

Simulink[®]

Simulation and Model-Based Design

- Modeling
- Simulation
- Implementation

Writing S-Functions

Version 6



How to Contact The MathWorks:



www.mathworks.com Web
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
service@mathworks.com Order status, license renewals, passcodes
info@mathworks.com Sales, pricing, and general information



508-647-7000 Phone



508-647-7001 Fax



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

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

Writing S-Functions

© COPYRIGHT 1998–2005 by The MathWorks, Inc.

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

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

Trademarks

MATLAB, Simulink, Stateflow, Handle Graphics, Real-Time Workshop, and xPC TargetBox are registered trademarks of The MathWorks, Inc.

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

Patents

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

Revision History

October 1998	First printing	Revised for Simulink 3.0 (Release R11)
November 2000	Second printing	Revised for Simulink 4.0 (Release R12)
July 2002	Third printing	Revised for Simulink 5.0 (Release R13)
April 2003	Online only	Revised for Simulink 5.1 (Release R13SP1)
April 2004	Online only	Revised for Simulink 5.1.1 (Release R13SP1+)
June 2004	Online only	Revised for Simulink 6.0 (Release R14)
October 2004	Online only	Revised for Simulink 6.1 (Release R14SP1)
March 2005	Online only	Revised for Simulink 6.2 (Release R14SP2)
September 2005	Online Only	Revised for Simulink 6.3 (Release R14SP3)

Overview of S-Functions

1

What Is an S-Function?	1-2
Using S-Functions in Models	1-3
Passing Parameters to S-Functions	1-4
When to Use an S-Function	1-5
How S-Functions Work	1-6
Mathematics of Simulink Blocks	1-6
Simulation Stages	1-6
S-Function Callback Methods	1-8
Implementing S-Functions	1-9
M-File S-Functions	1-9
MEX-File S-Functions	1-10
S-Function Concepts	1-12
Direct Feedthrough	1-12
Dynamically Sized Arrays	1-13
Setting Sample Times and Offsets	1-14
S-Function Examples	1-18
M-File S-Function Examples	1-19
C S-Function Examples	1-21
Fortran S-Function Examples	1-25
C++ S-Function Examples	1-26
Ada S-Function Examples	1-26

Writing S-Functions in M

2

Introduction	2-2
Writing Level-2 M-File S-Functions	2-3
About Level-2 M-File S-Functions	2-3
Level-2 M-File S-Function API	2-3
M-File S-Function Demos	2-4
S-Function Template	2-4
Instantiating a Level-2 M-File S-Function	2-5
Generating Code from a Level-2 M-File S-Function	2-5
Callback Methods	2-5
Setup Method	2-6
Run-time Object	2-7
Maintaining Level-1 M-File S-Functions	2-8
S-Function Arguments	2-8
S-Function Outputs	2-9
Defining S-Function Block Characteristics	2-10
Processing S-Function Parameters	2-11

Writing S-Functions in C

3

Introduction	3-2
Creating C MEX S-Functions	3-3
Building S-Functions Automatically	3-5
Deploying the Generated S-Function	3-7
How the S-Function Builder Builds an S-Function	3-7
Setting the Include Path	3-8
S-Function Builder Dialog Box	3-9
Parameters/S-Function Name Pane	3-10
Port/Parameter Pane	3-11
Initialization Pane	3-11
Data Properties Pane	3-13

Input Ports Pane	3-14
Output Ports Pane	3-15
Parameters Pane	3-17
Data Type Attributes Pane	3-18
Libraries Pane	3-18
Outputs Pane	3-20
Continuous Derivatives Pane	3-24
Discrete Update Pane	3-25
Build Info Pane	3-27
Example of a Basic C MEX S-Function	3-29
Defines and Includes	3-31
Callback Implementations	3-32
Simulink/Real-Time Workshop Interface	3-34
Building the Timestwo Example	3-34
Templates for C S-Functions	3-35
S-Function Source File Requirements	3-35
The SimStruct	3-37
Compiling C S-Functions	3-37
How Simulink Interacts with C S-Functions	3-39
Process View	3-39
Data View	3-43
Writing Callback Methods	3-47
Converting Level 1 C MEX S-Functions to Level 2	3-48
Obsolete Macros	3-50

Creating C++ S-Functions

4

Source File Format	4-2
Making C++ Objects Persistent	4-6

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	5-7
Callbacks Invoked by Simulink	5-7
Implementing Callbacks	5-8
Omitting Optional Callback Methods	5-8
SimStruct Functions	5-8
Building an Ada S-Function	5-10
Ada Compiler Requirements	5-10
Example of an Ada S-Function	5-11
Timestwo Package Specification	5-12
Timestwo Package Body	5-13
mdlInitializeSizes	5-16
mdlOutputs	5-16
Building the Timestwo Example	5-17

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
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
Example C MEX S-Function Calling Fortran Code	6-14
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

Dialog Parameters	7-2
Tunable Parameters	7-3
Run-Time Parameters	7-6
Creating Run-Time Parameters	7-7
Updating Run-Time Parameters	7-8
Tuning Runtime Parameters	7-9
Creating Input and Output Ports	7-10
Creating Input Ports	7-10
Creating Output Ports	7-12
Scalar Expansion of Inputs	7-13
Masked Multiport S-Functions	7-15
Custom Data Types	7-16
Sample Times	7-17
Block-Based Sample Times	7-18
Specifying Port-Based Sample Times	7-21

Hybrid Block-Based and Port-Based Sample Times	7-25
Multirate S-Function Blocks	7-26
Synchronizing Multirate S-Function Blocks	7-28
Specifying Model Reference Sample Time Inheritance	7-29
Work Vectors	7-31
Work Vectors and Zero Crossings	7-33
Example Involving a Pointer Work Vector	7-33
Memory Allocation	7-35
Function-Call Subsystems	7-36
Processing Frame-Based Signals	7-39
Handling Errors	7-41
Exception Free Code	7-41
ssSetErrorStatus Termination Criteria	7-43
Checking Array Bounds	7-43
S-Function Examples	7-44
Example of a Continuous State S-Function	7-44
Example of a Discrete State S-Function	7-49
Example of a Hybrid System S-Function	7-54
Example of a Variable-Step S-Function	7-59
Example of a Zero Crossing S-Function	7-63
Example of a Time-Varying Continuous Transfer Function	7-78

S-Function Callback Methods — Alphabetical List

8

SimStruct Functions — Categorical List

9

Introduction	9-2
Language Support	9-2
The SimStruct	9-2
SimStruct Macros and Functions Listed by Usage	9-3
Dialog Box Parameters	9-9
Run-Time Parameters	9-10

Index

Overview of S-Functions

S-functions (system-functions) provide a powerful mechanism for extending the capabilities of Simulink®. The following sections explain what an S-function is and when and why you might use one and how to write your own S-functions.

What Is an S-Function? (p. 1-2)	Brief overview of S-functions.
Using S-Functions in Models (p. 1-3)	How to insert S-functions as blocks in a model and pass parameters to them.
How S-Functions Work (p. 1-6)	How Simulink invokes S-functions when simulating a model that includes them.
Implementing S-Functions (p. 1-9)	How to write S-functions.
S-Function Concepts (p. 1-12)	Key concepts needed to write certain types of S-functions.
S-Function Examples (p. 1-18)	Examples that illustrate the creation of various types of S-functions and S-function features.

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 (see Building MEX-Files in the External Interfaces User's Guide). As with other MEX-files, they are dynamically linked into MATLAB when needed.

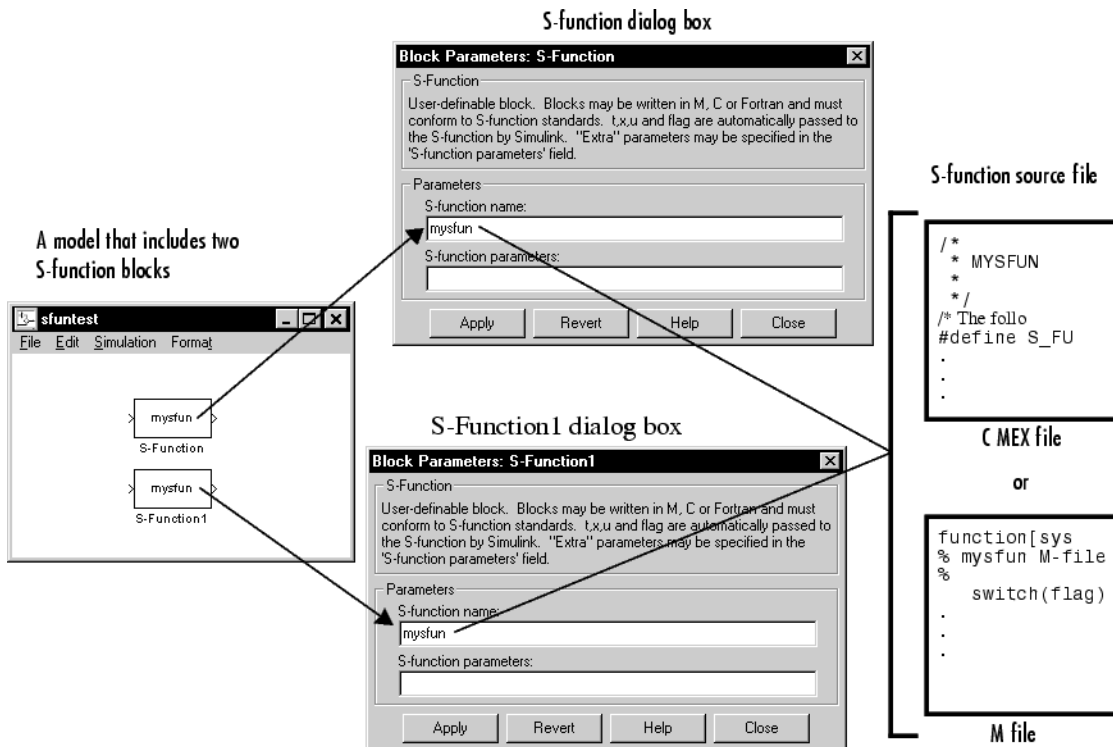
S-functions use a special calling syntax that enables you to interact with Simulink equation solvers. This interaction is very similar to the interaction that takes place between the solvers and built-in Simulink blocks. The form of an S-function is very general and can accommodate continuous, discrete,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

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 write your S-function and place its name in an S-Function block (available in the User-Defined Functions block library), you can customize the user interface by using masking.

You can use S-functions with Real-Time Workshop®. You can also customize the code generated by Real-Time Workshop for S-functions by writing a Target Language Compiler (TLC) file. See “Writing S-Functions for Real-Time Workshop” in the “Real-Time Workshop User's Guide” Real-Time Workshop User's Guide for more information.

Using S-Functions in Models

To incorporate an S-function into a Simulink model, drag an S-Function block from the Simulink User-Defined Functions 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 following figure.



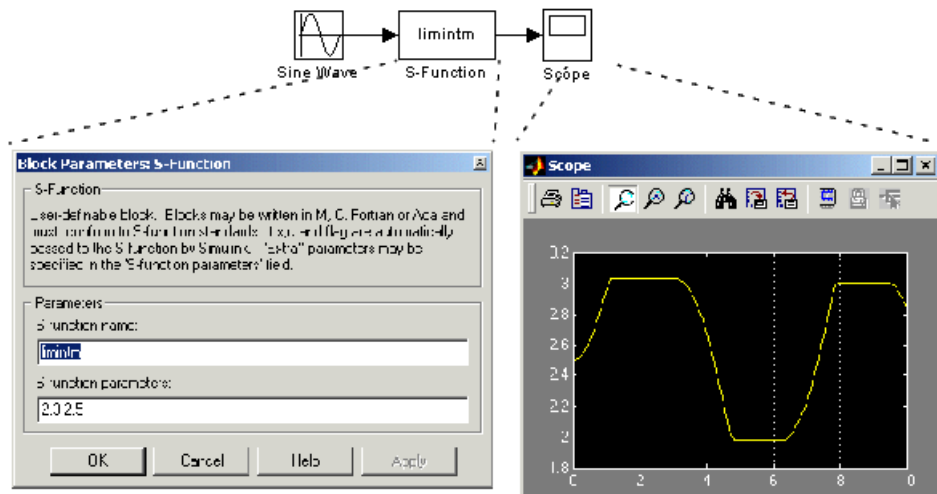
Relationship Between an S-Function Block, Its Dialog Box, and the Source File That Defines the Block's Behavior

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 have 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 the parameters the S-function requires and the order in which the function requires them. (If you do not know, consult the S-function's author, documentation, or source code.) Enter the parameters, separated by a comma, in the order required by the S-function. The parameter values can be constants, names of variables defined in the model's workspace, or MATLAB expressions.

The following example illustrates usage of the **S-function parameters** field to enter user-defined parameters.



The model in this example incorporates `limintm`, a sample S-function that comes with Simulink. The function's source code resides in `toolbox/simulink/blocks`. The `limintm` function accepts three parameters: a lower bound, an upper bound, and an initial condition. It outputs the time integral of the input signal if the time integral is between the lower and upper bounds, the lower bound if the time integral is less than the lower bound, and the upper bound if the time integral is greater than the upper bound. The dialog box in the example specifies a lower and upper bound and an initial

condition of 2, 3, and 2.5, respectively. The scope shows the resulting output when the input is a sine wave of amplitude 1.

See “Processing S-Function Parameters” on page 2-11 and “Handling Errors” on page 7-41 for information on how to access user-specified parameters in an S-function.

You can use the Simulink masking facility to create custom dialog boxes and icons for your S-function blocks. Masked dialog boxes can make it easier to specify additional parameters for S-functions. For discussions of additional parameters and masking, see the Using Simulink documentation.

When to Use an S-Function

The most common use of S-functions is to create custom Simulink blocks. You can use S-functions for a variety of applications, including

- Adding new general purpose blocks to Simulink
- Adding blocks that represent hardware device drivers
- Incorporating existing C code into a simulation
- Describing a system as a set of mathematical equations
- Using graphical animations (see the inverted pendulum demo, penddemo)

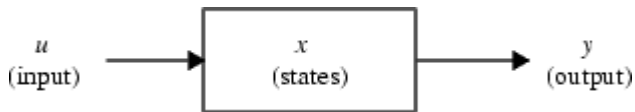
An advantage of using S-functions is that you can build a general-purpose block that you can use many times in a model, varying parameters with each instance of the block.

How S-Functions Work

To create S-functions, you need to know how S-functions work. Understanding how S-functions work, in turn, requires understanding how Simulink simulates a model, and this, in turn, requires an understanding of the mathematics of blocks. This section therefore begins by explaining the mathematical relationship between a block's inputs, states, and outputs.

Mathematics of Simulink Blocks

A Simulink block consists of a set of inputs, a set of states, and a set of outputs, where the outputs are a function of the sample time, the inputs, and the block's states.



The following equations express the mathematical relationships between the inputs, outputs, and the states.

$$y = f_0(t, x, u) \quad (\text{Output})$$

$$\dot{x}_c = f_d(t, x, u) \quad (\text{Derivative})$$

$$x_{d_{k+1}} = f_u(t, x, u) \quad (\text{Update})$$

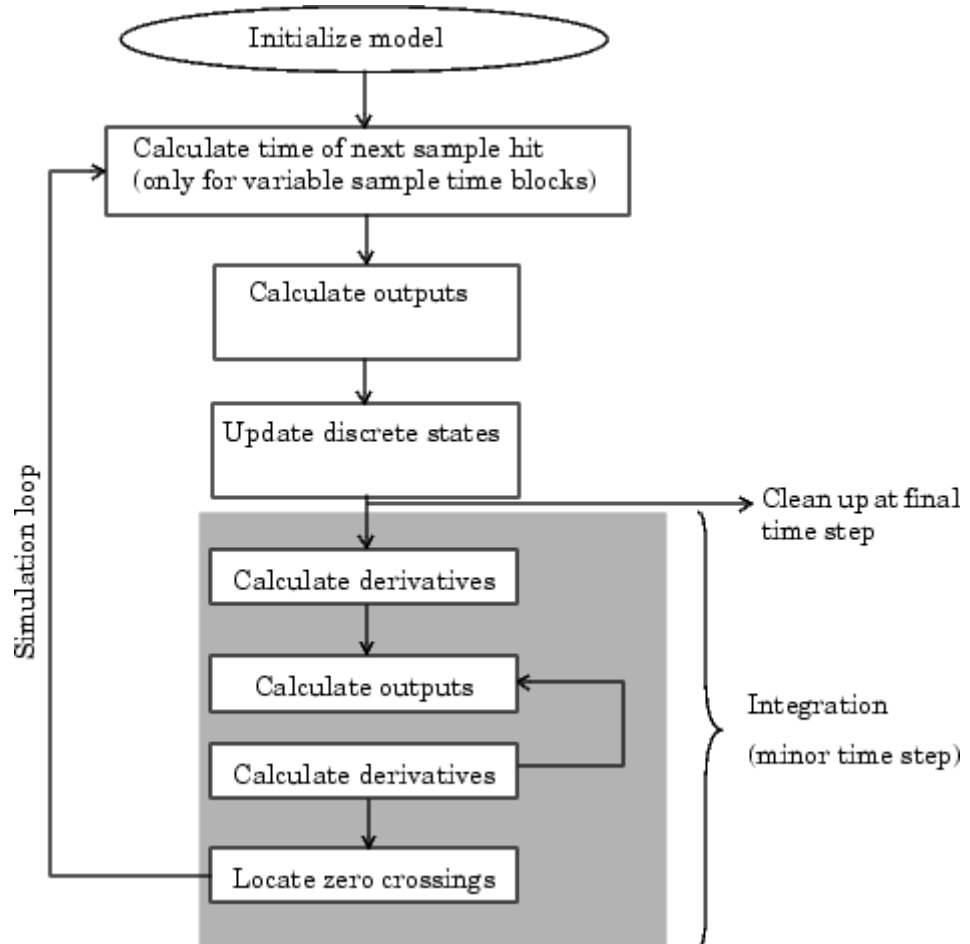
$$\text{where } x = x_c + x_d$$

Simulation Stages

Execution of a Simulink model proceeds in stages. First comes the initialization phase. In this phase, Simulink incorporates library blocks into the model, propagates widths, data types, and sample times, evaluates block parameters, determines block execution order, and allocates memory. Then Simulink enters a *simulation loop*, where each pass through the loop is referred to as a *simulation step*. During each simulation step, Simulink executes each of the model's blocks in the order determined during

initialization. For each block, Simulink invokes functions that compute the block's states, derivatives, and outputs for the current sample time. This continues until the simulation is complete.

The following figure illustrates the stages of a simulation.



How Simulink Performs Simulation

S-Function Callback Methods

An S-function comprises a set of *S-function callback methods* that perform tasks required at each simulation stage. During simulation of a model, at each simulation stage, Simulink calls the appropriate methods for each S-Function block in the model. Tasks performed by S-function methods include

- Initialization — Prior to the first simulation loop, Simulink initializes the S-function. During this stage, Simulink
 - Initializes the `SimStruct`, a simulation structure that contains information about the S-function
 - Sets the number and dimensions of input and output ports
 - Sets the block sample times
 - Allocates storage areas and the `sizes` array
- Calculation of next sample hit — If you've created a variable sample time block, this stage calculates the time of the next sample hit; that is, it calculates the next step size.
- Calculation of outputs in the major time step — After this call is complete, all the output ports of the blocks are valid for the current time step.
- Update of discrete states in the major time step — In this call, all blocks should perform once-per-time-step activities such as updating discrete states for next time around the simulation loop.
- Integration — This applies to models with continuous states and/or nonsampled zero crossings. If your S-function has continuous states, Simulink calls the output and derivative portions of your S-function at minor time steps. This is so Simulink can compute the states for your S-function. If your S-function (C MEX only) has nonsampled zero crossings, Simulink calls the output and zero-crossings portions of your S-function at minor time steps so that it can locate the zero crossings.

Note See “How Simulink Works” in the Using Simulink documentation for an explanation of major and minor time steps.

Implementing S-Functions

You can implement an S-function as either an M-file or a Mex-file. The following sections describe these alternative implementations and discuss the advantages of each.

M-File S-Functions

An M-file S-function consists of a MATLAB function of the following form:

```
[sys,x0,str,ts]=f(t,x,u,flag,p1,p2,...)
```

where *f* is the S-function's name, *t* is the current time, *x* is the state vector of the corresponding S-function block, *u* is the block's inputs, *flag* indicates a task to be performed, and *p1*, *p2*, ... are the block's parameters. During simulation of a model, Simulink repeatedly invokes *f*, using *flag* to indicate the task to be performed for a particular invocation. Each time the S-function performs the task, it returns the result in a structure having the format shown in the syntax example.

A template implementation of an M-file S-function, *sfuntmpl.m*, resides in *matlabroot/toolbox/simulink/blocks*. The template consists of a top-level function and a set of skeleton subfunctions, each of which corresponds to a particular value of *flag*. The top-level function invokes the subfunction indicated by *flag*. The subfunctions, called S-function callback methods, perform the tasks required of the S-function during simulation. The following table lists the contents of an M-file S-function that follows this standard format.

Simulation Stage	S-Function Routine	Flag
Initialization	<i>mdlInitializeSizes</i>	<i>flag</i> = 0
Calculation of next sample hit (variable sample time block only)	<i>mdlGetTimeOfNextVarHit</i>	<i>flag</i> = 4
Calculation of outputs	<i>mdlOutputs</i>	<i>flag</i> = 3
Update of discrete states	<i>mdlUpdate</i>	<i>flag</i> = 2

Simulation Stage	S-Function Routine	Flag
Calculation of derivatives	mdlDerivatives	flag = 1
End of simulation tasks	mdlTerminate	flag = 9

We recommend that you follow the structure and naming conventions of the template when creating M-file S-functions. This makes it easier for others to understand and maintain M-file S-functions that you create. See Chapter 2, “Writing S-Functions in M” for information on creating M-file S-functions.

MEX-File S-Functions

Like an M-file S-function, a MEX-file function consists of a set of callback routines that Simulink invokes to perform various block-related tasks during a simulation. Significant differences exist, however. For one, MEX-file functions are implemented in a different programming language: C, C++, Ada, or Fortran. Also, Simulink invokes MEX S-function routines directly instead of via a flag value as with M-file S-functions. Because Simulink invokes the functions directly, MEX-file functions must follow standard naming conventions specified by Simulink.

Other key differences exist. For one, the set of callback functions that MEX functions can implement is much larger than can be implemented by M-file functions. A MEX function also has direct access to the internal data structure, called the SimStruct, that Simulink uses to maintain information about the S-function. MEX-file functions can also use the MATLAB MEX-file API to access the MATLAB workspace directly.

A C MEX-file S-function template, called `sfuntmpl_basic.c`, resides in the `matlabroot/simulink/src` directory. The template contains skeleton implementations of all the required and optional callback routines that a C MEX-file S-function can implement. For a more amply commented version of the template, see `sfuntmpl_doc.c` in the same directory.

MEX-File Versus M-File S-Functions

M-file and MEX-file S-functions each have advantages. The advantage of M-file S-functions is speed of development. Developing M-file S-functions avoids the time-consuming compile-link-execute cycle required by

development in a compiled language. M-file S-functions also have easier access to MATLAB and toolbox functions.

The primary advantage of MEX-file functions is versatility. The larger number of callbacks and access to the SimStruct enable MEX-file functions to implement functionality not accessible to M-file S-functions. Such functionality includes the ability to handle data types other than double, complex inputs, matrix inputs, and so on.

S-Function Concepts

Understanding these key concepts should enable you to build S-functions correctly:

- Direct feedthrough
- Dynamically sized inputs
- Setting sample times and offsets

Direct Feedthrough

Direct feedthrough means that the output (or the variable sample time for variable sample time blocks) is controlled directly by the value of an input port. A good rule of thumb is that an S-function input port has direct feedthrough if

- The output function (`mdlOutputs` or `flag==3`) is a function of the input u . That is, there is direct feedthrough if the input u is accessed in `mdlOutputs`. Outputs can also include graphical outputs, as in the case of an XY Graph scope.
- The “time of next hit” function (`mdlGetTimeOfNextVarHit` or `flag==4`) of a variable sample time S-function accesses the input u .

An example of a system that requires its inputs (i.e., has direct feedthrough) is the operation $y = k \times u$, where u is the input, k is the gain, and y is the output.

An example of a system that does not require its inputs (i.e., does not have direct feedthrough) is this simple integration algorithm

Outputs: $y = x$

Derivative: $\dot{x} = u$

where x is the state, \dot{x} is the state derivative with respect to time, u is the input, and y is the output. Note that x is the variable that Simulink integrates. It is very important to set the direct feedthrough flag correctly because it affects the execution order of the blocks in your model and is used to detect algebraic loops.

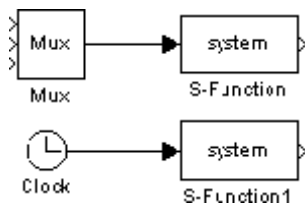
Dynamically Sized Arrays

S-functions can be written to support arbitrary input dimensions. In this case, the actual input dimensions are determined dynamically when a simulation is started by evaluating the dimensions of the input vector driving the S-function. The input dimensions can also be used to determine the number of continuous states, the number of discrete states, and the number of outputs.

M-file S-functions can have only one input port and that input port can accept only one-dimensional (vector) signals. However, the signals can be of varying widths. Within an M-file S-function, to indicate that the input width is dynamically sized, specify a value of -1 for the appropriate fields in the `sizes` structure, which is returned during the `mdlInitializeSizes` call. You can determine the actual input width when your S-function is called by using `length(u)`. If you specify a width of 0, the input port is removed from the S-function block.

A C S-function can have multiple I/O ports and the ports can have different dimensions. The number of dimensions and the size of each dimension can be determined dynamically.

For example, the following illustration shows two instances of the same S-Function block in a model.



The upper S-Function block is driven by a block with a three-element output vector. The lower S-Function block is driven by a block with a scalar output. By specifying that the S-Function block has dynamically sized inputs, the same S-function can accommodate both situations. Simulink automatically calls the block with the appropriately sized input vector. Similarly, if other block characteristics, such as the number of outputs or the number of discrete or continuous states, are specified as dynamically sized, Simulink defines these vectors to be the same length as the input vector.

C S-functions give you more flexibility in specifying the widths of input and output ports. See “Creating Input and Output Ports” on page 7-10.

Setting Sample Times and Offsets

Both M-file and C MEX S-functions allow a high degree of flexibility in specifying when an S-function executes. Simulink provides the following options for sample times:

- Continuous sample time — For S-functions that have continuous states and/or nonsampled zero crossings (see “How Simulink Works” in Using Simulink documentation for explanation of zero crossings). For this type of S-function, the output changes in minor time steps.
- Continuous but fixed in minor time step sample time — For S-functions that need to execute at every major simulation step, but do not change value during minor time steps.
- Discrete sample time — If your S-Function block’s behavior is a function of discrete time intervals, you can define a sample time to control when Simulink calls the block. You can also define an offset that delays each sample time hit. The value of the offset cannot exceed the corresponding sample time.

A *sample time hit* occurs at time values determined by the formula

$$\text{TimeHit} = (n * \text{period}) + \text{offset}$$

where n , an integer, is the current simulation step. The first value of n is always zero.

If you define a discrete sample time, Simulink calls the S-function `mdlOutput` and `mdlUpdate` routines at each sample time hit (as defined in the above equation).

- Variable sample time — A discrete sample time where the intervals between sample hits can vary. At the start of each simulation step, S-functions with variable sample times are queried for the time of the next hit.
- Inherited sample time — Sometimes an S-Function block has no inherent sample time characteristics (that is, it is either continuous or discrete, depending on the sample time of some other block in the system). You can

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 `INHERITED_SAMPLE_TIME` in C S-functions as the sample time. For more information on the propagation of sample times, see "Displaying 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 that 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

```
[INHERITED_SAMPLE_TIME, 0.0]
```

or

```
[INHERITED_SAMPLE_TIME, FIXED_IN_MINOR_STEP_OFFSET]
```

where

```
INHERITED_SAMPLE_TIME = -1.0
```

The following guidelines might 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 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 `mdlGetTimeOfNextVarHit` 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, 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 of its driving block, either continuous or discrete, but should never run in minor steps. If it did, the scope display would show the intermediate computations of the solver rather than the final result at each time point.

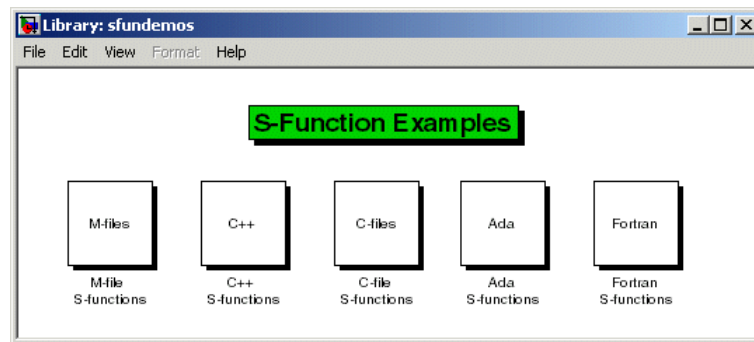
S-Function Examples

Simulink comes with a library of S-function examples.

To run an example:

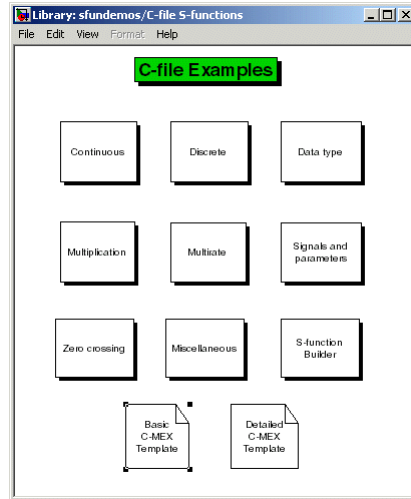
- 1 Enter `sfundemos` at the MATLAB command line.

MATLAB displays the S-function demo library.



Each block represents a category of S-function examples.

- 2 Click a category to display the examples that it includes.



3 Click a block to open and run the example that it represents.

It might be helpful to examine some sample S-functions as you read the next chapters. Code for the examples is stored in these subdirectories under the MATLAB root directory:

M-files	toolbox/simulink/blocks
C, C++, and Fortran	simulink/src
Ada	simulink/ada/examples

M-File S-Function Examples

The simulink/blocks directory contains many M-file S-functions. Consider starting off by looking at these files.

Filename	Description
csfunc.m	Define a continuous system in state-space format.
dsfunc.m	Define a discrete system in state-space format.

