

SIMULINK[®]

Dynamic System Simulation for MATLAB[®]

Modeling
└─

Simulation
└─

Implementation
└─



Writing S-Functions
Version 4

How to Contact The MathWorks:



508-647-7000 Phone



508-647-7001 Fax



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



<http://www.mathworks.com> Web
<ftp.mathworks.com> Anonymous FTP server
<comp.soft-sys.matlab> Newsgroup



support@mathworks.com Technical support
suggest@mathworks.com Product enhancement suggestions
bugs@mathworks.com Bug reports
doc@mathworks.com Documentation error reports
subscribe@mathworks.com Subscribing user registration
service@mathworks.com Order status, license renewals, passcodes
info@mathworks.com Sales, pricing, and general information

Writing S-Functions

© COPYRIGHT 1998 - 2000 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 or for the federal government of the United States. By accepting delivery of the Program, the government hereby agrees that this software qualifies as "commercial" computer software within the meaning of FAR Part 12.212, DFARS Part 227.7202-1, DFARS Part 227.7202-3, DFARS Part 252.227-7013, and DFARS Part 252.227-7014. The terms and conditions of The MathWorks, Inc. Software License Agreement shall pertain to the government's use and disclosure of the Program and Documentation, and shall supersede any conflicting contractual terms or conditions. If this license fails to meet the government's minimum needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to MathWorks.

MATLAB, Simulink, Stateflow, Handle Graphics, and Real-Time Workshop are registered trademarks, and Target Language Compiler is a trademark of The MathWorks, Inc.

Other product or brand names are trademarks or registered trademarks of their respective holders.

Printing History: October 1998 First printing Revised for Simulink 3.0 (Release 11)
November 2000 Second printing Revised for Simulink 4.0 (Release 12)

Overview of S-Functions

1

Introduction	1-2
What Is an S-Function?	1-2
Using S-Functions in Models	1-2
Passing Parameters to S-Functions	1-3
When to Use an S-Function	1-5
How S-Functions Work	1-5
Implementing S-Functions	1-9
S-Function Concepts	1-11
S-Function Examples	1-16

Writing M S-Functions

2

Introduction	2-2
S-Function Arguments	2-2
S-Function Outputs	2-3
Defining S-Function Block Characteristics	2-4
A Simple M-File S-Function Example	2-5
Examples of M-File S-Functions	2-8
Example - Continuous State S-Function	2-8
Example - Discrete State S-Function	2-11
Example - Hybrid System S-Functions	2-13
Example - Variable Sample Time S-Functions	2-16
Processing S-Function Parameters	2-19

Writing S-Functions in C

3

Introduction	3-2
Example of a Basic C MEX S-Function	3-3
Templates for C S-Functions	3-9
S-Function Source File Requirements	3-9
The SimStruct	3-11
Compiling C S-Functions	3-12
How Simulink Interacts with C S-Functions	3-13
Process View	3-13
Data View	3-17
Writing Callback Methods	3-21
Converting Level 1 C MEX S-Functions to Level 2	3-22
Obsolete Macros	3-24

Creating C++ S-Functions

4

Overview	4-2
Source File Format	4-3
Making C++ Objects Persistent	4-7
Building C++ S-Functions	4-8

Creating Ada S-Functions

5

Introduction	5-2
Ada S-Function Source File Format	5-3
Ada S-Function Specification	5-3
Ada S-Function Body	5-4
Writing Callback Methods in Ada	5-6
Callbacks Invoked By Simulink	5-6
Implementing Callbacks	5-7
Omitting Optional Callback Methods	5-7
SimStruct Functions	5-7
Building an Ada S-Function	5-9
Using an Ada S-Function in a Model	5-10
Example of an Ada S-Function	5-11

Creating Fortran S-Functions

6

Introduction	6-2
Level 1 Versus Level 2 S-Functions	6-2
Creating Level 1 Fortran S-Functions	6-3
The Fortran MEX Template File	6-3
Example	6-3
Inline Code Generation Example	6-6
Creating Level 2 Fortran S-Functions	6-7
Template File	6-7
C/Fortran Interfacing Tips	6-7
Constructing the Gateway	6-11
An Example C-MEX S-Function Calling Fortran Code	6-13

Porting Legacy Code	6-15
Find the States	6-15
Sample Times	6-15
Multiple Instances	6-15
Use Flints If Needed	6-16
Considerations for Real Time	6-16

Implementing Block Features

7

Introduction	7-2
Dialog Parameters	7-3
Tunable Parameters	7-4
Run-Time Parameters	7-6
Creating Run-Time Parameters	7-7
Updating Run-Time Parameters	7-7
Input and Output Ports	7-9
Creating Input Ports	7-9
Creating Output Ports	7-11
Scalar Expansion of Inputs	7-12
Masked Multiport S-Functions	7-13
Custom Data Types	7-15
Sample Times	7-16
Block-Based Sample Times	7-16
Port-Based Sample Times	7-19
Specifying the Number of Sample Times in mdlInitializeSizes	7-20
Hybrid Block-Based and Port-Based Sample Times	7-20
Multirate S-Function Blocks	7-21
Synchronizing Multirate S-Function Blocks	7-22
Work Vectors	7-24
Work Vectors and Zero Crossings	7-26

An Example Involving a Pointer Work Vector	7-26
Memory Allocation	7-28
Function-Call Subsystems	7-29
Handling Errors	7-31
Exception Free Code	7-31
ssSetErrorStatus Termination Criteria	7-32
S-Function Examples	7-34
Example - Continuous State S-Function	7-34
Example - Discrete State S-Function	7-38
Example - Hybrid System S-Functions	7-42
Example - Variable Step S-Function	7-45
Example - Zero Crossing S-Function	7-48
Example - Time Varying Continuous Transfer Function	7-60

8 Writing S-Functions for Real-Time Workshop

8

Introduction	8-2
Classes of Problems Solved by S-Functions	8-2
Types of S-Functions	8-3
Basic Files Required for Implementation	8-5
Noninlined S-Functions	8-7
S-Function Module Names for Real-Time Workshop Builds	8-7
Writing Wrapper S-Functions	8-9
The MEX S-Function Wrapper	8-9
The TLC S-Function Wrapper	8-14
The Inlined Code	8-18
Fully Inlined S-Functions	8-19
Multiport S-Function Example	8-19
Fully Inlined S-Function with the mdlRTW Routine	8-21

S-Function RTWdata for Generating Code with Real-Time Workshop	8-22
The Direct-Index Lookup Table Algorithm	8-23
The Direct-Index Lookup Table Example	8-24

S-Function Callback Methods

9

Callback Method Reference	9-2
mdlCheckParameters	9-3
mdlDerivatives	9-5
mdlGetTimeOfNextVarHit	9-6
mdlInitializeConditions	9-7
mdlInitializeSampleTimes	9-9
mdlInitializeSizes	9-13
mdlOutputs	9-17
mdlProcessParameters	9-18
mdlRTW	9-20
mdlSetDefaultPortComplexSignals	9-21
mdlSetDefaultPortDataTypes	9-22
mdlSetDefaultPortDimensionInfo	9-23
mdlSetInputPortComplexSignal	9-24
mdlSetInputPortDataType	9-25
mdlSetInputPortDimensionInfo	9-26
mdlSetInputPortFrameData	9-28
mdlSetInputPortSampleTime	9-29
mdlSetInputPortWidth	9-31
mdlSetOutputPortComplexSignal	9-32
mdlSetOutputPortDataType	9-33
mdlSetOutputPortDimensionInfo	9-34
mdlSetOutputPortSampleTime	9-36
mdlSetOutputPortWidth	9-37
mdlSetWorkWidths	9-38
mdlStart	9-39
mdlTerminate	9-40
mdlUpdate	9-41
mdlZeroCrossings	9-42

Introduction	10-2
Language Support	10-2
The SimStruct	10-2
SimStruct Macros and Functions Listed by Usage	10-3
Miscellaneous	10-3
Error Handling and Status	10-3
I/O Port	10-4
Dialog Box Parameters	10-6
Run-Time Parameters	10-7
Sample Time	10-8
State and Work Vector	10-9
Simulation Information	10-12
Function Call	10-12
Data Type	10-13
Real-Time Workshop	10-13
Macro Reference	10-15
ssCallExternalModeFcn	10-16
ssCallSystemWithTid	10-17
ssGetAbsTol	10-18
ssGetContStateAddress	10-19
ssGetContStates	10-20
ssGetDataTypeName	10-21
ssGetDataTypeId	10-22
ssGetDataTypeIdFromMxArray	10-23
ssGetDataTypeIdFromMxArray	10-23
ssGetDataTypeIdZero	10-24
ssGetDiscStates	10-25
ssGetDTypeIdFromMxArray	10-26
ssGetDWorkComplexSignal	10-28
ssGetDWorkDataType	10-29
ssGetDWorkName	10-30
ssGetDWorkUsedAsDState	10-31
ssGetDWorkWidth	10-32
ssGetdX	10-33
ssGetErrorStatus	10-34
ssGetInputPortBufferDstPort	10-35

ssGetInputPortConnected	10-36
ssGetInputPortComplexSignal	10-37
ssGetInputPortDataType	10-38
ssGetInputPortDimensionInfo	10-39
ssGetInputPortDimensions	10-40
ssGetInputPortDirectFeedThrough	10-41
ssGetInputPortFrameData	10-42
ssGetInputPortNumDimensions	10-43
ssGetInputPortOffsetTime	10-44
ssGetInputPortOverWritable	10-45
ssGetInputPortRealSignal	10-46
ssGetInputPortRealSignalPtrs	10-47
ssGetInputPortRequiredContiguous	10-48
ssGetInputPortReusable	10-49
ssGetInputPortSampleTime	10-50
ssGetInputPortSampleTimeIndex	10-51
ssGetInputPortSignal	10-52
ssGetInputPortSignalAddress	10-54
ssGetInputPortSignalPtrs	10-55
ssGetInputPortWidth	10-56
ssGetIWork	10-57
ssGetModelName	10-58
ssGetModeVector	10-59
ssGetModeVectorValue	10-60
ssGetNonsampledZCs	10-61
ssGetNumContStates	10-62
ssGetNumDataTypes	10-63
ssGetNumDiscStates	10-64
ssGetNumDWork	10-65
ssGetNumInputPorts	10-66
ssGetNumIWork	10-67
ssGetNumModes	10-68
ssGetNumNonsampledZCs	10-69
ssGetNumOutputPorts	10-70
ssGetNumParameters	10-71
ssGetNumRunTimeParams	10-72
ssGetNumPWork	10-73
ssGetNumRWork	10-74
ssGetNumSampleTimes	10-75
ssGetNumSFcnParams	10-76

ssGetOutputPortBeingMerged	10-77
ssGetOutputPortComplexSignal	10-78
ssGetOutputPortDataType	10-79
ssGetOutputPortDimensions	10-80
ssGetOutputPortFrameData	10-81
ssGetOutputPortNumDimensions	10-82
ssGetOutputPortOffsetTime	10-83
ssGetOutputPortRealSignal	10-84
ssGetOutputPortReusable	10-85
ssGetOutputPortSampleTime	10-86
ssGetOutputPortSignal	10-87
ssGetOutputPortSignalAddress	10-88
ssGetOutputPortWidth	10-89
ssGetPath	10-90
ssGetParentSS	10-91
ssGetPlacementGroup	10-92
ssGetPWork	10-93
ssGetRealDiscStates	10-94
ssGetRootSS	10-95
ssGetRunTimeParamInfo	10-96
ssGetRWork	10-97
ssGetSampleTimeOffset	10-98
ssGetSampleTimePeriod	10-99
ssGetSFcnParam	10-100
ssGetSFcnParamsCount	10-101
ssGetSimMode	10-102
ssGetSolverName	10-103
ssGetStateAbsTol	10-104
ssGetT	10-105
ssGetTNext	10-106
ssGetTaskTime	10-107
ssGetTFinal	10-108
ssGetTStart	10-109
ssIsContinuousTask	10-110
ssGetUserData	10-111
ssIsFirstInitCond	10-112
ssIsMajorTimeStep	10-113
ssIsMinorTimeStep	10-114
ssIsSampleHit	10-115
ssIsSpecialSampleHit	10-116

ssIsVariableStepSolver	10-117
ssPrintf	10-118
ssRegAllTunableParamsAsRunTimeParams	10-119
ssRegisterDataType	10-120
ssSetCallSystemOutput	10-121
ssSetDataTypeSize	10-122
ssSetDataTypeZero	10-123
ssSetDWorkComplexSignal	10-125
ssSetDWorkDataType	10-126
ssSetDWorkName	10-127
ssSetDWorkUsedAsDState	10-128
ssSetDWorkWidth	10-129
ssSetErrorStatus	10-130
ssSetExternalModeFcn	10-131
ssSetInputPortComplexSignal	10-132
ssSetInputPortDataType	10-133
ssSetInputPortDimensionInfo	10-134
ssSetInputPortFrameData	10-136
ssSetInputPortDirectFeedThrough	10-137
ssSetInputPortMatrixDimensions	10-138
ssSetInputPortOffsetTime	10-139
ssSetInputPortOverWritable	10-140
ssSetInputPortReusable	10-141
ssSetInputPortRequiredContiguous	10-143
ssSetInputPortSampleTime	10-144
ssSetInputPortSampleTimeIndex	10-145
ssSetInputPortVectorDimension	10-146
ssSetInputPortWidth	10-147
ssSetModeVectorValue	10-148
ssSetNumContStates	10-149
ssSetNumDiscStates	10-150
ssSetNumDWork	10-151
ssSetNumInputPorts	10-152
ssSetNumIWork	10-153
ssSetNumModes	10-154
ssSetNumNonsampledZCs	10-155
ssSetNumOutputPorts	10-156
ssSetNumPWork	10-157
ssSetNumRunTimeParams	10-158
ssSetNumRWork	10-159

ssSetNumSampleTimes	10-160
ssSetNumSFcnParams	10-161
ssSetOffsetTime	10-162
ssSetOptions	10-163
ssSetOutputPortComplexSignal	10-167
ssSetOutputPortDataType	10-168
ssSetOutputPortDimensionInfo	10-169
ssSetOutputPortFrameData	10-170
ssSetOutputPortMatrixDimensions	10-171
ssSetOutputPortOffsetTime	10-172
ssSetOutputPortReusable	10-173
ssSetOutputPortSampleTime	10-174
ssSetOutputPortVectorDimension	10-175
ssSetOutputPortWidth	10-176
ssSetParameterName	10-177
ssSetParameterTunable	10-178
ssSetPlacementGroup	10-179
ssSetRunTimeParamInfo	10-180
ssSetSampleTime	10-183
ssSetSFcnParamNotTunable	10-184
ssSetSFcnParamTunable	10-185
ssSetSolverNeedsReset	10-186
ssSetStopRequested	10-187
ssSetTNext	10-188
ssSetUserData	10-189
ssSetVectorMode	10-190
ssUpdateAllTunableParamsAsRunTimeParams	10-191
ssUpdateRunTimeParamData	10-192
ssUpdateRunTimeParamInfo	10-193
ssWarning	10-194
ssWriteRTWMxVectParam	10-195
ssWriteRTWMx2dMatParam	10-196
ssWriteRTWParameters	10-197
ssWriteRTWParamSettings	10-201
ssWriteRTWScalarParam	10-205
ssWriteRTWStr	10-206
ssWriteRTWStrParam	10-207
ssWriteRTWStrVectParam	10-208
ssWriteRTWVectParam	10-209
ssWriteRTWWorkVect	10-210

ssWriteRTW2dMatParam	10-211
----------------------------	--------

Overview of S-Functions

Introduction	1-2
What Is an S-Function?	1-2
Using S-Functions in Models	1-2
Passing Parameters to S-Functions	1-3
When to Use an S-Function	1-5
How S-Functions Work	1-5
Implementing S-Functions	1-9
S-Function Concepts	1-11
S-Function Examples	1-16

Introduction

S-functions (system-functions) provide a powerful mechanism for extending the capabilities of Simulink[®]. The introductory sections of this chapter describe what an S-function is and when and why you might use one. This chapter then presents a comprehensive description of how to write your own S-functions.

S-functions allow you to add your own blocks to Simulink models. You can create your blocks in MATLAB[®], C, C++, Fortran, or Ada. By following a set of simple rules, you can implement your algorithms in an S-function. After you have written your S-function and placed its name in an S-Function block (available in the Functions & Tables block library), you can customize the user interface by using masking.

S-functions can be used with the Real-Time Workshop. You can also customize the code generated by the Real Time Workshop[®] for S-functions by writing a Target Language Compiler[™] (TLC) file. See the Target Language Compiler Reference Guide and the Real-Time Workshop User's Guide for more information.

What Is an S-Function?

An S-function is a computer language description of a Simulink block. S-functions can be written in MATLAB, C, C++, Ada, or Fortran. C, C++, Ada, and Fortran S-functions are compiled as MEX-files using the mex utility described in the Application Program Interface Guide. As with other MEX-files, they are dynamically linked into MATLAB when needed.

S-functions use a special calling syntax that enables you to interact with Simulink's equation solvers. This interaction is very similar to the interaction that takes place between the solvers and built-in Simulink blocks. The form of an S-function is very general and can accommodate continuous, discrete, and hybrid systems.

Using S-Functions in Models

To incorporate an S-function into an Simulink model, drag an S-Function block from Simulink's Functions & Tables block library into the model. Then specify the name of the S-function in the S-function name field of the S-Function block's dialog box as illustrated in the figure below.

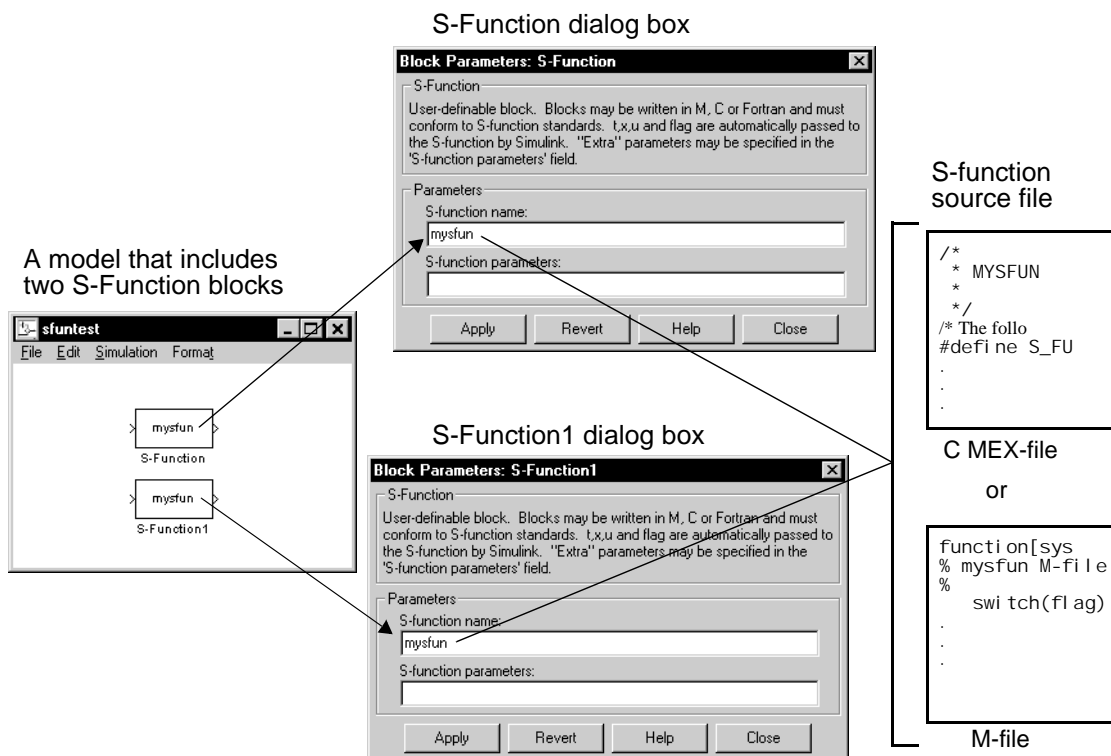


Figure 1-1: The Relationship Between an S-Function Block, Its Dialog Box, and the Source File That Defines the Block's Behavior

In this example, the model contains two instances of an S-Function block. Both blocks reference the same source file (mysfun, which can be either a C MEX-file or an M-file). If both a C MEX-file and an M-file exist with the same name, the C MEX-file takes precedence and is the file that the S-function uses.

Passing Parameters to S-Functions

The S-function block's S-function parameters field allows you to specify parameter values to be passed to the corresponding S-function. To use this field, you must know what parameters the S-function requires and the order in which the function requires them. (If you do not know, consult the S-function's

specify that the block's sample time is inherited. A simple example of this is a Gain block that inherits its sample time from the block driving it.

A block can inherit its sample time from:

- The driving block
- The destination block
- The fastest sample time in the system

To set a block's sample time as inherited, use -1 in M-file S-functions and `INHHERITED_SAMPLE_TIME` in C S-functions as the sample time. For more information on the propagation of sample times, see "Sample Time Colors" in Using Simulink.

S-functions can be either single or multirate; a multirate S-function has multiple sample times.

Sample times are specified in pairs in this format: [`sample_time`, `offset_time`]. The valid sample time pairs are

```
[CONTINUOUS_SAMPLE_TIME, 0.0]
[CONTINUOUS_SAMPLE_TIME, FIXED_IN_MINOR_STEP_OFFSET]
[discrete_sample_time_period, offset]
[VARIABLE_SAMPLE_TIME, 0.0]
```

where

```
CONTINUOUS_SAMPLE_TIME = 0.0
FIXED_IN_MINOR_STEP_OFFSET = 1.0
VARIABLE_SAMPLE_TIME = -2.0
```

and the italics indicate a real value is required.

Alternatively, you can specify that the sample time is inherited from the driving block. In this case the S-function can have only one sample time pair

```
[INHHERITED_SAMPLE_TIME, 0.0]
```

or

```
[INHHERITED_SAMPLE_TIME, FIXED_IN_MINOR_STEP_OFFSET]
```

where

```
INHHERITED_SAMPLE_TIME = -1.0
```

The following guidelines may help you specify sample times:

- A continuous S-function that changes during minor integration steps should register the `[CONTINUOUS_SAMPLE_TIME, 0.0]` sample time.
- A continuous S-function that does not change during minor integration steps should register the `[CONTINUOUS_SAMPLE_TIME, FIXED_IN_MINOR_STEP_OFFSET]` sample time.
- A discrete S-function that changes at a specified rate should register the discrete sample time pair, `[discrete_sample_time_period, offset]`, where

$$discrete_sample_period > 0.0$$

and

$$0.0 \leq offset < discrete_sample_period$$

- A discrete S-function that changes at a variable rate should register the variable step discrete sample time. `[VARIABLE_SAMPLE_TIME, 0.0]`

The mdl `GetTimeOfNextVarHit` routine is called to get the time of the next sample hit for the variable step discrete task.

If your S-function has no intrinsic sample time, then you must indicate that your sample time is inherited. There are two cases:

- An S-function that changes as its input changes, even during minor integration steps, should register the `[INHERITED_SAMPLE_TIME, 0.0]` sample time.
- An S-function that changes as its input changes, but doesn't change during minor integration steps (that is, remains fixed during minor time steps), should register the `[INHERITED_SAMPLE_TIME, FIXED_IN_MINOR_STEP_OFFSET]` sample time.

The Scope block is a good example of this type of block. This block should run at the rate, either continuous or discrete, of its driving block, but should never run in minor step. If it did, the scope display would show the intermediate computations of the solver rather than the final result at each time point.

The following table describes the values that flag can assume and lists the corresponding S-function method for each value.

Table 2-1: Flag Argument

Flag	S-Function Routine	Description
0	mdlInitializes	Defines basic S-Function block characteristics, including sample times, initial conditions of continuous and discrete states, and the sizes array.
1	mdlDerivatives	Calculates the derivatives of the continuous state variables.
2	mdlUpdate	Updates discrete states, sample times, and major time step requirements.
3	mdlOutputs	Calculates the outputs of the S-function.
4	mdlGetTimeOfNextVarHit	Calculates the time of the next hit in absolute time. This routine is used only when you specify a variable discrete-time sample time in mdlInitializes.
9	mdlTerminate	Performs any necessary end of simulation tasks.

S-Function Outputs

An M-file returns an output vector containing the following elements:

- `sys`, a generic return argument. The values returned depend on the flag value. For example, for `flag = 3`, `sys` contains the S-function outputs.
- `x0`, the initial state values (an empty vector if there are no states in the system). `x0` is ignored, except when `flag = 0`.

- `str`, reserved for future use. M-file S-functions must set this to the empty matrix, `[]`.
- `ts`, a two column matrix containing the sample times and offsets of the block. Continuous systems have their sample time set to zero. The hybrid example, which starts on page 2-13, demonstrates an S-function with multiple sample times.

Sample times should be declared in ascending order. For example, if you want your S-function to execute at `[0 0.1 0.25 0.75 1.0 1.1 1.25, etc.]`, set `ts` equal to a two row matrix.

```
ts = [.25 0; 1.0 .1];
```

Defining S-Function Block Characteristics

For Simulink to recognize an M-file S-function, you must provide it with specific information about the S-function. This information includes the number of inputs, outputs, states, and other block characteristics.

To give Simulink this information, call the `si_msi zes` function at the beginning of `mdl_initializeSi zes`.

```
si zes = si_msi zes;
```

This function returns an uninitialized `si zes` structure. You must load the `si zes` structure with information about the S-function. The table below lists the `si zes` structure fields and describes the information contained in each field.

Table 2-2: Fields in the `si zes` Structure

Field Name	Description
<code>si zes.NumContStates</code>	Number of continuous states
<code>si zes.NumDi scStates</code>	Number of discrete states
<code>si zes.NumOutputs</code>	Number of outputs
<code>si zes.NumI nputs</code>	Number of inputs
<code>si zes.Di rFeedthrough</code>	Flag for direct feedthrough
<code>si zes.NumSampl eTi mes</code>	Number of sample times

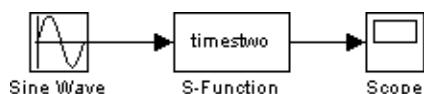
After you initialize the `si zes` structure, call `si msi zes` again.

```
sys = si msi zes(si zes);
```

This passes the information in the `si zes` structure to `sys`, a vector that holds the information for use by Simulink.

A Simple M-File S-Function Example

The easiest way to understand how S-functions work is to look at a simple example. This block takes an input scalar signal, doubles it, and plots it to a scope.



The M-file code that contains the S-function is modeled on an S-function template called `sfuntmpl.m`, which is included with Simulink. By using this template, you can create an M-file S-function that is very close in appearance to a C MEX S-function. This is useful because it makes a transition from an M-file to a C MEX-file much easier.

Below is the M-file code for the `timestwo.m` S-function.

```
function [sys, x0, str, ts] = timestwo(t, x, u, flag)
% Dispatch the flag. The switch function controls the calls to
% S-function routines at each simulation stage.
switch flag,

    case 0
        [sys, x0, str, ts] = mdlInitializeSizes; % Initialization

    case 3
        sys = mdlOutputs(t, x, u); % Calculate outputs

    case { 1, 2, 4, 9 }
        sys = []; % Unused flags

    otherwise
        error(['Unhandled flag = ', num2str(flag)]); % Error handling
end;
```

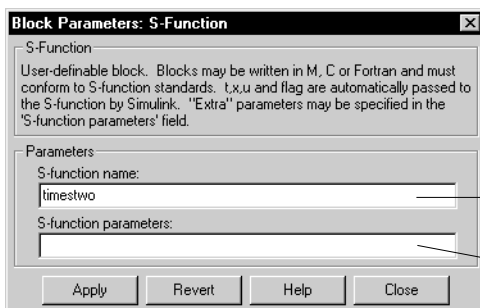
```
% End of function timestwo.
```

Below are the S-function subroutines that timestwo.m calls.

```
%=====
% Function mdlInitializeSizes initializes the states, sample
% times, state ordering strings (str), and sizes structure.
%=====
function [sys,x0,str,ts] = mdlInitializeSizes
% Call function simsizes to create the sizes structure.
sizes = simsizes;
% Load the sizes structure with the initialization information.
sizes.NumContStates= 0;
sizes.NumDiscStates= 0;
sizes.NumOutputs= 1;
sizes.NumInputs= 1;
sizes.DirFeedthrough=1;
sizes.NumSampleTimes=1;
% Load the sys vector with the sizes information.
sys = simsizes(sizes);
%
x0 = []; % No continuous states
%
str = []; % No state ordering
%
ts = [-1 0]; % Inherited sample time
% End of mdlInitializeSizes.
%=====
% Function mdlOutputs performs the calculations.
%=====
function sys = mdlOutputs(t,x,u)
sys = 2*u;

% End of mdlOutputs.
```

To test this S-function in Simulink, connect a sine wave generator to the input of an S-Function block. Connect the output of the S-Function block to a Scope. Double-click on the S-Function block to open the dialog box.



Enter the function name here. In this example, type timestwo.

If you have additional parameters to pass to the block, enter their names here, separating them with commas. In this example, there are no additional parameters.

You can now run this simulation.

Examples of M-File S-Functions

The simple example discussed above (ti mestwo) has no states. Most S-Function blocks require the handling of states, whether continuous or discrete. The sections that follow discuss four common types of systems you can model in Simulink using S-functions:

- Continuous
- Discrete
- Hybrid
- Variable-step

All examples are based on the M-file S-function template found in `sfuntmpl.m`.

Example - Continuous State S-Function

Simulink includes a function called `csfunc.m`, which is an example of a continuous state system modeled in an S-function. Here is the code for the M-file S-function.

```
function [sys, x0, str, ts] = csfunc(t, x, u, flag)
% CSFUNC An example M-file S-function for defining a system of
% continuous state equations:
%      x' = Ax + Bu
%      y = Cx + Du
%
% Generate a continuous linear system:
A=[-0.09  -0.01
    1      0];
B=[ 1  -7
    0  -2];
C=[ 0  2
    1 -5];
D=[-3  0
    1  0];
%
% Dispatch the flag.
%
switch flag,
```

```

case 0
    [sys, x0, str, ts]=mdlInitializeSizes(A, B, C, D); % Initialization

case 1
    sys = mdlDerivatives(t, x, u, A, B, C, D); % Calculate derivatives

case 3
    sys = mdlOutputs(t, x, u, A, B, C, D); % Calculate outputs

case { 2, 4, 9 } % Unused flags
    sys = [];
otherwise
    error(['Unhandled flag = ', num2str(flag)]); % Error handling
end
% End of csfunc.
%=====
% mdlInitializeSizes
% Return the sizes, initial conditions, and sample times for the
% S-function.
%=====
%
function [sys, x0, str, ts] = mdlInitializeSizes(A, B, C, D)
%
% Call simsizes for a sizes structure, fill it in and convert it
% to a sizes array.
%
sizes = simsizes;
sizes.NumContStates = 2;
sizes.NumDiscStates = 0;
sizes.NumOutputs = 2;
sizes.NumInputs = 2;
sizes.DirFeedthrough = 1; % Matrix D is nonempty.
sizes.NumSampleTimes = 1;
sys = simsizes(sizes);
%
% Initialize the initial conditions.
%
x0 = zeros(2, 1);
%
% str is an empty matrix.

```

```

%
str = [];
%
% Initialize the array of sample times; in this example the sample
% time is continuous, so set ts to 0 and its offset to 0.
%
ts = [0 0];
% End of mdlInitializeSizes.
%
%=====
% mdl Derivatives
% Return the derivatives for the continuous states.
%=====
function sys = mdlDerivatives(t, x, u, A, B, C, D)
sys = A*x + B*u;
% End of mdlDerivatives.
%
%=====
% mdl Outputs
% Return the block outputs.
%=====
function sys = mdlOutputs(t, x, u, A, B, C, D)
sys = C*x + D*u;
% End of mdlOutputs.

```

The above example conforms to the simulation stages discussed earlier in this chapter. Unlike `ti_mestwo.m`, this example invokes `mdlDerivatives` to calculate the derivatives of the continuous state variables when `flag = 1`. The system state equations are of the form

$$\begin{aligned}x' &= Ax + Bu \\y &= Cx + Du\end{aligned}$$

so that very general sets of continuous differential equations can be modeled using `csfunc.m`. Note that `csfunc.m` is similar to the built-in State-Space block. This S-function can be used as a starting point for a block that models a state-space system with time-varying coefficients.

Each time the `mdlDerivatives` routine is called it must explicitly set the value of all derivatives. The derivative vector does not maintain the values from the

last call to this routine. The memory allocated to the derivative vector changes during execution.

Example - Discrete State S-Function

Simulink includes a function called `dsfunc.m`, which is an example of a discrete state system modeled in an S-function. This function is similar to `csfunc.m`, the continuous state S-function example. The only difference is that `mdlUpdate` is called instead of `mdlDerivative`. `mdlUpdate` updates the discrete states when the `flag = 2`. Note that for a single-rate discrete S-function, Simulink calls the `mdlUpdate`, `mdlOutput`, and `mdlGetTimeOfNextVarHit` (if needed) routines only on sample hits. Here is the code for the M-file S-function.

```
function [sys, x0, str, ts] = dsfunc(t, x, u, flag)
% An example M-file S-function for defining a discrete system.
% This S-function implements discrete equations in this form:
%     x(n+1) = Ax(n) + Bu(n)
%     y(n)   = Cx(n) + Du(n)
%
% Generate a discrete linear system:
A=[-1.3839 -0.5097
    1.0000      0];
B=[-2.5559      0
    0      4.2382];
C=[ 0      2.0761
    0      7.7891];
D=[ -0.8141  -2.9334
    1.2426      0];

switch flag,
case 0
    sys = mdlInitializeSizes(A, B, C, D); % Initialization

case 2
    sys = mdlUpdate(t, x, u, A, B, C, D); % Update discrete states

case 3
    sys = mdlOutputs(t, x, u, A, B, C, D); % Calculate outputs

case {1, 4, 9} % Unused flags
    sys = [];
```

```

        otherwise
            error(['unhandled flag = ', num2str(flag)]); % Error handling
        end
    % End of dsfunc.

%=====
% Initialization
%=====

function [sys, x0, str, ts] = mdlInitializeSizes(A, B, C, D)

% Call simsizes for a sizes structure, fill it in, and convert it
% to a sizes array.

sizes = simsizes;
sizes.NumContStates = 0;
sizes.NumDiscStates = 2;
sizes.NumOutputs = 2;
sizes.NumInputs = 2;
sizes.DirFeedthrough = 1; % Matrix D is non-empty.
sizes.NumSampleTimes = 1;
sys = simsizes(sizes);
x0 = ones(2, 1); % Initialize the discrete states.
str = []; % Set str to an empty matrix.
ts = [1 0]; % sample time: [period, offset]
% End of mdlInitializeSizes.

%=====
% Update the discrete states
%=====

function sys = mdlUpdates(t, x, u, A, B, C, D)
sys = A*x + B*u;
% End of mdlUpdate.

%=====
% Calculate outputs
%=====

function sys = mdlOutputs(t, x, u, A, B, C, D)
sys = C*x + D*u;

```

```
% End of mdl Outputs.
```

The above example conforms to the simulation stages discussed earlier in chapter 1. The system discrete state equations are of the form

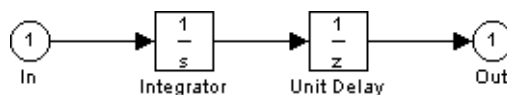
$$\begin{aligned}x(n+1) &= Ax(n) + Bu(n) \\ y(n) &= Cx(n) + Du(n)\end{aligned}$$

so that very general sets of difference equations can be modeled using `dsfunc.m`. This is similar to the built-in Discrete State-Space block. You can use `dsfunc.m` as a starting point for modeling discrete state-space systems with time-varying coefficients.

Example - Hybrid System S-Functions

Simulink includes a function called `mixedm.m`, which is an example of a hybrid system (a combination of continuous and discrete states) modeled in an S-function. Handling hybrid systems is fairly straightforward; the `flag` parameter forces the calls to the correct S-function subroutine for the continuous and discrete parts of the system. One subtlety of hybrid S-functions (or any multirate S-function) is that Simulink calls the `mdlUpdate`, `mdlOutput`, and `mdlGetTimeOfNextVarHit` routines at all sample times. This means that in these routines you must test to determine which sample hit is being processed and only perform updates that correspond to that sample hit.

`mixed.m` models a continuous Integrator followed by a discrete Unit Delay. In Simulink block diagram form, the model looks like this.



Here is the code for the M-file S-function.

```
function [sys, x0, str, ts] = mixedm(t, x, u, flag)
% A hybrid system example that implements a hybrid system
% consisting of a continuous integrator (1/s) in series with a
% unit delay (1/z).
%
% Set the sampling period and offset for unit delay.
dperiod = 1;
doffset = 0;
```

```

switch flag,

    case 0          % Initialization
        [sys, x0, str, ts] = mdlInitializeSizes(dperiod, doffset);

    case 1
        sys = mdlDerivatives(t, x, u); % Calculate derivatives

    case 2
        sys = mdlUpdate(t, x, u, dperiod, doffset); % Update disc states

    case 3
        sys = mdlOutputs(t, x, u, doffset, dperiod); % Calculate outputs
    case {4, 9}
        sys = []; % Unused flags

    otherwise
        error(['unhandled flag = ', num2str(flag)]); % Error handling
end
% End of mixedm.
%
%=====
% mdlInitializeSizes
% Return the sizes, initial conditions, and sample times for the
% S-function.
%=====
function [sys, x0, str, ts] = mdlInitializeSizes(dperiod, doffset)
sizes = sizes;
sizes.NumContStates = 1;
sizes.NumDiscStates = 1;
sizes.NumOutputs = 1;
sizes.NumInputs = 1;
sizes.DirFeedthrough = 0;
sizes.NumSampleTimes = 2;
sys = sizes(sizes);
x0 = ones(2, 1);
str = [];
ts = [0, 0 % sample time
      dperiod, doffset];
% End of mdlInitializeSizes.

```

```

%
%=====
% mdl Derivatives
% Compute derivatives for continuous states.
%=====
%
function sys = mdlDerivatives(t, x, u)
sys = u;
% end of mdlDerivatives.
%
%=====
% mdl Update
% Handle discrete state updates, sample time hits, and major time
% step requirements.
%=====
%
function sys = mdlUpdate(t, x, u, dperiod, doffset)
% Next discrete state is output of the integrator.
% Return next discrete state if we have a sample hit within a
% tolerance of 1e-8. If we don't have a sample hit, return [] to
% indicate that the discrete state shouldn't change.
%
if abs(round((t-doffset)/dperiod)-(t-doffset)/dperiod) < 1e-8
    sys = x(1); % mdlUpdate is "latching" the value of the
                % continuous state, x(1), thus introducing a delay.
else
    sys = []; % This is not a sample hit, so return an empty
end          % matrix to indicate that the states have not
            % changed.
% End of mdlUpdate.
%
%=====
% mdl Outputs
% Return the output vector for the S-function.
%=====
%
function sys = mdlOutputs(t, x, u, doffset, dperiod)
% Return output of the unit delay if we have a
% sample hit within a tolerance of 1e-8. If we

```

```

% don't have a sample hit then return [] indicating
% that the output shouldn't change.
%
if abs(round((t-doffset)/dperiod)-(t-doffset)/dperiod) < 1e-8
    sys = x(2);

else
    sys = []; % This is not a sample hit, so return an empty
end          % matrix to indicate that the output has not changed

% End of mdlOutputs.

```

Example - Variable Sample Time S-Functions

This M-file is an example of an S-function that uses a variable sample time. This example, in an M-file called `vsfunc.m`, calls `mdlGetTimeOfNextVarHit` when `flag = 4`. Because the calculation of a next sample time depends on the input `u`, this block has direct feedthrough. Generally, all blocks that use the input to calculate the next sample time (`flag = 4`) require direct feedthrough. Here is the code for the M-file S-function.

```

function [sys,x0,str,ts] = vsfunc(t,x,u,flag)
% This example S-function illustrates how to create a variable
% step block in Simulink. This block implements a variable step
% delay in which the first input is delayed by an amount of time
% determined by the second input.
%
%      dt      = u(2)
%      y(t+dt) = u(t)
%
switch flag,

case 0
    [sys,x0,str,ts] = mdlInitializeSizes; % Initialization

case 2
    sys = mdlUpdate(t,x,u); % Update Discrete states

case 3
    sys = mdlOutputs(t,x,u); % Calculate outputs

```

```

case 4
    sys = mdlGetTimeOfNextVarHit(t, x, u); % Get next sample time

case { 1, 9 }
    sys = []; % Unused flags
otherwise
    error(['Unhandled flag = ', num2str(flag)]); % Error handling
end
% End of vsfunc.
%=====
% mdlInitializeSizes
% Return the sizes, initial conditions, and sample times for the
% S-function.
%=====
%
function [sys, x0, str, ts] = mdlInitializeSizes
%
% Call simsizes for a sizes structure, fill it in and convert it
% to a sizes array.
%
sizes = simsizes;
sizes.NumContStates = 0;
sizes.NumDiscStates = 1;
sizes.NumOutputs = 1;
sizes.NumInputs = 2;
sizes.DirFeedthrough = 1; % flag=4 requires direct feedthrough
                           % if input u is involved in
                           % calculating the next sample time
                           % hit.

sizes.NumSampleTimes = 1;
sys = simsizes(sizes);
%
% Initialize the initial conditions.
%
x0 = [0];
%
% Set str to an empty matrix.
%
str = [];
%
```

```

% Initialize the array of sample times.
%
ts = [-2 0];      % variable sample time
% End of mdlInitializeSizes.
%
%=====
% mdl Update
% Handle discrete state updates, sample time hits, and major time
% step requirements.
%=====
%
function sys = mdlUpdate(t, x, u)
sys = u(1);
% End of mdlUpdate.
%
%=====
% mdl Outputs
% Return the block outputs.
%=====
%
function sys = mdlOutputs(t, x, u)
sys = x(1);
% end mdlOutputs
%
%=====
% mdl GetTimeOfNextVarHit
% Return the time of the next hit for this block. Note that the
% result is absolute time.
%=====
%
function sys = mdlGetTimeOfNextVarHit(t, x, u)
sys = t + u(2);
% End of mdlGetTimeOfNextVarHit.

```

mdlGetTimeOfNextVarHit returns the “time of the next hit,” the time in the simulation when vsfunc is next called. This means that there is no output from this S-function until the time of the next hit. In vsfunc, the time of the next hit is set to $t + u(2)$, which means that the second input, $u(2)$, sets the time when the next call to vsfunc occurs.

Processing S-Function Parameters

When invoking an M-file S-function, Simulink always passes the standard block parameters, `t`, `x`, `u`, and `flag`, to the S-function as function arguments. Simulink can pass additional, block-specific parameters specified by the user to the S-function. The user specifies the parameters in the S-function parameters field of the S-function's block parameter dialog (see "Passing Parameters to S-Functions" on page 1-3). If the block dialog specifies additional parameters, Simulink passes the parameters to the S-function as additional function arguments. The additional arguments follow the standard arguments in the S-function argument list in the order in which the corresponding parameters appear in the block dialog. You can use this block-specific S-function parameter capability to allow the same S-function to implement various processing options. See the `limintm.m` example in the `toolbox/simulink/blocks` directory for an example of an S-function that uses block-specific parameters in this way.

Writing S-Functions in C

Introduction	3-2
Example of a Basic C MEX S-Function	3-3
Templates for C S-Functions	3-9
S-Function Source File Requirements	3-9
The SimStruct	3-11
Compiling C S-Functions	3-12
How Simulink Interacts with C S-Functions	3-13
Process View	3-13
Data View	3-17
Writing Callback Methods	3-21
Converting Level 1 C MEX S-Functions to Level 2	3-22
Obsolete Macros	3-24

Introduction

A C MEX-file that defines an S-Function block must provide information about the model to Simulink during the simulation. As the simulation proceeds, Simulink, the ODE solver, and the MEX-file interact to perform specific tasks. These tasks include defining initial conditions and block characteristics, and computing derivatives, discrete states, and outputs.

