SIMULINK®

Model-Based and System-Based Design

Modeling

Simulation

Implementation

Writing S-Functions

Version 5

The MathWorks

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

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

Writing S-Functions

Printing History:			
	October 1998	First printing	Revised for Simulink 3.0 (Release 11)
	November 2000	Second printing	Revised for Simulink 4.0 (Release 12)
	June 2001	Online only	Revised for Simulink 4.1 (Release 12.1)
	July 2002	Online only	Revised for Simulink 5.0 (Release 13)

Contents

1 Overview of S-Functions

2 Writing M S-Functions

3 Writing S-Functions in C

4 Creating C++ S-Functions

5 Creating Ada S-Functions

6 Creating Fortran S-Functions

7 Implementing Block Features

8 Writing S-Functions for Real-Time Workshop

9 S-Function Callback Methods

10 SimStruct Functions

1

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?

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 online MATLAB documentation). As with other MEX-files, they are dynamically linked into MATLAB when needed.

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

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 Functions & Tables block library), you can customize the user interface by using masking.

You can use S-functions with the Real-Time Workshop®. You can also customize the code generated by the Real Time Workshop for S-functions by writing a Target Language Compiler™ (TLC) file. See "Writing S-Functions for Real-Time Workshop" on page 8-1 and the Real-Time Workshop documentation for more information.

Using S-Functions in Models

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

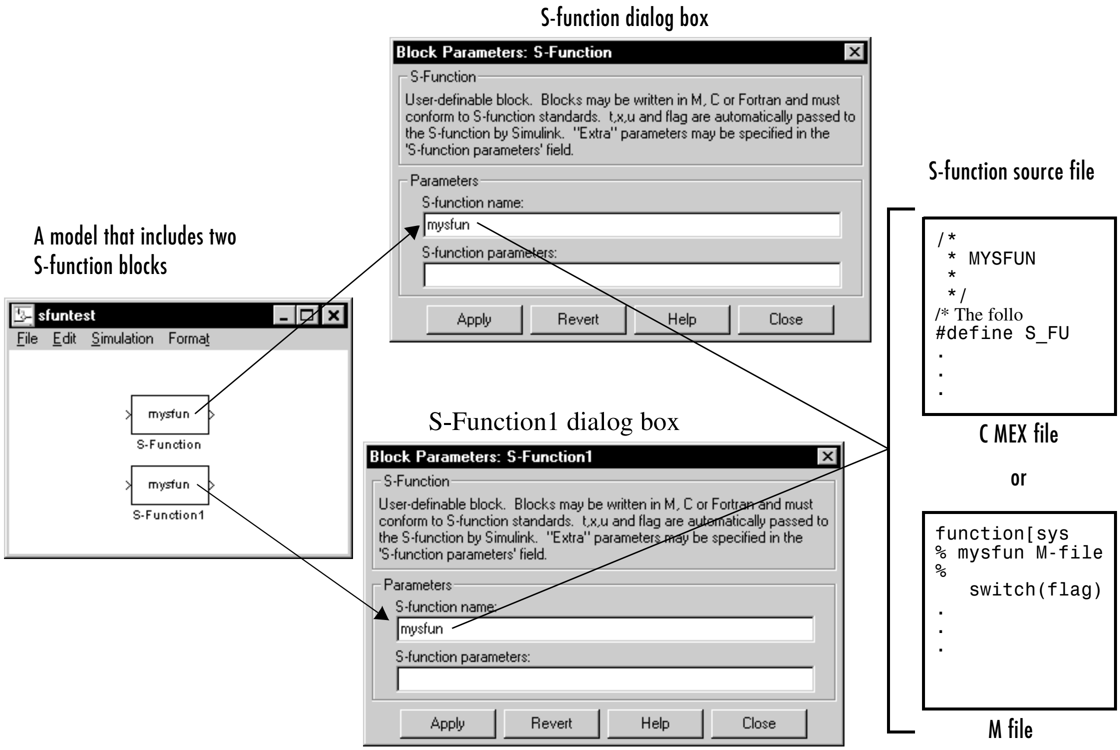


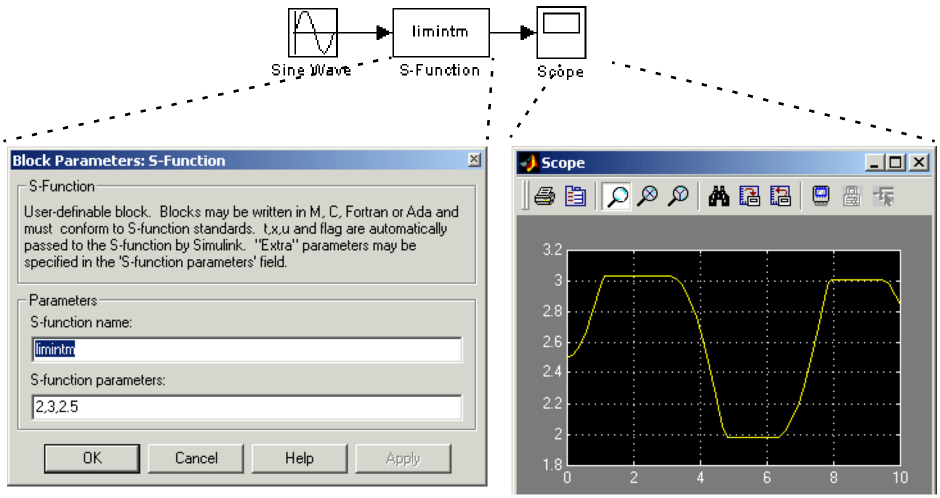
Figure 1-1: Relationship Between an S-Function Block, Its Dialog Box, and the Source File That Defines the Block's Behavior

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 2-6 and "Handling Errors" on page 7-33 for information on how to access user-specified parameters in an S-function.

You can use Simulink's 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 *Using Simulink*.

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`)

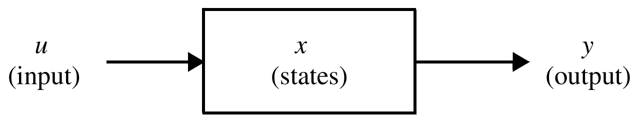
An advantage of using S-functions is that you can build a general purpose block that you can use many times in a model, varying parameters with each instance of the block.

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.

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

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

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

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.

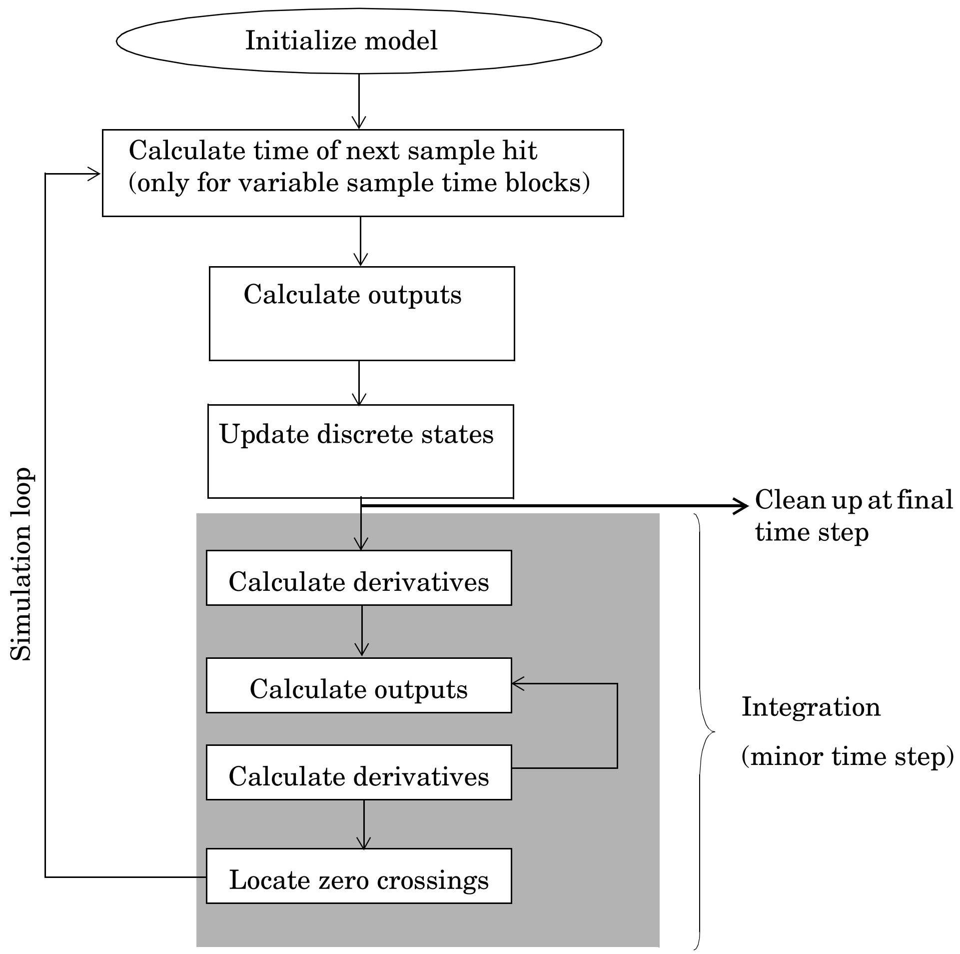


Figure 1-2: How Simulink Performs Simulation

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	`mdlInitializeSizes`	`flag = 0`
Calculation of next sample hit (variable sample time block only)	`mdlGetTimeOfNextVarHit`	`flag = 4`
Calculation of outputs	`mdlOutputs`	`flag = 3`
Update of discrete states	`mdlUpdate`	`flag = 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 M S-Functions,” 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 $\dot{x}$ 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.

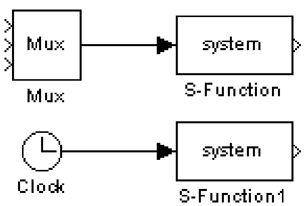
Dynamically Sized Arrays

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 7-8.

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* 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

  ```
  TimeHit = (n * period) + 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 "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.

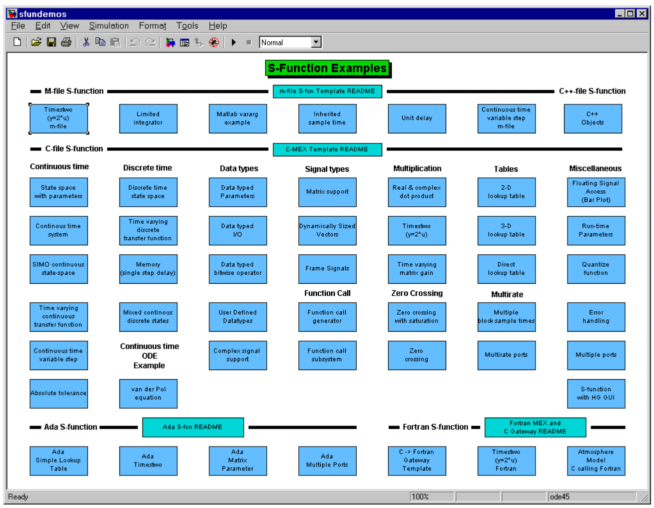
S-Function Examples

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 an S-function example.

2 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.
`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`	Example state-space M-file S-function with internal `A`, `B`, `C`, and `D` matrices. This S-function implements
	`dx/at = Ax + By`
	`y = Cx + Du`
	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.

`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.
`sfun_varargm.m`	Example M-file S-function showing how to use the MATLAB `vararg` facility.

Filename	Description
`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.
`mixedm.c`	Implement a hybrid dynamic system consisting of a continuous integrator (1/s) in series with a unit delay (1/z).
`mixedmex.c`	Implement a hybrid dynamic system with a single output and two inputs.

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

Filename	Description
`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 `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.
`sfunmem.c`	A one-integration-step delay and hold memory function.

Filename	Description

`simomex.c`	Implements a single-output, two-input state-space dynamic system described by these state-space equations
	`dx/dt = Ax + Bu`
	`y = Cx + Du`
	where `x` is the state vector, `u` is vector of inputs, and `y` is the vector of outputs.

`smatrxcat.c`	Matrix concatenation.
`sreshape.c`	Reshape the input signal.
`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.
`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.

Filename	Description
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
sfun_counter_cpp.cpp	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
matrix_gain	Implement a Matrix Gain block.
multi_port	Multiport block.
simple_lookup	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 RTW Ada Coder.
times_two	Output twice its input.

2

Writing M S-Functions

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

Introduction

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 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.

Table 2-1: Flag Argument

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

S-Function Outputs

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

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

- `str`, reserved for future use. M-file S-functions must set this to the empty matrix, `[]`.
- `ts`, a two-column matrix containing the sample times and offsets of the block (see "Specifying Sample Time" in the online documentation 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. See "Example - Hybrid System S-Function" on page 2-14) for an example of how to implement such a hybrid block.

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.

Table 2-2: 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.

Examples of M-File S-Functions

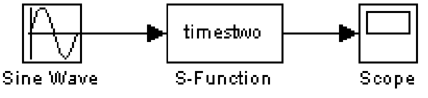
The easiest way to understand how S-functions work is to look at examples. This section starts off with a s simple example (`timestwo`) that has no states. Most S-Function blocks require the handling of states, whether continuous or discrete. The sections that follow discuss four common types of systems you can model in Simulink using S-functions:

- Continuous
- Discrete
- Hybrid
- Variable-step

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

Simple M-File S-Function Example

This block takes an input scalar signal, doubles it, and plots it to a scope.



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

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

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

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

   case 3
```

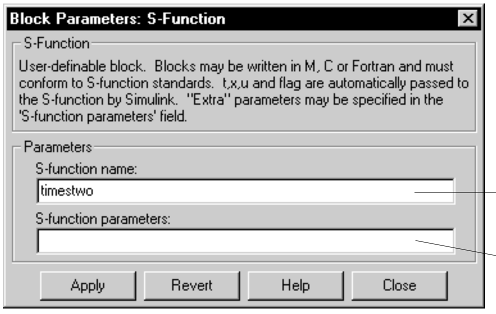
```
    sys = mdlOutputs(t,x,u); % Calculate outputs

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

  otherwise
    error(['Unhandled flag = ',num2str(flag)]); % Error handling
end;
% End of function timestwo.
```

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

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

```
% End of mdlOutputs.
```

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



Enter the function name here. In this example, enter `timestwo`.

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

You can now run this simulation.

Example - Continuous State S-Function

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

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

```
D=[-3    0
    1    0];
%
% Dispatch the flag.
%
switch flag,

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

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

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

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

```
sys = simsizes(sizes);
%
% Initialize the initial conditions.
%
x0 = zeros(2,1);
%
% str is an empty matrix.
%
str = [];
%
% Initialize the array of sample times; in this example the sample
% time is continuous, so set ts to 0 and its offset to 0.
%
ts = [0 0];
% End of mdlInitializeSizes.
%
%==============================================================
% mdlDerivatives
% Return the derivatives for the continuous states.
%==============================================================
function sys = mdlDerivatives(t,x,u,A,B,C,D)
sys = A*x + B*u;
% End of mdlDerivatives.
%
%==============================================================
% mdlOutputs
% Return the block outputs.
%==============================================================
%
function sys = mdlOutputs(t,x,u,A,B,C,D)
sys = C*x + D*u;
% End of mdlOutputs.
```

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

```
x'= Ax + Bu
y = Cx + Du
```

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

Each time the `mdlDerivatives` routine is called it must explicitly set the 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.

Example - Discrete State S-Function

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

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

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

  case 2
```

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

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

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

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

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

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

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

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

%==============================================================
% Update the discrete states
%==============================================================
function sys = mdlUpdates(t,x,u,A,B,C,D)
sys = A*x + B*u;
```

```
% End of mdlUpdate.

%==============================================================
% Calculate outputs
%==============================================================
function sys = mdlOutputs(t,x,u,A,B,C,D)
sys = C*x + D*u;
% End of mdlOutputs.
```

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

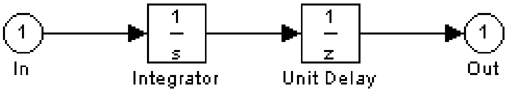
```
x(n+1) = Ax(n) + Bu(n)
y(n)   = Cx(n) + Du(n)
```

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

Example - Hybrid System S-Function

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

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



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

```
function [sys,x0,str,ts] = mixedm(t,x,u,flag)
```

```
% A hybrid system example that implements a hybrid system
% consisting of a continuous integrator (1/s) in series with a
% unit delay (1/z).
%
% Set the sampling period and offset for unit delay.
dperiod = 1;
doffset = 0;
switch flag,

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

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

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

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

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

```
sizes.NumSampleTimes = 2;
sys = simsizes(sizes);
x0  = ones(2,1);
str = [];
ts  = [0,       0           % sample time
       dperiod, doffset];
% End of mdlInitializeSizes.
%

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

```
% mdlOutputs
% Return the output vector for the S-function.
%=============================================================
%
function sys = mdlOutputs(t,x,u,doffset,dperiod)
% Return output of the unit delay if we have a
% sample hit within a tolerance of 1e-8. If we
% don't have a sample hit then return [] indicating
% that the output shouldn't change.
%
if abs(round((t-doffset)/dperiod)-(t-doffset)/dperiod) < 1e-8
  sys = x(2);

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

% End of mdlOutputs.
```

Example - Variable Sample Time S-Function

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

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

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

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

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

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

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

```
%
x0 = [0];
%
% Set str to an empty matrix.
%
str = [];
%
% Initialize the array of sample times.
%
ts = [-2 0];      % variable sample time
% End of mdlInitializeSizes.
%
%==============================================================
% mdlUpdate
% Handle discrete state updates, sample time hits, and major time
% step requirements.
%==============================================================
%
function sys = mdlUpdate(t,x,u)
sys = u(1);
% End of mdlUpdate.
%
%==============================================================
% mdlOutputs
% Return the block outputs.
%==============================================================
%
function sys = mdlOutputs(t,x,u)
sys = x(1);
% end mdlOutputs
%
%==============================================================
% mdlGetTimeOfNextVarHit
% Return the time of the next hit for this block.  Note that the
% result is absolute time.
%==============================================================
%
function sys = mdlGetTimeOfNextVarHit(t,x,u)
sys = t + u(2);
% End of mdlGetTimeOfNextVarHit.
```

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

3

Writing S-Functions in C

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

Introduction

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

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

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

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

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

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

static void mdlInitializeSizes(SimStruct *S)
```

```
{
}

<additional S-function routines/code>

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

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

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

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 S-function Builder, however, is limited in the kinds of S-functions that it can build. For example, it cannot build S-functions that have more than one input or output or that must handle data types other than `double`. You must create such S-functions 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 3-25 provides a step-by-step example of how to write a simple S-function from scratch.
- “Templates for C S-Functions” on page 3-31 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

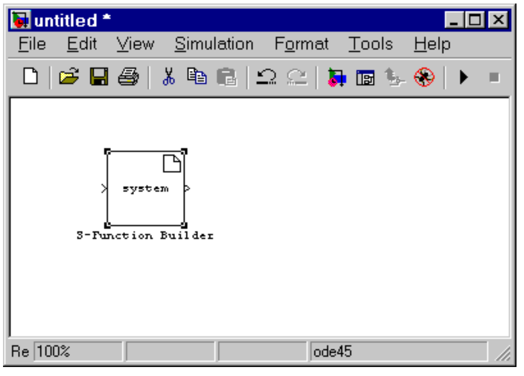
The S-Function Builder is a Simulink block that builds an S-function from specifications and C code that you supply. The S-Function Builder also serves as a wrapper for the generated S-function in models that use the S-function. This section explains how to use the S-Function Builder to build simple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

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

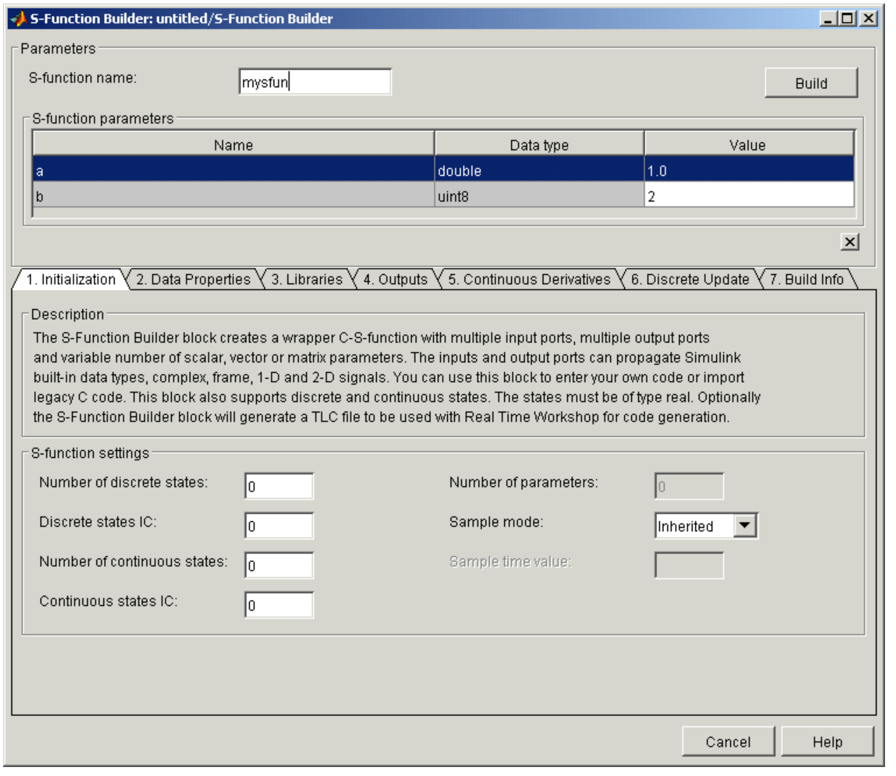
1 Set the MATLAB current directory to the directory in which you want to create the S-function.

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.



5 Enter the name of the S-function in the **S-function name** field.

6 If the S-function has parameters, enter default values for the parameters in the **S-function parameters** field.

7 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 3-8).

8 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.

9 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).

10 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 RTW to include this S-function in the code it generates.

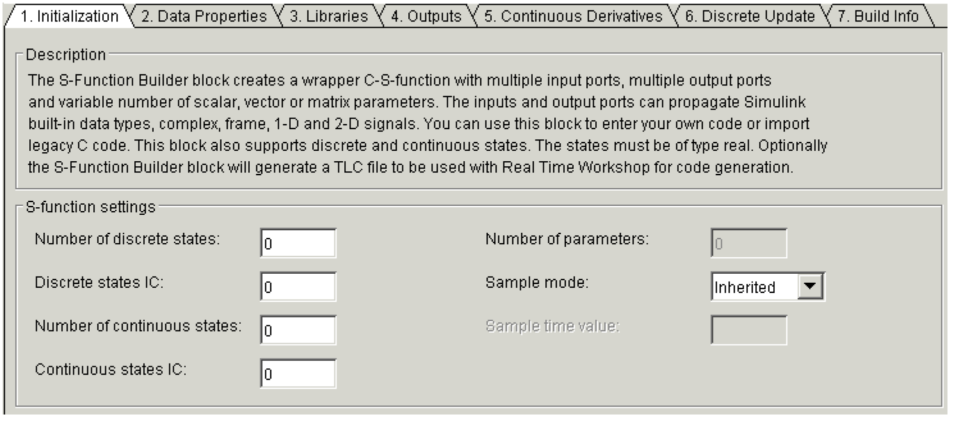
After generating the S-function source code, the S-Function Builder uses the MATLAB `mex` command to build the MEX file representation of the S-function from the generated source code and any external custom source code and libraries that you specified.

S-Function Builder Dialog Box

The tabbed panes on the **S-Function Builder** dialog box enable you to enter information and custom code required to tailor the S-function to a specific application. The dialog box contains the following panes.

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 3-35 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**.

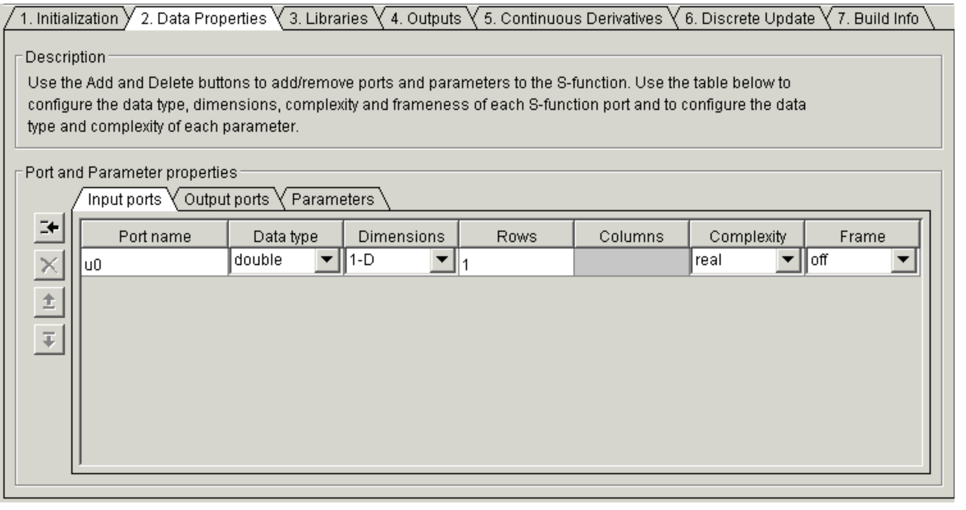
Input port width. Width of the S-function's input port. The width is the number of elements that a vector signal connected to the port must have. To permit connection of matrix (2-D) signals to the input port, specify `-1` as the input port width.

Output port width. Width of the S-function's output port. The width is the number of elements in the vector that this S-function outputs. If the S-function outputs matrix signals, specify `-1` as the port width.

Number of parameters. Number of parameters that this S-function accepts.

Data Properties Pane

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



This pane itself contains tabbed panes that respectively display the attributes of the S-function's

- Input ports (see "Input Ports Pane" on page 3-11)
- Output ports (see "Output Ports Pane" on page 3-12)
- Parameters (see "Parameters Pane" on page 3-13),

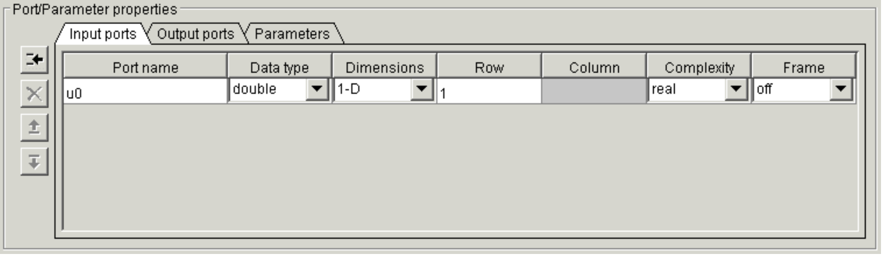
The column of buttons to the left of the panes allows you to add, delete, or reorder ports or parameters, depending on the currently selected pane.

- To add 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).

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.

Data type. Lists the data type of signals accepted by the port. Click the adjacent button to display a list of supported data types. To change the port's data type, select a new type from the list.

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.

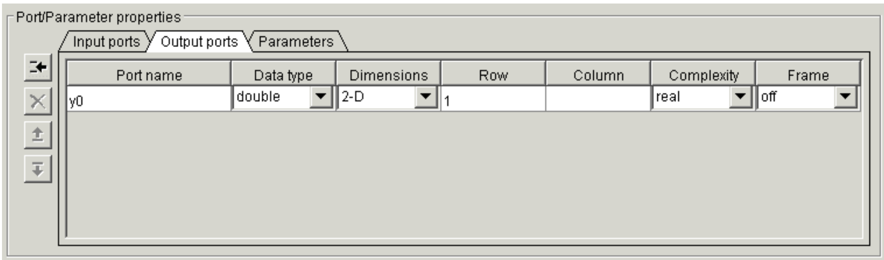
Column. Specifies the size of the input signal's second dimension (only if the input port accepts 2-D signals).

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

Frame. Specifies whether this port accepts frame-based signals generated by the Digital and Signal Processing Blockset and Communications Blockset. See the documentation for these blocksets for more information.

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.

Data type. Lists the data type of signals output by the port. Click the adjacent button to display a list of supported data types. To change the port's data type, select a new type from the list.

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.

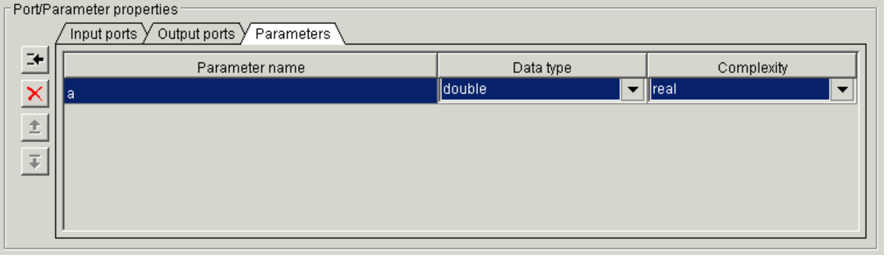
Column. Specifies the size of the output signal's second dimension (only if the output port accepts 2-D signals).

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

Frame. Specifies whether this port accepts frame-based signals generated by the Digital and Signal Processing Blockset and Communications Blockset. See the documentation for these blocksets for more information.

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 port. 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 parmeter as follows.

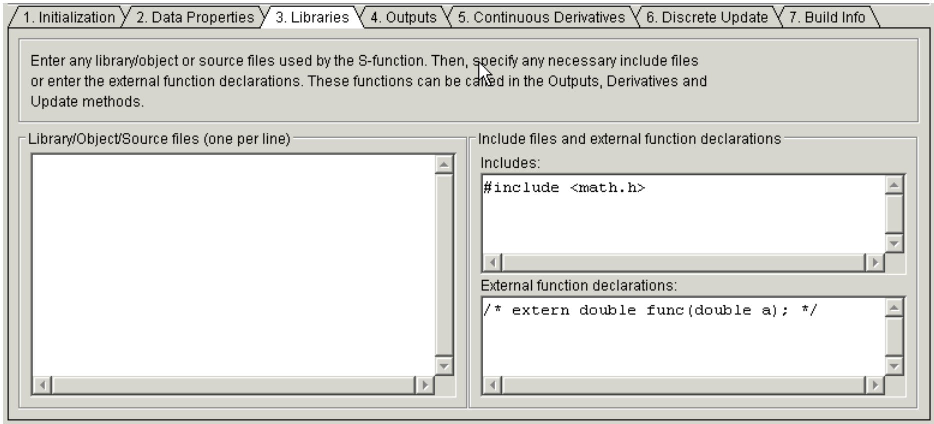
Parameter name. Name of the parameter. Edit this field to change the name.

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.

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.

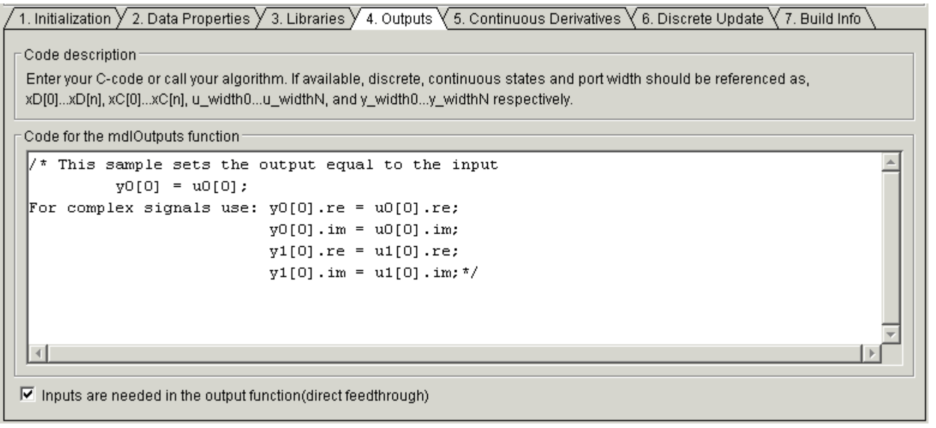
Library/Object/Source files. External library, object code, and source files referenced by custom code that you enter elsewhere on the **S-Function Builder** dialog box. List each file on a separate line. If the file resides in the current directory, you need specify only the file's name. If the file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

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 3-23).

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
`u`	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 `u_width` argument (see below).
`y`	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 `y_width` argument (see below). Use this array to pass the outputs that your code computes back to Simulink.
`xD`	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 3-21 for more information).

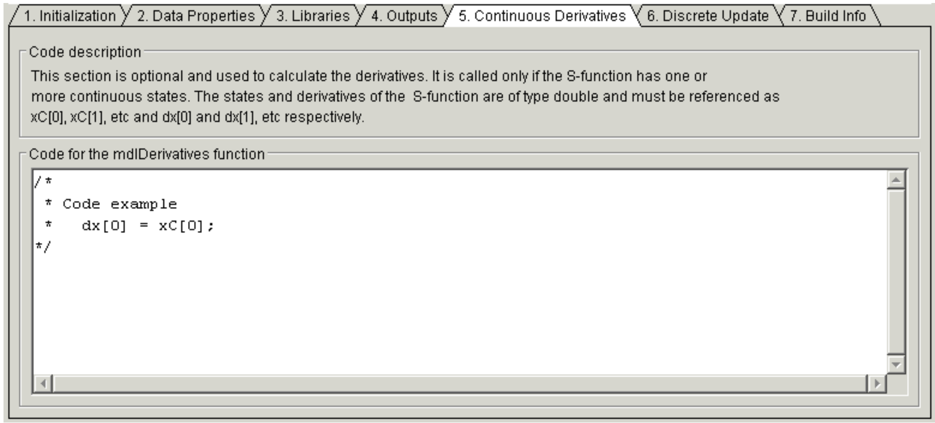
Argument	Description
xC	Pointer to an array containing the continuous states of the S-function. This argument appears only if you specified continuous states on the **Initialization** pane. At the first simulation time step, the continuous states have the initial values that you specified on the **Initialization** 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 3-19 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 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 3-35). Simulink's 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 3-15).

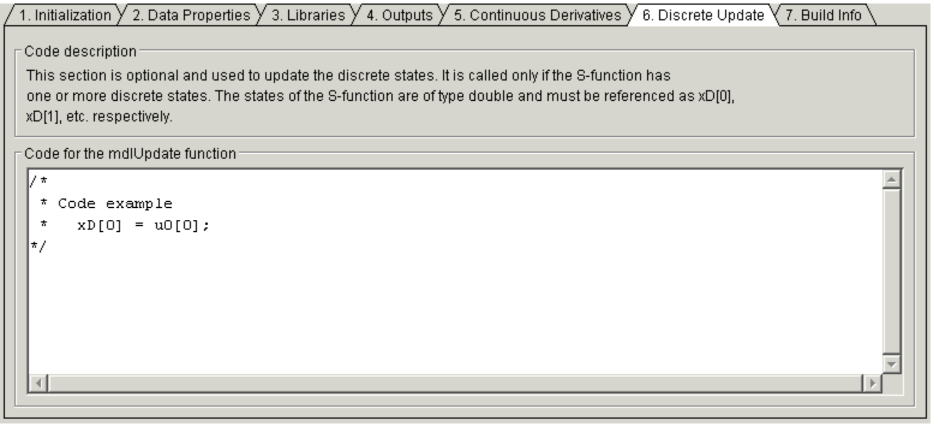
The generated S-function's `mdlDerivatives` callback method passes the following arguments to the derivatives wrapper function:

- `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 3-28 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 3-35). At the next time step, Simulink passes the updated states back to the S-function's `mdlOutputs` method (see "Outputs Pane" on page 3-15).

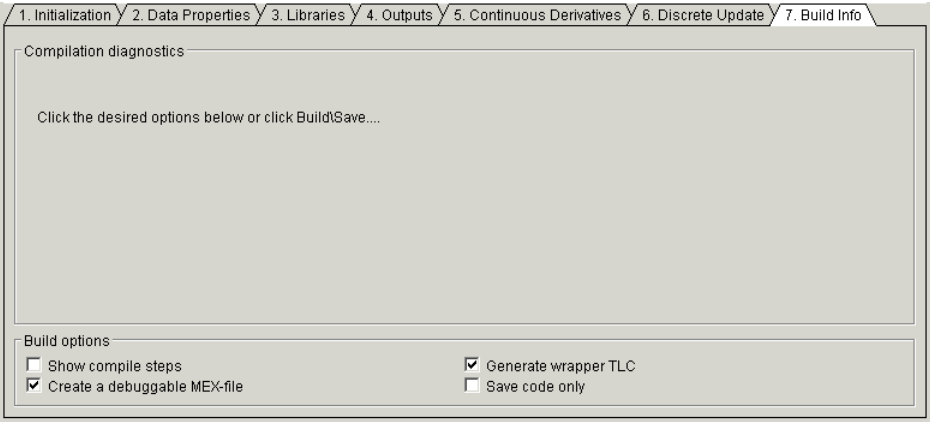
The generated S-function's `mdlUpdates` callback method passes the following arguments to the updates wrapper function:

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

See "mdlOutputs" on page 3-28 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.

Compilation diagnostics. Display diagnostic messages issued by the S-Function Builder when building the S-function.

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 TCL 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 RTW generates code.

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

Setting the Include Path

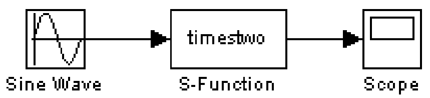
The S-Function Builder searches for custom header files in the directories specified by the MATLAB application data named `SfunctionBuilderIncludePath`. This data is associated with the model in which you create the S-Function Builder block. If your S-function uses custom header files and the custom 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)
```

Example of a Basic C MEX S-Function

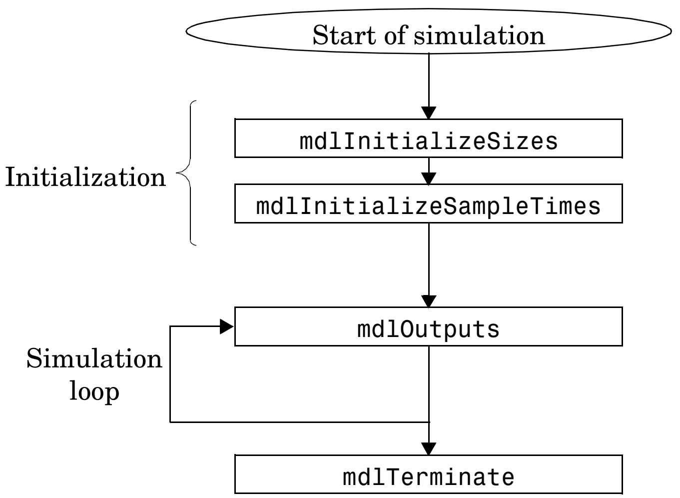
This section presents an example of a C MEX S-function that you can use as a model for creating simple C S-functions. The example is the `timestwo` S-function example that comes with Simulink (see *`matlabroot`*`/simulink/src/timestwo.c`). This S-function outputs twice its input.