Filename	Description
vsfunc.m	Illustrates how to create a variable sample time block. This block implements a variable step delay in which the first input is delayed by an amount of time determined by the second input.
mixed.m	Implement a hybrid system consisting of a continuous integrator in series with a unit delay.
vdpm.m	Implement the Van der Pol equation (similar to the demo model, vdp).
simom.m	<p>Example state-space M-file S-function with internal A, B, C, and D matrices. This S-function implements</p> $\frac{dx}{dt} = Ax + Bu$ $y = Cx + Du$ <p>where x is the state vector, u is the input vector, and y is the output vector. The A, B, C, and D matrices are embedded in the M-file.</p>
simom2.m	Example state-space M-file S-function with external A, B, C, and D matrices. The state-space structure is the same as in simom.m, but the A, B, C, and D matrices are provided externally as parameters to this file.
limintm.m	Implement a continuous limited integrator where the output is bounded by lower and upper bounds and includes initial conditions.

Filename	Description
sfun_varargm.m	Example M-file S-function showing how to use the MATLAB vararg facility.
vlimintm.m	Example of a continuous limited integrator S-function. This illustrates how to use the size entry of -1 to build an S-function that can accommodate a dynamic input/state width.
vdlimintm.m	Example of a discrete limited integrator S-function. This example is identical to vlimint.m, except that the limited integrator is discrete.

C S-Function Examples

The `simulink/src` directory also contains examples of C MEX S-functions, many of which have an M-file S-function counterpart. These C MEX S-functions are listed in this table.

Filename	Description
barplot.c	Access Simulink signals without using the standard block inputs.
csfunc.c	Example C MEX S-function for defining a continuous system.
dlimint.c	Implement a discrete-time limited integrator.
dsfunc.c	Example C MEX S-function for defining a discrete system.
fcncallgen.c	Execute function-call subsystems n times at the designated rate (sample time).
limintc.c	Implement a limited integrator.

Filename	Description
<code>mixedm.c</code>	Implement a hybrid dynamic system consisting of a continuous integrator (1/s) in series with a unit delay (1/z).
<code>mixedmex.c</code>	Implement a hybrid dynamic system with a single output and two inputs.
<code>quantize.c</code>	Example MEX-file for a vectorized quantizer block. Quantizes the input into steps as specified by the quantization interval parameter, q.
<code>sdotproduct</code>	Compute dot product (multiply-accumulate) of two real or complex vectors.
<code>sftable2.c</code>	Two-dimensional table lookup in S-function form.
<code>sfun_atol.c</code>	Set different absolute tolerances for each continuous state.
<code>sfun_bitop.c</code>	Perform the bitwise operations AND, OR, XOR, left shift, right shift, and one's complement on uint8, uint16, and uint32 inputs.
<code>sfun_cplx.c</code>	Complex signal add with one input port and one parameter.
<code>sfun_directlook.c</code>	Direct 1-D lookup.
<code>sfun_dtype_io.c</code>	Example of the use of Simulink data types for inputs and outputs.
<code>sfun_dtype_param.c</code>	Example of the use of Simulink data types for parameters.
<code>sfun_dynsize.c</code>	Simple example of how to size outputs of an S-function dynamically.
<code>sfun_errhdl.c</code>	Simple example of how to check parameters using the mdlCheckParams S-function routine.

Filename	Description
sfun_fcncall.c	Example of an S-function that is configured to execute function-call subsystems on the first and third output elements.
sfun_frmad.c	Frame-based A/D converter.
sfun_frmda.c	Frame-based D/A converter.
sfun_frmdft.c	Multichannel frame-based Discrete-Fourier transformation (and its inverse).
sfun_frmunbuff.c	Frame-based unbuffer block.
sfun_multiport.c	S-function that has multiple input and output ports.
sfun_manswitch.c	Manual switch.
sfun_matadd.c	Matrix add with one input port, one output port, and one parameter.
sfun_multirate.c	Demonstrate how to specify port-based sample times.
sfun_psbbreaker.c	Implement the logic for the breaker block in the Power System Blockset.
sfun_psbcontc.c	Continuous implementation of state-space system.
sfun_psbdiscc.c	Discrete implementation of state-space system.
sfun_runtime1.c	Run-time parameter example.
sfun_runtime2.c	Run-time parameter example.
sfun_zc.c	Demonstrate use of nonsampled zero crossings to implement $\text{abs}(u)$. This S-function is designed to be used with a variable-step solver.
sfun_zc_sat.c	Saturation example that uses zero crossings.

Filename	Description
sfunmem.c	A one-integration-step delay and hold memory function.
simomex.c	<p>Implements a single-output, two-input state-space dynamic system described by these state-space equations</p> $\begin{aligned} dx/dt &= Ax + Bu \\ y &= Cx + Du \end{aligned}$ <p>where x is the state vector, u is vector of inputs, and y is the vector of outputs.</p>
stspace.c	Implement a set of state-space equations. You can turn this into a new block by using the S-Function block and mask facility. This example MEX-file performs the same function as the built-in State-Space block. This is an example of a MEX-file where the number of inputs, outputs, and states is dependent on the parameters passed in from the workspace. Use this as a template for other MEX-file systems.
stvctf.c	Implement a continuous-time transfer function whose transfer function polynomials are passed in via the input vector. This is useful for continuous time adaptive control applications.

Filename	Description
stvdct.f	Implement a discrete-time transfer function whose transfer function polynomials are passed in via the input vector. This is useful for discrete-time adaptive control applications.
stvmgain.c	Time-varying matrix gain.
table3.c	3-D lookup table.
timestwo.c	Basic C MEX S-function that doubles its input.
vdlmint.c	Implement a discrete-time vectorized limited integrator.
vdpmex.c	Implement the Van der Pol equation.
vlimint.c	Implement a vectorized limited integrator.
vsfunc.c	Illustrate how to create a variable sample time 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.

Fortran S-Function Examples

The following table lists sample Fortran S-functions.

Filename	Description
sfun_timestwo_for.for	Sample Level 1 Fortran representation of a C timestwo S-function.
sfun_atmos.c	Calculation of the 1976 standard atmosphere to 86 km.
vdpmexf.for	Van der Pol system.

C++ S-Function Examples

The following table lists sample C++ S-functions.

Filename	Description
<code>sfun_counter_cpp.cpp</code>	Store a C++ object in the pointers vector PWork.

Ada S-Function Examples

The `simulink/ada/examples` directory contains the following examples of S-functions implemented in Ada.

Directory Name	Description
<code>matrix_gain</code>	Implement a Matrix Gain block.
<code>multi_port</code>	Multiport block.
<code>simple_lookup</code>	Lookup table. Illustrates use of a wrapper S-function that wraps stand-alone Ada code (i.e., Ada packages and procedures) both for use with Simulink as an S-function and directly with Ada code generated using the Real-Time Workshop Ada Coder.
<code>times_two</code>	Output twice its input.

Writing S-Functions in M

The following sections explain how to use the MATLAB M programming language to create S-functions.

Introduction (p. 2-2)

Introduction to writing S-functions in the MATLAB M language.

Writing Level-2 M-File S-Functions (p. 2-3)

Explains how to create M-file S-functions based on the current Level-2 M-file S-function application programming interface (API).

Maintaining Level-1 M-File S-Functions (p. 2-8)

Provides information on maintaining Level-1 M-file S-functions developed for earlier versions of Simulink.

Introduction

Simulink provides an application programming interface (API) that lets you create custom blocks whose properties and behavior are defined by M-file programs called M-file S-functions. The Level-2 M-file S-function API allows you to create blocks that have all of the features and capabilities of Simulink built-in blocks, including multiple input and output ports, the ability to accept vector or matrix signals of any data type supported by Simulink, real or complex signals, signal frames, and the ability to operate at multiple sample rates. For information on how to use the API to create custom blocks, see “Writing Level-2 M-File S-Functions” on page 2-3.

Note This version of Simulink also supports a predecessor API, called Level 1, for writing M-file S-functions. This is done to ensure that Simulink can simulate models that use M-file S-function blocks developed for use with earlier Simulink releases (see “Maintaining Level-1 M-File S-Functions” on page 2-8). You should not use the Level-1 API to develop new M-file S-functions. Instead, you should use the Level-2 API.

Writing Level-2 M-File S-Functions

The Level-2 M-file S-function application programming interface (API) allows you to use the MATLAB M language to create full-fledged custom blocks having multiple inputs and outputs and capable of handling any type of signal produced by a Simulink model, including matrix and frame signals of any data type. The Level-2 M-File S-Function API corresponds closely to the API for creating C MEX-file S-functions. Much of the documentation for creating C MEX-file S-functions (see Chapter 3, “Writing S-Functions in C” and Chapter 7, “Implementing Block Features”) applies also to Level-2 M-file S-functions. To avoid duplication, this section focuses on providing information that is specific to writing Level-2 M-file S-functions.

About Level-2 M-File S-Functions

First, a word about Level-2 M-File S-functions themselves. A Level-2 M-file S-function is an M-file that defines the properties and behavior of an instance of a Level-2 M-File S-Function block that references the M-file in a Simulink model. The M-file itself comprises a set of callback methods (see “Callback Methods” on page 2-5) that Simulink invokes when updating or simulating the model. The callback methods perform the actual work of initializing and computing the outputs of the block defined by the S-function.

To facilitate these tasks, Simulink passes a run-time object to the callback methods as an argument. The run-time object effectively serves as an M proxy for the S-function block, allowing the callback method to set and access the block’s properties during simulation or model updating (see “Run-time Object” on page 2-7 for more information).

Level-2 M-File S-Function API

The Level-2 M-File S-function API defines the signatures and general purpose of the callback methods that constitute a Level-2 M-file S-function. The S-function itself provides the implementations of these callback methods. The implementations in turn determine the block’s attributes (e.g., ports, parameters, and states) and behavior (e.g., the block’s outputs as a function of time and the block’s inputs, states, and parameters). By creating an S-function with an appropriate set of callback implementations, you can define a block type that meets the specific requirements of your application.

M-File S-Function Demos

Simulink provides a set of self-documenting demo models that illustrate creation and usage of Level-2 M-file S-functions. To view the demos”

- 1 Select **Help->Demos** from the MATLAB desktop or the Simulink editor menu bar.

The MATLAB Help Browser appears with the **Demos** pane selected.

- 2 Select **Simulink->Block Diagramming Features->Custom Code and Hand Coded Blocks** from the **Demos** pane.

The “Custom Code and Hand Coded Blocks” page appears in the help browser.

- 3 Click the “Open this model” hyperlink at the end of the page.

The **S-Functions Examples** window appears.

- 4 Double-click the M-File S-Functions block in the window.

The M-file S-functions window appears.

- 5 Double-click the Level-2 M-files block.

The Level-2 M-files window appears. Each block in the window corresponds to a demo.

- 6 Double-click any of the blocks to display the corresponding demo model.

S-Function Template

To give you a head start on creating Level-2 M-file S-functions, Simulink provides an annotated M-file template containing skeleton implementation of the callbacks defined by the Level-2 M-File S-function API. The template resides at

```
<matlabroot>/toolbox/simulink/blocks/msfuntmpl.m
```

To create an M-file S-function, make a copy of the template and edit the copy as necessary to reflect the desired behavior of the S-function you are creating. The comments in the template explain how to do this.

Instantiating a Level-2 M-File S-Function

To create an instance of the S-function in a model, first create an instance of the Level-2 M-File S-Function block in the model. Then open the block's parameter dialog box and enter the name of the M-file that implements your S-function in the dialog box's **M-file name** field. If your function uses any additional parameters, enter their values as a comma-separated list in the dialog box's **Parameters** field.

Generating Code from a Level-2 M-File S-Function

Generating code from a model containing a Level-2 M-file S-function requires that you provide a corresponding TLC file. You do not need a TLC file to run a model in accelerated mode as the Simulink Accelerator runs Level-2 M-file S-functions in interpreted mode.

Callback Methods

The Level-2 M-file S-function API specifies a set of callback methods that an M-file S-function must implement and others that it may choose to omit, depending on the requirements of the block that the S-function defines. The methods defined by the Level-2 M-file S-function API generally correspond to that of similarly named methods defined by the C MEX-file S-function API. For information on what each method does, see “How Simulink Works” in “Using Simulink” and Chapter 8, “S-Function Callback Methods — Alphabetical List”.

The following table lists the Level-2 M-file S-function callback methods and their C MEX-file counterparts.

Level-2 M-File Method	C MEX-File Method
setup method (see “Setup Method” on page 2-6)	mdlInitializeSizes
CheckParameters	mdlCheckParameters
Derivatives	mdlDerivatives
Disable	mdlDisable
Enable	mdlEnable
InitializeCondition	mdlInitializeConditions

Level-2 M-File Method	C MEX-File Method
Outputs	mdlOutputs
ProcessParameters	mdlProcessParameters
SetInputPortComplexSignal	mdlSetInputPortComplexSignal
SetInputPortDataType	mdlSetInputPortDataType
SetInputPortDimensions	mdlSetInputPortDimensionInfo
SetInputPortSampleTime	mdlSetInputPortSampleTime
SetInputPortSamplingMode	mdlSetInputPortFrameData
SetOutputPortComplexSignal	mdlSetOutputPortComplexSignal
SetOutputPortDataType	mdlSetOutputPortDataType
SetOutputPortDimensions	mdlSetOutputPortDimensionInfo
SetOutputPortSampleTime	mdlSetOutputPortSampleTime
Start	mdlStart
Update	mdlUpdate
WriteRTW	mdlRTW
ZeroCrossings	mdlZeroCrossings

Setup Method

The body of the setup method of a Level-2 M-file S-function initializes instances of the corresponding Level-2 M-File S-Function block in a model. In this respect, the main function is similar to the `mdlInitializeSizes` callback method implemented by C MEX S-functions. Setup tasks that the main function performs include:

- Setting up the number of input and output ports of the block.
- Setting attributes such as dimensions, data types, complexity, and sample times for these ports.
- Setting up the number of parameters and checking for the validity of these parameters.

- Registering the various block methods using the handles for other local functions in the M-file, using the `RegBlockMethod` method of the S-function block's run-time object passed to it. See the documentation for `Simulink.MSFcnRunTimeBlock` for information on using this method.

Run-time Object

When Simulink invokes a Level-2 M-file S-function callback method, it passes an instance of `Simulink.MSFcnRunTimeBlock` class to the method as an argument. This instance, known as the S-function block's run-time object, serves the same purpose for Level-2 M-file S-function callback methods as the `SimStruct` structure serves for C MEX-file S-function callback methods. It enables the method to provide and obtain information about various elements of the block: ports, parameters, states, and work vectors. The method does this by getting or setting properties or invoking methods of the block run-time object. See the documentation for `Simulink.MSFcnRunTimeBlock` class for information on getting and setting the run-time object's properties and invoking its methods.

Note Other M-file programs besides M-file S-functions can use run-time objects to obtain information about an M-file S-function or other blocks in a model that is simulating. See “Accessing Block Data During Simulation” in “Using Simulink” for more information.

Maintaining Level-1 M-File S-Functions

Note The information provided in this section is intended only for use in maintaining existing M-file S-functions based on Level-1 API. You should use the more capable Level-2 API to develop new M-file S-functions (see “Writing Level-2 M-File S-Functions” on page 2-3).

A Level-1 M-file S-function consists of a MATLAB function of the following form

$$[\text{sys}, \text{x0}, \text{str}, \text{ts}] = \text{f}(\text{t}, \text{x}, \text{u}, \text{flag}, \text{p1}, \text{p2}, \dots)$$

where *f* is the name of the S-function. During simulation of a model, Simulink repeatedly invokes *f*, using the *flag* argument to indicate the task (or tasks) to be performed for a particular invocation. Each time the S-function performs the task and returns the results in an output vector.

A template implementation of an M-file S-function, *sfuntmpl.m*, resides in *matlabroot/toolbox/simulink/blocks*. The template consists of a top-level function and a set of skeleton subfunctions, called S-function callback methods, each of which corresponds to a particular value of *flag*. The top-level function invokes the subfunction indicated by *flag*. The subfunctions perform the actual tasks required of the S-function during simulation.

S-Function Arguments

Simulink passes the following arguments to an S-function:

t	Current time
x	State vector
u	Input vector
flag	Integer value that indicates the task to be performed by the S-function

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

Flag Argument

Flag	S-Function Routine	Description
0	<code>mdlInitializeSizes</code>	Defines basic S-Function block characteristics, including sample times, initial conditions of continuous and discrete states, and the sizes array.
1	<code>mdlDerivatives</code>	Calculates the derivatives of the continuous state variables.
2	<code>mdlUpdate</code>	Updates discrete states, sample times, and major time step requirements.
3	<code>mdlOutputs</code>	Calculates the outputs of the S-function.
4	<code>mdlGetTimeOfNextVarHit</code>	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 <code>mdlInitializeSizes</code> .
9	<code>mdlTerminate</code>	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 (see “Specifying Sample Time” in the “Using Simulink” for information on how to specify a block’s sample time and offset).

For example, if you want your S-function to run at every time step (continuous sample time), set `ts` to `[0 0]`. If you want your S-function to run at the same rate as the block to which it is connected (inherited sample time), set `ts` to `[-1 0]`. If you want it to run every 0.25 seconds (discrete sample time) starting at 0.1 seconds after the simulation start time, set `ts` to `[0.25 0.1]`.

You can create S-functions that do multiple tasks, each at a different sample rate (i.e., a multirate S-function). In this case, `ts` should specify all the sample rates used by your S-function in ascending order by sample time. For example, suppose your S-function performs one task every 0.25 second starting from the simulation start time and another task every 1 second starting 0.1 second after the simulation start time. In this case, your S-function should set `ts` equal to `[.25 0; 1.0 .1]`. This will cause Simulink to execute the S-function at the following times: `[0 0.1 0.25 0.5 0.75 1 1.1 ...]`. Your S-function must decide at every sample time which task to perform at that sample time.

You can also create an S-function that performs some tasks continuously (i.e., at every time step) and others at discrete intervals.

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 `simsizes` function at the beginning of `mdlInitializeSizes`.

```
sizes = simsizes;
```

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

Fields in the sizes Structure

Field Name	Description
sizes.NumContStates	Number of continuous states
sizes.NumDiscStates	Number of discrete states
sizes.NumOutputs	Number of outputs
sizes.NumInputs	Number of inputs
sizes.DirFeedthrough	Flag for direct feedthrough
sizes.NumSampleTimes	Number of sample times

After you initialize the sizes structure, call `simsizes` again:

```
sys = simsizes(sizes);
```

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

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 1-4). 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

The following sections explain how to use the C programming language to create S-functions.

Introduction (p. 3-2)	Overview of writing a C S-function.
Building S-Functions Automatically (p. 3-5)	How to use the S-Function Builder to generate S-functions automatically from specifications that you supply.
S-Function Builder Dialog Box (p. 3-9)	Describes the S-Function Builder dialog box
Example of a Basic C MEX S-Function (p. 3-29)	Illustrates the code needed to create a C S-function.
Templates for C S-Functions (p. 3-35)	Describes code templates that you can use as starting points for writing your own C S-functions.
How Simulink Interacts with C S-Functions (p. 3-39)	Describes how Simulink interacts with a C S-function. This is information that you need to know in order to create and debug your own C S-functions.
Writing Callback Methods (p. 3-47)	How to write methods that Simulink calls as it executes your S-function.
Converting Level 1 C MEX S-Functions to Level 2 (p. 3-48)	How to convert S-functions written for earlier releases of Simulink to work with the current version.

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 C 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 8, “S-Function Callback Methods — Alphabetical List” 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

#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.

Creating C MEX S-Functions

The easiest way to create a C MEX S-function is to use the S-Function Builder (see “Building S-Functions Automatically” on page 3-5). This tool builds a C MEX S-function from specifications and code fragments that you supply. This eliminates the need for you to build the S-function from scratch.

The following sections provide information on writing C MEX S-functions from scratch:

- “Example of a Basic C MEX S-Function” on page 3-29 provides a step-by-step example of how to write a simple S-function from scratch.
- “Templates for C S-Functions” on page 3-35 describes a complete skeleton implementation of a C S-function that you can use as a starting point for creating your own S-functions.

Building S-Functions Automatically

The S-Function Builder is a Simulink block that builds an S-function from specifications and C code that you supply. The S-Function Builder also serves as a wrapper for the generated S-function in models that use the S-function. This section explains how to use the S-Function Builder to build simple C MEX S-functions.

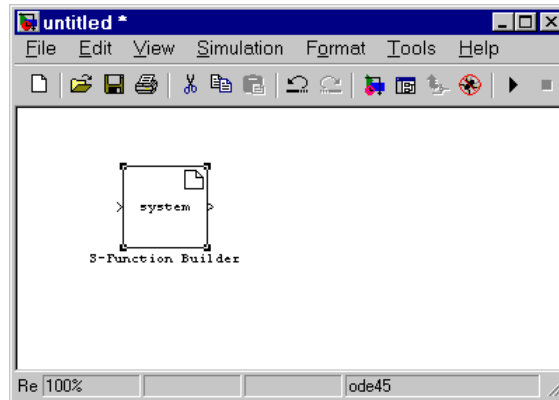
Note For examples of using the S-Function Builder to build S-functions, see the “C S-functions” section of the S-function demos provided with Simulink. To display the demos, enter `sfundemo` at the MATLAB command line (see “S-Function Examples” on page 1-18 for more information).

To build an S-function with the S-Function Builder:

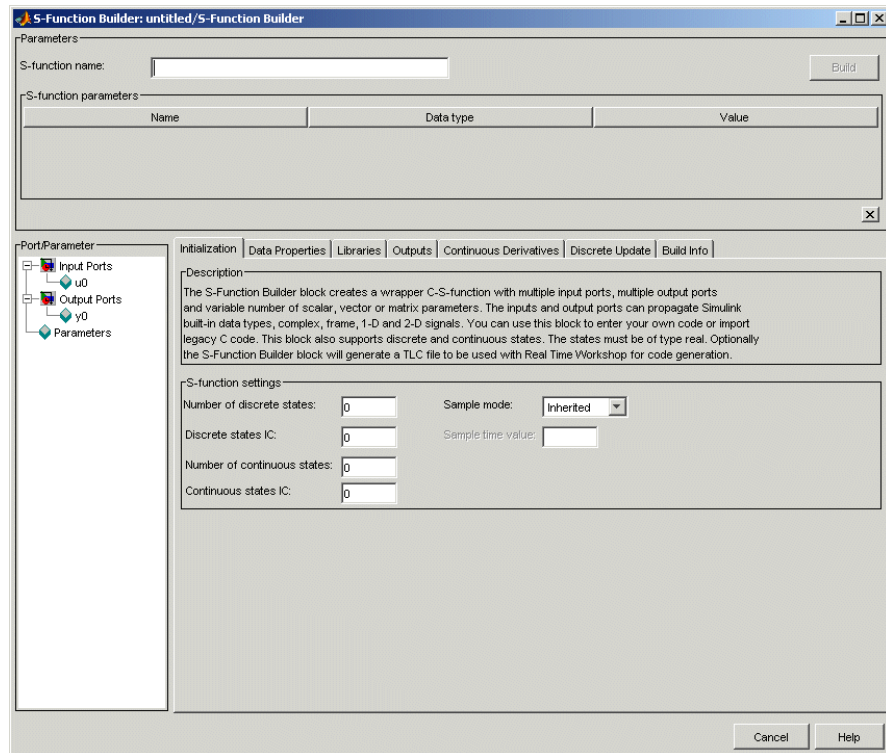
- 1 Set the MATLAB current directory to the directory in which you want to create the S-function.

Note This directory must be on the MATLAB path.

- 2 Create a new Simulink model.
- 3 Copy an instance of the S-Function Builder block from the Simulink User-Defined Functions library into the new model.



- 4 Double-click the block to open the S-Function Builder dialog box (see “S-Function Builder Dialog Box” on page 3-9).



- 5 Use the specification and code entry panes on the S-Function Builder dialog box to enter information and custom source code required to tailor the generated S-function to your application (see “S-Function Builder Dialog Box” on page 3-9).
- 6 If you have not already done so, configure the MATLAB mex command to work on your system.

To configure the mex command, type `mex -setup` at the MATLAB command line.

- 7 Click **Build** on the dialog box to start the build process.

Simulink builds a Mex-file that implements the specified S-function and saves the file in the current directory (see “How the S-Function Builder Builds an S-Function” on page 3-7).

- 8 Save the model containing the S-Function Builder block.

Deploying the Generated S-Function

To use the generated S-function in another model, first check to ensure that the directory containing the generated S-function is on the MATLAB path. Then copy the S-Function Builder block from the model used to create the S-function into the target model and set its parameters, if necessary, to the values required by the target model.

How the S-Function Builder Builds an S-Function

The S-Function Builder builds an S-function as follows. First, it generates the following source files in the current directory:

- `sfun.c`

where `sfun` is the name of the S-function that you specified in the **S-function name** field of the S-Function Builder’s dialog box. This file contains the C source code representation of the standard portions of the generated S-function.

- `sfun_wrapper.c`

This file contains the custom code that you entered in the S-Function Builder dialog box.

- `sfun.tlc`

This file permits Simulink to run the generated S-function in accelerated mode and Real-Time Workshop to include this S-function in the code it generates.

After generating the S-function source code, the S-Function Builder uses the MATLAB `mex` command to build the Mex-file representation of the S-function from the generated source code and any external custom source code and libraries that you specified.

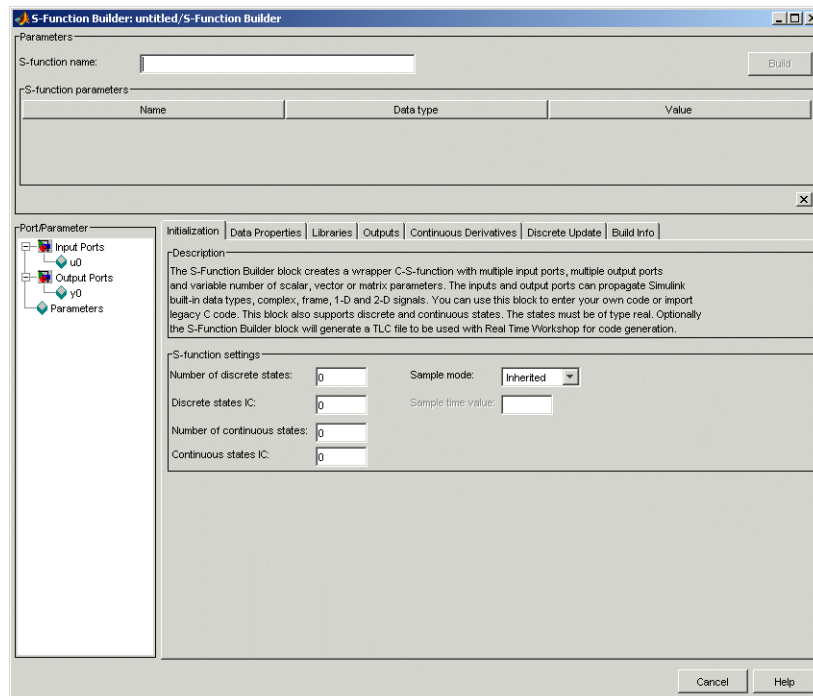
Setting the Include Path

The S-Function Builder searches for custom header files in the directories specified by the MATLAB application data named `SfunctionBuilderIncludePath`. This data is associated with the model in which you create the S-Function Builder block. If your S-function uses custom header files and the custom header files do not reside in the current directory (i.e., the directory containing the generated S-function), you must update `SfunctionBuilderIncludePath` to specify the locations of the directories containing the header files. `SfunctionBuilderIncludePath` is a three-element cell array that allows you to specify as many as three include directories. For example, the following MATLAB commands set `SfunctionBuilderIncludePath` to the paths of two include directories.

```
incPath = getappdata(0,'SfunctionBuilderIncludePath');
incPath{1} = '/home/jones/include';
incPath{2} = getenv('PROJECT_INCLUDE_DIR')
setappdata(0,'SfunctionBuilderIncludePath',incPath)
```

S-Function Builder Dialog Box

The S-Function Builder dialog box enables you to specify the attributes of an S-function to be built by an S-Function Builder block. To display the dialog box, click twice on the block's icon or select the block and then select **Open Block** from the model editor's **Edit** menu or the block's context menu. The dialog box appears.



The dialog box contains controls that let you enter information needed for the S-Function Builder block to build an S-function to your specifications. The controls are grouped into panes. See the following sections for information on the panes and the controls that they contain.

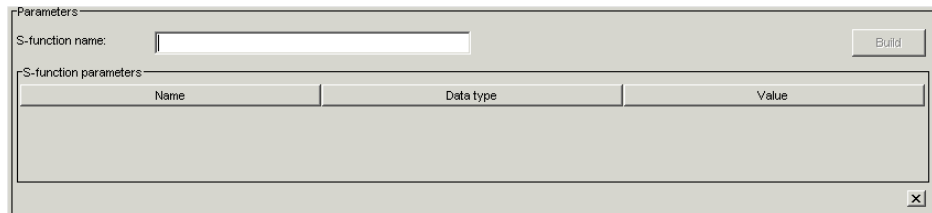
- “Parameters/S-Function Name Pane” on page 3-10
- “Port/Parameter Pane” on page 3-11
- “Initialization Pane” on page 3-11

- “Data Properties Pane” on page 3-13
- “Libraries Pane” on page 3-18
- “Outputs Pane” on page 3-20
- “Continuous Derivatives Pane” on page 3-24
- “Discrete Update Pane” on page 3-25
- “Build Info Pane” on page 3-27

Note The following sections use the term *target S-function* to refer to the S-function specified by the S-Function Builder dialog box.

Parameters/S-Function Name Pane

This pane displays the target S-function’s name and parameters.



The screenshot shows a dialog box titled "Parameters". It contains an "S-function name:" label followed by a text input field and a "Build" button. Below this is a section labeled "S-function parameters" which contains a table with three columns: "Name", "Data type", and "Value". The table is currently empty. There is a close button (X) in the bottom right corner of the dialog box.

The pane contains the following controls.

S-function name

Specifies the name of the target S-function.

S-function parameters

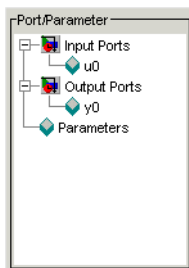
This table displays the parameters of the target S-function. Each row of the table corresponds to a parameter, and each column displays a property of the parameter as follows.

- **Name** — Name of the parameter. Define and modify this property from the “Parameters Pane” on page 3-17.

- **Data type** — Lists the data type of the parameter. Define and modify this property from the “Parameters Pane” on page 3-17.
- **Value** — Specifies the value of the parameter. Enter a valid MATLAB expression in this field.

Port/Parameter Pane

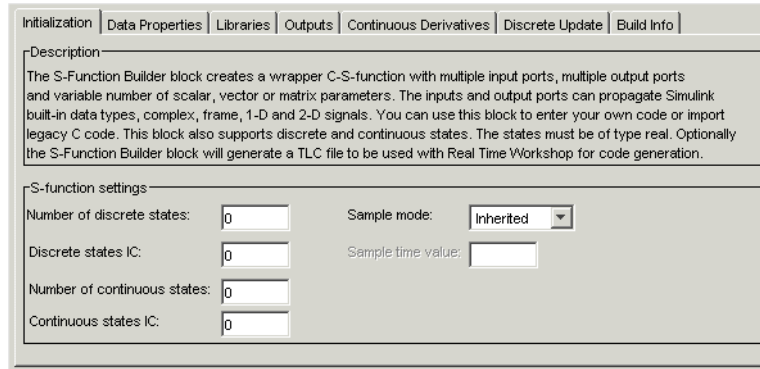
This pane displays the ports and parameters that the dialog box specifies for the target S-function.



