`Simulink.sdi.createRun` | `Simulink.sdi.exportRun`

More About

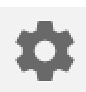
- "View Data in the Simulation Data Inspector" (Simulink)
- "Import Data from a CSV File into the Simulation Data Inspector" (Simulink)
- "Import Data Using a Custom File Reader" (Simulink)

Configure the Simulation Data Inspector

The Simulation Data Inspector supports a wide range of use cases for analyzing and visualizing data. You can modify preferences in the Simulation Data Inspector to match your visualization and analysis requirements. The preferences that you specify persist between MATLAB sessions.

By specifying preferences in the Simulation Data Inspector, you can configure options such as:

- How signals and metadata are displayed.
- Which data automatically imports from parallel simulations.
- Where prior run data is retained and how much prior data to store.
- How much memory is used during save operations.
- The system of units used to display signals.

To open the Simulation Data Inspector preferences, click Preferences. 

Note You can restore all preferences in the Simulation Data Inspector to default values by clicking **Restore Defaults** in the Preferences menu or by using the `Simulink.sdi.clearPreferences` function.

Logged Data Size and Location

By default, simulation data logs to disk with data loaded into memory on demand, and the maximum size of logged data is constrained only by available disk space. You can use the **Disk Management** settings in the Simulation Data Inspector to directly control the size and location of logged data.

The **Record mode** setting specifies whether logged data is retained after simulation. When you change the **Record mode** setting to **View during simulation only**, no logged data is available in the Simulation Data Inspector or the workspace after the simulation completes. Only use this mode when you do not want to save logged data. The **Record mode** setting reverts to **View and record data** each time you start MATLAB. Changing the **Record mode** setting can affect other applications, such as visualization tools. For details, see “View Data Only During Simulation” (Simulink).

To directly limit the size of logged data, you can specify a minimum amount of free disk space or a maximum size for the logged data. By default, logged data must leave at least 100 MB of free disk space with no maximum size limit. Specify the required disk space and maximum size in GB, and specify `0` to apply no disk space requirement or no maximum size limit.

When you specify a minimum disk space requirement or a maximum size for logged data, you can also specify whether to prioritize retaining data from the current simulation or data from prior simulations when approaching the limit. By default, the Simulation Data Inspector prioritizes retaining data for the current run by deleting data for prior runs. To prioritize retaining prior data, change the **When low on disk space** setting to **Keep prior runs and stop recording**. You see a warning message when prior runs are deleted and when recording is disabled. If recording is disabled due to the size of logged data, you need to change the **Record Mode** back to **View and**

record data to continue logging data, after you have freed up disk space. For more information, see "Specify a Minimum Disk Space Requirement or Maximum Size for Logged Data" (Simulink).

The **Storage Mode** setting specifies whether to log data to disk or to memory. By default, data logs to disk. When you configure a parallel worker to log data to memory, data transfer back to the host is not supported. Logging data to memory is not supported for rapid accelerator simulations or models deployed using Simulink Compiler™.

You can also specify the location of the temporary file that stores logged data. By default, data logs to the temporary files directory on your computer. You may change the file location when you need to log large amounts of data and a secondary drive provides more storage capacity. Logging data to a network location can degrade performance.

Programmatic Use

You can programmatically configure and check each preference value.

Preference	Functions

Record mode	`Simulink.sdi.setRecordData`
	
	`Simulink.sdi.getRecordData`

Required Free Space	`Simulink.sdi.setRequiredFreeSpace`
	
	`Simulink.sdi.getRequiredFreeSpace`

Max Disk Usage	`Simulink.sdi.setMaxDiskUsage`
	
	`Simulink.sdi.getMaxDiskUsage`

When low on disk space	`Simulink.sdi.setDeleteRunsOnLowSpace`
	
	`Simulink.sdi.getDeleteRunsOnLowSpace`

Storage Mode	`Simulink.sdi.setStorageMode`
	
	`Simulink.sdi.getStorageMode`

Storage Location	`Simulink.sdi.setStorageLocation`
	
	`Simulink.sdi.getStorageLocation`

Archive Behavior and Run Limit

When you run multiple simulations in a single MATLAB session, the Simulation Data Inspector retains results from each simulation so you can analyze the results together. Use the Simulation Data Inspector archive to manage runs in the user interface and control the number of runs the Simulation Data Inspector retains.

You can configure a limit for the number of runs to retain in the archive and whether the Simulation Data Inspector automatically moves prior runs into the archive.

Manage Runs Using the Archive

By default, the Simulation Data Inspector automatically archives simulation runs. When you simulate a model, the prior simulation run moves to the archive, and the Simulation Data Inspector updates the view to show data for aligned signals in the current run.

The archive does not impose functional limitations on the runs and signals it contains. You can plot signals from the archive, and you can use runs and signals in the archive in comparisons. You can drag runs of interest from the archive to the work area and vice versa whether **Automatically Archive** is selected or disabled.

To prevent the Simulation Data Inspector from automatically moving prior simulations runs to the archive, clear the **Automatically archive** setting. With automatic archiving disabled, the Simulation Data Inspector does not move prior runs into the **Archive** pane or automatically update plots to display data from the current simulation.

Tip To manually delete the contents of the archive, click Delete archived runs .

Control Number of Runs Retained in Simulation Data Inspector

You can specify a limit for the number of runs to retain in the archive. When the number of runs in the archive reaches the limit, the Simulation Data Inspector deletes runs in the archive on a first-in, first-out basis.

The run limit applies only to runs in the archive. For the Simulation Data Inspector to automatically limit the data it retains by deleting old runs, select **Automatically archive** and specify a size limit.

By default, the Simulation Data Inspector retains the last `20` runs moved to the archive. To remove the limit, select **No limit**. To specify the maximum number of runs to store in the archive, select **Last n runs** and enter the limit. A warning occurs if you specify a limit that would delete runs already in the archive.

Programmatic Use

You can programmatically configure and check the archive behavior and run limit.

Preference	Functions

Automatically archive	`Simulink.sdi.setAutoArchiveMode`
	
	`Simulink.sdi.getAutoArchiveMode`

Size	`Simulink.sdi.setArchiveRunLimit`
	
	`Simulink.sdi.getArchiveRunLimit`

Incoming Run Names and Location

You can configure how the Simulation Data Inspector handles incoming runs from import or simulation. You can choose whether new runs are added at the top of the work area or the bottom and specify a naming rule to use for runs created from simulation.

By default, the Simulation Data Inspector adds new runs below prior runs in the work area. The **Archive** settings also affect the location of runs. By default, prior runs are moved to the archive when a new simulation run is created.

The run naming rule is used to name runs created from simulation. You can create the run naming rule using a mix of literal text that is used in the run name as-is and one or more tokens that represent metadata about the run. By default, the Simulation Data Inspector names runs using the run index and model name: `Run <run_index>: <model_name>`.

Tip To rename an existing run, double-click the name in the work area and enter the new name, or modify the run name in the **Properties** pane.

Programmatic Use

You can programmatically configure and check incoming run names and locations.