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 3-44).

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

Callback Implementations

The next part of the `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-functions'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 3-35.

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 3-39.

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 the 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's use.

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

Templates for C S-Functions

Simulink provides skeleton implementations of C MEX S-functions, called templates, intended to serve as starting points for creating your own S-functions. The templates contain skeleton implementations of callback methods with comments that explain their use. The template file, `sfuntmpl_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 3-44. In addition, the code specifies that you are using the level 2 format of S-functions.

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

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

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

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

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

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

Header File	Description
matlabroot/extern/include/tmwtypes.h	General types, e.g., real_T
matlabroot/rtw/c/libsrc/rt_matrx.h	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 `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`.

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

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

Compiling C S-Functions

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

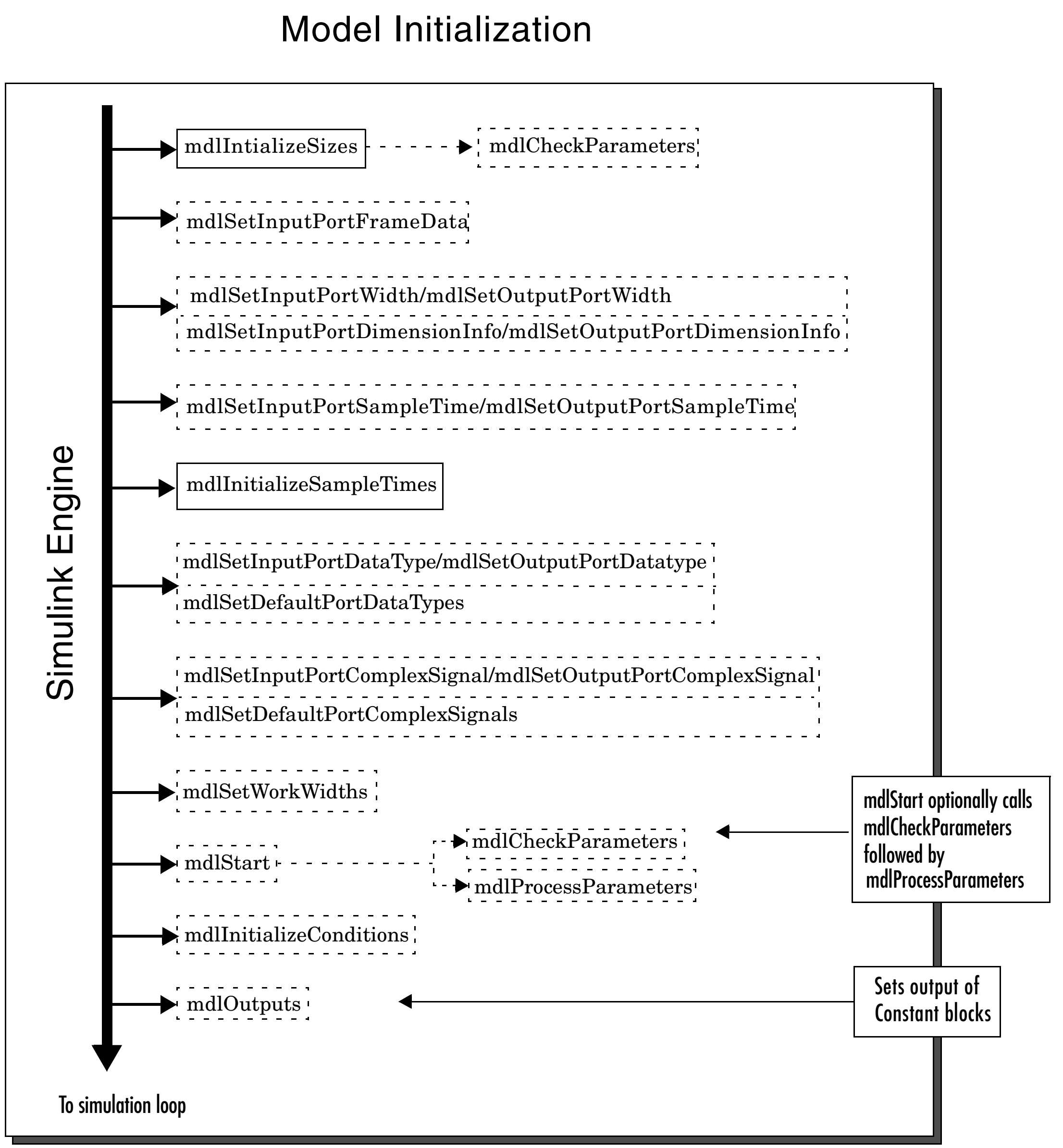
- `MATLAB_MEX_FILE` — Indicates that the S-function is being built as a MEX-file for use with Simulink.
- `RT` — Indicates that the S-function is being built with the Real-Time Workshop generated code for a real-time application using a fixed-step solver.
- `NRT` — Indicates that the S-function is being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

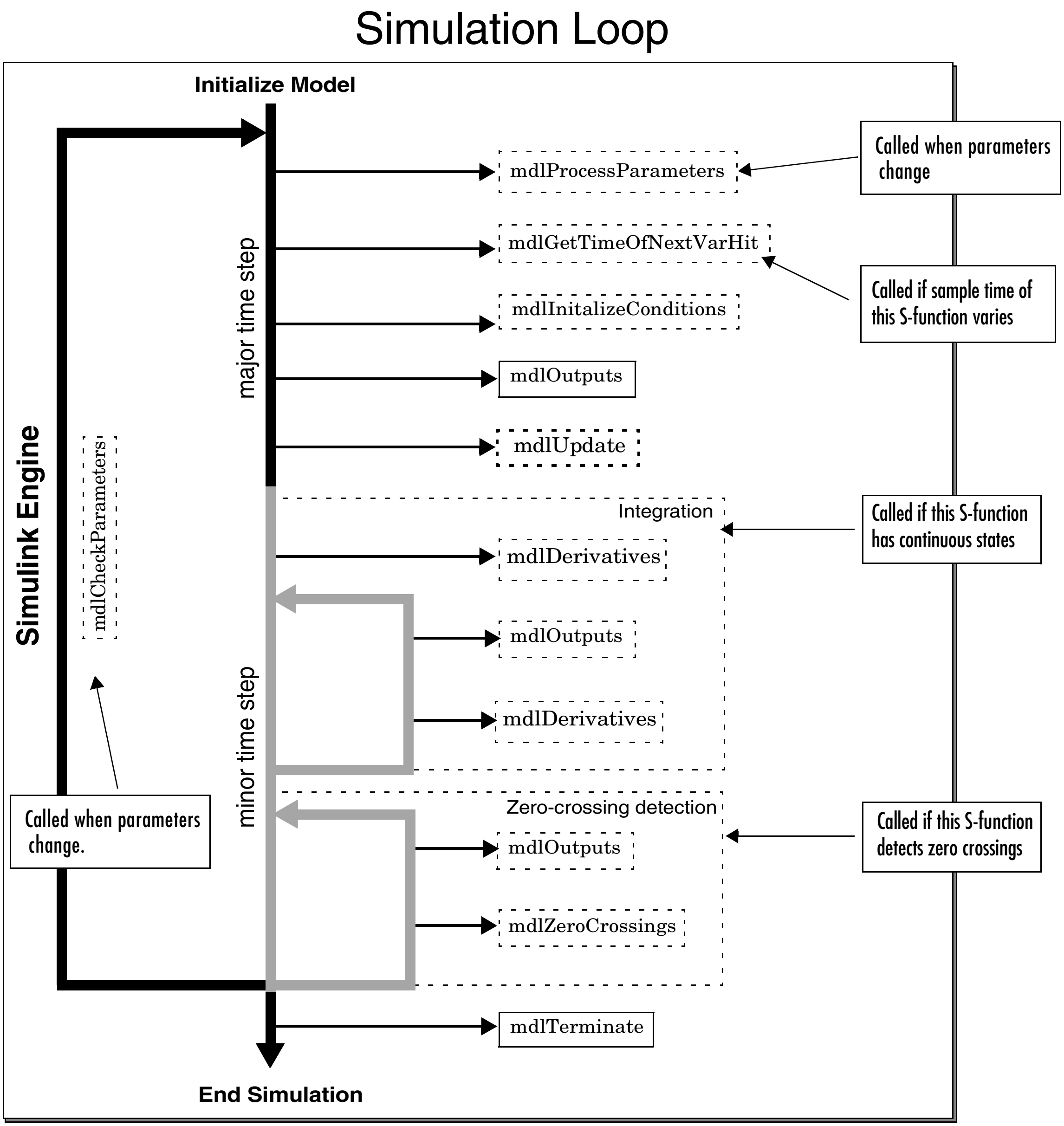
How Simulink Interacts with C S-Functions

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 "S-Function Callback Methods" on page 9-1 to determine the exact circumstances under which Simulink invokes the callback.


Model Initialization
Simulink Engine
mdlIntializeSizes
mdlCheckParameters
mdlSetInputPortFrameData
mdlSetInputPortWidth/mdlSetOutputPortWidth
mdlSetInputPortDimensionInfo/mdlSetOutputPortDimensionInfo
mdlSetInputPortSampleTime/mdlSetOutputPortSampleTime
mdlInitializeSampleTimes
mdlSetInputPortDataType/mdlSetOutputPortDatatype
mdlSetDefaultPortDataTypes
mdlSetInputPortComplexSignal/mdlSetOutputPortComplexSignal
mdlSetDefaultPortComplexSignals
mdlSetWorkWidths
mdlStart
mdlCheckParameters
mdlProcessParameters
mdlStart optionally calls mdlCheckParameters followed by mdlProcessParameters
mdlInitializeConditions
mdlOutputs
Sets output of Constant blocks
To simulation loop


Simulation Loop
Initialize Model
Simulink Engine
mdlCheckParameters
major time step
minor time step
mdlProcessParameters
mdlGetTimeOfNextVarHit
mdlInitalizeConditions
mdlOutputs
mdlUpdate
Integration
mdlDerivatives
mdlOutputs
mdlDerivatives
Zero-crossing detection
mdlOutputs
mdlZeroCrossings
mdlTerminate
End Simulation
Called when parameters change
Called if sample time of this S-function varies
Called if this S-function has continuous states
Called if this S-function detects zero crossings
Called when parameters change.

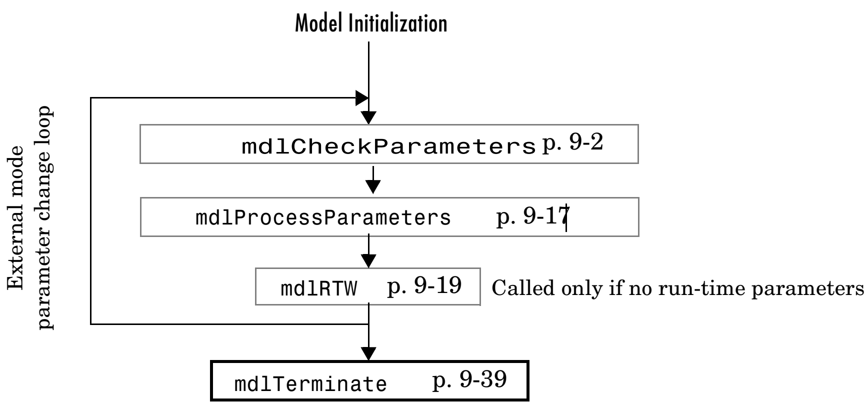
Calling Structure for the Real Time Workshop

When generating code, the 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 the Real-Time Workshop, `mdlTerminate`, and exits.

For more information about the Real-Time Workshop and how it interacts with S-functions, see the *Real-Time Workshop User's Guide* and the *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 the Real-Time Workshop. For more information about external mode, see the *Real-Time Workshop User's Guide*.

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

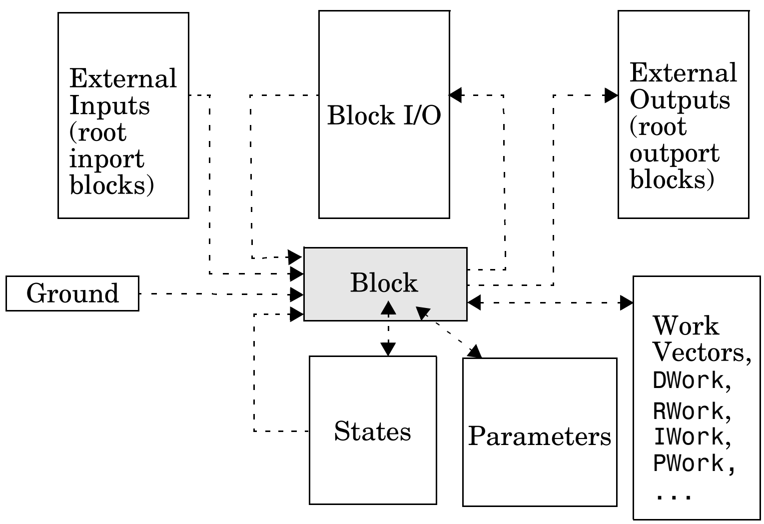
- External inputs via the root inport blocks
- Ground if the input signal is unconnected or grounded

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

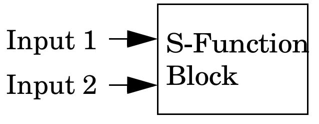
During the simulation loop, accessing the input signals is performed using

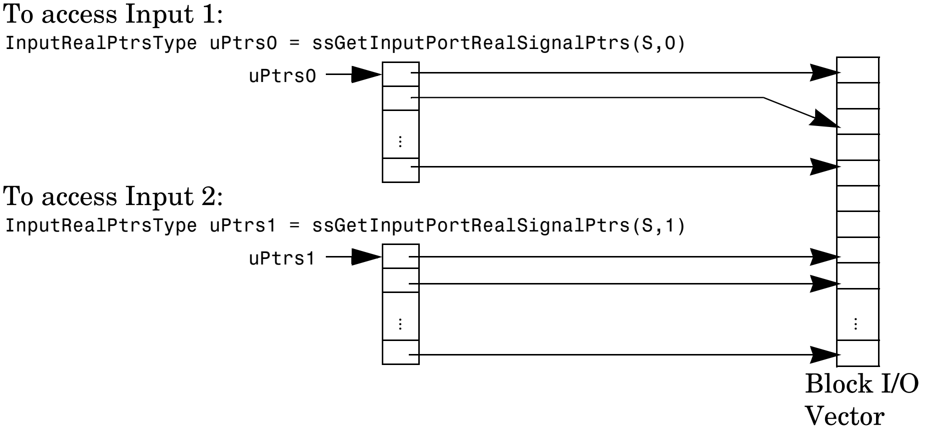
```
InputRealPtrsType uPtrs =
ssGetInputPortRealSignalPtrs(S,portIndex)
```

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 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

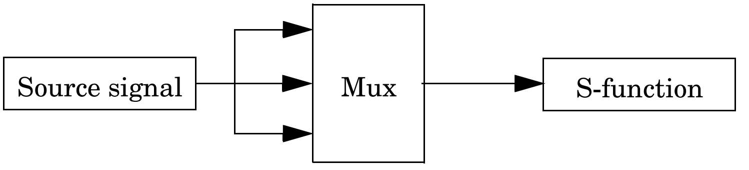
```
real_T *u = *uPtrs; /* Incorrect */
```

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 9, “S-Function Callback Methods.” 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 3-46 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*)

Obsolete Macro	Replace With
`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)`

4

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

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
#endif       // 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_sfun.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.

5

Creating Ada S-Functions

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

Introduction

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

Ada S-Function Source File Format

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

Ada S-Function Specification

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

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

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

package Times_Two is

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

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

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

end Times_Two;
```

Ada S-Function Body

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

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

package body Times_Two is

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

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

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

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

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


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

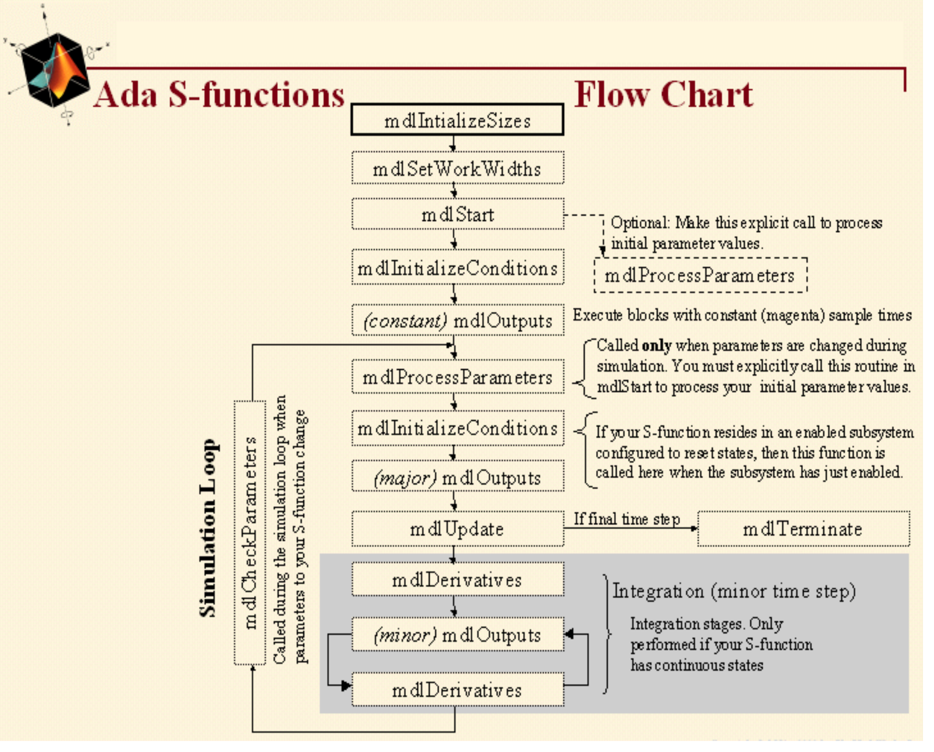
```
   --      Compute the S-Function's output,
   --       given its input: y = 2 * u
   --
   procedure mdlOutputs(S : in SimStruct; TID : in Integer) is

      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 occured 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 9, "S-Function Callback Methods," 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 5-9) provides a stub implementation.

SimStruct Functions

Simulink provides a set of functions that enable an Ada S-function to access the internal data structure (SimStruct) that Simulink maintains for the S-function. These functions consist of Ada wrappers around the SimStruct macros used to access the SimStruct from a C S-function (see Chapter 10, "SimStruct Functions"). Simulink provides Ada wrappers for a substantial subset of the SimStruct macros. The “Languages Supported” section of the reference page for a macro specifies whether it has an Ada wrapper.

Building an Ada S-Function

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

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

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

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

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

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

Ada Compiler Requirements

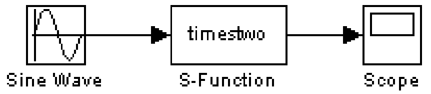
To build a MEX file from Ada source code, using the `mex` tool, you must have previously installed a copy of version 3.12 (or higher) of the GNAT Ada95 compiler on your system. You can obtain the latest Solaris, Windows, and GNU-Linux versions of the compiler at the GNAT ftp site (`ftp://cs.nyu.edu/pub/gnat`).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

The GNAT Ada95 compiler package used to include `gnatdll.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

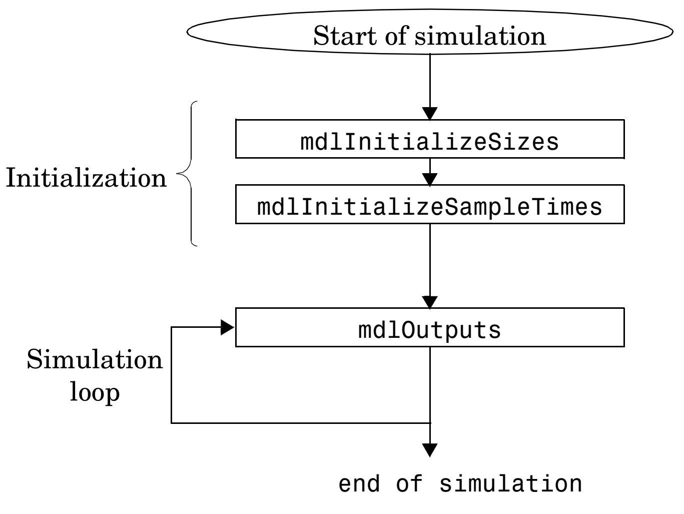
This section presents an example of a basic Ada S-function that you can use as a model when creating your own Ada S-functions. The example is the `timestwo` S-function example that comes with Simulink (see *`matlabroot`*`/simulink/ada/examples/timestwo.ads` and *`matlabroot`*`/simulink/ada/examples/timestwo.adb`). 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 10, “SimStruct Functions”).

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

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

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

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

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

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

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

Note When building an Ada S-function, MATLAB's `mex` tool uses the package specification to determine 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 attrubouts 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 occured 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.

6

Creating Fortran S-Functions

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

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 the 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.

Creating Level 1 Fortran S-Functions

The 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
      SUBROUTINE SIZES(SIZE)
C     .. Array arguments ..
      INTEGER*4        SIZE(*)
C     .. Parameters ..
      INTEGER*4       NSIZES
      PARAMETER       (NSIZES=6)

      SIZE(1) = 0
      SIZE(2) = 0
      SIZE(3) = 1
      SIZE(4) = 1
      SIZE(5) = 0
      SIZE(6) = 1

      RETURN
      END

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

      Y(1) = U(1) * 2.0

      RETURN
      END

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

```
C
C=====================================================

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

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

      SUBROUTINE DSTATES(T, X, U, XNEW)
      REAL*8           T, X(*), U(*), XNEW(*)
C --- Nothing to do.
      RETURN
      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` and read the *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 6-14).

Template File

The file `matlabroot/simulink/src/sfungate.c` is a C-MEX template file for calling into a Fortran subroutine. 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 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's 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 EÈÙ
0000136 EÈÙ
0000271 EÈº
0000467 ÇEÈ@
0000530 ÇEÈ
0000575 EÈÙEäØ5@
0001267 Cf¦VC-ò:C
0001323 :|.-:8Æ#8ýKw6
0001353 ?333@
0001364 333À
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/sfuntmpl_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 Simulink's state vector for integration instead of being integrated by the Fortran code. Likewise, all derivative calculations must be made available separately to be called from the `mdlDerivatives()` method in the S-function. Without these steps, any Fortran code with continuous states will not be compatible with variable-step solvers if the S-function is registered as a continuous block with continuous states.

Telltale signs of implicit advancement are incremented variables such as `M=M+1` or `X=X+0.05`. If the code has many of these constructs and you determine that it is impractical to recode the source 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.

7

Implementing Block Features

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

Dialog Parameters

A user can pass parameters to an S-function at the start of and, optionally, during the simulation, using the **S-Function parameters** field of the block's dialog box. Such parameters are called *dialog box parameters* to distinguish them from run-time parameters created by the S-function to facilitate code generation (see "Run-Time Parameters" on page 7-5). 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. 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 7-5).

Tuning Parameters in External Mode

When a user tunes parameters during simulation, Simulink invokes the S-function's `mdlCheckParameters` method to validate the changes and then the S-functions' `mdlProcessParameters` method to give the S-function a chance to process the parameters in some way. 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 and use internal representations of external dialog parameters called *run-time parameters*. Every run-time parameter corresponds to one or more dialog parameters and can have the same value and data type as its corresponding external 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.

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.

Note The first four characters of block's run-time parameter names 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.

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`.

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 7-9)

  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 7-9).
- Whether the input port allows scalar expansion of inputs (see "Scalar Expansion of Inputs" on page 7-11)
- Whether the input port has direct feedthrough, using `ssSetInputPortDirectFeedThrough`

  A port has direct feedthrough if the input is used in either the `mdlOutputs` or `mdlGetTimeOfNextVarHit` functions. The direct feedthrough flag for each input port can be set to either `1=yes` or `0=no`. It should be set to 1 if the input, `u`, is used in the `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 input ports that the S-function has, using `ssSetNumOutputPorts`. Then, for each output port, the method should specify

- Dimensions of the output port

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

- `ssSetOutputWidth`

  If you want the port's dimensions to depend on block connectivity, set the dimensions to `DYNAMICALLY_SIZED`. The S-function must then provide `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

Simulink supports blocks that execute at different rates. 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]
```

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 7-19)
- Constant sample time, i.e., the port's input or output never changes (see "Specifying Constant Sample Time for a Port" on page 7-20)

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 7-21).

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.

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 parameter**s option on the **Advanced** pane of the **Simulation 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 eheck 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 `sfcndemo_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` once per time step. 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`, use `ssGetPortBasedSampleTimeBlockIsTriggered` 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 (ssISampleHit(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.

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.

Table 7-1: 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 Table 7-1, Macros Used in Specifying Vector Widths, 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 Simulink's solvers. When you specify continuous states, you must return the states' derivatives in `mdlDerivatives`. The discrete state vector is used to maintain state information that changes at fixed intervals. Typically the discrete state vector is updated in place in `mdlUpdate`.

The integer, real, and pointer work vectors are storage locations that 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 `mxCalloc` 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

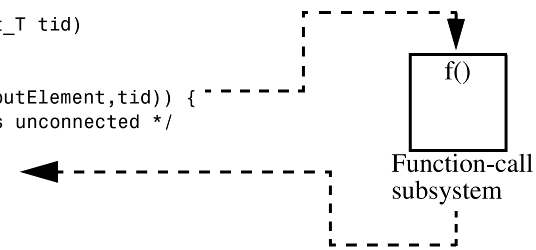
You can create a triggered subsystem whose execution is determined by logic internal to an S-function instead of by the value of a signal. A subsystem so configured is called a *function-call subsystem*. To implement a function-call subsystem:

- In the Trigger block, select **function-call** as the **Trigger type** parameter.
- In the S-function, use the `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.

```
void mdlOutputs(SimStruct *S, int_T tid)
{
  ...
  if (!ssCallSystemWithTid(S,outputElement,tid)) {
    return; /* error or output is unconnected */
  }
  <next statement>
  ...
}
```



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:

1 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 */
```

2 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.

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;
}
```

The `ssSetErrorStatus` error-handling approach is the suggested alternative to using the `mexErrMsgTxt` function. The function `mexErrMsgTxt` uses exception handling to immediately terminate S-function execution and return control to Simulink. In order to support exception handling inside 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 mx* 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 3-35). 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 3-35. 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 Simulink's 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 **Bounds checking** options list on the **Simulation Parameters** dialog box or enter the following command at the MATLAB command line.

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

S-Function Examples

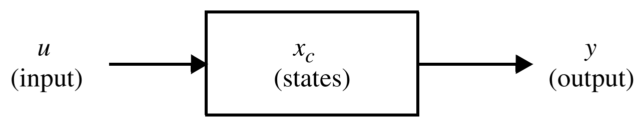
Most S-Function blocks require the handling of states, continuous or discrete. The following sections discuss common types of systems that you can model in Simulink with S-functions:

- 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 Simulink's solvers integrate using the following equations.



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

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

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 3-35. 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

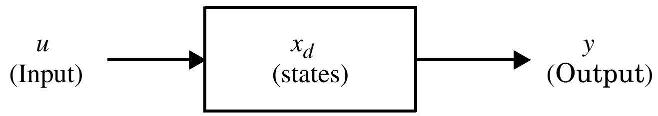
```
/*  File    : csfunc.c
 *  Abstract:
 *
 *      Example C-file S-function for defining a continuous system.
 *
 *      x' = Ax + Bu
 *      y  = Cx + Du
 *
 *      For more details about S-functions, see simulink/src/sfuntmpl_doc.c.
 *
 *  Copyright 1990-2000 The MathWorks, Inc.
 */

#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:
 *    Specifiy 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.



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

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

`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 3-35. 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;
    }
}
```

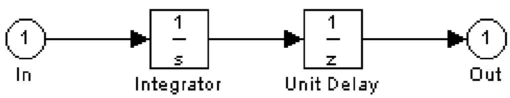
```
/* Function: mdlOutputs =======================================================
 * Abstract:
 *      y = Cx + Du
 */
static void mdlOutputs(SimStruct *S, int_T tid)
{
    real_T            *y    = ssGetOutputPortRealSignal(S,0);
    real_T            *x    = ssGetRealDiscStates(S);
    InputRealPtrsType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

#if defined(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 continusous 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 a equation switch definitely occurred
 *  during the time step.
 *
 *  Ideally, the ZC signal gives an estimate of when an equation switch
 *  occurred.  For example, if the ZC signal is -2 at the beginning and +6 at
 *  the end, then this suggests that the switch occurred
 *  25% = 100%*(-2)/(-2-(+6)) of the way into the time step.  It will almost
 *  never be true that 25% is perfectly correct.  There is no perfect choice
 *  for a ZC signal, but there are some good rules.  First, choose the ZC
 *  signal to be continuous.  Second, choose the ZC signal to give a monotonic
 *  measure of the "distance" to a signal switch; strictly monotonic is ideal.
 */
static void mdlZeroCrossings(SimStruct *S)
{
    int_T             iOutput;
    int_T             numOutput = ssGetOutputPortWidth(S,0);
    real_T            *zcSignals = ssGetNonsampledZCs(S);
    InputRealPtrsType uPtrs      = ssGetInputPortRealSignalPtrs(S,0);

    /*
     * Set index and increment for the input signal, upper limit, and lower
     * limit parameters so that each gives scalar expansion if needed.
     */
    int_T  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
 *      simulation 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. */
#if defined(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 sfuntmpl_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 continous 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.
 */
static void mdlOutputs(SimStruct *S, int_T tid)
{
    if (ssIsContinuousTask(S,tid)) {
```

```
        int                i;
        real_T             *num;
        int                nContStates = ssGetNumContStates(S);
        real_T             *x          = ssGetContStates(S);
        int_T              nCoeffs     = nContStates + 1;
        InputRealPtrsType uPtrs       = ssGetInputPortRealSignalPtrs(S,0);
        real_T             *y          = ssGetOutputPortRealSignal(S,0);
        int_T              *activeBank = ssGetIWork(S);

        /*
         * Switch banks because we've updated them in mdlUpdate and we're no
         * longer in a minor time step.
         */
        if (ssIsMajorTimeStep(S)) {
            int_T *banksUpdated = ssGetIWork(S) + 1;
            if (*banksUpdated) {
                *activeBank = !(*activeBank);
                *banksUpdated = 0;
                /*
                 * Need to tell the solvers that the derivatives are no
                 * longer valid.
                 */
                ssSetSolverNeedsReset(S);
            }
        }
        num = ssGetRWork(S) + (*activeBank) * (2*nCoeffs);

        /*
         * The continuous system is evaluated using a controllable state space
         * representation of the transfer function.  This implies that the
         * output of the system is equal to:
         *
         *     y(t) = Cx(t) + Du(t)
         *          = [ b1 b2 ... bn]x(t) + b0u(t)
         *
         * where b0, b1, b2, ... are the coefficients of the numerator
         * polynomial:
         *
         *    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
```

8

Writing S-Functions for Real-Time Workshop

The following sections explain how to write S-functions that work with the Real-Time Workshop.

Introduction

This chapter describes how to create S-functions that work seamlessly with the Real-Time Workshop. It begins with basic concepts and concludes with an example of how to create a highly optimized direct-index lookup table S-Function block.

This chapter assumes that you understand these concepts:

- Level 2 S-functions
- Target Language Compiler (TLC)
- The basics of how the Real-Time Workshop creates generated code

See the *Target Language Compiler Reference Guide* and the *Real-Time Workshop User's Guide* for more information about these subjects.

A note on terminology: when this chapter refers to actions performed by the Target Language Compiler, including parsing, caching, creating buffers, etc., the name Target Language Compiler is spelled out fully. When referring to code written in the Target Language Compiler syntax, this chapter uses the abbreviation TLC.

Note The guidelines presented in this chapter are for Real-Time Workshop users. Even if you do not currently use the Real-Time Workshop, we recommend that you follow the guidelines presented in this chapter when writing S-functions, especially if you are creating general-purpose S-functions.

Classes of Problems Solved by S-Functions

S-functions help solve various kinds of problems you might face when working with Simulink and the Real-Time Workshop (Real-Time Workshop). These problems include

- Extending the set of algorithms (blocks) provided by Simulink and Real-Time Workshop
- Interfacing legacy (hand-written) C-code with Simulink and Real-Time Workshop
- Generating highly optimized C-code for embedded systems

S-functions and S-function routines form an application program interface (API) that allows you to implement generic algorithms in the Simulink environment with a great deal of flexibility. This flexibility cannot always be maintained when you use S-functions with the Real-Time Workshop. For example, it is not possible to access the MATLAB workspace from an S-function that is used with the Real-Time Workshop. However, using the techniques presented in this chapter, you can create S-functions for most applications that work with the generated code from the Real-Time Workshop.

Although S-functions provide a generic and flexible solution for implementing complex algorithms in Simulink, they require significant memory and computation resources. Most often the additional resources are acceptable for real-time rapid prototyping systems. In many cases, though, additional resources are unavailable in real-time embedded applications. You can minimize memory and computational requirements by using the Target Language Compiler technology provided with the Real-Time Workshop to inline your S-functions.

Types of S-Functions

The implementation of S-functions changes based on your requirements. This chapter discusses the typical problems that you may face and how to create S-functions for applications that need to work with Simulink and the Real-Time Workshop. These are some (informally defined) common situations:

1. "I'm not concerned with efficiency. I just want to write one version of my algorithm and have it work in Simulink and the Real-Time Workshop automatically."

2. "I have a lot of hand-written code that I need to interface. I want to call my function from Simulink and the Real-Time Workshop in an efficient manner."

   or said another way:

   "I want to create a block for my blockset that will be distributed throughout my organization. I'd like it to be very maintainable with efficient code. I'd like my algorithm to exist in one place but work with both Simulink and the Real-Time Workshop."

3 “I want to implement a highly optimized algorithm in Simulink and the Real-Time Workshop that looks like a built-in block and generates very efficient code.”

The MathWorks has adopted terminology for these different requirements. Respectively, the situations described above map to this terminology:

1 Noninlined S-function

2 Wrapper S-function

3 Fully inlined S-function

Noninlined S-Functions

A noninlined S-function is a C-MEX S-function that is treated identically by Simulink and the Real-Time Workshop. In general, you implement your algorithm once according to the S-function API. Simulink and the Real-Time Workshop call the S-function routines (e.g., `mdlOutputs`) at the appropriate points during model execution.