The pane contains a tree control whose top nodes correspond to the target S-function’s input ports, output ports, and parameters, respectively. Expanding the Input Ports, Output Ports, or Parameter node displays the input ports, output ports, or parameters, respectively, specified for the target S-function. Selecting any of the port or parameter nodes selects the corresponding entry on the corresponding port or parameter specification pane.

Initialization Pane

The **Initialization** pane allows you to specify basic features of the S-function, such as the width of its input and output ports and its sample time.



The S-Function Builder uses the information that you enter on this pane to generate the S-function's `mdlInitializeSizes` callback method. Simulink invokes this method during the model initialization phase of the simulation to obtain basic information about the S-function. (See “How Simulink Interacts with C S-Functions” on page 3-39 for more information on the model initialization phase.)

The **Initialization** pane contains the following fields.

Number of discrete states

Number of discrete states that the S-function has.

Discrete states IC

Initial conditions of the S-function's discrete states. You can enter the values as a comma-separated list or as a vector (e.g., `[0 1 2]`). The number of initial conditions must equal the number of discrete states.

Number of continuous states

Number of continuous states that the S-function has.

Continuous states IC

Initial conditions of the S-function's continuous states. You can enter the values as a comma-separated list or as a vector (e.g., `[0 1 2]`). The number of initial conditions must equal the number of continuous states.

Sample mode

Sample mode of the S-function. The sample mode determines the length of the interval between the times when the S-function updates its output. You can select one of the following options:

- Inherited

The S-function inherits its sample time from the block connected to its input port.

- Continuous

The block updates its outputs at each simulation step.

- Discrete

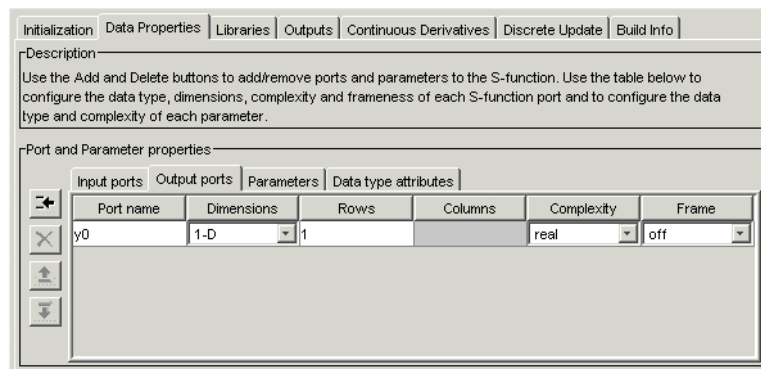
The S-function updates its outputs at the rate specified in the **Discrete sample time value** field of the S-Function Builder dialog box.

Sample time value

Interval between updates of the S-function's outputs. This field is enabled only if you have selected Discrete as the S-function's **Sample time**.

Data Properties Pane

The **Data Properties** pane allows you to add ports and parameters to your S-function.



The column of buttons to the left of the panes allows you to add, delete, or reorder ports or parameters, depending on the currently selected pane.

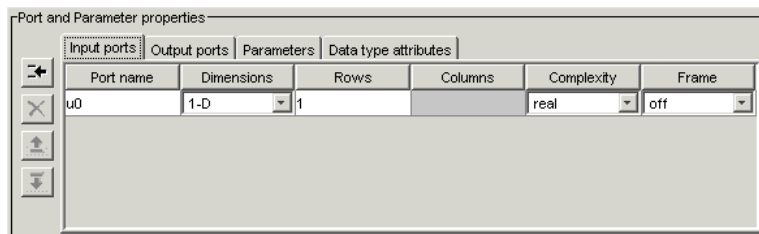
- To add a port or parameter, click the **Add** button (the top button in the column of buttons).
- To delete the currently selected port/parameter, click the **Delete** button (located beneath the **Add** button).
- To move the currently selected port/parameter up one position in the corresponding S-Function port/parameter list, click the **Up** button (beneath the **Delete** button).
- To move the currently selected port/parameter down one position in the corresponding S-function port/parameter list, click the **Down** button (beneath the **Up** button).

This pane also contains tabbed panes that enable you to specify the attributes of the ports and parameters that you create. See the following topics for more information.

- “Input Ports Pane” on page 3-14
- “Output Ports Pane” on page 3-15
- “Parameters Pane” on page 3-17
- “Data Type Attributes Pane” on page 3-18

Input Ports Pane

The **Input Ports** pane allows you to inspect and modify the properties of the S-function’s input ports.



The pane comprises an editable table that lists the properties of the input ports in the order in which the ports appear on the S-function block. Each row of the table corresponds to a port. Each entry in the row displays a property of the port as follows.

Port name

Name of the port. Edit this field to change the port name.

Dimensions

Lists the number of dimensions of input signals accepted by the port. To display a list of supported dimensions, click the adjacent button. To change the port's dimensionality, select a new value from the list. (Simulink signals can have at most two dimensions.)

Row

Specifies the size of the input signal's first (or only) dimension.

Column

Specifies the size of the input signal's second dimension (only if the input port accepts 2-D signals).

Complexity

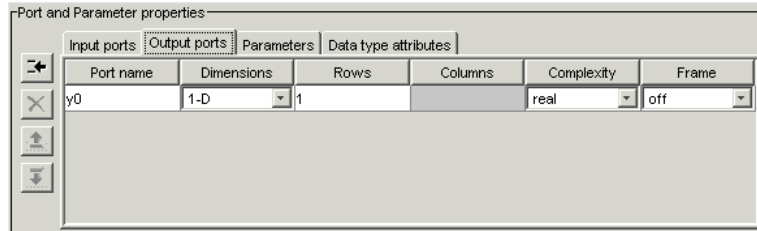
Specifies whether the input port accepts real or complex-valued signals.

Frame

Specifies whether this port accepts frame-based signals generated by the Signal Processing Blockset or Communications Blockset. For more information, see the documentation for these blocksets.

Output Ports Pane

The **Output Ports** pane allows you to inspect and modify the properties of the S-function's output ports.



The pane comprises an editable table that lists the properties of the output ports in the order in which the ports appear on the S-function block. Each row of the table corresponds to a port. Each entry in the row displays a property of the port as follows.

Port name

Name of the port. Edit this field to change the port name.

Dimensions

Lists the number of dimensions of signals output by the port. To display a list of supported dimensions, click the adjacent button. To change the port's dimensionality, select a new value from the list. (Simulink signals can have at most two dimensions).

Row

Specifies the size of the output signal's first (or only) dimension.

Column

Specifies the size of the output signal's second dimension (only if the output port accepts 2-D signals).

Complexity

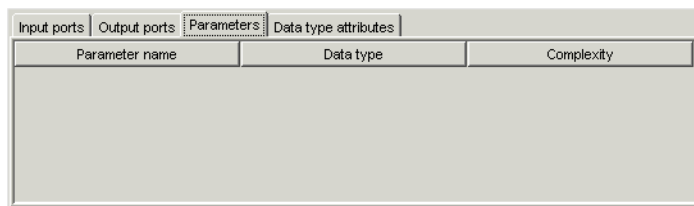
Specifies whether the port outputs real or complex-valued signals.

Frame

Specifies whether this port accepts frame-based signals generated by the Signal Processing Blockset or Communications Blockset. For more information, see the documentation for these blocksets.

Parameters Pane

The **Parameters** pane allows you to inspect and modify the properties of the S-function's parameters.



The pane comprises an editable table that lists the properties of the S-function's parameters. Each row of the table corresponds to a parameter. The order in which the parameters appear corresponds to the order in which the user must specify them. Each entry in the row displays a property of the parameter as follows.

Parameter name

Name of the parameter. Edit this field to change the name.

Data type

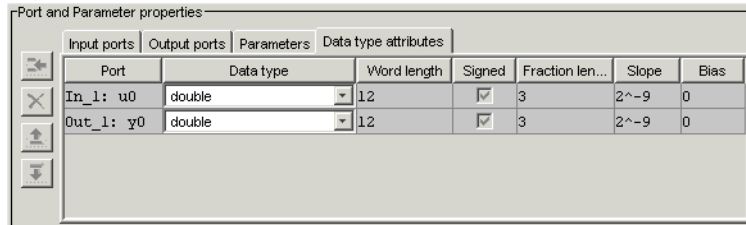
Lists the data type of the parameter. Click the adjacent button to display a list of supported data types. To change the parameter's data type, select a new type from the list.

Complexity

Specifies whether the parameter has real or complex values.

Data Type Attributes Pane

This pane allows you to specify the data type attributes of the target S-function's input and output ports.



The pane contains a table listing the data type attributes of each of the S-functions ports. Only some of the fields in the table are editable. Non-editable fields are grayed out. Each row corresponds to a port of the target S-function. Each column specifies an attribute of the corresponding port.

Port

Name of the port. This field is not editable.

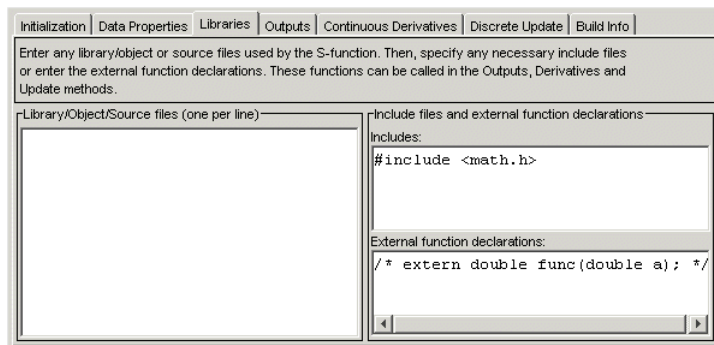
Data Type

Data type of the port. To display a list of specifiable data types, select the adjacent pulldown list control. To change the data type, select a different data type from the list.

The remaining fields on this pane are enabled only if the **Data Type** field specifies a fixed-point data type. See the Simulink Fixed Point documentation for more information.

Libraries Pane

The **Libraries** pane allows you to specify the location of external code files referenced by custom code that you enter in other panes of the S-Function Builder dialog box.



The **Libraries** pane includes the following fields.

Library/Object/Source files

External library, object code, and source files referenced by custom code that you enter elsewhere on the S-Function Builder dialog box. List each file on a separate line. If the file resides in the current directory, you need specify only the file's name. If the file resides in another directory, you must specify the full path of the file.

You can also use this field to specify search paths for libraries, object files, header files and source files. To do this, enter the tag `LIB_PATH`, `INC_PATH`, or `SRC_PATH`, respectively, followed by a comma- or semicolon-separated list of paths. You can make as many entries of this kind as you need but each must reside on a separate line.

For example, consider an S-Function Builder project that resides at `d:\matlab6p5\work` and needs to link against the following files:

- `c:\customfolder\customfunctions.lib`
- `d:\matlab7\customobjs\userfunctions.obj`
- `d:\externalsource\freesource.c`

The following entries enable the S-Function Builder to find these files:

```
Customfunctions.lib
Userfunctions.obj
```

```
LIB_PATH c:\customfolder\customfunctions.lib
LIB_PATH $MATLABROOT\customobjs
Freesource.c
SRC_PATH d:\externalsource
```

As this example illustrates, you can use `LIB_PATH` to specify both object and library file paths and the tag `$MATLABROOT` to indicate paths relative to the MATLAB installation. You can also include multiple `LIB_PATH` entries on separate lines. The paths are searched in the order specified.

You can also enter preprocessor (`-D`) directives in this field, e.g.,

```
-DDEBUG
```

Each directive must reside on a separate line.

Includes

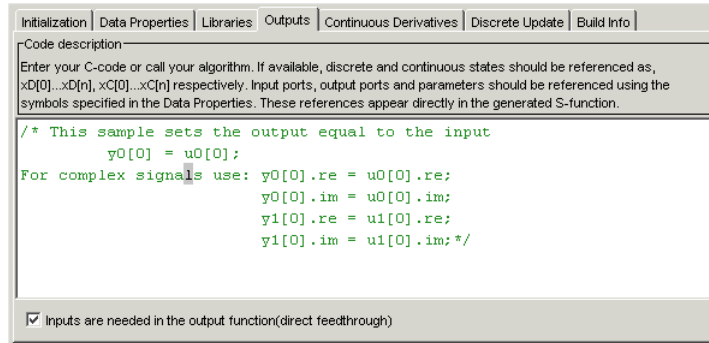
Header files containing declarations of functions, variables, and macros referenced by custom code that you enter elsewhere on the S-Function Builder dialog box. Specify each file on a separate line as `#include` statements. Use brackets to enclose the names of standard C header files (e.g., `#include <math.h>`). Use quotation marks to enclose names of custom header files (e.g., `#include "myutils.h"`). If your S-function uses custom include files that do not reside in the current directory, you must set the S-Function Builder's include path to the directories containing the include files (see "Setting the Include Path" on page 3-8).

External function declarations

Declarations of external functions not declared in the header files listed in the **Includes** field. Put each declaration on a separate line. The S-Function Builder includes the specified declarations in the S-function source file that it generates. This allows S-function code that computes the S-function's states or output to invoke the external functions.

Outputs Pane

Use the **Outputs** pane to enter code that computes the outputs of the S-function at each simulation time step.



The **Outputs** pane contains the following fields.

Code for the mdlOutputs function

Code that computes the output of the S-function at each simulation time step (or sample time hit, in the case of a discrete S-function). When generating the source code for the S-function, the S-Function Builder inserts the code in this field in a wrapper function of the form

```

void sfun_Outputs_wrapper(const real_T *u,
                          real_T      *y,
                          const real_T *xD, /* optional */
                          const real_T *xC, /* optional */
                          const real_T *param0, /* optional */
                          int_T p_width0 /* optional */
                          real_T *param1 /* optional */
                          int_t p_width1 /* optional */
                          int_T y_width, /* optional */
                          int_T u_width) /* optional */
{
    /* Your code inserted here */
}

```

where `sfun` is the name of the S-function. The S-Function Builder inserts a call to this wrapper function in the `mdlOutputs` callback method that it generates for your S-function. Simulink invokes the `mdlOutputs` method at each simulation time step (or sample time step in the case of a discrete

S-function) to compute the S-function's output. The S-function's `mdlOutputs` method in turn invokes the wrapper function containing your output code. Your output code then actually computes and returns the S-function's output.

The `mdlOutputs` method passes some or all of the following arguments to the outputs wrapper function.

Argument	Description
<code>u</code>	Pointer to an array containing the inputs to the S-function. The width of the array is the same as the input width you specified on the Initialization pane. If you specified -1 as the input width, the width of the array is specified by the wrapper function's <code>u_width</code> argument (see below).
<code>y</code>	Pointer to an array containing the output of the S-function. The width of the array is the same as the output width you specified on the Initialization pane. If you specified -1 as the output width, the width of the array is specified by the wrapper function's <code>y_width</code> argument (see below). Use this array to pass the outputs that your code computes back to Simulink.
<code>xD</code>	Pointer to an array containing the discrete states of the S-function. This argument appears only if you specified discrete states on the Initialization pane. At the first simulation time step, the discrete states have the initial values that you specified on the Initialization pane. At subsequent sample-time steps, the states are obtained from the values that the S-function computes at the preceding time step (see "Discrete Update Pane" on page 3-25 for more information).

Argument	Description
xC	Pointer to an array containing the continuous states of the S-function. This argument appears only if you specified continuous states on the Initialization pane. At the first simulation time step, the continuous states have the initial values that you specified on the Initialization pane. At subsequent time steps, the states are obtained by numerically integrating the derivatives of the states at the preceding time step (see “Continuous Derivatives Pane” on page 3-24 for more information).
param0, p_width0, param1, p_width1, ... paramN, p_widthN	param0, param1, ...paramN are pointers to arrays containing the S-function’s parameters, where N is the number of parameters specified on the Initialization pane. p_width0, p_width1, ...p_widthN are the widths of the parameter arrays. If a parameter is a matrix, the width equals the product of the dimensions of the arrays. For example, the width of a 3-by-2 matrix parameter is 6. These arguments appear only if you specify parameters on the Initialization pane.
y_width	Width of the array containing the S-function’s outputs. This argument appears in the generated code only if you specified -1 as the width of the S-function’s output. If the output is a matrix, y_width is the product of the dimensions of the matrix.
u_width	Width of the array containing the S-function’s inputs. This argument appears in the generated code only if you specified -1 as the width of the S-function’s input. If the input is a matrix, u_width is the product of the dimensions of the matrix.

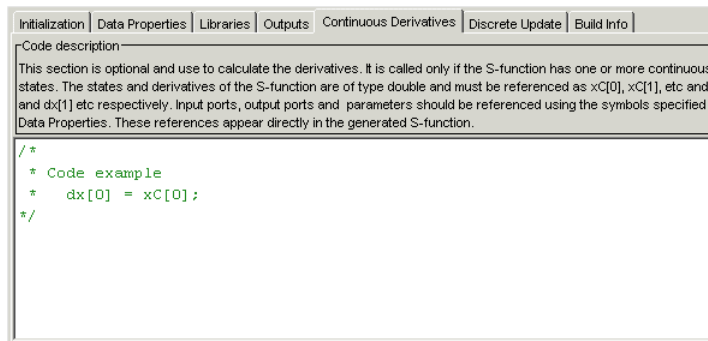
These arguments permit you to compute the output of the block as a function of its inputs and, optionally, its states and parameters. The code that you enter in this field can invoke external functions declared in the header files or external declarations on the **Libraries** pane. This allows you to use existing code to compute the outputs of the S-function.

Inputs are needed in the output function

Selected if the current values of the S-function's inputs are used to compute its outputs. Simulink uses this information to detect algebraic loops created by directly or indirectly connecting the S-function's output to its input.

Continuous Derivatives Pane

If the S-function has continuous states, use the **Continuous Derivatives** pane to enter code required to compute the state derivatives.



Enter code to compute the derivatives of the S-function's continuous states in the **Code for the mdlDerivatives function** field on this pane. When generating code, the S-Function Builder takes the code in this pane and inserts it in a wrapper function of the form

```

void sfun_Derivatives_wrapper(const real_T *u,
    const real_T *y,
    real_T *dx,
    real_T *xC,
    const real_T *param0, /* optional */
    int_T p_width0, /* optional */
    real_T *param1, /* optional */
    int_T p_width1, /* optional */
    int_T y_width, /* optional */
    int_T u_width) /* optional */
{
    /* Your code inserted here. */
}
  
```

```
}
```

where `sfun` is the name of the S-function. The S-Function Builder inserts a call to this wrapper function in the `mdlDerivatives` callback method that it generates for the S-function. Simulink calls the `mdlDerivatives` method at the end of each time step to obtain the derivatives of the S-function's continuous states (see “How Simulink Interacts with C S-Functions” on page 3-39). The Simulink solver numerically integrates the derivatives to determine the continuous states at the next time step. At the next time step, Simulink passes the updated states back to the S-function's `mdlOutputs` method (see “Outputs Pane” on page 3-20).

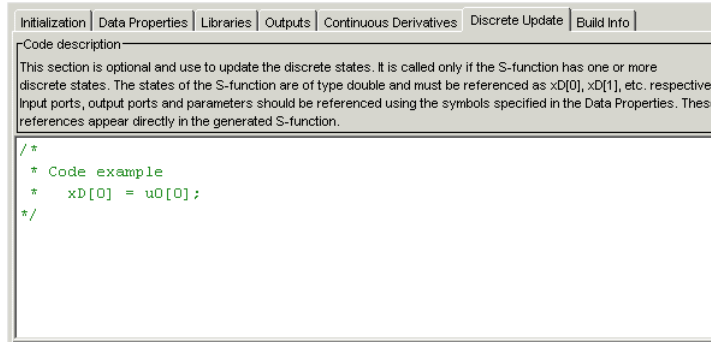
The generated S-function's `mdlDerivatives` callback method passes the following arguments to the derivatives wrapper function:

- `u`
- `y`
- `dx`
- `xC`
- `param0, p_width0, param1, p_width1, ... paramN, p_widthN`
- `y_width`
- `x-width`

The `dx` argument is a pointer to an array whose width is the same as the number of continuous derivatives specified on the **Initialization** pane. Your code should use this array to return the values of the derivatives that it computes. See “`mdlOutputs`” on page 3-33 for the meanings and usage of the other arguments. The arguments allow your code to compute derivatives as a function of the S-function's inputs, outputs, and, optionally, parameters. Your code can invoke external functions declared on the **Libraries** pane.

Discrete Update Pane

If the S-function has discrete states, use the **Discrete Update** pane to enter code that computes at the current time step the values of the discrete states at the next time step.



Enter code to compute the values of the S-function's discrete states in the **Code for the mdlUpdate function** field on this pane. When generating code, the S-Function Builder takes the code in this pane and inserts it in a wrapper function of the form

```
void sfun_Update_wrapper(const real_T *u,
    const real_T *y,
    real_T *xD,
    const real_T *param0, /* optional */
    int_T p_width0, /* optional */
    real_T *param1, /* optional */
    int_T p_width1, /* optional */
    int_T y_width, /* optional */
    int_T u_width) /* optional */
{
    /* Your code inserted here. */
}
```

where `sfun` is the name of the S-function. The S-Function Builder inserts a call to this wrapper function in the `mdlUpdate` callback method that it generates for the S-function. Simulink calls the `mdlUpdate` method at the end of each time step to obtain the values of the S-function's discrete states at the next time step (see "How Simulink Interacts with C S-Functions" on page 3-39). At the next time step, Simulink passes the updated states back to the S-function's `mdlOutputs` method (see "Outputs Pane" on page 3-20).

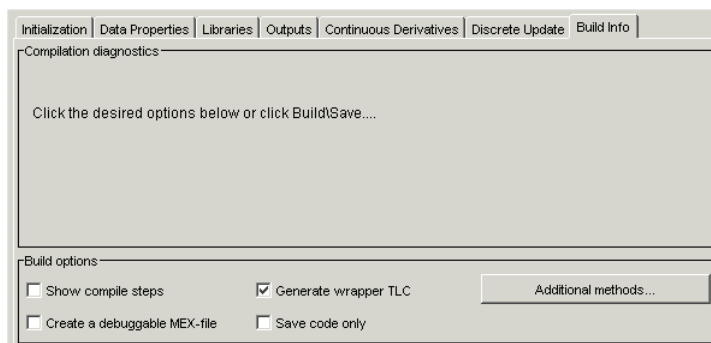
The generated S-function's `mdlUpdates` callback method passes the following arguments to the updates wrapper function:

- `u`
- `y`
- `xD`
- `param0, p_width0, param1, p_width1, ... paramN, p_widthN`
- `y_width`
- `x-width`

See “`mdlOutputs`” on page 3-33 for the meanings and usage of these arguments. Your code should use the `xD` (discrete states) variable to return the values of the derivatives that it computes. The arguments allow your code to compute the discrete states as functions of the S-function's inputs, outputs, and, optionally, parameters. Your code can invoke external functions declared on the **Libraries** pane.

Build Info Pane

Use the **Build Info** pane to specify options for building the S-function Mex-file.



This pane contains the following fields.

Show compile steps

Log each build step in the **Compilation diagnostics** field.

Create a debuggable MEX-file

Include debug information in the generated MEX-file.

Generate wrapper TLC

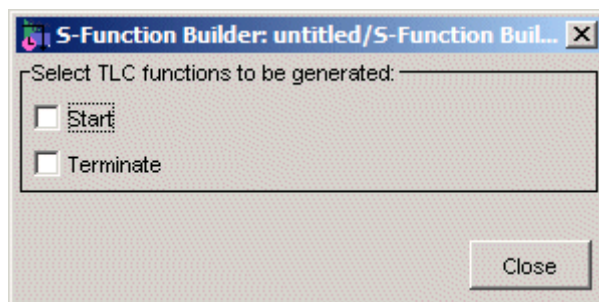
Generate a TLC file. You do not need to generate a TLC file if you do not expect the S-function ever to run in accelerated mode or be used in a model from which Real-Time Workshop generates code.

Save code only

Do not build a MEX-file from the generated source code.

Additional methods

Click this button to include TLC methods in your S-function. The following dialog box appears.

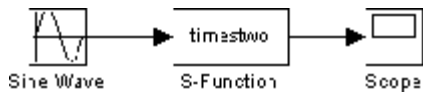


Check the methods you want to add and click the **Close** button to include the methods in your S-function. See “Block Target File Methods” for more information.

Example of a Basic C MEX S-Function

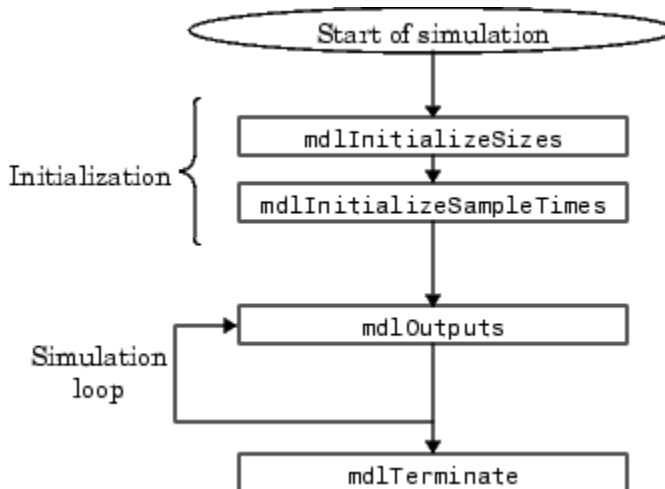
This section presents an example of a C MEX S-function that you can use as a model for creating simple C S-functions. The example is the `timestwo` S-function example that comes with Simulink (see `matlabroot/simulink/src/timestwo.c`). This S-function outputs twice its input.

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 `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 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 timestwo
#define S_FUNCTION_LEVEL 2

```

The first specifies the name of the S-function (`timestwo`). 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 3-48).

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 timestwo
#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 9, “SimStruct Functions — Categorical List”).

Callback Implementations

The next part of the `timestwo` 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 `timestwo` implementation of `mdlInitializeSizes` specifies the following size information:

- Zero parameters

This means that the **S-function parameters** field of the S-function’s dialog box must be empty. If it contains any parameters, Simulink reports 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 2 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 `timestwo` example specifies the actual value of the sample time in the `mdlInitializeSampleTimes` routine.

- The code is exception free.

Specifying exception-free code speeds up execution of your S-function. You must take care 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 3-39.

mdlInitializeSampleTimes

Simulink calls `mdlInitializeSampleTimes` to set the sample times of the S-function. A `timestwo` block executes whenever the driving block executes. Therefore, it has a single inherited sample time, `SAMPLE_TIME_INHERITED`.

mdlOutputs

Simulink calls `mdlOutputs` at each time step to calculate a block's outputs. The `timestwo` implementation of `mdlOutputs` takes the input, multiplies it by 2, and writes the answer to the output.

The `timestwo` `mdlOutputs` method uses a `SimStruct` macro,

```
InputRealPtrsType uPtrs = ssGetInputPortRealSignalPtrs(S,0);
```

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

```
*uPtrs[i]
```

For more details, see "Data View" on page 3-43.

The `timestwo` `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 the end of the simulation. This is a mandatory S-function routine. However, the `timestwo` 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 Real-Time Workshop.

```
#ifdef MATLAB_MEX_FILE
#include "simulink.c"
#else
#include "cg_sfun.h"
#endif
```

Building the Timestwo Example

To incorporate this S-function into Simulink, enter

```
mex timestwo.c
```

at the command line. The `mex` command compiles and links the `timestwo.c` file to create a dynamically loadable executable for Simulink to 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_basic.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_sfunction_name_here
#define S_FUNCTION_LEVEL 2
#include "simstruc.h"
```

where *your_sfunction_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 3-48. 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.

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 Real-Time Workshop, `simstruc.h` includes the following.

Header Files Included by `simstruc.h` When Used by 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_matrx.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.

```

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

```

```
#endif
```

These statements select the appropriate code for your particular application:

- `simulink.c` is included if the file is being compiled into a MEX-file.
- `cg_sfun.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/simstruc.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 9, “`SimStruct` Functions — Categorical List” 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 non-real-time application using a variable-step solver.

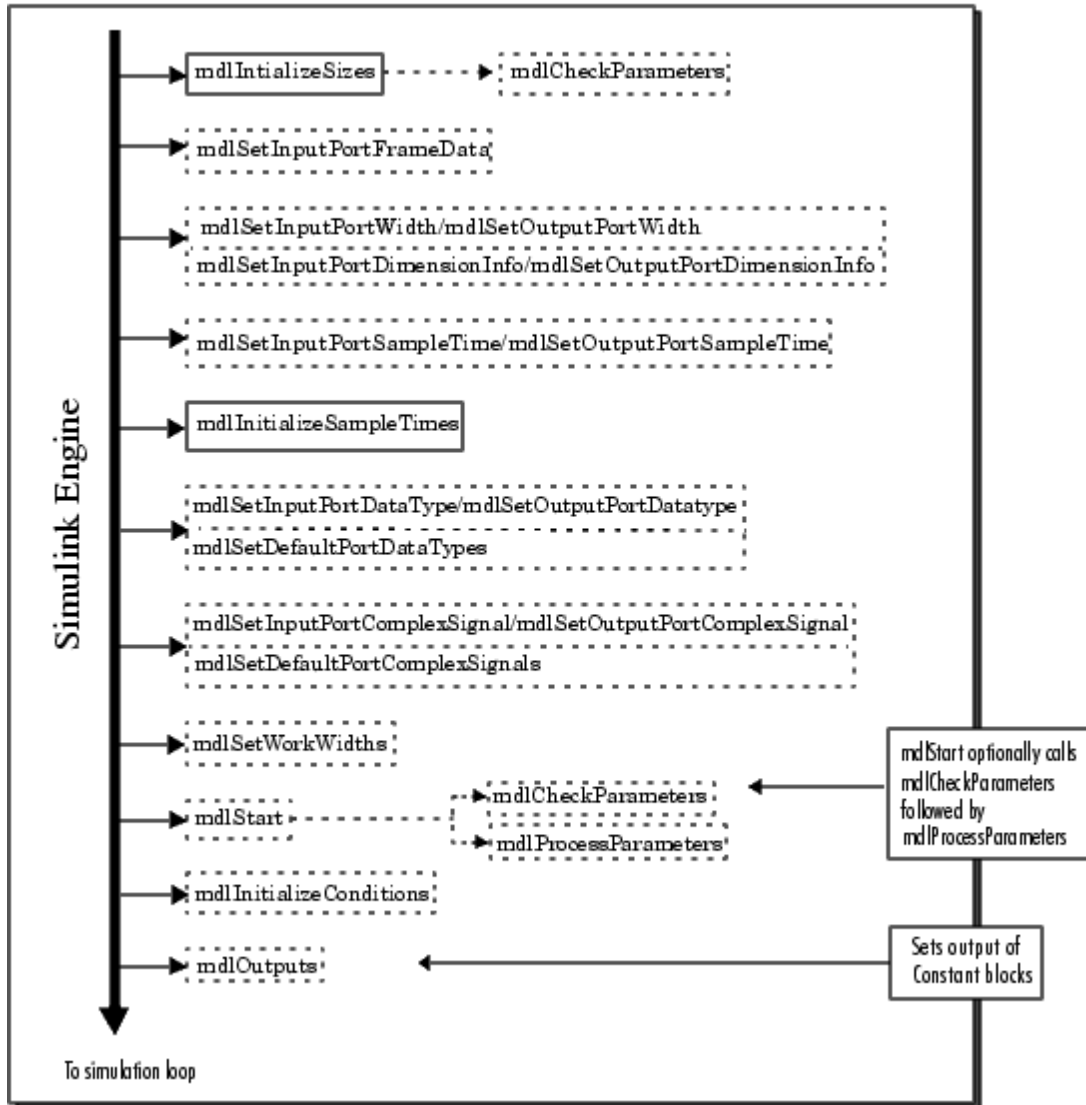
How Simulink Interacts with C S-Functions

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.