Preference	Functions
Add New Runs	`Simulink.sdi.setAppendRunToTop` `Simulink.sdi.getAppendRunToTop`

Naming Rule	`Simulink.sdi.setRunNamingRule`
	`Simulink.sdi.getRunNamingRule`
	`Simulink.sdi.resetRunNamingRule`

Signal Metadata to Display

You can control which signal metadata is displayed in the work area of the **Inspect** pane and in the results section on the **Compare** pane in the Simulation Data Inspector. You specify the metadata to display separately for each pane using the **Table Columns** preferences in the **Inspect** and **Compare** sections of the Preferences dialog, respectively.

Inspect Pane

By default, the signal name and the line style and color used to plot the signal are displayed on the **Inspect** pane. To display different or additional metadata in the work area on the **Inspect** pane, select the check box next to each piece of metadata you want to display in the **Table Columns** preference in the **Inspect** section. You can always view complete metadata for the selected signal in the **Inspect** pane using the **Properties** pane.

Note Metadata displayed in the work area on **Inspect** pane is included when you generate a report of plotted signals. You can also specify metadata to include in the report regardless of what is displayed in the work area when you create the report programmatically using the `Simulink.sdi.report` function.

Compare Pane

By default, the **Compare** pane shows the signal name, the absolute and relative tolerances used in the signal comparison, and the maximum difference from the comparison result. To display different or additional metadata in the results on the **Compare** pane, select the check box next to each piece of metadata you want to display in the **Table Columns** preference in the **Compare** section. You can always view complete metadata for the signals compared for a selected signal result using the **Properties** pane, where metadata that differs between the compared signals is highlighted. Signal metadata displayed on the **Compare** pane does not affect the contents of comparison reports.

Signal Selection on the Inspect Pane

You can configure how you select signals to plot on the selected subplot in the Simulation Data Inspector. By default, you use check boxes next to each signal to plot. You can also choose to plot signals based on selection in the work area. Use **Check Mode** when creating views and visualizations that represent findings and analysis of a data set. Use **Browse Mode** to quickly view and analyze data sets with a large number of signals.

For more information about creating visualizations using **Check Mode**, see “Create Plots Using the Simulation Data Inspector” (Simulink).

For more information about using **Browse Mode**, see “Visualize Many Logged Signals” (Simulink).

Note To use **Browse Mode**, your layout must include only **Time Plot** visualizations.

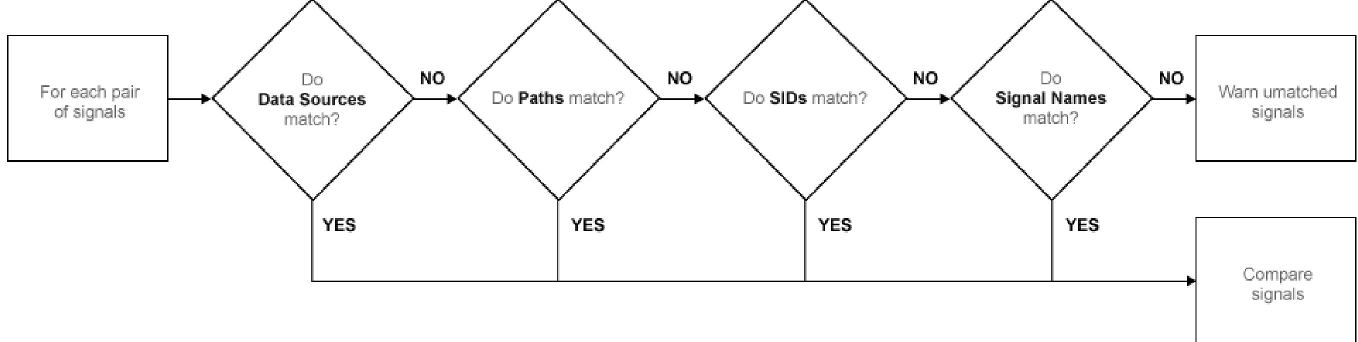
How Signals Are Aligned for Comparison

When you compare runs using the Simulation Data Inspector, the comparison algorithm pairs signals for signal comparison through a process called alignment. You can align signals between the compared runs using one or more of the signal properties shown in the table.

Property	Description
Data Source	Path of the variable in the MATLAB workspace for data imported from the workspace
Path	Block path for the source of the data in its model
SID	Automatically assigned Simulink identifier
Signal Name	Name of the signal

You can specify the priority for each piece of metadata used for alignment. The **Align By** field specifies the highest priority property used to align signals. The priority drops with each subsequent **Then By** field. You must specify a primary alignment property in the **Align By** field, but you can leave any number of **Then By** fields blank.

By default, the Simulation Data Inspector aligns signals between runs according to this flow chart.



For more information about configuring comparisons in the Simulation Data Inspector, see "How the Simulation Data Inspector Compares Data" (Simulink).

Colors Used to Display Comparison Results

You can configure the colors used to display comparison results using the Simulation Data Inspector preferences. You can specify whether to use the signal color from the **Inspect** pane or a fixed color for the baseline and compared signals. You can also choose colors for the tolerance and the difference signal.

By default, the Simulation Data Inspector displays comparison results using fixed colors for the baseline and compared signals. Using a fixed color allows you to avoid the baseline signal color and compared signal color being either the same or too similar to distinguish.

Signal Grouping

You can specify how to group signals within a run in the Simulation Data Inspector. The preferences apply to both the **Inspect** and **Compare** panes and comparison reports. You can group signals by:

- Domain — Signal type. For example, signals created by signal logging have a domain of `Signal`, while signals created from logging model outputs have a domain of `Outports`.
- Physical System Hierarchy — Signal Simscape™ physical system hierarchy. The option to group by physical system hierarchy is available when you have a Simscape license.
- Data Hierarchy — Signal location within structured data. For example, data hierarchy grouping reflects the hierarchy of a bus.
- Model Hierarchy — Signal location within model hierarchy. Grouping by model hierarchy can be helpful when you log data from a model that includes model or subsystem references.

Grouping signals adds rows for the hierarchical nodes, which you can expand to show the signals within that node. By default, the Simulation Data Inspector groups signals by domain, then by physical system hierarchy (if you have a Simscape license), and then by data hierarchy.

To remove grouping and display a flat list of signals in each run, select `None` for all grouping options.

Programmatic Use

To specify how to group signals programmatically, use the `Simulink.sdi.setTableGrouping` function.

Data to Stream from Parallel Simulations

When you run parallel simulations using the `parsim` function, you can stream logged simulation data to the Simulation Data Inspector. A dot next to the run name in the **Inspect** pane indicates the status of the simulation that corresponds to the run, so you can monitor simulation progress while visualizing the streamed data. You can control whether data streams from a parallel simulation based on the type of worker the data comes from.

By default, the Simulation Data Inspector is configured for manual import of data from parallel workers. You can use the Simulation Data Inspector programmatic interface to inspect the data on the worker and decide whether to send it to the client Simulation Data Inspector for further analysis. To manually move data from a parallel worker to the Simulation Data Inspector, use the `Simulink.sdi.sendWorkerRunToClient` function.

You may want to automatically stream data from parallel simulations that run on local workers or on local and remote workers. Streaming data from both local and remote workers may affect simulation performance, depending on how many simulations you run and how much data you log. When you choose to stream data from local workers or all parallel workers, all logged simulation data automatically shows in the Simulation Data Inspector.