As with M-file S-functions, Simulink interacts with a C MEX-file S-function by invoking callback methods that the S-function implements. Each method performs a predefined task, such as computing block outputs, required to simulate the block whose functionality the S-function defines. Simulink defines in a general way the task of each callback. The S-function is free to perform the task according to the functionality it implements. For example, Simulink specifies that the S-function's `mdlOutput` method must compute that block's outputs at the current simulation time. It does not specify what those outputs must be. This callback-based API allows you to create S-functions, and hence custom blocks, of any desired functionality.

The set of callback methods, hence functionality, that C MEX-files can implement is much larger than that available for M-file S-functions. See Chapter 9, "S-Function Callback Methods" for descriptions of the callback methods that a C MEX-file S-function can implement. Unlike M-file S-functions, C MEX-files can access and modify the data structure that Simulink uses internally to store information about the S-function. The ability to implement a broader set of callback methods and to access internal data structures allows C-MEX files to implement a wider set of block features, such as the ability to handle matrix signals and multiple data types.

C MEX-file S-functions are required to implement only a small subset of the callback methods that Simulink defines. If your block does not implement a particular feature, such as matrix signals, you are free to omit the callback methods required to implement a feature. This allows you to create simple blocks very quickly.

The general format of a C MEX S-function is shown below.

```
#define S_FUNCTION_NAME your_sfunction_name_here
#define S_FUNCTION_LEVEL 2
#include "simstruc.h"

static void mdlInitializeSizes(SimStruct *S)
```

```

{
}

<additional S-function routines/code>

static void mdlTerminate(SimStruct *S)
{
}
#ifdef MATLAB_MEX_FILE    /* Is this file being compiled as a
                           MEX-file? */
#include "simulink.c"      /* MEX-file interface mechanism */
#else
#include "cg_sfun.h"       /* Code generation registration
                           function */
#endif

```

`mdlInitializeSizes` is the first routine Simulink calls when interacting with the S-function. Simulink subsequently invokes other S-function methods (all starting with `mdl`). At the end of a simulation, Simulink calls `mdlTerminate`.

Note Unlike M-file S-functions, C MEX S-function methods do not each have a flag parameter. This is because Simulink calls each S-function method directly at the appropriate time during the simulation.

Example of a Basic C MEX S-Function

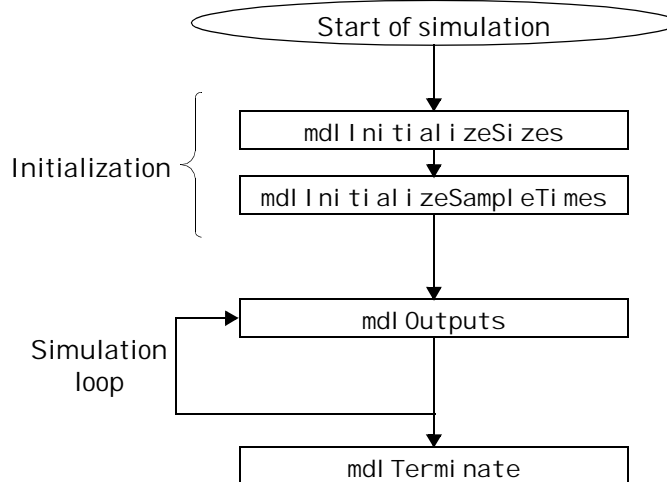
This section presents an example of a C MEX S-function that you can use as a model for creating simple C S-functions. The example is the `timestwo` S-function example that comes with Simulink (see `matlabroot/simulink/src/timestwo.c`). This S-function outputs twice its input.

The following model uses the `timestwo` S-function to double the amplitude of a sine wave and plot it in a scope.



The block dialog for the S-function specifies `ti mestwo` as the S-function name; the parameters field is empty.

The `ti mestwo` S-function contains the S-function callback methods shown in this figure.



The contents of `timestwo.c` are shown below.

```

#define S_FUNCTION_NAME  timestwo
#define S_FUNCTION_LEVEL 2

#include "simstruc.h"

static void mdlInitializeSizes(SimStruct *S)
{
    ssSetNumSFcnParams(S, 0);
    if (ssGetNumSFcnParams(S) != ssGetSFcnParamsCount(S)) {
        return; /* Parameter mismatch will be reported by Simulink */
    }

    if (!ssSetNumInputPorts(S, 1)) return;
    ssSetInputPortWidth(S, 0, DYNAMICALLY_SIZED);
    ssSetInputPortDirectFeedThrough(S, 0, 1);

    if (!ssSetNumOutputPorts(S, 1)) return;
    ssSetOutputPortWidth(S, 0, DYNAMICALLY_SIZED);

    ssSetNumSampleTimes(S, 1);

    /* Take care when specifying exception free code - see sfuntmpl.doc */
    ssSetOptions(S, SS_OPTION_EXCEPTION_FREE_CODE);
}

static void mdlInitializeSampleTimes(SimStruct *S)
{
    ssSetSampleTime(S, 0, INHERITED_SAMPLE_TIME);
    ssSetOffsetTime(S, 0, 0.0);
}

static void mdlOutputs(SimStruct *S, int_T tid)
{
    int_T      i;
    InputRealPtrsType uPtrs = ssGetInputPortRealSignalPtrs(S, 0);
    real_T      *y          = ssGetOutputPortRealSignal(S, 0);
    int_T      width = ssGetOutputPortWidth(S, 0);

    for (i=0; i<width; i++) {
        *y++ = 2.0 *(*uPtrs[i]);
    }
}

static void mdlTerminate(SimStruct *S){}

#ifdef MATLAB_MEX_FILE /* Is this file being compiled as a MEX-file? */
#include "simulink.c" /* MEX-file interface mechanism */
#else
#include "cg_sfun.h" /* Code generation registration function */
#endif

```

This example has three parts:

- Defines and includes
- Callback implementations
- Simulink (or Real-Time Workshop) interface

The following sections explain each of these parts.

Defines and Includes

The example starts with the following defines.

```
#define S_FUNCTION_NAME  timstwo
#define S_FUNCTION_LEVEL 2
```

The first specifies the name of the S-function (`timstwo`). The second specifies that the S-function is in the level 2 format (for more information about level 1 and level 2 S-functions, see “Converting Level 1 C MEX S-Functions to Level 2” on page 3-22).

After defining these two items, the example includes `simstruc.h`, which is a header file that gives access to the `SimStruct` data structure and the MATLAB Application Program Interface (API) functions.

```
#define S_FUNCTION_NAME  timstwo
#define S_FUNCTION_LEVEL 2
#include "simstruc.h"
```

The `simstruc.h` file defines a data structure, called the `SimStruct`, that Simulink uses to maintain information about the S-function. The `simstruc.h` file also defines macros that enable your MEX-file to set values in and get values (such as the input and output signal to the block) from the `SimStruct` (see Chapter 10, “SimStruct Functions”).

Callback Implementations

The next part of the `timstwo` S-function contains implementations of callback methods required by Simulink.

`mdlInitializeSizes`. Simulink calls `mdlInitializeSizes` to inquire about the number of input and output ports, sizes of the ports, and any other objects (such as the number of states) needed by the S-function.

The `ti mestwo` implementation of `mdl I ni ti al i zeSi zes` specifies the following size information:

- Zero parameters
This means that the S-function parameters field of the S-functions's dialog box must be empty. If it contains any parameters, Simulink will report a parameter mismatch.
- One input port and one output port
The widths of the input and output ports are dynamically sized. This tells Simulink to multiply each element of the input signal to the S-function by two and to place the result in the output signal. Note that the default handling for dynamically sized S-functions for this case (one input and one output) is that the input and output widths are equal.
- One sample time
The `ti mestwo` example specifies the actual value of the sample time in the `mdl I ni ti al i zeSampl eTi mes` routine.
- The code is exception free.

Specifying exception free code speeds up execution of your S-function. Care must be taken when specifying this option. In general, if your S-function isn't interacting with MATLAB, it is safe to specify this option. For more details, see "How Simulink Interacts with C S-Functions" on page 3–13.

`mdlInitializeSampleTimes`. Simulink calls `mdl I ni ti al i zeSampl eTi mes` to set the sample time(s) of the S-function. A `ti mestwo` block executes whenever the driving block executes. Therefore, it has a single inherited sample time, `SAMPLE_TI ME_I NHERI TED`.

`mdlOutputs`. Simulink calls `mdl Outputs` at each time step to calculate a block's outputs. The `ti mestwo` implementation of `mdl Outputs` takes the input, multiplies it by two, and writes the answer to the output.

The `ti mestwo` `mdl Outputs` method uses a `Si mStruct` macro,

```
InputReal PtrsType uPtrs = ssGetInputPortReal Signal Ptrs(S, 0);
```

to access the input signal. The macro returns a vector of pointers, which must be accessed using

```
*uPtrs[i ]
```

For more details, see “Data View” on page 3–17.

The `ti_mestwo mdlOutputs` method uses the macro

```
real_T *y = ssGetOutputPortRealSignal(S, 0);
```

To access the output signal. This macro returns a pointer to an array containing the block’s outputs.

The S-function uses

```
int_T width = ssGetOutputPortWidth(S, 0);
```

to get the width of the signal passing through the block. Finally the S-function loops over the inputs to compute the outputs.

`mdlTerminate`. Perform tasks at end of simulation. This is a mandatory S-function routine. However, the `ti_mestwo` S-function doesn’t need to perform any termination actions, so this routine is empty.

Simulink/Real-Time Workshop Interface

At the end of the S-function, specify code that attaches this example to either Simulink or the Real-Time Workshop.

```
#ifndef MATLAB_MEX_FILE
#include "simulink.c"
#else
#include "cg_sfuns.h"
#endif
```

Building the Timestwo Example

To incorporate this S-function into Simulink, type

```
mex timestwo.c
```

at the command line. The `mex` command compiles and links the `ti_mestwo.c` file to create a dynamically loadable executable for Simulink’s use.

The resulting executable is referred to as a MEX S-function, where MEX stands for “MATLAB EXecutable.” The MEX-file extension varies from platform to platform. For example, in Microsoft Windows, the MEX-file extension is `.dll`.

Templates for C S-Functions

Simulink provides skeleton implementations of C MEX S-functions, called templates, intended to serve as starting points for creating your own S-functions. The templates contain skeleton implementations of callback methods with comments that explain their use. The template file, `sfuntmpl_basi.c.c`, which can be found in the directory `simulink/src` below the MATLAB root directory, contains commonly used S-function routines. A template containing all available routines (as well as more comments) can be found in `sfuntmpl_doc.c` in the same directory.

Note We recommend that you use the C MEX-file template when developing MEX S-functions.

S-Function Source File Requirements

This section describes requirements that every S-function source file must meet to compile correctly. The S-function templates meet these requirements.

Statements Required at the Top of S-Functions

For S-functions to operate properly, each source module of your S-function that accesses the `SimStruct` must contain the following sequence of defines and include

```
#define S_FUNCTION_NAME your_sfuction_name_here
#define SFUNCTION_LEVEL 2
#include "simstruc.h"
```

Where *your_sfuction_name_here* is the name of your S-function (i.e., what you enter in the Simulink S-Function block dialog). These statements give you access to the `SimStruct` data structure that contains pointers to the data used by the simulation. The included code also defines the macros used to store and retrieve data in the `SimStruct`, described in detail in “Converting Level 1 C MEX S-Functions to Level 2” on page 3–22. In addition, the code specifies that you are using the level 2 format of S-functions.

Note All S-functions from Simulink 1.3 through 2.1 are considered to be level 1 S-functions. They are compatible with Simulink 3.0, but we recommend that you write new S-functions in the level 2 format.

The following headers are included by `matlabroot/simulink/include/simstruc.h` when compiling as a MEX-file.

Table 3-1: Header Files Included by Simstruc.h When Compiling as a MEX-File

Header File	Description
<code>matlabroot/extern/include/tmwtypes.h</code>	General data types, e.g., <code>real_T</code>
<code>matlabroot/extern/include/mex.h</code>	MATLAB MEX-file API routines
<code>matlabroot/extern/include/matrix.h</code>	MATLAB MEX-file API routines

When compiling your S-function for use with the Real-Time Workshop, `simstruc.h` includes the following.

Table 3-2: Header Files Included by Simstruc.h When Used by the Real-Time Workshop

Header File	Description
<code>matlabroot/extern/include/tmwtypes.h</code>	General types, e.g. <code>real_T</code>
<code>matlabroot/rtw/c/libsrc/rt_matrix.h</code>	Macros for MATLAB API routines

Statements Required at the Bottom of S-Functions

Include this trailer code at the end of your C MEX S-function main module only.

```
#ifndef MATLAB_MEX_FILE /* Is this being compiled as MEX-file? */
#include "simulink.c" /* MEX-file interface mechanism */
```

```
#el se  
#i ncl ude "cg_sf un. h"      /* Code generati on regi strati on func */  
#endi f
```

These statements select the appropriate code for your particular application:

- `si mul i nk. c` is included if the file is being compiled into a MEX-file.
- `cg_sf un. h` is included if the file is being used in conjunction with the Real-Time Workshop to produce a stand-alone or real-time executable.

Note This trailer code must not be in the body of any S-function routine.

The SimStruct

The file *matlabroot/simulink/include/simstruct.h* is a C language header file that defines the Simulink data structure and the `SimStruct` access macros. It encapsulates all the data relating to the model or S-function, including block parameters and outputs.

There is one `SimStruct` data structure allocated for the Simulink model. Each S-function in the model has its own `SimStruct` associated with it. The organization of these `SimStruct`s is much like a directory tree. The `SimStruct` associated with the model is the root `SimStruct`. The `SimStruct`s associated with the S-functions are the child `SimStruct`s.

Note By convention, port indices begin at 0 and finish at the total number of ports minus 1.

Simulink provides a set of macros that S-functions can use to access the fields of the `SimStruct`. See Chapter 10, "SimStruct Functions" for more information.

Compiling C S-Functions

S-functions can be compiled in one of three modes identified by the presence of one of the following defines:

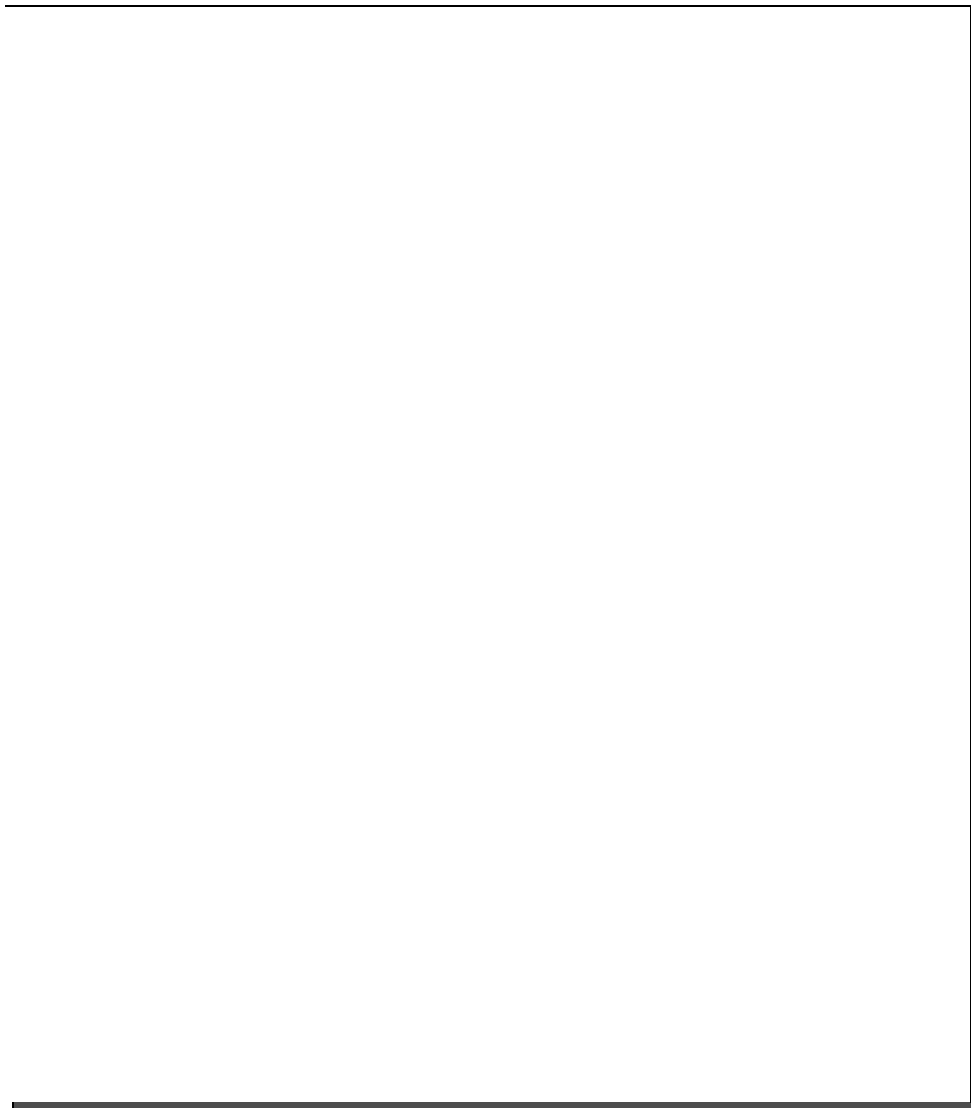
- `MATLAB_MEX_FILE` — Indicates that the S-function is being built as a MEX-file for use with Simulink.
- `RT` — Indicates that the S-function is being built with the Real-Time Workshop generated code for a real-time application using a fixed-step solver.
- `NRT` — Indicates that the S-function is being built with the Real-Time Workshop generated code for a nonreal-time application using a variable-step solver.

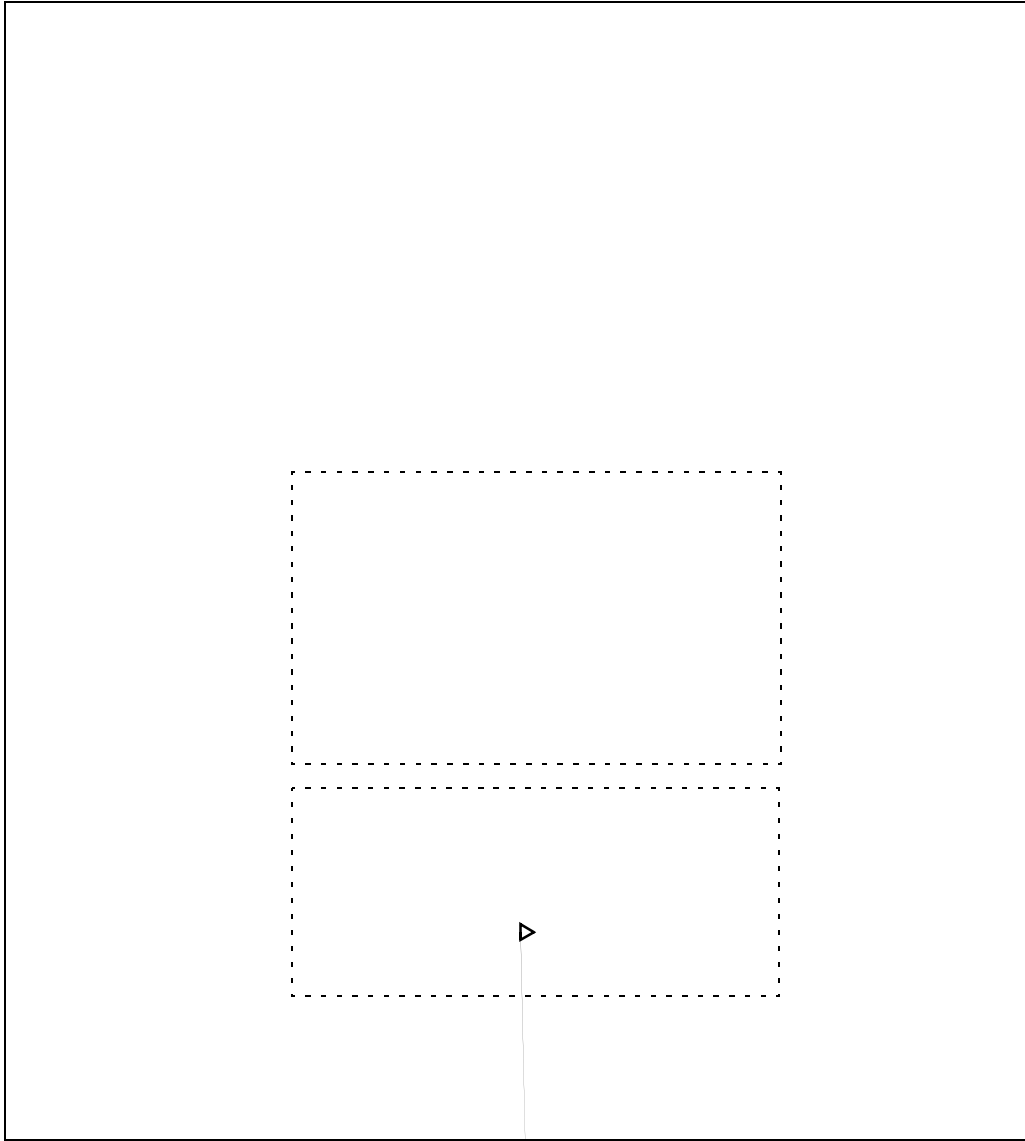
How Simulink Interacts with C S-Functions

It is helpful in writing C MEX-file S-functions to understand how Simulink interacts with S-functions. This section examines the interaction from two perspectives: a process perspective, i.e., at which points in a simulation Simulink invokes the S-function, and a data perspective, i.e., how Simulink and the S-function exchange information during a simulation.

Process View

The following figures shows the order in which Simulink invokes an S-function's callback methods.






```

*/
#include "simstruc.h"

/*=====
 * S-function methods *
 *=====*/

/* Function: mdlInitializeSizes =====
 * Abstract:
 *   The sizes information is used by Simulink to determine the S-function
 *   block's characteristics (number of inputs, outputs, states, etc.).
 */
static void mdlInitializeSizes(SimStruct *S)
{
    /* See sfuntmpl.doc for more details on the macros below */

    ssSetNumSFcnParams(S, 1); /* Number of expected parameters */
    if (ssGetNumSFcnParams(S) != ssGetSFcnParamsCount(S)) {
        /* Return if number of expected != number of actual parameters */
        return;
    }

    ssSetNumContStates(S, 0);
    ssSetNumDiscStates(S, 0);

    if (!ssSetNumInputPorts(S, 0)) return;

    if (!ssSetNumOutputPorts(S, 1)) return;
    ssSetOutputPortWidth(S, 0, 1);

    ssSetNumSampleTimes(S, 1);
    ssSetNumRWork(S, 0);
    ssSetNumIWork(S, 0);
    ssSetNumPWork(S, 1); // reserve element in the pointers vector
    ssSetNumModes(S, 0); // to store a C++ object
    ssSetNumNonsampledZCs(S, 0);

    ssSetOptions(S, 0);
}

/* Function: mdlInitializeSampleTimes =====
 * Abstract:
 *   This function is used to specify the sample time(s) for your
 *   S-function. You must register the same number of sample times as
 *   specified in ssSetNumSampleTimes.
 */
static void mdlInitializeSampleTimes(SimStruct *S)
{
    ssSetSampleTime(S, 0, mxGetScalar(ssGetSFcnParam(S, 0)));
    ssSetOffsetTime(S, 0, 0.0);
}

```



```

}

#define MDL_START /* Change to #undef to remove function */
#if defined(MDL_START)
/* Function: mdlStart =====
 * Abstract:
 * This function is called once at start of model execution. If you
 * have states that should be initialized once, this is the place
 * to do it.
 */
static void mdlStart(SimStruct *S)
{
    ssGetPWork(S)[0] = (void *) new counter; // store new C++ object in the
                                           // pointers vector
}
#endif /* MDL_START */

/* Function: mdlOutputs =====
 * Abstract:
 * In this function, you compute the outputs of your S-function
 * block. Generally outputs are placed in the output vector, ssGetY(S).
 */
static void mdlOutputs(SimStruct *S, int_T tid)
{
    counter *c = (counter *) ssGetPWork(S)[0]; // retrieve C++ object from
real_T *y = ssGetOutputPortRealSignal(S,0); // the pointers vector and use
y[0] = c->output(); // member functions of the
} // object

/* Function: mdlTerminate =====
 * Abstract:
 * In this function, you should perform any actions that are necessary
 * at the termination of a simulation. For example, if memory was
 * allocated in mdlStart, this is the place to free it.
 */
static void mdlTerminate(SimStruct *S)
{
    counter *c = (counter *) ssGetPWork(S)[0]; // retrieve and destroy C++
delete c; // object in the termination
} // function
/*=====
 * See sfuntmpl.doc for the optional S-function methods *
 *=====*/

/*=====
 * Required S-function trailer *
 *=====*/

#ifdef MATLAB_MEX_FILE /* Is this file being compiled as a MEX-file? */
#include "simulink.c" /* MEX-file interface mechanism */
#else
#include "cg_sfun.h" /* Code generation registration function */
#endif

```

```
#ifndef __cplusplus
} // end of extern "C" scope
#endif
```

Making C++ Objects Persistent

Your C++ callback methods may need to create persistent C++ objects, that is, objects that continue to exist after the method exits. For example, a callback method may need to access an object created during a previous invocation. Or one callback method may need to access an object created by another callback method. To create persistent C++ objects in your S-function:

- 1 Create a pointer work vector to hold pointers to the persistent object between method invocations.

```
static void mdlInitializeSizes(SimStruct *S)
{
    ...
    ssSetNumPWork(S, 1); // reserve element in the pointers vector
                        // to store a C++ object
    ...
}
```

- 2 Store a pointer to each object that you want to be persistent in the pointer work vector.

```
static void mdlStart(SimStruct *S)
{
    ssGetPWork(S)[0] = (void *) new counter; // store new C++ object in the
                                           // pointers vector
}
```

- 3 Retrieve the pointer in any subsequent method invocation to access the object.

```
static void mdlOutputs(SimStruct *S, int_T tid)
{
    counter *c = (counter *) ssGetPWork(S)[0]; // retrieve C++ object from
    real_T *y = ssGetOutputPortRealSignal(S, 0); // the pointers vector and use
    y[0] = c->output(); // member functions of the
                       // object
}
```

- 4 Destroy the objects when the simulation terminates.

```
static void mdlTerminate(SimStruct *S)
{
    counter *c = (counter *) ssGetPWork(S)[0]; // retrieve and destroy C++
    delete c; // object in the termination
             // function
}
```

Building C++ S-Functions

Use the MATLAB `mex` command to build C++ S-functions exactly the way you use it to build C S-functions. For example, to build the C++ version of the `sfun_counter` example, enter

```
mex sfun_counter_cpp.cpp
```

at the MATLAB command line.

Note The extension of the source file for a C++ S-function must be `.cpp` to ensure that the compiler treats the file's contents as C++ code.

Creating Ada S-Functions

Introduction	5-2
Ada S-Function Source File Format	5-3
Ada S-Function Specification	5-3
Ada S-Function Body	5-4
Writing Callback Methods in Ada	5-6
Callbacks Invoked By Simulink	5-6
Implementing Callbacks	5-7
Omitting Optional Callback Methods	5-7
SimStruct Functions	5-7
Building an Ada S-Function	5-9
Using an Ada S-Function in a Model	5-10
Example of an Ada S-Function	5-11

Introduction

Simulink allows you to use the Ada programming language to create S-functions. As with S-functions coded in other programming languages, Simulink interacts with an Ada S-function by invoking callback methods that the S-function implements. Each method performs a predefined task, such as computing block outputs, required to simulate the block whose functionality the S-function defines. Creating an Ada S-function thus entails writing Ada implementations of the callback methods required to simulate the S-function and then compiling and linking the callbacks into a library that Simulink can load and invoke during simulation. The following sections explain how to perform these tasks.

Ada S-Function Source File Format

To create an Ada S-function, you must create an Ada package that implements the callback methods required to simulate the S-function. The S-function package comprises a specification and a body.

Ada S-Function Specification

The specification specifies the methods that the Ada S-function uses and implements. The specification must specify that the Ada S-function uses the `Simulink` package, which defines data types and functions that the S-function can use to access the internal data structure (`SimStruct`) that Simulink uses to store information about the S-function (see Chapter 10, “`SimStruct` Functions”). The specification and body of the `Simulink` package reside in the `matlabroot/simulink/ada/interface/` directory.

The specification should also specify each callback method that the S-function implements as an Ada procedure exported to C. The following is an example of an Ada S-function specification that meets these requirements.

```
-- The Simulink API for Ada S-Function
with Simulink; use Simulink;

package Times_Two is

    -- The S_FUNCTION_NAME has to be defined as a constant
    -- string.
    --
    S_FUNCTION_NAME : constant String := "times_two";

    -- Every S-Function is required to have the
    -- "mdlInitializeSIZES" method.
    -- This method needs to be exported as shown below, with the
    -- exported name being "mdlInitializeSIZES".
    --
    procedure mdlInitializeSIZES(S : in SimStruct);
    pragma Export(C, mdlInitializeSIZES, "mdlInitializeSIZES");

    procedure mdlOutputs(S : in SimStruct; TID : in Integer);
    pragma Export(C, mdlOutputs, "mdlOutputs");

end Times_Two;
```

Ada S-Function Body

The Ada S-Function body provides the implementations of the S-function callback methods as illustrated in the following example.

```

with Simulink; use Simulink;
with Ada.Exceptions; use Ada.Exceptions;

package body Times_Two is

  -- Function: mdlInitializeSizes -----
  -- Abstract:
  --   Setup the input and output port attributes for this
  --   S-Function.
  --
  procedure mdlInitializeSizes(S : in SimStruct) is

  begin
    -- Set the input port attributes
    --
    ssSetNumInputPorts(      S, 1);
    ssSetInputPortWidth(    S, 0, DYNAMICALLY_SIZED);
    ssSetInputPortDataType( S, 0, SS_DOUBLE);
    ssSetInputPortDirectFeedThrough(S, 0, TRUE);
    ssSetInputPortOverwritable( S, 0, FALSE);
    ssSetInputPortOptimizationLevel(S, 0, 3);

    -- Set the output port attributes
    --
    ssSetNumOutputPorts(    S, 1);
    ssSetOutputPortWidth(  S, 0, DYNAMICALLY_SIZED);
    ssSetOutputPortDataType( S, 0, SS_DOUBLE);
    ssSetOutputPortOptimizationLevel(S, 0, 3);

    -- Set the block sample time.
    ssSetSampleTime(      S, INHERITED_SAMPLE_TIME);

  exception
    when E : others =>
      if ssGetErrorStatus(S) = "" then
        ssSetErrorStatus(S,
          "Exception occurred in mdlInitializeSizes. " &
          "Name: " & Exception_Name(E) & ", " &
          "Message: " & Exception_Message(E) &
          " and " & "Information: " &
          Exception_Information(E));
        end if;
      end mdlInitializeSizes;

  -- Function: mdlOutputs -----
  -- Abstract:

```



```
--      Compute the S-Function's output,
--      given its input: y = 2 * u
--
procedure mdlOutputs(S : in SimStruct; TID : in Integer) is

    uWidth : Integer := ssGetInputPortWidth(S, 0);
    U       : array(0 .. uWidth-1) of Real_T;
    for U' Address use ssGetInputPortSignalAddress(S, 0);

    yWidth : Integer := ssGetOutputPortWidth(S, 0);
    Y       : array(0 .. yWidth-1) of Real_T;
    for Y' Address use ssGetOutputPortSignalAddress(S, 0);

begin
    if uWidth = 1 then
        for Idx in 0 .. yWidth-1 loop
            Y(Idx) := 2.0 * U(0);
        end loop;
    else
        for Idx in 0 .. yWidth-1 loop
            Y(Idx) := 2.0 * U(Idx);
        end loop;
    end if;

exception
    when E : others =>
        if ssGetErrorStatus(S) = "" then
            ssSetErrorStatus(S,
                "Exception occurred in mdlOutputs. " &
                "Name: " & Exception_Name(E) & ", " &
                "Message: " & Exception_Message(E) & " and " &
                "Information: " & Exception_Information(E));
        end if;
    end mdlOutputs;

end Times_Two;
```

Note When interacting with Ada S-functions, Simulink invokes only a subset of the callback methods that it invokes for C S-functions. The “Languages Supported” section of the reference page for each callback method specifies whether Simulink invokes that callback when interacting with an Ada S-function.

Implementing Callbacks

Simulink defines in a general way the task of each callback. The S-function is free to perform the task according to the functionality it implements. For example, Simulink specifies that the S-function's `mdlOutput` method must compute that block's outputs at the current simulation time. It does not specify what those outputs must be. This callback-based API allows you to create S-functions, and hence custom blocks, that meet your requirements.

Chapter 9, “S-Function Callback Methods” explains the purpose of each callbacks and provides guidelines for implementing them. Chapter 3, “Writing S-Functions in C” provides information on using these callbacks to implement specific S-function features, such as the ability to handle multiple signal data types.

Omitting Optional Callback Methods

The method `mdlInitializeSizes` is the only callback that an Ada S-function must implement. The source for your Ada S-function needs to include implementations only for callbacks that it must handle. If the source for your S-function does not include an implementation for a particular callback, the mex tool that builds the S-function (see “Building an Ada S-Function” on page 5-9) provides a stub implementation.

SimStruct Functions

Simulink provides a set of functions that enable an Ada S-function to access the internal data structure (SimStruct) that Simulink maintains for the S-function. These functions consist of Ada wrappers around the SimStruct macros used to access the SimStruct from a C S-function (see Chapter 10, “SimStruct Functions”). Simulink provides Ada wrappers for a substantial

subset of the SimStruct macros. The “Languages Supported” section of the reference page for a macro specifies whether it has an Ada wrapper.

Building an Ada S-Function

To use your Ada S-function with Simulink, you must build a MATLAB executable (MEX) file from the Ada source code for the S-function. Use the MATLAB `mex` command to perform this step.

The `mex` syntax for building an Ada S-function MEX file is

```
mex [-v] [-g] -ada SFCN.ads
```

where `SFCN.ads` is the name of the S-function's package specification.

For example, to build the `ti mestwo` S-function example that comes with Simulink, enter the command

```
mex -ada ti mestwo.ads
```

Note To build a MEX file from Ada source code, using the `mex` tool, you must have previously installed a copy of version 3.2 (or higher) of the GNAT Ada95 compiler on your system. You can obtain the latest Solaris, Windows, and GNU-Linux versions of the compiler at the GNAT ftp site ([ftp: // cs.nyu.edu/pub/gnat](ftp://cs.nyu.edu/pub/gnat)). Make sure that the compiler executable is in MATLAB's command path so that the `mex` tool can find it.

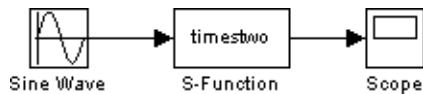
Using an Ada S-Function in a Model

The way to include an Ada S-function in a model is the same as that for including any other type of S-function. See “Using S-Functions in Models” on page 1–2 for more information.

Example of an Ada S-Function

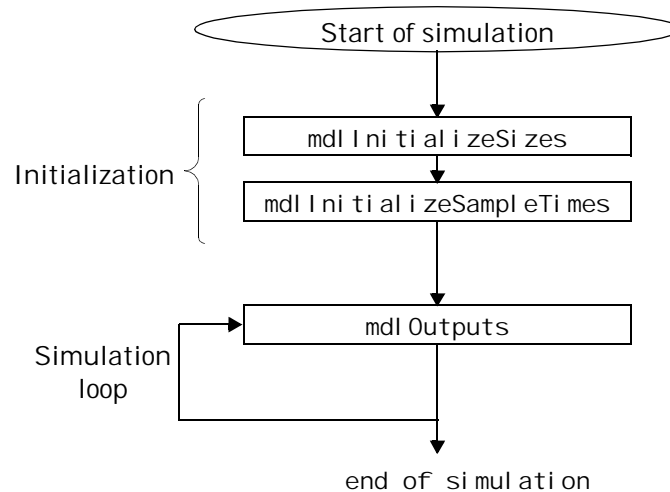
This section presents an example of a basic Ada S-function that you can use as a model when creating your own Ada S-functions. The example is the `timestwo` S-function example that comes with Simulink (see `matlabroot/simulink/ada/examples/timestwo.ads` and `matlabroot/simulink/ada/examples/timestwo.adb`). This S-function outputs twice its input.

The following model uses the `timestwo` S-function to double the amplitude of a sine wave and plot it in a scope.



The block dialog for the S-function specifies `timestwo` as the S-function name; the parameters field is empty.

The `timestwo` S-function contains the S-function callback methods shown in this figure.



The source code for the `timestwo` S-function comprises two parts:

- Package specification

- Package body

The following sections explains each of these parts.

TimesTwo Package Specification

The timesTwo package specification, timesTwo.ads, contains the following code.

```
-- The Simulink API for Ada S-Function

with Simulink; use Simulink;

package Times_Two is

    -- The S_FUNCTION_NAME has to be defined as a constant string. Note that
    -- the name of the S-Function (ada_times_two) is different from the name
    -- of this package (times_two). We do this so that it is easy to identify
    -- this example S-Function in the MATLAB workspace. Normally you would use
    -- the same name for S_FUNCTION_NAME and the package.
    --
    S_FUNCTION_NAME : constant String := "ada_times_two";

    -- Every S-Function is required to have the "mdlInitializeSizes" method.
    -- This method needs to be exported as shown below, with the exported name
    -- being "mdlInitializeSizes".
    --
    procedure mdlInitializeSizes(S : in SimStruct);
    pragma Export(C, mdlInitializeSizes, "mdlInitializeSizes");

    procedure mdlOutputs(S : in SimStruct; TID : in Integer);
    pragma Export(C, mdlOutputs, "mdlOutputs");

end Times_Two;
```

The package specification begins by specifying that the S-function uses the Simulink package.

```
with Simulink; use Simulink;
```

The Simulink package defines Ada procedures for accessing the internal data structure (SimStruct) that Simulink maintains for each S-function (see Chapter 10, "SimStruct Functions").

Next the specification specifies the name of the S-function.

```
S_FUNCTION_NAME : constant String := "ada_times_two";
```


The name `ada_times_two` serves to distinguish the MEX file generated from Ada source from those generated from the `timestwo` source coded in other languages.

Finally the specification specifies the callback methods implemented by the `timestwo` S-function.

```

procedure mdlInitializeSizes(S : in SimStruct);
pragma Export(C, mdlInitializeSizes, "mdlInitializeSizes");

procedure mdlOutputs(S : in SimStruct; TID : in Integer);
pragma Export(C, mdlOutputs, "mdlOutputs");

```

The specification specifies that the Ada compiler should compile each method as a C-callable function. This is because the Simulink engine assumes that callback methods are C functions.

Note When building an Ada S-function, MATLAB's `mex` tool uses the package specification to determine which callbacks the S-function does not implement. It then generates stubs for the non implemented methods.

TimesTwo Package Body

The `timestwo` package body, `timestwo.adb`, contains

```

with Simulink; use Simulink;
with Ada.Exceptions; use Ada.Exceptions;

package body Times_Two is

  -- Function: mdlInitializeSizes -----
  -- Abstract:
  --   Setup the input and output port attributes for this S-Function.
  --
  procedure mdlInitializeSizes(S : in SimStruct) is

    begin
      -- Set the input port attributes
      --
      ssSetNumInputPorts(      S, 1);
      ssSetInputPortWidth(    S, 0, DYNAMICALLY_SIZED);
      ssSetInputPortDataType( S, 0, SS_DOUBLE);
      ssSetInputPortDirectFeedThrough(S, 0, TRUE);
      ssSetInputPortOverwrite(S, 0, FALSE);
      ssSetInputPortOptimizationLevel(S, 0, 3);
    end;
  end;

```



```
0001607 .note
0001615 .comment
0003071 sfun_atmos_for. for
0003101 gcc2_compl ed.
0003120 rearth. 0
0003131 gmr. 1
0003137 htab. 2
0003146 ttab. 3
0003155 ptab. 4
0003164 gtab. 5
0003173 atmos_
0003207 exp
0003213 pow_d
```

Note that Atmos has been changed to atmos_ and the latter is what the C program must call to be successful.

With Compaq Visual Fortran, the symbol is suppressed, so that Atmos becomes ATMOS (no underscore).

Fortran Math Library

Fortran math library symbols may not match C math library symbols. For example A^B in Fortran will call library function pow_dd, which is not in the C math library. In these cases, you must tell mex to link in the Fortran math library. For gcc environments, these routines are usually found in /usr/local/lib/libf2c.a, /usr/lib/libf2c.a or equivalent.

The mex command becomes

```
mex -L/usr/local/lib -lf2c cmex_c_file fortran_object_file
```

Note On UNIX, the -lf2c option follows the conventional UNIX library linking syntax where '-l' is the library option itself and 'f2c' is the unique part of the library file's name, libf2c.a. Be sure to use the -L option for the library search path since -l is only followed while searching for include files.

The f2c package can be obtained for Windows and UNIX environments from the Internet. The file libf2c.a is usually part of g77 distributions, or else the file is not needed as the symbols match. In obscure cases, it must be installed separately, but even this is not difficult once the need for it is identified.

On Windows using Microsoft Visual C/C++ and Compaq Visual Fortran 6.0 (formerly known as Digital Fortran), this example can be compiled using the following mex commands (each command is on one line).

```
mex -v COMPFLAGS#" $COMPFLAGS /i face: cref" -c sfun_atmos_sub.for
-f . . \ . \bin\win32\mexopts\df60opts.bat
mex -v LINKFLAGS#" $LINKFLAGS dfor.lib dfconsole.lib dfport.lib
/LIBPATH: $DF_ROOT\DF98\LIB" sfun_atmos.c sfun_atmos_sub.obj
```

See *matlabroot*/simulink/src/sfuntmpl_fortran.txt and *matlabroot*/simulink/src/sfun_atmos.c for the latest information on compiling Fortran for C on Windows.

CFortran

Or try using CFortran to create an interface. CFortran is a tool for automated interface generation between C and Fortran modules, in either direction. Search the Web for cfortran or visit

<http://www-zeus.desy.de/~burow/cfortran/>

for downloading.

Obtaining a Fortran Compiler

On Windows using Visual C/C++ with Fortran is best done with Compaq Visual Fortran, Absoft, Lahey or other third-party compilers. See Compaq (www.compaq.com) and Absoft (www.absoft.com) for Windows, Linux, and Sun compilers and see Lahey (www.lahey.com) for more choices in Windows Fortran compilers.

For Sun (Solaris) and other commercial UNIX platforms, one can purchase the computer vendor's Fortran compiler, a third-party Fortran such as Absoft, or even use the Gnu Fortran port for that platform (if available).

As long as the compiler can output the same object (.o) format as the platform's C compiler, the Fortran compiler will work with the gateway C-MEX S-function technique.

Gnu Fortran (g77) can be obtained free for several platforms from many download sites, including [tap: //www.redhat.com](http://www.redhat.com) in the download area. A useful keyword on search engines is g77.

Constructing the Gateway

The `mdlInitializeSizes()` and `mdlInitializeSampleTimes()` methods are coded in C. It is unlikely that you will need to call Fortran routines from these S-function methods. In the simplest case, the Fortran is called only from `mdlOutputs()`.

Simple Case

The Fortran code must at least be callable in a “step at a time” fashion. If the code doesn’t have any states, it can be called from `mdlOutputs()` and no `mdlDerivatives()` or `mdlUpdate()` methods are required.