Significant memory and computation resources are required for each instance of a noninlined S-Function block. However, this routine of incorporating algorithms into Simulink and the Real-Time Workshop is typical during the prototyping phase of a project where efficiency is not important. The advantage gained by forgoing efficiency is the ability to change model parameters and/or structures rapidly.

Note that writing a noninlined S-function does not involve any TLC coding. Noninlined S-functions are the default case for the Real-Time Workshop in the sense that once you’ve built a C-MEX S-function in your model, there is no additional preparation prior to clicking **Build** in the **Real-Time Workshop** Page of the **Simulation Parameters** dialog box for your model.

Wrapper S-Functions

A wrapper S-function is ideal for interfacing hand-written code or a large algorithm that is encapsulated within a few procedures. In this situation, usually the procedures reside in modules that are separate from the C-MEX S-function. The S-function module typically contains a few calls to your procedures. Because the S-function module does not contain any parts of your algorithm, but only calls your code, it is referred to as a *wrapper S-function*.

In addition to the C-MEX S-function wrapper, you need to create a TLC wrapper that complements your S-function. The TLC wrapper is similar to the S-function wrapper in that it contains calls to your algorithm.

Fully Inlined S-Functions

A fully inlined S-function builds your algorithm (block) into Simulink and the Real-Time Workshop in a manner that is indistinguishable from a built-in block. Typically, a fully inlined S-function requires you to implement your algorithm twice: once for Simulink (C-MEX S-function) and once for the Real-Time Workshop (TLC file). The complexity of the TLC file depends on the complexity of your algorithm and the level of efficiency you're trying to achieve in the generated code. TLC files vary from simple to complex in structure.

Basic Files Required for Implementation

This section briefly describes what files and functions you'll need to create noninlined, wrapper, and fully inlined S-functions.

- Noninlined S-functions require the C-MEX S-function source code *`sfunction`*`.c`.
- Wrapper S-functions that inline a call to your algorithm (your C function) require an *`sfunction`*`.tlc` file.
- Fully inlined S-functions require an *`sfunction`*`.tlc` file. Fully inlined S-functions produce the optimal code for a parameterized S-function. This is an S-function that operates in a specific mode dependent upon fixed S-function parameters that do not change during model execution. For a given operating mode, the *`sfunction`*`.tlc` file specifies the exact code that is generated to implement the algorithm for that mode. For example, the direct-index lookup table S-function at the end of this chapter contains two operating modes — one for evenly spaced `x-data` and one for unevenly spaced `x-data`.
  - Fully inlined S-functions might require the placement of the `mdlRTW` routine in your S-function MEX-file *`sfunction`*`.c`. The `mdlRTW` routine lets you place information in *`model`*`.rtw`, which is the file that is processed by the Target Language Compiler prior to executing *`sfunction`*`.tlc` when generating code. This is useful when you want to introduce nontunable parameters into your TLC file.

For S-functions to work correctly in the Simulink environment, a certain amount of overhead code is necessary. When the Real-Time Workshop generates code from models that contain S-functions (without `sfunction.tlc` files), it embeds some of this overhead code in the generated C code. If you want to optimize your real-time code and eliminate some of the overhead code, you must *inline* (or embed) your S-functions. This involves writing a TLC (`sfunction.tlc`) file that directs the Real-Time Workshop to eliminate all overhead code from the generated code. The Target Language Compiler, which is part of the Real-Time Workshop, processes `sfunction.tlc` files to define how to inline your S-function algorithm in the generated code.

Note The term *inline* should not be confused with the C++ *inline* keyword. In MathWorks terminology, inline means to specify a textual string in place of the call to the general S-function API routines (e.g., `mdlOutputs`). For example, when we say that a TLC file is used to inline an S-function, we mean that the generated code contains the appropriate C code that would normally appear within the S-function routines and the S-function itself has been removed from the build process.

Noninlined S-Functions

Noninlined S-functions are identified by the *absence* of an *sfunction*.tlc file for your S-function (*sfunction*.mex). When placing a noninlined S-function in a model that is to be used with the Real-Time Workshop, the following MATLAB API functions are supported:

- mxGetEps
- mxGetInf
- mxGetM
- mxGetN
- mxGetNaN
- mxGetPr — Note that using mxGetPr on an empty matrix does not return NULL; rather, it returns a random value. Therefore, you should protect calls to mxGetPr with *mxIsEmpty*.
- mxGetScalar
- mxGetString
- mxIsEmpty
- mxIsFinite
- mxIsInf

In addition, parameters to S-functions can only be of type double precision or characters contained in scalars, vectors, or 2-D matrices. To obtain more flexibility in the type of parameters you can supply to S-functions or the operations in the S-function, you need to inline your S-function and (possibly) use an mdlRTW S-function routine.

S-Function Module Names for Real-Time Workshop Builds

If your S-function is built with multiple modules, you must provide the build process names of additional modules. You can do this through the Real-Time Workshop template makefile technology, or more conveniently by using the set_param MATLAB command. For example, if your S-function is built with multiple modules, as in

```
mex sfun_main.c sfun_module1.c sfun_module2.c
```

specify the names of the modules without the extension, using the command

```
set_param(sfun_block,'SFunctionModules','sfun_module1 sfun_module2')
```

The parameter can also be a variable, as in

```
modules = 'sfun_module1 sfun_module2'
set_param(sfun_block,'SFunctionModules','modules')
```

or a string to be evaluated (this is needed when the modules are valid identifiers).

```
set_param(sfun_block,'SFunctionModules','''sfun_module1 sfun_module2''')
```

Writing Wrapper S-Functions

This section describes how to create S-functions that work seamlessly with Simulink and the Real-Time Workshop using the *wrapper* concept. This section begins by describing how to interface your algorithms in Simulink by writing MEX S-function wrappers (`sfunction.mex`). It finishes with a description of how to direct the Real-Time Workshop to insert your algorithm into the generated code by creating a TLC S-function wrapper (`sfunction.tlc`).

MEX S-Function Wrapper

Creating S-functions using an S-function wrapper allows you to insert your C code algorithms in Simulink and the Real-Time Workshop with little or no change to your original C code function. A *MEX S-function wrapper* is an S-function that calls code that resides in another module. In effect, the wrapper binds your code to Simulink. A *TLC S-function wrapper* is a TLC file that specifies how the Real-Time Workshop should call your code (the same code that was called from the C-MEX S-function wrapper).

Suppose you have an algorithm (i.e., a C function) called `my_alg` that resides in the file `my_alg.c`. You can integrate `my_alg` into Simulink by creating a MEX S-function wrapper (e.g., `wrapsfcn.c`). Once this is done, Simulink can call `my_alg` from an S-Function block. However, the Simulink S-function contains a set of empty functions that Simulink requires for various API-related purposes. For example, although only `mdlOutputs` calls `my_alg`, Simulink calls `mdlTerminate` as well, even though this S-function routine performs no action.

You can integrate `my_alg` into the Real-Time Workshop generated code (i.e., embed the call to `my_alg` in the generated code) by creating a TLC S-function wrapper (e.g., `wrapsfcn.tlc`). The advantage of creating a TLC S-function wrapper is that the empty function calls can be eliminated and the overhead of executing the `mdlOutputs` function and then the `my_alg` function can be eliminated.

Wrapper S-functions are useful when you are creating new algorithms that are procedural in nature or when you are integrating legacy code into Simulink. However, if you want to create code that is

- Interpretive in nature in Simulink (i.e., highly parameterized by operating modes)

- Heavily optimized in the Real-Time Workshop (i.e., no extra tests to decide what mode the code is operating in)

then you must create a *fully inlined TLC file* for your S-function.

This figure illustrates the wrapper S-function concept.

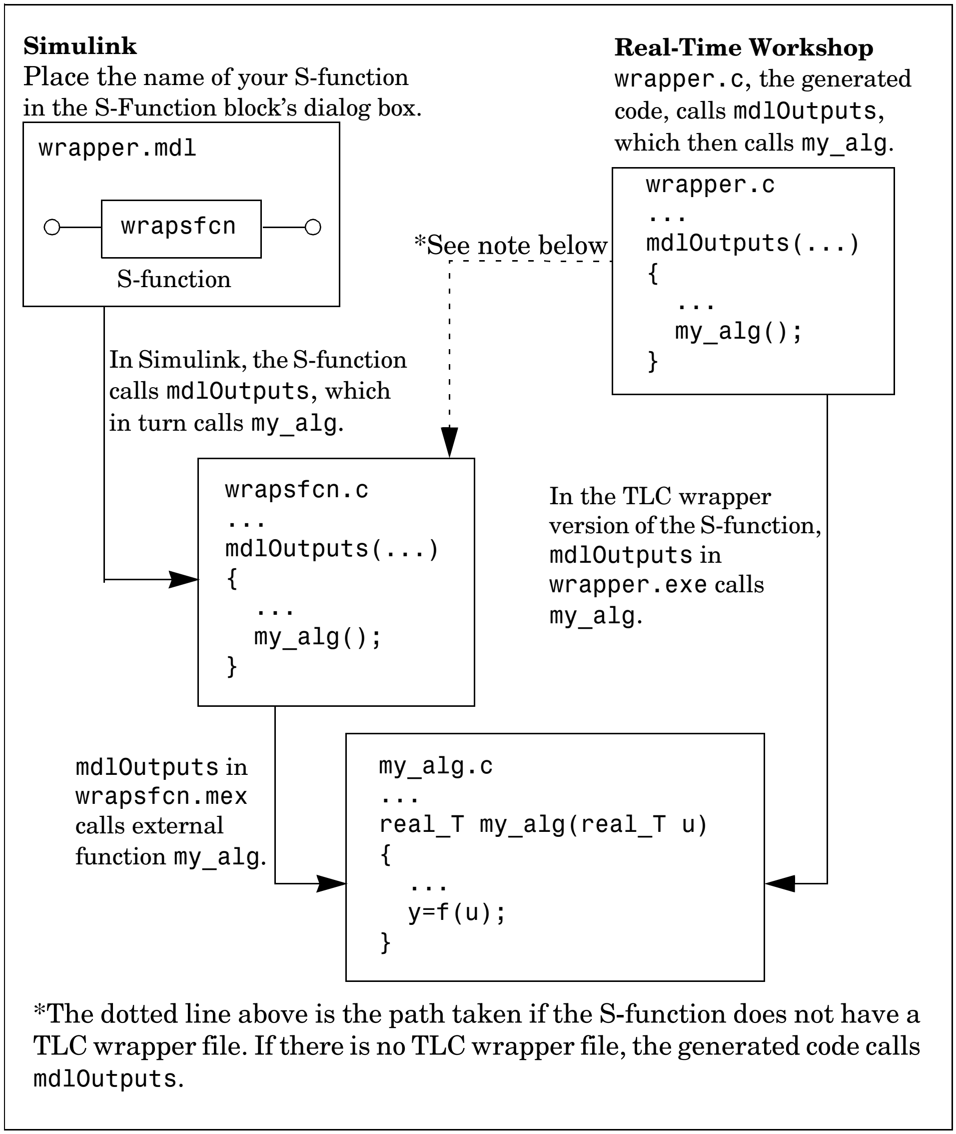


Figure 8-1: How S-Functions Interface with Hand-Written Code

Using an S-function wrapper to import algorithms in your Simulink model means that the S-function serves as an interface that calls your C code algorithms from `mdlOutputs`. S-function wrappers have the advantage that you can quickly integrate large stand-alone C code into your model without having to make changes to the code.

This is an example of a model that includes an S-function wrapper.

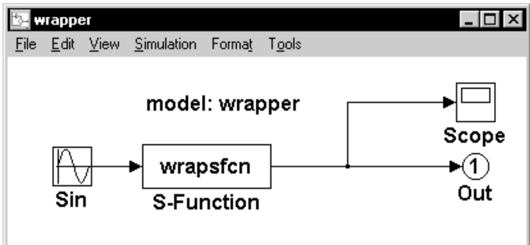


Figure 8-1: An Example Model That Includes an S-Function Wrapper

There are two files associated with the `wrapsfcn` block, the S-function wrapper and the C code that contains the algorithm. This is the S-function wrapper code for this example, called `wrapsfcn.c`.

Declare `my_alg` as `extern`.

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

extern real_T my_alg(real_T u);

/*
 * mdlInitializeSizes - initialize the sizes array
 */
static void mdlInitializeSizes(SimStruct *S)
{

    ssSetNumSFcnParams( S, 0); /*number of input arguments*/

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

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

    ssSetNumSampleTimes( S, 1);
}
```

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


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

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

Place the call to `my_alg` in `mdlOutputs`.

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

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

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

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

TLC S-Function Wrapper

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

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

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

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

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

```
set_param('wrapper/S-Function','SFunctionModules','my_alg')
```

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

```
<Generated code comments for wrapper model with noninlined wrapsfcn S-function>

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

/* Start the model */
void mdlStart(void)
{
  /* (no start code required) */
}

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

Noninlined S-functions create a `SimStruct` object and generate a call to the S-function routine

```
  /* Level2 S-Function Block: <Root>/S-Function (wrapsfcn) */
  {
    SimStruct *rts = ssGetSFunction(rtS, 0);
    sfcnOutputs(rts, tid);
  }

  /* Outport Block: <Root>/Out */
  rtY.Out = rtB.S_Function;
}

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

/* Terminate function */
void mdlTerminate(void)
{
  /* Level2 S-Function Block: <Root>/S-Function (wrapsfcn) */
  {
    SimStruct *rts = ssGetSFunction(rtS, 0);
    sfcnTerminate(rts);
  }
}

#include "wrapper.reg"

/* [EOF] wrapper.c */
```

Noninlined S-functions require a `SimStruct` object and the call to the S-function routine `mdlTerminate`.

In addition to the overhead outlined above, the `wrapper.reg` generated file contains the initialization of the `SimStruct` for the wrapper S-Function block. There is one child `SimStruct` for each S-Function block in your model. You can significantly reduce this overhead by creating a TLC wrapper for the S-function.

How to Inline

The generated code makes the call to your S-function, `wrapsfcn.c`, in `mdlOutputs` by using this code:

```
SimStruct *rts = ssGetSFunction(rtS, 0);
sfcnOutputs(rts, tid);
```

This call has a significant amount of computational overhead associated with it. First, Simulink creates a `SimStruct` data structure for the S-Function block. Second, the Real-Time Workshop constructs a call through a function pointer to execute `mdlOutputs`, then `mdlOutputs` calls `my_alg`. By inlining the call to your C algorithm, `my_alg`, you can eliminate both the `SimStruct` and the extra function call, thereby improving the efficiency and reducing the size of the generated code.

Inlining a wrapper S-function requires an *`sfunction`*`.tlc` file for the S-function; this file must contain the function call to `my_alg`. This picture shows the relationships between the algorithm, the wrapper S-function, and the *`sfunction`*`.tlc` file.

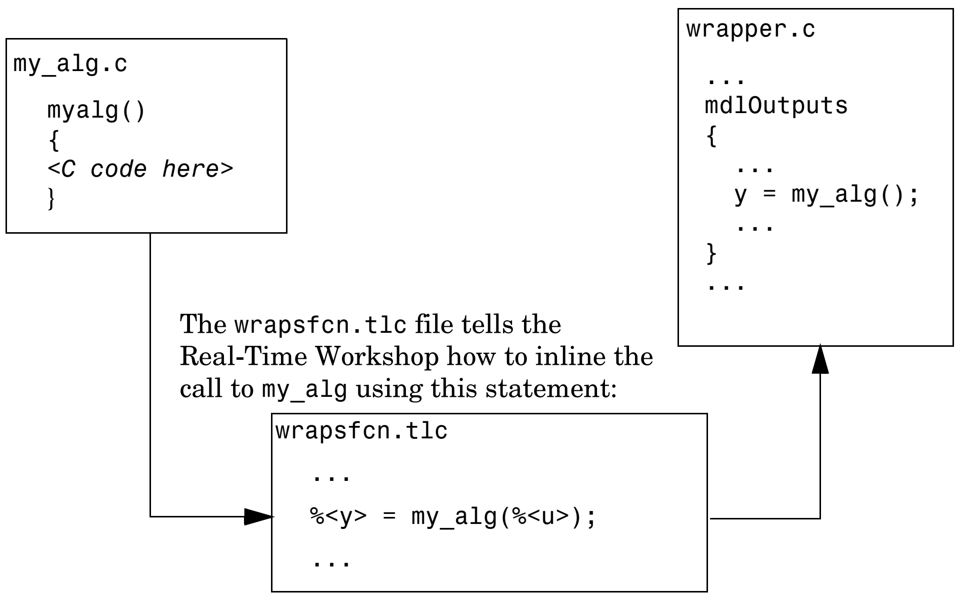


Figure 8-2: Inlining an Algorithm by Using a TLC File

To inline this call, you have to place your function call in an *`sfunction`*`.tlc` file with the same name as the S-function (in this example, `wrapsfcn.tlc`). This causes the Target Language Compiler to override the default method of placing calls to your S-function in the generated code.

This is the `wrapsfcn.tlc` file that inlines `wrapsfcn.c`.

```
%% File    : wrapsfcn.tlc
%% Abstract:
%%      Example inlined tlc file for S-function wrapsfcn.c
%%

%implements "wrapsfcn" "C"

%% Function: BlockTypeSetup ====================================================
%% Abstract:
%%      Create function prototype in model.h as:
%%      "extern real_T my_alg(real_T u);"
%%
%function BlockTypeSetup(block, system) void
  %openfile buffer
    extern real_T my_alg(real_T u);
  %closefile buffer
  %<LibCacheFunctionPrototype(buffer)>
%endfunction %% BlockTypeSetup

%% Function: Outputs ===========================================================
%% Abstract:
%%      y = my_alg( u );
%%
%function Outputs(block, system) Output
  /* %<Type> Block: %<Name> */
  %assign u = LibBlockInputSignal(0, "", "", 0)
  %assign y = LibBlockOutputSignal(0, "", "", 0)
  %% PROVIDE THE CALLING STATEMENT FOR "algorithm"
  %<y> = my_alg(%<u>);

%endfunction %% Outputs
```

This line is placed in `wrapper.h`.

This line is expanded and placed in `mdlOutputs` within `wrapper.c`.

The first section of this code directs the Real-Time Workshop to inline the `wrapsfcn` S-Function block and generate the code in C:

```
%implements "wrapsfcn" "C"
```

The next task is to tell the Real-Time Workshop that the routine `my_alg` needs to be declared external in the generated `wrapper.h` file for any `wrapsfcn` S-Function blocks in the model. You only need to do this once for all `wrapsfcn` S-Function blocks, so use the `BlockTypeSetup` function. In this function, you tell the Target Language Compiler to create a buffer and cache the `my_alg` as `extern` in the `wrapper.h` generated header file.

The final step is the inlining of the call to the function `my_alg`. This is done by the `Outputs` function. In this function, you load the input and output and place a direct call to `my_alg`. The call is embedded in `wrapper.c`.

The Inlined Code

The code generated when you inline your wrapper S-function is similar to the default generated code. The `mdlTerminate` function no longer contains a call to an empty function and the `mdlOutputs` function now directly calls `my_alg`.

Inlined call to the function `my_alg`.

```
void mdlOutputs(int_T tid)
{
  /* Sin Block: <Root>/Sin */
  rtB.Sin = rtP.Sin.Amplitude *
    sin(rtP.Sin.Frequency * ssGetT(rtS) + rtP.Sin.Phase);

  /* S-Function Block: <Root>/S-Function */
  rtB.S_Function = my_alg(rtB.Sin);

  /* Outport Block: <Root>/Out */
  rtY.Out = rtB.S_Function;
}
```

In addition, `wrapper.reg` no longer creates a child `SimStruct` for the S-function because the generated code is calling `my_alg` directly. This eliminates over 1K of memory usage.

Fully Inlined S-Functions

Continuing the example of the previous section, you could eliminate the call to `my_alg` entirely by specifying the explicit code (i.e., `2.0 * u`) in `wrapsfcn.tlc`. This is referred to as a *fully inlined S-function*. While this can improve performance, if your C code is large this can be a lengthy task. In addition, you now have to maintain your algorithm in two places, the C S-function itself and the corresponding TLC file. However, the performance gains might outweigh the disadvantages. To inline the algorithm used in this example, in the `Outputs` section of your `wrapsfcn.tlc` file, instead of writing

```
%<y> = my_alg(%<u>);
```

use

```
%<y> = 2.0 * %<u>;
```

This is the code produced in `mdlOutputs`:

```
void mdlOutputs(int_T tid)
{
  /* Sin Block: <Root>/Sin */
  rtB.Sin = rtP.Sin.Amplitude *
    sin(rtP.Sin.Frequency * ssGetT(rtS) + rtP.Sin.Phase);

  /* S-Function Block: <Root>/S-Function */
  rtB.S_Function = 2.0 * rtB.Sin;

  /* Outport Block: <Root>/Out */
  rtY.Out = rtB.S_Function;
}
```

This is the explicit embedding of the algorithm.

The Target Language Compiler has replaced the call to `my_alg` with the algorithm itself.

Multiport S-Function Example

A more advanced multiport inlined S-function example exists in *`matlabroot`*`/simulink/src/sfun_multiport.c` and *`matlabroot`*`/toolbox/simulink/blocks/tlc_c/sfun_multiport.tlc`. This S-function demonstrates how to create a fully inlined TLC file for an S-function that contains multiple ports. You might find that looking at this example aids in the understanding of fully inlined TLC files.

Fully Inlined S-Function with the mdlRTW Routine

You can make a more fully inlined S-function that uses the S-function `mdlRTW` routine. The purpose of the `mdlRTW` routine is to provide the code generation process with more information about how the S-function is to be inlined, including

- Renaming of tunable parameters in the generated code. This improves readability of the code by replacing `p1`, `p2`, etc., by names of your choice.
- Creating a parameter record of a nontunable parameter for use with a TLC file.

`mdlRTW` does this by placing information into the *`model`*`.rtw` file. The `mdlRTW` routine is described in the text file *`matlabroot`*`/simulink/src/sfuntmpl_doc.c`.

As an example of how to use the `mdlRTW` function, this section discusses the steps you must take to create a direct-index lookup S-function. Lookup tables are collections of ordered data points of a function. Typically, these tables use some interpolation scheme to approximate values of the associated function between known data points. To incorporate the example lookup table algorithm in Simulink, the first step is to write an S-function that executes the algorithm in `mdlOutputs`. To produce the most efficient C code, the next step is to create a corresponding TLC file to eliminate computational overhead and improve the performance of the lookup computations.

For your convenience, Simulink provides support for two general purpose lookup 1-D and 2-D algorithms. You can use these algorithms as they are or create a custom lookup table S-function to fit your requirements. This section demonstrates how to create a 1-D lookup S-function, `sfun_directlook.c`, and its corresponding inlined `sfun_directlook.tlc` file. (See the *Real-Time Workshop User's Guide* and the *Target Language Compiler Reference Guide* for more details on the Target Language Compiler.) This 1-D direct-index lookup table example demonstrates the following concepts that you need to know to create your own custom lookup tables:

- Error checking of S-function parameters
- Caching of information for the S-function that doesn't change during model execution

- How to use the `mdlRTW` routine to customize the Real-Time Workshop generated code to produce the optimal code for a given set of block parameters
- How to generate an inlined `TLC` file for an S-function in a combination of the fully inlined form and/or the wrapper form

S-Function RTWdata

There is a property of blocks called `RTWdata`, which can be used by the Target Language Compiler when inlining an S-function. `RTWdata` is a structure of strings that you can attach to a block. It is saved with the model and placed in the *model*`.rtw` file when generating code. For example, this set of MATLAB commands,

```
mydata.field1 = 'information for field1';
mydata.field2 = 'information for field2';
set_param(gcb,'RTWdata',mydata)
get_param(gcb,'RTWdata')
```

produces this result:

```
ans =

    field1: 'information for field1'
    field2: 'information for field2'
```

Inside the *model*`.rtw` file for the associated S-Function block is this information.

```
Block {
  Type                    "S-Function"
  RTWdata {
    field1                  "information for field1"
    field2                  "information for field2"
  }
```

The Direct-Index Lookup Table Algorithm

The 1-D lookup table block provided in the Simulink library uses interpolation or extrapolation when computing outputs. This extra accuracy is not needed in all situations. In this example, you create a lookup table that directly indexes the output vector (*y*-data vector) based on the current input (*x*-data) point.

This direct 1-D lookup example computes an approximate solution *p(x)* to a partially known function *f(x)* at *x=x0*, given data point pairs *(x,y)* in the form of an *x*-data vector and a *y*-data vector. For a given data pair (e.g., the *i*th pair), *y_i = f(x_i)*. It is assumed that the *x*-data values are monotonically increasing. If *x0* is outside the range of the *x*-data vector, the first or last point is returned.

The parameters to the S-function are

```
XData, YData, XEvenlySpaced
```

`XData` and `YData` are double vectors of equal length representing the values of the unknown function. `XDataEvenlySpaced` is a scalar, 0.0 for false and 1.0 for true. If the `XData` vector is evenly spaced, more efficient code is generated.

The following graph illustrates how the parameters `XData=[1:6]`and `YData=[1,2,7,4,5,9]` are handled. For example, if the input (x-value) to the S-Function block is 3, the output (y-value) is 7.

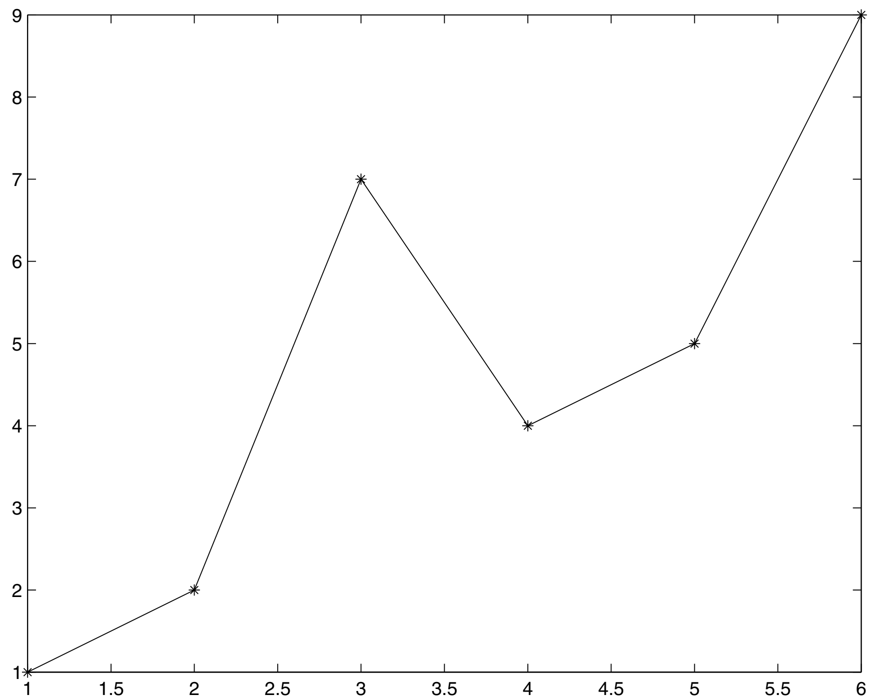


Figure 8-3: Typical Output from a Lookup Table Example

The Direct-Index Lookup Table Example

This section shows how to improve the lookup table by inlining a direct-index S-function with a TLC file. Note that this direct-index lookup table S-function doesn't require a TLC file to work with the Real-Time Workshop. Here the example uses a TLC file for the direct-index lookup table S-function to reduce the code size and increase efficiency of the generated code.

Implementation of the direct-index algorithm with inlined TLC file requires the S-function main module, `sfun_directlook.c`, and a corresponding `lookup_index.c` module. The `lookup_index.c` module contains the `GetDirectLookupIndex` routine that is used to locate the index in the `XData` for the current `x` input value when the `XData` is unevenly spaced. The `GetDirectLookupIndex` routine is called from both the S-function and the generated code. Here the example uses the wrapper concept for sharing C code between Simulink MEX-files and the generated code.

If the `XData` is evenly spaced, then both the S-function main module and the generated code contain the lookup algorithm (not a call to the algorithm) to compute the *y*-value of a given *x*-value, because the algorithm is short. This demonstrates the use of a fully inlined S-function for generating optimal code.

The inlined TLC file, which performs either a wrapper call or embeds the optimal C code, is `sfun_directlook.tlc` (see page -39).

Error Handling

In this example, the `mdlCheckParameters` routine on page -31 verifies that

- The new parameter settings are correct.
- `XData` and `YData` are vectors of the same length containing real finite numbers.
- `XDataEvenlySpaced` is a scalar.
- The `XData` vector is a monotonically increasing vector and evenly spaced if needed.

Note that the `mdlInitializeSizes` routine explicitly calls `mdlCheckParameters` after it verifies that the number of parameters passed to the S-function is correct. After Simulink calls `mdlInitializeSizes`, it then calls `mdlCheckParameters` whenever you change the parameters or there is a need to reevaluate them.

User Data Caching

The `mdlStart` routine on page -34 illustrates how to cache information that does not change during the simulation (or while the generated code is executing). The example caches the value of the `XDataEvenlySpaced` parameter in `UserData`, a field of the SimStruct. The line

```
ssSetSFcnParamTunable(S, XDATAEVENLYSPACED_PIDX, 0);
```

in `mdlInitializeSizes` tells Simulink to disallow changes to the `XDataEvenlySpaced` parameter. During execution, `mdlOutputs` accesses the value of `XDataEvenlySpaced` from `UserData` rather than calling the `mxGetPr` MATLAB API function. This results in a slight increase in performance.

mdlRTW Usage

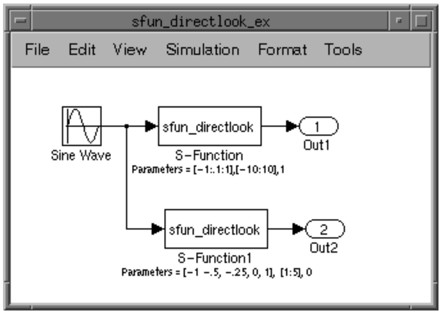
The Real-Time Workshop calls the `mdlRTW` routine while it (the Real-Time Workshop) generates the *`model`*`.rtw` file. You can add information to the *`model`*`.rtw` file about the mode in which your S-Function block is operating to produce optimal code for your Simulink model.

This example adds the following information to the *`model`*`.rtw` file:

- Parameters — These can be modified during execution by external mode. In this example, the `XData` and `YData` S-function parameters can change during execution and are written using the function `ssWriteRTWParameters`.
- Parameter settings — These do not change during execution. In this case the `XDataEvenlySpaced` S-function parameter cannot change during execution (`ssSetSFcnParamTunable` was specified as false (`0`) for it in `mdlInitializeSizes`). This example writes it out as a parameter setting (`XSpacing`) using the function `ssWriteRTWParamSettings`.

Example Model

Before examining the S-function and the inlined TLC file, consider the generated code for the following model.



When creating this model, you need to specify the following for each S-Function block.

```
set_param('sfun_directlook_ex/S-Function','SFunctionModules','lookup_index')
set_param('sfun_directlook_ex/S-Function1','SFunctionModules','lookup_index')
```

This informs the Real-Time Workshop build process that the module `lookup_index.c` is needed when creating the executable.

The generated code for the lookup table example model is

```
<Generated header for sfun_directlook_ex model>

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

/* Start the model */
void mdlStart(void)
{
  /* (no start code required) */
}

/* Compute block outputs */
void mdlOutputs(int_T tid)
{
  /* local block i/o variables */
  real_T rtb_Sine_Wave;
  real_T rtb_buffer2;

  /* Sin Block: <Root>/Sine Wave */
  rtb_Sine_Wave = rtP.Sine_Wave.Amplitude *
    sin(rtP.Sine_Wave.Frequency * ssGetT(rtS) + rtP.Sine_Wave.Phase);

  /* S-Function Block: <Root>/S-Function */
  {
    real_T *xData = &rtP.S_Function.XData[0];
    real_T *yData = &rtP.S_Function.YData[0];
    real_T spacing = xData[1] - xData[0];

    if ( rtb_Sine_Wave <= xData[0] ) {
      rtb_buffer2 = yData[0];
    } else if ( rtb_Sine_Wave >= yData[20] ) {
      rtb_buffer2 = yData[20];
    } else {
      int_T idx = (int_T)( ( rtb_Sine_Wave - xData[0] ) / spacing );
      rtb_buffer2 = yData[idx];
    }
  }

  /* Outport Block: <Root>/Out1 */
  rtY.Out1 = rtb_buffer2;
```

This is the code that is inlined for the top S-Function block in the `sfun_directlook_ex`. model.