Programmatic Use

You can configure Simulation Data Inspector support for parallel worker data programmatically using the `Simulink.sdi.enablePCTSupport` function.

Options for Saving and Loading Session Files

You can specify a maximum amount of memory to use while loading or saving a session file. By default, the Simulation Data Inspector uses a maximum of 100 MB of memory when you load or save a session file. You can specify a memory use limit as low as 50 MB.

To reduce the size of the saved session file, you can specify a compression option.

- `None` — Do not compress saved data.
- `Normal` — Compress the saved file as much as possible.
- `Fastest` — Compress the saved file less than `Normal` compression for faster save time.

Signal Display Units

Signals in the Simulation Data Inspector have two units properties: stored units and display units. The stored units represent the units of the data saved to disk. The display units specify how the Simulation Data Inspector displays the data. You can configure the Simulation Data Inspector to use a system of units to define the display units for all signals. You can choose either the **SI** or **US Customary** system of units, or you can display data using its stored units.

When you use a system of units to define display units for signals in the Simulation Data Inspector, the display units update for any signal with display units that are not valid for that unit system. For example, if you select **SI** units, the display units for a signal may update from `ft` to `m`.

Note The system of units you choose to use in the Simulation Data Inspector does not affect the stored units for any signal. You can convert the stored units for a signal using the `convertUnits` function. Conversion may result in loss of precision.

In addition to selecting a system of units, you can specify override units so that all signals of a given measurement type are displayed using consistent units. For example, if you want to visualize all signals that represent weight using units of `kg`, specify `kg` as an override unit.

Tip For a list of units supported by Simulink, enter `showunitslist` in the MATLAB Command Window.

You can also modify the display units for a specific signal using the **Properties** pane. For more information, see “Modify Signal Properties in the Simulation Data Inspector” (Simulink).

Programmatic Use

Configure the unit system and override units using the `Simulink.sdi.setUnitSystem` function. You can check the current units preferences using the `Simulink.sdi.getUnitSystem` function.

See Also

Functions

`Simulink.sdi.clearPreferences` | `Simulink.sdi.setRunNamingRule` | `Simulink.sdi.setTableGrouping` | `Simulink.sdi.enablePCTSupport` | `Simulink.sdi.setArchiveRunLimit` | `Simulink.sdi.setAutoArchiveMode`

More About

- “Iterate Model Design Using the Simulation Data Inspector” (Simulink)
- “How the Simulation Data Inspector Compares Data” (Simulink)
- “Compare Simulation Data” (Simulink)
- “Create Plots Using the Simulation Data Inspector” (Simulink)
- “Modify Signal Properties in the Simulation Data Inspector” (Simulink)

How the Simulation Data Inspector Compares Data

You can tailor the Simulation Data Inspector comparison process to fit your requirements in multiple ways. When comparing runs, the Simulation Data Inspector:

1. Aligns signal pairs in the **Baseline** and **Compare To** runs based on the **Alignment** settings.

   The Simulation Data Inspector does not compare signals that it cannot align.
2. Synchronizes aligned signal pairs according to the specified **Sync Method**.

   Values for time points added in synchronization are interpolated according to the specified **Interpolation Method**.
3. Computes the difference of the signal pairs.
4. Compares the difference result against specified tolerances.

When the comparison run completes, the results of the comparison are displayed in the navigation pane.

Status	Comparison Result
	Difference falls within the specified tolerance.
	Difference violates specified tolerance.
	The signal does not align with a signal from the **Compare To** run.

When you compare signals with differing time intervals, the Simulation Data Inspector compares the signals on their overlapping interval.

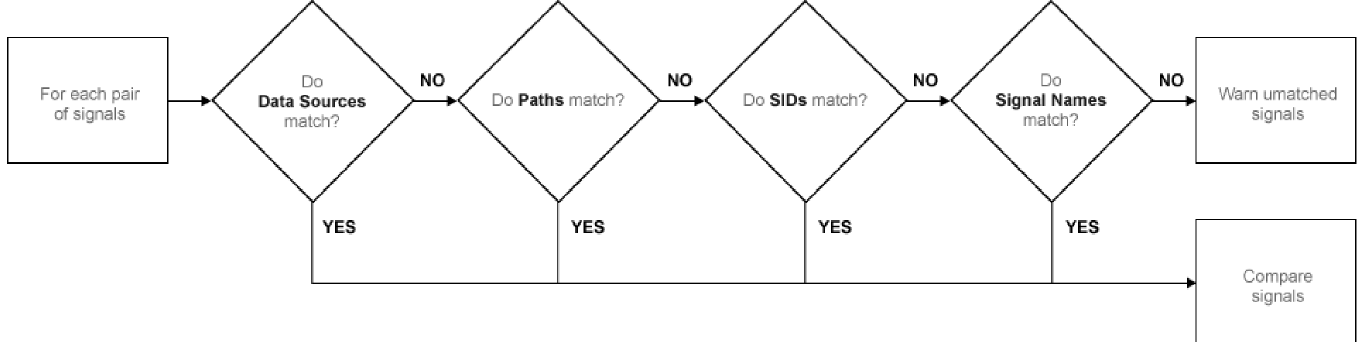
Signal Alignment

In the alignment step, the Simulation Data Inspector decides which signal from the **Compare To** run pairs with a given signal in the **Baseline** run. When you compare signals with the Simulation Data Inspector, you complete the alignment step by selecting the **Baseline** and **Compare To** signals.

The Simulation Data Inspector aligns signals using a combination of their Data Source, Path, SID, and Signal Name properties.

Property	Description
Data Source	Path of the variable in the MATLAB workspace for data imported from the workspace
Path	Block path for the source of the data in its model
SID	Automatically assigned Simulink identifier
Signal Name	Name of the signal in the model

With the default alignment settings, the Simulation Data Inspector aligns signals between runs according to this flow chart.

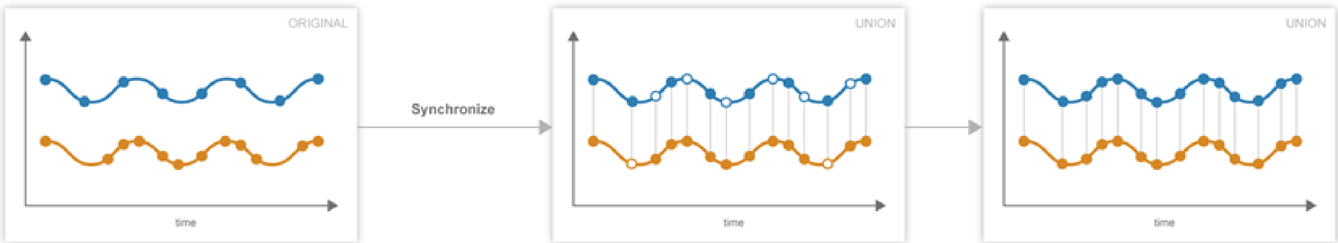


You can specify the priority for each of the signal properties used for alignment in the Simulation Data Inspector **Preferences**. The **Align By** field specifies the highest priority property used to align signals. The priority drops with each subsequent **Then By** field. You must specify a primary alignment property in the **Align By** field, but you can leave any number of the **Then By** fields blank.