Code with States

If the code has states, you must decide if the Fortran code can support a variable step solver or not. For fixed-step solver only support, the C gateway consists of a call to the Fortran code from `mdlUpdate()` and outputs are cached in an S-function `DWork` vector so that subsequent calls by Simulink into `mdlOutputs()` will work properly and the Fortran code won’t be called until the next invocation of `mdlUpdate()`. In this case, the states in the code can be stored however you like, typically in the work vector or as discrete states in Simulink.

If instead the code needs to have continuous time states with support for variable step solvers, the states must be registered and stored with Simulink as doubles. This is done in `mdlInitializeSizes()` (registering states), then the states are retrieved and sent to the Fortran code whenever you need to execute it. In addition, the main body of code has to be separable into a call form that can be used by `mdlDerivatives()` to get derivatives for the state integration and also by the `mdlOutputs()` and `mdlUpdate()` methods as appropriate.

Setup Code

If there is a lengthy setup calculation, it is best to make this part of the code separable from the “one step at a time” code and call it from `mdlStart()`. This can either be a separate SUBROUTINE called from `mdlStart()` that communicates with the rest of the code through COMMON blocks or argument I/O, or it can be part of the same piece of Fortran code that is isolated by an IF-THEN-ELSE construct. This construct can be triggered by one of the input arguments that tells the code if it is to either perform the setup calculations or the “one step” calculations.

SUBROUTINE Versus PROGRAM

To be able to call Fortran from Simulink directly without having to launch processes, etc., it is necessary to convert a Fortran PROGRAM into a SUBROUTINE. This consists of three steps. The first is trivial, the second and third can take a bit of examination:

- 1 Change the line PROGRAM to SUBROUTINE subName.

Now you can call it from C using C function syntax.

- 2 Identify variables that need to be inputs and outputs and put them in the SUBROUTINE argument list or in a COMMON block.

It is customary to strip out all hard-coded cases and output dumps. In the Simulink environment, you want to convert inputs and outputs into block I/O.

- 3 If you are converting a stand-alone simulation to work inside of Simulink, identify the "main loop" of time integration and remove both the loop and, if you want Simulink to integrate continuous states, remove any time integration code. Leave time integrations in the code if you intend to make a discrete time (sampled) S-function.

Arguments to a SUBROUTINE

Most Fortran compilers generate SUBROUTINE code that passes arguments "by reference." This means that the C code calling the Fortran code must use only pointers in the argument list.

```
PROGRAM . . .
```

becomes

```
SUBROUTINE somename( U, X, Y )
```

A SUBROUTINE never has a return value. I/O is achieved by using some of the arguments for input, the rest for output.

Arguments to a FUNCTION

A FUNCTION has a scalar return value passed by value, so a calling C program should expect this. The argument list is passed by reference (i.e., pointers) as in the SUBROUTINE.

If the result of a calculation is an array, then a subroutine should be used as a FUNCTION cannot return an array.

Interfacing to COMMON blocks

While there are several ways for Fortran COMMON blocks to be visible to C code, it is often recommended to use an input/output argument list to a SUBROUTINE or FUNCTION. If the Fortran code has already been written and uses COMMON blocks, it is a simple matter to write a small SUBROUTINE that has an input/output argument list and copies data into and out of the COMMON block.

The procedure for copying in and out of the COMMON block begins with a write of the inputs to the COMMON block before calling the existing SUBROUTINE. The SUBROUTINE is called, then the output values are read out of the COMMON block and copied into the output variables just before returning.

An Example C-MEX S-Function Calling Fortran Code

The subroutine Atmos is in file `sfun_atmos_sub.for`. The gateway C-MEX S-function is `sfun_atmos.c`, which is built on UNIX using the command

```
mex -L/usr/local/lib -lf2c sfun_atmos.c sfun_atmos_sub.o
```

On Windows, the command is

```
>> mex -v COMPFLAGS#"$COMPFLAGS /iface:crcf" -c sfun_atmos_sub.for
-f . . .\bin\win32\mexopts\df60opts.bat
>> mex -v LINKFLAGS#$LINKFLAGS dfor.lib dfconsol.lib dfport.lib
/LIBPATH: $DF_ROOT\DF98\LIB" sfun_atmos.c sfun_atmos_sub.obj
```

On some UNIX systems where the C and Fortran compiler were installed separately (or aren't aware of each other), you may need to reference the library `libf2c.a`. To do this, use the `-lf2c` flag.

UNIX only: if the `libf2c.a` library isn't on the library path, you need to add it the path to the `mex` process explicitly with the `-L` command, for instance:

```
mex -L/usr/local/lib/ -lf2c sfun_atmos.c sfun_atmos_sub.o
```

This sample is prebuilt and is on the MATLAB search path already, so you can see it working by opening the sample model `sfcndemo_atmos.mdl`. Just type

```
sfcndemo_atmos
```

at the command prompt, or to get all the S-function demos for Simulink, type `sfcndemos` at the MATLAB prompt.

Porting Legacy Code

Find the States

If a variable step solver is being used, it is critical that all continuous states are identified in the code and put into Simulink's state vector for integration instead of being integrated by the Fortran code. Likewise, all derivative calculations must be made available separately to be called from the `mdlDerivatives()` method in the S-function. Without these steps, any Fortran code with continuous states will not be compatible with variable step solvers, if the S-function is registered as a continuous block with continuous states.

Telltale signs of implicit advancement are incremented variables such as `M=M+1` or `X=X+0.05`. If the code has many of these constructs and you determine that it is impractical to recode the source to not "ratchet forward," you may need to try another approach using fixed step solvers.

If it is impractical to find all the implicit states and to separate out the derivative calculations for Simulink, another approach can be used, but you are limited to using fixed step solvers. The technique here is to call the Fortran code from the `mdlUpdate()` method so the Fortran code is only executed once per Simulink major integration step. Any block outputs must be cached in a work vector so that `mdlOutputs()` can be called as often as needed and output the values from the work vector instead of calling the Fortran routine again (which would cause it to inadvertently advance time). See *matlabroot/simulink/src/sfuntmpl_gate_fortran.c* for an example that uses DWork vectors.

Sample Times

Be sure if the code has an implicit step size in its algorithm, coefficients, etc., that you register the proper discrete sample time in the `mdlInitializeSampleTimes()` S-function method and only change the block's output values from the `mdlUpdate()` method.

Multiple Instances

If you plan on having multiple copies of this S-function used in one Simulink model, it is necessary to allocate storage for each copy of the S-function in the model. The recommended approach is to use DWork vectors, see *matlabroot/*

`simulink/include/simstruc.h` and `matlabroot/simulink/src/sfuntmpl.doc` for details on allocating data typed work vectors.

Use Flints If Needed

Use flints (floating-point ints) to keep track of time. Flints (for IEEE-754 floating-point numerics) have the useful property of not accumulating round off error when adding and subtracting flints. Using flint variables in DOUBLE PRECISION storage (with integer values) avoids round off error accumulation that would accumulate when floating point numbers are added together thousands of times.

```
DOUBLE PRECISION F
      :
      :
      F = F + 1.0
      TIME = 0.003 * F
```

This technique avoids a common pitfall in simulations.

Considerations for Real Time

Since very few Fortran applications are used in a real-time environment, it is more common to come across simulation code that is incompatible with a real-time environment. Common failures include unbounded (or large) iterations and sporadic but time-intensive side calculations. These must be dealt with directly if there is to be any hope of running in real time.

Conversely, it is still perfectly good practice to have iterative or sporadic calculations if the generated code is not being used for a real-time application.

Implementing Block Features

Introduction	7-2
Dialog Parameters	7-3
Run-Time Parameters	7-6
Input and Output Ports	7-9
Custom Data Types	7-15
Sample Times	7-16
Work Vectors	7-24
Function-Call Subsystems	7-29
Handling Errors	7-31
S-Function Examples	7-34

Introduction

This chapter explains how to use S-function callback methods to implement various block features.

Dialog Parameters

A user can pass parameters to an S-function at the start of and, optionally, during the simulation, using the S-Function parameters field of the block's dialog box. Such parameters are called dialog box parameters to distinguish them from run-time parameters created by the S-function to facilitate code generation (see "Run-Time Parameters" on page 7-6). Simulink stores the values of the dialog box parameters in the S-function's SimStruct structure. Simulink provides callback methods and SimStruct macros that allow the S-function to access and check the parameters and use them in the computation of the block's output.

If you want your S-function to be able to use dialog parameters, you must perform the following steps when you create the S-function:

- 1 Determine the order in which the parameters are to be specified in the block's dialog box.
- 2 In the `mdlInitializeSFunction` function, use the `ssSetNumSFcnParams` macro to tell Simulink how many parameters the S-function accepts. Specify `S` as the first argument and the number of parameters you are defining interactively as the second argument. If your S-function implements the `mdlCheckParameters` method, the `mdlInitializeSFunction` routine should call `mdlCheckParameters` to check the validity of the initial values of the parameters.
- 3 Access these input arguments in the S-function using the `ssGetSFcnParam` macro.

Specify `S` as the first argument and the relative position of the parameter in the list entered on the dialog box (0 is the first position) as the second argument. The `ssGetSFcnParam` returns a pointer to the `mxArray` containing the parameter. You can use `ssGetDataTypeFromMxArray` to get the data type of the parameter.

When running a simulation, the user must specify the parameters in the S-Function parameters field of the block's dialog box in the same order that you defined them in step 1 above. The user can enter any valid MATLAB expression as the value of a parameter, including literal values, names of workspace variables, function invocations, or arithmetic expressions. Simulink evaluates the expression and passes its value to the S-function.

For example, the following code is part of a device driver S-function. Four input parameters are used: `BASE_ADDRESS_PRM`, `GAIN_RANGE_PRM`, `PROG_GAIN_PRM`, and `NUM_OF_CHANNELS_PRM`. The code uses `#define` statements to associate particular input arguments with the parameter names.

```
/* Input Parameters */
#define BASE_ADDRESS_PRM(S)      ssGetSFcnParam(S, 0)
#define GAIN_RANGE_PRM(S)       ssGetSFcnParam(S, 1)
#define PROG_GAIN_PRM(S)        ssGetSFcnParam(S, 2)
#define NUM_OF_CHANNELS_PRM(S)  ssGetSFcnParam(S, 3)
```

When running the simulation, a user would enter four variable names or values in the S-Function parameters field of the block's dialog box. The first corresponds to the first expected parameter, `BASE_ADDRESS_PRM(S)`. The second corresponds to the next expected parameter, and so on.

The `mdlInitializeSizes` function contains this statement.

```
ssSetNumSFcnParams(S, 4);
```

Tunable Parameters

Dialog parameters can be either tunable or nontunable. A tunable parameter is a parameter that a user can change while the simulation is running. Use the macro `ssSetSFcnParamTunable` in `mdlInitializeSizes` to specify the tunability of each dialog parameter used by the macro.

Note Dialog parameters are tunable by default. Nevertheless, it is good programming practise to set the tunability of every parameter, even those that are tunable. If the user enables the simulation diagnostic, S-function upgrade needed, Simulink issues the diagnostic whenever it encounters an S-function that fails to specify the tunability of all its parameters.

The `mdlCheckParameters` method enables you to validate changes to tunable parameters during a simulation run. Simulink invokes the `mdlCheckParameters` method whenever a user changes the values of parameters during the simulation loop. This method should check the S-function's dialog parameters to ensure the changes are valid.

Note The S-function's `mdlInitializeSizes` routine should also invoke the `mdlCheckParameters` method to ensure that the initial values of the parameters are valid.

The optional `mdlProcessParameters` callback method allows an S-function to process changes to tunable parameters. Simulink invokes this method only if valid parameter changes have occurred in the previous time step. A typical use of this method is to perform computations that depend only on the values of parameters and hence need to be computed only when parameter values change. The method can cache the results of the parameter computations in work vectors or, preferably, as run-time parameters (see "Run-Time Parameters" on page 7-6).

Tuning Parameters in External Mode

When a user tunes parameters during simulation, Simulink invokes the S-function's `mdlCheckParameters` method to validate the changes and then the S-functions' `mdlProcessParameters` method to give the S-function a chance to process the parameters in some way. When running in external mode, Simulink invokes these methods as well but it passed the unprocessed changes onto the S-function target. Thus, if it is essential that your S-function process parameter changes, you need to create a Target Language Compiler (TLC) file that inlines the S-function, including its parameter processing code, during the code generation process. For information on inlining S-functions, see the Target Language Compiler Reference Guide.

Run-Time Parameters

Simulink allows an S-function to create and use internal representations of external dialog parameters called run-time parameters. Every run-time parameter corresponds to one or more dialog parameters and can have the same value and data type as its corresponding external parameter(s) or a different value or data type. If a run-time parameter differs in value or data type from its external counterpart, the dialog parameter is said to have been transformed to create the run-time parameter. The value of a run-time parameter that corresponds to multiple dialog parameter is typically a function of the values of the dialog parameters. Simulink allocates and frees storage for run-time parameters and provides functions for updating and accessing them, thus eliminating the need for S-functions to perform these tasks.

Run-time parameters facilitate the following kinds of S-function operations:

- Computed parameters

Often the output of a block is a function of the values of several dialog parameters. For example, suppose a block has two parameters, the volume and density of some object, and the output of the block is a function of the input signal and the weight of the object. In this case, the weight can be viewed as a third internal parameter computed from the two external parameters, volume and density. An S-function can create a run-time parameter corresponding to the computed weight, thereby eliminating the need to provide special case handling for weight in the output computation.

- Data type conversions

Often a block may need to change the data type of a dialog parameter to facilitate internal processing. For example, suppose that the output of the block is a function of the input and a parameter and the input and parameter are of different data types. In this case, the S-function can create a run-time parameter that has the same value as the dialog parameter but has the data type of the input signal and use the run-time parameter in the computation of the output.

- Code generation

During code generation, Real-Time Workshop writes all run-time parameters automatically to the *model.rtw file*, eliminating the need for the S-function to perform this task via a mdl RTW method.

Creating Run-Time Parameters

An S-function can create run-time parameters all at once or one by one.

Creating Run-Time Parameters All at Once

Use the `SimStruct` function, `ssRegAllTunableParamsAsRunTimeParams`, in `mdlSetWorkWidths` to create run-time parameters corresponding to all tunable parameters. This function requires that you pass it an array of names, one for each run-time parameter. Real-Time Workshop uses this name as the name of the parameter during code generation.

This approach to creating run-time parameters assumes that there is a one-to-one correspondence between an S-function's run-time parameters and its tunable dialog parameters. This may not be the case. For example, an S-function may want to use a computed parameter whose value is a function of several dialog parameters. In such cases, the S-function may need to create the run-time parameters individually.

Creating Run-Time Parameters Individually

To create run-time parameters individually, the S-function's `mdlSetWorkWidths` method should:

- 1 Specify the number of run-time parameters it intends to use, using `ssSetNumRunTimeParams`.
- 2 Specify the attributes of each run-time parameter, using `ssSetRunTimeParamInfo`.

Updating Run-Time Parameters

Whenever a user changes the values of an S-function's dialog parameters during a simulation run, Simulink invokes the S-function's `mdlCheckParameters` method to validate the changes. If the changes are valid, Simulink invokes the S-function's `mdlProcessParameters` method at the beginning of the next time step. This method should update the S-function's run-time parameters to reflect the changes in the dialog parameters.

Updating All Parameters at Once

If there is a one-to-one correspondence between the S-function's tunable dialog parameters and the run-time parameters, the S-function can use the

SimStruct function, `ssUpdateAllTunableParamsAsRunTimeParams`, to accomplish this task. This function updates each run-time parameter to have the same value as the corresponding dialog parameter.

Updating Parameters Individually

If there is not a one-to-one correspondence between the S-function's dialog and run-time parameters or the run-time parameters are transformed versions of the dialog parameters, the mdl `ProcessParameters` method must update each parameter individually.

If a run-time parameter and its corresponding dialog parameter differ only in value, the method can use the `SimStruct` macro, `ssUpdateRunTimeParamData`, to update the run-time parameter. This function updates the data field in the parameter's attributes record (`ssParamRec`) with a new value. Otherwise, the mdl `ProcessParameters` method must update the parameter's attributes record itself. To update the attributes record, the method should:

- 1 Get a pointer to the parameter's attributes record, using `ssGetRunTimeParamInfo`.
- 2 Update the attributes record to reflect the changes in the corresponding dialog parameter(s).
- 3 Register the changes, using `ssUpdateRunTimeParamInfo`.

Input and Output Ports

Simulink allows S-functions to create and use any number of block I/O ports. This section shows how to create and initialize I/O ports and how to change the characteristics of an S-function block's ports, such as dimensionality and data type, based on its connections to other blocks.

Creating Input Ports

To create and configure input ports, the `mdlInitializeSizes` method should first specify the number of input ports that the S-function has, using `ssSetNumInputPorts`. Then, for each input port, the method should specify:

- The dimensions of the input port (see “Initializing Input Port Dimensions” on page 7-10)

If you want your S-function to inherit its dimensionality from the port to which it is connected, you should specify that the port is dynamically sized in `mdlInitializeSizes` (see “Sizing an Input Port Dynamically” on page 7-10).

- Whether the input port allows scalar expansion of inputs (see “Scalar Expansion of Inputs” on page 7-12)
- Whether the input port has direct feedthrough, using `ssSetInputPortDirectFeedThrough`

A port has direct feedthrough if the input is used in either the `mdlOutputs` or `mdlGetTimeOfNextVarHit` functions. The direct feedthrough flag for each input port can be set to either 1=yes or 0=no. It should be set to 1 if the input, `u`, is used in the `mdlOutput` or `mdlGetTimeOfNextVarHit` routine. Setting the direct feedthrough flag to 0 tells Simulink that `u` will not be used in either of these S-function routines. Violating this will lead to unpredictable results.

- The data type of the input port, if not the default `double`

Use `ssSetInputPortDataType` to set the input port's data type. If you want the data type of the port to depend on the data type of the port to which it is connected, specify the data type as `DYNAMICALLY_TYPED`. In this case, you must provide implementations of the `mdlSetInputPortDataType` and `mdlSetDefaultPortDataTypes` methods to enable the data type to be set correctly during signal propagation.

- The numeric type of the input port, if the port accepts complex-valued signals. Use `ssSetInputComplexSignal` to set the input port's numeric type. If you want the numeric type of the port to depend on the numeric type of the port to which it is connected, specify the data type as inherited. In this case, you must provide implementations of the `mdlSetInputPortComplexSignal` and `mdlSetDefaultPortComplexSignal` methods to enable the numeric type to be set correctly during signal propagation.

Note The `mdlInitializeSizes` method must specify the number of ports before setting any properties. If it attempts to set a property of a port that doesn't exist, it will be accessing invalid memory and Simulink will crash.

Initializing Input Port Dimensions

The following options exist for setting the input port dimensions:

- If the input signal is one-dimensional, and the input port width is w , use `ssSetInputPortVectorDimension(S, inputPortIdx, w)`
- If the input signal is a matrix of dimension m -by- n , use `ssSetInputPortMatrixDimensions(S, inputPortIdx, m, n)`
- Otherwise use `ssSetInputPortDimensionInfo(S, inputPortIdx, dimInfo)`. This function can be used to fully or partially initialize the port dimensions (see next section).

Sizing an Input Port Dynamically

If your S-function does not require that an input signal have a specific dimensionality, you may want to set the dimensionality of the input port to match the dimensionality of the signal actually connected to the port. To dimension an input port dynamically, your S-function should:

- Specify some or all of the dimensions of the input port as dynamically sized in `mdlInitializeSizes`

- If the input port can accept a signal of any dimensionality, use `ssSetInputPortDimensionInfo(S, inputPortIdx, DYNAMIC_DIMENSION)` to set the dimensionality of the input port.
- If the input port can accept only vector (1-D) signals but the signals can be of any size, use `ssSetInputPortWidth(S, inputPortIdx, DYNAMICALLY_SIZED)` to specify the dimensionality of the input port.
If the input port can accept only matrix signals but can accept any row or column size, use `ssSetInputPortMatrixDimensions(S, inputPortIdx, m, n)` where *m* and/or *n* are `DYNAMICALLY_SIZED`.
- Provide a `mdlSetInputPortDimensionInfo` method that sets the dimensions of the input port to the size of the signal connected to it
Simulink invokes this method during signal propagation when it has determined the dimensionality of the signal connected to the input port.
- Provide a `mdlSetDefaultPortDimensionInfo` method that sets the dimensions of the block's ports to a default value
Simulink invokes this method during signal propagation when it cannot determine the dimensionality of the signal connected to some or all of the block's input ports. This can happen, for example, if an input port is unconnected. If the S-function does not provide this method, Simulink sets the dimension the block's ports to 1-D scalar.

Creating Output Ports

To create and configure output ports, the `mdlInitializeSizes` method should first specify the number of input ports that the S-function has, using `ssSetNumOutputPorts`. Then, for each output port, the method should specify:

- Dimensions of the output port
Simulink provides the following macros for setting the port's dimensions.
 - `ssSetOutputPortDimensionInfo`
 - `ssSetOutputPortMatrixDimensions`
 - `ssSetOutputPortVectorDimensions`

- `ssSetOutputWidth`

If you want the port's dimensions to depend on block connectivity, set the dimensions to `DYNAMICALLY_SIZED`. The S-function must then provide `mdlSetOutputPortDimensions` and `ssSetDefaultPortDimensions` methods to ensure that output port dimensions are set to the correct values in code generation.

- Data type of the output port

Use `ssSetOutputPortDataType` to set the output port's data type. If you want the data type of the port to depend on block connectivity, specify the data type as `DYNAMICALLY_TYPED`. In this case, you must provide implementations of the `mdlSetOutputPortDataType` and `mdlSetDefaultPortDataTypes` methods to enable the data type to be set correctly during signal propagation.

- The numeric type of the input port, if the port outputs complex-valued signals

Use `ssSetOutputComplexSignal` to set the output port's numeric type. If you want the numeric type of the port to depend on the numeric type of the port to which it is connected, specify the data type as `inherited`. In this case, you must provide implementations of the `mdlSetOutputPortComplexSignal` and `mdlSetDefaultPortComplexSignal` methods to enable the numeric type to be set correctly during signal propagation.

Scalar Expansion of Inputs

Scalar expansion of inputs refers conceptually to the process of expanding scalar input signals to have the same dimensions as the port to which they are connected. This is done by setting each element of the expanded signal to the value of the scalar input. An S-function's `mdlInitializeSizes` method can enable scalar expansion of inputs for its input ports by setting the `SS_OPTION_ALLOW_INPUT_SCALAR_EXPANSION` option, using `ssSetOptions`.

The best way to understand the scalar expansion rules is to consider a sum block with two input ports, where the first input signal is scalar, the second input signal is a 1-D vector with $w > 1$ elements, and the output signal is a 1-D vector with w elements. In this case, the scalar input is expanded to a 1-D vector with w elements in the output method, and each element of the expanded signal is set to the value of the scalar input.

```
Outputs
  <snip>
```



```

u1inc = (u1width > 1);
u2inc = (u2width > 1);
for (i=0; i<w; i++) {
    y[i] = *u1 + *u2;
    u1 += u1inc;
    u2 += u2inc;
}

```

If the block has more than two inputs, each input signal must be scalar, or the wide signals must have the same number of elements. In addition, if the wide inputs are driven by 1-D and 2-D vectors, the output will be a 2-D vector signal, and the scalar inputs are expanded to a 2-D vector signal.

The way scalar expansion actually works depends on whether the S-function manages the dimensions of its input and output ports using `mdlSetInputPortWidth` and `mdlSetOutputPortWidth` or `mdlSetInputPortDimensions`, `mdlSetOutputPortDimensions`, and `mdlSetDefaultPortDimensions`.

If the S-function does not specify/control the dimensions of its input and output ports using the above methods, Simulink uses a default method to set the input and output ports using the above methods, Simulink uses a default method to set the S-function port dimensions.

In `mdlInitializeSizes` method, the S-function can enable scalar expansion for its input ports by setting the `SS_OPTION_ALLOW_INPUT_SCALAR_EXPANSION` option, using `ssSetOptions`. Simulink default method uses the above option to allow or disallow scalar expansion for a block input ports. If the above option is not set by an S-function, Simulink assumes all ports (input and output ports) must have the same dimensions, and it sets all port dimensions to the same dimensions specified by one of the driving blocks.

If the S-function specifies/controls the dimensions of its input and output ports, Simulink ignores the `SCALAR_EXPANSION` option.

See `matlabroot/simulink/src/sfun_multiport.c` for an example.

Masked Multiport S-Functions

If you are developing masked multiport S-function blocks whose number of ports varies based on some parameter, and if you want to place them in a Simulink library, then you must specify that the mask modifies the appearance of the block. To do this, execute this command

```
set_param('block', 'MaskSel fModi fi abl e', 'on' )
```

at the MATLAB prompt before saving the library. Failure to specify that the mask modifies the appearance of the block means that an instance of the block in a model reverts to the number of ports in the library whenever you load the model or update the library link.

Custom Data Types

An S-function can accept and output user-defined as well as built-in Simulink data types. To use a user-defined data type, the S-function's `mdlInitializeSizes` routine must:

- 1 Register the data type, using `ssRegisterDataType`.
- 2 Specify the amount of memory in bytes required to store an instance of the data type, using `ssSetDataTypeSize`.
- 3 Specify the value that represents zero for the data type, using `ssSetDataTypeZero`.

Sample Times

Simulink supports blocks that execute at different rates. There are three methods by which you can specify the rates (i.e., sample times):

- Block-based sample times
- Port-based sample times
- Hybrid block-based and port-based sample times

In the case of block-based sample times, your S-function specifies all the sample rates of the block and processes inputs and outputs at the fastest rate specified if all the sample times are integer multiples of the fastest sample time. (If your sample times are not multiples of each other, Simulink behaves differently. See “Sample Time Colors” in chapter 9 of *Using Simulink* for more information.) When using port-based sample times, your S-function specifies the sample time for each input and output port. To compare block-based versus port-based sample times, consider two sample rates, 0.5 and 0.25 seconds respectively:

- In the block-based method, selecting 0.5 and 0.25 would direct the block to execute inputs and outputs at 0.25 second increments.
- In the port-based method, you could set the input port to 0.5 and the output port to 0.25, and the block would execute inputs at 2Hz and outputs at 4Hz.

You should use port-based sample times if your application requires unequal sample rates for input and output execution or if you don't want the overhead associated with running input and output ports at the highest sample rate of your block.

In some applications, an S-Function block may need to operate internally at one or more sample rates while inputting or outputting signals at other rates. The hybrid block- and port-based method of specifying sample rates allows you to create such blocks.

In typical applications, you will specify only one block-based sample time. Advanced S-functions may require the specification of port-based or multiple block sample times.

Block-Based Sample Times

The next two sections discuss how to specify block-based sample times. You must specify information in

- `mdlInitializeSizes`
- `mdlInitializeSampleTimes`

A third section presents a simple example that shows how to specify sample times in `mdlInitializeSampleTimes`.

Specifying the Number of Sample Times in `mdlInitializeSizes`. To configure your S-function block for block-based sample times, use

```
ssSetNumSampleTimes(S, numSampleTimes);
```

where `numSampleTimes > 0`. This tells Simulink that your S-function has block-based sample times. Simulink calls `mdlInitializeSampleTimes`, which in turn sets the sample times.

Setting Sample Times and Specifying Function Calls in `mdlInitializeSampleTimes`

`mdlInitializeSampleTimes` is used to specify two pieces of execution information:

- **Sample and offset times** — In `mdlInitializeSizes`, specify the number of sample times you'd like your S-function to have by using the `ssSetNumSampleTimes` macro. In `mdlInitializeSampleTimes`, you must specify the sampling period and offset for each sample time. Sample times can be a function of the input/output port widths. In `mdlInitializeSampleTimes`, you can specify that sample times are a function of `ssGetInputPortWidth` and `ssGetOutputPortWidth`.
- **Function calls** — In `ssSetCallSystemOutput`, specify which output elements are performing function calls. See `matlabroot/simulink/src/sfun_fcncall.c` for an example.

The sample times are specified as pairs [*sample_time*, *offset_time*] by using these macros

```
ssSetSampleTime(S, sampleTimePairIndex, sample_time)
ssSetOffsetTime(S, offsetTimePairIndex, offset_time)
```

where *sampleTimePairIndex* starts at 0.

The valid sample time pairs are (upper-case values are macros defined in `simstruc.h`).

```
[CONTINUOUS_SAMPLE_TIME, 0.0 ]
[CONTINUOUS_SAMPLE_TIME, FIXED_MINOR_STEP_OFFSET]
[discrete_sample_period, offset ]
[VARIABLE_SAMPLE_TIME, 0.0 ]
```

Alternatively, you can specify that the sample time is inherited from the driving block in which case the S-function can have only one sample time pair

```
[INHERITED_SAMPLE_TIME, 0.0 ]
```

or

```
[INHERITED_SAMPLE_TIME, FIXED_MINOR_STEP_OFFSET]
```

The following guidelines may help aid in specifying sample times:

- A continuous function that changes during minor integration steps should register the [CONTINUOUS_SAMPLE_TIME, 0.0] sample time.
- A continuous function that does not change during minor integration steps should register the [CONTINUOUS_SAMPLE_TIME, FIXED_MINOR_STEP_OFFSET] sample time.
- A discrete function that changes at a specified rate should register the discrete sample time pair [discrete_sample_period, offset]

where

discrete_sample_period > 0.0

and

$0.0 \leq \textit{offset} < \textit{discrete_sample_period}$

- A discrete function that changes at a variable rate should register the variable step discrete [VARIABLE_SAMPLE_TIME, 0.0] sample time. The mdlGetTimeOfNextVarHit function is called to get the time of the next sample hit for the variable step discrete task. The VARIABLE_SAMPLE_TIME can be used with variable step solvers only.

If your function has no intrinsic sample time, then you must indicate that it is inherited according to the following guidelines:

- A function that changes as its input changes, even during minor integration steps, should register the [INHERITED_SAMPLE_TIME, 0.0] sample time.

- A function that changes as its input changes, but doesn't change during minor integration steps (that is, held during minor steps), should register the `[INHERITED_SAMPLE_TIME, FIXED_IN_MINOR_STEP_OFFSET]` sample time.

To check for a sample hit during execution (in `mdl Outputs` or `mdl Update`), use the `ssSampleHit` or `ssContinuousTask` macro. For example, if your first sample time is continuous, then you used the following code fragment to check for a sample hit. Note that you would get incorrect results if you used `ssSampleHit(S, 0, tid)`.

```
if (ssContinuousTask(S, tid)) {
}
```

If, for example, you wanted to determine if the third (discrete) task has a hit, then you would use the following code-fragment.

```
if (ssSampleHit(S, 2, tid) {
}
```

Example: `mdlInitializeSampleTimes`

This example specifies that there are two discrete sample times with periods of 0.01 and 0.5 seconds.

```
static void mdlInitializeSampleTimes(SimStruct *S)
{
    ssSetSampleTime(S, 0, 0.01);
    ssSetOffsetTime(S, 0, 0.0);
    ssSetSampleTime(S, 1, 0.5);
    ssSetOffsetTime(S, 1, 0.0);
} /* End of mdlInitializeSampleTimes. */
```

Port-Based Sample Times

The next three sections discuss how to specify port-based sample times. You must specify information in:

- `mdlInitializeSizes`
- `mdlSetInputPortSampleTime`
- `mdlSetOutputPortSampleTime`

Specifying the Number of Sample Times in mdlInitializeSizes

To specify port-based sample times, use

```
ssSetNumSampleTimes(S, PORT_BASED_SAMPLE_TIMES)
```

with:

```
mdlSetInputPortSampleTime(S, idx, period)
mdlSetInputPortOffsetTime(S, idx, offset)
mdlSetOutputPortSampleTime(S, idx, period)
mdlSetOutputPortOffsetTime(S, idx, offset)
```

The `inputPortIndex` and `outputPortIndex` range from 0 to the number of input (output) ports minus 1.

When you specify port based sample times, Simulink will call `mdlSetInputPortSampleTime` and `mdlSetOutputPortSampleTime` to determine the rates of inherited signals. Once all rates have been determined completed, Simulink will also call `mdlInitializeSampleTimes` to configure function-call connections. If your S-function does not have any function-call connections this routine should be empty.

Note `mdlInitializeSizes` should not contain any `ssSetSampleTime` or `ssSetOffsetTime` calls when using port-based sample times.

Hybrid Block-Based and Port-Based Sample Times

The hybrid method of assigning sample times combines the block-based and port-based methods. You first specify, in `mdlInitializeSizes`, the total number of rates at which your block operates, including both internal and input and output rates, using `ssSetNumSampleTimes`. You then set the `SS_OPTION_PORT_SAMPLE_TIMES_ASSIGNED`, using `ssSetOption`, to tell the simulation engine that you are going to use the port-based method to specify the rates of the input and output ports individually. Next, as in the block-based method, you specify the period and offset of all of the block's rates, both internal and external, using

```
ssSetSampleTime
ssSetOffsetTime
```


Finally, as in the port-based method, you specify the rates for each port, using

```
ssSetInputPortSampleTime(S, idx, period)
ssSetInputPortOffsetTime(S, idx, offset)
ssSetOutputPortSampleTime(S, idx, period)
ssSetOutputPortOffsetTime(S, idx, offset)
```

Note that each of the assigned port rates must be the same as one of the previously declared block rates.

Multirate S-Function Blocks

In a multirate S-Function block, you can encapsulate the code that defines each behavior in the mdl Output and mdl Update functions with a statement that determines whether a sample hit has occurred. The `ssSampleHit` macro determines whether the current time is a sample hit for a specified sample time. The macro has this syntax

```
ssSampleHit(S, st_index, tid)
```

where `S` is the `SimStruct`, `st_index` identifies a specific sample time index, and `tid` is the task ID (`tid` is an argument to the mdl Output and mdl Update).

For example, these statements specify three sample times: one for continuous behavior, and two for discrete behavior.

```
ssSetSampleTime(S, 0, CONTINUOUS_SAMPLE_TIME);
ssSetSampleTime(S, 1, 0.75);
ssSetSampleTime(S, 2, 1.0);
```

In the mdl Update function, the following statement would encapsulate the code that defines the behavior for the sample time of 0.75 second.

```
if (ssSampleHit(S, 1, tid)) {
}
```

The second argument, 1, corresponds to the second sample time, 0.75 second.

Example - Defining a Sample Time for a Continuous Block

This example defines a sample time for a block that is continuous in nature.

```
/* Initialize the sample time and offset. */
static void mdlInitializeSampleTimes(SimStruct *S)
{
```

```
    ssSetSampleTime(S, 0, CONTINUOUS_SAMPLE_TIME);  
    ssSetOffsetTime(S, 0, 0.0);  
}
```

You must add this statement to the mdlInitializeSizes function.

```
ssSetNumSampleTimes(S, 1);
```

Example - Defining a Sample Time for a Hybrid Block

This example defines sample times for a hybrid S-Function block.

```
/* Initialize the sample time and offset. */  
static void mdlInitializeSampleTimes(SimStruct *S)  
{  
    /* Continuous state sample time and offset. */  
    ssSetSampleTime(S, 0, CONTINUOUS_SAMPLE_TIME);  
    ssSetOffsetTime(S, 0, 0.0);  
  
    /* Discrete state sample time and offset. */  
    ssSetSampleTime(S, 1, 0.1);  
    ssSetOffsetTime(S, 1, 0.025);  
}
```

In the second sample time, the offset causes Simulink to call the mdlUpdate function at these times: 0.025 second, 0.125 second, 0.225 second, and so on, in increments of 0.1 second.

The following statement, which indicates how many sample times are defined, also appears in the mdlInitializeSizes function.

```
ssSetNumSampleTimes(S, 2);
```

Synchronizing Multirate S-Function Blocks

If tasks running at different rates need to share data, you must ensure that data generated by one task is valid when accessed by another task running at a different rate. You can use the `ssIsSpecialSampleHit` macro in the `mdlUpdate` or `mdlOutputs` routines of a multirate S-Function to ensure that the shared data is valid. This macro returns true if a sample hit has occurred at one rate and a sample hit has also occurred at another rate in the same time step. It thus permits a higher rate task to provide data needed by a slower rate task at a rate the slower task can accommodate.

Suppose, for example, that your model has an input port operating at one rate, 0, and an output port operating at a slower rate, 1. Further, suppose that you want the output port to output the value currently on the input. The following example illustrates usage of this macro.

```
if (ssI SampleHit(S, 0, tid) {
    if (ssISpecialSampleHit(S, 0, 1, tid) {
        /* Transfer input to output memory. */
        ...
    }
}

if (ssISampleHit(S, 1, tid) {
    /* Emit output. */
    ...
}
```

In this example, the first block runs when a sample hit occurs at the input rate. If the hit also occurs at the output rate, the block transfers the input to the output memory. The second block runs when a sample hit occurs at the output rate. It transfers the output in its memory area to the block's output.

Note that higher-rate tasks always run before slower-rate tasks. Thus, the input task in the preceding example always runs before the output task, ensuring that valid data is always present at the output port.

Work Vectors

If your S-function needs persistent memory storage, use S-function work vectors instead of static or global variables. If you use static or global variables, they are used by multiple instances of your S-function. This occurs when you have multiple S-Function blocks in a Simulink model and the same S-function C MEX-file has been specified. The ability to keep track of multiple instances of an S-function is called re-entrancy.

You can create an S-function that is re-entrant by using work vectors. These are persistent storage locations that Simulink manages for an S-function. Integer, floating point (real), pointer, and general data types are supported. The number of elements in each vector can be specified dynamically as a function of the number of inputs to the S-function.

Work vectors have several advantages:

- Instance specific storage for block variables
- Integer, real, pointer, and general data types
- Elimination of static and global variables and the associated multiple instance problems

For example, suppose you'd like to track the previous value of each input signal element entering input port 1 of your S-function. Either the discrete-state vector or the real-work vector could be used for this, depending upon whether the previous value is considered a discrete state (that is, compare the unit delay and the memory block). If you do not want the previous value to be logged when states are saved, use the real-work vector, `rwork`. To do this, in `mdlInitializeSizes` specify the length of this vector by using `ssSetNumRWork`. Then in either `mdlStart` or `mdlInitializeConditions`, initialize the `rwork` vector, `ssGetRWork`. In `mdlOutputs`, you can retrieve the previous inputs by using `ssGetRWork`. In `mdlUpdate`, update the previous value of the `rwork` vector by using `ssGetInputPortRealSignalPtrs`.

Use the macros in this table to specify the length of the work vectors for each instance of your S-function in `mdlInitializeSes`.

Table 7-1: Macros Used in Specifying Vector Widths

Macro	Description
<code>ssSetNumContStates</code>	Width of the continuous-state vector
<code>ssSetNumDiscreteStates</code>	Width of the discrete-state vector
<code>ssSetNumDWork</code>	Width of the data type work vector
<code>ssSetNumRWork</code>	Width of the real-work vector
<code>ssSetNumIWork</code>	Width of the integer-work vector
<code>ssSetNumPWork</code>	Width of the pointer-work vector
<code>ssSetNumModes</code>	Width of the mode-work vector
<code>ssSetNumNonsampledZCs</code>	Width of the nonsampled zero-crossing vector

Specify vector widths in `mdlInitializeSes`. There are three choices:

- 0 (the default). This indicates that the vector is not used by your S-function.
- A positive nonzero integer. This is the width of the vector that will be available for use by `mdlStart`, `mdlInitializeConditions`, and S-function routines called in the simulation loop.
- The `DYNAMICALLY_SIZED` define. The default behavior for dynamically sized vectors is to set them to the overall block width. Simulink does this after propagating line widths and sample times. The block width is the width of the signal passing through your block. In general this is equal to the output port width.

If the default behavior of dynamically sized vectors does not meet your needs, use `mdlSetWorkWidths` and the macros listed in Table 7-1, Macros Used in Specifying Vector Widths to set explicitly the sizes of the work vectors. Also, `mdlSetWorkWidths` allows you to set your work vector lengths as a function of the block sample time and/or port widths.

The continuous states are used when you have a state that needs to be integrated by one of Simulink's solvers. When you specify continuous states, you must return the states' derivatives in `mdlDerivatives`. The discrete state vector is used to maintain state information that changes at fixed intervals. Typically the discrete state vector is updated in place in `mdlUpdate`.

The integer, real and pointer work vectors are storage locations that do not get logged by Simulink during simulations. They maintain persistent data between calls to your S-function.

Work Vectors and Zero Crossings

The mode-work vector and the nonsampled zero-crossing vector are typically used with zero crossings. Elements of the mode vector are integer values. You specify the number of mode-vector elements in `mdlInitializeSizes` using `ssSetNumModes(S, num)`. You can then access the mode vector using `ssGetModeVector`. The mode vector is used to determine how the `mdlOutput` routine should operate when the solvers are honing in on zero crossings. The zero crossings or state events (i.e., discontinuities in the first derivatives) of some signal, usually a function of an input to your S-function, are tracked by the solver by looking at the nonsampled zero crossings. To register nonsampled zero crossings, set the number of nonsampled zero crossings in `mdlInitializeSizes` using `ssSetNumNonsampledZCs(S, num)`. Then, define the `mdlZeroCrossings` routine to return the nonsampled zero crossings. See matlabroot/simulink/src/sfun_zc.c for an example.

An Example Involving a Pointer Work Vector

This example opens a file and stores the FILE pointer in the pointer-work vector.

The statement below, included in the `mdlInitializeSizes` function, indicates that the pointer-work vector is to contain one element.

```
ssSetNumPWork(S, 1) /* pointer-work vector */
```

The code below uses the pointer-work vector to store a FILE pointer, returned from the standard I/O function, `fopen`.

```
#define MDL_START /* Change to #undef to remove function. */
#if defined(MDL_START)
static void mdlStart(real_T *x0, SimStruct *S)
```

```

{
    FILE *fPtr;
    void **PWork = ssGetPWork(S);
    fPtr = fopen("file.data", "r");
    PWork[0] = fPtr;
}
#endif /* MDL_START */

```

This code retrieves the FILE pointer from the pointer-work vector and passes it to fclose to close the file.

```

static void mdlTerminate(SimStruct *S)
{
    if (ssGetPWork(S) != NULL) {
        FILE *fPtr;
        fPtr = (FILE *) ssGetPWorkValue(S, 0);
        if (fPtr != NULL) {
            fclose(fPtr);
        }
        ssSetPWorkValue(S, 0, NULL);
    }
}

```

Note If you are using mdlSetWorkWidths, then any work vectors you use in your S-function should be set to DYNAMICALLY_SIZED in mdlInitializeSizes, even if the exact value is known before mdlInitializeSizes is called. The size to be used by the S-function should be specified in mdlSetWorkWidths.

The synopsis is

```

#define MDL_SET_WORK_WIDTHS /* Change to #undef to remove function. */
#if defined(MDL_SET_WORK_WIDTHS) && defined(MATLAB_MEX_FILE)
static void mdlSetWorkWidths(SimStruct *S)
{
}
#endif /* MDL_SET_WORK_WIDTHS */

```

For an example, see *matlabroot*/simulink/src/sfun_dynsize.c.

Memory Allocation

When creating an S-function, it is possible that the available work vectors don't provide enough capability. In this case, you will need to allocate memory for each instance of your S-function. The standard MATLAB API memory allocation routines (`mxMalloc`, `mxFree`) should not be used with C MEX S-functions. The reason is that these routines are designed to be used with MEX-files that are called from MATLAB and not Simulink. The correct approach for allocating memory is to use the `stdlib.h` (`malloc`, `free`) library routines. In `mdlStart` allocate and initialize the memory and place the pointer to it either in pointer-work vector elements

```
ssGetPWork(S)[i] = ptr;
```

or attach it as user data.

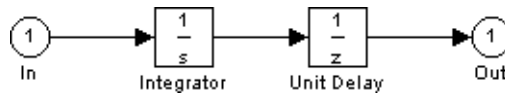
```
ssSetUserData(S, ptr);
```

In `mdlTerminate`, free the allocated memory.