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

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

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

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

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

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

#include "sfun_directlook_ex.reg"

/* [EOF] sfun_directlook_ex.c */
```

matlabroot/simulink/src/sfun_directlook.c

```
/*
 * File    :  sfun_directlook.c
 * Abstract:
 *
 *      Direct 1-D lookup. Here we are trying to compute an approximate
 *      solution p(x) to an unknown function f(x) at x=x0, given data point
 *      pairs (x,y) in the form of an x data vector and a y data vector. For a
 *      given data pair (say the ith pair), we have y_i = f(x_i). It is
 *      assumed that the x data values are monotonically increasing.  If the
 *      x0 is outside of the range of the x data vector, then the first or
 *      last point will be returned.
 *
 *      This function returns the "nearest" y0 point for a given x0. No
 *      interpolation is performed.
 *
 *      The S-function parameters are:
 *        XData              - double vector
 *        YData              - double vector
 *        XDataEvenlySpacing - double scalar 0 (false) or 1 (true)
 *        The third parameter cannot be changed during simulation.
```

```
 *
 *      To build:
 *        mex sfun_directlook.c lookup_index.c
 *
 * Copyright (c) 1990-1998 by The MathWorks, Inc. All Rights Reserved.
 *
 */

#define S_FUNCTION_NAME  sfun_directlook
#define S_FUNCTION_LEVEL 2


#include <math.h>
#include "simstruc.h"
#include <float.h>

/*=========*
 * Defines *
 *=========*/

#define XVECT_PIDX    0
#define YVECT_PIDX    1
#define XDATAEVENLYSPACED_PIDX 2
#define NUM_PARAMS    3

#define XVECT(S)         ssGetSFcnParam(S,XVECT_PIDX)
#define YVECT(S)         ssGetSFcnParam(S,YVECT_PIDX)
#define XDATAEVENLYSPACED(S) ssGetSFcnParam(S,XDATAEVENLYSPACED_PIDX)


/*==============*
 * misc defines *
 *==============*/
#if !defined(TRUE)
#define TRUE  1
#endif
#if !defined(FALSE)
#define FALSE 0
#endif

/*===========*
 * typedef's *
 *===========*/

typedef struct SFcnCache_tag {
    boolean_T evenlySpaced;
} SFcnCache;

/*===================================================================*
 * Prototype define for the function in separate file lookup_index.c *
 *===================================================================*/
extern int_T GetDirectLookupIndex(const real_T *x, int_T xlen, real_T u);
```

```
/*=========================*
 * Local Utility Functions *
 *=========================*/


/* Function: IsRealVect =====================================================
 * Abstract:
 *      Verify that the mxArray is a real vector.
 */
static boolean_T IsRealVect(const mxArray *m)
{
    if (mxIsNumeric(m) &&
        mxIsDouble(m) &&
        !mxIsLogical(m) &&
        !mxIsComplex(m) &&
        !mxIsSparse(m) &&
        !mxIsEmpty(m) &&
        mxGetNumberOfDimensions(m) == 2 &&
        (mxGetM(m) == 1 || mxGetN(m) == 1))
      {

          real_T *data = mxGetPr(m);
          int_T  numEl = mxGetNumberOfElements(m);
          int_T  i;

          for (i = 0; i < numEl; i++) {
              if (!mxIsFinite(data[i])) {
                  return(FALSE);
              }
          }

          return(TRUE);
      } else {
          return(FALSE);
      }
}
/* end IsRealVect */


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


#define MDL_CHECK_PARAMETERS           /* Change to #undef to remove function */
#if defined(MDL_CHECK_PARAMETERS) && defined(MATLAB_MEX_FILE)
/* Function: mdlCheckParameters =============================================
 * Abstract:
 *    This routine will be called after mdlInitializeSizes, whenever
 *    parameters change or get reevaluated. The purpose of this routine is
 *    to verify that the new parameter settings are correct.
 *
```

```
 *    You should add a call to this routine from mdlInitializeSizes
 *    to check the parameters. After setting your sizes elements, you should:
 *       if (ssGetSFcnParamsCount(S) == ssGetNumSFcnParams(S)) {
 *           mdlCheckParameters(S);
 *       }
 */
static void mdlCheckParameters(SimStruct *S)
{

    if (!IsRealVect(XVECT(S))) {
       ssSetErrorStatus(S,"1st, X-vector parameter must be a real finite vector");
        return;
    }

    if (!IsRealVect(YVECT(S))) {
        ssSetErrorStatus(S,"2nd, Y-vector parameter must be a real finite "
                         "vector");
        return;
    }

    /*
     * Verify that the dimensions of X and Y are the same.
     */
    if (mxGetNumberOfElements(XVECT(S)) != mxGetNumberOfElements(YVECT(S)) ||
        mxGetNumberOfElements(XVECT(S)) == 1) {
        ssSetErrorStatus(S,"X and Y-vectors must be of the same dimension "
                         "and have at least two elements");
        return;
    }

    /*
     * Verify we have a valid XDataEvenlySpaced parameter.
     */
    if (!mxIsNumeric(XDATAEVENLYSPACED(S)) ||
        !(mxIsDouble(XDATAEVENLYSPACED(S)) ||
          mxIsLogical(XDATAEVENLYSPACED(S))) ||
        mxIsComplex(XDATAEVENLYSPACED(S)) ||
        mxGetNumberOfElements(XDATAEVENLYSPACED(S)) != 1) {
        ssSetErrorStatus(S,"3rd, X-evenly-spaced parameter must be scalar "
                         "(0.0=false, 1.0=true)");
        return;
    }

    /*
     * Verify x-data is correctly spaced.
     */
    {
        int_T     i;
        boolean_T spacingEqual;
        real_T    *xData = mxGetPr(XVECT(S));
        int_T     numEl  = mxGetNumberOfElements(XVECT(S));

        /*
```

```
         * spacingEqual is TRUE if user  XDataEvenlySpaced
         */
        spacingEqual = (mxGetScalar(XDATAEVENLYSPACED(S)) != 0.0);

        if (spacingEqual) {    /* XData is 'evenly-spaced' */
            boolean_T badSpacing = FALSE;
            real_T    spacing    = xData[1] - xData[0];
            real_T    space;

            if (spacing <= 0.0) {
                badSpacing = TRUE;
            } else {
                real_T eps = DBL_EPSILON;

                for (i = 2; i < numEl; i++) {
                    space = xData[i] - xData[i-1];
                    if (space <= 0.0 ||
                        fabs(space-spacing) >= 128.0*eps*spacing ){
                        badSpacing = TRUE;
                        break;
                   }
                }
            }

            if (badSpacing) {
                ssSetErrorStatus(S,"X-vector must be an evenly spaced "
                                 "strictly monotonically increasing vector");
                return;
            }
        } else {     /* XData is 'unevenly-spaced' */
            for (i = 1; i < numEl; i++) {
                if (xData[i] <= xData[i-1]) {
                    ssSetErrorStatus(S,"X-vector must be a strictly "
                                     "monotonically increasing vector");
                    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)
{
    ssSetNumSFcnParams(S, NUM_PARAMS);  /* Number of expected parameters */
```

```
    /*
     * Check parameters passed in, providing the correct number was specified
     * in the S-function dialog box. If an incorrect number of parameters
     * was specified, Simulink will detect the error since ssGetNumSFcnParams
     * and ssGetSFcnParamsCount will differ.
     *   ssGetNumSFcnParams   - This sets the number of parameters your
     *                          S-function expects.
     *   ssGetSFcnParamsCount - This is the number of parameters entered by
     *                          the user in the Simulink S-function dialog box.
     */
#if defined(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

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

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

    ssSetInputPortTestPoint(S, 0, FALSE);
    ssSetInputPortOverWritable(S, 0, TRUE);

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

    ssSetOutputPortTestPoint(S, 0, FALSE);

    ssSetNumSampleTimes(S, 1);

    ssSetSFcnParamTunable(S, XDATAEVENLYSPACED_PIDX, 0);

    ssSetOptions(S, SS_OPTION_EXCEPTION_FREE_CODE);

} /* mdlInitializeSizes */


/* Function: mdlInitializeSampleTimes ==========================================
 * Abstract:
 *    The lookup inherits its sample time from the driving block.
 */
static void mdlInitializeSampleTimes(SimStruct *S)
{
    ssSetSampleTime(S, 0, INHERITED_SAMPLE_TIME);
    ssSetOffsetTime(S, 0, 0.0);
```

```
} /* end mdlInitializeSampleTimes */



#define MDL_START                    /* Change to #undef to remove function */
#if defined(MDL_START)
/* Function: mdlStart ===============================================================
 * Abstract:
 *      Here we cache the state (true/false) of the XDATAEVENLYSPACED parameter.
 *      We do this primarily to illustrate how to "cache" parameter values (or
 *      information that is computed from parameter values) that do not change
 *      for the duration of the simulation (or in the generated code). In this
 *      case, rather than repeated calls to mxGetPr, we save the state once.
 *      This results in a slight increase in performance.
 */
static void mdlStart(SimStruct *S)
{
    SFcnCache *cache = malloc(sizeof(SFcnCache));

    if (cache == NULL) {
        ssSetErrorStatus(S,"memory allocation error");
        return;
    }

    ssSetUserData(S, cache);

    if (mxGetScalar(XDATAEVENLYSPACED(S)) != 0.0){
        cache->evenlySpaced = TRUE;
    }else{
        cache->evenlySpaced = FALSE;
    }

}
#endif /*  MDL_START */



/* Function: mdlOutputs ==============================================================
 * Abstract:
 *    In this function, we compute the outputs of our S-function
 *    block. Generally outputs are placed in the output vector, ssGetY(S).
 */
static void mdlOutputs(SimStruct *S, int_T tid)
{
    SFcnCache         *cache = ssGetUserData(S);
    real_T            *xData = mxGetPr(XVECT(S));
    real_T            *yData = mxGetPr(YVECT(S));
    InputRealPtrsType uPtrs  = ssGetInputPortRealSignalPtrs(S,0);
    real_T            *y     = ssGetOutputPortRealSignal(S,0);
    int_T             ny     = ssGetOutputPortWidth(S,0);
    int_T             xLen   = mxGetNumberOfElements(XVECT(S));
    int_T             i;
```

```
    /*
     * When the XData is evenly spaced, we use the direct lookup algorithm
     * to calculate the lookup
     */
    if (cache->evenlySpaced) {
        real_T spacing = xData[1] - xData[0];
        for (i = 0; i < ny; i++) {
            real_T u = *uPtrs[i];

            if (u <= xData[0]) {
                y[i] = yData[0];
            } else if (u >= xData[xLen-1]) {
                y[i] = yData[xLen-1];
            } else {
                int_T idx = (int_T)((u - xData[0])/spacing);
                y[i] = yData[idx];
            }
        }
    } else {
        /*
         * When the XData is unevenly spaced, we use a bisection search to
         * locate the lookup index.
         */
        for (i = 0; i < ny; i++) {
            int_T idx = GetDirectLookupIndex(xData,xLen,*uPtrs[i]);
            y[i] = yData[idx];
        }
    }

} /* end mdlOutputs */



/* Function: mdlTerminate ======================================================
 * Abstract:
 *    Free the cache that was allocated in mdlStart.
 */
static void mdlTerminate(SimStruct *S)
{
    SFcnCache *cache = ssGetUserData(S);
    if (cache != NULL) {
        free(cache);
    }
} /* end mdlTerminate */



#define MDL_RTW                        /* Change to #undef to remove function */
#if defined(MDL_RTW) && (defined(MATLAB_MEX_FILE) || defined(NRT))
/* Function: mdlRTW ============================================================
 * Abstract:
 *    This function is called when the Real-Time Workshop is generating the
 *    model.rtw file. In this routine, you can call the following functions
```

```
 *    which add fields to the model.rtw file.
 *
 *    Important! Since this S-function has this mdlRTW routine, it must have
 *    a corresponding .tlc file to work with the Real-Time Workshop. You will find
 *    the sfun_directlook.tlc in the same directory as sfun_directlook.dll.
 */
static void mdlRTW(SimStruct *S)
{
    /*
     * Write out the [X,Y] data as parameters, i.e., these values can be
     * changed during execution.
     */
    {
        real_T *xData = mxGetPr(XVECT(S));
        int_T  xLen   = mxGetNumberOfElements(XVECT(S));
        real_T *yData = mxGetPr(YVECT(S));
        int_T  yLen   = mxGetNumberOfElements(YVECT(S));

        if (!ssWriteRTWParameters(S,2,
                                  SSWRITE_VALUE_VECT,"XData","",xData,xLen,
                                  SSWRITE_VALUE_VECT,"YData","",yData,yLen)) {
            return; /* An error occurred which will be reported by Simulink */
        }
    }
    /*
     * Write out the spacing setting as a param setting, i.e., this cannot be
     * changed during execution.
     */
    {
        boolean_T even = (mxGetScalar(XDATAEVENLYSPACED(S)) != 0.0);

        if (!ssWriteRTWParamSettings(S, 1,
                                     SSWRITE_VALUE_QSTR,
                                     "XSpacing",
                                     even ? "EvenlySpaced" : "UnEvenlySpaced")){
            return;/* An error occurred which will be reported by Simulink */
        }
    }
}
#endif /* MDL_RTW */


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

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

```
/* [EOF] sfun_directlook.c */
```

matlabroot/simulink/src/lookup_index.c

```
/* File    : lookup_index.c
 * Abstract:
 *
 *     Contains a routine used by the S-function sfun_directlookup.c to
 *     compute the index in a vector for a given data value.
 *
 *  Copyright (c) 1990-1998 by The MathWorks, Inc. All Rights Reserved.
 *
 */
#include "tmwtypes.h"

/*
 * Function: GetDirectLookupIndex ==============================================
 * Abstract:
 *     Using a bisection search to locate the lookup index when the x-vector
 *     isn't evenly spaced.
 *
 *     Inputs:
 *        *x   : Pointer to table, x[0] ....x[xlen-1]
 *        xlen : Number of values in xtable
 *        u    : input value to look up
 *
 *     Output:
 *        idx  : the index into the table such that:
 *              if u is negative
 *                 x[idx] <= u < x[idx+1]
 *              else
 *                 x[idx] < u <= x[idx+1]
 */
int_T GetDirectLookupIndex(const real_T *x, int_T xlen, real_T u)
{
    int_T idx    = 0;
    int_T bottom = 0;
    int_T top    = xlen-1;

    /*
     * Deal with the extreme cases first:
     *
     *  i] u <= x[bottom] then idx = bottom
     * ii] u >= x[top] then idx = top-1
     *
     */
    if (u <= x[bottom]) {
        return(bottom);
    } else if (u >= x[top]) {
        return(top);
    }
```

```
    /*
     * We have: x[bottom] < u < x[top], onward
     * with search for the appropriate index ...
     */
    for (;;) {
        idx = (bottom + top)/2;
        if (u < x[idx]) {
            top = idx;
        } else if (u > x[idx+1]) {
            bottom = idx + 1;
        } else {
            /*
             * We have:  x[idx] <= u <= x[idx+1], only need
             * to do two more checks and we have the answer.
             */
            if (u < 0) {
                /*
                 * We want right continuity, i.e.,
                 * if u == x[idx+1]
                 *    then x[idx+1] <= u < x[idx+2]
                 * else    x[idx  ] <= u < x[idx+1]
                 */
                return( (u == x[idx+1]) ? (idx+1) : idx);
            } else {
                /*
                 * We want left continuity, i.e.,
                 * if u == x[idx]
                 *    then x[idx-1] < u <= x[idx  ]
                 * else    x[idx  ] < u <= x[idx+1]
                 */
                return( (u == x[idx]) ? (idx-1) : idx);
            }
        }
    }
} /* end GetDirectLookupIndex */

/* [EOF] lookup_index.c */
```

matlabroot/toolbox/simulink/blocks/tlc_c/sfun_directlook.tlc

```
%% File    : sfun_directlook.tlc
%% Abstract:
%%      Level-2 S-function sfun_directlook block target file.
%%      It is using direct lookup algorithm without interpolation.
%%
%% Copyright (c) 1994-1998 by The MathWorks, Inc. All Rights Reserved.
%%


%implements "sfun_directlook" "C"

%% Function: BlockTypeSetup ====================================================
%% Abstract:
%%     Place include and function prototype in the model's header file.
%%
%function BlockTypeSetup(block, system) void

  %% Add this external function's prototype in the header of the generated
  %% file.
  %%
  %openfile buffer
  extern int_T GetDirectLookupIndex(const real_T *x, int_T xlen, real_T u);
  %closefile buffer

  %<LibCacheFunctionPrototype(buffer)>

%endfunction

%% Function: mdlOutputs ========================================================
%% Abstract:
%%      Direct 1-D lookup table S-function example.
%%      Here we are trying to compute an approximate solution p(x) to an
%%      unknown function f(x) at x=x0, given data point pairs (x,y) in the
%%      form of an x-data vector and a y-data vector. For a given data pair
%%      (say the ith pair), we have y_i = f(x_i). It is assumed that the x
%%      data values are monotonically increasing.  If the first or last x is
%%      outside of the range of the x data vector, then the first or last
%%      point will be returned.
%%
%%      This function returns the "nearest" y0 point for a given x0.
%%      No interpolation is performed.
%%
%%      The S-function parameters are:
%%        XData
%%        YData
%%        XEvenlySpaced: 0 or 1
%%      The third parameter cannot be changed during execution and is
%%      written to the model.rtw file in XSpacing field of the SFcnParamSettings
%%      record as "EvenlySpaced" or "UnEvenlySpaced". The first two parameters
%%      can change during execution and show up in the parameter vector.
%%
```

```
%function Outputs(block, system) Output
  /* %<Type> Block: %<Name> */
  {
    %assign rollVars = ["U", "Y"]
    %%
    %% Load XData and YData as local variables
    %%
    real_T *xData  = %<LibBlockParameterAddr(XData, "", "", 0)>;
    real_T *yData  = %<LibBlockParameterAddr(YData, "", "", 0)>;
    %assign xDataLen = SIZE(XData.Value, 1)
    %%
    %% When the XData is evenly spaced, we use the direct lookup algorithm
    %% to locate the lookup index.
    %%
    %if SFcnParamSettings.XSpacing == "EvenlySpaced"
      real_T spacing = xData[1] - xData[0];

      %roll idx = RollRegions, lcv = RollThreshold, block, "Roller", rollVars
        %assign u = LibBlockInputSignal(0, "", lcv, idx)
        %assign y = LibBlockOutputSignal(0, "", lcv, idx)
        if ( %<u> <= xData[0] ) {
          %<y> = yData[0];
        } else if ( %<u> >= yData[%<xDataLen-1>] ) {
          %<y> = yData[%<xDataLen-1>];
        } else {
          int_T idx = (int_T)( ( %<u> - xData[0] ) / spacing );
          %<y> = yData[idx];
        }
        %%
        %% Generate an empty line if we are not rolling,
        %% so that it looks nice in the generated code.
        %%
        %if lcv == ""

        %endif
      %endroll
    %else
      %% When the XData is unevenly spaced, we use a bisection search to
      %% locate the lookup index.
      int_T idx;

      %assign xDataAddr = LibBlockParameterAddr(XData, "", "", 0)
      %roll idx = RollRegions, lcv = RollThreshold, block, "Roller", rollVars
        %assign u = LibBlockInputSignal(0, "", lcv, idx)
        idx =  GetDirectLookupIndex(xData, %<xDataLen>, %<u>);
        %assign y = LibBlockOutputSignal(0, "", lcv, idx)
        %<y> = yData[idx];
        %%
        %% Generate an empty line if we are not rolling,
        %% so that it looks nice in the generated code.
        %%
        %if lcv == ""
```

```
        %endif
      %endroll
    %endif
  }

%endfunction

%% EOF: sfun_directlook.tlc
```

Creating Code-Reuse-Compatible S-Functions

The code reuse feature of the Real-Time Workshop generates code for a subsystem in the form of a function that is invoked wherever the subsystem occurs in the model (see “Nonvirtual Subsystem Code Generation Options” in the online Real-Time Workshop documentation). If a subsystem contains S-functions, the S-functions must be compatible with the code reuse feature. Otherwise, the Real-Time Workshop may not generate reusable code from the subsystem or may generate incorrect code.

If you want your S-function to support the subsystem code reuse feature, you must ensure that the S-function meets the following requirements:

- The S-function must be inlined.
- Code generated from the S-function must not use static variables.
- The TLC code that generates the inlined S-function code must not use the `BlockInstanceData` function.
- The S-function must initialize its pointer work vector in `mdlStart` and not before.
- The S-function must not be a sink that logs data to the workspace.
- The S-function must register its parameters as run time parameters in `mdlSetWorkWidths`. (It must not use `ssWriteRTWParameters` in its `mdlRTW` function for this purpose.)

In addition to meeting the preceding requirements, your S-function must set the `SS_OPTION_WORKS_WITH_CODE_REUSE` flag (see `ssSetOptions`). This flag assures RTW that your S-function meets the requirements for subsystem code reuse.

9

S-Function Callback Methods

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

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

Syntax `void mdlCheckParameters(SimStruct *S)`

Arguments

`S`
SimStruct 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.

```
#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 routine 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

See Also

`mdlProcessParameters, ssGetSFcnParamsCount`

mdlDerivatives

Purpose Compute the S-function's derivatives.

Syntax `void mdlDerivatives(SimStruct *S)`

Arguments `S`
SimStruct representing an S-Function block.

Description 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 `ssGetdX` to get a pointer to the derivatives vector.

Each time the `mdlDerivatives` 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.

Example For an example, see *`matlabroot`*`/simulink/src/csfunc.c`.

Languages Ada, C, M

See Also `ssGetdx`

Purpose Initialize the state vectors of this S-function.

Syntax `void mdlGetTimeOfNextVarHit(SimStruct *S)`

Arguments

`S`
SimStruct representing an S-Function block.

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

Note The time of 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`, `ssSetTNext`, `ssGetT`

mdlInitializeConditions

Purpose

Initialize the state vectors of this S-function.

Syntax

`void mdlInitializeConditions(SimStruct *S)`

Arguments

`S`
SimStruct representing an S-Function block.

Description

Simulink invokes this optional method at the beginning of a simulation. It should initialize the continuous and discrete states, if any, of this S-Function block. Use `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 `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

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]
```

  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`

Purpose Specify the number of inputs, outputs, states, parameters, and other characteristics of the S-function.

Syntax `void mdlInitializeSizes(SimStruct *S)`

Arguments `S`

SimStruct representing an S-Function block.

Description This is the first of the S-function's callback methods that Simulink calls. This method should perform the following tasks:

- Specify the number of parameters that this S-function supports, using `ssSetNumSFcnParams`.

  Use `ssSetSFcnParamTunable(S,paramIdx, 0)` when a parameter cannot change during simulation, where `paramIdx` 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 `ssSetNumContStates` and `ssSetNumDiscStates`.
- Configure the block's input ports.

  This entails the following tasks:

  - Specify the number of input ports that this S-function has, using `ssSetNumInputPorts`.
  - Specify the dimensions of the input ports.

    See"Dynamically Sized Block Features" on page 9-13 for more information.
  - For each input port, specify whether it has direct feedthrough, using `ssSetInputPortDirectFeedThrough`.

    A port has direct feedthrough if the input is used in either the `mdlOutputs` or `mdlGetTimeOfNextVarHit` 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 7-15 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`.

- 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 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 aparameter 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);

    /*
     * 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 */
```

mdlOutputs

Purpose Compute the signals that this block emits.

Syntax

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

Arguments

`S`
SimStruct representing an S-Function block.

`tid`
Task ID.

Description 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.

The `tid` (task ID) argument specifies the task running when the `mdlOutputs` routine is invoked. You can use this argument in the `mdlOutports` routine of a multirate S-Function block to encapsulate task-specific blocks of code (see "Multirate S-Function Blocks" on page 7-23).

For an example of an `mdlOutputs` routine that works with multiple input and output ports, see `matlabroot/simulink/src/sfun_multiport.c`.

Languages A, C, M

See Also `ssGetOutputPortSignal`, `ssGetOutputPortRealSignal`, `ssGetOutputPortComplexSignal`

Purpose Process the S-function's parameters.

Syntax

```
void mdlProcessParameters(SimStruct *S)
```

Arguments

`S`
SimStruct representing an S-Function block.

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

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

The synopsis is

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

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

```
#define MDL_PROCESS_PARAMETERS   /* Change to #undef to remove function */
```

```
#if defined(MDL_PROCESS_PARAMETERS) && defined(MATLAB_MEX_FILE)
static void mdlProcessParameters(SimStruct *S)
  {
    int_T  i;
    char_T *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)
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.

Syntax `void mdlRTW(SimStruct *S)`

Arguments `S`
SimStruct representing an S-Function block.

Description This function is called when the Real-Time Workshop is generating the *`model`*`.rtw` file. In this method, you can call the following functions that add fields to the *`model`*`.rtw` file:

- `ssWriteRTWParameters`
- `ssWriteRTWParamSettings`
- `ssWriteRTWWorkVect`
- `ssWriteRTWStr`
- `ssWriteRTWStrParam`
- `ssWriteRTWScalarParam`
- `ssWriteRTWStrVectParam`
- `ssWriteRTWVectParam`
- `ssWriteRTW2dMatParam`
- `ssWriteRTWMxVectParam`
- `ssWriteRTWMx2dMatParam`

Languages C

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 `void mdlSetDefaultPortComplexSignals(SimStruct *S)`

Arguments `S`
SimStruct representing an S-Function block.

Description 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 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`

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 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.

If the block does not implement this method and at least one port is known to be complex, Simulink sets the unknown ports to `COMPLEX_YES`; otherwise, it sets the unknown ports to `COMPLEX_NO`.

Languages

C

See Also

`ssSetOutputPortComplexSignal`, `ssSetInputPortComplexSignal`

mdlSetDefaultPortDimensionInfo

Purpose
Set the default dimensions of the signals accepted or emitted by an S-function's ports.

Syntax
`void mdlSetDefaultPortDimensionInfo(SimStruct *S, int_T port)`

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

Languages
C

See Also
`ssSetOutputPortMatrixDimensions`, `ssSetErrorStatus`

Purpose

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

Syntax

```
void mdlSetInputPortDataType(SimStruct *S, int_T port, CSignal_T
   csig)
```

Arguments

`S`
SimStruct representing an S-Function block.

`port`
Index of a port.

`csig`
Numeric type of signal.

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

See Also

`ssSetInputPortComplexSignal`, `ssSetErrorStatus`

mdlSetInputPortDataType

Purpose

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

Syntax

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

Arguments

`S`
SimStruct representing an S-Function block.

`port`
Index of a port.

`id`
Data type ID.

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

`ssSetInputPortDataType`, `ssSetErrorStatus`

mdlSetInputPortDimensionInfo

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

Syntax

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

Arguments

`S`
SimStruct representing an S-Function block.

`port`
Index of a port.

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

See `ssSetInputPortDimensionInfo` for a description of this structure.

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

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

See Also `ssSetErrorStatus`

Purpose Set frame data entering an input port.

Syntax

```
void mdlSetInputPortFrameData(SimStruct *S, int_T port,
   Frame_T frameData)
```

Arguments

`S`
SimStruct representing an S-Function block.

`port`
Index of a port.

`frameData`
Frame data.

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`.

Languages C

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.

Syntax

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

Arguments

`S`
SimStruct representing an S-Function block.

`port`
Index of a port.

`sampleTime`
Inherited sample time for `port`.

`offsetTime`
Inherited offset time for `port`.

Description

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

When inherited port-based sample times are specified, the sample time is guaranteed to be one of the following.

	Sample Time	Offset Time
Continuous	`0.0`	`0.0`
Discrete	`period`	`offset`

where `0.0 < period < inf` and `0.0 <= offset < period`. 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 [0.2499999999999, 0] to [0.25, 0].

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

Languages

C

See Also

ssSetInputPortSampleTime, ssSetOutputPortSampleTime, mdlInitializeSampleTimes

mdlSetInputPortWidth

Purpose

Set the width of an input port that accepts 1-D (vector) signals.

Syntax

```
void mdlSetInputPortWidth(SimStruct *S, int_T port, int_T width)
```

Arguments

`S`
SimStruct representing an S-Function block.

`port`
Index of a port.

`width`
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 `ssSetInputPortWidth`. If the size is unacceptable, an error should be generated via `ssSetErrorStatus`. 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 `ssSetInputPortWidth` or `ssSetOutputPortWidth`.

Languages

C

See Also

`ssSetInputPortWidth`, `ssSetOutputPortWidth`, `ssSetErrorStatus`

Purpose

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

Syntax

```
void mdlSetOutputPortDataType(SimStruct *S, int_T port, CSignal_T
   csig)
```

Arguments

`S`
SimStruct representing an S-Function block.

`port`
Index of a port.

`csig`
Numeric type of signal.

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.

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

See Also

`ssSetOutputPortComplexSignal`, `ssSetErrorStatus`

mdlSetOutputPortDataType

Purpose Set the data type of the signals emitted by an output port.

Syntax

```
void mdlSetOutputPortDataType(SimStruct *S, int_T port, DTypeId id)
```

Arguments

`S`
SimStruct representing an S-Function block.

`port`
Index of an output port.

`id`
Data type ID.

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

See Also `ssSetOutputPortDataType`, `ssSetErrorStatus`

mdlSetOutputPortDimensionInfo

Purpose

Set the dimensions of the signals accepted by an output port.

Syntax

```
void mdlSetOutputPortDimensionInfo(SimStruct *S, int_T port, const
   DimsInfo_T *dimsInfo)
```

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.

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

Example

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

See Also `ssSetOutputPortDimensionInfo`, `ssSetErrorStatus`

Purpose

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

Syntax

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

Arguments

`S`
SimStruct representing an S-Function block.

`port`
Index of a port.

`sampleTime`
Inherited sample time for `port`.

`offsetTime`
Inherited offset time for `port`.

Description

Simulink 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`. 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`.

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`

mdlSetOutputPortWidth

Purpose Set the width of an output port that outputs 1-D (vector) signals.

Syntax

```
void mdlSetOutputPortWidth(SimStruct *S, int_T port, int_T width)
```

Arguments

`S`
SimStruct representing an S-Function block.

`port`
Index of a port.

`width`
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 `ssSetOutputPortWidth`. If the size is unacceptable, an error should be generated via `ssSetErrorStatus`. 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 `ssSetInputPortWidth` or `ssSetOutputPortWidth`.

Languages C

See Also `ssSetInputPortWidth`, `ssSetOutputPortWidth`, `ssSetErrorStatus`

Purpose

Specify the sizes of the work vectors and create the run-time parameters required by this S-function.

Syntax

`void mdlSetWorkWidths(SimStruct *S)`

Arguments

`S`
SimStruct representing an 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 7-5). 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`.

Languages

Ada, C

See Also

`mdlInitializeSizes`

Purpose Initialize the state vectors of this S-function.

Syntax

```
void mdlStart(SimStruct *S)
```

Arguments

`S`
SimStruct representing an S-Function block.

Description Simulink invokes this optional method at the beginning of a simulation. It should initialize the continuous and discrete states, if any, of this S-Function block. Use `ssGetContStates` and/or `ssGetDiscStates` to get the states. This method can also perform any other initialization activities that this S-function requires.

Languages Ada, C

See Also `mdlInitializeConditions`, `ssGetContStates`, `ssGetDiscStates`

Purpose Perform any actions required at termination of the simulation.

Syntax `void mdlTerminate(SimStruct *S)`

Arguments

`S`
SimStruct representing an S-Function block.

Description This method should perform any actions, such as freeing memory, that must be performed at the end of simulation or when an S-Function block is destroyed (e.g., when it is deleted from a model). The option `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.

Syntax `void mdlUpdate(SimStruct *S, int_T tid)`

Arguments

`S`
SimStruct representing an S-Function block.

`tid`
Task ID.

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 7-23).

Example For an example, see `*matlabroot*/simulink/src/dsfunc.c`.

Languages Ada, C, M

See Also `mdlDerivatives`, `ssGetContStates`, `ssGetDiscStates`

Purpose Update zero-crossing vector.

Syntax

```
void mdlZeroCrossings(SimStruct *S)
```

Arguments

`S`
SimStruct representing an 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 `ssGetNonsampleZCs(S)` vector.

Example See *`matlabroot`*`/simulink/src/sfun_zc.c`.

Languages C

See Also `mdlInitializeSizes`, `ssGetNonsampledZCs`

10

SimStruct Functions

This sections describes SimStruct macros and functions.

Introduction

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

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

Language Support

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

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

The SimStruct

The file *`matlabroot`*`/simulink/include/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.

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.
`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

Macros	Description
`ssGetErrorStatus`	Get a string that identifies the last error.
`ssPrintf`	Print a variable-content `msg`.

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

I/O Port

Macro	Description
ssGetInputPortBufferDstPort	Determine the output port that is overwriting an input port's memory buffer.
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.
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.

Macro	Description
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.
ssGetInputPortReusable	Determine whether memory allocated to the input port is reusable.
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).
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.

Macro	Description
`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.
`ssGetOutputPortReusable`	Determine whether memory allocated to the output port is reusable
`ssGetOutputPortSampleTime`	Determine the sample time of an output port.
`ssGetOutputPortSignal`	Get the vector of signal elements emitted by an output port.
`ssGetOutputPortSignalAddress`	Get 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.

Macro	Description
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.
ssSetInputPortReusable	Specify whether an input port's memory buffer can be reused by other signals in the model.
ssSetInputPortSampleTime	Set the sample time of an input port.
ssSetInputPortSampleTimeIndex	Specify the sample time index of an input port.
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.

Macro	Description
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.
ssSetOutputPortReusable	Specify whether an output port's memory can be reused.
ssSetOutputPortSampleTime	Specify the sample time of an output port.
ssSetOutputPortVectorDimension	Specify dimension information for an output port that emits vector signals
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.

Dialog Box Parameters

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

Macro	Description
`ssGetDTypeIdFromMxArray`	Return 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.
`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).
`ssGetSFcnParamsCount`	Get the actual number of parameters specified by the user.
`ssSetSFcnParamNotTunable`	Obsolete.
`ssSetSFcnParamTunable`	Specify the tunability of a dialog box parameter.

Run-Time Parameters

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

Macro	Description
`ssRegDlgParamAsRunTimeParam`	Register a run-time parameter.
`ssUpdateDlgParamAsRunTimeParam`	Update a run-time parameter.
`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.
`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.
`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
ssGetSampleTimeOffset	Get the offset of the current sample time (Ada only).
ssGetSampleTimePeriod	Get the period of the current sample time (Ada only).
ssGetTNext	Get the time of the next sample hit in a discrete S-function with a variable sample time.
ssGetNumSampleTimes	Get the number of sample times an S-function has.
ssGetPortBasedSampleTimeBlockIsTriggered	Determine whether a block that uses port-based sample times resides in a triggered subsystem.
ssIsContinuousTask	Determine whether a specified rate is the continuous rate.
ssIsSampleHit	Determine the sample rate at which an S-function is operating.
ssIsSpecialSampleHit	Determine whether the current sample time hits two specified rates.
ssSampleAndOffsetAreTriggered	Determine whether a sample time and offset value pair indicate a triggered sample time.
ssSetNumSampleTimes	Set the number of sample times an S-function has.
ssSetOffsetTime	Specify the offset of a sample time.

Macro	Description
`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.
`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 (`int_T`) 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.

Macro	Description
`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 (`real_T`) work vector.
`ssGetPWork`	Get an S-function's pointer (`void *`) work vector.
`ssGetPWorkValue`	Get a pointer from a pointer work vector.
`ssGetRealDiscStates`	Get the real (`real_T`) values of an S-function's discrete state vector.
`ssGetRWork`	Get an S-function's real-valued (`real_T`) work vector.
`ssGetRWorkValue`	Get an element of an S-function's real-valued work vector.
`ssSetDWorkComplexSignal`	Specify whether the elements of a data type work vector are real or complex.

Macro	Description
`ssSetDWorkDataType`	Specify the data type of a data type work vector.
`ssSetDWorkName`	Specify the name of a data type work vector.
`ssSetDWorkUsedAsDState`	Specify that a data type work vector is used as a discrete state vector.
`ssSetDWorkWidth`	Specify the width of a data type work vector.
`ssSetIWorkValue`	Set an element of a block's integer work vector.
`ssSetModeVectorValue`	Set an element of a block's mode vector.
`ssSetNumContStates`	Specify the number of continuous states that an S-function has.
`ssSetNumDiscStates`	Specify the number of discrete states that an S-function has.
`ssSetNumDWork`	Specify the number of data type work vectors used by a block.
`ssSetNumIWork`	Specify the size of an S-function's integer (`int_T`) work vector.
`ssSetNumModes`	Specify the number of operating modes that an S-function has.
`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.

Macro	Description
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.
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.

Macro	Description
`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 if 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.
`ssSetSolverNeedsReset`	Ask Simulink to reset the solver.
`ssSetStopRequested`	Ask Simulink to terminate the simulation at the end of the current time step.

Function Call

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

Data Type

Macro	Description
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.
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.
`ssGetDWorkRTWTypeQualifier`	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
`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 `model.rtw` file.
`ssWriteRTWMx2dMatParam`	Write a MATLAB matrix parameter to the S-function's `model.rtw` file.

Macro	Description
ssWriteRTWMxVectParam	Write a MATLAB vector parameter to the S-function's model.rtw file.
ssWriteRTWParameters	Write tunable parameters to the S-function's model.rtw file.
ssWriteRTWParamSettings	Write settings for the S-function's parameters to the model.rtw file.
ssWriteRTWScalarParam	Write a scalar parameter to the S-function's model.rtw file.
ssWriteRTWStr	Write a string to the S-function's model.rtw file.
ssWriteRTWStrParam	Write a string parameter to the S-function's model.rtw file.
ssWriteRTWStrVectParam	Write a string vector parameter to the S-function's model.rtw file
ssWriteRTWVectParam	Write a Simulink vector parameter to the S-function's model.rtw file.
ssWriteRTWWorkVect	Write the S-function's work vectors to the model.rtw file.

Macro Reference

This section contains descriptions of each SimStruct macro.

Purpose

Invoke the external mode function for an S-function.

Syntax

```
void ssCallExternalModeFcn(SimStruct *S, SFunExtModeFcn *fcn)
```

Arguments

`S`
SimStruct representing an S-Function block or a Simulink model.

`fcn`
External mode function.

Description

Specifies the external mode function for `S`.

Languages

C

See Also

`ssSetExternalModeFcn`

ssCallSystemWithTid

Purpose

Specify that an output port is issuing a function call.

Syntax

```
ssCallSystemWithTid(SimStruct *S, port_index, tid)
```

Arguments

`S`
SimStruct representing an S-Function block or a Simulink model.

`port_index`
Index of the port that is issuing the function call.

`tid`
Task ID.

Description

Use in `mdlOutputs` to execute a function-call subsystem connected to the S-function. The invoking syntax is

```
if (!ssCallSystemWithTid(S,index, tid)) {
  /* Error occurred which will be reported by      Simulink */
  return;
}
```

Languages

C

See Also

`ssSetCallSystemOutput`

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

Syntax

```
real_T *ssGetAbsTol(SimStruct *S)
```

Arguments

S
SimStruct representing an S-Function block.

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

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

Languages C, C++

Example

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

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

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

See Also `ssGetStateAbsTol`, `ssIsVariableStepSolver`

ssGetBlockReduction

Purpose

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.

Syntax

```
unsigned int_T ssGetBlockReduction(SimStruct *S)
```

Arguments

`S`
SimStruct representing an S-Function block.

Description

The result of this function depends on when it is invoked. When invoked before the simulation loop has started, i.e., in `mdlSetWorkWidths` or earlier, this macro returns true if the block has previously requested that it be reduced. When invoked after the simulation loop has begun, this macro returns true if the block has actually been reduced, i.e., eliminated from the list of blocks to be executed during the simulation loop.

Note If a block has been reduced, the only callback method invoked for the block after the simulation loop has begun is the block's `mdlTerminate` method. Further, Simulink invokes the `mdlTerminate` method only if the block has set its `SS_OPTION_CALL_TERMINATE_AT_EXIT` option, using `ssSetOptions`. Thus, if your block needs to determine whether it has actually been reduced, it must set the `SS_OPTION_CALL_TERMINATE_AT_EXIT` option before the simulation loop has begun and invoke `ssGetBlockReduction` in its `mdlTerminate` method.

Languages

C

See Also

`ssSetBlockReduction`

ssGetContStateAddress

Purpose
Get the address of a block's continuous state vector.

Ada Syntax
`ssGetContStateAddress(S : in SimStruct) return System.Address`

Arguments
`S`
SimStruct representing an S-Function block.

Description
Can be used in the simulation loop, `mdlInitializeConditions`, or `mdlStart` routines to get the address of the S-function's continuous state vector. This vector has length `ssGetNumContStates(S)`. Typically, this vector is initialized in `mdlInitializeConditions` and used in `mdlOutputs`.