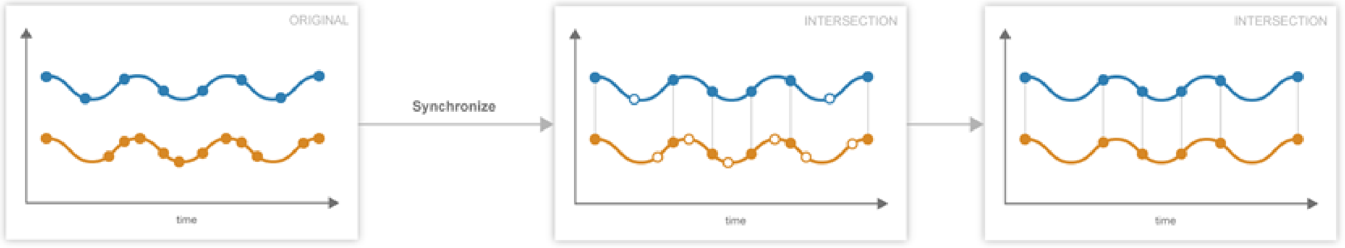
Synchronization

Often, signals that you want to compare don't contain the exact same set of time points. The synchronization step in Simulation Data Inspector comparisons resolves discrepancies in signals' time vectors. You can choose `union` or `intersection` as the synchronization method.

When you specify `union` synchronization, the Simulation Data Inspector builds a time vector that includes every sample time between the two signals. For each sample time not originally present in either signal, the Simulation Data Inspector interpolates the value. The second graph in the illustration shows the union synchronization process, where the Simulation Data Inspector identifies samples to add in each signal, represented by the unfilled circles. The final plot shows the signals after the Simulation Data Inspector has interpolated values for the added time points. The Simulation Data Inspector computes the difference using the signals in the final graph, so that the computed difference signal contains all the data points between the signals.



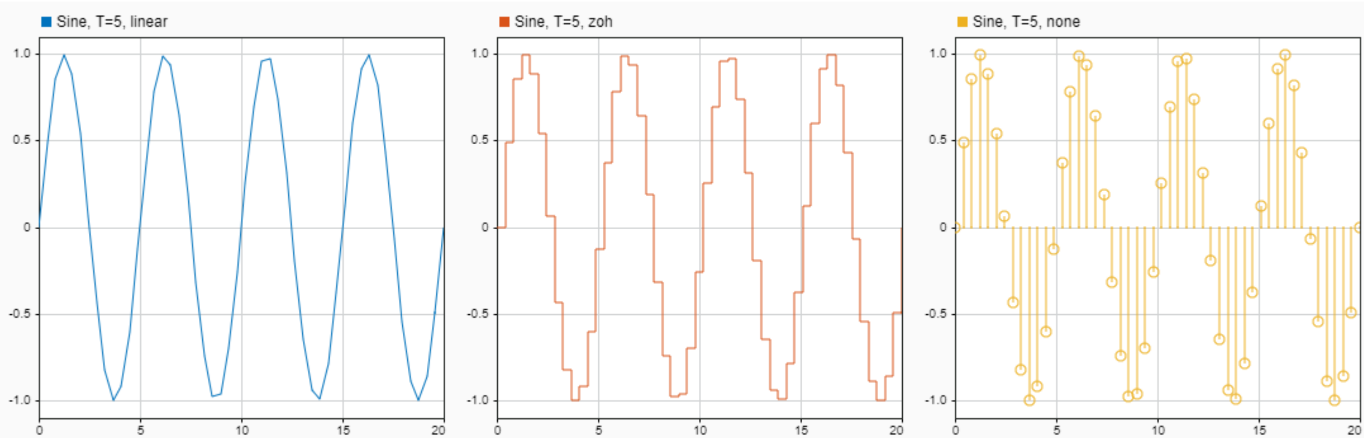
When you specify `intersection` synchronization, the Simulation Data Inspector uses only the sample times present in both signals in the comparison. In the second graph, the Simulation Data Inspector identifies samples that do not have a corresponding sample for comparison, shown as unfilled circles. The final graph shows the signals used for the comparison, without the samples identified in the second graph.



The choice between the synchronization options involves a trade off between speed and accuracy. The interpolation required by `union` synchronization takes time, but provides a more precise result. When you use `intersection` synchronization, the comparison finishes quickly because the Simulation Data Inspector computes the difference for fewer data points and does not interpolate. However, some data is discarded and precision lost with `intersection` synchronization.

Interpolation

The interpolation property of a signal determines how the Simulation Data Inspector displays the signal and how additional data values are computed in synchronization. You can choose to interpolate your data with a zero-order hold (zoh) or a linear approximation. You can also specify no interpolation.



When you specify `zoh` or `none` for the **Interpolation Method**, the Simulation Data Inspector replicates the data of the previous sample for interpolated sample times. When you specify `linear` interpolation, the Simulation Data Inspector uses samples on either side of the interpolated point to linearly approximate the interpolated value. Typically, discrete signals use `zoh` interpolation and continuous signals use `linear` interpolation. You can specify the **Interpolation Method** for your signals in the signal properties.

Tolerance Specification

The Simulation Data Inspector allows you to specify the scope and value of the tolerance for your signal. You can define a tolerance band using any combination of absolute, relative, and time tolerance values, and you can specify whether the specified tolerance applies to an individual signal or to all the signals in a run.

Tolerance Scope

In the Simulation Data Inspector, you can specify the tolerance for your data globally or for an individual signal. Global tolerance values apply to all signals in a run that do not have **Override Global Tol** set to `yes`. You can specify global tolerance values for your data at the top of the graphical viewing area in the **Compare** view. To specify signal specific tolerance values, edit the signal properties and ensure the **Override Global Tol** property is set to `yes`.

Tolerance Computation

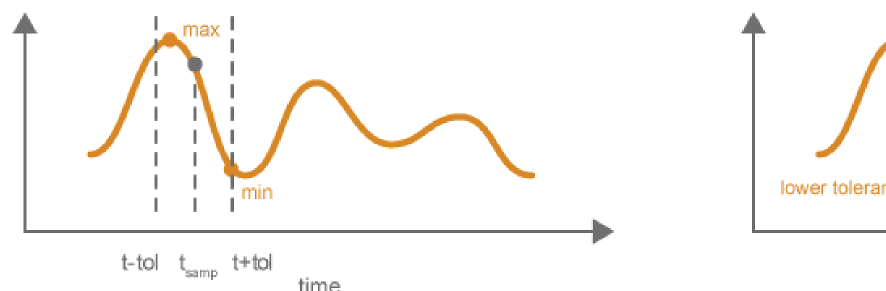
In the Simulation Data Inspector, you can specify a tolerance band for your run or signal using a combination of absolute, relative, and time tolerance values. When you specify the tolerance for your run or signal using multiple types of tolerances, each tolerance can yield a different answer for the tolerance at each point. The Simulation Data Inspector computes the overall tolerance band by selecting the most lenient tolerance result for each data point.