Example - Hybrid System S-Functions

The S-function, `matlabroot/simulink/src/mixedm.c`, is an example of a hybrid (a combination of continuous and discrete states) system. `mixedm.c` combines elements of `csfunc.c` and `dsfunc.c`. If you have a hybrid system, place your continuous equations in `mdlDerivative` and your discrete equations in `mdlUpdate`. In addition, you need to check for sample hits to determine at what point your S-function is being called.

In Simulink block diagram form, the S-function, `mixedm.c` looks like



which implements a continuous integrator followed by a discrete unit delay.

Since there are no tasks to complete at termination, `mdlTerminate` is an empty function. `mdlDerivatives` calculates the derivatives of the continuous states of the state vector x , and `mdlUpdate` contains the equations used to update the discrete state vector, x .

`matlabroot/simulink/src/mixedm.c`

```

/* File      : mixedm.c
 * Abstract:
 *
 *      An example C MEX S-function that implements a continuous integrator (1/s)
 *      in series with a unit delay (1/z)
 *
 *      For more details about S-functions, see simulink/src/sfuntmpl.doc.
 *
 * Copyright (c) 1990-1998 by The MathWorks, Inc. All Rights Reserved.
 * $Revision: 1.4 $
 */
#define S_FUNCTION_NAME mixedm
#define S_FUNCTION_LEVEL 2

#include "simstruc.h"

#define U(element) (*uPtrs[element]) /* Pointer to Input Port0 */

/*=====
 * S-function routines *
 *=====*/

/* Function: mdlInitializeSizes =====
 * Abstract:

```

```

*   The sizes information is used by Simulink to determine the S-Function
*   block's characteristics (number of inputs, outputs, states, etc.).
*/
static void mdlInitializeSizes(SimStruct *S)
{
    ssSetNumSFcnParams(S, 0); /* Number of expected parameters */
    if (ssGetNumSFcnParams(S) != ssGetSFcnParamsCount(S)) {
        return; /* Parameter mismatch will be reported by Simulink */
    }

    ssSetNumContStates(S, 1);
    ssSetNumDiscStates(S, 1);

    if (!ssSetNumInputPorts(S, 1)) return;
    ssSetInputPortWidth(S, 0, 1);
    ssSetInputPortDirectFeedThrough(S, 0, 1);

    if (!ssSetNumOutputPorts(S, 1)) return;
    ssSetOutputPortWidth(S, 0, 1);

    ssSetNumSampleTimes(S, 2);
    ssSetNumRWork(S, 0);
    ssSetNumIWork(S, 0);
    ssSetNumPWork(S, 0);
    ssSetNumModes(S, 0);
    ssSetNumNonsampledZCs(S, 0);

    /* Take care when specifying exception free code - see sfuntmpl.doc. */
    ssSetOptions(S, SS_OPTION_EXCEPTION_FREE_CODE);
}

/* Function: mdlInitializeSampleTimes =====
* Abstract:
*   Two tasks: One continuous, one with discrete sample time of 1.0
*/
static void mdlInitializeSampleTimes(SimStruct *S)
{
    ssSetSampleTime(S, 0, CONTINUOUS_SAMPLE_TIME);
    ssSetSampleTime(S, 1, 1.0);

    ssSetOffsetTime(S, 0, 0.0);
    ssSetOffsetTime(S, 1, 0.0);
}

#define MDL_INITIALIZE_CONDITIONS
/* Function: mdlInitializeConditions =====
* Abstract:
*   Initialize both continuous states to zero.
*/
static void mdlInitializeConditions(SimStruct *S)
{
    real_T *xC0 = ssGetContStates(S);
    real_T *xD0 = ssGetRealDiscStates(S);

```

```

        xCO[0] = 1.0;
        xDO[0] = 1.0;
    }

/* Function: mdlOutputs =====
 * Abstract:
 *     y = xD
 */
static void mdlOutputs(SimStruct *S, int_T tid)
{
    real_T *y = ssGetOutputPortRealSignal(S, 0);
    real_T *xD = ssGetRealDiscreteStates(S);

    /* y=xD */
    if (ssIsSampleHit(S, 1, tid)) {
        y[0]=xD[0];
    }
}

#define MDL_UPDATE
/* Function: mdlUpdate =====
 * Abstract:
 *     xD = xC
 */
static void mdlUpdate(SimStruct *S, int_T tid)
{
    real_T *xD = ssGetRealDiscreteStates(S);
    real_T *xC = ssGetContStates(S);

    /* xD=xC */
    if (ssIsSampleHit(S, 1, tid)) {
        xD[0]=xC[0];
    }
}

#define MDL_DERIVATIVES
/* Function: mdlDerivatives =====
 * Abstract:
 *     xdot = U
 */
static void mdlDerivatives(SimStruct *S)
{
    real_T *dx = ssGetdX(S);
    InputRealPtrsType uPtrs = ssGetInputPortRealSignalPtrs(S, 0);

    /* xdot=U */
    dx[0]=U(0);
}

/* Function: mdlTerminate =====
 * Abstract:
 *     No termination needed, but we are required to have this routine.

```



```

*/
static void mdlTerminate(SimStruct *S)
{
}

#ifdef MATLAB_MEX_FILE /* Is this file being compiled as a MEX-file? */
#include "simulink.c" /* MEX-file interface mechanism */
#else
#include "cg_sfun.h" /* Code generation registration function */
#endif

```

Example - Variable Step S-Function

The example S-function, `vsfunc.c` uses a variable step sample time. Variable step-size functions require a call to `mdlGetTimeOfNextVarHit`, which is an S-function routine that calculates the time of the next sample hit. S-functions that use the variable step sample time can only be used with variable step solvers. `vsfunc` is a discrete S-function that delays its first input by an amount of time determined by the second input.

This example demonstrates how to correctly work with the fixed and variable step solvers when the equations (functions) that are being integrated change during the simulation. In the transfer function used in this example, the parameters of the transfer function vary with time.

The output of `vsfunc` is simply the input `u` delayed by a variable amount of time. `mdlOutputs` sets the output `y` equal to state `x`. `mdlUpdate` sets the state vector `x` equal to `u`, the input vector. This example calls `mdlGetTimeOfNextVarHit`, an S-function routine that calculates and sets the "time of next hit," that is, the time when `vsfunc` is next called. In `mdlGetTimeOfNextVarHit` the macro `ssGetU` is used to get a pointer to the input `u`. Then this call is made.

```
ssSetTNext(S, ssGetT(S)(*u[1]));
```

The macro `ssGetT` gets the simulation time `t`. The second input to the block, `(*u[1])`, is added to `t`, and the macro `ssSetTNext` sets the time of next hit equal to `t+(*u[1])`, delaying the output by the amount of time set in `(*u[1])`.

`matlabroot/simulink/src/vsfunc.c`

```

/* File      : vsfunc.c
 * Abstract:
 *
 *          Example C-file S-function for defining a continuous system.
 *
 */

```

```

*      Variable step S-function example.
*      This example S-function illustrates how to create a variable step
*      block in Simulink. This block implements a variable step delay
*      in which the first input is delayed by an amount of time determined
*      by the second input:
*
*      dt      = u(2)
*      y(t+dt) = u(t)
*
*      For more details about S-functions, see simulink/src/sfuntmpl.doc.
*
*      Copyright (c) 1990-1998 by The MathWorks, Inc. All Rights Reserved.
*      $Revision: 1.6 $
*/

#define S_FUNCTION_NAME vsfunc
#define S_FUNCTION_LEVEL 2

#include "simstruc.h"

#define U(element) (*uPtrs[element]) /* Pointer to Input Port0 */

/* Function: mdlInitializeSizes =====
* Abstract:
*   The sizes information is used by Simulink to determine the S-function
*   block's characteristics (number of inputs, outputs, states, etc.).
*/
static void mdlInitializeSizes(SimStruct *S)
{
    ssSetNumSFcnParams(S, 0); /* Number of expected parameters */
    if (ssGetNumSFcnParams(S) != ssGetSFcnParamsCount(S)) {
        return; /* Parameter mismatch will be reported by Simulink */
    }

    ssSetNumContStates(S, 0);
    ssSetNumDiscStates(S, 1);

    if (!ssSetNumInputPorts(S, 1)) return;
    ssSetInputPortWidth(S, 0, 2);
    ssSetInputPortDirectFeedThrough(S, 0, 0);

    if (!ssSetNumOutputPorts(S, 1)) return;
    ssSetOutputPortWidth(S, 0, 1);

    ssSetNumSampleTimes(S, 1);
    ssSetNumRWork(S, 0);
    ssSetNumIWork(S, 0);
    ssSetNumPWork(S, 0);
    ssSetNumModes(S, 0);
    ssSetNumNonsampledZCs(S, 0);

```

```

    /* Take care when specifying exception free code - see sfuntmpl.doc */
    ssSetOptions(S, SS_OPTION_EXCEPTION_FREE_CODE);
}

/* Function: mdlInitializeSampleTimes =====
 * Abstract:
 *   Variable-Step S-function
 */
static void mdlInitializeSampleTimes(SimStruct *S)
{
    ssSetSampleTime(S, 0, VARIABLE_SAMPLE_TIME);
    ssSetOffsetTime(S, 0, 0.0);
}

#define MDL_INITIALIZE_CONDITIONS
/* Function: mdlInitializeConditions =====
 * Abstract:
 *   Initialize discrete state to zero.
 */
static void mdlInitializeConditions(SimStruct *S)
{
    real_T *x0 = ssGetRealDiscreteStates(S);

    x0[0] = 0.0;
}

#define MDL_GET_TIME_OF_NEXT_VAR_HIT
static void mdlGetTimeOfNextVarHit(SimStruct *S)
{
    InputRealPtrsType uPtrs = ssGetInputPortRealSignalPtrs(S, 0);

    /* Make sure input will increase time */
    if (U(1) <= 0.0) {
        /* If not, abort simulation */
        ssSetErrorStatus(S, "Variable step control input must be "
            "greater than zero");
        return;
    }
    ssSetTNext(S, ssGetT(S)+U(1));
}
/* Function: mdlOutputs =====
 * Abstract:
 *   y = x
 */
static void mdlOutputs(SimStruct *S, int_T tid)
{
    real_T *y = ssGetOutputPortRealSignal(S, 0);

```

```

    real_T *x = ssGetRealDiscreteStates(S);

    /* Return the current state as the output */
    y[0] = x[0];
}

#define MDL_UPDATE
/* Function: mdlUpdate =====
 * Abstract:
 * This function is called once for every major integration time step.
 * Discrete states are typically updated here, but this function is useful
 * for performing any tasks that should only take place once per integration
 * step.
 */
static void mdlUpdate(SimStruct *S, int_T tid)
{
    real_T *x = ssGetRealDiscreteStates(S);
    InputRealPtrsType uPtrs = ssGetInputPortRealSignalPtrs(S, 0);

    x[0]=U(0);
}

/* Function: mdlTerminate =====
 * Abstract:
 * No termination needed, but we are required to have this routine.
 */
static void mdlTerminate(SimStruct *S)
{
}

#ifdef MATLAB_MEX_FILE /* Is this file being compiled as a MEX-file? */
#include "simulink.c" /* MEX-file interface mechanism */
#else
#include "cg_sfun.h" /* Code generation registration function */
#endif

```

Example - Zero Crossing S-Function

The example S-function, `sfun_zc_sat` demonstrates how to implement a saturation block. This S-function is designed to work with either fixed or variable step solvers. When this S-function inherits a continuous sample time, and a variable step solver is being used, a zero crossings algorithm is used to locate the exact points at which the saturation occurs.

matlabroot/simulink/src/sfun_zc_sat.c

```

/* File      : sfun_zc_sat.c
 * Abstract:
 *
 *      Example of an S-function that has nonsampled zero crossings to
 *      implement a saturation function. This S-function is designed to be
 *      used with a variable or fixed step solver.
 *
 *      A saturation is described by three equations
 *
 *      (1)    y = UpperLimit
 *      (2)    y = u
 *      (3)    y = LowerLimit
 *
 *      and a set of inequalities that specify which equation to use
 *
 *      if                UpperLimit < u      then use (1)
 *      if    LowerLimit <= u <= UpperLimit    then use (2)
 *      if    u < LowerLimit                    then use (3)
 *
 *      A key fact is that the valid equation 1, 2, or 3, can change at
 *      any instant. Nonsampled zero crossing (ZC) support helps the variable step
 *      solvers locate the exact instants when behavior switches from one equation
 *      to another.
 *
 *      Copyright (c) 1990-1998 by The MathWorks, Inc. All Rights Reserved.
 *      $Revision: 1.5 $
 */

#define S_FUNCTION_NAME  sfun_zc_sat
#define S_FUNCTION_LEVEL 2

#include "tmwtypes.h"
#include "simstruc.h"
#ifdef MATLAB_MEX_FILE
#include "mex.h"
#endif

/*=====
 * General Defines/macros *
 *=====*/

/* index to Upper Limit */
#define I_PAR_UPPER_LIMIT 0

/* index to Lower Limit */
#define I_PAR_LOWER_LIMIT 1

/* total number of block parameters */
#define N_PAR                2

```



```

*   if u < LowerLimit           then use (3)
*
*   The mode vector is changed only at the beginning of a MajorTimeStep.
*
*   During a minor time step, the equation specified by the mode vector
*   is used without question. Most of the time, the value of u will agree
*   with the equation specified by the mode vector. However, sometimes u's
*   value will indicate a different equation. Nonetheless, the equation
*   specified by the mode vector must be used.
*
*   When the mode and u indicate different equations, the corresponding
*   calculations are not correct. However, this is not a problem. From
*   the ZC function, the solver will know that an equation switch occurred
*   in the middle of the last MajorTimeStep. The calculations for that
*   time step will be discarded. The ZC function will help the solver
*   find the exact instant at which the switch occurred. Using this knowledge,
*   the length of the MajorTimeStep will be reduced so that only one equation
*   is valid throughout the entire time step.
*/
static void mdlOutputs(SimStruct *S, int_T tid)
{
    InputRealPtrsType uPtrs    = ssGetInputPortRealSignalPtrs(S,0);
    real_T            *y        = ssGetOutputPortRealSignal(S,0);
    int_T             numOutput = ssGetOutputPortWidth(S,0);
    int_T             iOutput;

    /*
     * Set index and increment for input signal, upper limit, and lower limit
     * parameters so that each gives scalar expansion if needed.
     */
    int_T uldx          = 0;
    int_T ulnc          = ( ssGetInputPortWidth(S,0) > 1 );
    real_T *upperLimit  = mxGetPr( P_PAR_UPPER_LIMIT );
    int_T upperLimitInc = ( mxGetNumberOfElements( P_PAR_UPPER_LIMIT ) > 1 );
    real_T *lowerLimit  = mxGetPr( P_PAR_LOWER_LIMIT );
    int_T lowerLimitInc = ( mxGetNumberOfElements( P_PAR_LOWER_LIMIT ) > 1 );

    if (ssGetNumNonsampledZCs(S) == 0) {
        /*
         * This block is being used with a fixed-step solver or it has
         * a noncontinuous sample time, so we always saturate.
         */
        for (iOutput = 0; iOutput < numOutput; iOutput++) {
            if (*uPtrs[uldx] >= *upperLimit) {
                *y++ = *upperLimit;
            } else if (*uPtrs[uldx] > *lowerLimit) {
                *y++ = *uPtrs[uldx];
            } else {
                *y++ = *lowerLimit;
            }

            upperLimit += upperLimitInc;
            lowerLimit += lowerLimitInc;
        }
    }
}

```

```

        uldx      += ulnc;
    }
} else {
    /*
    * This block is being used with a variable-step solver.
    */
    int_T *mode = ssGetModeVector(S);

    /*
    * Specify indices for each equation.
    */
    enum { UpperLimitEquation, NonLimitEquation, LowerLimitEquation };

    /*
    * Update the mode vector ONLY at the beginning of a MajorTimeStep.
    */
    if ( ssIsMajorTimeStep(S) ) {
        /*
        * Specify the mode, that is, the valid equation for each output scalar.
        */
        for ( iOutput = 0; iOutput < numOutput; iOutput++ ) {
            if ( *uPtrs[uldx] > *upperLimit ) {
                /*
                * Upper limit eq is valid.
                */
                mode[iOutput] = UpperLimitEquation;
            } else if ( *uPtrs[uldx] < *lowerLimit ) {
                /*
                * Lower limit eq is valid.
                */
                mode[iOutput] = LowerLimitEquation;
            } else {
                /*
                * Nonlimit eq is valid.
                */
                mode[iOutput] = NonLimitEquation;
            }
        }
        /*
        * Adjust indices to give scalar expansion if needed.
        */
        uldx      += ulnc;
        upperLimit += upperLimitInc;
        lowerLimit += lowerLimitInc;
    }

    /*
    * Reset index to input and limits.
    */
    uldx      = 0;
    upperLimit = mxGetPr( P_PAR_UPPER_LIMIT );
    lowerLimit = mxGetPr( P_PAR_LOWER_LIMIT );
}

```

```

} /* end IsMajorTimeStep */

/*
 * For both MinorTimeSteps and MajorTimeSteps calculate each scalar
 * output using the equation specified by the mode vector.
 */
for ( iOutput = 0; iOutput < numOutput; iOutput++ ) {
    if ( mode[iOutput] == UpperLimitEquation ) {
        /*
         * Upper limit eq.
         */
        *y++ = *upperLimit;
    } else if ( mode[iOutput] == LowerLimitEquation ) {
        /*
         * Lower limit eq.
         */
        *y++ = *lowerLimit;
    } else {
        /*
         * Nonlimit eq.
         */
        *y++ = *uPtrs[uldx];
    }

    /*
     * Adjust indices to give scalar expansion if needed.
     */
    uldx      += ulnc;
    upperLimit += upperLimitInc;
    lowerLimit += lowerLimitInc;
}
} /* end mdlOutputs */

```

```

#define MDL_ZERO_CROSSINGS
#if defined(MDL_ZERO_CROSSINGS) && (defined(MATLAB_MEX_FILE) || defined(NRT))

/* Function: mdlZeroCrossings =====
 * Abstract:
 * This will only be called if the number of nonsampled zero crossings is
 * greater than 0, which means this block has a continuous sample time and the
 * the model is using a variable step solver.
 *
 * Calculate ZC signals that help the solver find the
 * exact instants at which equation switches occur:

```

```

*
*   if                UpperLimit < u   then use (1)
*   if   LowerLimit <= u <= UpperLimit then use (2)
*   if   u < LowerLimit                 then use (3)
*
* The key words are help find. There is no choice of a function that will
* direct the solver to the exact instant of the change. The solver will
* track the zero crossing signal and do a bisection style search for the
* exact instant of equation switch.
*
* There is generally one ZC signal for each pair of signals that can
* switch. The three equations above would break into two pairs (1)&(2)
* and (2)&(3). The possibility of a "long jump" from (1) to (3) does
* not need to be handled as a separate case. It is implicitly handled.
*
* When a ZCs are calculated, the value is normally used twice. When it is
* first calculated, it is used as the end of the current time step. Later,
* it will be used as the beginning of the following step.
*
* The sign of the ZC signal always indicates an equation from the pair. In the
* context of S-functions, which equation is associated with a positive ZC and
* which is associated with a negative ZC doesn't really matter. If the ZC is
* positive at the beginning and at the end of the time step, this implies that the
* positive equation was valid throughout the time step. Likewise, if the
* ZC is negative at the beginning and at the end of the time step, this
* implies that the negative equation was valid throughout the time step.
* Like any other nonlinear solver, this is not fool proof, but it is an
* excellent indicator. If the ZC has a different sign at the beginning and
* at the end of the time step, then a equation switch definitely occurred
* during the time step.
*
* Ideally, the ZC signal gives an estimate of when an equation switch
* occurred. For example, if the ZC signal is -2 at the beginning and +6 at
* the end, then this suggests that the switch occurred
* 25% = 100%*(-2)/(-2-(+6)) of the way into the time step. It will almost
* never be true that 25% is perfectly correct. There is no perfect choice
* for a ZC signal, but there are some good rules. First, choose the ZC
* signal to be continuous. Second, choose the ZC signal to give a monotonic
* measure of the "distance" to a signal switch; strictly monotonic is ideal.
*/
static void mdlZeroCrossings(SimStruct *S)
{
    int_T          iOutput;
    int_T          numOutput = ssGetOutputPortWidth(S, 0);
    real_T         *zcSignals = ssGetNonsampledZCs(S);
    InputRealPtrsType uPtrs    = ssGetInputPortRealSignalPtrs(S, 0);

    /*
     * Set index and increment for the input signal, upper limit, and lower
     * limit parameters so that each gives scalar expansion if needed.
     */
    int_T uldx      = 0;
    int_T ulnc      = ( ssGetInputPortWidth(S, 0) > 1 );

```

```

real_T *upperLimit = mxGetPr( P_PAR_UPPER_LIMIT );
int_T upperLimitInc = ( mxGetNumberOfElements( P_PAR_UPPER_LIMIT ) > 1 );
real_T *lowerLimit = mxGetPr( P_PAR_LOWER_LIMIT );
int_T lowerLimitInc = ( mxGetNumberOfElements( P_PAR_LOWER_LIMIT ) > 1 );

/*
 * For each output scalar, give the solver a measure of "how close things
 * are" to an equation switch.
 */
for ( iOutput = 0; iOutput < numOutput; iOutput++ ) {

    /* The switch from eq (1) to eq (2)
     *
     * if          UpperLimit < u      then use (1)
     * if    LowerLimit <= u <= UpperLimit      then use (2)
     *
     * is related to how close u is to UpperLimit. A ZC choice
     * that is continuous, strictly monotonic, and is
     *   u - UpperLimit
     * or it is negative.
     */
    zcSignals[2*iOutput] = *uPtrs[uldx] - *upperLimit;

    /* The switch from eq (2) to eq (3)
     *
     * if    LowerLimit <= u <= UpperLimit      then use (2)
     * if    u < LowerLimit                      then use (3)
     *
     * is related to how close u is to LowerLimit. A ZC choice
     * that is continuous, strictly monotonic, and is
     *   u - LowerLimit.
     */
    zcSignals[2*iOutput+1] = *uPtrs[uldx] - *lowerLimit;

    /*
     * Adjust indices to give scalar expansion if needed.
     */
    uidx += ulnc;
    upperLimit += upperLimitInc;
    lowerLimit += lowerLimitInc;
}
}

#endif /* end mdlZeroCrossings */

/* Function: mdlTerminate =====
 * Abstract:
 * No termination needed, but we are required to have this routine.
 */
static void mdlTerminate(SimStruct *S)
{
}

```



```

int_T          nContStates = ssGetNumContStates(S);
int_T          nCoeffs    = nContStates + 1;
int_T          bankToUpdate = !ssGetIWork(S)[0];
real_T         *num        = ssGetRWork(S)+bankToUpdate*2*nCoeffs;
real_T         *den        = num + nCoeffs;

real_T         den0;
int_T          allZero;

/*
 * Get the first denominator coefficient. It will be used
 * for normalizing the numerator and denominator coefficients.
 *
 * If all inputs are zero, we probably could have unconnected
 * inputs, so use the parameter as the first denominator coefficient.
 */
den0 = *uPtrs[uldx+nCoeffs];
if (den0 == 0.0) {
    den0 = mxGetPr(DEN(S))[0];
}

/*
 * Grab the numerator.
 */
allZero = 1;
for (i = 0; (i < nCoeffs) && allZero; i++) {
    allZero &= *uPtrs[uldx+i] == 0.0;
}

if (allZero) { /* if numerator is all zero */
    const real_T *numParam = mxGetPr(NUM(S));
    int_T        numParamLen = mxGetNumberOfElements(NUM(S));
    /*
     * Move the input to the denominator input and
     * get the denominator from the input parameter.
     */
    uldx += nCoeffs;
    num += nCoeffs - numParamLen;
    for (i = 0; i < numParamLen; i++) {
        *num++ = *numParam++ / den0;
    }
} else {
    for (i = 0; i < nCoeffs; i++) {
        *num++ = *uPtrs[uldx++] / den0;
    }
}

/*
 * Grab the denominator.
 */
allZero = 1;
for (i = 0; (i < nCoeffs) && allZero; i++) {

```


Place the call to
my_alg in
mdlOutputs.

```

/*
 * mdlInitializeSampleTimes - indicate that this S-function runs
 * at the rate of the source (driving block)
 */
static void mdlInitializeSampleTimes(SimStruct *S)
{
    ssSetSampleTime(S, 0, INHERITED_SAMPLE_TIME);
    ssSetOffsetTime(S, 0, 0.0);
}

/*
 * mdlOutputs - compute the outputs by calling my_alg, which
 * resides in another module, my_alg.c
 */
static void mdlOutputs(SimStruct *S, int_T tid)
{
    InputRealPtrsType uPtrs = ssGetInputPortRealSignalPtrs(S, 0);
    real_T *y = ssGetOutputPortRealSignal(S, 0);
    *y = my_alg(*uPtrs[0]);
}
/*
 * mdlTerminate - called when the simulation is terminated.
 */
static void mdlTerminate(SimStruct *S)
{
}

#ifdef MATLAB_MEX_FILE /* Is this file being compiled as a MEX-file? */
#include "simulink.c" /* MEX-file interface mechanism */
#else
#include "cg_sfun.h" /* Code generation registration function */
#endif

```

The S-function routine mdlOutputs contains a function call to my_alg, which is the C function that contains the algorithm that the S-function performs. This is the code for my_alg.c.

```

#include "tmwtypes.h"
real_T my_alg(real_T u)
{
    return(u * 2.0);
}

```

The wrapper S-function (wrapsfcn) calls my_alg, which computes $u * 2.0$. To build wrapsfcn.mex, use the following command.

```
mex wrapsfcn.c my_alg.c
```

The TLC S-Function Wrapper

This section describes how to inline the call to `my_alg` in the Mdl Outputs section of the generated code. In the above example, the call to `my_alg` is embedded in the mdl Outputs section as

```
*y = my_alg(*uPtrs[0]);
```

When creating a TLC S-function wrapper, the goal is to have the Real-Time Workshop embed the same type of call in the generated code.

It is instructive to look at how the Real-Time Workshop executes S-functions that are not inlined. A noninlined S-function is identified by the absence of the file `sfuncti on. tlc` and the existence of `sfuncti on. mex`. When generating code for a noninlined S-function, the Real-Time Workshop generates a call to mdl Outputs through a function pointer that, in this example, then calls `my_alg`.

The wrapper example contains one S-function (`wrapsfcn. mex`). You must compile and link an additional module, `my_alg`, with the generated code. To do this, specify

```
set_param(' wrapper/S-Functi on' , ' SFuncti onModul es' , ' my_alg' )
```

The code generated when using `grt. tlc` as the system target file without `wrapsfcn. tlc` is

```
<Generated code comments for wrapper model wi th noninlined wrapsfcn S-functi on>

#include <math.h>
#include <string.h>
#include "wrapper.h"
#include "wrapper.prm"

/* Start the model */
void Mdl Start(void)
{
    /* (no start code requi red) */
}

/* Compute block outputs */
void Mdl Outputs(int_T tid)
{
    /* Sin Block: <Root>/Sin */
    rtB.Sin = rtP.Sin.Ampl i tude *
        sin(rtP.Sin.Frequency * ssGetT(rtS) + rtP.Sin.Phase);
```


This is the code that is inlined for the bottom S-function block in the `sfun_directlook_ex` model.

```

rtY.Out1 = rtb_buffer2;

/* S-Function Block: <Root>/S-Function1 */
{
  real_T *xData = &rtP.S_Function1.XData[0];
  real_T *yData = &rtP.S_Function1.YData[0];
  int_T idx;

  idx = GetDirectLookupIndex(xData, 5, rtb_Sine_Wave);
  rtb_buffer2 = yData[idx];
}

/* Output Block: <Root>/Out2 */
rtY.Out2 = rtb_buffer2;
}

/* Perform model update */
void MdlUpdate(int_T tid)
{
  /* (no update code required) */
}

/* Terminate function */
void MdlTerminate(void)
{
  /* (no terminate code required) */
}

#include "sfun_directlook_ex.reg"

/* [EOF] sfun_directlook_ex.c */

```

matlabroot/simulink/src/sfun_directlook.c

```

/*
 * File      : sfun_directlook.c
 * Abstract:
 *
 * Direct 1-D lookup. Here we are trying to compute an approximate
 * solution, p(x) to an unknown function f(x) at x=x0, given data point
 * pairs (x,y) in the form of a x data vector and a y data vector. For a
 * given data pair (say the i'th pair), we have y_i = f(x_i). It is
 * assumed that the x data values are monotonically increasing. If the
 * x0 is outside of the range of the x data vector, then the first or
 * last point will be returned.
 *
 * This function returns the "nearest" y0 point for a given x0. No
 * interpolation is performed.
 *
 * The S-function parameters are:
 * XData      - double vector
 * YData      - double vector
 */

```


S-Function Callback Methods

Callback Method Reference	9-2
mdlCheckParameters	9-3
mdlDerivatives	9-5
mdlGetTimeOfNextVarHit	9-6
mdlInitializeConditions	9-7
mdlInitializeSampleTimes	9-9
mdlInitializeSizes	9-13
mdlOutputs	9-17
mdlProcessParameters	9-18
mdlSetDefaultPortComplexSignals	9-21
mdlSetDefaultPortDataTypes	9-22
mdlSetDefaultPortDimensionInfo	9-23
mdlSetInputPortComplexSignal	9-24
mdlSetInputPortDataType	9-25
mdlSetInputPortDimensionInfo	9-26
mdlSetInputPortFrameData	9-28
mdlSetInputPortSampleTime	9-29
mdlSetInputPortWidth	9-31
mdlSetOutputPortComplexSignal	9-32
mdlSetOutputPortDataType	9-33
mdlSetOutputPortDimensionInfo	9-34
mdlSetOutputPortSampleTime	9-36
mdlSetOutputPortWidth	9-37
mdlSetWorkWidths	9-38
mdlStart	9-39
mdlTerminate	9-40
mdlUpdate	9-41
mdlZeroCrossings	9-42

Callback Method Reference

Every user-written S-function must implement a set of methods, called callback methods or simply callbacks, that Simulink invokes when simulating a model that contains the S-function. Some callback methods are optional. Simulink invokes an optional callback only if the S-function defines the callback. This section describes the purpose and syntax of all callback methods that an S-function can implement. In each case, the documentation for a callback method indicates whether it is required or optional.

Purpose	Check the validity of an S-function's parameters.
Syntax	<code>void mdlCheckParameters(SimStruct *S)</code>
Arguments	S Simstruct representing an S-function block.
Description	Verifies new parameter settings whenever parameters change or are re-evaluated during a simulation.

When a simulation is running, changes to S-function parameters can occur at any time during the simulation loop; that is, either at the start of a simulation step or during a simulation step. When the change occurs during a simulation step, Simulink calls this routine twice to handle the parameter change. The first call during the simulation step is used to verify that the parameters are correct. After verifying the new parameters, the simulation continues using the original parameter values until the next simulation step at which time the new parameter values will be used. Redundant calls are needed to maintain simulation consistency.

Note You cannot access the work, state, input, output, and other vectors in this routine. Use this routine only to validate the parameters. Additional processing of the parameters should be done in `mdlProcessParameters`.

Example This example checks the first S-function parameter to verify that it is a real nonnegative scalar.

```
#define PARAM1(S) ssGetSFcnParam(S,0)
#define MDL_CHECK_PARAMETERS /* Change to #undef to remove function */
#if defined(MDL_CHECK_PARAMETERS) && defined(MATLAB_MEX_FILE)
static void mdlCheckParameters(SimStruct *S)
{
    if (mxGetNumberOfElements(PARAM1(S)) != 1) {
        ssSetErrorStatus(S, "Parameter to S-function must be a scalar");
        return;
    } else if (mxGetPr(PARAM1(S))[0] < 0) {
        ssSetErrorStatus(S, "Parameter to S-function must be non-negative");
        return;
    }
}
#endif /* MDL_CHECK_PARAMETERS */
```

mdlCheckParameters

In addition to the above routine, you must add a call to this routine from `mdlInitializeSIZES` to check parameters during initialization since `mdlCheckParameters` is only called while the simulation is running. To do this, in `mdlInitializeSIZES`, after setting the number of parameters you expect in your S-function by using `ssSetNumSFcnParams`, use this code:

```
static void mdlInitializeSIZES(SimStruct *S)
{
    ssSetNumSFcnParams(S, 1); /* Number of expected parameters */
#ifdef MATLAB_MEX_FILE
    if(ssGetNumSFcnParams(s) == ssGetSFcnParamsCount(s) {
        mdlCheckParameters(S);
        if(ssGetErrorStates(S) != NULL) return;
    } else {
        return; /* Simulink will report a mismatch error. */
    }
#endif
    ...
}
```

Note The macro `ssGetSFcnParamsCount` returns the actual number of parameters entered in the dialog box.

See *matlabroot*/simulink/src/sfun_errhdl.c for an example.

Languages Ada, C

See Also `mdlProcessParameters`, `ssGetSFcnParamsCount`

Purpose	Compute the S-function's derivatives.
Syntax	<code>void mdlDerivatives(SimStruct *S)</code>
Arguments	S Simstruct representing an S-function block.
Description	<p>Simulink invokes this optional method at each time step to compute the derivatives of the S-function's continuous states. This method should store the derivatives in the S-function's state derivatives vector. This method can use <code>ssGetdX</code> to get a pointer to the derivatives vector.</p> <p>Each time the <code>mdlDerivatives</code> routine is called, it must explicitly set the value of all derivatives. The derivative vector does not maintain the values from the last call to this routine. The memory allocated to the derivative vector changes during execution.</p>
Example	For an example, see <i>matlabroot</i> /simulink/src/csfunc.c.
Languages	Ada, C, M
See Also	<code>ssGetdx</code>

mdlGetTimeOfNextVarHit

Purpose Initialize the state vectors of this S-function.

Syntax `void mdlGetTimeOfNextVarHit(SimStruct *S)`

Arguments S
Simstruct representing an S-function block.

Description Simulink invokes this optional method at every major integration step to get the time of the next sample time hit. This method should set the time of next hit, using `ssSetTNext`. The time of the next hit must be greater than the current simulation time as returned by `ssGetT`. The S-function must implement this method if it operates at a discrete, variable-step sample time.

Note The time of next hit can be a function of the input signal(s).

Languages C, M

Example

```
static void mdlGetTimeOfNextVarHit(SimStruct *S)
{
    time_T offset = getOffset();
    time_T timeOfNextHit = ssGetT(S) + offset;
    ssSetTNext(S, timeOfNextHit);
}
```

See Also `mdlInitializeSampleTimes`, `ssSetTNext`, `ssGetT`

Purpose	Initialize the state vectors of this S-function.
Syntax	<code>void mdlInitializeConditions(SimStruct *S)</code>
Arguments	S Simstruct representing an S-function block.
Description	<p>Simulink invokes this optional method at the beginning of a simulation. It should initialize the continuous and discrete states, if any, of this S-function block. Use <code>ssGetContStates</code> and/or <code>ssGetDi scStates</code> to get the states. This method can also perform any other initialization activities that this S-function requires.</p> <p>If this S-function resides in an enabled subsystem configured to reset states, Simulink also calls this method when the enabled subsystem restarts execution. This method can use <code>ssIsFi rstI ni tCond</code> macro to determine if it is being called for the first time.</p>

Example This example is an S-function with both continuous and discrete states; it initializes both sets of states to 1.0:

```
#define MDL_INITIALIZE_CONDITIONS /* Change to #undef to remove function */
#if defined(MDL_INITIALIZE_CONDITIONS)

static void mdlInitializeConditions(SimStruct *S)
{
    int i;
    real_T *xcont = ssGetContStates(S);
    int_T nCStates = ssGetNumContStates(S);
    real_T *xdisc = ssGetRealDi scStates(S);
    int_T nDStates = ssGetNumDi scStates(S);

    for (i = 0; i < nCStates; i++) {
        *xcont++ = 1.0;
    }

    for (i = 0; i < nDStates; i++) {
        *xdisc++ = 1.0;
    }

}
#endif /* MDL_INITIALIZE_CONDITIONS */
```

For another example which initializes only the continuous states, see <matlabroot/simulink/src/resetint.c>.

mdlInitializeConditions

Languages C

See Also mdlStart, sslsFirstInitCond, ssGetContStates, ssGetDiscreteStates

Purpose Specify the sample rates at which this S-function operates.

Syntax void mdlInitializeSampleTimes(SimStruct *S)

Arguments S
Simstruct representing an S-function block.

Description This method should specify the sample time and offset time for each sample rate at which this S-function operates via the following paired macros

```
ssSetSampleTime(S, sampleTimeIndex, sample_time)
ssSetOffsetTime(S, offsetTimeIndex, offset_time)
```

where sampleTimeIndex runs from 0 to one less than the number of sample times specified in mdlInitializeSizes via ssSetNumSampleTimes.

If the S-function operates at one or more sample rates, this method can specify any of the following sample time and offset values for a given sample time:

- [CONTINUOUS_SAMPLE_TIME, 0.0]
- [CONTINUOUS_SAMPLE_TIME, FIXED_IN_MINOR_STEP_OFFSET]
- [discrete_sample_period, offset]
- [VARIABLE_SAMPLE_TIME, 0.0]

The upper case values are macros defined in simstruc.h.

If the S-function operates at one rate, this method can alternatively set the sample time to one of the following sample/offset time pairs.

- [INHERITED_SAMPLE_TIME, 0.0]
- [INHERITED_SAMPLE_TIME, FIXED_IN_MINOR_STEP_OFFSET]

If the number of sample times is 0, Simulink assumes that the S-function inherits its sample time from the block to which it is connected, i.e., that the sample time is

```
[INHERITED_SAMPLE_TIME, 0.0]
```

This method can therefore return without doing anything.

mdlInitializeSampleTimes

Use the following guidelines when specifying sample times.

- A continuous function that changes during minor integration steps should set the sample time to

[CONTINUOUS_SAMPLE_TIME, 0.0]

- A continuous function that does not change during minor integration steps should set the sample time to

[CONTINUOUS_SAMPLE_TIME, FIXED_MINOR_STEP_OFFSET]

- A discrete function that changes at a specified rate should set the sample time to

[discrete_sample_period, offset]

where

discrete_sample_period > 0.0

and

0.0 <= offset < discrete_sample_period

- A discrete function that changes at a variable rate should set the sample time to

[VARIABLE_SAMPLE_TIME, 0.0]

Simulink invokes mdlGetTimeOfNextVarHit function to get the time of the next sample hit for the variable step discrete task.

Note that VARIABLE_SAMPLE_TIME requires a variable step solver.

- To operate correctly in a triggered subsystem or a periodic system, a discrete S-function should:

- Specify a single sample time set to

[INHERITED_SAMPLE_TIME, 0.0]

- Set the SS_DISSALLOW_CONSTANT_SAMPLE_TIME simulation option in

mdlInitializeSizes

- Verify that it was assigned a discrete or triggered sample time in mdlSetWorkWidths:

```
if (ssGetSampleTime(S, 0) == CONTINUOUS_SAMPLE_TIME) {  
    ssSetErrorStatus(S,  
        "This block cannot be assigned a continuous sample time");  
}
```

After propagating sample times throughout the block diagram, Simulink assigns the sample time

```
[INHERITED_SAMPLE_TIME, INHERITED_SAMPLE_TIME]  
to discrete blocks residing in triggered subsystems.
```

If this function has no intrinsic sample time, it should set its sample time to inherited according to the following guidelines:

- A function that changes as its input changes, even during minor integration steps, should set its sample time to

```
[INHERITED_SAMPLE_TIME, 0.0]
```

A function that changes as its input changes, but doesn't change during minor integration steps (i.e., held during minor steps) should set its sample time to

```
[INHERITED_SAMPLE_TIME, FIXED_MINOR_STEP_OFFSET]
```

The S-function should use the `ssIsSampleHit` or `ssIsContinuousTask` macros to check for a sample hit during execution (in `mdlOutputs` or `mdlUpdate`). For example, if the block's first sample time is continuous, the function can use the following code-fragment to check for a sample hit.

```
if (ssIsContinuousTask(S, tid)) {  
}
```

Note The function would get incorrect results if it used `ssIsSampleHit(S, 0, tid)`.

mdlInitializeSampleTimes

If the function wanted to determine if the third (discrete) task has a hit, it could use the following code-fragment.

```
if (ssIsSampleHit(S, 2, tid) {  
}
```

Languages C

See Also mdlSetInputPortSampleTime, mdlSetOutputPortSampleTime

Purpose	Specify the number of inputs, outputs, states, parameters, and other characteristics of the S-function.
Syntax	<code>void mdlInitializeSizes(SimStruct *S)</code>
Arguments	S Simstruct representing an S-function block.
Description	<p>This is the first of the S-function's callback methods that Simulink calls. This method should perform the following tasks:</p> <ul style="list-style-type: none"> • Specify the number of parameters that this S-function supports, using <code>ssSetNumSFcnParams</code>. Use <code>ssSetSFcnParamNotTunable(S, paramIdx)</code> when a parameter cannot change during simulation, where <code>paramIdx</code> starts at 0. When a parameter has been specified as "not tunable," Simulink will issue an error during simulation (or the Real-Time Workshop external mode) if an attempt is made to change the parameter. • Specify the number of states that this function has, using <code>ssSetNumContStates</code> and <code>ssSetNumDiscStates</code>. • Configure the block's input ports. This entails the following tasks. <ul style="list-style-type: none"> - Specify the number of input ports that this S-function has, using <code>ssSetNumInputPorts</code> - Specify the dimensions of the input ports. See "Dynamically Sized Block Features" on page 9-14 for more information. - Specify for each input port whether it has direct feedthrough, using <code>ssSetInputPortDirectFeedthrough</code> A port has direct feedthrough if the input is used in either the <code>mdlOutputs</code> or <code>mdlGetTimeOfNextVarHit</code> functions. The direct feedthrough flag for each input port can be set to either 1=yes or 0=no. It should be set to 1 if the input, <code>u</code>, is used in the <code>mdlOutput</code> or <code>mdlGetTimeOfNextVarHit</code> routine. Setting the direct feedthrough flag to 0 tells Simulink that <code>u</code> will not be used in either of these S-function routines. Violating this will lead to unpredictable results.

- Configure the block's output ports.

This entails the following tasks.

- Specify the number of output ports that the block has, using `ssSetNumOutputPorts`
- Specify the dimensions of the output ports
See `mdlSetOutputPortDimensions` and `ssSetOutputPortDimensions` for more information.

If your S-function outputs are discrete (e.g., can only take on the values, 1 and 2), then specify `SS_OPTION_DISCRETE_VALUED_OUTPUT`.

- Set the number of sample times (i.e., sample rates) at which the block operates.

There are two ways of specifying sample times:

- Port-based sample times
- Block-based sample times

See "Sample Times" on page 7-16 for a complete discussion of sample time issues.

For multi-rate S-functions, the suggested approach to setting sample times is via the port based sample times method. When you create a multirate S-function, care needs to be taken to verify that when slower tasks are preempted that your S-function correctly manages data as to avoid race conditions. When port based sample times are specified, the block cannot inherit a constant sample time at any port.

- Set the size of the block's work vectors, using `ssSetNumRWork`, `ssSetNumIWork`, `ssSetNumPWork`, `ssSetNumModes`, `ssSetNumNonsampledZCs`
- Set the simulation options that this block implements, using `ssSetOptions`. All options have the form `SS_OPTION_<name>`. See `ssSetOptions` for information on each option. The options should be bitwise or'd together as in `ssSetOptions(S, (SS_OPTION_name1 | SS_OPTION_name2))`

Dynamically Sized Block Features

You can set the parameters `NumContStates`, `NumDiscStates`, `NumInputs`, `NumOutputs`, `NumRWork`, `NumIWork`, `NumPWork`, `NumModes`, and `NumNonsampledZCs` to a fixed nonnegative integer or tell Simulink to size them dynamically:

- DYNAMICALLY_SIZED — Sets lengths of states, work vectors, and so on to values inherited from the driving block. It sets widths to the actual input width, according to the scalar expansion rules unless you use mdlSetWorkWidths to set the widths.
- 0 or positive number — Sets lengths (or widths) to the specified value. The default is 0.