Languages
Ada

See Also
`ssGetNumContStates`, `ssGetRealDiscStates`, `ssGetdX`, `mdlInitializeConditions`, `mdlStart`

ssGetContStates

Purpose

Get a block's continuous states.

Syntax

```
real_T *ssGetContStates(SimStruct *S)
```

Arguments

`S`
SimStruct representing an S-Function block.

Description

Can be used in the simulation loop, `mdlInitializeConditions`, or `mdlStart` routines to get the `real_T` continuous state vector. This vector has length `ssGetNumContStates(S)`. Typically, this vector is initialized in `mdlInitializeConditions` and used in `mdlOutputs`.

Languages

C

See Also

`ssGetNumContStates`, `ssGetRealDiscStates`, `ssGetdX`, `mdlInitializeConditions`, `mdlStart`

Purpose Get the ID of a data type.

Syntax

```
DTypeID ssGetDataTypeId(SimStruct *S, char *name)
```

Arguments

`S`
SimStruct representing an S-Function block.

`name`
Name of a data type.

Description Returns the ID of the data type specified by `name` if `name` is a registered type name. Otherwise, this macro returns `INVALID_DTYPE_IDL` and reports an error. Because this macro reports any error that occurs, you do not need to use `ssSetErrorStatus` to report the error.

Languages C

Example The following example gets the ID of the data type named `Color`.

```
int_T id = ssGetDataTypeId (S, "Color");
if(id == INVALID_DTYPE_ID) return;
```

See Also `ssRegisterDataType`

ssGetDataTypeName

Purpose

Get the name of a data type.

Syntax

```
char *ssGetDataTypeName(SimStruct *S, DTypeId id)
```

Arguments

`S`
SimStruct representing an S-Function block.

`id`
ID of data type.

Description

Returns the name of the data type specified by `id`, if `id` is valid. Otherwise, this macro returns `NULL` and reports an error. Because this macro reports any error that occurs, you do not need to use `ssSetErrorStatus` to report the error.

Example

The following example gets the name of a custom data type.

```
const char *dtypeName = ssGetDataName(S, id);
if(dtypeName == NULL) return;
```

Languages

C

See Also

`ssRegisterDataType`

ssGetDataTypeSize

Purpose

Get the size of a custom data type.

Syntax

```
GetDataTypeSize(SimStruct *S, DTypeId id)
```

Arguments

`S`
SimStruct representing an S-Function block.

`id`
ID of data type.

Description

Returns the size of the data type specified by `id`, if `id` is valid and the data type's size has been set. Otherwise, this macro returns `INVALID_DTYPE_SIZE` and reports an error.

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

Languages

C

Example

The following example gets the size of the `int16` data type.

```
int_T size = ssGetDataTypeSize(S, SS_INT16);
if(size == INVALID_DTYPE_SIZE) return;
```

See Also

`ssSetDataTypeSize`

ssGetDataTypeZero

Purpose Get the zero representation of a data type.

Syntax

```
void* ssGetDataTypeZero(SimStruct *S, DTypeId id)
```

Arguments

`S`
SimStruct representing an S-Function block.

`id`
ID of data type.

Description Returns a pointer to the zero representation of the data type specified by `id`, if `id` is valid and the data type's size has been set. Otherwise, this macro returns `NULL` and reports an error. Because this macro reports any error that occurs, you do not need to use `ssSetErrorStatus` to report the error.

Languages C

Example The following example gets the zero representation of a custom data type.

```
const void *myZero = ssGetDataTypeZero(S, id);
if(myZero == NULL) return;
```

See Also `ssRegisterDataType`, `ssSetDataTypeSize`, `ssSetDataTypeZero`

Purpose Get a block's discrete states.

Syntax

```
real_T *ssGetDiscStates(SimStruct *S)
```

Arguments

`S`
SimStruct representing an S-Function block.

Description Returns a block's discrete state vector has an array of `real_T` elements of length `ssGetNumDiscStates(S)`. Typically, the state vector is initialized in `mdlInitializeConditions`, updated in `mdlUpdate`, and used in `mdlOutputs`. You can use this macro in the simulation loop, `mdlInitializeConditions`, or `mdlStart` routines.

Languages C

See Also `ssGetNumDiscStates`, `mdlInitializeConditions`, `mdlUpdate`, `mdlOutputs`, `mdlStart`

ssGetDTypeIdFromMxArray

Purpose

Get the data type of an S-function parameter.

Syntax

```
DTypeId ssGetDTypeIdFromMxArray(const mxArray *m)
```

Arguments

`m`
MATLAB array representing the parameter.

Description

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

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

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

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

Example

See the example in `matlabroot/simulink/src/sfun_dtype_param.c` to learn how to use data typed parameters in an S-function.

Languages

C

See Also

`ssGetSFcnParam`

ssGetDWork

Purpose Get a DWork vector.

Syntax

```
void *ssGetDWork(SimStruct *S, int_T vector)
```

Arguments

`S`
SimStruct representing an S-Function block.

`vector`
Index of a data type work vector, where the index is one of `0`, `1`, `2`, `...` `ssGetNumDWork(S)`.

Description Returns a pointer to the specified vector.

Languages C, C++

See Also `ssSetNumDWork`

Purpose

Determine whether the elements of a data type work vector are real or complex numbers.

Syntax

```
CSignal_T ssGetDWorkComplexSignal(SimStruct *S, int_T vector)
```

Arguments

`S`
SimStruct representing an S-Function block.

`vector`
Index of a data type work vector, where the index is one of `0`, `1`, `2`, `...` `ssGetNumDWork(S)`.

Description

Returns `COMPLEX_YES` if the specified vector contains complex numbers; otherwise, `COMPLEX_NO`.

Languages

C, C++

See Also

`ssSetDWorkComplexSignal`

ssGetDWorkDataType

Purpose Get the data type of a data type work vector.

Syntax

```
DTypeId ssGetDWorkDataType(SimStruct *S, int_T vector)
```

Arguments

`S`
SimStruct representing an S-Function block.

`vector`
Index of a data type work vector, where the index is one of `0`, `1`, `2`, `...` `ssGetNumDWork(S)`.

Description Returns the data type of the specified data type work vector.

Languages C, C++

See Also `ssSetDWorkDataType`

Purpose Get the name of a data type work vector.

Syntax

```
char_T *ssGetDWorkName(SimStruct *S, int_T vector)
```

Arguments

`S`
SimStruct representing an S-Function block.

`vector`
Index of the work vector, where the index is one of `0`, `1`, `2`, `...` `ssGetNumDWork(S)`.

Description Returns the name of the specified data type work vector.

Languages C, C++

See Also `ssSetDWorkName`

ssGetDWorkRTWIdentifier

Purpose

Get the identifier used to declare a DWork vector in code generated from the associated S-function.

Syntax

```
char_T * ssGetDWorkRTWIdentifier(SimStruct* S, int idx)
```

Arguments

`S`
SimStruct representing an S-Function block.

`idx`
Index of the work vector, where the index is one of `0`, `1`, `2`, `...` `ssGetNumDWork(S)`.

Description

Returns the identifier used in code generated by the Real-Time Workshop to declare the DWork vector specified by `idx`.

Languages

C, C++

See Also

`ssSetDWorkRTWIdentifier`

Purpose Get the storage class of a DWork vector in code generated from the associated S-function.

Syntax

```
ssRTWStorageType ssGetDWorkRTWStorageClass(SimStruct* S, int idx)
```

Arguments

`S`
SimStruct representing an S-Function block.

`idx`
Index of the work vector, where the index is one of `0`, `1`, `2`, `...` `ssGetNumDWork(S)`.

Description Returns the storage class of the the DWork vector specified by `idx`. The storage class is a code-generation attribute that determines how the code generated by the Real-Time Workshop for this S-function allocates memory for this work vector (see "Signal Storage Concepts" in the online documentation for the Real-Time Workshop). The returned storage class specifier is a value of type `ssRTWStorageType`:

```
typedef enum {
    SS_RTW_STORAGE_AUTO = 0,
    SS_RTW_STORAGE_EXPORTED_GLOBAL,
    SS_RTW_STORAGE_IMPORTED_EXTERN,
    SS_RTW_STORAGE_IMPORTED_EXTERN_POINTER
} ssRTWStorageType;
```

Languages C, C++

See Also `ssSetDWorkRTWStorageClass`

ssGetDWorkRTWTypeQualifier

Purpose Get the C type qualifier (e.g., `const`) used to declare a DWork vector in code generated from the associated S-function.

Syntax

```
char_T * ssGetDWorkRTWTypeQualifier(SimStruct* S, int idx)
```

Arguments

`S`
SimStruct representing an S-Function block.

`idx`
Index of the work vector, where the index is one of `0`, `1`, `2`, `...` `ssGetNumDWork(S)`.

Description Returns the C type qualifier (e.g., `const`) used to declare the DWork vector specified by `idx` in code generated by the Real-Time Workshop from the associated S-function.

Languages C, C++

See Also `ssSetDWorkRTWTypeQualifier`

Purpose Determine whether a data type work vector is used as a discrete state vector.

Syntax

```
int_T ssGetDWorkUsedAsDState(SimStruct *S, int_T vector)
```

Arguments

`S`
SimStruct representing an S-Function block.

`vector`
Index of a data type work vector, where the index is one of `0`, `1`, `2`, `...` `ssGetNumDWork(S)`.

Description Returns `SS_DWORK_USED_AS_DSTATE` if this vector is used to store a block's discrete states.

Languages C, C++

See Also `ssSetDWorkUsedAsDState`

ssGetDWorkWidth

Purpose Get the size of a data type work vector.

Syntax

```
int_T ssGetDWorkWidth(SimStruct *S, int_T vector)
```

Arguments

`S`
SimStruct representing an S-Function block.

`vector`
Index of a work vector, where the index is one of `0`, `1`, `2`, `...` `ssGetNumDWork(S)`.

Description Returns the number of elements in the specified work vector.

Languages C, C++

See Also `ssSetDWorkWidth`

Purpose Get the derivatives of a block's continuous states.

Syntax

```
(real_T *) ssGetdX(SimStruct *S)
```

Arguments

`S`
SimStruct representing an S-Function block.

Description Returns a pointer to an array containing the continuous states of S, which can be a block or the model. Use `ssGetNumContStates(S)` to get the length of the array. Use this macro in `mdlDerivatives` to get the derivatives of a model or block's continuous states.

Note The pointer returned by this macro changes as the solver evaluates different integration stages to compute the integral.

Languages C

See Also `ssGetNumContStates`, `ssGetContStates`

ssGetErrorStatus

Purpose Get a string that identifies the last error.

C Syntax

```
const char_T *ssGetErrorStatus(SimStruct *S)
```

Ada Syntax

```
const char_T *ssGetErrorStatus(SimStruct *S)
```

Arguments

`S`
SimStruct representing an S-Function block.

Description Returns a string that identifies the last error.

Languages Ada, C

See Also `ssSetErrorStatus`

Purpose

Determine whether the user has set the inline parameters option for the model containing this S-function.

Syntax

```
boolean_T ssGetInlineParameters(SimStruct *S)
```

Arguments

`S`
SimStruct representing an S-Function block.

Description

Returns `TRUE` if the user has checked the **Inline parameters** option on the **Advanced** pane of the **Simulation parameters** dialog box (see “The Advanced Pane” in the online Simulink documentation).

Languages

C

ssGetInputPortBufferDstPort

Purpose Determine the output port that is sharing this input port's buffer.

Syntax

```
ssGetInputPortBufferDstPort(SimStruct *S, int_T inputPortIdx)
```

Arguments

`S`
SimStruct representing an S-Function block.

`inputPortIdx`
Index of the port overwritten by an output port.

Description Use in any run-time S-function callback routine to determine the output port that is overwriting the specified input port. This can be used when you have specified the following:

- The input port and some output port on an S-function are *not* test points (`ssSetInputPortTestPoint` and `ssSetOutputPortTestPoint`).
- The input port is overwritable (`ssSetInputPortOverWritable`).

If you have this set of conditions, Simulink can use the same memory buffer for an input port and an output port. Simulink determines which ports share memory buffers. Use this function any time after model initialization to get the index of the output port that reuses the specified input port's buffer. If none of the S-function's output ports reuse this input port buffer, this macro returns `INVALID_PORT_IDX (= -1)`.

Languages C

See Also `ssSetNumInputPorts`, `ssSetInputPortOverWritable`

Purpose Get the numeric type (complex or real) of an input port.

Syntax

```
DTypeId ssGetInputPortComplexSignal(SimStruct *S,input_T port)
```

Arguments

`S`
SimStruct representing an S-Function block.

`port`
Index of an input port.

Description Returns the numeric type of `port`.

Languages C

See Also `ssSetInputPortComplexSignal`

ssGetInputPortConnected

Purpose Determine whether a port is connected to a nonvirtual block.

Syntax

```
int_T ssGetInputPortConnected(SimStruct *S, int_T port)
```

Arguments

`S`
SimStruct representing an S-Function block or a Simulink model.

`port`
Port whose connection status is needed.

Description Returns true if the specified `port` on the block represented by `S` is connected to a nonvirtual block. Can be invoked anywhere except in `mdlInitializeSizes` or `mdlCheckParameters`. The S-function must have previously set the number of input ports in `mdlInitializeSizes`, using `ssSetNumInputPorts`.

Languages C

See Also `ssSetNumInputPorts`

Purpose Get the data type of an input port.

C Syntax

```
DTypeId ssGetInputPortDataType(SimStruct *S,input_T port)
```

Ada Syntax

```
function  ssGetInputPortDataType(S : in SimStruct; port : in Integer
   := 0) return Integer;
```

Arguments

`S`
SimStruct representing an S-Function block or a Simulink model.

`port`
Index of an input port.

Description Returns the data type of the input port specified by `port`.

Languages Ada, C

See Also `ssSetInputPortDataType`

ssGetInputPortDimensions

Purpose Get the dimensions of the signal accepted by an input port.

Syntax

```
int_T *ssGetInputPortDimensions(SimStruct *S, int_T port)
```

Arguments

`S`
SimStruct representing an S-Function block.

`port`
Index of an input port.

Description Returns an array of integers that specifies the dimensions of the signal accepted by `port`, e.g., `[4 2]` for a 4-by-2 matrix array. The size of the dimensions array is equal to the number of signal dimensions accepted by the port, e.g., 1 for a vector signal or 2 for a matrix signal.

Languages C

See Also `ssGetInputPortNumDimensions`

Purpose Determine whether a port has direct feedthrough.

C Syntax

```
int_T ssGetInputPortDirectFeedThrough(SimStruct *S, int_T port)
```

Ada Syntax

```
function  ssGetInputPortDirectFeedThrough(S : in SimStruct;
   port : in Integer := 0) return Boolean;
```

Arguments

`S`
SimStruct representing an S-Function block.

`port`
Index of the port whose direct feedthrough property is required.

Description Use in any routine (except `mdlInitializeSizes`) to determine whether an input port has direct feedthrough.

Languages Ada, C

See Also `ssSetInputPortDirectFeedThrough`

ssGetInputPortFrameData

Purpose

Determine whether a port accepts signal frames.

Syntax

```
int_T ssGetInputPortFrameData(SimStruct *S,  int_T port)
```

Arguments

`S`
SimStruct representing an S-Function block.

`port`
Index of an input port.

Description

Returns one of the following:

- `-1`

  Port accepts either frame or unframed input.
- `0`

  Port accepts unframed input only.
- `1`

  Port accepts frame input only.

Languages

C

See Also

`ssSetInputPortFrameData`, `mdlSetInputPortFrameData`

Purpose Get the dimensionality of the signals accepted by an input port.

Syntax

```
int_T ssGetInputPortNumDimensions(SimStruct *S, int_T port)
```

Arguments

`S`
SimStruct representing an S-Function block.

`port`
Index of an input port.

Description Returns the number of dimensions of `port` or `DYNAMICALLY_SIZED`, if the number of dimensions is unknown.

Languages C

See Also `ssGetInputPortDimensions`

ssGetInputPortOffsetTime

Purpose Get the offset time of an input port.

Syntax

```
ssGetInputPortOffsetTime(SimStruct *S,inputPortIdx)
```

Arguments

`S`
SimStruct representing an S-Function block.

`inputPortIdx`
Index of the port whose offset time is required.

Description Use in any routine (except `mdlInitializeSizes`) to determine the offset time of an input port. This should only be used if you have specified the sample times as port-based.

Languages C

See Also `ssSetInputPortOffsetTime`, `ssGetInputPortSampleTime`

Purpose Determine whether an input port can be overwritten.

C Syntax

```
int_T ssGetInputPortOverWritable(SimStruct *S, int_T port)
```

Ada Syntax

```
function ssGetInputPortOverWritable(S : in SimStruct; port : in
   Integer := 0) return Boolean;
```

Arguments

`S`
SimStruct representing an S-Function block or a Simulink model.

`port`
Index of the input port whose overwritability is being set.

Description Returns true if the input port can be overwritten.

Languages Ada, C

See Also `ssSetInputPortOverWritable`

ssGetInputPortRealSignal

Purpose Get the address of a real, contiguous signal entering an input port.

Syntax

```
const real_T *ssGetInputPortRealSignal(SimStruct *S, inputPortIdx)
```

Arguments

`S`
SimStruct representing an S-Function block.

`inputPortIdx`
Index of the port whose sample time is required.

Description Returns the address of a real signal on the specified input port. A method should use this macro only if the input signal is known to be real and `mdlInitializeSizes` has specified that the elements of the input signal be contiguous, using `ssSetInputPortRequiredContiguous`.

Languages C, C++

Example The following code demonstrates the use of `ssGetInputPortRealSignal`.

Set flags to require that the input ports be contiguous:

```
void mdlInitializeSizes(SimStruct* S) {
   int_T i;
   /* snip */
   if (!ssSetNumInputPorts(S,2)) return;
       for (i = 0; i < 2; i++) {
           /* snip */
           ssSetInputPortDirectFeedThrough(S,i,1);
           ssSetInputPortRequiredContiguous(S,i,1);
       }
                /* snip */
   }
```

You can now use `ssGetInputPortRealSignal` in `mdlOutputs`:

```
void mdlOutputs(SimStruct* S, int_T tid) {
   int_T i;

   /* snip */

   for (i = 0; i < 2; i++) {
```

```
            int_T nu = ssGetInputPortWidth(S,i);
            const real_T* u  = ssGetInputPortRealSignal(S,i);
            UseInputVectorInSomeFunction(u, nu);
        }

        /* snip */

    }
```

See Also

ssSetInputPortRequiredContiguous, ssGetInputPortSignal, mdlInitializeSizes

ssGetInputPortRealSignalPtrs

Purpose

Get pointers to signals of type `double` connected to an input port.

Syntax

```
InputRealPtrsType ssGetInputPortRealSignalPtrs(SimStruct *S, int_T
   port)
```

Arguments

`S`
SimStruct representing an S-Function block.

`port`
Index of port whose signal is required.

Description

Returns pointers to the elements of a signal of type `double` connected to `port`. The input port index starts at 0 and ends at the number of input ports minus 1. This macro returns a pointer to an array of pointers to the `real_T` input signal elements. The length of the array of pointers is equal to the width of the input port.

Languages

C

Example

The following example reads all input port signals.

```
int_T i,j;
int_T nInputPorts = ssGetNumInputPorts(S);
for (i = 0; i < nInputPorts; i++) {
  InputRealPtrsType uPtrs =
                 ssGetInputPortRealSignal(S,i);
  int_T nu = ssGetInputPortWidth(S,i);
  for (j = 0; j < nu; j++) {
    SomeFunctionToUseInputSignalElement(*uPtrs
                        [j]);
  }
}
```

See Also

`ssGetInputPortWidth`, `ssGetInputPortDataType`, `ssGetInputPortSignalPtrs`

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

Syntax

```
int_T ssGetInputPortRequiredContiguous(SimStruct *S, int_T port)
```

Arguments

`S`
SimStruct representing an S-Function block or a Simulink model.

`port`
Index of an input port.

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

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

Languages C, C++

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

ssGetInputPortReusable

Purpose

Determine whether memory allocated to the input port is reusable.

Syntax

```
int_T ssGetInputPortReusable(SimStruct *S, int_T port)
```

Arguments

`S`
SimStruct representing an S-Function block or a Simulink model.

`port`
Index of the input port.

Description

Returns true if the input port memory buffer can be reused by other signals in the model.

Languages

C, C++

See Also

`ssSetInputPortReusable`

Purpose Get the sample time of an input port.

Syntax

```
ssGetInputPortSampleTime(SimStruct *S, inputPortIdx)
```

Arguments

`S`
SimStruct representing an S-Function block.

`inputPortIdx`
Index of port whose sample time is required.

Description Use in any routine (except `mdlInitializeSizes`) to determine the sample time of an input port. You should use this macro only if you have specified the sample times as port-based.

Languages C

See Also `ssSetInputPortSampleTime`, `ssGetInputPortOffsetTime`

ssGetInputPortSampleTimeIndex

Purpose

Get the sample time index of an input port.

Syntax

```
int_T ssGetInputPortSampleTimeIndex(SimStruct *S,
   int_T inputPortIdx)
```

Arguments

`S`
SimStruct representing an S-Function block or a Simulink model.

`inputPortIdx`
Index of the input port whose sample time index is being set.

Description

Returns the index of the sample time for the port.

Languages

C, C++

See Also

`ssSetInputPortSampleTimeIndex`

ssGetInputPortSignal

Purpose Get the address of a contiguous signal entering an input port.

Syntax

```
const void* ssGetInputPortSignal(SimStruct *S, inputPortIdx)
```

Arguments

`S`
SimStruct representing an S-Function block.

`inputPortIdx`
Index of port whose sample time is required.

Description Returns the address of the specified input port. A method should use this macro only if `mdlInitializeSizes` has specified that the elements of the input signal be contiguous, using `ssSetInputPortRequiredContiguous`.

Languages C, C++

Example The following code demonstrates the use of `ssGetInputPortSignal`.

```
nInputPorts = ssGetNumInputPorts(S);
    for (i = 0; i < nInputPorts; i++) {
         int_T nu = ssGetInputPortWidth(S,i);

    if ( ssGetInputPortRequiredContiguous(S,i) ) {

         const void *u = ssGetInputPortSignal(S,i);
         UseInputVectorInSomeFunction(u, nu);

         } else {

         InputPtrsType u  = ssGetInputPortSignalPtrs(S,i);
              for (j = 0; j < nu; j++) {
              UseInputInSomeFunction(*u[j]);
              }
         }
    }
```

If you know that the inputs are always `real_T` signals, the `ssGetInputPortSignal` line in the above code snippet would be

```
const real_T *u = ssGetInputPortRealSignal(S,i);
```

See Also `ssSetInputPortRequiredContiguous`, `ssGetInputPortRealSignal`

ssGetInputPortSignalAddress

Purpose

Get the address of an input port's signal.

Syntax

```
function ssGetInputPortSignalAddress(S : in SimStruct;
  port : in Integer := 0) return System.Address;
```

Arguments

`S`
SimStruct representing an S-Function block.

`port`
Index of an input port.

Description

Returns the address of the signal connected to `port`.

Languages

Ada

Example

The following code gets the signal connected to a block's input port.

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

See Also

`ssGetInputPortWidth`

ssGetInputPortSignalPtrs

Purpose

Get pointers to an input port's signal elements.

Syntax

```
InputPtrsType ssGetInputPortSignalPtrs(SimStruct *S, int_T port)
```

Arguments

`S`
SimStruct representing an S-Function block.

`port`
Index of an input port.

Description

Returns a pointer to an array of signal element pointers for the specified input port. For example, if the input port width is 5, this function returns a pointer to a 5-element pointer array. Each element in the pointer array points to the specific element of the input signal.

You must use `ssGetInputPortRealSignalPtrs` to get pointers to signals of type `double` (`real_T`).

Languages

C

Example

Assume that the input port data types are `int8_T`.

```
int_T nInputPorts = ssGetNumInputPorts(S);
for (i = 0; i < nInputPorts; i++) {
InputPtrsType      u     = ssGetInputPortSignalPtrs(S,i);
InputInt8PtrsType  uInt8 = (InputInt8PtrsType)u;
int_T              nu    = ssGetInputPortWidth(S,i);
for (j = 0; j < nu; j++) {
   /* u[j] is an int8_T pointer that points to the j-th element
       of the input signal.
     */
 UseInputInSomeFunction(*u[j]);
}
```

See Also

`ssGetInputPortRealSignalPtrs`

Purpose

Get the width of an input port.

C Syntax

```
int_T ssGetInputPortWidth(SimStruct *S, int_T port)
```

Ada Syntax

```
function  ssGetInputPortWidth(S : in SimStruct;
   port : in Integer := 0) return Integer;
```

Arguments

`S`
SimStruct representing an S-Function block.

`port`
Index of port whose width is required.

Description

Gets the input port number of elements. If the input port is a 1-D array with `w` elements, this function returns `w`. If the input port is an M-by-N matrix, this function returns `m*n`. If `m` or `n` is unknown, this function returns `DYNAMICALLY_SIZED`. Use in any routine (except `mdlInitializeSizes`) to determine the width of an input port.

Languages

Ada, C

See Also

`ssSetInputPortWidth`

Purpose Get a block's integer work vector.

Syntax

```
int_T* ssGetIWork(SimStruct *S)
```

Arguments

`S`
SimStruct representing an S-Function block.

Description Returns the integer work vector used by the block represented by `S`. The vector consists of elements of type `int_T` and is of length `ssGetNumIWork(S)`. Typically, this vector is initialized in `mdlStart` or `mdlInitializeConditions`, updated in `mdlUpdate`, and used in `mdlOutputs`. You can use this macro in the simulation loop, `mdlInitializeConditions`, or `mdlStart` routines.

Languages C

See Also `ssGetNumIWork`, `ssSetIWorkValue`, `ssGetIWorkValue`

ssGetIWorkValue

Purpose Get an element of a block's integer work vector.

Syntax

```
int_T ssGetIWorkValue(SimStruct *S, int_T idx)
```

Arguments

`S`
SimStruct representing an S-Function block.

`idx`
Index of the element returned by this function

Description Returns the `idx` element of the the integer vector used by the block represented by `S`. The vector consists of elements of type `int_T` and is of length `ssGetNumIWork(S)`. Typically, this vector is initialized in `mdlStart` or `mdlInitializeConditions`, updated in `mdlUpdate`, and used in `mdlOutputs`. You can use this macro in the simulation loop, `mdlInitializeConditions`, or `mdlStart` routines.

Example The following statement

```
int_T v = ssGetIWorkValue(s, 0);
```

is equivalent to

```
int_T* wv = ssGetIWork(s);
int_T v = wv[0];
```

Languages C

See Also `ssGetNumIWork`, `ssGetIWork`, `ssSetIWorkValue`

Purpose Get the model name.

Syntax

```
ssGetModelName(SimStruct *S)
```

Arguments

`S`
SimStruct representing an S-Function block or a Simulink model.

Description If `S` is a SimStruct for an S-Function block, this macro returns the name of the S-function MEX-file associated with the block. If `S` is the root `SimStruct`, this macro returns the name of the Simulink block diagram.

Languages C

See Also `ssGetPath`

ssGetModeVector

Purpose Get the mode vector.

Syntax

```
int_T *ssGetModeVector(SimStruct *S)
```

Arguments

`S`
SimStruct representing an S-Function block.

Description Returns a pointer (`int_T *`) to the mode vector.

This vector has length `ssGetNumModes(S)`. Typically, this vector is initialized in `mdlInitializeConditions` if the default value of 0 isn't acceptable. It is then used in `mdlOutputs` in conjunction with nonsampled zero crossings to determine when the output function should change mode. For example, consider an absolute value function. When the input is negative, negate it to create a positive value; otherwise, take no action. This function has two modes. The output function should be designed not to change modes during minor time steps. You can also use the mode vector in the `mdlZeroCrossings` routine to determine the current mode.

Languages C, C++

See Also `ssSetNumModes`

Purpose Get an element of a block's mode vector.

Syntax

```
int_T ssGetModeVectorValue(SimStruct *S, element)
```

Arguments

`S`
SimStruct representing an S-Function block.

`elementx`
Index of a mode vector element.

Description Returns the specified mode vector element.

Languages C, C++

See Also `ssSetModeVectorValue`, `ssGetModeVector`

ssGetNonsampledZCs

Purpose Get the zero-crossing signal values.

Syntax `ssGetNonsampledZCs(SimStruct *S)`

Arguments `S`
SimStruct representing an S-Function block.

Description Returns a pointer to the vector containing the current values of the signals that the variable-step solver monitors for zero crossings. The variable-step solver tracks the signs of these signals to bracket points where they cross zero. The solver then takes simulation time steps at the points where the zero crossings occur. This vector has length `ssGetNumNonsampledZCs(S)`.

Example The following excerpt from `matlabroot/simulink/src/sfun_zc.c` illustrates usage of this macro to update the zero-crossing array in the `mdlZeroCrossings` callback function.

```
static void mdlZeroCrossings(SimStruct *S)
{
       int_T i;
      real_T *zcSignals = ssGetNonsampledZCs(S);
      InputRealPtrsType uPtrs = ssGetInputPortRealSignalPtrs(S,0);
      int_T nZCSignals = ssGetNumNonsampledZCs(S);

      for (i = 0; i < nZCSignals; i++) {
          zcSignals[i] = *uPtrs[i];
      }
  }
```

Languages C

See Also `ssGetNumNonsampledZCs`

Purpose Get the number of continuous states that a block has.

C Syntax

```
int_T ssGetNumContStates(SimStruct *S)
```

Ada Syntax

```
function  ssGetNumContStates(S : in SimStruct) return Integer;
```

Arguments

`S`
SimStruct representing an S-Function block or model.

Description Returns the number of continuous states in the block or model represented by `S`. You can use this macro in any routine except `mdlInitializeSizes`.

Languages Ada, C

See Also `ssSetNumContStates`, `ssGetNumDiscStates`, `ssGetContStates`

ssGetNumDataTypes

Purpose

Get number of data types registered for this simulation, including built-in types.

Syntax

```
int_T ssGetNumDataTypes(SimStruct *S)
```

Arguments

`S`
SimStruct representing an S-Function block.

Description

Returns the number of data types registered for this simulation. This includes all custom data types registered by custom S-Function blocks and all built-in data types.

Note S-functions register their data types in their implementations of the `mdlInitializeSizes` callback function. Therefore, to ensure that this macro returns an accurate count, your S-function should invoke it only after the point in the simulation at which Simulink invokes the `mdlInitializeSizes` callback function.

Languages

C

See Also

`ssRegisterDataType`

Purpose Get the number of discrete states that a block has.

Syntax `int_T ssGetNumDiscStates(SimStruct *S)`

Arguments `S`
SimStruct representing an S-Function block.

Description Use in any routine (except `mdlInitializeSizes`) to determine the number of discrete states that the S-function has.

Languages C

See Also `ssSetNumDiscStates`, `ssGetNumContStates`

ssGetNumDWork

Purpose
Get the number of data type work vectors used by a block.

Syntax
```
int_T ssGetNumDWork(SimStruct *S)
```

Arguments
`S`
SimStruct representing an S-Function block.

Description
Returns the number of data type work vectors used by S.

Languages
C, C++

See Also
`ssSetNumDWork`

Purpose Get the number of input ports that a block has.

C Syntax

```
int_T ssGetNumInputPorts(SimStruct *S)
```

Ada Syntax

```
function  ssGetNumInputPorts(S : in SimStruct) return Integer;
```

Arguments

`S`
SimStruct representing an S-Function block.

Description Use in any routine (except `mdlInitializeSizes`) to determine how many input ports a block has.

Languages Ada, C

See Also `ssGetNumOutputPorts`

ssGetNumIWork

Purpose Get the size of a block's integer work vector.

Syntax `int_T ssGetNumIWork(SimStruct *S)`

Arguments `S`
SimStruct representing an S-Function block.

Description Returns the size of the integer (`int_T`) work vector used by the block represented by `S`. You can use this macro in any routine except `mdlInitializeSizes`.

Languages C

See Also `ssSetNumIWork`, `ssGetNumRWork`

Purpose Get the size of the mode vector.

Syntax `ssGetNumModes(SimStruct *S)`

Arguments `S`
SimStruct representing an S-Function block.

Description Returns the size of the modes vector. You can use this macro in any routine except `mdlInitializeSizes`.

Languages C

See Also `ssSetNumNonsampledZCs`, `ssGetNonsampledZCs`

ssGetNumNonsampledZCs

Purpose Get the size of the zero-crossing vector.

Syntax `ssGetNumNonsampledZCs(SimStruct *S)`

Arguments `S`
SimStruct representing an S-Function block.

Description Returns the size of the zero-crossing vector. You can use this macro in any routine except `mdlInitializeSizes`.

Languages C

See Also `ssSetNumNonsampledZCs`, `ssGetNonsampledZCs`

Purpose Get the number of output ports that a block has.

C Syntax

```
int_T ssGetNumOutputPorts(SimStruct *S)
```

Ada Syntax

```
function  ssGetNumOutputPorts(S : in SimStruct) return Integer;
```

Arguments

`S`
SimStruct representing an S-Function block.

Description Use in any routine (except `mdlInitializeSizes`) to determine how many output ports a block has.

Languages Ada, C

See Also `ssGetNumInputPorts`

ssGetNumParameters

Purpose Get the number of parameters that this block has.

Syntax

```
function ssGetNumParameters(S : in SimStruct) return Integer;
```

Arguments

`S`
SimStruct representing an S-Function block.

Description Returns the number of parameters that this block has.

Languages Ada

ssGetNumRunTimeParams

Purpose Get the number of run-time parameters created by this S-function.

Syntax

```
int_T ssGetNumRunTimeParams(SimStruct *S)
```

Arguments

`S`
SimStruct representing an S-Function block.

Description Use this function to get the number of run-time parameters created by this S-function.

Languages C

See Also `ssSetNumRunTimeParams`

ssGetNumPWork

Purpose

Get the size of a block's pointer work vector.

Syntax

```
int_T ssGetNumPWork(SimStruct *S)
```

Arguments

`S`
SimStruct representing an S-Function block.

Description

Returns the size of the pointer work vector used by the block represented by `S`. You can use this macro in any routine except `mdlInitializeSizes`.

Languages

C

See Also

`ssSetNumPWork`

Purpose Get the size of a block's floating-point work vector.

Syntax

```
int_T ssGetNumRWork(SimStruct *S)
```

Arguments

`S`
SimStruct representing an S-Function block.

Description Returns the size of the floating-point (`real_T`) work vector used by the block represented by `S`. You can use this macro in any routine except `mdlInitializeSizes`.

Languages C

See Also `ssSetNumRWork`

ssGetNumSampleTimes

Purpose Get the number of sample times that a block has.

Syntax

```
int_T ssGetNumSampleTimes(SimStruct *S)
```

Arguments

`S`
SimStruct representing an S-Function block.

Description Use in any routine (except `mdlInitializeSizes`) to determine the number of sample times `S` has.

Languages C

See Also `ssSetNumSampleTimes`

Purpose Get the number of parameters that an S-Function block expects.

Syntax

```
int_T ssGetNumSFcnParams(SimStruct *S)
```

Arguments

`S`
SimStruct representing an S-Function block.

Description Returns the number of parameters that `S` expects the user to enter.

Languages C

See Also `ssSetNumSFcnParams`

ssGetOutputPortBeingMerged

Purpose

Determine whether the output of this block is connected to a Merge block.

Syntax

```
int_T ssGetOutputPortBeingMerged(SimStruct *S, int_T port)
```

Arguments

`S`
SimStruct representing an S-Function block or a Simulink model.

`port`
Index of the output port.

Description

Returns true if this output port signal is being merged with other signals (this happens if the S-Function block's output port is connected to a Merge block directly or via connection type blocks). This macro returns the correct answer in and after the S-function's `mdlSetWorkWidths` method.

Languages

C, C++

See Also

`mdlSetWorkWidths`

Purpose Get the numeric type (complex or real) of an output port.

Syntax

```
CSignal_T ssGetOutputPortComplexSignal(SimStruct *S, int_T port)
```

Arguments

`S`
SimStruct representing an S-Function block.

`port`
Index of an output port.

Description Returns the numeric type of `port`: `COMPLEX_NO` (real signal), `COMPLEX_YES` (complex signal) or `COMPLEX_INHERITED` (dynamically determined).

Languages C

See Also `ssSetOutputPortComplexSignal`

ssGetOutputPortDataType

Purpose

Get the data type of an output port.

C Syntax

```
DTypeId ssGetOutputPortDataType(SimStruct *S, int_T port)
```

Ada Syntax

```
function  ssGetOutputPortDataType (S : in SimStruct;
   port : in Integer := 0) return Integer;
```

Arguments

`S`
SimStruct representing an S-Function block or a Simulink model.

`port`
Index of an output port.

Description

Returns the data type of the output port specified by `port`.