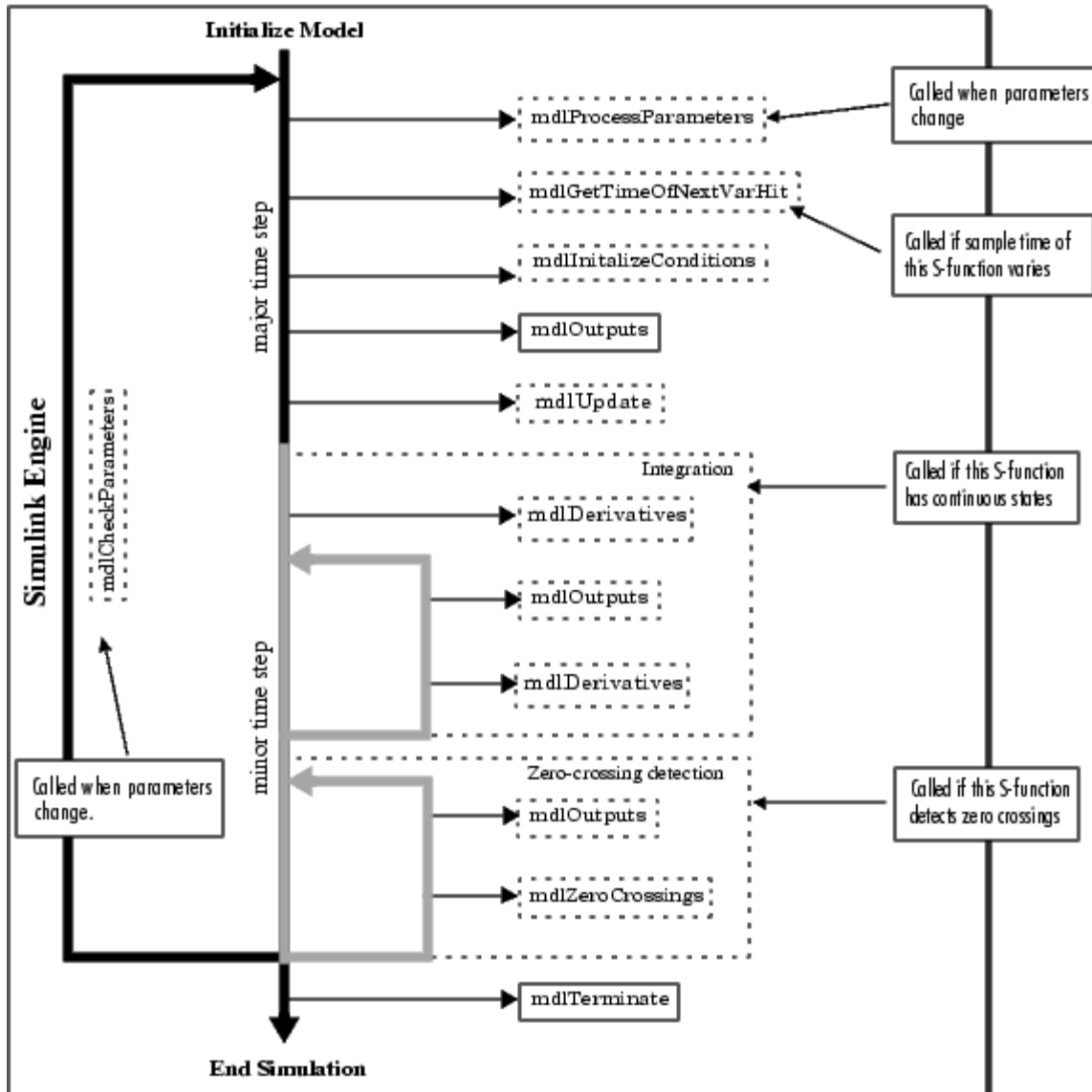
Process View

The following figures show the order in which Simulink invokes an S-function's callback methods. Solid rectangles indicate callbacks that always occur during model initialization and/or at every time step. Dotted rectangles indicate callbacks that may occur during initialization and/or at some or all time steps during the simulation loop. See the documentation for each callback method in Chapter 8, "S-Function Callback Methods — Alphabetical List" to determine the exact circumstances under which Simulink invokes the callback.

Model Initialization



Simulation Loop



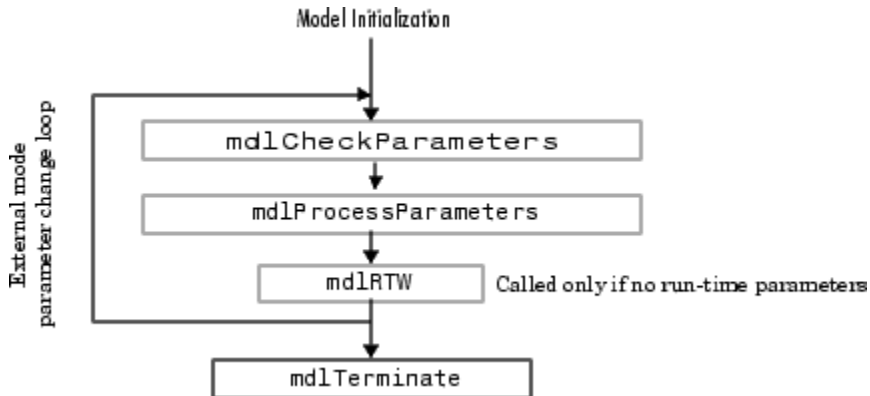
Calling Structure for Real-Time Workshop

When generating code, Real-Time Workshop does not go through the entire calling sequence outlined above. After initializing the model as outlined in the preceding section, Simulink calls mdlRTW, an S-function routine unique to Real-Time Workshop, mdlTerminate, and exits.

For more information about Real-Time Workshop and how it interacts with S-functions, see the Real-Time Workshop documentation and the “Real-Time Workshop Target Language Compiler Reference Guide”.

Alternate Calling Structure for External Mode

When you are running Simulink in external mode, the calling sequence for S-function routines changes. This picture shows the correct sequence for external mode.



Simulink calls mdlRTW once when it enters external mode and again each time a parameter changes or when you select **Update Diagram** under your model's **Edit** menu.

Note Running Simulink in external mode requires Real-Time Workshop. For more information about external mode, see the Real-Time Workshop documentation.

Data View

S-function blocks have input and output signals, parameters, and internal states, plus other general work areas. In general, block inputs and outputs are written to, and read from, a block I/O vector. Inputs can also come from

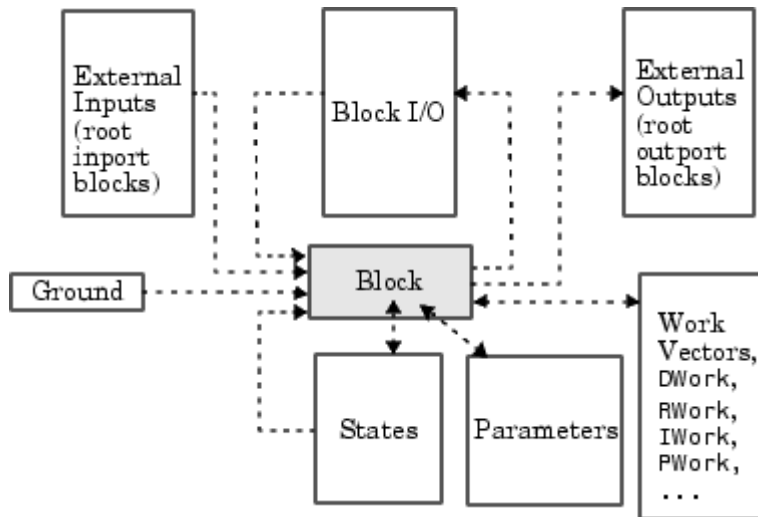
- External inputs via the root inport blocks
- Ground if the input signal is unconnected or grounded

Block outputs can also go to the external outputs via the root outport blocks. In addition to input and output signals, S-functions can have

- Continuous states
- Discrete states
- Other working areas such as real, integer or pointer work vectors

You can parameterize S-function blocks by passing parameters to them using the S-function block dialog box.

The following figure shows the general mapping between these various types of data.



An S-function's `mdlInitializeSizes` routine sets the sizes of the various signals and vectors. S-function methods called during the simulation loop can determine the sizes and values of the signals.

An S-function method can access input signals in two ways:

- Via pointers
- Using contiguous inputs

Accessing Signals Using Pointers

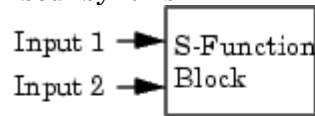
During the simulation loop, accessing the input signals is performed using

```
InputRealPtrsType uPtrs =  
    ssGetInputPortRealSignalPtrs(S,portIndex)
```

This is an array of pointers, where *portIndex* starts at 0. There is one for each input port. To access an element of this signal you must use

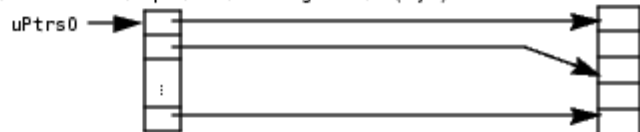
```
*uPtrs[element]
```

as described by this



To access Input 1:

```
InputRealPtrsType uPtrs0 = ssGetInputPortRealSignalPtrs(S,0)
```



To access Input 2:

```
InputRealPtrsType uPtrs1 = ssGetInputPortRealSignalPtrs(S,1)
```

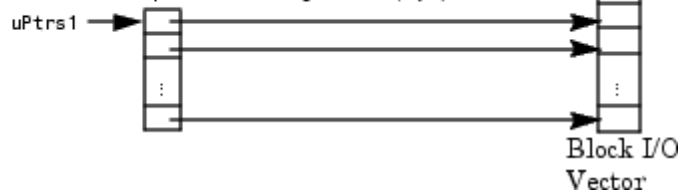


figure.

Note that input array pointers can point at noncontiguous places in memory. You can retrieve the output signal by using this code.

```
real_T *y = ssGetOutputPortSignal(S,outputPortIndex);
```

Accessing Contiguous Input Signals

An S-function's `mdlInitializeSizes` method can specify that the elements of its input signals must occupy contiguous areas of memory, using `ssSetInputPortRequiredContiguous`. If the inputs are contiguous, other methods can use `ssGetInputPortSignal` to access the inputs.

Accessing Input Signals of Individual Ports

This section describes how to access all input signals of a particular port and write them to the output port. The preceding figure shows that the input array of pointers can point to noncontiguous entries in the block I/O vector. The output signals of a particular port form a contiguous vector. Therefore, the

correct way to access input elements and write them to the output elements (assuming the input and output ports have equal widths) is to use this code.

```
int_T element;
int_T portWidth = ssGetInputPortWidth(S,inputPortIndex);
InputRealPtrsType uPtrs = ssGetInputPortRealSignalPtrs(S,inputPortIndex);
real_T *y = ssGetOutputPortSignal(S,outputPortIdx);

for (element=0; element<portWidth; element++) {
    y[element] = *uPtrs[element];
}
```

A common mistake is to try to access the input signals via pointer arithmetic. For example, if you were to place

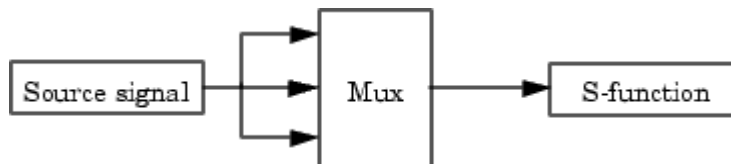
```
real_T *u = *uPtrs; /* Incorrect */
```

just below the initialization of `uPtrs` and replace the inner part of the above loop with

```
*y++ = *u++; /* Incorrect */
```

the code compiles, but the MEX-file might crash Simulink. This is because it is possible to access invalid memory (which depends on how you build your model). When accessing the input signals incorrectly, a crash occurs when the signals entering your S-function block are not contiguous. Noncontiguous signal data occurs when signals pass through virtual connection blocks such as the Mux or Selector blocks.

To verify that you are correctly accessing wide input signals, pass a replicated signal to each input port of your S-function. You do this by creating a Mux block with the number of input ports equal to the width of the desired signal entering your S-function. Then the driving source should be connected to each input port as shown in this figure.



Writing Callback Methods

Writing an S-function basically involves creating implementations of the callback functions that Simulink invokes during a simulation. For guidelines on implementing a particular callback, see the documentation for the callback in Chapter 8, “S-Function Callback Methods — Alphabetical List”. For information on using callbacks to implement specific block features, such as parameters or sample times, see Chapter 7, “Implementing Block Features”.

Converting Level 1 C MEX S-Functions to Level 2

Level 2 S-functions were introduced with Simulink 2.2. Level 1 S-functions refer to S-functions that were written to work with Simulink 2.1 and previous releases. Level 1 S-functions are compatible with Simulink 2.2 and subsequent releases; you can use them in new models without making any code changes. However, to take advantage of new features in S-functions, level 1 S-functions must be updated to level 2 S-functions. Here are some guidelines:

- Start by looking at `simulink/src/sfunctmpl_doc.c`. This template S-function file concisely summarizes level 2 S-functions.
- At the top of your S-function file, add this define:

```
#define S_FUNCTION_LEVEL 2
```

- Update the contents of `mdlInitializeSizes`. In particular, add the following error handling for the number of S-function parameters:

```
ssSetNumSFcnParams(S, NPARAMS); /*Number of expected parameters*/
if (ssGetNumSFcnParams(S) != ssGetSFcnParamsCount(S)) {
    /* Return if number of expected != number of actual parameters */
    return;
}
Set up the inputs using:
if (!ssSetNumInputPorts(S, 1)) return; /*Number of input ports */
ssSetInputPortWidth(S, 0, width);      /* Width of input
                                         port one (index 0)*/
ssSetInputPortDirectFeedThrough(S, 0, 1); /* Direct feedthrough
                                         or port one */

ssSetInputPortRequiredContiguous(S, 0);
Set up the outputs using:
if (!ssSetNumOutputPorts(S, 1)) return;
ssSetOutputPortWidth(S, 0, width);     /* Width of output port
                                         one (index 0) */
```

- If your S-function has a nonempty `mdlInitializeConditions`, update it to the following form:

```
#define MDL_INITIALIZE_CONDITIONS
```

```
static void mdlInitializeConditions(SimStruct *S)
{
}

```

Otherwise, delete the function.

- Access the continuous states using `ssGetContStates`. The `ssGetX` macro has been removed.
- Access the discrete states using `ssGetRealDiscStates(S)`. The `ssGetX` macro has been removed.
- For mixed continuous and discrete state S-functions, the state vector no longer consists of the continuous states followed by the discrete states. The states are saved in separate vectors and hence might not be contiguous in memory.
- The `mdlOutputs` prototype has changed from

```
static void mdlOutputs( real_T *y, const real_T *x,
    const real_T *u, SimStruct *S, int_T tid)

```

to

```
static void mdlOutputs(SimStruct *S, int_T tid)

```

Since `y`, `x`, and `u` are not explicitly passed in to level-2 S-functions, you must use

- `ssGetInputPortSignal` to access inputs
 - `ssGetOutputPortSignal` to access the outputs
 - `ssGetContStates` or `ssGetRealDiscStates` to access the states
 - The `mdlUpdate` function prototype has changed from
- ```
void mdlUpdate(real_T *x, real_T *u, Simstruct *S, int_T tid)
```
- to
- ```
void mdlUpdate(SimStruct *S, int_T tid)
```
- If your S-function has a nonempty `mdlUpdate`, update it to this form:

```
#define MDL_UPDATE
static void mdlUpdate(SimStruct *S, int_T tid)
{
}
```

Otherwise, delete the function.

- If your S-function has a nonempty `mdlDerivatives`, update it to this form:

```
#define MDL_DERIVATIVES
static void mdlDerivatives(SimStruct *S, int_T tid)
{
}
```

Otherwise, delete the function.

- Replace all obsolete `SimStruct` macros. See “Obsolete Macros” on page 3-50 for a complete list of obsolete macros.
- When converting level 1 S-functions to level 2 S-functions, you should build your S-functions with full (i.e., highest) warning levels. For example, if you have `gcc` on a UNIX system, use these options with the `mex` utility.

```
mex CC=gcc CFLAGS=-Wall sfcn.c
```

If your system has `Lint`, use this code.

```
lint -DMATLAB_MEX_FILE -I<matlabroot>/simulink/include
      -Imatlabroot/extern/include sfcn.c
```

On a PC, to use the highest warning levels, you must create a project file inside the integrated development environment (IDE) for the compiler you are using. Within the project file, define `MATLAB_MEX_FILE` and add

```
matlabroot/simulink/include
matlabroot/extern/include
```

to the path (be sure to build with alignment set to 8).

Obsolete Macros

The following macros are obsolete. Each obsolete macro should be replaced with the specified macro.

Obsolete Macro	Replace With
ssGetU(S), ssGetUPtrs(S)	ssGetInputPortSignalPtrs(S,port)
ssGetY(S)	ssGetOutputPortRealSignal(S,port)
ssGetX(S)	ssGetContStates(S), ssGetRealDiscStates(S)
ssGetStatus(S)	Normally not used, but ssGetErrorStatus(S) is available.
ssSetStatus(S,msg)	ssSetErrorStatus(S,msg)
ssGetSizes(S)	Specific call for the wanted item (i.e., ssGetNumContStates(S))
ssGetMinStepSize(S)	No longer supported.
ssGetPresentTimeEvent(S,sti)	ssGetTaskTime(S,sti)
ssGetSampleTimeEvent(S,sti)	ssGetSampleTime(S,sti)
ssSetSampleTimeEvent(S,t)	ssSetSampleTime(S,sti,t)
ssGetOffsetTimeEvent(S,sti)	ssGetOffsetTime(S,sti)
ssSetOffsetTimeEvent(S,sti,t)	ssSetOffsetTime(S,sti,t)
ssIsSampleHitEvent(S,sti,tid)	ssIsSampleHit(S,sti,tid)
ssGetNumInputArgs(S)	ssGetNumSFcnParams(S)
ssSetNumInputArgs(S, numInputArgs)	ssSetNumSFcnParams(S,numInputArgs)
ssGetNumArgs(S)	ssGetSFcnParamsCount(S)
ssGetArg(S,argNum)	ssGetSFcnParam(S,argNum)
ssGetNumInputs	ssGetNumInputPorts(S) and ssGetInputPortWidth(S,port)
ssSetNumInputs	ssSetNumInputPorts(S,nInputPorts) and ssSetInputPortWidth(S,port,val)
ssGetNumOutputs	ssGetNumOutputPorts(S) and ssGetOutputPortWidth(S,port)
ssSetNumOutputs	ssSetNumOutputPorts(S,nOutputPorts) and ssSetOutputPortWidth(S,port,val)

Creating C++ S-Functions

The procedure for creating C++ S-functions is nearly the same as that for creating C S-functions (see Chapter 3, “Writing S-Functions in C”). The following sections explain the differences.

Source File Format (p. 4-2)	Explains the differences between the source file structure of a C++ S-function and a C S-function.
Making C++ Objects Persistent (p. 4-6)	How to create C++ objects that persist across invocations of the S-function.
Building C++ S-Functions (p. 4-8)	How to build a C++ S-function.

Source File Format

The format of the C++ source for an S-function is nearly identical to that of the source for an S-function written in C. The main difference is that you must tell the C++ compiler to use C calling conventions when compiling the callback methods. This is necessary because the Simulink simulation engine assumes that callback methods obey C calling conventions.

To tell the compiler to use C calling conventions when compiling the callback methods, wrap the C++ source for the S-function callback methods in an extern "C" statement. The C++ version of the `sfun_counter` S-function example (`matlabroot/simulink/src/sfun_counter_cpp.cpp`) illustrates usage of the extern "C" directive to ensure that the compiler generates Simulink-compatible callback methods:

```
/* File   : sfun_counter_cpp.cpp
 * Abstract:
 *
 * Example of an C++ S-function which stores an C++ object in
 * the pointers vector PWork.
 *
 * Copyright 1990-2000 The MathWorks, Inc.
 *
 */

#include "iostream.h"

class counter {
    double x;
public:
    counter() {
        x = 0.0;
    }
    double output(void) {
        x = x + 1.0;
        return x;
    }
};

#ifdef __cplusplus
```



```
extern "C" { // use the C fcn-call standard for all functions
#ifdef // defined within this scope

#define S_FUNCTION_LEVEL 2
#define S_FUNCTION_NAME  sfun_counter_cpp

/*
 * Need to include simstruc.h for the definition of the SimStruct and
 * its associated macro definitions.
 */
#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.c 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.c 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_sfuns.h" /* Code generation registration function */
#endif

#ifdef __cplusplus
} // end of extern "C" scope
#endif

```

Making C++ Objects Persistent

Your C++ callback methods might need to create persistent C++ objects, that is, objects that continue to exist after the method exits. For example, a callback method might need to access an object created during a previous invocation. Or one callback method might 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

The following sections explain how to use the Ada programming language to create S-functions.

Introduction (p. 5-2)	Overview of creating Ada S-functions.
Ada S-Function Source File Format (p. 5-3)	Source code structure of an Ada S-function.
Writing Callback Methods in Ada (p. 5-7)	How to use Ada to implement S-function callback methods.
Building an Ada S-Function (p. 5-10)	Compiling and linking an Ada S-function.
Example of an Ada S-Function (p. 5-11)	An Ada version of the timestwo S-function example.

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 9, “SimStruct Functions — Categorical List”). 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 occured 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;

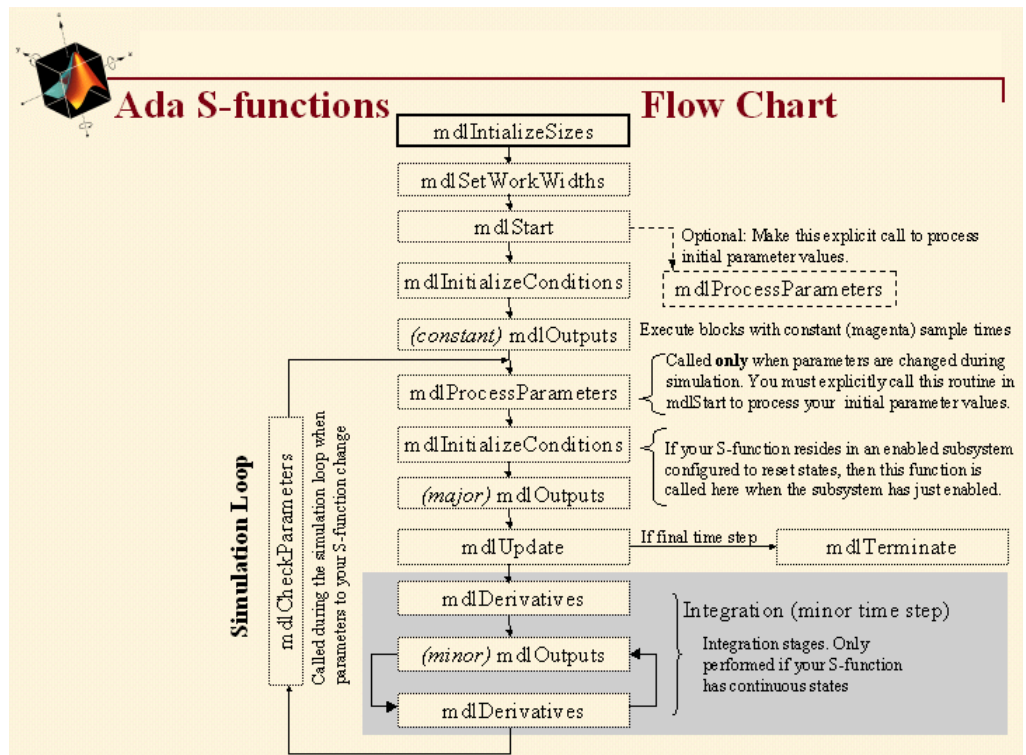
end Times_Two;
```

Writing Callback Methods in Ada

Simulink interacts with an Ada S-function by invoking callback methods that the S-function implements. This section specifies the callback methods that an Ada S-function can implement and provides guidelines for implementing them.

Callbacks Invoked by Simulink

The following diagram shows the callback methods that Simulink invokes when interacting with an Ada S-function during a simulation and the order in which Simulink invokes them.



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 `mdlOutputs` 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 8, “S-Function Callback Methods — Alphabetical List” explains the purpose of each callback 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 5-10) 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 9, “SimStruct Functions — Categorical List”). 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 `timestwo` S-function example that comes with Simulink, enter the command

```
mex -ada timestwo.ads
```

Ada Compiler Requirements

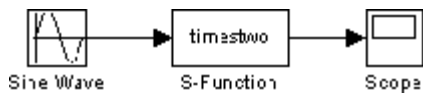
To build a Mex-file from Ada source code, using the `mex` tool, you must have previously installed a copy of Version 3.12 (or higher) of the GNAT Ada95 compiler on your system. You can obtain the latest Solaris, Windows, and GNU-Linux versions of the compiler at the GNAT ftp site (<ftp://cs.nyu.edu/pub/gnat>). Make sure that the compiler executable is in the MATLAB command path so that the `mex` tool can find it.

The GNAT Ada95 compiler package used to include `gnatd11.exe`, a tool for building DLLs on Windows. This tool, which is required to build Ada Mex-files on Windows, now comes as part of a separate `gnatwin` package containing Windows-specific files. If you want to build Ada S-functions on a Windows system, you must download and install the `gnatwin` package as well as the GNAT Ada95 compiler.

Example of an Ada S-Function

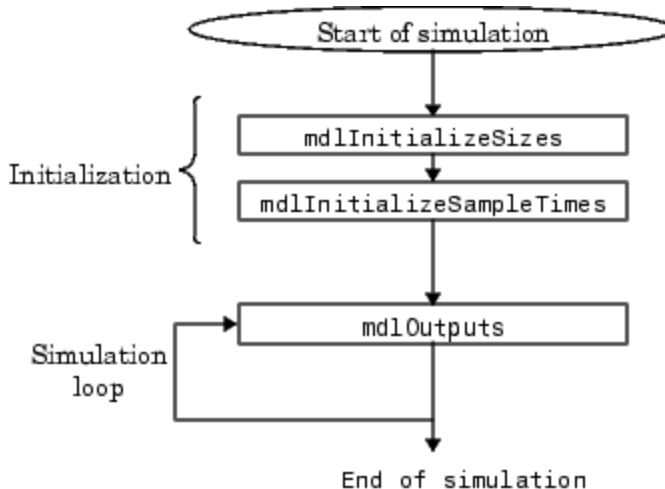
This section presents an example of a basic Ada S-function that you can use as a model when creating your own Ada S-functions. The example is the `timestwo` S-function example that comes with Simulink (see `matlabroot/simulink/ada/examples/timestwo.ads` and `matlabroot/simulink/ada/examples/timestwo.adb`). 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 explain 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 9, “SimStruct Functions — Categorical List”).

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, the MATLAB `mex` tool uses the package specification to determine the callbacks that the S-function does not implement. It then generates stubs for the nonimplemented 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);
    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;

```

The package body contains implementations of the callback methods needed to implement the `timestwo` example.

mdlInitializeSizes

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

The `timestwo` implementation of `mdlInitializeSizes` uses `SimStruct` functions defined in the Simulink package to specify the following size information:

- One input port and one output port

The widths of the input and output port are dynamically sized. This tells Simulink to multiply each element of the input signal to the S-function by 2 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

Finally the method provides an exception handler to handle any errors that occur in invoking the `SimStruct` functions.

mdlOutputs

Simulink calls `mdlOutputs` at each time step to calculate a block's outputs. The `timestwo` implementation of `mdlOutputs` takes the input, multiplies it by 2, and writes the answer to the output.

The `timestwo` implementation of the `mdlOutputs` method uses the `SimStruct` functions `ssGetInputPortWidth` and `ssGetInputPortSignalAddress` to access the input signal.

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

Similarly, the `mdlOutputs` method uses the functions `ssGetOutputPortWidth` and `ssGetOutputPortSignalAddress` to access the output signal.

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

Finally the method loops over the inputs to compute the outputs.

Building the Timestwo Example

To build this S-function into Simulink, enter

```
mex -ada timestwo.abs
```

at the command line.

Creating Fortran S-Functions

The following sections explain how to use the Fortran programming language to create S-functions.

Introduction (p. 6-2)

Overview of approaches to writing Fortran S-functions.

Creating Level 1 Fortran S-Functions (p. 6-3)

Describes a purely Fortran approach to creating an S-function.

Creating Level 2 Fortran S-Functions (p. 6-7)

Describes a hybrid C/Fortran approach to writing an S-function that enables creation of more capable blocks.

Porting Legacy Code (p. 6-15)

How to wrap an S-function around existing Fortran code.

Introduction

There are two main strategies to executing Fortran code from Simulink. One is from a Level 1 Fortran-MEX (F-MEX) S-function, the other is from a Level 2 gateway S-function written in C. Each has its advantages and both can be incorporated into code generated by Real-Time Workshop.

Level 1 Versus Level 2 S-Functions

The original S-function interface was called the Level 1 API. As the capabilities of Simulink grew, the S-function API was rearchitected into the more extensible Level 2 API. This allows S-functions to have all the capabilities of a full Simulink model (except automatic algebraic loop identification and solving) and to grow as Simulink grows.

Note The Level 1 API supports creation of S-functions having only continuous sample time. If you want to create a Fortran S-function with a discrete sample time, you must use the Level 2 API.