Languages

Ada, C, M

Example

```
static void mdlInitializeSizes(SimStruct *S)
{
    int_T nInputPorts = 1; /* number of input ports */
    int_T nOutputPorts = 1; /* number of output ports */
    int_T needsInput = 1; /* direct feed through */

    int_T inputPortIdx = 0;
    int_T outputPortIdx = 0;

    ssSetNumSFcnParams(S, 0); /* Number of expected parameters */
    if (ssGetNumSFcnParams(S) != ssGetSFcnParamsCount(S)) {
        /*
         * If the the number of expected input parameters is not
         * equal to the number of parameters entered in the
         * dialog box, return. Simulink will generate an error
         * indicating that there is a parameter mismatch.
         */
        return;
    } else {
        mdlCheckParameters(S);

        if (ssGetErrorStatus(s) != NULL)
            return;
    }

    ssSetNumContStates(S, 0);
    ssSetNumDiscStates(S, 0);

    /*
     * Configure the input ports. First set the number of input
     * ports.
     */
    if (!ssSetNumInputPorts(S, nInputPorts)) return;
    /*
     * Set input port dimensions for each input port index
     */
}
```

mdlInitializeSizes

```
    * starting at 0.
    */
    if(!ssSetInputPortDimensionInfo(S, inputPortIdx,
        DYNAMIC_DIMENSION)) return;
    /*
    * Set direct feedthrough flag (1=yes, 0=no).
    */
    ssSetInputPortDirectFeedThrough(S, inputPortIdx, needsInput);

    /*
    * Configure the output ports. First set the number of
    * output ports.
    */
    if(!ssSetNumOutputPorts(S, nOutputPorts)) return;

    /*
    * Set output port dimensions for each output port index
    * starting at 0.
    */
    if(!ssSetOutputPortDimensionInfo(S, outputPortIdx,
        DYNAMIC_DIMENSION)) return;

    /*
    * Set the number of sample times.      */
    ssSetNumSampleTimes(S, 1);

    /*
    * Set size of the work vectors.
    */
    ssSetNumRWork(S, 0); /* real vector */
    ssSetNumIWork(S, 0); /* integer vector */
    ssSetNumPWork(S, 0); /* pointer vector */
    ssSetNumModes(S, 0); /* mode vector */
    ssSetNumNonsampledZCs(S, 0); /* zero crossings */

    ssSetOptions(S, 0);

} /* end mdlInitializeSizes */
```

Purpose	Compute the signals that this block emits.
Syntax	<code>void mdlOutputs(SimStruct *S, int_T tid)</code>
Arguments	<code>S</code> Simstruct representing an S-function block. <code>tid</code> Task id
Description	<p>Simulink invokes this required method at each simulation time step. The method should compute the S-function's outputs at the current time step and store the results in the S-function's output signal arrays.</p> <p>The <code>tid</code> (task ID) argument specifies the task running when the <code>mdlOutputs</code> routine is invoked. You can use this argument in the <code>mdlOutputs</code> routine of a multirate S-Function block to encapsulate task-specific blocks of code (see "Multirate S-Function Blocks" on page 7-21).</p> <p>For an example of an <code>mdlOutputs</code> routine that works with multiple input and output ports, see <i>matlabroot</i>/simulink/src/sfun_multiport.c.</p>
Languages	A, C, M
See Also	<code>ssGetOutputPortSignal</code> , <code>ssGetOutputPortRealSignal</code> , <code>ssGetOutputPortComplexSignal</code>

mdlProcessParameters

Purpose Process the S-function's parameters.

Syntax `void mdlProcessParameters(SimStruct *S)`

Arguments S
Simstruct representing an S-function block.

Description This is an optional routine that Simulink calls after `mdlCheckParameters` changes and verifies parameters. The processing is done at the top of the simulation loop when it is safe to process the changed parameters. This routine can only be used in a C MEX S-function.

The purpose of this routine is to process newly changed parameters. An example is to cache parameter changes in work vectors. Simulink does not call this routine when it is used with the Real-Time Workshop. Therefore, if you use this routine in an S-function designed for use with the Real-Time Workshop, you must write your S-function so that it doesn't rely on this routine. To do this, you must inline your S-function by using the Target Language Compiler. See "The Target Language Compiler Reference Guide" for information on inlining S-functions.

The synopsis is

```
#define MDL_PROCESS_PARAMETERS /* Change to #undef to remove function */
#if defined(MDL_PROCESS_PARAMETERS) && defined(MATLAB_MEX_FILE)
static void mdlProcessParameters(SimStruct *S)
{
}
#endif /* MDL_PROCESS_PARAMETERS */
```

Example This example processes a string parameter that `mdlCheckParameters` has verified to be of the form '+++' (where there could be any number of '+' or '-' characters).

```
#define MDL_PROCESS_PARAMETERS /* Change to #undef to remove function */
#if defined(MDL_PROCESS_PARAMETERS) && defined(MATLAB_MEX_FILE)
static void mdlProcessParameters(SimStruct *S)
{
    int_T i;
    char_T *pl usMi nusStr;
    int_T nInputPorts = ssGetNumInputPorts(S);
    int_T *iwork = ssGetIWork(S);
    if ((pl usMi nusStr=(char_T*)malloc(nInputPorts+1)) == NULL) {
        ssSetErrorStatus(S, "Memory allocation error in mdlStart");
        return;
    }
}
```

```

    }
    if (mxGetString(SIGNS_PARAM(S), plusMinusStr, nInputPorts+1) != 0) {
        free(plusMinusStr);
        ssSetErrorStatus(S, "mxGetString error in mdl Start");
        return;
    }
    for (i = 0; i < nInputPorts; i++) {
        iwork[i] = plusMinusStr[i] == '+' ? 1: -1;
    }
    free(plusMinusStr);
}
#endif /* MDL_PROCESS_PARAMETERS */

```

mdl ProcessParameters is called from mdl Start to load the signs string prior to the start of the simulation loop.

```

#define MDL_START
#if defined(MDL_START)
static void mdl Start(SimStruct *S)
{
    mdl ProcessParameters(S);
}
#endif /* MDL_START */

```

For more details on this example, see *matlabroot/simulink/src/sfun_mulptiport.c*.

Languages

Ada, C, M

See Also

mdl CheckParameters

mdlRTW

Purpose	Generate code generation data.
Syntax	<code>void mdlRTW(SimStruct *S)</code>
Arguments	S Simstruct representing an S-function block.
Description	<p>This function is called when the Real-Time Workshop is generating the <code>model.rtw</code> file. In this method, you can call the following functions which add fields to the <code>model.rtw</code> file:</p> <ul style="list-style-type: none">• <code>ssWriteRTWParameters</code>• <code>ssWriteRTWParamSettings</code>• <code>ssWriteRTWorkVect</code>• <code>ssWriteRTWStr</code>• <code>ssWriteRTWStrParam</code>• <code>ssWriteRTWScalarParam</code>• <code>ssWriteRTWStrVectParam</code>• <code>ssWriteRTWVectParam</code>• <code>ssWriteRTW2dMatParam</code>• <code>ssWriteRTWMxVectParam</code>• <code>ssWriteRTWMx2dMatParam</code>
Languages	C
See Also	<code>ssSetInputPortFrameData</code> , <code>ssSetOutputPortFrameData</code> , <code>ssSetErrorStatus</code>

Purpose	Set the numeric type (real, complex, or inherited) of ports whose numeric type cannot be determined from block connectivity.
Syntax	<code>void mdlSetDefaultPortComplexSignals(SimStruct *S)</code>
Arguments	S Simstruct representing an S-function block.
Description	<p>Simulink invokes this method if the block has ports whose numeric type cannot be determined from connectivity. (This usually happens when the block is unconnected or is part of a feedback loop.) This method must set the data type of all ports whose data type is not set.</p> <p>If the block does not implement this method and Simulink cannot determine the data types of any of its ports, Simulink sets the data types of all the ports to double. If the block does not implement this method and Simulink cannot determine the data types of some, but not all, of its ports, Simulink sets the unknown ports to the data type of the port whose data type has the largest size.</p>
Languages	C
See Also	<code>ssSetOutputPortDataType</code> , <code>ssSetInputPortDataType</code>

mdlSetDefaultPortDataTypes

Purpose	Set the data type of ports whose data type cannot be determined from block connectivity.
Syntax	<code>void mdlSetDefaultPortDataTypes(SimStruct *S)</code>
Arguments	S Simstruct representing an S-function block.
Description	<p>Simulink invokes this method if the block has ports whose numeric type cannot be determined from connectivity. (This usually happens when the block is unconnected or is part of a feedback loop.) This method must set the numeric type of all ports whose numeric type is not set.</p> <p>If the block does not implement this method and at least one port is known to be complex, Simulink sets the unknown ports to <code>COMPLEX_YES</code>; otherwise, it sets the unknown ports to <code>COMPLEX_NO</code>.</p>
Languages	C
See Also	<code>ssSetOutputPortComplexSignal</code> , <code>ssSetInputPortComplexSignal</code>

Purpose	Set the default dimensions of the signals accepted or emitted by an S-function's ports.
Syntax	<code>void mdlSetDefaultPortDimensionInfo(SimStruct *S, int_T port)</code>
Arguments	S Simstruct representing an S-function block.
Description	Simulink calls this method during signal dimension propagation when a model does not supply enough information to determine the dimensionality of signals that can enter or leave the block represented by S. This method should set the dimensions of any input and output ports that are dynamically sized to default values. If S does not implement this method, Simulink set the dimensions of dynamically sized ports for which dimension information is unavailable to scalar, i.e., 1-D signals containing one element.
Example	See <code>matlabroot/simulink/src/sfun_matadd.c</code> for an example of how to use this function.
Languages	C
See Also	<code>ssSetOutputPortDimensionInfo</code> , <code>ssSetOutputPortDimensionInfo</code> , <code>ssSetErrorStatus</code>

mdlSetInputPortComplexSignal

Purpose	Set the numeric type (real, complex, or inherited) of the signals accepted by an input port.
Syntax	<pre>void mdlSetInputPortDataType(SimStruct *S, int_T port, CSIGNAL_T csig)</pre>
Arguments	<p>S Simstruct representing an S-function block.</p> <p>port Index of a port</p> <p>csig Numeric type of signal</p>
Description	<p>Simulink calls this routine to set the input port signal type. The S-function must check if the specified signal type is a valid type for the specified port. If it is valid, the s-function must set the signal type of the specified input port. Otherwise, it must report an error using <code>ssSetErrorStatus</code>. The s-function can also set the signal type of other input and output ports with unknown signal types. Simulink reports an error if the S-function changes the signal type of a port whose signal type is known.</p> <p>If the S-function does not implement this routine, Simulink assumes that the S-function accepts a real or complex signal and sets the input port signal type to the specified value.</p>
Languages	C
See Also	<code>ssSetInputPortComplexSignal</code> , <code>ssSetErrorStatus</code>

Purpose	Set the data type of the signals accepted by an input port.
Syntax	<code>void mdlSetInputPortDataType (SimStruct *S, int_T port, DTypel d id)</code>
Arguments	<p><code>S</code> Simstruct representing an S-function block.</p> <p><code>port</code> Index of a port</p> <p><code>id</code> Data type id</p>
Description	<p>Simulink calls this routine to set the data type of <code>port</code>. The S-function must check if the specified data type is a valid data type for the specified port. If it is a valid data type, it must set the data type of the input port. Otherwise, it must report an error using <code>ssSetErrorStatus</code>.</p> <p>The S-function can also set the data type of other input and output ports if they are unknown. Simulink reports an error if the S-function changes the data type of a port whose data type has been set.</p> <p>If the block does not implement this routine, Simulink assumes that the block accepts any data type and sets the input port data type to the specified value.</p>
Languages	C
See Also	<code>ssSetInputPortDataType</code> , <code>ssSetErrorStatus</code>

mdlSetInputPortDimensionInfo

Purpose	Set the dimensions of the signals accepted by an input port.
Syntax	<pre>void mdlSetInputPortDimensionInfo(SimStruct *S, int_T port, const DimInfo_T *dimInfo)</pre>
Arguments	<p>S Simstruct representing an S-function block.</p> <p>port Index of a port</p> <p>dimInfo Structure that specifies the signal dimensions supported by port See <code>ssSetInputPortDimensionInfo</code> for a description of this structure.</p>
Description	Simulink calls this method during dimension propagation with candidate dimensions, <code>dimInfo</code> , for <code>port</code> . If the proposed dimensions are acceptable, this method should go ahead and set the actual port dimensions, using <code>ssSetInputPortDimensionInfo</code> . If they are unacceptable, this method should generate an error via <code>ssSetErrorStatus</code> .

Note This method can set the dimensions of any other input or output port whose dimensions derive from the dimensions of `port`.

By default, Simulink calls this method only if it can fully determine the dimensionality of `port` from the port to which it is connected. If it cannot completely determine the dimensionality from port connectivity, it invokes `mdlSetDefaultPortDimensionInfo`. If an S-function can fully determine the port dimensionality from partial information, the function should set the option, `SS_OPTION_ALLOW_PARTIAL_DIMENSIONS_CALL`, in `mdlInitializeSizes`, using `ssSetOptions`. If this option is set, Simulink invokes `mdlSetInputPortDimensionInfo` even if it can only partially determine the dimensionality of the input port from connectivity.

Languages	C
-----------	---

Example See `matlabroot\simulink\src\sfun_matadd.c` for an example of how to use this function.

See Also `ssSetInputPortDimensionInfo`, `ssSetErrorStatus`

mdlSetInputPortFrameData

Purpose	Set frame data entering an input port.
Syntax	<pre>void mdlSetInputPortFrameData(SimStruct *S, int_T port, Frame_T frameData)</pre>
Arguments	<p>S Simstruct representing an S-function block.</p> <p>port Index of a port</p> <p>frameData frame data</p>
Description	<p>This method is called with the candidate frame setting (FRAME_YES, or FRAME_NO) for an input port. If the proposed setting is acceptable, the method should go ahead and set the actual frame data setting using <code>ssSetInputPortFrameData</code>. If the setting is unacceptable an error should be generated via <code>ssSetErrorStatus</code>. Note that any other dynamic frame input or output ports whose frame data setting are implicitly defined by virtue of knowing the frame data setting of the given port can also have their frame data settings set via calls to <code>ssSetInputPortFrameData</code> and <code>ssSetOutputPortFrameData</code>.</p>
Languages	C
See Also	<code>ssSetInputPortFrameData</code> , <code>ssSetOutputPortFrameData</code> , <code>ssSetErrorStatus</code>

Purpose Set the sample time of an input port that inherits its sample time from the port to which it is connected.

Syntax `void mdlSetInputPortSampleTime(SimStruct *S, int_T port, real_T sampleTime, real_T offsetTime)`

Arguments

`S`
Simstruct representing an S-function block.

`port`
Index of a port

`sampleTime`
Inherited sample time for port

`offsetTime`
Inherited offset time for port

Description Simulink invokes this method with the sample time that port inherits from the port to which it is connected. If the inherited sample time is acceptable, this method should set the sample time of port to the inherited time, using `ssSetInputPortSampleTime`. If the sample time is unacceptable, this method should generate an error via `ssSetErrorStatus`. Note that any other inherited input or output ports whose sample times are implicitly defined by virtue of knowing the sample time of the given port can also have their sample times set via calls to `ssSetInputPortSampleTime` or `ssSetOutputPortSampleTime`.

When inherited port based sample times are specified, we are guaranteed that the sample time will be one of the following:

	Sample Time	Offset Time
Continuous	0.0	0.0
Discrete	period	offset

where $0.0 < \text{period} < \text{inf}$ and $0.0 \leq \text{offset} < \text{period}$. Constant, triggered, and variable step sample times are not be propagated to S-functions with port-based sample times.

mdlSetInputPortSampleTime

Generally `mdlSetInputPortSampleTime` is called once with the input port sample time. However, there can be cases where this function will be called more than once. This happens when the simulation engine is converting continuous sample times to continuous but fixed in minor steps sample times. When this occurs, the original values of the sample times specified in `mdlInitializeSizes` will be restored before calling this method again.

The final sample time specified at the port may be different from (but equivalent to) the sample time specified by this method. This occurs when:

- The model uses a fixed step solver and the port has a continuous but fixed in minor step sample time. In this case, Simulink converts the sample time to the fundamental sample time for the model.
- Simulink adjusts the sample time to be as numerically sound as possible. For example, Simulink converts `[0.2499999999999, 0]` to `[0.25, 0]`.

The S-function can examine the final sample times in `mdlInitializeSampleTimes`.

Languages

C

See Also

`ssSetInputPortSampleTime`, `ssSetOutputPortSampleTimes`,
`mdlInitializeSampleTimes`

Purpose	Set the width of an input port that accepts 1-D (vector) signals.
Syntax	<code>void mdlSetInputPortWidth (SimStruct *S, int_T port, int_T width)</code>
Arguments	<p>S Simstruct representing an S-function block.</p> <p>port Index of a port</p> <p>width Width of signal</p>
Description	This method is called with the candidate width for a dynamically sized port. If the proposed width is acceptable, the method should go ahead and set the actual port width using <code>ssSetInputPortWidth</code> . If the size is unacceptable an error should be generated via <code>ssSetErrorStatus</code> . Note that any other dynamically sized input or output ports whose widths are implicitly defined by virtue of knowing the width of the given port can also have their widths set via calls to <code>ssSetInputPortWidth</code> or <code>ssSetOutputPortWidth</code> .
Languages	C
See Also	<code>ssSetInputPortWidth</code> , <code>ssSetOutputPortWidth</code> , <code>ssSetErrorStatus</code>

mdlSetOutputPortComplexSignal

Purpose	Set the numeric type (real, complex, or inherited) of the signals accepted by an output port.
Syntax	<pre>void mdlSetOutputPortDataType(SimStruct *S, int_T port, CSIGNAL_T csig)</pre>
Arguments	<p>S Simstruct representing an S-function block.</p> <p>port Index of a port</p> <p>csig Numeric type of signal</p>
Description	<p>Simulink calls this routine to set the output port signal type. The S-function must check if the specified signal type is a valid type for the specified port. If it is valid, the s-function must set the signal type of the specified output port. Otherwise, it must report an error using <code>ssSetErrorStatus</code>. The s-function can also set the signal type of other input and output ports with unknown signal types. Simulink reports an error if the S-function changes the signal type of a port whose signal type is known.</p> <p>If the S-function does not implement this routine, Simulink assumes that the S-function accepts a real or complex signal and sets the output port signal type to the specified value.</p>
Languages	C
See Also	<code>ssSetOutputPortComplexSignal</code> , <code>ssSetErrorStatus</code>

Purpose	Set the data type of the signals emitted by an output port.
Syntax	<code>void mdlSetOutputPortDataType (SimStruct *S, int_T port, DType id)</code>
Arguments	<p><code>S</code> Simstruct representing an S-function block.</p> <p><code>port</code> Index of an output port</p> <p><code>id</code> Data type id</p>
Description	<p>Simulink calls this routine to set the data type of <code>port</code>. The S-function must check if the specified data type is a valid data type for the specified port. If it is a valid data type, it must set the data type of <code>port</code>. Otherwise, it must report an error using <code>ssSetErrorStatus</code>.</p> <p>The S-function can also set the data type of other input and output ports if their data types have not been set. Simulink reports an error if the S-function changes the data type of a port whose data type has been set.</p> <p>If the block does not implement this method, Simulink assumes that the block accepts any data type and sets the input port data type to the specified value.</p>
Languages	C
See Also	<code>ssSetOutputPortDataType</code> , <code>ssSetErrorStatus</code>

mdlSetOutputPortDimensionInfo

Purpose	Set the dimensions of the signals accepted by an output port.
Syntax	<pre>void mdlSetOutputPortDimensionInfo(SimStruct *S, int_T port, const DimensionInfo_T *dimensionInfo)</pre>
Arguments	<p>S Simstruct representing an S-function block or a Simulink model.</p> <p>port Index of a port</p> <p>dimensionInfo Structure that specifies the signal dimensions supported by port See <code>ssSetInputPortDimensionInfo</code> for a description of this structure.</p>
Description	<p>Simulink calls this method with candidate dimensions, <code>dimensionInfo</code>, for <code>port</code>. If the proposed dimensions are acceptable, this method should go ahead and set the actual port dimensions, using <code>ssSetOutputPortDimensionInfo</code>. If they are unacceptable, this method should generate an error via <code>ssSetErrorStatus</code>.</p> <hr/> <p>Note This method can set the dimensions of any other input or output port whose dimensions derive from the dimensions of <code>port</code>.</p> <hr/> <p>By default, Simulink calls this method only if it can fully determine the dimensionality of <code>port</code> from the port to which it is connected. If it cannot completely determine the dimensionality from port connectivity, it invokes <code>mdlSetDefaultPortDimensionInfo</code>. If an S-function can fully determine the port dimensionality from partial information, the function should set the option, <code>SS_OPTION_ALLOW_PARTIAL_DIMENSIONS_CALL</code>, in <code>mdlInitializeSizes</code>, using <code>ssSetOptions</code>. If this option is set, Simulink invokes <code>mdlSetOutputPortDimensionInfo</code> even if it can only partially determine the dimensionality of the input port from connectivity.</p>
Languages	C
Example	See <code>matlabroot/simulink/src/sfun_matadd.c</code> for an example of how to use this function.

See Also

[ssSetOutputPortDimensionInfo](#), [ssSetErrorStatus](#)

mdlSetOutputPortSampleTime

Purpose	Set the sample time of an output port that inherits its sample time from the port to which it is connected.
Syntax	<pre>void mdlSetOutputPortSampleTime(SimStruct *S, int_T port, real_T sampleTime, real_T offsetTime)</pre>
Arguments	<p>S Simstruct representing an S-function block.</p> <p>port Index of a port</p> <p>sampleTime Inherited sample time for port</p> <p>offsetTime Inherited offset time for port</p>
Description	<p>Simulink calls this method with the sample time that port inherits from the port to which it is connected. If the inherited sample time is acceptable, this method should set the sample time of port to the inherited sample time, using <code>ssSetOutputPortSampleTime</code>. If the inherited sample time is unacceptable, this method should generate an error generated via <code>ssSetErrorStatus</code>. Note that this method can set the sample time of any other input or output port whose sample time derives from the sample time of port, using <code>ssSetInputPortSampleTime</code> or <code>ssSetOutputPortSampleTime</code>.</p> <p>Normally, sample times are propagated forwards, however if sources feeding this block have an inherited sample time, Simulink may choose to back propagate known sample times to this block. When back propagating sample times, we call this method in succession for all inherited output port signals.</p> <p>See <code>mdlSetInputPortSampleTime</code> for more information about when this method is called.</p>
Languages	C
See Also	<code>ssSetOutputPortSampleTime</code> , <code>ssSetErrorStatus</code> , <code>ssSetInputPortSampleTime</code> , <code>ssSetOutputPortSampleTime</code> , <code>mdlSetInputPortSampleTime</code>

Purpose	Set the width of an output port that outputs 1-D (vector) signals.
Syntax	<code>void mdlSetOutputPortWidth(SimStruct *S, int_T port, int_T width)</code>
Arguments	<p>S Simstruct representing an S-function block.</p> <p>port Index of a port</p> <p>width Width of signal</p>
Description	This method is called with the candidate width for a dynamically sized port. If the proposed width is acceptable, the method should go ahead and set the actual port width using <code>ssSetOutputPortWidth</code> . If the size is unacceptable an error should be generated via <code>ssSetErrorStatus</code> . Note that any other dynamically sized input or output ports whose widths are implicitly defined by virtue of knowing the width of the given port can also have their widths set via calls to <code>ssSetInputPortWidth</code> or <code>ssSetOutputPortWidth</code> .
Languages	C
See Also	<code>ssSetInputPortWidth</code> , <code>ssSetOutputPortWidth</code> , <code>ssSetErrorStatus</code>

mdlSetWorkWidths

Purpose	Specify the sizes of the work vectors and create the runtime parameters required by this S-function.
Syntax	<code>void mdlSetWorkWidths(SimStruct *S)</code>
Arguments	S Simstruct representing an S-function block.
Description	<p>Simulink calls this optional method to enable this S-function to set the sizes of state and work vectors that it needs to store global data and to create runtime parameters (see “Run-Time Parameters” on page 7-6). Simulink invokes this method after it has determined the input port width, output port width, and sample times of the S-function. This allows the S-function to size the state and work vectors based on the number and sizes of inputs and outputs and/or the number of sample times. This method specifies the state and work vector sizes via the macros <code>ssNumContStates</code>, <code>ssSetNumDiscStates</code>, <code>ssSetNumRWork</code>, <code>ssSetNumIWork</code>, <code>ssSetNumPWork</code>, <code>ssSetNumModes</code>, and <code>ssSetNumNonsampledZCs</code>.</p> <p>The S-function needs to implement this method only if it does not know the sizes of all the work vectors it requires when Simulink invokes the function's <code>mdlInitializeSizes</code> method. If this S-function implements <code>mdlSetWorkWidths</code>, it should initialize the sizes of any work vectors that it needs to DYNAMICALLY_SIZE in <code>mdlInitializeSizes</code>, even for those whose exact size it knows at that point. The S-function should then specify the actual size in <code>mdlSetWorkWidths</code>.</p>
Languages	Ada, C
See Also	<code>mdlInitializeSizes</code>

Purpose	Initialize the state vectors of this S-function.
Syntax	<code>void mdlStart(SimStruct *S)</code>
Arguments	S Simstruct representing an S-function block.
Description	Simulink invokes this optional method at the beginning of a simulation. It should initialize the continuous and discrete states, if any, of this S-function block. Use <code>ssGetContStates</code> and/or <code>ssGetDiscreteStates</code> to get the states. This method can also perform any other initialization activities that this S-function requires.
Languages	Ada, C
See Also	<code>mdlInitializeConditions</code> , <code>ssGetContStates</code> , <code>ssGetDiscreteStates</code>

mdlTerminate

Purpose	Perform any actions required at termination of the simulation.
Syntax	<code>void mdlTerminate(SimStruct *S)</code>
Arguments	S Simstruct representing an S-function block.
Description	This method should perform any actions, such as freeing memory, that must be performed at the end of simulation or when an S-function block is destroyed (e.g., when it is deleted from a model). The option <code>SS_OPTION_CALL_TERMINATE_ON_EXIT</code> (see <code>ssSetOptions</code>) determines whether Simulink invokes this method. If this option is not set, Simulink invokes <code>mdlTerminate</code> at the end of simulation only if the <code>mdlStart</code> method of at least one block in the model has executed during simulation. If this option is set, Simulink always invokes the <code>mdlTerminate</code> method at the end of a simulation run and whenever it destroys a block.
Languages	Ada, C, M
Example	Suppose your S-function allocates blocks of memory in <code>mdlStart</code> and saves pointers to the blocks in a <code>PWork</code> vector. The following code fragment would free this memory. <pre>{ int i; for (i = 0; i < ssGetNumPWork(S); i++) { if (ssGetPWorkValue(S, i) != NULL) { free(ssGetPWorkValue(S, i)); } } }</pre>

Purpose	Update a block's states.
Syntax	<code>void mdlUpdate(SimStruct *S, int_T tid)</code>
Arguments	<code>S</code> Simstruct representing an S-function block. <code>tid</code> Task ID
Description	<p>Simulink invokes this optional method at each major simulation time step. The method should compute the S-function's states at the current time step and store the states in the S-function's state vector. The method can also perform any other tasks that the S-function needs to perform at each major time step.</p> <p>Use this code if your S-function has one or more discrete states or does not have direct feedthrough.</p> <p>The reason for this is that most S-functions that do not have discrete states but do have direct feedthrough do not have update functions. Therefore, Simulink is able to eliminate the need for the extra call in these circumstances.</p> <p>If your S-function needs to have its <code>mdlUpdate</code> routine called and it does not satisfy either of the above two conditions, specify that it has a discrete state using the <code>ssSetNumDiscreteStates</code> macro in the <code>mdlInitializeSizes</code> function.</p> <p>The <code>tid</code> (task ID) argument specifies the task running when the <code>mdlOutputs</code> routine is invoked. You can use this argument in the <code>mdlUpdate</code> routine of a multirate S-Function block to encapsulate task-specific blocks of code (see "Multirate S-Function Blocks" on page 7-21).</p>
Example	For an example, see <i>matlabroot</i> /simulink/src/dsfunc.c
Languages	Ada, C, M
See Also	<code>mdlDerivatives</code> , <code>ssGetContStates</code> , <code>ssGetDiscreteStates</code>

mdlZeroCrossings

Purpose	Update zero-crossing vector.
Syntax	<code>void mdlZeroCrossings(SimStruct *S)</code>
Arguments	S Simstruct representing an S-function block.
Description	<p>An S-function needs to provide this optional method only if it does zero-crossing detection. This method should update the S-function's zero-crossing vector, using <code>ssGetNonsampledZCs</code>.</p> <p>You can use the optional <code>mdlZeroCrossings</code> routine, when your S-function has registered the <code>CONTINUOUS_SAMPLE_TIME</code> and has nonsampled zero crossings (<code>ssGetNumNonsampledZCs(S) > 0</code>). The <code>mdlZeroCrossings</code> routine is used to provide Simulink with signals that are to be tracked for zero crossings. These are typically:</p> <ul style="list-style-type: none">• Continuous signals entering the S-function• Internally generated signals that cross zero when a discontinuity would normally occur in mdl Outputs <p>Thus, the zero crossing signals are used to locate the discontinuities and end the current time step at the point of the zero crossing. To provide Simulink with zero crossing signal(s), <code>mdlZeroCrossings</code> updates the <code>ssGetNonsampledZCs(S)</code> vector.</p>
Example	See <i>matlabroot</i> /simulink/src/sfun_zc.c.
Languages	C
See Also	<code>mdlInitializeSizes</code> , <code>ssGetNonsampledZCs</code>

SimStruct Functions

Introduction	10-2
Language Support	10-2
The SimStruct	10-2
SimStruct Macros and Functions Listed by Usage	10-3
Miscellaneous	10-3
Error Handling and Status	10-3
I/O Port	10-4
Dialog Box Parameters	10-6
Run-Time Parameters	10-7
Sample Time	10-8
State and Work Vector	10-9
Simulation Information	10-12
Function Call	10-12
Data Type	10-13
Real-Time Workshop	10-13
Macro Reference	10-15

Introduction

Simulink provides a set of functions for accessing the fields of an S-function's simulation data structure (SimStruct). S-function callback methods use these functions to store and retrieve information about an S-function.

This reference describes the syntax and usage of each SimStruct function. The descriptions appear alphabetically by name to facilitate location of a particular macro. This section also provides listings of functions by usage to speed location of macros for specific purposes, such as implementing data type support.

Language Support

Some SimStruct functions are available only in some of the languages supported by Simulink. The reference page for each SimStruct function lists the languages in which it is available. If the SimStruct function is available in C, the reference page gives its C syntax. Otherwise, it gives its syntax in the language in which it is available.

Note Most SimStruct functions available in C are implemented as C macros.

The SimStruct

The file `matlabroot/simulink/include/simstruct.h` is a C language header file that defines the Simulink data structure and the SimStruct access macros. It encapsulates all the data relating to the model or S-function, including block parameters and outputs.

There is one SimStruct data structure allocated for the Simulink model. Each S-function in the model has its own SimStruct associated with it. The organization of these SimStructs is much like a directory tree. The SimStruct associated with the model is the root SimStruct. The SimStructs associated with the S-functions are the child SimStructs.

SimStruct Macros and Functions Listed by Usage

This section groups SimStruct macros by usage.

Miscellaneous

Macro	Description
ssGetModelName	Get the name of an S-function block or model containing the S-function.
ssGetParentSS	Get the parent of an S-function.
ssGetPath	Get the path of an S-function or the model containing the S-function.
ssGetRootSS	Return the root (model) SimStruct.
ssSetOptions	Set various simulation options.
ssSetPlacementGroup	Specify the execution order of a sink or source S-function.

Error Handling and Status

Macros	Description
ssGetSimMode	Determine context in which an S-function is being invoked: normal simulation, external-mode simulation, model editor, etc.
ssGetSolverName	Get name of the solver being used for the simulation.
ssIsVariableStepSolver	Determine if the current solver is a variable step solver.
ssPrintf	Print a variable-content msg.

Macros	Description
ssSetErrorStatus	Report errors.
ssWarning	Display a warning message.

I/O Port

Macro	Description
ssGetInputPortBufferDstPort	Determine the output port that is overwriting an input port's memory buffer.
ssGetInputPortConnected	Determine if an S-function block port is connected to a nonvirtual block.
ssGetInputPortDirectFeedThrough	Determine if an input port has direct feedthrough.
ssGetInputPortOffsetTime	Determine the offset time of an input port.
ssGetInputPortRealSignalPtrs	Access the signal elements connected to an input port.
ssGetInputPortSampleTime	Determine the sample time of an input port.
ssGetInputPortSignalPtrs	Get pointers to input signal elements of type other than double.
ssGetInputPortWidth	Determine the width of an input port.
ssGetNumInputPorts	Determine how many input ports a block has.
ssGetNumOutputPorts	Can be used in any routine (except mdlInitializeSizes) to determine how many output ports you have set.

Macro	Description
ssGetOutputPortOffsetTime	Determine the offset time of an output port.
ssGetOutputPortRealSignal	Access the elements of a signal connected to an output port.
ssGetOutputPortSampleTime	Determine the sample time of an output port.
ssGetOutputPortWidth	Determine the width of an output port.
ssSetInputPortDirectFeedThrough	Specify that an input port is a direct feedthrough port.
ssSetInputPortOffsetTime	Specify the sample time offset for an input port.
ssSetInputPortOverwritable	Specify whether an input port is overwritable by an output port.
ssSetInputPortReusable	Specify whether an input port's memory buffer can be reused by other signals in the model.
ssSetInputPortSampleTime	Set the sample time of an input port.
ssSetInputPortWidth	Set width of an input port.
ssSetNumInputPorts	Set the number of input ports on an S-function block.
ssSetNumOutputPorts	Specify the number of output ports on an S-function block.
ssSetOutputPortComplexSignal	Specify the numeric type (real or complex) of this port.
ssSetOutputPortDataType	Specify the data type of an output port.

Macro	Description
ssSetOutputPortOffsetTime	Specify the sample time offset value of an output port.
ssSetOutputPortReusable	Specify whether an output port's memory can be reused.
ssSetOutputPortSampleTime	Specify the sample time of an output port.
ssSetOutputPortWidth	Specify width of a 1-D (vector) output port.
ssSetOutputPortDimensionInfo	Specify the dimensions of an output port.
ssSetOutputPortMatrixDimensions	Specify the dimensions of a 2-D (matrix) signal.
ssSetOutputPortVectorDimension	Specify the dimension of a 1-2 (vector) signal.

Dialog Box Parameters

These macros enable an S-function to access and set the tunability of parameters that a user specifies in the S-function's dialog box.

Macro	Description
ssGetDTypeIDFromMxArray	Returns the Simulink data type of a dialog parameter.
ssGetNumSFcnParams	Get the number of parameters that an S-function expects.
ssGetSFcnParam	Get a parameter entered by a user in the S-function block dialog box.
ssSetNumSFcnParams	Set the number of parameters that an S-function expects.

Macro	Description
ssGetSfcnParamCount	Get the actual number of parameters specified by the user.
ssSetSfcnParamNotTunable	Obsolete.
ssSetSfcnParamTunable	Specify the tunability of a dialog box parameter.

Run-Time Parameters

These macros allow you to create, update, and access run-time parameters corresponding to a block's dialog parameters.

Macro	Description
ssGetNumRunTimeParams	Gets the number of run-time parameters created by this S-function.
ssGetRunTimeParamInfo	Gets attributes of a specified run-time parameter.
ssRegAllTunableParamsAsRunTimeParams	Register all tunable dialog parameters as run-time parameters.
ssSetNumRunTimeParams	Specify the number of run-time parameters to be created by this S-function.
ssSetRunTimeParamInfo	Specify attributes of a specified run-time parameter.
ssUpdateAllTunableParamsAsRunTimeParams	Update all run-time parameters corresponding to tunable dialog parameters.
ssUpdateRunTimeParamData	Update the value of a specified run-time parameter.
ssUpdateRunTimeParamInfo	Update the attributes of a specified run-time from the attributes of the corresponding dialog parameters.

Sample Time

Macro	Description
ssGetTNext	Get the time of the next sample hit in a discrete S-function with a variable sample time.
ssGetNumSampleTimes	Get the number of sample times an S-function has.
ssIsContinuousTask	Determine if a specified rate is the continuous rate.
ssIsSampleHit	Determine the sample rate at which an S-function is operating.
ssIsSpecialSampleHit	Determine if the current sample time hits two specified rates.
ssSetNumSampleTimes	Set the number of sample times an S-function has.
ssSetOffsetTime	Specify the offset of a sample time.
ssSetSampleTime	Specify a sample time for an S-function.
ssSetTNext	Specify time of next sample hit in an S-function.

State and Work Vector

Macro	Description
<code>ssGetContStates</code>	Get an S-function's continuous states.
<code>ssGetDi scStates</code>	Get an S-function's discrete states.
<code>ssGetDWorkCompl exSi gnal</code>	Determine whether the elements of a data type work vector are real or complex numbers.
<code>ssGetDWorkDataType</code>	Get the data type of a data type work vector.
<code>ssGetDWorkName</code>	Get the name of a data type work vector.
<code>ssGetDWorkUsedAsDState</code>	Determine whether a data type work vector is used as a discrete state vector.
<code>ssGetDWorkWi dth</code>	Get the size of a data type work vector.
<code>ssGetdX</code>	Get the derivatives of the continuous states of an S-function.
<code>ssGetI Work</code>	Get an S-function's integer-valued (<code>int_T</code>) work vector.
<code>ssGetModeVector</code>	Get an S-function's mode work vector.
<code>ssGetNonsampl edZCs</code>	Get an S-function's zero-crossing signals vector.
<code>ssGetNumContStates</code>	Determine the number of continuous states that an S-function has.
<code>ssGetNumDi scStates</code>	Determine the number of discrete states that an S-function has.
<code>ssGetNumDWork</code>	Get the number of data type work vectors used by a block

Macro	Description
ssGetNumIWork	Get the size of an S-function's integer work vector.
ssGetNumModes	Determine the size of an S-function's mode vector.
ssGetNumNonsampledZCs	Determine the number of nonsampled zero crossings that an S-function detects.
ssGetNumPWork	Determine the size of an S-function's pointer work vector.
ssGetNumRWork	Determine the size of an S-function's real-valued (real_T) work vector.
ssGetPWork	Get an S-function's pointer (void *) work vector.
ssGetRealDiscreteStates	Get the real (real_T) values of an S-function's discrete state vector.
ssGetRWork	Get an S-function's real-valued (real_T) work vector.
ssSetDWorkComplexSignal	Specify whether the elements of a data type work vector are real or complex.
ssSetDWorkDataType	Specify the data type of a data type work vector.
ssSetDWorkName	Specify the name of a data type work vector.
ssSetDWorkUsedAsDState	Specify that a data type work vector is used as a discrete state vector.
ssSetDWorkWidth	Specify the width of a data type work vector.
ssSetNumContStates	Specify the number of continuous states that an S-function has.

Macro	Description
<code>ssSetNumDiscreteStates</code>	Specify the number of discrete states that an S-function has.
<code>ssSetNumDWork</code>	Specify the number of data type work vectors used by a block.
<code>ssSetNumIWork</code>	Specify the size of an S-function's integer (<code>int_T</code>) work vector.
<code>ssSetNumModes</code>	Specify the number of operating modes that an S-function has.
<code>ssSetNumNonsampledZCs</code>	Specify the number of zero crossings that an S-function detects.
<code>ssSetNumPWork</code>	Specify the size of an S-function's pointer (<code>void *</code>) work vector.
<code>ssSetNumRWork</code>	Specify the size of an S-function's real (<code>real_T</code>) work vector.

Simulation Information

Macro	Description
ssGetT	Get the current base simulation time.
ssGetTaskTime	Get the current time for a task.
ssGetTFinal	Get the end time of the current simulation.
ssGetTStart	Get the start time of the current simulation.
ssIsMajorTimeStep	Determine if the current time step is a major time step.
ssIsMinorTimeStep	Determine if the current time step is a minor time step.
ssSetSolverNeedsReset	Ask Simulink to reset the solver.
ssSetStopRequested	Ask Simulink to terminate the simulation at the end of the current time step.

Function Call

Macro	Description
ssCallSystemWithTid	Execute a function-call subsystem connected to an S-function.
ssSetCallSystemOutput	Specify that an output port element issues a function call.

Data Type

Macro	Description
<code>ssGetDataTypeId</code>	Get the id for a data type.
<code>ssGetDataTypeName</code>	Get a data type's name.
<code>ssGetDataTypeSize</code>	Get a data type's size.
<code>ssGetDataTypeZero</code>	Get the zero representation of a data type.
<code>ssGetInputPortDataType</code>	Get the data type of an input port.
<code>ssGetNumDataTypes</code>	Get the number of data types defined by an S-function or the model.
<code>ssGetOutputPortDataType</code>	Get the data type of an output port.
<code>ssGetOutputPortSignal</code>	Get an output signal of any type except double.
<code>ssRegisterDataType</code>	Register a data type.
<code>ssSetDataTypeSize</code>	Specify the size of a data type.
<code>ssSetDataTypeZero</code>	Specify the zero representation of a data type.
<code>ssSetInputPortDataType</code>	Specify the data type of signals accepted by an input port.

Real-Time Workshop

Macro	Description
<code>ssWriteRTWParameter</code>	Write tunable parameters to the S-function's <code>model.rtw</code> file.
<code>ssWriteRTWParamSettings</code>	Write settings for the S-function's parameters to the <code>model.rtw</code> file.

Macro	Description
ssWriteRTWorkVect	Write the S-function's work vectors to the model . rtw file.
ssWriteRTWStr	Write a string to the S-function's model . rtw file.
ssWriteRTWStrParam	Write a string parameter to the S-function's model . rtw file.
ssWriteRTWScalarParam	Write a scalar parameter to the S-function's model . rtw file.
ssWriteRTWStrVectParam	Write a string vector parameter to the S-function's model . rtw file
ssWriteRTWVectParam	Write a Simulink vector parameter to the S-function's model . rtw file.
ssWriteRTW2dMatParam	Write a Simulink matrix parameter to the S-function's model . rtw file.
ssWriteRTWMxVectParam	Write a MATLAB vector parameter to the S-function's model . rtw file.
ssWriteRTWMx2dMatParam	Write a MATLAB matrix parameter to the S-function's model . rtw file.

Macro Reference

This section contains descriptions of each SimStruct macro.

ssCallExternalModeFcn

Purpose	Invoke the external mode function for an S-function.
Syntax	<code>void ssCallExternalModeFcn(SimStruct *S, SFunExtModeFcn *Fcn)</code>
Arguments	S SimStruct representing an S-function block or a Simulink model. Fcn external mode function
Description	Specifies the external mode function for S.
Languages	C
See Also	ssSetExternalModeFcn

Purpose	Specify that an output port is issuing a function call.
Syntax	<code>ssCallSystemWithTid(SimStruct *S, port_index, tid)</code>
Arguments	<p><code>S</code> SimStruct representing an S-function block or a Simulink model.</p> <p><code>port_index</code> Index of port that is issuing the function call</p> <p><code>tid</code> Task ID.</p>
Description	<p>Use in mdl Outputs to execute a function-call subsystem connected to the S-function. The invoking syntax is:</p> <pre> if (!ssCallSystemWithTid(S, <i>index</i>, <i>tid</i>)) { /* Error occurred which will be reported by Simulink */ return; } </pre>
Languages	C
See Also	<code>ssSetCallSystemOutput</code>

ssGetAbsTol

Purpose Get the absolute tolerances used by the model's variable step solver.