Languages

Ada, C

See Also

`ssSetOutputPortDataType`

Purpose

Get the dimensions of the signal leaving an output port.

Syntax

```
int_T *ssGetOutputPortDimensions(SimStruct *S, int_T port)
```

Arguments

`S`
SimStruct representing an S-Function block.

`port`
Index of an output port.

Description

Returns an array of integers that specifies the dimensions of the signal leaving `port`, e.g., `[4 2]` for a 4-by-2 matrix array. The size of the dimensions array is equal to the number of signal dimensions accepted by the port, e.g., 1 for a vector signal or 2 for a matrix signal.

Languages

C

See Also

`ssGetOutputPortNumDimensions`

ssGetOutputPortFrameData

Purpose Determine whether a port outputs signal frames.

Syntax

```
int_T ssGetOutputPortFrameData(SimStruct *S,  int_T port)
```

Arguments

`S`
SimStruct representing an S-Function block.

`port`
Index of an output port.

Description Returns one of the following:

- `-1`

  Port outputs either frame or unframed data.
- `0`

  Port outputs unframed data only.
- `1`

  Port outputs frame data only.

Languages C

See Also `ssSetOutputPortFrameData`

Purpose Get the number of dimensions of an output port.

Syntax

```
int_T ssGetOutputPortNumDimensions(SimStruct *S, int_T port)
```

Arguments

`S`
SimStruct representing an S-Function block.

`port`
Index of an output port.

Description Returns the number of dimensions of `port`.

Languages C

ssGetOutputPortOffsetTime

Purpose Get the offset time of an output port.

Syntax

```
real_T ssGetOutputPortOffsetTime(SimStruct *S,outputPortIdx)
```

Arguments

`S`
SimStruct representing an S-Function block.

`outputPortIdx`
Index of an output port.

Description Use in any routine (except `mdlInitializeSizes`) to determine the sample time of an output port. This macro should only be used if you have specified port-based sample times.

Languages C

See Also `ssSetOutputPortOffsetTime`, `ssGetOutputPortSampleTime`

Purpose

Get a pointer to an output signal of type `double` (`real_T`).

Syntax

```
real_T *ssGetOutputPortRealSignal(SimStruct *S, int_T port)
```

Arguments

`S`
SimStruct representing an S-Function block.

`port`
Index of an output port.

Description

Use in any simulation loop routine, `mdlInitializeConditions`, or `mdlStart` to access an output port signal where the output port index starts at 0 and must be less than the number of output ports. This returns a contiguous `real_T` vector of length equal to the width of the output port.

Example

To write to all output ports, you would use

```
int_T i,j;
int_T nOutputPorts = ssGetNumOutputPorts(S);
for (i = 0; i < nOutputPorts; i++) {
  real_T *y = ssGetOutputPortRealSignal(S,i);
  int_T  ny = ssGetOutputPortWidth(S,i);
  for (j = 0; j < ny; j++) {
    y[j] = SomeFunctionToFillInOutput();
  }
}
```

Languages

C

See Also

`ssGetInputPortRealSignalPtrs`

ssGetOutputPortReusable

Purpose

Determine whether memory allocated to the output port is reusable.

Syntax

```
int_T ssGetOutputPortReusable(SimStruct *S, int_T port)
```

Arguments

`S`
SimStruct representing an S-Function block or a Simulink model.

`port`
Index of an output port.

Description

Returns true if the output port memory buffer can be reused by other signals in the model.

Languages

C, C++

See Also

`ssSetOutputPortReusable`

Purpose Get the sample time of an output port.

Syntax

```
ssGetOutputPortSampleTime(SimStruct *S,outputPortIdx)
```

Arguments

`S`
SimStruct representing an S-Function block.

`outputPortIdx`
Index of an output port.

Description Use in any routine (except `mdlInitializeSizes`) to determine the sample time of an output port. This macro should only be used if you have specified port-based sample times.

Languages C

See Also `ssSetOutputPortSampleTime`

ssGetOutputPortSignal

Purpose Get the vector of signal elements emitted by an output port.

Syntax

```
void *ssGetOutputPortSignal(SimStruct *S, int_T port)
```

Arguments

`S`
SimStruct representing an S-Function block.

`port`
Index of an output port.

Description Returns a pointer to the vector of signal elements output by `port`.

Note If the port outputs a signal of type `double` (`real_T`), you must use `ssGetOutputPortRealSignal` to get the signal vector.

Example Assume that the output port data types are `int16_T`.

```
nOutputPorts = ssGetNumOutputPorts(S);
for (i = 0; i < nOutputPorts; i++) {
 int16_T *y      = (int16_T *)ssGetOutputPortSignal(S,i);
 int_T   ny      = ssGetOutputPortWidth(S,i);
 for (j = 0; j < ny; j++) {
     SomeFunctionToFillInOutput(y[j]);
 }
}
```

Languages C

See Also `ssGetOutputPortRealSignal`

Purpose Get address of an output port's signal.

Syntax

```
ssGetOutputPortSignalAddress(S : in SimStruct; port : in Integer :=
  0) return System.Address
```

Arguments

`S`
SimStruct representing an S-Function block.

`port`
Index of an output port.

Description Returns the address of the signal connected to `port`.

Languages Ada

Example The following code gets the signal connected to a block's input port.

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

See Also `ssGetOutputPortWidth`

ssGetOutputPortWidth

Purpose

Get the width of an output port.

C Syntax

```
int_T ssGetOutputPortWidth(SimStruct *S, int_T port)
```

Ada Syntax

```
function  ssGetOutputPortWidth(S : in SimStruct; port : in Integer
   := 0) return Integer;
```

Arguments

`S`
SimStruct representing an S-Function block.

`port`
Index of an output port.

Description

Use in any routine (except `mdlInitializeSizes`) to determine the width of an output port where the output port index starts at 0 and must be less than the number of output ports.

Languages

Ada, C

See Also

`ssSetOutputPortWidth`

Purpose

Get the parent of a `SimStruct`.

Syntax

```
SimStruct *ssGetParentSS(SimStruct *S)
```

Arguments

`S`
SimStruct representing an S-Function block or a Simulink model.

Description

Returns the parent `SimStruct` of `S`, or `NULL` if `S` is the root `SimStruct`.

Note There is one `SimStruct` for each S-function in your model and one for the model itself. The structures are arranged as a tree with the model `SimStruct` as the root. User-written S-functions should not use the `ssGetParentSS` macro.

Languages

C

See Also

`ssGetRootSS`

ssGetPath

Purpose

Get the path of a block.

C Syntax

```
const char_T *ssGetPath(SimStruct *S)
```

Ada Syntax

```
function  ssGetPath(S : in SimStruct) return String;
```

Arguments

`S`
SimStruct representing an S-Function block or a Simulink model.

Description

If `S` is an S-Function block, this macro returns the full Simulink path to the block. If `S` is the root SimStruct of the model, this macro returns the model name. In a C MEX S-function, in `mdlInitializeSizes`, if

```
strcmp(ssGetModelName(S),ssGetPath(S))==0
```

the S-function is being called from MATLAB and is not part of a simulation.

Languages

Ada, C

See Also

`ssGetModelName`

Purpose

Get the name of the placement group of a block.

Syntax

```
const char *ssGetPlacementGroup(SimStruct *S)
```

Arguments

`S`
SimStruct representing an S-Function block or a Simulink model. The block must be either a source block (i.e., a block without input ports) or a sink block (i.e., a block without output ports).

Description

Use this macro in `mdlInitializeSizes` to get the name of this block's placement group.

Note This macro is typically used to create Real-Time Workshop device driver blocks.

Languages

C

See Also

`ssSetPlacementGroup`

ssGetPortBasedSampleTimeBlockIsTriggered

Purpose Determine whether a block that uses port-based sample times resides in a triggered subsystem.

Syntax

```
boolean_T ssGetPortBasedSampleTimeBlockIsTriggered(SimStruct *S)
```

Arguments

`S`
SimStruct representing an S-Function block.

Description Returns `TRUE` if S uses port-based sample times and resides in a triggered subsystem. Use this macro in `mdlOutputs` and `mdlUpdate` to decode whether to use the block's triggered or non-triggered algorithms to compute its states and outputs.

Note This macro returns a valid result only after sample time propagation. Thus, you cannot use it in `mdlSetInputPortSampleTime` and `mdlSetOutputPortSampleTime` to determine whether a port's sample time is triggered. Use `ssSampleAndOffsetAreTriggered` instead.

Languages C

Purpose Get a block's pointer work vector.

Syntax

```
void** ssGetPWork(SimStruct *S)
```

Arguments

`S`
SimStruct representing an S-Function block.

Description Returns the pointer work vector used by the block represented by `S`. The vector consists of elements of type `void *` and is of length `ssGetNumPWork(S)`. Typically, this vector is initialized in `mdlStart` or `mdlInitializeConditions`, updated in `mdlUpdate`, and used in `mdlOutputs`. You can use this macro in the simulation loop, `mdlInitializeConditions`, or `mdlStart` routines.

Languages C

See Also `ssGetNumPWork`

ssGetPWorkValue

Purpose

Get a pointer from a block's pointer work vector.

Syntax

`void* ssGetPWorkValue(SimStruct *S, int_T idx)`

Arguments

`S`
SimStruct representing an S-function block.

`idx`
Index of the pointer returned by this function

Description

Returns the `idx` element of the the pointer work vector used by the block represented by `S`. The vector consists of elements of type `void *` and is of length `ssGetNumPWork(S)`. Typically, this vector is initialized in `mdlStart` or `mdlInitializeConditions`, updated in `mdlUpdate`, and used in `mdlOutputs`. You can use this macro in the simulation loop, `mdlInitializeConditions`, or `mdlStart` routines.

Example

The following statement

```
void* v = ssGetPWorkValue(s, 0);
```

is equivalent to

```
void** wv = ssGetPWork(s);
void* v = wv[0];
```

Languages

C

See Also

`ssGetNumPWork`, `ssGetPWork`, `ssSetPWorkValue`

Purpose Get a block's discrete state vector.

Syntax `real_T *ssGetRealDiscStates(SimStruct *S)`

Arguments

`S`
SimStruct representing an S-Function block.

Description Same as `ssGetDiscStates`.

Languages C

See Also `ssGetDiscStates`

ssGetRootSS

Purpose
Get the root of a SimStruct hierarchy.

Syntax
```
SimStruct *ssGetRootSS(SimStruct *S)
```

Arguments
`S`
SimStruct representing an S-Function block or a Simulink model.

Description
Returns the root of the `SimStruct` hierarchy containing `S`.

Languages
C

See Also
`ssGetParentSS`

ssGetRunTimeParamInfo

Purpose

Gets the attributes of a run-time parameter.

Syntax

```
ssParamRec *ssGetRunTimeParamInfo(SimStruct *S, int_T param)
```

Arguments

`S`
SimStruct representing an S-Function block.

`param`
Index of a run-time parameter.

Description

Returns the attributes of the run-time parameter specified by `param`. See the documentation for `ssSetRunTimeParamInfo` for a description of the `ssParamRec` structure returned by this function.

Languages

C

See Also

`ssSetRunTimeParamInfo`

ssGetRWork

Purpose Get a block's floating-point work vector.

Syntax `real_T* ssGetRWork(SimStruct *S)`

Arguments `S`
SimStruct representing an S-Function block.

Description Returns the floating-point work vector used by the block represented by `S`. The vector consists of elements of type `real_T` and is of length `ssGetNumRWork(S)`. Typically, this vector is initialized in `mdlStart` or `mdlInitializeConditions`, updated in `mdlUpdate`, and used in `mdlOutputs`. You can use this macro in the simulation loop, `mdlInitializeConditions`, or `mdlStart` routines.

Languages C

See Also `ssGetNumRWork`, `ssGetRWorkValue`, `ssSetRWorkValue`

Purpose Get an element of a block's floating-point work vector.

Syntax `real_T ssGetRWorkValue(SimStruct *S, int_T idx)`

Arguments

`S`
SimStruct representing an S-Function block.

`idx`
Index of the element returned by this function

Description Returns the `idx` element of the the floating-point work vector used by the block represented by `S`. The vector consists of elements of type `real_T` and is of length `ssGetNumRWork(S)`. Typically, this vector is initialized in `mdlStart` or `mdlInitializeConditions`, updated in `mdlUpdate`, and used in `mdlOutputs`. You can use this macro or `ssGetRWork` to get the current values of the work vector in the simulation loop, `mdlInitializeConditions`, or `mdlStart` routines.

Example The following statement

```
real_T v = ssGetRWorkValue(s, 0);
```

is equivalent to

```
real_T* wv = ssGetRWork(s);
real_T v = wv[0];
```

Languages C

See Also `ssGetNumRWork`, `ssGetRWork`, `ssSetRWorkValue`

ssGetSampleTimeOffset

Purpose Get the offset of the current sample time.

Syntax

```
function ssGetSampleTimeOffset(S : in SimStruct) return time_T;
```

Arguments

`S`
SimStruct representing an S-Function block.

Description Returns the offset of the current sample time.

Languages Ada

See Also `ssGetSampleTimePeriod`

Purpose Get the period of the current sample time.

Syntax

```
function ssGetSampleTimePeriod(S : in SimStruct) return time_T;
```

Arguments

`S`
SimStruct representing an S-Function block.

Description Returns the period of the current sample time.

Languages Ada

See Also `ssGetSampleTimeOffset`

ssGetSFcnParam

Purpose Get a parameter of an S-Function block.

Syntax

```
const mxArray *ssGetSFcnParam(SimStruct *S, int_T index)
```

Arguments

`S`
SimStruct representing an S-Function block.

`index`
Index of the parameter to be returned.

Description Use in any routine to access a parameter entered in the S-Function's block dialog box, where *`index`* starts at 0 and is less than `ssGetSFcnParamsCount(S)`.

Languages C

See Also `ssGetSFcnParamsCount`

Purpose

Get the number of block dialog parameters that an S-Function block has.

Syntax

```
int_T ssGetSFcnParamsCount(SimStruct *S)
```

Arguments

`S`
SimStruct representing an S-Function block.

Description

Returns the number of parameters that a user can set for the block represented by `S`.

Languages

C

See Also

`ssGetNumSFcnParams`

ssGetSimMode

Purpose Get the simulation mode an S-Function block.

Syntax ssGetSimMode(SimStruct *S)

Arguments

`S`
SimStruct representing an S-Function block or a Simulink model.

Description Returns the simulation mode of the block represented by `S`:

- `SS_SIMMODE_NORMAL`
  Running in a normal Simulink simulation
- `SS_SIMMODE_SIZES_CALL_ONLY`
  Invoked by editor to obtain number of ports
- `SS_SIMMODE_RTWGEN`
  Generating code
- `SS_SIMMODE_EXTERNAL`
  External mode simulation

Languages C

See Also `ssGetSolverName`

Purpose Get the solver mode being used to solve the S-function.

Syntax SolverMode ssGetSolverMode(SimStruct *S)

Arguments

`S`
SimStruct representing an S-Function block or a Simulink model.

Description Returns one of

- `SOLVER_MODE_AUTO`
- `SOLVER_MODE_SINGLETASKING`
- `SOLVER_MODE_MULTITASKING`

This macro can return `SOLVER_MODE_AUTO` in `mdlInitializeSizes`. However, in `mdlSetWorkWidths` and any methods called after `mdlSetWorkWidths`, solver mode is either `SOLVER_MODE_SINGLETASKING` or `SOLVER_MODE_MULTITASKING`.

Languages C, C++

See Also `ssGetSimMode`, `ssIsVariableStepSolver`

ssGetSolverName

Purpose Get the name of the solver being used to solve the S-function.

Syntax ssGetSolverName(SimStruct *S)

Arguments `S`
SimStruct representing an S-Function block or a Simulink model.

Description Returns a pointer (`char *`) to the name of the solver being used to solve the S-function represented by `S`.

Languages C

See Also `ssGetSimMode`, `ssIsVariableStepSolver`

ssGetStateAbsTol

Purpose Get the absolute tolerance used by the model's variable-step solver for a specified state.

Syntax

```
real_T ssGetStateAbsTol(SimStruct *S, int_T state)
```

Arguments

`S`
SimStruct representing an S-Function block.

Description Use in `mdlStart` to get the absolute tolerance for a particular state.

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

Languages C, C++

See Also `ssGetAbsTol`, `ssIsVariableStepSolver`

ssGetStopRequested

Purpose

Get the value of the simulation stop requested flag.

Syntax

```
int_T ssGetStopRequested(SimStruct *S)
```

Arguments

`S`
SimStruct representing an S-Function block or a Simulink model.

Description

Gets the value of the simulation stop requested flag. If the value is not `0`, Simulink halts the simulation at the end of the current time step.

Languages

C

See Also

`ssSetStopRequested`

Purpose Get the current simulation time.

C Syntax

```
ssGetT(SimStruct *S)
```

Ada Syntax

```
function  ssGetT(S : in SimStruct) return Real_T;
```

Arguments

`S`
SimStruct representing an S-Function block.

Description Returns the current base simulation time (`time_T`) for the model. You can use this macro in `mdlOutputs` and `mdlUpdate` to compute the output of your block.

Note Use this macro only if your block operates at the base rate of the model, for example, if your block operates at a single continuous rate. If your block operates at multiple rates or operates at a single rate that is different from the model's base, use `ssGetTaskTime` to get the correct time for the current task.

Languages Ada, C

See Also `ssGetTaskTime`, `ssGetTStart`, `ssGetTFinal`

ssGetTaskTime

Purpose Get the current time for the current task.

Syntax `ssGetTaskTime(SimStruct *S, st_index)`

Arguments

`S`
SimStruct representing an S-Function block.

`st_index`
Index of the sample time corresponding to the task for which the current time is to be returned.

Description Returns the current time (`time_T`) of the task corresponding to the sample rate specified by `st_index`. You can use this macro in `mdlOutputs` and `mdlUpdate` to compute the output of your block.

Languages C

See Also `ssGetT`

Purpose Get the simulation stop time.

C Syntax

```
time_T ssGetTFinal(SimStruct *S)
```

Ada Syntax

```
function ssGetTFinal(S : in SimStruct) return Real_T;
```

Arguments

`S`
SimStruct representing an S-Function block.

Description Returns the stop time of the current simulation.

Languages Ada, C

See Also `ssGetT`, `ssGetTStart`

ssGetTNext

Purpose Get the time of the next sample hit.

Syntax

```
time_T ssGetTNext(SimStruct *S)
```

Arguments

`S`
SimStruct representing an S-Function block.

Description Returns the next time that a sample hit occurs in a discrete S-function with a variable sample time.

Languages C

See Also `ssSetTNext`, `mdlGetTimeOfNextVarHit`

Purpose Get the simulation start time.

C Syntax

```
time_T ssGetTStart(SimStruct *S)
```

Ada Syntax

```
function  ssGetTStart(S : in SimStruct) return Real_T;
```

Arguments

`S`
SimStruct representing an S-Function block.

Description Returns the start time of the current simulation.

Languages Ada, C

See Also `ssGetT`, `ssGetTFinal`

ssGetUserData

Purpose Access user data.

Syntax

```
void ssGetUserData(SimStruct *S)
```

Arguments

`S`
SimStruct representing an S-Function block.

Description Retrieves a pointer to user data.

Languages C, C++

See Also `ssSetUserData`

Purpose Determine whether a task is continuous.

Syntax `ssIsContinuousTask(SimStruct *S,st_index,tid)`

Arguments

`S`
SimStruct representing an S-Function block.

`tid`
Task ID.

Description Use in `mdlOutputs` or `mdlUpdate` when your S-function has multiple sample times to determine whether your S-function is executing in the continuous task. You should not use this in single-rate S-functions, or if you did not register a continuous sample time.

Languages C

See Also `ssSetNumContStates`

ssIsFirstInitCond

Purpose

Determine whether this is the first call to `mdlInitializeConditions`.

Syntax

```
int_T ssIsFirstInitCond(SimStruct *S)
```

Arguments

`S`
SimStruct representing an S-Function block.

Description

Returns true if the current simulation time is equal to the simulation start time.

Languages

C

See Also

`mdlInitializeConditions`

Purpose Determine whether the simulation is in a major step.

C Syntax

```
int_T ssIsMajorTimeStep(SimStruct *S)
```

Ada Syntax

```
function ssIsMajorTimeStep(S : in SimStruct) return Boolean;
```

Arguments

`S`
SimStruct representing an S-Function block.

Description Returns 1 if the simulation is in a major time step.

Languages Ada, C

See Also `ssIsMinorTimeStep`

ssIsMinorTimeStep

Purpose Determine whether the simulation is in a minor step.

Syntax

```
int_T ssIsMinorTimeStep(SimStruct *S)
```

Arguments

`S`
SimStruct representing an S-Function block.

Description Returns 1 if the simulation is in a minor time step.

Languages C

See Also `ssIsMajorTimeStep`

Purpose

Determine whether the sample time is hit.

Syntax

```
ssIsSampleHit(SimStruct *S,st_index,tid)
```

Arguments

`S`
SimStruct representing an S-Function block.

`st_index`
Index of the sample time.

`tid`
Task ID.

Description

Use in `mdlOutputs` or `mdlUpdate` when your S-function has multiple sample times to determine the task your S-function is executing in. You should not use this in single-rate S-functions or for an *`st_index`* corresponding to a continuous task.

Languages

C

See Also

`ssIsContinuousTask`, `ssIsSpecialSampleHit`

ssIsSpecialSampleHit

Purpose Determine whether sample is hit.

Syntax `ssIsSpecialSampleHit(SimStruct *S, sti1, sti2, tid)`

Arguments

`S`
SimStruct representing an S-Function block.

`sti1`
Index of the sample time.

`sti2`
Index of the sample time.

`tid`
Task ID.

Description Returns true if a sample hit has occurred at `sti1` and a sample hit has also occurred at `sti2` in the same time step. You can use this macro in `mdlUpdate` and `mdlOutputs` to ensure the validity of data shared by multiple tasks running at different rates. For more information, see “Synchronizing Multirate S-Function Blocks” on page 7-24.

Languages C

See Also `ssIsSampleHit`

Purpose Get the name of the solver being used to solve the S-function.

Syntax

```
ssIsVariableStepSolver(SimStruct *S)
```

Arguments

`S`
SimStruct representing an S-Function block or a Simulink model.

Description Returns 1 if the solver being used to solve `S` is a variable-step solver. This is useful when you are creating S-functions that have zero crossings and an inherited sample time.

Languages C

See Also `ssGetSimMode`, `ssGetSolverName`

ssPrintf

Purpose Print a variable-content message.

Syntax

```
ssPrintf(msg, ...)
```

Arguments

`msg`
Message. Must be a string with optional variable replacement parameters.

`...`
Optional replacement arguments.

Description Prints variable-content `msg`. This macro expands to `mexPrintf` when the S-function is compiled via `mex` for use with Simulink. When the S-function is compiled for use with the Real-Time Workshop, this macro expands to `printf` if the target has `stdio` facilities; otherwise, it becomes a call to an empty function (`rtPrintfNoOp`). In the case of Real-Time Workshop, you can avoid a call altogether, using the `SS_STDIO_AVAILABLE` macro. For example:

```
#if defined(SS_STDIO_AVAILABLE)
   ssPrintf("my message ...");
#endif
```

Languages C

See Also `ssWarning`

ssRegDlgParamAsRunTimeParam

Purpose Register a dialog parameter as a run-time parameter.

Syntax

```
ssRegDlgParamAsRunTimeParam(S, dlgIdx, rtIdx, name, dtId)
```

Arguments

`S`
SimStruct representing an S-Function block or a Simulink model.

`dlgIdx`
Index of the dialog parameter

`rtIdx`
Index of the run-time parameter

`name`
Name of the parameter

`dtId`
Value of type `DTypeId` that specifies the data type of the run-time parameter

Description Use this function in `mdlSetWorkWidths` to register the dialog parameter specified by `dlgIdx` as a run-time parameter specified by `rtIdx` and having the name and data type specified by `name` and `dtId`, respectively. This function also initializes the run-time parameter to the initial value of the dialog parameter, converting the value to the specified data type if necessary.

Note The first four characters of block's run-time parameter names 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.

Languages C

See Also `ssRegAllTunableParamsAsRunTimeParams`

ssRegAllTunableParamsAsRunTimeParams

Purpose

Register all tunable parameters as run-time parameters.

Syntax

```
void ssRegAllTunableParamsAsRunTimeParams(S,
   const char_T *names[]))
```

Arguments

`S`
SimStruct representing an S-Function block.

`names`
Array of names for the run-time parameters.

Note The first four characters of block's run-time parameter names 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.

Description

Use this function in `mdlSetWorkWidths` to register all tunable dialog parameters as run-time parameters. Specify the names of the run-time versions of the parameters in the `names` array.

Note Simulink assumes that the `names` array is always available. Therefore, you must allocate the `names` array in such a way that it persists throughout the simulation.

You can register dialog parameters individually as run-time parameters, using `ssSetNumRunTimeParams` and `ssSetRunTimeParamInfo`.

Languages

C

See Also

`mdlSetWorkWidths`, `ssSetNumRunTimeParams`, `ssSetRunTimeParamInfo`

Purpose

Register a custom data type.

Syntax

```
DTypeId ssRegisterDataType(SimStruct *S, char *name)
```

Arguments

`S`
SimStruct representing an S-Function block.

`name`
Name of custom data type.

Description

Register a custom data type. Each data type must be a valid MATLAB identifier. That is, the first char is an alpha and all subsequent characters are alphanumeric or "_". The `name` length must be less than 32. Data types must be registered in `mdlInitializeSizes`.

If the registration is successful, the function returns the `DataTypeId` associated with the registered data type; otherwise, it reports an error and returns `INVALID_DTYPE_ID`.

After registering the data type, you must specify its size, using `ssSetDataTypeSize`.

Note You can call this function to get the data type ID associated with a registered data type.

Example

The following example registers a custom data type named `Color`.

```
DTypeId id = ssRegisterDataType(S, "Color");
if(id == INVALID_DTYPE_ID) return;
```

Languages

C

See Also

`ssSetDataTypeSize`

ssSampleAndOffsetAreTriggered

Purpose

Determine whether a sample time and offset value pair indicate a triggered sample time.

Syntax

```
boolean_T ssSampleAndOffsetAreTriggered(real_T st, real_T ot)
```

Arguments

`st`
The sample time

`ot`
The offset time

Description

Returns TRUE if both `st` and `ot` are equal to `INHERITED_SAMPLE_TIME`. Simulink sets the sample time and offset pairs of a block or its ports (for port-based sample times) to `INHERITED_SAMPLE_TIME` if the block resides in a triggered subsystem. By invoking this macro on its sample time/offset pairs, an S-function can determine whether it resides in a triggered subsystem.

Languages

C

Purpose

Request that Simulink attempt to reduce a block.

Syntax

```
ssSetBlockReduction(SimStruct *S, unsigned int_T flag)
```

Arguments

`S`
SimStruct representing an S-Function block.

`flag`
If true, Simulink should attempt to reduce this block.

Description

Use this macro to ask Simulink to reduce this block. A block is reducible if it can be eliminated from the model without affecting the model's behavior. Simulink optimizes performance by skipping execution of reducible blocks during model simulation. In particular, Simulink does not invoke the `mdlStart`, `mdlUpdate`, and `mdlOutputs` methods of reducible blocks. Further, Simulink executes the `mdlTerminate` method of a reduced block only if the block has set the `SS_OPTION_CALL_TERMINATE_AT_EXIT` option before the simulation loop has begun, using `ssSetOptions`.

A block must meet certain criteria to be considered reducible. For example, a block must have at least one input, must have the same number of outputs as inputs or no outputs, and none of the block's inputs can be a bus signal. If a block fails to meet any of these criteria, Simulink includes the block in the simulation regardless of whether the block has requested reduction.

Your S-function must invoke this macro before Simulink would otherwise invoke the S-function's `mdlStart` method (see the callback flow diagram in "How Simulink Interacts with C S-Functions" on page 3-35). This means your S-function must invoke this macro no later than its `mdlSetWorkWidths` method to be considered a candidate for block reduction.

Languages

C

See Also

`ssGetBlockReduction`

ssSetCallSystemOutput

Purpose Specify that an output port is issuing a function call.

Syntax

```
ssSetCallSystemOutput(SimStruct *S, port_index)
```

Arguments

`S`
SimStruct representing an S-Function block or a Simulink model.

`port_index`
Index of the port that is issuing the function call.

Description Use in `mdlInitializeSampleTimes` to specify that the output port element specified by *`port_index`* is issuing a function call by using `ssCallSystemWithTid(S,`*`index,tid`*`)`. The *`index`* specified starts at 0 and must be less than `ssGetOutputPortWidth(S,0)`.

Languages C

See Also `ssCallSystemWithTid`

Purpose

Set the size of a custom data type.

Syntax

```
int_T ssSetDataTypeSize(SimStruct *S, DTypeId id, int_T size)
```

Arguments

`S`
SimStruct representing an S-Function block.

`id`
ID of data type.

`size`
Size of the custom data type in bytes.

Description

Sets the size of the data type specified by `id` to `size`. If the call is successful, the macro returns 1 (true), otherwise, it returns 0 (false). Use this macro in `mdlInitializeSizes` to set the size of a data type you have registered.

Example

The following example registers and sets the size of the custom data type named `Color` to 4 bytes.

```
int_T   status;
DTypeId id;

id = ssRegisterDataType(SimStruct *S, "Color");
if(id == INVALID_DTYPE_ID) return;

status = ssSetDataTypeSize(S, id, 4);
if(status == 0) return;
```

Languages

C

See Also

`ssRegisterDataType`, `ssGetDataTypeSize`

ssSetDataTypeZero

Purpose Set zero representation of a data type.

Syntax `int_T ssSetDataTypeZero(SimStruct *S, DTypeId id, void* zero)`

Arguments

`S`
SimStruct representing an S-Function block.

`id`
ID of data type.

`zero`
Zero representation of the data type specified by `id`.

Description Sets the zero representation of the data type specified by `id` to `0` and returns `1` (true) if `id` is valid, the size of the data type has been set, and the zero representation has not already been set. Otherwise, this macro returns `0` (false) and reports an error. Because this macro reports any error that occurs, you do not need to use `ssSetErrorStatus` to report the error.

Note This macro makes a copy of the zero representation of the data type for Simulink's use. Thus, your S-function does not have to maintain the original in memory.

Languages C

Example The following example registers and sets the size and zero representation of a custom data type named `myDataType`.

```
typedef struct{
   int8_T   a;
   uint16_T b;
}myStruct;

int_T    status;
DTypeId  id;
myStruct tmp;

id = ssRegisterDataType(S, "myDataType");
```

```
if(id == INVALID_DTYPE_ID) return;

status = ssSetDataTypeSize(S, id, sizeof(tmp));
if(status == 0) return;

tmp.a = 0;
tmp.b = 1;
status = ssSetDataTypeZero(S, id, &tmp);
if(status == 0) return;
```

See Also

`ssRegisterDataType`, `ssSetDataTypeSize`, `ssGetDataTypeZero`

ssSetDWorkComplexSignal

Purpose Specify whether the elements of a data type work vector are real or complex.

Syntax

```
void ssSetDWorkComplexSignal(SimStruct *S, int_T vector,
   CSignal_T numType)
```

Arguments

`S`
SimStruct representing an S-Function block.

`vector`
Index of a data type work vector, where the index is one of `0`, `1`, `2`, `...` `ssGetNumDWork(S)`.

`numType`
Numeric type, either `COMPLEX_YES` or `COMPLEX_NO`.

Description Use in `mdlInitializeSizes` or `mdlSetWorkWidths` to specify whether the values of the specified work vector are complex numbers (`COMPLEX_YES`) or real numbers (`COMPLEX_NO`, the default).

Languages C, C++

See Also `ssSetDWorkDataType`, `ssGetNumDWork`

Purpose Specify the data type of a data type work vector.

Syntax

```
void ssSetDWorkDataType(SimStruct *S, int_T vector, DTypeId dtID)
```

Arguments

`S`
SimStruct representing an S-Function block.

`vector`
Index of a data type work vector, where the index is one of `0`, `1`, `2`, `...` `ssGetNumDWork(S)`.

`dtID`
ID of a data type.

Description Use in `mdlInitializeSizes` or `mdlSetWorkWidths` to set the data type of the specified work vector.

Languages C, C++

See Also `ssSetDWorkWidth`, `ssGetNumDWork`

ssSetDWorkName

Purpose Specify the name of a data type work vector.

Syntax

```
void ssSetDWorkName(SimStruct *S, int_T vector, char_T *name)
```

Arguments

`S`
SimStruct representing an S-Function block.

`vector`
Index of the work vector, where the index is one of `0`, `1`, `2`, `...` `ssGetNumDWork(S)`.

`name`
Name of a work vector.

Description Use in `mdlInitializeSizes` or in `mdlSetWorkWidths` to specify a name for the specified data type work vector. The Real-Time Workshop uses this name to label the work vector in generated code. If you do not specify a name, the Real-Time Workshop generates a name for the work vector.

Languages C, C++

See Also `ssGetDWorkName`, `ssSetNumDWork`

Purpose Specify the identifier used to declare a DWork vector in code generated from the associated S-function.

Syntax

```
void ssSetDWorkRTWIdentifier(SimStruct* S, int idx, char_T * id)
```

Arguments

`S`
SimStruct representing an S-Function block.

`idx`
Index of the work vector, where the index is one of `0`, `1`, `2`, `...` `ssGetNumDWork(S)`.

`id`
Identifier

Description Specifies `id` as the identifier used in code generated by the Real-Time Workshop to declare the DWork vector specified by `idx`.

Languages C, C++

See Also `ssSetDWorkRTWIdentifier`

ssSetDWorkRTWStorageClass

Purpose

Specify the storage class of a DWork vector in code generated from the associated S-function.

Syntax

`void ssSetDWorkRTWStorageClass(SimStruct* S, int idx,` ssRTWStorageType sc)

Arguments

`S`
SimStruct representing an S-Function block.

`idx`
Index of the work vector, where the index is one of `0`, `1`, `2`, `...` `ssGetNumDWork(S)`.

`sc`
Storage class of the work vector. Must be one of the values enumerated by `ssRTWStorageType` in `simstruc.h`:

```
typedef enum {
    SS_RTW_STORAGE_AUTO = 0,
    SS_RTW_STORAGE_EXPORTED_GLOBAL,
    SS_RTW_STORAGE_IMPORTED_EXTERN,
    SS_RTW_STORAGE_IMPORTED_EXTERN_POINTER
} ssRTWStorageType
```

Description

Sets `sc` as the storage class of the the DWork vector specified by `idx`. The storage class is a code-generation attribute that determines how the code generated by the Real-Time Workshop for this S-function allocates memory for this work vector (see "Signal Storage Concepts" in the online documentation for the Real-Time Workshop).

Languages

C, C++

See Also

`ssGetDWorkRTWStorageClass`

Purpose Specify the C type qualifier (e.g., `const`) used to declare a DWork vector in code generated from the associated S-function.

Syntax

```
void ssSetDWorkRTWTypeQualifier(SimStruct* S, int idx, char_T * tq)
```

Arguments

`S`
SimStruct representing an S-Function block.

`idx`
Index of the work vector, where the index is one of `0`, `1`, `2`, `...` `ssGetNumDWork(S)`.

`tq`
type qualifier

Description Sets `tq` as the C type qualifier (e.g., `const`) used to declare the DWork vector specified by `idx` in code generated by the Real-Time Workshop from the associated S-function.

Languages C, C++

See Also `ssGetDWorkRTWTypeQualifier`

ssSetDWorkUsedAsDState

Purpose Specify that a data type work vector is used as a discrete state vector.

Syntax

```
void ssSetDWorkUsedAsDState(SimStruct *S, int_T vector,
   int_T usage)
```

Arguments

`S`
SimStruct representing an S-Function block.

`vector`
Index of a data type work vector, where the index is one of `0`, `1`, `2`, `...` `ssGetNumDWork(S)`.

`usage`
How this vector is used.

Description Use in `mdlInitializeSizes` or `mdlSetWorkWidths` to specify the usage of the specified work vector, either `SS_DWORK_USED_AS_DSTATE` (used to store the block's discrete states) or `SS_DWORK_USED_AS_DWORK` (used as a work vector, the default).

Note Specify the usage as `SS_DWORK_USED_AS_DSTATE` if the following conditions are true. You want to use the vector to store discrete states and you want Simulink to log the discrete states to the workspace at the end of a simulation, if the user has selected the **Save to Workspace** option on Simulink's **Simulation Parameters** dialog.

Languages C, C++

See Also `ssGetDWorkUsedAsDState`

Purpose

Specify the width of a data type work vector.

Syntax

```
void ssSetDWorkWidth(SimStruct *S, int_T vector, int_T width)
```

Arguments

`S`
SimStruct representing an S-Function block.