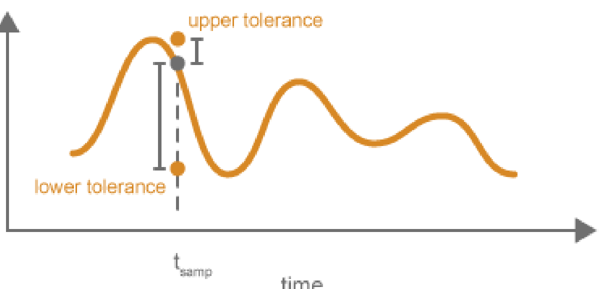
When you define your tolerance using only the absolute and relative tolerance properties, the Simulation Data Inspector computes the tolerance for each point as a simple maximum.

```
tolerance = max(absoluteTolerance,relativeTolerance*abs(baselineData));
```

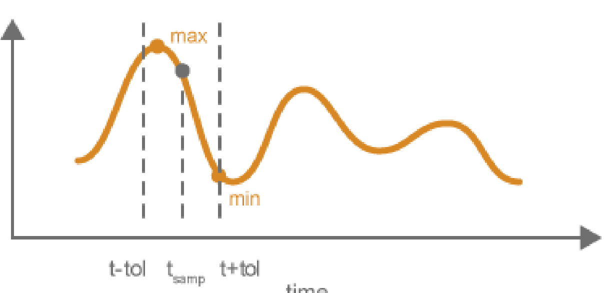
The upper boundary of the tolerance band is formed by adding `tolerance` to the **Baseline** signal. Similarly, the Simulation Data Inspector computes the lower boundary of the tolerance band by subtracting `tolerance` from the **Baseline** signal.

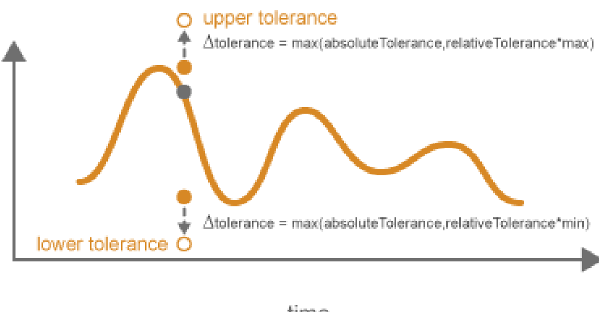
When you specify a time tolerance, the Simulation Data Inspector evaluates the time tolerance first, over a time interval defined as $[(t_{samp}-tol), (t_{samp}+tol)]$ for each sample. The Simulation Data Inspector builds the lower tolerance band by selecting the minimum point on the interval for each sample. Similarly, the maximum point on the interval defines the upper tolerance for each sample.





If you specify a tolerance band using an absolute or relative tolerance in addition to a time tolerance, the Simulation Data Inspector applies the time tolerance first, and then applies the absolute and relative tolerances to the maximum and minimum points selected with the time tolerance.






```
upperTolerance = max + max(absoluteTolerance,relativeTolerance*max)

lowerTolerance = min - max(absoluteTolerance,relativeTolerance*min)
```

Limitations

The Simulation Data Inspector does not support comparing:

- Signals of data types `int64` or `uint64`.
- Variable-size signals.

See Also

Related Examples

- "Compare Simulation Data" (Simulink)

Save and Share Simulation Data Inspector Data and Views

After you inspect, analyze, or compare your data in the Simulation Data Inspector, you can share your results with others. The Simulation Data Inspector provides several options for sharing and saving your data and results, depending on your needs. With the Simulation Data Inspector, you can:

- Save your data and layout modifications in a Simulation Data Inspector session.
- Share your layout modifications in a Simulation Data Inspector view.
- Share images and figures of plots you create in the Simulation Data Inspector.
- Create a Simulation Data Inspector report.
- Export data to the workspace.
- Export data to a file.

Save and Load Simulation Data Inspector Sessions

If you want to save or share data along with a configured view in the Simulation Data Inspector, save your data and settings in a Simulation Data Inspector session. You can save sessions as MAT- or MLDATX-files. The default format is MLDATX. When you save a Simulation Data Inspector session, the session file contains:

- All runs, data, and properties from the **Inspect** pane, including which run is the current run and which runs are in the archive.
- Plot display selection for signals in the **Inspect** pane.
- Subplot layout and line style and color selections.

Note Comparison results and global tolerances are not saved in Simulation Data Inspector sessions.

To save a Simulation Data Inspector session:

1. Hover over the save icon on the left side bar. Then, click **Save As**.

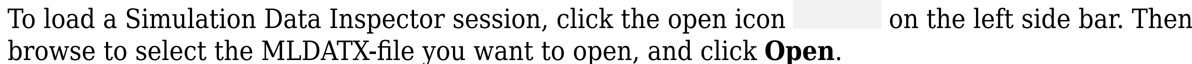
2. Name the file.
3. Browse to the location where you want to save the session, and click **Save**.

   For large datasets, a status overlay in the bottom right of the graphical viewing area displays information about the progress of the save operation and allows you to cancel the save operation.

The **Save** tab of the Simulation Data Inspector preferences menu on the left side bar allows you to configure options related to save operations for MLDATX-files. You can set a limit as low as 50MB on the amount of memory used for the save operation. You can also select one of three **Compression** options:

- `None`, the default, applies no compression during the save operation.

- `Normal` creates the smallest file size.
- `Fastest` creates a smaller file size than you would get by selecting `None`, but provides a faster save time than `Normal`.



To load a Simulation Data Inspector session, click the open icon on the left side bar. Then, browse to select the MLDATX-file you want to open, and click **Open**.

Alternatively, you can double-click the MLDATX-file. MATLAB and the Simulation Data Inspector open if they are not already open.

When the Simulation Data Inspector already contains runs and you open a session, all of the runs in the session move to the archive. The view updates to show plotted signals from the session file. You can drag runs between the work area and archive as desired.

When the Simulation Data Inspector does not contain runs and you open a session, the Simulation Data Inspector puts runs in the work area and archive as specified in the file.

Share Simulation Data Inspector Views

When you have different sets of data that you want to visualize the same way, you can save a view. A view saves the layout and appearance characteristics of the Simulation Data Inspector without saving the data. Specifically, a view saves:

- Plot visualization type, layout, axis ranges, linking characteristics, and normalized axes
- Location of signals in the plots, including plotted signals in the archive
- Signal grouping and columns on display in the **Inspect** pane
- Signal color and line styling

To save a view:

1. Click Visualizations and layouts .
2. In **Saved Views**, click **Save current view**.
3. In the dialog box, specify a name for the view and browse to the location where you want to save the MLDATX-file.
4. Click **Save**.

To load a view:

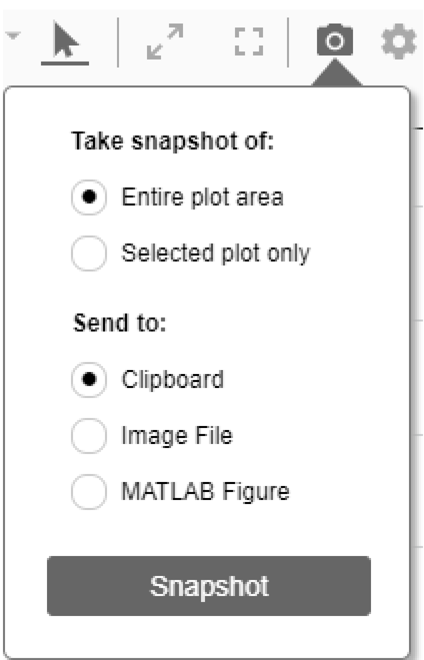
1. Click Visualizations and layouts .
2. In **Saved Views**, click **Open saved view**.
3. Browse to the view you would like to load, and click **Open**.