Syntax `real_T *ssGetAbsTol (SimStruct *S)`

Arguments S
SimStruct representing an S-function block.

Description Use in mdlStart to get the absolute tolerances used by the variable step solver for this simulation. Returns a pointer to an array that contains the tolerance for each continuous state.

Note Absolute tolerances are not allocated for fixed step solvers. Therefore, you should not invoke this macro until you have verified that the simulation is using a variable step solver, using `ssIsVariableStepSolver`.

Languages C, C++

Example

```
{
    int iSVarSolver = ssIsVariableStepSolver(S);

    if (iSVarSolver) {
        real_T *absTol = ssGetAbsTol(S);
        int nCStates = ssGetNumContStates(S);

        absTol[0] = whatever_value;
        ...
        absTol[nCStates-1] = whatever_value;
    }
}
```

See Also `ssGetStateAbsTol`, `ssIsVariableStepSolver`

Purpose	Get the address of a block's continuous states.
Ada Syntax	<code>ssGetContStateAddress(S : in SimStruct) return System.Address</code>
Arguments	S SimStruct representing an S-function block.
Description	Can be used in the simulation loop, <code>mdlInitializeConditions</code> , or <code>mdlStart</code> routines to get the address of the S-function's continuous state vector. This vector has length <code>ssGetNumContStates(S)</code> . Typically, this vector is initialized in <code>mdlInitializeConditions</code> and used in <code>mdlOutputs</code> .
Languages	Ada
See Also	<code>ssGetNumContStates</code> , <code>ssGetRealDiscreteStates</code> , <code>ssGetdX</code> , <code>mdlInitializeConditions</code> , <code>mdlStart</code>

ssGetContStates

Purpose	Get a block's continuous states.
Syntax	<code>real_T *ssGetContStates(SimStruct *S)</code>
Arguments	S SimStruct representing an S-function block.
Description	Can be used in the simulation loop, mdlInitializeConditions, or mdlStart routines to get the real_T continuous state vector. This vector has length <code>ssGetNumContStates(S)</code> . Typically, this vector is initialized in <code>mdlInitializeConditions</code> and used in <code>mdlOutputs</code> .
Languages	C
See Also	<code>ssGetNumContStates</code> , <code>ssGetRealDiscreteStates</code> , <code>ssGetdX</code> , <code>mdlInitializeConditions</code> , <code>mdlStart</code>

Purpose	Get the name of a data type.
Syntax	<code>char *ssGetDataTypeName(SimStruct *S, DTypeId id)</code>
Arguments	<code>S</code> SimStruct representing an S-function block. <code>id</code> ID of data type
Description	Returns the name of the data type specified by <code>id</code> , if <code>id</code> is valid. Otherwise, this macro returns <code>NULL</code> and reports an error. Because this macro reports any error that occurs, you do not need to use <code>ssSetErrorStatus</code> to report the error.
Example	The following example gets the name of a custom data type. <pre>const char *dtypeName = ssGetDataName(S, id); if(dtypeName == NULL) return;</pre>
Languages	C
See Also	<code>ssRegisterDataType</code>

ssGetDataTypeId

Purpose	Get the id of a data type.
Syntax	<code>DTypeId ssGetDataTypeId(SimStruct *S, char *name)</code>
Arguments	<code>S</code> SimStruct representing an S-function block. <code>name</code> Name of data type
Description	Returns the id of the data type specified by <code>name</code> , if <code>name</code> is a registered type name. Otherwise, this macro returns <code>INVALID_DTYPE_IDL</code> and reports an error. Because this macro reports any error that occurs, you do not need to use <code>ssSetErrorStatus</code> to report the error.
Languages	C
Example	The following example gets the id of the data type named <code>Color</code> . <pre>int_T id = ssGetDataTypeId(S, "Color"); if(id == INVALID_DTYPE_ID) return;</pre>
See Also	<code>ssRegisterDataType</code>

Purpose	Get the size of a custom data type.
Syntax	<code>GetDataTypeSize(SimStruct *S, DataType id)</code>
Arguments	<p><code>S</code> SimStruct representing an S-function block.</p> <p><code>id</code> ID of data type</p>
Description	Returns the size of the data type specified by <code>id</code> , if <code>id</code> is valid and the data types size has been set. Otherwise, this macro returns <code>INVALID_DTYPE_SIZE</code> and reports an error.

Note Because this macro reports any error that occurs when it is invoked, you do not need to use `ssSetErrorStatus` to report the error.

Languages	C
Example	<p>The following example gets the size of the <code>int16</code> data type.</p> <pre>int_T size = ssGetDataTypeSize(S, SS_INT16); if(size == INVALID_DTYPE_SIZE) return;</pre>
See Also	<code>ssSetDataTypeSize</code>

ssGetDataTypeZero

Purpose	Get the zero representation of a data type.
Syntax	<code>void* ssGetDataTypeZero(SimStruct *S, DTypeId id)</code>
Arguments	<p><code>S</code> SimStruct representing an S-function block.</p> <p><code>id</code> ID of data type</p>
Description	Returns a pointer to the zero representation of the data type specified by <code>id</code> , if <code>id</code> is valid and the data type's size has been set. Otherwise, this macro returns <code>NULL</code> and reports an error. Because this macro reports any error that occurs, you do not need to use <code>ssSetErrorStatus</code> to report the error.
Languages	C
Example	The following example gets the zero representation of a custom data type. <pre>const void *myZero = ssGetDataTypeZero(S, id); if(myZero == NULL) return;</pre>
See Also	<code>ssRegisterDataType</code> , <code>ssSetDataTypeSize</code> , <code>ssSetDataTypeZero</code>

Purpose	Get a block's discrete states.
Syntax	<code>real_T *ssGetDiscStates(SimStruct *S)</code>
Arguments	S SimStruct representing an S-function block.
Description	Returns a block's discrete state vector has an array of <code>real_T</code> elements of length <code>ssGetNumDiscStates(S)</code> . Typically, the state vector is initialized in <code>mdlInitializeConditions</code> , updated in <code>mdlUpdate</code> , and used in <code>mdlOutputs</code> . You can use this macro in the simulation loop, <code>mdlInitializeConditions</code> , or <code>mdlStart</code> routines.
Languages	C
See Also	<code>ssGetNumDiscStates</code> , <code>mdlInitializeConditions</code> , <code>mdlUpdate</code> , <code>mdlOutputs</code> , <code>mdlStart</code>

ssGetDTypeIdFromMxArray

Purpose Get the data type of an S-function parameter.

Syntax `DTypeId ssGetDTypeIdFromMxArray(const mxArray *m)`

Arguments `m`
MATLAB array representing the parameter

Description Returns the data type of an S-function parameter represented by a MATLAB array. This macro returns an enumerated type representing the data type. The enumerated type, `DTypeId`, is defined in `simstruc.h`. The following table shows the equivalency of Simulink, MATLAB, and C data types.

Simulink Data Type DTypeId	MATLAB DATA TYPE mxClassID	C- Data Type
SS_DOUBLE	mxDOUBLE_CLASS	real_T
SS_SINGLE	mxSINGLE_CLASS	real32_T
SS_INT8	mxINT8_CLASS	int8_T
SS_UINT8	mxUINT8_CLASS	uint8_T
SS_INT16	mxINT16_CLASS	int16_T
SS_UINT16	mxUINT16_CLASS	uint16_T
SS_INT32	mxINT32_CLASS	int32_T
SS_UINT32	mxUINT32_CLASS	uint32_T
SS_BOOLEAN	mxUINT8_CLASS+ Logical	boolean_T

`ssGetDTypeIdFromMxArray` returns `INVALID_DTYPE_ID` if the `mxClassID` does not map to any built-in Simulink data type id. For example, if `mxId == mxSTRUCT_CLASS`, the return value is `INVALID_DTYPE_ID`. Otherwise the return value is one of the enum values in `BuiltInDTypeId`. For example if `mxId == mxUINT16_CLASS`, the return value is `SS_UINT16`.

Note Use `ssGetSFcnParam` to get the array representing the parameter.

Example	See the example in <code>matlabroot/simulink/src/sfun_dtype_param.c</code> to learn how to use a data typed parameters in an S-function.
Languages	C
See Also	<code>ssGetSFcnParam</code>

ssGetDWorkComplexSignal

Purpose	Determine whether the elements of a data type work vector are real or complex numbers.
Syntax	<code>CSignal_T ssGetDWorkComplexSignal (SimStruct *S, int_T vector)</code>
Arguments	<p>S SimStruct representing an S-function block.</p> <p>vector Index of a data type work vector, where the index is one of 0, 1, 2, . . . <code>ssGetNumDWork(S)</code></p>
Description	Returns <code>COMPLEX_YES</code> if the specified vector contains complex numbers; otherwise, <code>COMPLEX_NO</code>
Languages	C, C++
See Also	<code>ssSetDWorkComplexSignal</code>

Purpose	Get the data type of a data type work vector.
Syntax	<code>DTypeId ssGetDWorkDataType(SimStruct *S, int_T vector)</code>
Arguments	<p>S SimStruct representing an S-function block.</p> <p>vector Index of a data type work vector, where the index is one of 0, 1, 2, . . . <code>ssGetNumDWork(S)</code></p>
Description	Returns the data type of the specified data type work vector.
Languages	C, C++
See Also	<code>ssSetDWorkDataType</code>

ssGetDWorkName

Purpose	Get the name of a data type work vector.
Syntax	<code>char_T *ssSetDWorkName(SimStruct *S, int_T vector)</code>
Arguments	<p>S SimStruct representing an S-function block.</p> <p>name Index of the work vector, where the index is one of 0, 1, 2, ... <code>ssGetNumDWork(S)</code></p>
Description	Returns the name of the specified data type work vector.
Languages	C, C++
See Also	<code>ssSetDWorkName</code>

Purpose	Determine whether a data type work vector is used as a discrete state vector.
Syntax	<code>int_T ssGetDWorkUsedAsDState(SimStruct *S, int_T vector)</code>
Arguments	<p>S SimStruct representing an S-function block.</p> <p>vector Index of a data type work vector, where the index is one of 0, 1, 2, . . . <code>ssGetNumDWork(S)</code></p>
Description	Returns <code>SS_DWORK_USED_AS_DSTATE</code> if this vector is used to store a block's discrete states.
Languages	C, C++
See Also	<code>sSetDWorkUsedAsDState</code>

ssGetDWorkWidth

Purpose	Get the size of a data type work vector.
Syntax	<code>int_T ssGetDWorkWidth(SimStruct *S, int_T vector)</code>
Arguments	<code>S</code> SimStruct representing an S-function block. <code>vector</code> Index of a work vector, where the index is one of 0, 1, 2, . . . <code>ssGetNumDWork(S)</code>
Description	Returns the number of elements in the specified work vector.
Languages	C, C++
See Also	<code>ssSetDWorkWidth</code>

Purpose	Get the derivatives of a block's continuous states.
Syntax	<code>ssGetContStates(SimStruct *S)</code>
Arguments	S SimStruct representing an S-function block.
Description	Use in mdlDerivatives to get the derivatives of a block's continuous states. This macro returns a vector that has length <code>ssGetNumContStates(S)</code> .
Languages	C
See Also	<code>ssGetNumContStates</code> , <code>ssGetContStates</code>

ssGetErrorStatus

Purpose	Get a string that identifies the last error.
C Syntax	<code>const char_T *ssGetContStates(SimStruct *S)</code>
Ada Syntax	<code>const char_T *ssGetContStates(SimStruct *S)</code>
Arguments	S SimStruct representing an S-function block.
Description	Returns a string that identifies the last error.
Languages	Ada, C
See Also	ssSetErrorString

Purpose	Determine the output port that is sharing this input port's buffer.
Syntax	<code>ssGetInputPortBufferDstPort(SimStruct *S, int_T inputPortIdx)</code>
Arguments	<p>S SimStruct representing an S-function block.</p> <p>inputPortIdx Index of port overwritten by an output port.</p>
Description	<p>Use in any run-time S-function callback routine to determine the output port that is overwriting the specified input port. This can be used when you have specified the following:</p> <ul style="list-style-type: none">• The input port and some output port on an S-Function are not test points (<code>ssSetInputPortTestPoint</code> and <code>ssSetOutputPortTestPoint</code>)• The input port is overwritable (<code>ssSetInputPortOverwritable</code>) <p>If you have this set of conditions, Simulink may use the same memory buffer for an input port and an output port. Simulink determines which ports share memory buffers. Use this function any time after model initialization to get the index of the output port that reuses the specified input port's buffer. If none of the S-function's output ports reuse this input port buffer, this macro returns <code>INVALID_PORT_IDX</code> (= -1).</p>
Languages	C
See Also	<code>ssSetNumInputPorts</code> , <code>ssSetInputPortOverwritable</code>

ssGetInputPortConnected

Purpose	Determine whether a port is connected to a nonvirtual block.
Syntax	<code>int_T ssGetInputPortConnected(SimStruct *S, int_T port)</code>
Arguments	<p><code>S</code> SimStruct representing an S-function block or a Simulink model.</p> <p><code>port</code> Port whose connection status is needed.</p>
Description	Returns true if the specified port on the block represented by <code>S</code> is connected to a nonvirtual block. Can be invoked anywhere except in <code>mdlInitializeSizes</code> or <code>mdlCheckParameters</code> . The S-function must have previously set the number of input ports in <code>mdlInitializeSizes</code> , using <code>ssSetNumInputPorts</code> .
Languages	C
See Also	<code>ssSetNumInputPorts</code>

Purpose	Get the numeric type (complex or real) of an input port.
Syntax	<code>DTypeId ssGetInputPortDataType(SimStruct *S, input_T port)</code>
Arguments	<code>S</code> SimStruct representing an S-function block. <code>port</code> Index of an input port
Description	Returns the numeric type of port:.
Languages	C
See Also	<code>ssSetInputPortComplexSignal</code>

ssGetInputPortDataType

Purpose	Get the data type of an input port.
C Syntax	<code>DTypeId ssGetInputPortDataType(SimStruct *S, input_T port)</code>
Ada Syntax	<code>function ssGetInputPortDataType(S : in SimStruct; port : in Integer := 0) return Integer;</code>
Arguments	<p>S SimStruct representing an S-function block or a Simulink model.</p> <p>port Index of an input port</p>
Description	Returns the data type of the input port specified by port.
Languages	Ada, C
See Also	<code>ssSetInputPortDataType</code>

Purpose	Specify information about the dimensionality of an input port.
Syntax	<code>DimInfo_T *ssGetInputPortDimensionInfo(SimStruct *S, int_T port)</code>
Arguments	<code>S</code> SimStruct representing an S-function block. <code>port</code> Index of an input port
Description	Gets the dimension information for port.
Languages	C, C++
See Also	<code>ssSetInputPortDimensionInfo</code>

ssGetInputPortDimensions

Purpose	Get the dimensions of the signal accepted by an input port.
Syntax	<code>int_T *ssGetInputPortDimensions(SimStruct *S, int_T port)</code>
Arguments	<code>S</code> SimStruct representing an S-function block. <code>port</code> Index of an input port
Description	Returns an array of integers that specifies the dimensions of the signal accepted by <code>port</code> , e.g., [4 2] for a 4-by-2 matrix array. The size of the dimensions array is equal to the number of signal dimensions accepted by the port, e.g., 1 for a vector signal or 2 for a matrix signal.
Languages	C
See Also	<code>ssGetInputPortNumDimensions</code>

Purpose	Determine whether a port has direct feedthrough.
C Syntax	<code>int_T ssGetInputPortDirectFeedThrough(SimStruct *S, int_T port)</code>
Ada Syntax	<pre>function ssGetInputPortDirectFeedThrough(S : in SimStruct; port : in Integer := 0) return Boolean;</pre>
Arguments	<p>S SimStruct representing an S-function block.</p> <p>port Index of port whose direct feedthrough property is required.</p>
Description	Use in any routine (except <code>mdlInitializeSizes</code>) to determine if an input port has direct feedthrough.
Languages	Ada, C
See Also	<code>ssSetInputPortDirectFeedThrough</code>

ssGetInputPortFrameData

Purpose	Determine if a port accepts signal frames.
Syntax	<code>int_T ssGetInputPortFrameData(SimStruct *S, int_T port)</code>
Arguments	<code>S</code> SimStruct representing an S-function block. <code>port</code> Index of an input port
Description	Returns one of the following <ul style="list-style-type: none">• -1 Port accepts either frame or unframed input.• 0 Port accepts unframed input only.• 1 Port accepts frame input only.
Languages	C
See Also	<code>ssSetInputPortFrameData</code> , <code>mdlSetInputPortFrameData</code>

ssGetInputPortNumDimensions

Purpose	Get the dimensionality of the signals accepted by an input port.
Syntax	<code>int_T ssGetInputPortNumDimensions(SimStruct *S, int_T port)</code>
Arguments	<code>S</code> SimStruct representing an S-function block. <code>port</code> Index of an input port
Description	Returns the number of dimensions of <code>port</code> or <code>DYNAMICALLY_SIZED</code> , if the number of dimensions is unknown.
Languages	C
See Also	<code>ssGetInputPortDimensions</code>

ssGetInputPortOffsetTime

Purpose	Get the offset time of an input port.
Syntax	<code>ssGetInputPortOffsetTime(SimStruct *S, inputPortIdx)</code>
Arguments	<code>S</code> SimStruct representing an S-function block. <code>inputPortIdx</code> Index of port whose offset time is required.
Description	Use in any routine (except <code>mdlInitializeSizes</code>) to determine the offset time of an input port. This should only be used if you have specified the sample times as port-based.
Languages	C
See Also	<code>ssSetInputPortOffsetTime</code> , <code>ssGetInputPortSampleTime</code>

Purpose	Determine whether an input port can be overwritten.
C Syntax	<code>int_T ssGetInputPortOverWritable(SimStruct *S, int_T port)</code>
Ada Syntax	<code>function ssGetInputPortOverWritable(S : in SimStruct; port : in Integer := 0) return Boolean;</code>
Arguments	<p>S SimStruct representing an S-function block or a Simulink model.</p> <p>port Index of the input port whose overwritability is being set.</p>
Description	Returns true if input port can be overwritten.
Languages	Ada, C
See Also	<code>ssSetInputPortOverWritable</code>

ssGetInputPortRealSignal

Purpose	Get the address of a real, contiguous signal entering an input port.
Syntax	<code>const real_T *ssGetInputPortRealSignal (SimStruct *S, inputPortIdx)</code>
Arguments	<p><code>S</code> SimStruct representing an S-function block.</p> <p><code>inputPortIdx</code> Index of port whose sample time is required.</p>
Description	Returns the address of a real signal on the specified input port. A method should use this macro only if the input signal is known to be real and <code>mdlIntializeSIZES</code> has specified that the elements of the input signal be contiguous, using <code>ssSetInputPortRequiredContiguous</code> .
Languages	C, C++
Example	<p>The following code demonstrates the use of <code>ssGetInputPortRealSignal</code>.</p> <pre>numInputPorts = ssGetNumInputPorts(S); for (i = 0; i < numInputPorts; i++) { int_nu = ssGetInputPortWidth(S, i); if (ssGetInputPortRequiredContiguous(S, i)) { const real_T *u = ssGetInputPortRealSignal (S, i); UseInputVectorInSomeFunction(u, nu); } else { InputPtrsType u = ssGetInputPortSignalPtrs(S, i); for (j = 0; j < nu; j++) { UseInputInSomeFunction(*u[j]); } } }</pre>
See Also	<code>ssSetInputPortRequiredContiguous</code> , <code>ssGetInputPortSignal</code> , <code>mdlIntializeSIZES</code>

Purpose	Get pointers to signals of type <code>double</code> connected to an input port.
Syntax	<code>InputRealPtrsType ssGetInputPortRealSignalPtrs(SimStruct *S, int_T port)</code>
Arguments	<p><code>S</code> SimStruct representing an S-function block.</p> <p><code>port</code> Index of port whose signal is required.</p>
Description	Returns pointers to the elements of a signal of type <code>double</code> connected to <code>port</code> . The input port index starts at 0 and ends at the number of input ports minus 1. This macro returns a pointer to an array of pointers to the <code>real_T</code> input signal elements. The length of the array of pointers is equal to the width of the input port.
Languages	C
Example	<p>The following example read all input port signals.</p> <pre> int_T i, j; int_T nInputPorts = ssGetNumInputPorts(S); for (i = 0; i < nInputPorts; i++) { InputRealPtrsType uPtrs = ssGetInputPortRealSignal(S, i); int_T nu = ssGetInputPortWidth(S, i); for (j = 0; j < nu; j++) { <i>SomeFunctionToUseInputSignalElement</i>(*uPtrs [j]); } } </pre>
See Also	<code>ssGetInputPortWidth</code> , <code>ssGetInputPortDataType</code> , <code>ssGetInputPortSignalPtrs</code>

ssGetInputPortRequiredContiguous

Purpose Determine whether the signal elements entering a port must be contiguous.

Syntax `int_T ssSetInputPortRequiredContiguous(SimStruct *S, int_T port)`

Arguments

- `S`
SimStruct representing an S-function block or a Simulink model.
- `port`
Index of an input port

Description Returns true if the signal elements entering the specified port must occupy contiguous areas of memory. If the elements are contiguous, a method can access the elements of the signal simply by incrementing the signal pointer returned by `ssGetInputPortSignal`.

Note The default setting for this flag is false. Hence, the default method for accessing the input signals is `ssGetInputSignalPtrs`.

Languages C, C++

See Also `ssSetInputPortRequiredContiguous`, `ssGetInputPortSignal`, `ssGetInputPortSignalPtrs`

Purpose	Determine whether memory allocated to input port is reusable.
Syntax	<code>int_T ssGetInputPortReusable(SimStruct *S, int_T port)</code>
Arguments	<code>S</code> SimStruct representing an S-function block or a Simulink model. <code>inputPortIdx</code> Index of the input port
Description	Returns TRUE if input port memory buffer can be reused by other signals in the model.
Languages	C, C++
See Also	<code>ssSetInputPortReusable</code>

ssGetInputPortSampleTime

Purpose	Get the sample time of an input port.
Syntax	<code>ssGetInputPortSampleTime(SimStruct *S, inputPortIdx)</code>
Arguments	<code>S</code> SimStruct representing an S-function block. <code>inputPortIdx</code> Index of port whose sample time is required.
Description	Use in any routine (except <code>mdlInitializeSizes</code>) to determine the sample time of an input port. You should use this macro only if you have specified the sample times as port-based.
Languages	C
See Also	<code>ssSetInputPortSampleTime</code> , <code>ssGetInputPortOffsetTime</code>

Purpose	Get the sample time index of an input port.
Syntax	<code>int_T ssGetInputPortSampleTimeIndex(SimStruct *S, int_T inputPortIdx)</code>
Arguments	<code>S</code> SimStruct representing an S-function block or a Simulink model. <code>inputPortIdx</code> Index of the input port whose sample time index is being set.
Description	Returns the index of the sample time for the port.
Languages	C, C++
See Also	<code>ssSetInputPortSampleTimeIndex</code>

ssGetInputPortSignal

Purpose	Get the address of a contiguous signal entering an input port.
Syntax	<code>const void* ssGetInputPortSignal (SimStruct *S, inputPortIdx)</code>
Arguments	<code>S</code> SimStruct representing an S-function block. <code>inputPortIdx</code> Index of port whose sample time is required.
Description	Returns the address of the specified input port. A method should use this macro only if mdlIntializesSizes has specified that the elements of the input signal be contiguous, using <code>ssSetInputPortRequiredContiguous</code> .
Languages	C, C++
Example	The following code demonstrates the use of <code>ssGetInputPortSignal</code> .

```
numInputPorts = ssGetNumInputPorts(S);
for (i = 0; i < numInputPorts; i++) {
    int_nu = ssGetInputPortWidth(S, i);

    if ( ssGetInputPortRequiredContiguous(S, i) ) {

        const void *u = ssGetInputPortSignal (S, i);
        UseInputVectorInSomeFunction(u, nu);

    } else {

        InputPtrsType u = ssGetInputPortSignalPtrs(S, i);
        for (j = 0; j < nu; j++) {
            UseInputInSomeFunction(*u[j]);
        }
    }
}
```

If you know that the inputs are always `real_T` signals, the `ssGetInputPortSignal` line in the above code snippet would be:

```
const real_T *u = ssGetInputPortRealSignal (S, i);
```

See Also

[ssSetInputPortRequiredContinuous](#), [ssGetInputPortRealSignal](#)

ssGetInputPortSignalAddress

Purpose	Get address of an input port's signal.
Syntax	<pre>function ssGetInputPortSignalAddress(S : in SimStruct; port : in Integer := 0) return System.Address;</pre>
Arguments	<p>S SimStruct representing an S-function block.</p> <p>port Index of an input port</p>
Description	Returns the address of the signal connected to port.
Languages	Ada
Example	<p>The following code gets the signal connected to a block's input port.</p> <pre>uWidth : Integer := ssGetInputPortWidth(S, 0); U : array(0 .. uWidth-1) of Real_T; for U' Address use ssGetInputPortSignalAddress(S, 0);</pre>
See Also	sGetInputPortWidth

Purpose	Get pointers to an input port's signal elements.
Syntax	<code>InputPtrsType ssGetInputPortSignalPtrs(SimStruct *S, int_T port)</code>
Arguments	<p><code>S</code> SimStruct representing an S-function block.</p> <p><code>port</code> Index of an input port</p>
Description	<p>Returns a pointer to an array of signal element pointers for the specified input port. For example, if the input port width is 5, this function returns a pointer to a 5-element pointer array. Each element in the pointer array points to the specific element of the input signal.</p> <p>You must use <code>ssGetInputPortRealSignalPtrs</code> to get pointers to signals of type <code>double (real_T)</code>.</p>
Languages	C
Example	<p>Assume that the input port data types are <code>int8_T</code>.</p> <pre> int_T nInputPorts = ssGetNumInputPorts(S); for (i = 0; i < nInputPorts; i++) { InputPtrsType u = ssGetInputPortSignalPtrs(S, i); InputInt8PtrsType uInt8 = (InputInt8PtrsType)u; int_T nu = ssGetInputPortWidth(S, i); for (j = 0; j < nu; j++) { /* u[j] is an int8_T pointer that points to the j-th element of the input signal. */ UseInputInSomeFunction(*u[j]); } } </pre>
See Also	<code>ssGetInputPortRealSignalPtrs</code>

ssGetInputPortWidth

Purpose	Get the width of an input port.
C Syntax	<code>int_T ssGetInputPortWidth(SimStruct *S, int_T port)</code>
Ada Syntax	<code>function ssGetInputPortWidth(S : in SimStruct; port : in Integer := 0) return Integer;</code>
Arguments	<p>S SimStruct representing an S-function block.</p> <p>port Index of port whose width is required.</p>
Description	Get the input port number of elements. If the input port is a 1-D array with w elements, this function returns w. If the input port is an M-by-N matrix, this function returns m*n. If m or n is unknown, this function returns DYNAMICALLY_SIZED. Use in any routine (except mdlInitializeSizes) to determine the width of an input port.
Languages	Ada, C
See Also	ssSetInputPortWidth

Purpose	Get a block's integer work vector.
Syntax	<code>ssGetIWork(SimStruct *S)</code>
Arguments	<code>S</code> SimStruct representing an S-function block.
Description	Returns the integer work vector used by the block represented by <code>S</code> . The vector consists of elements of type <code>int_T</code> and is of length <code>ssGetNumRWork(S)</code> . Typically, this vector is initialized in <code>mdlStart</code> or <code>mdlInitializeConditions</code> , updated in <code>mdlUpdate</code> , and used in <code>mdlOutputs</code> . You can use this macro in the simulation loop, <code>mdlInitializeConditions</code> , or <code>mdlStart</code> routines.
Languages	C
See Also	<code>ssGetNumIWork</code>

ssGetModelName

Purpose	Get the model name.
Syntax	<code>ssGetModelName(SimStruct *S)</code>
Arguments	S SimStruct representing an S-function block or a Simulink model.
Description	If S is a SimStruct for an S-function block, this macro returns the name of the S-function MEX-file associated with the block. If S is the root SimStruct, this macro returns the name of the Simulink block diagram.
Languages	C
See Also	<code>ssGetPath</code>

Purpose	Get the mode vector.
Syntax	<code>int_T *ssGetModeVector(SimStruct *S)</code>
Arguments	S SimStruct representing an S-function block.
Description	<p>Returns a pointer (<code>int_T *</code>) to the mode vector.</p> <p>This vector has length <code>ssGetNumModes(S)</code>. Typically, this vector is initialized in <code>mdlInitializeConditions</code> if the default value of zero isn't acceptable. It is then used in <code>mdlOutputs</code> in conjunction with nonsampled zero crossings to determine when the output function should change mode. For example consider an absolute value function. When the input is negative, negate it to create a positive value, otherwise take no action. This function has two modes. The output function should be designed not to change modes during minor time steps. The mode vector may also be used in the <code>mdlZeroCrossings</code> routine to determine the current mode.</p>
Languages	C, C++
See Also	<code>ssSetNumModes</code>

ssGetModeVectorValue

Purpose	Get an element of a block's mode vector.
Syntax	<code>int_T ssGetModeVectorValue(SimStruct *S, element)</code>
Arguments	<code>S</code> SimStruct representing an S-function block. <code>elementx</code> Index of a mode vector element
Description	Returns the specified mode vector element.
Languages	C, C++
See Also	<code>ssSetModeVectorValue</code> , <code>ssGetModeVector</code>

Purpose	Get the zero-crossing signal values.
Syntax	<code>ssGetNumNonSampledZCs(SimStruct *S)</code>
Arguments	S SimStruct representing an S-function block.
Description	Returns a pointer to the vector containing the current values of the signals that the variable-step solver monitors for zero crossings. The variable step solver tracks the signs of these signals to bracket points where they cross zero. The solver then takes simulation time steps at the points where the zero crossings occur. This vector has length <code>ssGetNumNonSampledZCs(S)</code> .
Example	<p>The following excerpt from <code>matlabroot/simulink/src/sfun_zc.c</code> illustrates usage of this macro to update the zero-crossing array in the mdl <code>ZeroCrossings</code> callback function.</p> <pre>static void mdlZeroCrossings(SimStruct *S) { int_T i; real_T *zcSignals = ssGetNonsampledZCs(S); InputRealPtrsType uPtrs = ssGetInputPortRealSignalPtrs(S, 0); int_T nZCSignals = ssGetNumNonSampledZCs(S); for (i = 0; i < nZCSignals; i++) { zcSignals[i] = *uPtrs[i]; } }</pre>
Languages	C
See Also	<code>ssGetNumNonSampledZCs</code>

ssGetNumContStates

Purpose	Get the number of continuous states that a block has.
C Syntax	<code>int_T ssGetNumContStates(SimStruct *S)</code>
Ada Syntax	<code>function ssGetNumContStates(S : in SimStruct) return Integer;</code>
Arguments	S SimStruct representing an S-function block or model.
Description	Returns the number of continuous states in the block or model represented by S. You can use this macro in any routine except mdl InitializeSes.
Languages	Ada, C
See Also	ssSetNumContStates, ssGetNumDiscreteStates, ssGetContStates

Purpose	Get number of data types registered for this simulation, including built-in types.
Syntax	<code>int_T ssGetNumDataTypes(SimStruct *S)</code>
Arguments	S SimStruct representing an S-function block.
Description	Returns the number of data types registered for this simulation. This includes all custom data types registered by custom S-function blocks and all built-in data types. <hr/> <p>Note S-functions register their data types in their implementations of the <code>mdlInitializeSizes</code> callback function. Therefore, to ensure that this macro returns an accurate count, your S-function should invoke it only after the point in the simulation at which Simulink invokes the <code>mdlInitializeSizes</code> callback function.</p> <hr/>
Languages	C
See Also	<code>ssRegisterDataType</code>

ssGetNumDiscStates

Purpose	Get the number of discrete states that a block has.
Syntax	<code>int_T ssGetNumDiscStates(SimStruct *S)</code>
Arguments	S SimStruct representing an S-function block.
Description	Use in any routine (except <code>mdlInitializeSizes</code>) to determine the number of discrete states that the S-function has.
Languages	C
See Also	<code>ssSetNumDiscStates</code> , <code>ssGetNumContStates</code>

Purpose	Get the number of data type work vectors used by a block.
Syntax	<code>int_T ssGetNumDWork(SimStruct *S)</code>
Arguments	S SimStruct representing an S-function block.
Description	Returns the number of data type work vectors used by S.
Languages	C, C++
See Also	<code>ssSetNumDWork</code>

ssGetNumInputPorts

Purpose	Get the number of input ports that a block has.
C Syntax	<code>int_T ssGetNumInputPorts(SimStruct *S)</code>
Ada Syntax	<code>function ssGetNumInputPorts(S : in SimStruct) return Integer;</code>
Arguments	S SimStruct representing an S-function block.
Description	Use in any routine (except <code>mdlInitializeSizes</code>) to determine how many input ports a block has.
Languages	Ada, C
See Also	<code>ssGetNumOutputPorts</code>

Purpose	Get the size of a block's integer work vector.
Syntax	<code>int_T ssGetNumIWork(SimStruct *S)</code>
Arguments	S SimStruct representing an S-function block.
Description	Returns the size of the integer (int_T) work vector used by the block represented by S. You can use this macro in any routine except <code>mdlInitializeSizes</code>
Languages	C
See Also	<code>ssSetNumIWork</code> , <code>ssGetNumRWork</code>

ssGetNumModes

Purpose	Get the size of the mode vector.
Syntax	<code>ssGetNumModes(SimStruct *S)</code>
Arguments	S SimStruct representing an S-function block.
Description	Returns the size of the modes vector. You can use this macro in any routine except <code>mdlInitializeSizes</code> .
Languages	C
See Also	<code>ssSetNumNonsampledZCs</code> , <code>ssGetNonsampledZCs</code>

Purpose	Get the size of the zero-crossing vector.
Syntax	<code>ssGetNumNonSampledZCs(SimStruct *S)</code>
Arguments	S SimStruct representing an S-function block.
Description	Returns the size of the zero-crossing vector. You can use this macro in any routine except <code>mdlInitializeSizes</code> .
Languages	C
See Also	<code>ssSetNumNonSampledZCs</code> , <code>ssGetNonSampledZCs</code>

ssGetNumOutputPorts

Purpose	Get the number of output ports that a block has.
C Syntax	<code>int_T ssGetNumOutputPorts(SimStruct *S)</code>
Ada Syntax	<code>function ssGetNumOutputPorts(S : in SimStruct) return Integer;</code>
Arguments	S SimStruct representing an S-function block.
Description	Use in any routine (except <code>mdlInitializeSizes</code>) to determine how many output ports a block has.
Languages	Ada, C
See Also	<code>ssGetNumInputPorts</code>

Purpose	Get the number of parameters that this block has.
Syntax	<code>function ssGetNumParameters(S : in SimStruct) return Integer;</code>
Arguments	S SimStruct representing an S-function block.
Description	Returns the number of parameters that this block has.
Languages	Ada
See Also	<code>ssGetParameterName</code>

ssGetNumRunTimeParams

Purpose	Get the number of run-time parameters created by this S-function.
Syntax	<code>int_T ssGetNumRunTimeParams(SimStruct *S)</code>
Arguments	S SimStruct representing an S-function block.
Description	Use this function to get the number of run-time parameters created by this S-function.
Languages	C
See Also	<code>ssSetNumRunTimeParams</code>

Purpose	Get the size of a block's pointer work vector.
Syntax	<code>int_T ssGetNumPWork(SimStruct *S)</code>
Arguments	S SimStruct representing an S-function block.
Description	Returns the size of the pointer work vector used by the block represented by S. You can use this macro in any routine except <code>mdlInitializeSizes</code> .
Languages	C
See Also	<code>ssSetNumPWork</code>

ssGetNumRWork

Purpose	Get the size of a block's floating-point work vector.
Syntax	<code>int_T ssGetNumRWork(SimStruct *S)</code>
Arguments	S SimStruct representing an S-function block.
Description	Returns the size of the floating-point (real_T) work vector used by the block represented by S. You can use this macro in any routine except <code>mdlInitializeSizes</code>
Languages	C
See Also	<code>ssSetNumRWork</code>

Purpose	Get the number of sample times that a block has.
Syntax	<code>int_T ssGetNumOutputPorts(SimStruct *S)</code>
Arguments	S SimStruct representing an S-function block.
Description	Use in any routine (except <code>mdlInitializeSizes</code>) to determine the number of sample times S has.
Languages	C
See Also	<code>ssSetNumSampleTimes</code>

ssGetNumSFcnParams

Purpose	Get the number of parameters that an S-function block expects.
Syntax	<code>int_T ssGetNumSFcnParams(SimStruct *S)</code>
Arguments	S SimStruct representing an S-function block.
Description	Returns the number of parameters that S expects the user to enter.
Languages	C
See Also	<code>ssSetSFcnNumSFcnParams</code>

Purpose	Determine whether the output of this block is connected to a Merge block.
Syntax	<code>int_T ssGetOutputPortBeingMerged(SimStruct *S, int_T port)</code>
Arguments	<p>S SimStruct representing an S-function block or a Simulink model.</p> <p>port Index of the output port</p>
Description	Returns TRUE if this output port signal is being merged with other signals (this happens if the S-function block output port is directly or via connection type blocks is connected to a Merge block). This macro returns the correct answer in and after the S-function's <code>mdlSetWorkWidths</code> method.
Languages	C, C++
See Also	<code>mdlSetWorkWidths</code>

ssGetOutputPortComplexSignal

Purpose	Get the numeric type (complex or real) of an output port.
Syntax	<code>DTypeId ssGetOutputPortDataType(SimStruct *S, input_T port)</code>
Arguments	<code>S</code> SimStruct representing an S-function block. <code>port</code> Index of an output port
Description	Returns the numeric type of port: <code>COMPLEX_NO</code> (real signal), <code>COMPLEX_YES</code> (complex signal) or <code>COMPLEX_INHERITED</code> (dynamically determined).
Languages	C
See Also	<code>ssSetOutputPortComplexSignal</code>

Purpose	Get the data type of an output port.
C Syntax	<code>DTypeId ssSetOutputPortDataType(SimStruct *S, input_T port)</code>
Ada Syntax	<pre>function ssGetOutputPortDataType (S : in SimStruct; port : in Integer := 0) return Integer;</pre>
Arguments	<p>S SimStruct representing an S-function block or a Simulink model.</p> <p>port Index of an output port</p>
Description	Returns the data type of the output port specified by port.
Languages	Ada, C
See Also	<code>ssSetOutputPortDataType</code>

ssGetOutputPortDimensions

Purpose	Get the dimensions of the signal leaving an output port.
Syntax	<code>int_T *ssGetOutputPortDimensions(SimStruct *S, int_T port)</code>
Arguments	<code>S</code> SimStruct representing an S-function block. <code>port</code> Index of an output port
Description	Returns an array of integers that specifies the dimensions of the signal leaving <code>port</code> , e.g., [4 2] for a 4-by-2 matrix array. The size of the dimensions array is equal to the number of signal dimensions accepted by the port, e.g., 1 for a vector signal or 2 for a matrix signal.
Languages	C
See Also	<code>ssGetOutputPortNumDimensions</code>

Purpose	Determine if a port accepts signal frames.
Syntax	<code>int_T ssGetOutputPortFrameData(SimStruct *S, int_T port)</code>
Arguments	<p>S SimStruct representing an S-function block.</p> <p>port Index of an output port</p>
Description	<p>Returns one of the following</p> <ul style="list-style-type: none">• -1 Port outputs either frame or unframed data.• 0 Port outputs unframed data only.• 1 Port outputs frame data only.
Languages	C
See Also	<code>ssSetOutputPortFrameData</code> , <code>mdlSetOutputPortFrameData</code>

ssGetOutputPortNumDimensions

Purpose	Get the offset time of an output port.
Syntax	<code>int_T ssGetOutputPortNumDimensions(SimStruct *S, int_T port)</code>
Arguments	<code>S</code> SimStruct representing an S-function block. <code>port</code> Index of output port.
Description	Returns number of dimensions of port.
Languages	C
See Also	<code>ssSetOutputPortDimensionInfo</code>

Purpose	Get the offset time of an output port.
Syntax	<code>real_T ssGetOutputPortOffsetTime(SimStruct *S, outputPortIdx)</code>
Arguments	<code>S</code> SimStruct representing an S-function block. <code>outputPortIdx</code> Index of output port.
Description	Use in any routine (except <code>mdlInitializeSizes</code>) to determine the sample time of an output port. This macro should only be used if you have specified port-based sample times.
Languages	C
See Also	<code>ssSetOutputOffsetTime</code> , <code>ssGetOutputPortSampleTime</code>

ssGetOutputPortRealSignal

Purpose	Get a pointer to an output signal of type <code>double</code> (<code>real_T</code>).
Syntax	<code>real_T *ssGetOutputPortRealSignal(SimStruct *S, int_T port)</code>
Arguments	<code>S</code> SimStruct representing an S-function block. <code>port</code> Index of output port.
Description	Use in any simulation loop routine, <code>mdlInitializeConditions</code> , or <code>mdlStart</code> to access an output port signal where the output port index starts at 0 and must be less than the number of output ports. This returns a contiguous <code>real_T</code> vector of length equal to the width of the output port.
Example	To write to all output ports, you would use <pre>int_T i, j; int_T nOutputPorts = ssGetNumOutputPorts(S); for (i = 0; i < nOutputPorts; i++) { real_T *y = ssGetOutputPortRealSignal(S, i); int_T ny = ssGetOutputPortWidth(S, i); for (j = 0; j < ny; j++) { y[j] = <i>SomeFunctionToFillInOutput</i>(); } }</pre>
Languages	C
See Also	<code>ssGetInputPortRealSignalPtrs</code>

Purpose	Determine whether memory allocated to output port is reusable.
Syntax	<code>int_T ssGetOutputPortReusable(SimStruct *S, int_T port)</code>
Arguments	<code>S</code> SimStruct representing an S-function block or a Simulink model. <code>port</code> Index of the output port
Description	Returns TRUE if output port memory buffer can be reused by other signals in the model.
Languages	C, C++
See Also	<code>ssSetOutputPortReusable</code>

ssGetOutputPortSampleTime

Purpose	Get the sample time of an output port.
Syntax	<code>ssGetOutputPortSampleTime(SimStruct *S, outputPortIdx)</code>
Arguments	<code>S</code> SimStruct representing an S-function block. <code>outputPortIdx</code> Index of output port.
Description	Use in any routine (except <code>mdlInitializeSizes</code>) to determine the sample time of an output port. This macro should only be used if you have specified port-based sample times.
Languages	C
See Also	<code>ssSetOutputSampleTime</code>

Purpose Get the vector of signal elements emitted by an output port.

Syntax void *ssGetOutputPortSignal (SimStruct *S, int_T port)

Arguments S
SimStruct representing an S-function block.

port
Index of output port.

Description Returns a pointer to the vector of signal elements output by port.

Note If the port outputs a signal of type double (real_T), you must use ssGetOutputPortRealSignal to get the signal vector.