Creating Level 1 Fortran S-Functions

Fortran MEX Template File

A template file for Fortran MEX S-functions is located at *matlabroot/simulink/src/sfuntmpl_fortran.for*. The template file compiles as is and copies the input to the output.

To use the template to create a new Fortran S-function:

- 1 Create a copy under another filename.
- 2 Edit the copy to perform the operations you need.
- 3 Compile the edited file into a Mex-file, using the `mex` command.
- 4 Include the Mex-file in your model, using the S-Function block.

Example

The example file, *matlabroot/simulink/src/sfun_timestwo_for.for*, implements an S-function that multiplies its input by 2.

```

C
C File:  SFUN_TIMESTWO_FOR.F
C
C Abstract:
C   A sample Level 1 FORTRAN representation of a
C   timestwo S-function.
C
C   The basic mex command for this example is:
C
C   >> mex sfun_timestwo_for.for simulink.for
C
C   Copyright 1990-2000 The MathWorks, Inc.
C
C
C
C=====
C   Function:  SIZES
C

```

```
C   Abstract:
C     Set the size vector.
C
C     SIZES returns a vector which determines model
C     characteristics. This vector contains the
C     sizes of the state vector and other
C     parameters. More precisely,
C     SIZE(1) number of continuous states
C     SIZE(2) number of discrete states
C     SIZE(3) number of outputs
C     SIZE(4) number of inputs
C     SIZE(5) number of discontinuous roots in
C             the system
C     SIZE(6) set to 1 if the system has direct
C             feedthrough of its inputs,
C             otherwise 0
C
C=====
C
C   SUBROUTINE SIZES(SIZE)
C     .. Array arguments ..
C     INTEGER*4      SIZE(*)
C     .. Parameters ..
C     INTEGER*4      NSIZES
C     PARAMETER      (NSIZES=6)
C
C     SIZE(1) = 0
C     SIZE(2) = 0
C     SIZE(3) = 1
C     SIZE(4) = 1
C     SIZE(5) = 0
C     SIZE(6) = 1
C
C     RETURN
C     END
C
C=====
C
C   Function:  OUTPUT
```

```
C
C   Abstract:
C     Perform output calculations for continuous
C     signals.
C
C=====
C   .. Parameters ..
C   SUBROUTINE OUTPUT(T, X, U, Y)
C   REAL*8          T
C   REAL*8          X(*), U(*), Y(*)

C   Y(1) = U(1) * 2.0

C   RETURN
C   END

C
C=====
C
C   Stubs for unused functions.
C
C=====

C   SUBROUTINE INITCOND(X0)
C   REAL*8          X0(*)
C --- Nothing to do.
C   RETURN
C   END

C   SUBROUTINE DERIVS(T, X, U, DX)
C   REAL*8          T, X(*), U(*), DX(*)
C --- Nothing to do.
C   RETURN
C   END

C   SUBROUTINE DSTATES(T, X, U, XNEW)
C   REAL*8          T, X(*), U(*), XNEW(*)
C --- Nothing to do.
C   RETURN
C   END
```

```
        SUBROUTINE DOUTPUT(T, X, U, Y)
        REAL*8          T, X(*), U(*), Y(*)
C --- Nothing to do.
        RETURN
        END

        SUBROUTINE TSAMPL(T, X, U, TS, OFFSET)
        REAL*8          T,TS,OFFSET,X(*),U(*)
C --- Nothing to do.
        RETURN
        END

        SUBROUTINE SINGUL(T, X, U, SING)
        REAL*8          T, X(*), U(*), SING(*)
C --- Nothing to do.
        RETURN
        END
```

A Level 1 S-function's input/output is limited to using the REAL*8 data type, (DOUBLE PRECISION), which is equivalent to a double in C. Of course, the internal calculations can use whatever data types you need.

To see how this S-function works, enter

```
sfcndemo_timestwo_for
```

at the MATLAB prompt and run the model.

Inline Code Generation Example

Real-Time Workshop users can use a sample block target file for `sfun_timestwo_for.mex` to generate code for `sfcndemo_timestwo_for`. If you want to learn how to inline your own Fortran Mex-file, see the example at `matlabroot/toolbox/simulink/blocks/tlc_c/sfun_timestwo_for.tlc`. Read the "Real-Time Workshop Target Language Compiler Reference Guide".

Creating Level 2 Fortran S-Functions

To use the features of a level 2 S-function with Fortran code, you must write a skeleton S-function in C that has code for interfacing to Simulink and also calls your Fortran code.

Using the C MEX S-function as a gateway is quite simple if you are writing the Fortran code from scratch. If instead your Fortran code already exists as a stand-alone simulation, there is some work to be done to identify parts of the code that need to be registered with Simulink, such as identifying continuous states if you are using variable-step solvers or getting rid of static variables if you want to have multiple copies of the S-function in a Simulink model (see “Porting Legacy Code” on page 6-15).

Template File

The file contains a template for creating a C MEX-file S-function that invokes a Fortran subroutine in its `mdlOutputs` method. It works with a simple Fortran subroutine if you modify the Fortran subroutine name in the code.

C/Fortran Interfacing Tips

The following are some tips for creating the C-to-Fortran gateway S-function.

MEX Environment

Remember that `mex -setup` needs to find both the MATLAB C and the Fortran compilers. If you install or change compilers, you must run `mex -setup`.

Test the installation and setup using sample Mex-files from MATLAB’s C and Fortran MEX examples in `matlabroot/extern/examples/mex`, as well as Simulink examples, which are located in `matlabroot/simulink/src`.

Compiler Compatibility

Your C and Fortran compilers need to use the same object format. If you use the compilers explicitly supported by the `mex` command this is not a problem. When you use the C gateway to Fortran, it is possible to use Fortran compilers not supported by the `mex` command, but only if the object file format

is compatible with the C compiler format. Common object formats include ELF and COFF.

The compiler must also be configurable so that the caller cleans up the stack instead of the callee. Compaq Visual Fortran (formerly known as Digital Fortran) is one compiler whose default stack cleanup is the callee.

Symbol Decorations

Symbol decorations can cause run-time errors. For example, g77 decorates subroutine names with a trailing underscore when in its default configuration. You can either recognize this and adjust the C function prototype or alter the Fortran compiler's name decoration policy via command-line switches, if the compiler supports this. See the Fortran compiler manual about altering symbol decoration policies.

If all else fails, use utilities such as `od` (octal dump) to display the symbol names. For example, the command

```
od -s 2 <file>
```

lists strings and symbols in binary (`.obj`) files.

These binary utilities can be obtained for Windows as well. MKS is one company that has commercial versions of powerful UNIX utilities, although most can also be obtained free on the Web. `hexdump` is another common program for viewing binary files. As an example, here is the output of

```
od -s 2 sfun_atmos_for.o
```

on Linux.

```
0000115 E€
0000136 E€
0000271 E€
0000467 E€@
0000530 E€
0000575 E€ E 5@
0001267 CfVC- :C
0001323 :|. -:8#8 Kw6
0001353 ?333@
```



```
0001364 333
0001414 01.01
0001425 GCC: (GNU) egcs-2.91.66 19990314/Linux
0001522 .symtab
0001532 .strtab
0001542 .shstrtab
0001554 .text
0001562 .rel.text
0001574 .data
0001602 .bss
0001607 .note
0001615 .comment
0003071 sfun_atmos_for.for
0003101 gcc2_compiled.
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_`, which 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 might not match C math library symbols. For example, A^B in Fortran calls 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, because `-I` 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#"$COMPFLAGS /iface:cref" -c sfun_atmos_sub.for  
-f ..\..\bin\win32\mexopts\df60opts.bat  
mex -v LINKFLAGS#"$LINKFLAGS dfor.lib dfconsol.lib dfport.lib  
/LIBPATH:$DF_ROOT\DF98\LIB" sfun_atmos.c sfun_atmos_sub.obj
```

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

CFortran

Or you can 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, you 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` 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 one-step-at-a-time fashion. If the code doesn't have any states, it can be called from `mdlOutputs()` and no `mdlDerivatives()` or `mdlUpdate()` method is required.

Code with States

If the code has states, you must decide whether 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. You do this 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 perform either 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., you must 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 Simulink, identify the main loop of time integration and remove 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. You manage I/O 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 you should use a subroutine, 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.

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#$COMPFLAGS /iface:cref" -c sfun_atmos_sub.for  
-f ..\..\bin\win32\mexopts\df60opts.bat  
>> mex -v LINKFLAGS#$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 compilers were installed separately (or aren't aware of each other), you might 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 the path to the `mex` process explicitly with the `-L` command. For example:

```
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`. Enter

```
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 the Simulink 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 so as not to “ratchet forward,” you might 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 (causing it to inadvertently advance time). See *matlabroot/simulink/src/sfuntmpl_gate_fortran.c* for an example that uses `DWork` vectors.

Sample Times

If the code has an implicit step size in its algorithm, coefficients, etc., ensure 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 to have multiple copies of this S-function used in one Simulink model, you need 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.c* 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 roundoff error when adding and subtracting flints. Using flint variables in DOUBLE PRECISION storage (with integer values) avoids roundoff 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 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. You must deal with these directly if you expect to run 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

The following sections explain how to use S-function callback methods to implement various block features.

Dialog Parameters (p. 7-2)	How to process parameters passed via the S-function block's dialog box.
Run-Time Parameters (p. 7-6)	How to create and use run-time parameters.
Creating Input and Output Ports (p. 7-10)	How to create input and output ports on a block.
Custom Data Types (p. 7-16)	How to create custom data types for the values of a block's signals and parameters.
Sample Times (p. 7-17)	How to specify the rate or rates at which your block operates.
Work Vectors (p. 7-31)	How to create and use work vectors.
Function-Call Subsystems (p. 7-36)	How to create a function-call subsystem.
Processing Frame-Based Signals (p. 7-39)	How to create an S-function that process frame-based signals.
Handling Errors (p. 7-41)	How to handle errors in an S-function.
S-Function Examples (p. 7-44)	Examples of S-functions.

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 `mdlInitializeSizes` 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 `mdlInitializeSizes` 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` macro returns a pointer to the `mxArray` containing the parameter. You can use `ssGetDTypeIdFromMxArray` 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.

Note You cannot use the Model Explorer, the S-Function block dialog box, or a mask to tune the parameters of a source S-function, i.e., an S-function that has outputs but no inputs, while a simulation is running. See "Changing Source Block Parameters" for more information.

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 enters 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 practice 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 that 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-function's `mdlProcessParameters` method to give the S-function a chance to process the parameters in some way. Simulink also invokes these methods when running in external mode, but it passes the unprocessed changes on to 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 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 parameters 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 parameters 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 needs 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 an *mdlRTW* 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.

Note The first four characters of the names of a block's run-time parameters must be unique. If they are not, Simulink signals an error. For example, trying to register a parameter named `param2` triggers an error if a parameter named `param1` already exists. This restriction allows Real-time Workshop to generate variable names that are unique within a pre-specified number of characters.

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 might not be the case. For example, an S-function might want to use a computed parameter whose value is a function of several dialog parameters. In such cases, the S-function might 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 Use `ssRegDlgParamAsRunTimeParam` to register a run-time parameter that corresponds to a single, untransformed dialog parameter or `ssSetRunTimeParamInfo` to set the attributes of a run-time parameter that corresponds to more than one dialog parameter or a transformed dialog parameter.

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 `mdlProcessParameters` method must update each parameter individually.

If a run-time parameter and its corresponding dialog parameter differ only in value, the method can use `ssUpdateRunTimeParamData` to update the run-time parameter. This function updates the data field in the parameter's attributes record, `ssParamRec`, with a new value. If the run-time parameter and the dialog parameter differ only in value and data type, the method can use `ssUpdateDlgParamAsRunTimeParam` to update the run-time parameter. Otherwise, the `mdlProcessParameters` 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 parameters.
- 3 Register the changes, using `ssUpdateRunTimeParamInfo`.

Tuning Runtime Parameters

Tuning a dialog parameter tunes the corresponding runtime parameter during simulation and in code generated from the model only if the dialog parameter meets the following conditions:

- The S-function marks the dialog parameter tunable, using `ssSetParameterTunable`.
- The dialog parameter is a MATLAB array of values of the standard data types supported by Simulink.
- The S-function has one or more input ports.

Note that you cannot tune a runtime parameter whose value is a cell array or structure.

Creating 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 7-11)

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 7-11).

- Whether the input port allows scalar expansion of inputs (see “Scalar Expansion of Inputs” on page 7-13)
- Whether the input port has direct feedthrough, using `ssSetInputPortDirectFeedThrough`

A port has direct feedthrough if the input is used in either the `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 `mdlOutputs` or `mdlGetTimeOfNextVarHit` routine. Setting the direct feedthrough flag to 0 tells Simulink that `u` is not used in either of these S-function routines. Violating this leads 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 is accessing invalid memory and Simulink crashes.

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, dimsInfo)
```

You can use this function 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 might want to set the dimensionality of the input port to match the dimensionality of the signal 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 an `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 an `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 of 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 output 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 `mdlSetOutputPortDimensionInfo` and `ssSetDefaultPortDimensionInfo` 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 ports 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 is 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 `mdlSetInputPortDimensionInfo`, `mdlSetOutputPortDimensionInfo`, and `mdlSetDefaultPortDimensionInfo`.

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

In the `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`. The Simulink default method uses the preceding option to allow or disallow scalar expansion for a block's input ports. If the preceding option is not set

by an S-function, Simulink assumes that 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, you must specify that the mask modifies the appearance of the block. To do this, execute the command

```
set_param('block','MaskSelfModifiable','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

This section explains how to specify the sample-time behavior of your function, e.g., whether it inherits its rates from the blocks that drive it or defines its own rates and, if it defines its own rates, what the rates are.

An S-function block can specify its rates (i.e., sample times) as

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

With block-based sample times, the S-function specifies a set of operating rates for the block as a whole during the initialization phase of the simulation. With port-based sample times, the S-function specifies a sample time for each input and output port individually during initialization. During the execution phase, with block-based sample times, the S-function processes all inputs and outputs each time a sample hit occurs for the block. By contrast, with port-based sample times, the block processes a particular port only when a sample hit occurs for that port.

For example, 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 process 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 might 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 specify only one block-based sample time. Advanced S-functions might 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 the output elements that are performing function calls. See `matlabroot/simulink/src/sfun_fcncall.c` for an example.

You specify the sample times as pairs [*sample_time*, *offset_time*], using these macros

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

where *sampleTimePairIndex* starts at 0.

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

```
[CONTINUOUS_SAMPLE_TIME, 0.0 ]
[CONTINUOUS_SAMPLE_TIME, FIXED_IN_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_IN_MINOR_STEP_OFFSET]
```

Note If your S-function inherits its sample time, you should specify whether it is safe to use the S-function in a submodel, i.e., a model referenced by another model. See “Specifying Model Reference Sample Time Inheritance” on page 7-29 for more information.

The following guidelines might help 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_IN_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 <= offset < 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, 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, 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 `mdlOutputs` or `mdlUpdate`), use the `ssIsSampleHit` or `ssIsContinuousTask` 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 get incorrect results if you use `ssIsSampleHit(S,0,tid)`.

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

```
}

```

If, for example, you wanted to determine whether the third (discrete) task has a hit, you would use the following code fragment:

```
if (ssIsSampleHit(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. */

```

Specifying Port-Based Sample Times

If you want your S-function to use port-based sample times, you must specify the number of sample times as port-based in the S-function's mdlInitializeSizes method:

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

You must also specify the sample time of each input and output port in the S-function's mdlInitializeSizes method, using the following macros

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

```

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

For any given port, you can specify

- A specific sample time and period

For example, the following code sets the sample time of the S-function's first input port to every 0.1 s starting with the simulation start time.

```
ssSetInputPortSampleTime(S, 0, 0.1);  
ssSetInputPortOffsetTime(S, 0, 0);
```

- Inherited sample time, i.e., the port inherits its sample time from the port to which it is connected (see “Specifying Inherited Sample Time for a Port” on page 7-22)
- Constant sample time, i.e., the port's input or output never changes (see “Specifying Constant Sample Time for a Port” on page 7-23)

Note To be usable in a triggered subsystem, all of your S-function's ports must have either inherited or constant sample time (see “Configuring Port-Based Sample Times for Use in Triggered Subsystems” on page 7-24).

Specifying Inherited Sample Time for a Port

To specify that a port's sample time is inherited, the `mdlInitializeSizes` method should set its period to -1 and its offset to 0. For example, the following code specifies inherited sample time for the S-function's first input port:

```
ssSetInputPortSampleTime(S, 0, -1);  
ssSetInputPortOffsetTime(S, 0, 0);
```

When you specify port-based sample times, Simulink calls `mdlSetInputPortSampleTime` and `mdlSetOutputPortSampleTime` to determine the rates of inherited signals.

Once all rates have been determined, Simulink calls `mdlInitializeSampleTimes`. Even though there is no need to initialize port-based sample times at this point, Simulink invokes this method to give your S-function an opportunity to configure function-call connections. Your S-function must thus provide an implementation for this method regardless of whether it uses port-based sample times or function-call connections. Although you can provide an empty implementation, you might want to use it to check the appropriateness of the sample times that the block inherited during sample time propagation.

Note If you specify that your S-function's ports inherit their sample time, you should also specify whether it is safe to use the S-function in a submodel, i.e., a model referenced by another model. See "Specifying Model Reference Sample Time Inheritance" on page 7-29 for more information.

Specifying Constant Sample Time for a Port

If your S-function uses port-based sample times, it can specify that any of its ports has a constant sample time. This means that the signal entering or leaving the port never changes from its initial value at the start of the simulation.

Before specifying constant sample time for an output port whose output depends on the S-function's parameters, the S-function should use `ssGetInlineParameters` to check whether the user has specified the **Inline parameters** option on the **Optimization** pane of the **Configuration parameters** dialog box. If the user has not checked this option, it is possible for the user to change the values the S-function's parameters and hence its outputs during the simulation. In this case, the S-function should not specify a constant sample time for any ports whose outputs depend on the S-function's parameters.

To specify constant sample time for a port, the S-function must perform the following tasks

- Tell Simulink that it supports constant port sample times in its `mdlInitializeSizes` method:

```
ssSetOptions(S, SS_OPTION_ALLOW_CONSTANT_PORT_SAMPLE_TIME);
```

Note By setting this option, your S-function is in effect telling Simulink that all of its ports support a constant sample time including ports that inherit their sample times from other blocks. If any of the S-function's inherited sample time ports cannot have a constant sample time, your S-function's `mdlSetInputPortSampleTime` and `mdlSetOutputPortSampleTime` methods must check whether that port has inherited a constant sample time. If the port has inherited a constant sample time, your S-function should throw an error.

- Set the port's period to `inf` and its offset to 0, e.g.,

```
ssSetInputPortSampleTime(S, 0, mxGetInf());  
ssSetInputPortOffsetTime(S, 0, 0);
```

- Check in `mdlOutputs` whether the method's `tid` argument equals `CONSTANT_TID` and if so, set the value of the port's output if it is an output port.

See `sfun_port_constant.c`, the source file for the `sfcdemo_port_constant` demo, for an example of how to create ports with a constant sample time.

Configuring Port-Based Sample Times for Use in Triggered Subsystems

To be usable in a triggered subsystem, your port-based sample time S-function must perform the following tasks.

- Tell Simulink in its `mdlInitializeSizes` method that it can run in a triggered subsystem:

```
ssSetOptions(S,  
SS_OPTION_ALLOW_PORT_BASED_SAMPLE_TIME_IN_TRIGSS);
```

- Set all of its ports to have either inherited or constant sample time in its `mdlInitializeSizes` method.

- Handle inheritance of a triggered sample time in `mdlSetInputPortSampleTime` and `mdlSetOutputPortSampleTime` methods as follows.

If the S-function resides in a triggered subsystem, Simulink invokes either `mdlSetInputPortSampleTime` or `mdlSetOutputPortSampleTime` during sample time propagation. Whichever method is called must set the sample time and offset of the port for which it is called to `INHERITED_SAMPLE_TIME (-1)`, e.g.,

```
ssSetInputPortSampleTime(S, 0, INHERITED_SAMPLE_TIME);
ssSetInputPortOffsetTime(S, 0, INHERITED_SAMPLE_TIME);
```

Setting a port's sample time and offset both to `INHERITED_SAMPLE_TIME` indicates that the sample time of the port is triggered, i.e., it produces an output or accepts an input only when the subsystem in which it resides is triggered. The method must also set the sample times and offsets of all of the S-function's other input and output ports to have either triggered or constant sample time, whichever is appropriate.

There is no way for an S-function residing in a triggered subsystem to predict whether Simulink will call `mdlSetInputPortSampleTime` or `mdlSetOutputPortSampleTime` to set its port sample times. For this reason, both methods must be able to set the sample times correctly.

- In `mdlUpdate` and `mdlOutputs`, `usessGetPortBasedSampleTimeBlockIsTriggered` to check whether the S-function resides in a triggered subsystem and if so, use appropriate algorithms for computing its states and outputs.

See `sfun_port_triggered.c`, the source file for the `sfcndemo_port_triggered` demo, for an example of how to create ports with a constant sample time.

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 `ssSetOptions`, 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 periods and offsets 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 `mdlOutputs` and `mdlUpdate` functions with a statement that determines whether a sample hit has occurred. The `ssIsSampleHit` macro determines whether the current time is a sample hit for a specified sample time. The macro has this syntax:

```
ssIsSampleHit(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 `mdlOutputs` and `mdlUpdate` functions).

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 `mdlUpdate` function, the following statement encapsulates the code that defines the behavior for the sample time of 0.75 second.

```
if (ssIsSampleHit(S, 1, tid)) {
```

```
}
```

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

Example of Defining a Sample Time for a Continuous Block

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

```
/* 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 of 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` routine 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 (ssIsSampleHit(S, 0, tid) {
    if (ssIsSpecialSampleHit(S, 0, 1, tid) {
        /* Transfer input to output memory. */
        ...
    }
}

if (ssIsSampleHit(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.

Specifying Model Reference Sample Time Inheritance

If your S-function inherits its sample times from the blocks that drive it, it should specify whether it is safe to use the S-function in a submodel. It is safe only if the S-function's output does not depend on the inherited sample time. If the S-function's output does not depend on its inherited sample time, use the `ssSetModelReferenceSampleTimeInheritanceRule` macro to set the S-function's sample time inheritance rule to `USE_DEFAULT_FOR_DISCRETE_INHERITANCE`. Otherwise, set the rule to `DISALLOW_SAMPLE_TIME_INHERITANCE`. Specifying the inheritance rule allows Simulink to disallow sample-time inheritance for submodels that include S-functions whose outputs depend on their inherited sample time and thereby avoid inadvertent simulation errors.

Note If your S-function does not set this flag, Simulink assumes that it does not preclude a submodel containing it from inheriting a sample time. However, Simulink optionally warns the user that the submodel contains S-functions that do not specify a sample-time inheritance rule (see "Model Referencing Options" in the online Simulink help).

If you are uncertain whether an existing S-function's output depends on its inherited sample time, check whether it invokes any of the following C macros:

- `ssGetSampleTime`
- `ssGetInputPortSampleTime`
- `ssGetOutputPortSampleTime`
- `ssGetInputPortOffsetTime`
- `ssGetOutputPortOffsetTime`
- `ssGetSampleTimePtr`
- `ssGetInputPortSampleTimeIndex`
- `ssGetOutputPortSampleTimeIndex`
- `ssGetSampleTimeTaskID`
- `ssGetSampleTimeTaskIDPtr`

or TLC functions:

- LibBlockSampleTime
- CompiledModel.SampleTime
- LibBlockInputSignalSampleTime
- LibBlockInputSignalOffsetTime
- LibBlockOutputSignalSampleTime
- LibBlockOutputSignalOffsetTime

If the S-function does not invoke any of these macros or functions, its output does not depend on its inherited sample time and hence it is safe to use in submodels that inherit their sample time.

Sample-Time Inheritance Rule Example

As an example of an S-function that precludes a submodel from inheriting its sample time, consider an S-function that has the following mdlOutputs method:

```
static void mdlOutputs(SimStruct *S, int_T tid) {
    const real_T *u = (const real_T*)
        ssGetInputPortSignal(S,0);
    real_T      *y = ssGetOutputPortSignal(S,0);
    y[0] = ssGetSampleTime(S,tid) * u[0];
}
```

This output of this S-function is its inherited sample time, hence its output depends on its inherited sample time, and hence it is unsafe to use in a submodel. For this reason, this S-function should specify its model reference inheritance rule as follows:

```
ssSetModelReferenceSampleTimeInheritanceRule
(S, DISALLOW_SAMPLE_TIME_INHERITANCE);
```

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 *reentrancy*.

You can create an S-function that is reentrant 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 `mdlInitializeSizes`.

Macros Used in Specifying Vector Widths

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

Specify vector widths in `mdlInitializeSizes`. 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 is 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 [Macros Used in Specifying Vector Widths](#) on page 7-32, to set the sizes of the work vectors explicitly. `mdlSetWorkWidths` also allows you to set your work vector lengths as functions 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 the Simulink 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 are not 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 `mdlOutputs` routine should operate when the solvers are homing 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.

Example Involving a Pointer Work Vector

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

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

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

The following code 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`, 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 you are creating an S-function, the available work vectors might not provide enough capability. In this case, you need to allocate memory for each instance of your S-function. The standard MATLAB API memory allocation routines `mxMalloc` and `mxFree` should not be used with C MEX S-functions, because 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` library routines `calloc` and `free`. 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.

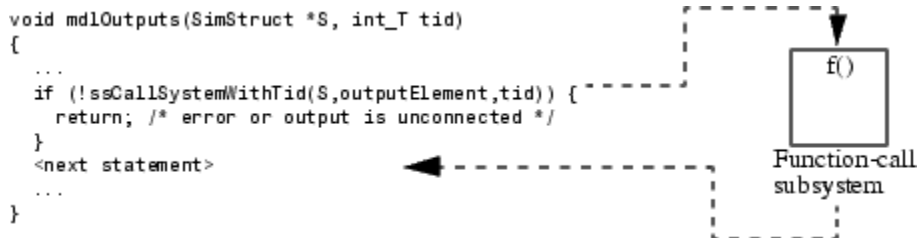
Function-Call Subsystems

You can create a triggered subsystem whose execution is determined by logic internal to an S-function instead of by the value of a signal. A subsystem so configured is called a *function-call subsystem*. To implement a function-call subsystem:

- In the Trigger block, select **function-call** as the **Trigger type** parameter.
- In the S-function, use the `ssEnableSystemWithTid` and `ssDisableSystemWithTid` to enable or disable the triggered subsystem and the `ssCallSystemWithTid` macro to call the triggered subsystem.
- In the model, connect the S-Function block output directly to the trigger port.

Note Function-call connections can only be performed on the first output port.

Function-call subsystems are not executed directly by Simulink; rather, the S-function determines when to execute the subsystem. When the subsystem completes execution, control returns to the S-function. This figure illustrates the interaction between a function-call subsystem and an S-function.



In this figure, `ssCallSystemWithTid` executes the function-call subsystem that is connected to the first output port element. `ssCallSystemWithTid` returns 0 if an error occurs while executing the function-call subsystem or if the output is unconnected. After the function-call subsystem executes, control is returned to your S-function.

Function-call subsystems can only be connected to S-functions that have been properly configured to accept them.

To configure an S-function to call a function-call subsystem:

- Specify the elements that are to execute the function-call system in `mdlInitializeSampleTimes`. For example:

```
ssSetCallSystemOutput(S,0); /* call on 1st element */
ssSetCallSystemOutput(S,2); /* call on 3rd element */
```

- Specify in `mdlInitializeSampleTimes` whether you want the S-function to be able to enable or disable the function-call subsystem. For example:

```
ssSetExplicitFCSSCtrl(SimStruct *S, TRUE);
```

- Execute the subsystem in the appropriate `mdlOutputs` or `mdlUpdate` S-function routine. For example:

```
static void mdlOutputs(...)
{
    if (((int)*uPtrs[0]) % 2 == 1) {
        if (!ssCallSystemWithTid(S,0,tid)) {
            /* Error occurred, which will be reported by Simulink */
            return;
        }
    } else {
        if (!ssCallSystemWithTid(S,2,tid)) {
            /* Error occurred, which will be reported by Simulink */
            return;
        }
    }
    ...
}
```

See `simulink/src/sfun_fcncall.c` for an example.

Function-call subsystems are a powerful modeling construct. You can configure Stateflow® blocks to execute function-call subsystems, thereby

extending the capabilities of the blocks. For more information on their use in Stateflow, see the Stateflow documentation.

Processing Frame-Based Signals

This section explains how to create an S-function that accepts and/or produces frame-based signals. See "Frame-Based Signals" in the "Working with Signals" section of the Signal Processing Blockset documentation for a comprehensive discussion of the use of frame-based signals in Simulink models.

Note Simulating a model containing the S-function that you develop requires a Signal Processing Blockset license.

To accept or produce frame-based signals, an S-function must perform the following tasks:

- The S-function's `mdlInitializeSizes` callback method must set the port frame status to `FRAME_YES`, `FRAME_NO`, or `FRAME_INHERITED` for each of the S-function's I/O ports, using the `ssSetInputPortFrameData` and `ssSetOutputPortFrameData` functions.
- If the frame status of any of the S-function's input ports is inherited, the S-function should define a `mdlSetInputPortFrameData` callback method. Simulink passes the frame status that it assigns to the port, based on frame signal propagation rules, as an argument to this callback method. The callback method should in turn use the `ssSetInputPortFrameData` function to set the port to the assigned status if it is acceptable or signal an error if it is not. If the frame status of other ports of the S-function depend on the status inherited by one of its input ports, the callback method can also use `ssSetInputPortFrameData` to set the frame status of the other ports based on the status that the input port inherits.
- The S-function's `mdlOutputs` method should include code to process the signals.
- The S-function's should specify the dimensions of the signals that its frame-based ports accept or produce in its `mdlInitializeSizes` or `mdlSetInputPortDimensionInfo` and `mdlSetOutputPortDimensionInfo` callback methods. Note that frame-based signals must be dimensioned as 2-D arrays.

See the frame-based A/D converter S-function example (`sfun_frmad.c`) for an example of how to create a frame-based S-function.

Handling Errors

When working with S-functions, it is important to handle unexpected events such as invalid parameter values correctly.

If your S-function has parameters whose contents you need to validate, use the following technique to report errors encountered.

```
ssSetErrorStatus(S,"error encountered due to ...");  
return;
```

Note that the second argument to `ssSetErrorStatus` must be persistent memory. It cannot be a local variable in your procedure. For example, the following causes unpredictable errors.