Share Simulation Data Inspector Plots

Use the snapshot feature to share the plots you generate in the Simulation Data Inspector. You can export your plots to the clipboard to paste into a document, as an image file, or to a MATLAB figure.

You can choose to capture the entire plot area, including all subplots in the plot area, or to capture only the selected subplot.

Click the camera icon on the toolbar to access the snapshot menu. Use the radio buttons to select the area you want to share and how you want to share the plot. After you make your selections, click **Snapshot** to export the plot.



If you create an image, select where you would like to save the image in the file browser.

You can create snapshots of your plots in the Simulation Data Inspector programmatically using `Simulink.sdi.snapshot`.

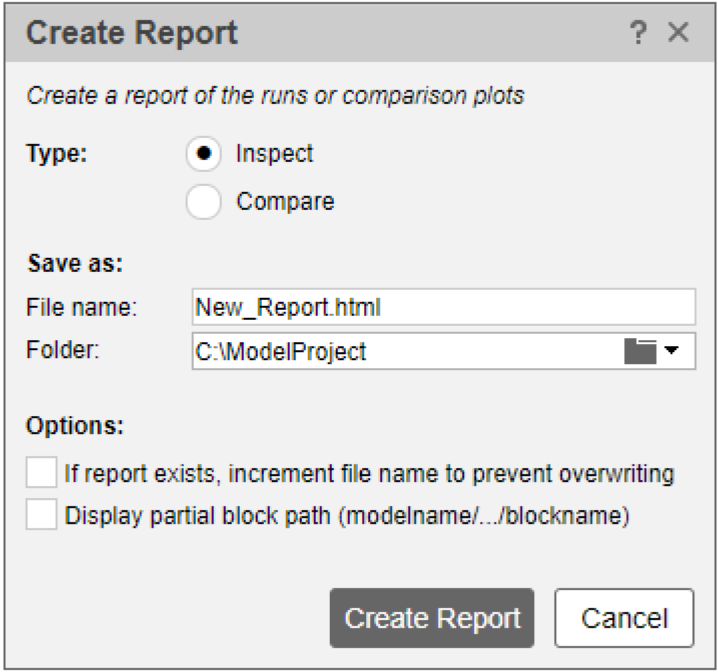
Create Simulation Data Inspector Report

To generate documentation of your results quickly, create a Simulation Data Inspector report. You can create a report of your data in either the **Inspect** or the **Compare** pane. The report is an HTML file that includes information about all the signals and plots in the active pane. The report includes all signal information displayed in the signal table in the navigation pane. For more information about configuring the table, see “Inspect Metadata” (Simulink).

To generate a Simulation Data Inspector Report:

1. Click the create report icon on the left bar.
2. Specify the type of report you want to create.
   - Select **Inspect** to include the plots and signals from the **Inspect** pane.

- Select **Compare** to include the data and plots from the **Compare** pane. When you generate a **Compare Runs** report, you can choose to **Report only mismatched signals** or to **Report all signals**. If you select **Report only mismatched signals**, the report shows only signal comparisons that are not within the specified tolerances.

3. Specify a **File name** for the report, and navigate to the **Folder** where you want to save the report.
4. Click **Create Report**.

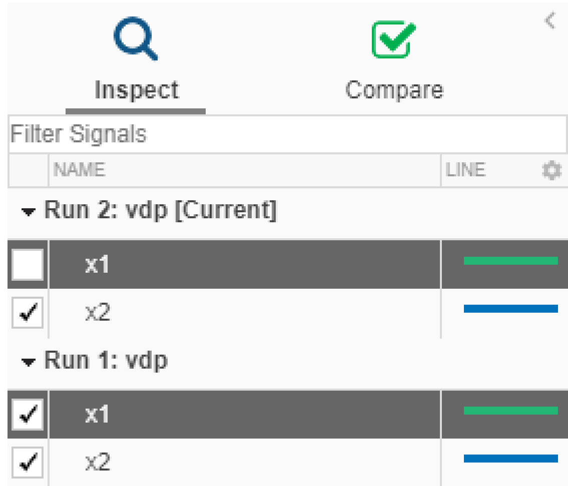
   The generated report automatically opens in your default browser.

Export Data to the Workspace or a File

You can use the Simulation Data Inspector to export data to the base workspace, a MAT file, or a Microsoft Excel file. You can export a selection of runs and signals, runs in the work area, or all runs in the **Inspect** pane, including the **Archive**.

When you export a selection of runs and signals, make the selection of data to export before clicking the export button .

Only the selected runs and signals are exported. In this example, only the `x1` signals from Run 1 and Run 2 are exported. The check box selections for plotting data do not affect whether a signal is exported.



When you export a single signal to the workspace or a MAT file, the signal is exported to a `timeseries` object. Data exported to the workspace or a MAT file for a run or multiple signals is stored as a `Simulink.SimulationData.Dataset` object.

To export data to a file, select the **File** option in the **Export** dialog. You can specify a file name and browse to the location where you want to save the exported file. When you export data to a MAT file, a single exported signal is stored as a `timeseries` object, and runs or multiple signals are stored as a `Simulink.SimulationData.Dataset` object. When you export data to a Microsoft Excel file, the data is stored using the format described in “Microsoft Excel Import, Export, and Logging Format” (Simulink).

To export to a Microsoft Excel file, select the XLSX extension from the drop-down. When you export data to a Microsoft Excel file, you can specify additional options for the format of the data in the exported file. If the file name you provided already exists, you can choose to overwrite the entire file or to only overwrite sheets containing data that corresponds to the exported data. You can also choose which metadata to include and whether signals with identical time data share a time column in the exported file.

Export Video Signal to an MP4 File

You can export a 2D or 3D signal that contains RGB or monochrome video data to an MP4 file using the Simulation Data Inspector. For example, when you log a video signal in a simulation, you can export the data to an MP4 file and view the video using a video player. To export a video signal to an MP4 file:

1. Select the signal you want to export.
2. Click Export in the toolbar on the left or right-click the signal and select **Export**.
3. In the Export dialog box, choose to export **Selected runs and signals** to a file.
4. Specify a file name and the path to the location where you want to save the file.
5. Select `MP4 video file` from the list and click **Export**.

For the option to export to an MP4 file to be available:

- You must export only one signal at a time.
- The selected signal must be 2D or 3D and contain RGB or monochrome video data.
- The selected signal must be represented in the Simulation Data Inspector as a single signal with multidimensional sample values.

  You may need to convert the signal representation before exporting the signal data. For more information, see “Analyze Multidimensional Signal Data” (Simulink).
- The data type for the signal values must be `double`, `single`, or `uint8`.

Exporting a video signal to an MP4 file is not supported for Linux operating systems.

See Also

Functions
`Simulink.sdi.saveView`

Related Examples

- “View Data in the Simulation Data Inspector” (Simulink)
- “Inspect Simulation Data” (Simulink)
- “Compare Simulation Data” (Simulink)

Inspect and Compare Data Programmatically

You can harness the capabilities of the Simulation Data Inspector from the MATLAB command line using the Simulation Data Inspector API.