Example Assume that the output port data types are int16_T.

```
nOutputPorts = ssGetNumOutputPorts(S);
for (i = 0; i < nOutputPorts; i++) {
    int16_T *y      = (int16_T *)ssGetOutputPortSignal (S, i);
    int_T   ny      = ssGetOutputPortWidth(S, i);
    for (j = 0; j < ny; j++) {
        SomeFunctionToFillInOutput(y[j]);
    }
}
```

Languages C

See Also ssGetOutputPortRealSignal

ssGetOutputPortSignalAddress

Purpose	Get address of an output port's signal.
Syntax	<code>ssGetOutputPortSignalAddress(S : in SimStruct; port : in Integer := 0) return System.Address</code>
Arguments	<code>S</code> SimStruct representing an S-function block. <code>port</code> Index of an output port
Description	Returns the address of the signal connected to port.
Languages	Ada
Example	The following code gets the signal connected to a block's input port. <pre>yWidth : Integer := ssGetOutputPortWidth(S, 0); Y : array(0 .. yWidth-1) of Real_T; for Y' Address use ssGetOutputPortSignalAddress(S, 0);</pre>
See Also	<code>ssGetOutputPortWidth</code>

Purpose	Get the width of an output port.
C Syntax	<code>int_T ssGetOutputPortWidth(SimStruct *S, int_T port)</code>
Ada Syntax	<code>function ssGetOutputPortWidth(S : in SimStruct; port : in Integer := 0) return Integer;</code>
Arguments	<p>S SimStruct representing an S-function block.</p> <p>outputPortIdx Index of output port.</p>
Description	Use in any routine (except <code>mdlInitializeSizes</code>) to determine the width of an output port where the output port index starts at 0 and must be less than the number of output ports.
Languages	Ada, C
See Also	<code>ssSetOutputPortWidth</code>

ssGetPath

Purpose	Get the path of a block.
C Syntax	<code>const char_T *ssGetPath(SimStruct *S)</code>
Ada Syntax	<code>function ssGetPath(S : in SimStruct) return String;</code>
Arguments	S SimStruct representing an S-function block or a Simulink model.
Description	If S is an S-function block, this macro returns the full Simulink path to the block. If S is the root SimStruct of the model, this macro returns the model name. In a C MEX S-function, in <code>mdlInitializeSizes</code> , if <pre>strcmp(ssGetModelName(S), ssGetPath(S))==0</pre> the S-function is being called from MATLAB and is not part of a simulation.
Languages	Ada, C
See Also	<code>ssGetModelName</code>

Purpose	Get the parent of a SimStruct.
Syntax	SimStruct *ssGetParentSS(SimStruct *S)
Arguments	S SimStruct representing an S-function block or a Simulink model.
Description	Returns the parent SimStruct of S, or NULL if S is the root SimStruct.

Note There is one SimStruct for each S-Function in your model and one for the model itself. The structures are arranged as a tree with the model SimStruct as the root. User-written S-functions should not use the ssGetParentSS macro.

Languages	C
See Also	ssGetRoot

ssGetPlacementGroup

Purpose	Get the name of the placement group of a block.
Syntax	<code>const char *ssGetPlacementGroup(SimStruct *S)</code>
Arguments	S SimStruct representing an S-function block or a Simulink model. The block must be either a source block (i.e., a block without input ports) or a sink block (i.e., a block without output ports).
Description	Use this macro in <code>mdlInitializeSizes</code> to get the name of this block's placement group. <hr/> <p>Note This macro is typically used to create Real-Time Workshop device driver blocks.</p> <hr/>
Languages	C
See Also	<code>ssGetPlacementGroup</code>

Purpose	Get a block's pointer work vector.
Syntax	<code>ssGetPWork(SimStruct *S)</code>
Arguments	<code>S</code> SimStruct representing an S-function block.
Description	Returns the pointer work vector used by the block represented by <code>S</code> . The vector consists of elements of type <code>void *</code> and is of length <code>ssGetNumRWork(S)</code> . Typically, this vector is initialized in <code>mdlStart</code> or <code>mdlInitializeConditions</code> , updated in <code>mdlUpdate</code> , and used in <code>mdlOutputs</code> . You can use this macro in the simulation loop, <code>mdlInitializeConditions</code> , or <code>mdlStart</code> routines.
Languages	C
See Also	<code>ssGetNumPWork</code>

ssGetRealDiscStates

Purpose	Get a block's discrete state vector.
Syntax	<code>real_T *ssGetRealDiscStates(SimStruct *S)</code>
Arguments	S SimStruct representing an S-function block.
Description	Same as <code>ssGetDiscStates</code> .
Languages	C
See Also	<code>ssGetDiscStates</code>

Purpose	Get the root of a SimStruct hierarchy.
Syntax	<code>SimStruct *ssGetRootSS(SimStruct *S)</code>
Arguments	S SimStruct representing an S-function block or a Simulink model.
Description	Returns the root of the SimStruct hierarchy containing S.
Languages	C
See Also	ssGetParent

ssGetRunTimeParamInfo

Purpose	Gets the attributes of a run-time parameter.
Syntax	<code>ssParamRec *ssSetRunTimeParamInfo(SimStruct *S, int_T param)</code>
Arguments	<code>S</code> SimStruct representing an S-function block. <code>param</code> Index of a run-time parameter
Description	Returns the attributes of the run-time parameter specified by <code>param</code> . See the documentation for <code>ssSetRunTimeParamInfo</code> for a description of the <code>ssParamRec</code> structure returned by this function.
Languages	C
See Also	<code>ssSetRunTimeParamInfo</code>

Purpose	Get a block's floating-point work vector.
Syntax	<code>ssGetRWork(SimStruct *S)</code>
Arguments	<code>S</code> SimStruct representing an S-function block.
Description	Returns the floating-point work vector used by the block represented by <code>S</code> . The vector consists of elements of type <code>real_T</code> and is of length <code>ssGetNumRWork(S)</code> . Typically, this vector is initialized in <code>mdlStart</code> or <code>mdlInitializeConditions</code> , updated in <code>mdlUpdate</code> , and used in <code>mdlOutputs</code> . You can use this macro in the simulation loop, <code>mdlInitializeConditions</code> , or <code>mdlStart</code> routines.
Languages	C
See Also	<code>ssGetNumRWork</code>

ssGetSampleTimeOffset

Purpose	Get the period of the current sample time.
Syntax	<code>function ssGetSampleTimeOffset(S : in SimStruct) return time_T;</code>
Arguments	S SimStruct representing an S-function block.
Description	Returns the offset of the current sample time.
Languages	Ada
See Also	<code>ssGetSampleTimePeriod</code>

Purpose	Get the period of the current sample time.
Syntax	<code>function ssGetSampleTimePeriod(S : in SimStruct) return time_T;</code>
Arguments	S SimStruct representing an S-function block.
Description	Returns the period of the current sample time.
Languages	Ada
See Also	<code>ssGetSampleTimeOffset</code>

ssGetSFcnParam

Purpose	Get a parameter of an S-function block.
Syntax	<code>const mxArray *ssGetSFcnParam(SimStruct *S, int_T index)</code>
Arguments	<p><code>S</code> SimStruct representing an S-function block.</p> <p><code>index</code> Index of the parameter to be returned.</p>
Description	Use in any routine to access a parameter entered in the S-function's block dialog box where <i>index</i> starts at 0 and is less than <code>ssGetSFcnParamsCount(S)</code> .
Languages	C
See Also	<code>ssGetSFcnParamsCount</code>

Purpose	Get the number of block dialog parameters that an S-function block has.
Syntax	<code>ssGetSFcnParamsCount(SimStruct *S)</code>
Arguments	S SimStruct representing an S-function block.
Description	Returns the number of parameters that a user can set for the block represented by S.
Languages	C
See Also	<code>ssGetNumSFcnParams</code>

ssGetSimMode

Purpose	Get the simulation mode an S-function block.
Syntax	ssGetSimMode(SimStruct *S)
Arguments	S SimStruct representing an S-function block or a Simulink model.
Description	Returns the simulation mode of the block represented by S: <ul style="list-style-type: none">• SS_SIMMODE_NORMAL Running in a normal Simulink simulation• SS_SIMMODE_SIZES_CALL_ONLY Invoked by editor to obtain number of ports• SS_SIMMODE_RTWGEN Generating code• SS_SIMMODE_EXTERNAL External mode simulation
Languages	C
See Also	ssGetSolverName

Purpose	Get the name of the solver being used to solve the S-function.
Syntax	<code>ssGetSolverName(SimStruct *S)</code>
Arguments	S SimStruct representing an S-function block or a Simulink model.
Description	Returns a pointer (<code>char *</code>) to the name of the solver being used to solve the S-function represented by S.
Languages	C
See Also	<code>ssGetSimMode</code> , <code>ssIsVariableStepSolver</code>

ssGetStateAbsTol

Purpose Get the absolute tolerance used by the model's variable step solver for a specified state.

Syntax `real_T ssGetStateAbsTol (SimStruct *S, int_T state)`

Arguments S
SimStruct representing an S-function block.

Description Use in mdlStart to get the absolute tolerance for a particular state.

Note Absolute tolerances are not allocated for fixed step solvers. Therefore, you should not invoke this macro until you have verified that the simulation is using a variable step solver, using `ssIsVariableStepSolver`.

Languages C, C++

See Also `ssGetAbsTol`, `ssIsVariableStepSolver`

Purpose	Get the current simulation time.
C Syntax	<code>ssGetT(SimStruct *S)</code>
Ada Syntax	<code>function ssGetT(S : in SimStruct) return Real_T;</code>
Arguments	S SimStruct representing an S-function block.
Description	Returns the current base simulation time (<code>time_T</code>) for the model. You can use this macro in <code>mdlOutputs</code> and <code>mdlUpdate</code> to compute the output of your block. <hr/> <p>Note Use this macro only if your block operates at the base rate of the model, for example, if your block operates at a single, continuous rate. If your block operates at multiple rates or operates at a single rate that is different from the model's base, use <code>ssGetTaskTime</code> to get the correct time for the current task.</p> <hr/>
Languages	Ada, C
See Also	<code>ssGetTaskTime</code> , <code>ssGetTStart</code> , <code>ssGetTFinal</code>

ssGetTNext

Purpose	Get the time of the next sample hit.
Syntax	<code>time_T ssGetTNext(SimStruct *S)</code>
Arguments	S SimStruct representing an S-function block
Description	Returns the next time that a sample hit occurs in a discrete S-function with a variable sample time.
Languages	C
See Also	<code>ssSetTNext</code> , <code>mdlGetTimeOfNextVarHit</code>

Purpose	Get the current time for the current task.
Syntax	<code>ssGetTaskTime(SimStruct *S, st_index)</code>
Arguments	<p><code>S</code> SimStruct representing an S-function block.</p> <p><code>st_index</code> Index of the sample time corresponding to the task for which the current time is to be returned.</p>
Description	Returns the current time (<code>time_T</code>) of the task corresponding to the sample rate specified by <code>st_index</code> . You can use this macro in <code>mdlOutputs</code> and <code>mdlUpdate</code> to compute the output of your block.
Languages	C
See Also	<code>ssGetT</code>

ssGetTFinal

Purpose	Get the simulation stop time.
C Syntax	<code>time_T ssGetTFinal (SimStruct *S)</code>
Ada Syntax	<code>function ssGetTFinal (S : in SimStruct) return Real_T;</code>
Arguments	S SimStruct representing an S-function block.
Description	Returns the stop time of the current simulation.
Languages	Ada, C
See Also	ssGetT, ssGetTStart

Purpose	Get the simulation start time.
C Syntax	<code>time_T ssGetTStart(SimStruct *S)</code>
Ada Syntax	<code>function ssGetTStart(S : in SimStruct) return Real_T;</code>
Arguments	S SimStruct representing an S-function block.
Description	Returns the start time of the current simulation.
Languages	Ada, C
See Also	ssGetT, ssGetTFinal

ssIsContinuousTask

Purpose	Determine if a task is continuous.
Syntax	<code>ssIsContinuousTask(SimStruct *S, st_index, tid)</code>
Arguments	<code>S</code> SimStruct representing an S-function block <code>tid</code> task ID
Description	Use in <code>mdlOutputs</code> or <code>mdlUpdate</code> when your S-function has multiple sample times to determine if your S-function is executing in the continuous task. This should not be used in single rate S-functions, or if you did not register a continuous sample time.
Languages	C
See Also	<code>ssSetNumContStates</code>

Purpose	Access user data.
Syntax	<code>void ssGetUserData(SimStruct *S, void * data)</code>
Arguments	<p>S SimStruct representing an S-function block.</p> <p>data User data</p>
Description	Retrieves pointer to user data.
Languages	C, C++
See Also	<code>ssSetUserData</code>

ssIsFirstInitCond

Purpose	Determine whether this is the first call to <code>mdlInitializeConditions</code> .
Syntax	<code>int_T ssGetFirstInitCond(SimStruct *S)</code>
Arguments	S SimStruct representing an S-function block.
Description	Returns true if the current simulation time is equal to the simulation start time.
Languages	C
See Also	<code>mdlInitializeConditions</code>

Purpose	Determine if the simulation is in a major step.
C Syntax	<code>int_T ssIsMajorTimeStep(SimStruct *S)</code>
Ada Syntax	<code>function ssIsMajorTimeStep(S : in SimStruct) return Boolean;</code>
Arguments	S SimStruct representing an S-function block
Description	Returns 1 if the simulation is in a major time step.
Languages	Ada, C
See Also	<code>ssIsMinorTimeStep</code>

ssIsMinorTimeStep

Purpose	Determine if the simulation is in a minor step.
Syntax	<code>int_T ssIsMinorTimeStep(SimStruct *S)</code>
Arguments	S SimStruct representing an S-function block
Description	Returns 1 if the simulation is in a minor time step.
Languages	C
See Also	<code>ssIsMajorTimeStep</code>

Purpose	Determine if sample is hit.
Syntax	<code>ssIsSampleHit(SimStruct *S, st_index, tid)</code>
Arguments	<p><code>S</code> SimStruct representing an S-function block</p> <p><code>st_index</code> Index of the sample time</p> <p><code>tid</code> task ID</p>
Description	Use in <code>mdlOutputs</code> or <code>mdlUpdate</code> when your S-function has multiple sample times to determine what task your S-function is executing in. This should not be used in single rate S-functions or for an <i>st_index</i> corresponding to a continuous task.
Languages	C
See Also	<code>ssIsContinuousTask</code> , <code>ssIsSpecialSampleHit</code>

ssIsSpecialSampleHit

Purpose	Determine if sample is hit.
Syntax	<code>ssIsSpecialSampleHit(SimStruct *S, sti1, sti2, tid)</code>
Arguments	<p>S SimStruct representing an S-function block</p> <p>sti1 Index of the sample time</p> <p>sti2 Index of the sample time</p> <p>tid task ID</p>
Description	Returns true if a sample hit has occurred at sti1 and a sample hit has also occurred at sti2 in the same time step. You can use this macro in mdl Update and mdl Outputs to ensure the validity of data shared by multiple tasks running at different rates. For more information, see "Synchronizing Multirate S-Function Blocks" on page 7-22.
Languages	C
See Also	<code>ssIsSampleHit</code>

Purpose	Get the name of the solver being used to solve the S-function.
Syntax	<code>ssGetSolverName(SimStruct *S)</code>
Arguments	S SimStruct representing an S-function block or a Simulink model.
Description	Returns 1 if the solver being used to solve S is a variable step solver. This is useful when creating S-functions that have zero crossings and an inherited sample time.
Languages	C
See Also	<code>ssGetSimMode</code> , <code>ssGetSolverName</code>

ssPrintf

Purpose	Print a variable-content message.
Syntax	<code>ssPrintf(msg, ...)</code>
Arguments	<code>msg</code> Message. Must be a string with optional variable replacement parameters. ... Optional replacement arguments.
Description	Prints variable-content <code>msg</code> . This macro expands to <code>mexPrintf</code> when the S-function is compiled via <code>mex</code> for use with Simulink. When the S-function is compiled for use with the Real-Time Workshop, this macro expands to <code>printf</code> , if the target has <code>stdio</code> facilities; otherwise, it becomes a call to an empty function (<code>rtPrintfNoOp</code>). In the case of Real-Time Workshop, you can avoid a call altogether, using the <code>SS_STDI0_AVAI LABLE</code> macro, e.g., <pre>#if defined(SS_STDI0_AVAI LABLE) ssPrintf("my message ..."); #endif</pre>
Languages	C
See Also	<code>ssWarning</code>

ssRegAllTunableParamsAsRunTimeParams

Purpose Register all tunable parameters as run-time parameters.

Syntax `void ssRegAllTunableParamsAsRunTimeParams(S,
const char_T *names[])`

Arguments `S`
SimStruct representing an S-function block.
`names`
Array of names for the run-time parameters

Description Use this function in `mdlSetWorkWidths` to register all tunable dialog parameters as run-time parameters. Specify the names of the run-time versions of the parameters in the `names` array.

Note Simulink assumes that the `names` array is always available. Therefore, you must allocate the `names` array in such a way that it persists throughout the simulation.

You can register dialog parameters individually as run-time parameters, using `ssSetNumRunTimeParameters` and `ssSetRunTimeParamInfo`.

Languages C

See Also `mdlSetWorkWidths`, `ssSetNumRunTimeParameters`, `ssSetRunTimeParamInfo`

ssRegisterDataType

Purpose Register a custom data type.

Syntax `DtypeId = ssRegisterDataType(SimStruct *S, char *name)`

Arguments

- `S`
SimStruct representing an S-function block.
- `name`
Name of custom data type

Description Register a custom data type. Each data type must be a valid MATLAB identifier. That is, the first char is an alpha and all subsequent characters are alphanumeric or "_". The name length must be less than 32. Data types must be registered in `mdlInitializeSes`.

If the registration is successful, the function returns the `DataTypeId` associated with the registered data type, otherwise, it reports an error and returns `INVALID_DTYPE_ID`.

After registering the data type, you must specify its size, using `ssSetDataTypeSize`.

Note You can call this function to get the data type id associated with a registered data type.

Example The following example registers a custom data type named `Color`.

```
DtypeId = ssRegisterDataType(S, "Color");  
if(Id == INVALID_DTYPE_ID) return;
```

Languages C

See Also `ssSetDataTypeSize`

Purpose	Specify that an output port is issuing a function call.
Syntax	<code>ssSetCallSystemOutput(SimStruct *S, port_index)</code>
Arguments	<p>S SimStruct representing an S-function block or a Simulink model.</p> <p>port_index Index of port that is issuing the function call</p>
Description	Use in mdlInitializeSampleTimes to specify that the output port element specified by <i>index</i> is issuing a function call by using <code>ssCallSystemWithTid(S, index, tid)</code> . The <i>index</i> specified starts at 0 and must be less than <code>ssGetOutputPortWidth(S, 0)</code> .
Languages	C
See Also	<code>ssCallSystemWithTid</code>

ssSetDataTypeSize

Purpose	Set the size of a custom data type.
Syntax	<code>int_T ssSetDataTypeSize(SimStruct *S, DTypeId id, int_T size)</code>
Arguments	<p><code>S</code> SimStruct representing an S-function block.</p> <p><code>id</code> ID of data type</p> <p><code>size</code> Size of the custom data type in bytes</p>
Description	Sets the size of the data type specified by <code>id</code> to <code>size</code> . If the call is successful, the macro returns 1 (true), otherwise, it returns 0 (false). Use this macro in <code>mdlInitializeSizes</code> to set the size of a data type you have registered.
Example	<p>The following example registers and sets the size of the custom data type named <code>Color</code> to four bytes.</p> <pre>int_T status; DTypeId id; id = ssRegisterDataType(SimStruct *S, "Color"); if(id == INVALID_DTYPE_ID) return; status = ssSetDataTypeSize(S, id, 4); if(status == 0) return;</pre>
Languages	C
See Also	<code>ssRegisterDataType</code> , <code>ssGetDataTypeSize</code>

Purpose	Set zero representation of a data type.
Syntax	<code>int_T ssSetDataTypeZero(SimStruct *S, DTypeId id, void* zero)</code>
Arguments	<p><code>S</code> SimStruct representing an S-function block.</p> <p><code>id</code> ID of data type</p> <p><code>zero</code> Zero representation of the data type specified by <code>id</code></p>
Description	<p>Sets the zero representation of the data type specified by <code>id</code> to <code>zero</code> and returns 1 (true), if <code>id</code> valid, and the size of the data type has been set, and the zero representation has not already been set. Otherwise, this macro returns 0 (false) and reports an error. Because this macro reports any error that occurs, you do not need to use <code>ssSetErrorStatus</code> to report the error.</p> <hr/> <p>Note This macro makes a copy of the zero representation of the data type for Simulink's use. Thus, your S-function does not have to maintain the original in memory.</p> <hr/>
Languages	C
Example	<p>The following example registers and sets the size and zero representation of a custom data type named <code>myDataType</code>.</p> <pre> typedef struct{ int8_T a; uint16_T b; }myStruct; int_T status; DtypeId id; myStruct tmp; id = ssRegisterDataType(S, "myDataType"); </pre>

ssSetDataTypeZero

```
if(id == INVALID_DTYPE_ID) return;

status = ssSetDataTypeSize(S, id, sizeof(tmp));
if(status == 0) return;

tmp.a = 0;
tmp.b = 1;
status = ssSetDataTypeZero(S, id, &tmp);
if(status == 0) return;
```

See Also [ssRegisterDataType](#), [ssSetDataTypeSize](#), [ssGetDataTypeZero](#)

Purpose	Specify whether the elements of a data type work vector are real or complex.
Syntax	<code>void ssSetDWorkComplexSignal (SimStruct *S, int_T vector, CSIGNAL_T numType)</code>
Arguments	<p><code>S</code> SimStruct representing an S-function block.</p> <p><code>vector</code> Index of a data type work vector, where the index is one of 0, 1, 2, ... <code>ssGetNumDWork(S)</code></p> <p><code>numType</code> Numeric type, either <code>COMPLEX_YES</code> or <code>COMPLEX_NO</code>.</p>
Description	Use in <code>mdlInitializeSizes</code> or <code>mdlSetWorkWidths</code> to specify whether the values of the specified work vector are complex numbers (<code>COMPLEX_YES</code>) or real numbers (<code>COMPLEX_NO</code> , the default).
Languages	C, C++
See Also	<code>ssSetDWorkDataType</code> , <code>ssGetNumDWork</code>

ssSetDWorkDataType

Purpose	Specify the data type of a data type work vector.
Syntax	<code>void ssSetDWorkDataType(SimStruct *S, int_T vector, DTypel dtID)</code>
Arguments	<p><code>S</code> SimStruct representing an S-function block.</p> <p><code>vector</code> Index of a data type work vector, where the index is one of 0, 1, 2, . . . <code>ssGetNumDWork(S)</code></p> <p><code>dtID</code> Id of a data type</p>
Description	Use in <code>mdlInitializeSizes</code> or <code>mdlSetWorkWidths</code> to set the data type of the specified work vector.
Languages	C, C++
See Also	<code>ssSetDWorkWidth</code> , <code>ssGetNumDWork</code>

Purpose	Specify the name of a data type work vector.
Syntax	<code>void ssSetDWorkName(SimStruct *S, int_T vector, char_T *name)</code>
Arguments	<p><code>S</code> SimStruct representing an S-function block.</p> <p><code>name</code> Index of the work vector, where the index is one of 0, 1, 2, ... <code>ssGetNumDWork(S)</code></p> <p><code>name</code> Name of work vector.</p>
Description	Use in <code>mdlInitializeSizes</code> or <code>mdlSetWorkWidths</code> to specify a name for the specified data type work vector. The Real-Time Workshop uses this name to label the work vector in generated code. If you do not specify a name, the Real-Time Workshop generates a name for the work vector.
Languages	C, C++
See Also	<code>ssGetDWorkName</code> , <code>ssSetNumDWork</code>

ssSetDWorkUsedAsDState

Purpose Specify that a data type work vector is used as a discrete state vector.

Syntax `void ssSetDWorkUsedAsDState(SimStruct *S, int_T vector, int_T usage)`

Arguments `S`
SimStruct representing an S-function block.

`vector`
Index of a data type work vector, where the index is one of 0, 1, 2, ...
`ssGetNumDWork(S)`

Usage
How this vector is used

Description Use in `mdlInitializeSizes` or `mdlSetWorkWidths` to specify the usage of the specified work vector, either `SS_DWORK_USED_AS_DSTATE` (used to store the block's discrete states) or `SS_DWORK_USED_AS_DWORK` (used as a work vector, the default).

Note Specify the usage as `SS_DWORK_USED_AS_DSTATE` if the following conditions are true. You want to use the vector to store discrete states and you want Simulink to log the discrete states to the workspace at the end of a simulation, if the user has selected the Save to Workspace option on Simulink's Simulation Parameters dialog.

Languages C, C++

See Also `sGetDWorkUsedAsDState`

Purpose	Specify the width of a data type work vector.
Syntax	<code>void ssSetDWorkWidth(SimStruct *S, int_T vector, int_T width)</code>
Arguments	<p><code>S</code> SimStruct representing an S-function block.</p> <p><code>vector</code> Index of the work vector, where the index is one of 0, 1, 2, ... <code>ssGetNumDWork(S)</code></p> <p><code>width</code> Number of elements in the work vector.</p>
Description	Use in <code>mdlInitializeSizes</code> or in <code>mdlSetWorkWidths</code> to set the number of elements in the specified data type work vector.
Languages	C, C++
See Also	<code>ssGetDWorkWidth</code> , <code>ssSetWorkDataType</code> , <code>ssSetNumDWork</code>

ssSetErrorStatus

Purpose	Report an error.
C Syntax	<code>void ssSetErrorStatus(SimStruct *S, const char_T *msg)</code>
Ada Syntax	<code>procedure ssSetErrorStatus(S : in SimStruct; msg : in String);</code>
Arguments	<p>S SimStruct representing an S-function block or a Simulink model.</p> <p>msg Error message</p>
Description	<p>Use this function to report errors that occur in your S-function, e.g.,</p> <pre>ssSetErrorStatus(S, "error message"); return;</pre> <hr/> <p>Note The error message string must be in persistent memory; it cannot be a local variable.</p> <hr/>
Languages	Ada, C
See Also	<code>ssWarning</code>

Purpose	Specify the external mode function for an S-function.
Syntax	<code>void ssSetExternalModeFcn(SimStruct *S, SFunExtModeFcn *Fcn)</code>
Arguments	<code>S</code> SimStruct representing an S-function block or a Simulink model. <code>fcn</code> external mode function
Description	Specifies the external mode function for S.
Languages	C
See Also	<code>ssCallExternalModeFcn</code>

ssSetInputPortComplexSignal

Purpose	Set the numeric type (real or complex) of an input port.
Syntax	<pre>void ssSetInputPortComplexSignal (SimStruct *S, input_T port, CSIGNAL_T csignal)</pre>
Arguments	<p>S SimStruct representing an S-function block or a Simulink model.</p> <p>port Index of an input port</p> <p>csignal Numeric type of the signals accepted by port. Valid values are COMPLEX_NO (real signal), COMPLEX_YES (complex signal), COMPLEX_INHERITED (numeric type inherited from driving block).</p>
Description	Use this function in mdlInitializeSizes to initialize input port signal type. If the numeric type of the input port is inherited from the block to which it is connected, set the numeric type to COMPLEX_INHERITED. The default numeric type of an input port is real.
Languages	C
Example	<p>Assume that an S-function has three input ports. The first input port accepts real (non-complex) signals. The second input port accepts complex signal. The third port accepts signals of either type. The following example specifies the correct numeric type for each port.</p> <pre>ssSetInputPortComplexSignal (S, 0, COMPLEX_NO) ssSetInputPortComplexSignal (S, 1, COMPLEX_YES) ssSetInputPortComplexSignal (S, 2, COMPLEX_INHERITED)</pre>
See Also	ssGetInputPortComplexSignal

Purpose	Set the data type of an input port.
C Syntax	<code>void ssSetInputPortDataType(SimStruct *S, input_T port, DType id)</code>
Ada Syntax	<pre>procedure ssSetInputPortDataType(S : in SimStruct; port : in Integer := 0; id : in Integer);</pre>
Arguments	<p><code>S</code> SimStruct representing an S-function block or a Simulink model.</p> <p><code>port</code> Index of an input port</p> <p><code>id</code> Id of data type accepted by port</p>
Description	<p>Use this function in mdlInitializeSizes to set the data type of the input port specified by <code>port</code>. If the input port's data type is inherited from the block connected to the port, set the data type to DYNAMICALLY_TYPED.</p> <hr/> <p>Note The data type of an input port is double (real_T) by default.</p> <hr/>
Languages	Ada, C
Example	<p>Suppose that you want to create an S-function with two input ports, the first of which inherits its data type the driving block and the second of which accepts inputs of type <code>int8_T</code>. The following code sets up the data types.</p> <pre>ssSetInputPortDataType(S, 0, DYNAMICALLY_TYPED) ssSetInputPortDataType(S, 1, SS_INT8)</pre>
See Also	<code>ssGetInputPortDataType</code>

ssSetInputPortDimensionInfo

Purpose Specify information about the dimensionality of an input port.

Syntax `void ssSetInputPortDimensionInfo(SimStruct *S, int_T port, DimInfo_T *dimInfo)`

Arguments `S`
SimStruct representing an S-function block.

`port`
Index of an input port

`dimInfo`
Structure of type `DimInfo_T` that specifies the dimensionality of the signals accepted by port.

The structure is defined as

```
typedef struct DimInfo_tag{
    int width; /* number of elements */
    int numDims /* Number of dimensions */
    int *dims; /* Dimensions. */
    [snip]
}DimInfo_T;
```

where:

- `numDims` specifies the number of dimensions of the signal, e.g., 1 for a 1-D (vector) signal or 2 for a 2-D (matrix) signal, or `DYNAMICALLY_SIZED` if the number of dimensions is determined dynamically
- `dims` is an integer array that specifies the size of each dimension, e.g., `[2 3]` for a 2-by-3 matrix signal, or `DYNAMICALLY_SIZED` for each dimension that is determined dynamically, e.g., `[2 DYNAMICALLY_SIZED]`
- `width` equals the total number of elements in the signal, e.g., 12 for a 3-by-4 matrix signal or 8 for an 8-element vector signal, or `DYNAMICALLY_SIZED` if the total number of elements is determined dynamically

Note Use the macro, `DECL_AND_INIT_DIMINFO`, to declare and initialize an instance of this structure.

Description	Specifies the dimension information for port. Use this function in <code>mdlInitializeSizes</code> to initialize the input port dimension information. If you want the port to inherit its dimensions from the port to which it is connected, specify <code>DYNAMIC_DIMENSION</code> as the <code>dimInfo</code> for port.
Languages	C
Example	The following example specifies that input port 0 accepts 2-by-2 matrix signals. <pre>DECL_AND_INIT_DIMINFO(di); di.numDims = 2; int dims[2]; dims[0] = 2; dims[1] = 2; di.dims = &dims; di.width = 4; ssSetInputPortDimensionInfo(S, 0, &di);</pre>
See Also	<code>ssSetInputPortMatrixDimensions</code> , <code>ssSetInputPortVectorDimensions</code>

ssSetInputPortFrameData

Purpose	Specify whether a port accepts signal frames.
Syntax	<code>void ssSetInputPortFrameData(SimStruct *S, int_T port, int_T acceptsFrames)</code>
Arguments	<p><code>S</code> SimStruct representing an S-function block.</p> <p><code>port</code> Index of an input port</p> <p><code>acceptsFrames</code> Type of signal accepted by port. Acceptable values are -1 (either frame or unframed input), 0 (unframed input only), 1 (framed input only).</p>
Description	Use in <code>mdlSetInputPortFrameData</code> to specify whether a port accepts frame data only, unframed data only, or both.
Languages	C
See Also	<code>ssGetInputPortFrameData</code> , <code>mdlSetInputPortFrameData</code>

Purpose	Specify the direct feedthrough status of a block's ports.
C Syntax	<pre>void ssSetInputPortDirectFeedThrough(SimStruct *S, int_T port, int_T dirFeed)</pre>
Ada Syntax	<pre>procedure ssSetInputPortDirectFeedThrough(S : in SimStruct; port : in Integer := 0; dirFeed : in Boolean);</pre>
Arguments	<p>S SimStruct representing an S-function block or a Simulink model.</p> <p>port Index of the input port whose direct feedthrough property is being set.</p> <p>dirFeed Direct feedthrough status of block specified by inputPortIdx.</p>
Description	Use in mdlInitializeSizes (after ssSetNumInputPorts) to specify the direct feedthrough (0 or 1) for each input port index. If not specified, the default direct feedthrough is 0. Setting direct feedthrough to 0 for an input port is equivalent to saying that the corresponding input port signal is not used in mdlOutputs or mdlGetTimeOfNextVarHit. If it is used, you may or may not see a delay of one simulation step in the input signal. This may cause the simulation solver to issue an error due to simulation inconsistencies.
Languages	Ada, C
See Also	ssSetInputPorts

ssSetInputPortMatrixDimensions

Purpose	Specify dimension information for an input port that accepts matrix signals.
Syntax	<pre>void ssSetInputPortMatrixDimensions(SimStruct *S, int_T port, int_T m, int_T n)</pre>
Arguments	<p>S SimStruct representing an S-function block.</p> <p>port Index of an input port</p> <p>m Row dimension of matrix signals accepted by port or DYNAMICALLY_SIZED</p> <p>n Column dimension of matrix signals accepted by port or DYNAMICALLY_SIZED</p>
Description	Specifies that port accepts an m-by-n matrix signal. If either dimension is DYNAMICALLY_SIZED, the other must be DYNAMICALLY_SIZED or 1.
Languages	C
Example	The following example specifies that input port 0 accepts 2-by-2 matrix signals. <pre>ssSetInputPortMatrixDimensions(S, 0, 2, 2);</pre>
See Also	ssSetInputPortDimensionInfo

Purpose	Specify the offset time of an input port.
Syntax	<pre>void ssSetInputPortOffsetTime(SimStruct *S, int_T inputPortIdx, int_T period)</pre>
Arguments	<p>S SimStruct representing an S-function block or a Simulink model.</p> <p>inputPortIdx Index of the input port whose offset time is being set.</p> <p>offset Offset time</p>
Description	Use in <code>mdlInitializeSizes</code> (after <code>ssSetNumInputPorts</code>) to specify the sample time offset for each input port index. You can use this macro in conjunction with <code>ssSetInputPortSampleTime</code> if you have specified port-based sample times for your S-function.
Languages	C
See Also	<code>ssSetNumInputPorts</code> , <code>ssSetInputPortSampleTime</code>

ssSetInputPortOverWritable

Purpose	Specify whether an input port can be overwritten.
C Syntax	<pre>void ssSetInputPortOverWritable(SimStruct *S, int_T port, int_T isOverWritable)</pre>
Ada Syntax	<pre>procedure ssSetInputPortOverWritable(S : in SimStruct; port : in Integer := 0; isOverWritable : in Boolean);</pre>
Arguments	<p>S SimStruct representing an S-function block or a Simulink model.</p> <p>port Index of the input port whose overwritability is being set.</p> <p>isOverWritable Value specifying whether port is overwritable.</p>
Description	<p>Use in <code>mdlInitializeSizes</code> (after <code>ssSetNumInputPorts</code>) to specify whether the input port is overwritable by an output port. The default is <code>isOverWritable=0</code>, which means that the input port does not share memory with an output port. When <code>isOverWritable=1</code>, the input port shares memory with an output port.</p> <hr/> <p>Note <code>ssSetInputPortReusable</code> and <code>ssSetOutputPortReusable</code> must both be set to 0, meaning that neither port involved can have global and persistent memory.</p> <hr/>
Languages	Ada, C
See Also	<code>ssSetNumInputPorts</code> , <code>ssSetInputPortReusable</code> , <code>ssSetOutputPortReusable</code> , <code>ssGetInputPortBufferDstPort</code>

Purpose	Specify whether where memory allocated to port is reusable.
Syntax	<code>void ssSetInputPortReusable(SimStruct *S, int_T port, int_T isReusable)</code>
Arguments	<p><code>S</code> SimStruct representing an S-function block or a Simulink model.</p> <p><code>inputPortIdx</code> Index of the input port whose reusability is being set.</p> <p><code>isReusable</code> Value specifying whether port is reusable.</p>
Description	<p>Use in <code>mdlInitializeSizes</code> (after <code>ssSetNumInputPorts</code>) to specify whether the input port memory buffer can be reused by other signals in the model. This macro can take on two values:</p> <ul style="list-style-type: none"> • Off (<code>isReusable=0</code>) — specifies that the input port is not reusable. This is the default. • On (<code>isReusable=1</code>) — specifies that the input port is reusable. <p>In Simulink, reusable signals share the same memory space. When this macro is turned on, the input port signal to the S-function may be reused by other signals in the model. This reuse results in less memory use during Simulink simulation and more efficiency in the Real-Time Workshop generated code.</p> <p>You must use caution when using this macro; you can safely turn it on only if the S-function reads its input port signal in its <code>mdlOutputs</code> routine and does not access this input port signal until the next call to <code>mdlOutputs</code>.</p> <p>When an S-function's input port signal is reused, other signals in the model overwrite it prior to the execution of <code>mdlUpdate</code>, <code>mdlDerivatives</code>, or other run-time S-function routines. For example, if the S-function reads the input port signal in its <code>mdlUpdate</code> routine, or reads the input port signal in the <code>mdlOutputs</code> routine and expects this value to be persistent until the execution of its <code>mdlUpdate</code> routine, turning this attribute on is incorrect and will lead to erroneous results.</p> <p>The default setting, off, is safe. It prevents any reuse of the S-function input port signals, which means that the input port signals have the same value in</p>

ssSetInputPortReusable

any run-time S-function routine during a single execution of the simulation loop.

Note that this is a suggestion and not a requirement for the Simulink engine. If Simulink cannot resolve buffer reuse in local memory, it resets `val ue=0` and places the input port signals into global memory

Languages C

See Also `ssSetNumInputPorts`, `ssSetInputPortOverwrite`, `ssSetOutputPortReusable`

ssSetInputPortRequiredContiguous

Purpose Specify that the signal elements entering a port must be contiguous.

Syntax `void ssSetInputPortRequiredContiguous(SimStruct *S, int_T port)`

Arguments

- `S`
SimStruct representing an S-function block or a Simulink model.
- `port`
Index of an input port

Description Specifies that the signal elements entering the specified port must occupy contiguous areas of memory. This allows a method to access the elements of the signal simply by incrementing the signal pointer returned by `ssGetInputPortSignal`. The S-function can set the value of this attribute as early as in the `mdlInitializeSizes` method and at the latest in the `mdlSetWorkWidths` method.

Note The default setting for this flag is false. Hence, the default method for accessing the input signals is `ssGetInputSignalPtrs`.