```
mdlOutputs()  
{  
    char msg[256]; /* ILLEGAL: should be "static char msg[256];" */  
    sprintf(msg,"Error due to %s", string);  
    ssSetErrorStatus(S,msg);  
    return;  
}
```

Because `ssSetErrorStatus` does not generate exceptions, using it to report errors in your S-function is preferable to using `mexErrMsgTxt`. The `mexErrMsgTxt` function uses exception handling to terminate S-function execution and return control to Simulink. To support exception handling in S-functions, Simulink must set up exception handlers prior to each S-function invocation. This introduces overhead into simulation.

Exception Free Code

You can avoid this overhead by ensuring that your S-function contains entirely *exception free code*. Exception free code refers to code that never long-jumps. Your S-function is not exception free if it contains any routine that, when called, has the potential of long-jumping. For example, `mexErrMsgTxt` throws an exception (i.e., long-jumps) when called, thus ending execution of your S-function. Using `mxCalloc` can cause unpredictable results in the event of a memory allocation error, because `mxCalloc` long-jumps. If memory allocation

is needed, use the `stdlib.h` `calloc` routine directly and perform your own error handling.

If you do not call `mexErrMsgTxt` or other API routines that cause exceptions, use the `SS_OPTION_EXCEPTION_FREE_CODE` S-function option. You do this by issuing the following command in the `mdlInitializeSizes` function.

```
ssSetOptions(S, SS_OPTION_EXCEPTION_FREE_CODE);
```

Setting this option increases the performance of your S-function by allowing Simulink to bypass the exception-handling setup that is usually performed prior to each S-function invocation. You must take extreme care to verify that your code is exception free when using `SS_OPTION_EXCEPTION_FREE_CODE`. If your S-function generates an exception when this option is set, unpredictable results occur.

All `mex*` routines have the potential of long-jumping. Several `mex*` routines also have the potential of long-jumping. To avoid any difficulties, use only the API routines that retrieve a pointer or determine the size of parameters. For example, the following never throw an exception: `mxGetPr`, `mxGetData`, `mxGetNumberOfDimensions`, `mxGetM`, `mxGetN`, and `mxGetNumberOfElements`.

Code in *run-time routines* can also throw exceptions. Run-time routines refer to certain S-function routines that Simulink calls during the simulation loop (see “How Simulink Interacts with C S-Functions” on page 3-39). The run-time routines include

- `mdlGetTimeOfNextVarHit`
- `mdlOutputs`
- `mdlUpdate`
- `mdlDerivatives`

If all run-time routines within your S-function are exception free, you can use this option:

```
ssSetOptions(S, SS_OPTION_RUNTIME_EXCEPTION_FREE_CODE);
```

The other routines in your S-function do not have to be exception free.

ssSetErrorStatus Termination Criteria

When you call `ssSetErrorStatus` and return from your S-function, Simulink stops the simulation and posts the error. To determine how the simulation shuts down, refer to the flow chart figure on “How Simulink Interacts with C S-Functions” on page 3-39. If `ssSetErrorStatus` is called prior to `mdlStart`, no other S-function routine is called. If `ssSetErrorStatus` is called in `mdlStart` or later, `mdlTerminate` is called.

Checking Array Bounds

If your S-function causes otherwise inexplicable errors, the reason might be that the S-function is writing beyond its assigned areas in memory. You can verify this possibility by enabling the Simulink array bounds checking feature. This feature detects any attempt by an S-Function block to write beyond the areas assigned to it for the following types of block data:

- Work vectors (R, I, P, D, and mode)
- States (continuous and discrete)
- Outputs

To enable array bounds checking, select **warning** or **error** from the **Array bounds exceeded** options list on the **Diagnostics** pane of the **Configuration Parameters** dialog box or enter the following command at the MATLAB command line.

```
set_param(modelName, 'ArrayBoundsChecking', 'none' | 'warning' |  
'error')
```

S-Function Examples

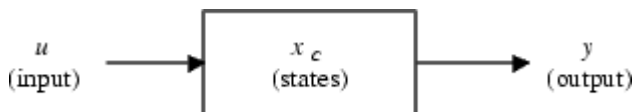
Most S-Function blocks require the handling of states, continuous or discrete. The following sections discuss common types of systems that you can model in Simulink with S-functions:

- Continuous state
- Discrete state
- Hybrid
- Variable step sample time
- Zero crossings
- Time-varying continuous transfer function

All examples are based on the C MEX-file S-function template `sfuntmpl_basic.c` and on `sfuntmpl_doc.c`, which contains a discussion of the S-function template.

Example of a Continuous State S-Function

The `matlabroot/simulink/src/csfunc.c` example shows how to model a continuous system with states in a C MEX S-function. In continuous state integration, there is a set of states that the Simulink solvers integrate using the following equations.



$$y = f_0(t, x_c, u) \quad (\text{output})$$

$$\dot{x}_c = f_d(t, x_c, u) \quad (\text{derivative})$$

S-functions that contain continuous states implement a state-space equation. The output portion is placed in `mdlOutputs` and the derivative portion in `mdlDerivatives`. To visualize how the integration works, refer to the flowchart in “How Simulink Interacts with C S-Functions” on page 3-39. The output equation above corresponds to the `mdlOutputs` in the major time

step. Next, the example enters the integration section of the flowchart. Here Simulink performs a number of minor time steps during which it calls `mdlOutputs` and `mdlDerivatives`. Each of these pairs of calls is referred to as an *integration stage*. The integration returns with the continuous states updated and the simulation time moved forward. Time is moved forward as far as possible, providing that error tolerances in the state are met. The maximum time step is subject to constraints of discrete events such as the actual simulation stop time and the user-imposed limit.

Note that `csfunc.c` specifies that the input port has direct feedthrough. This is because matrix `D` is initialized to a nonzero matrix. If `D` is set equal to a zero matrix in the state-space representation, the input signal isn't used in `mdlOutputs`. In this case, the direct feedthrough can be set to 0, which indicates that `csfunc.c` does not require the input signal when executing `mdlOutputs`.

matlabroot/simulink/src/csfunc.c

```

/* File      : csfunc.c
 * Abstract:
 *
 *      Example C-file S-function for defining a continuous system.
 *
 *       $x' = Ax + Bu$ 
 *       $y = Cx + Du$ 
 *
 *      For more details about S-functions, see simulink/src/sfuntmpl_doc.c.
 *
 *      Copyright 1990-2000 The MathWorks, Inc.
 */

#define S_FUNCTION_NAME csfunc
#define S_FUNCTION_LEVEL 2

#include "simstruc.h"

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

static real_T A[2][2]={ { -0.09, -0.01 } ,
                        { 1 , 0 }

```

```

        };

static real_T B[2][2]={ { 1   , -7   } ,
                        { 0   , -2   }
                        };

static real_T C[2][2]={ { 0   , 2    } ,
                        { 1   , -5   }
                        };

static real_T D[2][2]={ { -3   , 0    } ,
                        { 1   , 0    }
                        };

/*=====
 * 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)
{
    ssSetNumSFcnParams(S, 0); /* Number of expected parameters */
    if (ssGetNumSFcnParams(S) != ssGetSFcnParamsCount(S)) {
        return; /* Parameter mismatch will be reported by Simulink */
    }

    ssSetNumContStates(S, 2);
    ssSetNumDiscStates(S, 0);

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

    if (!ssSetNumOutputPorts(S, 1)) return;
    ssSetOutputPortWidth(S, 0, 2);

```

```
    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.c */
    ssSetOptions(S, SS_OPTION_EXCEPTION_FREE_CODE);
}

/* Function: mdlInitializeSampleTimes =====
 * Abstract:
 *   Specify that we have a continuous sample time.
 */
static void mdlInitializeSampleTimes(SimStruct *S)
{
    ssSetSampleTime(S, 0, CONTINUOUS_SAMPLE_TIME);
    ssSetOffsetTime(S, 0, 0.0);
}

#define MDL_INITIALIZE_CONDITIONS
/* Function: mdlInitializeConditions =====
 * Abstract:
 *   Initialize both continuous states to zero.
 */
static void mdlInitializeConditions(SimStruct *S)
{
    real_T *x0 = ssGetContStates(S);
    int_T lp;

    for (lp=0;lp<2;lp++) {
        *x0++=0.0;
    }
}
}
```

```

/* Function: mdlOutputs =====
 * Abstract:
 *      y = Cx + Du
 */
static void mdlOutputs(SimStruct *S, int_T tid)
{
    real_T      *y      = ssGetOutputPortRealSignal(S,0);
    real_T      *x      = ssGetContStates(S);
    InputRealPtrsType uPtrs = ssGetInputPortRealSignalPtrs(S,0);

    UNUSED_ARG(tid); /* not used in single tasking mode */

    /* y=Cx+Du */
    y[0]=C[0][0]*x[0]+C[0][1]*x[1]+D[0][0]*U(0)+D[0][1]*U(1);
    y[1]=C[1][0]*x[0]+C[1][1]*x[1]+D[1][0]*U(0)+D[1][1]*U(1);
}

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

    /* xdot=Ax+Bu */
    dx[0]=A[0][0]*x[0]+A[0][1]*x[1]+B[0][0]*U(0)+B[0][1]*U(1);
    dx[1]=A[1][0]*x[0]+A[1][1]*x[1]+B[1][0]*U(0)+B[1][1]*U(1);
}

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

```



```

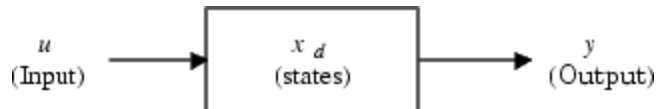
*/
static void mdlTerminate(SimStruct *S)
{
    UNUSED_ARG(S); /* unused input argument */
}

#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 of a Discrete State S-Function

The `matlabroot/simulink/src/dsfunc.c` example shows how to model a discrete system in a C MEX S-function. Discrete systems can be modeled by the following set of equations.



$$y = f_0(t, x_d, u) \quad (\text{Output})$$

$$x_{d+1} = f_u(t, x_d, u) \quad (\text{Update})$$

`dsfunc.c` implements a discrete state-space equation. The output portion is placed in `mdlOutputs` and the update portion in `mdlUpdate`. To visualize how the simulation works, refer to the flowchart in “How Simulink Interacts with C S-Functions” on page 3-39. The output equation above corresponds to the `mdlOutputs` in the major time step. The preceding update equation corresponds to the `mdlUpdate` in the major time step. If your model does not contain continuous elements, the integration phase is skipped and time is moved forward to the next discrete sample hit.

`matlabroot/simulink/src/dsfunc.c`

```

/* File      : dsfunc.c
 * Abstract:

```

```
*
*   Example C-file S-function for defining a discrete system.
*
*    $x(n+1) = Ax(n) + Bu(n)$ 
*    $y(n) = Cx(n) + Du(n)$ 
*
*   For more details about S-functions, see simulink/src/sfuntmpl_doc.c.
*
*   Copyright 1990-2000 The MathWorks, Inc.
*/

#define S_FUNCTION_NAME dsfunc
#define S_FUNCTION_LEVEL 2

#include "simstruc.h"

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

static real_T A[2][2]={ { -1.3839, -0.5097 } ,
                        { 1      , 0      }
                      };

static real_T B[2][2]={ { -2.5559, 0      } ,
                        { 0      , 4.2382 }
                      };

static real_T C[2][2]={ { 0      , 2.0761 } ,
                        { 0      , 7.7891 }
                      };

static real_T D[2][2]={ { -0.8141, -2.9334 } ,
                        { 1.2426, 0      }
                      };

/*=====
 * 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)
{
    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, 2);

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

    if (!ssSetNumOutputPorts(S, 1)) return;
    ssSetOutputPortWidth(S, 0, 2);

    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.c */
    ssSetOptions(S, SS_OPTION_EXCEPTION_FREE_CODE);
}

/* Function: mdlInitializeSampleTimes =====
* Abstract:
*   Specifiy that we inherit our sample time from the driving block.
*/
static void mdlInitializeSampleTimes(SimStruct *S)
{

```

```

        ssSetSampleTime(S, 0, 1.0);
        ssSetOffsetTime(S, 0, 0.0);
    }

#define MDL_INITIALIZE_CONDITIONS
/* Function: mdlInitializeConditions =====
 * Abstract:
 *   Initialize both discrete states to one.
 */
static void mdlInitializeConditions(SimStruct *S)
{
    real_T *x0 = ssGetRealDiscStates(S);
    int_T lp;

    for (lp=0;lp<2;lp++) {
        *x0+=1.0;
    }
}

/* Function: mdlOutputs =====
 * Abstract:
 *   y = Cx + Du
 */
static void mdlOutputs(SimStruct *S, int_T tid)
{
    real_T      *y      = ssGetOutputPortRealSignal(S,0);
    real_T      *x      = ssGetRealDiscStates(S);
    InputRealPtrsType uPtrs = ssGetInputPortRealSignalPtrs(S,0);

    UNUSED_ARG(tid); /* not used in single tasking mode */

    /* y=Cx+Du */
    y[0]=C[0][0]*x[0]+C[0][1]*x[1]+D[0][0]*U(0)+D[0][1]*U(1);
    y[1]=C[1][0]*x[0]+C[1][1]*x[1]+D[1][0]*U(0)+D[1][1]*U(1);
}

```

```

#define MDL_UPDATE
/* Function: mdlUpdate =====
 * Abstract:
 *      xdot = Ax + Bu
 */
static void mdlUpdate(SimStruct *S, int_T tid)
{
    real_T      tempX[2] = {0.0, 0.0};
    real_T      *x      = ssGetRealDiscStates(S);
    InputRealPtrsType uPtrs = ssGetInputPortRealSignalPtrs(S,0);

    UNUSED_ARG(tid); /* not used in single tasking mode */

    /* xdot=Ax+Bu */
    tempX[0]=A[0][0]*x[0]+A[0][1]*x[1]+B[0][0]*U(0)+B[0][1]*U(1);
    tempX[1]=A[1][0]*x[0]+A[1][1]*x[1]+B[1][0]*U(0)+B[1][1]*U(1);

    x[0]=tempX[0];
    x[1]=tempX[1];
}

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

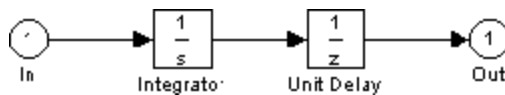
#ifdef MATLAB_MEX_FILE /* Is this file being compiled as a MEX-file? */
#include "simulink.c" /* MEX-file interface mechanism */
#else
#include "cg_sfun.h" /* Code generation registration function */
#endif

```

Example of a Hybrid System S-Function

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 `mdlDerivatives` 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.

Because 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 S-function illustrating multiple sample times by implementing
 *      integrator -> ZOH(Ts=1second) -> UnitDelay(Ts=1second)
 *      with an initial condition of 1.
 *      (e.g. an integrator followed by unit delay operation).
 *
 *      For more details about S-functions, see simulink/src/sfuntmpl_doc.c
 *
 *      Copyright 1990-2000 The MathWorks, Inc.
 */

#define S_FUNCTION_NAME mixedm
#define S_FUNCTION_LEVEL 2
  
```

```

#include "simstruc.h"

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

/*=====
 * 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)
{
    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);
    ssSetNumRWork(S, 1); /* for zoh output feeding the delay operator */

    if (!ssSetNumInputPorts(S, 1)) return;
    ssSetInputPortWidth(S, 0, 1);
    ssSetInputPortDirectFeedThrough(S, 0, 1);
    ssSetInputPortSampleTime(S, 0, CONTINUOUS_SAMPLE_TIME);
    ssSetInputPortOffsetTime(S, 0, 0.0);

    if (!ssSetNumOutputPorts(S, 1)) return;
    ssSetOutputPortWidth(S, 0, 1);
    ssSetOutputPortSampleTime(S, 0, 1.0);
    ssSetOutputPortOffsetTime(S, 0, 0.0);

    ssSetNumSampleTimes(S, 2);

    /* Take care when specifying exception free code - see sfuntmpl_doc.c. */
    ssSetOptions(S, (SS_OPTION_EXCEPTION_FREE_CODE |
                    SS_OPTION_PORT_SAMPLE_TIMES_ASSIGNED));
}

```

```
    } /* end mdlInitializeSizes */

/* 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);
    ssSetOffsetTime(S, 0, 0.0);

    ssSetSampleTime(S, 1, 1.0);
    ssSetOffsetTime(S, 1, 0.0);
} /* end mdlInitializeSampleTimes */

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

    xC0[0] = 1.0;
    xD0[0] = 1.0;
} /* end mdlInitializeConditions */

/* Function: mdlOutputs =====
 * Abstract:
```



```

*      y = xD, and update the zoh internal output.
*/
static void mdlOutputs(SimStruct *S, int_T tid)
{
    /* update the internal "zoh" output */
    if (ssIsContinuousTask(S, tid) ) {
        if (ssIsSpecialSampleHit(S, 1, 0, tid) ) {
            real_T *zoh = ssGetRWork(S);
            real_T *xC = ssGetContStates(S);
            *zoh = *xC;
        }
    }

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

} /* end mdlOutputs */

#define MDL_UPDATE
/* Function: mdlUpdate =====
* Abstract:
*      xD = xC
*/
static void mdlUpdate(SimStruct *S, int_T tid)
{
    UNUSED_ARG(tid); /* not used in single tasking mode */

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

```

```
    } /* end mdlUpdate */

#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);
} /* end mdlDerivatives */

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

#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 of a 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.

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 the 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 the 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.c.
```

```
*
* Copyright 1990-2000 The MathWorks, Inc.
*/

#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, 1);

    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);

    if (ssGetSimMode(S) == SS_SIMMODE_RTWGEN && !ssIsVariableStepSolver(S)) {
        ssSetErrorStatus(S, "S-function vsfunc.c cannot be used with RTW "
            "and Fixed-Step Solvers because it contains variable"
            " sample time");
    }

    /* Take care when specifying exception free code - see sfuntmpl_doc.c */
    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 = ssGetRealDiscStates(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 = ssGetRealDiscStates(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      = ssGetRealDiscStates(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 of a 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 which 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 support helps the variable step
* solvers locate the exact instants when behavior switches from one equation
* to another.
*
* Copyright 1990-2000 The MathWorks, Inc.
*/

#define S_FUNCTION_NAME  sfun_zc_sat
#define S_FUNCTION_LEVEL 2

#include "simstruc.h"

/*****
 * 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
```



```

/*
 * Make access to mxArray pointers for parameters more readable.
 */
#define P_PAR_UPPER_LIMIT ( ssGetSFcnParam(S,I_PAR_UPPER_LIMIT) )
#define P_PAR_LOWER_LIMIT ( ssGetSFcnParam(S,I_PAR_LOWER_LIMIT) )

#define MDL_CHECK_PARAMETERS
#if defined(MDL_CHECK_PARAMETERS) && defined(MATLAB_MEX_FILE)

/* Function: mdlCheckParameters =====
 * Abstract:
 * Check that parameter choices are allowable.
 */
static void mdlCheckParameters(SimStruct *S)
{
    int_T      i;
    int_T      numUpperLimit;
    int_T      numLowerLimit;
    const char *msg = NULL;

    /*
     * check parameter basics
     */
    for ( i = 0; i < N_PAR; i++ ) {
        if ( mxIsEmpty(  ssGetSFcnParam(S,i) ) ||
             mxIsSparse( ssGetSFcnParam(S,i) ) ||
             mxIsComplex( ssGetSFcnParam(S,i) ) ||
             !mxIsNumeric( ssGetSFcnParam(S,i) ) ) {
            msg = "Parameters must be real vectors.";
            goto EXIT_POINT;
        }
    }

    /*
     * Check sizes of parameters.
     */
    numUpperLimit = mxGetNumberOfElements( P_PAR_UPPER_LIMIT );
    numLowerLimit = mxGetNumberOfElements( P_PAR_LOWER_LIMIT );

```

```

        if ( ( numUpperLimit != 1          ) &&
            ( numLowerLimit != 1         ) &&
            ( numUpperLimit != numLowerLimit ) ) {
            msg = "Number of input and output values must be equal.";
            goto EXIT_POINT;
        }

        /*
         * Error exit point
         */
EXIT_POINT:
        if (msg != NULL) {
            ssSetErrorStatus(S, msg);
        }
    }
#endif /* MDL_CHECK_PARAMETERS */

/* Function: mdlInitializeSizes =====
 * Abstract:
 *   Initialize the sizes array.
 */
static void mdlInitializeSizes(SimStruct *S)
{
    int_T numUpperLimit, numLowerLimit, maxNumLimit;

    /*
     * Set and Check parameter count
     */
    ssSetNumSFcnParams(S, N_PAR);

#ifdef MATLAB_MEX_FILE
    if (ssGetNumSFcnParams(S) == ssGetSFcnParamsCount(S)) {
        mdlCheckParameters(S);
        if (ssGetErrorStatus(S) != NULL) {
            return;
        }
    } else {

```

```
        return; /* Parameter mismatch will be reported by Simulink */
    }
#endif

    /*
    * Get parameter size info.
    */
    numUpperLimit = mxGetNumberOfElements( P_PAR_UPPER_LIMIT );
    numLowerLimit = mxGetNumberOfElements( P_PAR_LOWER_LIMIT );

    if (numUpperLimit > numLowerLimit) {
        maxNumLimit = numUpperLimit;
    } else {
        maxNumLimit = numLowerLimit;
    }

    /*
    * states
    */
    ssSetNumContStates(S, 0);
    ssSetNumDiscStates(S, 0);

    /*
    * outputs
    * The upper and lower limits are scalar expanded
    * so their size determines the size of the output
    * only if at least one of them is not scalar.
    */
    if (!ssSetNumOutputPorts(S, 1)) return;

    if ( maxNumLimit > 1 ) {
        ssSetOutputPortWidth(S, 0, maxNumLimit);
    } else {
        ssSetOutputPortWidth(S, 0, DYNAMICALLY_SIZED);
    }

    /*
    * inputs
    * If the upper or lower limits are not scalar then
    * the input is set to the same size. However, the
```

```
* ssSetOptions below allows the actual width to
* be reduced to 1 if needed for scalar expansion.
*/
if (!ssSetNumInputPorts(S, 1)) return;

ssSetInputPortDirectFeedThrough(S, 0, 1 );

if ( maxNumLimit > 1 ) {
    ssSetInputPortWidth(S, 0, maxNumLimit);
} else {
    ssSetInputPortWidth(S, 0, DYNAMICALLY_SIZED);
}

/*
 * sample times
 */
ssSetNumSampleTimes(S, 1);

/*
 * work
 */
ssSetNumRWork(S, 0);
ssSetNumIWork(S, 0);
ssSetNumPWork(S, 0);

/*
 * Modes and zero crossings:
 * If we have a variable-step solver and this block has a continuous
 * sample time, then
 *   o One mode element will be needed for each scalar output
 *     in order to specify which equation is valid (1), (2), or (3).
 *   o Two ZC elements will be needed for each scalar output
 *     in order to help the solver find the exact instants
 *     at which either of the two possible "equation switches"
 *     One will be for the switch from eq. (1) to (2);
 *     the other will be for eq. (2) to (3) and vice versa.
 * otherwise
 *   o No modes and nonsampled zero crossings will be used.
 *

```

```

    */
    ssSetNumModes(S, DYNAMICALLY_SIZED);
    ssSetNumNonsampledZCs(S, DYNAMICALLY_SIZED);

    /*
    * options
    *   o No mexFunctions and no problematic mxFunctions are called
    *     so the exception free code option safely gives faster simulations.
    *   o Scalar expansion of the inputs is desired. The option provides
    *     this without the need to write mdlSetOutputPortWidth and
    *     mdlSetInputPortWidth functions.
    */
    ssSetOptions(S, ( SS_OPTION_EXCEPTION_FREE_CODE |
                     SS_OPTION_ALLOW_INPUT_SCALAR_EXPANSION));

} /* end mdlInitializeSizes */

/* Function: mdlInitializeSampleTimes =====
 * Abstract:
 *   Specify that the block is continuous.
 */
static void mdlInitializeSampleTimes(SimStruct *S)
{
    ssSetSampleTime(S, 0, INHERITED_SAMPLE_TIME);
    ssSetOffsetTime(S, 0, 0);
}

#define MDL_SET_WORK_WIDTHS
#if defined(MDL_SET_WORK_WIDTHS) && defined(MATLAB_MEX_FILE)
/* Function: mdlSetWorkWidths =====
 * The width of the Modes and the ZCs depends on the width of the output.
 * This width is not always known in mdlInitializeSizes so it is handled
 * here.
 */
static void mdlSetWorkWidths(SimStruct *S)
{

```

```

int nModes;
int nNonsampledZCs;

if (ssIsVariableStepSolver(S) &&
    ssGetSampleTime(S,0) == CONTINUOUS_SAMPLE_TIME &&
    ssGetOffsetTime(S,0) == 0.0) {

    int numOutput = ssGetOutputPortWidth(S, 0);

    /*
     * modes and zero crossings
     *   o One mode element will be needed for each scalar output
     *     in order to specify which equation is valid (1), (2), or (3).
     *   o Two ZC elements will be needed for each scalar output
     *     in order to help the solver find the exact instants
     *     at which either of the two possible "equation switches"
     *     One will be for the switch from eq. (1) to (2);
     *     the other will be for eq. (2) to (3) and vice versa.
     */
    nModes          = numOutput;
    nNonsampledZCs = 2 * numOutput;
} else {
    nModes          = 0;
    nNonsampledZCs = 0;
}
ssSetNumModes(S,nModes);
ssSetNumNonsampledZCs(S,nNonsampledZCs);
}
#endif /* MDL_SET_WORK_WIDTHS */

/* Function: mdlOutputs =====
 * Abstract:
 *
 * A saturation is described by three equations
 *
 * (1)   y = UpperLimit
 * (2)   y = u
 * (3)   y = LowerLimit

```

```

*
* When this block is used with a fixed-step solver or it has a noncontinuous
* sample time, the equations are used as it
*
* Now consider the case of this block being used with a variable-step solver
* and it has a continuous sample time. Solvers work best on smooth problems.
* In order for the solver to work without chattering, limit cycles, or
* similar problems, it is absolutely crucial that the same equation be used
* throughout the duration of a MajorTimeStep. To visualize this, consider
* the case of the Saturation block feeding an Integrator block.
*
* To implement this rule, the mode vector is used to specify the
* valid equation based on the following:
*
*   if                UpperLimit < u      then use (1)
*   if   LowerLimit <= u <= UpperLimit    then use (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 uIdx      = 0;
int_T uInc      = ( ssGetInputPortWidth(S,0) > 1 );
const real_T *upperLimit = mxGetPr( P_PAR_UPPER_LIMIT );
int_T upperLimitInc = ( mxGetNumberOfElements( P_PAR_UPPER_LIMIT ) > 1 );
const real_T *lowerLimit = mxGetPr( P_PAR_LOWER_LIMIT );
int_T lowerLimitInc = ( mxGetNumberOfElements( P_PAR_LOWER_LIMIT ) > 1 );

UNUSED_ARG(tid); /* not used in single tasking mode */

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[uIdx] >= *upperLimit) {
            *y++ = *upperLimit;
        } else if (*uPtrs[uIdx] > *lowerLimit) {
            *y++ = *uPtrs[uIdx];
        } else {
            *y++ = *lowerLimit;
        }

        upperLimit += upperLimitInc;
        lowerLimit += lowerLimitInc;
        uIdx      += uInc;
    }
} 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, ie the valid equation for each output scalar.
     */
    for ( iOutput = 0; iOutput < numOutput; iOutput++ ) {
        if ( *uPtrs[uIdx] > *upperLimit ) {
            /*
             * Upper limit eq is valid.
             */
            mode[iOutput] = UpperLimitEquation;
        } else if ( *uPtrs[uIdx] < *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.
     */
    uIdx      += uInc;
    upperLimit += upperLimitInc;
    lowerLimit += lowerLimitInc;
}

/*
 * Reset index to input and limits.
 */
```

```
        uIdx      = 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[uIdx];
        }

        /*
         * Adjust indices to give scalar expansion if needed.
         */
        uIdx      += uInc;
        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
 * model is using a variable-step solver.
 *
 * Calculate zero crossing (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 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. For
 * 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 foolproof, 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 an 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
*  $25\% = 100\% \cdot (-2) / (-2 - (+6))$  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 uIdx      = 0;
    int_T uInc      = ( 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[uIdx] - *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[uIdx] - *lowerLimit;

/*
 * Adjust indices to give scalar expansion if needed.
 */
uIdx      += uInc;
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)
{
    UNUSED_ARG(S); /* unused input argument */
}

```

```
#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 of a Time-Varying Continuous Transfer Function

The S-function `stvctf` is an example of a time-varying continuous transfer function. It demonstrates how to work with the solvers so that the simulation maintains *consistency*, which means that the block maintains smooth and consistent signals for the integrators although the equations that are being integrated are changing.