`vector`
Index of the work vector, where the index is one of `0`, `1`, `2`, `...` `ssGetNumDWork(S)`.

`width`
Number of elements in the work vector.

Description

Use in `mdlInitializeSizes` or in `mdlSetWorkWidths` to set the number of elements in the specified data type work vector.

Languages

C, C++

See Also

`ssGetDWorkWidth`, `ssSetDWorkDataType`, `ssSetNumDWork`

ssSetErrorStatus

Purpose

Report an error.

C Syntax

```
void ssSetErrorStatus(SimStruct *S, const char_T *msg)
```

Ada Syntax

```
procedure ssSetErrorStatus(S : in SimStruct; msg : in String);
```

Arguments

`S`
SimStruct representing an S-Function block or a Simulink model.

`msg`
Error message.

Description

Use this function to report errors that occur in your S-function. For example:

```
ssSetErrorStatus(S, "error message");
return;
```

Note The error message string must be in persistent memory; it cannot be a local variable.

Languages

Ada, C

See Also

`ssWarning`

Purpose Specify the external mode function for an S-function.

Syntax

```
void ssSetExternalModeFcn(SimStruct *S, SFunExtModeFcn *fcn)
```

Arguments

`S`
SimStruct representing an S-Function block or a Simulink model.

`fcn`
External mode function.

Description Specifies the external mode function for `S`.

Languages C

See Also `ssCallExternalModeFcn`

ssSetInputPortComplexSignal

Purpose

Set the numeric type (real or complex) of an input port.

Syntax

```
void ssSetInputPortComplexSignal(SimStruct *S, input_T port,
  CSignal_T csig)
```

Arguments

`S`
SimStruct representing an S-Function block or a Simulink model.

`port`
Index of an input port.

`csignal`
Numeric type of the signals accepted by `port`. Valid values are `COMPLEX_NO` (real signal), `COMPLEX_YES` (complex signal), and `COMPLEX_INHERITED` (numeric type inherited from driving block).

Description

Use this function in `mdlInitializeSizes` to initialize input port signal type. If the numeric type of the input port is inherited from the block to which it is connected, set the numeric type to `COMPLEX_INHERITED`. The default numeric type of an input port is `real`.

Languages

C

Example

Assume that an S-function has three input ports. The first input port accepts real (noncomplex) signals. The second input port accepts complex signals. The third port accepts signals of either type. The following example specifies the correct numeric type for each port.

```
ssSetInputPortComplexSignal(S, 0, COMPLEX_NO)
ssSetInputPortComplexSignal(S, 1, COMPLEX_YES)
ssSetInputPortComplexSignal(S, 2, COMPLEX_INHERITED)
```

See Also

`ssGetInputPortComplexSignal`

Purpose

Set the data type of an input port.

C Syntax

```
void ssSetInputPortDataType(SimStruct *S,input_T port, DTypeId id)
```

Ada Syntax

```
procedure ssSetInputPortDataType(S : in SimStruct;
   port : in Integer := 0; id : in Integer);
```

Arguments

`S`
SimStruct representing an S-Function block or a Simulink model.

`port`
Index of an input port.

`id`
ID of the data type accepted by `port`.

Description

Use this function in `mdlInitializeSizes` to set the data type of the input port specified by `port`. If the input port's data type is inherited from the block connected to the port, set the data type to `DYNAMICALLY_TYPED`.

Note The data type of an input port is `double` (`real_T`) by default.

Languages

Ada, C

Example

Suppose that you want to create an S-function with two input ports, the first of which inherits its data type from the driving block and the second of which accepts inputs of type `int8_T`. The following code sets up the data types.

```
ssSetInputPortDataType(S, 0, DYNAMICALLY_TYPED)
ssSetInputPortDataType(S, 1, SS_INT8)
```

See Also

`ssGetInputPortDataType`

ssSetInputPortDimensionInfo

Purpose Specify information about the dimensionality of an input port.

Syntax

```
void ssSetInputPortDimensionInfo(SimStruct *S,  int_T port,
  DimsInfo_T *dimsInfo)
```

Arguments

`S`
SimStruct representing an S-function block.

`port`
Index of an input port

`dimsInfo`
Structure of type DimsInfo_T that specifies the dimensionality of the signals accepted by `port`.

The structure is defined as

```
typedef struct DimsInfo_tag{
    int  width;/* number of elements   */
    int  numDims/* Number of dimensions */
    int  *dims;/* Dimensions.          */
    [snip]
}DimsInfo_T;
```

where:

- `numDims` specifies the number of dimensions of the signal, e.g., 1 for a 1-D (vector) signal or 2 for a 2-D (matrix) signal, or `DYNAMICALLY_SIZED` if the number of dimensions is determined dynamically
- `dims` is an integer array that specifies the size of each dimension, e.g., `[2 3]` for a 2-by-3 matrix signal, or `DYNAMICALLY_SIZED for each dimension that is determined dynamically, e.g., [2 DYNAMICALL_SIZED]`
- `width` equals the total number of elements in the signal, e.g., 12 for a 3-by-4 matrix signal or 8 for an 8-element vector signal, or `DYNAMICALLY_SIZED` if the total number of elements is determined dynamically

---

Note Use the macro, `DECL_AND_INIT_DIMSINFO`, to declare and initialize an instance of this structure.

Description

Specifies the dimension information for `port`. Use this function in `mdlInitializeSizes` to initialize the input port dimension information. If you want the port to inherit its dimensions from the port to which it is connected, specify `DYNAMIC_DIMENSION` as the `dimsInfo` for `port`.

Languages

C

Example

The following example specifies that input port 0 accepts 2-by-2 matrix signals.

```
{
    DECL_AND_INIT_DIMSINFO(di);
    int dims[2];

    di.numDims = 2;
    dims[0] = 2;
    dims[1] = 2;
    di.dims = &dims;
    di.width = 4;
    ssSetInputPortDimensionInfo(S, 0, &di);
}
```

See Also

`ssSetInputPortMatrixDimensions`, `ssSetInputPortVectorDimension`

ssSetInputPortDirectFeedThrough

Purpose

Specify the direct feedthrough status of a block's ports.

C Syntax

```
void ssSetInputPortDirectFeedThrough(SimStruct *S, int_T port,
   int_T dirFeed)
```

Ada Syntax

```
procedure ssSetInputPortDirectFeedThrough(S : in SimStruct; port :
   in Integer := 0; dirFeed : in Boolean);
```

Arguments

`S`
SimStruct representing an S-Function block or a Simulink model.

`port`
Index of the input port whose direct feedthrough property is being set.

`dirFeed`
Direct feedthrough status of block specified by `port`.

Description

Use in `mdlInitializeSizes` (after `ssSetNumInputPorts`) to specify the direct feedthrough (0 or 1) for each input port index. If not specified, the default direct feedthrough is 0. Setting direct feedthrough to 0 for an input port is equivalent to saying that the corresponding input port signal is not used in `mdlOutputs` or `mdlGetTimeOfNextVarHit`. If it is used, you might or might not see a delay of one simulation step in the input signal. This might cause the simulation solver to issue an error due to simulation inconsistencies.

Languages

Ada, C

See Also

`ssGetInputPortDirectFeedThrough`

Purpose

Specify whether a port accepts signal frames.

Syntax

```
void ssSetInputPortFrameData(SimStruct *S,  int_T port,
   int_T acceptsFrames)
```

Arguments

`S`
SimStruct representing an S-Function block.

`port`
Index of an input port.

`acceptsFrames`
Type of signal accepted by `port`. Acceptable values are `-1` (either frame or unframed input), `0` (unframed input only), and `1` (framed input only).

Description

Use in `mdlSetInputPortFrameData` to specify whether a port accepts frame data only, unframed data only, or both.

Languages

C

See Also

`ssGetInputPortFrameData`, `mdlSetInputPortFrameData`

ssSetInputPortMatrixDimensions

Purpose

Specify dimension information for an input port that accepts matrix signals.

Syntax

```
int_T ssSetInputPortMatrixDimensions(SimStruct *S,  int_T port,
   int_T m, int_T n)
```

Arguments

`S`
SimStruct representing an S-Function block.

`port`
Index of an input port.

`m`
Row dimension of matrix signals accepted by `port` or `DYNAMICALLY_SIZED`.

`n`
Column dimension of matrix signals accepted by `port` or `DYNAMICALLY_SIZED`.

Description

Specifies that `port` accepts an `m`-by-`n` matrix signal. If either dimension is `DYNAMICALLY_SIZED`, the other must be `DYNAMICALLY_SIZED` or 1. Returns `1` if successful; otherwise, `0`.

Languages

C

Example

The following example specifies that input port 0 accepts 2-by-2 matrix signals.

```
ssSetInputPortMatrixDimensions(S,  0, 2, 2);
```

Purpose Specify the offset time of an input port.

Syntax

```
void ssSetInputPortOffsetTime(SimStruct *S,
   int_T inputPortIdx, int_T period)
```

Arguments

`S`
SimStruct representing an S-Function block or a Simulink model.

`inputPortIdx`
Index of the input port whose offset time is being set.

`offset`
Offset time.

Description Use in `mdlInitializeSizes` (after `ssSetNumInputPorts`) to specify the sample time offset for each input port index. You can use this macro in conjunction with `ssSetInputPortSampleTime` if you have specified port-based sample times for your S-function.

Languages C

See Also `ssSetNumInputPorts`, `ssSetInputPortSampleTime`

ssSetInputPortOverWritable

Purpose

Specify whether an input port can be overwritten.

C Syntax

```
void ssSetInputPortOverWritable(SimStruct *S, int_T port, int_T
   isOverwritable)
```

Ada Syntax

```
procedure ssSetInputPortOverWritable(S : in SimStruct; port : in
   Integer := 0; isOverwritable : in Boolean);
```

Arguments

`S`
SimStruct representing an S-Function block or a Simulink model.

`port`
Index of the input port whose overwritability is being set.

`isOverwritable`
Value specifying whether `port` is overwritable.

Description

Use in `mdlInitializeSizes` (after `ssSetNumInputPorts`) to specify whether the input port is overwritable by an output port. The default is `isOverwritable=0`, which means that the input port does not share memory with an output port. When `isOverwritable=1`, the input port shares memory with an output port.

Note `ssSetInputPortReusable` and `ssSetOutputPortReusable` must both be set to 1, meaning that neither port involved can have global and persistent memory.

Languages

Ada, C

See Also

`ssSetNumInputPorts`, `ssSetInputPortReusable`, `ssSetOutputPortReusable`, `ssGetInputPortBufferDstPort`

Purpose

Specify that the signal elements entering a port must be contiguous.

Syntax

```
void ssSetInputPortRequiredContiguous(SimStruct *S, int_T port,
   int_T flag)
```

Arguments

`S`
SimStruct representing an S-Function block or a Simulink model.

`port`
Index of an input port.

`flag`
True if signal elements must be contiguous.

Description

Specifies that the signal elements entering the specified port must occupy contiguous areas of memory. This allows a method to access the elements of the signal simply by incrementing the signal pointer returned by `ssGetInputPortSignal`. The S-function can set the value of this attribute as early as in the `mdlInitializeSizes` method and at the latest in the `mdlSetWorkWidths` method.

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

Languages

C, C++

See Also

`mdlInitializeSizes`, `mdlSetWorkWidths`, `ssGetInputPortSignal`, `ssGetInputPortSignalPtrs`

ssSetInputPortReusable

Purpose Specify whether where memory allocated to port is reusable.

Syntax

```
void ssSetInputPortReusable(SimStruct *S, int_T port, int_T
   isReusable)
```

Arguments

`S`
SimStruct representing an S-Function block or a Simulink model.

`port`
Index of the input port whose reusability is being set.

`isReusable`
Value specifying whether `port` is reusable.

Description Use in `mdlInitializeSizes` (after `ssSetNumInputPorts`) to specify whether the input port memory buffer can be reused by other signals in the model. This macro can take one of two values:

- `Off (isReusable=0)` specifies that the input port is not reusable. This is the default.
- `On (isReusable=1)` specifies that the input port is reusable.

In Simulink, reusable signals share the same memory space. When this macro is turned on, the input port signal to the S-function can be reused by other signals in the model. This reuse results in less memory use during Simulink simulation and more efficiency in the Real-Time Workshop generated code.

You must use caution when using this macro; you can safely turn it on only if the S-function reads its input port signal in its `mdlOutputs` routine and does not access this input port signal until the next call to `mdlOutputs`.

When an S-function's input port signal is reused, other signals in the model overwrite it prior to the execution of `mdlUpdate`, `mdlDerivatives`, or other run-time S-function routines. For example, if the S-function reads the input port signal in its `mdlUpdate` routine, or reads the input port signal in the `mdlOutputs` routine and expects this value to be persistent until the execution of its `mdlUpdate` routine, turning this attribute on is incorrect and leads to erroneous results.

The default setting, `off`, is safe. It prevents any reuse of the S-function input port signals, which means that the inport port signals have the same values in any run-time S-function routine during a single execution of the simulation loop.

Note that this is a suggestion and not a requirement for the Simulink engine. If Simulink cannot resolve buffer reuse in local memory, it resets `value=0` and places the input port signals into global memory.

Languages

C

See Also

`ssSetNumInputPorts`, `ssSetInputPortOverwritable`, `ssSetOutputPortReusable`

ssSetInputPortSampleTime

Purpose

Specify the sample time of an input port.

Syntax

```
ssSetInputPortSampleTime(SimStruct *S,inputPortIdx,period)
```

Arguments

`S`
SimStruct representing an S-Function block or a Simulink model.

`inputPortIdx`
Index of the input port whose sample time is being set.

`period`
Sample period.

Description

Use in `mdlInitializeSizes` (after `ssSetNumInputPorts`) to specify the sample time period as continuous or as a discrete value for each input port. Input port index numbers start at 0 and end at the total number of input ports minus 1. You should use this macro only if you have specified port-based sample times.

Languages

C

See Also

`ssSetNumInputPorts`, `ssSetInputPortOffsetTime`

Purpose Specify the sample time index of an input port.

Syntax

```
void ssSetInputPortSampleTimeIndex(SimStruct *S,
   int_T inputPortIdx, int_T sampleTimeIdx)
```

Arguments

`S`
SimStruct representing an S-Function block or a Simulink model.

`inputPortIdx`
Index of the input port whose sample time index is being set.

`sampleTimeIdx`
Sample time index.

Description Use in `mdlInitializeSizes` (after `ssSetNumInputPorts`) to specify the index of the sample time for the port to be used in `mdlOutputs` and `mdlOutputs` when checking for sample hits.

Note This should only be used when the `PORT_BASED_SAMPLE_TIMES` has been specified for `ssSetNumSampleTimes` in `mdlInitializeSizes`.

Languages C, C++

See Also `ssGetInputPortSampleTimeIndex`, `mdlInitializeSizes`, `ssSetNumInputPorts`, `mdlOutputs`, `mdlOutputs`

ssSetInputPortVectorDimension

Purpose

Specify dimension information for an input port that accepts vector signals.

Syntax

```
int_T ssSetInputPortVectorDimension(SimStruct *S, int_T port, int_T
   w)
```

Arguments

`S`
SimStruct representing an S-Function block.

`port`
Index of an input port.

`w`
Width of vector or `DYNAMICALLY_SIZED`.

Description

Specifies that `port` accepts a `w`-element vector signal. Returns `1` if successful; otherwise, `0`.

Note This macro and `ssSetInputPortWidth` are functionally identical.

Languages

C

Example

The following example specifies that input port 0 accepts an 8-element matrix signal.

```
ssSetInputPortVectorDimension(S, 0, 8);
```

See Also

`ssSetInputPortWidth`

Purpose

Specify the width of an input port.

C Syntax

```
void ssSetInputPortWidth(SimStruct *S, int_T port, int_T width)
```

Ada Syntax

```
procedure ssSetInputPortWidth (S : in SimStruct;
   port : in Integer := 0; width : in Integer);
```

Arguments

`S`
SimStruct representing an S-Function block or a Simulink model.

`port`
Index of the input port whose width is being set.

`width`
Width of the input port.

Description

Use in `mdlInitializeSizes` (after `ssSetNumInputPorts`) to specify a nonzero positive integer width or `DYNAMICALLY_SIZED` for each input port index starting at 0.

Languages

Ada, C

See Also

`ssSetNumInputPorts`, `ssSetOutputPortWidth`

ssSetIWorkValue

Purpose Set an element of a block's integer work vector.

Syntax

```
int_T ssSetIWorkValue(SimStruct *S, int_T idx, int_T value)
```

Arguments

`S`
SimStruct representing an S-Function block.

`idx`
Index of the element to be set

`value`
New value of element

Description Sets the `idx` element of `S`'s integer work vector to `value`. The vector consists of elements of type `int_T` and is of length `ssGetNumIWork(S)`. Typically, this vector is initialized in `mdlStart` or `mdlInitializeConditions`, updated in `mdlUpdate`, and used in `mdlOutputs`. You can use this macro in the simulation loop, `mdlInitializeConditions`, or `mdlStart` routines. This macro returns the value that it sets.

Example The following statement

```
ssSetIWorkValue(s, 0, 1);
```

sets the first element of the work vector to `1`.

Languages C

See Also `ssGetNumIWork`, `ssGetIWork`, `ssGetIWorkValue`

Purpose

Set an element of a block's mode vector.

Syntax

```
void ssSetModeVectorValue(SimStruct *S, int_T element, int_T value)
```

Arguments

`S`
SimStruct representing an S-Function block.

`element`
Index of a mode vector element.

`value`
Mode vector value.

Description

Sets the specified mode vector element to the specified value.

Languages

C, C++

See Also

`ssGetModeVectorValue`, `ssGetModeVector`

Purpose Specify the number of continuous states that a block has.

C Syntax

```
void ssSetNumContStates(SimStruct *S, int_T n)
```

Ada Syntax

```
procedure ssSetNumContStates(S : in SimStruct; n : in Integer);
```

Arguments

`S`
SimStruct representing an S-Function block.

`n`
Number of continuous states to be set for the block represented by `S`.

Description Use in `mdlInitializeSizes` to specify the number of continuous states as 0, a positive integer, or `DYNAMICALLY_SIZED`. If you specify `DYNAMICALLY_SIZED`, you can specify the true (positive integer) width in `mdlSetWorkWidths`; otherwise, the width used is the width of the signal passing through the block. If your S-function has continuous states, it needs to return the derivatives of the states in `mdlDerivatives` so that the solvers can integrate them. Continuous states are logged if the **States** option is selected on the **Workspace I/O** pane of the **Simulation Parameters** dialog box.

Languages Ada, C

See Also `ssSetNumDiscStates`, `ssGetNumContStates`

Purpose Specify the number of discrete states that a block has.

Syntax

```
ssSetNumDiscStates(SimStruct *S, int_T nDiscStates)
```

Arguments

`S`
SimStruct representing an S-Function block.

`nDiscStates`
Number of discrete states to be set for the block represented by `S`.

Description Use in `mdlInitializeSizes` to specify the number of discrete states as 0, a positive integer, or `DYNAMICALLY_SIZED`. If you specify `DYNAMICALLY_SIZED`, you can specify the true (positive integer) width in `mdlSetWorkWidths`; otherwise, the width used is the width of the signal passing through the block. If your S-function has discrete states, it should return the next discrete state (in place) in `mdlUpdate`. Discrete states are logged if the **States** option is selected on the **Workspace I/O** page of the **Simulation Parameters** dialog box.

Languages C

See Also `ssSetNumContStates`, `ssGetNumDiscStates`

ssSetNumDWork

Purpose Specify the number of data type work vectors used by a block.

Syntax `void ssSetNumDWork(SimStruct *S, int_T nDWork)`

Arguments

`S`
SimStruct representing an S-Function block.

`nDWork`
Number of data type work vectors.

Description Use in `mdlInitializeSizes` to specify the number of data type work vectors as 0, a positive integer, or `DYNAMICALLY_SIZED`. If you specify `DYNAMICALLY_SIZED`, you can specify the true (positive integer) number of vectors in `mdlSetWorkWidths`.

You can specify the size and data type of each work vector, using the macros `ssSetDWorkWidth` and `ssSetDWorkDataType`, respectively. You can also specify that the work vector holds complex values, using `ssSetDWorkComplexSignal`.

Languages C, C++

See Also `ssGetNumDWork`, `ssSetDWorkWidth`, `ssSetDWorkDataType`, `ssSetDWorkComplexSignal`

Purpose Specify the number of input ports that a block has.

C Syntax

```
void ssSetNumInputPorts(SimStruct *S, int_T nInputPorts)
```

Ada Syntax

```
procedure ssSetNumInputPorts(S : in SimStruct;
   nInputPorts : in Integer);
```

Arguments

`S`
SimStruct representing an S-Function block.

`nInputPorts`
Number of input ports on the block represented by `S`. Must be a nonnegative integer.

Description Use in `mdlInitializeSizes` to set the number of input ports to a nonnegative integer. Invoke it using

```
if (!ssSetNumInputPorts(S,nInputPorts)) return;
```

where `ssSetNumInputPorts` returns 0 if *`nInputPorts`* is negative or an error occurs while creating the ports. When this happens, Simulink displays an error.

Languages Ada, C

See Also `ssSetInputPortWidth`, `ssSetNumOutputPorts`

ssSetNumIWork

Purpose Specify the size of a block's integer work vector.

Syntax

```
void ssSetNumIWork(SimStruct *S, int_T nIWork)
```

Arguments

`S`
SimStruct representing an S-Function block.

`nIWork`
Number of elements in the integer work vector.

Description Use in `mdlInitializeSizes` to specify the number of `int_T` work vector elements as 0, a positive integer, or `DYNAMICALLY_SIZED`. If you specify `DYNAMICALLY_SIZED`, you can specify the true (positive integer) width in `mdlSetWorkWidths`; otherwise, the width used is the width of the signal passing through the block.

Languages C

See Also `ssSetNumRWork`, `ssSetNumPWork`

Purpose Specifies the size of the block's mode vector.

Syntax

```
ssSetNumModes(SimStruct *S,nModes)
```

Arguments

`S`
SimStruct representing an S-Function block.

`nModes`
Size of the mode vector for the block represented by `S`. Valid values are 0, a positive integer, or `DYNAMICALLY_SIZED`.

Description

Sets the size of the block's mode vector to `nModes`. If `nModes` is `DYNAMICALLY_SIZED`, you can specify the true (positive integer) width in `mdlSetWorkWidths`; otherwise, the width used is the width of the signal passing through the block. Use this macro in `mdlInitializeSizes` to specify the number of `int_T` elements in the mode vector. Simulink allocates the mode vector and initializes its elements to 0. If the default value of 0 is not appropriate, you can set the elements of the array to other initial values in `mdlInitializeConditions`. Use `ssGetModeVector` to access the mode vector.

The mode vector, combined with zero-crossing detection, allows you to create blocks that have distinct operating modes, depending on the current values of input or output signals. For example, consider a block that outputs the absolute value of its input. Such a block operates in two distinct modes, depending on whether its input is positive or negative. If the input is positive, the block outputs the input unchanged. If the input is negative, the block outputs the negative of the input. You can use zero-crossing detection to detect when the input changes sign and update the single-element mode vector accordingly (for example, by setting its element to 0 for negative input and 1 for positive input). You can then use the mode vector in `mdlOutputs` to determine the mode in which the block is currently operating.

Languages C

See Also `ssGetNumModes`, `ssGetModeVector`

ssSetNumNonsampledZCs

Purpose

Specify the number of states for which a block detects zero crossings that occur between sample points.

Syntax

```
ssSetNumNonsampledZCs(SimStruct *S, nNonsampledZCs)
```

Arguments

`S`
SimStruct representing an S-Function block.

`nNonsampledZCs`
Number of nonsampled zero crossings that a block detects.

Description

Use in `mdlInitializeSizes` to specify the number of states for which the block detects nonsampled zero crossings (`real_T`) as 0, a positive integer, or `DYNAMICALLY_SIZED`. If you specify `DYNAMICALLY_SIZED`, you can specify the true (positive integer) width in `mdlSetWorkWidths`; otherwise, the width used is the width of the signal passing through the block.

Languages

C

See Also

`ssSetNumModes`

Purpose

Specify the number of output ports that a block has.

C Syntax

```
void ssSetNumOutputPorts(SimStruct *S, int_T nOutputPorts)
```

Ada Syntax

```
procedure ssSetNumOutputPorts(S : in SimStruct;
   nOutputPorts : in Integer);
```

Arguments

`S`
SimStruct representing an S-Function block.

`nOutputPorts`
Number of output ports on the block represented by `S`. Must be a nonnegative integer.

Description

Use in `mdlInitializeSizes` to set the number of output ports to a nonnegative integer. It should be invoked using

```
if (!ssSetNumOutputPorts(S,nOutputPorts)) return;
```

where `ssSetNumOutputPorts` returns 0 if *`nOutputPorts`* is negative or an error occurs while creating the ports. When this occurs, and you return out of your S-function, Simulink displays an error message.

Languages

Ada, C

See Also

`ssSetInputPortWidth`, `ssSetNumInputPorts`

ssSetNumPWork

Purpose Specify the size of a block's pointer work vector.

Syntax

```
void ssSetNumPWork(SimStruct *S, int_T nPWork)
```

Arguments

`S`
SimStruct representing an S-Function block.

`nPWork`
Number of elements to be allocated to the pointer work vector of the block represented by `S`.

Description Use in `mdlInitializeSizes` to specify the number of pointer (`void *`) work vector elements as 0, a positive integer, or `DYNAMICALLY_SIZED`. If you specify `DYNAMICALLY_SIZED`, you can specify the true (positive integer) width in `mdlSetWorkWidths`; otherwise, the width used is the width of the signal passing through the block.

Languages C

See Also `ssGetNumPWork`

Purpose Specify the number of run-time parameters created by this S-function.

Syntax

```
void ssSetNumRunTimeParams(S, int_T num)
```

Arguments

`S`
SimStruct representing an S-Function block.

`num`
Number of run-time parameters.

Description Use this function in `mdlSetWorkWidths` to specify the number of run-time parameters created by this S-function.

Languages C

See Also `mdlSetWorkWidths`, `ssGetNumRunTimeParams`, `ssSetRunTimeParamInfo`

ssSetNumRWork

Purpose Specify the size of a block's floating-point work vector.

Syntax

```
void ssSetNumRWork(SimStruct *S, int_T nRWork)
```

Arguments

`S`
SimStruct representing an S-Function block.

`nRWork`
Number of elements in the floating-point work vector.

Description Use in `mdlInitializeSizes` to specify the number of `real_T` work vector elements as 0, a positive integer, or `DYNAMICALLY_SIZED`. If you specify `DYNAMICALLY_SIZED`, you can specify the true (positive integer) width in `mdlSetWorkWidths`; otherwise, the width used is the width of the signal passing through the block.

Languages C

See Also `ssSetNumIWork`, `ssSetNumPWork`

Purpose

Specify the number of sample times that an S-Function block has.

Syntax

```
void ssSetNumSampleTimes(SimStruct *S, int_T nSampleTimes)
```

Arguments

`S`
SimStruct representing an S-Function block.

`nSampleTimes`
Number of sample times that `S` has.

Description

Use in `mdlInitializeSizes` to set the number of sample times `S` has. This must be a positive integer greater than 0.

Languages

C

See Also

`ssGetNumSampleTimes`

ssSetNumSFcnParams

Purpose Specify the number of parameters that an S-Function block has.

Syntax `ssSetNumSFcnParams(SimStruct *S, int_T nSFcnParams)`

Arguments

`S`
SimStruct representing an S-Function block.

`nSFcnParams`
Number of parameters that `S` has.

Description Use in `mdlInitializeSizes` to set the number of S-function parameters.

Languages C

See Also `ssGetNumSFcnParams`

Purpose Set the offset time of a block.

Syntax

```
ssSetOffsetTime(SimStruct *S, st_index, period)
```

Arguments

`S`
SimStruct representing an S-Function block.

`st_index`
Index of the sample time whose offset is to be set.

`offset`
Offset of the sample time specified by `st_index`.

Description Use this macro in `mdlInitializeSizes` to specify the offset of the sample time where `st_index` starts at 0.

Languages C

See Also `ssSetSampleTime`, `ssSetInputPortOffsetTime`, `ssSetOutputPortOffsetTime`

ssSetOptions

Purpose Specify S-function options.

Syntax

```
void ssSetOptions(SimStruct *S, uint_T options)
```

Arguments

`S`
SimStruct representing an S-Function block.

`options`
Options.

Description Use in `mdlInitializeSizes` to specify S-function options (see following). The options must be joined using the `OR` operator. For example:

```
ssSetOption(S, (SS_OPTION_EXCEPTION_FREE_CODE |
                SS_OPTION_DISCRETE_VALUED_OUTPUT));
```

S-Function Options

An S-function can specify the following options, using `ssSetOptions`:

- `SS_OPTION_EXCEPTION_FREE_CODE`

  If your S-function does not use `mexErrMsgTxt`, `mxCalloc`, or any other routines that can throw an exception when called, you can set this option for improved performance.
- `SS_OPTION_RUNTIME_EXCEPTION_FREE_CODE`

  Similar to `SS_OPTION_EXCEPTION_FREE_CODE` except it only applies to the run-time routines `mdlGetTimeOfNextVarHit`, `mdlOutputs`, `mdlUpdate`, and `mdlDerivatives`.
- `SS_OPTION_DISCRETE_VALUED_OUTPUT`

  Specify this if your S-function has discrete valued outputs. This is checked when your S-function is placed within an algebraic loop. If your S-function has discrete valued outputs, its outputs are not assigned algebraic variables.
- `SS_OPTION_PLACE_ASAP`

  Use to specify that your S-function should be placed as soon as possible. This is typically used by devices connecting to hardware.
- `SS_OPTION_ALLOW_INPUT_SCALAR_EXPANSION`

  Use to specify that the input to your S-function input ports can be either 1 or the size specified by the port, which is usually referred to as the block width.

- SS_OPTION_DISALLOW_CONSTANT_SAMPLE_TIME

  Use to disable an S-Function block from inheriting a constant sample time.
- SS_OPTION_ASYNCHRONOUS

  This option applies only to S-functions that have 0 or 1 input ports and 1 output port. The output port must be configured to perform function calls on every element. If any of these requirements is not met, the SS_OPTION_ASYNCHRONOUS option is ignored. Use this option when driving function-call subsystems to attached to interrupt service routines.
- SS_OPTION_ASYNC_RATE_TRANSITION

  Use this option to create a read-write pair of blocks intended to guarantee correct data transfers between a synchronously and an asynchronously executing subsystem or between two asynchronously executing subsystems. Both your "read" S-function and your "write" S-function should set this option. See the comment for SS_OPTION_ASYNC_RATE_TRANSITION in symstruc.h for more information.
- SS_OPTION_PORT_SAMPLE_TIMES_ASSIGNED

  Use this when you have registered multiple sample times (ssSetNumSampleTimes > 1) to specify the rate at which each input and output port is running. The simulation engine needs this information when checking for illegal rate transitions.
- SS_OPTION_SFUNCTION_INLINED_FOR_RTW

  Set this if you have a .tlc file for your S-function and do not have an mdlRTW method. Setting this option has no effect if you have an mdlRTW method.
- SS_OPTION_ALLOW_PARTIAL_DIMENSIONS_CALL

  Indicates that the S-function can handle dynamically dimensioned signals. See mdlSetDefaultPortDimensionInfo for more information.
- SS_OPTION_FORCE_NONINLINED_FCNCALL

  Use this flag if the block requires that all function-call subsystems that it calls should be generated as procedures instead of possibly being generated as inlined code.
- SS_OPTION_USE_TLC_WITH_ACCELERATOR

  Use this to force the Accelerator to use the TLC inlining code for an S-function, which speeds up execution of the S-function. By default, the Accelerator uses the mex version of the S-function even though a TLC file for the S-function exists. This option should not be set for device driver blocks

(A/D) or when there is an incompatibility between running the `mex Start/InitializeConditions` functions together with the TLC `Outputs/Update/Derivatives`.

- `SS_OPTION_SIM_VIEWING_DEVICE`

  This S-function is a `SimViewingDevice`. As long as it meets the other requirements for this type of block (no states, no outputs, etc.), it is considered to be an external mode block (it show up in the external mode GUI and no code is generated for it). During an external mode simulation, this block is run on the host only.

- `SS_OPTION_CALL_TERMINATE_ON_EXIT`

  This option allows S-function authors to better manage the data cached in run-time parameters and `UserData`. Setting this option guarantees that the `mdlTerminate` function is called if `mdlInitializeSizes` is called. This means that `mdlTerminate` is called

  - When a simulation ends

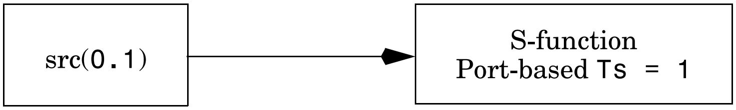
    Note that it does not matter if the simulation fails and at what stage the simulation fails. Therefore, if the `mdlSetWorkWidths` of some block errors out, the model's other blocks have a chance to free the memory during a call to `mdlTerminate`.

  - Every time an S-Function block is destroyed
  - If the user is editing the S-function graphically
  - If the S-Function block was reduced as a result of invoking `ssSetBlockReduction`

  If this option is not set, `mdlTerminate` is called only if at least one of the blocks has had its `mdlStart` called.

- `SS_OPTION_REQ_INPUT_SAMPLE_TIME_MATCH`

  Use this to option to specify that the input signal sample times match the sample time assigned to the block input port. For example:



  generates an error if this option is set. If the block (or input port) sample time is inherited, no error is generated.

- `SS_OPTION_WORKS_WITH_CODE_REUSE`

  Signifies that this S-function is compatible with the subsystem code reuse feature of the Real-Time Workshop (see "Creating Code-Reuse-Compatible S-Functions" on page 8-42).
- `SS_OPTION_ALLOW_CONSTANT_PORT_SAMPLE_TIME`

  Set this option in `mdlInitializeSizes` to allow your S-function's ports to specify or inherit a constant sample time (see "Specifying Constant Sample Time for a Port" on page 7-20 for more information).
- `SS_OPTION_ALLOW_PORT_BASED_SAMPLE_TIME_IN_TRIGSS`

  Set this option in `mdlInitializeSizes` to allow an S-function that uses port-based sample times to operate in a triggered subsystem (see "Configuring Port-Based Sample Times for Use in Triggered Subsystems" on page 7-21 for more information).

Languages C, C++

ssSetOutputPortComplexSignal

Purpose

Set the numeric type (real or complex) of an output port.

Syntax

```
void ssSetOutputPortComplexSignal(SimStruct *S, input_T port,
   CSignal_T csig)
```

Arguments

S
SimStruct representing an S-Function block or a Simulink model.

port
Index of an output port.

csignal
Numeric type of the signals emitted by port. Valid values are COMPLEX_NO (real signal), COMPLEX_YES (complex signal), and COMPLEX_INHERITED (dynamically determined).

Description

Use this function in mdlInitializeSizes to initialize an input port signal type. If the numeric type of the input port is determined dynamically, e.g., by a parameter setting, set the numeric type to COMPLEX_INHERITED. The default numeric type of an output port is real.

Languages

C

Example

Assume that an S-function has three output ports. The first output port emits real (noncomplex) signals. The second input port emits a complex signal. The third port emits signals of a type determined by a parameter setting. The following example specifies the correct numeric type for each port.

```
ssSetOutputPortComplexSignal(S, 0, COMPLEX_NO)
ssSetOutputPortComplexSignal(S, 1, COMPLEX_YES)
ssSetOutputPortComplexSignal(S, 2, COMPLEX_INHERITED)
```

See Also

ssGetOutputPortComplexSignal

Purpose

Set the data type of an output port.

C Syntax

```
void ssSetOutputPortDataType(SimStruct *S, int_T port, DTypeId id)
```

Ada Syntax

```
procedure ssSetOutputPortDataType(S : in SimStruct;
   port  : in Integer := 0; id : in Integer);
  Arguments
```

S
SimStruct representing an S-Function block or a Simulink model.

port
Index of an output port.

id
ID of the data type accepted by port.

Description

Use this function in mdlInitializeSizes to set the data type of the output port specified by port. If the output port's data type is determined dynamically, for example, from the data type of a block parameter, set the data type to DYNAMICALLY_TYPED.

Note The data type of an output port is double (real_T) by default.

Languages

Ada, C

Example

Suppose that you want to create an S-function with two output ports, the first of which gets its data type from a block parameter and the second of which outputs signals of type int16_T. The following code sets up the data types.

```
ssSetOutputPortDataType(S, 0, DYNAMICALLY_TYPED)
ssSetOutputPortDataType(S, 1, SS_INT16)
```

See Also

ssGetOutputPortDataType

ssSetOutputPortDimensionInfo

Purpose

Specify information about the dimensionality of an output port.