Languages C, C++

See Also `mdlInitializeSizes`, `mdlSetWorkWidths`, `ssGetInputPortSignal`, `ssGetInputPortSignalPtrs`

ssSetInputPortSampleTime

Purpose	Specify the sample time of an input port.
Syntax	<code>ssSetInputPortSampleTime(SimStruct *S, inputPortIdx, period)</code>
Arguments	<p><code>S</code> SimStruct representing an S-function block or a Simulink model.</p> <p><code>inputPortIdx</code> Index of the input port whose sample time is being set.</p> <p><code>period</code> Sample period.</p>
Description	Use in <code>mdlInitializeSizes</code> (after <code>ssSetNumInputPorts</code>) to specify the sample time period as continuous or as a discrete value for each input port. Input port index numbers start at 0 and end at the total number of input ports minus 1. You should use this macro only if you have specified port-based sample times.
Languages	C
See Also	<code>ssSetNumInputPorts</code> , <code>ssSetInputPortOffsetTime</code>

Purpose	Specify the sample time index of an input port.
Syntax	<pre>void ssSetInputPortSampleTimeIndex(SimStruct *S, int_T inputPortIdx, int_T sampleTimeIdx)</pre>
Arguments	<p>S SimStruct representing an S-function block or a Simulink model.</p> <p>inputPortIdx Index of the input port whose sample time index is being set.</p> <p>sampleTimeIdx Sample time index.</p>
Description	<p>Use in mdlInitializeSizes (after ssSetNumInputPorts) to specify the index of the sample time for the port to be used in mdlOutputs and mdlOutputs when checking for sample hits.</p> <hr/> <p>Note This should only be used when the PORT_BASED_SAMPLE_TIMES has been specified for ssSetNumSamples in mdlInitializeSizes.</p> <hr/>
Languages	C, C++
See Also	ssGetInputPortSampleTimeIndex, mdlInitializeSizes, ssSetNumInputPorts, mdlOutputs, mdlOutputs

ssSetInputPortVectorDimension

Purpose	Specify dimension information for an input port that accepts vector signals.
Syntax	<code>void ssSetInputPortVectorDimension(SimStruct *S, int_T port, int_T w)</code>
Arguments	<p><code>S</code> SimStruct representing an S-function block.</p> <p><code>port</code> Index of an input port</p> <p><code>w</code> Width of vector or DYNAMICALLY_SIZED</p>
Description	Specifies that <code>port</code> accepts a <code>w</code> -element vector signal.
	<hr/> <p>Note This macro and <code>ssSetInputPortWidth</code> are functionally identical.</p> <hr/>
Languages	C
Example	The following example specifies that input port 0 accepts an 8-element matrix signal. <pre>ssSetInputPortVectorDimension(S, 0, 8);</pre>
See Also	<code>ssSetInputPortDimensionInfo</code> , <code>ssSetInputPortWidth</code>

Purpose	Specify the number of input ports that a block has.
C Syntax	<code>void ssSetInputPortWidth(SimStruct *S, int_T port, int_T width)</code>
Ada Syntax	<pre>procedure ssSetInputPortWidth (S : in SimStruct; port : in Integer := 0; width : in Integer);</pre>
Arguments	<p>S SimStruct representing an S-function block or a Simulink model.</p> <p>port Index of the input port whose width is being set.</p> <p>width Width of input port.</p>
Description	Use in <code>mdlInitializeSizes</code> (after <code>ssSetNumInputPorts</code>) to specify a nonzero positive integer width or <code>DYNAMICALLY_SIZED</code> for each input port index starting at 0.
Languages	Ada, C
See Also	<code>ssSetNumInputPorts</code> , <code>ssSetOutputPortWidth</code>

ssSetModeVectorValue

Purpose	Set an element of a block's mode vector.
Syntax	<code>void ssSetModeVectorValue(SimStruct *S, int_T element, int_T value)</code>
Arguments	<p>S SimStruct representing an S-function block.</p> <p>element Index of a mode vector element</p> <p>value Mode vector value</p>
Description	Sets the specified mode vector element to the specified value.
Languages	C, C++
See Also	<code>ssGetModeVectorValue</code> , <code>ssGetModeVector</code>

Purpose	Specify the number of continuous states that a block has.
C Syntax	<code>void ssSetNumContStates(SimStruct *S, int_T n)</code>
Ada Syntax	<code>procedure ssSetNumContStates(S : in SimStruct; n : in Integer);</code>
Arguments	<p><code>S</code> SimStruct representing an S-function block.</p> <p><code>n</code> Number of continuous states to be set for the block represented by <code>S</code>.</p>
Description	Use in mdl InitializeSizes to specify the number of continuous states as 0, a positive integer, or DYNAMICALLY_SIZED. If you specify DYNAMICALLY_SIZED, you can specify the true (positive integer) width in mdl SetWorkWidths, otherwise the width to is the width of the signal passing through the block. If your S-function has continuous states, it needs to return the derivatives of the states in mdl Derivatives so that the solvers can integrate them. Continuous states are logged if the States option is checked on the Workspace I/O pane of the Simulation Parameters dialog box.
Languages	Ada, C
See Also	ssSetNumDiscStates, ssGetNumContStates

ssSetNumDiscStates

Purpose	Specify the number of discrete states that a block has.
Syntax	<code>ssSetNumDiscStates(SimStruct *S, int_T nDiscStates)</code>
Arguments	<p><code>S</code> SimStruct representing an S-function block.</p> <p><code>nDiscStates</code> Number of discrete states to be set for the block represented by <code>S</code>.</p>
Description	Use in <code>mdlInitializeSizes</code> to specify the number of discrete states as 0, a positive integer, or <code>DYNAMICALLY_SIZED</code> . If you specify <code>DYNAMICALLY_SIZED</code> , you can specify the true (positive integer) width in <code>mdlSetWorkWidths</code> , otherwise the width used is the width of the signal passing through the block. If your S-function has discrete states, it should return the next discrete state (in place) in <code>mdlUpdate</code> . Discrete states are logged if the <code>States</code> is checked on the <code>Workspace I/O</code> page of the Simulation Parameters dialog box.
Languages	C
See Also	<code>ssSetNumContStates</code> , <code>ssGetNumDiscStates</code>

Purpose	Specify the number of data type work vectors used by a block.
Syntax	<code>void ssSetNumDWork(SimStruct *S, int_T nDWork)</code>
Arguments	<p><code>S</code> SimStruct representing an S-function block.</p> <p><code>DWork</code> Number of data type work vectors.</p>
Description	<p>Use in mdlInitializeSizes to specify the number of data type work vectors as 0, a positive integer, or DYNAMICALLY_SIZED. If you specify DYNAMICALLY_SIZED, you can specify the true (positive integer) number of vectors in mdlSetWorkWidths.</p> <p>You can specify the size and data type of each work vector, using the macros <code>ssSetDWorkWidth</code> and <code>ssSetDWorkDataType</code>, respectively. You can also specify that the work vector holds complex values, using <code>ssSetDWorkComplexSignal</code>.</p>
Languages	C, C++
See Also	<code>ssGetNumDWork</code> , <code>ssSetDWorkWidth</code> , <code>ssSetDWorkDataType</code> , <code>ssSetDWorkComplexSignal</code>

ssSetNumInputPorts

Purpose	Specify the number of input ports that a block has.
C Syntax	<pre>void ssSetNumInputPorts(SimStruct *S, int_T nInputPorts)</pre>
Ada Syntax	<pre>procedure ssSetNumInputPorts(S : in SimStruct; nInputPorts : in Integer);</pre>
Arguments	<p>S SimStruct representing an S-function block.</p> <p>nInputPorts Number of input ports on the block represented by S. Must be a nonnegative integer.</p>
Description	<p>Used in mdlInitializeSizes to set to the number of input ports to a nonnegative integer. It should be invoked using</p> <pre>if (!ssSetNumInputPorts(S, <i>nInputPorts</i>)) return;</pre> <p>where ssSetNumInputPorts returns 0 if <i>nInputPorts</i> is negative or an error occurred while creating the ports. When this occurs, Simulink displays an error.</p>
Languages	Ada, C
See Also	ssSetInputPortWidth, ssSetNumOutputPorts

Purpose	Specify the size of a block's integer work vector.
Syntax	<code>void ssSetNumIWork(SimStruct *S, int_T nIWork)</code>
Arguments	<p><code>S</code> SimStruct representing an S-function block.</p> <p><code>nIWork</code> Number of elements in the integer work vector.</p>
Description	Use in mdlInitializeSizes to specify the number of int_T work vector elements as 0, a positive integer, or DYNAMICALLY_SIZED. If you specify DYNAMICALLY_SIZED, you can specify the true (positive integer) width in mdlSetWorkWidths, otherwise the width used is the width of the signal passing through the block.
Languages	C
See Also	ssSetNumRWork, ssSetNumPWork

ssSetNumModes

Purpose	Specifies the size of the block's mode vector.
Syntax	<code>ssSetNumModes(SimStruct *S, nModes)</code>
Arguments	<p><i>S</i> SimStruct representing an S-function block.</p> <p><i>nModes</i> Size of the mode vector for the block represented by <i>S</i>. Valid values are 0, a positive integer, or DYNAMICALLY_SIZED.</p>
Description	<p>Sets the size of the block's mode vector to <i>nModes</i>. If <i>nModes</i> is DYNAMICALLY_SIZED, you can specify the true (positive integer) width in <code>mdlSetWorkWidths</code>, otherwise the width used is the width of the signal passing through the block. Use this macro in <code>mdlInitializeSizes</code> to specify the number of <code>int_T</code> elements in the mode vector. Simulink allocates the mode vector and initializes its elements to 0. If the default value of 0 is not appropriate, you can set the elements of the array to other initial values in <code>mdlInitializeConditions</code>. Use <code>ssGetModeVector</code> to access the mode vector.</p> <p>The mode vector, combined with zero-crossing detection, allows you to create blocks that have distinct operating modes, depending on the current values of input or output signals. For example, consider a block that outputs the absolute value of its input. Such a block operates in two distinct modes, depending on whether its input is positive or negative. If the input is positive, the block outputs the input unchanged. If the input is negative, the block outputs the negative of the input. You can use zero-crossing detection to detect when the input changes sign and update the single-element mode vector accordingly (for example, by setting its element to 0 for negative input and 1 for positive input). You can then use the mode vector in <code>mdlOutputs</code> to determine the mode in which the block is currently operating.</p>
Languages	C
See Also	<code>ssGetNumModes</code> , <code>ssGetModeVector</code>

Purpose	Specify the number of states for which a block detects zero crossings that occur between sample points.
Syntax	<code>ssSetNumNonsampledZCs(SimStruct *S, nNonsampledZCs)</code>
Arguments	<p><code>S</code> SimStruct representing an S-function block.</p> <p><code>nNonsampledZCs</code> Number of nonsampled zero crossings that a block detects.</p>
Description	Use in mdlInitializeSizes to specify the number of states for which the block detects nonsampled zero crossings (real_T) as 0, a positive integer, or DYNAMICALLY_SIZED. If you specify DYNAMICALLY_SIZED, you can specify the true (positive integer) width in mdlSetWorkWidths, otherwise the width to be used will be the width of the signal passing through the block.
Languages	C
See Also	<code>ssSetNumModes</code>

ssSetNumOutputPorts

Purpose	Specify the number of output ports that a block has.
C Syntax	<code>void ssSetNumOutputPorts(SimStruct *S, int_T nOutputPorts)</code>
Ada Syntax	<code>procedure ssSetNumOutputPorts(S : in SimStruct; nOutputPorts : in Integer);</code>
Arguments	<p>S SimStruct representing an S-function block.</p> <p>nOutputPorts Number of output ports on the block represented by S. Must be a nonnegative integer.</p>
Description	<p>Use in mdlInitializeSizes to set to the number of output ports to a nonnegative integer. It should be invoked using</p> <pre>if (!ssSetNumOutputPorts(S, <i>nOutputPorts</i>)) return;</pre> <p>where <code>ssSetNumOutputPorts</code> returns a 0 if <i>nOutputPorts</i> is negative or an error occurred while creating the ports. When this occurs, and you return out of your S-function, Simulink will display an error message.</p>
Languages	Ada, C
See Also	<code>ssSetInputPortWidth</code> , <code>ssSetNumInputPorts</code>

Purpose	Specify the size of a block's pointer work vector.
Syntax	<code>void ssSetNumPWork(SimStruct *S, int_T nPWork)</code>
Arguments	<p>S SimStruct representing an S-function block.</p> <p>nPWork Number of elements to be allocated to the pointer work vector of the block represented by S.</p>
Description	Use in <code>mdlInitializeSizes</code> to specify the number of pointer (<code>void *</code>) work vector elements as 0, a positive integer, or <code>DYNAMICALLY_SIZED</code> . If you specify <code>DYNAMICALLY_SIZED</code> , you can specify the true (positive integer) width in <code>mdlSetWorkWidths</code> , otherwise the width used is the width of the signal passing through the block.
Languages	C
See Also	<code>ssSetNumIWork</code> , <code>ssSetNumPWork</code>

ssSetNumRunTimeParams

Purpose	Specify the number of run-time parameters created by this S-function.
Syntax	<code>void ssSetNumRunTimeParams(S, int_T num)</code>
Arguments	S SimStruct representing an S-function block. num Number of run-time parameters
Description	Use this function in mdl SetWorkWidths to specify the number of run-time parameters created by this S-function.
Languages	C
See Also	mdl SetWorkWidths, ssGetNumRunTimeParams, ssSetRunTimeParamInfo

Purpose	Specify the size of a block's floating-point work vector.
Syntax	<code>void ssSetNumRWork(SimStruct *S, int_T nRWork)</code>
Arguments	<p><code>S</code> SimStruct representing an S-function block.</p> <p><code>nRWork</code> Number of elements in the floating-point work vector.</p>
Description	Use in <code>mdlInitializeSizes</code> to specify the number of <code>real_T</code> work vector elements as 0, a positive integer, or <code>DYNAMICALLY_SIZED</code> . If you specify <code>DYNAMICALLY_SIZED</code> , you can specify the true (positive integer) width in <code>mdlSetWorkWidths</code> , otherwise the width used is the width of the signal passing through the block.
Languages	C
See Also	<code>ssSetNumIWork</code> , <code>ssSetNumPWork</code>

ssSetNumSampleTimes

Purpose	Specify the number of sample times that an S-function block has.
Syntax	<code>void ssSetNumSampleTimes(SimStruct *S, int_T nSampleTimes)</code>
Arguments	<code>S</code> SimStruct representing an S-function block. <code>nSampleTimes</code> Number of sample times that S has.
Description	Use in <code>mdlInitializeSizes</code> to set the number of sample times S has. This must be a positive integer greater than 0.
Languages	C
See Also	<code>ssGetNumSampleTimes</code>

Purpose	Specify the number of parameters that an S-function block has.
Syntax	<code>ssSetNumSFcnParams(SimStruct *S, int_T nSFcnParams)</code>
Arguments	<p><code>S</code> SimStruct representing an S-function block.</p> <p><code>nSFcnParams</code> Number of parameters that S has.</p>
Description	Use in mdlInitializeSizes to set the number of S-function parameters.
Languages	C
See Also	<code>ssGetSFcnNumParams</code>

ssSetOffsetTime

Purpose	Set the offset time of a block.
Syntax	<code>ssSetOffsetTime(SimStruct *S, st_index, period)</code>
Arguments	<p><code>S</code> SimStruct representing an S-function block.</p> <p><code>st_index</code> Index of sample time whose offset is to be set.</p> <p><code>offset</code> Offset of the sample time specified by <code>st_index</code></p>
Description	Use this macro in <code>mdlInitializeSizes</code> to specify the offset of the sample time where <code>st_index</code> starts at 0.
Languages	C
See Also	<code>ssSetSampleTime</code> , <code>ssSetInputPortOffsetTime</code> , <code>ssSetOutputPortOffsetTime</code>

Purpose	Specify S-function options.
Syntax	<code>void ssSetOptions(SimStruct *S, uint_T options)</code>
Arguments	<p>S SimStruct representing an S-function block.</p> <p>options Options</p>
Description	Use in mdlInitializeSizes to specify S-function options (see below). The options must be joined using the OR operator. For example:

```
ssSetOptions(S, (SS_OPTION_EXCEPT_ON_FREE_CODE |
                SS_OPTION_DISCRETE_VALUED_OUTPUT));
```

S-Function Options

An S-function can specify the following options, using `ssSetOptions`:

- `SS_OPTION_EXCEPT_ON_FREE_CODE`
If your S-function does not use `mexErrMsgTxt`, `mxCallLoc`, or any other routines that can throw an exception when called, you can set this option for improved performance.
- `SS_OPTION_RUNTIME_EXCEPT_ON_FREE_CODE`
Similar to `SS_OPTION_EXCEPT_ON_FREE_CODE` except it only applies to the “run-time” routines: `mdlGetTimeOfNextVarHit`, `mdlOutputs`, `mdlUpdate`, and `mdlDerivatives`.
- `SS_OPTION_DISCRETE_VALUED_OUTPUT`
Specify this if your S-function has discrete valued outputs. This is checked when your S-function is placed within an algebraic loop. If your S-function has discrete valued outputs, then its outputs will not be assigned algebraic variables.
- `SS_OPTION_PLACE_ASAP`
Used to specify that your S-function should be placed as soon as possible. This is typically used by devices connecting to hardware.

ssSetOptions

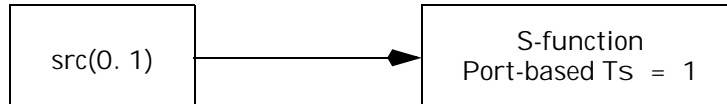
- `SS_OPTION_ALLOW_INPUT_SCALAR_EXPANSION`
Used to specify that the input to your S-function input ports can be either 1 or the size specified by the port, which is usually referred to as the block width.
- `SS_OPTION_DISALLOW_CONSTANT_SAMPLE_TIME`
Use to disable an S-function block from inheriting a constant sample time.
- `SS_OPTION_ASYNCHRONOUS`
This option applies only to S-functions that have 0 or 1 input ports and 1 output port. The output port must be configured to perform function calls on every element. If any of these requirements are not met, the `SS_OPTION_ASYNCHRONOUS` is ignored. Use this option when driving function-call subsystems that will be attached to interrupt service routines.
- `SS_OPTION_ASYNC_RATE_TRANSITION`
Use this when your S-function converts a signal from one rate to another rate.
- `SS_OPTION_RATE_TRANSITION`
Use this option when your S-function is behaving as a unit delay or a ZOH. This macro support these two operations only. It identifies a unit delay by the presence of `mdlUpdate`; if `mdlUpdate` is absent, the operation is taken to be ZOH.
- `SS_OPTION_PORT_SAMPLE_TIMES_ASSIGNED`
Use this when you have registered multiple sample times (`ssSetNumSampleTimes > 1`) to specify the rate at when each input and output port is running at. The simulation engine needs this information when checking for illegal rate transitions.
- `SS_OPTION_SFUNCTION_ILLEGAL_FOR_RTW`
Set this if you have a .tlc file for your S-function and do not have a `mdlRTW` method. Setting option has no effect if you have a `mdlRTW` method.
- `SS_OPTION_ALLOW_PARTIAL_DIMENSIONS_CALL`
Indicates that the S-function can handle dynamically dimensioned signals. See `mdlSetInputPortDimensions`, `mdlSetOutputPortDimensions`, or `mdlSetDefaultPortDimensions` for more information.

- `SS_OPTION_FORCE_NONINLINED_FCNCALL`
 Use this flag if the block requires that all function-call subsystems that it calls should be generated as procedures instead of possibly being generated as inlined code.
- `SS_OPTION_USE_TLC_WITH_ACCELERATOR`
 Use this to force the Accelerator to use the TLC inlining code for a S-function which will speed up execution of the S-function. By default, the Accelerator will use the mex version of the S-function even though a TLC file for the S-function exists. This option should not be set for device driver blocks (A/D) or when there is an incompatibility between running the mex Start/Initialize/ConditionalizeConditions functions together with the TLC Outputs/Update/Derivatives.
- `SS_OPTION_SIM_VIewing_DEVICE`
 This S-function is a SimViewingDevice. As long as it meets the other requirement for this type of block (no states, no outputs, etc), it will be considered to be an external mode block (show up in the external mode GUI and no code is generated for it). During an external mode simulation, this block is run on the host only.
- `SS_OPTION_CALL_TERMINATE_ON_EXIT`
 This option allows S-function authors to better manage the data cached in run-time parameters and UserData. Setting this option guarantees that the mdl Terminate function is called if mdl InitializeSes is called. This means that mdl Terminate is called:
 - When a simulation ends.
 Note that it does not matter if the simulation failed and at what stage the simulation failed. Therefore, if the mdl SetWorkWdths of some block errors out, the model's other blocks have a chance to free the memory during a call to mdl Terminate.
 - Every time an S-function block is destroyed.
 - If the user is editing the S-function graphically.
 If this option is not set, mdl Terminate is called only if at least one of the blocks has had its mdl Start called.

ssSetOptions

- SS_OPTION_REQ_INPUT_SAMPLE_TIME_MATCH

Use this option to specify that the input signal sample time(s) match the sample time assigned to the block input port. For example,



generates an error if this option is set. If the block (or input port) sample time is inherited, then there will be no error generated.

Languages

C, C++

Purpose	Set the numeric type (real or complex) of an output port.
Syntax	<code>void ssSetOutputPortComplexSignal (SimStruct *S, input_T port, CSIGNAL_T csignal)</code>
Arguments	<p><code>S</code> SimStruct representing an S-function block or a Simulink model.</p> <p><code>port</code> Index of an output port</p> <p><code>csignal</code> Numeric type of the signals emitted by port. Valid values are COMPLEX_NO (real signal), COMPLEX_YES (complex signal), COMPLEX_INHERITED (dynamically determined).</p>
Description	Use this function in mdlInitializeSizes to initialize input port signal type. If the numeric type of the input port is determined dynamically, e.g., by a parameter setting, set the numeric type to COMPLEX_INHERITED. The default numeric type of an output port is real.
Languages	C
Example	<p>Assume that an S-function has three output ports. The first output port emits real (non-complex) signals. The second input port emits a complex signal. The third port emits signals of a type determined by a parameter setting. The following example specifies the correct numeric type for each port.</p> <pre>ssSetOutputPortComplexSignal (S, 0, COMPLEX_NO) ssSetOutputPortComplexSignal (S, 1, COMPLEX_YES) ssSetOutputPortComplexSignal (S, 2, COMPLEX_INHERITED)</pre>
See Also	<code>ssGetOutputPortComplexSignal</code>

ssSetOutputPortDataType

Purpose	Set the data type of an output port.
C Syntax	<code>void ssSetOutputPortDataType(SimStruct *S, input_T port, DType id)</code>
Ada Syntax	<pre>procedure ssSetOutputPortDataType(S : in SimStruct; port : in Integer := 0; id : in Integer);</pre>
Arguments	<p>S SimStruct representing an S-function block or a Simulink model.</p> <p>port Index of an input port</p> <p>id Id of data type accepted by port</p>
Description	<p>Use this function in mdl I n i t i a l i z e S i z e s to set the data type of the output port specified by port. If the input port's data type is determined dynamically, for example, from the data type of a block parameter, set the data type to DYNAMI CALLY_TYPED.</p> <hr/> <p>Note The data type of an output port is doubl e (real _T) by default.</p> <hr/>
Languages	Ada, C
Example	<p>Suppose that you want to create an S-function with two input ports, the first of which gets its data type from a block parameter and the second of which outputs signals of type i n t 16_T. The following code sets up the data types.</p> <pre>ssSetI nputPortDataType(S, 0, DYNAMI CALLY_TYPED) ssSetI nputPortDataType(S, 1, SS_I NT16)</pre>
See Also	ssGetOutputPortDataType

Purpose	Specify information about the dimensionality of an output port.
Syntax	<code>void ssSetInputPortDimensionInfo(SimStruct *S, int_T port, DimInfo_T *dimInfo)</code>
Arguments	<p><code>S</code> SimStruct representing an S-function block.</p> <p><code>port</code> Index of an output port</p> <p><code>dimInfo</code> Structure of type <code>DimInfo_T</code> that specifies the dimensionality of the signals emitted by port</p> <p>See <code>ssSetInputPortDimensionInfo</code> for a description of this structure.</p>
Description	Specifies the dimension information for port. Use this function in <code>mdlInitializeSizes</code> to initialize the output port dimension info. If you want the port to inherit its dimensionality from the block to which it is connected, specify <code>DYNAMIC_DIMENSION</code> as the <code>dimInfo</code> for port.
Languages	C
Example	<p>The following example specifies that input port 0 accepts 2-by-2 matrix signals.</p> <pre>DECL_AND_INIT_DIMINFO(dim); dim.numDims = 2; int dims[2]; dims[0] = 2; dims[1] = 2; dim.dims = &dims; dim.width = 4; ssSetOutputPortDimensionInfo(S, 0, &dim);</pre>
See Also	<code>ssSetInputPortDimensionInfo</code>

ssSetOutputPortFrameData

Purpose	Specify whether a port outputs framed data.
Syntax	<code>void ssSetOutputPortFrameData(SimStruct *S, int_T port, int_T outputsFrames)</code>
Arguments	<p><code>S</code> SimStruct representing an S-function block.</p> <p><code>port</code> Index of an output port</p> <p><code>outputsFrames</code> Type of signal output by port. Acceptable values are -1 (either frame or unframed input), 0 (unframed input only), 1 (framed input only).</p>
Description	Use in <code>mdlSetInputPortFrameData</code> to specify whether an output port issues frame data only, unframed data only, or both.
Languages	C
See Also	<code>ssGetOutputPortFrameData</code> , <code>mdlSetInputPortFrameData</code>

ssSetOutputPortMatrixDimensions

Purpose	Specify dimension information for an output port that emits matrix signals.
Syntax	<pre>void ssSetOutputPortMatrixDimensions(SimStruct *S, int_T port, int_T m, int_T n)</pre>
Arguments	<p>S SimStruct representing an S-function block.</p> <p>port Index of an input port</p> <p>m Row dimension of matrix signals emitted by port or DYNAMICALLY_SIZED</p> <p>n Column dimension of matrix signals emitted by port or DYNAMICALLY_SIZED</p>
Description	Specifies that port emits an m-by-n matrix signal. If either dimension is DYNAMICALLY_SIZED, the other must be DYNAMICALLY_SIZED or 1.
Languages	C
Example	The following example specifies that input port 0 emits 2-by-2 matrix signals. <pre>ssSetOutputPortDimensions(S, 0, 2, 2);</pre>
See Also	ssSetOutputPortDimensions

ssSetOutputPortOffsetTime

Purpose	Specify the offset time of an output port.
Syntax	<code>ssSetOutputPortOffsetTime(SimStruct *S, outputPortIdx, offset)</code>
Arguments	<p><code>S</code> SimStruct representing an S-function block.</p> <p><code>outputPortIdx</code> Index of the output port whose sample time is being set.</p> <p><code>period</code> Sample time of output port.</p>
Description	Use in <code>mdlInitializeSizes</code> (after <code>ssSetNumOutputPorts</code>) to specify the sample time offset value for each output port index. This should only be used if you have specified the S-function's sample times as port-based.
Languages	C
See Also	<code>ssSetNumOutputPorts</code> , <code>ssSetOutputPortSampleTime</code>

Purpose	Specify that an output port is reusable.
Syntax	<code>ssSetOutputPortReusable(SimStruct *S, <i>outputPortIdx</i>, <i>isReusable</i>)</code>
Arguments	<p><i>S</i> SimStruct representing an S-function block.</p> <p><i>outputPortIdx</i> Index of the output port whose reusability is being set.</p> <p><i>isReusable</i> Value specifying reusability of port</p>
Description	<p>Use in <code>mdlInitializeSizes</code> (after <code>ssSetNumOutputPorts</code>) to specify whether output ports have a test point. This macro can take on two values:</p> <ul style="list-style-type: none"> • Off (<i>isReusable</i>=0) — specifies that the output port is not reusable. This is the default. • On (<i>isReusable</i>=1) — specifies that the output port is reusable. <p>In Simulink, reusable signals share the same memory space. When this macro is turned on, the output port signal to the S-function may be reused by other signals in the model. This reuse results in less memory use during Simulink simulation and more efficiency in the Real-Time Workshop generated code.</p> <p>When you mark an output port as reusable, your S-function must update the output once in <code>mdlOutputs</code>. It cannot expect the previous output value to be persistent.</p> <p>By default, the output port signals are not reusable. This forces Simulink's simulation engine (and the Real-Time Workshop) to allocate global memory for these output port signals. Hence this memory is only written to by your S-function and persists between model execution steps.</p>
Languages	C
See Also	<code>ssSetNumOutputPorts</code> , <code>ssSetInputPortReusable</code>

ssSetOutputPortSampleTime

Purpose	Specify the sample time of an output port.
Syntax	<code>ssSetOutputPortSampleTime(SimStruct *S, outputPortIdx, period)</code>
Arguments	<p><code>S</code> SimStruct representing an S-function block.</p> <p><code>outputPortIdx</code> Index of the output port whose sample time is being set.</p> <p><code>period</code> Sample time of output port.</p>
Description	Use in <code>mdlInitializeSizes</code> (after <code>ssSetNumOutputPorts</code>) to specify the sample time period as continuous or as a discrete value for each output port index. This should only be used if you have specified port-based sample times.
Languages	C
See Also	<code>ssSetNumOutputPorts</code> , <code>ssSetOutputPortOffsetTime</code>

Purpose	Specify dimension information for an output port that emits vector signals.
Syntax	<code>void ssSetOutputPortVectorDimension(SimStruct *S, int_T port, int_T w)</code>
Arguments	<p><code>S</code> SimStruct representing an S-function block.</p> <p><code>port</code> Index of an output port</p> <p><code>w</code> Width of vector or DYNAMICALLY_SIZED</p>
Description	Specifies that port emits a w-element vector signal.
<hr/> <p>Note This macro and <code>ssSetOutputPortWidth</code> are functionally identical.</p> <hr/>	
Example	<p>The following example specifies that output port 0 emits an 8-element matrix signal.</p> <pre>ssSetOutputPortVectorDimension(S, 0, 8);</pre>
Languages	C
See Also	<code>ssSetOutputPortDimensionInfo</code> , <code>ssSetOutputPortWidth</code>

ssSetOutputPortWidth

Purpose	Specify the width of an output port.
C Syntax	<code>void ssSetOutputPortWidth(SimStruct *S, int_T port, int_T width)</code>
Ada Syntax	<code>procedurw ssSetOutputPortWidth(S : in SimStruct; port : in Integer := 0; Width : in Integer);</code>
Arguments	<p>S SimStruct representing an S-function block.</p> <p>port Index of the output port whose width is being set.</p> <p>width Width of output port.</p>
Description	Use in <code>mdlInitializeSizes</code> (after <code>ssSetNumOutputPorts</code>) to specify a nonzero positive integer width or <code>DYNAMICALLY_SIZED</code> for each output port index starting at 0.
Languages	Ada, C
See Also	<code>ssSetNumOutputPorts</code> , <code>ssSetInputPortWidth</code>

Purpose	Set the name of a parameter.
Syntax	<pre>procedure ssSetParameterName (S : in SimStruct; Parameter : in Integer; Name : in String);</pre>
Arguments	<p>S SimStruct representing an S-function block.</p> <p>Parameter Index of a parameter</p> <p>Name Name of the parameter</p>
Description	Sets the name of Parameter to Name.
Languages	Ada

ssSetParameterTunable

Purpose	Set the tunability of a parameter.
Syntax	<pre>procedure ssSetParameterTunable (S : in SimStruct; Parameter : in Integer; IsTunable : in Boolean);</pre>
Arguments	<p>S SimStruct representing an S-function block.</p> <p>Parameter Index of a parameter</p> <p>IsTunable true indicates that the parameter is tunable.</p>
Description	Sets the tunability of Parameter to the value of IsTunable.
Languages	Ada

Purpose	Specify the name of the placement group of a block.
Syntax	<code>void ssSetPlacementGroup(SimStruct *S, const char *groupName)</code>
Arguments	<p><code>S</code> SimStruct representing an S-function block. The block must be either a source block (i.e., a block without input ports) or a sink block (i.e., a block without output ports).</p> <p><code>groupName</code> Name of placement group name of the block represented by <code>S</code>.</p>
Description	<p>Use this macro to specify the name of the placement group to which the block represented by <code>S</code> belongs. S-functions that share the same placement group name are placed adjacent to each other in the block execution order list for the model. This macro should be invoked in <code>mdlInitializeSes</code>.</p> <hr/> <p>Note This macro is typically used to create Real-Time Workshop device driver blocks.</p> <hr/>
Languages	C
See Also	<code>ssGetPlacementGroup</code>

ssSetRunTimeParamInfo

Purpose	Specify the attributes of a run-time parameter.
Syntax	<code>void ssSetRunTimeParamInfo(SimStruct *S, int_T param, ssParamRec *info)</code>
Arguments	<code>S</code> SimStruct representing an S-function block. <code>param</code> Index of a run-time parameter
Description	Use this function in <code>mdlSetWorkWidths</code> or <code>mdlProcessParameters</code> to specify information about a run-time parameter. Use a <code>ssParamRec</code> structure to pass the parameter attributes to the function.

ssParamRec Structure

The `simstruc.h` macro defines this structure as follows.

```
typedef struct ssParamRec_tag {
    const char *name;
    int_T      nDimensions;
    int_T      *dimensions;
    DTypeId    dataTypeId;
    boolean_T  complexSignal;
    void       *data;
    const void *dataAttributes;
    int_T      nDlgParamIndices;
    int_T      *dlgParamIndices;
    TransformedFlag transformed; /* Transformed status */
    boolean_T  outputAsMatrix; /* Write out parameter as a
vector (false)
                                * [default] or a matrix (true)
                                */
} ssParamRec;
```

The record contains the following fields.

`name`. Name of the parameter. This must point to persistent memory. Do not set to a local variable (`static char name[32]` or strings name are okay).

`nDimensions`. Number of dimensions that this parameter has

Purpose	Write a vector parameter in MATLAB format to the model . rtw file.
Syntax	<pre>int_T ssWriteRTWMxVectParam(SimStruct *S, const char_T *name, const void *rValue, const void *iValue, int_T dataType, int_T size)</pre>
Arguments	<p>S SimStruct representing an S-function block.</p> <p>name Parameter name</p> <p>rValue Real values of parameter</p> <p>cValue Complex values of parameter</p> <p>dataType Data type of parameter elements (see "Specifying Data Type Info" on page 10-199)</p> <p>size Number of elements in vector</p>
Description	Use this function in mdl RTW to write a vector parameter in Simulink format to this S-function's model . rtw file. This function returns TRUE if successful.
Languages	C
See Also	mdl RTW, ssWriteRTWMxVectParam

ssWriteRTWMx2dMatParam

Purpose	Write a matrix parameter in MATLAB format to the model . rtw file.
Syntax	<pre>int_T ssWriteRTWMx2dMatParam(SimStruct *S, const char_T *name, const void *rValue, const void *iValue, int_T dataType, int_T nRows, int_T nCols)</pre>
Arguments	<p>S SimStruct representing an S-function block.</p> <p>name Parameter name</p> <p>rValue Real elements of parameter array</p> <p>iValue Imaginary elements of parameter array</p> <p>dataType Data type of parameter elements (see “Specifying Data Type Info” on page 10-199)</p> <p>nRows Number of rows in matrix</p> <p>nColumns Number of columns in matrix</p>
Description	Use this function in mdl RTW to write a matrix parameter in MATLAB format to this S-function’s model . rtw file. This function returns TRUE if successful.
Languages	C
See Also	mdl RTW, ssWriteRTW2dMatParam

Purpose	Write tunable parameter information to model .rtw file.
Syntax	<code>int_T ssWriteRTWParameters(SimStruct *S, int_T nParams, int_T paramType, const char_T *paramName, const char_T *stringInfo, ...)</code>
Arguments	<p><code>S</code> SimStruct representing an S-function block.</p> <p><code>nParams</code> Number of tunable parameters</p> <p><code>paramType</code> Type of parameter (see "Parameter Type-Specific Arguments")</p> <p><code>paramName</code> Name of parameter</p> <p><code>stringInfo</code> General information about the parameter, such as how it was derived</p> <p>...</p> <p>Remaining arguments depend on parameter type (see "Parameter Type-Specific Arguments").</p>
Description	Use this function in mdl RTW to write tunable parameter information to this S-function's model .rtw file. This function returns TRUE if successful.

Note This function is provided for compatibility with S-functions that do not use run-time parameters. It is suggested that you use run-time parameters (see "Run-Time Parameters" on page 7-6). If you do use run-time parameters, you do not need to use this function.

Parameter Type-Specific Arguments

This section lists the parameter-specific arguments required by each parameter type.

ssWriteRTWParameters

- SS_WRITE_VALUE_VECT (vector parameter)

Argument	Description
const real_T *valueVect	Pointer to array of vector values
int_T vectLen	Length of vector

- SSWRITE_VALUE_2DMAT (matrix parameter)

Argument	Description
const real_T *valueMat	Pointer to array of matrix elements
int_T nRows	Number of rows in matrix
int_T nCols	Number of columns in matrix

- SSWRITE_VALUE_DTYPE_2DMAT

Argument	Description
const real_T *valueMat	Pointer to array of matrix elements
int_T nRows	Number of rows in matrix
int_T nCols	Number of columns in matrix
int_T dtInfo	Data type of matrix elements (see "Specifying Data Type Info" on page 10-199)

- SSWRITE_VALUE_DTYPE_ML_VECT

Argument	Description
const void *rValueVect	Real component of complex vector
const void *iValueVect	Imaginary component of complex vector

Argument	Description
<code>int_T vectLen</code>	Length of vector
<code>int_T dtInfo</code>	Data type of vector (see “Specifying Data Type Info” on page 10-199)

- `SSWRITE_VALUE_DTYPE_ML_2DMAT`

Argument	Description
<code>const void *rValueMat</code>	Real component of complex matrix
<code>const void *iValueMat</code>	Imaginary component of complex matrix
<code>int_T nRows</code>	Number of rows in matrix
<code>int_T nCols</code>	Number of columns in matrix
<code>int_T dtInfo</code>	Data type of matrix

Specifying Data Type Info

The data type of value argument passed to the `ssWriteRTW` macros is obtained using

```
DTINFO(dTypeId, isComplex),
```

where `dTypeId` can be any one of the enum values in `BuiltInTypeID` (`SS_DOUBLE`, `SS_SINGLE`, `SS_INT8`, `SS_UINT8`, `SS_INT16`, `SS_UINT16`, `SS_INT32`, `SS_UINT32`, `SS_BOOLEAN`) defined in `simstuc_types.h`. The `isComplex` argument is either 0 or 1.

For example, `DTINFO(SS_INT32, 0)` is a noncomplex 32-bit signed integer.

If `isComplex==1`, it is assumed that the array of values has the real and imaginary parts arranged in an interleaved manner (i.e., Simulink format). If you prefer to pass the real and imaginary parts as two separate arrays, you should use the macros `ssWriteRTWxVectParam` or `ssWriteRTWx2dMatParam`.

Example

See `simulink/src/sfun_multiply.c` for an example that uses this function.

ssWriteRTWParameters

Languages C

See Also mdl RTW

Purpose Write tunable parameter settings to model . rtw file.

Syntax `int_T ssWriteRTWParamSettings(SimStruct *S, int_T nParamSettings, int_T paramType, const char_T *settingName, ...)`

Arguments

`S`
SimStruct representing an S-function block.

`nParamSettings`
Number of tunable parameter settings

`settingType`
Type of parameter (see "Parameter Setting Type-Specific Arguments")

`settingName`
Name of parameter setting

...

Remaining arguments depend on parameter type (see "Parameter Setting Type-Specific Arguments").

Description Use this function in mdl RTW to write tunable parameter setting information to this S-function's model . rtw file. This function returns TRUE if successful.

Parameter Setting Type-Specific Arguments

This section lists the parameter-specific arguments required by each parameter type.

- SSWRITE_VALUE_STR (unquoted string)

Argument	Description
<code>const char_T *value</code>	string (Example: U. S. A.)

- SSWRITE_VALUE_QSTR (quoted string)

Argument	Description
<code>const char_T *value</code>	string (Example: "U. S. A. ")

ssWriteRTWParamSettings

- SSWRITE_VALUE_VECT_STR (vector of strings)

Argument	Description
<code>const char_T *value</code>	Vector of strings (e.g., ["USA", "Mexico"])
<code>int_T nItemsInVect</code>	Size of vector

- SSWRITE_VALUE_NUM (number)

Argument	Description
<code>const real_T value</code>	Number (e.g., 2)

- SSWRITE_VALUE_VECT (vector of numbers)

Argument	Description
<code>const real_T *value</code>	Vector of numbers (e.g., [300, 100])
<code>int_T vectLen</code>	Size of vector

- SSWRITE_VALUE_2DMAT (matrix of numbers)

Argument	Description
<code>const real_T *value</code>	Matrix of numbers (e.g., [[170, 130], [60, 40]])
<code>int_T nRows</code>	Number of rows in vector
<code>int_T nCols</code>	Number of columns in vector

- SSWRITE_VALUE_DTYPE_NUM (data typed number)

Argument	Description
const void *value	Number (e.g., [3+4i])
int_T dtInfo	Data type (see "Specifying Data Type Info" on page 10-199)

- SSWRITE_VALUE_DTYPE_VECT (data typed vector)

Argument	Description
const void *value	Data typed vector (e.g., [1+2i, 3+4i])
int_T vectLen	Size of vector
int_T dtInfo	Data type (see "Specifying Data Type Info" on page 10-199)

- SSWRITE_VALUE_DTYPE_2DMAT (data typed matrix)

Argument	Description
const void *value	Matrix (e.g., [1+2i 3+4i; 5 6])
int_T nRows	Number of rows in matrix
int_T nCols	Number of columns in matrix
int_T dtInfo	Data type (see "Specifying Data Type Info" on page 10-199)

- SSWRITE_VALUE_DTYPE_ML_VECTOR (data typed MATLAB vector)

Argument	Description
const void *RValue	Real component of vector (e.g., [1 3])
const void *IValue	Imaginary component of vector (e.g., [2 5])

ssWriteRTWParamSettings

Argument	Description
<code>int_T vectLen</code>	Number of elements in vector
<code>int_T dtInfo</code>	Data type (see "Specifying Data Type Info" on page 10-199)

- `SSWRITE_VALUE_DTYPE_ML_2DMAT` (data typed MATLAB matrix)

Argument	Description
<code>const void *RValue</code>	Real component of matrix (e.g., [1 5 3 6])
<code>const void *IValue</code>	Real component of matrix (e.g., [2 0 4 0])
<code>int_T nRows</code>	Number of rows in matrix
<code>int_T nCols</code>	Number of columns in matrix
<code>int_T dtInfo</code>	Data type (see "Specifying Data Type Info" on page 10-199)

Example See `simulink/src/sfun_multipart.c` for an example that uses this function.

Languages C

See Also `mdlRTW`

Purpose	Write a scalar parameter to the model .rtw file.
Syntax	<pre>int_T ssWriteRTWStr(SimStruct *S, const char_T *name, const void *value)</pre>
Arguments	<p>S SimStruct representing an S-function block.</p> <p>name Parameter name</p> <p>value Parameter value</p>
Description	Use this function in mdl RTW to write scalar parameters to this S-function's model .rtw file. This function returns TRUE if successful.
Languages	C
See Also	mdl RTW

ssWriteRTWStr

Purpose	Write a string to the model . rtw file.
Syntax	<code>int_T ssWriteRTWStr(SimStruct *S, const char_T *str)</code>
Arguments	<code>S</code> SimStruct representing an S-function block. <code>str</code> String
Description	Use this function in mdl RTW to write strings to this S-function's model . rtw file. This function returns TRUE if successful.
Languages	C
See Also	mdl RTW

Purpose	Write a string parameter to the model .rtw file.
Syntax	<code>int_T ssWriteRTWStr(SimStruct *S, const char_T *name, const char_T *value)</code>
Arguments	<p><code>S</code> SimStruct representing an S-function block.</p> <p><code>name</code> Parameter name</p> <p><code>value</code> Parameter value</p>
Description	Use this function in mdl RTW to write string parameters to this S-function's model .rtw file. This function returns TRUE if successful.
Languages	C
See Also	mdl RTW

ssWriteRTWStrVectParam

Purpose	Write a string vector parameter to the model .rtw file.
Syntax	<code>int_T ssWriteRTWStrVectParam(SimStruct *S, const char_T *name, const void *value, int_T size)</code>
Arguments	<p>S SimStruct representing an S-function block.</p> <p>name Parameter name</p> <p>value Parameter values</p> <p>size Number of elements in vector</p>
Description	Use this function in mdl RTW to write a vector of string parameters to this S-function's model .rtw file. This function returns TRUE if successful.
Languages	C
See Also	mdl RTW

Purpose	Write a vector parameter to the model . rtw file.
Syntax	<code>int_T ssWriteRTWStrVectParam(SimStruct *S, const char_T *name, const void *value, int_T dataType, int_T size)</code>
Arguments	<p><code>S</code> SimStruct representing an S-function block.</p> <p><code>name</code> Parameter name</p> <p><code>value</code> Parameter values</p> <p><code>dataType</code> Data type of parameter elements (see "Specifying Data Type Info" on page 10-199)</p> <p><code>size</code> Number of elements in vector</p>
Description	Use this function in mdl RTW to write a vector parameter in Simulink format to this S-function's model . rtw file. This function returns TRUE if successful.
Languages	C
See Also	mdl RTW, ssWriteRTWMxVectParam

ssWriteRTWorkVect

Purpose	Write work vectors to model.rtw file.
Syntax	<pre>int_T ssWriteRTWorkVect(SimStruct *S, const char_T *vectName, int_T nNames, const char_T *name1, int_T size1, ..., const char_T *nameN, int_T sizeN)</pre>
Arguments	<p>S SimStruct representing an S-function block.</p> <p>vectName Name of work vector (must be "RWork", "IWork" or "PWork")</p> <p>settingType Type of parameter (see "Parameter Setting Type-Specific Arguments")</p> <p>name1 ... nameN Names of groups of work vector elements</p> <p>size1 ... sizeN Size of each element group (the total of the sizes must equal the size of the work vector)</p>
Description	Use this function in mdl RTW to write work vectors to this S-function's model.rtw file. This function returns TRUE if successful.
Languages	C
See Also	mdl RTW

Purpose	Write a matrix parameter to the model . rtw file.
Syntax	<code>int_T ssWriteRTW2dMatParam(SimStruct *S, const char_T *name, const void *value, int_T dataType, int_T nRows, int_T nCols)</code>
Arguments	<p><code>S</code> SimStruct representing an S-function block.</p> <p><code>name</code> Parameter name</p> <p><code>value</code> Parameter values</p> <p><code>dataType</code> Data type of parameter elements (see "Specifying Data Type Info" on page 10-199)</p> <p><code>nRows</code> Number of rows in matrix</p> <p><code>nColumns</code> Number of columns in matrix</p>
Description	Use this function in mdl RTW to write a vector of numeric parameters to this S-function's model . rtw file. This function returns TRUE if successful.
Languages	C
See Also	mdl RTW

ssWriteRTW2dMatParam

A

additional parameters for S-functions 2-19

B

block-based sample times 7-16

C

C MEX S-functions 1-2, 4-1, 5-1, 6-1

callback methods 1-9

continuous block, setting sample time 7-21

continuous state S-function example (C MEX)
7-34

continuous state S-function example (M-file) 2-8

D

direct feedthrough 1-11

direct index lookup table example 8-24

discrete state S-function example (C MEX) 7-38

discrete state S-function example (M-file) 2-11

dynamically sized inputs 1-12

E

examples

continuous state S-function (C MEX) 7-34

continuous state S-function (M-file) 2-8

direct index lookup table 8-24

discrete state S-function (C MEX) 7-38

discrete state S-function (M-file) 2-11

hybrid system S-function (C MEX) 7-42

hybrid system S-function (M-file) 2-13

pointer work vector 7-26

sample time for continuous block 7-21

sample time for hybrid block 7-22

variable step S-function (C MEX) 7-45

variable step S-function (M-file) 2-16

exception free code 7-31

H

hybrid block, setting sample time 7-22

hybrid system S-function example (C MEX) 7-42

hybrid system S-function example (M-file) 2-13

I

input arguments for M-file S-functions 2-6

inputs, dynamically sized 1-12

M

masked multiport S-functions 7-13

matrix.h 3-10

mdl CheckParameters 9-3

mdl Derivatives 9-5

mdl GetTimeOfNextVarHit 9-6

mdl InitializeConditions 9-7

mdl InitializeSampleTimes 9-9

mdl InitializeSizes 1-12, 2-4, 9-13

mdl OutputFunction 7-21

mdl Outputs 9-17

mdl ProcessParameters 9-18

mdl RTW 8-21, 9-20

mdl SetDefaultPortComplexSignals 9-21

mdl SetDefaultPortDataTypes 9-22

mdl SetDefaultPortDimensionInfo 9-23

mdl SetInputPortComplexSignal 9-24

mdl SetInputPortDataType 9-25

mdl SetInputPortDimensionInfo 9-26

mdl SetInputPortFrameData 9-28