`matlabroot/simulink/src/stvctf.c`

```
/*
 * File : stvctf.c
 * Abstract:
 *     Time Varying Continuous Transfer Function block
 *
 *     This S-function implements a continuous time transfer function
 *     whose transfer function polynomials are passed in via the input
 *     vector. This is useful for continuous time adaptive control
 *     applications.
 *
 *     This S-function is also an example of how to use banks to avoid
 *     problems with computing derivatives when a continuous output has
 *     discontinuities. The consistency checker can be used to verify that
 *     your S-function is correct with respect to always maintaining smooth
 *     and consistent signals for the integrators. By consistent we mean that
 *     two mdlOutputs calls at major time t and minor time t are always the
 *     same. The consistency checker is enabled on the diagnostics page of the
 *     Configuration parameters dialog box. The update method of this S-function
 *     modifies the coefficients of the transfer function, which cause the
 *     output to "jump." To have the simulation work properly, we need to let
 *     the solver know of these discontinuities by setting
```

```

*      ssSetSolverNeedsReset and then we need to use multiple banks of
*      coefficients so the coefficients used in the major time step output
*      and the minor time step outputs are the same. In the simulation loop
*      we have:
*      Loop:
*          o Output in major time step at time t
*          o Update in major time step at time t
*          o Integrate (minor time step):
*              o Consistency check: recompute outputs at time t and compare
*                with current outputs.
*              o Derivatives at time t
*              o One or more Output,Derivative evaluations at time t+k
*                where k <= step_size to be taken.
*              o Compute state, x
*              o t = t + step_size
*          End_Integrate
*      End_Loop
*      Another purpose of the consistency checker is to verify that when
*      the solver needs to try a smaller step_size, the recomputing of
*      the output and derivatives at time t doesn't change. Step size
*      reduction occurs when tolerances aren't met for the current step size.
*      The ideal ordering would be to update after integrate. To achieve
*      this we have two banks of coefficients. And the use of the new
*      coefficients, which were computed in update, is delayed until after
*      the integrate phase is complete.
*
* This block has multiple sample times and will not work correctly
* in a multitasking environment. It is designed to be used in
* a single tasking (or variable step) simulation environment.
* Because this block accesses the input signal in both tasks,
* it cannot specify the sample times of the input and output ports
* (SS_OPTION_PORT_SAMPLE_TIMES_ASSIGNED).
*
* See simulink/src/sfuntmpl_doc.c.
*
* Copyright 1990-2000 The MathWorks, Inc.
*/

#define S_FUNCTION_NAME stvctf
#define S_FUNCTION_LEVEL 2

```

```

#include "simstruc.h"

/*
 * Defines for easy access to the numerator and denominator polynomials
 * parameters
 */
#define NUM(S)  ssGetSFcnParam(S, 0)
#define DEN(S)  ssGetSFcnParam(S, 1)
#define TS(S)   ssGetSFcnParam(S, 2)
#define NPARAMS 3

#define MDL_CHECK_PARAMETERS
#if defined(MDL_CHECK_PARAMETERS) && defined(MATLAB_MEX_FILE)
/* Function: mdlCheckParameters =====
 * Abstract:
 *   Validate our parameters to verify:
 *   o The numerator must be of a lower order than the denominator.
 *   o The sample time must be a real positive nonzero value.
 */
static void mdlCheckParameters(SimStruct *S)
{
    int_T i;

    for (i = 0; i < NPARAMS; i++) {
        real_T *pr;
        int_T  el;
        int_T  nEls;
        if (mxIsEmpty(  ssGetSFcnParam(S,i) ||
            mxIsSparse(  ssGetSFcnParam(S,i) ||
            mxIsComplex( ssGetSFcnParam(S,i) ||
            !mxIsNumeric( ssGetSFcnParam(S,i) ) ) {
            ssSetErrorStatus(S,"Parameters must be real finite vectors");
            return;
        }
        pr  = mxGetPr(ssGetSFcnParam(S,i));
        nEls = mxGetNumberOfElements(ssGetSFcnParam(S,i));
        for (el = 0; el < nEls; el++) {
            if (!mxIsFinite(pr[el])) {

```



```

        ssSetErrorStatus(S,"Parameters must be real finite vectors");
        return;
    }
}

if (mxGetNumberOfElements(NUM(S)) > mxGetNumberOfElements(DEN(S)) &&
    mxGetNumberOfElements(DEN(S)) > 0 && *mxGetPr(DEN(S)) != 0.0) {
    ssSetErrorStatus(S,"The denominator must be of higher order than "
        "the numerator, nonempty and with first "
        "element nonzero");
    return;
}

/* xxx verify finite */
if (mxGetNumberOfElements(TS(S)) != 1 || mxGetPr(TS(S))[0] <= 0.0) {
    ssSetErrorStatus(S,"Invalid sample time specified");
    return;
}
}
#endif /* MDL_CHECK_PARAMETERS */

/* 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)
{
    int_T nContStates;
    int_T nCoeffs;

    /* See sfuntmpl_doc.c for more details on the macros below. */

    ssSetNumSFcnParams(S, NPARAMS); /* Number of expected parameters. */
#ifdef MATLAB_MEX_FILE
    if (ssGetNumSFcnParams(S) == ssGetSFcnParamsCount(S)) {
        mdlCheckParameters(S);
        if (ssGetErrorStatus(S) != NULL) {

```

```

        return;
    }
} else {
    return; /* Parameter mismatch will be reported by Simulink. */
}
#endif

/*
 * Define the characteristics of the block:
 *
 * Number of continuous states:    length of denominator - 1
 * Inputs port width              2 * (NumContStates+1) + 1
 * Output port width              1
 * DirectFeedThrough:            0 (Although this should be computed.
 *                               We'll assume coefficients entered
 *                               are strictly proper).
 * Number of sample times:        2 (continuous and discrete)
 * Number of Real work elements:  4*NumCoeffs
 *                               (Two banks for num and den coeff's:
 *                               NumBank0Coeffs
 *                               DenBank0Coeffs
 *                               NumBank1Coeffs
 *                               DenBank1Coeffs)
 * Number of Integer work elements: 2 (indicator of active bank 0 or 1
 *                               and flag to indicate when banks
 *                               have been updated).
 *
 * The number of inputs arises from the following:
 *   o 1 input (u)
 *   o the numerator and denominator polynomials each have NumContStates+1
 *     coefficients
 */
nCoeffs      = mxGetNumberOfElements(DEN(S));
nContStates = nCoeffs - 1;

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

if (!ssSetNumInputPorts(S, 1)) return;

```

```

    ssSetInputPortWidth(S, 0, 1 + (2*nCoeffs));
    ssSetInputPortDirectFeedThrough(S, 0, 0);
    ssSetInputPortSampleTime(S, 0, mxGetPr(TS(S))[0]);
    ssSetInputPortOffsetTime(S, 0, 0);

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

    ssSetNumSampleTimes(S, 2);

    ssSetNumRWork(S, 4 * nCoeffs);
    ssSetNumIWork(S, 2);
    ssSetNumPWork(S, 0);

    ssSetNumModes(S, 0);
    ssSetNumNonsampledZCs(S, 0);

    /* Take care when specifying exception free code - see sfunmpl_doc.c */
    ssSetOptions(S, (SS_OPTION_EXCEPTION_FREE_CODE));

} /* end mdlInitializeSizes */

/* Function: mdlInitializeSampleTimes =====
 * Abstract:
 * This function is used to specify the sample time(s) for the
 * S-function. This S-function has two sample times. The
 * first, a continuous sample time, is used for the input to the
 * transfer function, u. The second, a discrete sample time
 * provided by the user, defines the rate at which the transfer
 * function coefficients are updated.
 */
static void mdlInitializeSampleTimes(SimStruct *S)
{
    /*
     * the first sample time, continuous
     */

```

```

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

        /*
         * the second, discrete sample time, is user provided
         */
        ssSetSampleTime(S, 1, mxGetPr(TS(S))[0]);
        ssSetOffsetTime(S, 1, 0.0);

    } /* end mdlInitializeSampleTimes */

#define MDL_INITIALIZE_CONDITIONS
/* Function: mdlInitializeConditions =====
 * Abstract:
 *      Initalize the states, numerator and denominator coefficients.
 */
static void mdlInitializeConditions(SimStruct *S)
{
    int_T i;
    int_T nContStates = ssGetNumContStates(S);
    real_T *x0          = ssGetContStates(S);
    int_T nCoeffs       = nContStates + 1;
    real_T *numBank0    = ssGetRWork(S);
    real_T *denBank0    = numBank0 + nCoeffs;
    int_T *activeBank   = ssGetIWork(S);

    /*
     * The continuous states are all initialized to zero.
     */
    for (i = 0; i < nContStates; i++) {
        x0[i]          = 0.0;
        numBank0[i]    = 0.0;
        denBank0[i]    = 0.0;
    }
    numBank0[nContStates] = 0.0;
    denBank0[nContStates] = 0.0;

    /*

```

```

    * Set up the initial numerator and denominator.
    */
    {
        const real_T *numParam = mxGetPr(NUM(S));
        int          numParamLen = mxGetNumberOfElements(NUM(S));

        const real_T *denParam = mxGetPr(DEN(S));
        int          denParamLen = mxGetNumberOfElements(DEN(S));
        real_T       den0       = denParam[0];

        for (i = 0; i < denParamLen; i++) {
            denBank0[i] = denParam[i] / den0;
        }

        for (i = 0; i < numParamLen; i++) {
            numBank0[i] = numParam[i] / den0;
        }
    }

    /*
    * Normalize if this transfer function has direct feedthrough.
    */
    for (i = 1; i < nCoeffs; i++) {
        numBank0[i] -= denBank0[i]*numBank0[0];
    }

    /*
    * Indicate bank0 is active (i.e. bank1 is oldest).
    */
    *activeBank = 0;

} /* end mdlInitializeConditions */

/* Function: mdlOutputs =====
* Abstract:
*   The outputs for this block are computed by using a controllable state-
*   space representation of the transfer function.
*/

```



```

        *
        *   B(s) = b0 s^n + b1 s^n-1 + b2 s^n-2 + ... + bn-1 s + bn
        */
        *y = *num++ * (*uPtrs[0]);
        for (i = 0; i < nContStates; i++) {
            *y += *num++ * *x++;
        }
    }

} /* end mdlOutputs */

#define MDL_UPDATE
/* Function: mdlUpdate =====
 * Abstract:
 *   Every time through the simulation loop, update the
 *   transfer function coefficients. Here we update the oldest bank.
 */
static void mdlUpdate(SimStruct *S, int_T tid)
{
    UNUSED_ARG(tid); /* not used in single tasking mode */

    if (ssIsSampleHit(S, 1, tid)) {
        int_T          i;
        InputRealPtrsType uPtrs      = ssGetInputPortRealSignalPtrs(S,0);
        int_T          uIdx          = 1; /*1st coeff is after signal input*/
        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[uIdx+nCoeffs];
    if (den0 == 0.0) {
        den0 = mxGetPr(DEN(S))[0];
    }

    /*
    * Grab the numerator.
    */
    allZero = 1;
    for (i = 0; (i < nCoeffs) && allZero; i++) {
        allZero &= *uPtrs[uIdx+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.
        */
        uIdx += nCoeffs;
        num += nCoeffs - numParamLen;
        for (i = 0; i < numParamLen; i++) {
            *num++ = *numParam++ / den0;
        }
    } else {
        for (i = 0; i < nCoeffs; i++) {
            *num++ = *uPtrs[uIdx++] / den0;
        }
    }
}

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



```

    }

    if (allZero) { /* If denominator is all zero. */
        const real_T *denParam = mxGetPr(DEN(S));
        int_T denParamLen = mxGetNumberOfElements(DEN(S));

        den0 = denParam[0];
        for (i = 0; i < denParamLen; i++) {
            *den++ = *denParam++ / den0;
        }
    } else {
        for (i = 0; i < nCoeffs; i++) {
            *den++ = *uPtrs[uIdx++] / den0;
        }
    }
}

/*
 * Normalize if this transfer function has direct feedthrough.
 */
num = ssGetRWork(S) + bankToUpdate*2*nCoeffs;
den = num + nCoeffs;
for (i = 1; i < nCoeffs; i++) {
    num[i] -= den[i]*num[0];
}

/*
 * Indicate oldest bank has been updated.
 */
ssGetIWork(S)[1] = 1;
}

} /* end mdlUpdate */

#define MDL_DERIVATIVES
/* Function: mdlDerivatives =====
 * Abstract:
 * The derivatives for this block are computed by using a controllable
 * state-space representation of the transfer function.

```

```

*/
static void mdlDerivatives(SimStruct *S)
{
    int_T          i;
    int_T          nContStates = ssGetNumContStates(S);
    real_T         *x          = ssGetContStates(S);
    real_T         *dx         = ssGetdX(S);
    int_T          nCoeffs     = nContStates + 1;
    int_T          activeBank   = ssGetIWork(S)[0];
    const real_T   *num        = ssGetRWork(S) + activeBank*(2*nCoeffs);
    const real_T   *den        = num + nCoeffs;
    InputRealPtrsType uPtrs    = ssGetInputPortRealSignalPtrs(S,0);

    /*
    * The continuous system is evaluated using a controllable state-space
    * representation of the transfer function. This implies that the
    * next continuous states are computed using:
    *
    *
    *      dx = Ax(t) + Bu(t)
    *          = [-a1 -a2 ... -an] [x1(t)] + [u(t)]
    *            [ 1  0 ...  0] [x2(t)] + [0]
    *            [ 0  1 ...  0] [x3(t)] + [0]
    *            [ .  . ...  .]  .      + .
    *            [ .  . ...  .]  .      + .
    *            [ .  . ...  .]  .      + .
    *            [ 0  0 ... 1 0] [xn(t)] + [0]
    *
    * where a1, a2, ... are the coefficients of the numerator polynomial:
    *
    *      A(s) = s^n + a1 s^{n-1} + a2 s^{n-2} + ... + an-1 s + an
    */
    dx[0] = -den[1] * x[0] + *uPtrs[0];
    for (i = 1; i < nContStates; i++) {
        dx[i] = x[i-1];
        dx[0] -= den[i+1] * x[i];
    }

} /* end mdlDerivatives */

```

```
/* Function: mdlTerminate =====
 * Abstract:
 *   Called when the simulation is terminated.
 *   For this block, there are no end of simulation tasks.
 */
static void mdlTerminate(SimStruct *S)
{
    UNUSED_ARG(S); /* unused input argument */
} /* end mdlTerminate */

#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
```


S-Function Callback Methods — Alphabetical List

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.

- mdlCheckParameters
- mdlDerivatives
- mdlDisable
- mdlEnable
- mdlGetTimeOfNextVarHit
- mdlInitializeConditions
- mdlInitializeSampleTimes
- mdlInitializeSizes
- mdlOutputs
- mdlProcessParameters
- mdlRTW
- mdlSetDefaultPortComplexSignals
- mdlSetDefaultPortDataTypes
- mdlSetDefaultPortDimensionInfo
- mdlSetInputPortComplexSignal
- mdlSetInputPortDataType

mdlSetInputPortDimensionInfo
mdlSetInputPortFrameData
mdlSetInputPortSampleTime
mdlSetInputPortWidth
mdlSetOutputPortComplexSignal
mdlSetOutputPortDataType
mdlSetOutputPortDimensionInfo
mdlSetOutputPortSampleTime
mdlSetOutputPortWidth
mdlSetWorkWidths
mdlSimStatusChange
mdlStart
mdlTerminate
mdlUpdate
mdlZeroCrossings

Purpose Check the validity of an S-function's parameters

C Syntax `void mdlCheckParameters(SimStruct *S)`

C Arguments S
SimStruct representing an S-Function block.

M Syntax `CheckParameters(s)`

M Arguments s
Instance of `Simulink.MSFcnRunTimeBlock` class representing an S-Function block.

Description Verifies new parameter settings whenever parameters change or are reevaluated 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 are 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.

mdlCheckParameters

```
#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 nonnegative");
        return;
    }
}
#endif /* MDL_CHECK_PARAMETERS */
```

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

```
static void mdlInitializeSizes(SimStruct *S)
{
    ssSetNumSFcnParams(S, 1); /* Number of expected parameters */
#if defined(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, M

See Also

`mdlProcessParameters`, `ssGetSFcnParamsCount`

mdlDerivatives

Purpose	Compute the S-function's derivatives
C Syntax	<code>void mdlDerivatives(SimStruct *S)</code>
C Arguments	S SimStruct representing an S-Function block.
M Syntax	<code>Derivatives(s)</code>
M Arguments	s Instance of <code>Simulink.MSFcnRunTimeBlock</code> class representing the 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 values 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 <code>matlabroot/simulink/src/csfunc.c</code> .
Languages	Ada, C, M
See Also	<code>ssGetdx</code>

Purpose	Respond to disabling of an enabled system containing this block
C Syntax	<code>void mdlDisable(SimStruct *S)</code>
C Arguments	S SimStruct representing an S-Function block.
M Syntax	<code>Disable(s)</code>
M Arguments	s Instance of <code>Simulink.MSFcnRunTimeBlock</code> class representing the S-Function block.
Description	Simulink invokes this optional method if this block resides in an enabled subsystem and the enabled subsystem changes from an enabled to a disabled state at the current time step. Your S-function can use this method to perform any actions required by the disabling of the containing subsystem.
Languages	Ada, C, M
See Also	<code>mdlEnable</code>

mdlEnable

Purpose	Respond to enabling of an enabled system containing this block
C Syntax	<code>void mdlEnable(SimStruct *S)</code>
C Arguments	S SimStruct representing an S-Function block.
M Syntax	<code>Enable(s)</code>
M Arguments	s Instance of <code>Simulink.MSFcnRunTimeBlock</code> class representing the S-Function block.
Description	Simulink invokes this optional method if this block resides in an enabled subsystem and the enabled subsystem changes from a disabled to an enabled state at the current time step. Your S-function can use this method to perform any actions required by the enabling of the containing subsystem.
Languages	Ada, C, M
See Also	<code>mdlDisable</code>

Purpose Specify time of the next sample time hit

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 the next hit can be a function of the input signals.

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`, `ssGetT`, `ssSetTNext`

mdlInitializeConditions

Purpose Initialize the state vectors of this S-function

C Syntax void mdlInitializeConditions(SimStruct *S)

C Arguments S
SimStruct representing an S-Function block.

M Syntax InitializeConditions(s)

M Arguments s
Instance of Simulink.MSFcnRunTimeBlock class representing the 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 ssGetContStates and/or ssGetDiscStates to get the states. This method can also perform any other initialization activities that this S-function requires.

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 the ssIsFirstInitCond macro to determine whether it is being called for the first time.

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   = ssGetRealDiscStates(S);
int_T   nDStates = ssGetNumDiscStates(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 that initializes only the continuous states, see *matlabroot/simulink/src/resetint.c*.

Languages

C, C++, M

See Also

mdlStart, ssIsFirstInitCond, ssGetContStates, ssGetDiscStates

mdlInitializeSampleTimes

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 uppercase 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.

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_IN_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]
```

mdlInitializeSampleTimes

Simulink invokes the `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]
```

- Use `ssSetOptions` to set the `SS_OPTION_DISALLOW_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., is held during minor steps) should set its sample time to

```
[ INHERITED_SAMPLE_TIME, FIXED_IN_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 receives incorrect results if it uses `ssIsSampleHit(S,0,tid)`.

If the function wants to determine whether the third (discrete) task has a hit, it can use the following code fragment.

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

Languages

C

See Also

`mdlSetInputPortSampleTime`, `mdlSetOutputPortSampleTime`

mdlInitializeSizes

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>ssSetSFcnParamTunable(S,paramIdx, 0)</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 issues 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 8-18 for more information.▪ For each input port, specify 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> function. 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 `mdlOutputs` or `mdlGetTimeOfNextVarHit` routine. Setting the direct feedthrough flag to 0 tells Simulink that *u* is not used in either of these S-function routines. Violating this leads 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 `mdlSetOutputPortDimensionInfo` for more information.

If your S-function outputs are discrete (e.g., can only take the values 1 and 2), 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 7-17 for a complete discussion of sample time issues.

For multirate S-functions, the suggested approach to setting sample times is via the port-based sample times method. When you create a multirate S-function, you must take care to verify that, when slower tasks are preempted, your S-function correctly manages data so 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`.

mdlInitializeSizes

- 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 widths, 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 values. 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 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
 * starting at 0.

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

/*
```

mdlInitializeSizes

```
* 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	<p><code>S</code> SimStruct representing an S-Function block.</p> <p><code>tid</code> Task ID.</p>
M Syntax	<code>Outputs(s)</code>
M Arguments	<p><code>s</code> Instance of <code>Simulink.MSFcnRunTimeBlock</code> class representing the S-Function block.</p>
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 7-26).</p> <p>For an example of an <code>mdlOutputs</code> routine that works with multiple input and output ports, see <code>matlabroot/simulink/src/sfun_multiport.c</code>.</p>
Languages	A, C, C++, M
See Also	<code>ssGetOutputPortComplexSignal</code> , <code>ssGetOutputPortRealSignal</code> , <code>ssGetOutputPortSignal</code>

mdlProcessParameters

Purpose Process the S-function's parameters

C Syntax `void mdlProcessParameters(SimStruct *S)`

C Arguments S
SimStruct representing an S-Function block.

M Syntax `ProcessParameters(s)`

M Arguments s
Instance of `Simulink.MSFcnRunTimeBlock` class representing the 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 Real-Time Workshop. Therefore, if you use this routine in an S-function designed for use with 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 "Real-Time Workshop 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 *plusMinusStr;  
    int_T nInputPorts = ssGetNumInputPorts(S);  
    int_T *iwork = ssGetIWork(S);  
    if ((plusMinusStr=(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 mdlStart");  
        return;  
    }  
    for (i = 0; i < nInputPorts; i++) {  
        iwork[i] = plusMinusStr[i] == '+'? 1: -1;  
    }  
    free(plusMinusStr);  
}  
#endif /* MDL_PROCESS_PARAMETERS */
```

`mdlProcessParameters` is called from `mdlStart` to load the signs string prior to the start of the simulation loop.

```
#define MDL_START  
#if defined(MDL_START)
```

mdlProcessParameters

```
static void mdlStart(SimStruct *S)
{
    mdlProcessParameters(S);
}
#endif /* MDL_START */
```

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

Languages Ada, C, M

See Also mdlCheckParameters

Purpose	Generate code generation data
C Syntax	<code>void mdlRTW(SimStruct *S)</code>
C Arguments	S SimStruct representing an S-Function block.
M Syntax	<code>PostPropagationSetup(s)</code>
M Arguments	s Instance of <code>Simulink.MSFcnRunTimeBlock</code> class representing the S-Function block.
Description	<p>This function is called when Real-Time Workshop is generating the <code>model.rtw</code> file. In this method, you can call the following functions that 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, C++, M

See Also `ssSetInputPortFrameData`, `ssSetOutputPortFrameData`,
`ssSetErrorStatus`

mdlSetDefaultPortComplexSignals

Purpose	Set the numeric types (real, complex, or inherited) of ports whose numeric types 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 types 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 types of all ports whose numeric types are 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>

mdlSetDefaultPortDataTypes

Purpose Set the data types of ports whose data types cannot be determined from block connectivity

Syntax `void mdlSetDefaultPortDataTypes(SimStruct *S)`

Arguments S
SimStruct representing an S-Function block.

Description Simulink invokes this method if the block has ports whose data types cannot be determined from block connectivity. (This usually happens when the block is unconnected or is part of a feedback loop.) This method must set the data types of all ports whose data types are not set. 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.

Languages C

See Also `ssSetOutputPortDataType`, `ssSetInputPortDataType`

mdlSetDefaultPortDimensionInfo

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 sets 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>ssSetErrorStatus</code> , <code>ssSetOutputPortMatrixDimensions</code>

mdlSetInputPortComplexSignal

Purpose Set the numeric types (real, complex, or inherited) of the signals accepted by an input port

C Syntax `void mdlSetInputPortComplexSignal(SimStruct *S, int_T port, CSignal_T csig)`

C Arguments

`S` SimStruct representing an S-Function block.

`port` Index of a port.

`csig` Numeric type of signal.

M Syntax `SetInputPortComplexSignal(s, port, typeId)`

M Arguments

`s` Instance of `Simulink.MSFcnRunTimeBlock` class representing the S-Function block.

`port` Integer value specifying index of port to be set.

`typeId` Integer value specifying whether the port accepts real (0) or complex (1) signals.

Description Simulink calls this routine to set the input port signal type. The S-function must check whether 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 `ssSetErrorStatus`. The S-function can also set the signal types 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.

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.

Languages C, C++, M

See Also `ssSetErrorStatus`, `ssSetInputPortComplexSignal`

mdlSetInputPortDataType

Purpose Set the data types of the signals accepted by an input port

C Syntax `void mdlSetInputPortDataType(SimStruct *S, int_T port, DTypeId id)`

C Arguments

`S` SimStruct representing an S-Function block.

`port` Index of a port.

`id` Data type ID.

M Syntax `SetInputPortDataType(s, port, typeId)`

M Arguments

`s` Instance of `Simulink.MSFcnRunTimeBlock` class representing the S-Function block.

`port` Integer value specifying index of port to be set.

`typeId` Integer value specifying ID of port's data type. Use `s.getDatatypeName(typeId)` to get the data type's name.

Description Simulink calls this routine to set the data type of port. The S-function must check whether 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 `ssSetErrorStatus`.

The S-function can also set the data types 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.

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.

Languages C

See Also `ssSetErrorStatus`, `ssSetInputPortDataType`

mdlSetInputPortDimensionInfo

Purpose Set the dimensions of the signals accepted by an input port

C Syntax `void mdlSetInputPortDimensionInfo(SimStruct *S, int_T port, const DimsInfo_T *dimsInfo)`

C Arguments

`S` SimStruct representing an S-Function block.

`port` Index of a port.

`dimsInfo` Structure that specifies the signal dimensions supported by the port.

See `ssSetInputPortDimensionInfo` for a description of this structure.

M Syntax `SetInputPortDimensions(s, port, dims)`

M Arguments

`s` Instance of `Simulink.MSFcnRunTimeBlock` class representing the S-Function block.

`port` Integer value specifying index of port to be set.

`dims` 1-D array that specifies the signal dimensions supported by the port, e.g., `[5]` for a 5-element vector signal or `[3 3]` for a 3-by-3 matrix signal.

Description Simulink calls this method during dimension propagation with candidate dimensions `dimsInfo` for `port`. If the proposed dimensions are acceptable, this method should go ahead and set the actual port dimensions, using `ssSetInputPortDimensionInfo`. If they

are unacceptable, this method should generate an error via `ssSetErrorStatus`.

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, C++, M

Example

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

See Also

`ssSetErrorStatus`

mdlSetInputPortFrameData

Purpose Specify whether an input port accepts frame data

C Syntax `void mdlSetInputPortFrameData(SimStruct *S, int_T port, rame_T frameData)`

C Arguments

`S` SimStruct representing an S-Function block.

`port` Index of a port.

`frameData` Frame data.

M Syntax `SetInputPortSamplingMode(s, port, mode)`

M Arguments

`s` Instance of Simulink.MSFcnRunTimeBlock class representing the S-Function block.

`port` Integer value specifying the index of port whose sampling mode is to be set.

`mode` Integer value specifying the sampling mode of the port (0 = sample, 1 = frame).

Description 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 `ssSetInputPortFrameData`. If the setting is unacceptable, an error should be generated via `ssSetErrorStatus`. Note that any other dynamic frame input or output ports whose frame data settings 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 `ssSetInputPortFrameData` and `ssSetOutputPortFrameData`.

The use of frame-based signals (`mode` has a value of 1) requires a Signal Processing Blockset license.

Languages

C, C++, M

See Also

`ssSetInputPortFrameData`, `ssSetOutputPortFrameData`,
`ssSetErrorStatus`

mdlSetInputPortSampleTime

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

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

C 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.

M Syntax `SetInputPortSampleTime(s, port, time)`

M Arguments

`s` Instance of `Simulink.MSFcnRunTimeBlock` class representing the S-Function block.

`port` Integer value specifying the index of port whose sampling mode is to be set.

`time` Two-element array, `[period offset]`, that specifies the period and offset of the times that this port samples its input.

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, the sample time is guaranteed to be one of the following where $0.0 < \text{period} < \text{inf}$ and $0.0 \leq \text{offset} < \text{period}$.

	Sample Time	Offset Time
Continuous	0.0	0.0
Discrete	period	offset

Constant, triggered, and variable-step sample times are not propagated to S-functions with port-based sample times.

Generally mdlSetInputPortSampleTime is called once with the input port sample time. However, there can be cases where this function is 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 are restored before this method is called again.

The final sample time specified at the port can 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 $[0.249999999999999, 0]$ to $[0.25, 0]$.

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

mdlSetInputPortSampleTime

Languages C, C++, M

See Also `ssSetInputPortSampleTime`, `ssSetOutputPortSampleTime`,
`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><code>S</code> SimStruct representing an S-Function block.</p> <p><code>port</code> Index of a port.</p> <p><code>width</code> Width of signal.</p>
Description	<p>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>.</p>
Languages	C
See Also	<code>ssSetInputPortWidth</code> , <code>ssSetOutputPortWidth</code> , <code>ssSetErrorStatus</code>

mdlSetOutputPortComplexSignal

Purpose Set the numeric types (real, complex, or inherited) of the signals accepted by an output port

C Syntax `void mdlSetOutputPortDataType(SimStruct *S, int_T port, CSignal_T csig)`

C Arguments

`S` SimStruct representing an S-Function block.

`port` Index of a port.

`csig` Numeric type of signal.

M Syntax `SetOutputPortComplexSignal(s, port, typeId)`

M Arguments

`s` Instance of `Simulink.MSFcnRunTimeBlock` class representing the S-Function block.

`port` Integer value specifying the index of port to be set.

`typeId` Integer value specifying whether the port produces real (0) or complex (1) signals.

Description Simulink calls this routine to set the output port signal type. The S-function must check whether 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 `ssSetErrorStatus`. The S-function can also set the signal types 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.

mdlSetOutputPortComplexSignal

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.

Languages

C, C++, M

See Also

ssSetOutputPortComplexSignal, ssSetErrorStatus

mdlSetOutputPortDataType

Purpose Set the data type of the signals emitted by an output port

C Syntax `void mdlSetOutputPortDataType(SimStruct *S, int_T port, DTypeId id)`

C Arguments

`S` SimStruct representing an S-Function block.

`port` Index of an output port.

`id` Data type ID.

M Syntax `SetOutputPortDataType(s, port, typeId)`

M Arguments

`s` Instance of `Simulink.MSFcnRunTimeBlock` class representing the S-Function block.

`port` Integer value specifying index of port to be set.

`typeId` Integer value specifying ID of port's data type. Use `s.getDatatypeName(typeId)` to get the data type's name.

Description Simulink calls this routine to set the data type of port. The S-function must check whether 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 port. Otherwise, it must report an error, using `ssSetErrorStatus`.

The S-function can also set the data types 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.

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.

Languages

C, C++, M

See Also

ssSetOutputPortDataType, ssSetErrorStatus

mdlSetOutputPortDimensionInfo

Purpose Set the dimensions of the signals accepted by an output port

C Syntax `void mdlSetOutputPortDimensionInfo(SimStruct *S, int_T port, const DimsInfo_T *dimsInfo)`

C Arguments

`S` SimStruct representing an S-Function block or a Simulink model.

`port` Index of a port.

`dimsInfo` Structure that specifies the signal dimensions supported by port.

See `ssSetInputPortDimensionInfo` for a description of this structure.

M Syntax `SetOutputPortDimensions(s, port, dims)`

M Arguments

`s` Instance of `Simulink.MSFcnRunTimeBlock` class representing the S-Function block.

`port` Integer value specifying the index of the port to be set.

`dims` 1-D array that specifies the signal dimensions supported by the port, e.g., `[5]` for a 5-element vector signal or `[3 3]` for a 3-by-3 matrix signal.

Description Simulink calls this method with candidate dimensions `dimsInfo` for port. If the proposed dimensions are acceptable, this method should go ahead and set the actual port dimensions, using `ssSetOutputPortDimensionInfo`. If they are unacceptable, this method should generate an error via `ssSetErrorStatus`.

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 `mdlSetOutputPortDimensionInfo` even if it can only partially determine the dimensionality of the input port from connectivity.

Languages

C, C++, M

Example

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

See Also

`ssSetErrorStatus`, `ssSetOutputPortDimensionInfo`

mdlSetOutputPortSampleTime

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

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

C 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.

M Syntax `SetOutputPortSampleTime(s, port, time)`

M Arguments

`s` Instance of Simulink.MSFcnRunTimeBlock class representing the S-Function block.

`port` Integer value specifying the index of port whose sampling mode is to be set.

`time` Two-element array, [period offset], that specifies the period and offset of the times that this port produces output.

Description 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 `ssSetOutputPortSampleTime` or

```
pd = s.OutputPort(port);  
pd.SampleTime = time;
```

in the case of a Level-2 M-file S-function.

If the inherited sample time is unacceptable, this method should generate an error via `ssSetErrorStatus`. 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 `ssSetInputPortSampleTime` or `ssSetOutputPortSampleTime` or the `SampleTime` property of the `Simulink.BlockPortData` object associated with the port in the case of Level-2 M-file S-functions.