Syntax

```
void ssSetInputPortDimensionInfoSimStruct *S,  int_T port,
  DimsInfo_T *dimsInfo)
```

Arguments

`S`
SimStruct representing an S-function block.

`port`
Index of an output port

`dimsInfo`
Structure of type `DimsInfo_T` that specifies the dimensionality of the signals emitted by `port`

See `ssSetInputPortDimensionInfo` for a description of this structure.

Description

Specifies the dimension information for `port`. Use this function in `mdlInitializeSizes` to initialize the output port dimension info. If you want the port to inherit its dimensionality from the block to which it is connected, specify `DYNAMIC_DIMENSION` as the `dimsInfo` for `port`.

Languages

C

Example

The following example specifies that input port 0 accepts 2-by-2 matrix signals.

```
DECL_AND_INIT_DIMSINFO(di);
di.numDims = 2;
int dims[2];
dims[0] = 2;
dims[1] = 2;
di.dims = &dims;
di.width = 4;
ssSetOutputPortDimensionInfo(S,  0, &di);
```

See Also

`ssSetInputPortDimensionInfo`

Purpose Specify whether a port outputs framed data.

Syntax

```
void ssSetOutputPortFrameData(SimStruct *S,  int_T port,
   int_T outputsFrames)
```

Arguments

`S`
SimStruct representing an S-Function block.

`port`
Index of an output port.

`outputsFrames`
Type of signal output by `port`. Acceptable values are `-1` (either frame or unframed input), `0` (unframed input only), and `1` (framed input only).

Description Use in `mdlSetInputPortFrameData` to specify whether an output port issues frame data only, unframed data only, or both.

Languages C

See Also `ssGetOutputPortFrameData`, `mdlSetInputPortFrameData`

ssSetOutputPortMatrixDimensions

Purpose

Specify dimension information for an output port that emits matrix signals.

Syntax

```
int_T ssSetOutputPortMatrixDimensions(SimStruct *S, int_T port,
   int_T m, in_T n)
```

Arguments

`S`
SimStruct representing an S-Function block.

`port`
Index of an output port.

`m`
Row dimension of matrix signals emitted by `port` or `DYNAMICALLY_SIZED`.

`n`
Column dimension of matrix signals emitted by `port` or `DYNAMICALLY_SIZED`.

Description

Specifies that `port` emits an `m`-by-`n` matrix signal. If either dimension is `DYNAMICALLY_SIZED`, the other must be `DYNAMICALLY_SIZED` or 1. Returns `1` if successful; otherwise, `0`.

Languages

C

Example

The following example specifies that input port 0 emits 2-by-2 matrix signals.

```
ssSetOutputPortMatrixDimensions(S,  0, 2, 2);
```

Purpose Specify the offset time of an output port.

Syntax

```
ssSetOutputPortOffsetTime(SimStruct *S,outputPortIdx,offset)
```

Arguments

`S`
SimStruct representing an S-Function block.

`outputPortIdx`
Index of the output port whose sample time is being set.

`offset`
Sample time of an output port.

Description Use in `mdlInitializeSizes` (after `ssSetNumOutputPorts`) to specify the sample time offset value for each output port index. This should only be used if you have specified the S-function's sample times as port-based.

Languages C

See Also `ssSetNumOutputPorts`, `ssSetOutputPortSampleTime`

ssSetOutputPortReusable

Purpose

Specify that an output port is reusable.

Syntax

```
ssSetOutputPortReusable(SimStruct *S,outputPortIdx,isReusable)
```

Arguments

`S`
SimStruct representing an S-Function block.

`outputPortIdx`
Index of the output port whose reusability is being set.

`isReusable`
Value specifying reusability of the port.

Description

Use in `mdlInitializeSizes` (after `ssSetNumOutputPorts`) to specify whether output ports have a test point. This macro can take one of two values:

- Off `(isReusable=0)` specifies that the output port is not reusable. This is the default.
- On `(isReusable=1)` specifies that the output port is reusable.

In Simulink, reusable signals share the same memory space. This macro allows an S-function to tell Simulink that it can store the S-function's outputs temporarily in memory used for storing other signals in the model. This reuse results in less memory use during simulation and more efficiency in the Real-Time Workshop generated code.

When you mark an output port as reusable, your S-function must update the output once in `mdlOutputs`. It cannot expect the previous output value to be persistent.

By default, the output port signals are not reusable. This forces Simulink's simulation engine (and the Real-Time Workshop) to allocate global memory for these output port signals. Hence this memory is only written to by your S-function and persists between model execution steps.

Note If you want to allow users to connect the output of your S-function to a Merge block, you must use this macro to specify that your S-function's output ports are reusable.

Languages C

See Also `ssSetNumOutputPorts`, `ssSetInputPortReusable`

ssSetOutputPortSampleTime

Purpose Specify the sample time of an output port.

Syntax

```
ssSetOutputPortSampleTime(SimStruct *S,outputPortIdx,period)
```

Arguments

`S`
SimStruct representing an S-Function block.

`outputPortIdx`
Index of the output port whose sample time is being set.

`period`
Sample time of output port.

Description Use in `mdlInitializeSizes` (after `ssSetNumOutputPorts`) to specify the sample time period as continuous or as a discrete value for each output port index. This should only be used if you have specified port-based sample times.

Languages C

See Also `ssSetNumOutputPorts`, `ssSetOutputPortOffsetTime`

Purpose Specify dimension information for an output port that emits vector signals.

Syntax

```
int_T ssSetOutputPortVectorDimension(SimStruct *S,  int_T port,
   int_T w)
```

Arguments

`S`
SimStruct representing an S-Function block.

`port`
Index of an output port.

`w`
Width of vector or `DYNAMICALLY_SIZED`.

Description Specifies that `port` emits a `w`-element vector signal. Returns `1` if successful; otherwise, `0`.

Note This macro and `ssSetOutputPortWidth` are functionally identical.

Example The following example specifies that output port 0 emits an 8-element matrix signal.

```
ssSetOutputPortVectorDimension(S, 0, 8);
```

Languages C

See Also `ssSetOutputPortWidth`

ssSetOutputPortWidth

Purpose Specify the width of an output port.

C Syntax

```
void ssSetOutputPortWidth(SimStruct *S, int_T port, int_T width)
```

Ada Syntax

```
procedure ssSetOutputPortWidth(S : in SimStruct;
   port : in Integer := 0; Width : in Integer);
```

Arguments

`S`
SimStruct representing an S-Function block.

`port`
Index of the output port whose width is being set.

`width`
Width of an output port.

Description Use in `mdlInitializeSizes` (after `ssSetNumOutputPorts`) to specify a nonzero positive integer width or `DYNAMICALLY_SIZED` for each output port index starting at 0.

Languages Ada, C

See Also `ssSetNumOutputPorts`, `ssSetInputPortWidth`

Purpose Set the name of a parameter.

Syntax

```
procedure ssSetParameterName(S : in SimStruct; Parameter : in
   Integer; Name : in String);
```

Arguments

`S`
SimStruct representing an S-Function block.

`Parameter`
Index of a parameter.

`Name`
Name of the parameter.

Description Sets the name of `Parameter` to `Name`.

Languages Ada

ssSetParameterTunable

Purpose

Set the tunability of a parameter.

Syntax

```
procedure ssSetParameterTunable(S : in SimStruct; Parameter : in
   Integer; IsTunable : in Boolean);
```

Arguments

`S`
SimStruct representing an S-Function block.

`Parameter`
Index of a parameter.

`IsTunable`
True indicates that the parameter is tunable.

Description

Sets the tunability of `Parameter` to the value of `IsTunable`.

Languages

Ada

Purpose Specify the name of the placement group of a block.

Syntax

```
void ssSetPlacementGroup(SimStruct *S, const char *groupName)
```

Arguments

`S`
SimStruct representing an S-Function block. The block must be either a source block (i.e., a block without input ports) or a sink block (i.e., a block without output ports).

`groupName`
Name of the placement group of the block represented by `S`.

Description Use this macro to specify the name of the placement group to which the block represented by `S` belongs. S-functions that share the same placement group name are placed adjacent to each other in the block execution order list for the model. This macro should be invoked in `mdlInitializeSizes`.

Note You typically use this macro is to create Real-Time Workshop device driver blocks.

Languages C

See Also `ssGetPlacementGroup`

ssSetPWorkValue

Purpose Set an element of a block's pointer work vector.

Syntax

```
void* ssSetPWorkValue(SimStruct *S, int_T idx, void* pointer)
```

Arguments

`S`
SimStruct representing an S-Function block.

`idx`
Index of the element to be set

`pointer`
New pointer element

Description Sets the `idx` element of `S`'s pointer work vector to `pointer`. The vector consists of elements of type `void*` and is of length `ssGetNumPWork(S)`. Typically, this vector is initialized in `mdlStart` or `mdlInitializeConditions`, updated in `mdlUpdate`, and used in `mdlOutputs`. You can use this macro in the simulation loop, `mdlInitializeConditions`, or `mdlStart` routines. This macro returns the pointer that it sets.

Example The following statement

```
typedef struct Color_tag {int r; int b; int g;} Color;
Color* p = malloc(sizeof(Color));
ssSetPWorkValue(s, 0, p);
```

sets the first element of the pointer work vector to a pointer to the allocated `Color` structure.

Languages C

See Also `ssGetNumPWork`, `ssGetPWork`, `ssGetPWorkValue`

Purpose Set an element of a block's floating-point work vector.

Syntax

```
real_T ssSetRWorkValue(SimStruct *S, int_T idx, real_T value)
```

Arguments

`S`
SimStruct representing an S-Function block.

`idx`
Index of the element to be set

`value`
New value of element

Description Sets the `idx` element of `S`'s floating-point work vector to `value`. The vector consists of elements of type `real_T` and is of length `ssGetNumRWork(S)`. Typically, this vector is initialized in `mdlStart` or `mdlInitializeConditions`, updated in `mdlUpdate`, and used in `mdlOutputs`. You can use this macro in the simulation loop, `mdlInitializeConditions`, or `mdlStart` routines. This macro returns the value that it sets.

Example The following statement

```
ssSetRWorkValue(s, 0, 1.0);
```

sets the first element of the work vector to `1.0`.

Languages C

See Also `ssGetNumRWork`, `ssGetRWork`, `ssGetRWorkValue`

ssSetRunTimeParamInfo

Purpose

Specify the attributes of a run-time parameter.

Syntax

```
void ssSetRunTimeParamInfo(SimStruct *S, int_T param, ssParamRec
   *info)
```

Arguments

`S`
SimStruct representing an S-Function block.

`param`
Index of a run-time parameter.

Description

Use this function in `mdlSetWorkWidths` or `mdlProcessParameters` to specify information about a run-time parameter. Use an `ssParamRec` structure to pass the parameter attributes to the function.

ssParamRec Structure

The `simstruc.h` macro defines this structure as follows:

```
typedef struct ssParamRec_tag {
    const char *name;
    int_T      nDimensions;
    int_T      *dimensions;
    DTypeId    dataTypeId;
    boolean_T  complexSignal;
    void       *data;
    const void *dataAttributes;
    int_T      nDlgParamIndices;
    int_T      *dlgParamIndices;
    TransformedFlag transformed;  /* Transformed status */
    boolean_T  outputAsMatrix;    /* Write out parameter as a
vector (false)
                                 * [default] or a matrix (true)
                                   */
} ssParamRec;
```

The record contains the following fields.

name. Name of the parameter. This must point to persistent memory. Do not set to a local variable (`static char name[32]` or strings name are okay).

Note The first four characters of block's run-time parameter names 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.

---

nDimensions. Number of dimensions that this parameter has.

dimensions. Array giving the size of each dimension of the parameter.

dataTypeId. Data type of the parameter. For built-in data types, see `BuiltInDTypeId` in `simstruc_types.h`.

complexSignal. Specifies whether this parameter has complex numbers (true) or real numbers (false) as values.

data. Pointer to the value of this run-time parameter. If the parameter is a vector or matrix or a complex number, this field points to an array of values representing the parameter elements. Complex Simulink signals are stored interleaved. Likewise complex run-time parameters must be stored interleaved. Note that `mxArrays` stores the real and complex parts of complex matrices as two separate contiguous pieces of data instead of interleaving the real and complex parts.

dataAttributes. The data attributes pointer is a persistent storage location where the S-function can store additional information describing the data and then recover this information later (potentially in a different function).

nDlgParamIndices.

Number of dialog parameters used to compute this run-time parameter.

dlgParamIndices. Indices of dialog parameters used to compute this run-time parameter.

transformed. Specifies the relationship between this run-time parameter and the dialog parameters specified by `dlgParamIndices`. This field can have any of the following values defined by `TransformFlag` in `simstruc.h`.

- `RTPARAM_NOT_TRANSFORMED`

  Specifies that this run-time parameter corresponds to a single dialog parameter (`nDialogParamIndices` is one) and has the same value as the dialog parameter.

- `RTPARAM_TRANSFORMED`

  Specifies that the value of this run-time parameter depends on the values of multiple dialog parameters (`nDialogParamIndices > 1`) or that this run-time parameter corresponds to one dialog parameter but has a different value or data type.

- `RTPARAM_MAKE_TRANSFORMED_TUNABLE`

  Specifies that this run-time parameter corresponds to a single tunable dialog parameter (`nDialogParamIndices` is one) and that the run-time parameter's value or data type differs from the dialog parameter's. During code generation, Real-Time Workshop writes the data type and value of the run-time parameter (rather than the dialog parameter) out to the Real-Time Workshop file. For example, suppose that the dialog parameter contains a workspace variable `k` of type `double` and value `1`. Further, suppose the S-function sets the data type of the corresponding run-time variable to `int8` and the run-time parameter's value to `2`. In this case, during code generation, the Real-Time Workshop writes `k` out to the Real-Time Workshop file as an `int8` variable with an initial value of `2`.

outputAsMatrix. Specifies whether to write the values of this parameter out to the `model.rtw` file as a matrix (true) or as a vector (false).

Languages

C

See Also

`mdlSetWorkWidths`, `mdlProcessParameters`, `ssGetNumRunTimeParams`, `ssGetRunTimeParamInfo`

Purpose
Set the period of a sample time.

C Syntax

```
void ssSetSampleTime(SimStruct *S, st_index, time_T period)
```

Ada Syntax

```
procedure ssSetSampleTime(S      : in SimStruct; Period : in time_T;
      st_index : in time_T := 0.0);
```

Arguments

`S`
SimStruct representing an S-Function block.

`st_index`
Index of the sample time whose period is to be set.

`period`
Period of the sample time specified by `st_index`.

Description
Use this macro in `mdlInitializeSizes` to specify the period of the sample time where *`st_index`* starts at 0.

Languages
Ada, C

See Also
`ssSetInputPortSampleTime`, `ssSetOutputPortSampleTime`, `ssSetOffsetTime`

ssSetSFcnParamNotTunable

Purpose Make a block parameter nontunable.

Syntax

```
void ssSetSFcnParamNotTunable(SimStruct *S, int_T index)
```

Arguments

`S`
SimStruct representing an S-Function block.

`index`
Index of the parameter to be made nontunable.

Description

Use this macro in `mdlInitializeSizes` to specify that a parameter doesn't change during the simulation, where *`index`* starts at 0 and is less than `ssGetSFcnParamsCount(S)`. This improves efficiency and provides error handling in the event that an attempt is made to change the parameter.

Note This macro is obsolete. It is provided only for compatibility with S-functions created with earlier versions of Simulink.

Languages C

See Also `ssSetSFcnParamTunable`, `ssGetSFcnParamsCount`

Purpose

Make a block parameter tunable.

Syntax

```
void ssSetSFcnParamTunable(SimStruct *S, int_T param,
   int_T isTunable)
```

Arguments

`S`
SimStruct representing an S-Function block.

`param`
Index of the parameter.

`isTunable`
Valid values are 1 (tunable) or 0 (not tunable).

Description

Use this macro in `mdlInitializeSizes` to specify whether a user can change a dialog parameter during the simulation. The parameter index starts at 0 and is less than `ssGetSFcnParamsCount(S)`. This improves efficiency and provides error handling in the event that an attempt is made to change the parameter.

Note Dialog parameters are tunable by default. However, an S-function should declare the tunability of all parameters, whether tunable or not, to avoid programming errors. 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.

Languages

C

See Also

`ssGetSFcnParamsCount`

ssSetSolverNeedsReset

Purpose Ask Simulink to reset the solver.

Syntax `void ssSetSolverNeedsReset(SimStruct *S)`

Arguments `S`
SimStruct representing an S-Function block or a Simulink model.

Description This macro causes the solver for the current simulation to reinitialize variable step size and zero-crossing computations. This happens only if the solver is a variable-step, continuous solver. (The macro has no effect if the user has selected another type of solver for the current simulation.) An S-function should invoke this macro whenever changes occur in the dynamics of the S-function, e.g., a discontinuity in a state or output, that might invalidate the solver's step-size computations. Otherwise, the solver might take unnecessarily small steps, slowing down the simulation.

Note If a change in the dynamics of the S-function necessitates reinitializing its continuous states, the S-function should reinitialize the states before invoking this macro to ensure accurate computation of the next step size.

Languages C

Example The following example uses this macro to ask Simulink to reset the solver.

```
static void mdlOutputs(SimStruct *S, int_T tid)
{
    :
    : <snip>
    :
    if ( under_certain_conditions ) {
         double *x = ssGetContStates(S);
         /* reset the states */
         for (i=0; i<nContStates; i++) {
              x[i] = 0.0;
         }
         /* Ask Simulink to reset the solver. */
         ssSetSolverNeedsReset(S);
    }
}
```

Also see the source code for the Time-Varying Continuous Transfer Function (`matlabroot/simulink/src/stvctf.c`) for an example of where and how to use this macro.

ssSetStopRequested

Purpose Set the simulation stop requested flag.

Syntax

```
ssSetStopRequested(SimStruct *S, val)
```

Arguments

`S`
SimStruct representing an S-Function block or a Simulink model.

`val`
Boolean value (`int_T`) specifying whether stopping the simulation has been requested (1) or not (0).

Description Sets the simulation stop requested flag to `val`. If `val` is not 0, Simulink halts the simulation at the end of the current time step.

Languages C

See Also `ssGetStopRequested`

Purpose Set the time of the next sample hit.

Syntax

```
void ssSetTNext(SimStruct *S, time_T tnext)
```

Arguments

`S`
SimStruct representing an S-Function block.

`tnext`
Time of the next sample hit.

Description A discrete S-function with a variable sample time should use this macro in `mdlGetTimeOfNextVarHit` to specify the time of the next sample hit.

Languages C

See Also `ssGetTNext`, `ssGetT`, `mdlGetTimeOfNextVarHit`

ssSetUserData

Purpose Specify user data.

Syntax

```
void ssSetUserData(SimStruct *S, void * data)
```

Arguments

`S`
SimStruct representing an S-Function block.

`data`
User data.

Description Specifies user data.

Languages C, C++

See Also `ssGetUserData`

Purpose Specify the vector mode that an S-function supports.

Syntax

```
void ssSetVectorMode(SimStruct *S,  ssVectorMode mode)
```

Arguments

`S`
SimStruct representing an S-Function block.

`mode`
Vector mode.

Description Specifies the types of vector-like signals that an S-Function block's input and output ports support. Simulink uses this information during signal dimension propagation to check the validity of signals connected to the block or emitted by the block. The enumerated type `ssVectorMode` defines the set of values that `mode` can have.

Mode Value	Signal Dimensionality Supported
`SS_UNKNOWN_MODE`	Unknown
`SS_1_D_OR_COL_VECT`	1-D (vector) or single-column 2-D (column vector)
`SS_1_D_OR_ROW_VECT`	1-D or single-row 2-D (row vector) signals
`SS_1_D_ROW_OR_COL_VECT`	Vector or row or column vector
`SS_1_D_VECT`	Vector
`SS_COL_VECT`	Column vector
`SS_ROW_VECT`	Row vector

Languages C

Example See `simulink/src/sfun_bitop.c` for examples that use this macro.

ssUpdateAllTunableParamsAsRunTimeParams

Purpose

Update the values of run-time parameters to be the same as those of the corresponding tunable dialog parameters.

Syntax

```
void ssUpdateAllTunableParamsAsRunTimeParams(SimStruct *S)
```

Arguments

`S`

Description

Use this macro in the S-function's `mdlProcessParameters` method to update the values of all run-time parameters created by the `ssRegAllTunableParamsAsRunTimeParams` macro.

Languages

C

See Also

`mdlProcessParameters`, `ssUpdateRunTimeParamInfo`, `ssRegAllTunableParamsAsRunTimeParams`

Purpose Update the value of a run-time parameter.

Syntax

```
void ssUpdateRunTimeParamData(SimStruct *S, int_T param, void *data)
```

Arguments

`S`
SimStruct representing an S-Function block.

`param`
Index of a run-time parameter.

`data`
New value of the parameter.

Description Use this macro in the S-function's `mdlProcessParameters` method to update the value of the run-time parameter specified by `param`.

Languages C

See Also `mdlProcessParameters`, `ssGetRunTimeParamInfo`, `ssUpdateAllTunableParamsAsRunTimeParams`, `ssRegAllTunableParamsAsRunTimeParams`

ssUpdateDlgParamAsRunTimeParam

Purpose Update a run-time parameter that corresponds to a dialog parameter.

Syntax

```
ssUpdateDlgParamAsRunTimeParam(S, rtIdx)
```

Arguments

`S`
SimStruct representing an S-Function block or a Simulink model.

`rtIdx`
Index of the run-time parameter

Description Use in `mdlProcessParameters` to set the value of the run-time parameter specified by rtIdx to the current value of the dialog parameter specified by `dlgIdx`. If necessary, this function converts the data type of the value to the data type specified by `dtId`.

Languages C

See Also `ssUpdateAllTunableParamsAsRunTimeParams`

Purpose Update the attributes of a run-time parameter.

Syntax

```
void ssUpdateRunTimeParamInfo(SimStruct *S, int_T param, ssParamRec
   *info)
```

Arguments

`S`
SimStruct representing an S-Function block.

`param`
Index of a run-time parameter.

`info`
Attributes of the run-time parameter.

Description Use this macro in the S-function's `mdlProcessParameters` method to update specific run-time parameters. For each parameter to be updated, the method should first obtain a pointer to the parameter's attributes record (`ssParamRec`), using `ssGetRunTimeParamInfo`. The method should then update the record and pass it back to Simulink, using this macro.

Note If you used `ssRegAllTunableParamsAsRunTimeParams` to create the run-time parameters, use `ssUpdateAllTunableParamsAsRunTimeParams` to update the parameters.

Languages C

See Also `mdlProcessParameters`, `ssGetRunTimeParamInfo`, `ssUpdateAllTunableParamsAsRunTimeParams`, `ssRegAllTunableParamsAsRunTimeParams`

ssWarning

Purpose Display a warning message.

Syntax `ssWarning(SimStruct *S, msg)`

Arguments

`S`
SimStruct representing an S-Function block or a Simulink model.

`msg`
Warning message.

Description Displays `msg`. Expands to `mexWarnMsgTxt` when compiled for use with Simulink. When compiled for use with the Real-Time Workshop, expands to `printf("Warning:%s from '%s'\n",msg, ssGetPath(S));`, if the target has `stdio` facilities; otherwise, it expands to a comment.

Languages C

See Also `ssSetErrorStatus`, `ssPrintf`

Purpose

Write a matrix parameter to the `model.rtw` file.

Syntax

```
int_T ssWriteRTW2dMatParam(SimStruct *S,  const char_T *name,
  const void *value, int_T dataType, int_T nRows, int_T nCols)
```

Arguments

`S`
SimStruct representing an S-Function block.

`name`
Parameter name.

`value`
Parameter values.

`dataType`
Data type of parameter elements (see “Specifying Data Type Info” on page 10-230).

`nRows`
Number of rows in the matrix.

`nColumns`
Number of columns in the matrix.

Description

Use this function in `mdlRTW` to write a vector of numeric parameters to this S-function’s `model.rtw` file. This function returns true if successful.

Languages

C

See Also

`mdlRTW`

ssWriteRTWMx2dMatParam

Purpose Write a matrix parameter in MATLAB format to the `model.rtw` file.

Syntax

```
int_T ssWriteRTWMx2dMatParam(SimStruct *S,  const char_T *name,
  const void *rValue, const void *iValue, int_T dataType, int_T
  nRows, int_T nCols)
```

Arguments

`S`
SimStruct representing an S-Function block.

`name`
Parameter name.

`rValue`
Real elements of the parameter array.

`iValue`
Imaginary elements of the parameter array.

`dataType`
Data type of the parameter elements (see “Specifying Data Type Info” on page 10-230).

`nRows`
Number of rows in the matrix.

`nColumns`
Number of columns in the matrix.

Description Use this function in `mdlRTW` to write a matrix parameter in MATLAB format to this S-function’s `model.rtw` file. This function returns true if successful.

Languages C

See Also `mdlRTW`, `ssWriteRTW2dMatParam`

Purpose Write a vector parameter in MATLAB format to the `model.rtw` file.

Syntax

```
int_T ssWriteRTWMxVectParam(SimStruct *S,  const char_T *name,
  const void *rValue, const void *iValue, int_T dataType, int_T
  size)
```

Arguments

`S`
SimStruct representing an S-Function block.

`name`
Parameter name.

`rValue`
Real values of the parameter.

`cValue`
Complex values of the parameter.

`dataType`
Data type of the parameter elements (see “Specifying Data Type Info” on page 10-230).

`size`
Number of elements in the vector.

Description Use this function in `mdlRTW` to write a vector parameter in Simulink format to this S-function’s `model.rtw` file. This function returns true if successful.

Languages C

See Also `mdlRTW`, `ssWriteRTWMxVectParam`

ssWriteRTWParameters

Purpose Write tunable parameter information to the `model.rtw` file.

Syntax

```
int_T ssWriteRTWParameters(SimStruct *S,  int_T nParams, int_T
  paramType, const char_T *paramName, const char_T *stringInfo,
  ...)
```

Arguments

`S`
SimStruct representing an S-Function block.

`nParams`
Number of tunable parameters.

`paramType`
Type of parameter (see "Parameter Type-Specific Arguments").

`paramName`
Name of the parameter.

`stringInfo`
General information about the parameter, such as how it was derived.

`...`
Remaining arguments depend on the parameter type (see "Parameter Type-Specific Arguments").

Description Use this function in `mdlRTW` to write tunable parameter information to this S-function's `model.rtw` file. Your S-function must write the parameters out in the same order as they are declared at the beginning of the S-function. This function returns true if successful.

Note This function is provided for compatibility with S-functions that do not use run-time parameters. It is suggested that you use run-time parameters (see "Run-Time Parameters" on page 7-5). If you do use run-time parameters, you do not need to use this function.

Parameter Type-Specific Arguments

This section lists the parameter-specific arguments required by each parameter type.

- `SS_WRITE_VALUE_VECT` (vector parameter)

Argument	Description
`const real_T *valueVect`	Pointer to an array of vector values
`int_T vectLen`	Length of the vector

- `SSWRITE_VALUE_2DMAT` (matrix parameter)

Argument	Description
`const real_T *valueMat`	Pointer to an array of matrix elements
`int_T nRows`	Number of rows in the matrix
`int_T nCols`	Number of columns in the matrix

- `SSWRITE_VALUE_DTYPE_2DMAT`

Argument	Description
`const real_T *valueMat`	Pointer to an array of matrix elements
`int_T nRows`	Number of rows in the matrix
`int_T nCols`	Number of columns in the matrix
`int_T dtInfo`	Data type of matrix elements (see "Specifying Data Type Info" on page 10-230)

- `SSWRITE_VALUE_DTYPE_ML_VECT`

Argument	Description
`const void *rValueVect`	Real component of the complex vector
`const void *iValueVect`	Imaginary component of the complex vector

Argument	Description
int_T vectLen	Length of the vector
int_T dtInfo	Data type of the vector (see "Specifying Data Type Info" on page 10-230)

- SSWRITE_VALUE_DTYPE_ML_2DMAT

Argument	Description
const void *rValueMat	Real component of the complex matrix
const void *iValueMat	Imaginary component of the complex matrix
int_T nRows	Number of rows in the matrix
int_T nCols	Number of columns in the matrix
int_T dtInfo	Data type of matrix

Specifying Data Type Info

You obtain the data type of the value argument passed to the ssWriteRTW macros using

```
DTINFO(dTypeId, isComplex)
```

where dTypeId can be any one of the enum values in BuiltInDTypeID (SS_DOUBLE, SS_SINGLE, SS_INT8, SS_UINT8, SS_INT16, SS_UINT16, SS_INT32, SS_UINT32, SS_BOOLEAN) defined in simstuc_types.h. The isComplex argument is either 0 or 1.

For example, DTINFO(SS_INT32,0) is a noncomplex 32-bit signed integer.

If isComplex==1, the array of values is assumed to have the real and imaginary parts arranged in an interleaved manner (i.e., Simulink format). If you prefer to pass the real and imaginary parts as two separate arrays, you should use the macro ssWriteRTWMxVectParam or ssWriteRTWMx2dMatParam.

Example

See simulink/src/sfun_multiport.c for an example that uses this function.

Languages C

See Also `mdlRTW`

ssWriteRTWParamSettings

Purpose Write values of nontunable parameters to the `model.rtw` file.

Syntax

```
int_T ssWriteRTWParamSettings(SimStruct *S,  int_T nParamSettings,
   int_T paramType, const char_T *settingName, ...)
```

Arguments

`S`
SimStruct representing an S-Function block.

`nParamSettings`
Number of parameter settings.

`paramType`
Type of parameter (see “Parameter Setting Type-Specific Arguments” on page 10-232).

`settingName`
Name of parameter.

`...`
Remaining arguments depend on the parameter type (see “Parameter Setting Type-Specific Arguments”).

Description

Use this function in `mdlRTW` to write nontunable parameter setting information to this S-function’s `model.rtw` file. A nontunable parameter is any parameter that the S-function has declared as nontunable, using the `ssSetParameterTunable` macro. You can also use this macro to write out other constant values required to generate code for this S-function.

This function returns true if successful.

Parameter Setting Type-Specific Arguments

This section lists the parameter-specific arguments required by each parameter type.

- `SSWRITE_VALUE_STR` (unquoted string)

Argument	Description
`const char_T *value`	String (e.g., `U.S.A.`)

- SSWRITE_VALUE_QSTR (quoted string)

Argument	Description
const char_T *value	String (e.g., "U.S.A.")

- SSWRITE_VALUE_VECT_STR (vector of strings)

Argument	Description
const char_T *value	Vector of strings (e.g., ["USA", "Mexico"])
int_T nItemsInVect	Size of the vector

- SSWRITE_VALUE_NUM (number)

Argument	Description
const real_T value	Number (e.g., 2)

- SSWRITE_VALUE_VECT (vector of numbers)

Argument	Description
const real_T *value	Vector of numbers (e.g., [300, 100])
int_T vectLen	Size of the vector

- SSWRITE_VALUE_2DMAT (matrix of numbers)

Argument	Description
const real_T *value	Matrix of numbers (e.g., [[170, 130],[60, 40]])
int_T nRows	Number of rows in the vector
int_T nCols	Number of columns in the vector

- `SSWRITE_VALUE_DTYPE_NUM` (data typed number)

Argument	Description
`const void *value`	Number (e.g., `[3+4i]`)
`int_T dtInfo`	Data type (see "Specifying Data Type Info" on page 10-230)

- `SSWRITE_VALUE_DTYPE_VECT` (data typed vector)

Argument	Description
`const void *value`	Data-typed vector (e.g., `[1+2i, 3+4i]`)
`int_T vectLen`	Size of the vector
`int_T dtInfo`	Data type (see "Specifying Data Type Info" on page 10-230)

- `SSWRITE_VALUE_DTYPE_2DMAT` (data-typed matrix)

Argument	Description
`const void *value`	Matrix (e.g., `[1+2i 3+4i; 5 6]`)
`int_T nRows`	Number of rows in the matrix
`int_T nCols`	Number of columns in the matrix
`int_T dtInfo`	Data type (see "Specifying Data Type Info" on page 10-230)

- SSWRITE_VALUE_DTYPE_ML_VECTOR (data-typed MATLAB vector)

Argument	Description
const void *RValue	Real component of the vector (e.g., [1 3])
const void *IValue	Imaginary component of the vector (e.g., [2 5])
int_T vectLen	Number of elements in the vector
int_T dtInfo	Data type (see "Specifying Data Type Info" on page 10-230)

- SSWRITE_VALUE_DTYPE_ML_2DMAT (data typed MATLAB matrix)

Argument	Description
const void *RValue	Real component of the matrix (e.g., [1 5 3 6])
const void *IValue	Real component of the matrix (e.g., [2 0 4 0])
int_T nRows	Number of rows in the matrix
int_T nCols	Number of columns in the matrix
int_T dtInfo	Data type (see "Specifying Data Type Info" on page 10-230)

Example

See simulink/src/sfun_multiport.c for an example that uses this function.

Languages

C

See Also

mdlRTW, ssSetParameterTunable

ssWriteRTWScalarParam

Purpose

Write a scalar parameter to the model.rtw file.

Syntax

```
int_T ssWriteRTWScalarParam(SimStruct *S,  const char_T *name,
   const void *value, int_T type)
```

Arguments

S
SimStruct representing an S-Function block.

name
Parameter name.

value
Parameter value.

type
Integer ID of the type of the parameter value, for example, the ID of one of Simulink's built-in data types (see BuiltInDTypeId in simstruc_types.h in the MATLAB simulink/include subdirectory) or the ID of a user-defined type (see "Custom Data Types" on page 7-14).

Description

Use this function in mdlRTW to write scalar parameters to this S-function's model.rtw file. This function returns true if successful.

Languages

C

See Also

mdlRTW

Purpose Write a string to the `model.rtw` file.

Syntax

```
int_T ssWriteRTWStr(SimStruct *S,  const char_T *str)
```

Arguments

`S`
SimStruct representing an S-Function block.

`str`
String.

Description Use this function in `mdlRTW` to write strings to this S-function's `model.rtw` file. This function returns true if successful.

Languages C

See Also `mdlRTW`

ssWriteRTWStrParam

Purpose Write a string parameter to the `model.rtw` file.

Syntax

```
int_T ssWriteRTWStrParam(SimStruct *S,  const char_T *name,
   const char_T *value)
```

Arguments

`S`
SimStruct representing an S-Function block.

`name`
Parameter name.

`value`
Parameter value.

Description Use this function in `mdlRTW` to write string parameters to this S-function's `model.rtw` file. This function returns true if successful.

Languages C

See Also `mdlRTW`

Purpose

Write a string vector parameter to the `model.rtw` file.

Syntax

```
int_T ssWriteRTWStrVectParam(SimStruct *S,  const char_T *name,
   const void *value, int_T size)
```

Arguments

`S`
SimStruct representing an S-Function block.

`name`
Parameter name.

`value`
Parameter values.

`size`
Number of elements in the vector.

Description

Use this function in `mdlRTW` to write a vector of string parameters to this S-function's `model.rtw` file. This function returns true if successful.

Languages

C

See Also

`mdlRTW`

ssWriteRTWVectParam

Purpose Write a vector parameter to the `model.rtw` file.

Syntax

```
int_T ssWriteRTWVectParam(SimStruct *S,  const char_T *name,
   const void *value, int_T dataType, int_T size)
```

Arguments

`S`
SimStruct representing an S-Function block.

`name`
Parameter name.

`value`
Parameter values.

`dataType`
Data type of the parameter elements (see “Specifying Data Type Info” on page 10-230).

`size`
Number of elements in the vector.

Description Use this function in `mdlRTW` to write a vector parameter in Simulink format to this S-function’s `model.rtw` file. This function returns true if successful.

Languages C

See Also `mdlRTW`, `ssWriteRTWMxVectParam`

Purpose Write work vectors to the `model.rtw` file.

Syntax

```
int_T ssWriteRTWWorkVect(SimStruct *S,  const char_T *vectName,
   int_T nNames, const char_T *name1, int_T size1,  ...,
   const char_T * nameN, int_T sizeN)
```

Arguments

`S`
SimStruct representing an S-Function block.

`vectName`
Name of work vector (must be `"RWork"`, `"IWork"`, or `"PWork"`).

`nNames`
NEED DESCRIPTION

`name1 ... nameN`
Names of groups of work vector elements.

`size1 ... sizeN`
Size of each element group (the total of the sizes must equal the size of the work vector).

Description Use this function in `mdlRTW` to write work vectors to this S-function's `model.rtw` file. This function returns true if successful.

Languages C

See Also `mdlRTW`

Index

N

O

P

R

S

T

V

W

Z