The Simulation Data Inspector organizes data in runs and signals, assigning a unique numeric identification to each run and signal. Some Simulation Data Inspector API functions use the run and signal IDs to reference data, rather than accepting the run or signal itself as an input. To access the run IDs in the workspace, you can use `Simulink.sdi.getAllRunIDs` or `Simulink.sdi.getRunIDByIndex`. You can access signal IDs through a `Simulink.sdi.Run` object using the `getSignalIDByIndex` method.

The `Simulink.sdi.Run` and `Simulink.sdi.Signal` classes provide access to your data and allow you to view and modify run and signal metadata. You can modify the Simulation Data Inspector preferences using functions like `Simulink.sdi.setSubPlotLayout`, `Simulink.sdi.setRunNamingRule`, and `Simulink.sdi.setMarkersOn`. To restore the Simulation Data Inspector's default settings, use `Simulink.sdi.clearPreferences`.

Create a Run and View the Data

This example shows how to create a run, add data to it, and then view the data in the Simulation Data Inspector.

Create Data for the Run

Create `timeseries` objects to contain data for a sine signal and a cosine signal. Give each `timeseries` object a descriptive name.

```
time = linspace(0,20,100);

sine_vals = sin(2*pi/5*time);
sine_ts = timeseries(sine_vals,time);
sine_ts.Name = 'Sine, T=5';

cos_vals = cos(2*pi/8*time);
cos_ts = timeseries(cos_vals,time);
cos_ts.Name = 'Cosine, T=8';
```

Create a Run and Add Data

Use the `Simulink.sdi.view` function to open the Simulation Data Inspector.

```
Simulink.sdi.view
```

To import data into the Simulation Data Inspector from the workspace, create a `Simulink.sdi.Run` object using the `Simulink.sdi.Run.create` function. Add information about the run to its metadata using the `Name` and `Description` properties of the `Run` object.

```
sinusoidsRun = Simulink.sdi.Run.create;
sinusoidsRun.Name = 'Sinusoids';
sinusoidsRun.Description = 'Sine and cosine signals with different frequencies';
```

Use the `add` function to add the data you created in the workspace to the empty run.

```
add(sinusoidsRun,'vars',sine_ts,cos_ts);
```

Plot the Data in the Simulation Data Inspector

Use the `getSignalByIndex` function to access `Simulink.sdi.Signal` objects that contain the signal data. You can use the `Simulink.sdi.Signal` object properties to specify the line style and color for the signal and plot it in the Simulation Data Inspector. Specify the `LineColor` and `LineDashed` properties for each signal.

```
sine_sig = getSignalByIndex(sinusoidsRun,1);
sine_sig.LineColor = [0 0 1];
sine_sig.LineDashed = '-.';

cos_sig = sinusoidsRun.getSignalByIndex(2);
cos_sig.LineColor = [0 1 0];
cos_sig.LineDashed = '--';
```

Use the `Simulink.sdi.setSubPlotLayout` function to configure a 2-by-1 subplot layout in the Simulation Data Inspector plotting area. Then use the `plotOnSubplot` function to plot the sine signal on the top subplot and the cosine signal on the lower subplot.

```
Simulink.sdi.setSubPlotLayout(2,1);

plotOnSubPlot(sine_sig,1,1,true);
plotOnSubPlot(cos_sig,2,1,true);
```

Close the Simulation Data Inspector and Save Your Data

When you have finished inspecting the plotted signal data, you can close the Simulation Data Inspector and save the session to an MLDATX file.

```
Simulink.sdi.close('sinusoids.mldatx')
```

Compare Two Signals in the Same Run

You can use the Simulation Data Inspector programmatic interface to compare signals within a single run. This example compares the input and output signals of an aircraft longitudinal controller.

First, load the session that contains the data.

```
Simulink.sdi.load('AircraftExample.mldatx');
```

Use the `Simulink.sdi.Run.getLatest` function to access the latest run in the data.

```
aircraftRun = Simulink.sdi.Run.getLatest;
```

Then, you can use the `Simulink.sdi.getSignalsByName` function to access the `Stick` signal, which represents the input to the controller, and the `alpha, rad` signal that represents the output.

```
stick = getSignalsByName(aircraftRun,'Stick');
alpha = getSignalsByName(aircraftRun,'alpha, rad');
```

Before you compare the signals, you can specify a tolerance value to use for the comparison. Comparisons use tolerance values specified for the baseline signal in the comparison, so set an absolute tolerance value of `0.1` on the `Stick` signal.

```
stick.AbsTol = 0.1;
```

Now, compare the signals using the `Simulink.sdi.compareSignals` function. The `Stick` signal is the baseline, and the `alpha, rad` signal is the signal to compare against the baseline.

```
comparisonResults = Simulink.sdi.compareSignals(stick.ID,alpha.ID);
match = comparisonResults.Status
```

```
match = 
  ComparisonSignalStatus enumeration

    OutOfTolerance
```

The comparison result is out of tolerance. You can use the `Simulink.sdi.view` function to open the Simulation Data Inspector to view and analyze the comparison results.

Compare Runs with Global Tolerance

You can specify global tolerance values to use when comparing two simulation runs. Global tolerance values are applied to all signals within the run. This example shows how to specify global tolerance values for a run comparison and how to analyze and save the comparison results.

First, load the session file that contains the data to compare. The session file contains data for four simulations of an aircraft longitudinal controller. This example compares data from two runs that use different input filter time constants.

```
Simulink.sdi.load('AircraftExample.mldatx');
```

To access the run data to compare, use the `Simulink.sdi.getAllRunIDs` (Simulink) function to get the run IDs that correspond to the last two simulation runs.

```
runIDs = Simulink.sdi.getAllRunIDs;
runID1 = runIDs(end - 1);
runID2 = runIDs(end);
```

Use the `Simulink.sdi.compareRuns` (Simulink) function to compare the runs. Specify a global relative tolerance value of `0.2` and a global time tolerance value of `0.5`.

```
runResult = Simulink.sdi.compareRuns(runID1,runID2,'reltol',0.2,'timetol',0.5);
```

Check the `Summary` property of the returned `Simulink.sdi.DiffRunResult` object to see whether signals were within the tolerance values or out of tolerance.

```
runResult.Summary
```

```
ans = struct with fields:
       OutOfTolerance: 0
      WithinTolerance: 3
            Unaligned: 0
        UnitsMismatch: 0
                Empty: 0
             Canceled: 0
          EmptySynced: 0
     DataTypeMismatch: 0
```

```
        TimeMismatch: 0
    StartStopMismatch: 0
         Unsupported: 0
```

All three signal comparison results fell within the specified global tolerance.

You can save the comparison results to an MLDATX file using the `saveResult` (Simulink) function.

```
saveResult(runResult,'InputFilterComparison');
```

Analyze Simulation Data Using Signal Tolerances

You can programmatically specify signal tolerance values to use in comparisons performed using the Simulation Data Inspector. In this example, you compare data collected by simulating a model of an aircraft longitudinal flight control system. Each simulation uses a different value for the input filter time constant and logs the input and output signals. You analyze the effect of the time constant change by comparing results using the Simulation Data Inspector and signal tolerances.