Normally, sample times are propagated forward; however, if sources feeding this block have inherited sample times, Simulink might 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.

See `mdlSetInputPortSampleTime` for more information about when this method is called.

Languages

C

See Also

`ssSetOutputPortSampleTime`, `ssSetErrorStatus`,
`ssSetInputPortSampleTime`, `ssSetOutputPortSampleTime`,
`mdlSetInputPortSampleTime`, `Simulink.MSFcnRunTimeBlock`,
`Simulink.BlockPortData`

mdlSetOutputPortWidth

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	<table><tr><td><code>S</code></td><td>SimStruct representing an S-Function block.</td></tr><tr><td><code>port</code></td><td>Index of a port.</td></tr><tr><td><code>width</code></td><td>Width of signal.</td></tr></table>	<code>S</code>	SimStruct representing an S-Function block.	<code>port</code>	Index of a port.	<code>width</code>	Width of signal.
<code>S</code>	SimStruct representing an S-Function block.						
<code>port</code>	Index of a port.						
<code>width</code>	Width of signal.						
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>						

Purpose Specify the sizes of the work vectors and create the run-time parameters required by this S-function

C Syntax `void mdlSetWorkWidths(SimStruct *S)`

C Arguments S
SimStruct representing an S-Function block.

M Syntax `PostPropagationSetup(s)`

M Arguments s
Instance of `Simulink.MSFcnRunTimeBlock` class representing the S-Function block.

Description 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 run-time 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 `ssGetNumContStates`, `ssSetNumDiscStates`, `ssSetNumRWork`, `ssSetNumIWork`, `ssSetNumPWork`, `ssSetNumModes`, and `ssSetNumNonsampledZCs`.

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 `mdlInitializeSizes` method. If this S-function implements `mdlSetWorkWidths`, it should initialize the sizes of any work vectors that it needs to `DYNAMICALLY_SIZED` in `mdlInitializeSizes`, even for those whose exact size it knows at that point. The S-function should then specify the actual size in `mdlSetWorkWidths`.

mdlSetWorkWidths

Languages Ada, C, M

See Also mdlInitializeSizes

Purpose	Respond to a pause or resumption of the simulation of the model that contains this S-function.
Syntax	<pre>void mdlSimStatusChange(SimStruct *s, ssSimStatusChangeType simStatus)</pre>
Arguments	<p>S SimStruct representing an S-Function block.</p> <p>simStatus Status of the simulation, either SIM_PAUSE or SIM_CONTINUE.</p>
Description	Simulink calls this routine when a simulation of the model containing S pauses or resumes.
Example	<pre>#if defined(MATLAB_MEX_FILE) #define MDL_SIM_STATUS_CHANGE static void mdlSimStatusChange(SimStruct *S, ssSimStatusChangeType simStatus) { if (simStatus == SIM_PAUSE) { slPrintf("Pause has been called! \n"); } else if (simStatus == SIM_CONTINUE) { slPrintf("Continue has been called! \n"); } } #endif</pre>
Languages	C

mdlStart

Purpose	Initialize the state vectors of this S-function
C Syntax	<code>void mdlStart(SimStruct *S)</code>
C Arguments	S SimStruct representing an S-Function block.
M Syntax	Start(s)
M Arguments	s Instance of Simulink.MSFcnRunTimeBlock class representing the 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>ssGetDiscStates</code> to get the states. This method can also perform any other initialization activities that this S-function requires.
Languages	Ada, C, M
See Also	<code>mdlInitializeConditions</code> , <code>ssGetContStates</code> , <code>ssGetDiscStates</code>

Purpose Perform any actions required at termination of the simulation

Syntax void mdlTerminate(SimStruct *S)

Arguments S
SimStruct representing an S-Function block.

See Also ssSetOptions

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 `SS_OPTION_CALL_TERMINATE_ON_EXIT` (see `ssSetOptions`) determines whether Simulink invokes this method. If this option is not set, Simulink invokes `mdlTerminate` at the end of simulation only if the `mdlStart` method of at least one block in the model has executed during simulation. If this option is set, Simulink always invokes the `mdlTerminate` 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 `mdlStart` and saves pointers to the blocks in a `PWork` vector. The following code fragment would free this memory.

```
{
    int i;
    for (i = 0; i < ssGetNumPWork(S); i++) {
        if (ssGetPWorkValue(S,i) != NULL) {
            free(ssGetPWorkValue(S,i));
        }
    }
}
```

mdlUpdate

Purpose Update a block's states

C Syntax `void mdlUpdate(SimStruct *S, int_T tid)`

C Arguments

S
SimStruct representing an S-Function block.

tid
Task ID.

M Syntax Update(s)

M Arguments

s
Instance of Simulink.MSFcnRunTimeBlock class representing the S-Function block.

Description 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.

Use this code if your S-function has one or more discrete states or does *not* have direct feedthrough.

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.

If your S-function needs to have its mdlUpdate routine called and it does not satisfy either of the above two conditions, specify that it has a discrete state, using the ssSetNumDiscStates macro in the mdlInitializeSizes function.

The tid (task ID) argument specifies the task running when the mdlOutputs routine is invoked. You can use this argument in the

mdlUpdate routine of a multirate S-Function block to encapsulate task-specific blocks of code (see “Multirate S-Function Blocks” on page 7-26).

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

Languages Ada, C, C++, M

See Also mdlDerivatives, ssGetContStates, ssGetDiscStates

mdlZeroCrossings

Purpose Update zero-crossing vector

C Syntax `void mdlZeroCrossings(SimStruct *S)`

C Arguments S
SimStruct representing an S-Function block.

M Syntax `ProcessParameters(s)`

M Arguments s
Instance of `Simulink.MSFcnRunTimeBlock` class representing the S-Function block.

Description 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 `ssGetNonsampledZCs`.

You can use the optional `mdlZeroCrossings` routine when your S-function has registered the `CONTINUOUS_SAMPLE_TIME` and has nonsampled zero crossings (`ssGetNumNonsampledZCs(S) > 0`). The `mdlZeroCrossings` routine is used to provide Simulink with signals that are to be tracked for zero crossings. These are typically

- Continuous signals entering the S-function
- Internally generated signals that cross zero when a discontinuity would normally occur in `mdlOutputs`

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 signals, `mdlZeroCrossings` updates the `ssGetNonsampledZCs(S)` vector.

Example For an example, see `matlabroot/simulink/src/sfun_zc.c`.

Languages C, C++, M

See Also mdlInitializeSizes, ssGetNonsampledZCs

SimStruct Functions — Categorical List

The following sections describe SimStruct macros and functions.

Introduction (p. 9-2)

Overview of SimStruct macros and functions.

SimStruct Macros and Functions
Listed by Usage (p. 9-3)

SimStruct functions listed by usage.

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 or function. The descriptions appear alphabetically by name to facilitate location of a particular function. This section also provides listings of functions by usage to speed location of macros and functions 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 macro or 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/simstruc.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.

SimStruct Macros and Functions Listed by Usage

Miscellaneous on page 9-3

Error Handling and Status on page 9-4

I/O Port on page 9-4

Dialog Box Parameters on page 9-9

Run-Time Parameters on page 9-10

Sample Time on page 9-11

State and Work Vector on page 9-13

Simulation Information on page 9-16

Function call on page 9-18

Data Type on page 9-18

Real-Time Workshop on page 9-19

Miscellaneous

Macro	Description
ssCallExternalModeFcn	Invoke the external mode function for an S-function.
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.
ssGetUserData	Access user data.

Miscellaneous (Continued)

Macro	Description
ssSetExternalModeFcn	Specify the external mode function for an S-function.
ssSetOptions	Set various simulation options.
ssSetPlacementGroup	Specify the execution order of a sink or source S-function.
ssSetUserData	Specify user data.

Error Handling and Status

Macro	Description
ssGetErrorStatus	Get a string that identifies the last error.
ssPrintf	Print a variable-content msg.
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.
ssGetInputPortComplexSignal	Get the numeric type (complex or real) of an input port.
ssGetInputPortConnected	Determine whether an S-Function block port is connected to a nonvirtual block.
ssGetInputPortDataType	Get the data type of an input port.

I/O Port (Continued)

Macro	Description
ssGetInputPortDimensions	Get the dimensions of the signal accepted by an input port.
ssGetInputPortDirectFeedThrough	Determine whether an input port has direct feedthrough.
ssGetInputPortFrameData	Determine whether a port accepts signal frames.
ssGetInputPortNumDimensions	Get the dimensionality of the signals accepted by an input port.
ssGetInputPortOffsetTime	Determine the offset time of an input port.
ssGetInputPortOverWritable	Determine whether an input port can be overwritten.
ssGetInputPortRealSignal	Get the address of a real, contiguous signal entering an input port.
ssGetInputPortRealSignalPtrs	Access the signal elements connected to an input port.
ssGetInputPortRequiredContiguous	Determine whether the signal elements entering a port must be contiguous.
ssGetInputPortSampleTime	Determine the sample time of an input port.
ssGetInputPortSampleTimeIndex	Get the sample time index of an input port.
ssGetInputPortSignal	Get the address of a contiguous signal entering an input port.
ssGetInputPortSignalAddress	Get the address of an input port's signal (Ada only).

I/O Port (Continued)

Macro	Description
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.
ssGetOutputPortBeingMerged	Determine whether the output of this block is connected to a Merge block.
ssGetOutputPortComplexSignal	Get the numeric type (complex or real) of an output port.
ssGetOutputPortDataType	Get the data type of an output port.
ssGetOutputPortDimensions	Get the dimensions of the signal leaving an output port.
ssGetOutputPortFrameData	Determine whether a port outputs signal frames.
ssGetOutputPortNumDimensions	Get the number of dimensions of an output port.
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.

I/O Port (Continued)

Macro	Description
ssGetOutputPortSignal	Get the vector of signal elements emitted by an output port.
ssGetOutputPortSignalAddress	Get the address of an output port's signal (Ada only).
ssGetOutputPortWidth	Determine the width of an output port.
ssSetInputPortComplexSignal	Set the numeric type (real or complex) of an input port.
ssSetInputPortDataType	Set the data type of an input port.
ssSetInputPortDimensionInfo	Set the dimensionality of an input port.
ssSetInputPortDirectFeedThrough	Specify that an input port is a direct-feedthrough port.
ssSetInputPortFrameData	Specify whether a port accepts signal frames.
ssSetInputPortMatrixDimensions	Specify dimension information for an input port that accepts matrix signals.
ssSetInputPortOffsetTime	Specify the sample time offset for an input port.
ssSetInputPortOverWritable	Specify whether an input port is overwritable by an output port.
ssSetInputPortRequiredContiguous	Specify that the signal elements entering a port must be contiguous.
ssSetInputPortSampleTime	Set the sample time of an input port.

I/O Port (Continued)

Macro	Description
ssSetInputPortVectorDimension	Specify dimension information for an input port that accepts vector signals.
ssSetInputPortWidth	Set the 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.
ssSetOneBasedIndexInputPort	Specify that an input port expects one-based indices.
ssSetOneBasedIndexOutputPort	Specify that an output port emits one-based indices.
ssSetOutputPortComplexSignal	Specify the numeric type (real or complex) of this port.
ssSetOutputPortDataType	Specify the data type of an output port.
ssSetOutputPortDimensionInfo	Specify the dimensionality of an output port.
ssSetOutputPortFrameData	Specify whether a port outputs framed data.
ssSetOutputPortMatrixDimensions	Specify dimension information for an output port that emits matrix signals.
ssSetOutputPortOffsetTime	Specify the sample time offset value of an output port.
ssSetOutputPortSampleTime	Specify the sample time of an output port.
ssSetOutputPortVectorDimension	Specify dimension information for an output port that emits vector signals.

I/O Port (Continued)

Macro	Description
ssSetOutputPortWidth	Specify the width of a 1-D (vector) output port.
ssSetOutputPortMatrixDimensions	Specify the dimensions of a 2-D (matrix) signal.
ssSetOutputPortVectorDimension	Specify the dimension of a 1-2 (vector) signal.
ssSetVectorMode	Specify the vector mode that an S-function supports.
ssSetZeroBasedIndexInputPort	Specify that an input port expects zero-based indices.
ssSetZeroBasedIndexOutputPort	Specify that an output port emits zero-based indices.

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.

Dialog Box Parameters

Macro	Description
ssGetDTypeIdFromMxArray	Get the Simulink data type of a dialog parameter.
ssGetNumParameters	Get the number of parameters that this block has (Ada only).
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.
ssGetSFcnParamsCount	Get the actual number of parameters specified by the user.

Dialog Box Parameters (Continued)

Macro	Description
ssSetNumSFcnParams	Set the number of parameters that an S-function expects.
ssSetParameterName	Set the name of a parameter (Ada only).
ssSetParameterTunable	Set the tunability of a parameter (Ada only).
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.

Run-Time Parameters

Macro	Description
ssGetNumRunTimeParams	Get the number of run-time parameters created by this S-function.
ssGetRunTimeParamInfo	Get the attributes of a specified run-time parameter.
ssRegAllTunableParamsAsRunTimeParams	Register all tunable dialog parameters as run-time parameters.
ssRegDlgParamAsRunTimeParam	Register a run-time parameter.

Run-Time Parameters (Continued)

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

Sample Time

Macro	Description
ssGetInputPortSampleTime	Determine the sample time of an input port.
ssGetInputPortSampleTimeIndex	Get the sample time index of an input port.

Sample Time (Continued)

Macro	Description
<code>ssGetNumSampleTimes</code>	Get the number of sample times an S-function has.
<code>ssGetOutputPortSampleTime</code>	Determine the sample time of an output port.
<code>ssGetPortBasedSampleTimeBlockIsTriggered</code>	Determine whether a block that uses port-based sample times resides in a triggered subsystem.
<code>ssGetSampleTimeOffset</code>	Get the offset of the current sample time (Ada only).
<code>ssGetSampleTimePeriod</code>	Get the period of the current sample time (Ada only).
<code>ssGetTNext</code>	Get the time of the next sample hit in a discrete S-function with a variable sample time.
<code>ssIsContinuousTask</code>	Determine whether a specified rate is the continuous rate.
<code>ssIsSampleHit</code>	Determine the sample rate at which an S-function is operating.
<code>ssIsSpecialSampleHit</code>	Determine whether the current sample time hits two specified rates.
<code>ssSampleAndOffsetAreTriggered</code>	Determine whether a sample time and offset value pair indicate a triggered sample time.
<code>ssSetInputPortSampleTime</code>	Set the sample time of an input port.

Sample Time (Continued)

Macro	Description
ssSetModelReferenceSampleTimeInheritance	Specify whether use of an S-function in a submodel prevents the submodel from inheriting its sample time from the parent model.
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 the time of the next sample hit in an S-function.

State and Work Vector

Macro	Description
ssGetContStateAddress	Get the address of a block's continuous state vector.
ssGetContStates	Get an S-function's continuous states.
ssGetDiscStates	Get an S-function's discrete states.
ssGetDWork	Get a DWork vector.
ssGetDWorkComplexSignal	Determine whether the elements of a data type work vector are real or complex numbers.
ssGetDWorkDataType	Get the data type of a data type work vector.
ssGetDWorkName	Get the name of a data type work vector.
ssGetDWorkUsedAsDState	Determine whether a data type work vector is used as a discrete state vector.

State and Work Vector (Continued)

Macro	Description
ssGetDWorkWidth	Get the size of a data type work vector.
ssGetdX	Get the derivatives of the continuous states of an S-function.
ssGetIWork	Get an S-function's integer-valued (<code>int_T</code>) work vector.
ssGetIWorkValue	Get a value from a block's integer work vector.
ssGetModeVector	Get an S-function's mode work vector.
ssGetModeVectorValue	Get an element of a block's mode vector.
ssGetNonsampledZCs	Get an S-function's zero-crossing signals vector.
ssGetNumContStates	Determine the number of continuous states that an S-function has.
ssGetNumDiscStates	Determine the number of discrete states that an S-function has.
ssGetNumDWork	Get the number of data type work vectors used by a block.
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 (<code>real_T</code>) work vector.
ssGetPWork	Get an S-function's pointer (<code>void *</code>) work vector.

State and Work Vector (Continued)

Macro	Description
<code>ssGetPWorkValue</code>	Get a pointer from a pointer work vector.
<code>ssGetRealDiscStates</code>	Get the real (<code>real_T</code>) values of an S-function's discrete state vector.
<code>ssGetRWork</code>	Get an S-function's real-valued (<code>real_T</code>) work vector.
<code>ssGetRWorkValue</code>	Get an element of an S-function's real-valued work vector.
<code>ssSetDWorkComplexSignal</code>	Specify whether the elements of a data type work vector are real or complex.
<code>ssSetDWorkDataType</code>	Specify the data type of a data type work vector.
<code>ssSetDWorkName</code>	Specify the name of a data type work vector.
<code>ssSetDWorkUsedAsDState</code>	Specify that a data type work vector is used as a discrete state vector.
<code>ssSetDWorkWidth</code>	Specify the width of a data type work vector.
<code>ssSetIWorkValue</code>	Set an element of a block's integer work vector.
<code>ssSetModeVectorValue</code>	Set an element of a block's mode vector.
<code>ssSetNumContStates</code>	Specify the number of continuous states that an S-function has.
<code>ssSetNumDiscStates</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.

State and Work Vector (Continued)

Macro	Description
ssSetNumNonsampledZCs	Specify the number of zero crossings that an S-function detects.
ssSetNumPWork	Specify the size of an S-function's pointer (void *) work vector.
ssSetNumRWork	Specify the size of an S-function's real (real_T) work vector.
ssSetPWorkValue	Set an element of a block's pointer work vector.
ssSetRWorkValue	Set an element of a block's floating-point work vector.

Simulation Information

Macro	Description
ssGetAbsTol	Get the absolute tolerances used by a model's variable-step solver.
ssGetBlockReduction	Determine whether a block has requested block reduction before the simulation has begun and whether it has actually been reduced after the simulation loop has begun.
ssGetErrorStatus	Get a string that identifies the last error.
ssGetInlineParameters	Determine whether the user has set the inline parameters option for the model containing this S-function.
ssGetSimMode	Determine the context in which an S-function is being invoked: normal simulation, external-mode simulation, model editor, etc.
ssGetSolverMode	Get the solver mode being used to solve the S-function.

Simulation Information (Continued)

Macro	Description
ssGetSolverName	Get the name of the solver being used for the simulation.
ssGetStateAbsTol	Get the absolute tolerance used by the model's variable-step solver for a specified state.
ssGetStopRequested	Get the value of the simulation stop requested flag.
ssGetT	Get the current base simulation time.
ssGetTaskTime	Get the current time for a task.
ssGetTFinal	Get the end time of the current simulation.
ssGetTNext	Get the time of the next sample hit.
ssGetTStart	Get the start time of the current simulation.
ssIsFirstInitCond	Determine whether this is the first call to mdlInitializeConditions.
ssIsMajorTimeStep	Determine whether the current time step is a major time step.
ssIsMinorTimeStep	Determine whether the current time step is a minor time step.
ssIsVariableStepSolver	Determine whether the current solver is a variable-step solver.
ssSetBlockReduction	Request that Simulink attempt to reduce a block.

Simulation Information (Continued)

Macro	Description
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.
ssDisableSystemWithTid	Disable a function-call subsystem connected to this S-function block.
ssEnableSystemWithTid	Enable a function-call subsystem connected to this S-function.
ssSetCallSystemOutput	Specify that an output port element issues a function call.
ssSetExplicitFCSSCtrl	Specify whether an S-function explicitly enables and disables the function-call subsystem that it calls.

Data Type

Macro	Description
ssGetDataTypeId	Get the ID for a data type.
ssGetDataTypeName	Get a data type's name.
ssGetDataTypeSize	Get a data type's size.
ssGetDataTypeZero	Get the zero representation of a data type.
ssGetInputPortDataType	Get the data type of an input port.
ssGetNumDataTypes	Get the number of data types defined by an S-function or the model.

Data Type (Continued)

Macro	Description
ssGetOutputPortDataType	Get the data type of an output port.
ssGetOutputPortSignal	Get an output signal of any type except double.
ssRegisterDataType	Register a data type.
ssSetDataTypeSize	Specify the size of a data type.
ssSetDataTypeZero	Specify the zero representation of a data type.
ssSetInputPortDataType	Specify the data type of signals accepted by an input port.

Real-Time Workshop

Macro	Description
ssGetDWorkRTWIdentifier	Get the identifier used to declare a DWork vector in code generated from the associated S-function.
ssGetDWorkRTWStorageClass	Get the storage class of a DWork vector in code generated from the associated S-function.
ssGetDWorkRTWTypeQualifier	Get the C type qualifier (e.g., const) used to declare a DWork vector in code generated from the associated S-function.
ssSetDWorkRTWTypeQualifier	Set the identifier used to declare a DWork vector in code generated from the associated S-function.
ssGetPlacementGroup	Get the name of the placement group of a block.

Real-Time Workshop (Continued)

Macro	Description
ssSetDWorkRTWIdentifier	Set the storage class of a DWork vector in code generated from the associated S-function.
ssSetDWorkRTWStorageClass	Set the C type qualifier (e.g., const) used to declare a DWork vector in code generated from the associated S-function.
ssSetPlacementGroup	Specify the name of the placement group of a block.
ssWriteRTW2dMatParam	Write a Simulink matrix parameter to the S-function's <i>model.rtw</i> file.
ssWriteRTWMx2dMatParam	Write a MATLAB matrix parameter to the S-function's <i>model.rtw</i> file.
ssWriteRTWMxVectParam	Write a MATLAB vector parameter to the S-function's <i>model.rtw</i> file.
ssWriteRTWParameters	Write tunable parameters to the S-function's <i>model.rtw</i> file.
ssWriteRTWParamSettings	Write settings for the S-function's parameters to the <i>model.rtw</i> file.
ssWriteRTWScalarParam	Write a scalar parameter to the S-function's <i>model.rtw</i> file.
ssWriteRTWStr	Write a string to the S-function's <i>model.rtw</i> file.
ssWriteRTWStrParam	Write a string parameter to the S-function's <i>model.rtw</i> file.
ssWriteRTWStrVectParam	Write a string vector parameter to the S-function's <i>model.rtw</i> file.

Real-Time Workshop (Continued)

Macro	Description
ssWriteRTWVectParam	Write a Simulink vector parameter to the S-function's <i>model.rtw</i> file.
ssWriteRTWorkVect	Write the S-function's work vectors to the <i>model.rtw</i> file.

A

- Ada S-functions
 - creating 5-3
 - example 5-11
 - GNAT Ada95 compiler 5-10
 - mex syntax 5-10
 - source file format 5-3
 - specification 5-3
- additional parameters for M-file S-functions 2-11
- array bounds
 - checking 7-43

B

- block I/O ports 7-10
- block-based sample times 7-18
 - specifying 7-18
- Build Info pane
 - S-Function Builder 3-27

C

- C language header file
 - matlabroot/simulink/include/simstruc.h 9-2
- C MEX S-functions
 - advantages 3-2
 - converting from level 1 to level 2 3-48
 - creating 3-3
 - definition 1-2
 - example 3-29
 - modes for compiling 3-37
 - S-Function Builder 3-5
 - Simulink interaction 3-39
- C++ objects
 - making persistent 4-6
- C++ S-functions
 - building 4-8
 - mex command 4-8
- C-to-Fortran gateway S-function 6-7

- callback methods 1-9
- CFortran 6-10
- cg_sfun.h 3-37
- checking array bounds 7-43
- compiler compatibility
 - Fortran 6-7
- continuous blocks
 - setting sample time 7-27
- Continuous Derivatives pane
 - S-Function Builder 3-24
- continuous state S-function example (C MEX) 7-44
- creating persistent C++ objects 4-6

D

- data types
 - using user-defined 7-16
- direct feedthrough 1-12
- discrete state S-function example (C MEX) 7-49
- Discrete Update pane
 - S-Function Builder 3-25
- dynamically sized inputs 1-13

E

- error handling
 - checking array bounds 7-43
 - exception free code 7-41
- examples
 - Ada S-function specification 5-3
 - C MEX S-function 3-29
 - continuous state S-function (C MEX) 7-44
 - discrete state S-function (C MEX) 7-49
 - Fortran MEX S-function 6-3
 - hybrid system S-function (C MEX) 7-54
 - pointer work vector 7-33
 - sample time for continuous block 7-27
 - sample time for hybrid block 7-27

- time-varying continuous transfer function
(C MEX) 7-78
- variable-step S-function (C MEX) 7-59
- zero-crossing S-function (C MEX) 7-63

exception free code 7-41

extern "C" statement 4-2

F

Fortran compilers 6-11

Fortran math library 6-9

Fortran MEX S-functions

- example 6-3
- template file 6-3

frame-based signals, implementing in
S-functions 7-39

function-call subsystems 7-36

H

header files 3-36

hybrid blocks

- setting sample time 7-27

hybrid sample times

- specifying 7-25

hybrid system S-function example (C
MEX) 7-54

I

Initialization pane

- S-Function Builder 3-11

input ports

- how to create 7-10

inputs, dynamically sized 1-13

L

level 1 C MEX S-functions

- converting to level 2 3-48

Libraries pane

S-Function Builder 3-18

M

M-file S-functions

- arguments 2-8
- creating 2-8
- defining characteristics 2-10
- definition 2-8
- passing additional parameters 2-11
- routines 2-8

masked multiport S-functions 7-15

matlabroot/simulink/include/simstruc.h

- C language header file 9-2

matlabroot/simulink/src/csfunc.c

- example file 7-45

matlabroot/simulink/src/dsfunc.c

- example file 7-49

matlabroot/simulink/src/mixedm.c

- example file 7-54

matlabroot/simulink/src/sfun_counter_cpp.cpp

- ensuring Simulink compatibility of C++
S-functions 4-2

matlabroot/simulink/src/sfun_timestwo_for.for

- Fortran example file 6-3

matlabroot/simulink/src/sfun_zc_sat.c

- example file 7-63

matlabroot/simulink/src/stvctf.c

- example file 7-78

matlabroot/simulink/src/vsfunc.c

- example file 7-59

matrix.h 3-36

mdlCheckParameters 8-3

mdlDerivatives 8-6

mdlGetTimeOfNextVarHit 8-9

mdlInitializeConditions 8-10

mdlInitializeSampleTimes 8-12

mdlInitializeSizes 8-16

- and sizes structure 1-13
- calling sizes 2-10

- mdlOutputs 8-21
- mdlProcessParameters 8-22
- mdlRTW 8-25
- mdlSetDefaultPortComplexSignals 8-27
- mdlSetDefaultPortDataTypes 8-28
- mdlSetDefaultPortDimensionInfo 8-29
- mdlSetInputPortComplexSignal 8-30
- mdlSetInputPortDataType 8-32
- mdlSetInputPortDimensionInfo 8-34
- mdlSetInputPortFrameData 8-36
- mdlSetInputPortSampleTime 8-38
- mdlSetInputPortWidth 8-41
- mdlSetOutputPortComplexSignal 8-42
- mdlSetOutputPortDataType 8-44 8-53
- mdlSetOutputPortDimensionInfo 8-46
- mdlSetOutputPortSampleTime 8-48
- mdlSetOutputPortWidth 8-50
- mdlSetWorkWidths 8-51
- mdlStart 8-54
- mdlTerminate 8-55
- mdlUpdate 8-56
- mdlZeroCrossings 8-58
- memory allocation 7-35
- memory and work vectors 7-31
- mex command
 - building Ada S-functions 5-10
 - building C MEX S-functions 3-34
 - building C++ S-functions 4-8
- mex.h 3-36
- multirate S-Function blocks 7-26
 - synchronizing 7-28

O

- obsolete macros 3-50
- output ports
 - how to create 7-12
- Outputs pane
 - S-Function Builder 3-20

P

- parameters
 - M-file S-functions 2-11
 - passing to S-functions 1-4
 - run-time parameters 7-6
 - tunable parameters 7-3
- penddemo demo 1-5
- persistence
 - C++ objects 4-6
- port-based sample times 7-21
 - constant 7-23
 - inherited 7-22
 - specifying 7-21
 - triggered 7-24

R

- reentrancy 7-31
- run-time parameter names, uniqueness of 7-7
- run-time parameters 7-6
- run-time routines 7-42

S

- S-Function blocks
 - multirate 7-26
 - S-functions parameters field 7-2
 - synchronizing multirate 7-28
- S-Function Builder
 - Build Info pane 3-27
 - Continuous Derivatives pane 3-24
 - customizing 3-9
 - Discrete Update pane 3-25
 - for C MEX S-functions 3-5
 - Initialization pane 3-11
 - Libraries pane 3-18
 - Outputs pane 3-20
 - setting the include path 3-8
- S-function routines
 - M-file 2-8

- S_FUNCTION_LEVEL 2, #define 3-35
 - S_FUNCTION_NAME, #define 3-35
 - S-functions 2-8 3-2 4-1 5-2 6-3
 - building C++ 4-8
 - C MEX 1-2
 - creating Ada 5-3
 - creating C MEX 3-3
 - creating Fortran 6-3
 - creating level 2 with Fortran 6-7
 - creating persistent C++ objects 4-6
 - creating run-time parameters 7-7
 - definition 1-2
 - direct feedthrough 1-12
 - exception free code 7-41
 - level 1 and level 2 6-2
 - masked multiport 7-15
 - purpose 1-5
 - routines 1-8
 - run-time parameters 7-6
 - run-time routines 7-42
 - using in models 1-3
 - when to use 1-5
 - writing in C++ 4-2
 - See also* Ada S-functions; C MEX S-functions; C++ S-functions; Fortran MEX S-functions; M-file S-functions
 - S-functions parameters field
 - S-Function block 7-2
 - sample times
 - block-based 7-18
 - continuous block example 7-27
 - hybrid block example 7-27
 - port-based 7-21
 - specifying block-based 7-17 to 7-18
 - specifying hybrid 7-25
 - specifying port-based 7-21
 - scalar expansion of inputs 7-13
 - sfuntmpl.c template 3-35
 - sfuntmpl_fortran.for template 6-3
 - sfuntmpl.m template
 - M-file S-function 2-8
 - simsizes function
 - M-file S-function 2-10
 - SimStruct 3-37
 - SimStruct macros 9-3
 - simulation loop 1-6
 - simulation stages 1-6
 - simulink.c 3-37
 - sizes structure
 - fields
 - M-file S-function 2-10
 - returned in mdlInitializeSizes 1-13
 - ssGetModeVectorValue 9-14
 - synchronizing multirate S-Function
 - blocks 7-28
- T**
- templates
 - M-file S-function 2-8
 - time-varying continuous transfer function
 - example (C MEX) 7-78
 - tmwtypes.h 3-36
 - tunable parameters 7-3
- V**
- variable-step S-function example (C MEX) 7-59
- W**
- work vectors 7-31
 - writing S-functions in Ada 5-3
 - writing S-functions in C++ 4-2
 - writing S-functions in MATLAB 2-8
- Z**
- zero-crossing S-function example (C MEX) 7-63