First, load the session file that contains the simulation data.

```
Simulink.sdi.load('AircraftExample.mldatx');
```

The session file contains four runs. In this example, you compare data from the first two runs in the file. Access the `Simulink.sdi.Run` objects for the first two runs loaded from the file.

```
runIDs = Simulink.sdi.getAllRunIDs;
runIDTs1 = runIDs(end-3);
runIDTs2 = runIDs(end-2);
```

Now, compare the two runs without specifying any tolerances.

```
noTolDiffResult = Simulink.sdi.compareRuns(runIDTs1,runIDTs2);
```

Use the `getResultByIndex` function to access the comparison results for the `q` and `alpha` signals.

```
qResult = getResultByIndex(noTolDiffResult,1);
alphaResult = getResultByIndex(noTolDiffResult,2);
```

Check the `Status` of each signal result to see whether the comparison result fell within our out of tolerance.

```
qResult.Status
```

```
ans = 
  ComparisonSignalStatus enumeration

    OutOfTolerance
```

```
alphaResult.Status
```

```
ans = 
  ComparisonSignalStatus enumeration
```

```
    OutOfTolerance
```

The comparison used a value of `0` for all tolerances, so the `OutOfTolerance` result means the signals are not identical.

You can further analyze the effect of the time constant by specifying tolerance values for the signals. Specify the tolerances by setting the properties for the `Simulink.sdi.Signal` objects that correspond to the signals being compared. Comparisons use tolerances specified for the baseline signals. This example specifies a time tolerance and an absolute tolerance.

To specify a tolerance, first access the `Signal` objects from the baseline run.

```
runTs1 = Simulink.sdi.getRun(runIDTs1);
qSig = getSignalsByName(runTs1,'q, rad/sec');
alphaSig = getSignalsByName(runTs1,'alpha, rad');
```

Specify an absolute tolerance of `0.1` and a time tolerance of `0.6` for the `q` signal using the `AbsTol` and `TimeTol` properties.

```
qSig.AbsTol = 0.1;
qSig.TimeTol = 0.6;
```

Specify an absolute tolerance of `0.2` and a time tolerance of `0.8` for the `alpha` signal.

```
alphaSig.AbsTol = 0.2;
alphaSig.TimeTol = 0.8;
```

Compare the results again. Access the results from the comparison and check the `Status` property for each signal.

```
tolDiffResult = Simulink.sdi.compareRuns(runIDTs1,runIDTs2);
qResult2 = getResultByIndex(tolDiffResult,1);
alphaResult2 = getResultByIndex(tolDiffResult,2);

qResult2.Status

ans = 
  ComparisonSignalStatus enumeration

    WithinTolerance


alphaResult2.Status

ans = 
  ComparisonSignalStatus enumeration

    WithinTolerance
```

See Also

Simulation Data Inspector

Related Examples

- "Compare Simulation Data" (Simulink)
- "How the Simulation Data Inspector Compares Data" (Simulink)
- "Create Plots Using the Simulation Data Inspector" (Simulink)

Limit the Size of Logged Data

In this section...

“Limit the Number of Runs Retained in the Simulation Data Inspector Archive” on page 32-48
“Specify a Minimum Disk Space Requirement or Maximum Size for Logged Data” on page 32-48
“View Data Only During Simulation” on page 32-49
“Reduce the Number of Data Points Logged from Simulation” on page 32-49

Logging simulation data can produce large amounts of data that fill up disk space. Such situations include logging many signals, logging data for long simulations, and running many simulations without deleting run data from the Simulation Data Inspector. You can choose among several options to limit the size of logged simulation data. You can:

- Limit the number of runs retained in the Simulation Data Inspector archive.
- Reduce the number of data points logged in each simulation.
- Specify a minimum disk space requirement or maximum size for logged data.
- Configure logging for only viewing data during simulation.

Depending on your requirements, you can use more than one strategy to limit the size of logged data.

Limit the Number of Runs Retained in the Simulation Data Inspector Archive

When you run multiple simulations in a single MATLAB session, logged simulation data accumulates in the Simulation Data Inspector even if you overwrite the logging data in the MATLAB workspace. To reduce the amount of data the Simulation Data Inspector retains, you can configure a limit for the number of runs stored in the archive. When the number of runs in the archive reaches the size limit, the Simulation Data Inspector starts to delete runs from the archive on a first-in, first-out basis.

Configure the archive **Size** setting in the Simulation Data Inspector preferences. The size limit only applies to runs in the archive. For the Simulation Data Inspector to automatically limit data retention, select **Automatically archive** and specify the maximum number of runs to retain in the archive. By default, **Automatically archive** is enabled with an archive size limit of twenty runs. If you experience issues with logged data consuming too much disk space, consider adjusting the size limit for the archive in the Simulation Data Inspector preferences.

Specify a Minimum Disk Space Requirement or Maximum Size for Logged Data

You can use preferences in the Simulation Data Inspector to directly limit the size of logged data by specifying a minimum amount of disk space to leave free or by specifying a maximum size for logged data on disk. Each setting accounts for all kinds of logged data. By default, logged data must leave at least 100 MB of free disk space with no maximum size limit. Specify the required disk space and maximum size in GB, and specify `0` to apply no disk space requirement or no maximum size limit.

When you specify a minimum disk space requirement or a maximum size for logged data, you can also specify whether to prioritize retaining data from the current simulation or data from prior simulations when approaching the limit. By default, the Simulation Data Inspector prioritizes

retaining data for the current run. As the free disk space or logged data size approaches the limit, prior runs are deleted first to free up space for data being logged in the current run. If deleting runs does not free up enough space, recording is disabled. To prioritize retaining prior data, change the **When low on disk space** setting to **Keep prior runs and stop recording**. You see a warning message when prior runs are deleted and when recording is disabled. If recording is disabled due to the size of logged data, you need to change the **Record Mode** back to **View and record data** to continue logging data, after you have freed up disk space.

View Data Only During Simulation

In some situations, you may want to only view the data for logged signals and not save the values. For example, when using the Simulation Data Inspector to visualize data streaming from hardware, you may only want to view the data live and not record it. You can change the **Record mode** in the Simulation Data Inspector preferences to **View during simulation only** so that logged data is not saved and you can still view the data during simulation. The **Record mode** is reset to **View and record data** at the start of each MATLAB session.

When you change the **Record mode** to **View during simulation only**:

- Logged data is not available in the Simulation Data Inspector or workspace after simulation.
- You can view data using dashboard blocks, scopes, and the Simulation Data Inspector, but plots clear when you pan or zoom.
- You cannot access logged data during simulation using the Simulation Data Inspector programmatic interface.

Reduce the Number of Data Points Logged from Simulation

Model configuration parameters and signal properties allow you to limit the number of data points logged in a simulation. Be sure to carefully consider data requirements when limiting logged data points. Limiting data can skip critical time points, and can lead to aliasing, if your effective sample rate is too low.

You can reduce the number of data points using:

- Decimation — Log every *n*th signal value.
- Limit data points to last — Only log the last *n* signal values.
- Logging intervals — Specify specific time intervals in which to log data.

For details, see “Specify Signal Values to Log” (Simulink).

See Also

Tools
Simulation Data Inspector

Related Examples

- “Specify Signal Values to Log” (Simulink)
- “Configure the Simulation Data Inspector” (Simulink)